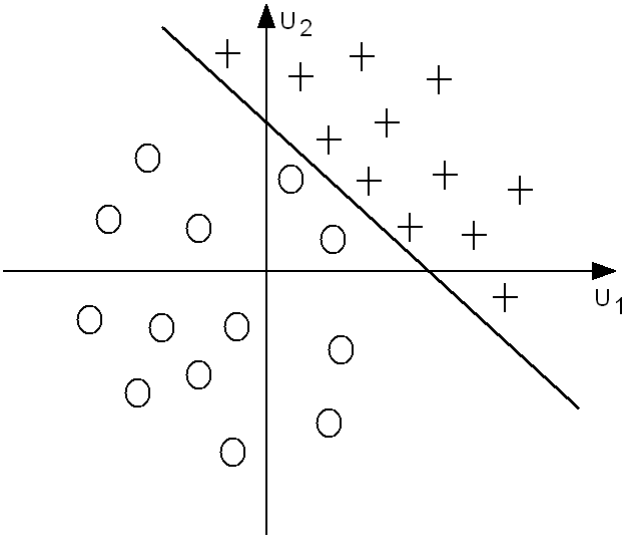
Ćwiczenie nr 5

Sztuczne sieci neuronowe

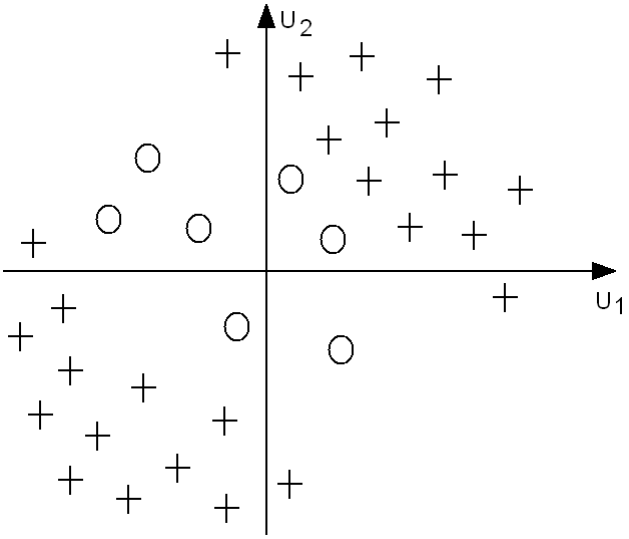
Sieci wielowarstwowe MLP (Multi Layer Perceptron), trenowanie sieci wielowarstwowych

1. Wprowadzenie

W poprzednim ćwiczeniu trenowaliśmy pojedyncze sztuczne neurony do problemów klasyfikacji. Wytrenowany perceptron umożliwiał podział przestrzeni klasyfikacyjnej na dwie klasy. W przypadku problemów liniowo separowalnych zastosowanie pojedynczego perceptronu dawało poprawne rezultaty, lecz w przypadku problemów liniowo nieseparowalnych (np. XOR) stosowanie pojedynczego perceptronu nie umożliwiało poprawnej klasyfikacji danych tam zawartych. Dla przykładu na rys. 1a przedstawiono problem liniowo separowalny a na rys. 1b problem liniowo nieseparowalny. Oba przypadki są dwuwymiarowe.

