Scenariusz 5

Białek Tomasz, gr. 1

Temat ćwiczenia: Budowa i działanie sieci Kohonena dla WTA.

1. Cel ćwiczenia

Celem ćwiczenia jest poznanie budowy i działania sieci Kohonena przy wykorzystaniu reguły WTA do odwzorowywania istotnych cech kwiatów.

2. Opis budowy sieci i algorytmów uczenia.

Celem zbudowanej sieci jest podział danych uczących na określone grupy i przyporządkowanie danej grupie danego elementu wyjściowego. Podział na grupy polega na tym, żeby elementy w danej grupie były jak najbardziej podobne do siebie jednocześnie będąc zupełnie inne od elementów z innych grup.

Długość działki kielicha	Szerokość działki kielicha	Długość płatka	Szerokość płatka	Rodzaj
5.2	3.5	1.4	0.2	*I. setosa*
4.9	3.0	1.4	0.2	*I. setosa*
4.7	3.2	1.3	0.2	*I. setosa*
4.6	3.1	1.5	0.2	*I. setosa*
5.0	3.6	1.4	0.3	*I. setosa*
5.4	3.9	1.7	0.4	*I. setosa*
4.6	3.4	1.4	0.3	*I. setosa*
5.0	3.4	1.5	0.2	*I. setosa*
4.4	2.9	1.4	0.2	*I. setosa*
4.9	3.1	1.5	0.1	*I. setosa*
5.4	3.7	1.5	0.2	*I. setosa*
4.8	3.4	1.6	0.2	*I. setosa*
4.8	3.0	1.4	0.1	*I. setosa*
4.3	3.0	1.1	0.1	*I. setosa*
5.8	4.0	1.2	0.2	*I. setosa*
5.7	4.4	1.5	0.4	*I. setosa*
5.4	3.9	1.3	0.4	*I. setosa*
5.1	3.5	1.4	0.3	*I. setosa*
5.7	3.8	1.7	0.3	*I. setosa*
5.1	3.8	1.5	0.3	*I. setosa*
5.4	3.4	1.7	0.2	*I. setosa*

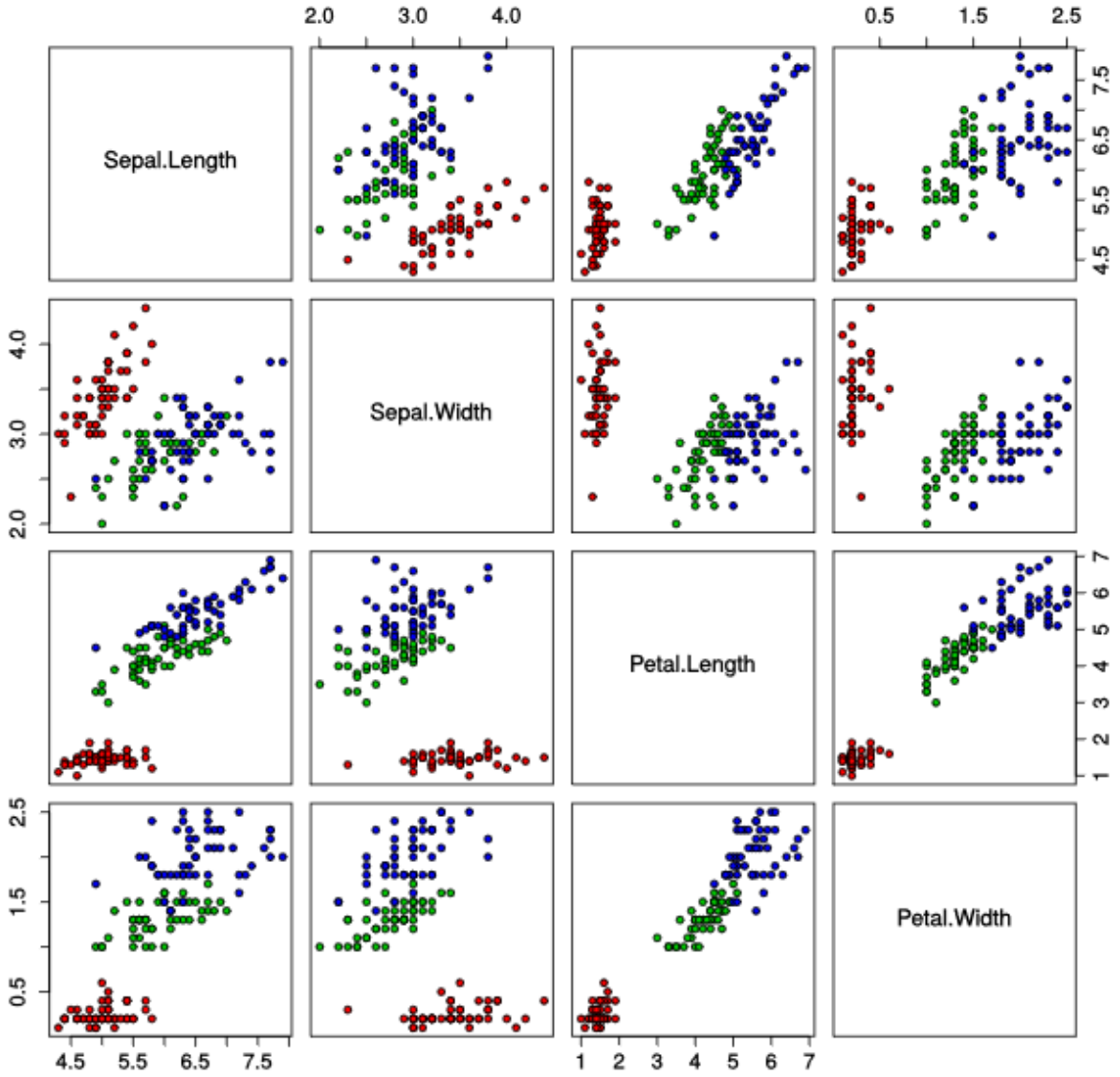
Długość działki kielicha	Szerokość działki kielicha	Długość płatka	Szerokość płatka	Rodzaj
5.1	3.7	1.5	0.4	*I. setosa*
4.6	3.6	1.0	0.2	*I. setosa*
5.1	3.3	1.7	0.5	*I. setosa*
4.8	3.4	1.9	0.2	*I. setosa*
5.0	3.0	1.6	0.2	*I. setosa*
5.0	3.4	1.6	0.4	*I. setosa*
5.2	3.5	1.5	0.2	*I. setosa*
5.2	3.4	1.4	0.2	*I. setosa*
4.7	3.2	1.6	0.2	*I. setosa*
4.8	3.1	1.6	0.2	*I. setosa*
5.4	3.4	1.5	0.4	*I. setosa*
5.2	4.1	1.5	0.1	*I. setosa*
5.5	4.2	1.4	0.2	*I. setosa*
4.9	3.1	1.5	0.2	*I. setosa*
5.0	3.2	1.2	0.2	*I. setosa*
5.5	3.5	1.3	0.2	*I. setosa*
4.9	3.6	1.4	0.1	*I. setosa*
4.4	3.0	1.3	0.2	*I. setosa*
5.1	3.4	1.5	0.2	*I. setosa*
5.0	3.5	1.3	0.3	*I. setosa*
4.5	2.3	1.3	0.3	*I. setosa*
4.4	3.2	1.3	0.2	*I. setosa*
5.0	3.5	1.6	0.6	*I. setosa*
5.1	3.8	1.9	0.4	*I. setosa*
4.8	3.0	1.4	0.3	*I. setosa*
5.1	3.8	1.6	0.2	*I. setosa*
4.6	3.2	1.4	0.2	*I. setosa*
5.3	3.7	1.5	0.2	*I. setosa*
5.0	3.3	1.4	0.2	*I. setosa*
7.0	3.2	4.7	1.4	*I. versicolor*
6.4	3.2	4.5	1.5	*I. versicolor*
6.9	3.1	4.9	1.5	*I. versicolor*
5.5	2.3	4.0	1.3	*I. versicolor*

