Implementacja Sieci Neuronowych typu MLP

Bartosz Jezierski
Nr indeksu: 327280

13 kwietnia 2025

Spis treści

1 Wprowadzenie

Celem laboratoriów było poznanie budowy i działania sieci neuronowych typu MLP. W ramach projektu realizowane były kolejne etapy, obejmujące:

- Bazową implementację sieci (NN1),
- Uczenie sieci metodą wstecznej propagacji błędu (NN2),
- Implementację usprawnień: momentu oraz normalizacji gradientu RMSProp (NN3),
- Rozwiązywanie zadania klasyfikacji z wykorzystaniem funkcji softmax (NN4),
- Testowanie różnych funkcji aktywacji (NN5),
- Implementację regularyzacji zapobiegającej przeuczeniu (NN6).

2 Opis wykonanej pracy i wyniki

2.1 Implementacja bazowej sieci MLP (NN1)

2.1.1 Opis działania

W pierwszym etapie zaimplementowałem sieć neuronową umożliwiającą definiowanie:

- liczby warstw,
- liczby neuronów w każdej warstwie,
- wartości wag oraz biasów,
- wyboru funkcji aktywacji i na wyjściu.

```
x = data[:, 0].reshape(-1, 1) # Dane w odpowiednim formacie
net = MLP([1, 5, 10, 1], sigmoid, linear) # Definicja sieci
net.forward(x)
```

Listing 1: Przykładowa definicja sieci wraz z użyciem

`[1, 5, 10, 1]` to liczba neuronów w kolejnych warstwach, a `sigmoid` i `linear` to kolejno funkcja aktywacji i funkcja na wyjściu z sieci.
Uwaga! W mojej implementacji inaczej niż na laboratoriach zdefiniowana była macierz wag w warstwie. Moim zdaniem bardziej intuicyjne było, aby waga $w_{ij}^{(L)}$ reprezentowała połączenie i-tego neuronu wejścia z j-tym neuronem wyjścia w kolejnej warstwie. Przez to musiałem później bardziej uważać przy implementowaniu propagacji wstecznej błędu.
Sieć zaimplementowałem tak, żeby wewnętrznie standaryzowała dane i wszystkie operacje przeprowadzane są na danych zestandaryzowanych.

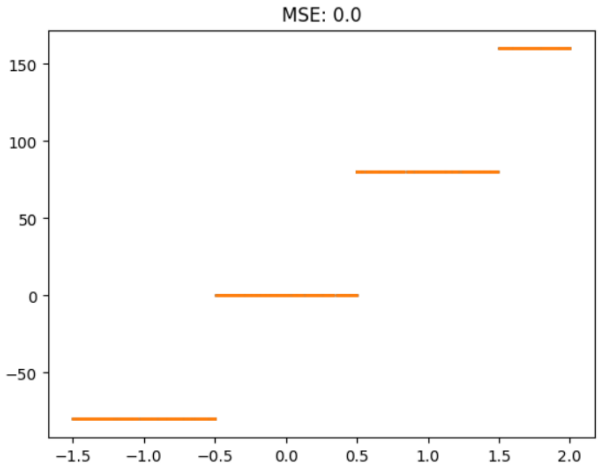
2.1.2 Eksperymenty

Należało dobrać parametry do danych **steps-large** i **square-simple** tak, aby wartość MSE nie przekraczała 9 na nieznormalizowanych danych.
Dla danych **steps-large** udało mi się wymyślić na kartce rozwiązanie idealnych wag dla tego zbioru:

```
net = MLP([1, 5, 1], sigmoid, linear)
a = 1000000
w0 = np.array([[a, a, a, 0, 0]])
w1 = np.array([[80],[80],[80],[0],[0]])
b0 = np.array([0.5*a,   -0.5*a,   -1.5*a,  0, 0])
b1 = np.array([-80])
```

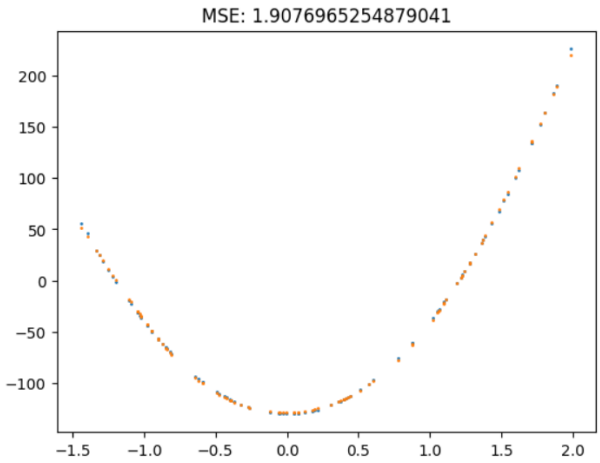
Listing 2: Rozwiązanie steps-large

Co skutkowało dopasowaniem dla tego zbioru:



Rysunek 1: Predykcja punktów dla zbioru **steps-large**

Z kolei dla danych **square-simple** próbowałem różne możliwe wagi ręcznie, aż natrafiłem na odpowiednie:



Rysunek 2: Predykcja punktów dla zbioru **square-simple**

2.2 Propagacja wsteczna błędu (NN2)

2.2.1 Opis działania

W kolejnym etapie rozszerzyłem implementację o uczenie sieci metodą wstecznej propagacji błędu (na przykładzie funkcji sigmoid). W tym etapie wykonanałem:

- Uczenie sieci z wykorzystaniem propagacji wstecznej,
- Wizualizację wartości wag podczas kolejnych iteracji,
- Implementacje możliwości zdefiniowania batch-size

W mojej implementacji aby przyspieszyć proces w kroku wstecz do pochodnej przekazywane jest wyjście z funkcji aktywacji, a nie suma ważona wartości w neuronach. Poniżej przedstawiam fragment kodu implementujący krok wstecz dla jednej warstwy oraz :

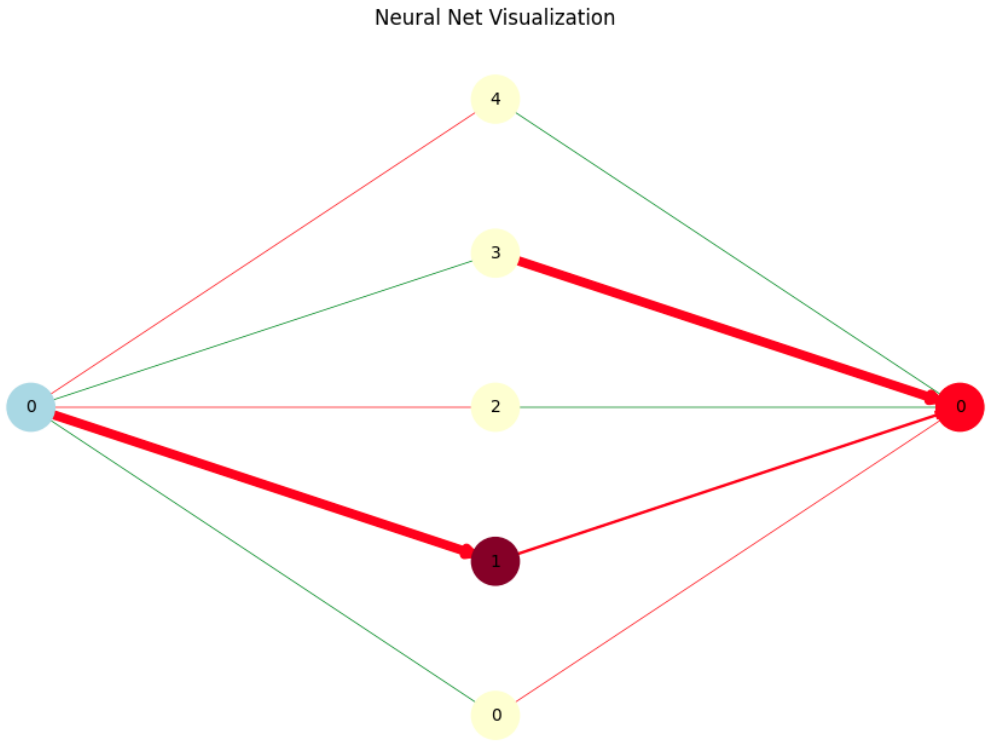
```
def backward(self, dA):
        # dC / dz
```

```
        dZ = self.act_derivative(self.a_out) * dA # zakladamy, ze
            uzywamy act_derivative na "a", a nie na "z".
        # Zmiana na wagach (L)
        dW = np.dot(self.a_in.T, dZ)
        # Zmiana na bias (L)
        db = np.sum(dZ, axis=0)
        # Zmiana dA (L-1)
        dA_in = np.dot(dZ, self.weights.T)
        return dA_in, dW, db

def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def sigmoid_deriv(output):
    return output * (1 - output)
```

