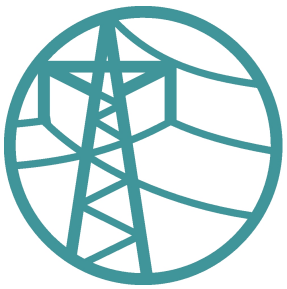


WYDZIAŁ
ELEKTROTECHNIKI
I INFORMATYKI
POLITECHNIKI RZESZOWSKIEJ

**Sztuczna Inteligencja**

**Projekt**

Temat: Zrealizować sieć neuronową uczoną algorytmem wstecznej propagacji błędu z przyśpieszeniem metodą adaptacyjnego współczynnika uczenia (trainbpa) uczącą się rozpoznawania rodzaju wina

**Szymon Kmieć**
2EF-DI, P2

Rzeszów, 2022

# Spis treści

# 1. Opis problemu

Głównym celem projektu było zaprojektowanie oraz implementacja sieci neuronowej służącej do rozpoznawania gatunku wina. Do tego celu użyto własnej implementacji sieci neuronowej w języku Python 3.10.5 uczonej algorytmem wstecznej propagacji błędu. Jako metodę przyśpieszenia uczenia użyto adaptacyjnego współczynnika uczenia. W ramach projektu zbadano wpływ następujących parametrów na szybkość uczenia się sieci:

- S1 - liczba neuronów w pierwszej warstwie sieci
- S2 - liczba neuronów w drugiej warstwie sieci
- lr - współczynnik uczenia sieci
- er - współczynnik maksymalnego dopuszczalnego przyrostu błędu
- $\mathrm{lr}_{\mathrm{dec}}$ - modyfikator współczynnika uczenia w przypadku przekroczeniu maksymalnego dopuszczalnego przyrostu błędu
- $\mathrm{lr}_{\mathrm{inc}}$ - modyfikator współczynnika uczenia w przypadku spadku błędu

Opis problemu oraz wykorzystane w projekcie dane zostały przedstawione na stronie [1]

# 2. Specyfikacja danych

Dane pobrane z wspomnianej powyżej strony zawierały 178 wierszy danych, z których to każdy posiadał 14 atrybutów (w tym atrybut klasowy). Dane są wynikiem analizy chemicznej win uprawianych w tym samym regionie Włoch, ale pochodzących z trzech różnych odmian. Badany zestaw danych nie zawiera niekompletnych rekordów, oraz wartości niepoprawnych. W zbiorze danych pierwszy atrybut określa rodzaj wina - opisany liczbą całkowitą 1, 2, 3, natomiast pozostałe 13 atrybutów to

- (Alcohol) - Procentowa zawartość alkoholu (wartość ciągła)
- (Malic acid) - Procentowa zawartość kwasu jabłkowego (wartość ciągła)
- (Ash) - Zawartość popiołu (wartość ciągła)
- (Alkalinity of ash) - Zasadowość popiołu (wartość ciągła)
- (Mg) - Zawartość magnezu (wartość ciągła)
- (Total phenols) - Zawartość fenoli (wartość ciągła)
- (Flavonoids) - Flawonoidy (wartość ciągła)
- (Non Flavonoid phenols) - Fenole nieflawonoidowe (wartość ciągła)
- (Proanthocyanins) - Proantocyjanidyny (wartość ciągła)
- (Colour intensity) - Intensywność barwy (wartość ciągła)
- (Hue) - Odcień (wartość ciągła)
- (OD280/0D315) (wartość ciągła)
- (Proline) - Prolina (wartość dyskretna)

## 2.1. Normalizacja danych

W celu usprawnienia procesu uczenia dane wejściowe zostały poddane normalizacji metodą *min - max* określoną wzorem [3]:

$$f(x) = \frac{x - \min(x)}{\max(x) - \min(x)} \tag{2.1}$$

Dzięki takiej normalizacji, możliwe jest dynamiczne normalizowanie danych wejściowych do zadanego zakresu. W przypadku tego projektu został zastosowany przedział normalizacyjny $[0,1]$.

```
1     for column in norm_data.columns:
2         if column !=0:
3             norm_data[column] = (norm_data[column] - norm_data[
    column].min()) / (norm_data[column].max() - norm_data[column
    ].min())
```

Listing 1: Algorytm normalizacji

Numery klas, stanowiące dane wyjściowe, zostały natomiast przekształcone przy pomocy kodowania 1 z N do postaci trójelementowego wektora, zawierającego wartość 1 na pozycji odpowiadającej danej klasie oraz wartości 0 na pozostałych pozycjach.

# 3. Zagadnienia teoretyczne

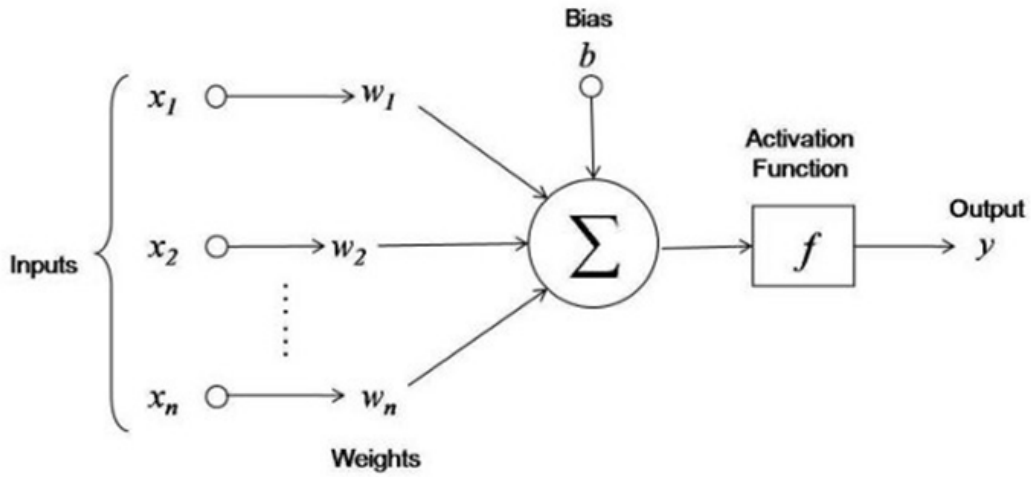
## 3.1. Model sztucznego neuronu

Sieć neuronową tworzą pojedyncze neurony, które ułożone są w poszczególnych warstwach sieci. Neuron jest jednostką przetwarzającą informacje otrzymane na wejciu, oraz zwraca wynik przetwarzania w postaci wartości wyjściowej.

Początkowo neuron miał co najmniej jedno wejście binarne i tylko jedno binarne wyjście. Wyjście było aktywowane, gdy osiągnięta została określona liczba wejść. W 1957 roku uczony Frank Rosenblatt zmodyfikował prosty sztuczny neuron binarny, tworząc w ten sposób perceptron, czyli jedną z najprostszych sieci neuronowych. Posiada on następującą charakterystykę [6]:

- Na wejściu i wyjściu mogą być dowolne liczby, nie tylko wartości binarne
- Połączenia węzłów mają określoną wagę
- Wartość wyjściowa węzła składa się z dwóch części: sumy wartości z warstw poprzednich pomnożonej przez wagi oraz nałożonej na tą sumę funkcji aktywacji.

