Laboratorium Sztuczna Inteligencja

Zadanie 3 - Implementacja sieci neuronowej

Imię i nazwisko:	Bartłomiej Koba
Numer indeksu:	266493
Termin zajęć:	Piątek, 13:15
Numer grupy ćwiczeniowej:	5
Data wykonania ćwiczenia:	15/12/2023
Prowadzący kurs:	dr inż. Magdalena Turowska

1 Opis problemu

1.1 Zadanie

Naszym zadaniem było zaimplementowanie sieci neuronowej realizującej dwuargumentową funkcję logiczną XOR. Implementacja miała być wykonana bez użycia bibliotek specjalnie zaprojektowanych do tworzenia skomplikowanych sieci neuronowych. Warto rozpocząć o wprowadzenia i wytłumaczenia podstawowych pojęć używanych w temacie sieci neuronowych.

1.2 Siec neuronowa - wprowadzenie

Sieć neuronowa to model obliczeniowy inspirowany biologicznym układem nerwowym. Składa się z połączonych sztucznych neuronów, które przetwarzają i przesyłają informacje między sobą. Sieć neuronowa składa się z:

1.2.1 Warstw

- **Warstwa wejściowa**: Przyjmuje dane wejściowe. Każdy neuron reprezentuje cechę danych wejściowych.
- **Warstwy ukryte**: Przetwarzają dane między warstwą wejściową a warstwą wyjściową. Obliczają złożone relacje między danymi.
- **Warstwa wyjściowa**: Generuje wyniki obliczeń sieci. Każdy neuron reprezentuje konkretne wyjście sieci.

1.2.2 Wagi i Połączenia

Każde połączenie między neuronami ma wagę, reprezentującą siłę połączenia. Wagi są modyfikowane podczas uczenia się sieci.

1.2.3 Funkcje Aktywacji

Neurony używają funkcji aktywacji, decydując, czy dane powinny być przekazane dalej. Przykładowe funkcje to Sigmoid, ReLU, tanh itp.

1.2.4 Uczenie

Sieć dostosowuje wagi na podstawie danych treningowych w celu minimalizacji błędu między predykcjami a rzeczywistością.

1.2.5 Propagacja Wsteczna

Algorytm propagacji wstecznej aktualizuje wagi w sieci, minimalizując błąd prognoz. Zaczyna od warstwy wyjściowej i przechodzi wstecz przez sieć.

i1	i2	z
0	0	0
0	1	1
1	0	1
1	1	0

2 Realizacja

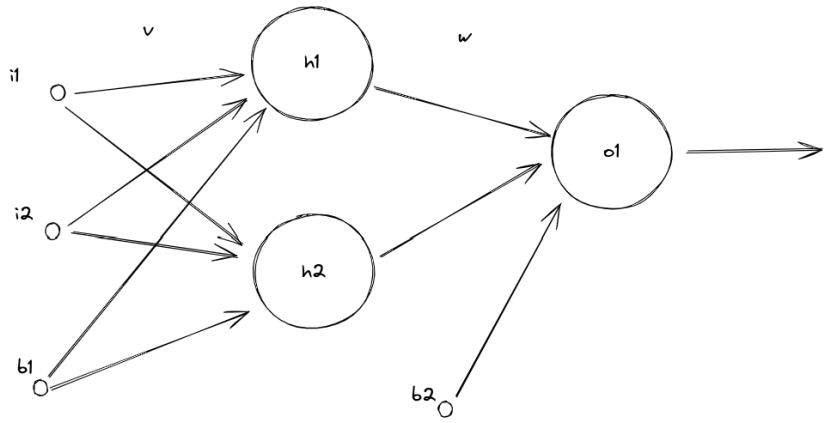
Realizacja została oparta o silnie typowany język Rust.

2.1 Rust - opis

Język Rust to systemowy język programowania o wysokim poziomie bezpieczeństwa i wydajności. Posiada zaawansowany system typów, eliminujący wiele błędów związanych z zarządzaniem pamięcią, takich jak wycieki pamięci czy odwołania null.

