Informatyka, 1 rok, niestacjonarne – gr. wyk. W2 / ćw. 5

Wprowadzenie do programowania w języku C#

Wykładowca: dr Mirosław Chodacki

Prowadzący laboratoria: dr Jacek Mosakowski

Projekt i implementacja Perceptronu

Jarosław Stajuda

Wyższa Szkoła Ekonomii i Informatyki w Krakowie

Kraków, rok akadem. 2017/2018

stajudaj@gmail.com

Rys historyczny

Ludzie nauki od wieków dociekają zasad działania żywych organizmów, w szczególności ciała człowieka. Rozwój technologiczny pozwala nie tylko na coraz to bardziej dogłębną analizę, ale również na próbę odtworzenia elementów biologicznych w postaci komponentów elektromechanicznych lub czysto elektronicznych.

Jak opisuje Jurgen Thorwald (lit. 6) – świętym graalem badań był ludzki mózg. Najtrudniej dostępny, najbardziej złożony organ jeszcze do niedawna pozostawał tajemnicą. Sytuacja zmieniła się w XX wieku, kiedy to dynamiczny rozwój technologii pozwolił na mozolne odkrywanie „kruchego domu duszy".

Frank Rosenblatt, profesor na Uniwersytecie Cornell w USA, zainteresowany był w szczególności procesami uczenia się. Przeprowadzając badania z udziałem zwierząt doszedł do wniosku, że możliwe jest stworzenie systemu składającego się ze sztucznych neuronów, zdolnego do gromadzenia wiedzy.

System ten nazwał Perceptronem (w psychologii: percepcja, czyli umiejętność rozpoznawania i klasyfikowania sygnałów z otoczenia) – sieć złożoną z elektronicznych neuronów, zdolną do samodzielnego uczenia się.

Sztuczny neuron to (w wielkim uproszczeniu) odwzorowanie komórki nerwowej – układ obliczeniowy posiadający wiele wejść i jedno wyjście, które służy do przesyłania sygnału do odbiorników.

Odbiornikami mogą być również inne neurony, co umożliwia tworzenie złożonych przetwarzających informacje sieci neuronowych.

W systemie Rosenblatta sygnały wejściowe płynęły z prymitywnej kamery cyfrowej, natomiast sygnał wyjściowy manifestowany był za pomocą lampek. Sztuczne neurony syntetyzowały ważone sygnały wejściowe i kalkulowały wyjście z tym założeniem, że obróbka sygnałów wejściowych podlegała procesowi uczenia się. Efekt ten został osiągnięty przez zastosowanie potencjometrów do dostrajania wag sygnałów wejściowych. Potencjometry kontrolowane były przez pojedyncze silniczki, które modyfikowały wagi w zależności od poprawności wykonania zadania (w tym wypadku rozpoznania kształtu). Poprawność rozpoznania sygnalizowana była przez nadzorującego działanie Perceptronu nauczyciela.

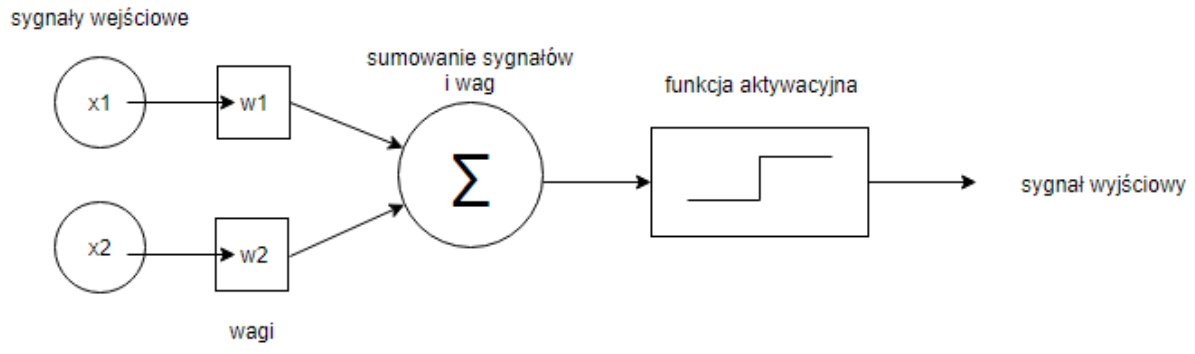
Efekty działania Perceptronu były zdumiewające – maszyna nie wykonująca konkretnego algorytmu była zdolna nauczyć się wykonywać określone zadanie na podstawie prób i korygowania błędów. Stworzone zostały w ten sposób fundamenty dla wszechobecnych współcześnie zaawansowanych sieci neuronowych.

