

## MLP - Perceptrony Wielowarstwowe

---

### SPRAWOZDANIE

Kornel Tłaczała

16 czerwca 2025

### Spis treści

<b>NN1:</b> Bazowa implementacja	<b>2</b>
<b>NN2:</b> Propagacja wsteczna	<b>4</b>
<b>NN3:</b> Implementacja momentu i normalizacji gradientu (RMSProp)	<b>7</b>
<b>NN4:</b> Klasyfikacja - implementacja z softmax	<b>10</b>
<b>NN5:</b> Testowanie różnych funkcji aktywacji	<b>15</b>
<b>NN6:</b> Regularyzacja i przeuczanie	<b>20</b>

# NN1: Bazowa implementacja

## Cel

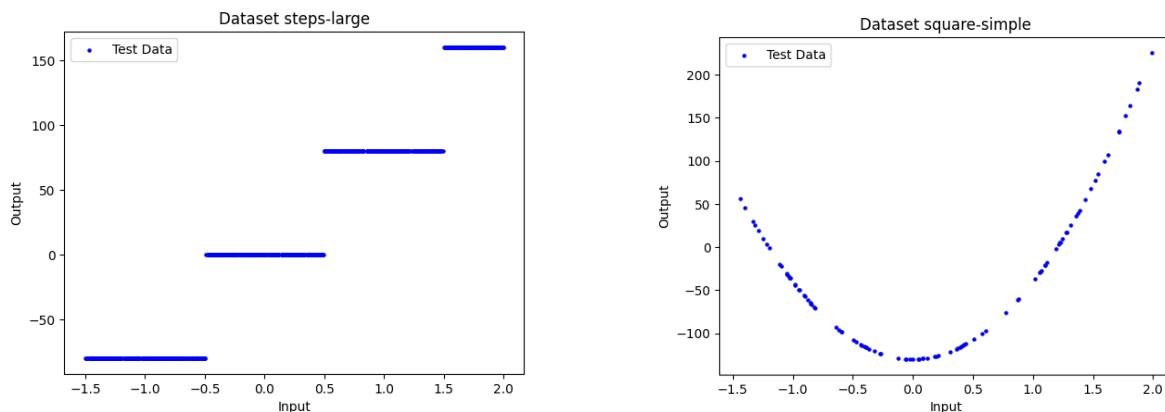
- Stworzenie własnej, podstawowej implementacji sieci neuronowej, która będzie w stanie uczyć się na podstawie danych wejściowych i przewidywać wyniki zadania regresji. Zapewnienie możliwości łatwej zmiany architektury sieci oraz funkcji aktywacji.
- Ręczne dobranie parametrów sieci dla 3 architektur:
  - 1 warstwa ukryta, 5 neuronów
  - 1 warstwa ukryta, 10 neuronów
  - 2 warstwy ukryte, 5 neuronów w każdej
 oraz przetestowanie ich na 2 zbiorach danych:
  - square-simple
  - steps-large

## Implementacja

Model został zaimplementowany w Pythonie. W celu zapewnienia elastyczności oraz łatwości w rozwijaniu projektu implementacja została podzielona na klasy:

- MLP** - klasa reprezentująca sieć neuronową, zawierająca metody do trenowania i przewidywania oraz wizualizacji wyników.
- Layer** - klasa reprezentująca pojedynczą warstwę sieci, zawierająca neurony oraz metody do obliczania aktywacji. Konieczne, ale i wygodne okazało się rozszerzenie tej klasy do klas **FirstLayer** i **LastLayer**, które reprezentują odpowiednio warstwę wejściową i wyjściową i miały częstą inną funkcję aktywacji, bądź sposób aktualizacji wag. To właśnie każda warstwa przechowywała informację o wartościach wag, biasów oraz informację o wykorzystywanej funkcji aktywacji.
- MLPArchitecture** - klasa reprezentująca architekturę MLP, zapewniająca możliwość łatwego tworzenia różnych konfiguracji sieci neuronowych poprzez definiowanie liczby warstw i neuronów w każdej z nich
- Dataset** - klasa reprezentująca zbiór danych, zawierająca metody do ładowania danych z plików CSV oraz dzielenia ich na zbiory treningowe i testowe. Automatyzuje ona normalizację danych. Wspomaga klasę MLP w reprezentacji zbioru danych. Jej metody przyjmując przewidywanie sieci, zwracają odpowiednie metryki jakości predykcji, w tym liczbę funkcji straty.

## Zbiorы danych

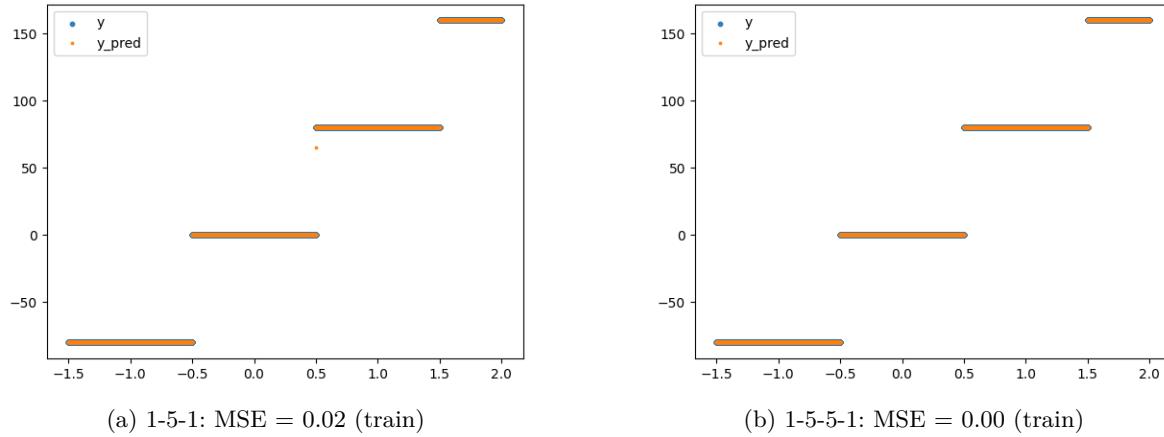


Rysunek 1: Prezentacja zbiorów steps-large i square-simple

## Wyniki - steps-large

Architektura	MSE (train)	MSE (test)
1-5-1	0.02	0.00
1-10-1	0.02	0.00
1-5-5-1	0.00	0.00

Tabela 1: Wartości funkcji straty (MSE) dla zbioru steps-large



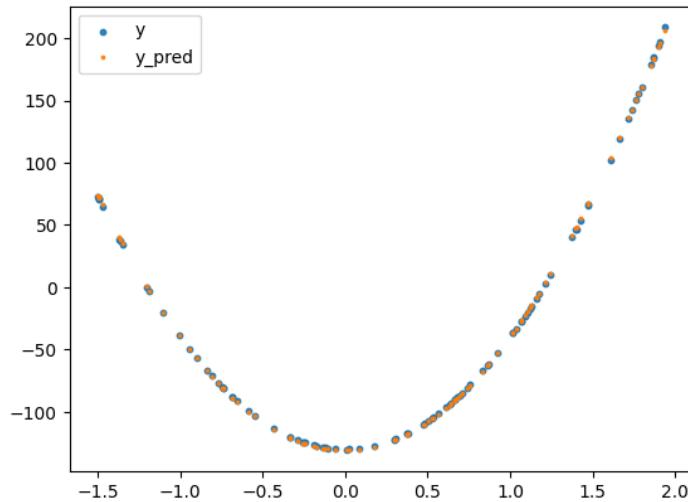
Rysunek 2: Porównanie wyników dla architektur 1-5-1 i 1-5-5-1 na zbiorze steps-large

## Wyniki - square-simple

Dla zbioru square-simple najsensowniejsze dopasowanie udało się uzyskać dla architektury 1-5-5-1. Dla tej architektury MSE wynosiło mniej niż 1 zarówno na zbiorze treningowym, jak i testowym.

	MSE (train)	MSE (test)
Wynik	0.74	0.78

Tabela 2: Wartości funkcji straty (MSE) dla zbioru square-simple



Rysunek 3: 1-5-5-1: MSE = 0.74 (train)

## NN2: Propagacja wsteczna

### Cel

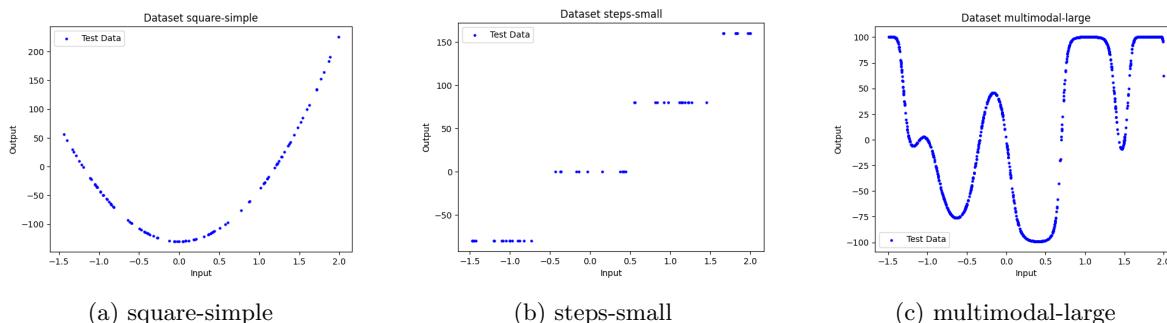
- a) Zaimplementowanie algorytmu propagacji wstecznej (backpropagation) w celu automatycznego wyznaczania gradientów i aktualizacji wag w sieci neuronowej.
- b) Przeprowadzenie eksperymentów porównujących skuteczność uczenia sieci z propagacją wsteczną na zbiorach danych:
  - square-simple
  - steps-small
  - multimodal-large
- c) Analiza wpływu liczby epok, współczynnika uczenia oraz architektury sieci na proces uczenia i jakość predykcji.

