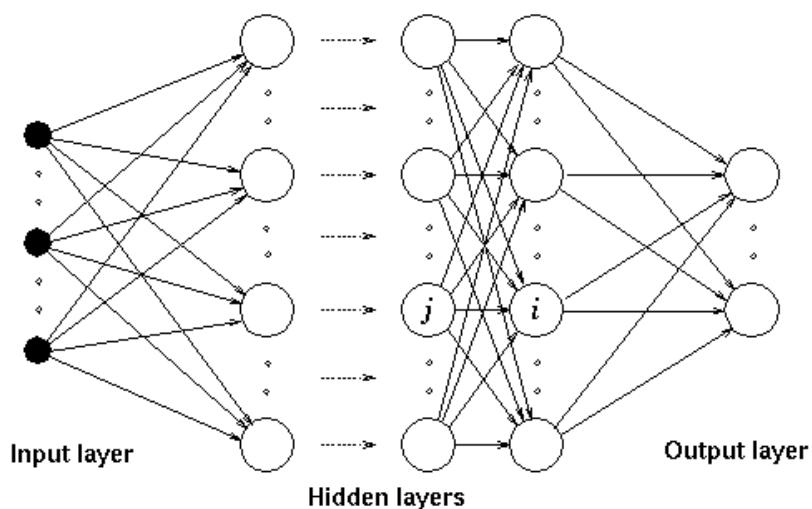


## Lab 3 - Wielowarstwowe sieci jednokierunkowe i propagacja wsteczna

### 1. Wielowarstwowe sieci jednokierunkowe

Wielowarstwowe sieci jednokierunkowe, zwane także wielowarstwowymi perceptronami (ang. *MLP* – *Multilayer Perceptron*), składają się z kilku warstw neuronów, a sygnały przepływają przez nie w jednym kierunku, od warstwy wejściowej przez warstwy ukryte do warstwy wyjściowej.



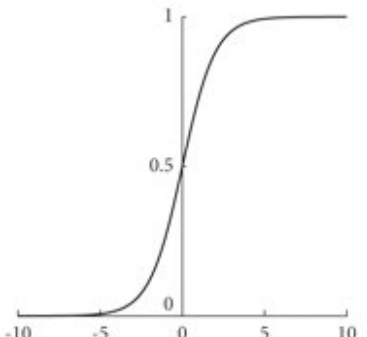
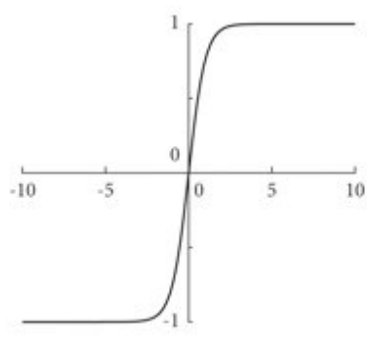
Sieć składa się z **warstwy wejściowej**, jednej lub więcej **warstw ukrytych** oraz **warstwy wyjściowej**. Warstwa wejściowa przyjmuje dane wejściowe, które następnie są przetwarzane przez warstwy ukryte, a wynik jest przekazywany do warstwy wyjściowej, która generuje końcowy wynik modelu. Każdy neuron w warstwie otrzymuje sygnały z poprzedniej warstwy, które są ważone i sumowane. Następnie wynik sumowania jest przekształcany przez funkcję aktywacji. Sieć uczy się, minimalizując funkcję błędu (np. błąd średniokwadratowy lub entropię krzyżową) za pomocą algorytmu propagacji wstecznej.

## 2. Funkcje aktywacyjne

Funkcje aktywacyjne wprowadzają nieliniowość do obliczeń, dzięki czemu sieć może modelować skomplikowane wzorce w danych. Każdy neuron w sieci przekształca swoje wejścia za pomocą funkcji aktywacyjnej, co decyduje, jakie informacje zostaną przekazane do następnej warstwy.