Zasada działania

Perceptron jest prostą siecią neuronową, którego działanie polega na analizowaniu danych wejściowych i wyrażaniu wyniku analizy poprzez jeden z dwóch stanów – najczęściej stosuje się umowne wartości 1, -1.

Przedmiotem analizy jest prosty Perceptron złożony z pojedynczego neuronu, przyjmujący dwa sygnały wejściowe. Każdy z sygnałów ma przypisaną wagę, która to jest kluczowym elementem w procesie uczenia.

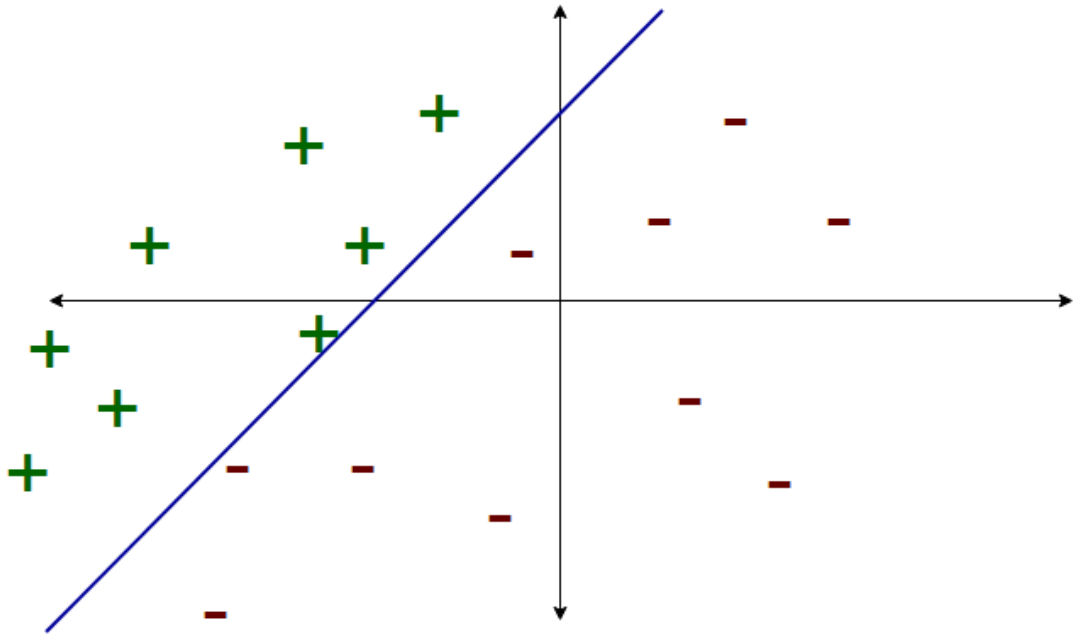
Sygnały wejściowe są następnie integrowane, a produkt integracji przekazywany do funkcji aktywacyjnej – od wyniku której zależy, czy neuron na wyjściu przekaże sygnał oznaczający stan wzbudzenia (1), czy też będzie to sygnał hamujący (-1).



Rysunek 1 Schemat omawianego Perceptronu

Prosty Perceptron służy do binarnej klasyfikacji – determinuje na podstawie danych wejściowych, czy obiekt reprezentowany tymi danymi przypisać do jednej z dwóch klas. Graficznie przedstawić tę

zasadę można posługując się wykresem funkcji liniowej oraz zbiorem losowo położonych na płaszczyźnie punktów.



