

Projekt z przedmiotu

Sztuczna Inteligencja

**Zrealizować sieć neuronową LVQ uczącą się rozpoznawania kwiatów irysa**

**Imię i nazwisko:** Rafał Nazarko

**Grupa projektowa:** P5

**Numer indeksu:** 163982

Rzeszów 2021

**Spis treści**

[1. Opis projektu 2](#_Toc73010133)

[2. Zbiór danych 2](#_Toc73010134)

[2.1 Przedstawienie zbioru 2](#_Toc73010135)

[2.2 Przygotowanie i normalizacja danych 2](#_Toc73010136)

[3. Wprowadzenie teoretyczne 4](#_Toc73010137)

[3.1 Sieć neuronowa 4](#_Toc73010138)

[3.1.1 Model sztucznego neuronu 5](#_Toc73010139)

[3.1.2 Sieć jednowarstwowa 6](#_Toc73010140)

[3.1.3 Sieć wielowarstwowa 7](#_Toc73010141)

[3.1.4 Sieć jednokierunkowa 8](#_Toc73010142)

[3.1.5 Sieć rekurencyjna 9](#_Toc73010143)

[3.1.6 Uczenie sieci neuronowej 9](#_Toc73010144)

[3.2 Algorytm LVQ 11](#_Toc73010145)

[3.2.1 LVQ1 12](#_Toc73010146)

[3.2.2 LVQ2.1 13](#_Toc73010147)

[3.2.3 LVQ3 14](#_Toc73010148)

[3.2.4 Warstwa Kohonena 15](#_Toc73010149)

[3.3 Działanie sieci nauczonej 15](#_Toc73010150)

[3.4 Walidacja krzyżowa 16](#_Toc73010151)

[4. Kod programu 17](#_Toc73010152)

[5. Eksperymenty 23](#_Toc73010153)

[5.1 Eksperyment pierwszy 23](#_Toc73010154)

[5.2 Eksperyment drugi 24](#_Toc73010155)

[5.3 Eksperyment trzeci 25](#_Toc73010156)

[6. Wnioski 26](#_Toc73010157)

[7. Bibliografia 26](#_Toc73010158)

[7.1 Literatura 26](#_Toc73010159)

[7.2 Źródła internetowe 26](#_Toc73010160)

# Opis projektu

Celem projektu jest stworzenie sieci neuronowej LVQ uczącej się rozpoznawania kwiatów irysa. Zakres realizacji obejmował przygotowanie danych, stworzenie algorytmu w języku Python oraz przeprowadzenie niezbędnych eksperymentów, w ramach których poszukiwane były najbardziej optymalne konfiguracje parametrów sieci a także najkrótszy czas wykonania.

# Zbiór danych

## Przedstawienie zbioru

Baza danych Iris Data Set [1] wykorzystana w tym projekcie pochodzi z repozytorium uczenia maszynowego UC Irvine. Jest to prawdopodobnie najbardziej znana baza danych, jaką można znaleźć w literaturze dotyczącej rozpoznawania wzorców. Zawiera 3 klasy po 50 instancji. Nie posiada brakujących/nieokreślonych wartości dlatego nie jest konieczne usuwanie rekordów. Każda z instancji składa się z 4 *atrybutów* w formacie liczb rzeczywistych oraz atrybutu klasy, która odnosi się do typu rośliny.

**Informacje o atrybutach:**

1. **Długość kielicha** (w centymetrach)
2. **Szerokość kielicha** (w centymetrach)
3. **Długość płatka** (w centymetrach)
4. **Szerokość płatka** (w centymetrach)
5. **Klasa:**
   * Iris Setosa
   * Iris Versicolour
   * Iris Virginica

## Przygotowanie i normalizacja danych

Klasy należało przekształcić z literałów na wartości liczbowe aby ujednolicić dane oraz sprawniej przeprowadzać operacje porównywania. Literały klas wymienione w poprzednim podpunkcie przekształcono odpowiednio w liczby naturalne z przedziału od 1 do 3. Separator atrybutów zamieniono z przecinka (‘,’) na średnik (‘;’).

Kolejnym krokiem była normalizacja danych w celu umożliwienia ich wzajemnego porównywania i dalszej analizy. Wszystkie atrybuty poza klasami zostały przeliczone na odpowiadające im wartości w przedziale od -1 do 1. Zostało to osiągnięte dzięki zastosowaniu poniższego wzoru [3]:

(2.1)

gdzie:

* – wartość znormalizowana
* – maksymalna wartość dla danej cechy w zbiorze po normalizacji
* – minimalna wartość dla danej cechy w zbiorze po normalizacji
* – maksymalna wartość dla danej cechy w zbiorze przed normalizacją
* – minimalna wartość dla danej cechy zbiorze przed normalizacją
* – wartość przed normalizacją

Poniżej zostały zamieszczone dane oryginalne oraz odpowiadające im dane po przygotowaniu i normalizacji, aby zobrazować zaistniałe zmiany.

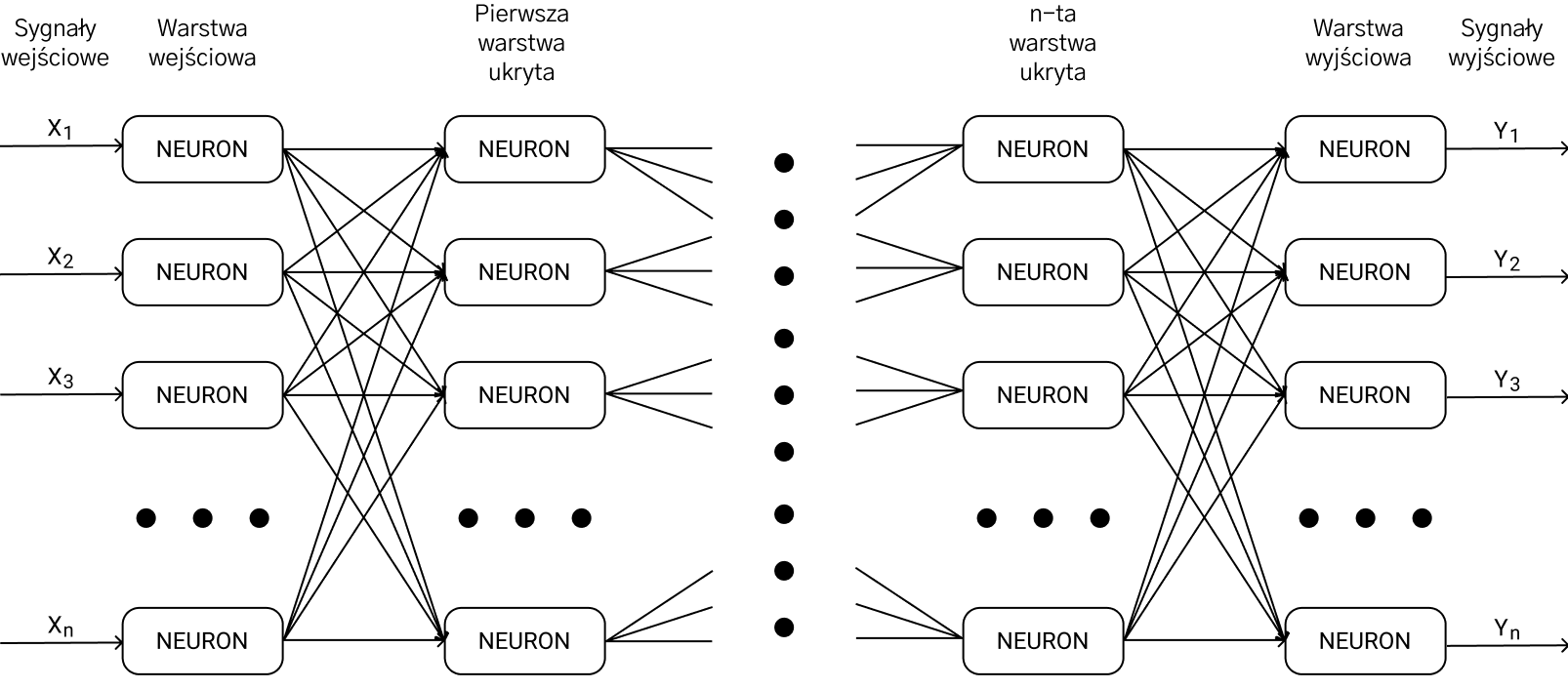
|  |
| --- |
| **Dane oryginalne** |
| 5.0,3.3,1.4,0.2,Iris-setosa  7.0,3.2,4.7,1.4,Iris-versicolor |

|  |
| --- |
| **Dane po przygotowaniu i normalizacji** |
| -0.611111;0.083333;-0.864407;-0.916667;1  0.5;0.0;0.254237;0.083333;2 |

# Wprowadzenie teoretyczne

## Sieć neuronowa

Termin *sztuczne sieci neuronowe* (SSN) lub również często określane jako po prostu *sieci neuronowe* (NN), obejmują rodzinę nieliniowych metod obliczeniowych, które przynajmniej we wczesnej fazie ich rozwoju były inspirowane funkcjonowaniem ludzkiego mózgu. Są to bardzo wyrafinowane narzędzia obliczeniowe zdolne do modelowania niezwykle złożonych funkcji. Składa się ona z dużej liczby elementów przetwarzających informacje. Elementy te, zwane *neuronami*, powiązane są ze sobą w sieć wzajemnie zależnych od siebie współczynników. Połączenia oraz ich parametry są podstawą działania sieci a jej topologia może różnić się w zależności od obranego algorytmu. Strukturę taką dzieli się na warstwy: *wejściową, wyjściową* oraz warstwy *ukryte*. W każdej sieci obligatoryjnie znajduje się warstwa wejściowa – przyjmująca zestaw danych, który ma zostać poddany analizie, oraz wyjściową – zwracającą wyniki działania algorytmu. Warstw ukrytych może istnieć dowolna ilość a decyduje o tym model wybranej sieci. W większości współczesnych zastosowań realizuje się sieci o modelu wielowarstwowym. *Sygnały* przekazywane są pomiędzy neuronami poszczególnych warstw aż do najwyższej warstwy dającej odpowiedź, a ich wpływ na odpowiedź neuronu jest zależna od *wag*, które są odpowiednio dopasowywane w procesie *uczenia sieci.*

