Praca projektowa z przedmiotu Sztuczna Inteligencja

Sieć neuronowa uczona algorytmem wstecznej propagacji błędu z przyśpieszeniem metodą adaptacyjnego współczynnika uczenia

Cezary Bober

Rzeszów, 2019

Spis treści

1 Opis projektu

Celem projektu jest stworzenie prostej sieci neuronowej, która nauczy się rozpoznawać rodzaje wina na podstawie kilku jego parametrów. Oprócz zbudowania sieci należało także wykonać serię eksperymentów służących zbadaniu optymalnych parametrów uczenia sieci dla podanego zestawu danych.

Do stworzenia zadanej sieci neuronowej wykorzystany został język programowania *Python* wraz z biblioteką *Numpy* dodającą wsparcie dla m.in. wielowymiarowych macierzy.

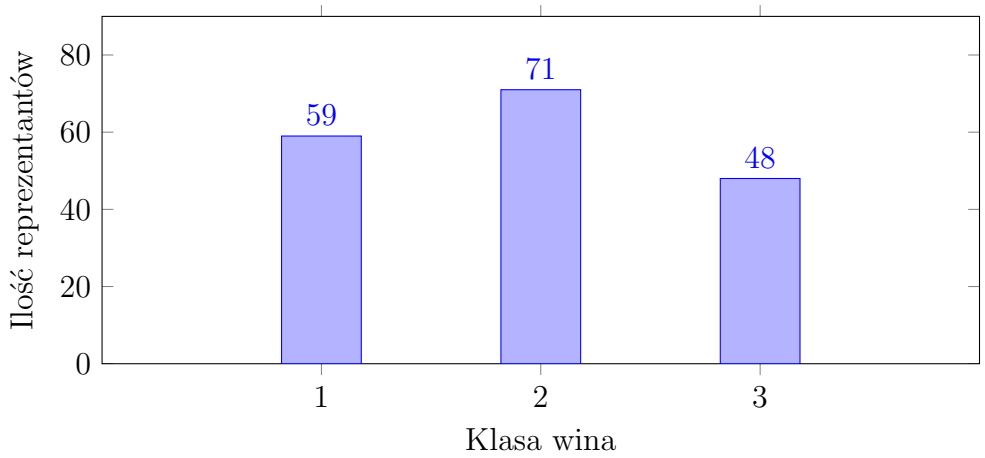
1.1 Zestaw danych

Dane, na których uczyła się sieć pochodzą z zestawu *Wine* [1]. Jest to zestaw zawierający 178 rekordów opisujących wyniki analizy chemicznej 3 różnych odmian win z pewnego regionu Włoch. Każdy rekord składa się z 14 kolumn opisujących poszczególne parametry:

1. Class identifier — klasa identyfikująca rodzaj wina
2. Alcohol — procentowa zawartość alkoholu w danym winie
3. Malic acid — zawartość kwasu jabłkowego w winie
4. Ash — zawarość popiołu w winie
5. Alkalinity of ash — zasadowość popiołu
6. Magnesium — zawartość magnezu w winie
7. Total phenols — zawartość fenoli w winie
8. Flavanoids — zawartość flawonoidów w winie
9. Nonflavanoid phenols — zawartość fenoli nieflawonoidowych
10. Proanthocyanidins — zawartość proantocyjanidyn
11. Color intensity — intensywność koloru wina
12. Hue — barwa wina
13. OD280/OD315 — zawartość OD280/OD315 w rozcieńczonym winie
14. Proline — zawartość proliny w winie

Wszystkie parametry (oprócz klasy wina, która przyjmuje wartości ze zbioru $\{1, 2, 3\}$) mają wartości ciągłe. Klasa wina jest wartością aproksymowaną przez sieć, a pozostałe parametry tworzą wektor danych wejściowych. Zestaw nie zawiera brakujących lub niepotrzebnych danych w związku z czym do nauki wykorzystany został w całości.

Przed przystąpieniem do uczenia sieci zestaw został poddany odpowiedniemu przygotowaniu. Dane zostały podzielone na część testową zawierającą 20% wszystkich rekordów oraz część treningową zawierającą pozostałe 80%. Rekordy są przydzielane do tych części losowo.



Rysunek 1: Rozkład poszczególnych klas wina w zestawie

Z analizy danych przedstawionej na rysunku 1.1 wynika, że poszczególne klasy wina są reprezentowane przez zbliżoną ilość rekordów dzięki czemu po rozlosowaniu wszystkich danych każda z części jest w pełni reprezentatywna.

Dodatkowo, dane wejściowe (wszytko oprócz klasy wina) zostały znormalizowane do zakresu $[-1, 1]$. Do normalizacji danych wykorzystano algorytm *min-max*, który pozwala znormalizować dane do wskazanego zakresu.

$$X' = \frac{X - X_{min}}{X_{max} - X_{min}} \cdot (Y_{max} - Y_{min}) + Y_{min} \tag{1.1}$$

gdzie Y_{max} i Y_{min} są granicami zakresu do którego chcemy znormalizować dane.

2 Działanie sieci neuronowej

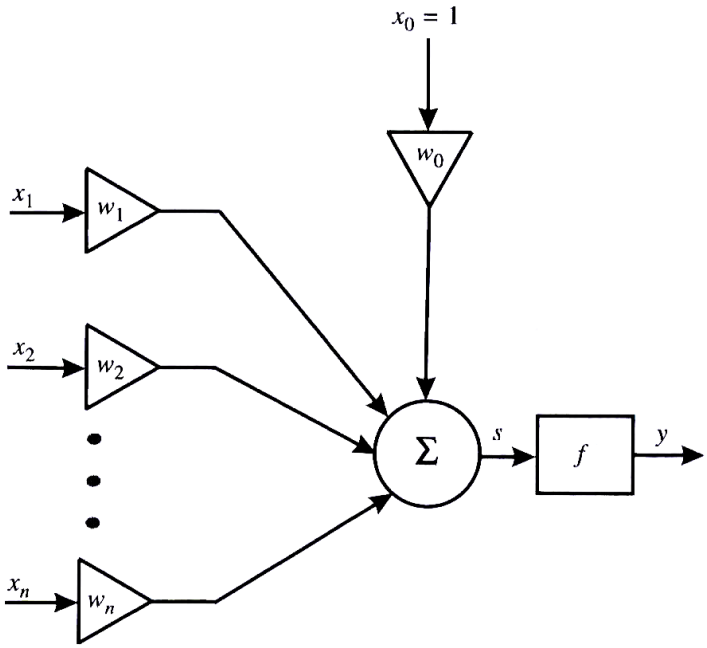
2.1 Działanie neuronu

Komputerowe sieci neuronowe powstały na bazie wiedzy o działaniu systemu nerwowego istot żywych w otaczającym nas świecie. Poznanie mechanizmów działania poszczególnych neuronów jest istotne dla zrozumienia procesów przetwarzania informacji zachodzących w sieciach neuronowych.

Neuron posiada ciało, z którego wyrastają liczne wypustki łączące się z innymi neuronami. Wypustki dzielimy na liczne, cienkie *dendryty* oraz oraz pojedyńczy gruby *akson*. Sygnały wejściowe doprowadzane są do komórki za pośrednictwem synaps, zaś sygnał wyjściowy odprowadzany jest za pomocą aksonu. Transmisja

sygnałów wewnątrz systemu nerwowego jest bardzo skomplikowanym procesem chemiczno-elektrycznym [4].

2.2 Model neuronu sigmoidalnego



Rysunek 2: Model neuronu (Źródło: [5])

$x_1, \ldots, x_n$ — sygnały wejściowe neuronu

$w_1, \ldots, w_n$ — wagi neuronu

w_0 — próg (bias) neuronu

f — funkcja aktywacji neuronu

s — łączne pobudzenie neuronu

y — wartość wyjściowa neuronu

Działanie takiego neuronu jest bardzo proste. Najpierw sygnały wejściowe $x_0, x_1, \ldots, x_n$ zostają pomnożone przez odpowiadające im wagi $w_0, w_1, \ldots, w_n$. Otrzymane w ten sposób wartości należy następnie zsumować. W wyniku powstaje sygnał s nazywany łącznym pobudzeniem neuronu [2] odzwierciedlający działanie części liniowej neuronu. Sygnał ten jest poddawany działaniu *funkcji aktywacji*. Zakładamy, że wartość sygnału x_0 jest równa 1, natomiast wagę w_0 nazywa się progiem (ang. *bias*) [5].