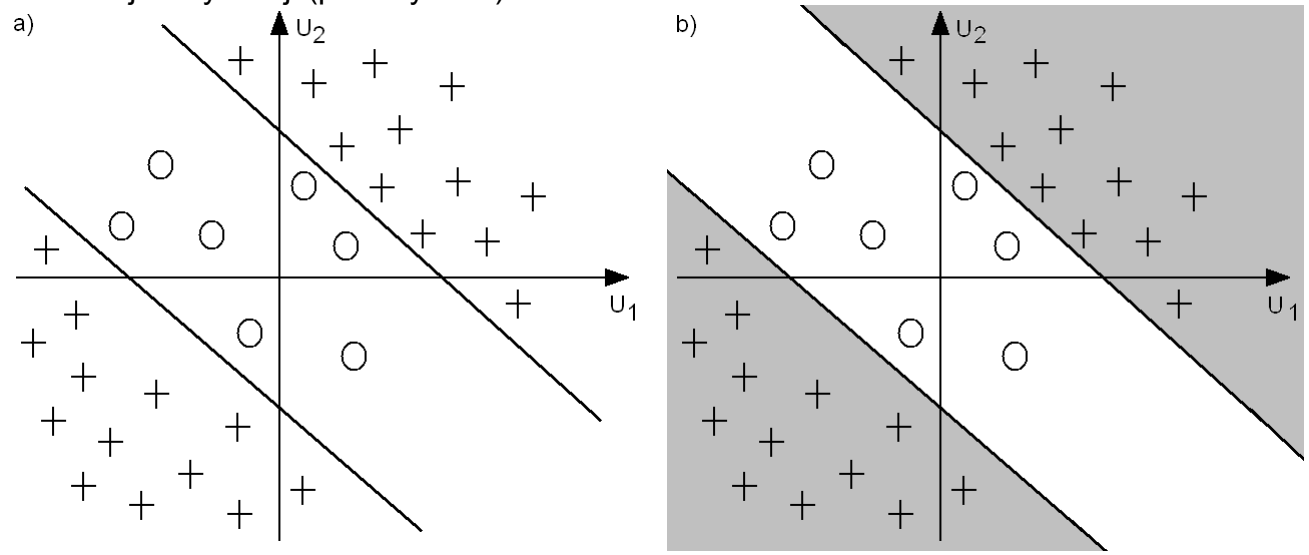


Rys. 1a – Problem liniowo separowalny



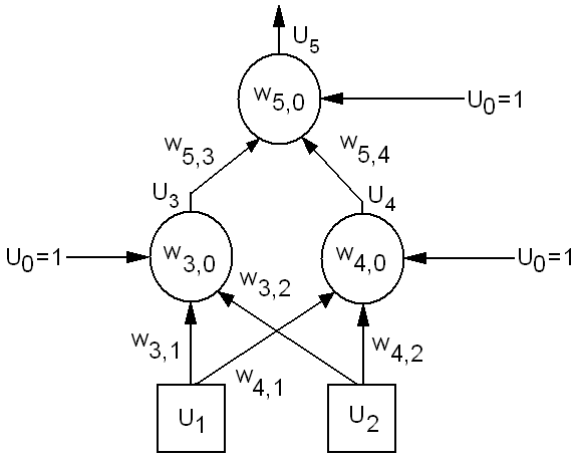
Rys. 1b – Problem liniowo nieseparowalny

Z rys. 1a widać, że do poprawnej klasyfikacji danych wystarczy jeden perceptron z odpowiednio wytrenowanymi wagami, gdyż przy użyciu jednej linii decyzyjnej klasyfikujemy dane do poprawnych klas. W przypadku przedstawionym na rys. 1b, nie jesteśmy w stanie przy użyciu jednej linii klasyfikacyjnej dokonać poprawnej klasyfikacji. Dlatego do rozwiązania tego problemu potrzebne są dwa neurony w warstwie ukrytej, które rozgraniczą te liniowo nieseparowalne obszary (patrz rys. 2a) oraz jeden neuron w warstwie wyjściowej dokonujący ostatecznej klasyfikacji (patrz rys. 2b).



Rys. 2 – Klasyfikacja problemu liniowo nieseparowalnego

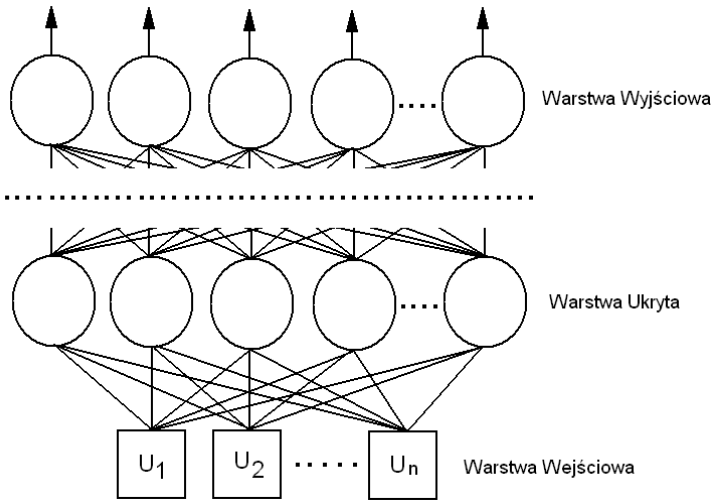
Strukturę sieci realizującą problem klasyfikacji z rys. 2a lub 2b przedstawiono na rys. 3.



Rys. 3 – Struktura sieci wielowarstwowej do klasyfikacji danych z rys. 2b

Sieć neuronowa z rys. 3 składa się z 2 wejść, 2 neuronów w warstwie ukrytej i 1 neuronu w warstwie wyjściowej, posiada 1 wyjście oraz 9 wag, których wartości muszą zostać ustalone w wyniku procesu trenowania. Dla uproszczenia w rysowaniu sieci przyjmuje się że każdy neuron zawiera w sobie blok sumujący sygnały wejściowe oraz blok zawierający funkcję aktywacji.

Widać, więc że w przypadku problemów liniowo nieseparowalnych do ich rozwiązania wymagane jest stosowanie sieci wielowarstwowych tzw. MLP (Multi Layer Perceptron). Strukturę sieci MLP przedstawiono na rys. 4.

