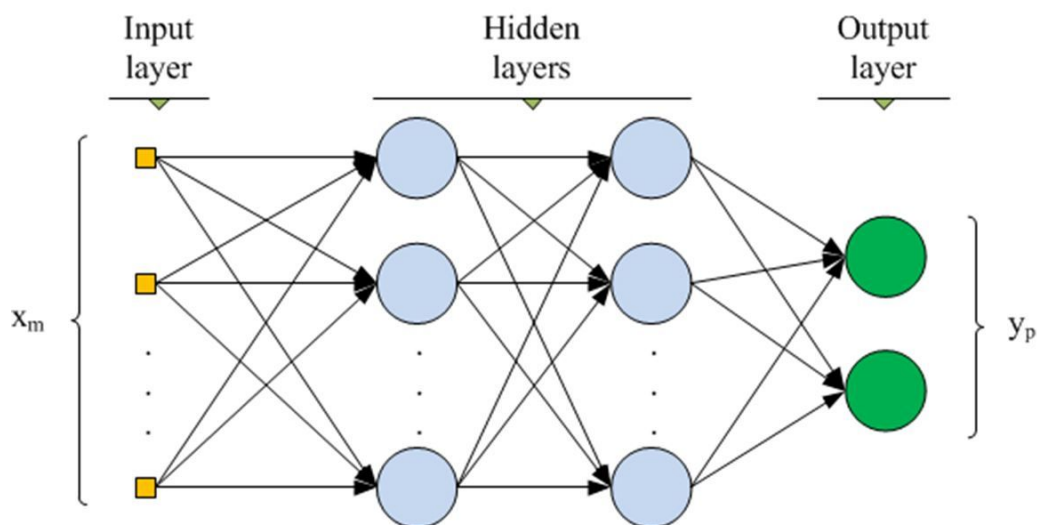


Imię Nazwisko Grupa Michał Słowikowski Gr 4	Temat Scenariusz 3	Data 24.11.2017r.
---	-----------------------	----------------------

Celem ćwiczenia była budowa sieci wielowarstwowej z użyciem algorytmu wstecznej propagacji błędu oraz naukę aproksymacji funkcji Rastrigin.

Syntetyczny opis algorytmu uczenia

Do ćwiczenia wykorzystałem sieć typu feedforward zbudowaną z różnej ilości warstw i znajdujących się w niej neuronów z sigmoidalną funkcją aktywacji.



Jako funkcję aktywacji wykorzystałem unipolarną funkcję sigmoidalną:

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

Jako algorytmu użyłem Backpropagation - wstecznej propagacji błędu.

Błąd potrzebny do korekcji wag dla każdego neuronu w ostatniej warstwie obliczałem za pomocą wzoru:

$$\frac{\partial E}{\partial W_{jk}} = O_j \delta_k$$

$$\delta_k = O_k(1 - O_k)(O_k - t_k)$$

Natomiast każdą wcześniejszą

$$\frac{\partial E}{\partial W_{ij}} = O_i \delta_j$$

$$\delta_j = O_j(1 - O_j) \sum_{k \in K} \delta_k W_{jk}$$

Korekcja wag ze wzoru:

$$\Delta W = -\eta \delta_{\ell} O_{\ell-1}$$
$$\Delta \theta = \eta \delta_{\ell}$$

, gdzie η - learning rate, δ - błąd na neuronie, O wynik z poprzedniej warstwy

Łączny błąd liczyłem z MSE, czyli

$$E_{total} = \sum \frac{1}{2} (target - output)^2$$

Dane

Dane potrzebne do zadania wyliczałem z funkcji podanej przez prowadzącego, którą można znaleźć pod tym linkiem: https://en.wikipedia.org/wiki/Rastrigin_function jako przedział przyjąłem dane dla x i y należących do przedziału $[-2, 2]$.

Wygenerowałem łącznie 1600 danych z czego 400 posłużyło mi do testowania sieci. Każdy wynik przed porównaniem przeskalowałem, żeby znajdowała się w przedziale od 0 do 1 (tak jak signum unipolarne). Przed skalowaniem obliczyłem, że największa wartość jaką przyjmuje funkcja to ~ 40 a najmniejsza to 0. Dla bezpieczeństwa przyjąłem pierwotną skalę od 0 do 50.

