Politechnika Wrocławska
Wydział Informatyki i Telekomunikacji

Sztuczna inteligencja

Sprawozdanie z laboratorium

Autor

Aleksandra Walczybok

nr albumu: **272454**

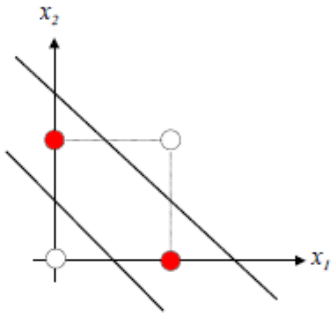
kierunek: **Inżynieria systemów**

1 grudnia 2024

1 Wstęp

Celem niniejszego sprawozdania jest przedstawienie procesu tworzenia i treningu sieci neuronowej typu perceptron wielowarstwowy (MLP) w celu rozwiązania klasycznego problemu logicznego, jakim jest funkcja XOR.

Funkcja XOR, będąca przykładem problemu nieliniowego, jest powszechnie wykorzystywana do testowania zdolności sieci neuronowych do modelowania złożonych zależności.



Rysunek 1: Wykres funkcji XOR

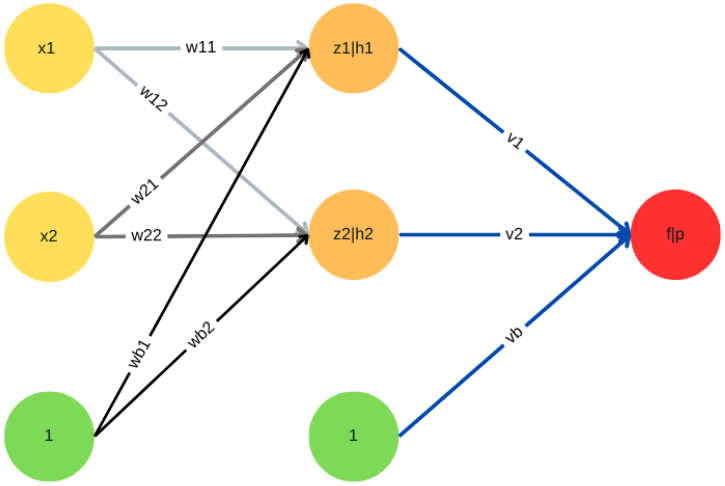
A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

Tabela 1: Tabela prawdy funkcji XOR

W ramach tego sprawozdania, krok po kroku, zostanie przedstawiony proces budowy sieci MLP, jej treningu przy użyciu algorytmu propagacji wstecznej oraz oceny jej skuteczności w rozwiązywaniu problemu XOR. Analizie poddane zostaną również aspekty dotyczące doboru hiperparametrów sieci oraz optymalizacji procesu uczenia.

2 Opis rozwiązania

Jak widać na rysunku 1 XOR nie jest funkcją liniowo separowalną, co oznacza, że nie istnieje pojedyncza linia, która rozdzielałaby jej klasy. Z tego powodu do rozwiązania problemu potrzebne jest zaimplementowanie **MLP**, czyli złożonej sieci neuronowej składającej się z minimum jednej warstwy ukrytej. Mając na wejściu 2 neurony oraz 2 neurony w warstwie ukrytej, otrzymano dwie linie proste, które są w stanie odpowiednio rozdzielać klasy. Łącznie wykorzystano 6 neuronów, ponieważ uwzględniono również bias. Jako funkcję aktywacji wybrano sigmoidę. Budowę sieci przedstawiono na rysunku poniżej.