Rysunek 2 Binarna klasyfikacja liniowa

Na przedstawionym powyżej wykresie punkty przyporządkowane są do danej klasy (+ lub -) względem funkcji liniowej.

Poniżej przedstawiony zostanie program komputerowy, który przy użyciu Perceptronu uczy się klasyfikować losowo wygenerowane punkty pod kątem ich położenia względem wykresu zadanej funkcji liniowej.

Projekt i implementacja

W rozwiązaniu zadania zastosowana zostanie metoda uczenia nadzorowanego (ang. supervised learning), polegającą na przekazywaniu do sieci neuronowej danych wejściowych, dla których znana jest prawidłowa odpowiedź i w zależności od uzyskanego wyniku – korygowania wag w celu minimalizacji błędu.

Schemat postępowania

1. Przekazanie do Perceptronu danych wejściowych, dla których znane są prawidłowe odpowiedzi. Wagi inicjalizowane są wartościami losowymi z zakresu od -1 do 1.
2. Następuje integracja sygnałów wejściowych i przekazanie ich do funkcji aktywacji, która na podstawie zintegrowanego sygnału przekazuje na wyjście wartość 1 lub -1.

3. Następuje kalkulacja błędu.
4. Następuje skorygowanie wartości wag zgodnie z wartością błędu.
5. Powrót do pkt. 1

W rozpatrywanym przypadku danymi wejściowymi są współrzędne punktu na płaszczyźnie (**x1, x2**), którym początkowo przypisywane są losowe wagi z zakresu od -1 do 1.

Punkty reprezentowane są przeze egzemplarze klasy Point, a ich współrzędne inicjalizowane są wartościami losowymi.

[listing 1.1 - klasa Point]

```
public class Point : Input

    {
        private double x;
        private double y;

        public Point(double x, double y)
        {
            this.x = x;
            this.y = y;
        }

        public int GetLabel(Function linFunc)
        {
            return (y >= linFunc.GetValue(x)) ? 1 : -1;
        }

        public double[] GetInputs()
        {
            return new double[]{ this.x, this.y };
        }

        public override string ToString()
        {
            return string.Format("P(x = {0}, y = {1})", this.x, this.y);
        }

    }
```

Klasa Point realizuje interfejs Input, który wymusza implementację metody GetInputs() udostępniającej sygnały wejściowe dla neuronu, oraz metody GetLabel(Function linFunc), która określa prawidłową klasyfikację punktu względem podanej funkcji liniowej.

[listing 1.2 - interfejs Input]

```
public interface Input
    {
        double[] GetInputs();
        int GetLabel(Function linFunc);
    }
```

Funkcja liniowa reprezentowana jest egzemplarzem klasy LinearFunction, która pozwala łatwo dostosowywać jej charakterystykę poprzez modyfikację parametrów.

[listing 1.3 – klasa LinearFunction]

```
public class LinearFunction : Function
    {
        private double a;
        private double b;

           public LinearFunction(double a, double b)
           {
            this.a = a;
            this.b = b;
           }

        public double GetValue(double x)
        {
            return x * a + b;
        }

        public override string ToString()
        {
            string strA = "";
            if (b == 0) strA = "0";
            string strB = "";

            if( a != 0 )
            {
                if( a != 1) strA = string.Format("{0}x", a);
                else strA = string.Format("x");
            }

            if (b != 0)
            {
                if( b > 0) strB = string.Format("+ {0}", b);
                else if(b < 0) strB = string.Format("{0}", b);
            }

            return string.Format("Funkcja f(x) = {0} {1}", strA, strB);
        }
    }
```

Wygenerowane egzemplarze klasy Point przechowywane są przy użyciu generycznej klasy bibliotecznej List<T>, która inicjalizowana jest przy użyciu metody initPointsList().