Długość działki kielicha	Szerokość działki kielicha	Długość płatka	Szerokość płatka	Rodzaj
6.5	2.8	4.6	1.5	*I. versicolor*
5.7	2.8	4.5	1.3	*I. versicolor*
6.3	3.3	4.7	1.6	*I. versicolor*
4.9	2.4	3.3	1.0	*I. versicolor*
6.6	2.9	4.6	1.3	*I. versicolor*
5.2	2.7	3.9	1.4	*I. versicolor*
5.0	2.0	3.5	1.0	*I. versicolor*
5.9	3.0	4.2	1.5	*I. versicolor*
6.0	2.2	4.0	1.0	*I. versicolor*
6.1	2.9	4.7	1.4	*I. versicolor*
5.6	2.9	3.6	1.3	*I. versicolor*
6.7	3.1	4.4	1.4	*I. versicolor*
5.6	3.0	4.5	1.5	*I. versicolor*
5.8	2.7	4.1	1.0	*I. versicolor*
6.2	2.2	4.5	1.5	*I. versicolor*
5.6	2.5	3.9	1.1	*I. versicolor*
5.9	3.2	4.8	1.8	*I. versicolor*
6.1	2.8	4.0	1.3	*I. versicolor*
6.3	2.5	4.9	1.5	*I. versicolor*
6.1	2.8	4.7	1.2	*I. versicolor*
6.4	2.9	4.3	1.3	*I. versicolor*
6.6	3.0	4.4	1.4	*I. versicolor*
6.8	2.8	4.8	1.4	*I. versicolor*
6.7	3.0	5.0	1.7	*I. versicolor*
6.0	2.9	4.5	1.5	*I. versicolor*
5.7	2.6	3.5	1.0	*I. versicolor*
5.5	2.4	3.8	1.1	*I. versicolor*
5.5	2.4	3.7	1.0	*I. versicolor*
5.8	2.7	3.9	1.2	*I. versicolor*
6.0	2.7	5.1	1.6	*I. versicolor*
5.4	3.0	4.5	1.5	*I. versicolor*
6.0	3.4	4.5	1.6	*I. versicolor*
6.7	3.1	4.7	1.5	*I. versicolor*

