**Podstawy Sztucznej Inteligencji**

Scenariusz 3: Budowa i działanie sieci wielowarstwowej

Wszystkie projekty znajdują się w jednym programie na repozytorium

**Wykonał:**

**Bartłomiej Leja**

**Wydział Inżynierii Metali i Informatyki Przemysłowej**

**Informatyka Stosowana**

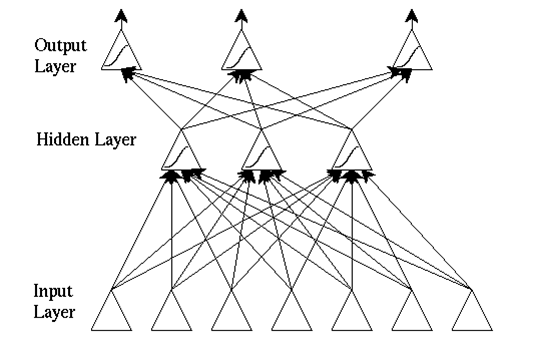
1. **Cel ćwiczenia**

Celem projektu było zapoznanie się z budową oraz działaniem wielowarstwowej sieci neuronowej poprzez uczenie z użyciem algorytmu wstecznej propagacji błędu rozpoznawania konkretnych liter alfabetu.

1. **Sieć wielowarstwowa**

Sieć wielowarstwowa jest stosunkowo prosta architektura polegająca na utworzeniu kilku (zwykle 3) warstw neuronów połączonych w ten sposób, że wyjścia neuronów należących do warstwy niższej połączone są z wejściami neuronów należących do warstwy wyższej (każdy z każdym), pozwala na tworzenie sieci o niemal dowolnej charakterystyce. Działanie takiej sieci polega na liczeniu odpowiedzi neuronów w kolejnych warstwach - najpierw w pierwszej, do której trafiają sygnały z wejść sieci, potem (na podstawie wyników pierwszej warstwy) liczymy odpowiedzi drugiej warstwy neuronów itd., przy czym odpowiedzi ostatniej warstwy traktowane są jako wyjścia z sieci. Taka sieć składa się z trzech typów warstw, które zostały pokazane na rys. 1:

* warstwa pierwsza - wejściowa (jednostki w tej warstwie są to jednostki wejściowe),
* warstwa ostatnia - wyjściowa (jednostki w tej warstwie są to jednostki wyjściowe),
* warstwy pomiędzy wejściową a wyjściową - ukryte (jednostki w tych warstwach są to jednostki ukryte).



Rys. 1

1. **Algorytm wstecznej propagacji błędu**

Algorytm wstecznej propagacji błędu jest podstawowy algorytm uczenia nadzorowanego wielowarstwowych, jednokierunkowych sieci neuronowych. Podaje on przepis na zmianę wag dowolnych połączeń elementów przetwarzających rozmieszczonych w sąsiednich warstwach sieci. Oparty jest on na minimalizacji sumy kwadratów błędów (lub innej funkcji błędu) uczenia z wykorzystaniem optymalizacyjnej metody największego spadku. 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.

Ogólny schemat procesu trenowania sieci wygląda następująco:

1. Ustalamy topologię sieci, tzn. liczbę warstw, liczbę neuronów w warstwach.
2. Inicjujemy wagi losowo (na małe wartości).
3. Dla danego wektora uczącego obliczamy odpowiedź sieci (warstwa po warstwie).
4. Każdy neuron wyjściowy oblicza swój błąd, oparty na różnicy pomiędzy obliczoną odpowiedzią *y* oraz poprawną odpowiedzią *t*.
5. Błędy propagowane są do wcześniejszych warstw.
6. Każdy neuron (również w warstwach ukrytych) modyfikuje wagi na podstawie wartości błędu i wielkości przetwarzanych w tym kroku sygnałów.
7. Powtarzamy od punktu 3. dla kolejnych wektorów uczących. Gdy wszystkie wektory zostaną użyte, losowo zmieniamy ich kolejność i zaczynamy wykorzystywać powtórnie.
8. **Zestaw danych uczących**

Dane uczące zostały zawarte w pliku letterLearningData.txt, który zawiera 20 liter polskiego alfabetu w postaci 35 elementowych wektorów utworzonych na bazie matryc 7x5. Dla przykładu litera A to:

**0 1 1 1 0  
1 0 0 0 1  
1 0 0 0 1  
1 1 1 1 1** 🡪 **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  
1 0 0 0 1   
1 0 0 0 1  
1 0 0 0 1**

Ponadto w pliku tym do każdej z 20 liter przyporządkowany jest wektor składający się z 20 cyfr: dziewiętnastu zer i jednej jedynki. Wektor ten służy do rozpoznania jaka to litera: dla przykładu literze A odpowiada wektor:

**1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0**

1. **Uzyskane wyniki**

Program stworzony na potrzeby tego projektu poddano testom polegającym na rozpoznaniu 2 liter z różną liczbą iteracji uczenia oraz różnym współczynnikiem uczenia. W wszystkich testach wykorzystano funkcję aktywacji w postaci tangensa hiperbolicznego.

