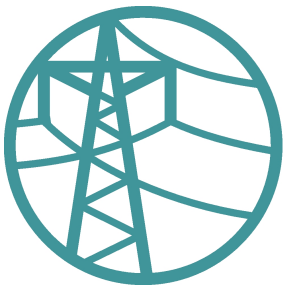


WYDZIAŁ
ELEKTROTECHNIKI
I INFORMATYKI
POLITECHNIKI RZESZOWSKIEJ

Karol Dudek

Realizacja sieci neuronowej uczonej algorytmem wstecznej propagacji błędu ucząca się rozpoznawać rodzaj schorzenia u pacjenta na podstawie wyników jego badań.

Praca projektowa z przedmiotu
"Sztuczna Inteligencja"

Opiekun pracy:
dr hab. inż. Roman Zajdel, prof. PRz

Rzeszów, 2022

Spis treści

1. Wstęp **5**
1.1. Cel projektu 5
1.2. Opis problemu 6
1.3. Opis zestawu danych 6
1.4. Przygotowanie danych 7
2. Zagadnienia teorytyczne **8**
2.1. Model sztucznego neuronu 8
2.2. Funkcja aktywacji 9
2.3. Model sieci wielowarstwowej 10
2.4. Uczenie sieci pod nadzorem (supervised learning) 12
2.5. Algorytm wstecznej propagacji błędu 14
3. Realizacja sieci **16**
3.1. Opis skryptu 16
4. Eksperymenty **21**
4.1. Eksperyment 1 21
4.2. Eksperyment 2 22
4.3. Eksperyment 3 23
4.4. Eksperyment 4 24
4.5. Eksperyment 5 25
4.6. Eksperyment 6 26
4.7. Eksperyment 7 27
4.8. Eksperyment 8 28
5. Wnioski **29**
Literatura **30**

1. Wstęp

1.1. Cel projektu

Głównym założeniem projektu jest realizacja sieci neuronowej uczonej za pomocą algorytmu wstecznej propagacji błędu, której zadaniem jest zdiagnozowanie zapalenia nerek lub zapalenia pęcherza na podstawie pewnych danych wejściowych. W ramach projektu zbadano wpływ poszczególnych parametrów sieci na proces jej uczenia:

- S1 - ilość neuronów w I warstwy ukrytej
- S2 - ilość neuronów w II warstwy ukrytej
- lr (learning rate) / eta - wartość współczynnika uczenia
- target error - maksymalny docelowy błąd sieci

Jako zbiór danych uczących wykorzystano zbiór "Acute Inflammations", a sama sieć została zrealizowana przy użyciu języka programowania Python.

1.2. Opis problemu

Realizowana sieć na podstawie podanych informacji wejściowych ma za zadanie sklasyfikować, do której klasy należą te dane i na wyjściu podać informacje o rodzaju zdiagnowzowanej choroby. W zbiorze danych uczących występuje 4 możliwe klasy:

- brak chorób
- występuje zapalenie nerek
- występuje zapalenie pęcherza
- występuje jednocześnie zapalenie pęcherza i nerek

1.3. Opis zestawu danych

Zestaw danych uczących został pobrany ze strony:
https://archive.ics.uci.edu/ml/datasets/Acute+Inflammations

Zbiór danych uczących zawiera 120 rekordów po 6 atrybutów w każdym wierszu. W przypadku atrybutów, będących parametrami wejściowymi są to:

- a1 - Temperatura pacjenta
- a2 - Zawroty głowy
- a3 - Ból lędzwiowy
- a4 - Ciągła potrzeba oddania moczu
- a5 - Bóle mikcyjne
- a6 - Pieczenie cewki moczowej

Oraz dwie możliwe decyzje wyjściowe:

- d1 - Zapalenie pęcherza
- d2 - Zapalenie nerek

1.4. Przygotowanie danych

Badany zestaw danych nie zawiera niekompletnych rekordów, oraz wartości niepoprawnych, dlatego też podczas przygotowywania danych nie napotkano żadnych nieprzewidzianych błędów.

W celu ujednolicenia danych do poźniejszego przetwarzania na danych przeprowadzono normalizację zgodnie z poniższym wzorem:

$$x_{norm} = \frac{x - x_{min}}{x_{max} - x_{min}} \tag{1.1}$$

gdzie:

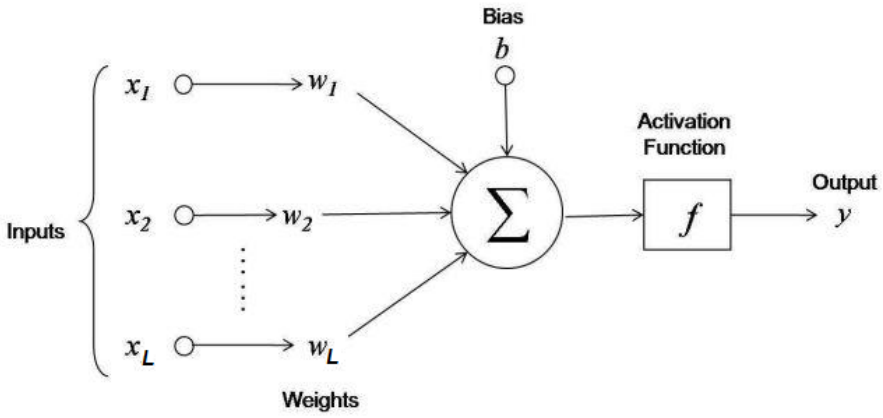
- x_{norm} - wartość po normalizacji
- x - wartośc przed normalizacją
- x_{min} - minimalna wartość w zbiorze
- x_{max} – maksymalna wartość w zbiorze

Dzięki zastosowaniu wzoru 1.1 pierwotne dane wejściowe każdego rekordu zostały zamienione na odpowiadające im wartości z zakresu od 0 do 1.