Listing 3: Część implementacji kroku wstecz oraz pochodna i funkcja sigmoid

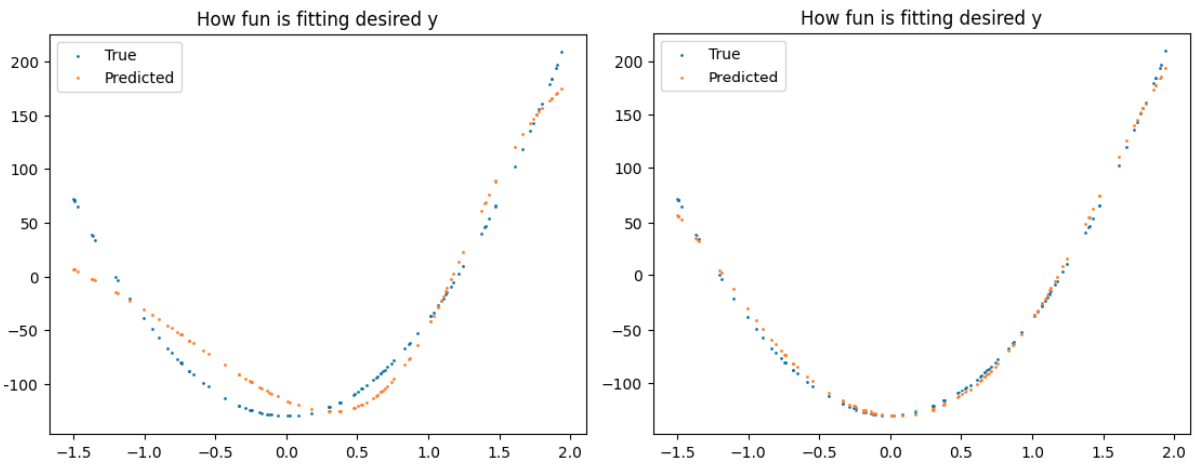
2.2.2 Eksperymenty

Zbiór **square-simple** był pierwszym, który testowałem. Na początku uczenie nie działało. Wydawało się, że sieć dąży do ogromnych wartości wag co udało mi się zobaczyć na wizualizacji wag:



Rysunek 3: Widać, że jedno połączenie w warstwie jest duże większe od innych.

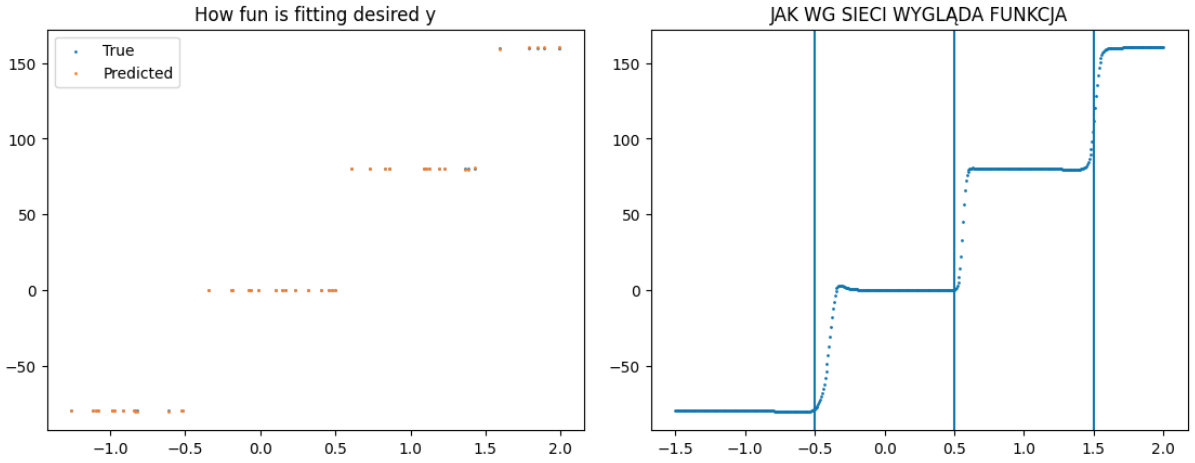
Problemem okazało się niepodzielenie zmiany przez wielkość batcha. Po naprawieniu wszystko działało jak trzeba.
Dodanie opcji wykonywania uczenia za pomocą mini-batch przyspieszyło uczenie (przynajmniej w sensie epok). Poniżej na lewym wykresie przedstawione jest dopasowanie się sieci po 400 epokach, kiedy w epoce brany był cały zbiór, a po prawej kiedy zbiór było podzielony na 32-elementowe batche:



Rysunek 4: Dopasowanie sieci do funkcji po 400 epokach dla **square-simple**.