$$y(t) = f(s) = f\left(\sum_{i=0}^{n} w_i x_i\right) \tag{2.1}$$

2.3 Funkcja aktywacji

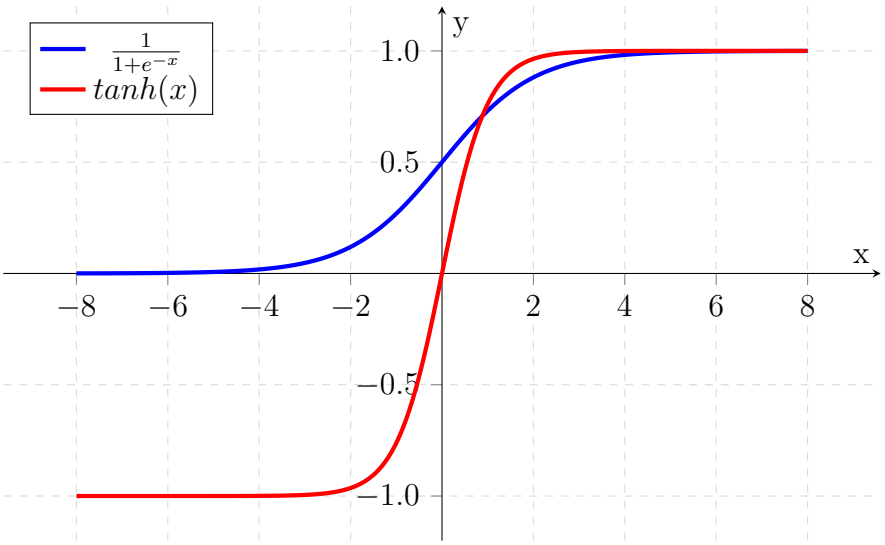
Funkcja aktywacji jest wykorzystywana do liczenia sygnału wyjściowego neuronu na podstawie jego pobudzenia. W neuronie sigmoidalnym wykorzystywana jest funkcja sigmoidalna dana wzorem:

$$f(x) = \frac{1}{1 + e^{-\beta x}} \tag{2.2}$$

oraz w wersji bipolarnej:

$$f(x) = tanh(\beta x) = \frac{e^{\beta x} - e^{-\beta x}}{e^{\beta x} + e^{-\beta x}} \tag{2.3}$$

Różnią się one między sobą wartościami, które zwracają. Funkcja (2.2) zwraca wartości z zakresu $[0, 1]$ natomiast (2.3) z zakresu $[-1, 1]$.



Rysunek 3: Wykres funkcji sigmoidalnej unipolarnej (niebieski) i bipolarnej (czerwony)

Zmieniając parametr β możemy modyfikować kształt funkcji sprawiając, aby był bardziej łagodny lub stromy. Przy odpowiednim ustawieniu tego parametru funkcja sigmoidalna ma charakter progowy.

Dużą zaletą funkcji sigmoidalnej jest to, iż jest ona różniczkowalna, oraz łatwo można policzyć jej pochodną [3], która ma postać:

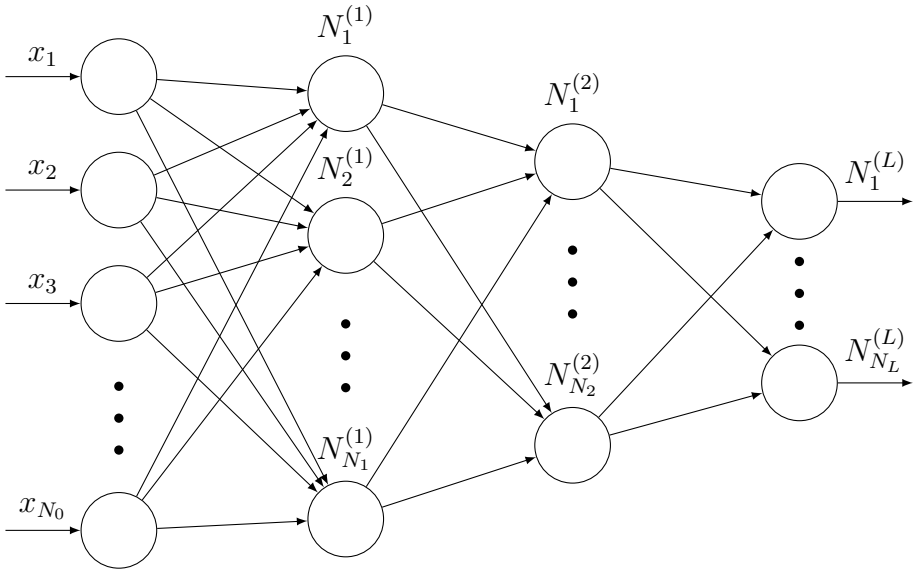
$$f'(x) = f(x)(1 - f(x)) \tag{2.4}$$

lub dla funkcji bipolarnej:

$$f'(x) = (1 - f^2(x)) \tag{2.5}$$

2.4 Budowa sieci neuronowej

Pojedyncze neurony są w stanie uczyć się rozwiązywania prostych zadań, ale już przykładowo naśladowanie funkcji XOR jest zbyt trudnym zadaniem dla pojedynczego neuronu. Z tego powodu neurony łączy się w struktury zwane warstwami. Sieć stworzona z wielu warstw jest już w stanie rozwiązywać bardziej skomplikowane zadania.



Rysunek 4: Schemat sieci neuronowej wielowarstwowej

Przedstawiona na rysunku 4 sieć składa się z warstwy wejściowej, następnie 2 warstw ukrytych oraz warstwy wyjściowej. Omawiana sieć ma N_0 wejść, na które podawane są sygnały $x_1(t), \ldots, x_{N_0}(t)$ zapisywane w postaci wektora:

$$\mathbf{x} = [x_1, \ldots, x_{N_0}]^T \qquad t = 1, 2, \ldots \tag{2.6}$$

Każdy neuron N_i^k ma przypisany wektor z wagami:

$$\mathbf{w}_i^{(k)} = [w_{i,0}^{(k)}, \ldots, w_{i,N_{k-1}}^{(k)}]^T \qquad k = 1, 2, \ldots, L \quad i = 1, \ldots, N_k \tag{2.7}$$

Sygnał wyjściowy i-tego neuronu w k-tej warstwie jest oznaczony jako:

$$y_i^{(k)} = f^{(k)}(s_i^{(k)}) \tag{2.8}$$

przy czym:

$$s_i^{(k)} = \sum_{j=0}^{N_{k-1}} w_{ij}^{(k)} x_j^{(k)} \tag{2.9}$$

Sygnały wyjściowe z ostatniej warstwy są jednocześnie sygnałami wyjściowymi całej sieci. Po każdej iteracji sieci są one porównywane z *sygnałami wzorcowymi* sieci [5]

$$d_1^{(L)}, d_2^{(L)}, \ldots, d_{N_L}^{(L)} \tag{2.10}$$

Dla poszczególnych neuronów w warstwach wyjście będzie miało postać:

$$y_m^{(1)} = f^{(1)}(s_m^{(1)}) = f^{(1)}\Big(\sum_{p=1}^{N_0} x_p \cdot w_{mp}^{(1)} + w_{m0}^{(1)}\Big) \tag{2.11}$$

$$\begin{aligned} y_j^{(2)} = f^{(2)}(s_j^{(2)}) = f^{(2)}\Big(\sum_{m=1}^{N_1} f^{(1)}(s_m^{(1)}) \cdot w_{jm}^{(2)} + w_{j0}^{(2)}\Big) = \\ = f^{(2)}\Big(\sum_{m=1}^{N_1} f^{(1)}\Big(\sum_{p=1}^{N_0} x_p \cdot w_{mp}^{(1)} + w_{m0}^{(1)}\Big) \cdot w_{jm}^{(2)} + w_{j0}^{(2)}\Big) \end{aligned} \tag{2.12}$$

$$\begin{aligned} y_i^{(3)} = f^{(3)}(s_i^{(3)}) = f^{(3)}\Big(\sum_{j=1}^{N_2} f^{(2)}(s_j^{(2)}) \cdot w_{ij}^{(3)} + w_{i0}^{(3)}\Big) = \\ f^{(3)}\Big(\sum_{j=1}^{N_2} f^{(2)}\Big(\sum_{m=1}^{N_1} f^{(1)}\Big(\sum_{p=1}^{N_0} x_p \cdot w_{mp}^{(1)} + w_{m0}^{(1)}\Big) \cdot w_{jm}^{(2)} + w_{j0}^{(2)}\Big) \cdot w_{ij}^{(3)} + w_{i0}^{(3)}\Big) \end{aligned} \tag{2.13}$$

Błąd sieci liczymy jako:

$$Q = \sum_{m=1}^{N_L} \Big(d_m^{(L)} - y_m^{(L)}\Big)^2 \tag{2.14}$$

2.5 Wsteczna propagacja błędu

Podczas uczenia sieci jednowarstwowych w łatwy sposób można zdefiniować błąd na wyjściu sieci ponieważ znamy wartość otrzymaną oraz wartość, która powinna być na wyjściu. Problem pojawia się jednak przy tworzeniu sieci wielowarstwowych. Nie znamy bowiem wartości jakie powinny być na wyjściu poszczególnych neuronów w warstwach ukrytych. W celu rozwiązania tego problemu wymyślono *metodę wstecznej propagacji błędów* [5].