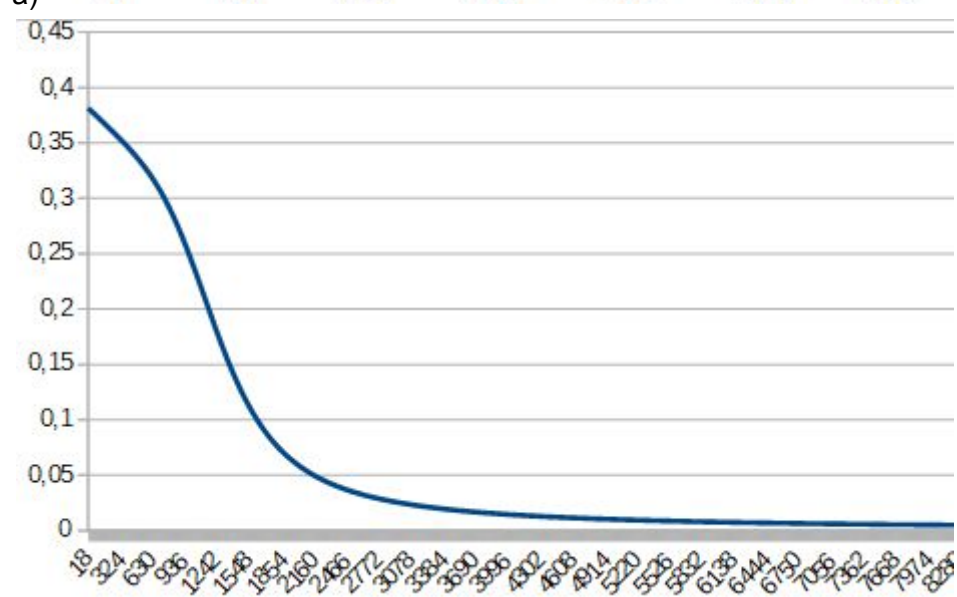
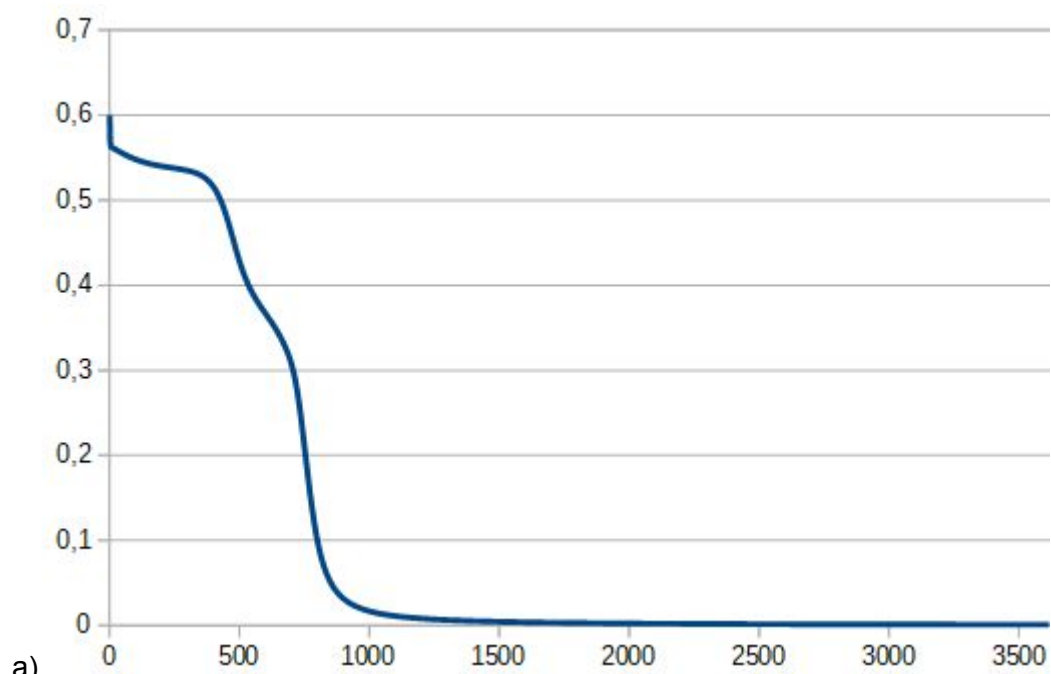
Sieci jakich użyłem to (ze względu na ilość neuronów znajdujących się w warstwie