[listing 1.4 – metoda initPoints]

```
public static void initPointsList(List<Point> points, int count)
        {
            Random rnd = new Random();

            for (int i = 0; i < count; i++)
            {
                Point p = new Point(rnd.Next(-100, 100), rnd.Next(-100, 100));
                points.Add(p);
            }
        }
```

Egzemplarz klasy Perceptron tworzony jest przy użyciu konstruktora, który jako parametr przyjmuje ilość sygnałów wejściowych przekazywanych do sieci neuronowej (w tym wypadku 2).
Dla każdego z sygnałów generowana jest losowa wartość wagi – metoda initWeights().

Konstruktor klasy Perceptron posiada też opcjonalny parametr learningRate (domyślna wartość 0.1), który może służyć do modyfikacji wartości korekty wag przy pojedynczej analizie danych wejściowych.

[listing 1.5 – konstruktor klasy Perceptron, metoda initWeights]

```
public Perceptron(int inputCount, double learningRate = 0.1)
           {
            weights = new double[inputCount];
            initWeights(weights);
            this.learningRate = learningRate;
        }

        private static void initWeights(double[] weights)
        {
            for (int i = 0; i < weights.Length; i++)
            {
                weights[i] = RandomNumberBetween(-1, 1);
            }
        }
```

Integracja sygnałów wejściowych polega na sumowaniu iloczynów sygnałów i ich wag.

ZSUMOWANY SYGNAŁ = x1 * waga1 + x2 * waga2

Wartość ta jest przekazywana do funkcji aktywacji, która determinuje sygnał wyjściowy spośród wartości {-1, 1}. Funkcja ta dla dodatniej wartości sumy zwraca wartość 1, dla ujemnej -1.

Jest to próba odgadnięcia przez Perceptron prawidłowego sygnału wyjściowego.

[listing 1.6 - metoda Guess]

```
public int Guess(double[] inputs)
        {
            double sum = 0;

            for (int i = 0; i < weights.Length; i++)
            {
                sum += weights[i] * inputs[i];
            }

            return Activate(sum);
        }
```

[listing 1.7 - metoda Activate]

```
private int Activate(double signal)
        {
            if (signal >= 0)
            {
                return 1;
            }
            return -1;
        }
```

Wartości błędu wylicza się wzorem:

BŁĄD = PRAWIDŁOWE WYJŚCIE – ODPOWIEDŹ PERCEPTRONU

Zgodnie z przyjętymi założeniami wartości błędu mogą być następujące:

Prawidłowe wyjście	Odpowiedź Perceptronu	Błąd
1	1	0
1	-1	2
-1	1	-2
-1	-1	0

Po skalkulowaniu wartości błędu kolejnym krokiem jest ustalenie nowej wagi dla danego sygnału. Proces ten można opisać wzorem:

NOWA WAGA = WAGA + ΔWAGI

ΔWAGI – zmiana wartości wagi przy kolejnej analizie sygnału wejściowego – następuje poprzez pomnożenie wartości sygnału przez wartość błędu. Warto zastosować również trzeci czynnik, w celu kontrolowania proporcji zmian wagi (zbyt gwałtowne skoki wartości mogą powodować powiększanie wartości błędu).

ΔWAGI = BŁĄD * WEJŚCIE (* WSPÓŁCZYNNIK UCZENIA)

Tak więc

NOWA WAGA = WAGA + BŁĄD * WEJŚCIE (* WSPÓŁCZYNNIK UCZENIA)

[listing 1.8 - metoda Train]

```
public bool Train(double[] inputs, int target)
        {
            int guess = Guess(inputs);
            int error = target - guess;

            if (guess != target)
            {
                for (int i = 0; i < inputs.Length; i++)
                {
                    weights[i] += error * inputs[i] * learningRate;
                }
                return true;
            }

            return false;
        }
```

Cały schemat działania realizowany jest przez metodę Perceptron.Run(), która jako argumenty przyjmuje listę obiektów realizujących interfejs Input, zadaną funkcję liniową, oraz flagę określającą pierwszą iterację (dla zobrazowania jaki jest początkowy odsetek prawidłowych odpowiedzi w pierwszym obiegu metoda dla każdego punktu wejściowego korzysta z metody Guess(), która nie dostraja wag w celu minimalizacji błędu).
Metoda Run() dostarcza również informacji o całkowitej ilości popełnionych w danym podejściu błędów. Korzystając z metody Report() wyświetlane są w konsoli szczegółowe informacje dotyczące analizy poszczególnych punktów.

