**Sprawozdanie z projektu nr 3** Monika Pawlik

**Temat ćwiczenia:**   
Budowa i działanie sieci wielowarstwowej

**Cel ćwiczenia:**  
Celem ćwiczenia jest poznanie budowy i działania wielowarstwowych sieci neuronowych poprze uczenie z użyciem algorytmu wstecznej propagacji błędu rozpoznawania konkretnych liter alfabetu.

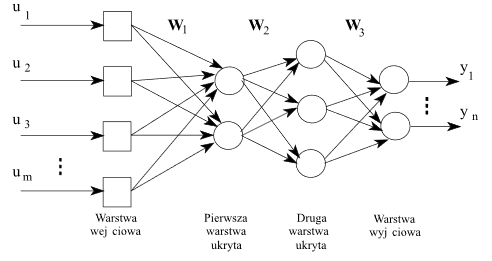
1. **Syntetyczny opis budowy wielowarstwowej sieci neuronowej**

Ten typ sieci cieszy się największym zainteresowaniem. Jej cechą charakterystyczną jest występowanie co najmniej jednej warstwy ukrytej neuronów, pośredniczącej w przekazywaniu sygnałów pomiędzy wejściami a wyjściami sieci. Wielowarstwowy perceptron jest siecią, w której neurony pogrupowane są w warstwy.

Można wyróżnić warstwę wejściową, jedną i więcej warstw ukrytych oraz warstwę wyjściową. Zadaniem warstwy wejściowej jest wstępne przetwarzanie danych wejściowych u=[u1,u2,…,up]T, np. skalowanie, filtracja czy normalizacja sygnałów oraz przekazanie ich do elementów pierwszej warstwy przetwarzania neuronowego. Główne przetwarzanie neuronowe odbywa się w kolejnych warstwach ukrytych i warstwie wyjściowej. Odpowiedź sieci neuronowej y otrzymuje się na wyjściu neuronów warstwy wyjściowej. Ogólnie proces przetwarzania neuronowego można opisać zależnością:

y = F3{W3F2 [W2F1(W1u)] }

gdzie F1,F2,F3 są nieliniowymi operatorami opisującymi transformację neuronową sygnału przez kolejne warstwy neuronowe, W1, W2, W3 oznaczają macierze współczynników wagowych, które określają przepustowość połączeń między neuronami dwóch sąsiednich warstw, u i y oznaczają odpowiednio wektory wejściowy i wyjściowy.



**Rys. Trójwarstwowa sieć neuronowa**

1. **Algorytm wstecznej propagacji błędu**

Podstawowym algorytmem uczenia sieci perceptronowych jest algorytm wstecznej propagacji błędów. Algorytm ten określa zasady uaktualniania wartości wagowych połączeń między neuronowych sąsiednich warstw sieci. Jest to procedura iteracyjna rozwiązująca zadanie minimalizacji przyjętego wskaźnika jakości wykorzystując metodę gradientową największego spadku. Modyfikacja wartości współczynników wagowych odbywa się zgodnie z zależnością:

gdzie w(k) oznacza wektor współczynników wagowych w chwili k, n jest krokiem uczenia oraz E(w(k)) oznacza gradient wskaźnika jakości E względem wektora wag w. Opisana sieć neuronowa jest siecią statyczną i można ją z powodeniem stosować do aproksymacji dowolnej nieliniowej, statycznej funkcji.

**Krok 1.** Dla każadej warstwy wyznacza się aktualne aktywności wszystkich elementów przetwarzających zgodnie z poniższą zależnością:



**Krok 2.** Błąd warstwy wyjściowej jest rzutowany wstecz zgodnie z poniższymi zależnościami:

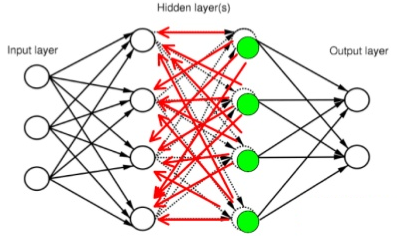
- dla warstwy wyjściowej:

- dla warstw ukrytych:



**Krok 3.** Korygujemy parametry sieci zgodnie z zależnością:



**Krok 4.** Sprawdzenie warunku stopu. Jeżeli jest spełniony to kończymy proces uczenia. W innym przypadku wykonujemy kroki 1-3. Warunkiem stopu jest zazwyczaj wykonanie określonej z góry liczby prezentacji wzorców uczących lub przekroczenie zamierzonej granicy błędu.