Długość działki kielicha	Szerokość działki kielicha	Długość płatka	Szerokość płatka	Rodzaj
6.3	2.3	4.4	1.3	*I. versicolor*
5.6	3.0	4.1	1.3	*I. versicolor*
5.5	2.5	4.0	1.3	*I. versicolor*
5.5	2.6	4.4	1.2	*I. versicolor*
6.1	3.0	4.6	1.4	*I. versicolor*
5.8	2.6	4.0	1.2	*I. versicolor*
5.0	2.3	3.3	1.0	*I. versicolor*
5.6	2.7	4.2	1.3	*I. versicolor*
5.7	3.0	4.2	1.2	*I. versicolor*
5.7	2.9	4.2	1.3	*I. versicolor*
6.2	2.9	4.3	1.3	*I. versicolor*
5.1	2.5	3.0	1.1	*I. versicolor*
5.7	2.8	4.1	1.3	*I. versicolor*
6.3	3.3	6.0	2.5	*I. virginica*
5.8	2.7	5.1	1.9	*I. virginica*
7.1	3.0	5.9	2.1	*I. virginica*
6.3	2.9	5.6	1.8	*I. virginica*
6.5	3.0	5.8	2.2	*I. virginica*
7.6	3.0	6.6	2.1	*I. virginica*
4.9	2.5	4.5	1.7	*I. virginica*
7.3	2.9	6.3	1.8	*I. virginica*
6.7	2.5	5.8	1.8	*I. virginica*
7.2	3.6	6.1	2.5	*I. virginica*
6.5	3.2	5.1	2.0	*I. virginica*
6.4	2.7	5.3	1.9	*I. virginica*
6.8	3.0	5.5	2.1	*I. virginica*
5.7	2.5	5.0	2.0	*I. virginica*
5.8	2.8	5.1	2.4	*I. virginica*
6.4	3.2	5.3	2.3	*I. virginica*
6.5	3.0	5.5	1.8	*I. virginica*
7.7	3.8	6.7	2.2	*I. virginica*
7.7	2.6	6.9	2.3	*I. virginica*
6.0	2.2	5.0	1.5	*I. virginica*

Długość działki kielicha	Szerokość działki kielicha	Długość płatka	Szerokość płatka	Rodzaj
6.9	3.2	5.7	2.3	*I. virginica*
5.6	2.8	4.9	2.0	*I. virginica*
7.7	2.8	6.7	2.0	*I. virginica*
6.3	2.7	4.9	1.8	*I. virginica*
6.7	3.3	5.7	2.1	*I. virginica*
7.2	3.2	6.0	1.8	*I. virginica*
6.2	2.8	4.8	1.8	*I. virginica*
6.1	3.0	4.9	1.8	*I. virginica*
6.4	2.8	5.6	2.1	*I. virginica*
7.2	3.0	5.8	1.6	*I. virginica*
7.4	2.8	6.1	1.9	*I. virginica*
7.9	3.8	6.4	2.0	*I. virginica*
6.4	2.8	5.6	2.2	*I. virginica*
6.3	2.8	5.1	1.5	*I. virginica*
6.1	2.6	5.6	1.4	*I. virginica*
6.3	3.4	5.6	2.4	*I. virginica*
6.4	3.1	5.5	1.8	*I. virginica*
6.0	3.0	4.8	1.8	*I. virginica*
6.9	3.1	5.4	2.1	*I. virginica*
6.7	3.1	5.6	2.4	*I. virginica*
6.9	3.1	5.1	2.3	*I. virginica*
5.8	2.7	5.1	1.9	*I. virginica*
6.8	3.2	5.9	2.3	*I. virginica*
6.7	3.3	5.7	2.5	*I. virginica*
6.7	3.0	5.2	2.3	*I. virginica*
6.3	2.5	5.0	1.9	*I. virginica*
6.5	3.0	5.2	2.0	*I. virginica*
6.2	3.4	5.4	2.3	*I. virginica*
5.9	3.0	5.1	1.8	*I. virginica*

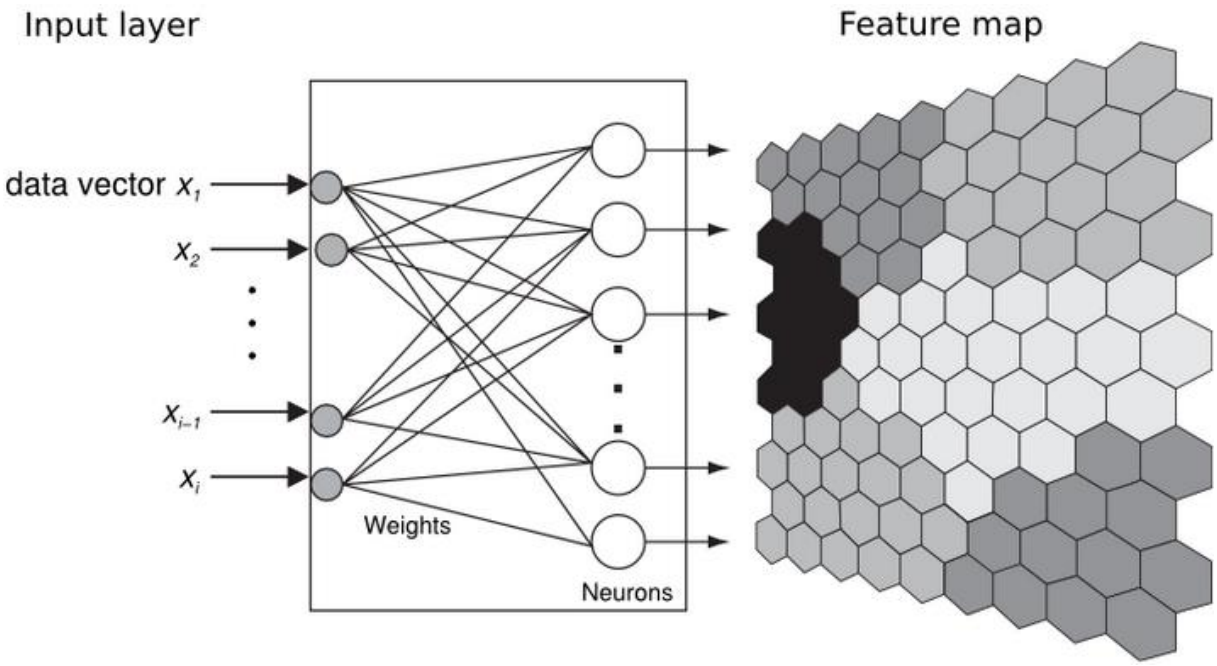
Tabela 1. Zestaw danych



Rys. 1 Cechy kwiatów (czerwony – setosa, green – versicolor, blue – virginica)