[listing 1.9 - metoda Run]

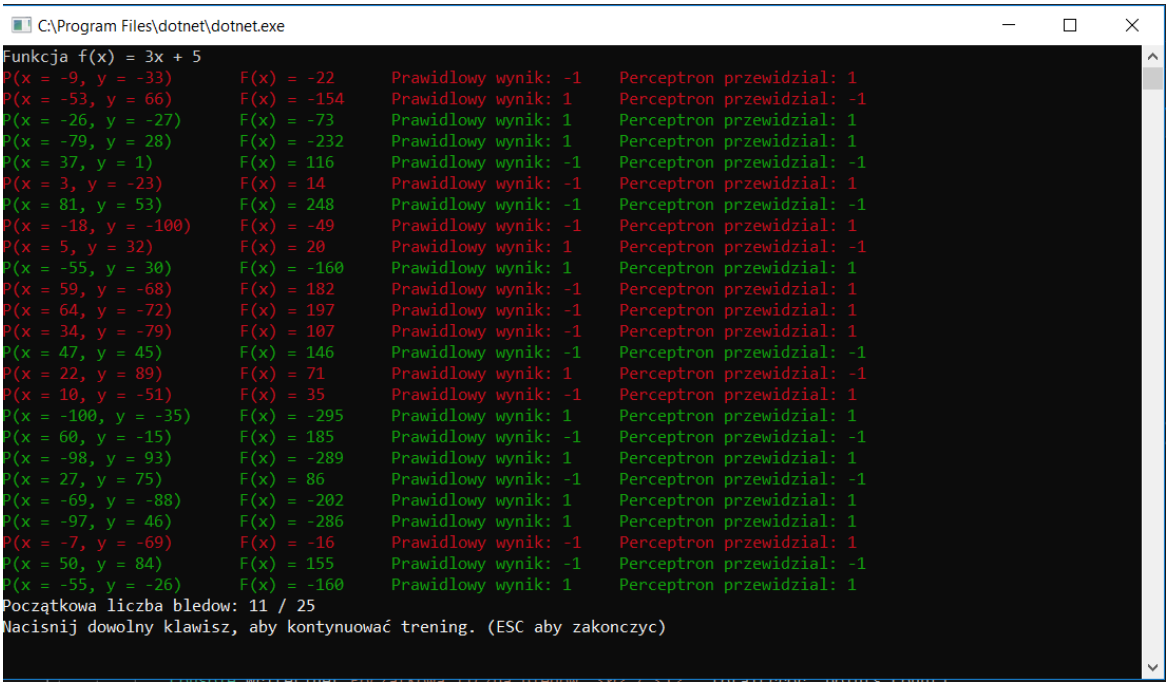
```
public int Run<T>(IEnumerable<T> inputs, Function linFunc, bool initial = false) where
T : Input
        {
            int totalError = 0;

            foreach (Input input in inputs)
            {
                int guess = Guess(input.GetInputs());
                int label = input.GetLabel(linFunc);
                double funcVal = linFunc.GetValue(input.GetInputs()[0]);
                string pointInfo = input.ToString();
                bool error = false;

                if( initial == true ) {
                    if ( guess != label )
                    {
                        totalError++;
                        error = true;
                    }
                }
                else
                {
                    if ( Train(input.GetInputs(), label) )
                    {
                        totalError++;
                        error = true;
                    }
                }

                Report(guess, label, funcVal, pointInfo, error);
            }

            return totalError;
        }
```

[listing 1.10 – metoda Report]