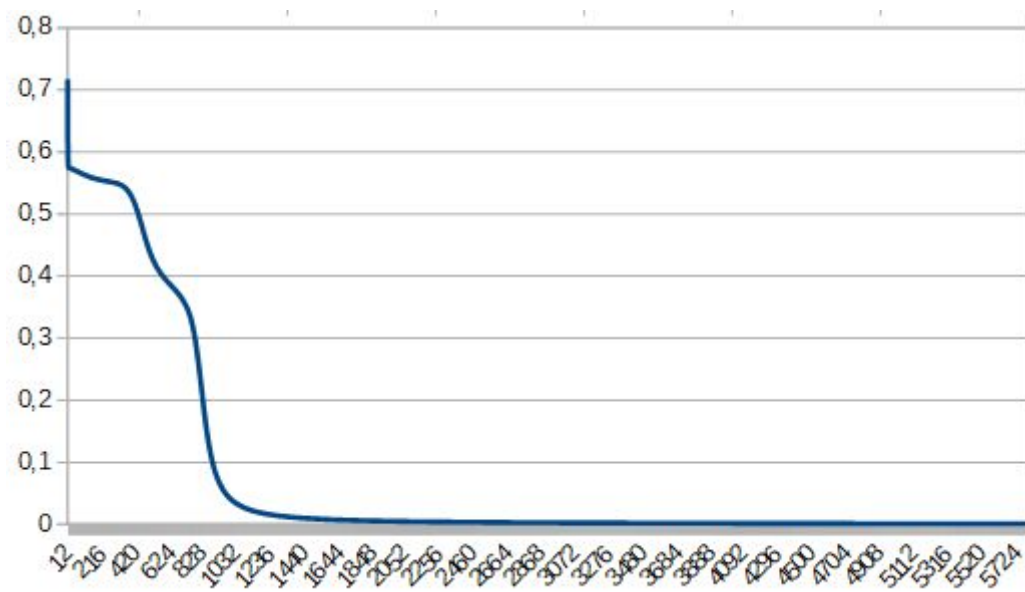
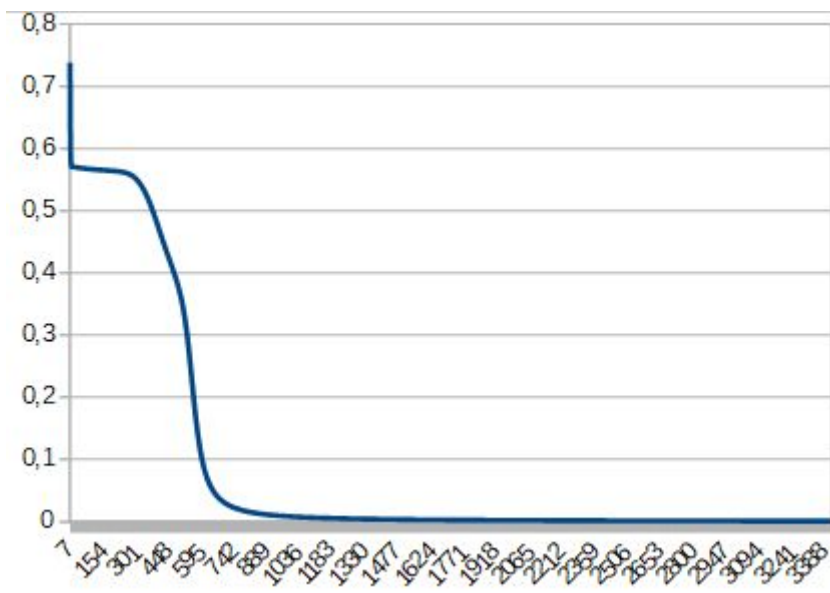
- a) 10-3-1
- b) 2-5-1
- c) 4-3-3-1

Do każdy zestaw przetestowałem z współczynnikami uczenia 0.1 i 0.01.

