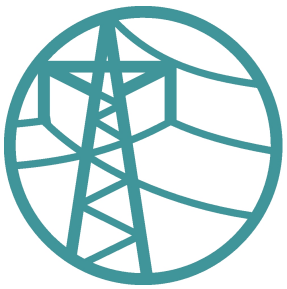


WYDZIAŁ
ELEKTROTECHNIKI
I INFORMATYKI
POLITECHNIKI RZESZOWSKIEJ

Sztuczna Inteligencja

Projekt

Temat: Zrealizować sieć neuronową uczoną algorytmem wstecznej propagacji błędu z przyśpieszeniem metodą adaptacyjnego współczynnika uczenia (trainbpa) uczącą się rozpoznawanie 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, oznaczany również jako MAX_PERF_INC
- $\mathrm{lr}_{\mathrm{dec}}$ - 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 zostaly 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

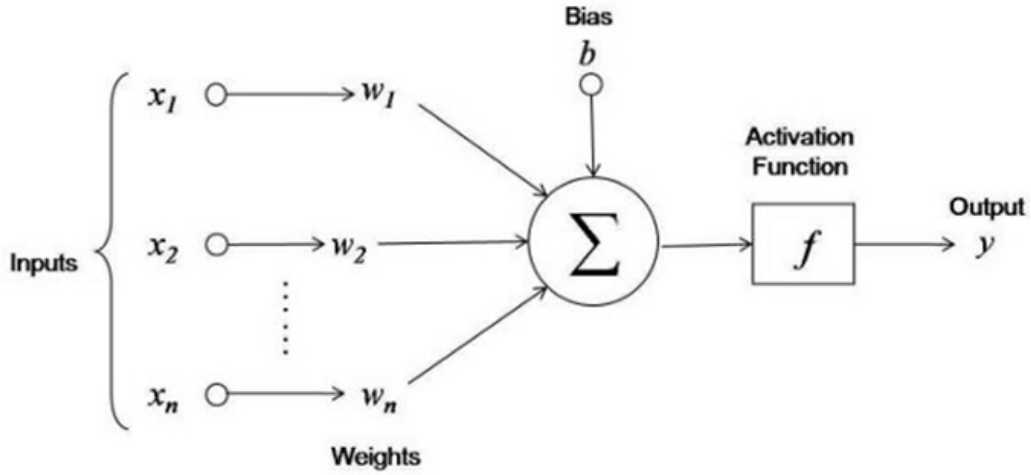
3.1. Model sztucznego neuronu

Podstawowym elementem z którego zbudowane są sieci neuronowe jest pojedynczy neuron. Neuron jest jednostką która przetwarza informacje wejściowe, 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 zamiast wartości binarnych mogą być dowolne liczby
- Połączenia węzłów mają nadaną wagę
- Wartość wyjściowa w węźle składa się z dwóch części: sumy wartości z warstw poprzednich pomnożonej przez wagi oraz nałożonej na tą sumę funkcji aktywacji.

Na rysunku 3.1 sygnał przechodzi z lewej do prawej.



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

Ze względu na funkcję aktywacji wyróżnia się różne typy neuronów. Najczęściej stosowanymi funkcjami aktywacji neuronu są funkcje liniowe (3.4) oraz sigmoidalne (3.5 oraz 3.6).

Funkcja liniowa ma postać:

$$f(x) = a \cdot x + b \tag{3.4}$$

Funkcja sigmoidalna unipolarna:

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

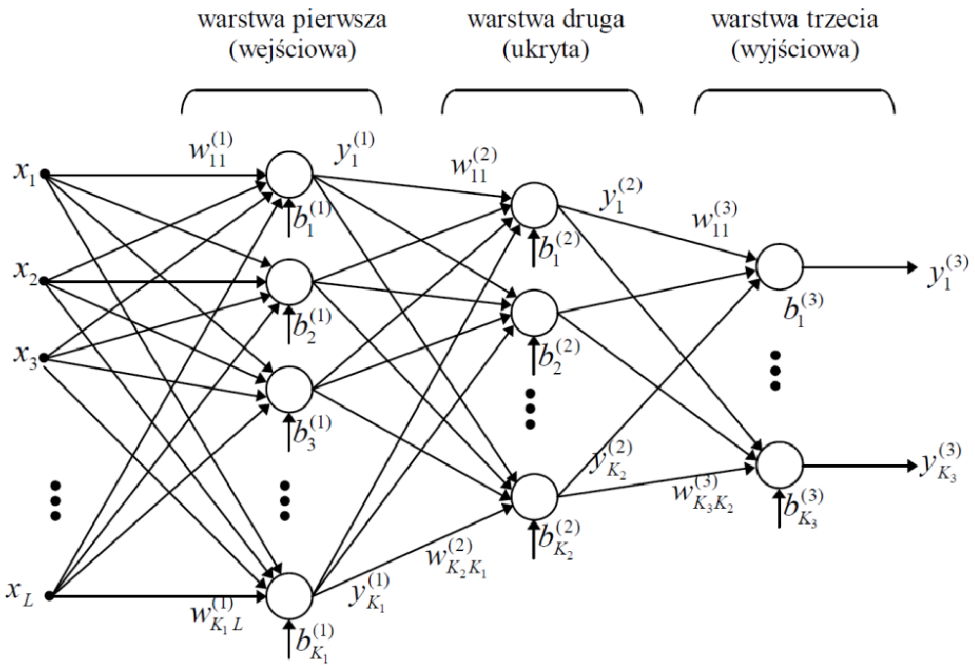
Funkcja sigmoidalna bipolarna:

$$f(x) = \frac{2}{1 + e^{-\beta x}} - 1 \tag{3.6}$$

gdzie parametr β określony jest z reguły w przedziale $[0, 1]$.

3.2. Sieć jednokierunkowa wielowarstwowa

Sztuczną sieć neuronowa uzyskuje się łącząc ze sobą warstwy neuronów. 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
- co najmniej jedną warstwę ukrytą (złożoną z neuronów sigmoidalnych)
- warstwę wyjściową (złożoną z neuronów sigmoidalnych lub liniowych)

Każda warstwa 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. Uczenie sieci algorytmem wstecznej propagacji błędu

Algorytm wstecznej propagacji błędu dominuje wśród metod uczenia sieci jednokierunkowych. Opiera się on na koncepcji poprawiania na każdym kroku procesu uczenia wartości korekty wag na podstawie oceny błędu popełnionego przez każdy neuron podczas uczenia sieci [7].

Do zastosowania algorytmu wstecznej propagacji błędu, wymagane jest, aby funkcje aktywacji neuronów były różniczkowalne, co pozwala 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 η, który również może zmieniać się podczas procesu uczenia sieci. Wagę dla $k+1$ kroku otrzymać można w następujący sposób:

$$w_{ij}(k+1) = w_{ij}(k) - \eta \frac{\partial E}{\partial w_{ij}(k)} \tag{3.12}$$

Zatem zmiany wag obliczane są ze wzoru:

$$\Delta w_{ij} = -\eta \frac{\partial E}{\partial w_{ij}} \tag{3.13}$$

Obliczanie wag neuronów rozpoczyna się od warstwy wyjściowej:

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

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 η 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
- ξ_d - współczynnik zmniejszania wartości współczynnika uczenia
- ξ_i - współczynnik zwiększania wartości współczynnika uczenia

4. Algorytm

Na potrzeby realizacji projektu zaimplementowano w języku Python algorytm pozwalający na uczenie dowolnej sieci neuronowej algorytmem wstecznej propagacji błędu, wraz z przyspieszeniem metodą adaptacyjnego współczynnika uczenia.

Python jest z natury językiem wolnym. Dzieje się tak, ponieważ należy do grupy języków interpretowanych. Z racji tego, implementacja sieci w samym Pythonie byłaby nieefektywna. W związku z tym do obliczeń wykorzystano moduł *numpy*. Biblioteka ta została napisana i skompilowana w języku C, dlatego wykonywanie obliczeń będzie znacznie przyspieszone. Do kroswalidacji użyto modułu *sklearn*.

