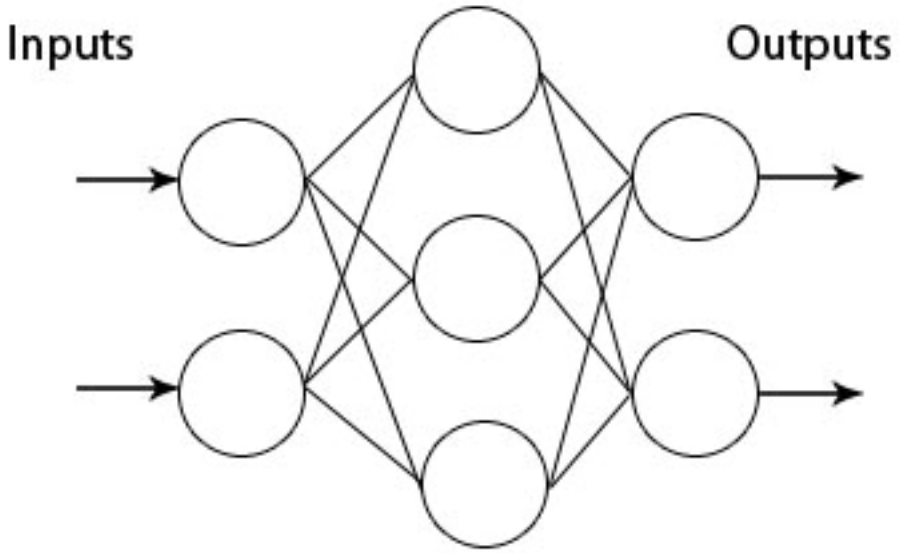
Scenariusz 3

Maciej Słaboń
Gr 4

Cel ćwiczenia;
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.



Unipolarną funkcję sigmoidalną wykorzystałem jako funkcję aktywacji:

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

Został użyty algorytm Backpropagation - wstecznej propagacji błędu.
Błąd potrzebny do korekcji wag dla każdego neuronu w ostatniej warstwie obliczałem za pomocą wzoru:

$$\frac{\partial E}{\partial W_{jk}} = \mathcal{O}_j \delta_k$$

$$\delta_k = \mathcal{O}_k(1 - \mathcal{O}_k)(\mathcal{O}_k - t_k)$$

Natomiast każdą wcześniejszą

$$\frac{\partial E}{\partial W_{ij}} = \mathcal{O}_i \delta_j$$

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

Korekcja wag ze wzoru:

$$\Delta W = -\eta \delta_\ell \mathcal{O}_{\ell-1}$$
$$\Delta \theta = \eta \delta_\ell$$

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

Łączny błąd liczyłem z MSE, czyli

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

Dane:

https://en.wikipedia.org/wiki/Rastrigin_function jako przedział przyjąłem dane dla x i y należących do przedziału [-2,2] .

Ilość danych do uczenia się była zmienna ponieważ zauważyłem, że im większa sieć tym więcej potrzebuje danych. 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.

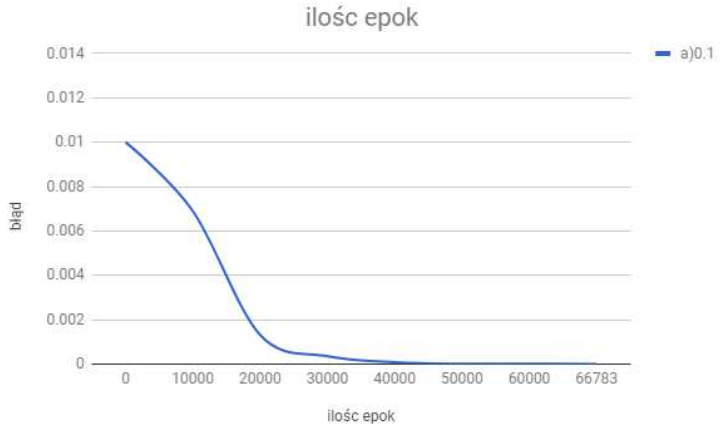
Sieci jakich użyłem to (ze względu na ilość neuronów znajdujących się w warstwie)
A)9-8-7-6
B)12-9-7-3
C)20-15-10

Do testów użyłem współczynników uczenia 0.1 i 0.01.