W metodzie tej do określania błędów na poszczególnych warstwach wykorzystuje się gradienty. Cały proces zaczyna się od końca sieci, stąd nazwa *wsteczna propagacja*. Do obliczenia zmiany Δw_{ij} musimy policzyć pochodną cząstkową z funkcji kosztu względem wagi w_{ij}.

$$\Delta w_{ij} = -\eta \frac{\partial Q}{\partial w_{ij}} \tag{2.15}$$

gdzie η to współczynnik uczenia sieci. Nie wiemy jak bezpośrednio policzyć taką pochodną dlatego musimy rozpisać ją tak, abyśmy otrzymali składniki, które umiemy policzyć.

Przykładowo dla przedstawionej na rysunku 4 sieci, aby obliczyć zmianę wag dla warstwy wyjściowej rozpiszemy [9]:

$$\frac{\partial Q}{\partial w_{ij}^{(L)}} = \frac{\partial Q}{\partial f^{(L)}} \frac{\partial f^{(L)}(s_i^{(L)})}{\partial s_i^{(L)}} \frac{\partial s_i^{(L)}}{\partial w_{ij}^{(L)}} \tag{2.16}$$

Możemy zauważyć, że

$$\frac{\partial Q}{\partial f^{(L)}} = 2(y_i^{(L)} - d_i^{(L)}) \tag{2.17}$$

oraz

$$\frac{\partial s_i^{(L)}}{\partial w_{ij}^{(L)}} = y_j^{(2)} \tag{2.18}$$

W związku z czym możemy równanie 2.16 zapisać jako:

$$\frac{\partial Q}{\partial w_{ij}^{(L)}} = 2(y_i^{(L)} - d_i^{(L)}) \frac{\partial f^{(L)}(s_i^{(L)})}{\partial s_i^{(L)}} y_j^{(2)} \tag{2.19}$$

Następnie postępujemy w analogiczny sposób dla warstwy 2:

$$\frac{\partial Q}{\partial w_{jm}^{(2)}} = \frac{\partial Q}{\partial f^{(L)}} \frac{\partial f^{(L)}(s_i^{(L)})}{\partial s_i^{(L)}} \frac{\partial s_i^{(L)}}{\partial f^{(2)}} \frac{\partial f^{(2)}(s_j^{(2)})}{\partial s_j^{(2)}} \frac{\partial s_j^{(2)}}{\partial w_{jm}^{(2)}} \tag{2.20}$$

$$\frac{\partial Q}{\partial w_{jm}^{(2)}} = \sum_{i=1}^{N_L} 2(y_i^{(L)} - d_i^{(L)}) \frac{\partial f^{(L)}(s_i^{(L)})}{\partial s_i^{(L)}} w_{ij}^{(L)} \frac{\partial f^{(2)}(s_j^{(2)})}{\partial s_j^{(2)}} y_m^{(1)} \tag{2.21}$$

oraz dla warstwy 1:

$$\frac{\partial Q}{\partial w_{mp}^{(1)}} = \frac{\partial Q}{\partial f^{(L)}} \frac{\partial f^{(L)}(s_i^{(L)})}{\partial s_i^{(L)}} \frac{\partial s_i^{(L)}}{\partial f^{(2)}} \frac{\partial f^{(2)}(s_j^{(2)})}{\partial s_j^{(2)}} \frac{\partial s_j^{(2)}}{\partial f^{(1)}} \frac{\partial f^{(1)}(s_m^{(1)})}{\partial s_m^{(1)}} \frac{\partial s_m^{(1)}}{\partial w_{mp}^{(1)}} \tag{2.22}$$

$$\frac{\partial Q}{\partial w_{mp}^{(1)}} = \sum_{i=1}^{N_L} 2(y_i^{(L)} - d_i^{(L)}) \frac{\partial f^{(L)}(s_i^{(L)})}{\partial s_i^{(L)}} \sum_{j=1}^{N_2} w_{ij}^{(L)} \frac{\partial f^{(2)}(s_j^{(2)})}{\partial s_j^{(2)}} w_{jm}^{(2)} \frac{\partial f^{(1)}(s_m^{(1)})}{\partial s_m^{(1)}} x_p \tag{2.23}$$

W ten sposób możemy policzyć zmianę wag dla dowolnie głębokiej sieci neuronowej.

W książce [5] przedstawiona została pewna modyfikacja tych wzorów, dzięki której implementacja tego algorytmu w kodzie jest łatwiejsza. Podczas liczenia zmiany wagi wprowadzany jest $\varepsilon_i^{(k)}$ w postaci:

$$\varepsilon_i^{(k)} = \sum_{m=1}^{N_{k+1}} \delta_m^{(k+1)} w_{mi}^{(k+1)} \qquad k = 1, \ldots, L-1 \tag{2.24}$$

następnie korzystając z obliczonej w 2.24 wartości liczymy $\delta_i^{(k)}$:

$$\delta_i^{(k)} = \varepsilon_i^{(k)} f'(s_i^{(k)}) \tag{2.25}$$

Zmiana wagi jest wtedy wyrażona jako:

$$\Delta w_{ij}^{(k)} = 2\eta \delta_i^{(k)} x_j^{(k)} \tag{2.26}$$

Od wzoru 2.15 różni się podstawieniem:

$$\delta_i^{(k)} = -\frac{1}{2} \frac{\partial Q}{\partial s_i^{(k)}} \tag{2.27}$$

$$\frac{\partial Q}{\partial w_{ij}^{(k)}} = -2\delta_i^{(k)} x_j^{(k)} \tag{2.28}$$

2.6 Adaptacyjny współczynnik uczenia

W trakcie uczenia się sieci ważne jest aby współczynnik uczenia nie był zbyt duży aby nie pominąć minimum funkcji kosztu. Jednakże przy zbyt niskim współczynniku proces uczenia może trwać bardzo długo. Rozwiązaniem tego problemu jest *adaptacyjny współczynnik uczenia*.

Podczas uczenia sieci należy obserwować błąd na wyjściu sieci. Jeżeli błąd w stosunku do poprzedniej chwili czasu wzrósł w sposób istotny

$$err(t+1) > err\_ratio \cdot err(t) \tag{2.29}$$

to nowo wyliczone wagi oraz biasy zostają odrzucone, a współczynnik uczenia zostaje pomniejszony o odpowiednią wartość:

$$\eta(t+1) = \eta(t) \cdot lr\_dec \tag{2.30}$$

Natomiast w przypadku gdy błąd zmalał w stosunku do poprzedniej iteracji

$$err(t+1) < err(t) \tag{2.31}$$

to współczynnik uczenia jest zwiększany

$$\eta(t+1) = \eta(t) \cdot lr\_inc \tag{2.32}$$

W innych przypadkach współczynnik uczenia zostaje bez zmian.

Dzięki zastosowaniu tej metody sieć będzie *przyspieszała* i *zwalniała* w odpowiednich momentach dzięki czemu proces nauczania sieci będzie krótszy.

Przykładowo w programie *MATLAB* standardowo przyjęte są wartości: $err\_ratio = 1.04$, $lr\_dec = 0.7$ oraz $lr\_inc = 1.05$. W rezultacie, jeżeli błąd zwiększy się o 4% to współczynnik uczenia zostanie zmniejszony o 30%, a gdy błąd zmniejszy się to współczynnik uczenia wzrośnie o 5% [6].

3 Realizacja sieci w języku Python

Opisana powyżej sieć została zaimplementowana w języku *Python*. Oprócz standardowej biblioteki program wykorzystuje pakiet *Numpy* ułatwiający pracę na macierzach oraz bibliotekę *Pyplot* do tworzenia wykresów. Całość kodu (oprócz funkcji do inicjalizacji wag i biasów) została napisana *od zera*.