Rysunek 2: Sieć neuronowa

2.1 Opis matematyczny

2.1.1 Macierze wag w sieci neuronowej

1. Macierz wag dla pierwszej warstwy ukrytej w_1 (łącząca x_1 i x_2 z h_1 oraz h_2):

$$W_1 = \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \end{bmatrix}$$

2. Macierz wag W_2:

$$W_2 = \begin{bmatrix} v_1 \\ v_2 \end{bmatrix}$$

3. Macierz biasu dla pierwszej warstwy:

$$b_1 = \begin{bmatrix} wb_1 & wb_2 \end{bmatrix}$$

4. Macierz biasu dla drugiej warstwy:

$$b_2 = \begin{bmatrix} v_b \end{bmatrix}$$

2.1.2 Forward propagation

Dla warstw ukrytych h_1 i h_2:

$$z_1 = w_{11}x_1 + w_{12}x_2 + w_{b1}$$

$$z_2 = w_{21}x_1 + w_{22}x_2 + w_{b2}$$
$$h_1 = \sigma(z_1)$$
$$h_2 = \sigma(z_2)$$

gdzie $\sigma(z) = \frac{1}{1+e^{-z}}$ to funkcja sigmoidalna.

Dla wyjścia p:

$$f = v_1h_1 + v_2h_2 + v_b$$
$$p = \sigma(f)$$

2.1.3 Funkcja kosztu

Jako funckję kosztu wybrano *cross-entropy*. Jest ona szczególnie efektywna w zadaniach klasyfikacji, zarówno dla klasyfikacji binarnej, jak i wieloklasowej.

Funkcja *cross-entropy* silnie penalizuje błędne przewidywania, szczególnie w przypadkach, gdy przewidywana wartość jest bardzo daleka od rzeczywistej etykiety. Dzięki temu proces uczenia jest bardziej ukierunkowany na poprawę jakości przewidywań w kolejnych iteracjach.

Dla klasyfikacji binarnej, funkcja kosztu *cross-entropy* dla całego zbioru danych N jest definiowana jako:

$$L = -\frac{1}{N}\sum_{i=1}^{N}(y_i \log(\hat{y}_i) + (1-y_i)\log(1-\hat{y}_i))$$

gdzie:

- N to liczba przykładów w zbiorze treningowym,
- y_i to prawdziwa etykieta dla i-tego przykładu,
- $\hat{y}_i$ to przewidywane prawdopodobieństwo dla i-tego przykładu.

2.1.4 Backpropagation

Aby sieć dostosowała się do danych, należy przeprowadzić proces backpropagation. Jest to algorytm, który pozwala na obliczenie gradientów funkcji kosztu względem wag i biasów w sieci neuronowej. Na podstawie obliczonych gradientów wagi i biasy są aktualizowane w kierunku zmniejszenia błędu.

Poniżej przedstawiono wzory obliczeniowe dla dwuwarstwowej sieci neuronowej.

1. Gradient wag dla warstwy wyjściowej (W_2):

$$\frac{\partial L}{\partial W_2} = H^T(P - Y)$$

gdzie:

- L to funkcja kosztu,
- H to wyjścia warstwy ukrytej,
- P to przewidywane wyjście sieci,

- Y to rzeczywiste etykiety.

2. Gradient wag dla warstwy ukrytej (W_1):

$$D_1 = (P - Y)W_2^T$$
$$D_2 = D_1 \odot f'(Z)$$
$$\frac{\partial L}{\partial W_1} = X^T D_2$$

gdzie:

- $f'(Z)$ to pochodna funkcji aktywacyjnej,
- X to wejście do sieci.

3. Gradient biasu dla warstwy wyjściowej (b_2):

$$\frac{\partial L}{\partial b_2} = \sum (P - Y)$$

4. Gradient biasu dla warstwy ukrytej (b_1):

$$\frac{\partial L}{\partial b_1} = \sum D_2$$

Następnie z wykorzystaniem obliczonych gradientów należy zaaktualizować wagi.

Dodatkowo trzeba również zdefiniować współczynnik uczenia (η), który kontroluje tempo, z jakim model aktualizuje swoje wagi podczas procesu uczenia.