2. Zagadnienia teorytyczne

2.1. Model sztucznego neuronu

Każda sieć neuronowa składa się z połączonych między sobą pojedynczych neuronów. Należy zatem zapoznać się z modelem pojedynczego neuronu w celu zrozumienia problemu sieci neuronowych. Przykładowy model pojedynczego neuronu został przedstawiony na rysunku 2.1



Rysunek 2.1: Model pojedynczego neuronu

Każdy neuron to układ składający się z wielu wejść i jednego wyjścia. Wszystkie wejścia posiadają tzw. współczynnik wagowy, który określa jak bardzo dane wejście wpływa na rezultat wynikowy danego neuronu. Oprócz wagi dodatkowym elementem wejściowych jest również bias, umożliwiający przesunięcie funkcji aktywacji w lewo lub w prawo danego neuronu. Całość z poprzednio podanych informacji do wejść neuronu trafia do sumatora gdzie odbywa się proces wyznaczania łącznego pobudzenia neuronu, wyrażanego z następującego wzoru 2.2:

$$z = \sum_{j=1}^{L} w_j x_j + b \tag{2.2}$$

Wyznaczona wartość następnie trafia do funkcji aktywacji, gdzie określany jest sygnał wyjściowy neuronu zgodnie z zależnością 2.3:

$$y = f(z) = f(\sum_{j=1}^{L} w_j x_j + b) \tag{2.3}$$

gdzie:

- y - wyjście neuronu
- x_j - j-ty sygnał wejściowy (j=1,2,...,L)
- w_j - waga j-tego wejścia
- b - bias

2.2. Funkcja aktywacji

Każda warstwa w swoich neuronach może wykorzystywać inną funkcję aktywacji. W przypadku sieci jednokierunkowych najbardziej powszechna jest funkcja sigmoidalna, w której wyróżnić można dwa typy:

- unipolarna funkcja aktywacji, która przyjmuje wartości w przedziału (0,1):

$$f(x) = \frac{1}{1+e^{-x}} \tag{2.4}$$

- bipolarna funkcja aktywacji, przyjmująca wartości z przedziału (-1,1):

$$f(x) = \frac{2}{1+e^{-x}} - 1 \tag{2.5}$$

Funkcje te są różniczkowalne i ich pochodne wyrazić można jako:

- w przypadku unipolarnej funkcji aktywacji:

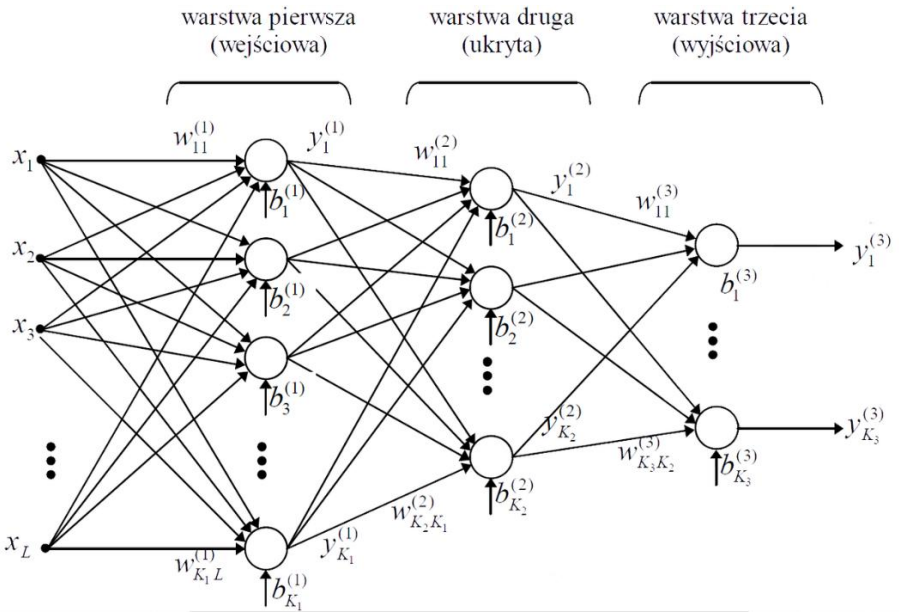
$$f(x) = \frac{1}{1+e^{-x}}(1 - \frac{1}{1+e^{-x}}) \tag{2.6}$$

- w przypadku bipolarnej funkcji aktywacji:

$$f(x) = 1 - (\frac{2}{1+e^{-x}})^2 \tag{2.7}$$

2.3. Model sieci wielowarstwowej

W każdej sieci jednokierunkowej wielowarstwowej wyróżnić można nastepujące warstwy: wejścia, wyścia i występujące pomiędzy nimi warstwy ukryte. W przypadku sieci jednokierunkowej dane mogą przepływać tylko w jednym kierunku od warstwy wejścia do warstwy wyjścia.



Rysunek 2.2: Model sieci jednokierunkowej wielowarstwowej

Każda pojedyncza warstwa neuronów posiada:

- macierz wag **w**
- wektor przesunieć **b**
- funkcje aktywacji **f**
- oraz wektor sygnałów wyjściowych **y**

W celu obliczenia sygnału wyjściowego sieci wielowarstwowej uogólniony wzór na wyjście pojedynczej warstwy przedstawiający się jako:

$$y^{(n)} = f^{(n)}(w^{(n)}y^{(n-1)} + b^{(n)}) \tag{2.8}$$

gdzie n to numer odpowiedniej warstwy.

Dla przykładu dla przedstawionej sieci 2.2 wzory opisujące wyjścia poszczególnych warstw przyjmą postać:

$$y^{(1)} = f^{(1)}(w^{(1)}x + b^{(1)}) \tag{2.9}$$

$$y^{(2)} = f^{(2)}(w^{(2)}y^{(1)} + b^{(2)}) \tag{2.10}$$

$$y^{(3)} = f^{(3)}(w^{(3)}y^{(2)} + b^{(3)}) \tag{2.11}$$

Na podstawie powyższych równań sygnał wyjściowy całej sieci 2.2 można opisać wzorem:

$$y^{(3)} = f^{(3)}(w^{(3)}f^{(2)}(w^{(2)}f^{(1)}(w^{(1)}x + b^{(1)}) + b^{(2)}) + b^{(3)}) \tag{2.12}$$

2.4. Uczenie sieci pod nadzorem (supervised learning)

Uczenie z nadzorem polega na wprowadzaniu do systemu uczącego się próbki danych (danych uczących) posiadającej pary danych w postaci tzw. wektora wejściowego i wektora wyjściowego, gdzie:

- wektor wejściowy - wektor podawany na wejście sieci
- wektor wyjściowy - wektor oczekiwanych wartości na wyjściu sieci

Zatem ten sposób uczenia zakłada, że każdemu wektorowi wejściowemu towarzyszy wektor wyjściowy. Taka para jest wykorzystywana w procesie uczenia, a dokładnie zmiany wag i biasów neuronów w kolejnych jego cyklach. W sieci nienauczonej w większości przypadków sygnał wyjściowy y będzie inny niż oczekiwany $\hat{y}$. Taki błąd nazywa się funkcją straty, która według wzoru dla pojedynczego neuronu wyjściowego przedstawić można jako:

$$e_i = y_i - \hat{y}_i \tag{2.13}$$

Funkcja straty w poźniejszej fazie wykorzystywana jest do obliczenia wartości tzw. funkcji kosztu, która w bezpośredni sposób bierze udział w procesie uczenia. Najczęsciej stosowaną funkcją kosztu dla klasyfikacji to:

- SSE (Sum Squared Error) - sumaryczny błąd kwadratowy:

$$E = \frac{1}{2}\sum_{i=1}^{K} e_i^2 \tag{2.14}$$

gdzie:

- i - numer neuronu w warstwie wyjściowej
- K - ilość neuronów w warstwie wyjściowej.