3.1 Kod programu

Uwaga: Z poniższego kodu wycięte zostały fragmenty nie mające związku z prezentowanym algorytmem tj. wypisywanie tekstu do konsoli, tworzenie wykresu, zapisywanie modelu itp. w celu zachowania przejrzystości kodu.

```
def normalize(data, min_v=-1, max_v=1):
    for i in range(data.shape[0]):
        data[i] = ((data[i] - data[i].min()) / (data[i].max() -
        ↪  data[i].min())) * (max_v - min_v) + min_v
    return data
```

Funkcja `normalize()` służy do normalizacji danych w macierzy dwuwymiarowej do wybranego zakresu.

```
def load_wine(test_count=20, sort_training=False):
    with open('wine.csv') as f:
        wines = np.array([list(map(float, x.strip().split(',')))
        ↪  for x in f])
        np.random.shuffle(wines)
        a = int((wines.shape[0] / 100) * test_count)
        wines[:, 1:] = normalize(wines[:, 1:].T).T

        test_wines = wines[:a]
        test_wines = test_wines[test_wines[:,0].argsort()]
        testing_data = (test_wines[:, 1:], test_wines[:, 0])

        wines = wines[a:]

        if sort_training:
            wines = wines[wines[:,0].argsort()]

        learning_data = (wines[:, 1:], wines[:, 0])

    return (learning_data, testing_data)
```

Wczytywanie danych odbywa się za pomocą funkcji `load_wine()`. Na początku dane są ładowane z pliku pobranego ze strony [1]. Następnie zostają one wczytane do tablicy jako liczby zmiennoprzecinkowe. Po wczytaniu kolejność rekordów jest mieszana, a zestaw jest normalizowany do zakresu $[-1, 1]$ korzystając z funkcji `normalize()`. Po normalizacji zestaw jest dzielony na dwie części (domyślnie na 20% i 80%). Jeżeli zmienna `sort_training` jest ustawiona na `True` to część treningowa danych jest sortowana według klasy.

```
class NeuralNetwork:
    def __init__(self, lr, epochs, layers, err_ratio, lr_inc,
    ↪  lr_dec, goal):
        self.lr = lr
        self.epochs = epochs
        self.layers = layers
        self.err_ratio = err_ratio
```

```
        self.lr_inc = lr_inc
        self.lr_dec = lr_dec
        self.goal = goal
        self.weights = None
```

Sieć zaimplementowana jest jako klasa `NeuralNetwork`. Podczas jej inicjalizacji odpowiednie pola ustawiane są na wartości przekazane w konstruktorze.

```
def feed_training_data(self, P, T):
    self.P = P
    self.T = T

def feed_test_data(self, P, T):
    self.test_P = P
    self.test_T = T
```

Funkcje `feed_training_data()` oraz `feed_test_data()` inicjalizują dane wykorzystywane przez sieć do uczenia i testowania.

```
def activation(self, x, derivative=False, beta=1):
    return np.tanh(beta * x) if not derivative else 1 -
    ↪  np.tanh(beta * x)**2

def linear(self, x, derivative=False):
    return 1 if derivative else x
```

Funkcja `activation()` w zależności od podanych parametrów zwraca obliczoną wartość funkcji sigmoidalnej bipolarnej dla danego argumentu lub wartość pochodnej. Funkcja `linear()` działa na tej samej zasadzie lecz dla funkcji liniowej.

```
def initnw(layer_size, input_size):
    beta = 0.7 * (layer_size ** (1.0 / input_size))
    w_rand = np.random.rand(layer_size, input_size) * 2 - 1
    w_rand = np.sqrt(1.0 /
    ↪  np.square(w_rand).sum(axis=1).reshape(layer_size, 1)) *
    ↪  w_rand

    w = beta * w_rand
    b = beta * np.linspace(-1, 1, layer_size) * np.sign(w[:, 0])

    return w, b
```

Funkcja `initnw()` jest zmodyfikowaną wersją kodu z biblioteki *NeuroLab* [7]. Służy ona do generowania współczynników wagowych oraz biasów zgodnie z postulatami

zawartymi w [8]. Na początku obliczany jest współczynnik szerokości przedziału, następnie wagi inicjowane są jako liczby z przedziału [-1, 1]. W kolejnym kroku liczona jest norma Euklidesowa, przez którą wraz z obliczonym współczynnikiem zostają przemnożone wagi. Dzięki zastosowaniu tego algorytmu wagi dalej pozostają w pewnym stopniu losowe, ale są równomiernie rozłożone względem danych wejściowych.

```
def init_weights_and_biases(self):
    self.weights, self.biases = [], []
    sizes = [self.P.shape[1], *self.layers]

    for i in range(1, len(sizes)):
        w, b = initnw(sizes[i], sizes[i-1])
        self.weights.append(w)
        self.biases.append(b)

    self.weights.append(np.random.rand(1, sizes[-1]))
    self.biases.append(np.random.rand(1))
```

W funkcji `init_weights_and_biases()` inicjalizowane są wagi generowane przy użyciu funkcji `initnw()`. Jedynie w ostatniej (wyjściowej) warstwie generowane dane korzystają z klasycznej funkcji `rand()`.

Zmienna `self.weights` jest listą zawierającą tyle elementów ile warstw ma sieć (nie licząc warstwy wejściowej). W każdym z nich znajduje się tablica o rozmiarze $M \times N$ gdzie M oznacza liczbę neuronów w warstwie, a N liczbę neuronów w warstwie poprzedzającej.

`self.biases` jest natomiast listą zawierającą tyle elementów ile warstw ma sieć, gdzie każdy element jest tablicą M-elementową, w której M oznacza liczbę neuronów w warstwie.

```
def forward(self, x):
    y, sum_inputs = [x], []
    for i in range(len(self.layers) + 1):
        f = self.linear if i == len(self.layers) else
        ↪  self.activation
        s = [np.dot(y[i], w) for w in self.weights[i]] +
        ↪  self.biases[i]
        y.append(f(s))
        sum_inputs.append(s)
    return y, sum_inputs
```

Przejście do przodu przez całą sieć realizowane jest przez funkcję `forward()`. Wewnątrz pętli, dla każdej warstwy wybierany jest rodzaj używanej funkcji aktywacji, następnie liczone jest całkowite pobudzenie neuronów korzystając z funkcji do liczenia iloczynu skalarnego macierzy. Funkcja na koniec oblicza wyjścia neuronów, zwraca listę zawierającą wyjścia neuronów oraz listę ze wszystkimi pobudzeniami.

```
def predict(self, x):
    return self.forward(x)[0][-1][0]
```

Funkcja `predict()` przyjmuje tablicę z danymi dotyczącymi jednego wina i zwraca jedną liczbę, którą jest wyjście z sieci.

```
def test(self, P, T):
    prediction = [self.predict(x) for x in P]
    error = np.array([d - y for y, d in zip(prediction, T)])
    cost = (error**2).sum()
    pk = int(((error**2) < 0.25).sum() / len(error) * 100)
    return prediction, cost, pk
```

Funkcja `test()` służy do przetestowania sieci na podanych w argumentach danych. Przeprowadzany jest test, obliczany koszt oraz poprawność klasyfikacji wyrażona w procentach. Na koniec wyjścia z sieci oraz obliczone wskaźniki są zwracane.

```
def errors(self, d, y, sum_inputs):
    delta = [d - y[-1]]
    for k in range(len(self.layers), 0, -1):
        epsilon = [np.dot(delta[0], w) for w in self.weights[k].T]
        delta.insert(0, np.array(epsilon *
        ↪  self.activation(sum_inputs[k-1], True)))
    return delta
```

Funkcja `errors()` odpowiedzialna jest za obliczenie odpowiednich wartości do aktualizacji wag wykorzystując metodę wstecznej propagacji błędów. W każdej iteracji pętli (która przechodzi od warstwy wyjściowej poprzez wszystkie warstwy ukryte) obliczany jest błąd warstwy, który dla pojedynczego neuronu ma postać jak w 2.24 oraz $\delta_i^{(k)}$. Na koniec obliczone delty są zwracane.

```
def update_weights_and_biases(self, delta, x):
    for i in range(len(self.layers) + 1):
        for j in range(len(self.weights[i])):
            factor = 2 * self.lr * delta[i][j]
            self.weights[i][j] += (factor * x[i])
            self.biases[i][j] += (factor * 1)
```

W funkcji `update_weights_and_biases()` następuje aktualizacja wag oraz biasów neuronów. Wykorzystywana jest $\delta_i^{(k)}$ policzona w funkcji `errors()`.

```
def update_learning_rate(self, cost):
    if len(self.costs) > 1:
        if cost > self.costs[-1] * self.err_ratio:
            self.lr = max(1e-10, self.lr * self.lr_dec)
            self.weights = self.last_weights
            self.biases = self.last_biases
            return False
        elif cost < self.costs[-1]:
            self.lr = min(1 - 1e-10, self.lr * self.lr_inc)
    return True
```

Algorytm adaptacyjnego współczynnika uczenia opisany w sekcji 2.6 jest realizowany za pomocą funkcji `update_learning_rate()`. Użyte funkcje `max()` i `min()` służą zabezpieczeniu aby współczynnik uczenia nie wyszedł poza zakres (0, 1).

```
def start_learning(self):
    self.init_weights_and_biases()
    self.costs, self.pks = [], []
    self.last_weights, self.last_biases = [], []

    for epoch in range(self.epochs):
        for x, d in zip(self.P, self.T):
            y, sum_inputs = self.forward(x)
            delta = self.errors(d, y, sum_inputs)
            self.update_weights_and_biases(delta, y)

        prediction, cost, pk = self.test(self.test_P, self.test_T)

        if self.update_learning_rate(cost):
            self.costs.append(cost)
            self.pks.append(pk)
        else:
            self.costs.append(self.costs[-1])
            self.pks.append(self.pks[-1])

        if cost <= self.goal or pk == 100:
            break

        self.last_weights = self.weights
        self.last_biases = self.biases

    return self.costs
```

Metoda `start_learning()` rozpoczyna proces uczenia się sieci. Na początku inicjowane są wagi i biasy, następnie uruchamiana jest pętla wykonująca się przez zadeklarowaną ilość epok oraz pętla podająca kolejne rekordy danych.