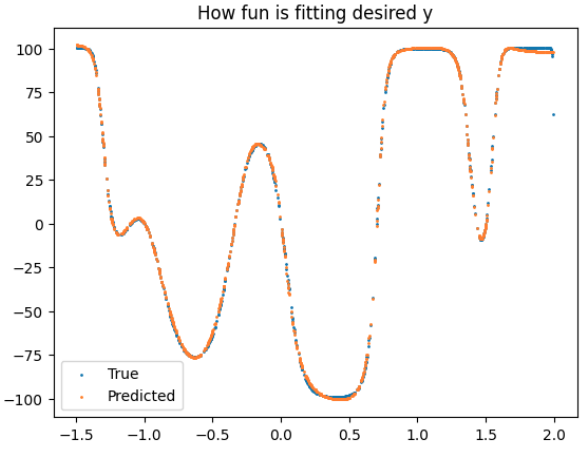
Na tym zbiorze bez problemu udało się spełnić wymagania dotyczące MSE.

Zbiór **steps-small** okazał się niemożliwy przy teraźniejszej implementacji sieci (bez momentu i innych usprawnień) do nauczenia, by wymagania dotyczące MSE były spełnione. Wynikało to z tego, że zbiór testowy miał punkty między skokami funkcji, których sieć nie mogła dobrze przewidzieć. Poniżej przedstawiam jak moja sieć przetwarzała tę funkcję:



Rysunek 5: Po lewo dopasowanie sieci na zbiorze treningowym, a po prawo jak sieć przewidywała funkcję (zbiór testowy posiadał punkty w miejscach wysokiej zmienności przewidywań).

Wymagania dla zbioru **multimodal-large** okazały się proste do spełnienia (MSE = 2.3).



Rysunek 6: Dopasowanie do funkcji na zbiorze testowym dla **multimodal-large**

2.3 Implementacja momentu i RMSProp (NN3)

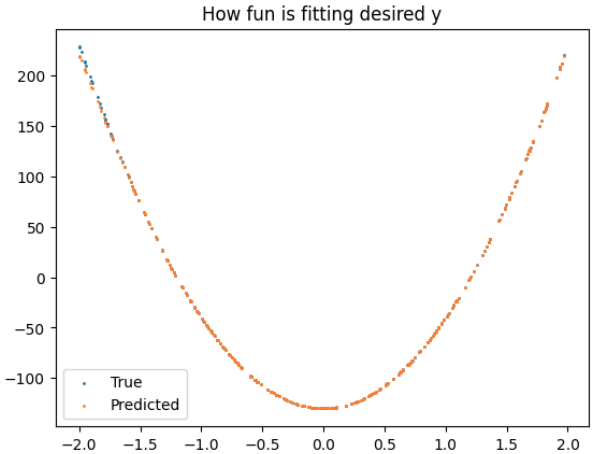
2.3.1 Opis działania

Na etapie NN3 zaimplementowałem:

- Uczenie z momentem,
- Normalizację gradientu metodą RMSProp,
- Możliwość zmiany typu inicjalizacji wag (tutaj tylko metoda Xavier, w kolejnym etapie dodałem He),
- Optimizer Adam (po nieudanych próbach z poprzednimi), który jest połączeniem uczenia z momentem i RMSProp.

2.3.2 Eksperymenty

Zbiór **steps-large** okazał się problematyczny. Najlepszy wynik udało mi się osiągnąć stosując uczenie z momentem (MSE = 1.99 na milionie epok...), jednak to dalej nie spełniało wymagań.



Rysunek 7: Dopasowanie do zbioru steps large (zbiór testowy)

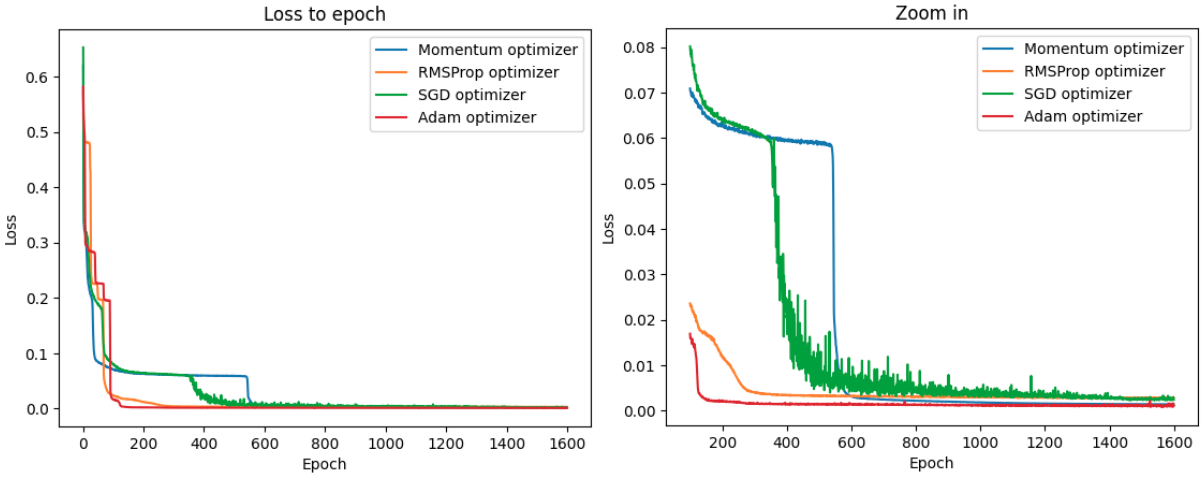
Przy testowaniu doszedłem do następujących wniosków:

- Po implementacji Xavier siec uczy sie dużo stabilniej,
- Duża sieć potrzebuje więcej epok, ale daje to lepsze rezultaty,

- Im większy batch_size tym więcej epok potrzeba, ale też im większy batch_size tym szybciej są wykonywane operacje,
- Żeby zastosować większą lambdę(moment), potrzeba czasami zmniejszyć learning rate, żeby sieć była w stanie się uczyć,
- Zwiększanie ilości layerów jest bardziej kosztowne (czasowo) niż znaczne zwiększenie ilości neuronów w warstwach.

Jako, że nie byłem w stanie spełnić wymagań zaimplementowałem wtedy dodatkowo optimizer Adam, który już spełnił oczekiwania (MSE = 0.43).

Dwa kolejne zbiory **steps-large** i **multimodal-large** nie sprawiły już większych problemów, w obu przypadkach udało się osiągnąć MSE < 3. Poniżej przedstawiam porównanie zbieżności różnych optimizerów dla **multimodal-large**.



Rysunek 8: Porównanie zbieżności różnych optimizerów dla **multimodal-large**.

Widać, że pomimo tego, że Adam nie uzyskuje od początku najlepszych rezultatów to w pewnym momencie deklasuje reszte konkurentów.

2.4 Rozwiązanie zadania klasyfikacji (NN4)

2.4.1 Opis działania

Na etapie NN4 trzeba było zaimplementować uczenie się z funkcją softmax na wyjściu, by następnie porównać jak radzi sobie sieć ucząc się klasyfikacji z softmax na wyjściu oraz z funkcją liniową na wyjściu. Ważnym punktem było odpowiednie zaimplementowanie softmax w celu uniknięcia numerycznej niestabilności problemu:

```
def softmax(z):
    d = -np.max(z, axis=1, keepdims=True)
    return np.exp(z + d) / np.sum(np.exp(z + d), axis=1, keepdims=True)
```

Listing 4: Implementacja softmax

2.4.2 Eksperymenty

Zostały przeprowadzone na zbiorach **rings3-regular**, **easy** oraz **xor3**, gdzie F-score na zbiorze testowym miał wynieść < 0.75, < 0.99 i <0.97. Wszystkie wymagania udało się spełnić bez żadnych komplikacji:

Dane	Score z funkcją liniową	Score z funkcją softmax
rings3-regular	0.9458	0.9810
easy	0.9980	0.9980
xor3	0.9610	0.9672

Tabela 1: Wyniki F-score na różnych danych (we wszystkich testach sieć miała 3 warstwy ukryte po 5 neuronów na każdą).

Ponieważ na danych xor3 początkowo nie spełniłem wymagań to jeszcze raz na większej liczbie epok nauczyłem sieć osiągając F-score = 0.975.
Można zauważyć, że rzeczywiście softmax poprawia jakość działania sieci.

2.5 Testowanie funkcji aktywacji (NN5)

2.5.1 Opis działania

W etapie NN5 rozszerzyłem implementację sieci o możliwość wyboru nowych funkcji aktywacji:

- Tanh,
- ReLU,
- LReLU

Żeby nie zmieniać kodu wewnątrz klasy sieci, postanowiłem, że również tutaj zaimplementuje pochodne tych funkcji tak, aby przyjmowały one output ze swojej funkcji aktywacji (a nie średnią ważoną wartości w neuronach i biasu):

```
# Nowe funkcje aktywacji
def tanh(x):
    return np.tanh(x)

def tanh_deriv(output):
    return 1 - np.power(output, 2)

def relu(x):
    return np.maximum(0, x)

def relu_deriv(a_out):
    return np.where(a_out > 0, 1.0, 0.0)

def leaky_relu(x, alpha=0.01):
    return np.where(x > 0, x, alpha * x)

def leaky_relu_deriv(a_out, alpha=0.01):
    return np.where(a_out > 0, 1.0, alpha)
```