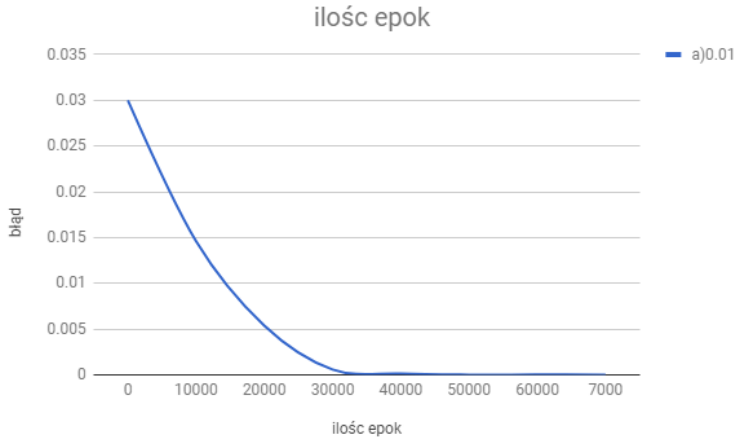
Wyniki:

A)

1) Średni błąd dla wartości przewidywanej : 0.0189%

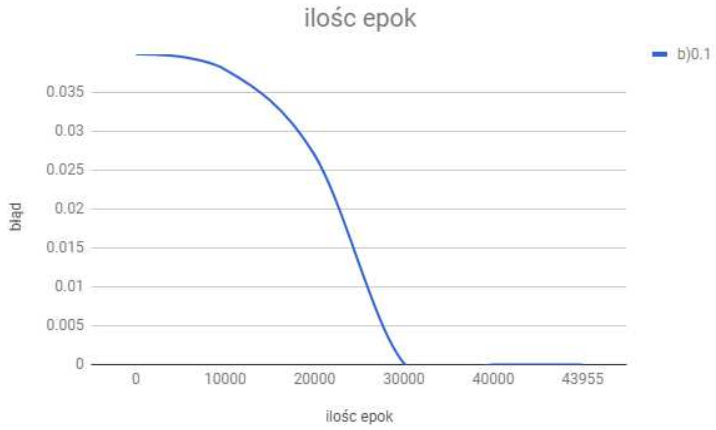


2) Średni błąd dla wartości przewidywanej : 0.0227%



B)

1) Średni błąd dla wartości przewidywanej. 0.0125%

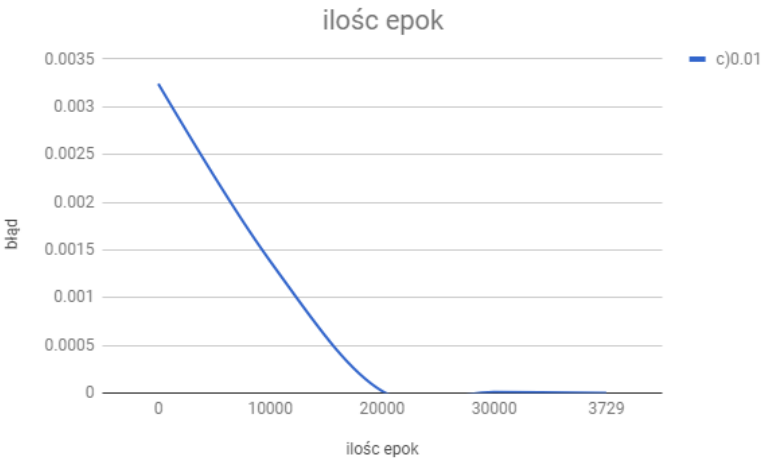


2) Średni błąd dla wartości przewidywanej: 0.0659%

c)

1) Średni błąd dla wartości przewidywanej: 0.0118%

2) Średni błąd dla wartości przewidywanej: 0.0237%



Analiza wyników:

Widać wpływ współczynnika uczenia na długość trwania procesu. Jak widać z wykresów wygląd funkcji zależności błędu od epoki zależy od wyglądu sieci (ilości warstw oraz ilości

neuronów w tych warstwach). Najlepiej wypadła sieć C to dzięki wiekszej ilości neuronów w każdej warstwie. Przy niskim współczynniku średni błąd dla b2 jest widocznie większy od pozostałych. Można zauważyć zależność, że wraz z niższym współczynnikiem wykres staje się gładszy.