W każdej iteracji liczone jest wyjście sieci oraz łączne pobudzenie neuronów, następnie liczone są błędy poszczególnych neuronów po czym aktualizowane są wagi oraz biasy. Podejście to nosi nazwę *przyrostowego uaktualniania wag* [5].

Po przejściu każdej epoki sieć jest testowana a wyniki zapisywane w celu późniejszego ich przedstawienia na wykresie. Następnie następuje aktualizacja współczynnika nauczania. Jeżeli koszt spadł poniżej zadanego progu lub sieć osiągnęła 100% poprawności klasyfikacji to następuje przerwanie procesu nauczania w przeciwnym razie rozpoczyna się kolejna epoka.

4 Eksperymenty

W ramach tej części projektu zbadane zostało jak pewne zmiany domyślnych wartości parametrów sieci wpływają na poprawę lub pogorszenie się procesu uczenia sieci.

4.1 Wpływ sposobu inicjalizacji wag na szybkość uczenia sieci

W tym eksperymencie sprawdzone zostało jaki wpływ na szybkość uczenia sieci ma użycie dwóch różnych metod inicjalizacji wag. Porównane zostało generowanie wag w sposób losowy, oraz za pomocą algorytmu *Nguyen'a-Widrow'a*, którego kod znajduje się w sekcji 3.1. Do przeprowadzenia testu został wykorzystany następujący program

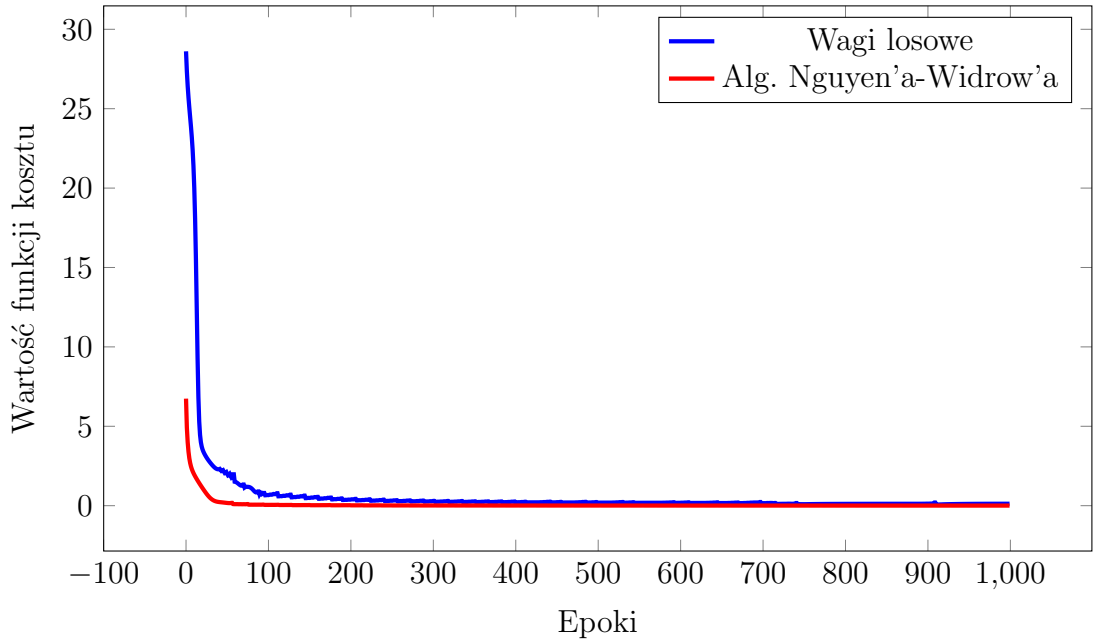
```
learning_data, testing_data = load_wine()
net = nn.NeuralNetwork(0.01, 1000, [10, 8], 1.04, 1.05, 0.7, 0.20)
net.feed_training_data(*learning_data)
net.feed_test_data(*testing_data)
costs = net.start_learning()
```

W teście sieć o 2 warstwach ukrytych o ilości neuronów $S_1 = 10$ i $S_2 = 8$ uczyła się przez 1000 epok.

Dodatkowo przy testowaniu losowych wag została zmodyfikowana funkcja `init_weights_and_biases()` w sposób następujący:

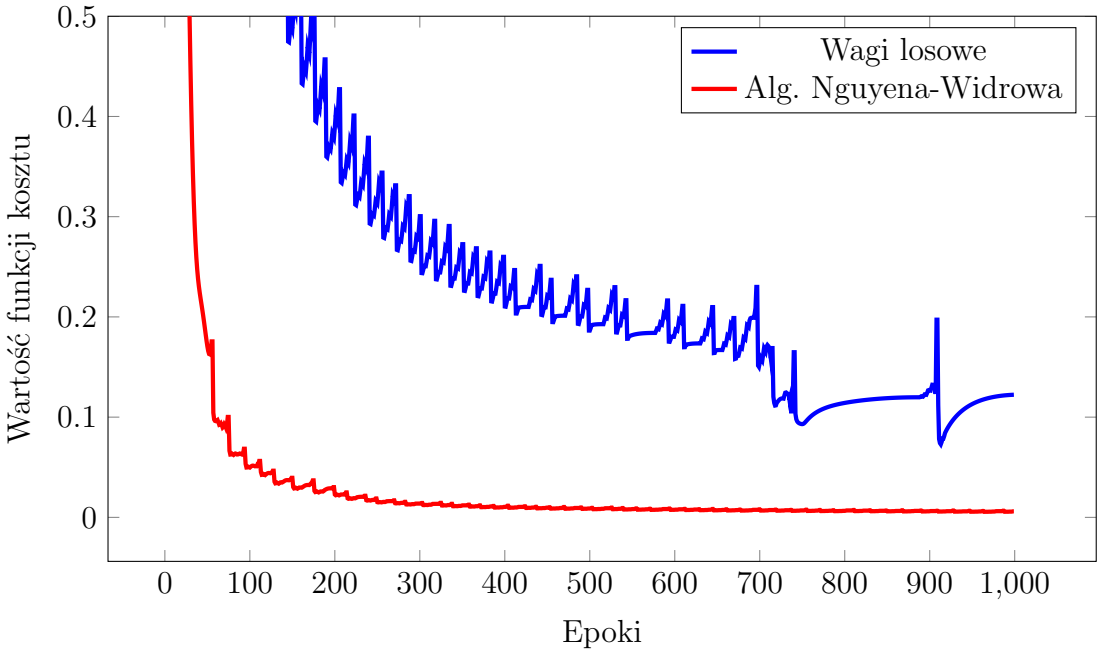
```
def init_weights_and_biases(self):
    self.weights, self.biases = [], []
    sizes = [self.P.shape[1], *self.layers, 1]

    for i in range(1, len(sizes)):
        self.weights.append(np.random.rand(sizes[i], sizes[i-1]))
        self.biases.append(np.random.rand(sizes[i]))
```

Zebrane wyniki eksperymentu przedstawiono na rysuku 5.



Rysunek 5: Poziom nauczenia sieci po 1000 epokach dla dwóch metod inicjalizacji wag

Eksperyment pokazał, że dla wag w pełni losowych początkowy koszt jest prawie 6-krotnie wyższy niż dla wag inicjalizowanych algorytmem *Nguyen'a-Widrow'a*. Dodatkowo funkcja kosztu dużo szybciej spada do niewielkich wartości.



Rysunek 6: Zbliżenie na wykres kosztu przy użyciu dwóch metod inicjalizacji wag

Na zbliżeniu widać także, że koszt maleje bez dużych skoków jak to było przy wagach losowych. W związku z tym do kolejnych eksperymentów algorytm ten będzie wykorzystywany przy inicjalizowaniu wag.