Listing 5: Implementacja nowych funkcji aktywacji

2.5.2 Eksperymenty

Należało przeprowadzić na zbiorach **multimodal-large**, **steps-large**, **rings5-regular** i **rings3-regular**.
Oto wyniki na zbiorze **multimodal-large**:

Architektura	Funkcja aktywacji	MSE
[1, 20, 20, 1]	sigmoid	3.055278
[1, 20, 20, 1]	tanh	3.223643
[1, 20, 20, 1]	leaky_relu	3.712780
[1, 20, 20, 1]	relu	7.396732
[1, 5, 5, 1]	tanh	53.903093
[1, 5, 5, 1]	sigmoid	57.417997
[1, 5, 5, 5, 1]	tanh	68.409272
[1, 5, 5, 5, 1]	relu	79.275383
[1, 5, 5, 5, 1]	leaky_relu	230.796000
[1, 5, 5, 1]	leaky_relu	234.785497
[1, 5, 1]	tanh	385.819878
[1, 5, 1]	sigmoid	427.517375
[1, 5, 5, 1]	relu	473.923893
[1, 5, 5, 5, 1]	sigmoid	1432.642456
[1, 5, 1]	relu	1682.510804
[1, 5, 1]	leaky_relu	1685.938096

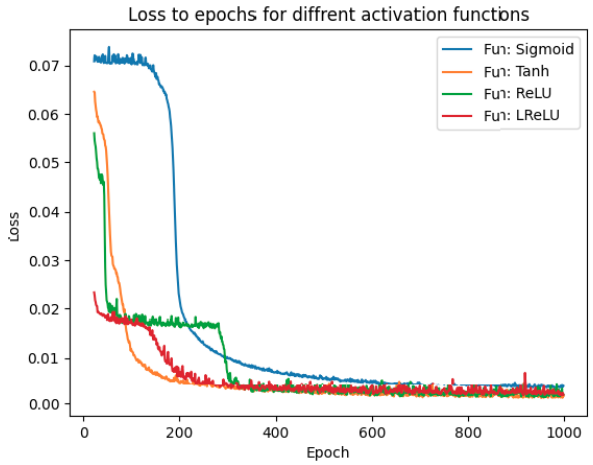
Tabela 2: Porównanie dla zbioru **multimodal-large**.

Nie zdziwiło mnie tutaj to, że najlepiej poradziły sobie funkcje "gładkie", jako, że brany był zbiór **multimodal-large**. Dlatego też w kolejnych testach ciągle brałem pod uwagę resztę funkcji aktywacji (np. wydawało mi się, że dla **steps-large** najodpowiedniejsze powinno być ReLU, bądź LReLU). Postanowiłem też, że od tego momentu testował będę już tylko dla architekury [1, 20, 20, 1].

Dane **steps-large**:

Architektura	Funkcja aktywacji	MSE
[1, 20, 20, 1]	relu	6.738343
[1, 20, 20, 1]	leaky_relu	7.794850
[1, 20, 20, 1]	tanh	8.628865
[1, 20, 20, 1]	sigmoid	17.975794

Tabela 3: Porównanie dla zbioru **steps-large**.



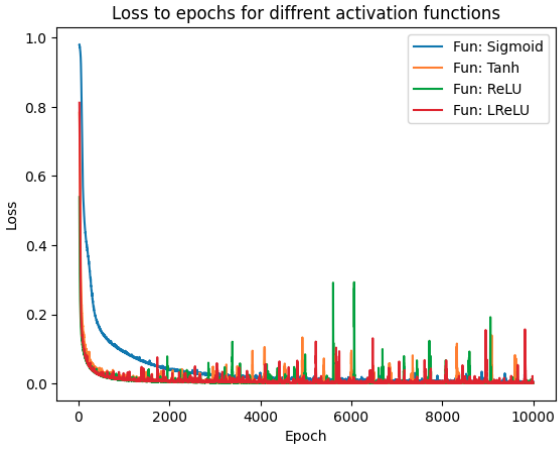
Rysunek 9: Proces uczenia się dla różnych funkcji aktywacji dla **steps-large**

I tak jak się spodziewałem, w tym przypadku lepiej poradziły sobie funkcje "niegładkie".