W celu uczenia sieci przy użyciu algorytmu wstecznej propagacji błędu konieczne jest obliczanie pochodnych cząstkowych względem wag, co wymaga stosowania ciągłych i różniczkowalnych funkcji aktywacji. W praktyce najczęściej stosowane są funkcje aktywacyjne sigmoidalne. Neurony z funkcją sigmoidalną mają konstrukcję podobną do perceptronu, z tą różnicą, że zamiast progowej funkcji aktywacji, stosuje się tutaj ciągłe funkcje sigmoidalne. Wśród nich wyróżniamy sigmoidalne funkcje unipolarne (wartości wyjściowe od 0 do 1) oraz bipolarne (wartości wyjściowe od -1 do 1).

Najczęściej stosowane są sigmoid i tanh:

funkcja	wykres	postać	Pierwsza pochodna
sigmoid		$\sigma(x) = \frac{1}{1+e^{-\beta x}}$	$\sigma'(x) = \beta \cdot \sigma(x) \cdot (1 - \sigma(x))$
tanh		$\tanh(x) = \frac{e^{\beta x} - e^{-\beta x}}{e^{\beta x} + e^{-\beta x}}$	$\tanh'(x) = \beta \cdot (1 - \tanh^2(x))$

Parameter  $\beta$  pozwala na modyfikację nachylenia funkcji sigmoidalnych.

Popularność tych funkcji spowodowana jest łatwością obliczania ich pochodnych, co jest konieczne przy użyciu algorytmów uczenia opartych na podejściu gradientowym.

### 3. Algorytm propagacji wstecznej - wstęp

Algorytm propagacji wstecznej (ang. *backpropagation*) jest jedną z podstawowych metod uczenia sieci neuronowych.

Ideą algorytmu propagacji wstecznej jest, na podstawie obliczonego błędu (lub straty), ponowne dostosowanie wartości wag w każdej warstwie – od ostatniej aż do warstwy wejściowej sieci. Najpierw obliczamy różnicę między przewidywanym wynikiem sieci a rzeczywistą wartością, a następnie aktualizujemy wszystkie wartości wag, od ostatniej warstwy aż do pierwszej, zawsze dążąc do zmniejszenia błędu.

W ujęciu matematycznym algorytm propagacji wstecznej określa strategię doboru wag w sieci wielowarstwowej, wykorzystując **gradientowe metody optymalizacji**. Algorytm ten można przedstawić w postaci następujących kroków:

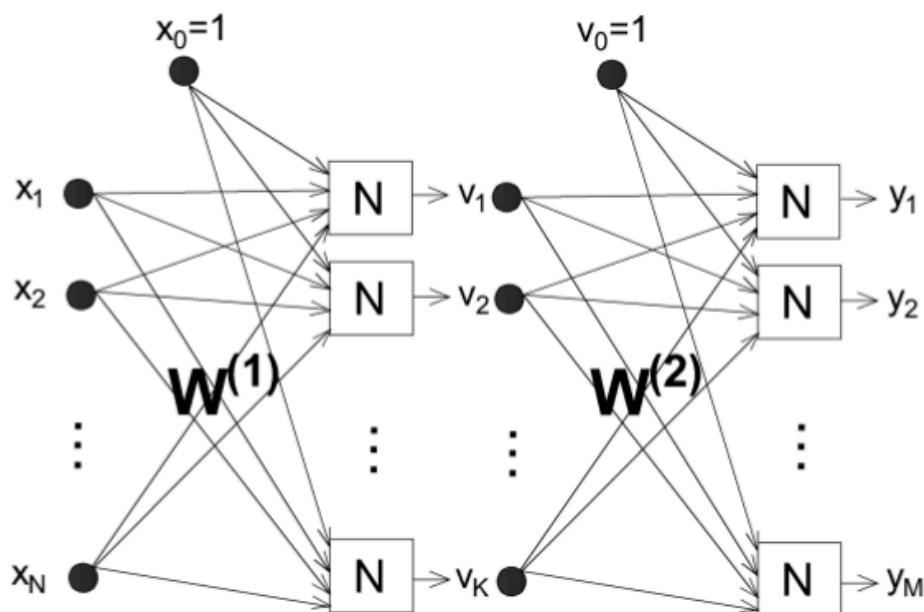
1. **Inicjalizacja wag:** Wszystkie wagi należy zainicjować małymi, losowymi wartościami.
2. **Obliczenie błędu:** Następnie wprowadza się dane do sieci, aby obliczyć wartość funkcji błędu przez porównanie przewidywanej wartości wyjściowej z wartością oczekiwaną. Istotne jest, aby funkcja błędu była różniczkowalna.
3. **Minimalizacja błędu:** Aby zminimalizować błąd, oblicza się **gradient funkcji błędu** względem każdej wagi. Z rachunku różniczkowego wiadomo, że wektor gradientu wskazuje kierunek największego wzrostu funkcji. Ponieważ wagi powinny zmieniać się w kierunku największego spadku funkcji błędu, należy podążać w kierunku przeciwnym do gradientu.
4. **Aktualizacja wag:** Po obliczeniu wektora gradientu każdą wagę aktualizuje się iteracyjnie. Gradienty są przeliczane na początku każdej iteracji treningowej, aż błąd zmniejszy się poniżej ustalonego progu lub osiągnięta zostanie maksymalna liczba iteracji.