Korzystając z wzoru opisującego sygnał wyjściowy sieci 2.12 można przedstawić pełne rozwinięcie funkcji celu:

$$\begin{aligned}
E &= \frac{1}{2}\sum_{i_3=1}^{K_3} e_{i_3}^2 = \frac{1}{2}\sum_{i_3=1}^{K_3}(y_{i_3}^{(3)} - \hat{y_{i_3}})^2 \\
&= \frac{1}{2}\sum_{i_3=1}^{K_3}(f^{(3)}(w^{(3)}f^{(2)}(w^{(2)}f^{(1)}(w^{(1)}x + b^{(1)}) + b^{(2)}) + b^{(3)}) - \hat{y_{i_3}})^2 \\
&= \frac{1}{2}\sum_{i_3=1}^{K_3}(f^{(3)}(\sum_{i_2=1}^{K_2} w_{i_3 i_2}^{(3)} y_{i_2} + b_{i_3}^{(3)}) - \hat{y_{i_3}})^2 \\
&= \frac{1}{2}\sum_{i_3=1}^{K_3}(f^{(3)}(\sum_{i_2=1}^{K_2} w_{i_3 i_2}^{(3)} f^{(2)}(\sum_{i_1=1}^{K_1} w_{i_2 i_1}^{(2)} y_{i_1} + b_{i_2}^{(2)}) + b_{i_3}^{(3)}) - \hat{y_{i_3}})^2 \\
&= \frac{1}{2}\sum_{i_3=1}^{K_3}(f^{(3)}(\sum_{i_2=1}^{K_2} w_{i_3 i_2}^{(3)} f^{(2)}(\sum_{i_1=1}^{K_1} w_{i_2 i_1}^{(2)} f^{(1)}(\sum_{j=1}^{L} w_{i_1 j}^{(1)} x_j + b_{i_1}^{(1)}) + b_{i_2}^{(2)}) + b_{i_3}^{(3)}) - \hat{y_{i_3}})^2
\end{aligned} \tag{2.15}$$

gdzie:

- $j = 1, \ldots, L$ numer wejścia warstwy I
- $j_1 = 1, \ldots, K_1,\ j_1 = 1, \ldots, K_1,\ j_1 = 1, \ldots, K_1$ odpowiednio numer wyjścia warstwy I, II, III

2.5. Algorytm wstecznej propagacji błędu

Podczas uczenia sieci neuronowych celem każdego zastosowanego algorytmu jest modyfikowanie wag danego neuronu w taki sposób, aby na wyjściu sieć popełniała jak najmniejszy błąd. Obecnie najbardziej rozpowszechnione są dwa rodzaje / sposoby uczenia sieci. Pierwszy sposób polega na aktulizacji wag po przejściu wszystkich danych uczących. Drugi natomiast za pomocą gradientów i reguły łańcuchowej powracaniu do początku sieci i rozkładaniu odpowiednio wartości błędów próbując poprawić wagi. Proces ten nazwę propagacji wstecznej.

Na początku jedyną rzeczą jaką można wyznaczyć to błędy neuronów warstwy wyjściowej na podstawie danych wyjściowych i danych docelowych zawartych w zbiorze uczącym. Kolejne błędy w warstwach wcześniejszych niż wyjściowa będą określane dzięki algorytmowi wstecznej propagacji. Wzór umożliwiający obliczenie zmiany wagi przedstawia się jako:

$$\Delta w_{ij} = -\eta \frac{\partial E}{\partial w_{ij}} \tag{2.16}$$

gdzie η oznacza wartość współczynnika uczenia Opisując algorytm można dojść do wniosku, że należy on do metod gradientowych, gdyż jego założenie opiera sie na stwierdzeniu, że gradient funkcji wskazuje na kierunek jej najszybszego wzrostu, a przy zmianie znaku na przeciwny kierunek jej najszybszego spadku. Zakładając, że każda zmiana wagi zależy od danej chwili uczenia t (tutaj zwykle epoki) wartość danej wagi można zapisać jako:

$$w_{ij}(t+1) = w_{ij}(t) + \Delta w_{ij} \tag{2.17}$$

Podstawienie równania 2.16 pozwala na przekształcenie powyższego wzoru do postaci:

$$w_{ij}(t+1) = w_{ij}(t) - \eta \frac{\partial E}{\partial w_{ij}} \tag{2.18}$$

Przy obliczaniu gradientu dla danej wagi, napotyka się problem o niemożności określenia go za pomocą bezpośrednich obliczeń, dlatego też najpierw należy określić wygląd poszczególnych pochodnych cząstkowych z zależności 2.15, a następnie na ich podstawie odpowiadający danej wadze gradient:

- dla warstwy wyjściowej:

$$\frac{\partial E}{\partial w^{(3)}_{i_3 i_2}} = \frac{\partial E}{\partial f^{(3)}} \frac{\partial f^{(3)}(z^{(3)}_{i_3})}{\partial z^{(3)}_{i_3}} \frac{\partial z^{(3)}_{i_3}}{\partial w^{(3)}_{i_3 i_2}} = (y_{i_3} - \hat{y_{i_3}}) \frac{\partial f^{(3)}(z^{(3)}_{i_3})}{\partial z^{(3)}_{i_3}} y^{(2)}_{i_2} \tag{2.19}$$

gdzie: $z^{(3)}_{i_3} = \sum_{i_2=1}^{K_2} w^{(3)}_{i_3 i_2} y_{i_2} + b^{(3)}_{i_3}$ - pobudzenie i_3-ego neuronu warstwy wyjściowej.