Wzory na aktualizacje wag bez momentum:

1. Aktualizacja wag dla warstwy wyjściowej (W_2):

$$W_2 \leftarrow W_2 - \eta \frac{\partial L}{\partial W_2}$$

2. Aktualizacja wag dla warstwy ukrytej (W_1):

$$W_1 \leftarrow W_1 - \eta \frac{\partial L}{\partial W_1}$$

3. Aktualizacja biasów dla warstwy wyjściowej (b_2):

$$b_2 \leftarrow b_2 - \eta \frac{\partial L}{\partial b_2}$$

4. Aktualizacja biasów dla warstwy ukrytej (b_1):

$$b_1 \leftarrow b_1 - \eta \frac{\partial L}{\partial b_1}$$

Drugim sposobem na zaaktualizowanie wag jest zastosowanie momentum.

Momentum wprowadza dodatkowy składnik v, który akumuluje wcześniejsze gradienty, co pozwala na szybsze i bardziej stabilne uczenie. Wzory dla aktualizacji wag z użyciem momentum są następujące:

1. Aktualizacja prędkości (v):

$$v_t = \beta v_{t-1} + (1 - \beta)\nabla L$$

gdzie:

- v_t to prędkość w kroku t,
- β to współczynnik momentum ($\beta \in [0, 1]$),
- ∇L to gradient funkcji kosztu w bieżącym kroku.

2. Aktualizacja wag (W):

$$W_t = W_{t-1} - \eta v_t$$

gdzie:

- η to współczynnik uczenia (*learning rate*),
- W_t to wagi w kroku t,
- v_t to prędkość w kroku t.

3 Trening i wybór parametrów

Trening sieci neuronowej składa się z kluczowych etapów, takich jak propagacja w przód (forward propagation), obliczanie wartości funkcji kosztu, wsteczna propagacja błędu (backpropagation) oraz aktualizacja wag. Każdy z tych elementów został szczegółowo opisany powyżej.

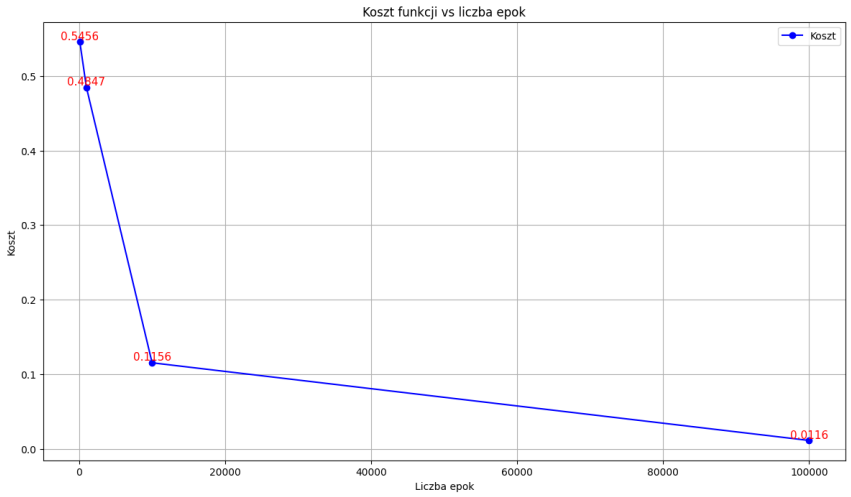
Nie mniej istotnym aspektem procesu uczenia jest dobór hiperparametrów, które mają kluczowy wpływ na skuteczność i szybkość działania modelu.

3.1 Liczba epok

Liczba epok w treningu sieci neuronowej odnosi się do liczby pełnych przejść przez cały zbiór danych treningowych, jakie wykonuje model podczas procesu uczenia.

Każda epoka pozwala modelowi dostosować wagi, aby lepiej dopasować się do danych treningowych.

Poniżej ukazano wykres na którym widać jak liczba epok wpływa na średni koszt podczas uczenia zbioru treningowego.