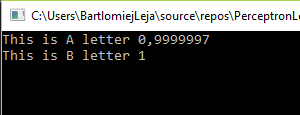
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Numer  próby | Współczynnik  uczenia η | Liczba danych  uczących | Liczba potrzebnych  iteracji do wyuczenia | Liczba błędów przy testowaniu |
| 1 | 0,01 | 20 | 1000 | 0 |
| 2 | 0,1 | 20 | 1000 | 0 |
| 3 | 0,2 | 20 | 1000 | 0 |
| 4 | 0,4 | 20 | 1000 | 0 |
| 5 | 0,01 | 20 | 5000 | 0 |
| 6 | 0,1 | 20 | 5000 | 0 |
| 7 | 0,2 | 20 | 5000 | 0 |
| 8 | 0,4 | 20 | 5000 | 0 |

Tab. 1

|  |  |  |
| --- | --- | --- |
| Numer próby | Wynik dla litery A | Wynik dla litery B |
| 1 | 0,93 | 0,92 |
| 2 | 0,95 | 0,94 |
| 3 | 0,96 | 0,96 |
| 4 | 0,96 | 0,97 |
| 5 | 0,98 | 0,98 |
| 6 | 0,97 | 0,97 |
| 7 | 0,99 | 0,99 |
| 8 | 1 | 1 |

Tab. 2

Przykładowy wydruk.



Rys. 2

Wykres zależności współczynnika uczenia od wyniku litery A dla 1000 powtórzeń uczenia.

Rys. 3

Wykres zależności współczynnika uczenia od wyniku litery A dla 5000 powtórzeń uczenia.

Rys. 4

1. **Wnioski**

Uzyskane wyniku pokazują iż sieć wielowarstwowa z algorytmem propagacji wstecznej jest bardzo skuteczna w pewnych wariantach można obserwować 100% pewność programu, że dana litera jest poprawna. Na podanych wykresach oraz w tabelach widać, iż czym więcej mamy iteracji uczenia oraz czym wyższy jest współczynnik uczenia uzyskujemy dokładniejszy wynik. Można z tego wywnioskować, iż nie zaszło w żadnym testowanym przypadku do przeuczenia algorytmu. Należy też wspomnieć o trudnościach doboru ilości warstw. W tym przypadku zostały stworzone 4 warstwy 2 ukryte jedna wyjścia i jedna wejścia i jak widać po wynikach był to dobry wybór. Dodatkowo jako ciekawostkę zauważyłem, iż przy 5 tysiącach powtórzeń uczenia zauważalna jest długość wykonywania, co świadczy o dużej ilości wykonywanych operacji.

1. **Literatura**

- <http://edu.pjwstk.edu.pl/wyklady/nai/scb/wyklad3/w3.htm>

-http://edward\_ch.republika.pl/sneuro.html

- <https://pl.wikipedia.org/wiki/Propagacja_wsteczna>

- <http://www-users.mat.umk.pl/~rudy/wsn/wyk/wsn-wyklad-05a-propag.pdf>

1. **Listning kodu**

**Klasa Layer**

using System;

namespace PerceptronLerning

{

/// <summary>

/// Each individual layer in the ML

/// </summary>

class Layer

{

int numberOfInputs; //# of neurons in the previous layer

int numberOfOuputs; //# of neurons in the current layer

public float[] outputs; //outputs of this layer

public float[] inputs; //inputs in into this layer

public float[,] weights; //weights of this layer

public float[,] weightsDelta; //deltas of this layer

public float[] gamma; //gamma of this layer

public float[] error; //error of the output layer

public static Random random = new Random(); //Static random class variable

/// <summary>

/// Constructor initilizes vaiour data structures

/// </summary>

/// <param name="numberOfInputs">Number of neurons in the previous layer</param>

/// <param name="numberOfOuputs">Number of neurons in the current layer</param>

public Layer(int numberOfInputs, int numberOfOuputs)

{

this.numberOfInputs = numberOfInputs;

this.numberOfOuputs = numberOfOuputs;

//initilize datastructures

outputs = new float[numberOfOuputs];

inputs = new float[numberOfInputs];

weights = new float[numberOfOuputs, numberOfInputs];

weightsDelta = new float[numberOfOuputs, numberOfInputs];

gamma = new float[numberOfOuputs];

error = new float[numberOfOuputs];

InitilizeWeights(); //initilize weights

}

/// <summary>

/// Initilize weights between -0.5 and 0.5

/// </summary>

public void InitilizeWeights()

{

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

{

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

{

weights[i, j] = (float)random.NextDouble() - 0.5f;

