WYDZIAŁ
ELEKTROTECHNIKI
I INFORMATYKI
POLITECHNIKI RZESZOWSKIEJ

Praca projektowa z przedmiotu "Sztuczna Inteligencja"

Temat:

Realizacja sieci neuronowej uczonej algorytmem wstecznej propagacji błędu z przyśpieszeniem metodą adaptacyjnego współczynnika uczenia zastosowanej do rozpoznawania przeżywalności pacjentów chorych na wirusowe zapalenie wątroby

Mariia Rybak

Czerwiec 11, 2024

WYDZIAŁ
ELEKTROTECHNIKI
I INFORMATYKI
POLITECHNIKI RZESZOWSKIEJ

Spis treści

1. Opis problemu

Głównym założeniem projektu jest realizacja sieci neuronowej uczonej za pomocą algorytmu wstecznej propagacji błędu z przyśpieszeniem metodą adaptacyjnego współczynnika uczenia, której zadaniem jest badanie wpływu poszczególnych czynników na przeżywalność pacjentów chorych na wirusowe zapalenie wątroby o danej płci i wieku.
W ramach projektu zostały przeprowadzone eksperymenty w celu wyznaczenia optymalnych wartości poniższych parametrów:

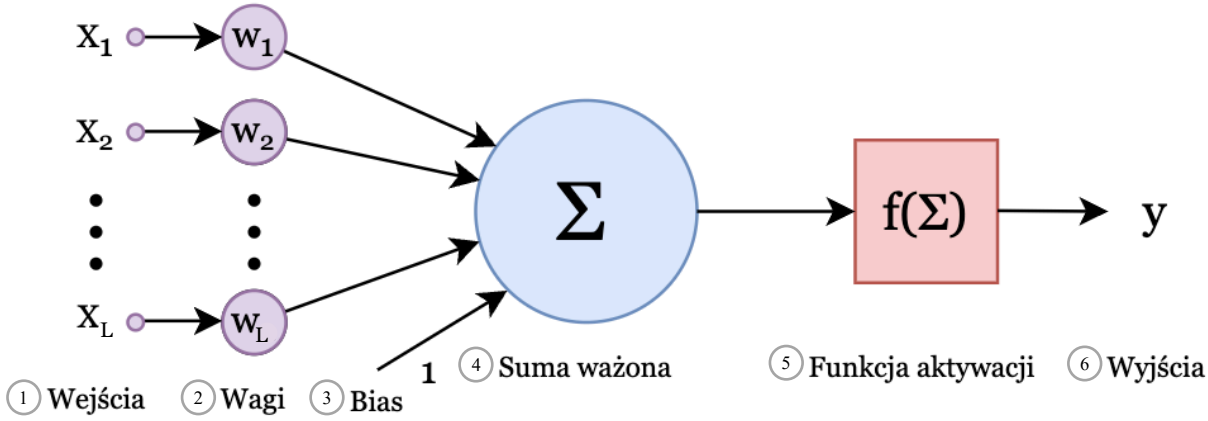
- `K1` – ilość neuronów w warstwie I
- `K2` – ilość neuronów w warstwie II
- `lr_inc` – współczynnik zwiększania współczynnika uczenia (lr)
- `lr_dec` – współczynnik zmniejszania współczynnika uczenia (lr)
- `err` – dopuszczalna krotność przyrostu błędu

Zbiór danych uczących to „*Hepatitis*”. Sieć została zrealizowana przy użyciu języka programowania Python oraz biblioteki „`nnet`”.

2. Część teoretyczna

2.1 Model neuronu

Model sztucznego neuronu to matematyczna reprezentacja biologicznego neuronu, który jest podstawowym elementem sztucznych sieci neuronowych stosowanych w dziedzinie uczenia maszynowego i sztucznej inteligencji. Sztuczne neurony, zwane także perceptronami, są fundamentalnym budulcem tych sieci i naśladują działanie neuronów biologicznych, przetwarzając sygnały wejściowe i przekazując je dalej przez sieć.



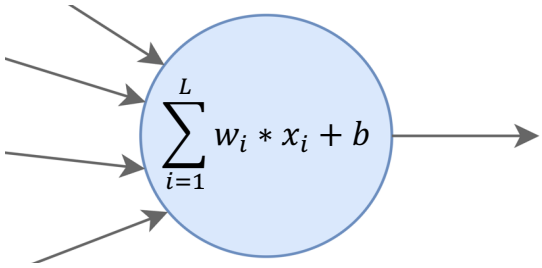
Rys.1 Sztuczny neuron według modelu *McCullocha-Pittsa*

Kluczowe elementy modelu sztucznego neuronu:

1. **Wejścia**: Są to wartości, które neuron otrzymuje z zewnętrznych źródeł lub innych neuronów. Każde wejście jest zwykle związane z wagą.
2. **Wagi**: Każde wejście jest mnożone przez odpowiednią wagę. Wagi określają, jak silny jest wpływ danego wejścia na wyjście neuronu.
3. **Bias**: Pozwala przesuwać funkcję aktywacji wzdłuż osi poziomej, co zwiększa elastyczność modelu w dopasowywaniu danych.
4. **Suma ważona**: Suma wszystkich iloczynów wejść i ich odpowiednich wag.
5. **Funkcja aktywacji**: Jest to funkcja, która przekształca sumę ważoną na wyjście neuronu. Funkcja aktywacji wprowadza nieliniowość, co pozwala sieci na rozwiązywanie bardziej złożonych problemów.
6. **Wyjście**: To wartość, którą neuron wysyła dalej do innych neuronów lub jako wynik końcowy. Jest to wynik zastosowania funkcji aktywacji na sumie ważonej.

2.2 Opis matematyczny sztucznego neuronu

Suma ważona wyraża się jako:



Rys.2 Suma ważona

gdzie w_i to waga, a x_i to wartość wejściowa.
Wyjście **y** neuronu można opisać matematyczną zależnością:

$$y = f(\sum_{i=1}^{L} w_i * x_i + b)$$

- w_i to waga,
- x_i to wartość wejściowa,
- b to bias (stała),
- L to liczba sygnałów wejściowych.

Schemat działania sztucznego neuronu

1. Neuron odbiera sygnały wejściowe $x_1, x_2, \dots, x_L$.
2. Każde wejście jest mnożone przez przypisaną wagę $w_1, w_2, \dots, w_L$.
3. Obliczana jest suma ważona $z = \sum_{i=1}^{L} w_i * x_i + b$.
4. Suma ważona jest przekazywana przez funkcję aktywacji, co daje wynik: $y = f(z)$ gdzie f to funkcja aktywacji.
5. Wyjście y jest przekazywane dalej.

Uproszczenie równania neuronu

W celu uproszczenia równania sztucznego neuronu, możemy wykorzystać notację wektorową i macierzową. Załóżmy, że mamy następujące definicje:

1. **Wektor wejściowy**: $x = [x_1, x_2, \dots, x_L]^T$ gdzie x jest wektorem sygnałów wejściowych o długości L, a T oznacza transpozycję, co sprawia, że wektor jest kolumnowy.
2. **Wektor wag**: $w = [w_1, w_2, \dots, w_L]$ gdzie w jest macierzą wierszową wag o tej samej długości L.
3. **Skalary**:
    - y: wyjście neuronu.
    - b: bias (przesunięcie).

Uproszczone równanie

Korzystając z notacji wektorowej, możemy zapisać równanie sumy ważonej wejść jako iloczyn skalarny wektorów w i x, a następnie dodać bias b:

$$z = w * x + b$$

gdzie: $w * x$ to iloczyn skalarny wektorów w i x, co oznacza $\sum_{i=1}^{L} w_i * x_i$

Zastosowanie funkcji aktywacji

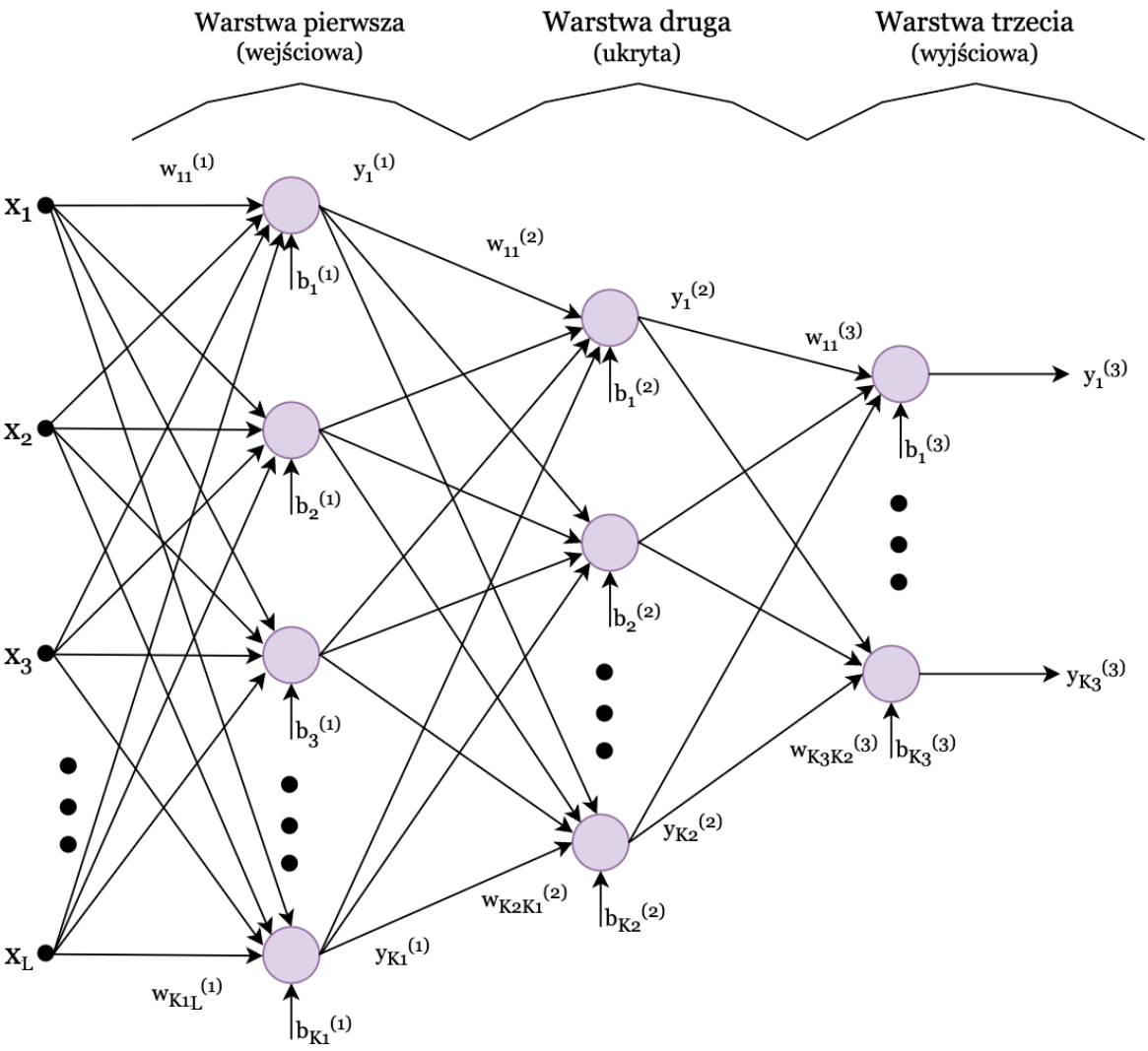
Wynik z jest następnie przekształcany przez funkcję aktywacji $f(x)$, aby uzyskać ostateczne wyjście y:

$$y = f(z) = f(w * x + b)$$

Użycie notacji wektorowej i macierzowej upraszcza zapis i analizę działania sztucznego neuronu. Równanie $z = w * x + b$ w elegancki sposób integruje wszystkie wejścia, wagi i bias, co jest bardziej zrozumiałe i łatwiejsze do obliczenia w praktyce. Funkcja aktywacji $f(z)$ przekształca liniową kombinację wejść na nieliniowe wyjście, co pozwala na efektywne modelowanie złożonych wzorców.

2.3 Sieci neuronowe jednokierunkowe wielowarstwowe

Sieci neuronowe jednokierunkowe wielowarstwowe, znane również jako Multi-Layer Perceptrons (MLP), są jedną z podstawowych architektur sztucznych sieci neuronowych. Charakteryzują się przepływem informacji w jednym kierunku – od warstwy wejściowej do warstwy wyjściowej – bez sprzężenia zwrotnego.



Rys.3 Sieć jednokierunkowa wielowarstwowa

Struktura MLP

MLP składa się z kilku typów warstw:

1. **Warstwa wejściowa**:
    - Ta warstwa zawiera neurony, które otrzymują dane wejściowe. Każdy neuron w tej warstwie reprezentuje jedną cechę wejściową.
2. **Warstwy ukryte**:
    - Te warstwy składają się z neuronów, które przetwarzają sygnały z warstwy wejściowej lub poprzednich warstw ukrytych. Liczba warstw ukrytych oraz liczba neuronów w każdej warstwie są hiperparametrami modelu.
3. **Warstwa wyjściowa**:
    - Ta warstwa zawiera neurony, które generują wyjście modelu. Liczba neuronów w tej warstwie zależy od rodzaju zadania (np. jeden neuron dla regresji, wiele neuronów dla klasyfikacji wieloklasowej).

Przepływ informacji

W MLP przepływ informacji odbywa się w następujący sposób:

1. **Przetwarzanie w warstwie wejściowej**:
    - Dane wejściowe są podawane do neuronów w warstwie wejściowej.
2. **Przetwarzanie w warstwach ukrytych**:
    - Każdy neuron w warstwie ukrytej oblicza sumę ważoną swoich wejść, dodaje bias, a następnie przepuszcza wynik przez funkcję aktywacji, aby wprowadzić nieliniowość. Wyjścia tej warstwy stają się wejściami dla kolejnej warstwy ukrytej lub warstwy wyjściowej.
3. **Generowanie wyniku w warstwie wyjściowej**:
    - Neurony w warstwie wyjściowej przetwarzają sygnały z ostatniej warstwy ukrytej i generują ostateczne wyniki modelu.