Dane **rings3-regular**:

Architektura	**Funkcja aktywacji**	**Fscore**
$[2, 20, 20, 3]$	sigmoid	0.981558
$[2, 20, 20, 3]$	tanh	0.979072
$[2, 20, 20, 3]$	relu	0.977705
$[2, 20, 20, 3]$	leaky_relu	0.972697

Tabela 4: Porównanie dla zbioru **rings3-regular**.

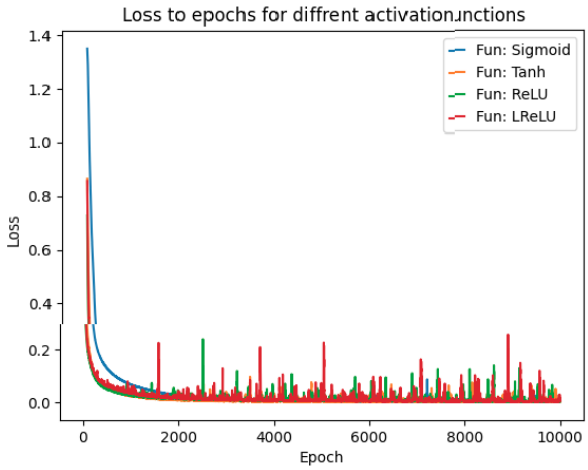


Rysunek 10: Proces uczenia się dla różnych funkcji aktywacji dla **rings3-regular**

Dane **rings5-regular**:

Architektura	**Funkcja aktywacji**	**Fscore**
$[2, 20, 20, 5]$	sigmoid	0.970043
$[2, 20, 20, 5]$	tanh	0.964636
$[2, 20, 20, 5]$	relu	0.957355
$[2, 20, 20, 5]$	leaky_relu	0.952728

Tabela 5: Porównanie dla zbioru **rings5-regular**.



Rysunek 11: Proces uczenia się dla różnych funkcji aktywacji dla **rings5-regular**

Można generalnie zauważyć, że ReLU i LReLU wydają się szybciej zbiegać, ale z większymi "turbulencjami"po drodze.

2.6 Regularyzacja (NN6)

2.6.1 Opis działania

W ostatnim etapie dodałem:

- Możliwość ustawienia Dropout,
- Możliwość wcześniejszego stopu,
- Możliwość l1 i l2 regularyzacji.

```
def backward(self, dA, optimizer, l1_lambda, l2_lambda):
        for layer in reversed(self.layers):
            dA, dW, db = layer.backward(dA)
            dW += l1_lambda * np.sign(layer.weights) + l2_lambda *
                layer.weights  # DODANIE REGULARYZACJI
            optimizer.update(layer, dW, db)
```

Listing 6: Część kodu zapewniająca możliwość regularyzacji.

Możliwosć stopu jest ustawiana za pomocą parametru early_stopping_patience w funkcji train (w sprawozdaniu nazywana patience). Po przekroczeniu epok (ustawionych jako patience) bez poprawienia się najlepszego wyniku loss (dla zbioru testowego) trening jest zatrzymywany, w celu zapobiegniecia zjawisku overfittingu. Ustawienie patience na 0 wyłącza early stopping.

2.6.2 Eksperymenty

Były przeprowadzane na zbiorach o niewielkim zbiorze treningowym (sparse w nazwie), bądź o niezbalansowanych targetach (balance w nazwie). Niestety nie wiem z jakiej przyczyny, ale aktywacja Dropout dawała tragiczne wyniki. Jako, że nie było to celem tego etapu, to postanowiłem, że pominę testowanie Dropout w sprawozdaniu. Poniżej zamieszczam jakie hiperapametry przeszukiwałem w testach:

```
archs = [[2, 20, 20, 5]]
funs = [sigmoid, tanh, relu, leaky_relu]
fun_derivs = [sigmoid_deriv, tanh_deriv, relu_deriv, leaky_relu_deriv]
dropouts = [0]
early_stops = [0, 50, 100, 200, 500]
```

```
l1_vals = [0, 0.01, 0.001, 0.0001]
l2_vals = [0, 0.01, 0.001, 0.0001]
```

Listing 7: Siatka przeszukiwań hiperparametrów.