**

*Rysunek 1. Przykładowy model sieci neuronowej.*

*Źródło: opracowanie własne*

### Model sztucznego neuronu

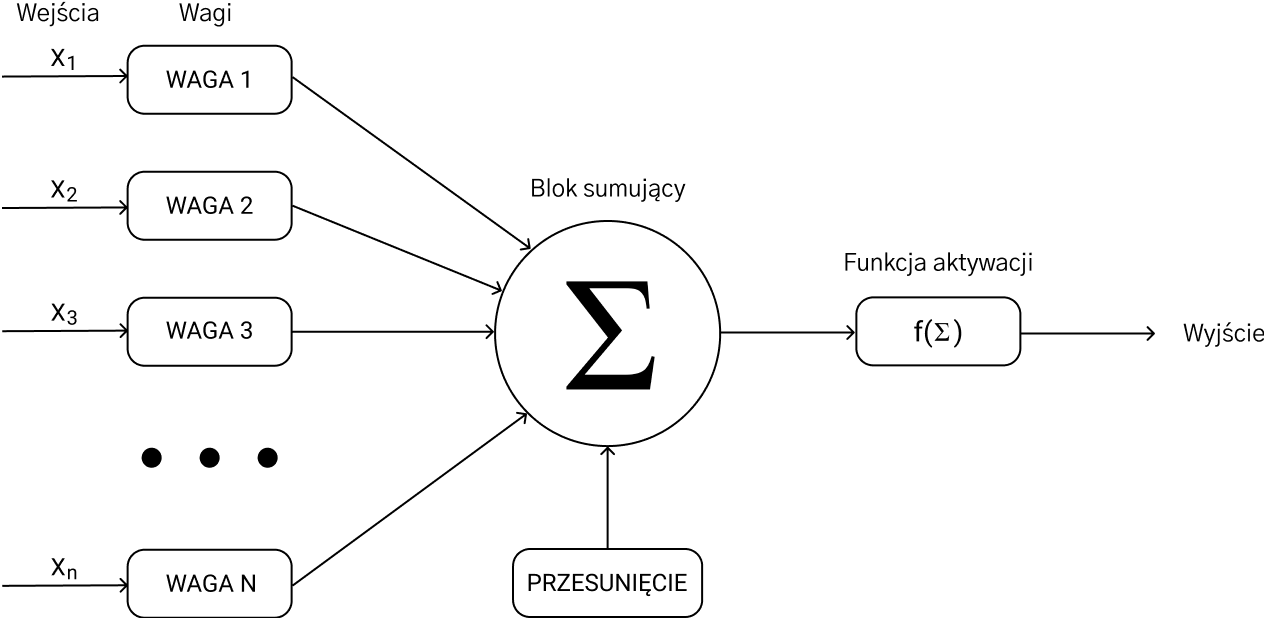
Mianem *sztucznego neuronu* określa się system przetwarzający sygnały przekazane do jego wejścia, na pojedynczą wartość wyjściową. Model sztucznego neuronu zwany *perceptronem* został stworzony na podstawie biologicznego neuronu.

Może on przyjmować dowolną ilość sygnałów wejściowych, najczęściej oznaczanych jako . Sygnały z wejść mnożone są z odpowiednimi wagami , a następnie sumowane wraz z przesunięciem w bloku sumującym [5]. Obliczona wartość poddawana jest działaniu funkcji aktywacji, która ostatecznie zwraca sygnał wyjściowy zwany *pobudzeniem neuronu*. Ogólny wzór na pobudzenie neuronu przedstawia poniższa zależność:

(3.1)

gdzie:

* – wartość i-tego sygnału wejściowego
* – waga i-tego sygnału wejściowego
* – ilość sygnałów wejściowych
* – przesunięcie
* – pobudzenie neuronu

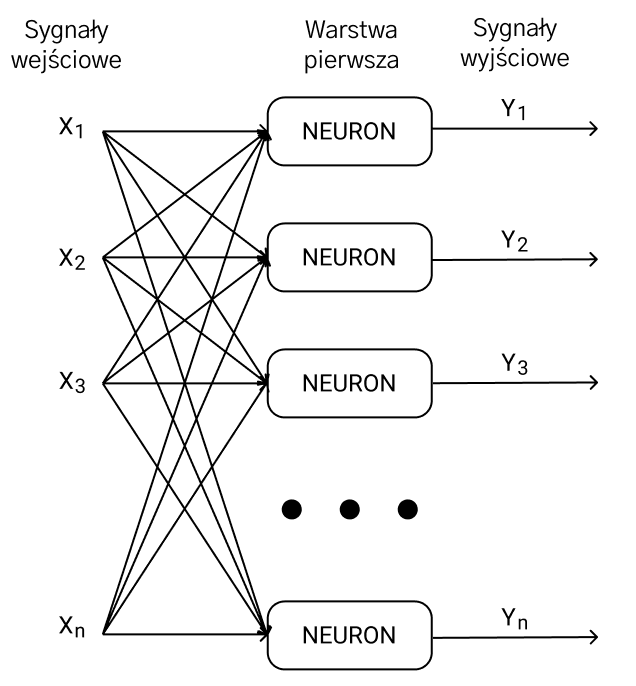


*Rysunek 2. Model sztucznego neuronu.*

*Źródło: opracowanie własne*

### Sieć jednowarstwowa

*Sieć jednowarstwowa* ma bardzo prymitywną budowę. Składa się jedynie z jednej warstwy neuronów oraz sygnałów wejściowych i wyjściowych. Nie posiada żadnych neuronów w warstwach ukrytych. Ma niewielkie znaczenie praktyczne, lecz nadal znajduje zastosowanie wszędzie tam, gdzie występuje niska złożoność problemu a istnienie jednej warstwy jest wystarczające do rozwiązania określonego zagadnienia. Neurony ułożone w pojedynczej warstwie działają niezależnie od siebie, stąd możliwości takiej sieci są ograniczone do możliwości pojedynczych neuronów [2]. Każdy z nich realizuje odwzorowanie funkcyjne pobudzenia neuronu (3.1). Przykładowa struktura sieci jednowarstwowej została zobrazowana poniżej.



*Rysunek 3. Model sieci jednowarstwowej.*

*Źródło: opracowanie własne*

### Sieć wielowarstwowa

*Sieci wielowarstwowe* znajdują zastosowanie w większości współczesnych realizacji sieci neuronowych. Tworzone są przez neurony ułożone w wielu warstwach [2]. Są rozszerzeniem sieci jednowarstwowej, jednak oprócz warstwy wejściowej i wyjściowej istnieje pomiędzy nimi co najmniej jedna warstwa ukryta. Sygnały wejściowe podawane są na wejścia neuronów warstwy wejściowej. Przetworzone dane są przekazywane na wyjścia neuronów, skąd trafiają na wejścia neuronów kolejnej warstwy. Proces ten jest powtarzany tyle razy ile jest warstw ukrytych. Na koniec, wyjścia ostatniej warstwy ukrytej są sygnałami wejściowymi ostatniej warstwy – warstwy wyjściowej. Działanie poszczególnych warstw można przedstawić według poniższej zależności:

(3.2)

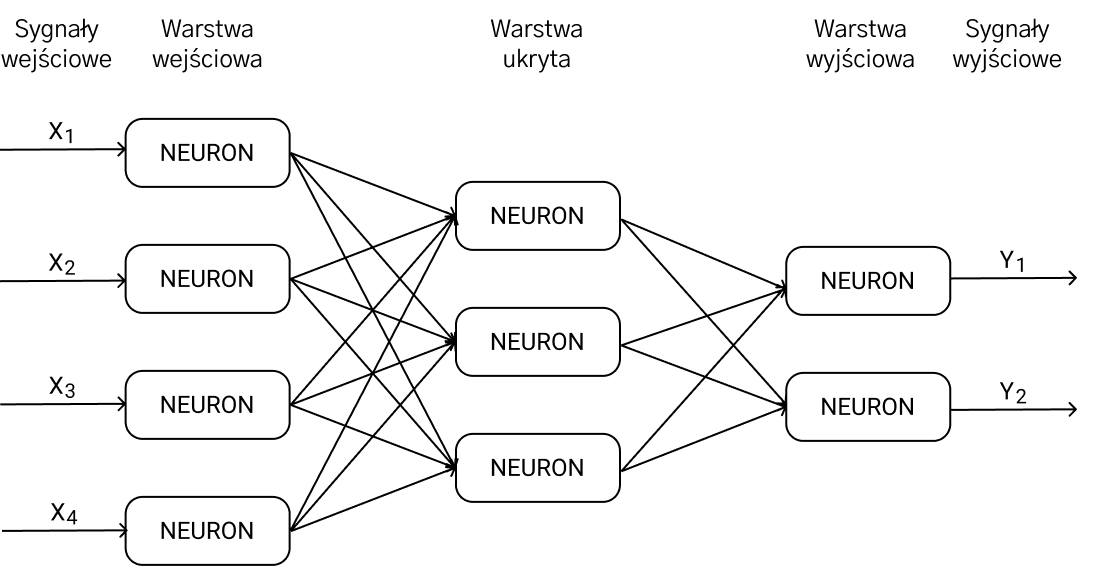
gdzie:

* – macierz wag dla n-tej warstwy
* – macierz wektorów wejściowych
* – wektor przesunięć dla n-tej warstwy
* – funkcja aktywacji dla n-tej warstwy
* – pobudzenie neuronu na n-tej warstwie