Funkcje aktywacji

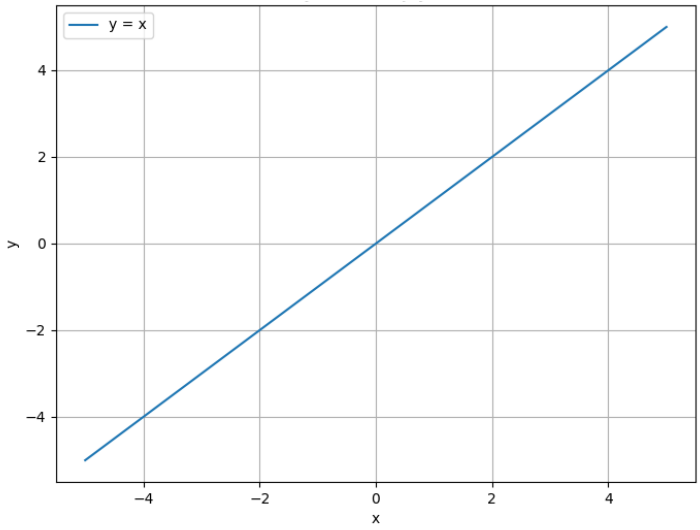
W MLP stosuje się różne funkcje aktywacji w warstwach ukrytych, aby wprowadzić nieliniowość do modelu. Przykłady funkcji aktywacji to:

- ***purelin***:

$$f(x) = ax$$

Pochodna tej funkcji:

$$f'(x) = x$$



Rys.4 Wykres funkcji *purelin*

- ***tansig***:

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

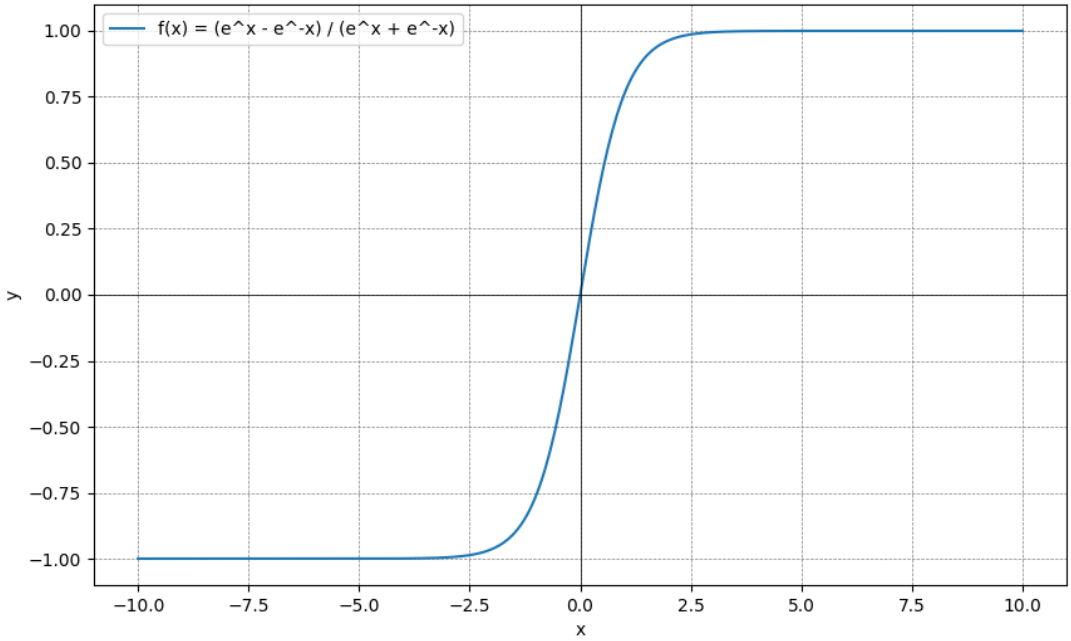
Pochodna tej funkcji:

$$f'(x) = \frac{(e^x + e^{-x})(e^x + e^{-x}) - (e^x - e^{-x})(e^x - e^{-x})}{(e^x + e^{-x})^2} =$$

$$= \frac{(e^x + e^{-x})^2 - (e^x - e^{-x})^2}{(e^x + e^{-x})^2} =$$

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

$$= 1 - f(x)^2$$



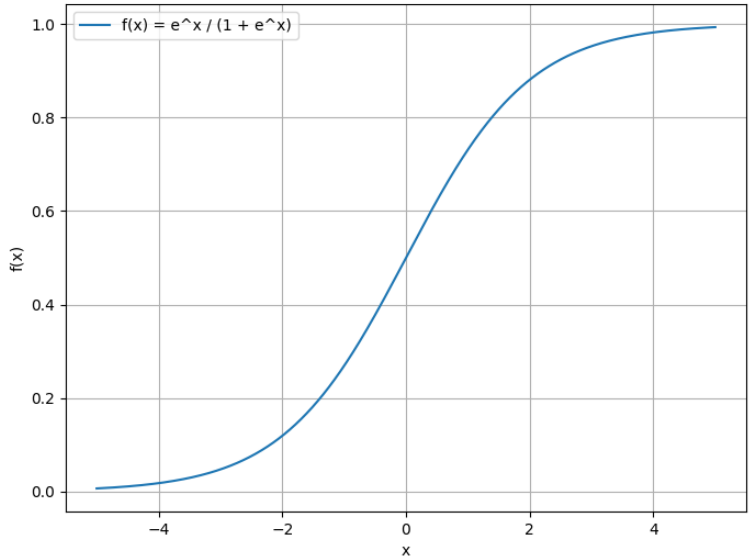
Rys.5 Wykres funkcji *tansig*

- ***logsin***:

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

Pochodna tej funkcji:

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



Rys.6 Wykres funkcji *logsig*

Każda warstwa neuronów ma swój macierz wag $\boldsymbol{w}$, wektor przesunięcia $\boldsymbol{b}$, funkcję aktywacji $\boldsymbol{f}$, a także wektor sygnałów wyjściowych $\boldsymbol{y}$. Dla rozróżnienia poszczególnych warstw do każdej z wielkości dodano numer warstwy, której dotyczy. W tym przypadku 1 oznacza warstwę wejściową, 2 warstwę ukrytą, a 3 warstwę wyjściową. Działanie poszczególnych warstw wygląda następująco:

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

gdzie x jest wektorem sygnałów wejściowych.
Działanie całej sieci można opisać zależnością:

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

Uczenie MLP
MLP są uczone za pomocą algorytmu wstecznej propagacji błędów (backpropagation) i optymalizowane za pomocą różnych metod optymalizacji. Proces uczenia polega na:

1. **Przepływie w przód (forward pass)**:
   - Obliczenie wyjścia sieci dla danych wejściowych.
2. **Obliczeniu błędu**:
   - Porównanie wyjścia sieci z rzeczywistymi wartościami (etykietami) za pomocą funkcji kosztu.
3. **Przepływie wstecz (backward pass)**:
   - Propagacja błędu wstecz przez sieć i aktualizacja wag na podstawie gradientu błędu.

2.4 Algorytm wstecznej propagacji błędu

Algorytm wstecznej propagacji błędu (backpropagation) to metoda stosowana do trenowania sztucznych sieci neuronowych. Jest to technika optymalizacji służąca do minimalizacji funkcji kosztu przez dostosowanie wag sieci neuronowej na podstawie obliczonego gradientu błędu. Algorytm wstecznej propagacji błędu składa się z dwóch głównych etapów: przepływu w przód (forward pass) i przepływu wstecz (backward pass).

Etapy algorytmu wstecznej propagacji błędu:

1. **Przepływ w przód**:
   - Dane wejściowe są podawane do sieci neuronowej i przepływają przez kolejne warstwy, aż do warstwy wyjściowej. Na każdym neuronie obliczana jest suma ważona wejść i stosowana jest funkcja aktywacji, co daje wyjście neuronu.
   - Wyjście z warstwy wyjściowej jest porównywane z rzeczywistą wartością, co pozwala obliczyć błąd (funkcję kosztu).
2. **Obliczenie błędu**:
   - Funkcja kosztu mierzy różnicę między przewidywanym a rzeczywistym wyjściem sieci. Ta różnica jest traktowana jako błąd, który sieć musi zminimalizować.
3. **Przepływ wstecz**:
   - Algorytm oblicza gradient funkcji kosztu względem wag sieci, zaczynając od warstwy wyjściowej i idąc wstecz do warstwy wejściowej. Gradienty są obliczane za pomocą reguły łańcuchowej z rachunku różniczkowego.
4. **Aktualizacja wag**:
   - Wagi sieci są aktualizowane na podstawie obliczonych gradientów i wybranego algorytmu optymalizacji. Wagi są korygowane w kierunku, który minimalizuje funkcję kosztu.

Następny wzór opisuje sposób aktualizacji wag w sieciach neuronowych w sposób iteracyjny. Dokładnie, wzór ten mówi, jak waga w_{ij} między neuronem j w warstwie poprzedzającej a neuronem i w bieżącej warstwie jest aktualizowana w kolejnej iteracji:

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

gdzie:

- $w_{ij}(t)$ to waga między neuronem j a neuronem i w iteracji t,
- $\Delta w_{ij}(t)$ to zmiana wagi w_{ij} w iteracji t,
- $w_{ij}(t+1)$ to waga w iteracji $t+1$.

Każdy wektor wejściowy $\boldsymbol{x}$ to zestaw cech lub danych, które są podawane na wejście sieci neuronowej. Każdemu wektorowi wejściowemu $\boldsymbol{x}$ towarzyszy pożądany wektor wyjściowy $\widehat{\boldsymbol{y}}$, który reprezentuje oczekiwaną odpowiedź sieci neuronowej na dane wejście.
Warstwa neuronów odpowiada na wektor wejściowy $\boldsymbol{x}$ generując wyjściowy wektor $\boldsymbol{y}$. Jeśli sieć nie jest jeszcze nauczona, wektor wyjściowy $\boldsymbol{y}$ różni się od pożądanego wektora $\widehat{\boldsymbol{y}}$. Różnica ta jest określana jako błąd $\boldsymbol{e}$.
Błąd można zdefiniować jako różnicę między rzeczywistym wyjściem $\boldsymbol{y}$ a pożądanym wyjściem $\widehat{\boldsymbol{y}}$:

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

gdzie:

- e to wektor błędu,
- y to rzeczywisty wektor wyjściowy sieci,
- $\hat{y}$ to pożądany wektor wyjściowy.

Celem uczenia pod nadzorem jest zminimalizowanie tego błędu $\boldsymbol{e}$. W praktyce oznacza to minimalizowanie funkcji kosztu, która mierzy całkowity błąd sieci dla zbioru treningowego. Przyjmuje ona postać błędu średniokwadratowego dla każdego neuronu w danej warstwie. Dla określonej warstwy neuronów błąd jest następujący:

$$E = \frac{1}{2}\sum_{i=1}^{K_3} {e_i}^2$$

W przypadku sieci wielowarstwowej (w tym przypadku trójwarstwowej) postać błędu jest następująca:

$$E = \frac{1}{2}\sum_{i_3=1}^{K_3} e_{i_3}^2 =$$

$$= \frac{1}{2}\sum_{i_3=1}^{K_3} \left(y_{i_3}^{(3)} - \hat{y}_{i_3}\right)^2 =$$

$$= \frac{1}{2}\sum_{i_3=1}^{K_3} \left(f^{(3)} \left(\sum_{i_2=1}^{K_2} w_{i_3 i_2}^{(3)} y_{i_2}^{(2)} + b_{i_3}^{(3)} \right) - \hat{y}_{i_3} \right)^2 =$$

$$= \frac{1}{2}\sum_{i_3=1}^{K_3} \left(f^{(3)} \left(\sum_{i_2=1}^{K_2} w_{i_3 i_2}^{(3)} f^{(2)} \left(\sum_{i_1=1}^{K_1} w_{i_2 i_1}^{(2)} y_{i_1}^{(1)} + b_{i_2}^{(2)} \right) + b_{i_3}^{(3)} \right) - \hat{y}_{i_3} \right)^2 =$$

$$= \frac{1}{2}\sum_{i_3=1}^{K_3} \left(f^{(3)} \left(\sum_{i_2=1}^{K_2} w_{i_3 i_2}^{(3)} f^{(2)} \left(\sum_{i_1=1}^{K_1} w_{i_2 i_1}^{(2)} f^{1} \left(\sum_{j=1}^{L} w_{i_1 j}^{(1)} x_j + b_{i_1}^{(1)} \right) + b_{i_2}^{(2)} \right) + b_{i_3}^{(3)} \right) - \hat{y}_{i_3} \right)^2$$

gdzie j oznacza numer wejścia warstwy pierwszej, a i_1, i_2, i_3 oznaczają wyjścia odpowiednio warstwy pierwszej, drugiej i trzeciej.

Jako że wartość funkcji kosztu zależy od wartości wag w modelu, to poszukiwanie minimum może być osiągnięte za pomocą metody gradientowej. Najczęściej stosowaną metodą gradientów zmiany wartości wag jest metoda największego spadku, w której wektor przyrostu wag wygląda następująco:

$$\Delta w = -\eta \nabla E(w)$$

gdzie ∇ jest gradientem, a η współczynnikiem uczenia.

Zmiana wag $\Delta w_{ij}(t)$ jest obliczana na podstawie gradientu funkcji kosztu względem tej wagi. Dla algorytmu gradient descent jest to:

$$\Delta w_{ij}(t) = -\eta \frac{\partial E}{\partial w_{ij}}$$

gdzie:

- η to współczynnik uczenia.
- $\frac{\partial y}{\partial w_{ij}}$ to gradient funkcji kosztu E względem wagi w_{ij}.