Model neuronu został przedstawiony na rysunku 3.1.




Rysunek 3.1: Model neuronu

Ważona suma wejść wraz z przesunięciem nazywana jest łącznym pobudzeniem neuronu i określana wzorem:

$$z = \sum_{i=1}^{n} (x_i \cdot w_i) + b \tag{3.2}$$

Ogólny wzór na wartość wyjścia neuronu przedstawiono równaniem 3.3

$$y = f\left(\sum_{i=1}^{n}(x_i \cdot w_i) + b\right) = f(z) \tag{3.3}$$

gdzie:

- $y$ - wyjście neuronu
- $f$ - funkcja aktywacji
- $n$ - ilość wejść
- $x$ - wektor wejciowy
- $w$ - wektor wag
- $b$ - bias

Typy neuronów dzieli się ze względu na ich fukncję aktywacji. Najczęściej stosowanymi funkcjami aktywacji neuronu są funkcje sigmoidalne (3.4 oraz 3.5) oraz liniowe (3.6).

Funkcja sigmoidalna unipolarna ma postać:

$$f(x) = \frac{1}{1 + e^{-\beta x}} \tag{3.4}$$

Funkcja sigmoidalna bipolarna:

$$f(x) = \frac{2}{1 + e^{-\beta x}} - 1 \tag{3.5}$$

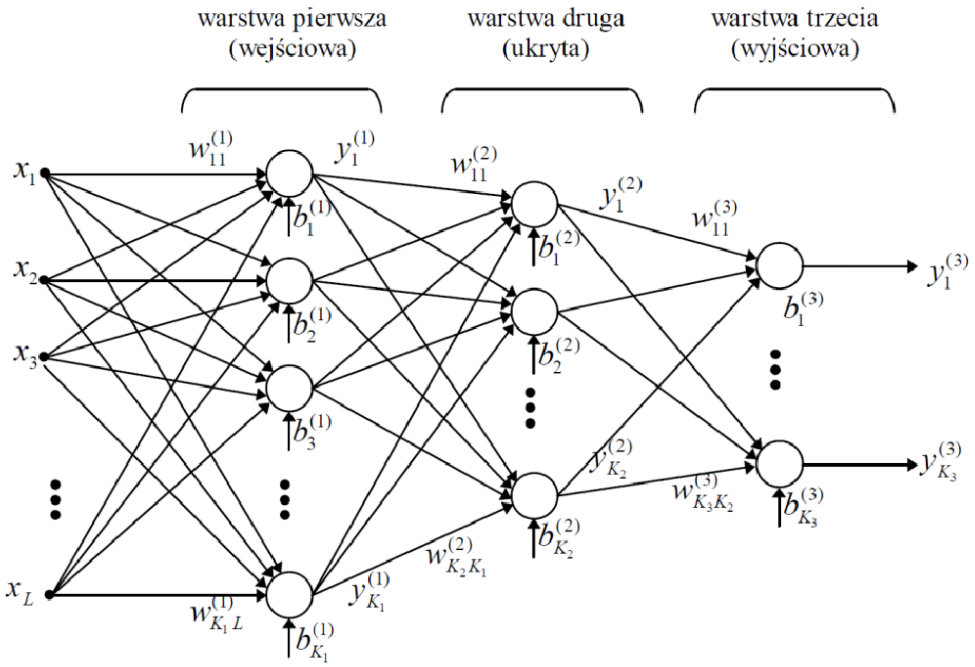
Funkcja liniowa:

$$f(x) = a \cdot x + b \tag{3.6}$$

gdzie parametr $\beta$ najczęściej jest z przedziału $[0, 1]$.

## 3.2. Sieć jednokierunkowa wielowarstwowa

Sztuczną sieć neuronowa uzyskuje się łącząc ze sobą warstwy neuronów. Neurony nie łączą się ze sobą w obrębie tej samej warstwy lub z pominięciem warstw między nimi. Przykładowy model sieci wielowarstwowej pokazano na rysunku 3.2.




Rysunek 3.2: Sieć jednokierunkowa wielowarstwowa [Źródło: [5]]

Sieć taka ma zwykle strukturę obejmującą:

- warstwę wejściową
- jedną lub więcej warstw ukrytych (złożonych z neuronów sigmoidalnych)
- warstwę wyjściową (złożoną z neuronów sigmoidalnych lub liniowych)

Charakterystyczną cechą dla warstwy wejściowej jest to, że nie bierze ona udziału w procesie uczenia, jej zadaniem zadaniem jest przekazanie wektora wejściowego sieci do warstw ukrytych, lub bezpośrednio do warstwy wyjściowej (w przypadku modelu sieci jednowarstwowej). Ilość warstw w sieci oraz ilość neuronów w warstwie są ściśle uzależnione od specyfiki problemu przed jakim zosataje postawiona sieć neuronowa.

Każda warstwa sieci posiada:

- Macierz wag neuronów - **w**
- Wektor przesunięć - **b**
- Wektor sygnałów wyjściowych - **y**

Działanie poszczególnych warstw sieci opisane jest wzorami:

$$\begin{aligned} y^{(1)} &= f^{(1)}(w^{(1)}x + b^{(1)}) \\ y^{(2)} &= f^{(2)}(w^{(2)}y^{(1)} + b^{(2)}) \\ y^{(3)} &= f^{(3)}(w^{(3)}y^{(2)} + b^{(3)}) \end{aligned} \tag{3.7}$$

Zatem działanie całej trójwarstwowej sieci można zapisać jako:

$$y^{(3)} = f^{(3)}\left(w^{(3)}f^{(2)}\left(w^{(2)}f^{(1)}\left(w^{(1)}x + b^{(1)}\right) + b^{(2)}\right) + b^{(3)}\right) \tag{3.8}$$

## 3.3. Algorytm wstecznej propagacji błędu

Algorytm wstecznej propagacji błędu, nazywany również algorytmem największego spadku gradientu jest algorytmem dominującym wśród metod uczenia sieci jednokierunkowych. Działanie algorytmu sprowadza się do wyznaczenia gradientu funkcji celu dla każdego z neuronów sieci.