Dla zobrazowania działania sieci wielowarstwowych, poniżej został przedstawiony opis działania sieci trójwarstwowej [3]:

(3.3)

Do czego przydatne są warstwy ukryte? Powodem jest to, że jeśli nie ma ukrytych warstw to odwzorowywane dane wejściowe, nie są ze sobą w żaden sposób powiązane na wyjściach poszczególnych neuronów. W rzeczywistych problemach zmienne wejściowe są zwykle wysoce współzależne i wpływają na wynik w skorelowany sposób. Neurony warstwy ukrytej pozwalają uchwycić subtelne interakcje pomiędzy danymi wejściowymi, które ostatecznie wpływają wyjście. Ponadto, ukryte warstwy reprezentują cechy lub atrybuty wyższego poziomu naszych danych. Każdy z neuronów w warstwie ukrytej waży dane wejściowe inaczej, ucząc się innej pośredniczącej charakterystyki danych, a neuron wyjściowy jest wtedy ich funkcją zamiast surowych danych wejściowych. Włączając więcej niż jedną ukrytą warstwę, sieć dostaje możliwość nauczenia się wielu poziomów abstrakcji oryginalnych danych wejściowych przed uzyskaniem ostatecznego wyniku. Dzięki wykorzystaniu wielu warstw, sieć potrafi rozwiązywać problemy, z którymi nie radzi sobie sieć jednowarstwowa. Funkcje aktywacji zwiększają jej zdolność do wychwytywania nieliniowych relacji między wejściami a wyjściami. Łącząc ze sobą wiele nieliniowych transformacji za pomocą warstw, radykalnie zwiększa to elastyczność i zastosowanie sieci neuronowych [4].



*Rysunek 4. Przykładowa struktura sieci trójwarstwowej.*

*Źródło: opracowanie własne*

### Sieć jednokierunkowa

*Sieć jednokierunkowa* jest jednym ze sposobów działanie sieci neuronowych. Taką architekturę posiadają najprostsze sieci neuronowe. Informacje są podawane tylko w jednym kierunku – każdy neuron reaguje na jeden zestaw danych wejściowych tylko raz. Są częściej stosowane ze względu na prostotę w analizie. Sieci takie nazywane są czasami *heteroasocjacyjnymi* i z założenia są rozpatrywane jako twory statyczne, ponieważ ewentualne procesy przejściowe, zachodzące w sieci podczas jej pracy, nie mają znaczenia z punktu widzenia celu funkcjonowania sieci i mogą być pomijane[5].

### Sieć rekurencyjna

Bardziej złożone sieci są *rekurencyjne*, co znaczy, że dane wychodzące z ukrytych warstw są podawane pośrednio lub bezpośrednio do jednej z poprzednich warstw. Sprzężenie zwrotne powoduje, że sieć po jakimś czasie osiąga stan równowagi i „decyduje się” na konkretną odpowiedź. Takie sieci określane bywają jako *autoasocjacyjne* i odznaczają się bogatymi własnościami dynamicznymi, ponieważ sygnały mogą krążyć w sieci dowolnie długo, tym samym powodując powstanie różnorodnych i ciekawych przebiegów sieci przejściowych [5].

### Uczenie sieci neuronowej

*Uczenie sieci* to proces dobierania odpowiedniego zestawu wag dla danego zadania. Można je podzielić na *uczenie nadzorowane* oraz *uczenie bez nauczyciela*. Metodę uczenia pod nadzorem stosuje się gdy problem do rozwiązania jest zrozumiały dla człowieka oraz spodziewana jest określona odpowiedź od sieci. Wiedząc jakie sygnały są poprawne na wyjściu sieci dla każdego sygnału możemy je porównać z aktualną odpowiedzią sieci, po czym w razie potrzeby zmodyfikować wagi neuronów. Dobór odpowiednich wag odpowiada procesowi minimalizacji funkcji błędu, którą można zapisać jako:

(3.4)

gdzie:

* – odpowiedź n-tego neuronu
* – poprawny sygnał dla n-tego neuronu
* – ilość neuronów
* – błąd dopasowania

W zależności od wybranego algorytmu, funkcja skorygowania wag sygnałów może wyglądać w inny sposób. Jednakże w większości przypadków funkcja ta będzie przybierać postać zbliżoną do poniższej:

(3.5)

gdzie:

* – nowa macierz wektorów wag
* – aktualna macierz wektorów wag
* – współczynnik uczenia
* – błąd dopasowania
* – macierz wektorów wejściowych

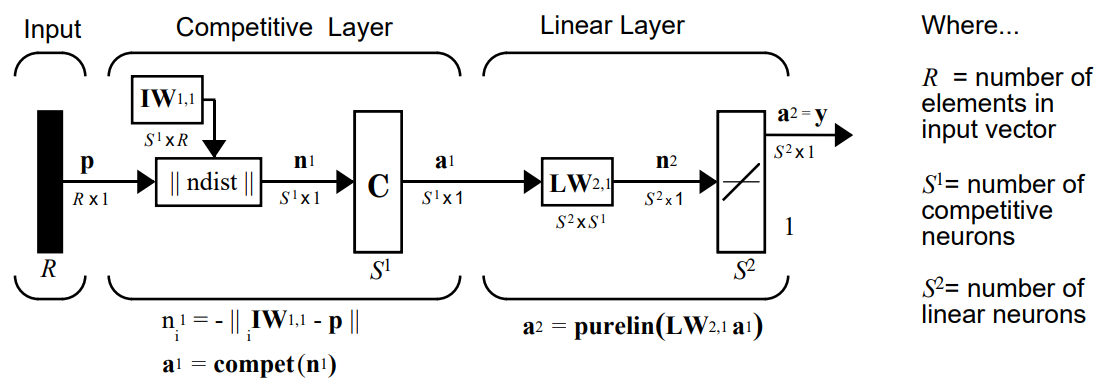
W sytuacjach gdy problem jest zbyt skomplikowany albo określone przez ludzi cechy prowadzą do niesatysfakcjonującego rozwiązania, można używać sposobu nauczania bez nauczyciela. Sieć ucząca się w taki sposób musi być odpowiednio złożona, wielowarstwowa, żeby mogła wykrywać w danych wejściowych różnorodne zależności. Ponieważ sieci uczone bez nadzoru nie dążą do zadanego wyniku, mają potencjał wykrywania prawidłowości i zależności, które wcześniej nie były znane. To powoduje, że są użyteczne do analizy zjawisk, których model jest niedokładny albo niekompletny – a to można powiedzieć prawie o wszystkich zjawiskach społecznych albo finansowych. Jednak stopień złożoności architektury takiej sieci jest wysoko zaawansowany a również nie pokrywa się z tematem projektu, toteż nie zostanie bardziej szczegółowo opisany w tym raporcie.

## Algorytm LVQ

Algorytm *Nauki Kwantyzacji Wektorów* (ang. Learning Vector Quantization) to adaptacyjna metoda klasyfikacji danych oparta na danych szkoleniowych z informacjami o klasie. Został opracowany i jest najlepiej rozumiany jako algorytm klasyfikacji. Jest siecią jednokierunkową wielowarstwową. Obsługuje zarówno binarne (dwuklasowe), jak i wieloklasowe problemy klasyfikacji. Jest bardzo podobny do algorytmu k-najbliższego sąsiada. Jest prekursorem *samoorganizujących się map cech* (SOFM) [6]. Został opracowany przez Teuvo Kohonena.

Neurony zwane tutaj również *wektorami kodującymi* są w rzeczywistości wektorami zdefiniowanymi w przestrzeni wejściowej z przypisanymi dla nich etykietami klas, przy czym liczba neuronów jest znacznie mniejsza od liczby *wektorów treningowych*. Koncepcja algorytmu LVQ bazuje na przyciąganiu do siebie wektorów kodujących przez wektory z tą samą etykietą klasy i odpychaniu od siebie wektorów kodujących o niezgodnych etykietach klas. Wektory kodujące są inicjalizowane dla losowo wybranych wartości z uczącego zestawu danych. Następnie, na przestrzeni kilku epok, są one dostosowywane, aby jak najlepiej podsumować dane szkoleniowe za pomocą algorytmu uczącego. To jaką liczbę epok, neuronów czy wartość współczynnika uczenia przyjąć, aby algorytm jak najlepiej klasyfikował dane, należy rozstrzygnąć eksperymentalnie. Zrobione to zostanie w rozdziale 5 tego raportu.

W architekturze sieci LVQ można wyróżnić dwie kluczowe warstwy – *warstwę konkurencyjną* oraz *warstwę liniową*. Warstwa konkurencyjna uczy się klasyfikować wektory wejściowe w podobny sposób, jak robią to samoorganizujące się mapy funkcji. Warstwa liniowa przekształca warstwę konkurencyjną na docelowe klasyfikacje zdefiniowane przez użytkownika. Warto zauważyć, że struktura nie uwzględnia przesunięcia (tzw. biasu).



*Rysunek 5. Struktura budowy sieci LVQ.*

*Źródło: Neural Network Toolbox 5 User’s Guide*

### LVQ1

Algorytm uczenia sieci typu LVQ w wersji pierwszej polega na iteracyjnej aktualizacji położenia wektorów kodujących tak by zminimalizować błąd klasyfikacji. Jest to wersja algorytmu, którą wykorzystano w tym projekcie. Proces iteracyjny składa się z głównej pętli, która wykonywana jest określoną liczbę epok, oraz pętli w której iteracja następuje po elementach zbioru treningowego. Elementy tego zbioru losowane są z książki kodowej a ich ilość jest równa ilości ustanowionych neuronów . Wówczas dla każdego wektora treningowego poszukiwany jest najbliżej leżący wektor kodujący. Odległość wektorów obliczana jest według wzoru na *odległość Euklidesową*, która ma postać:

(3.6)

gdzie:

* – wektor treningowy
* – wektor uczący
* – odległość Euklidesowa

Warstwa konkurencyjna opiera się na typie algorytmu *„zwycięzca bierze wszystko”* dlatego, że wybierany jest tylko jeden neuron z całego zbioru, którego pobudzenie będzie wynosiło 1 – reszta neuronów na wyjście podaje sygnał 0. Jeżeli obydwa wektory (kodujący i treningowy) należą do tej samej klasy wówczas aktualizacja wag wektora kodującego odbywa się wg. zależności (3.7) (zwanej również *Regułą uczenia Kohonenea* [7]) jeżeli natomiast klasy wektora kodującego i treningowego są różne aktualizacja wag odbywa się wg. zależności (3.8) [6].