### Implementacja

Na tym etapie do implementacji zostały dodane następujące funkcjonalności:

- klasa **Initializer()** - jej podklasy odpowiadają za inicjalizację wag i biasów w sieci neuronowej. Wykorzystano różne metody inicjalizacji, takie jak *Xavier* i *He*, które są dostosowane do różnych funkcji aktywacji.
- klasa **ActivationFunction()** - jej podklasy implementują różne funkcje aktywacji, takie jak *ReLU*, *sigmoid* i *tanh*. Każda z tych funkcji jest dostosowana do specyfiki danych wejściowych oraz architektury sieci.
- klasa **ModelHistory()** - odpowiada za przechowywanie i analizę historii uczenia, w tym wartości funkcji straty (MSE) dla zbiorów treningowych i testowych. Umożliwia zapamiętanie parametrów sieci neuronowej oraz wygenerowanie wykresów ilustrujących historię procesu uczenia.
- **Mechanizm backpropagacji** - podzielony na trzy etapy: forward (w celu wyliczenia aktualnej predykcji), backward (w celu wyliczenia błędów oraz zmian gradientów) oraz update (aktualizacja wag i biasów w sieci neuronowej). Zapewniona została możliwość trenowania modelu z wykorzystaniem SGD, mini-batch oraz batch (uczenia na całym zbiorze danych). Zauważalne było, że model pracuje szybciej (pod względem mijających epok) przy użyciu mniejszych partii danych, (mini-batch, lub w szczególności SGD). Wiązało się to jednak z mniejszą stabilnością wyników.

### Zbiory danych



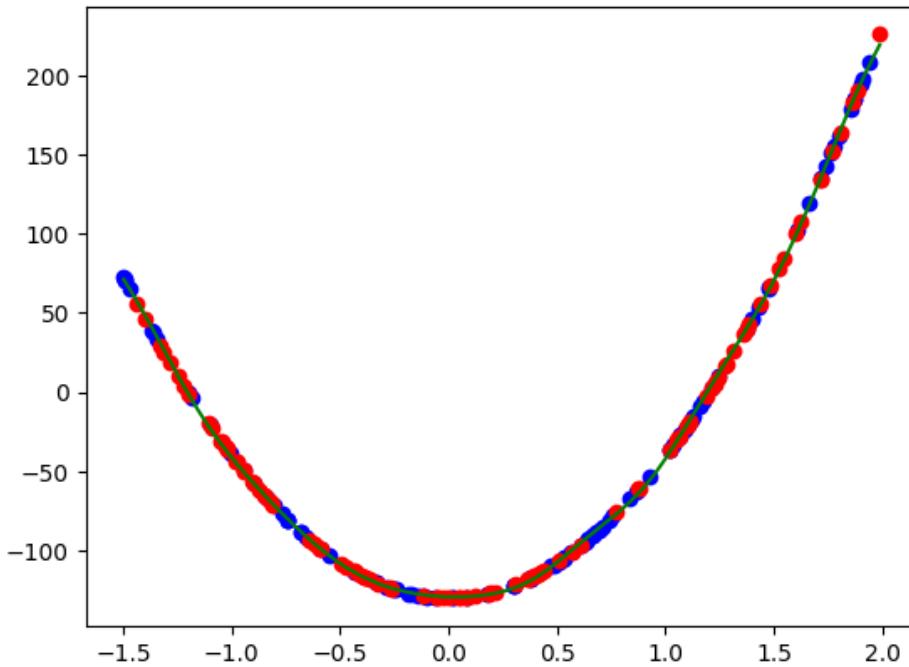
Rysunek 4: Prezentacja zbiorów danych: square-simple, steps-small, multimodal-large

## Trening

Dla każdego ze zbiorów został wytrenowany model. Trening odbywał się zarówno na całym zbiorze danych w każdej iteracji, jak i przy użyciu mini-batch oraz SGD. W przypadku zbioru multimodal-large, ze względu na jego dużą wielkość, lepsze rezultaty zdawał się osiągać trening z użyciem mini-batch. Dane treningowe w zbiorze steps-large okazały się przesunięte w stosunku do danych testowych, co w połączeniu ze skokową naturą danych, skutkowało dużymi wartościami funkcji straty (MSE) dla tego zbioru w przypadku ewaluacji na zbiorze treningowym.

### Square-simple

Wykorzystano architekturę **1-50-100-50-1**



Rysunek 5: Dopasowanie modelu do zbioru square-simple przy użyciu batch

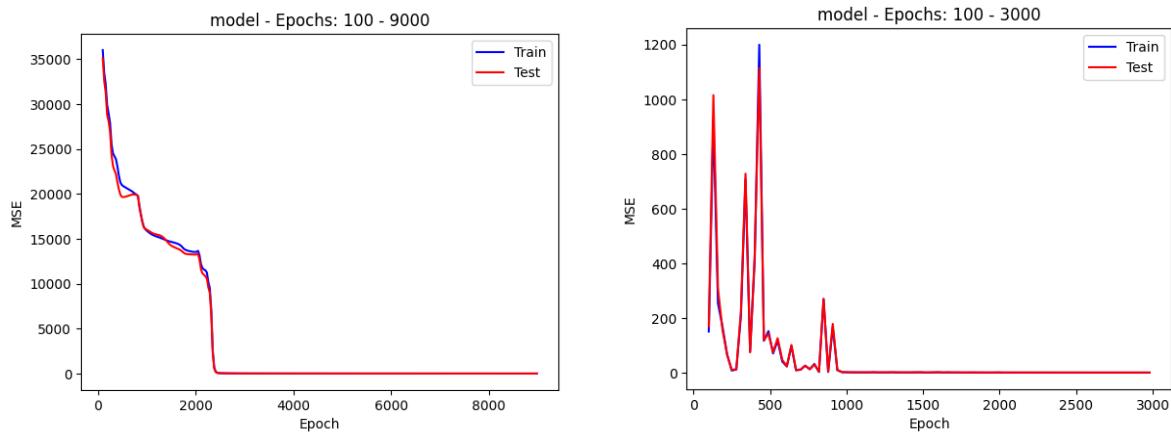
## Wyniki

Metoda	Liczba epok	MSE (train)	MSE (test)
batch	9000	1.51	1.98
mini-batch	3000	0.70	1.05

Tabela 3: Model na mini-batch uczył się krócej, ale osiągnął lepsze wyniki.

### Historia uczenia (loss curves)

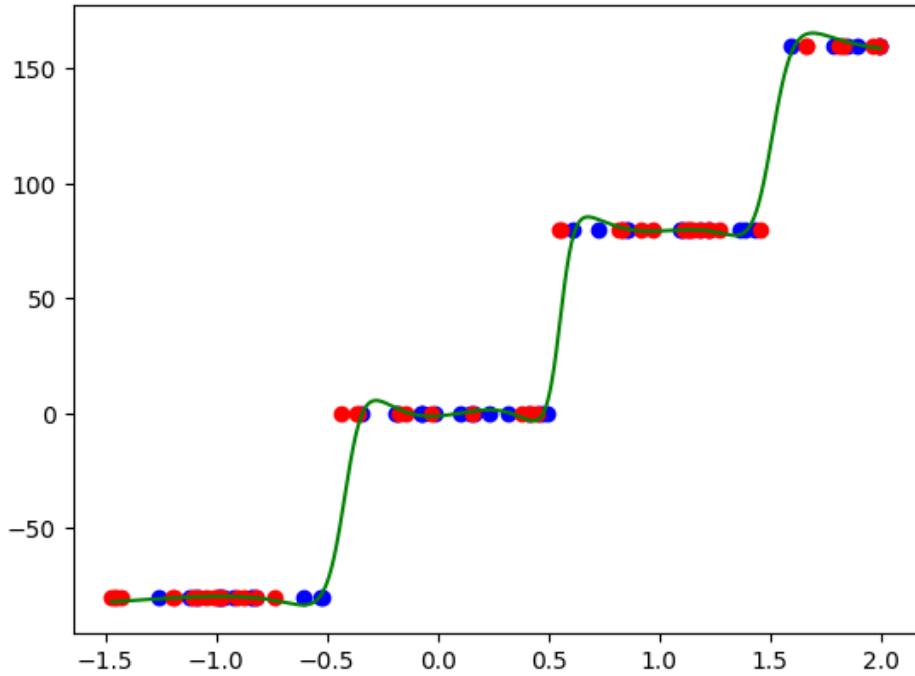
Poniższe wykresy przedstawiają porównanie procesu nauki modelu przy użyciu metod batch oraz mini-batch. Można zauważyć, że model przy uczeniu mini-batch działa bardziej chaotycznie, ale szybciej osiągał lepsze rezultaty. W przypadku batch model uczył się wolniej, ale stabilniej.