Aby możliwe było zastosowanie algorytmu wstecznej propagacji błędu, wymagane jest, aby funkcje aktywacji neuronów były różniczkowalne. Pozwala to na wyznaczenie pochodnej błędu po danych wyjściowych. Dla każdej pary $(x, \hat{y})$ sieć popełnia błąd, który można zdefiniować następująco:

$$e = y - \hat{y} \tag{3.9}$$

Celem uczenia sieci jest zminimalizowanie sumarycznego błędu kwadratowego, wyrażonego jako suma kwadratów błędów dla K neuronów w warstwie wyjściowej.

$$E = \frac{1}{2}\sum_{j=1}^{K} e_j^2 \tag{3.10}$$

W przypadku sieci z rysunku 3.2 funkcja 3.10 po uwzględnieniu zależności 3.7 przyjmie postać:

$$
\begin{aligned}
E =& \frac{1}{2}\sum_{i_3=1}^{K_3} e_{i_3}^2 = \frac{1}{2}\sum_{i_3=1}^{K_3}\left(y_{i_3}^{(3)} - \hat{y}_{i_3}\right)^2 = \\
=& \frac{1}{2}\sum_{i_3=1}^{K_3}\left(f^{(3)}\left(\sum_{i_2=1}^{K_2} w_{i_3 j_2}^{(3)} y_{i_2} + b_{i_3}^{(3)}\right) - \hat{y}_{i_3}\right)^2 = \\
=& \frac{1}{2}\sum_{i_3=1}^{K_3}\left(f^{(3)}\left(\sum_{i_2=1}^{K_2} w_{i_3 j_2}^{(3)} f^{(2)}\left(\sum_{i_1=1}^{K_1} w_{i_2 j_1}^{(2)} y_{i_1} + b_{i_2}^{(2)}\right) + b_{i_3}^{(3)}\right) - \hat{y}_{i_3}\right)^2 = \\
=& \frac{1}{2}\sum_{i_3=1}^{K_3}\left(f^{(3)}\left(\sum_{i_2=1}^{K_2} w_{i_3 j_2}^{(3)} f^{(2)}\left(\sum_{i_1=1}^{K_1} w_{i_2 j_1}^{(2)} f^{(1)}\left(\sum_{j=1}^{L} w_{i_1 j}^{(1)} x_j + b_{i_1}^{(1)}\right) + b_{i_2}^{(2)}\right) + b_{i_3}^{(3)}\right) - \hat{y}_{i_3}\right)^2
\end{aligned}
\tag{3.11}
$$

Zminimalizowanie błędu 3.10 osiąga się poprzez zmianę biasów neuronów oraz wag ich połączeń. Kolejna wartość wagi połączenia jest wyznaczana na podstawie pochodnej cząstkowej błędu po wartości tejże wagi w obecnym cyklu nauczania. W podobny sposób wyznaczana jest również nowa wartość biasu dla każdego z neuronów. Otrzymaną pochodną mnoży się przez współczynnik uczenia $\eta$, który również może zmieniać się podczas procesu uczenia sieci. Zmiany poszczególnych wag obliczane są ze wzoru:

$$\Delta w_{ij} = -\eta \frac{\partial E}{\partial w_{ij}} \tag{3.12}$$

Zatem wagę dla $k+1$ kroku otrzymać można w następujący sposób:

$$w_{ij}(k+1) = w_{ij}(k) - \eta \frac{\partial E}{\partial w_{ij}(k)} \tag{3.13}$$

Obliczanie wag neuronów rozpoczyna się od warstwy wyjściowej:

$$\frac{\partial E}{\partial w_{i_3 i_2}^{(3)}} = \frac{\partial E}{\partial f^{(3)}} \frac{\partial f^{(3)}\left(z_{i_3}^{(3)}\right)}{\partial z_{i_3}^{(3)}} \frac{\partial z_{i_3}^{(3)}}{\partial w_{i_3 i_2}^{(3)}} = \left(y_{i_3} - \hat{y}_{i_3}\right) \frac{\partial f^{(3)}\left(z_{i_3}^{(3)}\right)}{\partial z_{i_3}^{(3)}} y_{i_2}^{(2)} \tag{3.14}$$

Podobnie można obliczyć elementy gradientu względem wag warstwy ukrytej:

$$\begin{aligned}\frac{\partial E}{\partial w_{i_2 i_1}^{(2)}} &= \frac{\partial E}{\partial f^{(3)}} \frac{\partial f^{(3)}\left(z_{i_3}^{(3)}\right)}{\partial z_{i_3}^{(3)}} \frac{\partial z_{i_3}^{(3)}}{\partial f^{(2)}} \frac{\partial f^{(2)}\left(z_{i_2}^{(2)}\right)}{\partial z_{i_2}^{(2)}} \frac{\partial z_{i_2}^{(2)}}{\partial w_{i_2 i_1}^{(2)}} = \\ &= \sum_{i_3=1}^{K_3} \left(y_{i_3} - \hat{y}_{i_3}\right) \frac{\partial f^{(3)}\left(z_{i_3}^{(3)}\right)}{\partial z_{i_3}^{(3)}} w_{i_3 i_2}^{(3)} \frac{\partial f^{(2)}\left(z_{i_2}^{(2)}\right)}{\partial z_{i_2}^{(2)}} y_{i_1}^{(1)}\end{aligned} \tag{3.15}$$

Oraz dla warstwy wejściowej:

$$\begin{aligned}\frac{\partial E}{\partial w_{i_1 j}^{(1)}} &= \frac{\partial E}{\partial f^{(3)}} \frac{\partial f^{(3)}\left(z_{i_3}^{(3)}\right)}{\partial z_{i_3}^{(3)}} \frac{\partial z_{i_3}^{(3)}}{\partial f^{(2)}} \frac{\partial f^{(2)}\left(z_{i_2}^{(2)}\right)}{\partial z_{i_2}^{(2)}} \frac{\partial z_{i_2}^{(2)}}{\partial f^{(1)}} \frac{\partial f^{(1)}\left(z_{i_1}^{(1)}\right)}{\partial z_{i_1}^{(1)}} \frac{\partial z_{i_1}^{(1)}}{\partial w_{i_1 j}^{(1)}} = \\ &= \sum_{i_3=1}^{K_3} \left(y_{i_3} - \hat{y}_{i_3}\right) \frac{\partial f^{(3)}\left(z_{i_3}^{(3)}\right)}{\partial z_{i_3}^{(3)}} \sum_{i_2=1}^{K_2} w_{i_3 i_2}^{(3)} \frac{\partial f^{(2)}\left(z_{i_2}^{(2)}\right)}{\partial z_{i_2}^{(2)}} w_{i_2 i_1}^{(2)} \frac{\partial f^{(1)}\left(z_{i_1}^{(1)}\right)}{\partial z_{i_1}^{(1)}} x_j\end{aligned} \tag{3.16}$$

## 3.4. Adaptacyjny współczynik uczenia

Algorytm wstecznej propagacji błędu jest dość czasochłonny. W celu przyśpieszenia procesu uczenia sieci korzysta się z metod pozwalających na jego przyśpieszenie. Jedną z nich jest metoda adaptacyjnej korekty współczynnika uczenia. Decyzję o zmianie podejmuje się na podstawie porównania błędu kwadratowego z jego wartością uzyskaną w poprzednim cyklu nauczania. Kolejną wartość $\eta$ otrzymuje się na podstawie następującej zależności:

$$\eta(t+1) = \begin{cases} \eta(t)\xi_d & gdy\ SSE(t) > er \cdot SSE(t-1) \\ \eta(t)\xi_i & gdy\ SSE(t) < SSE(t-1) \\ \eta(t) & gdy\ SSE(t-1) \leq SSE(t) \leq er \cdot SSE(t-1) \end{cases} \tag{3.17}$$

gdzie:

- $er$ - dopuszczalna krotność przyrostu błędu
- $\xi_d$ - współczynnik zmniejszania wartości współczynnika uczenia
- $\xi_i$ - współczynnik zwiększania wartości współczynnika uczenia

# 4. Implementacja sieci neuronowej

Na potrzeby realizacji projektu zaimplementowano w języku Python program pozwalający na uczenie sieci neuronowej o dowolnej liczbie warstw algorytmem wstecznej propagacji błędu, wraz z przyspieszeniem metodą adaptacyjnego współczynnika uczenia na podstawie książki Michaela Nielsena [2].