(3.7)

(3.8)

gdzie:

* – nowy i-ty wektor wag
* – aktualny i-ty wektor wag
* – aktualny współczynnik uczenia (maleje liniowo w każdej epoce, obliczany ze wzoru:
* – j-ty wektor treningowy

### LVQ2.1

LVQ2.1 opiera się na takiej samej zasadzie klasyfikacji jak wersja pierwsza lecz różni się metodą wybierania wygranego neuronu. Podczas uczenia wybierane są dwa wektory z książki kodowej, które są najbliższymi sąsiadami wektora treningowego. To właśnie one będą podlegać modyfikacji wag. Aby jednak mogło się to dokonać muszą zostać spełnione trzy warunki:

1. Klasa najbliższego wektora kodującego musi być różna od klasy wektora treningowego,
2. Klasa drugiego najbliższego wektora musi być taka sama jak klasa wektora treningowego,
3. Wektor treningowy musi należeć do strefy wartości zwanej „oknem”, zdefiniowanym jako:

(3.9)

gdzie:

* – odległość Euklidesowa do najbliższego wektora
* – odległość Euklidesowa do drugiego najbliższego wektora
* – zmienna definująca szerokość okna (zalecana wartość zawiera się w przedziale od 0.2 do 0.3 [7])

Jeśli te warunki są spełnione, nieprawidłowy wektor kodujący jest odsunięty od wektora treningowego, podczas gdy prawidłowy wektor kodujący zbliża się do wektora treningowego, zgodnie odpowiednio z modyfikacją wag w algorytmie LVQ1 (zależności (3.7) oraz (3.8)).

Algorytm LVQ w wersji drugiej pozwala osiągnąć lepsze dokładności dopasowania, jednak **należy go stosować tylko po uprzednim zastosowaniu wersji pierwszej!** LVQ2.1 powinien być używany tylko w sposób zrównoważony, wykorzystując niewielką wartość współczynnika uczenia się i stosunkowo małą liczbę epok.

### LVQ3

Kolejną wariancją algorytmu LVQ jest wersja trzecia. Powstała z powodu problemów, które wykazuje wersja druga. Chociaż naukowcy odnotowali dobre wyniki z algorytmem LVQ2.1, zgłosili również pewne problemy. Po pierwsze dlatego, że poprawki LVQ2.1 są proporcjonalne do różnicy między wektorami kodującymi a wektorem trenującym , a korekta wektora o właściwej klasie ma znacznie większą wielkość niż poprawka na wektor o złej klasie. Rozwiązaniem tego problemu jest rozszerzenie warunków algorytmu w wersji drugiej o dodatkowy warunek sformułowany następująco:

Jeśli obydwa najbliższe wektory kodujące mają taką samą klasę jak wektor treningowy, zmodyfikuj wagi obydwu wektorów według następującego wzoru:

(3.10)

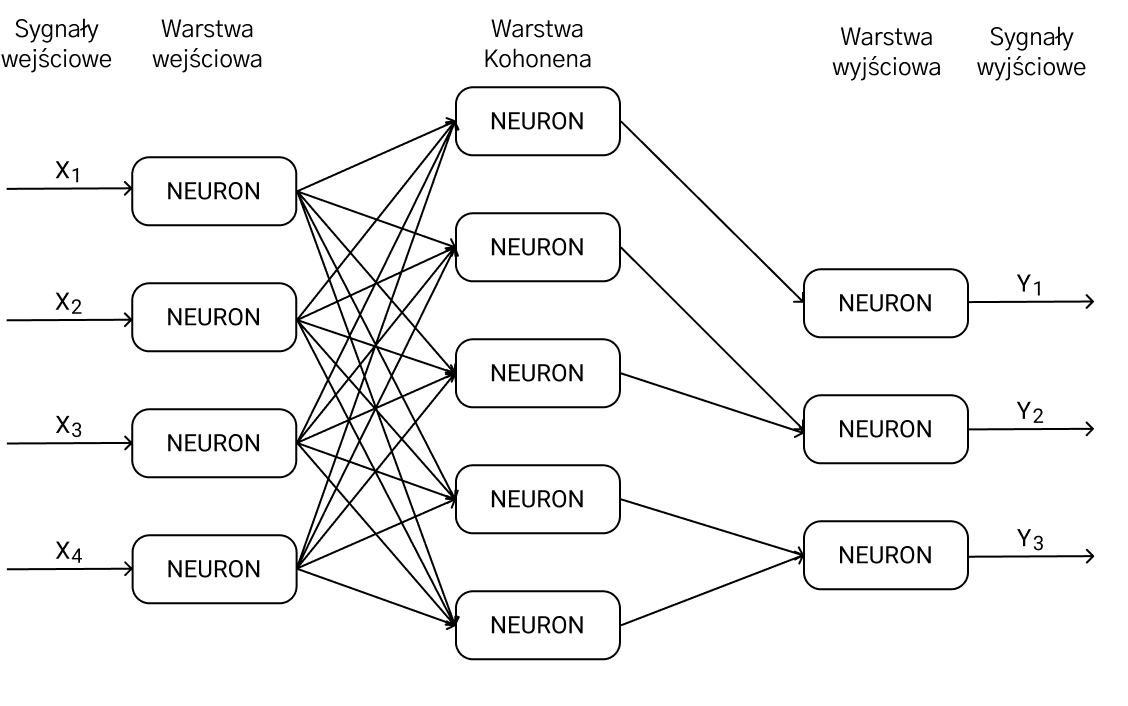
gdzie:

* – nowy i-ty oraz k-ty wektor wag
* – aktualny i-ty oraz k-ty wektor wag
* – aktualny współczynnik uczenia (maleje liniowo w każdej epoce, obliczany ze wzoru:
* – j-ty wektor treningowy
* – zmienna definująca wpływ modyfikacji na wagi (zalecana wartość zawiera się w przedziale od 0.1 do 0.5 [8])

Pozostałe zasady odnoszące się do szerokości okna oraz modyfikacji wag najbliższych wektorów kodujących, w przypadku gdy jeden z nich odpowiada klasie wektora treningowego a drugi nie, nie ulegają zmianie w porównaniu do poprzedniej wersji algorytmu. LVQ3 podobnie jak LVQ1 definiują bardziej niezawodny proces, w których wektory kodujące przyjmują właściwe wartości nawet po dłuższych okresach uczenia się. Warto w tym miejscu zauważyć, że algorytm LVQ1 modyfikuje tylko jeden wektor kodujący jednocześnie, podczas gdy LVQ2.1 oraz LVQ3 modyfikują jednocześnie dwa wektory.

### Warstwa Kohonena

*Warstwa Kohonena* jest warstwą w algorytmie LVQ odpowiadającą za uczenie sieci z rywalizacją. W tym miejscu wektory kodujące rywalizują ze sobą o najlepsze odwzorowanie wektora treningowego. We współzawodnictwie zwycięża jeden neuron, którego wagi najmniej różnią się od odpowiednich składowych wektora treningowego. Następnie wagi zwycięskiego neuronu są aktualizowane aby jeszcze lepiej dopasować się do reszty podobnych sobie wektorów. Proces ich aktualizacji dokładniej opisany jest w rozdziale 3.2.1. Najlepsze efekty uzyskuje się wprowadzając metody Kohonena dla sieci neuronowych dwuwymiarowych, to znaczy takich, w których neurony ułożone są w strukturę tablicy o pewnej liczbie wyróżnionych wierszy i kolumn [5]. Ilość neuronów w warstwie Kohonena jest uwarunkowana parametrami początkowymi sieci, zaś w warstwie wyjściowej odpowiada liczbie klas w zbiorze danych.

**

*Rysunek 6. Przykładowa struktura sieci z warstwą Kohonena.*

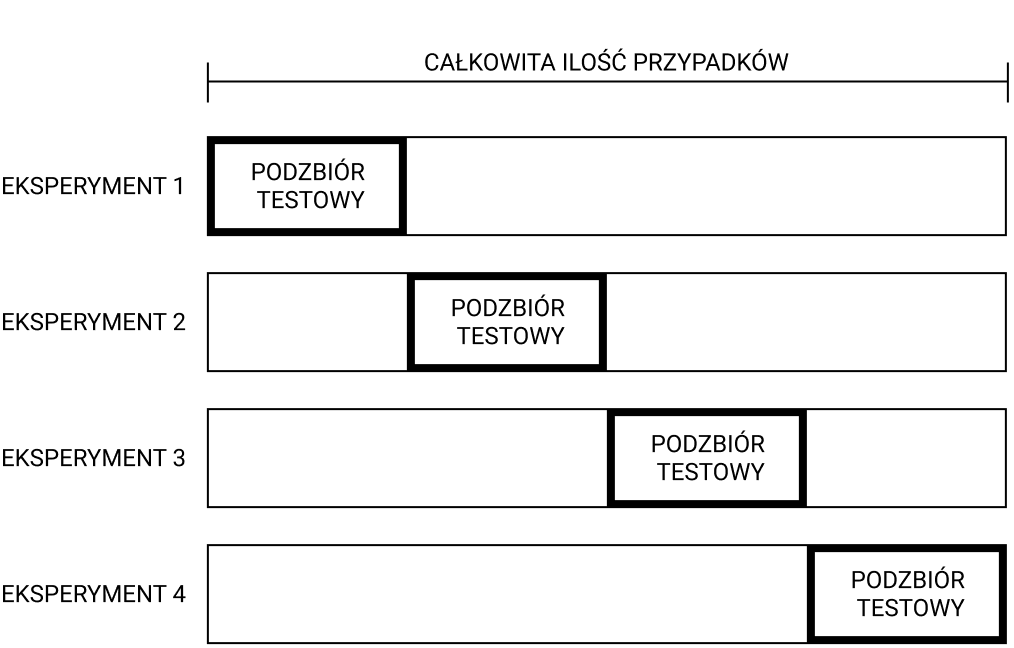
*Źródło: opracowanie własne*

## Działanie sieci nauczonej

Efektem uczenia sieci neuronowej LVQ jest *warstwa topologiczna*. Neurony należące do tej warstwy specjalizują się w identyfikowaniu poszczególnych obiektów, jakie w trakcie procesu nauczania były wprowadzane na wejście sieci. Każdy neuron tej warstwy ma przypisany do siebie obiekt, którego pojawienie się na wejściu sieci powoduje, że ten właśnie neuron zostaje zwycięzcą [11]. Proces uczenia odbywa się w warstwie rywalizacji, natomiast dopasowywanie obiektów do danej klasy dzieje się wewnątrz warstwy liniowej. Wynikiem działania sieci jest zbiór wektorów z przyporządkowanymi do nich klasami (które zostały rozpoznane poprawnie lub nie).