- dla warstwy ukrytej:

$$\begin{aligned}\frac{\partial E}{\partial w_{i_2 i_1}^{(3)}} &= \frac{\partial E}{\partial f^{(3)}} \frac{\partial f^{(3)}(z_{i_3}^{(3)})}{\partial z_{i_3}^{(3)}} \frac{\partial z_{i_3}^{(3)}}{\partial f^{(2)}} \frac{\partial f^{(2)}(z_{i_2}^{(2)})}{\partial z_{i_2}^{(2)}} \frac{\partial z_{i_2}^{(2)}}{\partial w_{i_2 i_1}^{(2)}} \\ &= \sum_{i_3=1}^{K_3} (y_{i_3} - \hat{y_{i_3}}) \frac{\partial f^{(3)}(z_{i_3}^{(3)})}{\partial z_{i_3}^{(3)}} w_{i_3 i_2}^{(3)} \frac{\partial f^{(2)}(z_{i_2}^{(2)})}{\partial z_{i_2}^{(2)}} y_{i_1}^{(1)}\end{aligned} \tag{2.20}$$

- dla warstwy wejściowej:

$$\begin{aligned}\frac{\partial E}{\partial w_{i_1 j}^{(3)}} &= \frac{\partial E}{\partial f^{(3)}} \frac{\partial f^{(3)}(z_{i_3}^{(3)})}{\partial z_{i_3}^{(3)}} \frac{\partial z_{i_3}^{(3)}}{\partial f^{(2)}} \frac{\partial f^{(2)}(z_{i_2}^{(2)})}{\partial z_{i_2}^{(2)}} \frac{\partial z_{i_2}^{(2)}}{\partial f^{(1)}} \frac{\partial f^{(1)}(z_{i_1}^{(1)})}{\partial z_{i_1}^{(1)}} \frac{\partial z_{i_1}^{(1)}}{\partial w_{i_1 j}^{(1)}} \\ &= \sum_{i_3=1}^{K_3} (y_{i_3} - \hat{y_{i_3}}) \frac{\partial f^{(3)}(z_{i_3}^{(3)})}{\partial z_{i_3}^{(3)}} \sum_{i_2=1}^{K_2} w_{i_3 i_2}^{(3)} \frac{\partial f^{(2)}(z_{i_2}^{(2)})}{\partial z_{i_2}^{(2)}} w_{i_2 i_1}^{(2)} \frac{\partial f^{(1)}(z_{i_1}^{(1)})}{\partial z_{i_1}^{(1)}} x_j\end{aligned} \tag{2.21}$$

3. Realizacja sieci

3.1. Opis skryptu

Na potrzeby realizacji sieci neuronowej stworzono skrypt w języku Python umożliwiającą w prosty sposób zrealizować dowolną sieć neuronową uczoną metodą wstecznej propagracji błędu. Skrypt został podzielony na 3 pliki: network.py, data.py oraz main.py. Skrypt data.py miał za zadanie prygotować dane pobrane ze zbioru danych do optymalnego formatu dla sieci neuronowej. Skrypt network.py zawierał cały kod sieci neuronowej, natomiast skrypt main.py łączył 2 wcześniej wspomniane skrypty.

```
from sklearn.model_selection import train_test_split
from csv import reader
import numpy as np

# Import function
def dataImport(name):
    with open(name, 'r', encoding='utf-16') as file:
        return [line for line in reader(file, delimiter='\t')]

# Normalization function
def normalizeMinMax(table):
    for row in range(0, len(table)):
        min_val, max_val = min(table[row]), max(table[row])
        table[row] = [(1 - 0) * (col - min_val) / (max_val - min_val) for col in table
    [row]]
    return table

# Data loader function
def loadData():
    # Import acute.tsv to dataFile
    dataFile = dataImport('acute.tsv')

    # Create numpy array from dataList
    dataFile = np.array(dataFile)

    # Convert array of strings to array of floats
    dataFile = dataFile.astype(float)

    # Data normalization
    dataFile = normalizeMinMax(dataFile.T).T

    # Data splitting into training and test data
    trainData, testData = train_test_split(dataFile, test_size=0.2, random_state=25)

    # Splitting data into 2 groups, inputData and outputdata
    testIn, testOut = testData[:,:6], testData[:,6:]
```

```
    trainIn, trainOut = trainData[:,:6], trainData[:,6:]

    # Combining inputData and outputData in a single tuple
    trainData = [(np.array(trainIn[i], ndmin=2).T, np.array(trainOut[i], ndmin=2).T)
    for i in range(0, len(trainOut))]
    testData = [(np.array(testIn[i], ndmin=2).T,np.array(testOut[i], ndmin=2).T) for i
     in range(0, len(testOut))]

    return (trainData, testData)
```

Listing 1: Plik przygotowujący dane- data.py

```
import random
import time
import numpy as np

class Network(object):

    # Constructor, takes list of layers and amount of neurons as parameter
    def __init__(self, sizes):

        #Applying Seed
        np.random.seed(7)

        # Assing 'sizes' vector to amount of layers in the network
        self.num_layers = len(sizes)
        self.sizes = sizes

        # Pseudo random generator used to assign weight and biases
        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:])]

    def feedforward(self, a):

        # Return neural network results for 'a' data
        for b, w in zip(self.biases, self.weights):
            a = sigmoid(np.dot(w, a)+b)
        return a

        # Mean Square Error
    def mse(self,_test_data):
        error=[pow(np.linalg.norm(self.feedforward(x)-y),2) for (x,y) in _test_data]
        return 1/len(_test_data)*sum(error)

    def SGD(self, training_data, epochs, mini_batch_size, eta,
            error_target=0.001, test_data=None):

        if test_data: n_test = len(test_data)
        n = len(training_data)
```

```
        for j in range(epochs):
            time1 = time.time()
            random.shuffle(training_data)
            mini_batches = [
                training_data[k:k+mini_batch_size]
                for k in range(0, n, mini_batch_size)]
            for mini_batch in mini_batches:
                self.update_mini_batch(mini_batch, eta)
            cur_err = self.mse(training_data)
            time2 = time.time()
            evalVal = self.evaluate(test_data)
            evalAcc = (evalVal/n_test*100)
            if cur_err < error_target or j == epochs-1:
                if test_data:
                    print("{0}, {2:.2f}, {3:.0f}%".format(
                        j, cur_err, evalAcc))
                    pass
                else:
                    print("Epoch {0} ".format(j))
                break

            print("{0}, {1:.6f}, {2:.0f}%".format(j, cur_err, evalAcc))

    def update_mini_batch(self, mini_batch, eta):

        # Updates weights and biases using SGD and backpropagation for each mini batch
        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:
            # Calculate gradient increase for each (x, y) pair
            delta_nabla_b, delta_nabla_w = self.backprop(x, y)

            # Calculate new gradient
            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)]

        # New weights and biases
        self.weights = [w-(eta/len(mini_batch))*nw
                        for w, nw in zip(self.weights, nabla_w)]
        self.biases = [b-(eta/len(mini_batch))*nb
                       for b, nb in zip(self.biases, nabla_b)]

    def backprop(self, x, y):

        #Return tuple representing the gradient of the cost function
        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
```

```
89          activations = [x] # list to store all the activations, layer by layer
90          zs = [] # list to store all the z vectors, layer by layer
91
92          # Calculate neuron activations
93          for b, w in zip(self.biases, self.weights):
94              z = np.dot(w, activation)+b
95              zs.append(z)
96              activation = sigmoid(z)
97              activations.append(activation)
98
99          # backward pass (gradient increase for output layer)
100         delta = self.cost_derivative(activations[-1], y) * \
101             sigmoid_prime(zs[-1])
102         nabla_b[-1] = delta
103         nabla_w[-1] = np.dot(delta, activations[-2].transpose())
104
105         # Calculate gradient increase for input and hidden layers
106         for l in range(2, self.num_layers):
107             z = zs[-l]
108             sp = sigmoid_prime(z)
109             delta = np.dot(self.weights[-l+1].transpose(), delta) * sp
110             nabla_b[-l] = delta
111             nabla_w[-l] = np.dot(delta, activations[-l-1].transpose())
112         return (nabla_b, nabla_w)
113
114     def evaluate(self, test_data):
115
116         test_results = [(self.feedforward(x), y)
117                         for (x, y) in test_data]
118
119                         # Approximation
120         return sum(int((y[0] == 0 and x[0] < 0.5) or (y[0] == 1 and x[0] > 0.5) and
121                        (y[1] == 0 and x[1] < 0.5) or (y[1] == 1 and x[1] > 0.5))
122                    for (x, y) in test_results)
123
124     def cost_derivative(self, output_activations, y):
125         # Return vector with difference between the neuron and the expected result
126         return (output_activations-y)
127
128 #### Miscellaneous functions
129 def sigmoid(z):
130     # Sigmoid function
131     return 1.0/(1.0+np.exp(-z))
132
133 def sigmoid_prime(z):
134     # Sigmoid prime function
135     return sigmoid(z)*(1-sigmoid(z))
```