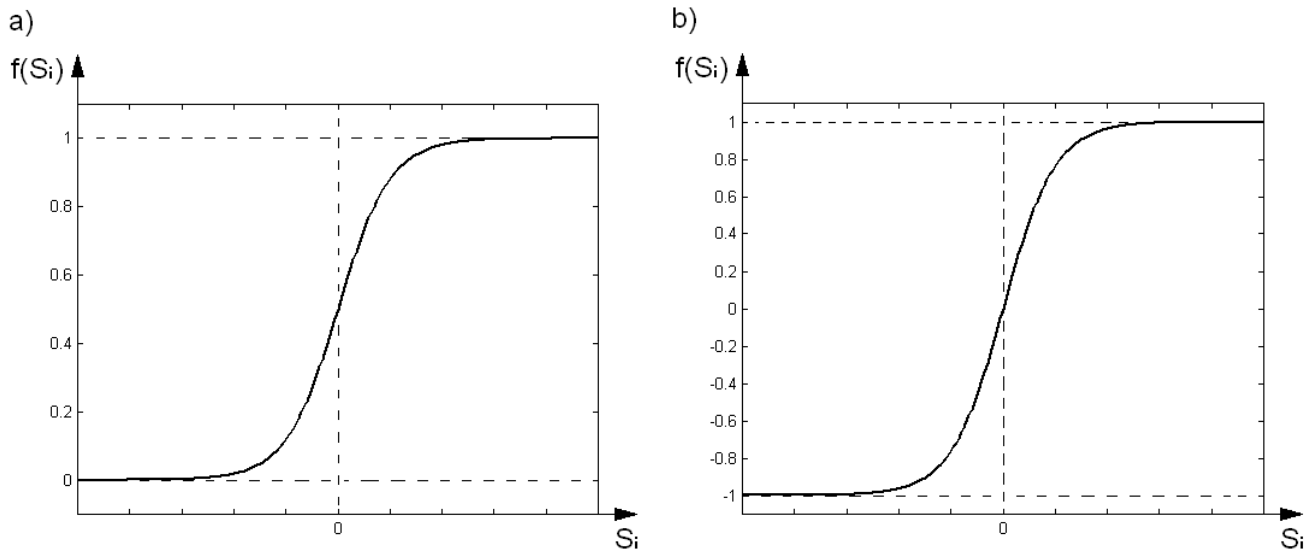


Rys. 4 – Struktura sieci MLP

Sieć MLP składa się z warstwy wejściowej (Input Layer), której zadaniem jest przekazanie na wejścia neuronów w warstwie ukrytej (Hidden Layer) sygnałów wejściowych. W warstwie ukrytej oraz wyjściowej znajdują się sztuczne neurony. Liczba warstw ukrytych, liczba sygnałów wejściowych, liczba neuronów w poszczególnych warstwach ukrytych oraz liczba neuronów w warstwie wyjściowej (Output Layer) odpowiedzialna za liczbę wyjść sieci może być dowolna. W praktyce najczęściej do większości zadań wystarczają sieci z dwoma warstwami ukrytymi. Na rys. 4 ze względu na czytelność nie oznaczono wag występujących przy poszczególnych połączeniach oraz nie oznaczono wag progowych $w_{i,0}$ (gdzie *i* oznacza numer danego neuronu)

2. Trenowanie sieci MLP – algorytm wstecznej propagacji błędu

Algorytm wstecznej propagacji błędu (ang. Error Back Propagation) jest jednym z najbardziej skutecznych i służy do trenowania jednokierunkowych (przepływ informacji w sieci odbywa się w jednym kierunku, nie występują sprzężenia), wielowarstwowych sieci neuronowych. Wykorzystuje się w nim metodę gradientowego najszybszego spadku przy minimalizacji błędu kwadratowego. Dla przeprowadzenia tego w sieci neuronowej wykorzystuje się sygnały ciągłe w zakresie [0, 1] lub w zakresie [-1, 1] i ciągłe funkcje aktywacji typu sigmoidalnego odwzorowujące ważone sumy sygnałów wejściowych w sygnały wyjściowe w zakresie [0, 1] lub [-1, 1], jak pokazano na rys. 5.



Rys. 5 – Przykłady sigmoidalnych funkcji aktywacji odwzorowujących sygnały wejściowe w ciągłe sygnały wyjściowe w zakresie a – [0, 1], b – [-1, 1]

Sigmoidalną funkcję aktywacji w postaci unipolarnej [0, 1] opisuje zależność:

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

Natomiast w postaci bipolarnej:

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

gdzie:

$$S_i = \sum_j w_{i,j} \cdot u_j \tag{3}$$

oraz:

$$u_i = f(S_i) \tag{4}$$

Algorytm wstecznej propagacji błędu przedstawiono na rys. 6.