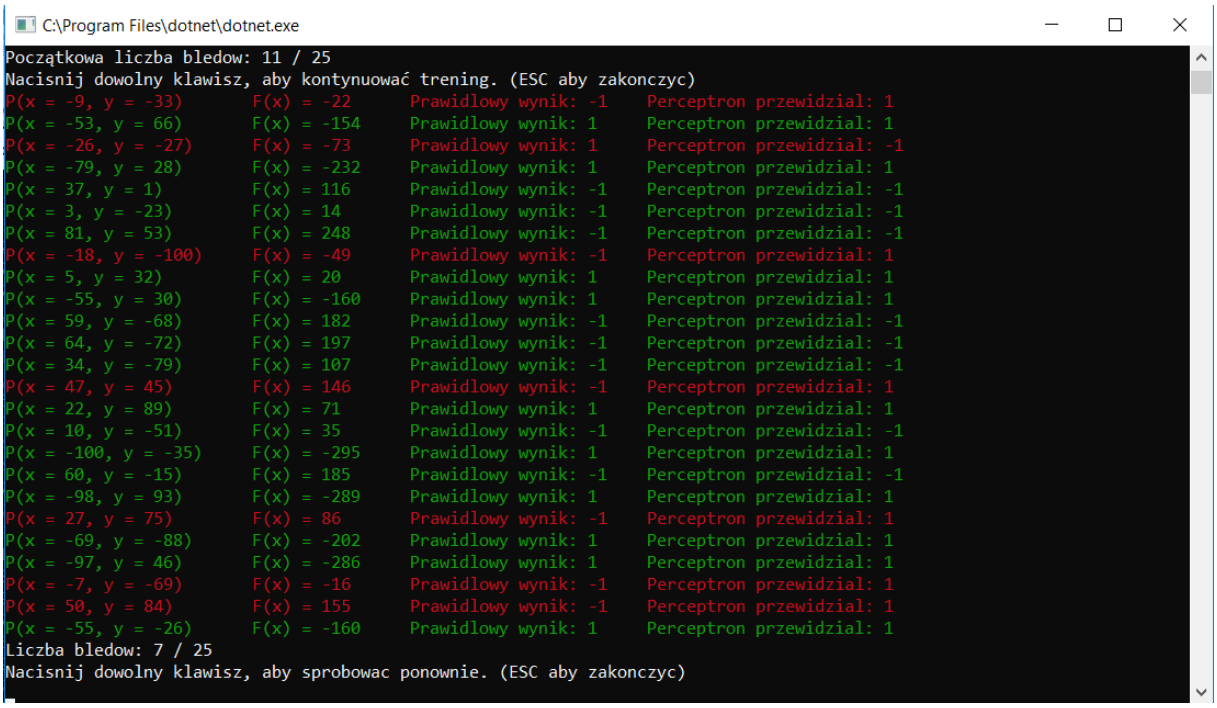
```
public void Report(int guess, int label, double funcVal, string pointInfo, bool error
= false)
        {
            if( error ) Console.ForegroundColor = ConsoleColor.DarkRed;
            else Console.ForegroundColor = ConsoleColor.DarkGreen;
            Console.Write(pointInfo + "\t");
            Console.Write(" F(x) = " + funcVal + "\t");
            Console.Write(" Prawidlowy wynik: " + label + "\t");
            Console.Write(" Perceptron przewidzial: " + guess + "\n");
        }
```

Schemat powtarza się do momentu, kiedy całkowita ilość błędów wynosi 0, lub użytkownik opuści program przy użyciu klawisza ESC.