Listing 2: Plik zawierający sieć - network.py

```
import data
import network

import numpy as np

trainData, testData = data.loadData()

# [input vector size, S1 neurons, S2 neurons, output]
net = network.Network([6,2])

# (training_data, epochs, batch_size, eta, target, test_data)
net.SGD(trainData, 100000, 1, 0.1, error_target=0.179,test_data=testData)
```

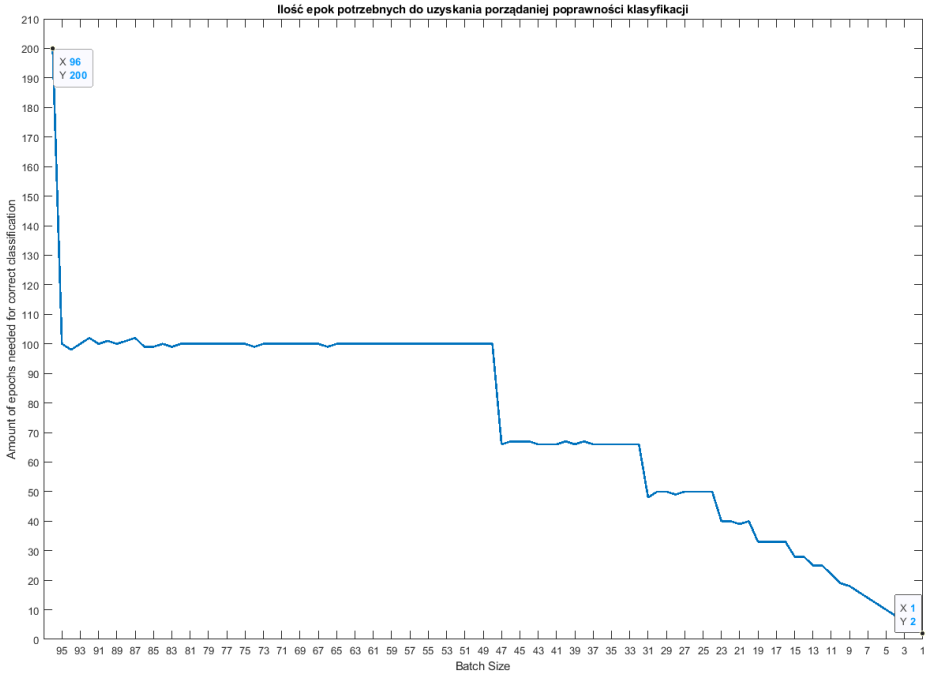
Listing 3: Plik wywołujący przykładową sieć - main.py

4. Eksperymenty

4.1. Eksperyment 1

Celem pierwszego eksperymentu było sprawdzenie jak wielkość partii danych uczących wpływa na szybkość uczenia się sieci. W tym eksperymencie parametry sieci jednowarstwowej były następujące:

- epoki - 10000
- learning rate - 0,3
- target error - 0,179



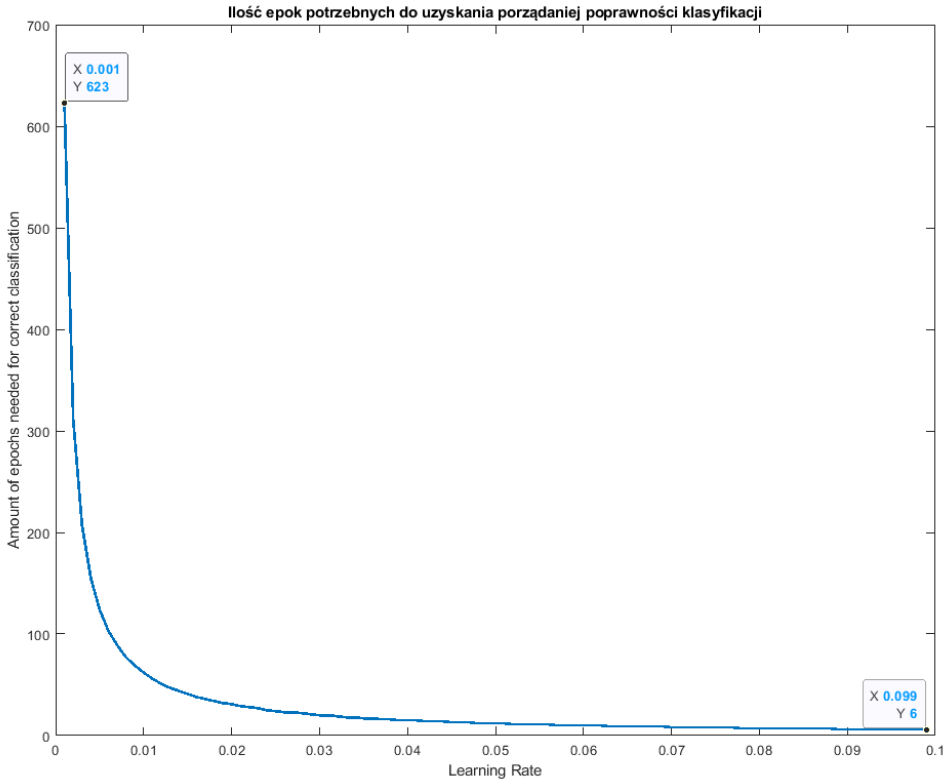
Rysunek 4.3: Wykres do eksperymentu pierwszego

Z powyższego wykresu możemy zauważyć, iż batch size wielkości 1 wypada najlepiej pod kątem szybkości uczenia się sieci.