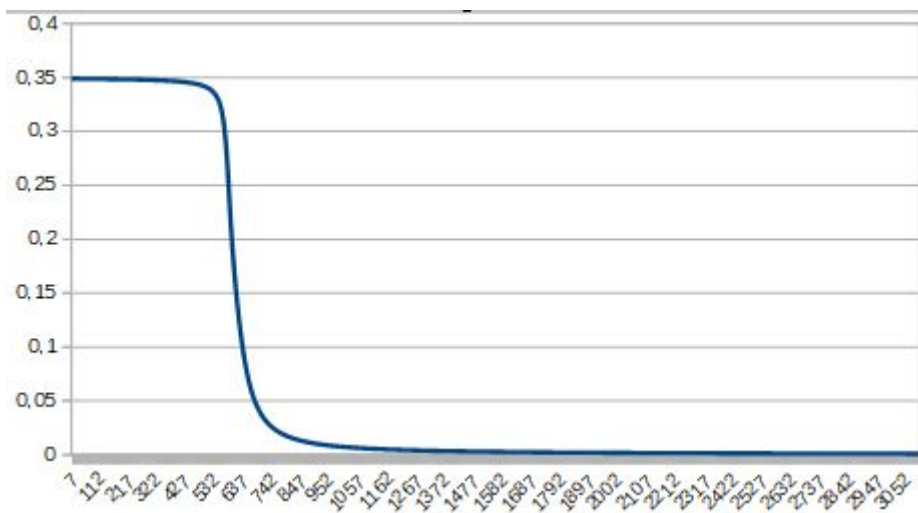
Wyniki

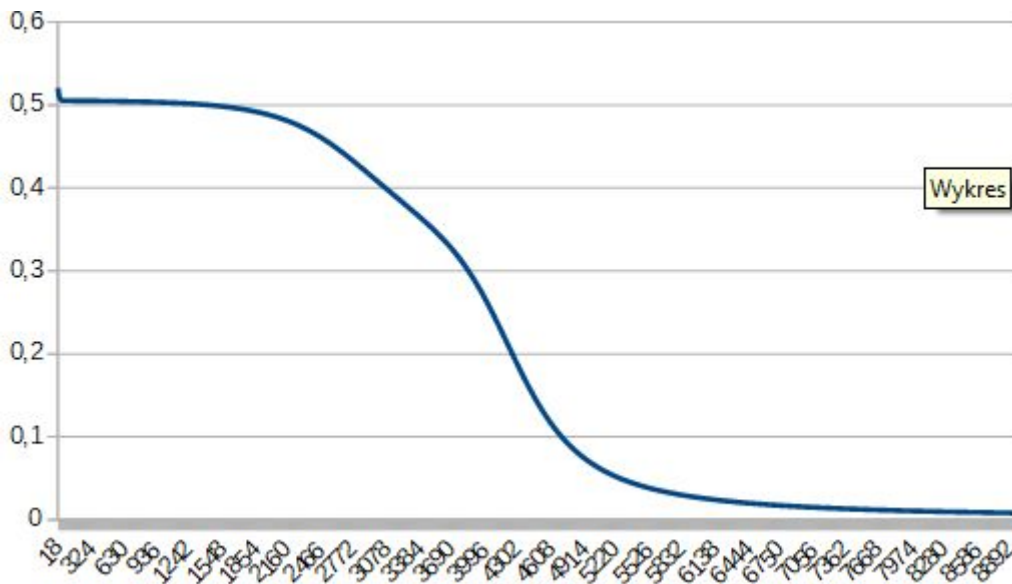
- a)





c)





Testowanie:

Podczas testowania wyliczałem średni błąd dla wartości przewidywanej i otrzymanej.

- a) 0.256%
0.245%
- b) 0.373%
0.285%
- c) 0.265%
0.254%

Analiza wyników:

Tak jak w poprzednich ćwiczeniach widać wpływ współczynnika uczenia na długość trwania tego procesu. Jak widać z wykresów wygląd funkcji zależności błędu od epoki zależy od wyglądu sieci (ilości warstw oraz ilości neuronów w tych warstwach). Ciężko określić, która z sieci wypadła najlepiej. Można jednak zauważyć zależność, że wraz z niższym współczynnikiem wykres staje się gładziej. W porównaniu do poprzednich ćwiczeń nauka sieci trwała dłużej, jednak bez tego nie byłibyśmy w stanie przewidywać skomplikowanych zjawisk.