Syntetyczny opis sieci

Nauka sieci odbywa się za pomocą uczenia rywalizującego (metoda uczenia sieci samoorganizujących). Podczas procesu uczenia neurony są nauczane rozpoznawania danych i zbliżają się do obszarów zajmowanych przez te dane. Po wejściu każdego wektora uczącego wybierany jest tylko jeden neuron (neuron będący najbliższemu prezentowanemu wzorcowi). Wszystkie neurony rywalizują między sobą, gdzie zwycięża ten neuron, którego wartość jest największa. Zwycięski neuron przyjmuje na wyjściu wartość 1, pozostałe 0.



Rys. 1 Sieć samoorganizująca

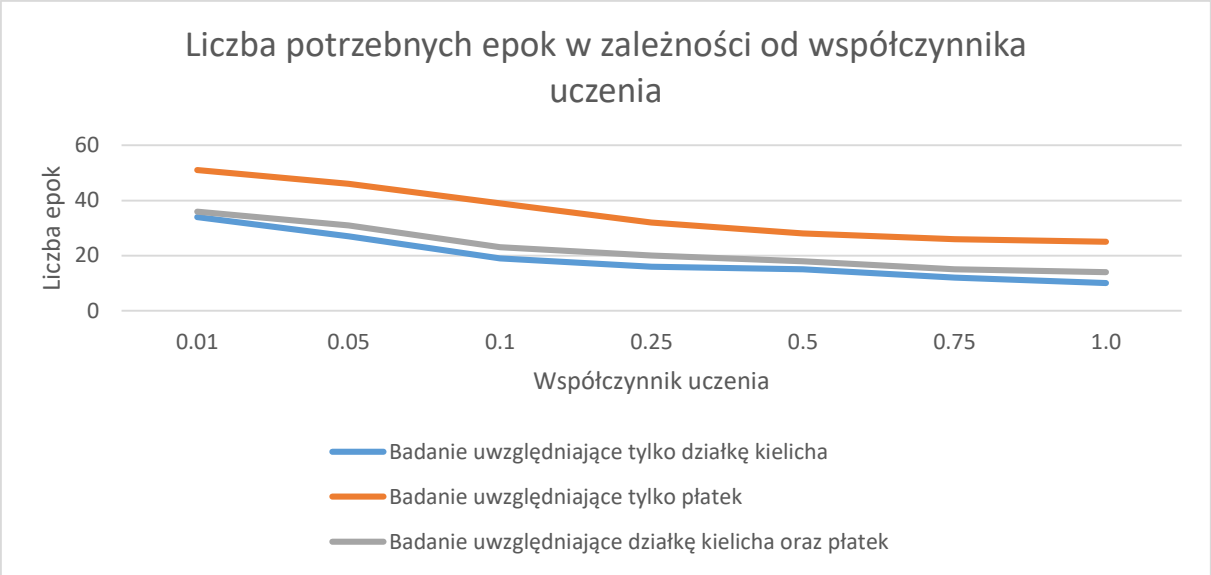
Proces uczenia przebiega według następującego schematu:

1. Normalizacja wszystkich danych
2. Wybór współczynnika uczenia η z przedziału (0; 1>
3. Wybór początkowych wartości wag z przedziału <0; 1)
4. Dla danego zbioru uczącego obliczamy odpowiedź sieci – dla każdego pojedynczego neuronu obliczana jest suma ilorazów sygnałów wejściowych oraz wag
5. Wybierany jest neuron, dla którego obliczona suma jest największa. Tylko dla tego neuronu następuje aktualizacja wag. Wzór na zaktualizowanie wagi jest następujący:

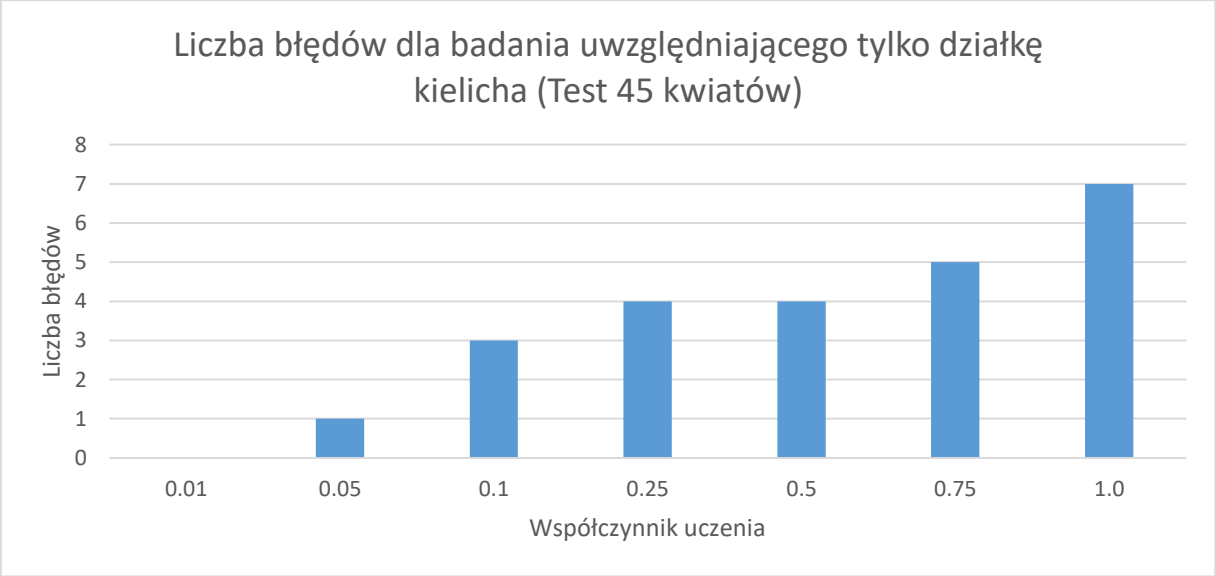
$$w_{i,j}(t+1) = w_{i,j}(t) * \eta * \Big(x_i * w_{i,j}(t)\Big)$$