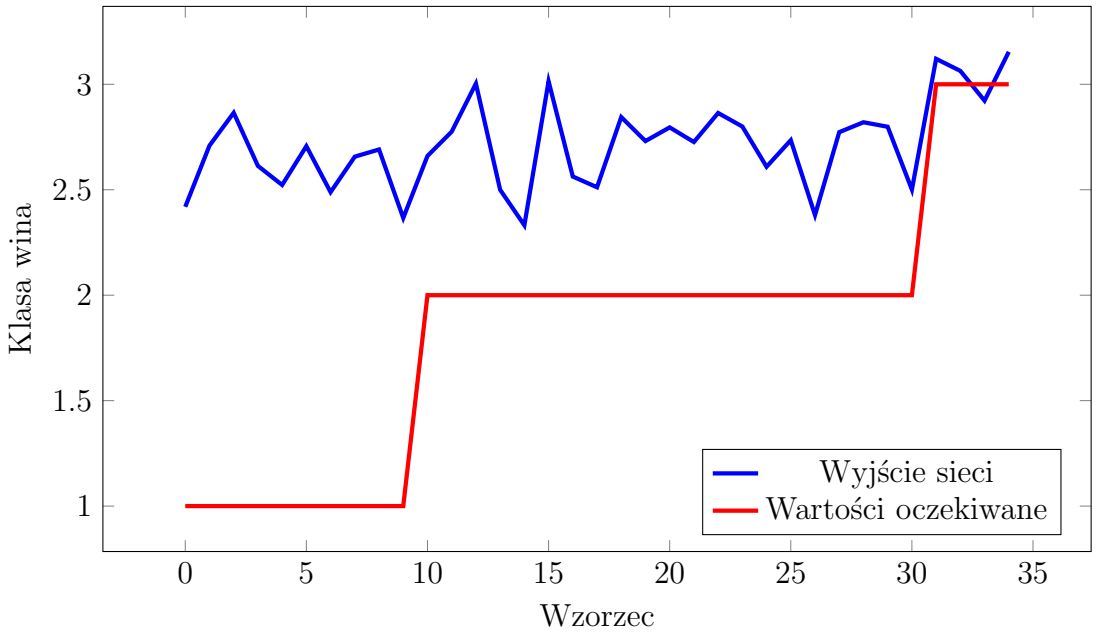
4.2 Wpływ rozkładu klas w danych wejściowych na szybkość uczenia

W tym eksperymecnie zestawione zostały ze sobą dwie wersje danych wejściowych — posortowane według klas, oraz przetasowane w losowej kolejności. Do przeprowadzenia eksperymentu użyto kodu:

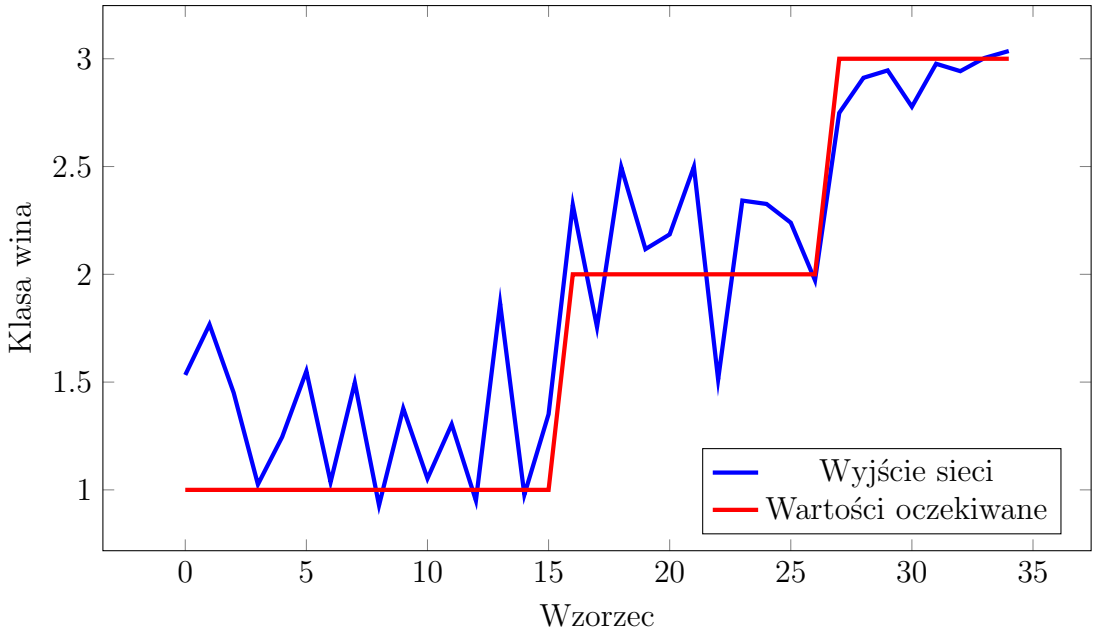
```
learning_data, testing_data = load_wine()
net = nn.NeuralNetwork(0.01, 20, [10, 8], 1.04, 1.05, 0.7, 0.20)
net.feed_training_data(*learning_data)
net.feed_test_data(*testing_data)
net.start_learning()
prediction, cost, pk = net.test(*testing_data)
```

Dodatkowo przy testowaniu danych posortowanych funkcja `load_wine()` została użyta z flagą `sort_training`, która odpowiada za posortowanie danych.

W pierwszej części eksperymentu zatrzymano sieć po 1 epoce aby zobaczyć jak prezentuje się poziom nauczenia sieci na samym początku.

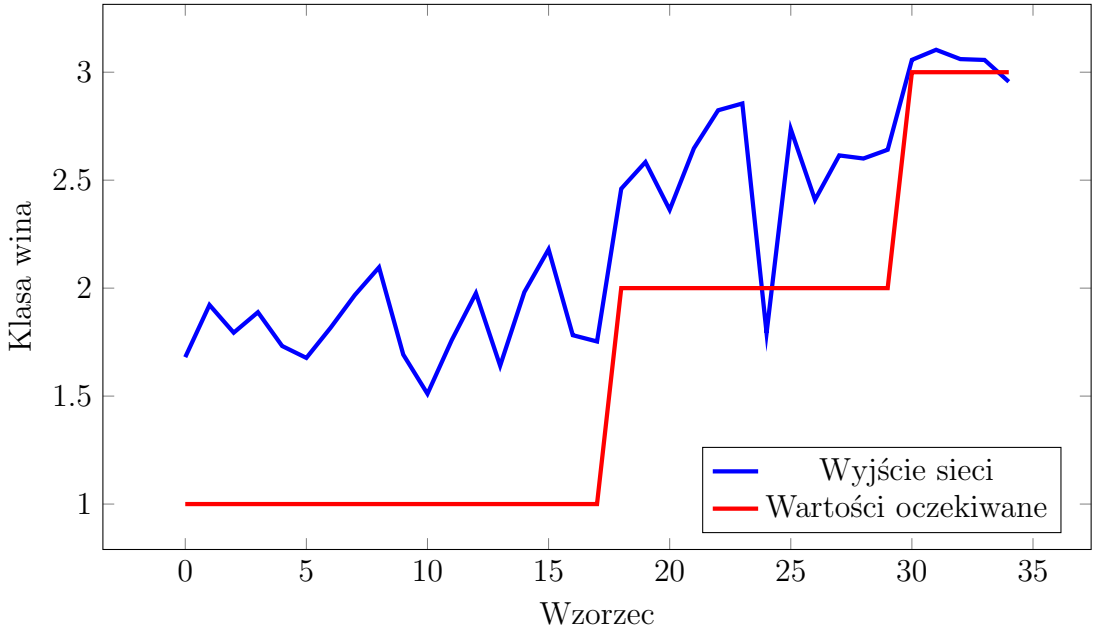


Rysunek 7: Klasyfikacja wina przez sieć po 1 epoce dla danych posortowanych

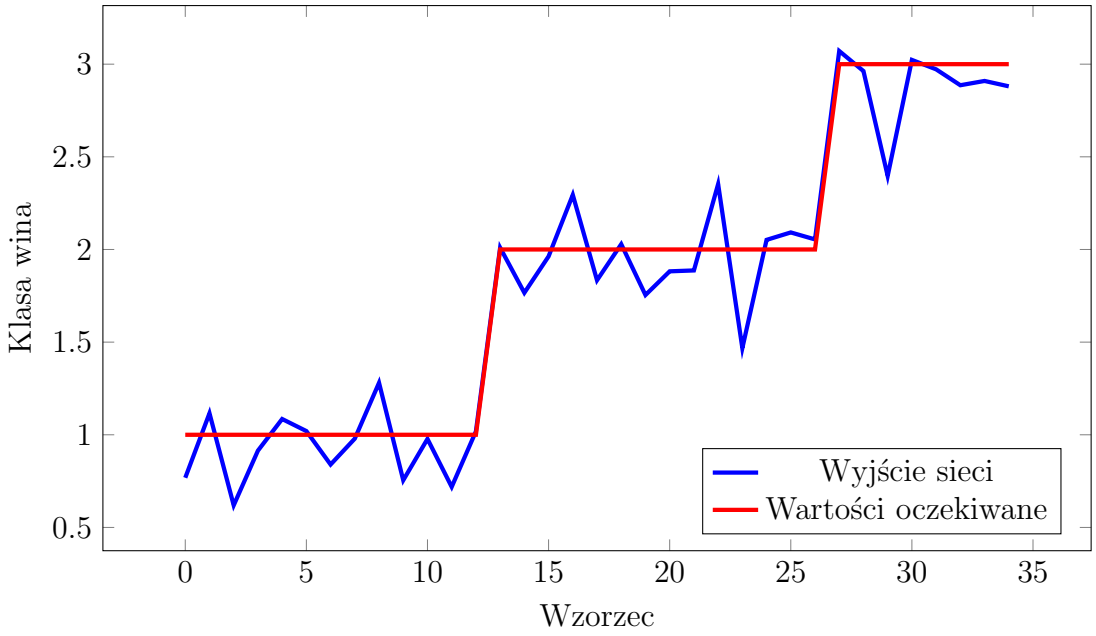


Rysunek 8: Klasyfikacja wina przez sieć po 1 epoce dla danych nieposortowanych

Z wykresów 7 i 8 wyraźnie widać, że już po 1 epoce sieć z danymi nieposortowanymi radziła sobie dużo lepiej, Poszczególne predykcje sieci były o wiele bliższe wartościom oczekiwanym niż jak to było przy danych posortowanych według klas.