1. Wybierz małą wartość $\rho>0$,
2. Wybierz losowo małe wartości wag $w_{i,j}$ dla każdego neuronu
3. Powtarzaj dopóki algorytm osiągnie zbieżność, tj. gdy zmiany wag i zmiany błędu kwadratowego $\varepsilon(w)$ staną się wystarczająco małe,
3a. Wybierz kolejną parę trenującą [Ek] wraz z poprawną odpowiedzią [Ck]
3b. **Faza propagacji wprzód**: oblicz kolejno dla wszystkich komórek sumy ważone S_i oraz aktywacje $u_i=f(S_i)$,
3c. **Faza propagacji wstecz**: poczynając od warstwy wyjściowej oblicz dla warstwy wyjściowej oraz warstw pośrednich pochodne funkcji aktywacji $f'(S_i)$:

$$f'(S_i)=u_i\cdot(1-u_i),\ \ dla\ \ f(S_i)=\frac{1}{1+e^{-S_i}}$$

lub

$$f'(S_i)=\frac{1}{2}\cdot(1-u_i^2),\ \ dla\ \ f(S_i)=\frac{1-e^{-S_i}}{1+e^{-S_i}}$$

oraz współczynniki $\delta_i=(C_i-u_i)\cdot f'(S_i)$, dla komórek warstwy wyjściowej
oraz:
$\delta_i=\left(\sum_{m>i}\delta_m\cdot w_{m,i}\right)\cdot f'(S_i)$, dla komórek warstw pośrednich (δ_m dotyczy komórki *m*, do której dołączona jest komórka *i*)
3d. uaktualnij wagi

$$w_{i,j}^{*}=w_{i,j}+\rho\cdot\delta_i\cdot u_j$$

Rys. 6 – Algorytm wstecznej propagacji błędu

Algorytm propagacji wstecznej stosuje się do tzw. sieci stałych, tj. takich, których struktura nie ulega zmianie w czasie trenowania. Zatem, przy rozważaniu danego problemu do rozwiązania najpierw wybiera się strukturę sieci, a następnie stosuje się trenowanie.

W celu pokazania w szczegółach jak oblicza się poszczególne współczynniki i funkcje w algorytmie rozpatrzmy sieć realizującą funkcję boolowską XOR. Strukturę sieci przyjmijmy z rys. 3, a wartości wag ustalmy następujące: $w_{3,0}=1$, $w_{3,1}=2$, $w_{3,2}=3$, $w_{4,0}=-2$, $w_{4,1}=4$, $w_{4,2}=5$, $w_{5,0}=-4$, $w_{5,3}=2$, $w_{5,4}=3$. Rozważmy jeden przykład trenujący [E] i [C] w postaci: $[E]=[u_0\ \ u_1\ \ u_2]=[1\ \ 1\ \ 0]$, $C=1$. Jako krok iteracji przyjęto wartość $\rho=0.2$, a jako funkcję aktywacji zależność (1).

Faza propagacji wprzód daje wyniki:

i	S_i	$u_i=f(S_i)$
1	-	$u_1=1$
2	-	$u_2=0$
3	$S_3=3$	$u_3=0.9526$
4	$S_4=2$	$u_4=0.8808$
5	$S_5=0.5476$	$u_5=0.6336$

a wymagana wartość u_5 wynosi $u_5=C=1$.

W fazie wstecz obliczamy kolejno:

<u>Warstwa wyjściowa:</u>

$f'(S_5)=u_5\cdot(1-u_5)=0.2322$
$\delta_5=(C-u_5)\cdot f'(S_5)=(1-0.6336)\cdot 0.2322=0.0851$

Warstwa pośrednia:

$f'(S_4)=u_4\cdot(1-u_4)=0.1050$
$\delta_4=w_{5,4}\cdot\delta_5\cdot f'(S_4)=3\cdot0.0851\cdot0.1050=0.0268$

$f'(S_3)= u_3\cdot(1-u_3)=0.9526\cdot(1-0.9526)=0.0451$
$\delta_3=w_{5,3}\cdot\delta_5\cdot f'(S_3)=2\cdot0.0851\cdot0.0451=0.0077$

Obecnie można przystąpić do obliczenia uaktualnionych wag, poczynając od wyjścia:

Warstwa wyjściowa:

$w^*_{5,4}=w_{5,4}+\rho\cdot\delta_5\cdot u_4=3+0.2\cdot0.0851\cdot0.8808=3.0150$
$w^*_{5,3}=w_{5,3}+\rho\cdot\delta_5\cdot u_3=2+0.2\cdot0.0851\cdot0.9526=2.0162$
$w^*_{5,0}=w_{5,0}+\rho\cdot\delta_5\cdot u_0=-4+0.2\cdot0.0851\cdot1=-3.9830$

Warstwa pośrednia:

$w^*_{4,2}=w_{4,2}+\rho\cdot\delta_4\cdot u_2=5+0.2\cdot0.0268\cdot0=5$
$w^*_{4,1}=w_{4,1}+\rho\cdot\delta_4\cdot u_1=4+0.2\cdot0.0268\cdot1=4.0054$
$w^*_{4,0}=w_{4,0}+\rho\cdot\delta_4\cdot u_0=-2+0.2\cdot0.0268\cdot1=-1.995$

$w^*_{3,2}=w_{3,2}+\rho\cdot\delta_3\cdot u_2=3+0.2\cdot0.0077\cdot0=3$
$w^*_{3,1}=w_{3,1}+\rho\cdot\delta_3\cdot u_1=2+0.2\cdot0.0077\cdot1=2.0015$
$w^*_{3,0}=w_{3,0}+\rho\cdot\delta_3\cdot u_0=1+0.2\cdot0.0077\cdot1=1.0015$

Z obliczeń tych wynika, że zmiany wag są niewielkie, co wskazuje, że algorytm propagacji wstecznej jest bardzo wolny.