## Walidacja krzyżowa

Do oceny zdolności sieci nauczonej można wykorzystać metodę określaną jako *walidacja krzyżowa*. Metoda ta polega na podziale zbioru danych wejściowych na podzbiory, pośród których wybiera się jeden z nich jako zbiór walidacyjny, natomiast pozostałe służą do uczenia sieci. Stosuje się ją szczególnie wtedy, kiedy liczba danych wejściowych jest ograniczona. Metoda walidacji krzyżowej posiada kilka odmian, jednak w tym projekcie zastosowano walidację *n-krotną*. Przy tym sposobie zbiór danych uczących jest dzielony na n rozłącznych podzbiorów. Operacja jest powtarzana n razy, tak aby każdy z podzbiorów pełnił rolę zbioru walidacyjnego. Skuteczność sieci określa się jako średnią poprawności klasyfikacji dla poszczególnych zbiorów walidacyjnych [11].



*Rysunek 7. Schemat działania walidacji krzyżowej.*

*Źródło: opracowanie własne*

# Kod programu

Algorytm realizujący sieć neuronową LVQ został napisany w języku Python [12]. Do tego celu nie została wykorzystana żadna biblioteka zewnętrzna oferująca gotowe rozwiązanie przedstawionego problemu. Do wizualizacji danych posłużono się biblioteką *matplotlib* [9]. Cały kod programu zamieszczonego poniżej można znaleźć na repozytorium [10].