Do implementacji wykorzystano język programowania Python 3.10.5, który jest językiem z natury wolnym, dlatego do obliczeń wykorzystano moduł *numpy*. Moduł ten napisany jest w C, dzięki czemu obliczenia na macierzach będą wykonywane o wiele szybciej niż w zwykłym pythonie. Do kroswalidacji wykorzystano moduł *sklearn*.

Program oferuje możliwość wyboru techniki aktualizacji wag:

- Wsadowa - parametry neuronów aktualizowane są dopiero po wczytaniu do sieci całego zestawu danych uczących
- Mini-batch - Zbiór danych podzielony jest na podzbiory, aktualizacja parametrów sieci następuje po zaprezentowaniu całego podzbioru

```
import random
import numpy as np

class Network(object):

    def __init__(self, sizes):
        np.random.seed(0x2f6eab9)
        self.num_layers = len(sizes)
        self.sizes = sizes
        self.biases = [np.random.randn(y, 1) for y in sizes[1:]]
        self.weights = [np.random.randn(y, x)
                        for x, y in zip(sizes[:-1], sizes[1:])]
        self.max_perf_inc = 1.04

    def __feedforward(self, a):
        for b, w in zip(self.biases, self.weights):
            a = self.__sigmoid(np.dot(w, a)+b)
        return a

    def __sse(self,_test_data):
        error=[pow(np.linalg.norm(self.__feedforward(x)-y),2)
    for (x,y) in _test_data]
        return 0.5*sum(error)

    def train(self, training_data, test_data,epochs=1000,eta
    =0.1,error_goal=0.25,inc=1.05,dec=0.7,mini_batch_size=0,
    err_inc=1.04):
        self.max_perf_inc=err_inc
        if mini_batch_size==0: mini_batch_size=len(training_data
    )
```

```
        n_test = len(test_data)
        n = len(training_data)
        for j in range(epochs):
            random.shuffle(training_data)
            mini_batches = [training_data[k:k+mini_batch_size]
                    for k in range(0, n, mini_batch_size)]
            old_error=self.__sse(test_data)
            backup_weights = self.weights.copy()
            backup_biases = self.biases.copy()
            for mini_batch in mini_batches:
                self.__update_mini_batch(mini_batch, eta)
            new_error=self.__sse(test_data)
            if new_error < error_goal:
                test=self.__evaluate(test_data)
                test2=test/n_test*100
                print("Epoch {0}: {1:.2f}\%".format(j+1, test2))
                return [j+1, test2]
            elif new_error < old_error:
                eta *= inc
            elif new_error > old_error * self.max_perf_inc:
                self.weights = backup_weights
                self.biases = backup_biases
                eta *= dec
            if j==epochs-1:
                test=self.__evaluate(test_data)
                test2=test/n_test*100
                print("Epoch {0}: {1:.2f}\%".format(j+1, test2))
                return [j+1, test2]
    def __update_mini_batch(self, mini_batch, eta):
        nabla_b = [np.zeros(b.shape) for b in self.biases]
        nabla_w = [np.zeros(w.shape) for w in self.weights]
        for x, y in mini_batch:
            delta_nabla_b, delta_nabla_w = self.__backprop(x, y)
            nabla_b = [nb+dnb for nb, dnb in zip(nabla_b,
    delta_nabla_b)]
            nabla_w = [nw+dnw for nw, dnw in zip(nabla_w,
    delta_nabla_w)]
        self.weights = [w-(eta/2)*nw
                        for w, nw in zip(self.weights, nabla_w)]
        self.biases = [b-(eta/2)*nb
                       for b, nb in zip(self.biases, nabla_b)]

    def __backprop(self, x, y):
        nabla_b = [np.zeros(b.shape) for b in self.biases]
        nabla_w = [np.zeros(w.shape) for w in self.weights]
        # feedforward
        activation = x
        activations = [x] # list to store all the activations,
    layer by layer
        zs = [] # list to store all the z vectors, layer by
    layer
        for b, w in zip(self.biases, self.weights):
            z = np.dot(w, activation)+b
            zs.append(z)
            activation = self.__sigmoid(z)
            activations.append(activation)
```

```
79          # backward pass
80          delta = self.__cost_derivative(activations[-1], y) *
    self.__sigmoid_prime(zs[-1])
81          nabla_b[-1] = delta
82          nabla_w[-1] = np.dot(delta, activations[-2].transpose())
83          for l in range(2, self.num_layers):
84              z = zs[-l]
85              sp = self.__sigmoid_prime(z)
86              delta = np.dot(self.weights[-l+1].transpose(), delta
    ) * sp
87              nabla_b[-l] = delta
88              nabla_w[-l] = np.dot(delta, activations[-l-1].
    transpose())
89          return (nabla_b, nabla_w)
90
91      def __evaluate(self, test_data):
92          test_results = [(np.argmax(self.__feedforward(x)), np.
    argmax(y)) for (x, y) in test_data]
93          return sum(int(x == y) for (x, y) in test_results)
94
95      def __cost_derivative(self, output_activations, y):
96          return (output_activations-y)
97
98      def __sigmoid(self,z):
99          return 1.0/(1.0+np.exp(-z))
100
101     def __sigmoid_prime(self,z):
102         return self.__sigmoid(z)*(1-self.__sigmoid(z))
```

Listing 2: Implementacja sieci

# 5. Eksperymenty

## 5.1. Eksperyment 1 - badanie wpływu S1 i S2 na szybkość uczenia sieci

Celem eksperymentu było znalezienie takiej kombinacji parametrów S1 oraz S2, dla której sieć osiągnie najlepszą poprawność klasyfikacji. Listing 3 przedstawia kod odpowiedzialny za tenże eksperyment. Każda instancja sieci neuronowej została zainicjalizowana takimi samymi wagami i biasami. Do przeprowadzenia eksperymentu przyjęto następujące wartości domyślne:

| | |
|---|---|
| S1 (neurony warstwy I) | $[1, 25]$ |
| S2 (neurony warstwy II) | $[1, 25]$ |
| Learning rate | 0.1 |
| Learning rate accelerate | 1.05 |
| Learning rate decelerate | 0.7 |
| Error ratio | 1.04 |
| Stosunek wielkości zbiorów | $80\% : 20\%$ |
| Oczekiwana wartość funkcji celu | SSE $<0.25$ |