(a) Przebieg funkcji straty (MSE) podczas uczenia na zbiorze square-simple (batch)

(b) Przebieg funkcji straty (MSE) podczas uczenia na zbiorze square-simple (mini-batch)

Rysunek 6: Porównanie procesu uczenia (batch vs mini-batch) na zbiorze square-simple

**Steps-small**Wykorzystano architekturę **1-50-100-50-1**

Rysunek 7: Dopasowanie modelu do zbioru steps-small przy użyciu mini-batch

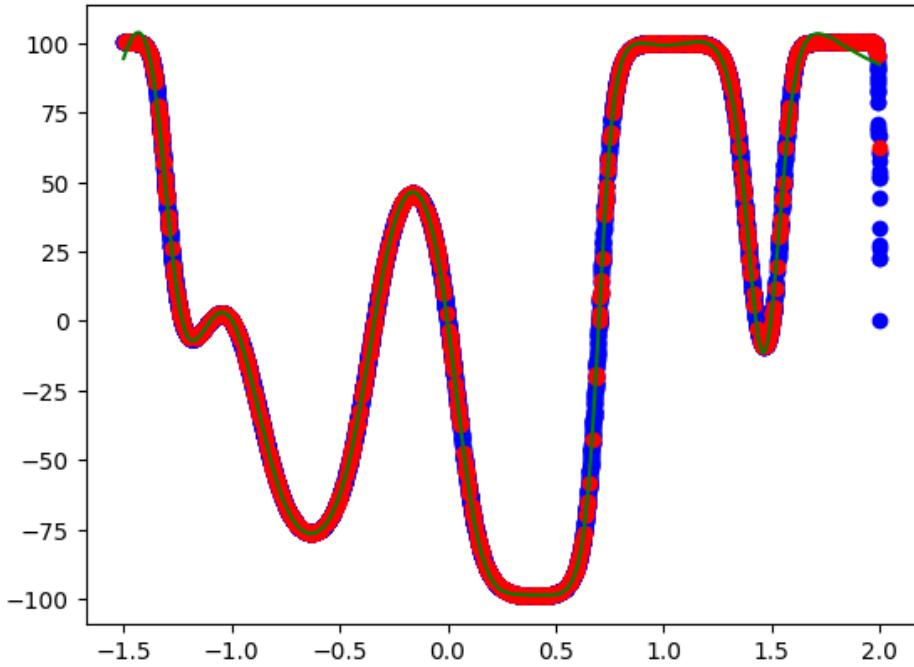
**Wyniki**

Metoda	Liczba epok	MSE (train)	MSE (test)
batch	103000	2.66	109.46
mini-batch	17000	3.98	104.91

Tabela 4: Porównanie wyników dla batch i mini-batch na zbiorze steps-small.

## Multimodal-large

Wykorzystano architekturę 1-50-100-50-1



Rysunek 8: Dopasowanie modelu do zbioru multimodal-large przy użyciu mini-batch

## Wyniki

Metoda	Liczba epok	MSE (train)	MSE (test)
mini-batch	700	7.47	3.78

Tabela 5: Porównanie wyników dla mini-batch na zbiorze multimodal-large.

## NN3: Implementacja momentu i normalizacji (RMSProp)

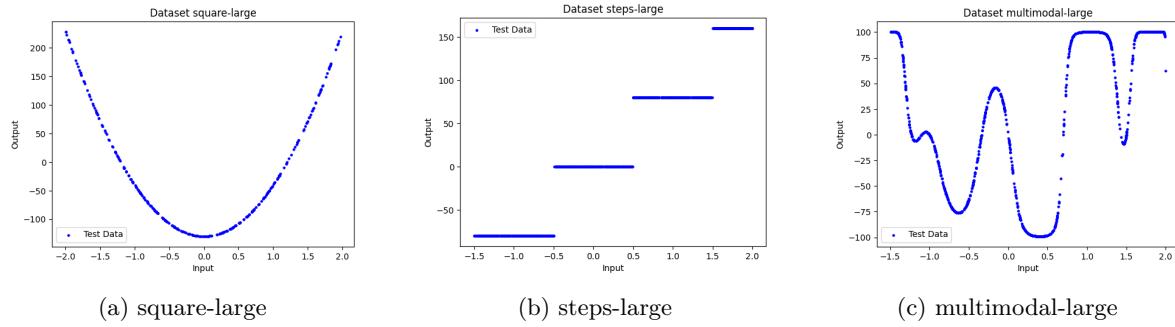
### Cel

- Dodanie do implementacji sieci neuronowej mechanizmu uczenia z momentem oraz normalizacji gradientu RMSProp w celu poprawy procesu uczenia na trudnych zbiorach danych.
- Przeprowadzenie eksperymentów porównujących skuteczność uczenia z momentem i RMSProp względem klasycznego SGD lub mini-batch na zbiorach danych:
  - square-large
  - steps-large
  - multimodal-large
- Analiza wpływu parametrów momentu i RMSProp (np. współczynnika momentu, parametru  $\beta$ ) na szybkość i stabilność uczenia.

### Implementacja

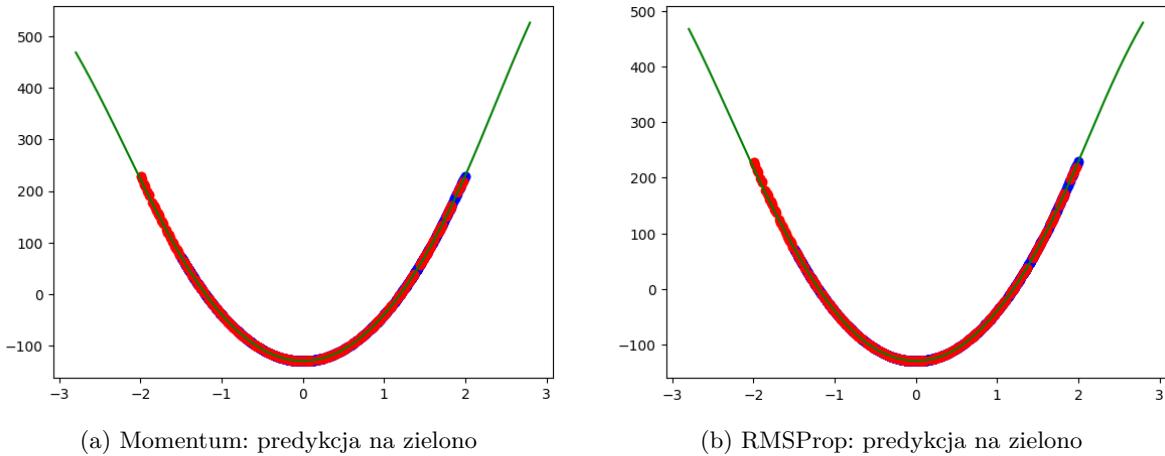
Do metody `train()` dodany został parametr `optimizer`, który pozwala na wybór metody optymalizacji. Dodatkowo można również podać parametry specyficzne dla danej metody, takie jak współczynnik momentu `momentum_lambda` dla metody z momentem oraz `rms_beta` dla RMSProp.

## Zbiory danych



Rysunek 9: Prezentacja zbiorów danych: square-large, steps-large, multimodal-large

## Square-large



Rysunek 10: Porównanie dopasowania na zbiorze square-large

## Wyniki

Metoda optymalizacji	Liczba epok	MSE (train)	MSE (test)
Momentum	3778	0.27	1.05
RMSProp	129	0.56	3.54

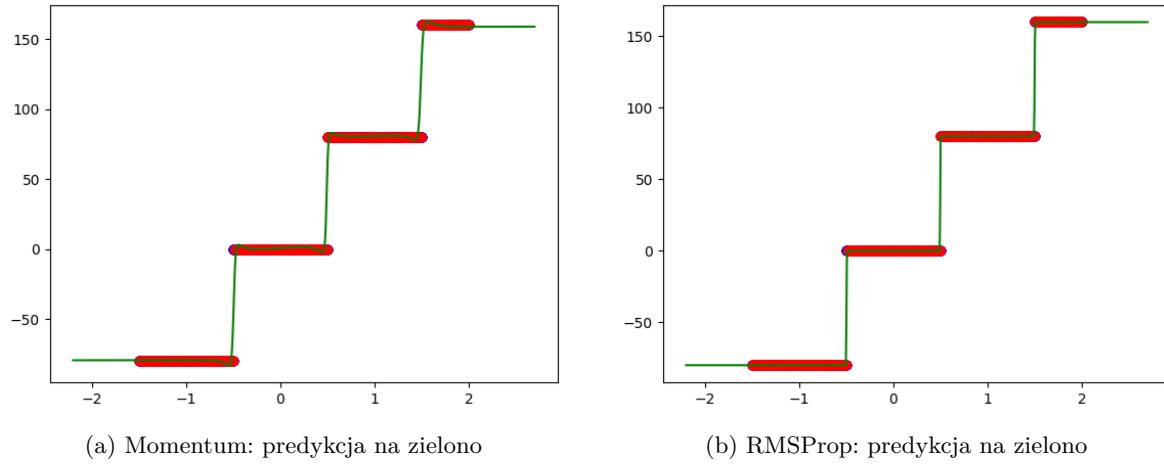
Tabela 6: Porównanie skuteczności optymalizatorów Momentum i RMSProp na zbiorze square-large (MSE dla zbioru treningowego i testowego).

## Obserwacje

Trening z wykorzystaniem momentu przebiegał w sposób jednostajny. Modelowi udało się "dociągnąć" odstające (przez brak danych treningowych) ramię, co pozwoliło na osiągnięcie satysfakcyjnych wyników na zbiorze testowym. W przypadku RMSProp model uczył się znacznie szybciej, (tylko 129 epok!) i osiągnął podobne wyniki, chociaż dla tego zbioru moment wydaje się być lepszym rozwiązaniem.

## Steps-large

Dla tego zbioru danych, przy treningu z momentem, najlepiej zdawało się funkcjonować SGD. Z kolei RMSProp najlepiej działał przy użyciu mini-batch.