Dla sprawdzenia o ile zmienił się sygnał wyjściowy u5 dokonujemy ponownie obliczenia fazy propagacji wprzód. Uzyskane wyniki są następujące:

$S_3= w^*_{3,0}\cdot u_0+ w^*_{3,1}\cdot u_1+ w^*_{3,2}\cdot u_2=3.003$
$u_3=f(S_3)=0.9527$

$S_4= w^*_{4,0}\cdot u_0+ w^*_{4,1}\cdot u_1+ w^*_{4,2}\cdot u_2=2.0104$
$u_4=f(S_4)=0.8819$

$S_5= w^*_{5,0}\cdot u_0+ w^*_{5,3}\cdot u_3+ w^*_{5,4}\cdot u_4=0.5968$
$u_5=f(S_5)=0.6449$

Otrzymany wzrost sygnału $\Delta u_5=0.6449 - 0.6336 = 0.0113$, co potwierdza bardzo wolną zbieżność algorytmu, gdyż u_5 powinno zmierzać do żądanej wartości wyjściowej C=1.

Jednym ze sposobów przyśpieszenia algorytmu propagacji wstecznej jest dobór większej wartości kroku ρ. Jednakże, zbyt duża jego wartość może doprowadzić do rozbieżności algorytmu. Innym sposobem przyśpieszenia, bez obawy o niestabilność, jest tzw. Metoda momentum.

Na rys. 7 przedstawiono przykładowy program trenujący sieć neuronową z rys. 3 do klasyfikacji problemu XOR, dla którego przyjęto następującą tablicę wektorów trenujących:

i	U_0	U_1	U_2	C
1	1	0	0	0
2	1	0	1	1
3	1	1	0	1
4	1	1	1	0

W programie z rys. 7 wartości wag zapisywane są w tablicy „W” w następującej kolejności:

$W=[w_{3,0}\ w_{3,1}\ w_{3,2}\ w_{4,0}\ w_{4,1}\ w_{4,2}\ w_{5,0}\ w_{5,3}\ w_{5,4}]$

Wartości sum ważonych zapisywane są w tablicy „S” jak następuje:

$S=[S_3\ S_4\ S_5]$,

Wartości funkcji aktywacji $u_i=f(S_i)$ zapisywane są w tablicy „U” w kolejności:

$U=[U_3\ U_4\ U_5]$

Wartości pochodnej funkcji aktywacji f’(Si) zapisywane są w tablicy „F” jak poniżej:

$F=[f'(S_3)\quad f'(S_4)\quad f'(S_5)]$

Wartości δ_l zapisywane są w tablicy “d” jak następuje:

$d=[\delta_3\quad \delta_4\quad \delta_5]$

Na rys. 7a przedstawiono realizację algorytmu wstecznej propagacji błędu w środowisku SCILAB.

```
//---- funkcja XOR
//---- algorytm wstecznej propagacji bledu
clear;
A(1,1)=1;    A(1,2)=0;    A(1,3)=0;    A(1,4)=0;
A(2,1)=1;    A(2,2)=0;    A(2,3)=1;    A(2,4)=1;
A(3,1)=1;    A(3,2)=1;    A(3,3)=0;    A(3,4)=1;
A(4,1)=1;    A(4,2)=1;    A(4,3)=1;    A(4,4)=0;
//---- wykreslenie obszaru klasyfikacji
Licz=0;
IleKrokow=50000;
for i=1:4
   if A(i,4)==1
      plot(A(i,2),A(i,3),'ko:');
   else
      plot(A(i,2),A(i,3),'r+:');
   end
end
mtlb_axis([-0.5 1.5 -0.5 1.5])

//---- utworzenie odpowiednich tablic na dane
W=zeros(1,9); S=zeros(1,3); U=zeros(1,3); F=zeros(1,3); d=zeros(1,3);

//---- losowa inicjalizacja wag poczatkowych
for i=1:9
   W(i)=rand()-0.5;
end

ro=0.2;
iteracja=0;
```

```
while (iteracja<IleKrokow)
iteracja=iteracja+1;
//---- losowe wybieranie wektora trenujacego
i=round(rand()*3)+1;

//---- faza propagacji w przod - warstwa posrednia
S(1)=W(1)*A(i,1)+W(2)*A(i,2)+W(3)*A(i,3);
S(2)=W(4)*A(i,1)+W(5)*A(i,2)+W(6)*A(i,3);

U(1)=1/(1+exp(-S(1)));
U(2)=1/(1+exp(-S(2)));

//---- faza propagacji w przod - warstwa wyjsciowa
S(3)=W(7)*A(i,1)+W(8)*U(1)+W(9)*U(2);
U(3)=1/(1+exp(-S(3)));

//---- faza propagacji wstecz - warstwa wyjsciowa
F(3)=U(3)*(1-U(3));
d(3)=(A(i,4)-U(3))*F(3);

//---- faza propagacji wstecz - warstwa posrednia
F(1)=U(1)*(1-U(1));
d(1)=W(8)*d(3)*F(1);
F(2)=U(2)*(1-U(2));
d(2)=W(9)*d(3)*F(2);

//---- uaktualnienie wag - warstwa wyjsciowa
W(7)=W(7)+(ro*d(3)*A(i,1));
W(8)=W(8)+(ro*d(3)*U(1));
W(9)=W(9)+(ro*d(3)*U(2));

//---- uaktualnienie wag - warstwa posrednia
W(1)=W(1)+(ro*d(1)*A(i,1));
W(2)=W(2)+(ro*d(1)*A(i,2));
W(3)=W(3)+(ro*d(1)*A(i,3));

W(4)=W(4)+(ro*d(2)*A(i,1));
W(5)=W(5)+(ro*d(2)*A(i,2));
W(6)=W(6)+(ro*d(2)*A(i,3));
end

//---- wykreslenie otrzymanej linii podzialu (neuron 1)
k=0;
for i=-2:0.01:2
k=k+1;
XX(k)=i;
YY(k)=-((W(2)/W(3))*i)-(W(1)*1)/W(3);
end
plot(XX,YY,'r');

//---- wykreslenie otrzymanej linii podzialu (neuron 2)
k=0;
for i=-2:0.01:2
k=k+1;
XX(k)=i;
YY(k)=-((W(5)/W(6))*i)-(W(4)*1)/W(6);
end
plot(XX,YY,'b');
mtlb_axis([-0.5 1.5 -0.5 1.5]);
```