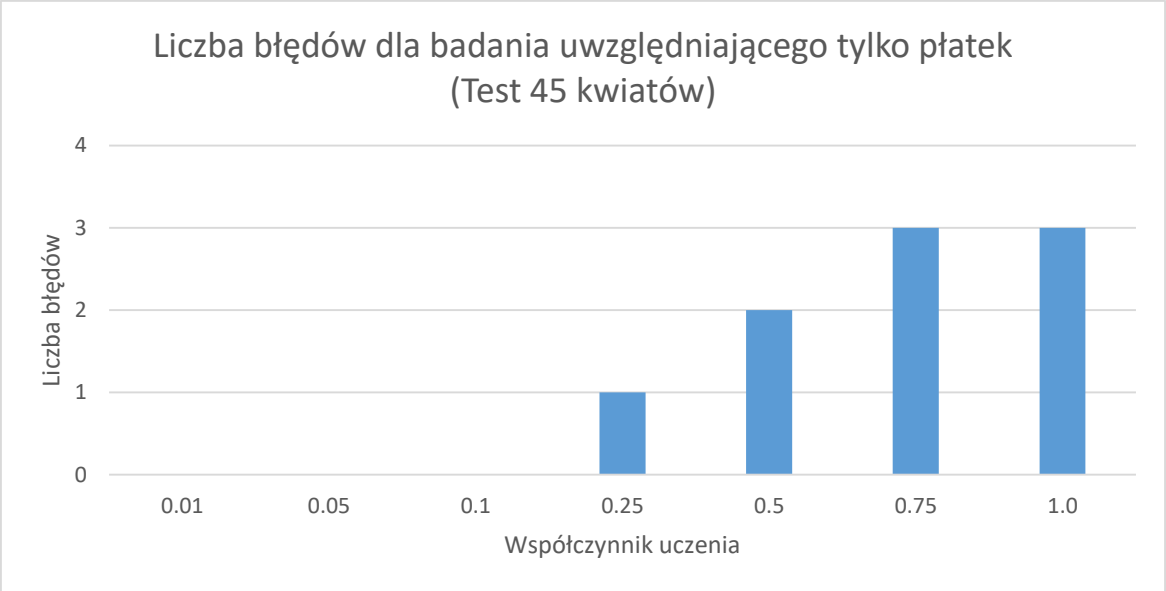
6. Znormalizowanie wartości nowego wektora wag
7. Zwycięski neuron daje odpowiedź na swoim wyjściu równą 1, a pozostałe 0.
8. Wczytanie kolejnego wektora uczącego.