Wnioski:

- Współczynnik uczenia i ilość neuronów w sieci ma wpływ na szybkość uczenia
- Wysoki błąd może wynikać z niewystarczającej ilości danych uczących lub źle obranej skali
- Wielowarstwowa sieć neuronowa umożliwia aproksymacje skomplikowanych funkcji

Listening Kodu:

```
public class Main {
    public static void main(String[] args) {
        int[] structure = new int[] { 3, 25, 15, 10, 5, 3, 1 }; //budowa sieci (ilość wejść, ilość neuronów warstwie,(...),ilość wyjść)
        Trainer t = new Trainer(structure,  learningRate: 0.1f,  max: 100000); //trener
        t.learn();  //uczenie
        t.test();   //testowanie
    }
}
```

```
public class DataProvider {
    public static double calculateResult(double x, double y) { //wartość funkcji dla podanego x,y
        return 20 + Math.pow(x, 2) + Math.pow(y, 2) - 10 * (Math.cos(2 * Math.PI * x) + Math.cos(2 * Math.PI * y));
    }
}
```

```java
import java.util.Random;

public class Layer {
    int nIn;
    int nOut;

    public float[] outputs;
    public float[] inputs;
    public float[][] weights;
    public float[][] weightsDelta;
    public float[] gamma;
    public float[] error;

    public static Random random = new Random();

    public Layer(int numberOfInputs, int numberOfOuputs) {
        this.nIn = numberOfInputs;
        this.nOut = numberOfOuputs;

        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];

        initWeights();
    }

    public void initWeights() {     //inicjalizacja wag
        for (int i = 0; i < nOut; i++) {
            for (int j = 0; j < nIn; j++) {
                weights[i][j] = (float) random.nextDouble() - 0.5f; //losowa liczba od -0.5 do 0.5
            }
        }
    }

    public float[] forward(float[] inputs) {    //do przodu, obliczanie wyniku z warstwy na podstawie wejścia
        this.inputs = inputs;

        for (int i = 0; i < nOut; i++) {
            outputs[i] = 0;
            for (int j = 0; j < nIn; j++) {
                outputs[i] += inputs[j] * weights[i][j];
            }

            outputs[i] = (float) Math.tanh(outputs[i]);
        }

        return outputs;
    }
```

```
    public float div(float value) { return 1 - (value * value); } //pochodna funkcji aktywacji

    public void backPropOutput(float[] expected) {  //algorytm propagacji wstecznej dla ostatniej warstwy zgodnie ze sprawozdaniem
        for (int i = 0; i < nOut; i++)
            error[i] = outputs[i] - expected[i];

        for (int i = 0; i < nOut; i++)
            gamma[i] = error[i] * div(outputs[i]);

        for (int i = 0; i < nOut; i++) {
            for (int j = 0; j < nIn; j++) {
                weightsDelta[i][j] = gamma[i] * inputs[j];
            }
        }
    }

    public void backPropHidden(float[] gammaForward, float[][] weightsFoward) { //algorytm propagacji wstecznej dla każdej kolejnej warstwy poza pierwszą zgodnie ze sprawozdaniem
        for (int i = 0; i < nOut; i++) {
            gamma[i] = 0;

            for (int j = 0; j < gammaForward.length; j++) {
                gamma[i] += gammaForward[j] * weightsFoward[j][i];
            }

            gamma[i] *= div(outputs[i]);
        }

        for (int i = 0; i < nOut; i++) {
            for (int j = 0; j < nIn; j++) {
                weightsDelta[i][j] = gamma[i] * inputs[j];
            }
        }
    }

    public void updateWeights(float learningRate) { //aktualizacja wag (wywoływana po algorytmie propagacji wstecznej
        for (int i = 0; i < nOut; i++) {
            for (int j = 0; j < nIn; j++) {
                weights[i][j] -= weightsDelta[i][j] * learningRate;
            }
        }
    }
}
```

```
public class Network {
    Layer[] layers;
    float learningRate;