Jest znany z bliskiej wydajności do języków niskopoziomowych, jak C++ czy C, a także umożliwia programowanie współbieżne i równoległe dzięki abstrakcjom takim jak ‘async‘ i ‘await‘.

2.2 Projekt sieci



Rysunek 1: Szkic projektu sieci

Tak prezentuje się szkic sieci, mamy w niej dwa inputy oraz wejście syntetyczne jako warstwę wejściową. Warstwę ukrytą stanowią dwa neurony, natomiast wyjście obsługiwane jest przez jeden neuron połączony z warstwą ukrytą oraz z biasem.

2.2.1 Niezbędne oznaczenia

Warto wprowadzić niezbędne oznaczenia:

- $i1, i2 \in \{0, 1\}$ - wejścia
- $z \in \{0, 1\}$ - wyjście (numer klasy)

Zdefiniujmy również jak interpretujemy bramkę logiczną XOR: jak widać po powyższej notacji przyjąłem klasyfikację $0, 1$ gdzie 0 to brak sygnału na wyjściu bramki XOR, a jeden świadczy o wyjściu sygnał z symulowanej bramki. Przy realizacji w kodzie zdecydowałem się na losowanie początkowych wag. zatem możemy zdefinować wagi dla każdej warstwy. Dla warstwy wejściowej mamy: $\boldsymbol{v} = \begin{bmatrix} .v_1 1 & v_1 2 & v_1 3 \\ v_2 1 & v_2 2 & v_2 3 \end{bmatrix} =$

Dla warstwy ukrytej mamy: $\boldsymbol{w} \begin{bmatrix} w1 \\ w2 \\ w3 \end{bmatrix}$

2.2.2 Aktywatory

Głównym aktywatorem jest najprostsza w implementacji sigmoida którą możemy zdefiniować jako:

$$\sigma(z_i) = \frac{1}{1 + \exp(-z_i)}$$
$$\sigma^{'}(z_1) = \frac{\exp(z_i)}{(1 + \exp(z_i))^2}$$

2.3 Kodowanie

Rust pozwala na dogłębne kontrolowanie procesów zachodzących podczas realizacji obliczeń. W swojej realizacji postanowiłem oprzeć wartości liczbowe na **f32** czyli 32-bitowych zmiennych zmiennoprzecinkowych, co w zupełności wystarcza do realizacji zadań.

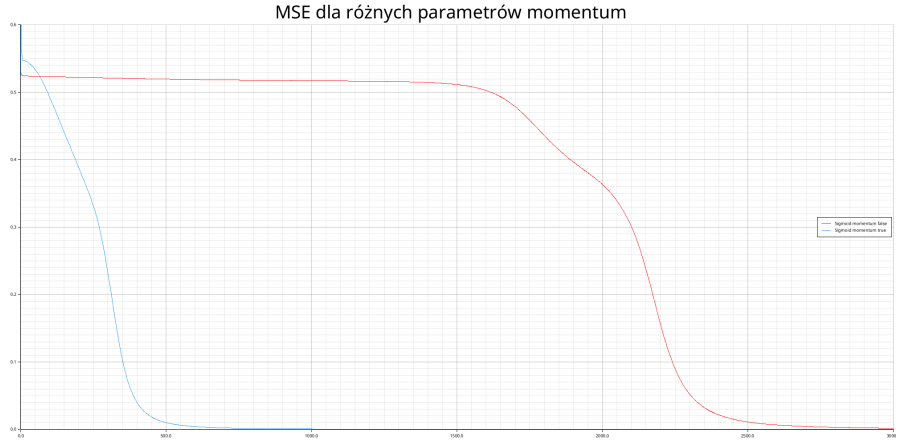
Tak natomiast prezentuje się przygotowanie danych do nauki modelu:

```
// declaring random weights for inputs nodes
let mut weights: Vec<Vec<f32>> = vec![
                                    vec![rng.gen::<f32>(),rng.gen::<f32>(),rng.gen::<f32>()], // w11, w12, w13
                                    vec![rng.gen::<f32>(),rng.gen::<f32>(),rng.gen::<f32>()], // w21, w22, w23
];

// declaring output_weights
let mut output_weights: Vec<f32> = vec![rng.gen::<f32>(),rng.gen::<f32>(),rng.gen::<f32>()]; //o1 o2 o3

let inputs: Vec<Vec<f32>> = vec![
    vec![0f32,0f32, -1f32],
    vec![0f32,1f32, -1f32],
    vec![1f32,0.0, -1f32],
    vec![1.0,1.0, -1f32]
];

let outputs: Vec<f32> = vec![0.0,1.0,1.0,0.0];
```