4.2. Eksperyment 2

Celem drugiego eksperymentu było sprawdzenie jak learning rate wpływa na szybkość uczenia się sieci. W tym eksperymencie parametry sieci jednowarstwowej były następujące:

- epoki - 10000
- batch size - 1
- target error - 0,18



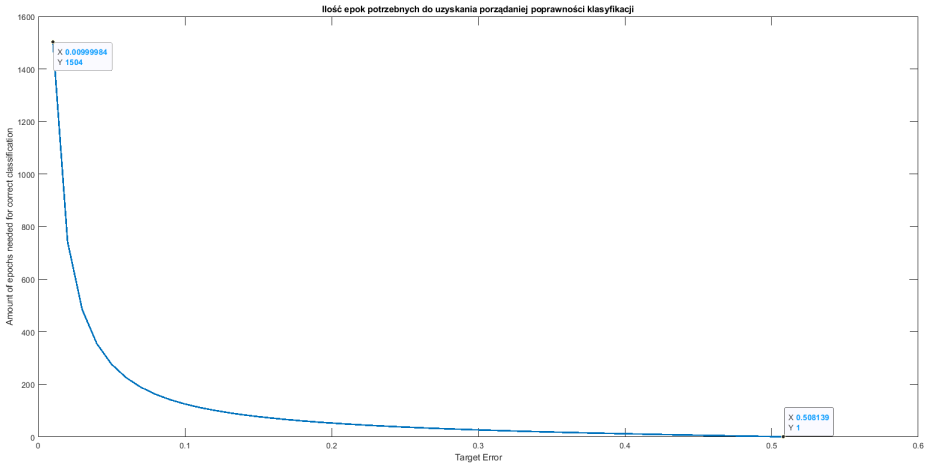
Rysunek 4.4: Wykres do eksperymentu drugiego

Z wykresu można zauważyć jak ilość epok potrzebnych do osiągniecia porządanej klasyfikacji spada wraz ze wzrostem learning rate.

4.3. Eksperyment 3

Celem trzeciego eksperymentu było sprawdzenie jak błąd docelowy wpływa na szybkość uczenia się sieci. W tym eksperymencie parametry sieci jednowarstwowej były następujące:

- epoki - 10000
- learning rate - 0,01
- batch size - 1



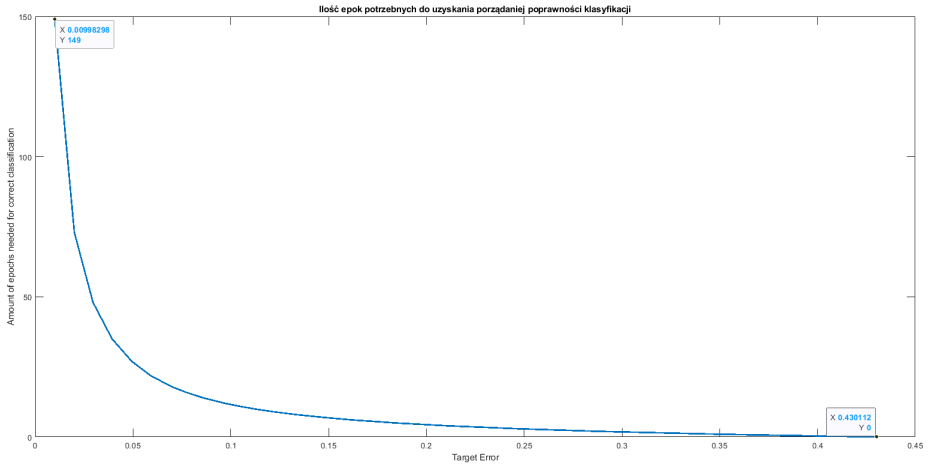
Rysunek 4.5: Wykres do eksperymentu trzeciego

Na wykresie możemy zauważyć spadek epok potrzebnych do osiągniecia porządanej klasyfikacji wraz ze wzrostem błędu docelowego.

4.4. Eksperyment 4

Celem czwartego eksperymentu było sprawdzenie jak błąd docelowy wpływa na szybkość uczenia się sieci. W tym eksperymencie parametry sieci jednowarstwowej były następujące:

- epoki - 10000
- learning rate - 0,1
- batch size - 1



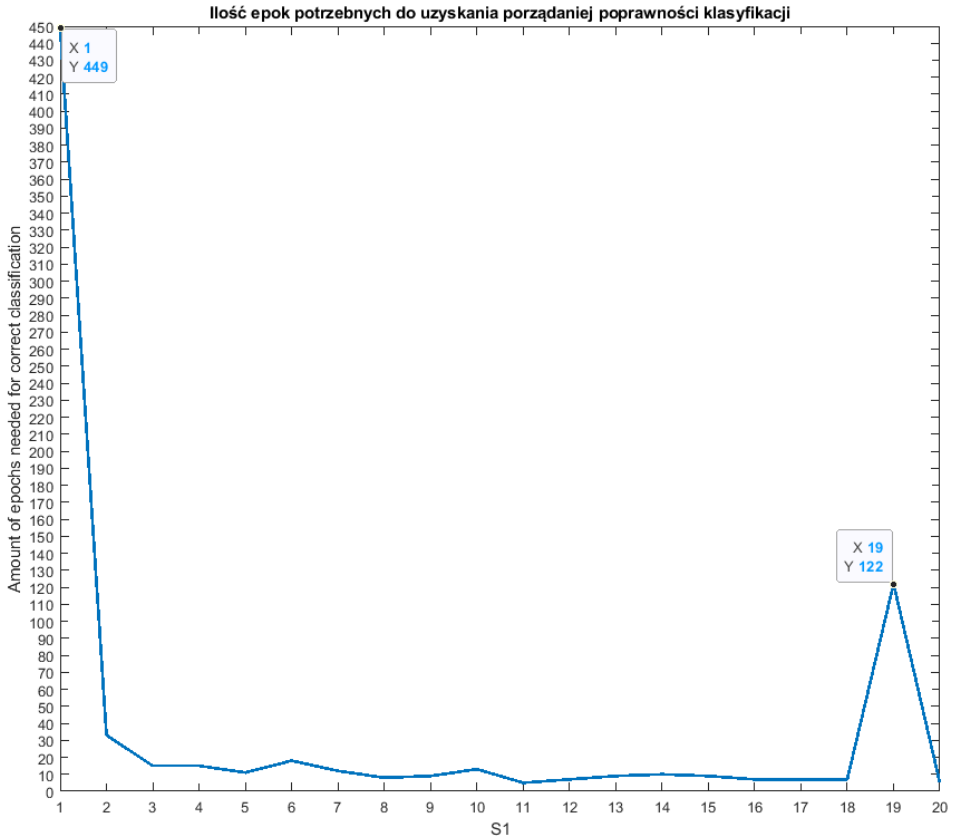
Rysunek 4.6: Wykres do eksperymentu czwartego

