

Sprawozdanie Scenariusz 2 Aleksandra Karaś

Budowa i działanie sieci jednowarstwowej

Cel ćwiczenia:

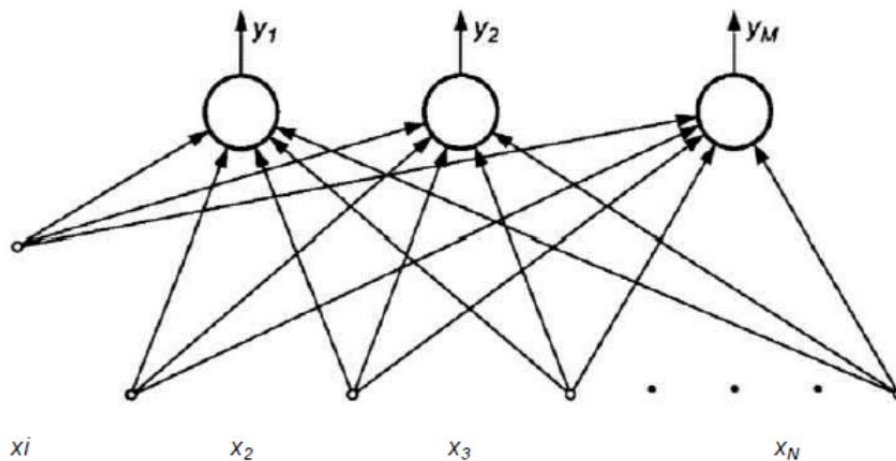
Celem ćwiczenia jest poznanie budowy i działania jednowarstwowych sieci neuronowych oraz uczenie ich rozpoznawania wielkości liter.

2. Opis budowy sieci i algorytmów uczenia.

W celu uczenia sieci neuronowej stworzyłam zestaw liter (10 dużych i 10 małych), które są reprezentowane w postaci dwuwymiarowej tablicy 5x7 pikseli dla jednej litery.

Ćwiczenie zostało wykonane w języku C++ (własna implementacja), w którym zastosowano uczenie nadzorowane (z pomocą nauczyciela). Wykorzystano model sieci Adaline oraz DeltaRule.

Schemat sieci neuronowej jednowarstwowej



Schemat uczenia sieci jednowarstwowej

Dane jest p par uczących: $\{(y_1, d_1), (y_2, d_2), \dots, (y_p, d_p)\}$, gdzie y_i ma rozmiar $J \times 1$, d_i ma rozmiar $K \times 1$. parametr l oznacza numer kroku cyklu uczenia.

1. Wybór $h > 0$, $E_{max} > 0$.
2. Wybór początkowych wartości elementów macierzy wag W jako niewielkich liczb losowych. Macierz W ma wymiar $K \times J$.
3. Ustawienie wartości początkowej licznika kroków oraz wyzerowanie wartości błędu:
 - 1) $l = 1$
 - 2) $E = 0$

4. Podanie danych na wejście i obliczenie sygnału wyjściowego $y = y_l$, $d = d_l$, $z_k = j(w_k T_y)$, $k = 1, 2, \dots, K$ (gdzie $w_k T$ jest k -tym wierszem macierzy W).
5. Uaktualnienie wag według wzoru:

a) dla Adaline: $w \leftarrow w + \eta(o - y)x$, gdzie:

η is the learning rate (some positive constant)

y is the output of the model

o is the target (desired) output

b) dla DeltaRule: $\Delta w_{ji} = \alpha(t_j - y_j)g'(h_j)x_i$, gdzie:

α is a small constant called *learning rate*

$g(x)$ is the neuron's activation function

t_j is the target output

h_j is the weighted sum of the neuron's inputs

y_j is the actual output

x_i is the i th input.