    public Network(int[] layer, float learningRate) {       //tworzenie sieci
        this.learningRate = learningRate;

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

        for (int i = 0; i < layers.length; i++) {
            layers[i] = new Layer(layer[i], layer[i + 1]);
        }
    }

    public float[] forward(float[] inputs) {    //algorytm feedforward
        layers[0].forward(inputs);  //zaczynamy od pierwszej warstwy (jako input przyjmuje input sieci)
        for (int i = 1; i < layers.length; i++) {
            layers[i].forward(layers[i - 1].outputs);   //każda kolejna warstwa jako dane wejściowe przyjmuje dane wyjściowe warstwy poprzedniej
        }

        return layers[layers.length - 1].outputs;   //zwracamy dane wyjściowe ostatniej warstwy jako wynik sieci
    }

    public void backPropagation(float[] expected) { //algorytm backPropagation
        for (int i = layers.length - 1; i >= 0; i--) { //zgodnie ze sprawozdaniem mamy dwie różne możliwości
            if (i == layers.length - 1) {   //jeżeli znajdujemy się w warstwie ostatniej
                layers[i].backPropOutput(expected);
            } else {    //jeżeli znajdujemy się w każdej innej warstwie
                layers[i].backPropHidden(layers[i + 1].gamma, layers[i + 1].weights);
            }
        }

        for (int i = 0; i < layers.length; i++) { //po przejściu wstecz aktualizujemy wszystkie warstwy o nowe wagi
            layers[i].updateWeights(learningRate);
        }
    }
}
```

```java
public class Trainer {
    private Network network;
    private int Max;

    public Trainer(int[] neuralStructure, float learningRate, int max) {
        network = new Network(neuralStructure, learningRate);
        Max = max;
    }

    public void learn() {   //algorytm uczenia
        int counter = 0;    //ilość epok
        double mseError = 0.0f;
        do {
            mseError = 0.0f;  //błąd średniokwadratowy
            for (float x = -2; x <= 2; x += 0.5f) { // x i y jako dane wejściowe dla obliczenia wartości funkcji oraz do nauki sieci
                for (float y = -2; y <= 2; y += 0.5f) {
                    float expected = (float) ((DataProvider.calculateResult(x, y) + 50) / 100); //obliczanie wartości funkcji w punkcie x,y oraz normalizacja do przedziału 0,1
                    float output = network.forward(new float[]{x, y, 1})[0]; //wynik sieci
                    mseError += Math.pow(output - expected, 2); //błąd średnikwadratowy
                    network.backPropagation(new float[]{expected}); //algorytm propagacji wstecznej z wyniku
                }
            }
            counter++;  //zwiększanie liczby epok
            if (counter % (Max / 10) == 0) { //co 10% epok z maksymalnej ilości epok
                System.out.println("Error: " + mseError);
                test();
            }
        } while (counter < Max && mseError > 0.000001);
        System.out.println("Error: " + mseError + " Counter: " + counter);
    }

    public void test() {    //testowanie podobne jak uczenie tylko nie zawiera algorytmów propagacji, zamiast tego tylko porównuje wyniki otrzymane z oczekiwanymi
        float error = 0.0f;
        int iterator = 0;
        for (float x = -2; x <= 2; x += 1f) {
            for (float y = -2; y <= 2; y += 1f) {
                float result = network.forward(new float[]{x, y, 1})[0];
                float expected = (float) ((DataProvider.calculateResult(x, y)));
                System.out.println(x + " " + y + " Got: " + (result * 100 - 50) + " Expected: " + expected);
                iterator++;
                error += Math.abs((result * 100 - 50) - expected);
            }
        }
        System.out.println("Prawdziwy blad sredni: " + (error / (float) iterator));
    }
}
```