Rysunek 3: Koszt vs liczba epok

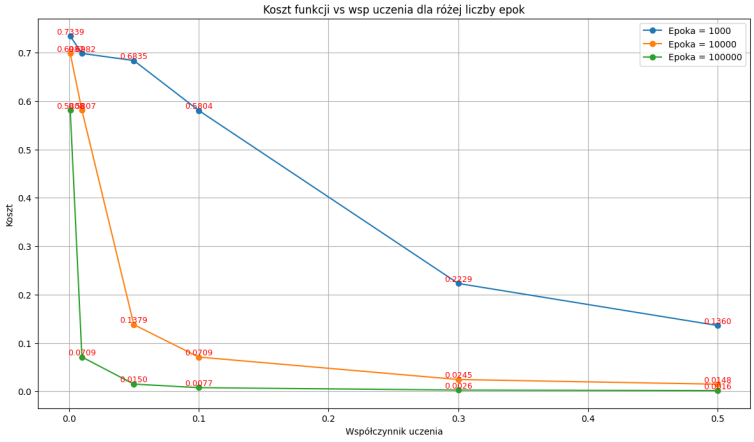
Widzimy zatem, że im więcej epok, tym koszt jest mniejszy. Podczas uczenia większych sieci neuronowych należy uważać na zjawisko przuczenia, ponieważ przy dużej liczbie epok model może zbyt dopasować się do danych treningowych i nie być w stanie dobrze klasyfikować nowych, nieznanych mu danych (overfitting).

3.2 Współczynnik uczenia

Współczynnik uczenia (η) to kluczowy hiperparametr, który kontroluje, jak duże kroki wykonuje model w kierunku minimalizacji funkcji kosztu podczas aktualizacji wag. Jego wartość musi być odpowiednio dobrana – zbyt mały współczynnik powoduje wolne uczenie, a zbyt duży może prowadzić do niestabilności i braku konwergencji modelu.

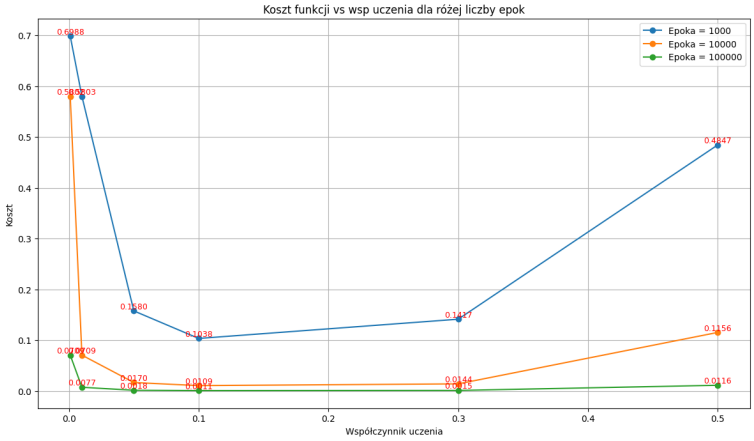
Zależności funkcji kosztu od różnych wartości współczynnika pokazano na wykresach:

1. Bez momentum



Rysunek 4: Koszt vs współczynnik uczenia

2. Z momentum ustalonym na wartość domyślną 0.9

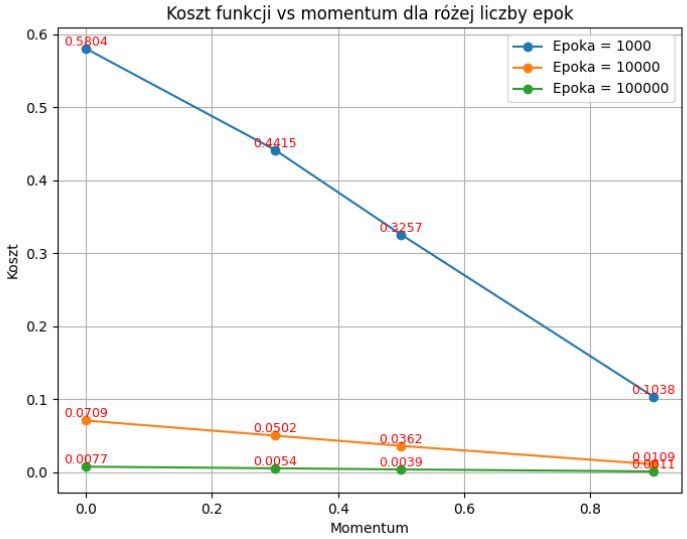


Rysunek 5: Koszt vs współczynnik uczenia

Dla przykładu bez używania momentum (czyli wartość momentum była równa 0) widać, że najlepszy współczynnik to wartość 0.5, natomiast z momentum najlepszy wynik osiągają modele dla wartości 0.1.