Wykorzystując zależności $E = E\left(y_i\left(z_i(w_{ij})\right)\right)$ pochodna będzie miała postać:

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial y_i}\frac{\partial y_i}{\partial z_i}\frac{\partial z_i}{\partial w_{ij}}$$

Uwzględniając następne zależności:

$$\frac{\partial E}{\partial y_i} = \frac{\partial}{\partial y_i}\frac{1}{2}\left(\sum_{i_3=1}^{K_3}\left(y_i - \hat{y}_{i_3}\right)^2\right) =$$

$$= \frac{\partial}{\partial y_i}\left(\frac{1}{2} * \left((y_1 - \hat{y}_1)^2 + (y_2 - \hat{y}_2)^2 + \cdots + (y_i - \hat{y}_i)^2 + \cdots + \left(y_{K_3} - \hat{y}_{K_3}\right)^2\right)\right) =$$

$$= (y_i - \hat{y}_i)$$

$$\frac{\partial z_i}{\partial w_{ij}} = \frac{\partial}{\partial w_{ij}}\left(\sum_{k=1}^{L} w_{ik}x_k + b_i\right) =$$

$$= \frac{\partial}{\partial w_{ij}}\left(w_{i1}x_1 + w_{i2}x_2 + \cdots + w_{ij}x_j + \cdots + w_{iL}x_L + b_i\right) =$$

$$= x_j$$

ten wzór będzie wyglądać następująco:

$$\frac{\partial E}{\partial w_{ij}} = (y_i - \hat{y}_i)\frac{\partial y_i}{\partial z_i}x_j$$

Dla warstwy wyjściowej ten wzór będzie miał postać:

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

Elementy gradientu względem wag warstwy ukrytej ma postać:

$$\frac{\partial E}{\partial w_{i_2 i_1}^{(2)}} = \sum_{i_3=1}^{K_3}(y_{i_3}^{(3)} - \hat{y}_{i_3})\frac{\partial f^{(3)}(z_{i_3}^{(3)})}{\partial z_{i_3}^{(3)}}w_{i_3 i_2}^{(3)}\frac{\partial f^{(2)}(z_{i_2}^{(2)})}{\partial z_{i_2}^{(2)}}y_{i_1}^{(1)}$$

Dla warstwy wejściowej ma postać:

$$\frac{\partial E}{\partial w_{i_1 j}^{(1)}} = \sum_{i_3=1}^{K_3} (y_{i_3}^{(3)} - \hat{y}_{i_3}) \frac{\partial f^{(3)}(z_{i_3}^{(3)})}{\partial z_{i_3}^{(3)}} \sum_{i_2=1}^{K_2} w_{i_3 i_2}^{(3)} \frac{\partial f^{(2)}(z_{i_2}^{(2)})}{\partial z_{i_2}^{(2)}} w_{i_2 i_1}^{(2)} \frac{\partial f^{(1)}(z_{i_1}^{(1)})}{\partial z_{i_1}^{(1)}} x_j$$

Elementy gradientu dla biasów względem wag warstwy wyjściowej wygląda następująco:

$$\frac{\partial E}{\partial b_{i_3}^{(3)}} = (y_{i_3}^{(3)} - \hat{y}_{i_3}) \frac{\partial f^{(3)}(z_{i_3}^{(3)})}{\partial z_{i_3}^{(3)}} * 1$$

Dla warstwy ukrytej:

$$\frac{\partial E}{\partial b_{i_2}^{(2)}} = \sum_{i_3=1}^{K_3} (y_{i_3}^{(3)} - \hat{y}_{i_3}) \frac{\partial f^{(3)}(z_{i_3}^{(3)})}{\partial z_{i_3}^{(3)}} w_{i_3 i_2}^{(3)} \frac{\partial f^{(2)}(z_{i_2}^{(2)})}{\partial z_{i_2}^{(2)}} * 1$$

i dla warstwy wejściowej:

$$\frac{\partial E}{\partial b_{i_1}^{(1)}} = \sum_{i_3=1}^{K_3} (y_{i_3}^{(3)} - \hat{y}_{i_3}) \frac{\partial f^{(3)}(z_{i_3}^{(3)})}{\partial z_{i_3}^{(3)}} \sum_{i_2=1}^{K_2} w_{i_3 i_2}^{(3)} \frac{\partial f^{(2)}(z_{i_2}^{(2)})}{\partial z_{i_2}^{(2)}} w_{i_2 i_1}^{(2)} \frac{\partial f^{(1)}(z_{i_1}^{(1)})}{\partial z_{i_1}^{(1)}} * 1$$

2.5 Metoda adaptacyjnego współczynnika uczenia

Metoda adaptacyjnego współczynnika uczenia (Adaptive Learning Rate Method) to technika stosowana w optymalizacji sieci neuronowych, mająca na celu dynamiczne dostosowywanie tempa uczenia się (learning rate) podczas treningu. Zamiast używać stałej wartości współczynnika uczenia, który może być nieoptymalny przez cały czas trwania treningu, adaptacyjne metody pozwalają na automatyczne dostosowanie tego parametru w celu poprawy konwergencji i skuteczności optymalizacji.
Na podstawie porównania wartości sumarycznego błędu kwadratowego SSE w chwili uczenia t z jej poprzednii wartością $\mathrm{SSE}(\mathrm{t}-1)$, sposób zmiany współczynnika uczenia η definiuje się jako:

$$\eta(t+1)=\begin{cases}\eta(t)\xi_d, & gdy\ SSE(t) > er * SSE(t-1)\\ \eta(t)\xi_i, & gdy\ SSE(t) < SSE(t-1)\\ \eta(t), & gdy\ SSE(t-1) \le SSE(t) \le er * SSE(t-1)\end{cases}$$

gdzie ξ_d jest współczynnikiem zmniejszania wartości η, ξ_i współczynnikiem zwiększania wartości η, a er dopuszczalną krotnością błędu .

3. Analiza danych

Zbiór danych uczących zawiera 155 rekordów, gdzie każdy zawiera 20 atrybutów. Zbiór znajduje się pod adresem: http://archive.ics.uci.edu/ml/datasets/Hepatitis. Zestaw zawiera parametry z brakującymi danymi, które zostały usunięte. Po usunięciu poszczególnych rekordów zawierających brakujące dane nastąpiło ich istotne zmniejszenie ze 155 do 80. Zadaniem perceptronu jest ocena przeżywalności (nie/tak) na podstawie danych zawierających poszczególne schorzenia u badanych. W przypadku wszystkich parametrów, gdzie zawarta jest informacja nie lub tak, liczba **1** odpowiada przypadkowi **nie**, natomiast liczba **2 – tak.**

Parametry występujące w zbiorze *„Hepatitis”* opisują poszczególne cechy:

1. **Class** – wartość liczbowa, określająca czy badana osoba żyje lub nie
2. **Age** – wartość całkowitoliczbowa, określająca wiek osoby
3. **Sex** – wartość liczbowa, określająca płeć osoby (1 – mężczyzna, 2 – kobieta)
4. **Steroid** – wartość liczbowa, określająca, czy dana osoba zażywa sterydy (nie/tak)
5. **Antivirals** – wartość liczbowa, określająca zażywanie leków przeciwwirusowych przez daną osobę (nie/tak)
6. **Fatigue** – wartość liczbowa, określająca zmęczenie danej osoby (nie/tak)
7. **Malaise** – wartość liczbowa, określająca złe samopoczucie danej osoby (nie/tak)
8. **Anorexia** – wartość liczbowa, określająca, czy dana osoba cierpi na brak apetytu (nie/tak)
9. **Liver big** – wartość liczbowa, określająca, czy osoba ma problem z powiększoną wątrobą (hepatomegalia) (nie/tak)
10. **Liver firm** – wartość liczbowa, określająca, czy osoba ma problem z twardością wątroby (nie/tak)
11. **Spleen Palpable** – wartość liczbowa, określająca czy dana osoba ma powiększoną śledzionę (nie/tak)
12. **Spiders** – wartość liczbowa, określająca, czy dana osoba posiada pajączki wątrobowe (nie/tak)
13. **Ascites** – wartość liczbowa, określająca, czy dana osoba ma wodobrzusze (nie/tak)
14. **Varices** - wartość liczbowa, określająca, czy u danej osoby występują żylaki przełyku (nie/tak)
15. **Bilirubin** – wartość ciągła, określająca poziom bilirubiny (barwnika żółciowego) w organizmie badanej osoby.
16. **Alk Phosphate** – wartość całkowitoliczbowa, określająca poziom fosfatazy alkalicznej (33, 80, 120, 160, 200, 250)
17. **SGOT** – wartość całkowitoliczbowa, określająca poziom aminotransferazy asparaginianowy w organizmie osoby (13, 100, 200, 300, 400, 500)
18. **Albumin** – wartość liczbowa zmiennoprzecinkowa, określająca poziom albuminy w organizmie (2.1, 3.0, 3.8, 4.5, 5.0, 6.0)

19. **Protime** – wartość całkowitoliczbowa, określająca czas protrombinowy badanej osoby (10, 20, 30, 40, 50, 60, 70, 80, 90)
20. **Histology** – wartość całkowitoliczbowa, określająca, czy dana osoba była badana w kierunku histopatologicznym.

Analizując powyższą strukturę danych, można stwierdzić, że pierwsza kolumna zostanie wykorzystana jako klasa, przyjmując tylko dwie wartości: 1 lub 2. Reprezentuje ona oczekiwaną wartość klasyfikacji i jest parametrem wyjściowym. Pozostałe kolumny, od 2 do 20, traktujemy jako parametry wejściowe, stanowiące dane wejściowe.

4. Skrypt programu

Przed przystąpieniem do realizacji sieci nastąpiła normalizacja danych.
Normalizacja danych jest procesem przekształcania wartości zmiennych wejściowych w taki sposób, aby ich zakres był spójny. Najczęściej stosowane metody normalizacji to skalowanie min-max. Normalizacja może poprawić wydajność algorytmów uczenia maszynowego, ponieważ zapobiega sytuacjom, w których zmienne o różnych skalach mają nieproporcjonalny wpływ na wynik modelu.
Normalizację wykonano za pomocą wzoru:

$$x_{norm} = \frac{x - x_{min}}{x_{max} - x_{min}} * 2 - 1$$

gdzie:

- x_{norm} to wartość po normalizacji,
- x to wartość przed normalizacją,
- x_{min} to minimalna wartość w zbiorze,
- x_{max} to maksymalna wartość w zbiorze.

Normalizacja danych w tym przypadku jest wykonana w przedziale <-1;1> .
Dodatkowo ważnym krokiem jest posortowanie zbioru wyjściowego y_t.

```
import hickle as hkl  # Importowanie biblioteki hickle do serializacji danych
import numpy as np  # Importowanie biblioteki numpy do operacji na tablicach
from numpy import array, loadtxt, transpose  # Importowanie funkcji array, loadtxt i transpose z numpy
import matplotlib.pyplot as plt  # Importowanie biblioteki matplotlib do tworzenia wykresów

# Wczytywanie danych z pliku
filename = open("/Users/mariiarybak/Desktop/Sztuczna inteligencja/PRO/hepatitis/hepatitis.txt")
data = loadtxt(filename, delimiter=',', dtype=str)  # Wczytanie danych z pliku z użyciem przecinka jako separatora, typ danych to string

# Usunięcie rekordów z brakującymi danymi
data = data[(data == '?').any(axis=1) == 0]  # Usunięcie rekordów z brakującymi danymi

# Ustawienie cech wejściowych
x = data[:, 1:].astype(float).T  # Pobranie wszystkich kolumn poza pierwszą i zamiana ich na float, a następnie transpozycja

# Ustawienie wyjścia pożądanego
y_t = data[:, 0].astype(float)  # Pobranie pierwszej kolumny danych i zamiana na float
y_t = y_t.reshape(1, y_t.shape[0])  # Zmiana kształtu y_t na jednowierszową tablicę

# Wydrukowanie minimalnych i maksymalnych wartości dla każdej cechy przed normalizacją
print(transpose([array(range(x.shape[0])), x.min(axis=1), x.max(axis=1)]))

# Inicjalizacja wartości minimalnych i maksymalnych dla każdej cechy
x_min = x.min(axis=1)
x_max = x.max(axis=1)

# Normalizacja danych
x_norm_max = 1  # Docelowa maksymalna wartość po normalizacji
x_norm_min = -1  # Docelowa minimalna wartość po normalizacji
x_norm = np.zeros(x.shape)  # Inicjalizacja znormalizowanej tablicy o tych samych wymiarach co x
for i in range(x.shape[0]):
    x_norm[i, :] = (x_norm_max - x_norm_min) / (x_max[i] - x_min[i]) *
```

```
(x[i, :] - x_min[i]) + x_norm_min  # Normalizacja każdego wiersza

# Wydrukowanie minimalnych i maksymalnych wartości dla każdej cechy po
normalizacji
print(transpose([array(range(x.shape[0])), x_norm.min(axis=1),
x_norm.max(axis=1)]))

# Sortowanie danych
y_t_s_ind = np.argsort(y_t)  # Indeksy sortujące y_t
x_n_s = np.zeros(x.shape)  # Inicjalizacja posortowanej tablicy x
y_t_s = np.zeros(y_t.shape)  # Inicjalizacja posortowanej tablicy y_t
for i in range(x.shape[1]):
    y_t_s[0, i] = y_t[0, y_t_s_ind[0, i]]  # Sortowanie y_t zgodnie z
indeksami
    x_n_s[:, i] = x_norm[:, y_t_s_ind[0, i]]  # Sortowanie x zgodnie z
indeksami

# Rysowanie wykresu posortowanego zbioru
plt.plot(y_t_s[0])
plt.show()

# Zapisanie danych do pliku za pomocą hickle
hkl.dump([x, y_t, x_norm, x_n_s, y_t_s],
"/Users/mariiarybak/Desktop/Sztuczna
inteligencja/PRO/hepatitis/hepatitis.hkl")

# Wczytanie danych z pliku za pomocą hickle
x, y_t, x_norm, x_n_s, y_t_s =
hkl.load("/Users/mariiarybak/Desktop/Sztuczna
inteligencja/PRO/hepatitis/hepatitis.hkl")
```

Listing 1. Przygotowanie danych do użycia w modelu uczenia maszynowego

W powyższym kodzie przedstawiono proces przetwarzania i normalizacji danych oraz przygotowania ich do użycia w modelu uczenia maszynowego:

1. **Importowanie bibliotek**: kod rozpoczyna się od importowania niezbędnych bibliotek, takich jak `hickle` do serializacji danych, `numpy` do operacji na tablicach oraz `matplotlib` do tworzenia wykresów.
2. **Wczytywanie danych**: następnie, dane są wczytywane z pliku tekstowego, w którym poszczególne wartości są oddzielone przecinkami.
3. **Usuwanie brakujących danych**: kod usuwa rekordy, które zawierają brakujące dane, oznaczone jako '?'.
4. **Ustawienie cech wejściowych i wyjścia pożądanego**: dane są rozdzielane na cechy wejściowe `(x)` oraz wyjście pożądane `(y_t)`. Wyjście jest następnie przekształcane do jednowierszowej tablicy.