Rysunek 9: Klasyfikacja wina przez sieć po 20 epokach dla danych posortowanych



Rysunek 10: Klasyfikacja wina przez sieć po 20 epokach dla danych nieposortowanych

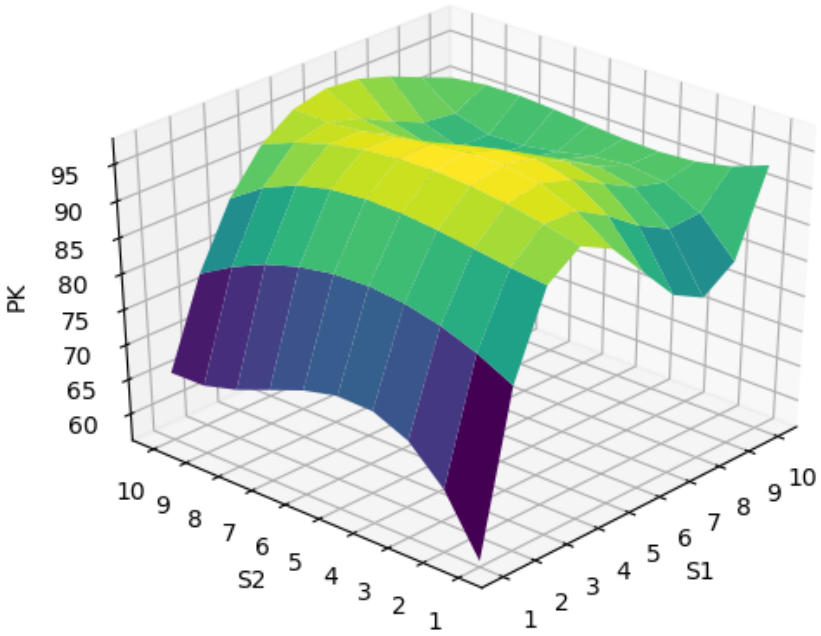
Po uruchomieniu sieci na 20 epok w dalszym ciągu widać dużą przewagę danych przetasowanych. Na wykresie 10 sieć już jest w stanie poprawnie zakwalifikować wino w większej części przypadków. W zestawieniu do tego sieć z posortowanymi danymi poprawnie kwalifikuje tylko pojedyncze przypadki. W kolejnych eksperymentach w celu przyspieszenia sieci dane będą przetasowane w kolejności losowej.

4.3 Wpływ ilości neuronów na poprawność klasyfikacji

W ramach kolejnego eksperymentu sprawdzone zostało przy jakiej ilości neuronów w poszczególnych warstwach sieć się najlepiej uczy. Sieć była uruchamiana korzystając z poniższego skryptu:

```
a, b = np.arange(1, 11), np.arange(1, 11)
S1, S2 = np.meshgrid(a, b)
result = np.empty(S2.shape)

for i, row in enumerate(zip(S1, S2)):
    for j, params in enumerate(zip(*row)):
        learning_data, testing_data = load_wine()
        net = nn.NeuralNetwork(0.01, 100, [*params], 1.04, 1.05,
        ↪  0.7, 0.020)
        net.feed_training_data(*learning_data)
        net.feed_test_data(*testing_data)
        net.start_learning()
        result[i][j] = net.test(*testing_data)[2]
```

Eksperyment przeprowadzono dla parametrów $S_1 = [1, 10]$ oraz $S_2 = [1, 10]$. Za każdym razem sieć uczyła się 100 epok. Wyniki eksperymentu widać na poniższym wykresie.



Rysunek 11: Wykres poprawności klasyfikacji sieci dla różnej ilości neuronów

Z wykresu widać, że dla małej ilości neuronów w pierwszej warstwie kiepsko się uczy. Wysokie wyniki sieć osiąga dla 5 neuronów w warstiwe pierwszej oraz 3-6 neuronów w warstwie drugiej, z najlepszym wynikiem 97% dla $S_1 = 5$ i $S_2 = 4$.

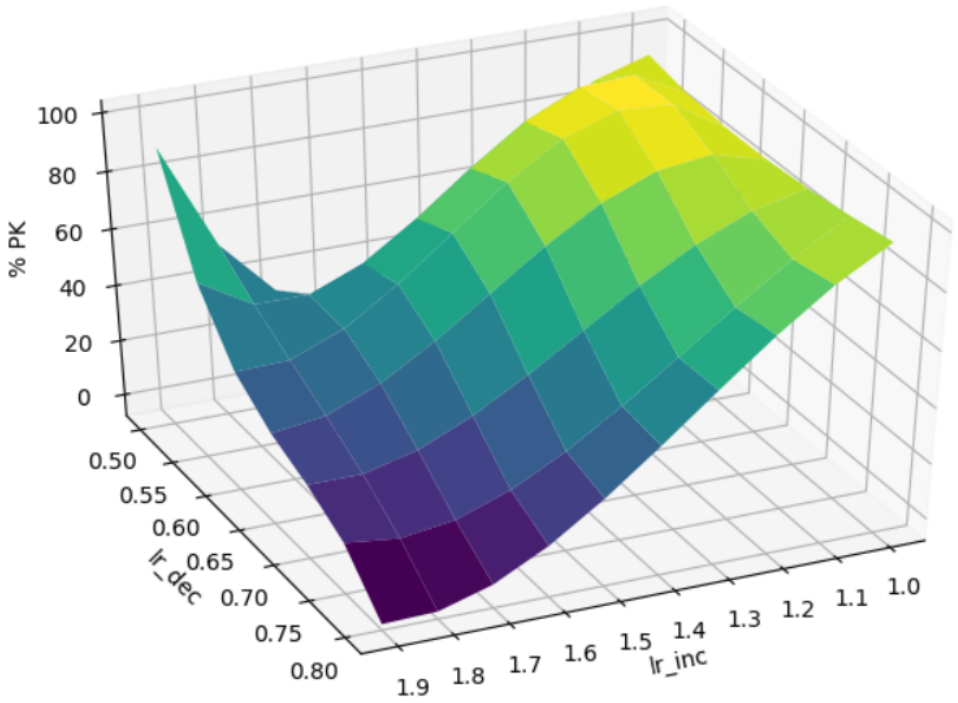
4.4 Wpływ współczynnika uczenia na poprawność klasyfikacji

Następnymi badanymi wartościami są lr_{inc} oraz lr_{dec} czyli współczynniki decydujące o tym jak bardzo współczynnik uczenia będzie rósł lub malał w trakcie nauki. Do badania przyjęto wartości w zakresie [1.0, 1.9] dla lr_{inc} oraz [0.5, 0.8] dla lr_{dec}. Do przeprowadzenia eksperymentu użyto poniższego kodu:

```
a, b = np.arange(1.00, 1.91, 0.1), np.arange(0.5, 0.81, 0.05)
lr_inc, lr_dec = np.meshgrid(a, b)
result = np.empty(lr_dec.shape)

for i, row in enumerate(zip(lr_inc, lr_dec)):
    for j, params in enumerate(zip(*row)):
        learning_data, testing_data = load_wine()
        net = nn.NeuralNetwork(0.01, 100, [5, 4], 1.04, *params,
        ↪  0.020)
        net.feed_training_data(*learning_data)
```

```
10          net.feed_test_data(*testing_data)
11          net.start_learning()
12          result[i][j] = net.test(*testing_data)[2]
```

Efektem działania tego skryptu jest poniższy wykres:



Rysunek 12: Wykres poprawności klasyfikacji sieci dla różnej wartości współczynników lr_{inc} i lr_{dec}

Jak widać na wykresie, dla zbyt dużych lr_{inc} i lr_{dec} sieć traci zbieżność i nie potrafi poprawnie klasyfikować wina. Natomiast dla lr_{inc} między 1.1, a 1.3 oraz lr_{dec} od 0.55 do 0.60 wyniki sięgają 97%.