Tutaj został powtórzony eksperyment trzeci lecz ze zmienionym learning rate na wartość 0,1 w przeciwieństwu to poprzedniego 0,01.

4.5. Eksperyment 5

Celem piątego eksperymentu było sprawdzenie jak ilość neuronów w pierwszej wartstwie wpływa na szybkość uczenia się sieci. W tym eksperymencie parametry sieci dwuwarstwowej były następujące:

- epoki - 10000
- learning rate - 0,1
- batch size - 1
- target error - 0,18



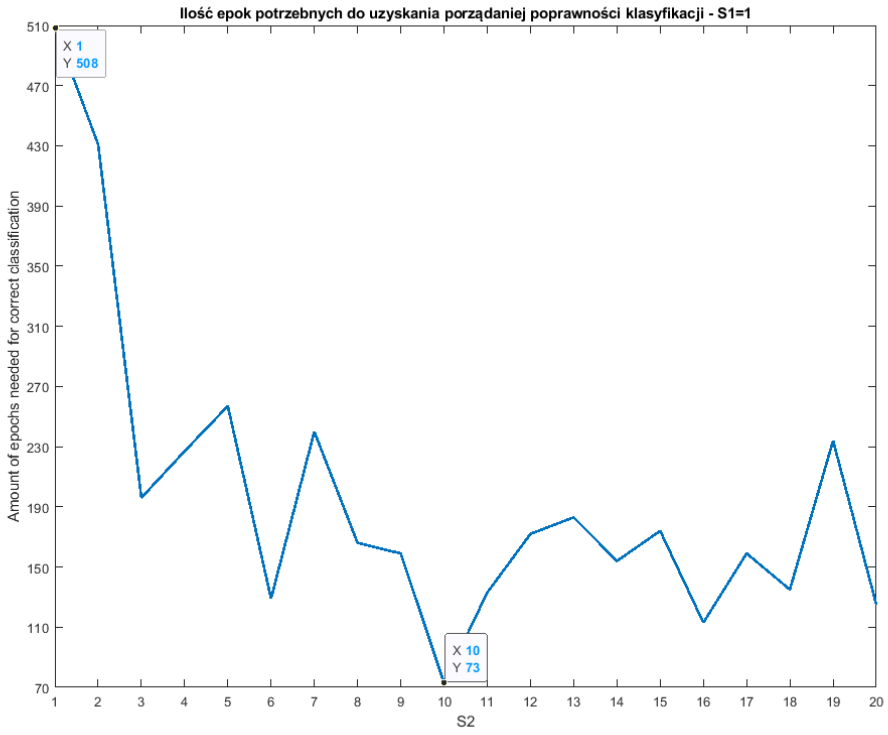
Rysunek 4.7: Wykres do eksperymentu piątego

Na wykresie możemy zauważyć ewidenty spadek ilości epok potrzebnych do osiągniecia porządanej klasyfikacji wraz ze wzrostem ilości neuronów w warstwie S1, za wyjątkiem jednego wzrostu w przypadku 19 neuronów.

4.6. Eksperyment 6

Celem szóstego eksperymentu było sprawdzenie jak ilość neuronów w drugiej wartstwie wpływa na szybkość uczenia się sieci. W tym eksperymencie parametry sieci trójwarstwowej były następujące:

- epoki - 10000
- learning rate - 0,1
- batch size - 1
- target error - 0,18
- S1 - 1



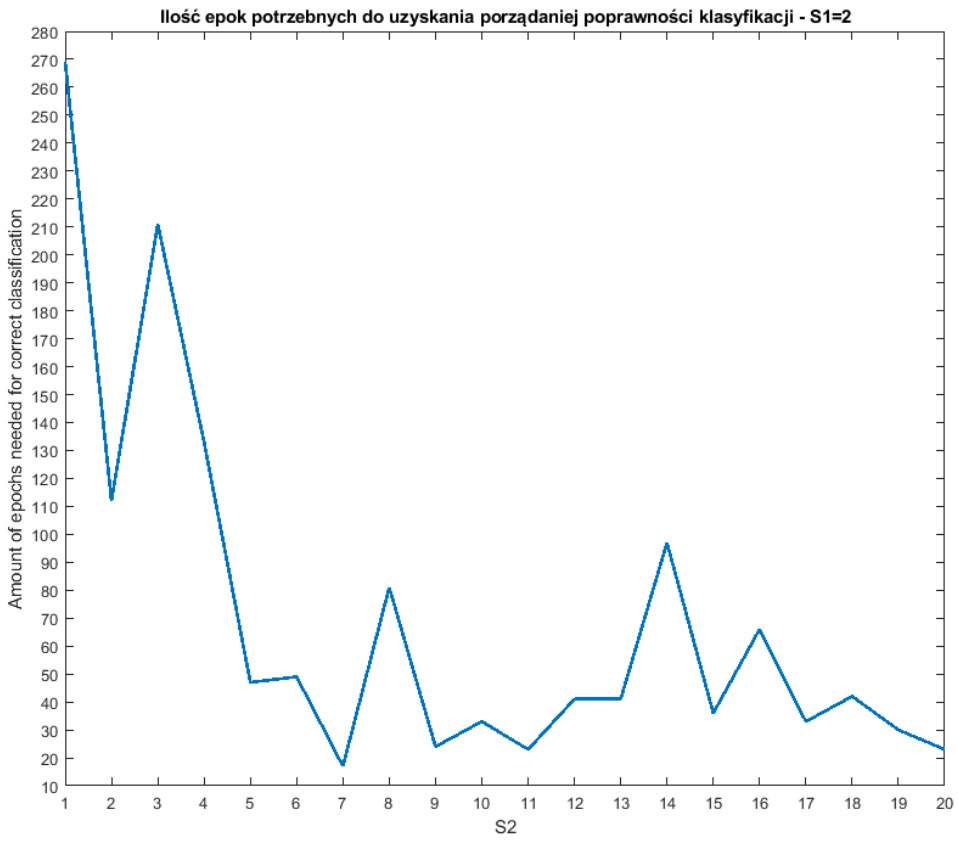
Rysunek 4.8: Wykres do eksperymentu szóstego

Pierwszy eksperyment zawierający sieć trójwarstwową, w tym eksperymencie chciałem sprawdzić jak sieć radzi sobie z nieoptymalną ilością neuronów w warstwie S1 (tylko jeden neuron), ilość neuronów w warstwie S2 jest natomiast zmienna w zakresie od 1 do 20.