Wnioski:

- Współczynnik uczenia ma wpływ na szybkość uczenia
- Stosunkowo wysoki błąd może wynikać z niewystarczającej ilości danych uczących lub źle obranej skali
- Wielowarstwowa sieć neuronowa pozwala aproksymować skomplikowane funkcje

Listening Kodu:

```

namespace Scen2ver2
{
    class Program
    {
        static void Main(string[] args)
        {
            int[,] struktura = new int[3, 2] { { 10, 2 }, { 3, 10 }, { 1, 3 } };
            double learningRate = 0.1;
            /*Console.WriteLine("Wygląd struktury: ");
            for(int i = 0; i < struktura.GetLength(0); i++)
            {
                Console.WriteLine("Warstwa nr: " + i + " neurony: " + struktura[i, 0] + " input: " + struktura[i, 1]);
            }*/
            double[] input = new double[2] { 1, 2 };
            Trainer t = new Trainer(struktura, learningRate);
            t.PrintNeuralNetwork();
            t.Learn();
            t.Test();

            Console.WriteLine(t.GetOutput(input));

            Console.ReadLine();
        }
    }
}

```

```

namespace Scen2ver2
{
    class Neuron
    {
        private double BIAS = 1;
        private double[] weights;

        public Neuron(double[] weights)
        {
            this.weights = new double[weights.Length];
            Array.Copy(weights, this.weights, weights.Length);
        }

        public double GetResult(double[] input)
        {
            double sum = InputSummary(input);
            return PerceptronActivation(sum);
        }

        public void Learn(double[] input, double error, double lr)
        {
            for (int i = 0; i < input.Length; i++)
                weights[i] += error * lr * input[i];

            weights[input.Length] += lr * error;
        }

        public double GetWeight(int numberOfInput)
        {
            return weights[numberOfInput];
        }

        public void PrintWeights()
        {
            for (int i = 0; i < weights.Length; i++)
            {
                if (i != weights.Length - 1)
                    Console.Write("Weight " + i);
                else
                    Console.Write("BIAS ");
                Console.WriteLine(" : " + weights[i]);
            }
        }

        public double InputSummary(double[] input)
        {
            double sum = 0;
            for (int i = 0; i < input.Length; i++)
            {
                sum += weights[i] * input[i];
            }

            sum += weights[input.Length];

            return sum;
        }

        protected double PerceptronActivation(double sum)
        {
            return 1 / (1 + Math.Exp(-sum));
        }

        protected double Derive(double x)
        {
            return (1 - PerceptronActivation(x)) * PerceptronActivation(x);
        }
    }
}

```



```
}  
public double[] CalculateOutput(double[] input)  
{  
    double[] result = new double[neurons.Length];  
    for(int i = 0; i < neurons.Length; i++)  
    {  
        result[i] = neurons[i].GetResult(input);  
    }  
    Array.Copy(result, lastOutput, lastOutput.Length);  
    return result;  
}  
  
public double[] GetWeights(int neuronNumber)  
{  
    double[] weights = new double[neurons.Length];  
    for(int i = 0; i < weights.Length; i++)  
    {  
        weights[i] = neurons[i].GetWeight(neuronNumber);  
    }  
  
    return weights;  
}  
public void CalculateNewWeights(double[] input)  
{  
    for(int i = 0; i < neurons.Length; i++)  
    {  
        neurons[i].Learn(input, errors[i], learningRate);  
    }  
    input = lastOutput;  
}  
}
```

```

namespace Scen2ver2
{
    class Layer
    {
        private Neuron[] neurons;
        private double[] errors;
        private double learningRate;
        public double[] lastOutput;
        public Layer(int numberOfNeurons, int numberOfInputs, double learningRate)
        {
            this.learningRate = learningRate;
            lastOutput = new double[numberOfNeurons];
            neurons = new Neuron[numberOfNeurons];
            errors = new double[numberOfNeurons];
            for(int i = 0; i < numberOfNeurons; i++)
            {
                errors[i] = 0.0;
            }
            Random r = new Random();
            double[] weights = new double[numberOfInputs + 1];

            for (int i = 0; i < numberOfNeurons; i++)
            {
                for (int j = 0; j < weights.Length; j++)
                {
                    weights[j] = r.NextDouble();
                }
                neurons[i] = new Neuron(weights);
            }
        }

        public void PrintLayer()
        {
            for(int i = 0; i < neurons.Length; i++)
            {
                Console.WriteLine("Neuron " + i);
                neurons[i].PrintWeights();
            }
        }

        public double[] CalculateLastError(double[] outputError)
        {
            for (int i = 0; i < errors.Length; i++)
            {
                errors[i] = lastOutput[i] * (1 - lastOutput[i]) * outputError[i];
            }
            return errors;
        }

        public double[] CalculateError(double[] errorsFromNextLayer, Layer nextLayer)
        {
            for(int i = 0; i < errors.Length; i++)
            {
                errors[i] = 0.0;
                for(int j = 0; j < errorsFromNextLayer.Length; j++)
                {
                    errors[i] = errorsFromNextLayer[j] * nextLayer.GetWeights(i)[j];
                }

                errors[i] *= lastOutput[i] * (1 - lastOutput[i]);
            }
            return errors;
        }
    }
}