**Gradient** to wektor składający się z pochodnych funkcji względem wszystkich jej zmiennych – wskazuje, jak funkcja zmienia się we wszystkich możliwych kierunkach jednocześnie.

**Gradient funkcji błędu względem wag** to zbiór pochodnych, które pokazują, jak zmienia się błąd w odpowiedzi na zmianę każdej wagi. Dzięki temu możemy określić, które wagi należy dostosować i o ile, aby zminimalizować błąd. Jeśli element gradientu dla danej wagi jest dodatni, oznacza to, że zwiększenie tej wagi zwiększy błąd, a zmniejszenie – zmniejszy błąd.

#### 4. Algorytm propagacji wstecznej - przykład

Rozważmy przykład sieci z jedną warstwą ukrytą, której schemat wraz z oznaczeniami przedstawia poniższy rysunek:



Oznaczenia:

- Wektor sygnałów wejściowych  $x = [1, x_1, x_2, \dots, x_N]$
- Macierz  $W^{(1)}$  o rozmiarach  $K \times (N + 1)$  zawiera wartości wag między sygnałami wejściowymi a warstwą ukrytą, gdzie  $w_{kn}^{(1)}$  oznacza wagę pomiędzy  $k$ -tym neuronem warstwy ukrytej a  $n$ -tym sygnałem wejściowym.
- Macierz  $W^{(2)}$  o rozmiarach  $M \times (K + 1)$  zawiera wartości wag między warstwą ukrytą a warstwą wyjściową, gdzie  $w_{mk}^{(2)}$  oznacza wagę pomiędzy  $m$ -tym neuronem warstwy wyjściowej a  $k$ -tym neuronem warstwy ukrytej.
- Wektor pożądanych odpowiedzi (danych treningowych dla wyjścia sieci)  $d = [1, d_1, d_2, \dots, d_M]$

Funkcje błędu zdefiniujemy jako MSE (błąd średniokwadratowy):

$$E(W) = \frac{1}{2} \sum_{m=1}^M [y_m - d_m]^2$$

Przy czym  $y_m$  wyraża się wzorem:

$$y_m = G \left( \sum_{k=0}^K w_{mk}^{(2)} \cdot v_k \right) = G \left( \sum_{k=0}^K w_{mk}^{(2)} \cdot F \left( \sum_{n=0}^N w_{kn}^{(1)} \cdot x_n \right) \right)$$

Najpierw wejścia  $x_n$  są sumowane w warstwie ukrytej z wagami  $w_{kn}^{(1)}$  i przekształcane funkcją aktywacji  $F$ , dając wartości  $v_k$ . Następnie te wartości  $v_k$  są sumowane w warstwie wyjściowej z wagami  $w_{mk}^{(2)}$  i przekształcane funkcją aktywacyjną  $G$ , co daje wynik  $y_m$ .