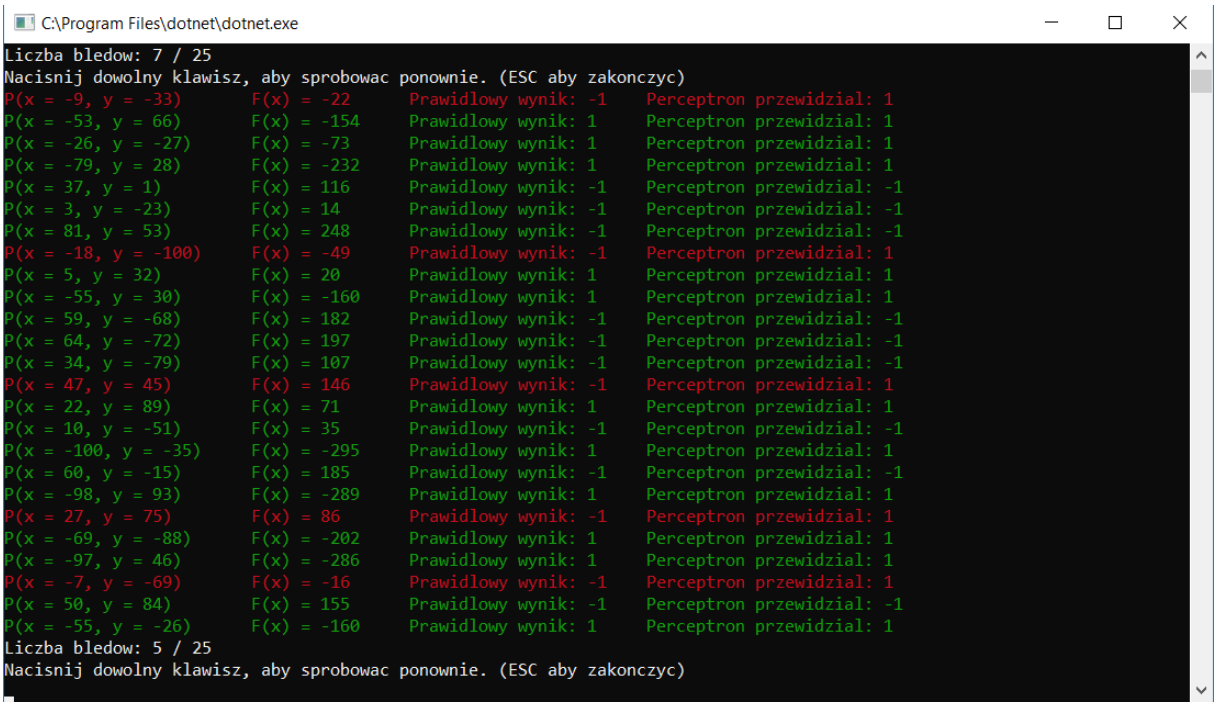
Zrzuty ekranu z poszczególnych faz wykonania programu



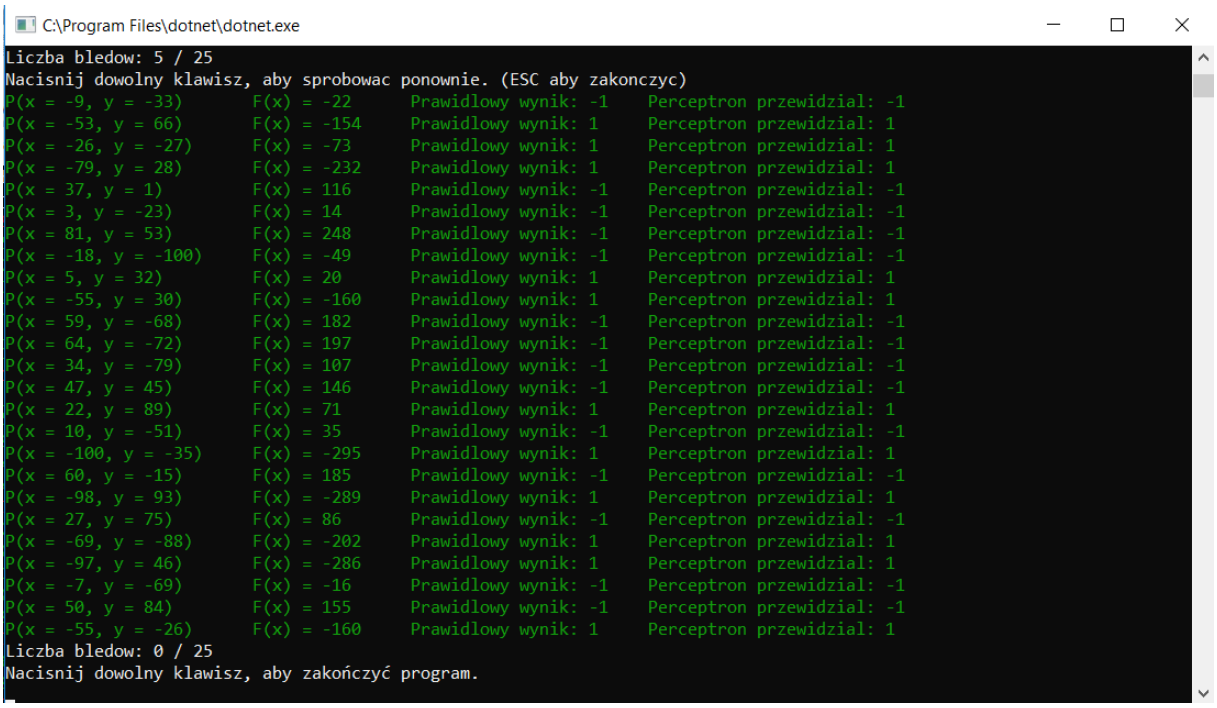
Rysunek 3 - Pierwsze wykonanie (wagi losowe) – liczba błędów 11