6. Obliczenie błędu łącznego:

$$Q(w) = \frac{1}{2} \varepsilon^2 = \frac{1}{2} \left[d - \sum_{i=0}^n w_i x_i \right]^2$$

a) dla Adaline:

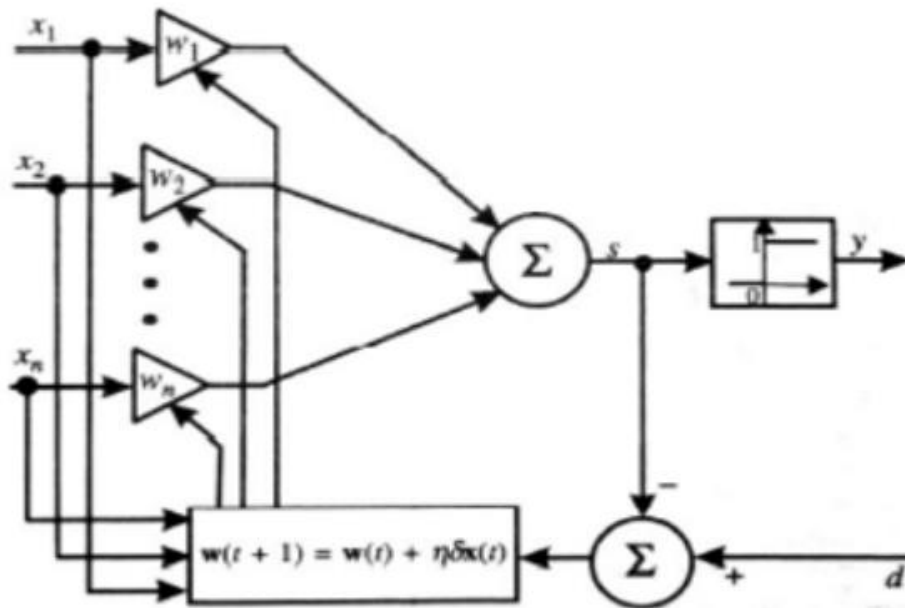
$$Q(w) = \frac{1}{2} \left[d - f \left(\sum_{i=0}^n w_i x_i \right) \right]^2$$

b) dla DeltaRule:

7. Jeżeli $l < p$, to $l = l + 1$ oraz przejście do kroku 4.
8. Cykl uczenia został zakończony, jeżeli $E < E_{\max}$ (Threshold). W przeciwnym wypadku rozpoczęcie nowego cyklu uczenia (powrót do kroku 3.).

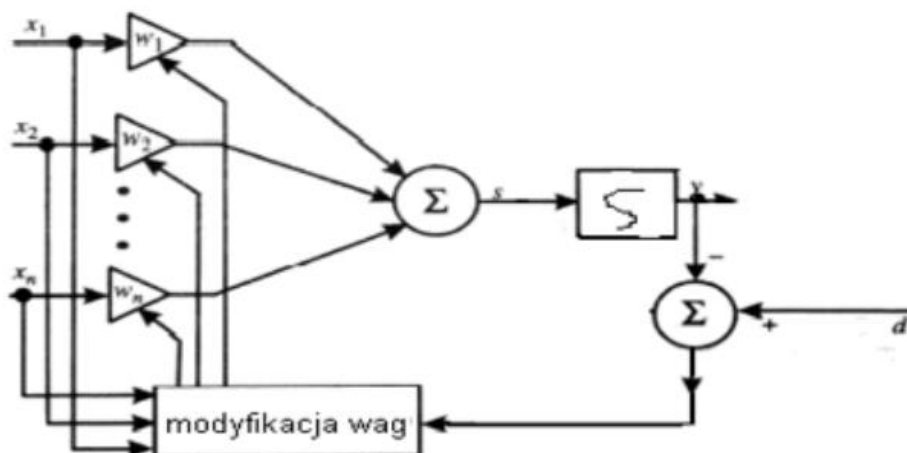
Schemat sieci Adaline

Model Adaline ma zbliżoną budowę do perceptronu. Różnią się one algorytmami uczenia. W modelu Adaline nie jest uwzględniana funkcja aktywacji przy porównywaniu sygnału wyjściowego z sygnałem wzorcowym.



Schemat sieci DeltaRule

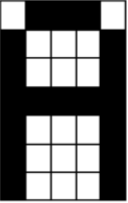
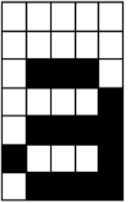
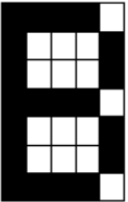
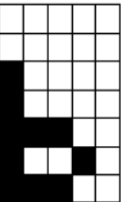
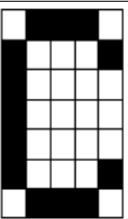
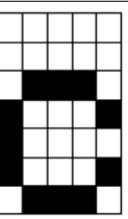
Model DeltaRule ma analogiczną budowę do modelu Adaline, jednakże funkcją aktywacji jest funkcja sigmoidalna, a przy aktualizacji wag uwzględnia się pochodną tejże funkcji.