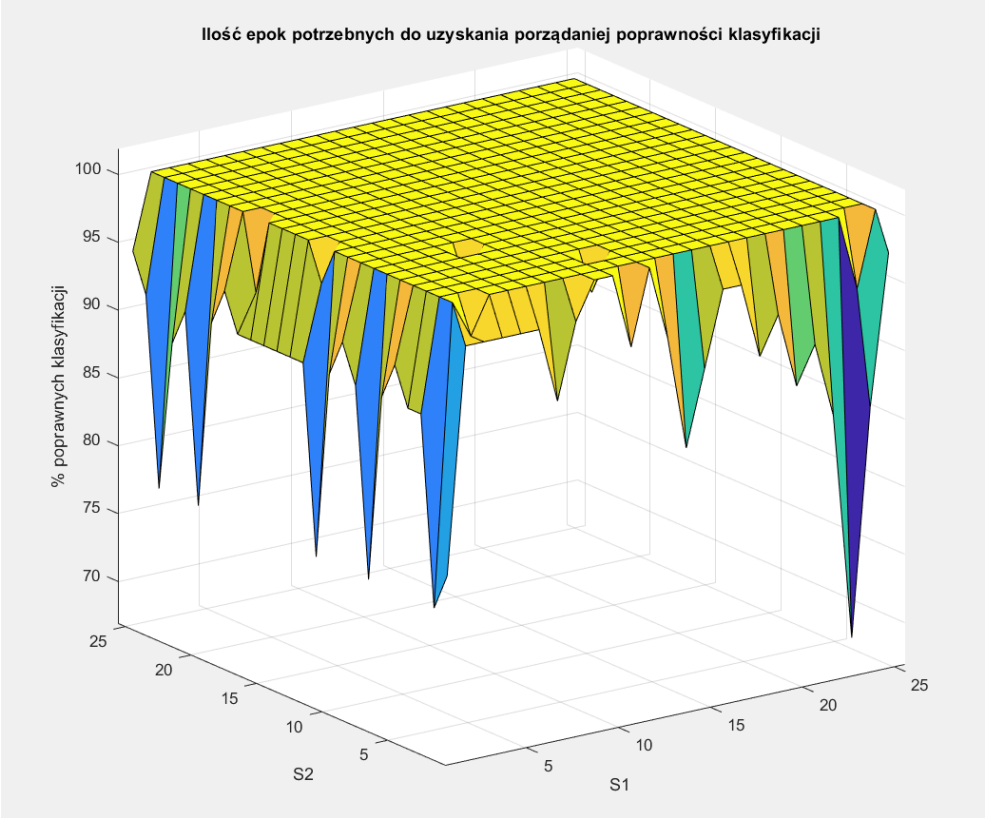
Tablica 5.1: Domyślne wartości parametrów

```
import network
from data import loadData
import pandas as pd
import numpy as np

trainData, testData = loadData()
s1_vec = np.arange(1,25.01,1,dtype=int)
s2_vec = np.arange(1,25.01,1,dtype=int)
results = []
for s1 in s1_vec:
  for s2 in s2_vec:
    result = [s1,s2]
    net = network.Network([13, s1, s2, 3])
    print('S1: {0}, S2: {1}'.format(s1,s2))
    result.extend(net.train(trainData,testData))
    results.append(result)

results=pd.DataFrame(results)
results.to_csv('S1_S2_1.csv',index=None,header=None)
```

Listing 3: Algorytm realizujący eksperyment 1

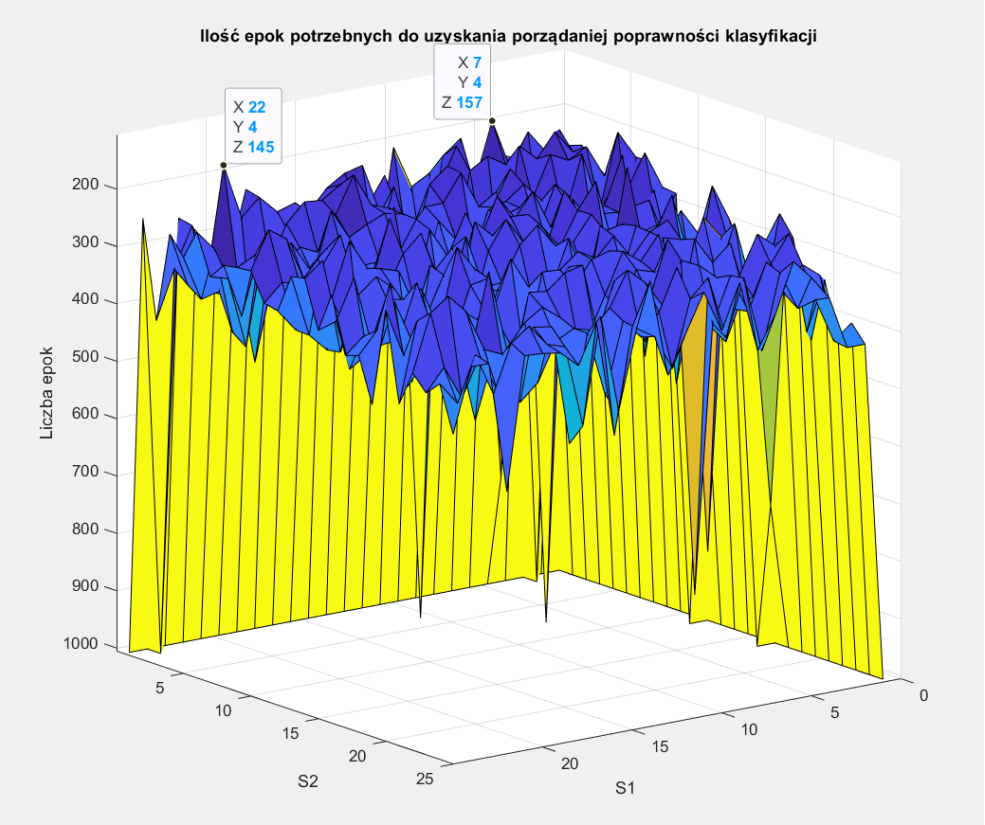
### 5.1.1. Badania dla 1000 epok

Celem pierwszego eksperymentu było znalezienie odpowiedniej liczby neuronów w warstwie pierwszej i drugiej. Przebadano liczby neuronów w obu warstwach z zakresu od 1 do 25 z krokiem co jeden neuron. Liczba epok została ustawiona na 1000. Pozostałe parametry ustawiono zgodnie z tabelą 5.1 Przebadane zostały liczby neuronów w obu warstwach z przedziału od 1 do 20 z krokiem co 1 neuron. Liczba epok wynosiła 1000.




Rysunek 5.3: Wykres wypływu S1, S2 na poprawność klasyfikacji (maks. 1000 epok)

Na wykresie 5.1.1 można zauważyć, że większość kombinacji liczby neuronów S1 i S2 proces uczenia zakończył się przed osiągnięciem maksymalnej ilości epok, co świadczy o tym, że przedstawiony zbiór danych jest niewymagający. W celu poprawnej analizy danych wykreślono powierzchnię funkcji minimalnej ilości epok potrzebnych do osiągnięcia zakładanego błędu uczenia SSE.