Rysunek 2: Wagi i dane do uczenia.

3 Wyniki

3.1 Wpływ momentum



Rysunek 3: Zmiana wywoływana przez momentum.

Inmplementacja momentum pozywynie wpływa na szybkość zmierzania błędu do akceptowalnych wartości.

3.2 Wpływ parametru momentum

Rysunek 4: Wpływ parametru α.

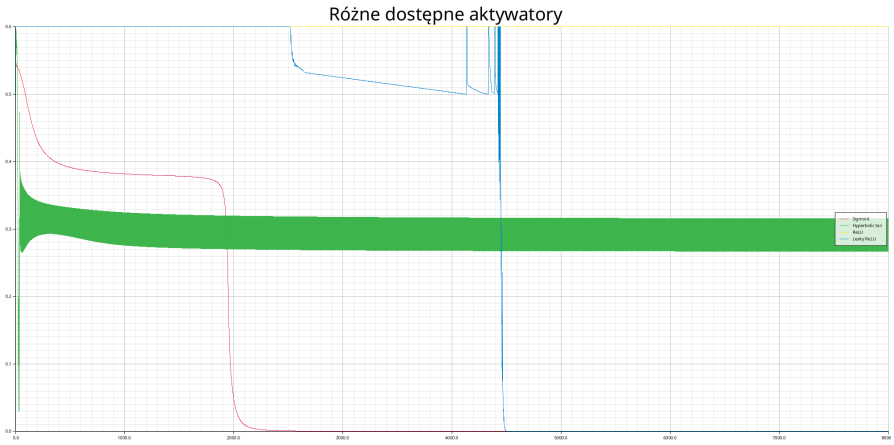
Porównując wyniki można zauważyć, że parametr α należy raczej umieszczać w górnych dostępnych granicach $(0 < \alpha < 1)$

3.3 Wpływ parametru β

Rysunek 5: Wpływ parametru β.

Porównując wyniki można zauważyć, że parametr β należy raczej umieszczać w górnych dostępnych granicach $(0 < \beta < 1)$

3.4 Różne aktywatory



Rysunek 6: Zróżnicowane aktywatory.

Wyniki są niejednoznaczne. Najbardziej stabilnym aktywatorem na przestrzeni wielu prób okazał się sigmoid, Leaky ReLU zachowuje się nieprzewidywalnie, zarówno z sferze osiągnięcia zadowalających wyników jak i przebiegu. Stabilnie ale bez schodzenia poniżej 0.1 sprawdza się Hyperboliczny tangens.

4 Wnioski

5 Czego się nauczyłem

Warto wykonać zadanie takiego typu aby zrozumieć (chociaż w części) zasady działania sieci neuronowych i tego jak działają "pod spodem". Nauczyłem się również pracy z wektorami w języku Rust oraz jak generować wykresy w tymże języku.

6 Co warto by zaimplementować

W przyszłości chciałby wzbogacić moją implementację sieci neuronowej o ograniczenie zbędnego liczenia gdy przechodzenie przez kolejne epoki nie zmienia wyraźnie wartości błędu średniokwadratowego.