Algorytm 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):
    """konstruktor obiektu sieci - przyjmuje jako parametr liste
    zawierajaca liczbe wejsc, liczbe neuronow w
    warstwach S1, S2, oraz liczbe neuronow na wyjsciu"""
    def __init__(self, sizes, err=1.04):
        #do ilosci warstw sieci przypisz dlugosc wektora "sizes"
        self.num_layers = len(sizes)
        #przypisanie wektora do zmiennej w obiekcie
        self.sizes = sizes
        #ustawia parametry generatora pseudolosowego
        #aby kolejnym sieciom byly przypisywane takie same wagi
        #i biasy
        np.random.seed(1)
        """inicjalizacja biasow i wag przy pomocy rozkladu
        gaussa N(0,1) dla wszystkich warstw poza zerowa
        (warstwa wejsciowa to nie neurony)"""
        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:])]
        #maksymalny przyrost bledu
        self.max_perf_inc = err

    def feedforward(self, a):
        #zwraca wynik sieci neuronowej dla zestawu danych "a"
        for b, w in zip(self.biases, self.weights):
```

```
30              a = sigmoid(np.dot(w, a)+b)
31          return a
32
33      #liczenie bledu sredniokwadratowego
34      def sse(self,_test_data):
35          error=[pow(np.linalg.norm(self.feedforward(x)-y),2) for
    (x,y) in _test_data]
36          return 0.5*sum(error)
37
38
39      def SGD(self, training_data, epochs, mini_batch_size, eta,
40              test_data,error_goal,inc,dec):
41          """Funkcja odpowiedzialna za proces uczenia.
42          Jako parametry przyjmuje: liste krotek (x, y)
43          gdzie x - wektor danych uczacych,
44          y - oczekiwane wyjscie sieci;
45          maksymalna liczbe epok; rozmiar podzbioru danych;
46          poczatkowy wspolczynnik uczenia; liste krotek (x, y)
47          zawierajacych dane testowe; docelowy koszt;
48          wspolczynnik przyrostu i spadku wspolczynnika uczenia"""
49          #przypisanie do zmiennej ilosci testow
50          n_test = len(test_data)
51          n = len(training_data)
52          #petla dla kazdej epoki
53          for j in range(epochs):
54              #przelosowanie zbioru uczacego
55              random.shuffle(training_data)
56              #generowanie podzbiorow
57              mini_batches = [training_data[k:k+mini_batch_size]
58                      for k in range(0, n, mini_batch_size)]
59              #obliczanie bledu dla starych parametrow
60              old_error=self.sse(test_data)
61              #kopia zapasowa wektorow z biasami i wagami
62              backup_weights = self.weights.copy()
63              backup_biases = self.biases.copy()
64              #aktualizowanie kazdego z podzbiorow
65              for mini_batch in mini_batches:
66                  self.update_mini_batch(mini_batch, eta)
67              #obliczenie bledu dla nowych wartosci bias i wag
68              new_error=self.sse(test_data)
69              #jezeli nowy blad < pozadany koszt
70              if new_error < error_goal:
71                  #obliczanie sprawnosci sieci
72                  #evaluate - liczba poprawnych dopasowan
73                  test=self.evaluate(test_data)
74                  #zamiana na wartosc procentowa
75                  test2=test/n_test*100
76                  print("Epoch {0}: , {1:.2f}%".format(j+1,  test2
    ))
77                  return [j+1, test2]
78              #jezeli nowy blad < starego
79              elif new_error < old_error:
80                  #zwiekszenie wspolczynnika uczenia
81                  eta *= inc
82              #jezeli nowy blad > stary * max przyrost bledu
83              elif new_error > old_error * self.max_perf_inc:
```

```
                #przywrocenie starych biasow i wag
                self.weights = backup_weights
                self.biases = backup_biases
                #zmniejszenie wspolczynnika uczenia
                eta *= dec
            #jezeli uplynela max liczba epok
            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):
        #Funkcja aktualizujaca wagi i biasy poprzez
        #zastosowanie metody gradientowej i wstecznej
        #propagacji dla danego podzioru. Jako parametry
        #przyjmuje: liste krotek (x, y) oraz wspolczynik uczenia
        #generowanie macierzy zerowej dla gradientu biasow i wag
        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:
            #dla kazdej pary (x, y) oblicz przyrost gradientu
            delta_nabla_b, delta_nabla_w = self.backprop(x, y)
            #oblicz nowy 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)]
        #obliczanie nowych wag i biasow
        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):
        """Zwraca krotke (nabla_b, nabla_w) rezprezentujaca
        gradient funkcji kosztu"""
        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
        #lista zawierajaca aktywacje wszystkich neuronow,
        #warstwa po warstwie
        activations = [x]
        # lista pobudzen, warstwa po warstwie
        zs = []
        #obliczanie aktywacji i pobudzen
        for b, w in zip(self.biases, self.weights):
            z = np.dot(w, activation)+b
            zs.append(z)
            activation = sigmoid(z)
            activations.append(activation)
        #obliczanie przyrostu gradientu dla warstwy wyjsciowej
```