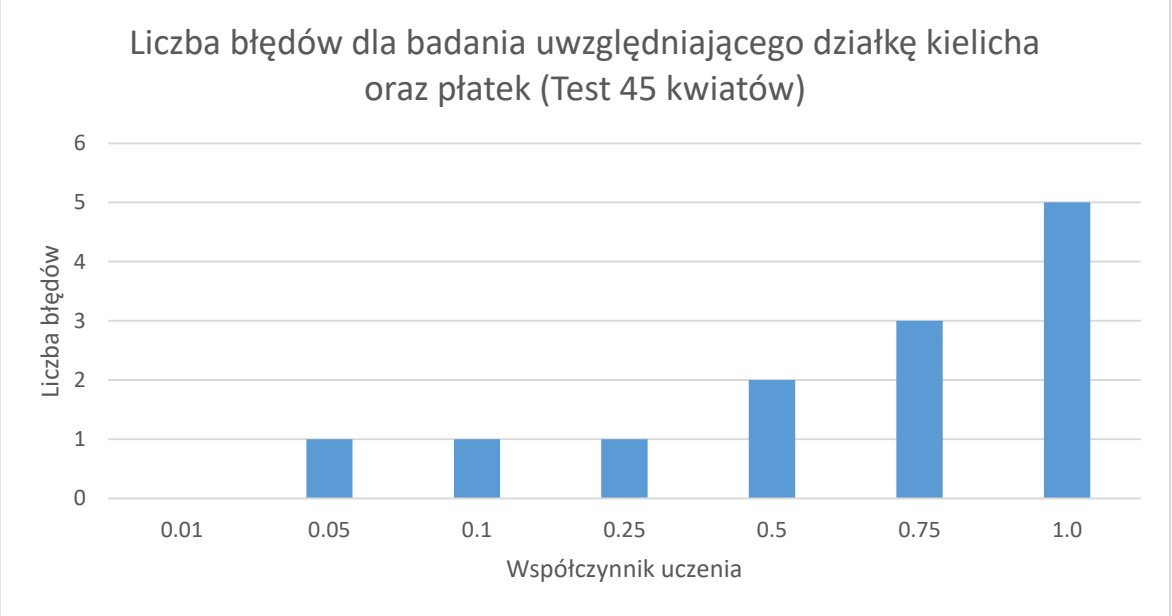
3. Zestawienie otrzymanych wyników



Z powyższego wykresu można odczytać, że sieć potrzebowała największej ilości epok do nauki dla badania uwzględniającego tylko płatek, najmniej dla badania uwzględniającego tylko działkę kielicha. Wynik pośredni przypadł badaniu uwzględniającego działkę kielicha oraz płatek. Powodem takich wyników może być sytuacja, gdy sieć mogła mieć już pogrupowane neurony ze względu na działkę kielicha, ale dane odpowiedzialne za płatek są dla tych trzech rodzajów bardzo podobne, co skutkuje w powolniejszym procesie uczenia.







Z powyższych wykresów wynika, że bez względu na dane badanie, największą efektywność uczenia sieć otrzymuje dla małych współczynników uczenia. Im większy współczynnik uczenia, tym powstaje więcej błędnych odpowiedzi. Spowodowane to może być, że dając duży współczynnik uczenia sieć wzmacnia za bardzo zwycięski neuron, przez co przy teście za dwa odmienne rodzaje kwiatów może być odpowiedzialny jeden neuron. Mniejsze współczynniki są efektywniejsze, lecz sieć wymaga dłuższej nauki, a z większymi współczynnikami uczenia sytuacja jest odwrotna.

4. Podsumowanie

Sieć Kohonena cechuje umiejętność podziału danych, które posiadają różne wartości dla poszczególnych cech, ponieważ jest to sieć samoorganizująca. Powoduje to, że odpowiedni podział na grupy może być wykonywane bez podawania wartości oczekiwanych (uczenie bez nauczyciela). Cały proces uczenia (jego efektywność) zależy od współczynnika uczenia. Im większy jest ten współczynnik, tym sieć uczy się szybciej. Jednakże wzrost efektywności nie jest wprost

proporcjonalny do współczynnika uczenia. Dla wzrostu przy małych współczynnikach uczenia następuje większa różnica w ilości potrzebnych epok aniżeli dla wzrostu przy dużych współczynnikach uczenia – występuje stabilizacja uczenia. Ponadto, sieci samoorganizujące potrafią się uczyć o wiele sprawniej od sieci podstawowych, co spowodowane jest typem danych uczących. W naszym przypadku było to 150 wektorów uczących, jednakże są one podzielone na trzy grupy, każda po 50 wektorów, co powoduje, że sieć „dostaje" w każdej epoce kilkadziesiąt bardzo podobnych danych. Podczas przygotowania danych trzeba również zwrócić uwagę na ich przedstawienie liczbowe, czyli na wartości w zestawie. Do poprawnego działania sieci potrzebna jest normalizacja, ponieważ w innym przypadku sieć nie będzie się uczyć – dane będą źle grupowane. Sieć dla odpowiednich współczynników uczenia pozwala uzyskać 100% poprawnych odpowiedzi, co powoduje, że sieć Kohonena sprawdza się przy dużej ilości różniących się pomiędzy sobą danych.