Rysunek 5.4: Wykres wypływu S1, S2 na ilość epok potrzebnych do osiągnięcia zakładanego SSE

W wyniku tego eksperymentu okazało się, że dla większości kombinacji S1 i S2 sieć osiąga 100% dokładności klasyfikacji. Skuteczność tą osiąga najszybciej dla kombinacji $< 22, 4 >$, którą sieć osiągnęła po 145 epokach. Drugą parą jest kombinacja $< 7, 4 >$, dla której zadaną skuteczność osiągnięto po 157 epokach.

## 5.2. Eksperyment 2 - badanie parametrów $lr_{inc}$ oraz $lr_{dec}$

Celem drugiego eksperymentu było wyznaczenie optymalnych wartości parametrów $lr_{inc}$ oraz $lr_{dec}$ odpowiedzialnych za modyfikację współczynnika uczenia na podstawie bieżącego błędu SSE. Na potrzeby eksperymentu przyjęto parametry uczenia określone w tabeli 5.1, zmianie poddane zostały parametry:

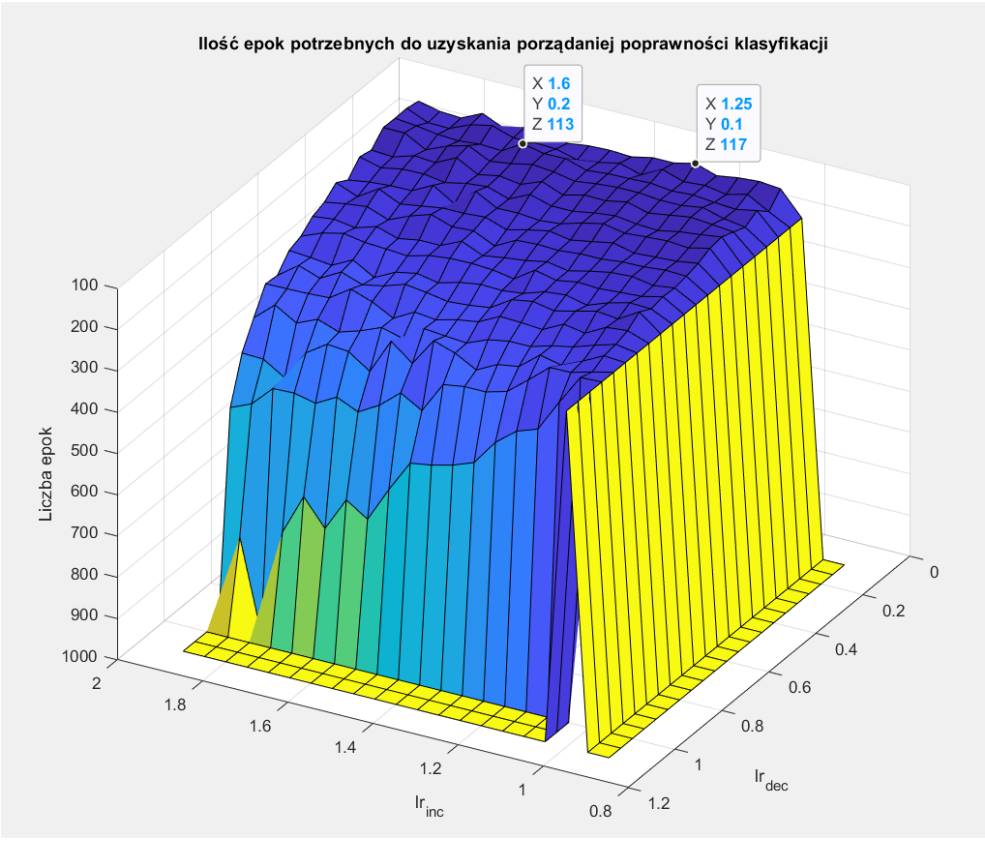
- learning rate accelerate - w przedziale $< 0.9; 1.9 >$, ze skokiem co 0.05
- learning rate decelerate - w przedziale $< 0.1; 1.1 >$, ze skokiem co 0.05

```
1 import sys
2 import network
3 from data import loadData
```

```
import pandas as pd
import numpy as np
trainData, testData = loadData()
lr_inc_vec=np.arange(0.9,1.91,0.05)
lr_dec_vec=np.arange(0.1,1.11,0.05)
results = []
name=sys.argv[3]
name='inc_dec__'+name+'.csv'
for lr_inc in lr_inc_vec:
  for lr_dec in lr_dec_vec:
    result = [lr_inc,lr_dec]
    net = network.Network([13, int(sys.argv[1]), int(sys.argv
   [2]), 3])
    result.extend(net.train(trainData,testData,inc=lr_inc,dec=
   lr_dec))
    results.append(result)

results=pd.DataFrame(results)
results.to_csv(name,index=None,header=None)
```

Listing 4: Algorytm realizujący eksperyment 2

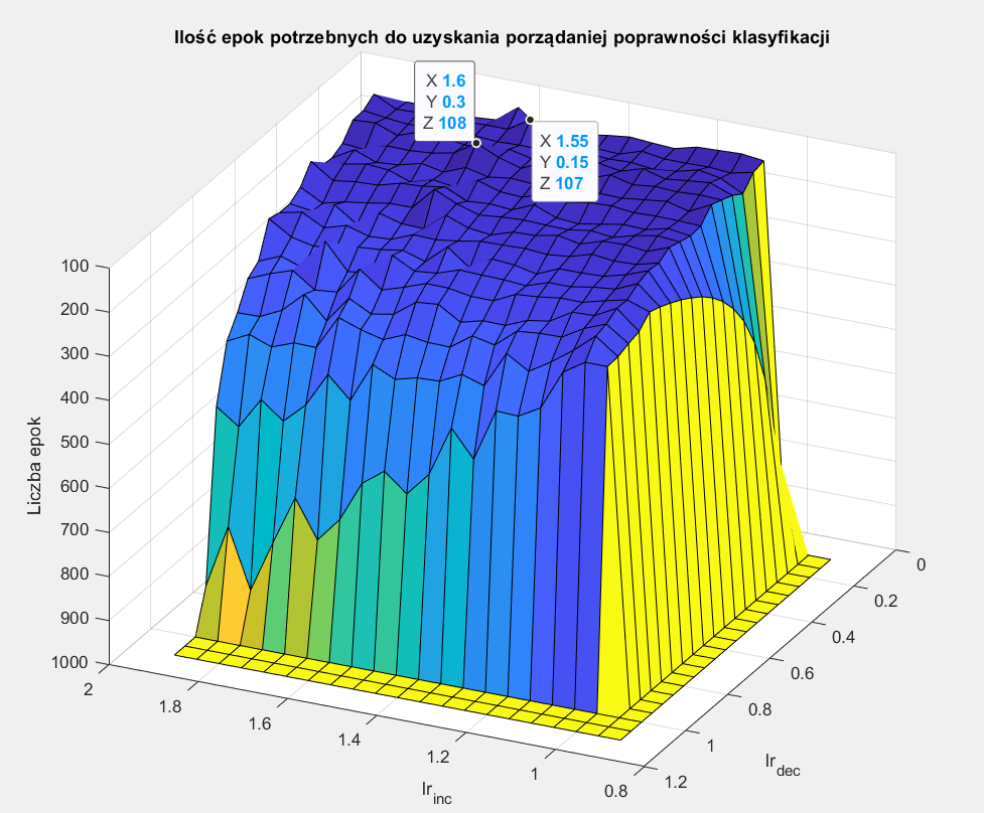
### 5.2.1. Badania dla $S1 = 22$, $S2 = 4$