**Rys. Schemat algorytmu wstecznej propagacji błędów**

1. **Wygenerowanie danych uczących i testujących**

Dane zawierają 20 dużych liter alfabetu łacińskiego w postaci dwuwymiarowej tablicy dla jednej litery. Prawidłowe rozpoznanie litery będzie polegało na uaktywnieniu jednego z 20 wyjść sieci, lub więcej w przypadku rozpoznania w prezentowanym wektorze cech kilku liter.

Przykładowa tablica liter:

„A” 0 1 1 1 0 1 0 0 0 1 1 0 0 0 1 1 1 1 1 1 1 0 0 0 1 1 0 0 0 1 1 0 0 0 1

„B” 1 1 1 1 0 1 0 0 0 1 1 0 0 0 1 1 1 1 1 0 1 0 0 0 1 1 0 0 0 1 1 1 1 1 0

1. **Zestawienie otrzymanych wyników**

**Tabela 1** przedstawia wyniki w zależności od współczynnika uczenia

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Współczynnik uczenia** | 0,2 | 0,4 | 0,6 | 0,8 | 1 |
| **Wartość rozpoznania** | 0,977 | 0,984 | 0,987 | 0,989 | 0,999 |
| **Czas algorytmu [s]** | 1,7 | 1,3 | 0.9 | 0.7 | 0.6 |

Im wyższy współczynnik uczenia, tym litery zostały szybciej i lepiej rozpoznane. Zbyt mały współczynnik proporcjonalności cechuje bardzo wolne, nieefektywne uczenie.

**Tabela 2** przedstawia różne wartości współczynnika bezwładności

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Wartość współczynnika bezwładności** | 0,5 | 0,8 | 0,9 | 1,1 |
| **Ilość potrzebnych iteracji** | 9999 | 7500 | 6900 | 8050 |

Sieć najlepiej się nauczyła dla współczynnika 0,9. Podczas testów okazało się, że współczynnik bezwładności nie może być za duży, ponieważ sieć będzie potrzebowała więcej iteracji do nauki.

1. **Analiza i dyskusja błędów uczenia i testowania opracowanej sieci w zależności od wartości współczynnika uczenia i bezwładności**

Wraz z upływem czasu błąd uczenia maleje i błąd testowania również. Różne wartości współczynnika uczenia mogą wywierać na przyspieszenie procesu uczenia. W opracowanej sieci im wyższy współczynnik, tym sieć szybciej się uczy.

Algorytm wstecznej propagacji błędu wykonuje wygładzanie poprawek zbioru wag, uzależnione od wartości współczynnika wygładzania. Analiza wykazuje, że wprowadzenie do algorytmu wartości bezwładności = 0,9 powoduje zwiększenie wartości poprawki wag. Jednak dla współczynnika powyżej 1 została ograniczona wielkość poprawek, co oznacza zbyt silną bezwładność i sieć uczy się trochę wolniej.

1. **Sformułowanie wniosków**

Algorytm wstecznej propagacji błędu wykazuje silną zależność pomiędzy przebiegiem procesu trenowania sieci, a wartością współczynnika uczenia. Zbyt mała jego wartość powoduje zwiększenie się liczby potrzebnych iteracji i tym samym czasu wymaganego na zakończenie treningu. Algorytm ma ponadto w takim przypadku tendencję to wygasania w minimach lokalnych funkcji celu. Ustalenie z kolei zbyt dużej wartości kroku grozi wystąpieniem oscylacji wokół poszukiwanego minimum, prowadzącej ostatecznie do zatrzymania procesu uczenia bez osiągniecie wymaganej wartości funkcji celu.

W procesie uczenia sieci ważna jest inicjalizacja początkowego układu wag połączeń między elementami sieci. Najskuteczniejszym rozwiązaniem jest losowy dobór wag, co praktycznie całkowicie zabezpiecza przed problemem z symetrią.

Wybór algorytmu wstecznej propagacji błędów – dzięki zastosowaniu specyficznego sposobu propagowania błędów uczenia sieci powstałych na jej wyjściu, tj. przesyłania ich od warstwy wyjściowej do wejściowej, algorytm propagacji wstecznej stał się jednym z najskuteczniejszych algorytmów uczenia sieci.

1. **Listing całego kodu oraz zrzuty konfiguracji i wykorzystania programu**

Screen działającego programu:



**Opis implementacji:**

Implementacja została wykonana w języku C#. Głównymi klasami są Program, wywołująca metody aplikacji i związana z interfejsem użytkownika oraz klasa Neutron odpowiedzialną za logikę programu.