5. **Wydrukowanie minimalnych i maksymalnych wartości przed normalizacją**: przed przystąpieniem do normalizacji, minimalne i maksymalne wartości dla każdej cechy są drukowane, aby można było je porównać z wartościami po normalizacji.
6. **Normalizacja danych**: dane wejściowe są normalizowane do zakresu od -1 do 1. Proces ten obejmuje obliczanie minimalnych i maksymalnych wartości dla każdej cechy oraz przeskalowanie danych.
7. **Wydrukowanie minimalnych i maksymalnych wartości po normalizacji**: po normalizacji, minimalne i maksymalne wartości są ponownie drukowane, aby upewnić się, że normalizacja została przeprowadzona poprawnie.
8. **Sortowanie danych**: dane są sortowane na podstawie wartości wyjściowych `(y_t)`. Indeksy sortujące są używane do przekształcenia zarówno cech wejściowych, jak i wyjścia pożądanego.
9. **Rysowanie wykresu posortowanego zbioru**: po sortowaniu, wartości wyjściowe są rysowane na wykresie, aby wizualnie sprawdzić rozkład danych.
10. **Zapisanie danych do pliku**: znormalizowane i posortowane dane są zapisywane do pliku za pomocą biblioteki `hickle`, co umożliwia ich późniejsze szybkie wczytanie.
11. **Wczytanie danych z pliku**: na koniec, dane są wczytywane z pliku, co pokazuje, jak można je ponownie załadować do dalszego przetwarzania lub trenowania modelu.

Cały proces zapewnia, że dane są odpowiednio przygotowane i gotowe do użycia w algorytmach uczenia maszynowego, minimalizując ryzyko błędów wynikających z niekompletnych lub źle przeskalowanych danych.

Rys.7 Zbiór posortowanego zbioru wyjściowego y_t

Na wykresie przedstawiono znormalizowane wartości cech po sortowaniu zgodnie z indeksami w tablicy y_t_s. Oś X reprezentuje indeksy próbek (Indeks próbki), natomiast oś Y przedstawia znormalizowane wartości cech (Znormalizowana wartość cechy). Wykres ten pozwala zobaczyć, jak dane zmieniają się po normalizacji i sortowaniu. Wartości te są w zakresie od -1 do 1, co jest wynikiem procesu normalizacji przeprowadzonego w kodzie.

```
import warnings  # Importowanie modułu do zarządzania ostrzeżeniami
import hickle as hkl  # Importowanie hickle do ładowania danych
from timeit import default_timer as timer  # Importowanie timera do
mierzenia czasu wykonania
import numpy as np  # Importowanie numpy do operacji na tablicach
import nnet as net  # Importowanie własnego modułu sieci neuronowej
import matplotlib.pyplot as plt  # Importowanie matplotlib do tworzenia
wykresów
from sklearn.model_selection import StratifiedKFold  # Importowanie
StratifiedKFold do walidacji krzyżowej
```

Listing 2. Wykorzystane biblioteki

- **`warnings`**: w kodzie użyto `warnings.filterwarnings('ignore')`, aby wyłączyć wyświetlanie ostrzeżeń. Jest to przydatne w kontekście trenowania sieci neuronowych, gdzie mogą pojawić się liczne, nieistotne ostrzeżenia, które nie wpływają na końcowy wynik, ale mogą zaśmiecać logi i utrudniać analizę wyników.
- **`hickle`**: biblioteka jest używana do zapisywania i wczytywania danych w formacie HDF5. Format ten pozwala na efektywne przechowywanie dużych zbiorów danych numerycznych. W kodzie używamy hickle do wczytywania zestawu danych (`hepatitis.hkl`) który zawiera wejściowe cechy (`x`, `x_n_s`) oraz etykiety (`y_t`, `y_t_s`).
- **`timer`**: w kodzie użyto `default_timer z biblioteki timeit` do zmierzenia całkowitego czasu wykonania głównej pętli, która trenuje modele sieci neuronowych. Dzięki temu można ocenić wydajność kodu i określić, jak długo trwa proces trenowania modelu.
- **`numpy`**: to podstawowa biblioteka do obliczeń numerycznych w Pythonie. Oferuje wsparcie dla wielowymiarowych tablic oraz różnych funkcji matematycznych. W kodzie `numpy` jest używane do manipulacji danymi, takich jak obliczenia macierzowe (np. `np.dot`), operacje na tablicach (np. `np.squeeze`) oraz funkcje matematyczne (np. `np.isnan`).
- **`nnet`** (jako `net`): `nnet` to biblioteka(pobrana ze strony http://materialy.prz-rzeszow.pl/materialy.php?przedmiot=30. Biblioteka udostępniona przez Prowadzącego zajęcia) do tworzenia i trenowania sieci neuronowych, zawierająca funkcje specyficzne dla implementacji. W kodzie korzystamy z funkcji tej biblioteki do inicjalizacji wag (`nwtan`, `rands`), obliczeń aktywacji (`tansig`, `purelin`), obliczeń błędów (`sumsqr`, `deltalin`, `deltatan`) oraz aktualizacji wag (`learnbp`).
Poszczególny opis wszystkich funkcji znajduje się na stronie nr. 48 tego raportu.
- **`matplotlib.pyplot`** (jako `plt`): to biblioteka do tworzenia wykresów w Pythonie. W kodzie jest używana do wizualizacji wyników, takich jak wykresy zmian współczynnika uczenia (`lr_vec`) oraz błędu średniokwadratowego (`SSE_vec`) w funkcji epok.
- **`sklearn.model_selection`**: zawiera narzędzia do podziału danych na zbiory treningowe i testowe oraz do walidacji krzyżowej. W kodzie używamy

`StratifiedKFold` do podziału danych na zbiory treningowe i testowe w sposób zapewniający równomierny rozkład klas w każdym podziale.

```
class mlp_a_3w:
    def __init__(self, x, y_t, K1, K2, lr, err_goal, disp_freq,
ksi_inc, ksi_dec, er, max_epoch):
        self.x = x  # Dane wejściowe
        self.L = self.x.shape[0]  # Liczba cech wejściowych
        self.y_t = y_t  # Docelowe wyjścia
        self.K1 = K1  # Liczba neuronów w pierwszej warstwie
        self.K2 = K2  # Liczba neuronów w drugiej warstwie
        self.lr = lr  # Współczynnik uczenia
        self.err_goal = err_goal  # Cel błędu do osiągnięcia
        self.disp_freq = disp_freq  # Częstotliwość wyświetlania stanu
nauki
        self.ksi_inc = ksi_inc  # Współczynnik inkrementacji
współczynnika uczenia
        self.ksi_dec = ksi_dec  # Współczynnik dekrementacji
współczynnika uczenia
        self.er = er  # Dopuszczalna krotność przyrostu błędu
        self.max_epoch = max_epoch  # Maksymalna liczba epok uczenia
        self.K3 = y_t.shape[0]  # Liczba neuronów w warstwie trzeciej
(wyjściowej)
        self.data = self.x.T  # Inicjalizacja danych do walidacji
krzyżowej
        self.target = self.y_t  # Inicjalizacja danych docelowych
        self.SSE_vec = []  # Inicjalizacja listy do przechowywania SSE
(Sum of Squared Errors)
        self.PK_vec = []  # Inicjalizacja listy do przechowywania PK
(Poprawność Klasyfikacji)

        self.w1, self.b1 = net.nwtan(self.K1, self.L)  # Inicjalizacja
wag i biasów dla pierwszej warstwy (wejściowej)
        self.w2, self.b2 = net.nwtan(self.K2, self.K1)  # Inicjalizacja
wag i biasów dla drugiej warstwy (ukrytej)
        self.w3, self.b3 = net.rands(self.K3, self.K2)  # Inicjalizacja
wag i biasów dla trzeciej warstwy (wyjściowej)
        self.SSE = 0  # Inicjalizacja sumy kwadratów błędów (SSE)
        self.lr_vec = list()  # Inicjalizacja listy do przechowywania
współczynnika uczenia

    warnings.filterwarnings('ignore')  # Wyłączenie ostrzeżeń
```

Listing 3. Inicjalizacja sieci

Parametry wejściowe do konstruktora:
`x`: Dane wejściowe (cechy).
`y_t`: Docelowe wyjścia (etykiety).
`K1`: Liczba neuronów w pierwszej warstwie.
`K2`: Liczba neuronów w drugiej warstwie.
`lr`: Współczynnik uczenia (learning rate).
`err_goal`: Docelowy błąd do osiągnięcia podczas treningu.
`disp_freq`: Częstotliwość wyświetlania stanu nauki.
`ksi_inc`: Współczynnik inkrementacji współczynnika uczenia.
`ksi_dec`: Współczynnik dekrementacji współczynnika uczenia.
`er`: Dopuszczalna krotność przyrostu błędu.
`max_epoch`: Maksymalna liczba epok uczenia.

Inicjalizacja zmiennych:
`self.x`: Przypisanie danych wejściowych.
`self.L`: Liczba cech wejściowych (liczba kolumn w x).
`self.y_t`: Przypisanie docelowych wyjść.
`self.K1`: Przypisanie liczby neuronów w pierwszej warstwie.
`self.K2`: Przypisanie liczby neuronów w drugiej warstwie.
`self.lr`: Przypisanie współczynnika uczenia.
`self.err_goal`: Przypisanie docelowego błędu.
`self.disp_freq`: Przypisanie częstotliwości wyświetlania stanu nauki.
`self.ksi_inc`: Przypisanie współczynnika inkrementacji współczynnika uczenia.
`self.ksi_dec`: Przypisanie współczynnika dekrementacji współczynnika uczenia.
`self.er`: Przypisanie dopuszczalnej krotności przyrostu błędu.
`self.max_epoch`: Przypisanie maksymalnej liczby epok uczenia.
`self.K3`: Liczba neuronów w trzeciej warstwie (wyjściowej), określona na podstawie liczby wierszy w y_t.
`self.data`: Transponowanie danych wejściowych do postaci ‘dane x cechy’ do walidacji krzyżowej.
`self.target`: Przypisanie danych docelowych.
`self.SSE_vec`: Inicjalizacja listy do przechowywania sumy kwadratów błędów (SSE).
`self.PK_vec`: Inicjalizacja listy do przechowywania współczynnika poprawności klasyfikacji (PK).

Inicjalizacja wag i biasów:
`self.w1, self.b1`: Inicjalizacja wag i biasów dla pierwszej warstwy (wejściowej) przy użyciu funkcji net.nwtan, która prawdopodobnie inicjalizuje wagi za pomocą losowych wartości z rozkładu tangens hiperboliczny.

`self.w2, self.b2`: Inicjalizacja wag i biasów dla drugiej warstwy (ukrytej) przy użyciu funkcji net.nwtan.
`self.w3, self.b3`: Inicjalizacja wag i biasów dla trzeciej warstwy (wyjściowej) przy użyciu funkcji net.rands, która prawdopodobnie inicjalizuje wagi za pomocą losowych wartości z rozkładu jednorodnego.
`self.SSE`: Inicjalizacja sumy kwadratów błędów (SSE) na 0.
`self.lr_vec`: Inicjalizacja listy do przechowywania historii współczynnika uczenia.

W powyższym fragmencie kodu zrealizowano konstrukcję klasy mlp_a_3w, która implementuje wielowarstwową sieć neuronową uczoną algorytmem wstecznej propagacji błędu z adaptacyjnym współczynnikiem uczenia.

Konstruktor klasy `__init__`
Konstruktor klasy `mlp_a_3w` inicjalizuje wszystkie niezbędne parametry i zmienne klasy potrzebne do działania modelu. Przyjmuje dane wejściowe `x`, docelowe `wyjścia y_t`, liczbę neuronów w warstwach `K1` i `K2`, współczynnik uczenia `lr`, cel błędu `err_goal`, częstotliwość wyświetlania statusu `disp_freq`, współczynniki inkrementacji `ksi_inc` i dekrementacji `ksi_dec` współczynnika uczenia, dopuszczalną krotność przyrostu błędu `er`, oraz maksymalną liczbę epok `max_epoch`.
Inicjalizacja parametrów modelu
Parametry wejściowe są przypisywane do odpowiednich zmiennych klasy. `self.L` jest ustalane jako liczba cech wejściowych. `self.K3` jest ustalana na podstawie liczby wierszy w `y_t`, co odpowiada liczbie neuronów w warstwie wyjściowej. `self.data` jest transponowaną wersją `x`, co ułatwia przetwarzanie podczas walidacji krzyżowej. `self.target` to docelowe wyjścia.
Listy do przechowywania wyników
`self.SSE_vec` i `self.PK_vec` są inicjalizowane jako puste listy, które będą przechowywać wartości sumy kwadratów błędów (`SSE`) i współczynnika poprawności klasyfikacji (`PK`) w trakcie trenowania modelu.
Inicjalizacja wag i biasów
Wagi i biasy dla każdej warstwy są inicjalizowane za pomocą funkcji z modułu `net`. `self.w1` i `self.b1` odpowiadają za pierwszą warstwę (wejściową), `self.w2` i `self.b2` za drugą warstwę (ukrytą), a `self.w3` i `self.b3` za trzecią warstwę(wyjściową). Inicjalizowane są za pomocą funkcji zewnętrznych `net.nwtan` i `net.rands`.

Proces inicjalizacji w metodzie Nguyen-Widrow(funkcja `nwtan`)
Wagi są początkowo wybierane losowo z rozkładu jednostajnego lub normalnego. To oznacza, że wartości wag są generowane w sposób losowy z określonego przedziału lub zgodnie z określonym rozkładem prawdopodobieństwa.
Następnie, wagi są normalizowane. Normalizacja polega na podzieleniu kolejnych wartości w wierszu przez normę tego wiersza. Dzięki temu, wagi mają stałą normę, co pomaga w utrzymaniu stabilności propagacji sygnału przez sieć.
Skalowanie wag: Po normalizacji, wagi są skalowane. Współczynnik skalowania jest obliczany według wzoru:

$$magw = 0.7 * s^{\frac{1}{p}}$$

Ten krok ma na celu dostosowanie zakresu wartości wag do liczby neuronów w warstwie, co pomaga w lepszym dostosowaniu sieci do danych.
Dla warstwy wyjściowej wagi oraz przesunięcia są obliczane za pomocą funkcji „`rands`" z biblioteki „nnet". Wyznacza ona wartości wag i przesunięć jako wartości losowe z przedziału <-1; 1).