```

```

public void Learn()
{
    double error;
    double totalError;
    int counter = 0;
    double output;
    double expected;
    double pivE;
    do
    {
        totalError = 0.0;
        for (double x = -2; x <= 2; x += 0.2)
        {
            for (double y = -2; y <= 2; y += 0.2)
            {
                output = 0.0;
                expected = (KastrignsProvider.CalculateResult(x, y) + 50) / 100;
                double[] input = new double[] { x, y };
                output = GetOutput(input);
                error = expected - output;
                // Console.WriteLine("True Expected: " + (expected*80-40) + " Expected: " + expected + " Got: " + output + " Error: " + error);
                totalError += Math.Pow(error, 2) / 2;
                BackPropagation(error, input);
            }
        }
        counter++;
        pivE = totalError;
        Console.WriteLine(totalError + "\t" + counter);
        // Console.WriteLine("-----");
    } while (totalError > 0.001 && counter < Max);
    Console.WriteLine(counter);
    Console.ReadLine();
}

public void Test()
{
}

public void BackPropagation(double error, double[] input)
{
    double[] errorsFromNextLayer = new double[] { error };
    errorsFromNextLayer = network[network.Length - 1].CalculateLastError(errorsFromNextLayer);
    for (int i = network.Length - 2; i >= 0; i--)
    {
        errorsFromNextLayer = network[i].CalculateError(errorsFromNextLayer, network[i + 1]);
    }
    CalculateNewWeights(input);
}

private void CalculateNewWeights(double[] input)
{
    double[] result = new double[input.Length];
    Array.Copy(input, result, input.Length);
    for (int i = 0; i < network.Length; i++)
    {
        network[i].CalculateNewWeights(result);
        result = network[i].lastOutput;
    }
}

```

```

namespace Scen2ver2
{
    class Trainer
    {
        private Layer[] network;
        private const int Max = 100000;

        public Trainer(int[,] neuralStructure, double learningRate)
        {
            network = new Layer[neuralStructure.GetLength(0)];
            for (int i = 0; i < neuralStructure.GetLength(0); i++)
            {
                network[i] = new Layer(neuralStructure[i, 0], neuralStructure[i, 1], learningRate);
            }
        }

        public void PrintNeuralNetwork()
        {
            int i = 1;
            foreach (Layer l in network)
            {
                Console.WriteLine("----- Warstwa " + i + " -----");
                l.PrintLayer();
                i++;
            }
        }

        public double GetOutput(double[] input)
        {
            double[] result = new double[input.Length];
            Array.Copy(input, result, input.Length);
            double output = 0.0;
            for (int i = 0; i < network.Length; i++)
            {
                result = network[i].CalculateOutput(result);
            }
            for (int i = 0; i < result.Length; i++)
            {
                output += result[i];
            }

            return output;
        }

        public void Learn()
        {

```

```

namespace Scen2ver2
{
    class RastrignsProvider
    {
        public static double CalculateResult(double x, double y)
        {
            return 20 + Math.Pow(x, 2) + Math.Pow(y, 2) - 10 * (Math.Cos(2 * Math.PI * x) + Math.Cos(2 * Math.PI * y));
        }

        private RastrignsProvider()
        {
        }
    }
}

```