|  |
| --- |
| from random import seed  from random import randrange  from copy import deepcopy  from mpl\_toolkits import mplot3d  import numpy as np  import matplotlib.pyplot as plt  from matplotlib import cm  from matplotlib.colors import LightSource  import datetime  ####################  # Zmienne globalne #  ####################  plikZrodlowy = open("iris.data", "r")  # wczytanie do zmiennej pliku z danymi  P = list()  # zbiór wejściowy (tu znajdują się cechy obiektu)  T = list()  # zbiór identyfikatorów (tu znajdują się identyfikatory, po których klasyfikowane są obiekty)  kolumnaT = 4  # indeks kolumny T  zbiorDanych = list()  czyNormalizowacDane = True  ################  # Algorytm LVQ #  ################  def lvq(daneUczace, daneTestowe, iloscNeuronow, wspUczenia, iloscEpok):      zestawDanychZmodyfikowanych = forward(daneUczace, iloscNeuronow,                            wspUczenia, iloscEpok)  # trenuj zestaw danych      prognozy = list()  # zainicjalizuj pustą listę prognoz      for wektor in daneTestowe:  # dla każdego wektora w danych testowych          output = prognoza(zestawDanychZmodyfikowanych, wektor)  # dopasuj klasyfikator          prognozy.append(output)  # dodaj wynik do listy prognoz      return prognozy  ###########  # Funkcje #  ###########  def rzutujListeNaTyp(lista, typ):      # zmienia typ danych dla całej listy a następnie zwraca ją      return list(map(typ, lista))  def odlegloscEuklidesowa(wektor1, wektor2):      # oblicza dystans pomiędzy dwoma wektorami danych (im mniejsza odległość tym mniej się różnią)      dystans = 0.0      for i in range(len(wektor1)-1):  # dla każdej cechy w wektorze utworz sume          dystans += (wektor1[i] - wektor2[i])\*\*2      return dystans\*\*(1/2)  def dopasujNajlepszyWektor(wylosowaneNeurony, wektor\_podstawowy):      # wyszukuje najlepiej dopasowany wektor ze zbioru danych dla podanego wektora podstawowego      dystanse = list()  # zainicjuj pustą listę      for wektor in wylosowaneNeurony:  # dla kazdego wektora w podanym zbiorze          # oblicz odległość do podanego wektora          dist = odlegloscEuklidesowa(wektor, wektor\_podstawowy)          dystanse.append((wektor, dist))  # i dodaj obliczną wartość do listy      # posortuj rosnąco listę z dystansami      dystanse.sort(key=lambda tup: tup[1])      # wybierz pierwszy element z listy (z najmniejszą odległością)      return dystanse[0][0]  def losowyWektorZDanych(daneUczace):      # dla podanego zbioru danych losuje jego losowy wektor      iloscRekordow = len(daneUczace)  # ile rekordow jest w zbiorze      return daneUczace[randrange(iloscRekordow)] # wylosuj wektor w przedziale  def forward(dane, iloscNeuronow, wspUczenia, epoki):      # procedura uczenia zestawu wektorów ze zbioru danych      wylosowaneNeurony = [losowyWektorZDanych(dane) for i in range(iloscNeuronow)] # wylosowanie z listy zbiorów podanej ilości neuronow      for epoch in range(epoki):  # dla podanej ilości epok          # aktualny współczynnik uczenia          rate = wspUczenia \* (1.0-(epoch/float(epoki)))          for wektorTreningowy in dane:  # dla każdego wektora w podanym zbiorze danych              # dopasuj najlepszy wektor z wylosowanego zbioru neuronów              bmu = dopasujNajlepszyWektor(wylosowaneNeurony, wektorTreningowy)              for i in range(len(wektorTreningowy)-1):  # dla każdej cechy                  error = wektorTreningowy[i] - bmu[i]  # oblicz różnicę cech                  if bmu[-1] == wektorTreningowy[-1]: # jeśli najbliższy neuron dla aktualnego wektora ma taką samą klasę                      bmu[i] += rate \* error # to przybliż neuron do wektora                  else:                      bmu[i] -= rate \* error  # to oddal neuron od wektora      return wylosowaneNeurony  def prognoza(zestawDanych, wektor):      # Dokonaj prognozy z wektorami danych z zestawu danych      # wyszukaj z zestawu danych nalepiej dopasowany wektor do podanego      bmu = dopasujNajlepszyWektor(zestawDanych, wektor)      return bmu[-1]  def podzielDane(dane, iloscPrzedzialow):      # Podziel dane na k przedziałów      podzialDanych = list()  # zainicjalizuj pustą liste podzialow      kopiaDanych = list(dane)  # utwórz kopie danych      # ustal szerokosc przedzialu (np. iloscPrzedzialow = 5; => 150/5 = 30 elementowe przedzialy)      szerokoscPrzedzialu = int(len(dane) / iloscPrzedzialow)      for i in range(iloscPrzedzialow):  # dla każdego przedziału          przedzial = list()  # zainicjalizuj pustą listę          # dopóki dlugość listy jest mniejsza niż ustalona szerokość przedziału          while len(przedzial) < szerokoscPrzedzialu:              # wylosuj wektor z zestawu danych              indeks = randrange(len(kopiaDanych))              # dodaj go do aktualnego przedziału i usuń ze zbioru do losowania              przedzial.append(kopiaDanych.pop(indeks))          podzialDanych.append(przedzial)  # dodaj przedział do listy przedziałów      return podzialDanych  def wskaznikPrecyzjiDopasowania(wlasciwy, otrzymany):      # Oblicz procent dokładności      blad = 0 # zainicjalizuj początkową wartość błędu średniokwadratowego      poprawnie = 0  # zainicjalizuj początkową wartość poprawności danych      for i in range(len(wlasciwy)):  # dla każdego z klasyfikatorów          blad += (otrzymany[i] - wlasciwy[i])\*\*2 # oblicz różnicę klasyfikatorów          if wlasciwy[i] == abs(round(float(otrzymany[i]))): # sprawdź czy otrzymany wynik pokrywa się z danymi uczącymi              poprawnie += 1  # jeśli tak to zwiększ poprawność      return [poprawnie / float(len(wlasciwy)) \* 100, blad / float(len(wlasciwy))]  def ocenAlgorytm(dane, iloscPrzedzialow, iloscNeuronow, wspolczynnikUczenia, iloscEpok):      # Ocenia algorytm używając podziału walidacji krzyżowej      przedzialy = podzielDane(dane, iloscPrzedzialow)  # utwórz przedziały      wyniki = list()  # zainicjalizuj pustą listę z wynikami procentowymi      bledy = list()  # zainicjalizuj pustą listę z błędami śreniokwadratowymi      for przedzial in przedzialy:  # dla każdego z utworzonych przedziałów          daneUczace = list(przedzialy)  # utwórz kopie przedziałów          daneUczace.remove(przedzial)  # usuń aktualny przedział          daneUczace = sum(daneUczace, [])          daneTestowe = list()  # zainicjalizuj pustą listę dla danych testowych          for wektor in przedzial:  # dla każdego wektora w przedziale              wektorKopia = list(wektor)  # utwórz jego kopie              daneTestowe.append(wektorKopia)  # dodaj wektor do danych testowych              # wektorKopia[-1] = None # usuń klasyfikator          otrzymany = lvq(daneUczace, daneTestowe, iloscNeuronow,                          wspolczynnikUczenia, iloscEpok)  # uzyskaj prognozy          wlasciwy = [wektor[-1]for wektor in przedzial]  # wyciągnij klasyfikatory          precyzja = wskaznikPrecyzjiDopasowania(wlasciwy, otrzymany)  # sprawdź poprawność klasyfikacji          wyniki.append(precyzja[0])  # dodaj wynik procentowy do wyników          bledy.append(precyzja[1])  # dodaj wynik procentowy do błędów      return [wyniki,bledy]  def rysujGraf(x,y,PK, MSE):      x, y = np.meshgrid(x, y) # utwórz przestrzeń XY      fig, (ax, bx) = plt.subplots(1,2, subplot\_kw=dict(projection='3d')) # zainicjalizuj osie      ls = LightSource(270,45) # ustaw kąt padania światła      rgb1 = ls.shade(PK, cmap=cm.brg, vert\_exag=0.1, blend\_mode='soft') # określ kolory wykresu      rgb2 = ls.shade(MSE, cmap=cm.summer, vert\_exag=0.1, blend\_mode='soft') # określ kolory wykresu      ax.plot\_surface(x, y, PK, rstride=1, cstride=1, facecolors=rgb1,                          linewidth=0, antialiased=False, shade=False) # utwórz wykres      bx.plot\_surface(x, y, MSE, rstride=1, cstride=1, facecolors=rgb2,                          linewidth=0, antialiased=False, shade=False)      ax.view\_init(30, -135) # ustaw początkowy punkt widzenia      bx.view\_init(30, -135) # ustaw początkowy punkt widzenia      ax.set\_xlabel('współczynnik uczenia') # tytuł osi x wykresu PK      ax.set\_ylabel('liczba neuronów') # tytuł osi y wykresu PK      ax.set\_zlabel('poprawność klasyfikacji') # tytuł osi z wykresu PK      bx.set\_xlabel('współczynnik uczenia') # tytuł osi x wykresu MSE      bx.set\_ylabel('liczba neuronów') # tytuł osi y wykresu MSE      bx.set\_zlabel('błąd średniokwadratowy') # tytuł osi z wykresu MSE      ax.set\_title('PK(S1, lr)') # tytuł wykresu PK      bx.set\_title('MSE(S1, lr)') # tytuł wykresu MSE      ax.set\_zlim(0, 100) #  określ przedział osi Z dla PK od 0 do 100      ax.zaxis.set\_major\_formatter('{x:.00f}%')      plt.show() # wyświetl graf  ########  # Main #  ########  # \*\*\* Przygotowanie danych \*\*\*  # tutaj przetrzymywane są całe kolumny zmiennych (czyli dla jednej cechy, ułatwia to wyszukanie minimum oraz maksimum)  kolumny = []  for linia in plikZrodlowy:  # dla każdej linii w pliku z danymi      # rozdziel ciąg znaków na znakach separacji (;) oraz utwórz z nich wektor liczbowy      temp = rzutujListeNaTyp(linia.replace('\n', '').split(';'), float)      P.append(temp[0:kolumnaT])  # dodaj wektor cech do zbioru P      T.append(temp[kolumnaT])  # dodaj identyfikator klasy do zbioru T      i = 0      for liczba in temp[0:kolumnaT]:  # każdą liczbę w wektorze          if(len(kolumny) != kolumnaT):              kolumny.append([liczba])          else:              # dodaj do kolumny dla odpowiedniej cechy              kolumny[i].append(liczba)          i = 1 + i  minP = []  maxP = []  Pn = [0] \* len(P)  # zainicjuj pustą liste dla znormalizowanych danych  for kolumna in kolumny:  # dla każdej cechy      minP.append(min(kolumna))  # znajdź minimum      maxP.append(max(kolumna))  # oraz maksimum  if czyNormalizowacDane:      aktualnyWiersz = 0      for wektor in P:  # dla każdego wektora w zbiorze P          aktualnaKolumna = 0          for liczba in wektor:  # dla każdej liczby w wektorze              znormalizowana = ((1 - (-1)) \* (liczba - minP[aktualnaKolumna]) / (                  maxP[aktualnaKolumna] - minP[aktualnaKolumna]) + (-1))  # znormalizuj ją do przedziału <-1,1>              if(aktualnaKolumna == 0):                  # utwórz listę dla znormalizowanych danych dla aktualnego wektora                  Pn[aktualnyWiersz] = [round(znormalizowana, 6)]              else:                  # dodaj znormalizowane dane dla aktualnego wektora                  Pn[aktualnyWiersz].append(round(znormalizowana, 6))              aktualnaKolumna = 1 + aktualnaKolumna          aktualnyWiersz = 1 + aktualnyWiersz  zbiorDanych = [0] \* len(Pn)  # inicjalizacja pustej listy zbioru danych  for i in range(len(Pn)):      # łączenie znormalizowanego zbioru cech ze zbiorem klasyfikatorów      zbiorDanych[i] = (Pn[i] if czyNormalizowacDane else P[i]) + [T[i]]  # \*\*\* Ewaluacja algorytmu \*\*\*  seed(1)  iloscPrzedzialow = 5 # ilość przedziałów na które zostanie podzielony zbiór danych (potrzebne do walidacji krzyżowej)  wspolczynnikUczenia = [0.001, 0.01, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 0.99] # lista współczynników uczenia do przetestowania  iloscEpok = 2 # ilość powtórzeń przez które uczone są dane  iloscNeuronow = range(10, 51, 10) # lista ilości neuronów do przetestowania  najlepszeKombinacjePK = list() # inicjalizacja listy z najlepszymi wynikami  najlepszeKombinacjeMSE = list() # inicjalizacja listy z najlepszymi wynikami  najlepszaDokladnosc = float('-inf') # zmienna do określenia najlepszej precyzji dopasowania danych  najmniejszyBlad = float('inf') # zmienna do określenia najmniejszego błedu średniokwardatowego  zPK = list() # zmienna przetrzymująca obliczone parametry dla wykresu PK  zMSE = list() # zmienna przetrzymująca obliczone parametry dla wykresu MSE  start = datetime.datetime.now() # rozpoczęcie pomiaru czasu wykonania skryptu  for neuron in iloscNeuronow: # dla każdej ilości neuronów w liście      wierszProcentow = list() # zainicjalizuj pusty wektor dla obliczonych danych      wierszBledow = list() # zainicjalizuj pusty wektor dla obliczonych danych      for lr in wspolczynnikUczenia: # dla każdego współczynnika uczenia          poczatek = datetime.datetime.now() # rozpocznij pomiar czasu dla danej konfiguracji          copy = deepcopy(zbiorDanych) # utwórz kopię zbioru danych          wyniki = ocenAlgorytm(copy, iloscPrzedzialow, neuron, lr, iloscEpok) # uzyskaj wyniki poprawności klasyfikacji dla każdego przedziału          koniec = datetime.datetime.now() # zakończ pomiar czasu dla danej konfiguracji          czasWykonania = round((koniec - poczatek).total\_seconds() \* 1000) # oblicz czas obliczania danej konfiguracji          sredniaProcentowa = round((sum(wyniki[0])/float(len(wyniki[0]))),5) # wyciągnij średnią z uzyskanych wyników poprawności klasyfikacji          sredniaBledu = round((sum(wyniki[1])/float(len(wyniki[1]))),5) # wyciągnij średnią z uzyskanych wyników błędu śreniokwadratowego (MSE)          wierszProcentow.append(sredniaProcentowa) # do wektora obliczonych srednich dodaj aktualną wartość PK          wierszBledow.append(sredniaBledu) # do wektora obliczonych srednich dodaj aktualną wartość MSE          print('S1:', neuron, ', lr:', lr, ', {0: .3f}%'.format(sredniaProcentowa), ', MSE {0: .3f}'.format(sredniaBledu), ', czas wykonania: ', czasWykonania, 'ms') # wypisz wyniki aktualnej konfiguracji          if sredniaProcentowa > najlepszaDokladnosc: # jeśli aktualny wynik PK jest lepszy od poprzeniego              najlepszeKombinacjePK = list() # wyczyść liste najlepszych konfiguracji PK              najlepszaDokladnosc = sredniaProcentowa # przypisz nową najlepszą wartość              najlepszeKombinacjePK.append([neuron,lr, sredniaProcentowa, sredniaBledu, czasWykonania]) # dodaj aktualną konfigurację do listy najlepszych konfiguracji PK          elif len(najlepszeKombinacjePK) > 0 and sredniaProcentowa == najlepszaDokladnosc: # jeśli wynik jest tak samo dobry              najlepszeKombinacjePK.append([neuron,lr, sredniaProcentowa, sredniaBledu, czasWykonania]) # dodaj aktualną konfigurację do listy najlepszych konfiguracji PK            if sredniaBledu < najmniejszyBlad: # jeśli aktualny wynik MSE jest lepszy od poprzeniego              najlepszeKombinacjeMSE = list() # wyczyść liste najlepszych konfiguracji MSE              najmniejszyBlad = sredniaBledu # przypisz nową najlepszą wartość              najlepszeKombinacjeMSE.append([neuron,lr, sredniaProcentowa, sredniaBledu, czasWykonania]) # dodaj aktualną konfigurację do listy najlepszych konfiguracji MSE          elif len(najlepszeKombinacjeMSE) > 0 and sredniaBledu == najmniejszyBlad: # jeśli wynik jest tak samo dobry              najlepszeKombinacjeMSE.append([neuron,lr, sredniaProcentowa, sredniaBledu, czasWykonania]) # dodaj aktualną konfigurację do listy najlepszych konfiguracji MSE      zPK.append(wierszProcentow) # dodaj wektor obliczonych danych do listy dla wykresu PK      zMSE.append(wierszBledow) # dodaj wektor obliczonych danych do listy dla wykresu MSE  stop = datetime.datetime.now() # zakończ pomiar czasu wykonania skryptu  czasWykonania = round((stop - start).total\_seconds() \* 1000) # oblicz czas wykonywania skryptu  # \*\*\* Podsumowanie wyników \*\*\*  print('Czas wykonania skryptu: ', czasWykonania, 'ms') # wypisz czas wykonywania skryptu  najlepszeKombinacjePK.sort(key=lambda tup: tup[4]) # posortuj nalepsze kombinacje PK względem czasu wykonania  najlepszeKombinacjeMSE.sort(key=lambda tup: tup[4]) # posortuj nalepsze kombinacje MSE względem czasu wykonania  print('najlepsze PK', najlepszeKombinacjePK) # wyświetl najlepsze kofiguracje PK  print('najlepsze MSE', najlepszeKombinacjeMSE) # wyświetl najlepsze kofiguracje MSE  zPK = np.array(zPK) # oś Z wykresu PK  zMSE = np.array(zMSE) # oś Z wykresu PK  x = np.array(wspolczynnikUczenia) # oś X wykresów  y = np.array(iloscNeuronow) # oś Y wykresów  rysujGraf(x,y,zPK,zMSE) # narysuj wykres powierzchniowy 3D |