4.7. Eksperyment 7

Celem siódmego eksperymentu było sprawdzenie jak ilość neuronów w drugiej wartstwie wpływa na szybkość uczenia się sieci. W tym eksperymencie parametry sieci trójwarstwowej były następujące:

- epoki - 10000
- learning rate - 0,1
- batch size - 1
- target error - 0,18
- S1 - 2



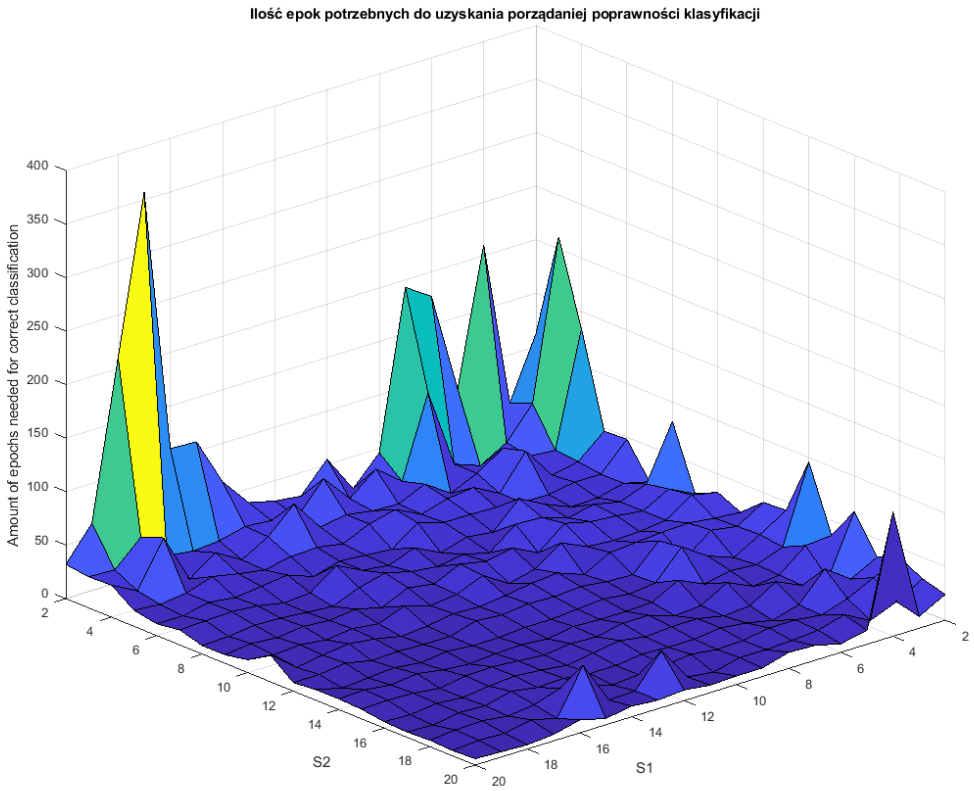
Rysunek 4.9: Wykres do eksperymentu siódmego

Tutaj został powtórzony eksperyment siódmy lecz ilość neuronów w warstwie S1 wynosiła 2, co znacznie poprawiło szybkość uczenia się sieci.

4.8. Eksperyment 8

Celem ósmego eksperymentu było sprawdzenie jak ilość neuronów w pierwszej i drugiej wartstwie wpływa na szybkość uczenia się sieci. W tym eksperymencie parametry sieci trójwarstwowej były następujące:

- epoki - 10000
- learning rate - 0,1
- batch size - 1
- target error - 0,18



Rysunek 4.10: Wykres do eksperymentu ósmego

W tym eksperymencie sprawdzam jak zmienna ilość neuronów w warstwie S1 i S2 w zakresie od 2 do 20 neuronów wpływa na szybkość uczenia się sieci. Możemy zauważyć, że już w przypadku 8 neuronów w obydwu warstwach sieć uczy się już bardzo szybko.

5. Wnioski

Celem projektu było zrealizowanie szutucznej sieci neuronowej uczonej algorytmem wstecznej propagacji błędu. W tym czelu użyto własnej implementacji sieci neuronowej napisanej w języku programowania Python na podstawie książki Michaela Nielsena.

Zbiór danych przeze mnie wykorzystywany okazał się być zbiorem liniowo separowalnym co wpłynęło na bardzo szybkie uczenie się sieci oraz pozwoliło na osiągnięcie 100% poprawności klasyfikacji.

Przeprowadzone przeze mnie eksperymenty pozwoliły określić najoptymalniejsze parametry sieci neuronowej pozwalające nauczyć sieć w jak najktrótrszym czasie zachowując przy tym jak największą dokładność. Finalnie w przypadku sieci jednowarstwowej najlepszymi parametrami okazały się:

- Ilosc neuronów w warstwie wyjściowej = 2
- ETA (learning rate) = 0,1
- Batch size = 1
- Target error = 0,18

Konfiguracja ta wynika z tego, iż zestaw danych uczących jest jak zostało wspomniane wcześniej liniowo separowalny dlatego też

Literatura

[1] https://archive.ics.uci.edu/ml/datasets/Acute+Inflammations

[2] Michael Nielsen, Neural Networks and Deep Learning.

[3] Zajdel.R „Ćwiczenie 6 Model Neuronu", Rzeszów, KIiA, PRz

[4] Zajdel.R „Ćwiczenie 8 Sieć jednokierunkowa jednowarstwowa", Rzeszów,KIiA,PRz

[5] Zajdel.R „Ćwiczenie 9 Sieć jednokierunkowa wielowarstwowa", Rzeszów,KIiA,PRz

[6] R.Tadeusiewicz, M.Szaleniec „Leksykon sieci neuronowych"