}

}

}

/// <summary>

/// Feedforward this layer with a given input

/// </summary>

/// <param name="inputs">The output values of the previous layer</param>

/// <returns></returns>

public float[] FeedForward(float[] inputs)

{

this.inputs = inputs;// keep shallow copy which can be used for back propagation

//feed forwards

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

{

outputs[i] = 0;

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

{

outputs[i] += inputs[j] \* weights[i, j];

}

outputs[i] = (float)Math.Tanh(outputs[i]);

}

return outputs;

}

/// <summary>

/// TanH derivate

/// </summary>

/// <param name="value">An already computed TanH value</param>

/// <returns></returns>

public float TanHDer(float value)

{

return 1 - (value \* value);

}

/// <summary>

/// Back propagation for the output layer

/// </summary>

/// <param name="expected">The expected output</param>

public void BackPropOutput(float[] expected)

{

//Error dervative of the cost function expectet value for example 1 1 1 las one is expected

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

error[i] = outputs[i] - expected[i];

//Gamma calculation

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

gamma[i] = error[i] \* TanHDer(outputs[i]);

//Caluclating detla weights

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

{

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

{

weightsDelta[i, j] = gamma[i] \* inputs[j];

}

}

}

/// <summary>

/// Back propagation for the hidden layers

/// </summary>

/// <param name="gammaForward">the gamma value of the forward layer</param>

/// <param name="weightsFoward">the weights of the forward layer</param>

public void BackPropHidden(float[] gammaForward, float[,] weightsFoward)

{

//Caluclate new gamma using gamma sums of the forward layer

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

{

gamma[i] = 0;

for (int j = 0; j < gammaForward.Length; j++)

{

gamma[i] += gammaForward[j] \* weightsFoward[j, i];

}

gamma[i] \*= TanHDer(outputs[i]);

}

//Caluclating detla weights

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

{

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

{

weightsDelta[i, j] = gamma[i] \* inputs[j];

}

}

}

/// <summary>

/// Updating weights w11-L\*w11'

/// </summary>

public void UpdateWeights()

{

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

{

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

{

weights[i, j] -= weightsDelta[i, j] \* 0.4f;

}

}

}

}

}

Klasa BackpropagationNeuralNetwork

using System;

namespace PerceptronLerning

{

class BackpropagationNeuralNetwork

{

int[] layer; //layer information

Layer[] layers; //layers in the network

/// <summary>

/// Constructor setting up layers

/// </summary>

/// <param name="layer">Layers of this network</param>

public BackpropagationNeuralNetwork(int[] layer)

{

//deep copy layers

this.layer = new int[layer.Length];

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

this.layer[i] = layer[i];

//creates neural layers

layers = new Layer[layer.Length - 1];

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

{

layers[i] = new Layer(layer[i], layer[i + 1]);

}

}

/// <summary>

/// High level feedforward for this network

/// </summary>

/// <param name="inputs">Inputs to be feed forwared</param>

/// <returns></returns>

public float[] FeedForward(float[] inputs)

{

//feed forward

layers[0].FeedForward(inputs);

for (int i = 1; i < layers.Length; i++)

{

layers[i].FeedForward(layers[i - 1].outputs);

}

return layers[layers.Length - 1].outputs; //return output of last layer

}

public void TestMethod(float[] inputs,char letter)

{

char[] letters = { 'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j' };

int index = 0;

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

{

if (letters[i] == letter) index = i;

}

var result =FeedForward(inputs);

Console.WriteLine($"This is {letter} letter {result[index]}");

}

/// <summary>

/// High level back porpagation

/// Note: It is expexted the one feed forward was done before this back prop.

/// </summary>

/// <param name="expected">The expected output form the last feedforward</param>

public void BackProp(float[] expected)

{

// run over all layers backwards

for (int i = layers.Length - 1; i >= 0; i--)

{

if (i == layers.Length - 1)

{

layers[i].BackPropOutput(expected); //back prop output

}

else

{

layers[i].BackPropHidden(layers[i + 1].gamma, layers[i + 1].weights); //back prop hidden

}

}

//Update weights

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

{

layers[i].UpdateWeights();

}

}

}

}

Main

var dictionaryInput = new LetterDataTextFileReader();

var floatLetterInput = dictionaryInput.ConvertTextForFloatArray(@"C:\Users\BartlomiejLeja\source\repos\PerceptronLerning\PerceptronLerning\letterLearningData.txt");

var net = new BackpropagationNeuralNetwork(new int[] { 35, 35, 35, 20 });

for (int i = 0; i < 5000; i++)

{

for (int j = 0; j < floatLetterInput.Count; j++)

{

net.FeedForward(floatLetterInput[j].LetterPattern);

net.BackProp(floatLetterInput[j].LetterResult);

}

}

net.TestMethod(new float[] { 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 }, 'A');

net.TestMethod(new float[] { 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 }, 'B');