Inicjalizacja sumy kwadratów błędów i współczynnika uczenia
`self.SSE` jest początkowo ustawiona na 0, co reprezentuje początkową sumę kwadratów błędów. `self.lr_vec` jest inicjalizowana jako pusta lista, która będzie przechowywać historię współczynnika uczenia się.
Wyłączenie ostrzeżeń
Ostrzeżenia są wyłączane za pomocą warnings.`filterwarnings('ignore')`, co jest przydatne w celu uniknięcia zaśmiecania wyjścia programu przez nieistotne ostrzeżenia, które nie wpływają na działanie modelu.

```
def predict(self, x):  # Definicja metody predict, która przyjmuje dane wejściowe x
    n = np.dot(self.w1, x)  # Obliczanie wejść do pierwszej warstwy
    self.y1 = net.tansig(n, self.b1 * np.ones(n.shape))  # Aktywacja neuronów pierwszej warstwy
    n = np.dot(self.w2, self.y1)  # Obliczanie wejść do drugiej warstwy
    self.y2 = net.tansig(n, self.b2 * np.ones(n.shape))  # Aktywacja neuronów drugiej warstwy
    n = np.dot(self.w3, self.y2)  # Obliczanie wejść do trzeciej warstwy
    self.y3 = net.purelin(n, self.b3 * np.ones(n.shape))  # Aktywacja neuronów trzeciej warstwy (wyjściowej)
    return self.y3  # Zwracanie wyniku końcowego jako wynik prognozy sieci neuronowej
```

Listing 4. Funkcja obliczania prognozy

Metoda `predict` jest odpowiedzialna za generowanie prognoz na podstawie danych wejściowych za pomocą wytrenowanej sieci neuronowej. Proces ten składa się z kilku kroków:

• **Obliczenie sumy ważonej dla pierwszej warstwy ukrytej:**
Dane wejściowe są przemnażane przez wagi dla pierwszej warstwy ukrytej, a następnie dodawane są odpowiednie biasy.
• **Stosowanie funkcji aktywacji dla pierwszej warstwy ukrytej:**
Suma ważona jest poddawana funkcji aktywacji, która jest tangensem hiperbolicznym (tansig). Wynik tej operacji staje się wyjściem dla pierwszej warstwy ukrytej.
• **Obliczenie sumy ważonej dla drugiej warstwy ukrytej:**
Wyjście z pierwszej warstwy ukrytej jest przemnażane przez wagi dla drugiej warstwy ukrytej, a następnie dodawane są biasy.
• **Stosowanie funkcji aktywacji dla drugiej warstwy ukrytej:**
Podobnie jak wcześniej, suma ważona jest przekazywana przez funkcję aktywacji, aby uzyskać wyjście dla drugiej warstwy ukrytej.
• **Obliczenie sumy ważonej dla warstwy wyjściowej:**
Wyjście z drugiej warstwy ukrytej jest przemnażane przez wagi dla warstwy wyjściowej, a następnie dodawane są biasy.
• **Stosowanie funkcji aktywacji dla warstwy wyjściowej:**
Ostatecznie, suma ważona jest przekazywana przez funkcję aktywacji, która może być funkcją liniową (purelin), aby uzyskać ostateczne wyjście sieci neuronowej.
• **Zwracanie prognoz:**
Ostatnie wyjście sieci, czyli wynik z warstwy wyjściowej, jest zwracane jako prognoza dla danych wejściowych.

```
def train_CV(self, CV, skfold):
    PK_vec = np.zeros(CV)  # Inicjalizacja wektora skuteczności (PK) dla każdego folda
    for i, (train, test) in enumerate(skfold.split(self.data, np.squeeze(self.target)), start=0):  # Pętla przez każdy fold w walidacji krzyżowej
        x_train, x_test = self.data[train], self.data[test]  # Podział danych na treningowe i testowe dla bieżącego folda
        y_train, y_test = np.squeeze(self.target)[train], np.squeeze(self.target)[test]
        self.train(x_train.T, y_train.T)  # Trenowanie modelu na danych treningowych
        result = self.predict(x_test.T)  # Przewidywanie na danych testowych
        n_test_samples = test.size  # Liczba próbek testowych
        PK_vec[i] = (1 - sum((abs(result - y_test) >= 0.5).astype(int)[0]) / n_test_samples) * 100  # Obliczanie skuteczności predykcji dla bieżącego folda
        print("Test #{:<2}: PK_vec {} test_size {}".format(i, PK_vec[i], n_test_samples))
    PK = np.mean(PK_vec)  # Obliczanie średniej skuteczności predykcji z wszystkich foldów
    return PK  # Zwracanie średniej skuteczności
```

Listing 5. Walidacja kszyżowa

`def train_CV(self, CV, skfold)`: definicja metody `train_CV`, która przyjmuje dwa parametry: `CV` oznaczający liczbę foldów walidacji krzyżowej oraz `skfold`, obiekt `StratifiedKFold` do przeprowadzania walidacji krzyżowej.

`PK_vec = np.zeros(CV)`: inicjalizacja wektora skuteczności (`PK`) dla każdego folda walidacji krzyżowej. Wektor ten będzie przechowywać skuteczność predykcji dla każdego folda.

`for i, (train, test) in enumerate(skfold.split(self.data, np.squeeze(self.target)), start=0)`: pętla iterująca przez każdy fold w walidacji krzyżowej. `skfold.split` dzieli dane na zbiory treningowe i testowe, a `enumerate` dodaje licznik i, zaczynający od 0.

`x_train, x_test = self.data[train], self.data[test]`: podział danych na zbiory treningowe i testowe dla bieżącego folda. `self.data[train]` to dane treningowe, a `self.data[test]` to dane testowe.

`y_train, y_test = np.squeeze(self.target)[train], np.squeeze(self.target)[test]`: podział docelowych wyjść (etykiet) na zbiory treningowe i testowe dla bieżącego folda. `np.squeeze` usuwa jednowymiarowe wymiary z `self.target`.

`self.train(x_train.T, y_train.T)`: trenowanie modelu na danych treningowych. `x_train.T` i `y_train.T` to transponowane macierze danych treningowych i etykiet.
`result = self.predict(x_test.T)`: przewidywanie na danych testowych za pomocą wytrenowanego modelu. `x_test`.T to transponowana macierz danych testowych.
`n_test_samples = test.size`: liczba próbek testowych w bieżącym foldzie.
`PK_vec[i] = (1 - sum((abs(result - y_test) >= 0.5).astype(int)[0]) / n_test_samples) * 100`: obliczanie skuteczności predykcji (`PK`) dla bieżącego folda. `abs(result - y_test) >= 0.5` sprawdza, czy różnica między prognozą a rzeczywistą etykietą jest większa lub równa 0.5, `astype(int)` zamienia wartości na 0 lub 1, a suma tych wartości dzielona przez liczbę próbek testowych daje procent błędnych klasyfikacji. Odjęcie tej wartości od 1 i pomnożenie przez 100 daje procent poprawnych klasyfikacji.
`print("Test #{:<2}: PK_vec {} test_size {}".format(i, PK_vec[i], n_test_samples))`: wyświetlanie informacji o bieżącym foldzie, w tym numer folda i, skuteczności predykcji `PK_vec[i]` oraz liczby próbek testowych `n_test_samples`.
`PK = np.mean(PK_vec)`: obliczanie średniej skuteczności predykcji (`PK`) z wszystkich foldów walidacji krzyżowej. `np.mean(PK_vec)` zwraca średnią wartość wektora `PK_vec`.
`return PK`: Zwracanie średniej skuteczności predykcji (`PK`) z wszystkich foldów walidacji krzyżowej.

W powyższym fragmencie kodu zrealizowano metodę `train_CV`, która wykonuje walidację krzyżową na danych wejściowych. Metoda ta dzieli dane na zbiory treningowe i testowe dla każdego folda, trenuje model na danych treningowych, a następnie testuje go na danych testowych, obliczając skuteczność predykcji (`PK`) dla każdego folda.
Pętla for iteruje przez wszystkie foldy walidacji krzyżowej, dzieląc dane na zbiory treningowe i testowe. Dla każdego folda model jest trenowany na danych treningowych, a następnie używany do przewidywania na danych testowych. Liczba próbek testowych jest obliczana, a skuteczność predykcji (`PK`) jest mierzona jako procent poprawnych klasyfikacji. Wynik ten jest przechowywany w wektorze `PK_vec`.
Na końcu metoda oblicza średnią skuteczność predykcji z wszystkich foldów i zwraca ją. Wyświetlane są również informacje o skuteczności dla każdego folda, co pozwala na monitorowanie postępów walidacji krzyżowej.

```
def train(self, x_train, y_train):
    for epoch in range(1, self.max_epoch + 1):  # Pętla po epokach
treningowych
        self.y3 = self.predict(x_train)  # Przewidywanie wyjść dla
danych treningowych
        self.e = y_train - self.y3  # Obliczanie błędu
        self.SSE_t_1 = self.SSE  # Zapisywanie poprzedniej sumy błędów
SSE
        self.SSE = net.sumsqr(self.e)  # Aktualizacja sumy kwadratów
błędów (SSE)

        self.PK = (1 - sum((abs(self.e) >= 0.5).astype(int)[0]) /
self.e.shape[1]) * 100  # Obliczanie skuteczności predykcji PK
        self.PK_vec.append(self.PK)  # Dodawanie wartości PK do wektora
skuteczności

        if self.SSE < self.err_goal or self.PK == 100:  # Warunek
zakończenia: jeśli SSE jest mniejsze niż cel błędu lub PK wynosi 100%
            break
        if np.isnan(self.SSE):  # Warunek zakończenia: jeśli SSE jest
NaN
            break
        else:
            if self.SSE > self.er * self.SSE_t_1:  # Jeśli SSE jest
większe niż er * SSE z poprzedniej epoki
                self.lr *= self.ksi_dec  # Zmniejszenie współczynnika
uczenia
            elif self.SSE < self.SSE_t_1:  # Jeśli SSE jest mniejsze
niż SSE z poprzedniej epoki
                self.lr *= self.ksi_inc  # Zwiększenie współczynnika
uczenia
        self.lr_vec.append(self.lr)  # Dodawanie współczynnika uczenia
do wektora

        self.d3 = net.deltalin(self.y3, self.e)  # Obliczanie
gradientów błędów dla warstwy wyjściowej
        self.d2 = net.deltatan(self.y2, self.d3, self.w3)  # Obliczanie
gradientów błędów dla warstwy ukrytej
        self.d1 = net.deltatan(self.y1, self.d2, self.w2)  # Obliczanie
gradientów błędów dla warstwy wejściowej
        self.dw1, self.db1 = net.learnbp(x_train, self.d1, self.lr)  #
Uczenie wag i biasów dla warstwy wejściowej
        self.dw2, self.db2 = net.learnbp(self.y1, self.d2, self.lr)  #
Uczenie wag i biasów dla warstwy ukrytej
        self.dw3, self.db3 = net.learnbp(self.y2, self.d3, self.lr)  #
```

```
        Uczenie wag i biasów dla warstwy wyjściowej
        self.w1 += self.dw1  # Aktualizacja wag warstwy wejściowej
        self.b1 += self.db1  # Aktualizacja biasów warstwy wejściowej
        self.w2 += self.dw2  # Aktualizacja wag warstwy ukrytej
        self.b2 += self.db2  # Aktualizacja biasów warstwy ukrytej
        self.w3 += self.dw3  # Aktualizacja wag warstwy wyjściowej
        self.b3 += self.db3  # Aktualizacja biasów warstwy wyjściowej
        self.SSE_vec.append(self.SSE)  # Zapisywanie aktualnej wartości
SSE
```

Listing 5. Funkcja trenowania sieci

Funckja `train` służy do uczenia sieci.

`def train(self, x_train, y_train)`: definicja metody `train`, która przyjmuje dane treningowe `x_train` i odpowiadające im etykiety `y_train`. Metoda ta trenuje model sieci neuronowej na tych danych.
`for epoch in range(1, self.max_epoch + 1)`: pętla iterująca przez wszystkie epoki treningowe, od 1 do maksymalnej liczby epok `max_epoch`.
`self.y3 = self.predict(x_train)`: przewidywanie wyjść dla danych treningowych za pomocą bieżących wag i biasów modelu.
`self.e = y_train - self.y3`: obliczanie błędu jako różnicy między docelowymi wyjściami `y_train` a przewidywanymi wyjściami `self.y3`.
`self.SSE_t_1 = self.SSE`: zapisywanie poprzedniej wartości sumy kwadratów błędów (`SSE`) dla porównania w kolejnej epoce.
`self.SSE = net.sumsqr(self.e)`: aktualizacja sumy kwadratów błędów (`SSE`) na podstawie bieżącego błędu `self.e`.
`self.PK = (1 - sum((abs(self.e) >= 0.5).astype(int)[0])/self.e.shape[1])*100`: obliczanie skuteczności predykcji (`PK`). Sprawdza, ile prognoz różni się od rzeczywistych wartości o mniej niż 0.5, przelicza to na procent poprawnych klasyfikacji.
`self.PK_vec.append(self.PK)`: dodawanie bieżącej wartości `PK` do wektora skuteczności predykcji.
`if self.SSE < self.err_goal or self.PK == 100`: warunek zakończenia treningu. Sprawdza, czy `SSE` jest mniejsze niż docelowy błąd lub `PK` wynosi 100%.
`break`: przerwanie pętli, jeśli warunek zakończenia jest spełniony.
`if np.isnan(self.SSE)`: sprawdza, czy `SSE` jest `NaN` (Not a Number), co może wskazywać na problemy w obliczeniach.
`break`: przerwanie pętli, jeśli `SSE` jest `NaN`.
`if self.SSE > self.er * self.SSE_t_1`: sprawdza, czy bieżące `SSE` jest większe niż er razy poprzednie `SSE`, co wskazuje na pogorszenie się błędu.