Rysunek 5.5: Zależność poprawności klasyfikacji od $lr_{inc}$ oraz $lr_{dec}$ dla $S1 = 22$, $S2 = 4$

Rysunek 5.5 przedstawia powierzchnię funkcji ilości epok potrzebnej do osiągnięcia zamierzonego błędu w zależności od wartości parametru learning rate accelerate oraz learning rate decelerate. Dla S1= 22, S2 = 4 największą poprawność uczenia osiągnięto dla wartości $lr_{inc} = 1.6$, oraz $lr_{dec} = 0.2$. Natomiast najgorszy wynik uzyskano gdy $lr_{inc}$ wynosił 0.9 lub $lr_{dec}$ był większy niż 1.

### 5.2.2. Badania dla $S1 = 7$, $S2 = 4$




Rysunek 5.6: Zależność poprawności klasyfikacji od $lr_{inc}$ oraz $lr_{dec}$ dla $S1 = 7$, $S2 = 4$

Rysunek 5.6 przedstawia powierzchnię funkcji ilości epok potrzebnej do osiągnięcia zamierzonego błędu w zależności od wartości parametru learning rate accelerate oraz learning rate decelerate. Dla S1= 7, S2 = 4 największą poprawność uczenia osiągnięto dla wartości $lr_{inc} = 1.55$, oraz $lr_{dec} = 0.15$. Natomiast najgorszy wynik uzyskano gdy $lr_{inc}$ wynosił 0.9 lub $lr_{dec}$ był większy niż 1.

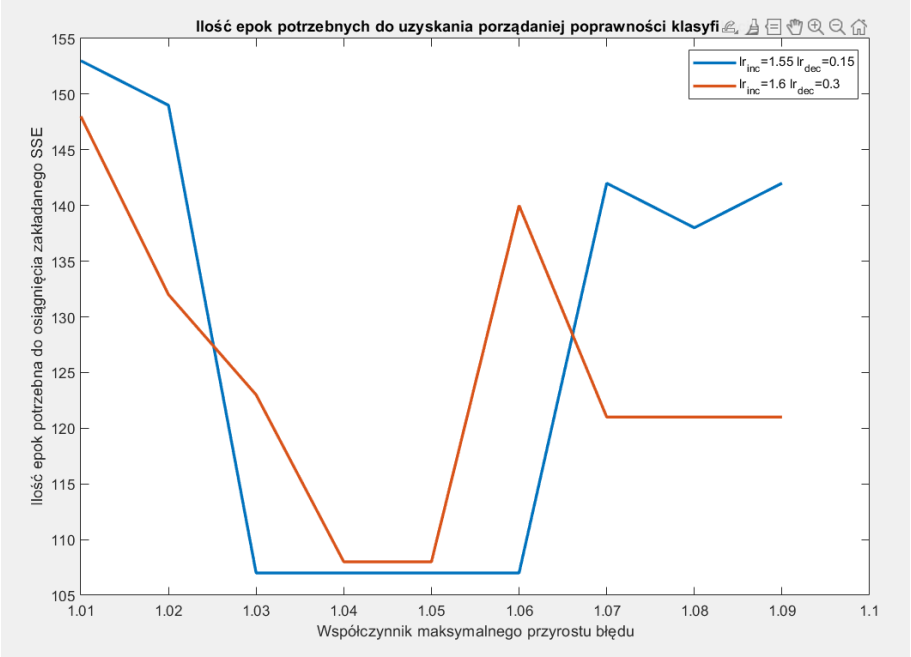
Porównując przedstawione wyżej wykresy i wartości, można wywnioskować, że sieć w parametrach S1= 7, S2 = 4 jest lepiej uwarunkowana dla badanego zestawu danych.

## 5.3. Eksperyment 3 - badanie parametru er

Celem trzeciego eksperymentu było zbadanie wpływu współczynnika maksymalnego przyrostu błędu *er* na ilość potrzebnych epok do osiągnięcia zakładanej wartości błędu SSE. Domyślnie parametr ten wynosił 1.04. Na potrzeby eksperymentu zmieniano wartość *er* w zakresie [1.01;1.09]. Badania przeprowadzono dla maksymalnej liczby 1000 epok, $lr_{inc} = 1.55$, $lr_{dec} = 0.15$ oraz $lr_{inc} = 1.6$, $lr_{dec} = 0.3$ dla parametrów S1 = 7, S2 = 4. Listing 5 przedstawia kod, którego użyto do eksperymentu.

```
import sys
import network
from data import loadData
import pandas as pd
import numpy as np
trainData, testData = loadData()
err_inc_vec=np.arange(1.01,1.091,0.01)
results = []
name=sys.argv[3]
name='err_inc'+name+'.csv'
for err in err_inc_vec:
  result = [err]
  net = network.Network([13, 7, 4, 3])
  result.extend(net.train(trainData,testData,inc=float(sys.argv
   [1]),dec=float(sys.argv[2]),err_inc=err))
  results.append(result)

results=pd.DataFrame(results)
results.to_csv(name,index=None,header=None)
```

Listing 5: Algorytm realizujący eksperyment 3




Rysunek 5.7: Wykres wpływu parametru er na poprawność klasyfikacji

Rysunek 5.7 obrazuje wyniki wykonanego badania. Na jego podstawie można ocenić, że dla pierwszej kombinacji optymalną wartością parametru *er* jest wartość z przedziału $< 1.03; 1.06 >$, natomiast dla drugiej pary jest to przedział $< 1.04; 1.05 >$.