Proces uczenia odbywa się poprzez aktualizację wag. Poprawa wag odbywa się w kierunku przeciwnym do gradientu. Dla warstwy wyjściowej  $W^{(2)}$ :

$$w_{mk}^{(2)} = w_{mk}^{(2)} - \eta \cdot \frac{\partial E(W)}{\partial w_{mk}^{(2)}}$$

gdzie,  $\eta$  - współczynnik uczenia.

Gradient z poprzedniego równania może być zdefiniowany używając reguły łańcuchowej jako:

$$\frac{\partial E(W)}{\partial w_{mk}^{(2)}} = \frac{\partial E(W)}{\partial d_m} \frac{\partial d_m}{\partial w_{mk}^{(2)}}$$

Pochodna cząstkowa funkcji błędu względem warstwy wyjściowej  $d_m$ :

$$\frac{\partial E(W)}{\partial d_m} = \frac{\partial}{\partial d_m} \left( \frac{1}{2} \sum_{m=1}^M [y_m - d_m]^2 \right) = y_m - d_m$$

a dla  $d_m$  względem  $w_{mk}^{(2)}$ :

$$\frac{\partial d_m}{\partial w_{mk}^{(2)}} = \frac{\partial}{\partial w_{mk}^{(2)}} \left( G \left( \sum_{k=0}^K w_{mk}^{(2)} \cdot v_k \right) \right) = G' \left( \sum_{k=0}^K w_{mk}^{(2)} \cdot v_k \right) \cdot v_k$$

Zatem gradient ostatecznie ma postać:

$$\frac{\partial E(W)}{\partial w_{mk}^{(2)}} = (y_m - d_m) \cdot G' \left( \sum_{k=0}^K w_{mk}^{(2)} \cdot v_k \right) \cdot v_k$$

Dla warstwy  $W^{(1)}$  aktualizacja wag odbywa się podobnie jak dla warstwy  $W^{(2)}$ :

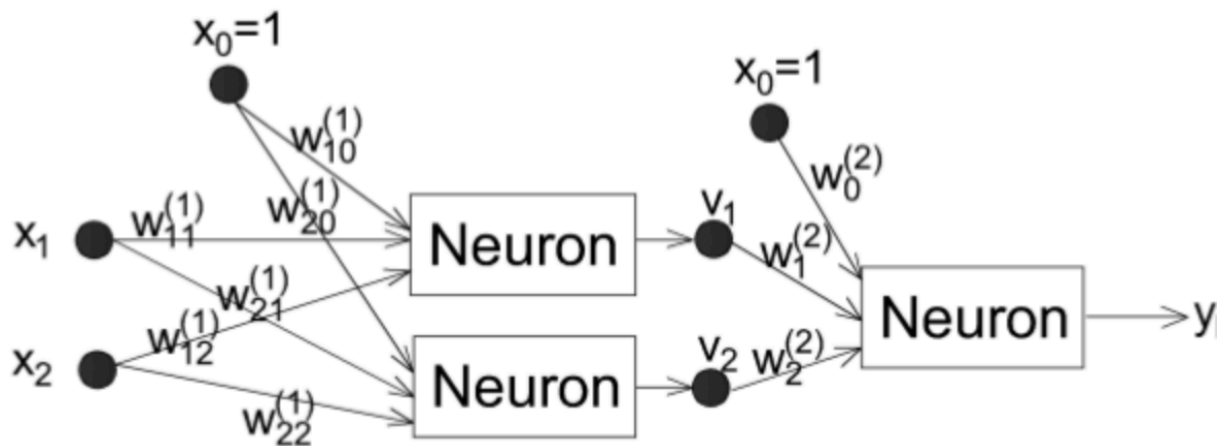
$$w_{kn}^{(1)} = w_{kn}^{(1)} - \eta \cdot \frac{\partial E(W)}{\partial w_{kn}^{(1)}}$$

Gradient dla  $W^{(1)}$  może być zdefiniowany jako:

$$\frac{\partial E(W)}{\partial w_{kn}^{(1)}} = \sum_{m=1}^M \left[ (y_m - d_m) \cdot G' \left( \sum_{k=0}^K w_{mk}^{(2)} \cdot v_k \right) \cdot w_{mk}^{(2)} \cdot F' \left( \sum_{n=0}^N w_{kn}^{(1)} \cdot x_n \right) \cdot x_n \right]$$

### 5. Zadanie 1

Zaimplementuj (w postaci funkcji) sieć neuronową o architekturze przedstawionej na poniższym rysunku.