```
137         delta = self.cost_derivative(activations[-1], y) * \
138             sigmoid_prime(zs[-1])
139         nabla_b[-1] = delta
140         nabla_w[-1] = np.dot(delta, activations[-2].transpose())
141         #obliczanie przyrostu gradientu dla warstwy ukrytej i
142         #wejsciowej
143         for l in range(2, self.num_layers):
144             z = zs[-l]
145             sp = sigmoid_prime(z)
146             delta = np.dot(self.weights[-l+1].transpose(), delta
    ) * sp
147             nabla_b[-l] = delta
148             nabla_w[-l] = np.dot(delta, activations[-l-1].
    transpose())
149         return (nabla_b, nabla_w)
150
151     def evaluate(self, test_data):
152         """Zwraca liczbe poprawnie dopasowanych rekordow
153         treningowych. Jako argument przyjmuje liste krotek (x,y)
154         z danymi testowymi"""
155         """tworzy liste krotek gdzie x - indeks neuronu ktory
156         posiada najwieksza wartosc, y - oczekiwana klasa"""
157         test_results = [(np.argmax(self.feedforward(x)), np.
    argmax(y))
158                         for (x, y) in test_data]
159         #suma poprawnych dopasowan
160         return sum(int(x == y) for (x, y) in test_results)
161
162     def cost_derivative(self, output_activations, y):
163         #Funkcja zwracajaca wektor
164         #(atywacja neuronow - oczekiwane wyniki)
165         return (output_activations-y)
166
167 def sigmoid(z):
168     """Funkcja sigmoidalna"""
169     return 1.0/(1.0+np.exp(-z))
170
171 def sigmoid_prime(z):
172     """Pochodna z funkcji sigmoidalnej"""
173     return sigmoid(z)*(1-sigmoid(z))
```

Listing 2: Algorytm uczenia 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)	[1,20]
S2 (neurony warstwy II)	[1,20]
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

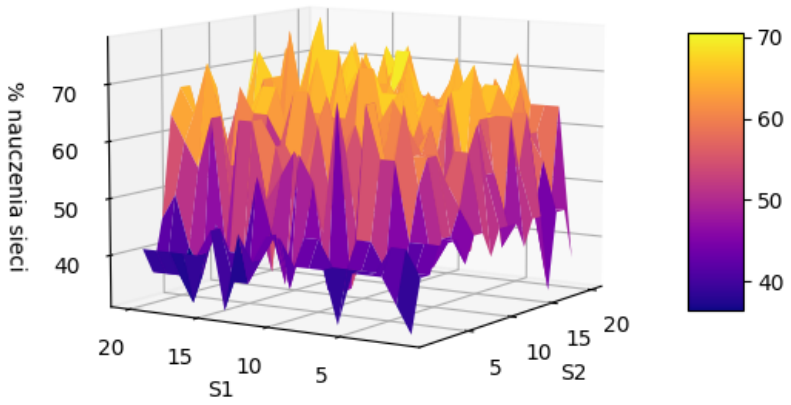
Tabela 5.1: Domyślne wartości parametrów