`self.lr = self.ksi_dec`: zmniejszenie współczynnika uczenia, jeśli `SSE` się pogorszyło.
`elif self.SSE < self.SSE_t_1`: sprawdza, czy bieżące `SSE` jest mniejsze niż poprzednie `SSE`, co wskazuje na poprawę błędu.
`self.lr = self.ksi_inc`: zwiększenie współczynnika uczenia, jeśli `SSE` się poprawiło.
`self.lr_vec.append(self.lr)`: dodawanie bieżącej wartości współczynnika uczenia do wektora.
`self.d3 = net.deltalin(self.y3, self.e`
`self.d2 = net.deltatan(self.y2, self.d3, self.w3)`
`self.d1 = net.deltatan(self.y1, self.d2, self.w2):`
obliczanie gradientów błędów dla warstwy wyjściowej, ukrytej i wejściowej.
`self.dw1, self.db1 = net.learnbp(x_train, self.d1, self.lr)`
`self.dw2, self.db2 = net.learnbp(self.y1, self.d2, self.lr`
`self.dw3, self.db3 = net.learnbp(self.y2, self.d3, self.lr):`
aktualizacja wag i biasów dla warstwy wejściowej, ukrytej i wyjściowej przy użyciu obliczonych gradientów i współczynnika uczenia
`self.w1 += self.dw1`
`self.b1 += self.db1`
`self.w2 += self.dw2`
`self.b2 += self.db2`
`self.w3 += self.dw3`
`self.b3 += self.db3`:
aktualizacja wag i biasów warstwy wejściowej, ukrytej i wyjściowej poprzez dodanie obliczonych zmian `dw1, dw2, dw3, db1, db2, db3`.
`self.SSE_vec.append(self.SSE)`:dodanie bieżącej wartości `SSE` do wektora, aby móc śledzić jego zmiany w czasie trenowania.

Pętla Treningowa: metoda `train` rozpoczyna pętlę, która będzie iterować przez epoki treningowe.
Generowanie Prognoz: dla każdej epoki, najpierw generowane są prognozy sieci neuronowej dla danych treningowych przy użyciu metody `predict`.
Obliczenie Błędu: następnie obliczany jest błąd między prognozami a rzeczywistymi etykietami wg. wzoru:

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

gdzie:

- y to rzeczywisty wektor wyjściowy sieci,
- $\hat{y}$ to pożądany wektor wyjściowy.

Aktualizacja Błędu Sumy Kwadratów (SSE): błąd sumy kwadratów (SSE) jest obliczany na podstawie błędu predykcji.
Sprawdzenie Warunków Zakończenia Treningu: sprawdzane są warunki zakończenia treningu, czyli czy SSE jest mniejszy niż celowy błąd (`err_goal`) lub czy dokładność prognozowania wynosi 100%.
Aktualizacja Współczynnika Uczenia: współczynnik uczenia (`lr`) jest dynamicznie aktualizowany w zależności od poprzedniego błędu (SSE).
Obliczenie Delty dla Każdej Warstwy: następnie obliczane są delty dla każdej warstwy sieci na podstawie błędu.
Aktualizacja Wagi i Biasów: wagi i biasy są aktualizowane zgodnie z algorytmem wstecznej propagacji błędu.
Dodanie Błędu do Listy: błąd sumy kwadratów (SSE) dla danej epoki jest dodawany do listy `SSE_vec`.
Kontynuacja Treningu lub Zakończenie: proces ten jest powtarzany dla kolejnych epok, aż jeden z warunków zakończenia treningu zostanie spełniony.

5. Eksperymenty

5.1 Wyznaczenie optymalnych wartości K1 oraz K2

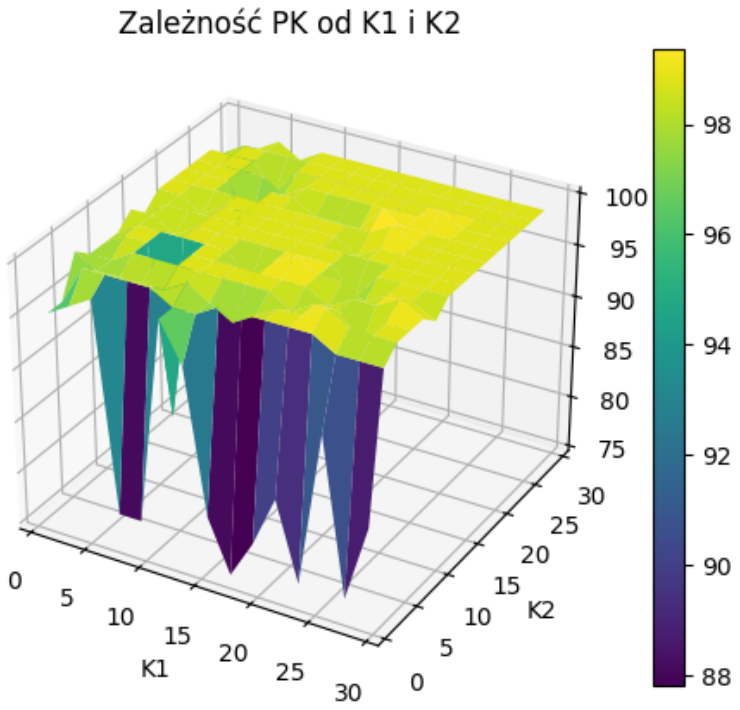
Pierwszy eksperyment polegał na wyznaczeniu optymalnych wartości parametrów **K1, K2.**

```
max_epoch = 200  # Maksymalna liczba epok treningu
err_goal = 0.25  # Docelowy błąd SSE do osiągnięcia podczas treningu
disp_freq = 10  # Częstotliwość wyświetlania statusu podczas treningu
lr = 1e-4  # Początkowy współczynnik uczenia
ksi_inc = 1.05  # Współczynnik zwiększania współczynnika uczenia
ksi_dec = 0.7  # Współczynnik zmniejszania współczynnika uczenia
er = 1.04  # Współczynnik do porównywania błędu SSE z poprzednią epoką
K1_vec = [1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29]  # Wektor
liczby neuronów w pierwszej warstwie do testowania
K2_vec = [1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29]  # Wektor
liczby neuronów w drugiej warstwie do testowania

CvN = 10  # Liczba foldów do walidacji krzyżowej
skfold = StratifiedKFold(n_splits=CVN)  # Inicjalizacja walidacji krzyżowej

start = timer()  # Rozpoczęcie mierzenia czasu wykonania
PK_2D_K1K2 = np.zeros([len(K1_vec), len(K2_vec)])  # Inicjalizacja macierzy do
przechowywania wyników PK dla różnych kombinacji K1 i K2
PK_2D_K1K2_max = 0  # Inicjalizacja zmiennej do przechowywania maksymalnej
wartości PK
k1_ind_max = 0  # Inicjalizacja indeksu K1 dla maksymalnej wartości PK
k2_ind_max = 0  # Inicjalizacja indeksu K2 dla maksymalnej wartości PK
for k1_ind in range(len(K1_vec)):  # Pętla po wszystkich wartościach K1
    for k2_ind in range(len(K2_vec)):  # Pętla po wszystkich wartościach K2
        mlpnet = mlp_a_3w(x_norm, y_t, K1_vec[k1_ind], K2_vec[k2_ind], lr,
err_goal, disp_freq, ksi_inc, ksi_dec, er, max_epoch)  # Inicjalizacja sieci
neuronowej z bieżącymi wartościami K1 i K2
        PK = mlpnet.train_CV(CVN, skfold)  # Trenowanie modelu z użyciem
walidacji krzyżowej i obliczenie skuteczności
        print("K1 {} | K2 {} | PK {}".format(K1_vec[k1_ind], K2_vec[k2_ind],
PK))  # Wyświetlanie wyników dla bieżących K1 i K2
        PK_2D_K1K2[k1_ind, k2_ind] = PK  # Przechowywanie skuteczności PK w
macierzy wyników
        if PK > PK_2D_K1K2_max:  # Sprawdzanie, czy bieżąca skuteczność PK
jest największa do tej pory
            PK_2D_K1K2_max = PK  # Aktualizacja maksymalnej skuteczności PK
            k1_ind_max = k1_ind  # Aktualizacja indeksu K1 dla maksymalnej
skuteczności
            k2_ind_max = k2_ind  # Aktualizacja indeksu K2 dla maksymalnej
skuteczności

print("Czas wykonania:", timer() - start)  # Wyświetlanie czasu wykonania
```

Listing 7. Wyznaczenie optymalnego K1 i K2

Testowane wartości dla K1 i K2 były następujące:
K1_vec = [1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29]
K2_vec = [1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29]

Danie wejściowe:
max_epoch = 5000
err_goal = 0.25
disp_freq = 10
lr = 1e-4
ksi_inc = 1.05
ksi_dec = 0.7
er = 1.04



Rys. 8 Zależność PK od K1 i K2

Dla każdej kombinacji wartości **K1** i **K2**, sieć neuronowa była trenowana przy użyciu walidacji krzyżowej z **10** foldami. Średnia skuteczność predykcji (PK) była obliczana dla każdej kombinacji, a wyniki były przechowywane w macierzy PK_2D_K1K2. Czas wykonania eksperymentu wyniósł **631.377207833** sekund.

Wyniki eksperymentu wskazują, że największa skuteczność predykcji (PK) na poziomie **100.0%** została osiągnięta dla kilku różnych kombinacji **K1** i **K2**:
K1=5,**K2**=23
K1=9,**K2**=27
K1=19,**K2**=21
K1=21,**K2**=19
K1=23,**K2**=23

Te wyniki sugerują, że umiarkowane wartości **K1 i K2** są korzystne dla osiągnięcia wysokiej skuteczności predykcji.
Dodatkowo, eksperymenty wskazują, że zbyt duża liczba neuronów w warstwach może prowadzić do przeuczenia modelu, co widać na przykładzie wartości **K1 = 29, K2 = 1,** gdzie PK spadło do **88.75%.**

Umiarkowane wartości **K1** i **K2** (w zakresie od **5** do **23**) są najczęściej związane z wysoką skutecznością predykcji. Kombinacje zbyt małej (np. **K1 = 1**) lub zbyt dużej (np. **K1 = 29**) liczby neuronów często prowadzą do niższej skuteczności.

5.2 Wyznaczenie optymalnych wartości Ir_inc i Ir_dec

Eksperyment przeprowadzono dla optymalnych wartości liczby neuronów w pierwszej (**K1**) i drugiej (**K2**) warstwie sieci neuronowej, ustalonych na podstawie wcześniejszych eksperymentów jako **K1=5** i **K2=23**.

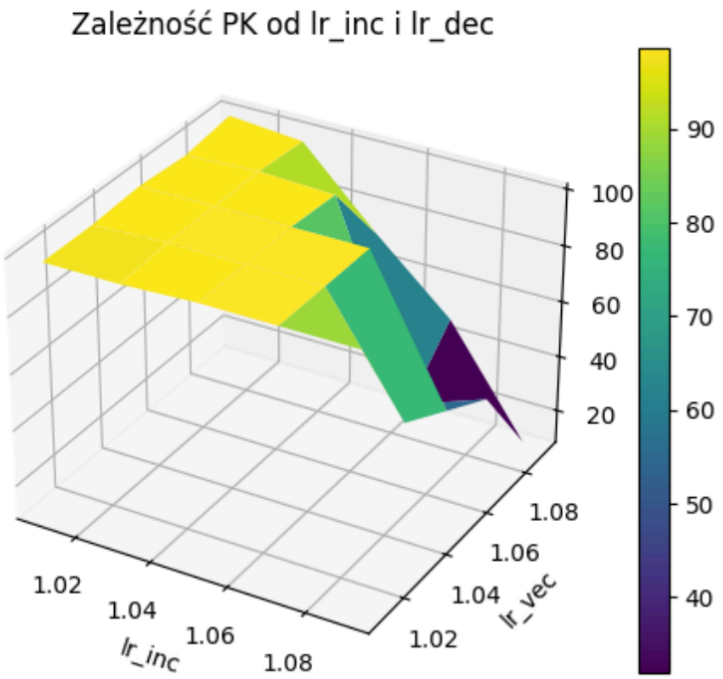
```
max_epoch = 5000  # Ustawienie maksymalnej liczby epok treningu na 5000
err_goal = 0.25  # Ustawienie docelowego błędu SSE (sumy kwadratów błędów) na
0.25
disp_freq = 10  # Ustawienie częstotliwości wyświetlania statusu podczas
treningu na co 10 epok
lr = 1e-4  # Ustawienie początkowego współczynnika uczenia na 0.0001
er = 1.04  # Ustawienie współczynnika do porównywania błędu SSE z poprzednią
epoką na 1.04
K1 = 5  # Ustawienie liczby neuronów w pierwszej warstwie na 5
K2 = 23  # Ustawienie liczby neuronów w drugiej warstwie na 23
ksi_inc_vec = [1.01, 1.03, 1.05, 1.07, 1.09]  # Lista wartości współczynnika
zwiększania współczynnika uczenia do testowania
ksi_dec_vec = [0.5, 0.6, 0.7, 0.8, 0.9]  # Lista wartości współczynnika
zmniejszania współczynnika uczenia do testowania
err = 1.05  # Ustawienie dopuszczalnej krotności przyrostu błędu na 1.05

CVN = 10  # Ustawienie liczby foldów do walidacji krzyżowej na 10
skfold = StratifiedKFold(n_splits=CVN)  # Inicjalizacja walidacji krzyżowej z
10 foldami

start = timer()  # Rozpoczęcie mierzenia czasu wykonania
PK_2D_K1K2 = np.zeros([len(ksi_inc_vec), len(ksi_dec_vec)])  # Inicjalizacja
macierzy do przechowywania wyników PK dla różnych kombinacji ksi_inc i ksi_dec
PK_2D_K1K2_max = 0  # Inicjalizacja zmiennej do przechowywania maksymalnej
wartości PK
ksi1_ind_max = 0  # Inicjalizacja indeksu ksi_inc dla maksymalnej wartości PK
ksi2_ind_max = 0  # Inicjalizacja indeksu ksi_dec dla maksymalnej wartości PK
for k1_ind in range(len(ksi_inc_vec)):  # Pętla iterująca przez wszystkie
wartości ksi_inc
    for k2_ind in range(len(ksi_dec_vec)):  # Pętla iterująca przez wszystkie
wartości ksi_dec
        mlpnet = mlp_a_3w(x_norm, y_t, K1, K2, lr, err_goal, disp_freq,
ksi_inc_vec[k1_ind], ksi_dec_vec[k2_ind], err, max_epoch)  # Inicjalizacja
sieci neuronowej z bieżącymi wartościami ksi_inc i


ksi_dec oraz pozostałymi parametrami treningu

PK = mlpnet.train_CV(CVN, skfold)  # Trenowanie modelu z użyciem walidacji
krzyżowej i obliczenie skuteczności PK
        print("ksi_inc{} | kis_dec {} | PK {}".format(ksi_inc_vec[k1_ind],
ksi_dec_vec[k2_ind], PK))  # Wyświetlanie wyników dla bieżących wartości
```

```
ksi_inc i ksi_dec oraz uzyskanej skuteczności predykcji (PK)

PK_2D_K1K2[k1_ind, k2_ind] = PK  # Przechowywanie uzyskanej skuteczności
predykcji (PK) w macierzy wyników dla bieżącej kombinacji ksi_inc i ksi_dec
        if PK > PK_2D_K1K2_max:  # Sprawdzenie, czy bieżąca skuteczność PK
jest największa do tej pory
            PK_2D_K1K2_max = PK  # Aktualizacja maksymalnej skuteczności PK
            k1_ind_max = k1_ind  # Aktualizacja indeksu ksi_inc dla
maksymalnej skuteczności
            k2_ind_max = k2_ind  # Aktualizacja indeksu ksi_dec dla
maksymalnej skuteczności
```