Rysunek 11: Porównanie dopasowania na zbiorze steps-large

## Wyniki

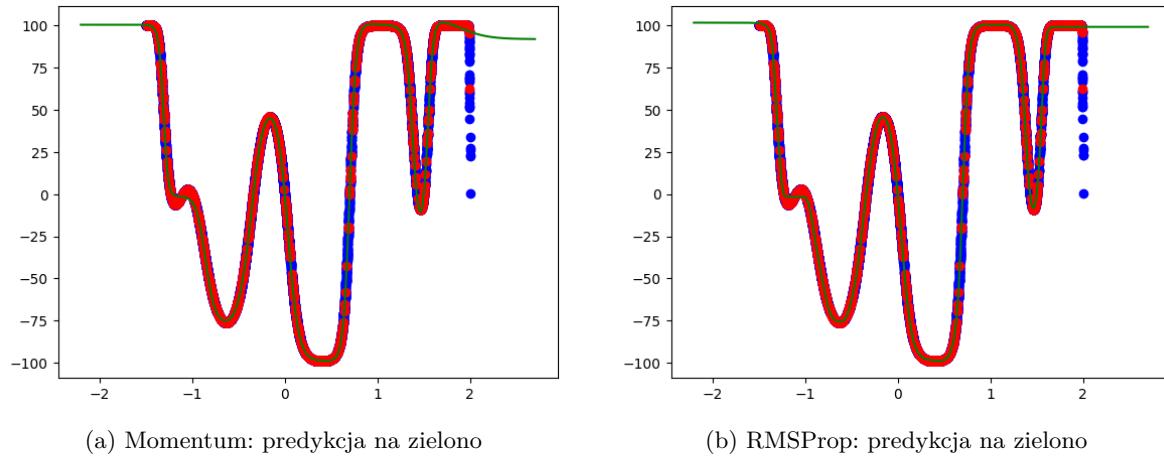
Metoda optymalizacji	Liczba epok	MSE (train)	MSE (test)
Momentum	180	25.71	15.27
RMSProp	2139	5.88	0.7

Tabela 7: Porównanie skuteczności optymalizatorów Momentum i RMSProp na zbiorze steps-large (MSE dla zbioru treningowego i testowego).

## Obserwacje

W wypadku tego zbioru danych, RMSProp okazał się znacznie lepszym rozwiązaniem niż Momentum. Model z wykorzystaniem RMSProp osiągnął bardzo dobre wyniki na zbiorze testowym, podczas gdy model z momentem po osiągnięciu MSE ok. 15, zaczął się przeuczać a wartość na zbiorze testowym wzrosła.

## Multimodal-large



Rysunek 12: Porównanie dopasowania na zbiorze multimodal-large

## Wyniki

Metoda optymalizacji	Liczba epok	MSE (train)	MSE (test)
Momentum	134	7.9	3.17
RMSProp	1194	7.34	2.21

Tabela 8: Porównanie skuteczności optymalizatorów Momentum i RMSProp na zbiorze multimodal-large (MSE dla zbioru treningowego i testowego).

### Obserwacje

Na tym zbiorze oba modele poradziły sobie bardzo dobrze.

### Ogólne spostrzeżenia

Niezależnie od modelu korzystne było stopniowe zmniejszanie learning-rate w trakcie uczenia. W przypadku uczenia z momentem, lambda w zakresie 0.7-0.9 dawała najlepsze rezultaty. Ważne było jednak stopniowe zwiększenie tego parametru, od wartości 0.3 przez 0.5 do 0.7. Pozwalało to uniknąć natychmiastowego wybicia wag w nieoczekiwany kierunku.

## NN4: Klasyfikacja - implementacja z softmax

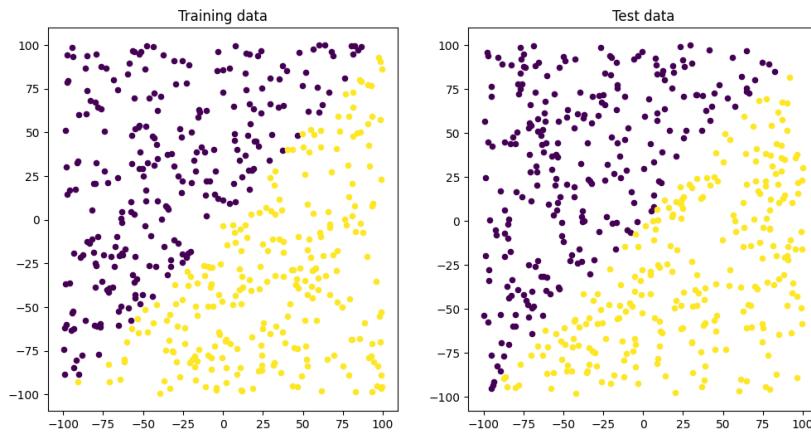
### Cel

- a) Rozszerzenie implementacji sieci neuronowej o możliwość rozwiązywania zadań klasyfikacyjnych poprzez dodanie funkcji aktywacji softmax.
- b) Implementacja funkcji kosztu cross-entropy odpowiedniej dla klasyfikacji wieloklasowej.
- c) Przetestowanie działania sieci na wybranych zbiorach danych klasyfikacyjnych oraz analiza jakości predykcji.

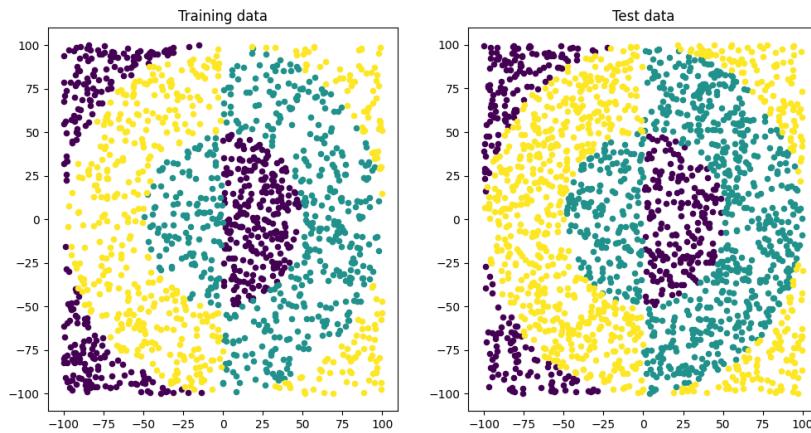
### Implementacja

Na tym etapie zaimplementowana została funkcja aktywacji softmax jako podklasa klasy *ActivationFunction()*, która jest wykorzystywana w warstwie wyjściowej sieci neuronowej do przekształcania wyników na prawdopodobieństwa dla poszczególnych klas. Dodatkowo, wprowadzono funkcję kosztu cross-entropy, która jest odpowiednia dla zadań klasyfikacyjnych i pozwala na efektywne uczenie się modelu.