```
import network
from data import loadData
import pandas as pd
import numpy as np
trainData, testData = loadData()
s1_vec = np.arange(1,20.01,1,dtype=int)
s2_vec = np.arange(1,20.01,1,dtype=int)
results = []
for s1 in s1_vec:
    for s2 in s2_vec:
        result = [s1,s2]
        net = network.Network([9, s1, s2,  6])
        print('S1: {0}, S2: {1}'.format(s1,s2))
        result.extend(net.SGD(trainData,2000,len(trainData),0.1,
   testData,0.25,1.05,0.7))
        results.append(result)
results=pd.DataFrame(results)
results.to_csv('new_results_2.csv',index=None,header=None)
```

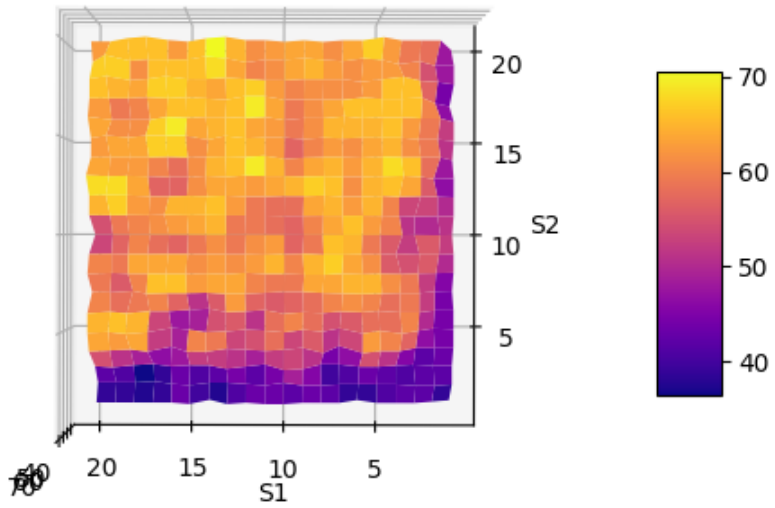
Listing 3: Algorytm realizujący eksperyment 1

5.1.1. Badania dla 1000 epok

Przebadane zostały liczby neuronów w obu warstwach z przedziału od 1 do 20 z krokiem co 1 neuron. Liczba epok wynosiła 1000. Pozostałe parametry ustawiono zgodnie z tabelą 5.1



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

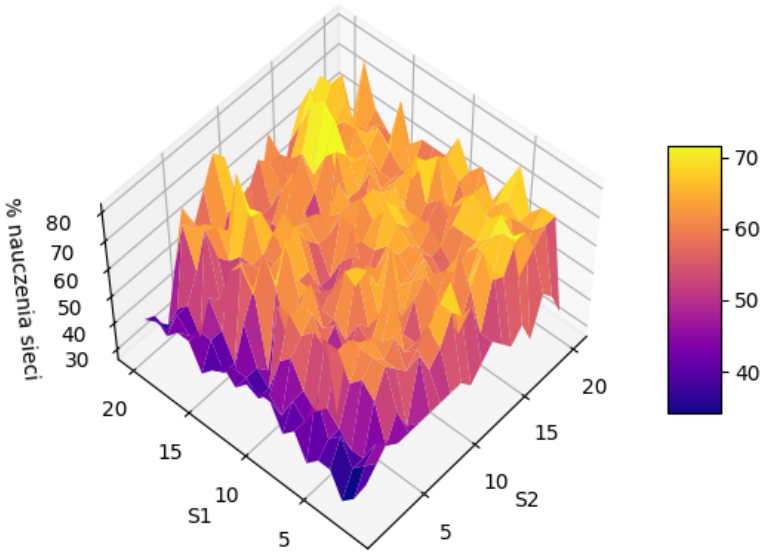


Rysunek 5.4: Wykres wypływu S1, S2 na poprawność klasyfikacji (maks. 1000 epok)

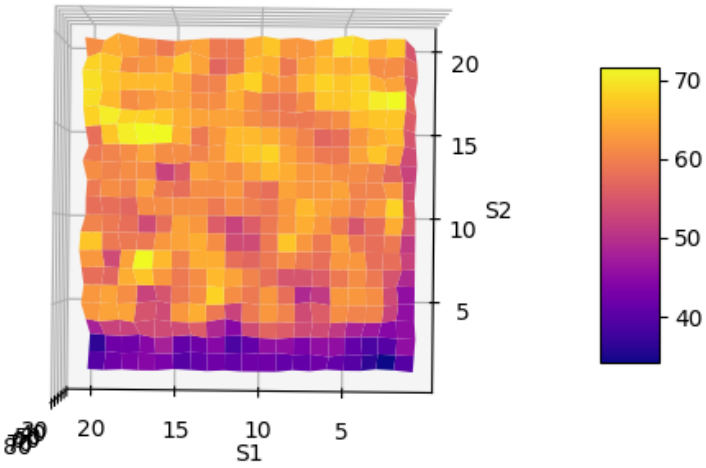
W wyniku tego eksperymentu okazało się, że dla tych parametrów sieć nie przekracza skuteczności równej $77,27\%$. Skuteczność tę osiąga dla 19 neuronów na warstwie I i 19 neuronów na warstwie II oraz dla 16 neuronów na warstwie I i 11 neuronów na warstwie II.

5.1.2. Badania dla 2000 epok

Ze względu na niezadowalające wyniki eksperymentu dla maksmalnej liczby epok równej 1000, liczba ta została zwiększona dwukrotnie. Dalsze eksperymenty przeprowadzono więc dla 2000 epok. Pozostałe parametry pozostały bez zmian.



Rysunek 5.5: Wykres wypływu S1, S2 na poprawność klasyfikacji (maks. 2000 epok)



Rysunek 5.6: Wykres wypływu S1, S2 na poprawność klasyfikacji (maks. 2000 epok)

Sieć osiąga najlepszą skuteczność równą 81,82% dla 17 neuronów na warstwie I i 7 neuronów na warstwie II oraz dla 17 neuronów na warstwie I i 15 neuronów na warstwie II. Został także przeprowadzony eksperyment dla maks. liczby epok równej 5000, jednak nie zmieniło to w żadnym stopniu skuteczności sieci.

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

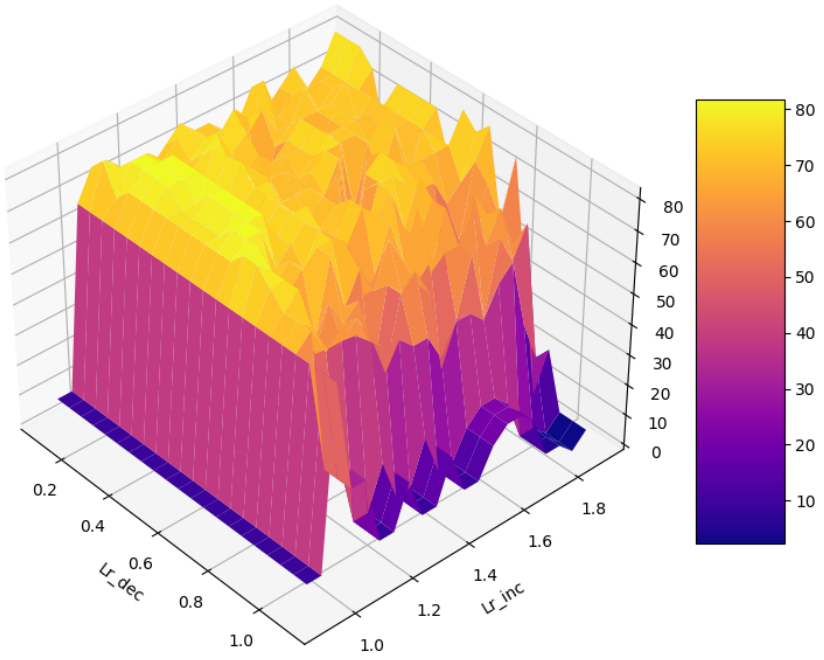
Na potrzeby eksperytmentu 2 przyjęto parametry uczenia określone w tabeli 5.1, zmianie poddane zostały parametry:

- 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

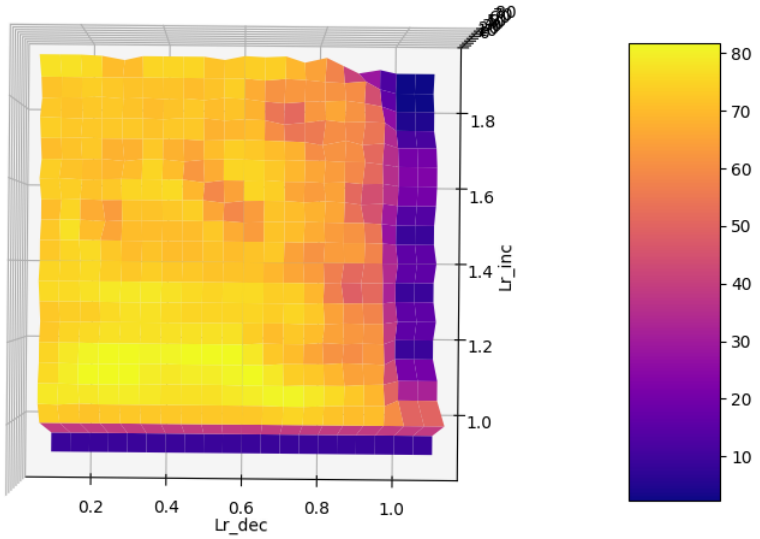
```
import sys
import network
from data import loadData
import pandas as pd
import numpy as np
from tabulate import tabulate
trainData, testData = loadData()
lr_inc_vec=np.arange(0.9,1.91,0.05)
lr_dec_vec=np.arange(0.1,1.11,0.05)
results = []
name=sys.argv[3]
name='result'+name+'.csv'
for lr_inc in lr_inc_vec:
    for lr_dec in lr_dec_vec:
        result = [lr_inc,lr_dec]
        net = network.Network([9, int(sys.argv[1]), int(sys.argv
    [2]), 6])
        result.extend(net.SGD(trainData,2000,len(trainData),0.1,
    testData,0.25,lr_inc,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 $S1 = 17$, $S2 = 7$