Klasa Neutron inicjuje warstwy sieci oraz wagi losowymi wartościami. Zawiera funkcje:  
TrainNetwork – trenującą sieć,   
CalculateOutput – obliczającą wejście, wyjście, wartość oczekiwaną ora błąd pierwszej warstwy, BackPropagation – metoda poprawiająca wagi warstwy wejściowej,  
GetError – zwracająca błędy.

Warstwa wejściowa zawiera wagi oraz wartość. Warstwa wyjściowa: sumę wejść, wyjście, błąd i wartość. Implementacja:

public struct InputLayer

{

public double Value;

public double[] Weights;

}

public struct OutputLayer

{

public double InputSum;

public double Output;

public double Error;

public double Target;

public string Value;

}

**Listing kodu:**

class Program

{

private readonly Network \_network = new Network();

private readonly string[] \_literki = {"A","B","C","D","E","F","G","H","I","J",   
 "K", "L", "M", "N", "O", "P", "R", "S", "T", "U" };

private static int imageSize = 5;

static void Main(string[] args)

{

var p = new Program();

p.\_network.maximumIteration = int.Parse(10000.ToString());

p.Test();

p.Check();

Console.ReadKey();

}

private void Test()

{

int length = \_literki.Length;

\_network.Initialize(imageSize \* imageSize, length);

double[][] inputs = new double[length][];

double[][] outputs = new double[length][];

for (int i = 0; i < length; ++i)

{

outputs[i] = new double[length];

for (int j = 0; j < length; ++j)

{

outputs[i][j] = i == j ? 1.0 : 0.0;

}

\_network.outputLayer[i].Value = \_literki[i];

}

inputs[0] = new double[25] {1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 1, 0, 0, 0, 1};

inputs[1] = new double[25] {1, 1, 1, 0, 0, 1, 0, 0, 1, 0, 1, 1, 1, 0, 0, 1, 0, 0, 1, 0, 1, 1, 1, 0, 0};

inputs[2] = new double[25] {1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 1, 1, 1};

inputs[3] = new double[25] {1, 1, 1, 0, 0, 1, 0, 1, 0, 0, 1, 0, 1, 0, 0, 1, 0, 1, 0, 0, 1, 1, 1, 0, 0};

inputs[4] = new double[25] {1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 1};

inputs[5] = new double[25] {1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0};

inputs[6] = new double[25] {1, 1, 1, 1, 0, 1, 0, 0, 0, 0, 1, 0, 1, 1, 0, 1, 0, 0, 1, 0, 1, 1, 1, 1, 0};

inputs[7] = new double[25] {1, 0, 0, 0, 1, 1, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 1, 0, 0, 0, 1};

inputs[8] = new double[25] {1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0};

inputs[9] = new double[25] {1, 1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 1, 1, 1, 0, 0};

inputs[10] = new double[25] {1, 1, 1, 0, 0, 1, 0, 1, 0, 0, 1, 0, 1, 0, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0};

inputs[11] = new double[25] {1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 1, 0, 0, 1, 0, 1, 0, 0, 1, 1, 1, 0, 0};

inputs[12] = new double[25] {0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0};

inputs[13] = new double[25] {0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1, 1, 1, 0, 0, 1, 0, 1, 0, 0, 1, 1, 1, 0, 0};

inputs[14] = new double[25] {0, 1, 1, 1, 0, 0, 1, 0, 1, 0, 0, 1, 1, 1, 0, 0, 1, 0, 0, 0, 0, 1, 1, 1, 0};

inputs[15] = new double[25] {0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 1, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0};

inputs[16] = new double[25] {1, 1, 1, 0, 0, 1, 0, 1, 0, 0, 1, 1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 1, 0, 0};

inputs[17] = new double[25] {0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 1, 0, 0, 1, 0, 1, 0, 0, 1, 0, 1, 0};

inputs[18] = new double[25] {0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0};

inputs[19] = new double[25] {0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 1, 0, 0};

\_network.TrainNetwork(inputs, outputs);

Console.WriteLine("Błąd rozpoznania :");

for (int i = 0; i < \_network.currentIteration; ++i)

{

Console.WriteLine( i + " - "+   
 \_network.errors[i].ToString("#0.000000"));

}

}

private void Check()