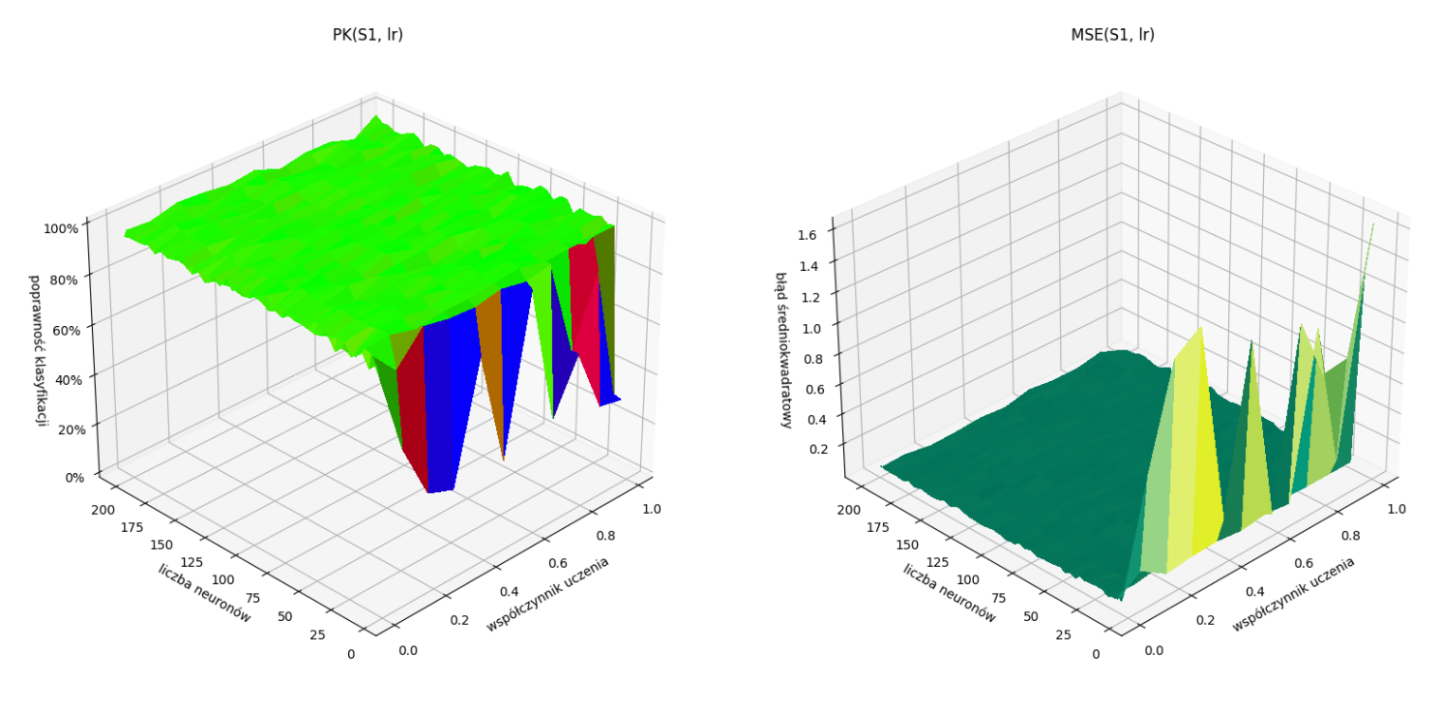
*Listing 1. Realizacja sieci neuronowej LVQ w programie Python.*

*Źródło: opracowanie własne*

# Eksperymenty

## Eksperyment pierwszy

Celem pierwszego eksperymentu było znalezienie dolnego kresu wartości parametrów współczynnika uczenia oraz ilości neuronów , przy których algorytm wyprowadzał zadowalająco dokładne wyniki a także górnego kresu, po którym zwiększanie tychże parametrów nie polepszy znacząco poprawności zwracanych wyników. Badanie rozpoczęto od wartości w granicach od 5 do 200 z krokiem 5 dla ilości neuronów, natomiast współczynniki uczenia dla każdej z tych wartości zdefiniowano następująco: 0.001, 0.01, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9 i 0.99. Ilość epok przyjęto na poziomie 50, a ilość przedziałów do walidacji krzyżowej na poziomie 5. Poniżej zamieszczono graficzną reprezentację uzyskanych wyników:



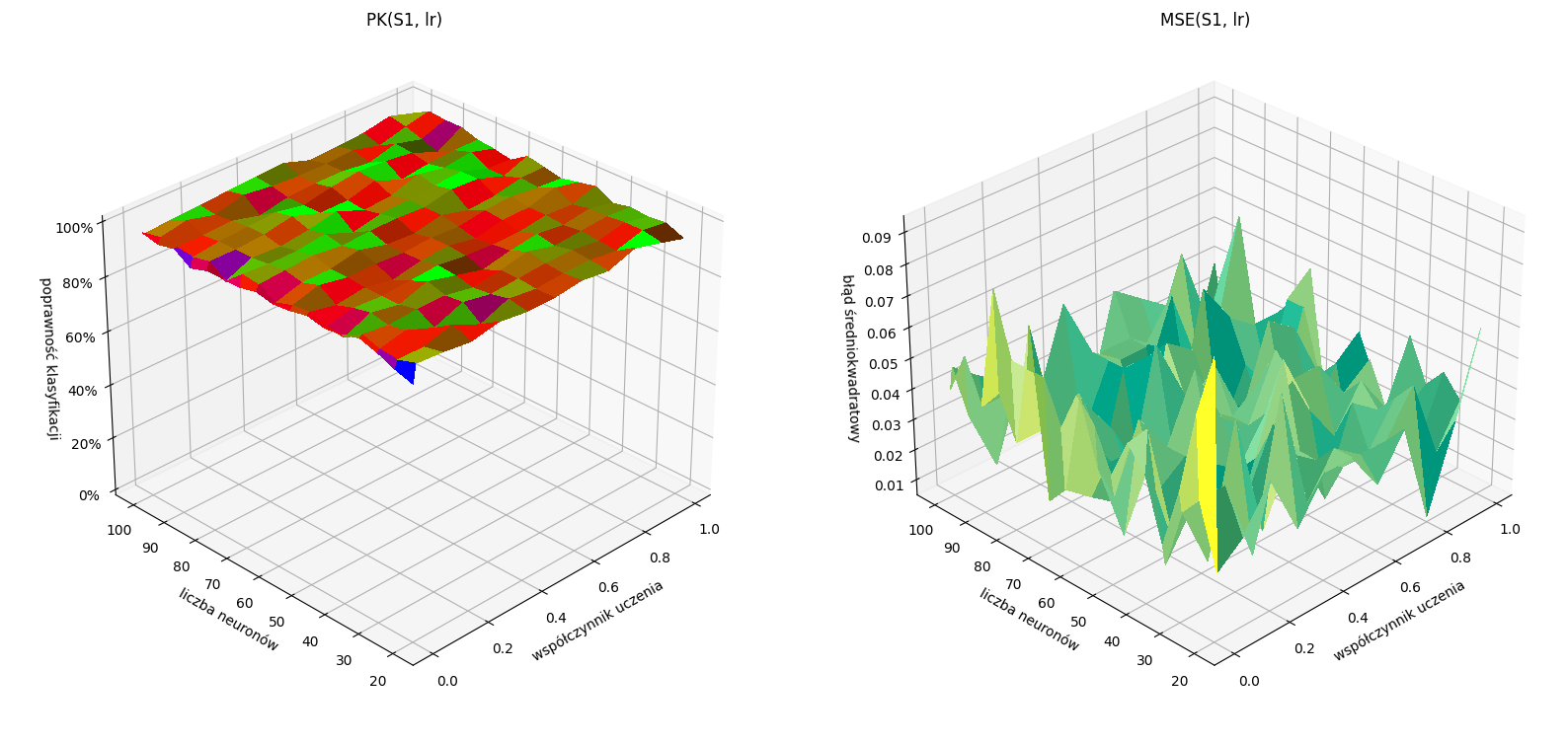
*Rysunek 8. Graficzne przedstawienie wyników eksperymentu pierwszego.*

*Źródło: opracowanie własne*

Poprawność klasyfikacji powyżej 90% została osiągnięta już od poziomu 15 neuronów. W tej fazie testów, nie można było zawęzić przedziału dla współczynnika uczenia, ponieważ dla każdej z obranej wartości wyniki były zadowalające. Przedział poszukiwania najlepszego , zawężono do przedziału od 20 do 100 z krokiem 5, poniżej tego przedziału poprawność klasyfikacji potrafiła spaść poniżej 50% przy wysokich błędach MSE co jest zdecydowanie nieakceptowanym wynikiem.