5. Kod programu

„Source.cpp"

```
#include <iostream>
#include <ctime>
#include <fstream>
#include <vector>

#include "Layer.h"

using namespace std;

//wczytanie do tablic danych wejsciowych
void setInputData(Neuron& neuron, vector<vector<double>> inputData, int
numberOfInputs, int inputDataRow);
//uczenie sieci
void learn(Layer& layer, vector<vector<double>> inputData);
//testowanie sieci
void test(Layer& layer, vector<vector<double>> inputData);
//wczytanie danych uczacych
void loadTrainingData(vector<vector<double>>&learnData, int numberOfInputs);
//wczytaniee danych testowych
void loadTestingData(vector<vector<double>>&testData, int numberOfInpus);

//strumienie do plikow sluzace do wczytania danych uczacych oraz zapisu wynikow
fstream OUTPUT_FILE_LEARNING, OUTPUT_FILE_TESTING_SUM, OUTPUT_FILE_TESTING_WINNER;
fstream TRAINING_DATA, TESTING_DATA;

int main() {
	srand(time(NULL));

	//wektory z danymi uczacymi oraz testujacymi
	vector<vector<double>> trainData;
	vector<vector<double>> testData;
	int numberOfNeurons = 10;
	int numberOfInputs = 4;
	double learningRate = 0.5;
	//stworzenie sieci Kohonena
	Layer kohonenNetwork(numberOfNeurons, numberOfInputs, learningRate);
	//wczytanie danych uczacych
	loadTrainingData(trainData, numberOfInputs);
	//wczytaniee danych testowych
	loadTestingData(testData, numberOfInputs);

	//"menu" programu
	do {
		cout << "1. Learn" << endl;
```

```
		cout << "2. Test" << endl;
		cout << "3. Exit" << endl;

		int choice;
		cin >> choice;
		switch (choice) {
		case 1:
			OUTPUT_FILE_LEARNING.open("output_learning_data.txt", ios::out);

			for (int epoch = 1, i = 0; i < 2; i++, epoch++) {
				//uczenie
				OUTPUT_FILE_LEARNING << "EPOCH: " << epoch << endl;
				cout << "EPOCH: " << epoch << endl;
				learn(kohonenNetwork, trainData);
			}
			OUTPUT_FILE_LEARNING.close();
			break;
		case 2:
			OUTPUT_FILE_TESTING_SUM.open("output_testing_data.txt", ios::out);
			OUTPUT_FILE_TESTING_WINNER.open("output_testing_neuron.txt",
ios::out);
			//testowanie
			test(kohonenNetwork, testData);
			break;
		case 3:
			OUTPUT_FILE_LEARNING.close();
			OUTPUT_FILE_TESTING_SUM.close();
			OUTPUT_FILE_TESTING_WINNER.close();
			return 0;
		default:
			cout << "BAD BAD BAD" << endl;
		}
	} while (true);

	return 0;
}

//wczytanie do tablic danych wejsciowych
void setInputData(Neuron& neuron, vector<vector<double>> inputData, int
numberOfInputs, int row)
{
	for (int i = 0; i < numberOfInputs; i++) {
		neuron.inputs[i] = inputData[row][i];
	}
}

//uczenie
void learn(Layer& layer, vector<vector<double>> inputData)
{
	int counter = 0;
	for (int rowOfData = 0; rowOfData < inputData.size(); rowOfData++) {
		for (int i = 0; i < layer.numberOfNeurons; i++) {
			//wczytanie danych do tablic
			setInputData(layer.neurons[i], inputData,
layer.neurons[i].getNumberOfInputs(), rowOfData);
			//wyliczenie sumy wejscia
			layer.neurons[i].calculateSumOfAllInputs();
		}
		//zmiana wag
		layer.changeWeights(true);
		//przeskoczenie na kolejny rodzaj kwiatka (wyzerowanie licznika)
		if (counter == 50) {
```

```
                        counter = 0;
                        OUTPUT_FILE_LEARNING << "Next flower" << endl;
                        cout << "Next flower" << endl;
                }
                OUTPUT_FILE_LEARNING << layer.winnerIndex << endl;
                cout << "Winner: " << layer.winnerIndex << endl;
                counter++;
        }
}

//testowanie
void test(Layer& layer, vector<vector<double>> inputData) {
        int counter = 0;
        for (int wierszDanych = 0; wierszDanych < inputData.size(); wierszDanych++) {
                for (int i = 0; i < layer.numberOfNeurons; i++) {
                        //wczytanie danych do tablic
                        setInputData(layer.neurons[i], inputData,
layer.neurons[i].getNumberOfInputs(), wierszDanych);
                        //wyliczenia sumy wejscia
                        layer.neurons[i].calculateSumOfAllInputs();
                }
                //przeskoczenie na kolejny rodzaj kwiatka (wyzerowanie licznika)
                if (counter == 15) {
                        counter = 0;
                        OUTPUT_FILE_TESTING_WINNER << "Next flower" << endl;
                        cout << "Next flower" << endl;
                }
                //wagi nie beda zaktualiowane dla zwyciezcy
                layer.changeWeights(false);
                OUTPUT_FILE_TESTING_SUM <<
layer.neurons[layer.winnerIndex].sumOfAllInputs << endl;
                OUTPUT_FILE_TESTING_WINNER << layer.winnerIndex << endl;
                cout << "Which neuron: " << layer.winnerIndex << endl;
                counter++;
        }
}

//wczytanie danych uczacych z pliku
void loadTrainingData(vector<vector<double>> &trainData, int numberOfInputs) {
        TRAINING_DATA.open("data.txt", ios::in);
        vector<double> row;

        do {
                row.clear();
                for (int i = 0; i < numberOfInputs; i++) {
                        double inputTmp = 0.0;
                        TRAINING_DATA >> inputTmp;
                        row.push_back(inputTmp);
                        if (i == numberOfInputs - 1) {
                                TRAINING_DATA >> inputTmp;
                                //row.push_back(inputTmp);
                        }
                }

                //znormalizowanie danych uczacych
                double length = 0.0;

                for (int i = 0; i < numberOfInputs; i++)
                        length += pow(row[i], 2);

                length = sqrt(length);
```

```
		for (int i = 0; i < numberOfInputs; i++)
			row[i] /= length;

		trainData.push_back(row);

	} while (!TRAINING_DATA.eof());

	TRAINING_DATA.close();
}

//wczytanie danych testujacych z pliku
void loadTestingData(vector<vector<double>> &testData, int numberOfInputs) {
	TESTING_DATA.open("datatest.txt", ios::in);
	vector<double> row;

	while (!TESTING_DATA.eof()) {
		row.clear();

		for (int i = 0; i < numberOfInputs; i++) {
			double inputTmp = 0.0;
			TESTING_DATA >> inputTmp;
			row.push_back(inputTmp);
			if (i == numberOfInputs - 1) {
				TRAINING_DATA >> inputTmp;
				//row.push_back(inputTmp);
			}
		}

		//znormalizowanie danych uczacych
		double length = 0.0;

		for (int i = 0; i < numberOfInputs; i++)
			length += pow(row[i], 2);

		length = sqrt(length);

		for (int i = 0; i < numberOfInputs; i++)
			row[i] /= length;

		testData.push_back(row);
	}

	TESTING_DATA.close();
}
```

„Layer.h”

```
#pragma once
#include <vector>
#include "Neuron.h"
using namespace std;

class Layer {
public:

	vector<Neuron> neurons; //wektor neuronow
	vector<double> sums; //wektor sum wejsc
	int numberOfNeurons; //liczba neuronow
	int winnerIndex; //indeks zwyciezcy

	//konstruktor
	Layer(int numberOfNeurons, int numberOfInputs, double learningRate);
```

```
	//zmiana wag (learning = true dla procesu uczenia, = false dla procesu
testowania)
	void changeWeights(bool learning);
	void findTheLargestSum(bool learning); //szukanie zwycieskiego neuronu
	void sumOfTheLayer(); //obliczenie sumy wszystkich wejsc
};
```