3.3 Momentum

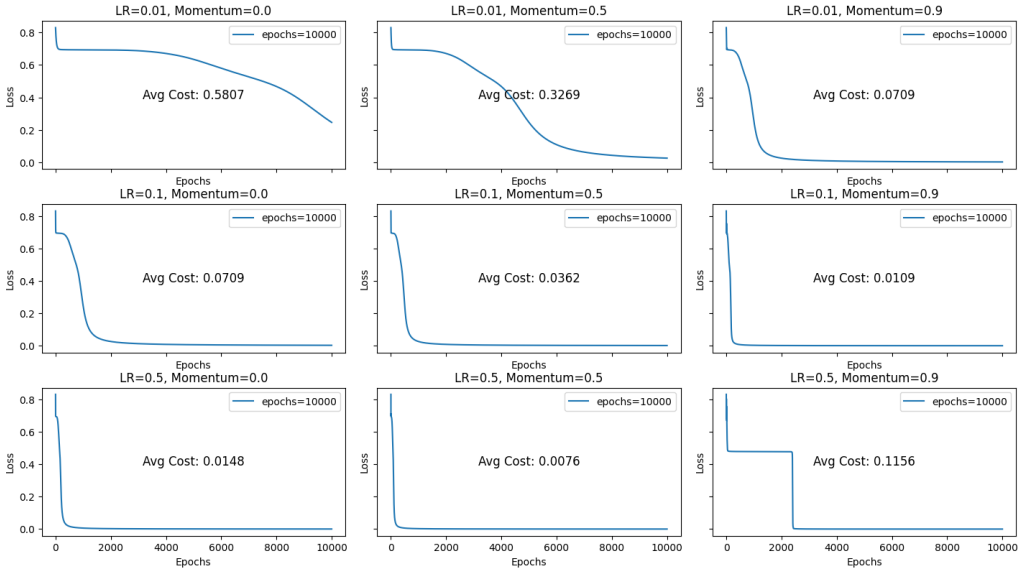
Momentum to technika optymalizacyjna, która dodaje "pamięć" o poprzednich krokach w procesie aktualizacji wag, aby przyspieszyć konwergencję i zmniejszyć oscylacje w dolinach funkcji kosztu. Dzięki temu model efektywniej porusza się w kierunkach zbieżnych i unika lokalnych minimów, co przekłada się na bardziej stabilny proces uczenia.



Rysunek 6: Koszt vs momentum

Na wykresie powyżej sprawdzano różne wartości funkcji kosztu dla różnych wielkości momentum. Założono, że współczynnik uczenia to 0.1. Widać zatem, że najlepsze wyniki osiąga momentum o wartości 0.9.

Warto jednak przeanalizować wszystkie wyżej wymienione hiperaparametry wspólnie.