Rys. 7a – Algorytm propagacji wstecznej do klasyfikacji problemu XOR (2 wejściowego) – realizacja w środowisku SCILAB

Na rys. 7b ukazano realizację algorytmu wstecznej propagacji błędu w środowisku Spyder (język Python).

```
#----funkcja XOR
#---- algorytm wstecznej propagacji bledu
import numpy as np
import matplotlib.pyplot as plt
import math
A=np.zeros((4,4))
A[0,0]=1; A[0,1]=0; A[0,2]=0; A[0,3]=0
A[1,0]=1; A[1,1]=0; A[1,2]=1; A[1,3]=1
A[2,0]=1; A[2,1]=1; A[2,2]=0; A[2,3]=1
A[3,0]=1; A[3,1]=1; A[3,2]=1; A[3,3]=0
#---- wykreslenie obszaru klasyfikacji
Licz=0
IleKrokow=50000
for i in range(0,4):
    if A[i,3]==1:
        plt.plot(A[i,1],A[i,2],'ko:')
    else:
        plt.plot(A[i,1],A[i,2],'r+:')
plt.axis([-0.5,1.5,-0.5,1.5])

#---- utworzenie odpowiednich tablic na dane
W=np.zeros((9));S=np.zeros((3));U=np.zeros((3));F=np.zeros((3));d=np.zeros((3))

#---- losowa inicjalizacja wag poczatkowych
for i in range(0,9):
    W[i]=np.random.rand()-0.5

ro=0.2
iteracja=0

while (iteracja<IleKrokow):
    iteracja=iteracja+1;

#---- losowe wybieranie wektora trenujacego
    i=np.random.randint(4)

#---- faza propagacji w przod - warstwa posrednia
    S[0]=W[0]*A[i,0]+W[1]*A[i,1]+W[2]*A[i,2]
    S[1]=W[3]*A[i,0]+W[4]*A[i,1]+W[5]*A[i,2]

    U[0]=1/(1+math.exp(-S[0]))
    U[1]=1/(1+math.exp(-S[1]))

#---- faza propagacji w przod - warstwa wyjsciowa
    S[2]=W[6]*A[i,0]+W[7]*U[0]+W[8]*U[1]
    U[2]=1/(1+math.exp(-S[2]))

#---- faza propagacji wstecz - warstwa wyjsciowa
    F[2]=U[2]*(1-U[2])
    d[2]=(A[i,3]-U[2])*F[2]

#---- faza propagacji wstecz - warstwa posrednia
    F[0]=U[0]*(1-U[0])
    d[0]=W[7]*d[2]*F[0]
    F[1]=U[1]*(1-U[1])
    d[1]=W[8]*d[2]*F[1]

#---- uaktualnienie wag - warstwa wyjsciowa
    W[6]=W[6]+ro*d[2]*A[i,0]
    W[7]=W[7]+ro*d[2]*U[0]
    W[8]=W[8]+ro*d[2]*U[1]
```

```
#---- uaktualnienie wag - warstwa posrednia
    W[0]=W[0]+ro*d[0]*A[i,0]
    W[1]=W[1]+ro*d[0]*A[i,1]
    W[2]=W[2]+ro*d[0]*A[i,2]

    W[3]=W[3]+ro*d[1]*A[i,0]
    W[4]=W[4]+ro*d[1]*A[i,1]
    W[5]=W[5]+ro*d[1]*A[i,2]

#---- wykreslenie otrzymanej linii podzialu (neuron 1)
XX=np.zeros((401))
YY=np.zeros((401))
k=0
for i in np.arange(-2,2.01,0.01):
    XX[k]=i
    YY[k]=-((W[1]/W[2])*i)-(W[0]*1)/W[2]
    k=k+1
plt.plot(XX,YY)

#---- wykreslenie otrzymanej linii podzialu (neuron 2)
k=0
for i in np.arange(-2,2.01,0.01):
    XX[k]=i
    YY[k]=-((W[4]/W[5])*i)-(W[3]*1)/W[5]
    k=k+1
plt.plot(XX,YY)
plt.axis([-0.5,1.5,-0.5,1.5])
```