„Layer.cpp”

```
#include "Layer.h"

//konstruktor
Layer::Layer(int numberOfNeurons, int numberOfInputs, double learningRate) {
	this->numberOfNeurons = numberOfNeurons;
	this->neurons.resize(numberOfNeurons);
	for (int i = 0; i < numberOfNeurons; i++)
		this->neurons[i].Neuron::Neuron(numberOfInputs, learningRate);
}

//obliczenie sum wszystkich wejsc, poszukiwanie tego o najwiekszej sumie i
aktualizacja jego wag
void Layer::changeWeights(bool learning) {
	sumOfTheLayer();
	findTheLargestSum(learning);
}

//obliczenie sumy wszystkich wejsc
void Layer::sumOfTheLayer() {
	this->sums.clear();
	for (int i = 0; i < this->numberOfNeurons; i++)
		this->sums.push_back(neurons[i].calculateSumOfAllInputs());
}

//poszukiwanie wejscia o najwiekszej sumie
void Layer::findTheLargestSum(bool learning) {
	double tmp = this->sums[0];
	this->winnerIndex = 0;
	for (int i = 1; i < this->sums.size(); i++) {
		if (tmp < this->sums[i]) {
			this->winnerIndex = i;
			tmp = this->sums[i];
		}
	}
	this->neurons[this->winnerIndex].activationFunction();
	if (learning == true) //aktualizacja wag
		this->neurons[this->winnerIndex].calculateNewWeight();
}
```

„Neuron.h”

```
#pragma once
#include <iostream>
#include <vector>

using namespace std;

class Neuron {
public:

	vector<double> inputs; //wejscia
	vector<double> weights; //wagi
	double sumOfAllInputs; //suma wszystkich wejsc
	double outputValue; //wartosc wyjsciowa
	double learningRate; //wspolczynnik uczenia

	Neuron(); //konstruktor
```

```
	Neuron(int numberOfInputs, double learningRate); //konstruktor

	double firstWeight(); //wylosowanie poczatkowych wag z zakresu <0;1)
	void normalizeWeight(); //znormalizowanie wag (podczas procesu uczenia)
	//stworzenie poczatkowych wejsc (ustawienie wejsc na 0, wykorzystanie metody
firstWeight())
	void createInputs(int numberOfInputs);
	void activationFunction(); //funkcja sigmoidalna obliczajaca wyjscie
	void calculateNewWeight(); //obliczenie nowej wagi dla zwycieskiego neuronu
	double calculateSumOfAllInputs(); //obliczenie sumy wszystkich wejsc

	int getNumberOfInputs() { //zwraca rozmiar wejsc
		return inputs.size();
	}

	int getNumberOfWeights() { //zwraca rozmiar wag
		return weights.size();
	}
};
```

„Neuron.cpp”

```
#include "Neuron.h"
#include <ctime>
#include <cmath>

//konstruktor
Neuron::Neuron() {
	this->inputs.resize(0);
	this->weights.resize(0);
	this->sumOfAllInputs = 0.0;
	this->outputValue = 0.0;
	this->learningRate = 0.0;
}

//konstruktor
Neuron::Neuron(int numberOfInputs, double learningRate) {
	createInputs(numberOfInputs);
	normalizeWeight();
	this->learningRate = learningRate;
	this->sumOfAllInputs = 0.0;
	this->outputValue = 0.0;
}

//stworzenie poczatkowych wejsc (ustawienie wejsc na 0, wykorzystanie metody
firstWeight())
void Neuron::createInputs(int numberOfInputs) {
	for (int j = 0; j < numberOfInputs; j++) {
		this->inputs.push_back(0);
		this->weights.push_back(firstWeight());
	}
}

//obliczenie sumy wszystkich wejsc
double Neuron::calculateSumOfAllInputs() {
	this->sumOfAllInputs = 0.0;
	for (int i = 0; i < getNumberOfInputs(); i++)
		this->sumOfAllInputs += inputs[i] * weights[i];
	return sumOfAllInputs;
}

//funkcja sigmoidalna obliczajaca wyjscie
void Neuron::activationFunction() {
	double beta = 1.0;
```

```
	this->outputValue = (1.0 / (1.0 + (exp(-beta * this->sumOfAllInputs))));
}

//obliczenie nowych wag
void Neuron::calculateNewWeight() {
	for (int i = 0; i < getNumberOfWeights(); i++)
		this->weights[i] += this->learningRate*(this->inputs[i] - this-
>weights[i]);
	normalizeWeight();
}

//ustalenie poczatkowych wag dla wszystkich wejsc - zakres <0;1)
double Neuron::firstWeight() {
	double max = 1.0;
	double min = 0.0;
	double weight = ((double(rand()) / double(RAND_MAX))*(max - min)) + min;
	return weight;
}

//znormalizowanie nowo obliczonej wagi zwycieskiego neuronu
void Neuron::normalizeWeight() {
	double vectorLength = 0.0;

	for (int i = 0; i < getNumberOfWeights(); i++)
		vectorLength += pow(this->weights[i], 2);

	vectorLength = sqrt(vectorLength);
	for (int i = 0; i < getNumberOfWeights(); i++)
		this->weights[i] /= vectorLength;
}
```