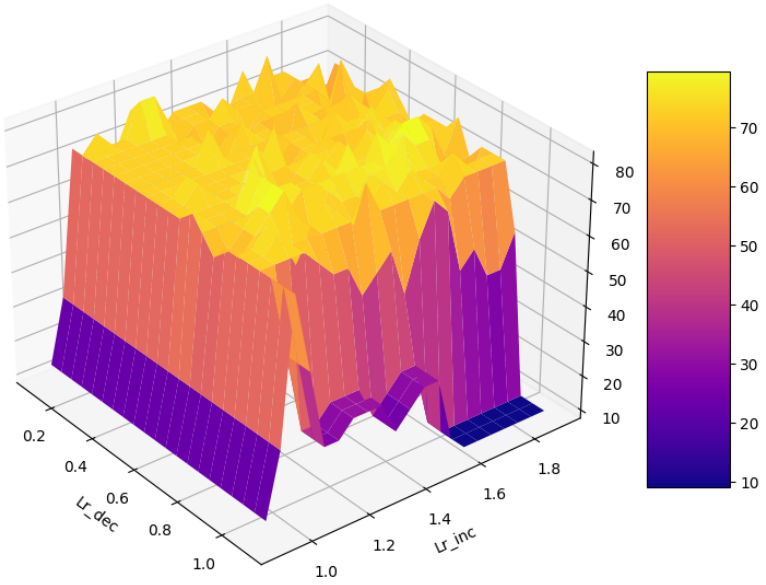
Rysunek 5.7: Zależność poprawności klasyfikacji od lr_{inc} oraz lr_{dec} dla $S1 = 17$, $S2 = 7$