Listing 8. Wyznaczeni optymalnego lr_inc i lr_dec

Testowane wartości dla lr_inc i lr_dec były następujące:
ksi_inc_vec = [1.01, 1.03, 1.05, 1.07, 1.09]
ksi_dec_vec = [0.5, 0.6, 0.7, 0.8, 0.9]

Danie wejściowe:
max_epoch = 5000
err_goal = 0.25
disp_freq = 10
lr = 1e-4
er = 1.04
K1 = 5
K2 = 23
err = 1.05



Rys9. Zależność PK od lr_inc i lr_dec

Testowane wartości współczynników **lr_inc** i **lr_dec** obejmowały szeroki zakres.
Osiągnięto następujące wyniki:
ksi_inc=1.01, ksi_dec=0.5: PK=97.5%
ksi_inc=1.01, ksi_dec=0.6: PK=97.5%
ksi_inc=1.01, ksi_dec=0.7: PK=98.75%
ksi_inc=1.01, ksi_dec=0.8: PK=98.75%
ksi_inc=1.01, ksi_dec=0.9: PK=98.75%
ksi_inc=1.03, ksi_dec=0.6: PK=98.75%

ksi_inc=1.05, ksi_dec=0.5: PK=98.75%
ksi_inc=1.07, ksi_dec=0.5: PK=97.5%
ksi_inc=1.09, ksi_dec=0.5: PK=100.0%

Najwyższa skuteczność predykcji (PK) na poziomie **100.0%** została osiągnięta dla kombinacji **lr_inc = 1.09** i **lr_dec = 0.5**.

Kombinacje **lr_inc** i **lr_dec**, które dały wysoką skuteczność na poziomie ≥ **98.75%**, obejmują:
lr_inc = 1.01 i **lr_dec** w zakresie od **0.7** do **0.9.**
lr_inc = 1.03 i **lr_dec** w zakresie od **0.6** do **0.8.**
lr_inc = **1.05** i **lr_dec** w zakresie od **0.5** do **0.8.**
lr_inc = 1.07 i l**r_dec** = **0.6.**

Pewne kombinacje **lr_inc** i **lr_dec** prowadzą do znacznego spadku skuteczności predykcji. Na przykład, dla **lr_inc = 1.09** i **lr_dec = 0.9**, skuteczność PK wyniosła

tylko **10.0%**. Sugeruje to, że zbyt duża wartość **lr_inc** w połączeniu z dużą wartością **lr_dec** może destabilizować proces uczenia.

Umiarkowane wartości **lr_inc** (**1.01 do 1.05**) i **lr_dec** (**0.5 do 0.7**) są najczęściej związane z wysoką skutecznością predykcji. Wartości te pozwalają na skuteczne uczenie modelu bez ryzyka destabilizacji.

5.3 Eksperyment dla najlepszych wartości err

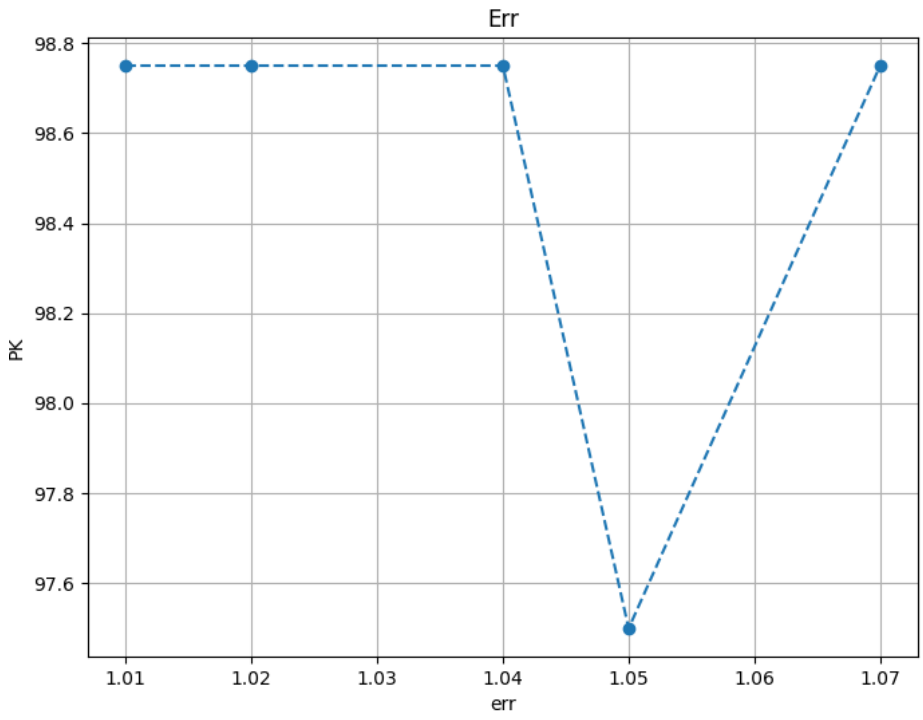
Przeprowadzono eksperyment mający na celu znalezienie optymalnej wartości współczynnika **err** przy ustalonych parametrach sieci neuronowej i warunkach treningu. Parametry **K1, K2, ksi_inc, ksi_dec** zostały ustalone na podstawie wcześniejszych eksperymentów.

```
# Ustawienie parametrów dla treningu sieci neuronowej
max_epoch = 5000  # Maksymalna liczba epok treningu
err_goal = 0.25  # Docelowy błąd SSE do osiągnięcia podczas treningu
disp_freq = 10  # Częstotliwość wyświetlania statusu podczas treningu
lr = 1e-4  # Początkowy współczynnik uczenia
er = 1.04  # Współczynnik do porównywania błędu SSE z poprzednią epoką
K1 = 5  # Liczba neuronów w pierwszej warstwie
K2 = 23  # Liczba neuronów w drugiej warstwie
ksi_inc = 1.09  # Współczynnik zwiększania współczynnika uczenia
ksi_dec = 0.5  # Współczynnik zmniejszania współczynnika uczenia
err_vec = [1.01, 1.02, 1.04, 1.05, 1.07]  # Lista wartości współczynnika er do
testowania

CVN = 10  # Liczba foldów do walidacji krzyżowej
skfold = StratifiedKFold(n_splits=CVN)  # Inicjalizacja walidacji krzyżowej z
10 foldami

start = timer()  # Rozpoczęcie mierzenia czasu wykonania
PK_2D_K1K2 = np.zeros([len(err_vec)])  # Inicjalizacja macierzy do
przechowywania wyników PK dla różnych wartości er
PK_2D_K1K2_max = 0  # Inicjalizacja zmiennej do przechowywania maksymalnej
wartości PK
k1_ind_max = 0  # Inicjalizacja indeksu dla maksymalnej wartości PK

for k1_ind in range(len(err_vec)):  # Pętla iterująca przez wszystkie wartości
er
    mlpnet = mlp_a_3w(x_norm, y_t, K1, K2, lr, err_goal, disp_freq, ksi_inc,
ksi_dec, err_vec[k1_ind], max_epoch)  # Inicjalizacja sieci neuronowej z
bieżącą wartością er oraz pozostałymi parametrami treningu
    PK = mlpnet.train_CV(CVN, skfold)  # Trenowanie modelu z użyciem walidacji
krzyżowej i obliczenie skuteczności PK
    print("err {} | PK {}".format(err_vec[k1_ind], PK))  # Wyświetlanie
wyników dla bieżącej wartości er oraz uzyskanej skuteczności predykcji (PK)
    PK_2D_K1K2[k1_ind] = PK  # Przechowywanie uzyskanej skuteczności predykcji
(PK) w macierzy wyników dla bieżącej wartości er
    if PK > PK_2D_K1K2_max:  # Sprawdzenie, czy bieżąca skuteczność PK jest
największa do tej pory
        PK_2D_K1K2_max = PK  # Aktualizacja maksymalnej skuteczności PK
        k1_ind_max = k1_ind  # Aktualizacja indeksu dla maksymalnej
skuteczności
```

Listing 9. Wyznaczeni eptymalnej wartości err

Testowane wartości dla err były następujące:
err_vec = [1.01, 1.02, 1.04, 1.05, 1.07]

Danie wejściowe:
max_epoch = 5000
err_goal = 0.25
disp_freq = 10
lr = 1e-4
er = 1.04
K1 = 5
K2 = 23
ksi_inc = 1.09
ksi_dec = 0.5



Rys. 10 Zależność PK od Err

Najwyższa skuteczność predykcji (PK) na poziomie **98.75%** została osiągnięta dla kilku wartości **err**, w tym **1.01, 1.02, 1.04, i 1.07**.
Wartość **err = 1.05** dała nieco niższą skuteczność predykcji, wynoszącą **97.5%.**
Nie ma wyraźnie jednej optymalnej wartości **err**, która daje najlepsze wyniki, ponieważ kilka wartości (**1.01, 1.02, 1.04, 1.07**) osiągnęło najwyższą skuteczność PK na poziomie **98.75%.**
Wartość **err = 1.05** osiągnęła skuteczność na poziomie **97.5%,** co jest nieco niższe niż pozostałe, ale nadal bardzo wysokie.

6. Podsumowanie i wnioski

Pod czas projektu skupiłam się na realizacji sieci neuronowej uczonej algorytmem wstecznej propagacji błędu z adaptacyjnym współczynnikiem uczenia. Głównym celem było zbadanie wpływu różnych parametrów sieci na jej zdolność do przewidywania przeżywalności osób chorych na wirusowe zapalenie wątroby na podstawie zbioru danych *Hepatitis*. W projekcie zrealizowano szereg eksperymentów, które miały na celu optymalizację parametrów sieci, takich jak liczba neuronów w warstwach ukrytych, współczynniki zwiększania i zmniejszania współczynnika uczenia oraz współczynnik błędu.

Optymalizacja liczby neuronów:
Eksperymenty wykazały, że liczba neuronów K1=5 i K2=23 była optymalna dla uzyskania wysokiej skuteczności klasyfikacji. Natomiast zbyt duża liczba neuronów prowadziła do uzyskiwania niskiej skuteczności klasyfikacji. Zbyt mała liczba neuronów uniemożliwiała efektywne uczenie.
Współczynniki adaptacyjnego uczenia:
Eksperymenty z różnymi wartościami współczynników zwiększania (ksi_inc) i zmniejszania (ksi_dec) wykazały, że umiarkowane wartości (ksi_inc = 1.09 i ksi_dec = 0.5) zapewniały stabilność i efektywność procesu uczenia, osiągając najwyższą skuteczność predykcji (PK) na poziomie 100%.
Wyższe wartości ksi_inc powodowały destabilizację procesu uczenia, co skutkowało spadkiem skuteczności PK.
Umiarkowane wartości ksi_inc (1.01 do 1.05) i ksi_dec (0.5 do 0.7) były najczęściej związane z wysoką skutecznością predykcji.
Eksperymenty z współczynnikiem błędu (err):
Testowane wartości err (1.01, 1.02, 1.04, 1.07) osiągnęły najwyższą skuteczność predykcji (PK) na poziomie 98.75%, co wskazuje na stabilność modelu przy tych ustawieniach.
Wartość err = 1.05 osiągnęła skuteczność na poziomie 97.5%, co jest nieco niższe niż pozostałe, ale nadal bardzo wysokie.

Projekt ukazał, że poprzez odpowiednią modyfikację parametrów sieci neuronowej, można osiągnąć wysoką dokładność predykcji.
Eksperymenty dostarczyły cennych informacji na temat wpływu poszczególnych parametrów na działanie sieci, co może być wykorzystane do dalszego doskonalenia modelu.
Adaptacyjne współczynniki uczenia okazały się jednymi z kluczowych dla stabilności i efektywności procesu trenowania.
Zbyt duża liczba neuronów w warstwach może prowadzić do niższej skuteczności, natomiast umiarkowane wartości K1 i K2 są najbardziej korzystne dla uzyskania wysokiej skuteczności predykcji.

Kombinacje wartości lr_inc i lr_dec muszą być starannie dobrane, aby uniknąć destabilizacji procesu uczenia.

Projekt dowiódł, że sieci neuronowe uczone algorytmem wstecznej propagacji błędu z adaptacyjnym współczynnikiem uczenia są skutecznym narzędziem do rozpoznawania przeżywalności pacjentów chorych na wirusowe zapalenie wątroby. Wyniki badań pokazują, że odpowiednia modyfikacja parametrów sieci może znacząco poprawić dokładność predykcji, co podkreśla potencjał tego podejścia w analizie danych.