{

double[] sample = new double[imageSize \* imageSize];

sample = new double[25] { 1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1,   
0, 0, 0, 1, 1, 0, 0, 0, 1 };

\_network.Recognize(sample);

Console.WriteLine("/nWyniki:");

var wyniki = new double[\_literki.Length];

for (int i = 0; i < \_network.outputLayer.Length; ++i)

{

wyniki[i] = \_network.outputLayer[i].Output;

Console.WriteLine(\_network.outputLayer[i].Value + " - "   
 +\_network.outputLayer[i].Output.ToString("#0.000000"));

}

int w = wyniki.ToList().IndexOf(wyniki.Max());

if(w > \_literki.Length/2)

Console.WriteLine("\nZostała wykryta litera: " +   
 \_network.outputLayer[w].Value);

else

{

Console.WriteLine("\nZostała wykryta litera: " +   
 \_network.outputLayer[w].Value);

}

}

}

class Network

{

public struct InputLayer

{

public double Value;

public double[] Weights;

}

public struct OutputLayer

{

public double InputSum;

public double Output;

public double Error;

public double Target;

public string Value;

}

public double learningRate = 1;

public int ImageSize = 0;

public int InputNum = 0;

public int OutputNum = 0;

public InputLayer[] inputLayer = null;

public OutputLayer[] outputLayer = null;

public double[] errors = null;

public int currentIteration = 0;

public int maximumIteration = 1000;

public void Initialize(int inputSize, int outputSize)

{

InputNum = inputSize;

OutputNum = outputSize;

inputLayer = new Network.InputLayer[inputSize];

outputLayer = new Network.OutputLayer[outputSize];

// Zainicjuj wagi losowymi wartościami z zakresu 0.01 - 0.02.

Random random = new Random();

for (int i = 0; i < InputNum; ++i)

{

inputLayer[i].Weights = new double[OutputNum];

for (int j = 0; j < OutputNum; ++j)

{

inputLayer[i].Weights[j] = random.Next(1, 3) / 100.0;

}

}

}

public bool TrainNetwork(double[][] inputs, double[][] outputs)

{

double currentError = 0.0, maximumError = 0.01;

currentIteration = 0;

// Utwórz tablicę do przechowywania wartości kolejnych błędów.

errors = new double[maximumIteration];

do

{

currentError = 0;

for (int i = 0; i < inputs.Length; ++i)

{

CalculateOutput(inputs[i], outputLayer[i].Value);

BackPropagation();

currentError += GetError();

}

errors[currentIteration] = currentError;

++currentIteration;

}

while (currentError > maximumError && currentIteration <   
 maximumIteration);

if (currentIteration <= maximumIteration)

{

return true;

}

return false;

}

private void CalculateOutput(double[] pattern, string output)

{

for (int i = 0; i < pattern.Length; i++)

{

inputLayer[i].Value = pattern[i];

}

for (int i = 0; i < OutputNum; i++)

{

double total = 0.0;

for (int j = 0; j < InputNum; j++)

{

total += inputLayer[j].Value \* inputLayer[j].Weights[i];

}

outputLayer[i].InputSum = total;

outputLayer[i].Output = Activation(total);

outputLayer[i].Target = outputLayer[i].Value.CompareTo(output) == 0 ?   
 1.0 : 0.0;

outputLayer[i].Error = (outputLayer[i].Target - outputLayer[i].Output)   
 \* (outputLayer[i].Output) \* (1 - outputLayer[i].Output);

}

}

private void BackPropagation()

{

for (int j = 0; j < OutputNum; j++)

{

for (int i = 0; i < InputNum; i++)

{

inputLayer[i].Weights[j] += learningRate \* (outputLayer[j].Error)   
 \* inputLayer[i].Value;

}

}

}

private double GetError()

{

double total = 0.0;

for (int j = 0; j < OutputNum; j++)

{

total += Math.Pow((outputLayer[j].Target - outputLayer[j].Output),   
 2.0) / 2.0;

}

return total;

}

private double Activation(double x)

{

return (1.0 / (1.0 + Math.Exp(-x)));

}

public void Recognize(double[] input)

{

for (int i = 0; i < InputNum; i++)

{

inputLayer[i].Value = input[i];

}

for (int i = 0; i < OutputNum; i++)

{

double total = 0.0;

for (int j = 0; j < InputNum; j++)

{

total += inputLayer[j].Value \* inputLayer[j].Weights[i];

}

outputLayer[i].InputSum = total;

outputLayer[i].Output = Activation(total);

}

}

}