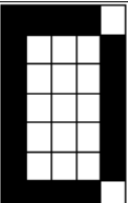
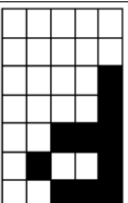
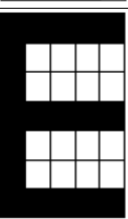
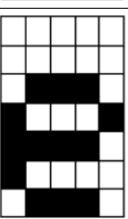
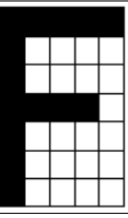
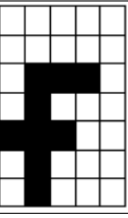


Duże litery wykorzystane w ćwiczeniu: ABCDEFGHIJ

Małe litery wykorzystane w ćwiczeniu: abcdefghij

Poniżej reprezentacja liter binarnie:

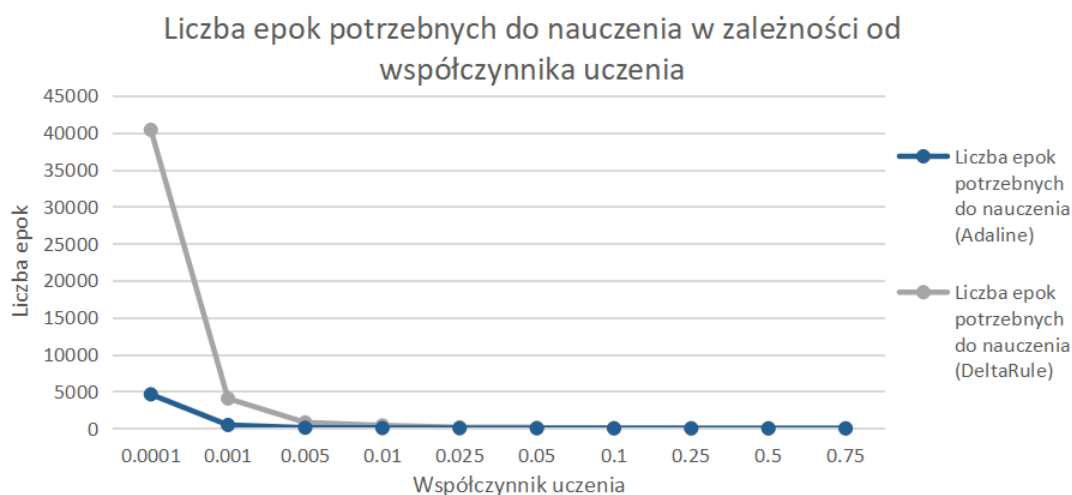
	01110 10001 10001 11111 10001 10001 10001		00000 00000 01110 00001 01111 10001 01111
	11110 10001 10001 11110 10001 10001 11110		00000 00000 10000 10000 11100 10010 11100
	01110 10001 10000 10000 10000 10001 01110		00000 00000 01110 10001 10000 10001 01110

	11110 10001 10001 10001 10001 10001 11110		00000 00000 00001 00001 00111 01001 00111
	11111 10000 10000 11110 10000 10000 11111		00000 00000 01110 10001 11110 10000 01110
	11111 10000 10000 11110 10000 10000 10000		00000 00000 01110 01000 11100 01000 01000

	11111 10001 10000 10111 10001 10001 01110		00000 00000 01111 10001 01111 00001 11110
	10001 10001 10001 11111 10001 10001 10001		00000 00000 10000 10000 11110 10001 10001
	01110 00100 00100 00100 00100 00100 01110		00000 00000 00100 00000 00100 00100 00110
	11111 00001 00001 00001 00001 10001 01110		00000 00000 00001 00111 00001 01001 00110

Wyniki:

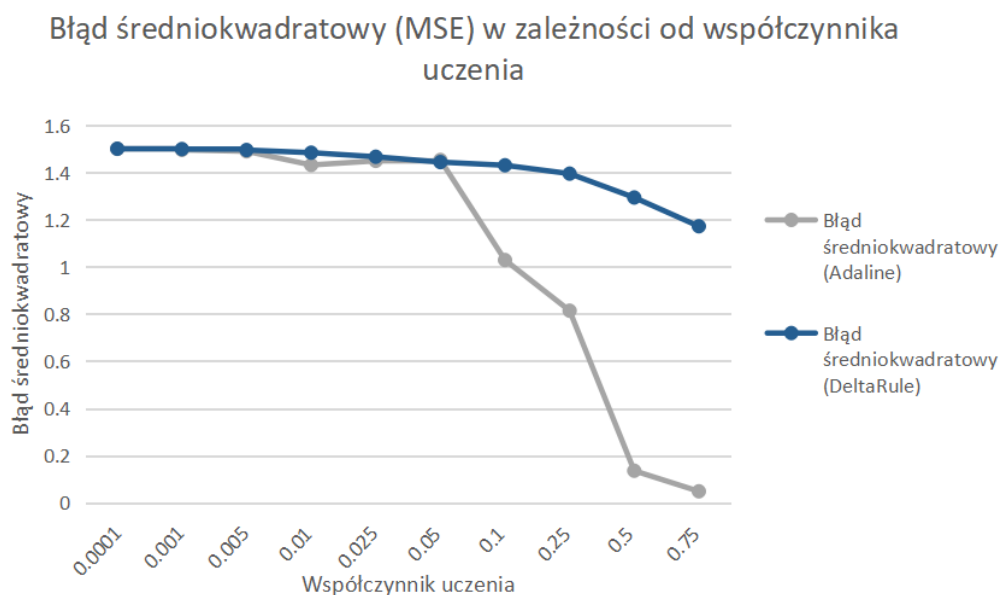
Learning Rate: 0.01	Learning Rate: 0.05	Learning Rate: 0.1
ADALINE - LEARNING	ADALINE - LEARNING	ADALINE - LEARNING
Epoch: 48	Epoch: 13	Epoch: 11
MSE error: 0.0997239	MSE error: 0.0911749	MSE error: 0.0994757
Test letters:	Test letters:	Test letters:
Letter A is: big	Letter A is: big	Letter A is: big
Letter B is: big	Letter B is: big	Letter B is: big
Letter C is: big	Letter C is: big	Letter C is: big
Letter D is: big	Letter D is: big	Letter D is: big
Letter E is: big	Letter E is: big	Letter E is: big
Letter F is: big	Letter F is: big	Letter F is: big
Letter G is: big	Letter G is: big	Letter G is: big
Letter H is: big	Letter H is: big	Letter H is: big
Letter I is: big	Letter I is: big	Letter I is: big
Letter J is: big	Letter J is: big	Letter J is: big
Letter a is: small	Letter a is: small	Letter a is: small
Letter b is: small	Letter b is: small	Letter b is: small
Letter c is: small	Letter c is: small	Letter c is: small
Letter d is: small	Letter d is: small	Letter d is: small
Letter e is: small	Letter e is: small	Letter e is: small
Letter f is: small	Letter f is: small	Letter f is: small
Letter g is: small	Letter g is: small	Letter g is: small
Letter h is: small	Letter h is: small	Letter h is: small
Letter i is: small	Letter i is: small	Letter i is: small
Letter j is: small	Letter j is: small	Letter j is: small
DELTARULE SIGMOID - LEARNING	DELTARULE SIGMOID - LEARNING	DELTARULE SIGMOID - LEARNING
Epoch: 405	Epoch: 82	Epoch: 42
MSE error: 0.0998097	MSE error: 0.0990114	MSE error: 0.0968022
Test letters:	Test letters:	Test letters:
Letter A is: big	Letter A is: big	Letter A is: big
Letter B is: big	Letter B is: big	Letter B is: big
Letter C is: big	Letter C is: big	Letter C is: big
Letter D is: big	Letter D is: big	Letter D is: big
Letter E is: big	Letter E is: big	Letter E is: big
Letter F is: big	Letter F is: big	Letter F is: big
Letter G is: big	Letter G is: big	Letter G is: big
Letter H is: big	Letter H is: big	Letter H is: big
Letter I is: big	Letter I is: big	Letter I is: big
Letter J is: big	Letter J is: big	Letter J is: big
Letter a is: small	Letter a is: small	Letter a is: small
Letter b is: small	Letter b is: small	Letter b is: small
Letter c is: small	Letter c is: small	Letter c is: small
Letter d is: small	Letter d is: small	Letter d is: small
Letter e is: small	Letter e is: small	Letter e is: small
Letter f is: small	Letter f is: small	Letter f is: small
Letter g is: small	Letter g is: small	Letter g is: small
Letter h is: small	Letter h is: small	Letter h is: small
Letter i is: small	Letter i is: small	Letter i is: small
Letter j is: small	Letter j is: small	Letter j is: small