- Dla neuronów warstwy ukrytej jako funkcję aktywacji przyjmij bipolarną funkcję sigmoidalną.
- Dla neuronu warstwy wyjściowej jako funkcję aktywacji przyjmij unipolarną funkcję sigmoidalną.
- Implementację wykonaj na reprezentacji wektorowo-macierzowej.
- Funkcja powinna zwracać odpowiedź sieci oraz wszystkie składniki potrzebne do obliczania pochodnych.

```
import numpy as np

def sigmoid(x, beta):
    return 1.0/(1.0+np.exp(-beta*x))

def tanh(x, beta):
    return np.tanh(beta*x)

# x - sygnał wejściowy [1,x1,x2,...,xN]
# w1 - wagi warstwy ukrytej, macierz
# w2 - wagi warstwy wyjściowej, wektor
# beta - parametr funkcji aktywacji, mogą być dwa różne
def mlp(x, w1, w2, beta):
    pass
```

Protips:

1. Zwróć uwagę, że wejściu dla warstwy wyjściowej jest  $x_0 = 1$ .
2. Można dodać do funkcji warunki sprawdzające poprawność wymiarów macierzy wag i sygnałów wejściowych.
3. Do obliczenia pochodnych potrzebujemy wyjść warstwy ukrytej.

## 6. Zadanie 2

Zaimplementuj algorytm wstecznej propagacji błędu dla sieci z poprzedniego zadania, stosując dwa warianty aktualizacji wag:

- aktualizacja wag po każdej próbce uczącej,
- aktualizacja wag po każdej epoce (po przetworzeniu wszystkich próbek uczących).

Następnie stwórz wykres przedstawiający zmiany wartości błędu sieci podczas uczenia (w zależności od epoki) dla problemu klasyfikacji XOR (dane podano poniżej).

Oba modele należy trenować aż do osiągnięcia błędu klasyfikacji równego 0 (lub maksymalnie przez 100 000 epok), przy zastosowaniu następujących zasad klasyfikacji:

- jeśli wyjście sieci jest większe niż 0,9, uznajemy, że próbka należy do klasy 1,
- jeśli wyjście sieci jest mniejsze niż 0,1, uznajemy, że próbka należy do klasy 0,
- w pozostałych przypadkach uznajemy, że sieć nie zwraca jednoznacznej odpowiedzi.

```
import numpy as np
import matplotlib.pyplot as plt

xx = np.array([[1,0,0],[1,0,1],[1,1,0],[1,1,1]])
d = np.array([0, 1, 1, 0])

def sigmoid_diff(y, beta):
    return beta*y*(1-y)

def tanh_diff(y, beta):
    return beta*(1-y*y)

def train_sample(xx, d, eta, beta):
    pass

def train_epoch(xx, d, eta, beta):
    pass
```

*Protips:*

1. Aby zbyt często nie utykać w minimach lokalnych, warto zamienić zerowe sygnały wejściowe na wartości niezerowe np. podane niżej dane uczące `xx` warto zamienić na `xx = np.array([[1,-1,-1],[1,-1,1],[1,1,-1],[1,1,1]])`.
2. Zwróć uwagę, że wejściu dla warstwy wyjściowej jest  $x_0 = 1$  - przy liczeniu gradientu należy "odciąć" bias i pobrać wartości wyjść tylko neuronów warstwy ukrytej.
3. Gradient dla warstwy ukrytej powinien mieć rozmiar (2,1).
4. Przy aktualizacji wag po każdej epoce należy aktualizować wagi na podstawie skumulowanych gradientów.