Rys. 7b – Algorytm propagacji wstecznej do klasyfikacji problemu XOR (2 wejściowego) – realizacja w środowisku Spyder (język Python)

3. Modyfikacja algorytmu wstecznej propagacji błędu metoda momentum RHW

Metoda ta jest modyfikacją algorytmu wstecznej propagacji błędu i polega ona na dodawaniu do aktualizowanych wag części ich poprzedniego przyrostu $\Delta w_{i,j}$. W takim przypadku krok 3d w algorytmie z rys. 6 zastępuje się przez:

3d:

$$w_{i,j}^{*} = w_{i,j} + \alpha \cdot \Delta w_{i,j} + \rho \cdot \delta_i \cdot u_j \tag{5}$$

gdzie:

$$\Delta w_{i,j} = w_{i,j}^{*} - w_{i,j} \tag{6}$$

jest przyrostem tego kroku.
Jako praktyczną wartość współczynnika α przyjmuje się: α=0.9.

4. Zadania do wykonania

a) zapoznać się z zagadnieniami bieżącego ćwiczenia

b) przepisać i uruchomić program (rys. 7) trenujący algorytmem wstecznej propagacji błędu sieć neuronową z rys. 3 do klasyfikacji problemu XOR

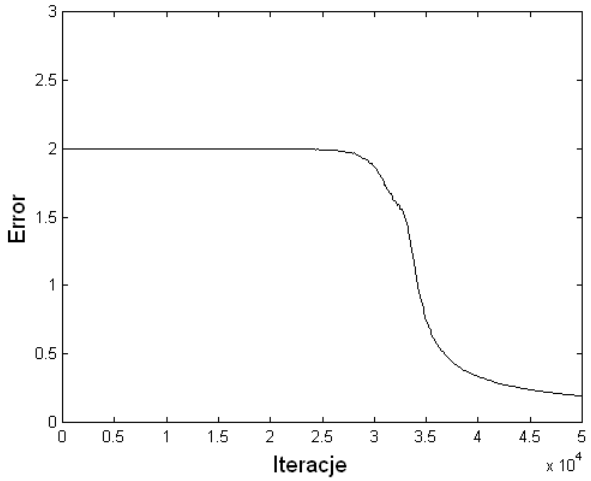
c) dodać do programu z rys. 7 możliwość testowania wytrenowanej sieci, tzn. umożliwić po wytrenowaniu i wyrysowaniu linii klasyfikujących wprowadzanie danych z klawiatury na wejścia sieci (u_1 i u_2), a następnie dla tak podanych danych wejściowych wyświetlać odpowiedź sieci. Wprowadzanie należy zapętlić, aby możliwe było wielokrotne powtarzanie wprowadzania danych na wejście wytrenowanej sieci neuronowej.

d) dopisać do programu z rys. 7 fragment kodu odpowiedzialny za wykreślenie błędu uczenia sieci. Jako błąd uczenia przyjąć zależność:

$$Error = \sum_{i=1}^{4} \left| A(i,4) - U(3)_i \right| \tag{7}$$

gdzie: A(i,4) oznacza poprawną odpowiedź dla i-tego wektora trenującego; $U(3)_i$ oznacza wartość wyjścia sieci neuronowej dla i-tego wektora trenującego.

Następnie w załóżmy co 100 iteracji obliczać błąd i zapisywać go do tablicy „ERR”. Po zakończeniu procesu trenowania wykreślić ten błąd w funkcji kolejnych iteracji. Wykres powinien być podobny do wykresu z rys. 8.