Proces uczenia sieci jednowarstwowej przebiega analogicznie do uczenia perceptronu. Można zauważyć, że im wyższy współczynnik uczenia, tym liczba potrzebnych epok była niższa. Dla współczynnika uczenia > 0.01 sieć została wyćwiczona już po kilkunastu epokach. Dla współczynnika uczenia $= 0.0001$ proces uczenia zajął ponad 4000 epok (dla metody DeltaRule).

W przypadku metody Adaline sieć dla bardzo małych współczynników uczenia nie potrzebowała aż tak dużej liczby, lecz ich ilość była zauważalna.

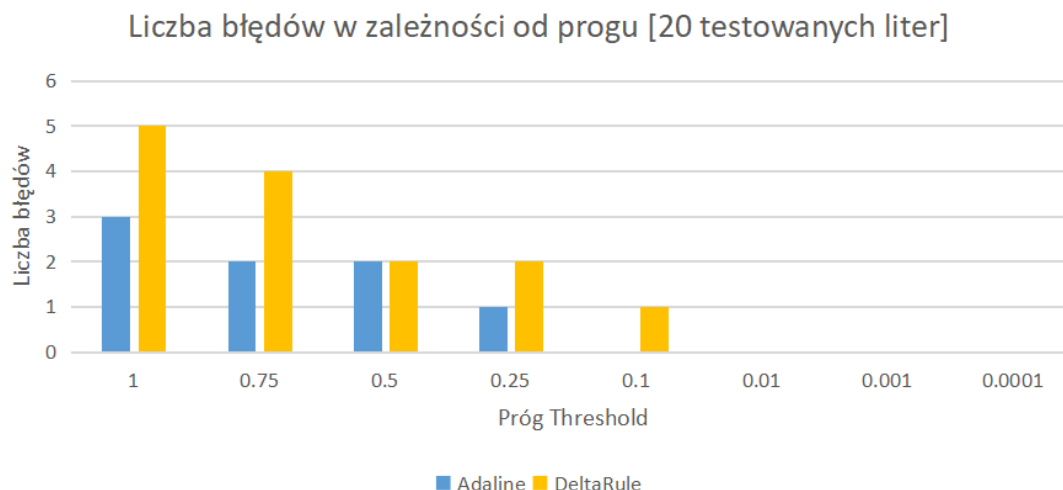
Dla współczynnika uczenia $= 0.01$ liczba epok uczenia praktycznie się wyrównała.



Z powyższego wykresu można odczytać, że współczynnik uczenia silnie wpływa na błąd średniokwadratowy. Im wyższy współczynnik uczenia, tym powstały błąd jest mniejszy.

Dla mniejszych współczynników uczenia błąd jest większy, co może wpływać na samo uczenie się sieci (nauka może trwać dłużej właśnie ze względu na powstały błąd).

Dla przypadku z DeltaRule błąd uczenia nie zmniejsza się jak przy modelu Adaline.



Z powyższego wykresu wynika, że sieć wykorzystująca model Adaline daje poprawniejsze wyniki aniżeli sieć z modelem DeltaRule. Model Adaline miał zawsze mniej lub taką samą ilość błędów co model DeltaRule.

Wnioski:

Skuteczność procesu uczenia zależy od współczynnika uczenia. Wraz z jego wzrostem proces uczenia jest poprawniejszy. Wartość jest większa tym przyrost wag, które na samym początku są niewielkie jest szybszy, więc proces uczenia przebiega szybciej. Sieć jednowarstwowa jest odporna na zaszumienie, jednakże tylko dla kilku bitów. Zbyt duże zaszumienie powoduje błędne odpowiedzi dawane przez sieć (w szczególności dla modelu DeltaRule). Przy projektowaniu sieci neuronowej trzeba wybrać odpowiedni model. Na jej efektywność również ma wpływ sama jej struktura, np. zastosowana funkcja aktywacji. Model Adaline pozwala na o wiele sprawniejsze uczenie sieci w porównaniu do modelu DeltaRule.