7. Opis biblioteki "nnet"

"nnet" to biblioteka do tworzenia i trenowania sieci neuronowych, zawierająca funkcje specyficzne dla implementacji. W kodzie wykorzystano funkcji z tej biblioteki do inicjalizacji wag (nwtan, rands), obliczeń aktywacji (tansig, purelin), obliczeń błędów (sumsqr, deltalin, deltatan) oraz aktualizacji wag (learnbp).
Biblioteka jest pobrana ze strony http://materialy.prz-rzeszow.pl/materialy.php?przedmiot=30. Biblioteka udostępniona przez Prowadzącego zajęcia)
Niżej jest przedstawiono poszczególny opis wszystkich funkcji z biblioteki "nnet".

```
# Funkcja randX tworzy macierz o wymiarach a x b z losowymi wartościami z
przedziału [0, 1]
def randX(a, b):
    P = np.random.rand(a, b)
    return P
```

Funkcja 1. `randX(a, b)`

Generuje macierz o wymiarach a×b z losowymi wartościami z przedziału [0, 1]. Używana do inicjalizacji wag i biasów z losowymi wartościami.

```
# Funkcja randn tworzy macierz o wymiarach a x b z losowymi wartościami z
przedziału [-1, 1]
def randn(a, b):
    w = randX(a, b) * 2.0 - 1.0
    return w
```

Funkcja 2. `randn(a, b)`

Generuje macierz o wymiarach a×b z losowymi wartościami z przedziału [-1, 1]. Używana do tworzenia losowych wag i biasów o wartościach z przedziału [-1, 1].

```
# Funkcja normRows normalizuje wiersze macierzy a
def normRows(a):
    P = a.copy()  # Tworzenie kopii macierzy a
    rows, columns = a.shape  # Pobieranie liczby wierszy i kolumn
    for x in range(0, rows):  # Iteracja przez każdy wiersz
        sumSq = 0  # Inicjalizacja sumy kwadratów
        for y in range(0, columns):  # Iteracja przez każdą kolumnę w wierszu
            v = P[x, y]  # Pobieranie wartości z macierzy
            sumSq += v ** 2.0  # Dodawanie kwadratu wartości do sumy kwadratów
        len = np.sqrt(sumSq)  # Obliczanie normy euklidesowej wiersza
        for y in range(0, columns):  # Iteracja przez każdą kolumnę w wierszu
            P[x, y] = P[x, y] / len  # Normalizacja wartości w wierszu przez
normę
    return P  # Zwracanie znormalizowanej macierzy
```

Funkcja 3. `normRows(a)`

Normalizuje wiersze macierzy a tak, aby każdy wiersz miał długość 1. Upewnia się, że wagi są znormalizowane, co pomaga w stabilności i szybkości konwergencji sieci neuronowej.

```
# Funkcja sumsqr oblicza sumę kwadratów wszystkich elementów w macierzy a
def sumsqr(a):
    rows, columns = a.shape  # Pobieranie liczby wierszy i kolumn
    sumSq = 0  # Inicjalizacja sumy kwadratów
    for x in range(0, rows):  # Iteracja przez każdy wiersz
        for y in range(0, columns):  # Iteracja przez każdą kolumnę w wierszu
            v = a[x, y]  # Pobieranie wartości z macierzy
            sumSq += v ** 2.0  # Dodawanie kwadratu wartości do sumy kwadratów
    return sumSq  # Zwracanie sumy kwadratów
```

Funkcja 4. `sumsqr(a)`

Oblicza sumę kwadratów wszystkich elementów w macierzy a.
Może być używana do obliczenia całkowitego błędu w sieci neuronowej.

```
# Funkcja rands tworzy macierz wag i biasów o losowych wartościach z
przedziału [-1, 1]
def rands(a, b):
    w = randX(a, b) * 2.0 - 1.0  # Macierz wag
    b = randX(a, 1) * 2.0 - 1.0  # Macierz biasów
    return w, b  # Zwracanie macierzy wag i biasów
```

Funkcja 5. `rands(a, b)`

Tworzy macierz wag w o wymiarach a×b i macierz biasów b o wymiarach a×1, obie z losowymi wartościami z przedziału [-1, 1].
Inicjalizacja wag i biasów sieci neuronowej.

```
# Funkcja nwtan inicjalizuje wagi i biasy dla sieci neuronowej z funkcją
aktywacji tangens hiperboliczny (tansig)
def nwtan(s, p):
    magw = 0.7 * s ** (1.0 / p)  # Obliczanie skalara do skalowania wag
    w = magw * normRows(randn(s, p))  # Normalizowanie i skalowanie losowych
wag
    b = magw * randn(s, 1)  # Skalowanie losowych biasów
    rng = np.zeros((1, p))  # Tworzenie macierzy zera o wymiarach 1 x p
    rng = rng + 2.0  # Zmiana wartości macierzy na 2.0
    mid = np.zeros((p, 1))  # Tworzenie macierzy zera o wymiarach p x 1
    w = 2.0 * w / np.dot(np.ones((s, 1)), rng)  # Skalowanie wag
    b = b - np.dot(w, mid)  # Korekta biasów
    return w, b  # Zwracanie wag i biasów
```

Funkcja 6. `nwtan(s, p)`

Inicjalizuje wagi i biasy dla sieci neuronowej z funkcją aktywacji tangens hiperboliczny (tansig). Skaluje je zgodnie z odpowiednią formułą.
Tworzy odpowiednie wagi i biasy do warstwy neuronowej z tansig jako funkcją aktywacji.

```
# Funkcja nwlog inicjalizuje wagi i biasy dla sieci neuronowej z funkcją
aktywacji log-sigmoid (logsig)
def nwlog(s, p):
    magw = 2.8 * s ** (1.0 / p)  # Obliczanie skalara do skalowania wag
    w = magw * normRows(randn(s, p))  # Normalizowanie i skalowanie losowych
wag
    b = magw * randn(s, 1)  # Skalowanie losowych biasów
    rng = np.zeros((1, p))  # Tworzenie macierzy zera o wymiarach 1 x p
    rng = rng + 2.0  # Zmiana wartości macierzy na 2.0
    mid = np.zeros((p, 1))  # Tworzenie macierzy zera o wymiarach p x 1
    w = 2.0 * w / np.dot(np.ones((s, 1)), rng)  # Skalowanie wag
    b = b - np.dot(w, mid)  # Korekta biasów
    return w, b  # Zwracanie wag i biasów
```

Funkcja 7. `nwlog(s, p)`

Inicjalizuje wagi i biasy dla sieci neuronowej z funkcją aktywacji log-sigmoid (logsig). Skaluje je zgodnie z odpowiednią formułą.
Tworzy odpowiednie wagi i biasy do warstwy neuronowej z logsig jako funkcją aktywacji.

```
# Funkcja tansig implementuje funkcję aktywacji tangens hiperboliczny z
biasami
def tansig(n, b):
    n = n + b  # Dodawanie biasów do wejścia
    a = 2.0 / (1.0 + np.exp(-2.0 * n)) - 1.0  # Obliczanie tangensa
hiperbolicznego
    rows, columns = a.shape  # Pobieranie liczby wierszy i kolumn
    for x in range(0, rows):  # Iteracja przez każdy wiersz
        for y in range(0, columns):  # Iteracja przez każdą kolumnę w wierszu
            v = a[x, y]  # Pobieranie wartości z macierzy
            if np.abs(v) == np.inf:  # Sprawdzanie wartości nieskończoności
                a[x, y] = np.sign(n[x, y])  # Zastępowanie wartości
nieskończoności przez znak wejścia
    return a  # Zwracanie znormalizowanej macierzy
```

Funkcja 8. `tansig(n, b)`

Implementuje funkcję aktywacji tangens hiperboliczny (tansig) z dodanymi biasami. Używana jako funkcja aktywacji w neuronach sieci, która przekształca sumę wagowaną wejść.
Więcej o funkcjach aktywacji na stronie nr.9 tego raportu.

```
# Funkcja logsig implementuje funkcję aktywacji log-sigmoid z biasami
def logsig(n, b):
    n = n + b  # Dodawanie biasów do wejścia
    a = 1.0 / (1.0 + np.exp(-n))  # Obliczanie log-sigmoidy
    rows, columns = a.shape  # Pobieranie liczby wierszy i kolumn
    for x in range(0, rows):  # Iteracja przez każdy wiersz
        for y in range(0, columns):  # Iteracja przez każdą kolumnę w wierszu
            v = a[x, y]  # Pobieranie wartości z macierzy
            if np.abs(v) == np.inf:  # Sprawdzanie wartości nieskończoności
                a[x, y] = np.sign(n[x, y])  # Zastępowanie wartości
nieskończoności przez znak wejścia
    return a  # Zwracanie znormalizowanej macierzy
```

Funkcja 9. `logsin(n, b)`

Implementuje funkcję aktywacji log-sigmoid (logsig) z dodanymi biasami.
Używana jako funkcja aktywacji w neuronach sieci, która przekształca sumę wagowaną wejść.

```
# Funkcja purelin implementuje liniową funkcję aktywacji z biasami
def purelin(n, b):
    a = n + b  # Dodawanie biasów do wejścia
    return a  # Zwracanie macierzy
```

Funkcja 10. `purelin(n, b)`

Implementuje liniową funkcję aktywacji z dodanymi biasami.
Używana jako funkcja aktywacji w neuronach sieci, która przekształca sumę wagowaną wejść w sposób liniowy.

```
# Funkcja deltatan oblicza gradient błędu dla funkcji aktywacji tansig
def deltatan(a, d, *w):
    if not w:  # Sprawdzanie, czy macierz wag jest pusta
        d = (1.0 - (a * a)) * d  # Obliczanie gradientu błędu
    else:
        d = (1.0 - (a * a)) * np.dot(np.transpose(w[0]), d)  # Obliczanie
gradientu błędu z wagami
    return d  # Zwracanie gradientu błędu
```

Funkcja 11. `deltatan(a, d, *w)`

Oblicza gradient błędu dla funkcji aktywacji tansig. Uwzględnia wagę jeśli jest podana.
Używana w algorytmie wstecznej propagacji do obliczania gradientu błędu dla neuronów z tansig jako funkcją aktywacji.

```
# Funkcja deltalog oblicza gradient błędu dla funkcji aktywacji logsig
def deltalog(a, d, *w):
    if not w:  # Sprawdzanie, czy macierz wag jest pusta
        d = a * (1.0 - a) * d  # Obliczanie gradientu błędu
    else:
        d = a * (1.0 - a) * np.dot(np.transpose(w[0]), d)  # Obliczanie
gradientu błędu z wagami
    return d  # Zwracanie gradientu błędu
```

Funkcja 12. `deltalog(a, d, *w)`

Oblicza gradient błędu dla funkcji aktywacji logsig. Uwzględnia wagę jeśli jest podana. Używana w algorytmie wstecznej propagacji do obliczania gradientu błędu dla neuronów z logsig jako funkcją aktywacji.

```
def deltalin(a, d):
    # Zwraca gradient błędu bez modyfikacji.
    # Wykorzystywane w warstwach z liniową funkcją aktywacji, gdzie gradient
jest przekazywany bez zmian.
    return d
```

Funkcja 13. `deltalin(a, d)`

Zwraca gradient błędu dla liniowej funkcji aktywacji (bez zmian).
Używana w algorytmie wstecznej propagacji do obliczania gradientu błędu dla neuronów z liniową funkcją aktywacji (purelin).

```
def learnbp(x, d, lr):
    # Aktualizacja wag i biasów w algorytmie wstecznej propagacji błędu.
    # Skalowanie gradientu błędu przez współczynnik uczenia
    d = lr * d
    # Obliczanie zmiany wag: iloczyn gradientu błędu i transpozycji wejścia
    dw = np.dot(d, np.transpose(x))
    # Obliczanie liczby próbek w wejściu
    Q = x.shape[1]
    # Obliczanie zmiany biasów: iloczyn gradientu błędu i macierzy
jednostkowej o wymiarze Q
    db = np.dot(d, np.ones((Q, 1)))

    # Zwracanie obliczonych zmian wag i biasów
    return dw, db
```

Funkcja 14. `learnbp(x, d, lr)`

Oblicza aktualizacje wag (dw) i biasów (db) dla algorytmu wstecznej propagacji błędu przy użyciu współczynnika uczenia (lr).
Realizuje algorytm wstecznej propagacji, aktualizując wagi i biasy na podstawie gradientu błędu i współczynnika uczenia.

Cel i rola funkcji w projekcie:
Te funkcje tworzą podstawowe operacje wymagane do zbudowania, inicjalizacji, trenowania i aktualizacji sieci neuronowej. W projekcie, który dotyczy realizacji sieci neuronowej uczonej algorytmem wstecznej propagacji błędu z przyśpieszeniem metodą adaptacyjnego współczynnika uczenia do rozpoznawania żywotności badanych osób, każda z tych funkcji pełni specyficzną rolę:

- **Inicjalizacja wag i biasów:** `randX, randn, rands, nwtan, nwlog`
- **Funkcje aktywacji:** `tansig, logsig, purelin`
- **Obliczanie gradientów:** `deltatan, deltalog, deltalin`
- **Trenowanie sieci:** `learnbp`
- **Normalizacja i obliczenia pomocnicze:** `normRows, sumsqr`

Całość pozwala na stworzenie sieci neuronowej, jej inicjalizację, obliczanie odpowiedzi na wejście, aktualizację wag podczas trenowania, co jest kluczowe w procesie uczenia maszynowego.

8. Bibliografia

1. http://archive.ics.uci.edu/ml/datasets/Hepatitis
2. Zajdel R. „Ćwiczenie 4 Model Neuronu” http://materialy.prz-rzeszow.pl/pracownik/pliki/34/ĆWICZENIE 4.pdf
4. Zajdel R. „Ćwiczenie 7 Sieć neuronowa jednokierunkowa wielowarstwowa” http://materialy.prz-rzeszow.pl/pracownik/pliki/34/ĆWICZENIE 7.pdf
6. Zajdel R. „Przykładowe skrypty ” http://materialy.prz-rzeszow.pl/pracownik/pliki/34/Przykładowe skrypty 3.pdf
7. Zajdel R. „Procedura przygotowania danych dla sieci neuronowych na potrzeby projektu z modułu sztuczna inteligencja” http://materialy.prz-rzeszow.pl/pracownik/pliki/34/SI_P2__Procedura przygotowania danych.pdf
8. https://pl.wikipedia.org/wiki/Neuron_McCullocha-Pittsa
9. https://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/