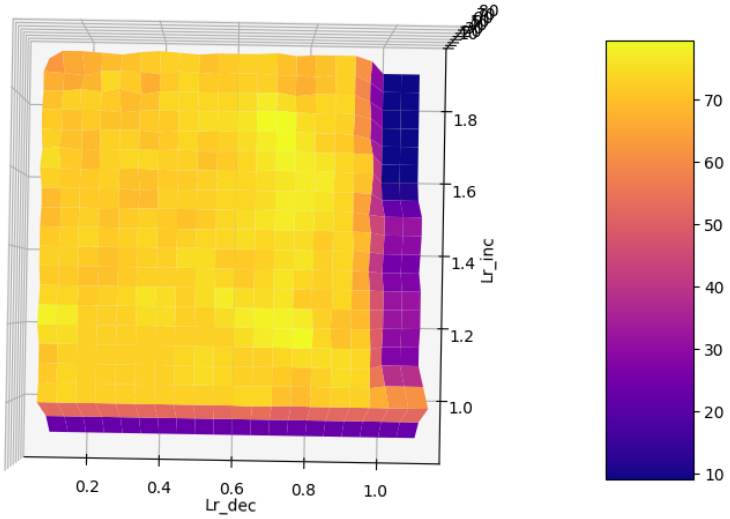
Rysunek 5.8: Zależność poprawności klasyfikacji od lr_{inc} oraz lr_{dec} dla $S1 = 17$, $S2 = 7$

Rysunki 5.7, oraz 5.8, przedstawiają powierzchnię funkcji procentowej wartości poprawności klasyfikacji w zależności od wartości parametru learning rate accelerate oraz learning rate decelerate. Dla S1= 17, S2 = 7 największą poprawność uczenia osiągnięto dla wartości z przedziału $lr_{inc} = [1.0; 1.1]$, oraz $lr_{dec} = [0.2; 0.6]$ - $81,81\%$. Natomiast najgorszy wynik uzyskano gdy lr_{inc} wynosił 0.9 lub lr_{dec} był większy niż 1.