4.5 Wpływ współczynnika błędu na poprawność klasyfikacji

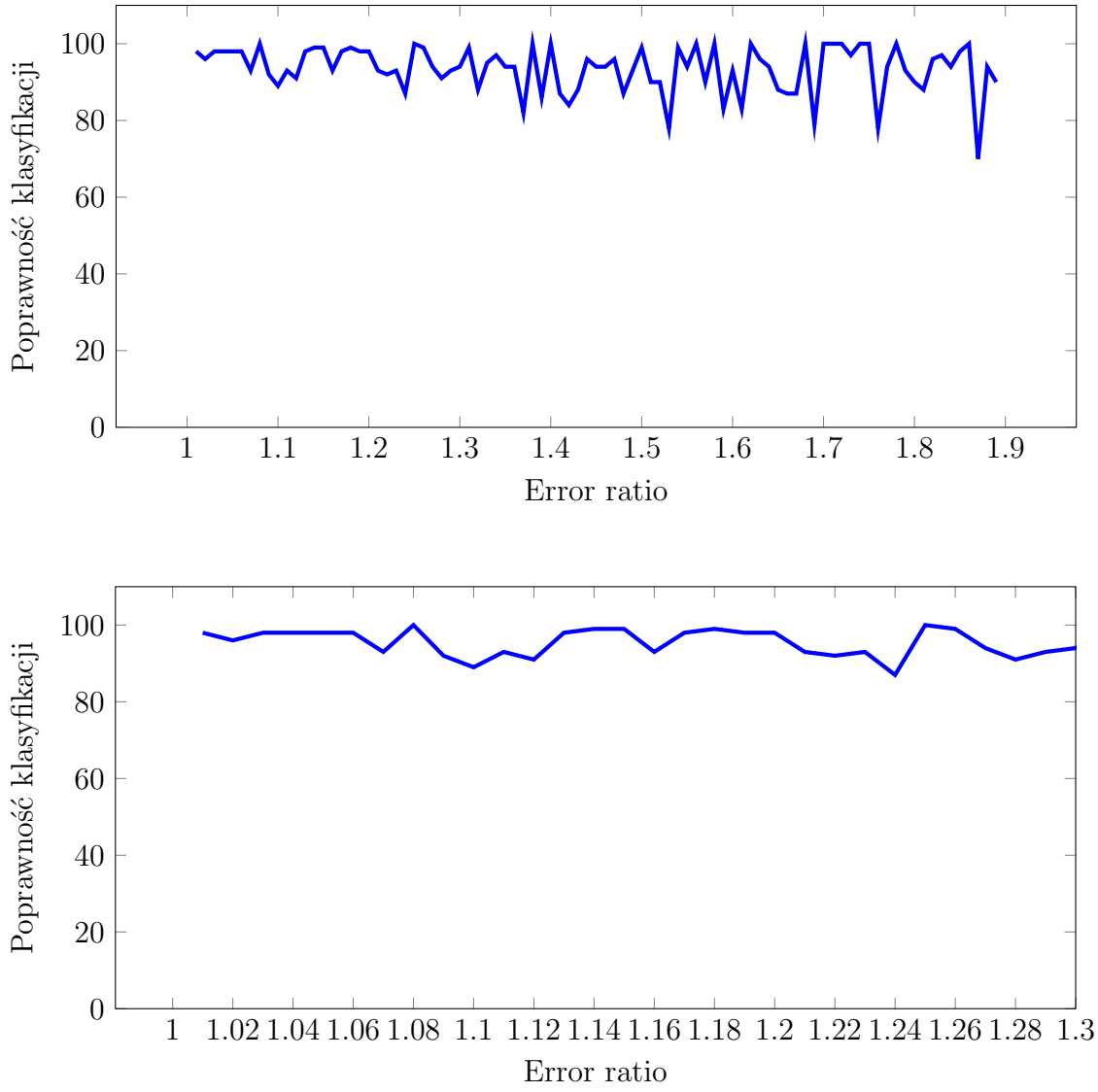
W ostatnim eksperymencie ustalone zostanie jaka wartość współczynnika błędu (err_ratio) najlepiej sobie radzi z wykorzystywanym zestawem danych.

Wykorzystany kod:

```
1  ratios = np.arange(1.01, 1.9, 0.01)
2  results = np.empty((1, len(ratios)))
3
4  for i, err_ratio in enumerate(ratios):
5      learning_data, testing_data = load_wine()
```

```
6       net = nn.NeuralNetwork(0.01, 100, [5, 4], 1.04, 1.2, 0.6,
        ↪  0.020)
7       net.feed_training_data(*learning_data)
8       net.feed_test_data(*testing_data)
9       net.start_learning()
10      pk = net.test(*testing_data)[2]
11      results[j][i] = pk
```

Efekty badania przedstawia poniższy wykres:



Rysunek 13: Wykres poprawności klasyfikacji sieci dla poszczególnych wartości *err_ratio*

Pomimo licznych prób przeprowadzenia eksperymentu za każdym razem wyniki różniły się od siebie i nie można było jednoznacznie określić dla jakich wartości *err_ratio* sieć uczy się lepiej lub gorzej. Jedyną zauważoną zależnością są mniejsze wahania PK dla mniejszego *error ratio*.

5 Wnioski

Dzięki wykonaniu eksperymentów można było zauważyć jak bardzo różne konfiguracje parametrów sieci neuronowej wpływają na proces nauczania. W większości przypadków wyniki eksperymentów były do siebie zbliżone z wyjątkiem ostatniego eksperymentu, w którym nie udało się skorelować zmian badanego współczynnika z konkretną poprawą lub pogorszeniem procesu nauczania.

Pierwszy eksperyment pokazał jak użycie algorytmu Nguyena-Widrowa wpływa już na pierwsze epoki uczenia. W porównaniu do sieci z losowymi wagami nauka od początku była obarczona mniejszym błędem, a wszelkie wahania kosztu odbywały się w niewielkim zakresie. Implementacja tego rozwiązania nie była skomplikowana, a rezultowała dużo lepszymi wynikami.

W drugim eksperymencie dodanie linijki odpowiedzialnej za sortowanie danych uczacych spowodowało pogorszenie się procesu nauczania. Sieć lepiej reagowała na rozlosowane dane dużo szybciej zbiegając do poprawnej klasyfikacji zadanych danych wejściowych.

Trzeci eksperyment był istotny dla tworzenia sieci ponieważ pozwolił ustalić przy jakich najmniejszych ilościach neuronów w 2 warstwach ukrytych sieć osiąga wysoki wynik poprawności klasyfikacji. Początkowe testy przesiewowe pozwoliły zawęzić przeszukiwany zakres neuronów do zakresu [1, 10], następnie właściwa część eksperymentu pozwoliła na określenie $S_1 = 5$ oraz $S_2 = 4$ jako najlepszych wartości. Dzięki temu kolejnych eksperymentów nie trzeba już było wykonywać na większych warstwach, przy których sieć być może osiągnęłaby podobne wyniki lecz przy dłuższym czasie uczenia.

W przypadku współczynnika zmniejszania oraz zwiększania współczynnika uczenia ustalono, że najlepsze wyniki sieć otrzymuje dla $lr_{dec} \approx 0.6$ oraz $lr_{inc} \approx 1.2$.

Wyniki ostatniego eksperymentu niestety nie były wystarczająco jednoznaczne przez co nie udało się ustalić czy wykorzystywany $err\_ratio = 1.04$ był wartością odpowiednią. Pomimo wielokrotnych prób otrzymywane dane różniły się od siebie i nie pozwalały na zauważenie konkretnej tendencji spadkowej lub wzrostowej.

Literatura

[1] S. Aeberhard, *Wine Data Set*, UCI - Machine Learning Repository, 1991. http://archive.ics.uci.edu/ml/datasets/wine

[2] R. Zajdel, *Sztuczna inteligencja, Laboratorium, Ćwiczenie 6 Model neuronu*, PRz, KIiA, Rzeszów.

[3] M. Percy, *Derivative of sigmoid function* $\sigma(x) = \frac{1}{1+e^{-x}}$, 2017. https://math.stackexchange.com/a/1225116

[4] S. Osowski, *Sieci neuronowe do przetwarzania informacji*, Oficyna Wydawnicza Politechniki Warszawskiej, Warszawa 2006.

[5] L. Rutkowski, *Metody i techniki sztucznej inteligencji*, Wydawnictwo Naukowe PWN, Warszawa, 2009.

[6] MathWorks Inc., *Gradient descent with adaptive learning rate backpropagation - MATLAB traingda* https://www.mathworks.com/help/deeplearning/ref/traingda.html

[7] *NeuroLab 0.3.5 documentation* https://pythonhosted.org/neurolab/_modules/neurolab/init.html

[8] D. Nguyen, B. Widrow, *Improving the Learning Speed of 2-Layer Neural Networks by Choosing Initial Values of the Adaptive Weights*, Stanford University, 1990, Stanford

[9] R. Zajdel, *Sztuczna inteligencja, Laboratorium, Ćwiczenie 6 Sieć Jednokierunkowa Wielowarstwowa*, PRz, KIiA, Rzeszów.