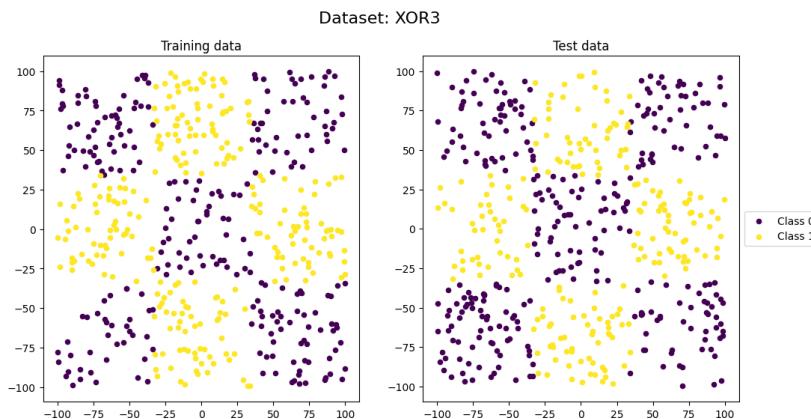
## Zbiory danych



Rysunek 13: Dataset: easy



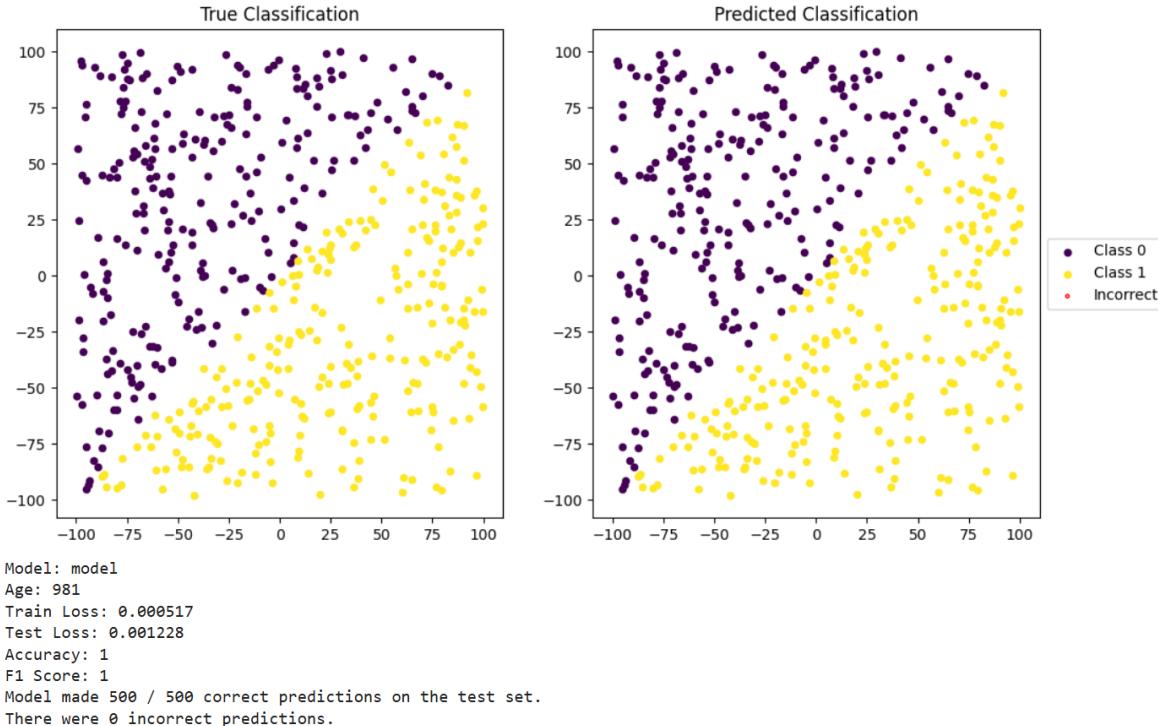
Rysunek 14: Dataset: rings3



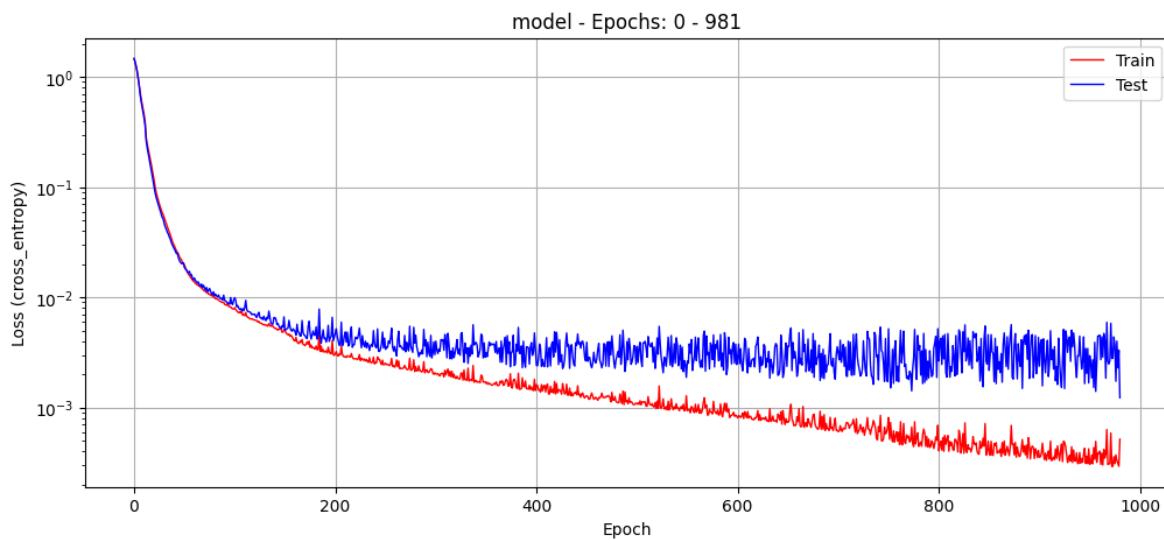
Rysunek 15: Dataset: xor3

## Simple

Ten zbiór był wyjątkowo prosty. Zarówno dla dedykowanej funkcji aktywacji *Softmax()* jak i dla funkcji aktywacji *Sigmoid()* model osiągnął 100% skuteczności na zbiorze testowym. W przypadku funkcji aktywacji *Softmax()* model uczył się szybciej, ale dla funkcji aktywacji *Sigmoid()* model również osiągnął bardzo dobre wyniki.



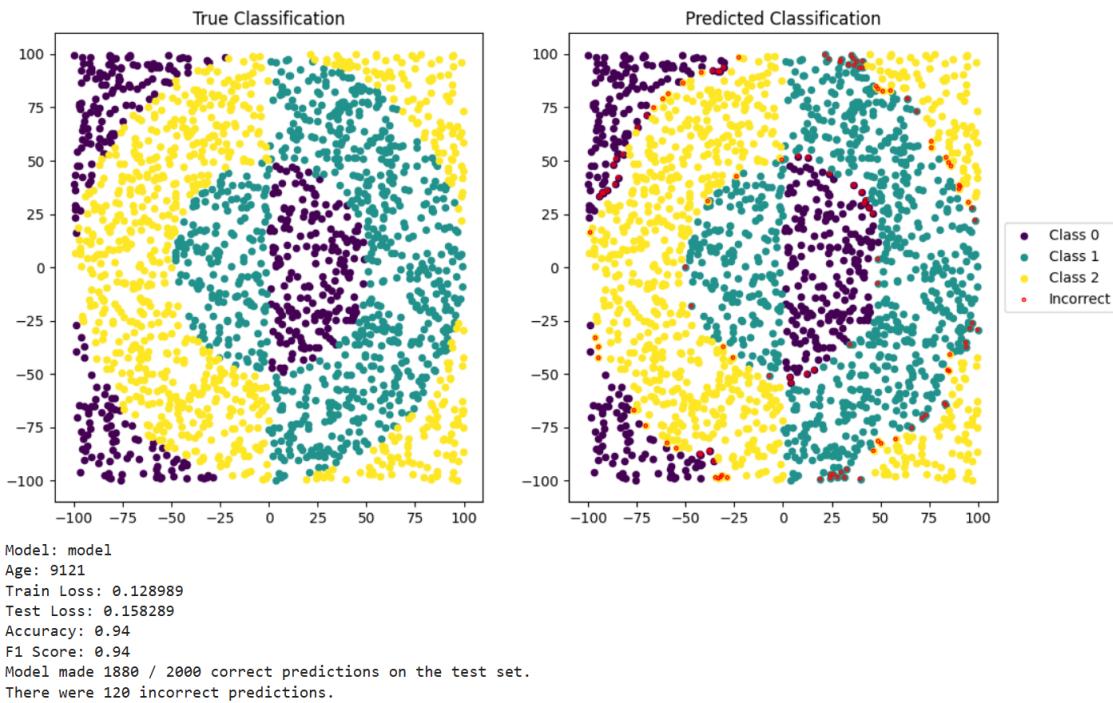
Rysunek 16: Simple: Dopasowanie modelu z obiema funkcjami dało 100% skuteczności.



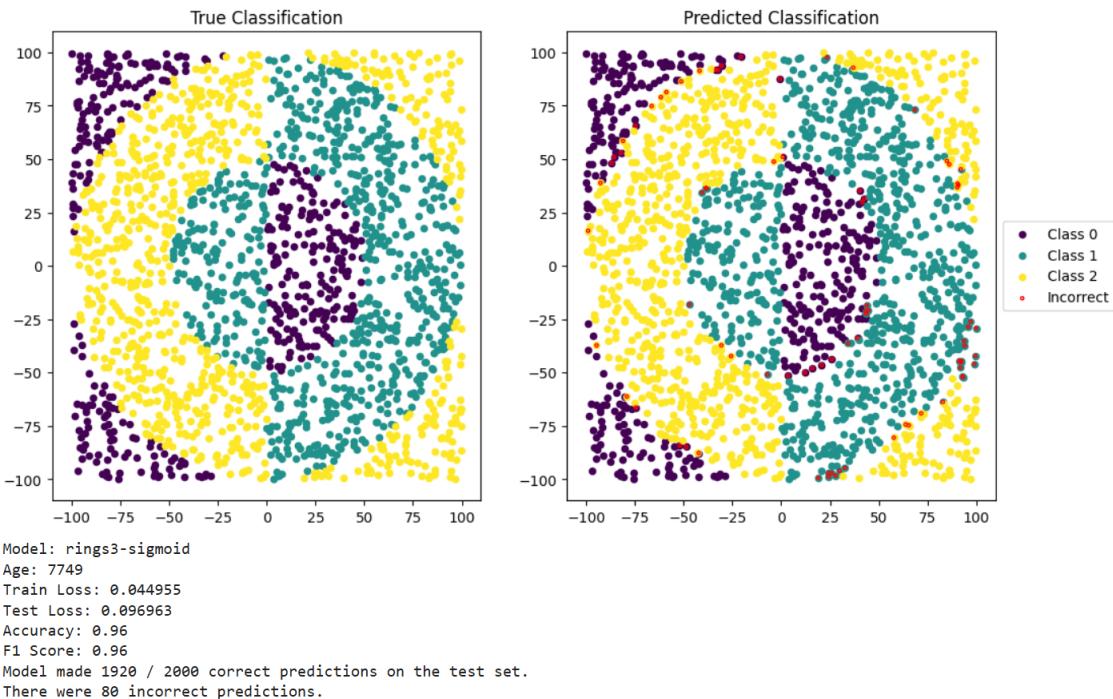
Rysunek 17: Simple: Historia uczenia modelu z funkcją aktywacji Softmax

## Rings3

Ten zbiór okazał się bardziej wymagający, ale również on nie sprawił modelowi większych problemów. Model korzystający z *Sigmoid()* poradził sobie nawet nieco lepiej niż model korzystający z *Softmax()*, co było zadziwiające.