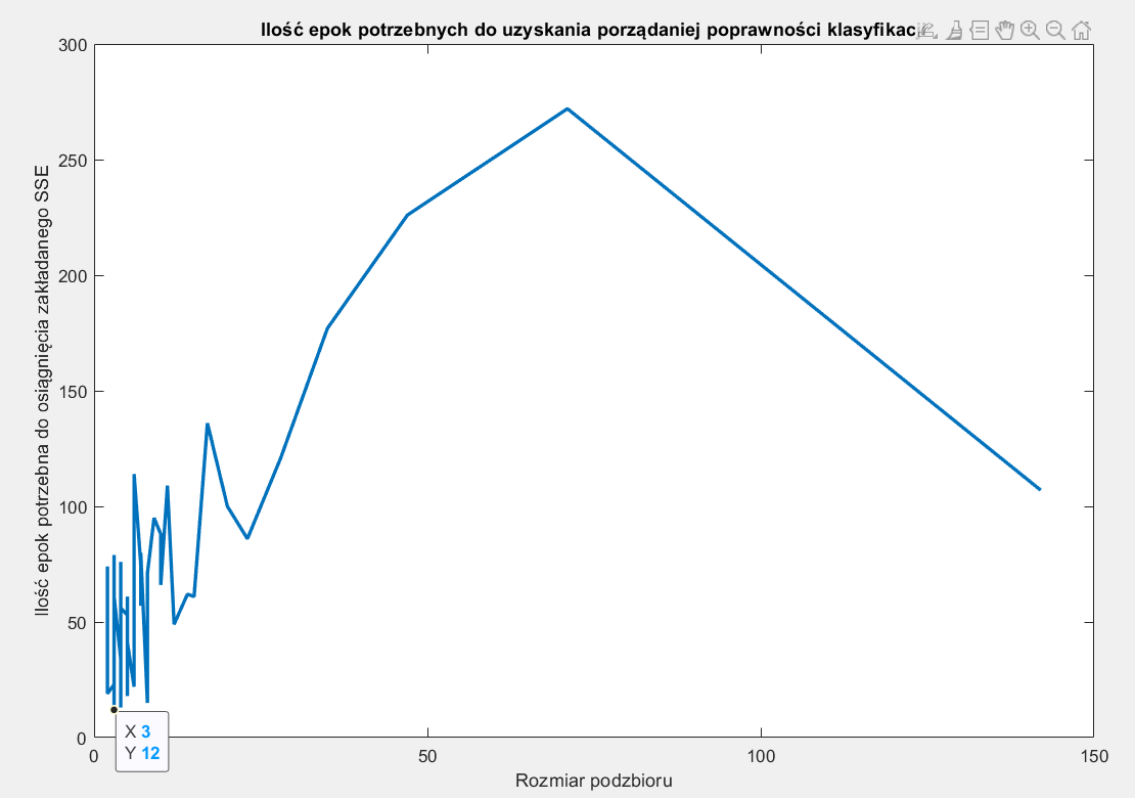
## 5.4. Eksperyment 4 - badanie wielkości podzbiorów

Celem czwartego eksperymentu było wyznaczenie optymalnej wielkości mini-batcha, dla którego obliczany jest gradient. Jako parametry przyjęto najoptymalniejsze z poprzednich eksperymentów: $S1 = 7; S2 = 4; lr_{inc} = 1.55; lr_{dec} = 0.15; err_inc = 1.04$. Rozmiar mini-batchy został określony według następującego wzoru $\frac{len(trainingData)}{n}$, gdzie $n$ należy do przedziału $< 1; 50 >$ ze skokiem co 1.

```
import network
from data import loadData
import pandas as pd
import numpy as np
from tabulate import tabulate
trainData, testData = loadData()
results = []
n=len(trainData)
size_vec = [n//x for x in range(1,50)]
for size in size_vec:
result = [size]
net = network.Network([13, 7, 4, 3])
result.extend(net.train(trainData,testData,inc=1.55,dec=0.15,
 err_inc=1.04,mini_batch_size=size))
results.append(result)
```

```
15
16  results=pd.DataFrame(results)
17  results.to_csv('batch.csv',index=None,header=None)
18
19
```

Listing 6: Algorytm realizujący eksperyment 3




Rysunek 5.8: Wykres wpływu parametru $mini_batch_size$ na poprawność klasyfikacji

Z wykresu 5.8 można zauważyć, że prędkość uczenia wzrasta wraz ze spadkiem rozmiaru pojedynczego podzbioru. Najbardziej optymalną wartością tego parametru jest wartość 3, dla której sieć potrzebuje tylko 12 epok do osiągnięcia zamierzonej wartości SSE.

# 6. Podsumowanie i wnioski końcowe

Cel projektu, tj. zrealizowanie sztucznej sieci neuronowej uczonej algorytmem wstecznej propagacji błędu z przyśpieszeniem metodą adaptacyjnego współczynnika uczenia (trainbpa) uczącej się diagnozowania choroby został zrealizowany. W tym celu wykorzystano własną implementację sieci neuronowej napisanej w jęyku Python na podstawie książki Michaela Nielsena [2].

Zadany zbiór okazał się być zbiorem trudnym do nauczenia. Procentowa dokładność klasyfikacji nie przekraczała nigdy $81,82\%$.

Eksperymenty pozwoliły na określenie optymalnych wartości parametrów, dla których sieć osiąga najlepsze wyniki.

Pierwszy eksperyment polegał na dostosowaniu ilości neuronów w warstwie I i II. Wyniki eksperymentu nie były zadowalające, najlepszy wynik osiągnięty przez sieć wynosił $81,82\%$ i został uzyskany dla maksymanej liczby epok równej 2000 oraz parametrów $S1 = 17$, $S2 = 7$ i $S1 = 17$, $S2 = 15$.

Celem drugiego eksperymentu było wyznaczenie optymalnych wartości parametrów $lr_{inc}$ oraz $lr_{dec}$. Badanie to przeprowadzono dla najlepszych wyników otrzymanych z eksperymentu 1. Okazało się, iż istnieje wiele wartości badanych parametrów, dla których sieć uczy się dobrze. Na podstawie uzyskanych wykresów można także wywnioskować, że sieć jest lepiej uwarunkowana dla $S1 = 17$ oraz $S2 = 7$.

Trzeci, ostatni eksperyment przeprowadzono w celu wyznaczenia najbardziej optymalnego parametru $er$. Obserwując wynik badania, można zauważyć, że sieć osiąga najlepsze wartości dla $er = \{1.04, 1.06, 1.07\}$.

Przeprowadzone eksperymenty pozwoliły na ustalenie optymalnych wartości parametrów, dla których sieć osiągała najlepszy wynik poprawności klasyfikacji. Badania te wykazały, że domyślne parametry z programu Matlab są najkorzystniejsze.

# Literatura


[1] UCI Machine Learning Repository
https://archive.ics.uci.edu/ml/datasets/Wine [Dostęp 14.05.2022 r.]

[2] Michael A. Nielsen, Ńeural Networks and Deep Learning", Determination Press, 2015

[3] http://sztuczna-inteligencja.eprace.edu.pl/1001,Funkcje-normalizacji.html [Dostęp 14.05.2022 r.]

[4] dr hab. inż. Roman Zajdel prof. PRz, PRz, KIiA, Sztuczna inteligencja, Laboratorium, Ćw8 Sieć jednokierunkowa jednowarstwowa
http://materialy.prz-rzeszow.pl/pracownik/pliki/34/sztuczna-inteligencja-cw8-siec-jednowarstw.pdf [Dostęp 09.06.2022 r.]

[5] dr hab. inż. Roman Zajdel prof. PRz, PRz, KIiA, Sztuczna inteligencja, Laboratorium, Ćw9 Sieć jednokierunkowa wielowarstwowa
http://materialy.prz-rzeszow.pl/pracownik/pliki/34/sztuczna-inteligencja-cw9-siec-wielowarstw.pdf [Dostęp 09.06.2022 r.]

[6] https://miroslawmamczur.pl/czym-jest-i-jak-sie-uczy-sztuczna-siec-neuronowa/ [Dostęp 21.05.2022 r.]

[7] R. Tadeusiewicz, M. Szaleniec „Leksykon sieci neuronowych”, Wrocław 2015