Kod:

Source.cpp

```
#include "stdafx.h"
#include "DeltaRule.h"
#include "Adaline.h"

int main()
{
    //Perceptron.h nie jest uzywana (sluzy tylko do zalaczania bibliotek i trzymania define)
    double learningRate = 0.1; //wspolczynnik uczenia

    cout << "Learning Rate: " << learningRate << endl;

    //Adaline
    Adaline adaline(learningRate);
    adaline.learn();
    adaline.test();

    //DeltaRule
    DeltaRule deltaRule(learningRate);
    deltaRule.learn();
    deltaRule.test();

    system("PAUSE");
    return 0;
}
```

DeltaRule.cpp

```
#include "stdafx.h"
#include "DeltaRule.h"

//konstruktor
DeltaRule::DeltaRule(double _learningRate) {
    delta = 0;
    numberOfWeights = BITS_OF_ONE_LETTER;
    numberOfSets = HOW_MANY_LETTERS;
    learningRate = _learningRate;
    error = 0;
    output = 0;
    weights = new double[numberOfWeights];

    for (int i = 0; i < BITS_OF_ONE_LETTER; i++)
        this->weights[i] = getRandomDouble();

    //wczytanie z pliku danych uczacych
    readTestData();
}

//losuje double'a z przedzialu <0;1>
double DeltaRule::getRandomDouble()
{
    double randValue = ((double)rand() / (double)RAND_MAX);
    return randValue;
}

//wczytanie danych uczacych z pliku
void DeltaRule::readTestData()
{
    fstream file;
    file.open("data_for_learning.txt");

    if (!file.good()) {
        cout << "----- I can't open the file with learning data -----" << endl;
        system("PAUSE");
        exit(0);
    }
}
```

```

//wczytaj z pliku dopoki są dane
while (!file.eof())
    for (int i = 0; i < HOW_MANY_LETTERS; i++) { // i oznacza indeks litery
        for (int j = 0; j < BITS_OF_ONE_LETTER; j++) // j oznacza ilość bitów na daną literę
            file >> this->inputData[i][j]; //wczytywanie do tablicy z wejściami
        file >> this->expectedResults[i]; //wczytanie z pliku czy dana litera jest duża (1) lub mała (0)
    }
file.close();
}

//funkcja aktywacji - funkcja sigmoidalna
double DeltaRule::activationFunction(double sum) {
    //Współczynnik beta = 1.0
    return (1 / (1 + exp(-1.0 * sum)));
}

//pochodna funkcji aktywacji
double DeltaRule::derivativeActivationFunction(double sum)
{
    return (1.0*exp(-1.0*sum)) / (pow(exp(-1.0*sum) + 1, 2));
}

//zwraca sumę danego wejścia
double DeltaRule::getSum(int letter[], double * weights)
{
    double sum = 0.0;
    for (int i = 0; i < numberOfWeights; i++)
        sum += letter[i] * weights[i];
    return sum;
}

//funkcja ucząca
void DeltaRule::learn() {
    cout << endl << "DELTARULE SIGMOID - LEARNING" << endl;

    bool acceptableError = false; //zmienna, stwierdzająca czy błąd jest możliwy do zaakceptowania

    int epoch = 0; //numer epoki

    /*
    for (int i = 0; i < numberOfWeights; i++)
        cout << "Weights are: w" << i+1 << " = " << weights[i] << endl;
    cout << endl;
    */

    do {
        epoch++; //zwiększenie numeru epoki
        error = 0.0; //zerowanie głównego błędu w celu sprawdzenia błędów podczas jednej iteracji
        for (int i = 0; i < numberOfSets; i++) {

            //wynik otrzymany
            output = activationFunction(getSum(inputData[i], weights));

            //obliczanie różnicy pomiędzy wynikiem oczekiwanym a wynikiem otrzymanym
            delta = expectedResults[i] - output;

            //aktualizowanie wag
            for (int j = 0; j < numberOfWeights; j++)
                weights[j] += learningRate*delta*inputData[i][j] * derivativeActivationFunction(getSum(inputData[i], weights));

            //aktualizowanie błędu głównego
            error += delta*delta;
        }
        error /= 2;

        //porównywanie błędu z progiem
        if (error > 0.1)
            acceptableError = false;
        else
            acceptableError = true;
    } while (!acceptableError);
}