5.2.2. Badania dla $S1 = 17$, $S2 = 15$



Rysunek 5.9: Zależność poprawności klasyfikacji od lr_{inc} oraz lr_{dec} dla $S1 = 17$, $S2 = 15$



Rysunek 5.10: Zależność poprawności klasyfikacji od lr_{inc} oraz lr_{dec} dla S1=17, S2=15

Na rysunkach 5.9, oraz 5.10 można zauważyć, że dla S1= 17, S2 = 15 największą poprawność uczenia osiągnięto dla wartości $lr_{inc} = [1.20; 1.25]$, $lr_{dec} = [0.65; 0.8]$ oraz $lr_{inc} = 1.4$, $lr_{dec} = 0.9$. Natomiast najgorsze wyniki uzyskano gdy lr_{inc} wynosił 0.9 lub lr_{dec} był większy niż 1.

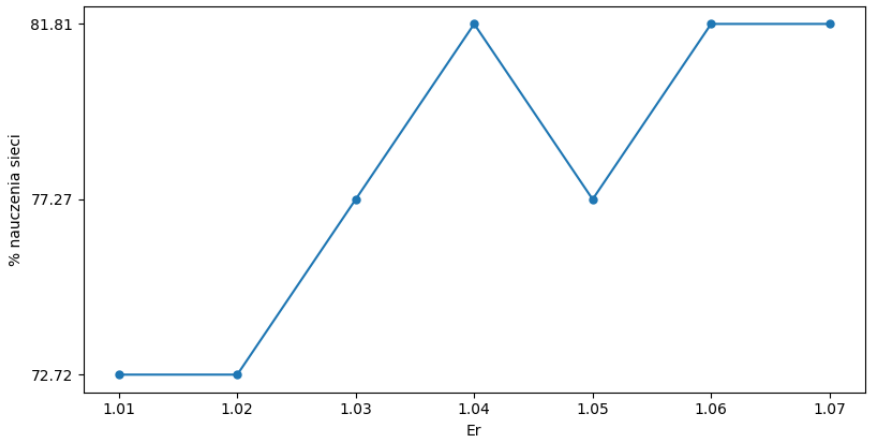
Porównując przedstawione wyżej wykresy i wartości, można wywnioskować, że sieć w parametrach S1= 17, S2 = 7 jest lepiej uwarunkowana dla badanego zestawu danych.

5.3. Eksperyment 3 - badanie parametru er

W kolejnym z eksperymentów zbadano wpływ parametru *er* na procentową wartość nauczenia sieci. Domyślnie parametr ten wynosił 1.04. Na potrzeby eksperymentu zmieniano wartość *er* w zakresie $[1.01; 1.07]$. Badania przeprowadzono dla maksymalnej liczby 2000 epok, $lr_{inc} = 1.05$, $lr_{dec} = 0.7$ oraz parametrów S1 = 17, S2 = 7. Listing 5 przedstawia kod, którego użyto do eksperymentu.

```
import network
from data import loadData
import pandas as pd
import numpy as np
from tabulate import tabulate
trainData, testData = loadData()
er_ratio_vec=np.arange(1.01,1.071,0.01)
results = []
for er in er_ratio_vec:
    result = [er]
    net = network.Network([9, 17, 7,  6],err=er)
    print('Er: {0}'.format(er))
    result.extend(net.SGD(trainData,2000,len(trainData),0.1,
   testData,0.25,1.05,0.7))
    results.append(result)

results=pd.DataFrame(results)
results.to_csv('new_resulty_1.csv',index=None,header=None)
```

Listing 5: Algorytm realizujący eksperyment 3



Rysunek 5.11: Wykres wpływu parametru er na poprawność klasyfikacji

Rysunek 5.11 obrazuje wyniki wykonanego badania. Warstość domyślna parametru $er = 1.04$ jest dobrze dobrana, ponieważ daje najwyższy wynik nauczenia sieci.

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, "Neural 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