## Eksperyment drugi

Jako cel drugiego eksperymentu obrano znalezienie takich wartości parametrów współczynnika uczenia oraz ilości neuronów , przy których osiągana jest najwyższa dokładność dopasowania danych przy jednoczesnym jak najkrótszym czasie działania algorytmu. Przedziały w jakich szukane były te wartości zdefiniowano w opracowaniu wyników eksperymentu pierwszego. Poniżej zamieszczono graficzną reprezentację uzyskanych wyników:



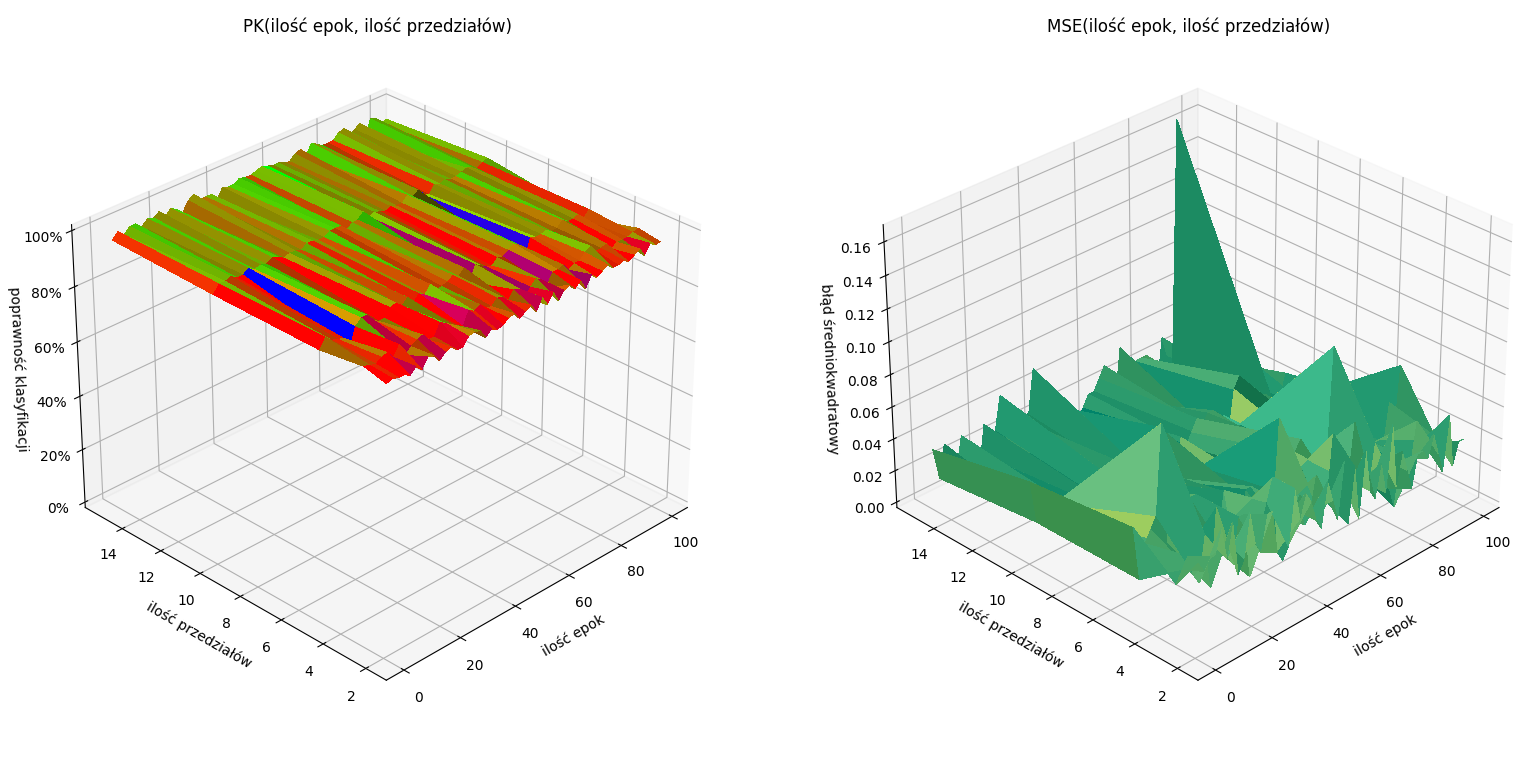
*Rysunek 9. Graficzne przedstawienie wyników eksperymentu drugiego.*

*Źródło: opracowanie własne*

Najlepsze wyniki na poziomie 99,3% wraz z najniższym MSE osiągnięto dla następujących par parametrów {,}: { 20, 0.8 }, { 40, 0.3 },{ 50, 0.9 },{ 65, 0.7 },{ 75, 0.4 },{ 75, 0.7 },{ 80, 0.3 },{ 85, 0.6 }. Najgorszy z wyników osiągnął 90,6% poprawności przy współczynniku uczenia równym 0,001. Z uzyskanych wyników można wywnioskować, że współczynnik uczenia powinien być dość wysoki i zawierać się w przedziale od 0.3 do 0.9 aby uzyskać najlepsze wyniki. Najbardziej optymalnym zestawem parametrów jest ten z najniższą liczbą neuronów, ponieważ czas wykonania algorytmu jest najmniejszy i wynosi jedynie 996ms. To właśnie oszczędność w czasie jest kryterium wyboru przemawiającym na korzyść zestawu = 20 oraz = 0.8 i to właśnie on będzie używany do przeprowadzenia następnego eksperymentu.

## Eksperyment trzeci

Eksperyment trzeci opiewał na znalezieniu jak najniższej ilości epok, przy której algorytm zachowuje swą dokładność, oraz ilości przedziałów do walidacji krzyżowej dającej najlepszą dokładność dopasowania danych. Przedział poszukiwania najniższej liczby epok ustanowiono w granicach od 1 do 100 z krokiem 2. Z racji tego, że dane uczące muszą się dzielić na równe przedziały w sposób zupełny, a zbiór liczy 150 rekordów, ilość przedziałów badano na wartościach zdefiniowanych następująco: 2, 3, 5, 10, 15. Podział zbioru na mniejsze przedziały, liczące poniżej 10 elementów, mógłby pozytywnie wpłynąć na poprawność klasyfikacji w danym przedziale, jednak nie byłaby to wartość prawdziwa a w większości losowa z racji na małą ilość próbek do porównania. Poniżej zamieszczono graficzną reprezentację uzyskanych wyników:



*Rysunek 10. Graficzne przedstawienie wyników eksperymentu trzeciego.*

*Źródło: opracowanie własne*

Podczas trwania eksperymentu udało się znaleźć kilka konfiguracji potrafiących w pełni poprawnie rozpoznać przedstawiane im dane. Najbardziej wydajną opcją jest ustalenie ilości przedziałów na poziomie 15, a ilość epok na 29. Osiągany czas wykonania algorytmu wynosi wówczas 1973ms. Drugim najlepszym zestawieniem jest ilość przedziałów wynosząca 10 natomiast ilość epok na 53. Czas jest zaledwie o 400ms dłuższy względem poprzednika. Jak wspomniano wcześniej duża ilość przedziałów może fikcyjnie zawyżać poprawność klasyfikacji, więc ostatecznie przyjęto za najlepszą konfigurację drugi z przedstawionych wyników.

# Wnioski

Obliczenia przeprowadzone „na kartce” pokrywają się z tymi obliczonymi w programie Octave. Wyniki dla poszczególnych metod plasują się następująco:

* Metoda bisekcji:
* Metoda regula falsi:
* Metoda siecznych:

Można więc stwierdzić, że największą dokładnością wykazuje się metodą siecznych. Cechuje się najmniejszą ilością kroków do osiągnięcia zbieżności, a co za tym idzie, również najmniejszą złożonością czasową. Jednakowoż metoda bisekcji jest najprostszą w obliczaniu metodą rozwiązywania równań nieliniowych.

# Bibliografia

## Literatura

[5] Tadeusiewicz R., Sieci neuronowe. Wydanie II, Akademicka Oficyna Wydawnicza, Warszawa 1993

[6] Kohonen T., Self-Organization and Associative Memory. Second Edition, Speinger Series in Information Sciences

[7] Demuth H., Beale M., Hogan M., Neural Network Toolbox 5 User’s Guide. The MathWorks Inc 2005

[8] Kohonen T., Hynninen J., Kangas K., Laaksonen J., Torkkola K., The Learning Vector Quantization Program Package. Version 3.1, Helsinki University of Technology 1995

[11] Tadeusiewicz R., Szaleniec M., Leksykon sieci neuronowych. Wydanie I, Wydawnictwo Fundacji „Projekt Nauka”, Wrocław 2015

## Źródła internetowe

[1] [Baza danych Iris Data Set](https://archive.ics.uci.edu/ml/datasets/iris)

[2] [Sieci jednokierunkowe wielowarstwowe typu sigmoidalnego](https://riad.pk.edu.pl/~pjarosz/si/%5b2%5d%20Sieci%20jednokierunkowe.pdf)

[3] [Instrukcja 9 do zajęć laboratoryjnych – Sieć jednokierunkowa wielowarstwowa](http://materialy.prz-rzeszow.pl/pracownik/pliki/34/sztuczna-inteligencja-cw9-siec-wielowarstw.pdf)

[4] [Machine Learning for Artists](https://ml4a.github.io/ml4a/neural_networks/)

[9] [Matplotlib: Visualization with Python](https://matplotlib.org/)

[10] [Repozytorium stworzonego algorytmu LVQ](https://github.com/Rav-Naz/LVQ)

[12] [How To Implement Learning Vector Quantization (LVQ) From Scratch With Python](https://machinelearningmastery.com/implement-learning-vector-quantization-scratch-python/)