Rysunek 18: Rings3: Dopasowanie modelu z funkcją aktywacji Softmax

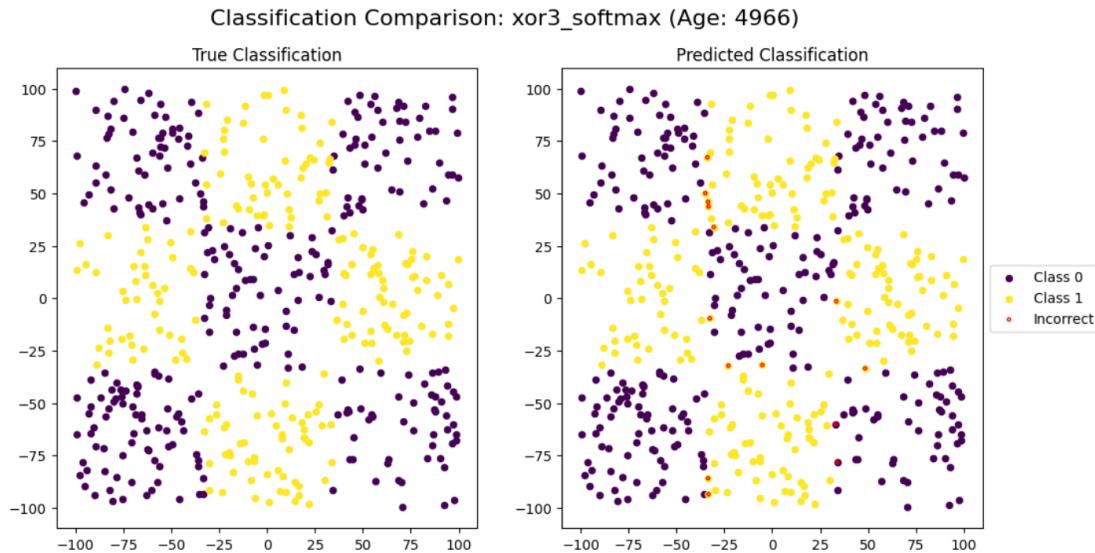


Rysunek 19: Rings3: Dopasowanie modelu z funkcją aktywacji Sigmoid

## Xor3

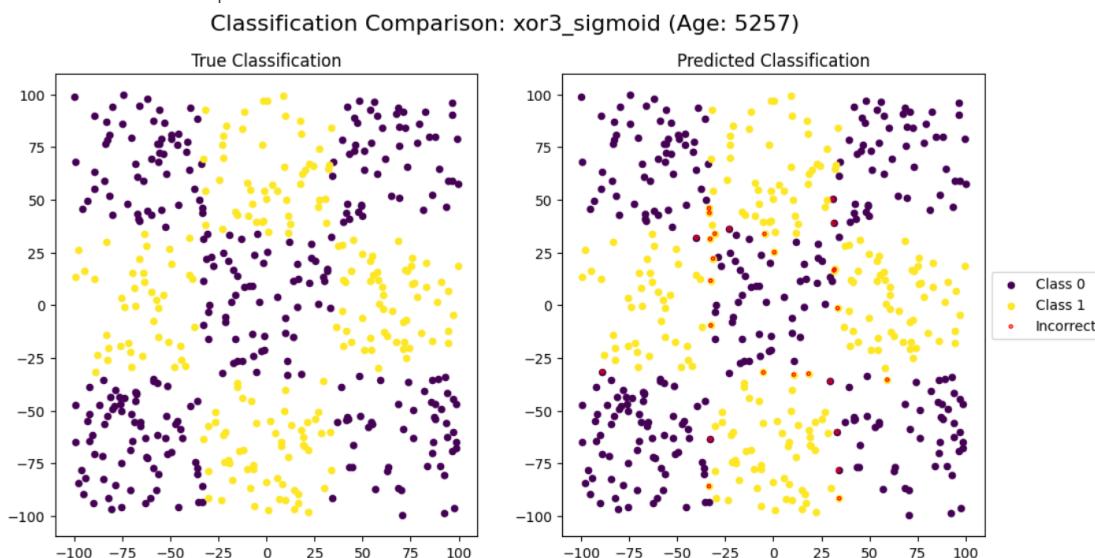
Zbiór xor3 również okazał się dosyć prosty. Jakość predykcji modeli w obu przypadkach była bardzo bardzo dobra, nieco lepsza dla Softmaxa.

```
Model: xor3_softmax
Age: 4966
Train Loss: 0.01442
Test Loss: 0.082856
Accuracy: 0.972
F1 Score: 0.9721
Model made 486 / 500 correct predictions on the test set.
There were 14 incorrect predictions.
```



Rysunek 20: Xor3: Dopasowanie modelu z funkcją aktywacji Softmax

```
Model: xor3_sigmoid
Age: 5257
Train Loss: 0.007942
Test Loss: 0.147957
Accuracy: 0.946
F1 Score: 0.9462
Model made 473 / 500 correct predictions on the test set.
There were 27 incorrect predictions.
```



Rysunek 21: Xor3: Dopasowanie modelu z funkcją aktywacji Sigmoid

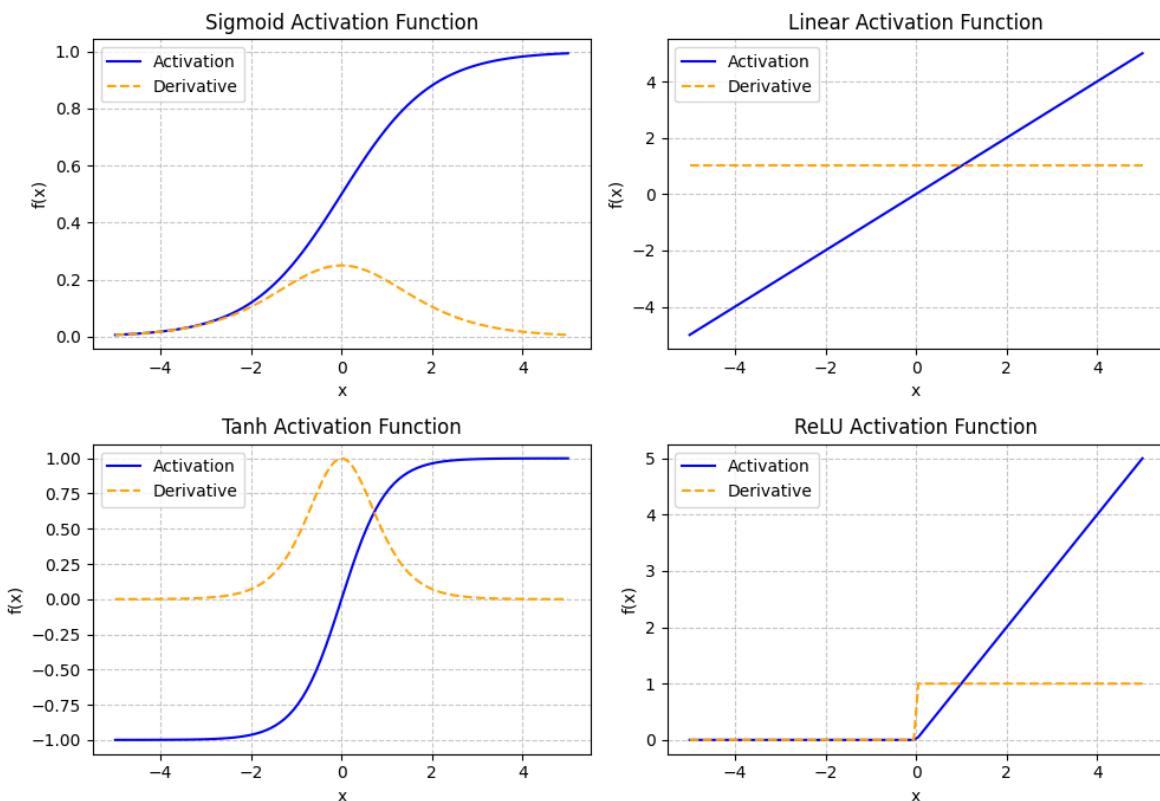
## NN5: Testowanie różnych funkcji aktywacji

### Cel

- a) Przetestowanie różnych kombinacji architektur i funkcji aktywacji w sieciach neuronowych, takich jak ReLU, tanh, sigmoid, funkcja liniowa oraz ich wpływu na proces uczenia i jakość predykcji.
- c) Szczegółowe testy dla 2 najlepszych kombinacji na zbiorach:
- steps-large
  - rings5-regular
  - rings3-regular

### Funkcje aktywacji i ich pochodne

Testowane były 4 funkcje aktywacji:



Rysunek 22: Funkcje aktywacji i ich pochodne

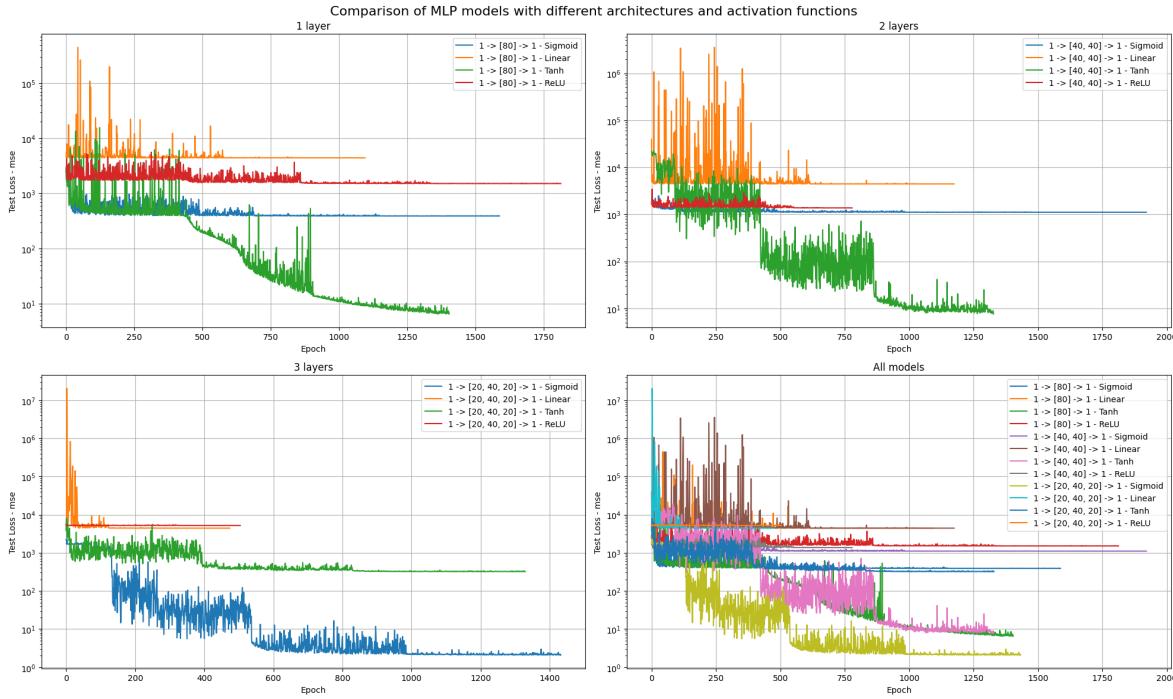
### Architektury

Testowane były 3 architektury, z 1, 2 oraz 3 warstwami ukrytymi. Każda architektura dzieliła między warstwy 80 neuronów:

- **1-80-1**
- **1-40-40-1**
- **1-20-40-20-1**

## Wstępne testy

Wstępne testy przeprowadzono na zbiorze *multimodal-large*.



Rysunek 23: Różne kombinacje architektur i funkcji aktywacji

## Wnioski

- Najlepsze wyniki osiągają modele z funkcją aktywacji **tanh** oraz **sigmoid**.
- Funkcja aktywacji **sigmoid** najlepiej funkcjonuje przy bardziej złożonej architekturze.
- W przypadku funkcji **tanh** zwiększenie liczby warstw architektury wpłynęło negatywnie na osiągnięty wynik. Najlepsze rezultaty uzyskano już przy 1 lub 2 warstwach.

Można wyróżnić 3 konkurencyjne modele:

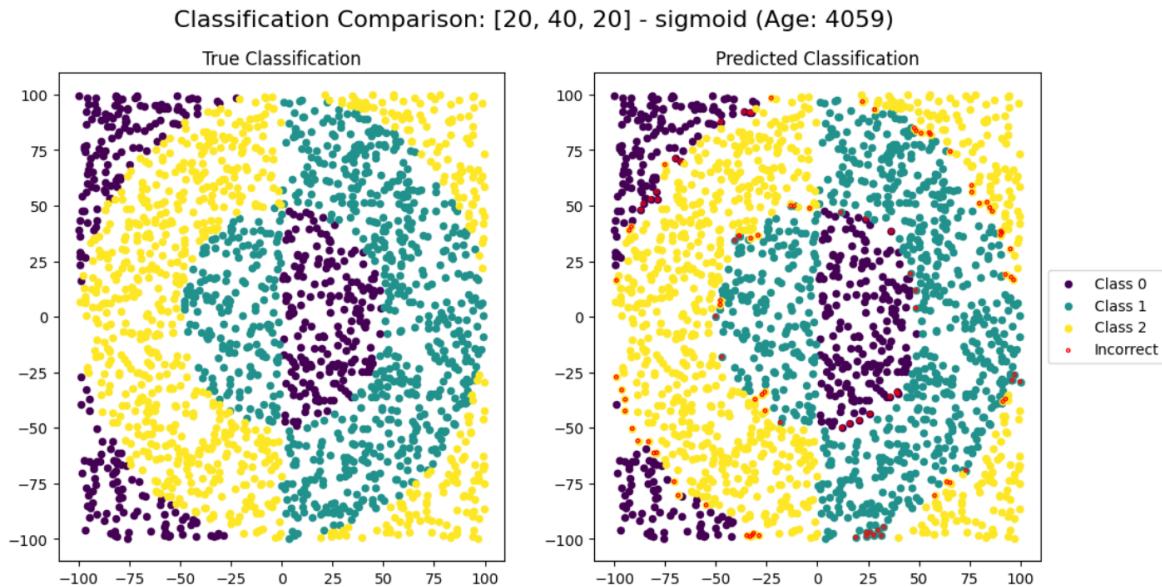
- **1-80-1** z funkcją aktywacji **tanh**
- **1-40-40-1** z funkcją aktywacji **tanh**
- **1-20-40-20-1** z funkcją aktywacji **sigmoid**

Te właśnie modele będą testowane na kolejnych zbiorach.

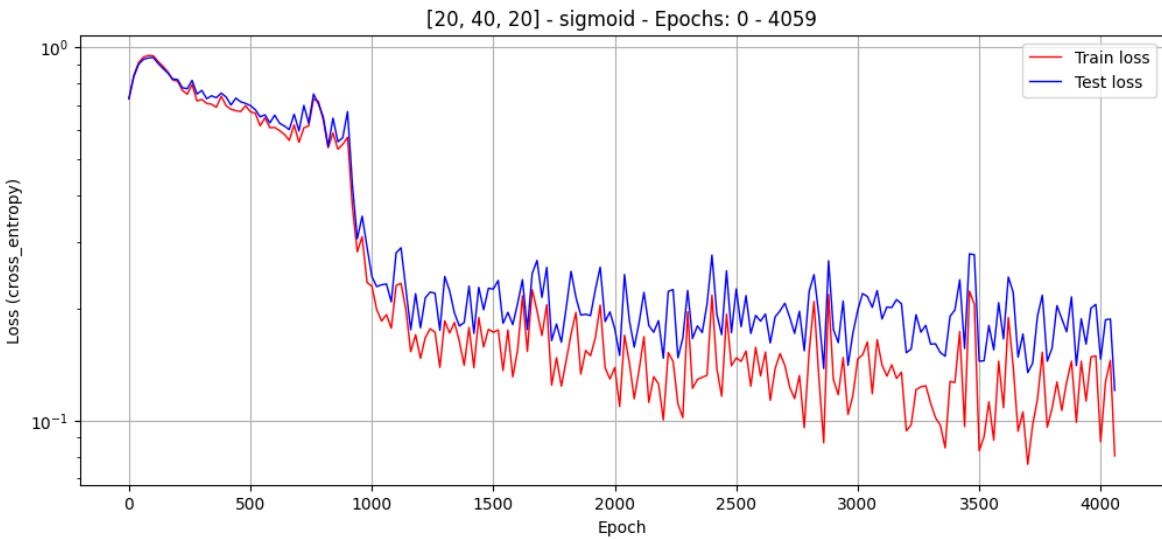
## Wyniki dla zbioru rings3-regular

```

Model: [20, 40, 20] - sigmoid
Age: 4059
Train Loss: 0.080453
Test Loss: 0.120444
Accuracy: 0.9495
F1 Score: 0.9495
Model made 1899 / 2000 correct predictions on the test set.
There were 101 incorrect predictions.
  
```



Rysunek 24: Dopasowanie modeli do zbioru rings3-regular



Rysunek 25: Historia uczenia modeli na zbiorze rings3-regular (cross\_entropy)

Wszystkie trzy testowane modele:

- **1-80-1** z funkcją aktywacji **tanh**
- **1-40-40-1** z funkcją aktywacji **tanh**
- **1-20-40-20-1** z funkcją aktywacji **sigmoid**

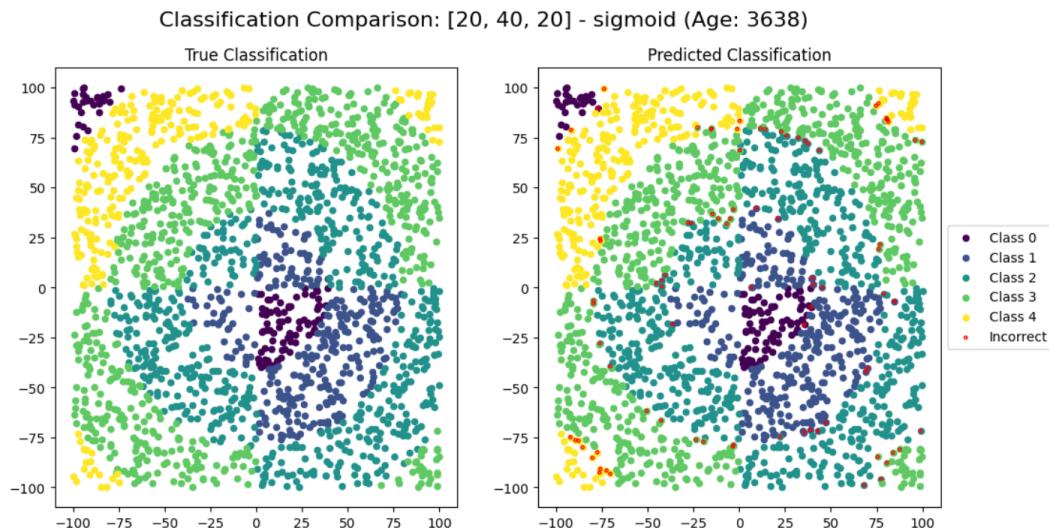
uzyskały bardzo zbliżone wyniki na zbiorze *rings3-regular*. Każdy z modeli osiągnął wartość F1-score na poziomie około 0,95 po 3000–5000 epokach treningu. Dalsze zwiększanie liczby epok nie przynosiło już istotnej poprawy jakości predykcji.