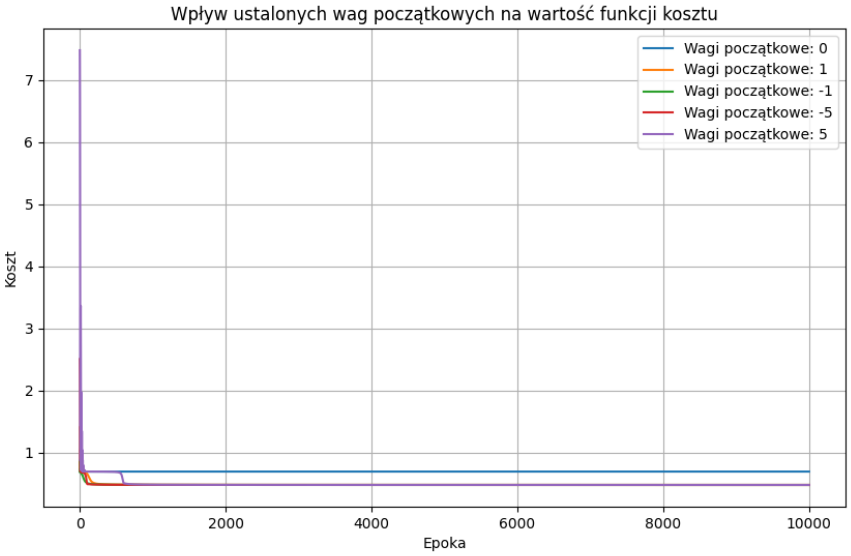
Rysunek 7: Symulacja hiperparametrów

Z powyższych wykresów zauważyć można, że dla liczby epok 10000 najlepsze wyniki osiąga współczynnik uczenia o wartości 0.5 z momentum równym 0.5, ale również podobną średnią wartość funkcji kosztu uzyskuje zestawienie momentum 0.9 oraz współczynnik uczenia 0.1.

3.4 Wagi początkowe

Wagi początkowe w sieciach neuronowych to losowo zainicjowane wartości, od których rozpoczyna się proces uczenia. Ich odpowiedni dobór jest kluczowy, ponieważ zbyt duże wagi mogą powodować problemy z eksplodującymi gradientami, a zbyt małe prowadzić do zanikających gradientów, co utrudnia efektywne uczenie modelu.

Przetestowano różne wartości wag początkowych. Na wykresie widoczne jest, iż początkowo istnieją spore rozbieżności w wartości kosztu dla różnych wag. Mimo to widać jednak, że w przypadku omawianej sieci po przejściu 10000 epok wagi te nie mają sporego znaczenia, gdyż końcowa wartość funkcji kosztu jest podobna dla testowanych wag. Zauważyć można jedynie lekki wzrost kosztu dla wag początkowych równych 0. Dlatego zmieniono losowanie wag z rozkładu normalnego na losowanie wag z rozkładu jednostajnego na przedziale (-1, 1).



Rysunek 8: Wpływ wag początkowych na funkcję kosztu

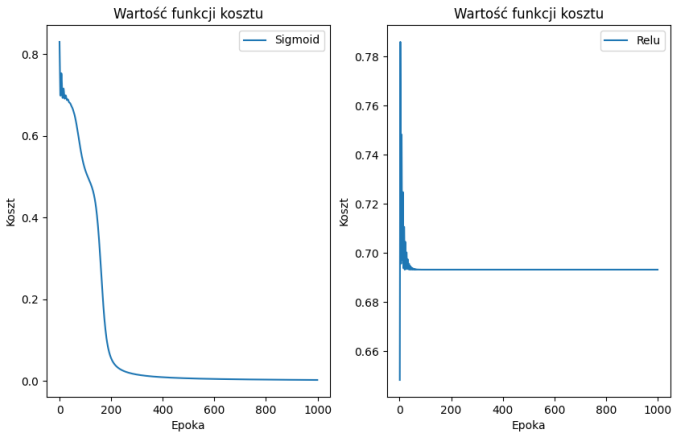
Wagi początkowe	**Koszt końcowy**
0	0.6931
1	0.4797
-1	0.4792
-5	0.4790
5	0.4824

3.5 Funkcja aktywacji

Funkcja aktywacji decyduje o tym, czy neuron powinien zostać aktywowany na podstawie sumy wejściowych sygnałów. Dzięki funkcjom aktywacji, takim jak sigmoid i ReLU, model jest w stanie uczyć się nieliniowych zależności, co jest kluczowe dla rozwiązywania bardziej skomplikowanych problemów.