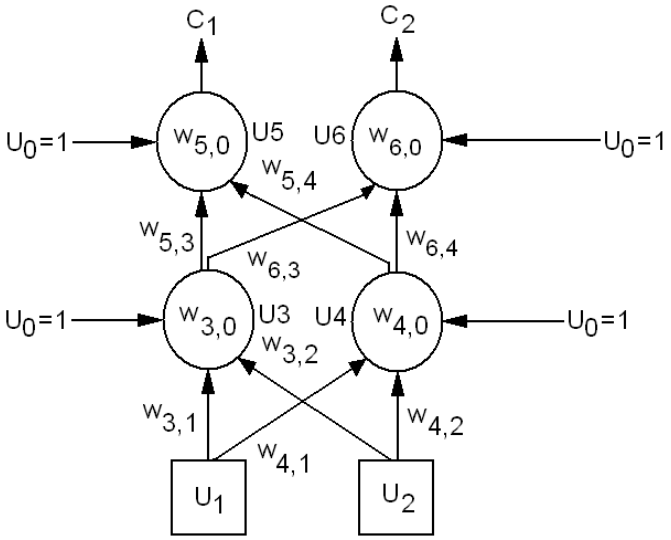
Rys. 8 – Przykładowy wykres błędu uczenia sieci neuronowej

e) do programu z rys. 7 dopisać fragment kodu realizujący metodę momentum RHW (patrz punkt 3) i przeprowadzić ponownie trenowanie sieci do klasyfikacji problemu XOR

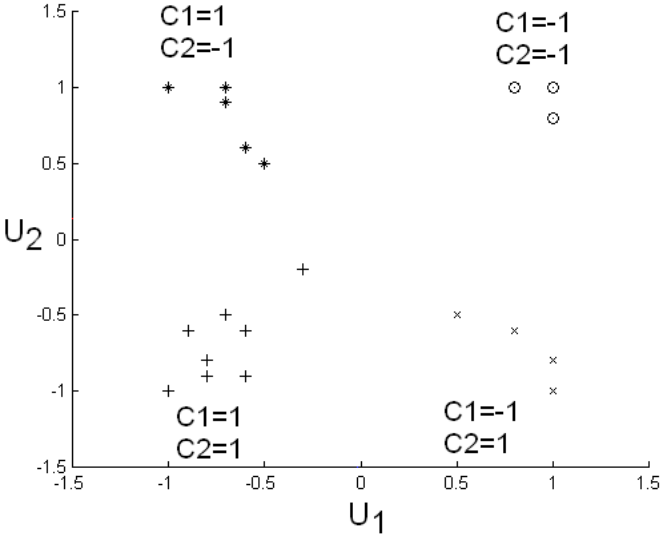
f). dla algorytmu powstałego w punkcie 4e wyznaczyć błąd trenowania sieci neuronowej. Przyjąć identyczną zależność na obliczenie błędu jak w punkcie 4d. Po otrzymaniu wykresu błędu trenowania, porównać go z wykresem otrzymanym w punkcie 4d.

g) dla programu powstałego w punkcie 4f dodać możliwość skalowania rozmiaru sieci w zależności od podanego parametru n. Dla przykładu jeśli użytkownik poda wartość n=4 program powinien samodzielnie utworzyć tablicę z wektorami trenującymi dla funkcji XOR z 4 wejściami. Tablica ta dla n=4 będzie posiadała 6 kolumn (U_0, U_1, U_2, U_3, U_4, C) i 16 wierszy. Sieć neuronowa będzie posiadać wówczas 4 wejścia, 4 neuronowy w warstwie ukrytej oraz 1 neuron w warstwie wyjściowej. Po wytrenowaniu sieci neuronowej program powinien umożliwiać użytkownikowi sprawdzenie poprawności jej działania poprzez wprowadzenie wartości wejść z klawiatury a następnie poprzez podanie odpowiedzi wygenerowanej przez wytrenowaną sieć neuronową.

h). Wytrenować sieć neuronową z rys. 9 do klasyfikacji problemu przedstawionego na rys. 10. Zastosować algorytm wstecznej propagacji błędu z momentum RHW.



Rys. 9 – Struktura sieci do wytrenowania



Rys. 10 – Problem do klasyfikacji

Poniżej w tabeli przedstawiono tablicę wektorów trenujących dla problemu z rys. 10:

i	U_0	U_1	U_2	C_1	C_2
1	1	1	1	-1	-1
2	1	1	0.8	-1	-1
3	1	0.8	1	-1	-1
4	1	1	-1	-1	1
5	1	0.8	-0.6	-1	1
6	1	1	-0.8	-1	1
7	1	0.5	-0.5	-1	1
8	1	-0.5	0.5	1	-1
9	1	-0.6	0.6	1	-1
10	1	-0.7	0.9	1	-1
11	1	-0.7	1	1	-1
12	1	-1	1	1	-1
13	1	-1	-1	1	1
14	1	-0.9	-0.6	1	1
15	1	-0.7	-0.5	1	1
16	1	-0.3	-0.2	1	1
17	1	-0.6	-0.6	1	1
18	1	-0.8	-0.8	1	1
19	1	-0.8	-0.9	1	1
20	1	-0.6	-0.9	1	1

Po wytrenowaniu narysować linie klasyfikacyjne dla neuronów U3 i U4 oraz dopisać fragment kodu odpowiedzialny za testowanie wytrenowanej sieci neuronowej.