```

```

        cout << "Epoch: " << epoch << endl;
        cout << "MSE error: " << error << endl;
        /*
        for (int i = 0; i < numberOfWeights; i++)
            cout << "Weights are: w" << i+1 << " = " << weights[i] << endl;
        */
    }

//funkcja testujaca
void DeltaRule::test()
{
    cout << "Test letters: " << endl;
    for (int i = 0; i < numberOfSets; i++) {
        cout << "Letter " << setTestLetters[i] << " is: ";
        if (activationFunction(getSum(setTest[i], weights)) > 0.5) {
            cout << "big";
        }
        else {
            cout << "small";
        }
        cout << endl;
    }
}

```

Adaline.cpp

```

#include "stdafx.h"
#include "Adaline.h"

Adaline::Adaline (double _learningRate)
{
    delta = 0;
    numberOfWeights = BITS_OF_ONE_LETTER;
    numberOfSets = HOW_MANY_LETTERS;
    learningRate = _learningRate;
    error = 0.0;
    weights = new double[numberOfWeights];

    for (int i = 0; i < BITS_OF_ONE_LETTER; i++)
        this->weights[i] = getRandomDouble();

    //wczytanie z pliku danych uczacych
    readTestData();
}

//wczytanie danych uczacych z pliku
void Adaline::readTestData()
{
    fstream file;
    file.open("data_for_learning.txt");

    if (!file.good()) {
        cout << "----- I can't open the file with learning data -----" << endl;
        system("PAUSE");
        exit(0);
    }

    //wczytaj z pliku dopóki są dane
    while (!file.eof())
        for (int i = 0; i < HOW_MANY_LETTERS; i++) { // i oznacza indeks litery
            for (int j = 0; j < BITS_OF_ONE_LETTER; j++) // j oznacza ilość bitów na daną literę
                file >> this->inputData[i][j]; //wczytywanie do tablicy z wejściami

            file >> this->expectedResults[i]; //wczytanie z pliku czy dana litera jest duża (1) lub mała (0)
        }
}

```

```

    }

    file.close();
}

//losuje double'a z przedzialu <0;1>
double Adaline::getRandomDouble()
{
    double randValue = ((double)rand() / (double)RAND_MAX);
    return randValue;
}

//funkcja aktywacji - funkcja progowa unipolarna
bool Adaline::activationFunction(double sum)
{
    if (sum > 0.5)
        return true;
    else
        return false;
}

//zwraca sume danego wejścia
double Adaline::getSum(int letter[], double * weights)
{
    double sum = 0.0;
    for (int i = 0; i < numberOfWeights; i++)
        sum += letter[i] * weights[i];
    return sum;
}

//funkcja ucząca
void Adaline::learn()
{
    cout << endl << "ADALINE - LEARNING" << endl;

    bool acceptableError = false; //zmienna, stwierdzająca czy błąd jest możliwy do zaakceptowania

    int epoch = 0; //numer epoki

    /*
    for (int i = 0; i < numberOfWeights; i++)
        cout << "Weights are: w" << i+1 << " = " << weights[i] << endl;;
    cout << endl;
    */

    do {
        epoch++; //zwiększenie epoki
        error = 0.0; //zerowanie głównego błędu w celu sprawdzenia błędów podczas jednej iteracji
        for (int i = 0; i < numberOfSets; i++) {

            //obliczanie różnicy pomiędzy wynikiem oczekiwanym a wynikiem otrzymanym
            delta = expectedResults[i] - getSum(inputData[i], weights);

            //aktualizowanie wag
            for (int j = 0; j < numberOfWeights; j++)
                weights[j] += learningRate*delta*inputData[i][j];

            //aktualizowanie błędu głównego
            error += delta*delta;
        }
        error /= 2;

        //porównywanie błędu z progiem
        if (error > 0.1) {
            acceptableError = false;
        }
        else {
            acceptableError = true;
        }
    } while (!acceptableError);

    cout << "Epoch: " << epoch << endl;
    cout << "MSE error: " << error << endl;
}

```

```
//funkcja testujaca
void Adaline::test()
{
    cout << "Test letters:" << endl;
    for (int i = 0; i<numberOfSets; i++) {
        cout << "Letter " << setTestLetters[i] << " is: ";
        if (activationFunction(getSum(setTest[i], weights))) {
            cout << "big";
        }
        else {
            cout << "small";
        }
        cout << endl;
    }
}
```