Architektura	Funkcja aktywacji	F1-score
1-80-1	tanh	0,9476
1-40-40-1	tanh	0,9475
1-20-40-20-1	sigmoid	0,9495

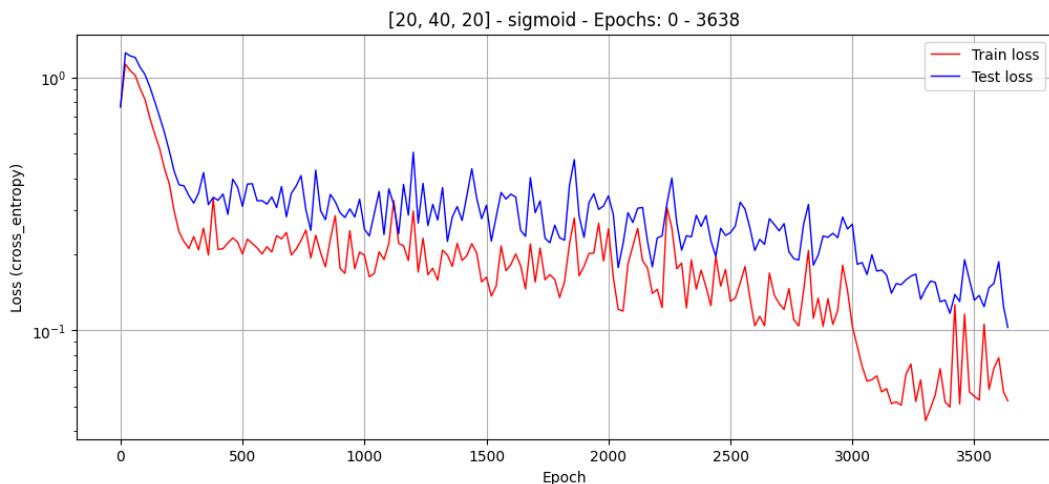
Wnioski: wszystkie testowane konfiguracje sieci neuronowych pozwalają na skuteczne rozwiązywanie zadania klasyfikacji na tym zbiorze danych.

## Wyniki dla zbioru rings5-regular

```
Model: [20, 40, 20] - sigmoid
Age: 3638
Train Loss: 0.052707
Test Loss: 0.102811
Accuracy: 0.957
F1 Score: 0.957
Model made 1914 / 2000 correct predictions on the test set.
There were 86 incorrect predictions.
```



Rysunek 26: Dopasowanie modelu do zbioru rings5-regular



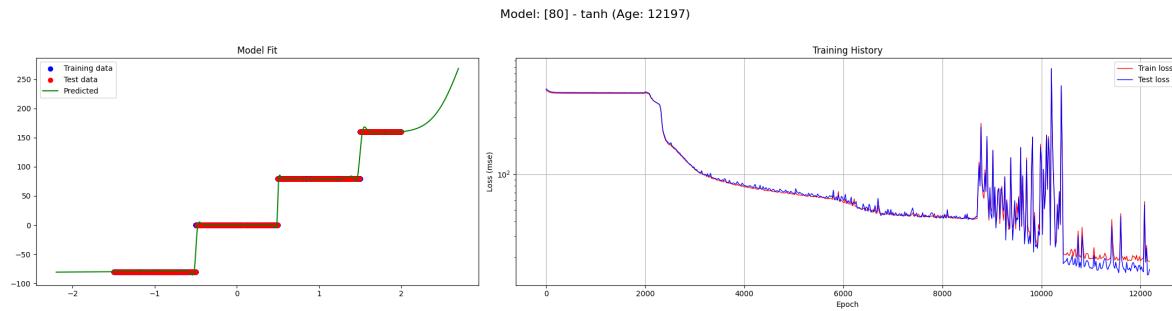
Rysunek 27: Historia uczenia modelu na zbiorze rings5-regular (cross\_entropy)

Zdecydowanie najlepszy okazał się model **1-20-40-20-1** z funkcją aktywacji **sigmoid**, który osiągnął wartość F1-score na poziomie około 0,957 po 3638 epokach treningu.

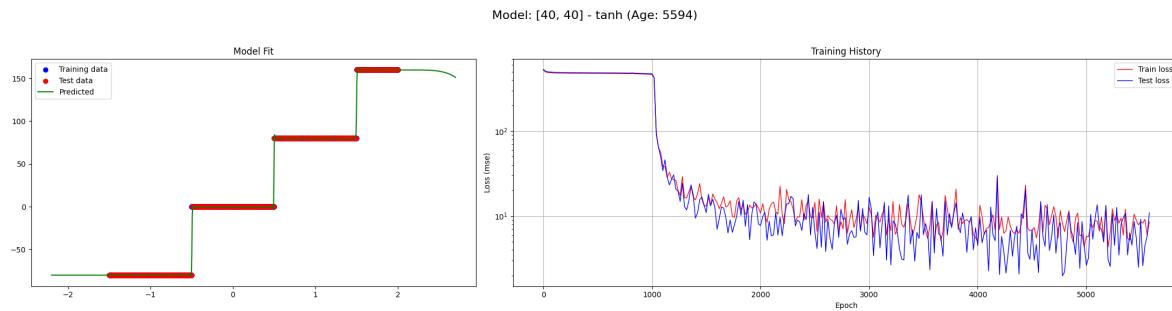
Architektura	Funkcja aktywacji	F1-score
1-80-1	tanh	0,9178
1-40-40-1	tanh	0,9339
1-20-40-20-1	sigmoid	0,957

Wszystkie modele pozwalają na rozwiązywanie zadania klasyfikacji na tym zbiorze danych, jednak model z funkcją aktywacji *sigmoid* okazał się zdecydowanie najlepszy.

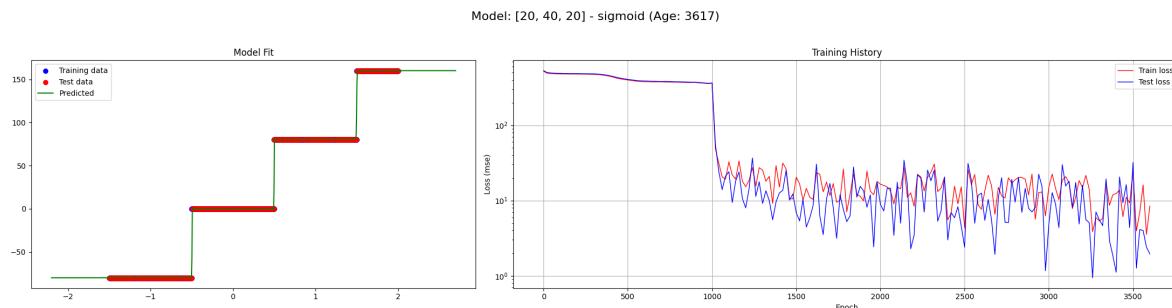
## Wyniki dla zbioru steps-large



Rysunek 28: Dopasowanie oraz historia uczenia (log(MSE)) modelu z funkcją aktywacji **tanh** oraz jedną warstwą ukrytą **(1-80-1)**



Rysunek 29: Dopasowanie oraz historia uczenia (log(MSE)) modelu z funkcją aktywacji **tanh** oraz dwiema warstwami ukrytymi **(1-40-40-1)**



Rysunek 30: Dopasowanie oraz historia uczenia (log(MSE)) modelu z funkcją aktywacji **sigmoid**

Architektura	Funkcja aktywacji	Liczba epok	MSE (train)	MSE (test)
1-80-1	tanh	12197	18.57	14.09
1-40-40-1	tanh	5594	6.56	1.44
1-20-40-20-1	sigmoid	3617	6.24	0.48

### Wnioski

Wszystkie modele poradziły sobie z zadaniem na przyzwoitym poziomie.

Model **1-80-1** z funkcją aktywacji **tanh** ze względu na prostą architekturę miał trudności z dokładnym odwzorowaniem przejść pomiędzy stopniami w zbiorze *steps-large*.

Modele **1-40-40-1** z funkcją **tanh** oraz **1-20-40-20-1** z funkcją **sigmoid** osiągnęły lepsze wyniki (MSE na poziomie około 1), szczególnie dobrze radząc sobie z odwzorowaniem miejsc przejść. Głębsze architektury oraz odpowiedni dobór funkcji aktywacji pozwoliły lepiej uchwycić złożoność zbioru danych. Ogólnie, zwiększenie głębokości modelu oraz tuning funkcji aktywacji poprawiły rezultaty na zbiorze *steps-large*.

## NN6: Regularyzacja i przeuczczanie

### Cel

- a) Zaimplementowanie mechanizmów regularyzacji w sieciach neuronowych, takich jak L1 oraz L2 w celu ograniczenia przeuczczania.
- b) Przeprowadzenie eksperymentów porównujących skuteczność regularyzacji na wybranych zbiorach danych, ze szczególnym uwzględnieniem wpływu na generalizację modelu.

### Implementacja

Na tym etapie do implementacji zostały dodane następujące mechanizmy regularyzacji:

- **Regularyzacja L1 oraz L2** – umożliwiająca ograniczenie wartości wag w sieci neuronowej, co pozwala na redukcję ryzyka przeuczczania. Współczynniki regularyzacji można ustawić osobno dla L1 i L2, co zapewnia elastyczność w doborze odpowiedniej metody do konkretnego zadania.
- **Parametr save\_till\_best** w funkcji **train()** – umożliwia automatyczne zapisywanie najlepszego modelu (na podstawie wyniku na zbiorze testowym) podczas procesu uczenia oraz przywrócenie tych parametrów po zakończeniu treningu. Takie podejście pozwala na uniknięcie sytuacji, w której model osiąga coraz lepsze wyniki na zbiorze treningowym, ale gorsze na zbiorze testowym, co jest typowym objawem przeuczczania.