Dane **multimodal-sparse**:

Arch	Fun	MSE	Patience	Net	L1	L2
[1, 20, 20, 1]	tanh	67.614738	0	MLP	0.0000	0.0000
[1, 20, 20, 1]	sigmoid	85.259411	200	MLP	0.0000	0.0000
[1, 20, 20, 1]	tanh	88.063898	0	MLP	0.0001	0.0000
[1, 20, 20, 1]	tanh	101.784085	100	MLP	0.0001	0.0000
[1, 20, 20, 1]	sigmoid	105.696081	0	MLP	0.0001	0.0000
[1, 20, 20, 1]	tanh	113.013384	50	MLP	0.0000	0.0000
[1, 20, 20, 1]	tanh	114.662063	200	MLP	0.0001	0.0001
[1, 20, 20, 1]	sigmoid	115.274965	0	MLP	0.0000	0.0000
[1, 20, 20, 1]	tanh	115.965850	500	MLP	0.0001	0.0000
[1, 20, 20, 1]	tanh	122.204717	0	MLP	0.0001	0.0001

Tabela 6: Najlepsze 10 kombinacji funkcji aktywacji, regularyzacji i patience względem MSE dla **multimodal-sparse**.

Dane **rings5-sparse**:

Arch	Fun	Patience	Net	L1	L2	Fscore
[2, 20, 20, 5]	relu	200	MLP	0.0010	0.0001	0.842862
[2, 20, 20, 5]	leaky_relu	200	MLP	0.0010	0.0001	0.843036
[2, 20, 20, 5]	leaky_relu	0	MLP	0.0010	0.0001	0.843820
[2, 20, 20, 5]	leaky_relu	500	MLP	0.0010	0.0001	0.844642
[2, 20, 20, 5]	leaky_relu	200	MLP	0.0000	0.0000	0.846231
[2, 20, 20, 5]	leaky_relu	200	MLP	0.0010	0.0000	0.847221
[2, 20, 20, 5]	sigmoid	200	MLP	0.0010	0.0000	0.848399
[2, 20, 20, 5]	sigmoid	200	MLP	0.0000	0.0000	0.851560
[2, 20, 20, 5]	sigmoid	500	MLP	0.0001	0.0001	0.854622
[2, 20, 20, 5]	relu	50	MLP	0.0000	0.0000	0.858219

Tabela 7: 10 najlepszych kombinacji względem Fscore dla **rings5-sparse**

Dane **rings3-balance**:

Arch	Fun	Patience	Net	L1	L2	Fscore
[2, 20, 20, 3]	tanh	0	MLP	0.0001	0.0	0.914554
[2, 20, 20, 3]	relu	50	MLP	0.0001	0.0	0.916002
[2, 20, 20, 3]	sigmoid	200	MLP	0.0000	0.0	0.916742
[2, 20, 20, 3]	leaky_relu	100	MLP	0.0000	0.0	0.919354
[2, 20, 20, 3]	tanh	0	MLP	0.0000	0.0	0.921156
[2, 20, 20, 3]	tanh	100	MLP	0.0000	0.0	0.921839
[2, 20, 20, 3]	relu	200	MLP	0.0000	0.0	0.923190
[2, 20, 20, 3]	leaky_relu	50	MLP	0.0000	0.0	0.925993
[2, 20, 20, 3]	relu	100	MLP	0.0000	0.0	0.927464
[2, 20, 20, 3]	tanh	50	MLP	0.0000	0.0	0.928748

Tabela 8: 10 najlepszych kombinacji względem Fscore dla **rings3-balance**

Dane **xor3-balance**:

Arch	Fun	Patience	Net	L1	L2	Fscore
[2, 20, 20, 2]	tanh	0	MLP	0.0001	0.0001	0.910651
[2, 20, 20, 2]	leaky_relu	50	MLP	0.0001	0.0000	0.911817
[2, 20, 20, 2]	leaky_relu	500	MLP	0.0001	0.0000	0.911817
[2, 20, 20, 2]	leaky_relu	50	MLP	0.0000	0.0000	0.911817
[2, 20, 20, 2]	tanh	0	MLP	0.0000	0.0001	0.914494
[2, 20, 20, 2]	tanh	500	MLP	0.0000	0.0000	0.914576
[2, 20, 20, 2]	tanh	500	MLP	0.0001	0.0001	0.915830
[2, 20, 20, 2]	tanh	100	MLP	0.0000	0.0000	0.918575
[2, 20, 20, 2]	tanh	200	MLP	0.0000	0.0000	0.921160
[2, 20, 20, 2]	tanh	500	MLP	0.0001	0.0000	0.923886

Tabela 9: 10 najlepszych kombinacji względem Fscore dla **xor3-balance**

2.6.3 Wnioski

- Najlepsze wyniki otrzymywały kombinacje z usuniętą, bądź bardzo małą wartością regularyzacji l1 i l2. W rzeczywistoci dla większych wartości sieć tak naprawdę przestawała się uczyć,
- Kluczową rolę w przedziwdziałaniu przeuczeniu stanowi early stopping,