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

$$\text{ReLU}(x) = \max(0, x)$$



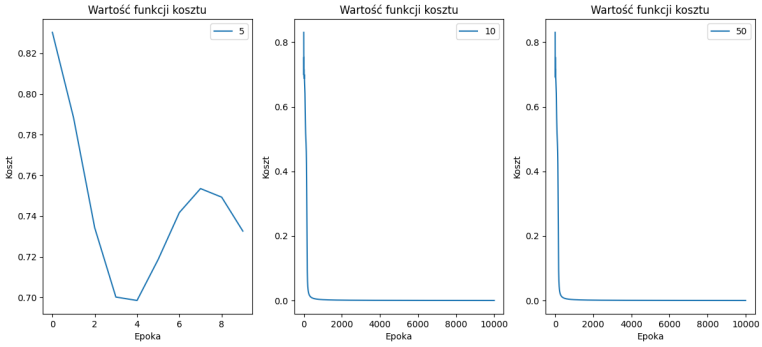
Rysunek 9: Funkcje aktywacji vs koszt

W omawianej sieci funkcja sigmoidalna zdecydowanie otrzymuje lepsze wyniki, co widać na powyższym wykresie.

3.6 Warunek stopu

W projekcie przyjęto sprawdzanie wartości kosztu funkcji i zapamiętywanie najlepszych wag podczas całego uczenia. Dodatkowo wprowadzono warunek stopu. Testowano różne wartości parametru *patience*, który mówi o tym, ile jest dopuszczalnych epok, podczas których nie zmienia się wartość funkcji kosztu.

Poniżej na wykresie widzimy, że w przypadku tej sieci wartość tego parametru równa 5 była zdecydowanie za mała, ponieważ model przerwał uczenie na samym początku, a funkcja kosztu była wysoka. Kolejne parametry okazało się, że nie wpływają na szybsze zakończenie, gdyż tak długie przestoje nie występowały w procesie uczenia sieci.



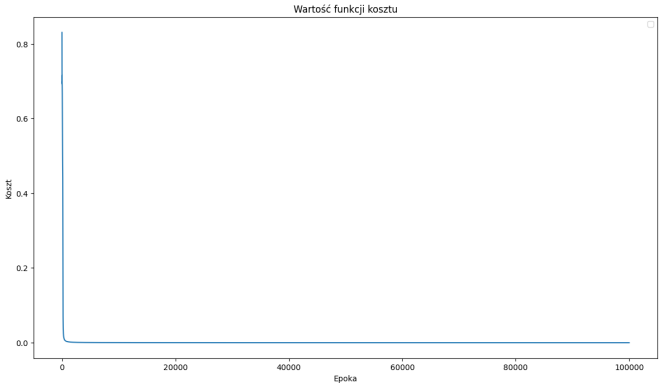
Rysunek 10: Warunek stopu

4 Wyniki

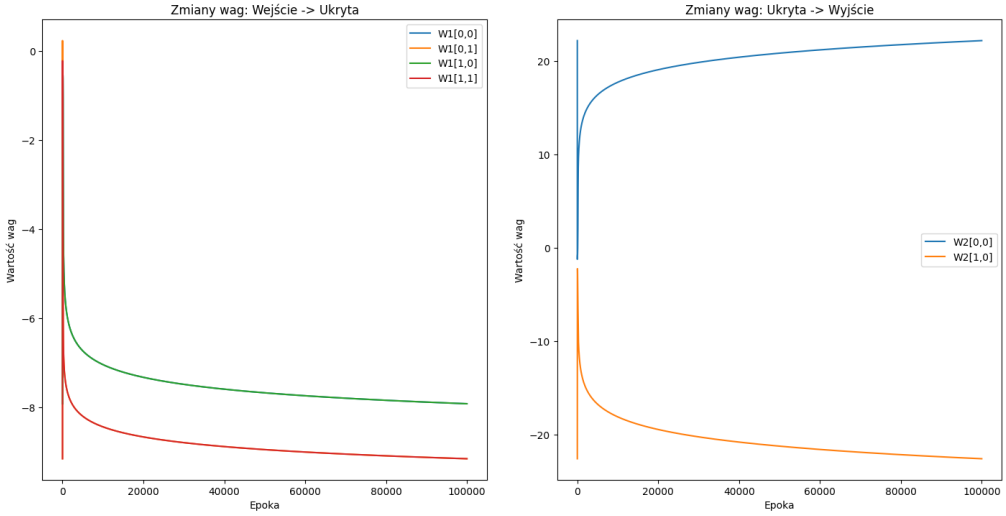
Po analizie hiperparametrów zdecydowano się użyć poniższych wartości:

- liczba epok: 100000
- współczynnik uczenia: 0.1
- momentum: 0.9
- funkcja aktywacji: sigmoida

4.1 Proces uczenia



Rysunek 11: Zmiana funkcji kosztu



Rysunek 12: Wykres zmian wag podczas etapu uczenia

4.2 Najlepsze wagi

$$w_1 = \begin{bmatrix} 9.24619812 & -8.89145538 \\ -8.9490941 & 9.19297115 \end{bmatrix}$$

$$w_2 = \begin{bmatrix} -16.01560446 \\ -16.0201618 \end{bmatrix}$$

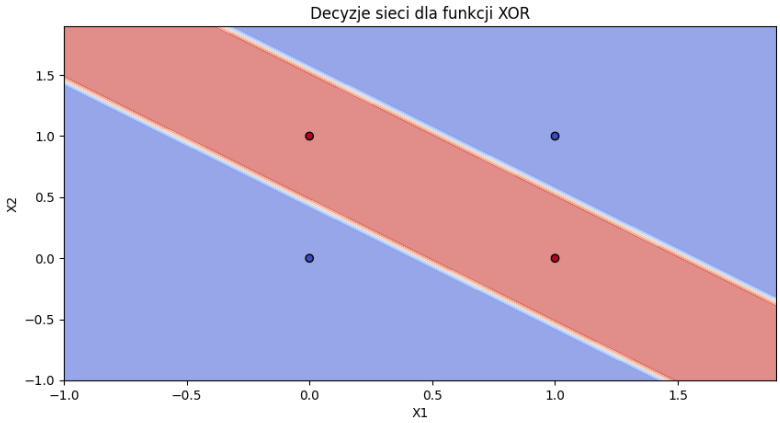
$$b_1 = \begin{bmatrix} 4.47470231 & 4.44436358 \end{bmatrix}$$

$$b_2 = \begin{bmatrix} 23.78494405 \end{bmatrix}$$

4.3 Predykcje

Po wytrenowaniu modelu, podczas predykcji wartości, odbywa się proces forward propagation, a następnie wartość prawdopodobieństwa przekształcana jest na wartość binarną przy pomocy funkcji progowej:

$$\text{step}(x) = \begin{cases} 1, & \text{jeśli } x \geqslant 0 \\ 0, & \text{jeśli } x < 0 \end{cases}$$



Rysunek 13: Ostateczna klasyfikacja

Wyniki z testowania sieci

Wejście: [1 0]
Predykcja: [[1]]
Prawidłowa odpowiedź: [1]

****
Wejście: [0 1]
Predykcja: [[1]]
Prawidłowa odpowiedź: [1]

****
Wejście: [0 0]
Predykcja: [[0]]
Prawidłowa odpowiedź: [0]

****
Wejście: [1 1]
Predykcja: [[0]]
Prawidłowa odpowiedź: [0]

5 Wnioski

W trakcie realizacji zadania udało się zaprojektować wielowarstwową sieć neuronową (MLP) zdolną do skutecznej klasyfikacji funkcji XOR, która jest klasycznym przykładem problemu nieliniowego. Zastosowanie warstwy ukrytej z odpowiednią liczbą neuronów oraz nieliniowej funkcji aktywacji okazało się kluczowe dla osiągnięcia poprawnych wyników. Uzyskane rezultaty potwierdzają, że

MLP jest w stanie efektywnie rozwiązywać zadania, które nie mogą być rozwiązane za pomocą prostych klasyfikatorów liniowych. Proces uczenia wymagał starannego doboru parametrów, takich jak liczba epok czy współczynnik uczenia, co podkreśla znaczenie optymalizacji w projektowaniu sieci neuronowych. Wyniki eksperymentu wskazują na potencjał MLP w bardziej złożonych zadaniach klasyfikacyjnych.