Rysunek 4 Druga iteracja – liczba błędów 7



Rysunek 5 Trzecia iteracja – liczba błędów 5



Rysunek 6 Czwarta iteracja - 0 błędów - koniec działania programu

Podsumowanie i wnioski

Zaprezentowana realizacja prostego Perceptronu Rosenblatta obrazuje fenomen sieci neuronowych. Program komputerowy nie posiadający konkretnego algorytmu służącego do rozwiązania problemu, posiadający w punkcie początkowym zupełnie losowe dane jest w stanie gromadzić wiedzę, ucząc się metodą prób i błędów, przypominając tym działaniem pracę mózgu ludzkiego.

W opracowaniu przedstawiono pierwotną koncepcję sztucznego neuronu – urządzenia elektronicznego sterowanego manualnie. Następnie zaprezentowano projekt i implementację tegoż neuronu przy pomocy języka C#.

Zaprezentowano zrzuty ekranu z poszczególnych etapów wykonania programu, które obrazują proces uczenia się i minimalizacji ilości popełnianych przez Perceptron błędów.

Zwrócono też uwagę, że przy specyficznym doborze sygnałów wejściowych i wygenerowanych wag sieć nie jest w stanie zredukować błędów do 0. Rozwiązaniem tego problemu może być manipulacja parametrem oznaczającym współczynnik uczenia się Perceptronu, wprowadzenie dodatkowego sygnału progowego (ang. bias), oraz zaimplementowanie zaawansowanej metody wyboru danych startowych.

W spisie literatury, z której korzystano w trakcie tworzenia tego opracowania pojawia się wiele pozycji autorstwa prof. Ryszarda Tadeusiewicza – znakomitego naukowca i popularyzatora wiedzy z zakresu biocybernetyki i sztucznej inteligencji.

Literatura

1. Tadeusiewicz Ryszard – Sieci Neuronowe – wyd. 2 (1993)
2. Tadeusiewicz Ryszard, Leper Bartosz, Borowik Barbara, Gąciarz Tomasz - Odkrywanie właściwości sieci neuronowych: przy użyciu programów w języku C# (2007)
3. http://ryszardtadeusiewicz.natemat.pl/129195,pierwszy-dzialajacy-techniczny-model-mozgu
4. https://en.wikipedia.org/wiki/Perceptron
5. https://www.researchgate.net/publication/294578763_LEKSYKON_SIECI_NEURONOWYCH_Lexicon_on_Neural_Networks
6. Jürgen Thorwald - Kruchy dom duszy (2010)
7. https://en.wikipedia.org/wiki/Frank_Rosenblatt
8. https://www.uci.agh.edu.pl/uczelnia/tad/
9. http://ryszardtadeusiewicz.natemat.pl/