Neural Network

Franciszek Szczepaniak - 320660

April 2024

Abstract

Poniższy dokument jest sprawozdaniem z moich postępów z zajęć Metod Inteligencji Obliczeniowej. Jest to pierwsze sprawozdanie z tego przedmiotu skupiające się na sieciach Neuronowych. Zawiera ono opis tematów, sposób rozwiązania, matematyczne objaśnienia oraz wyjaśnione podejścia do każdego problemu. Wszystkie wypisane wyniki MSE są na nieznormalizowanych danych!!!

1 NN1- Bazowa implementacja

1.1 Opis zadania

Pierwszym zadaniem było zbudowanie "szkieletu" sieci neuronowej. Sieć ta miała przyjmować jako agrumenty:

- liczba danych wejściowych
- liczba outputów
- liczba warstw ukrytych
- liczba węzłów w warstwach ukrytych
- funkcja aktywacji
- wagi i biasy

W pierwszym zadaniu funkcją aktywacji była sigmoidą. Sigmoida jest zdefiniowana jako:

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

Na wyjściu natomiast miała znajdować się funkcja liniowa (identyczności) :

$$f(x) = x$$

Zadaniem było ręcznie dobranie wag i biasów tak aby mse na zbiorach testowych wynosiło mniej niż 9 dla:

- steps-large
- square-simple

Mieliśmy zaimplementować to dla 3 architektur:

- jedna warstwa ukryta, 5 neuronów,
- jedna warstwa ukryta, 10 neuronów,
- dwie warstwy ukryte, po 5 neuronów każda.

1.2 Implementacja

Sieć została zaimplementowana za pomocą klas w pythonie. Stworzyłem klasę NeuralNetwork oraz klasę Layer. Sieć składała się z wektora wejściowego, warstwy wyjściowej oraz listy z wieloma Layer w środku.Każde Layer zawierało wyliczone wartości w formie wektora, swoje wagi i biasy.

Do wyliczania błędu użylem funkcji MSE:

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

gdzie y i $\hat{y}$ to rzeczywista wartość i przewidywana wartość odpowiednio

Zaimplementowałem również funkcję feedforward, która liczyła wartości w warstwach po podaniu inputu i wag. Wyglądało to w ten sposób

- n - liczba węzłów na wejściu do warstwy
- m - liczba węzłów na wyjściu z warstwy
- input - liczba danych treningowych

Dla warstwy wejściowej:

$$\begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_m \end{bmatrix} \begin{bmatrix} x_1 & x_2 & \cdots & x_{input} \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix}$$

A w następnych warstwach następująco:

$$\begin{bmatrix} w_{11} & w_{12} & \cdots & w_{1n} \\ w_{21} & w_{22} & \cdots & w_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{m1} & w_{m2} & \cdots & w_{mn} \end{bmatrix} \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1input} \\ x_{21} & x_{22} & \cdots & x_{2input} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n1} & x_{n2} & \cdots & x_{ninput} \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix}$$

Każda wartośc w macierzy wyjściowej w warstwie była poddawana funkcji aktywacji. W ten sposób dostawaliśmy macierz wynikową z wartośćiami w danej warstwie o wymiarach m x input. Każda kolumna reprezentowała wyniki dla różnch inputów, a każdy wiersz odpowiadał wartości w danym neuronie. Na wyjściu otrzymujemy macierz 1 x input.

1.3 eksperymenty

Aby znaleźć ręcznie parametry wag i biasów na początek losowałem losowe wartości z przedziału (-1,1). Następnie pomogłem sobie prostym algorytmem, który zmieniał minimalnie wybrane wagi i biasy i patrzyl na mse. Jeśli mse się poprawiało, wtedy wagi te pozostawały zmienione. Poniżej pokazuję pojedyncze wykresy pokazujące jak układają się przewydiwywane wartości. Stopniowo poprawiając sieć

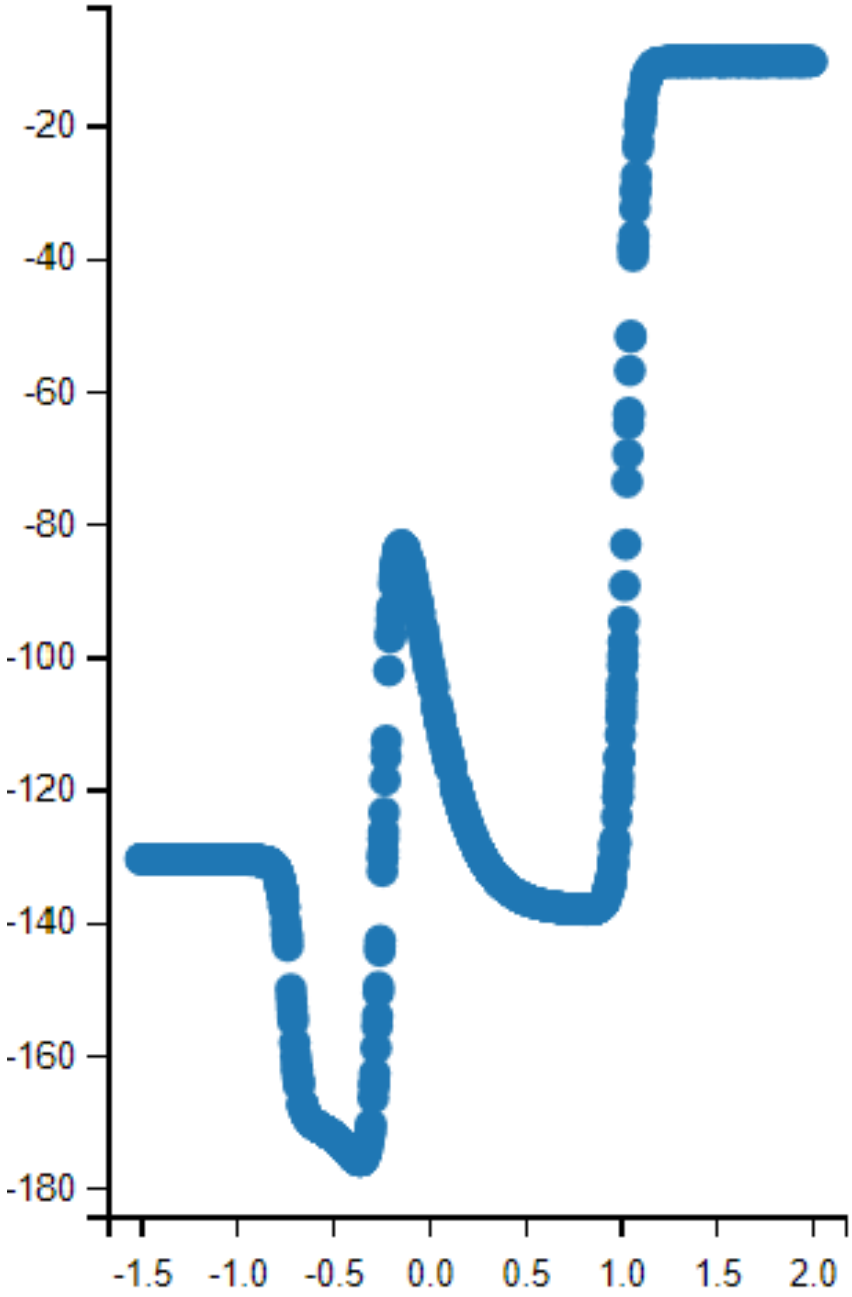


Figure 1: steps-large

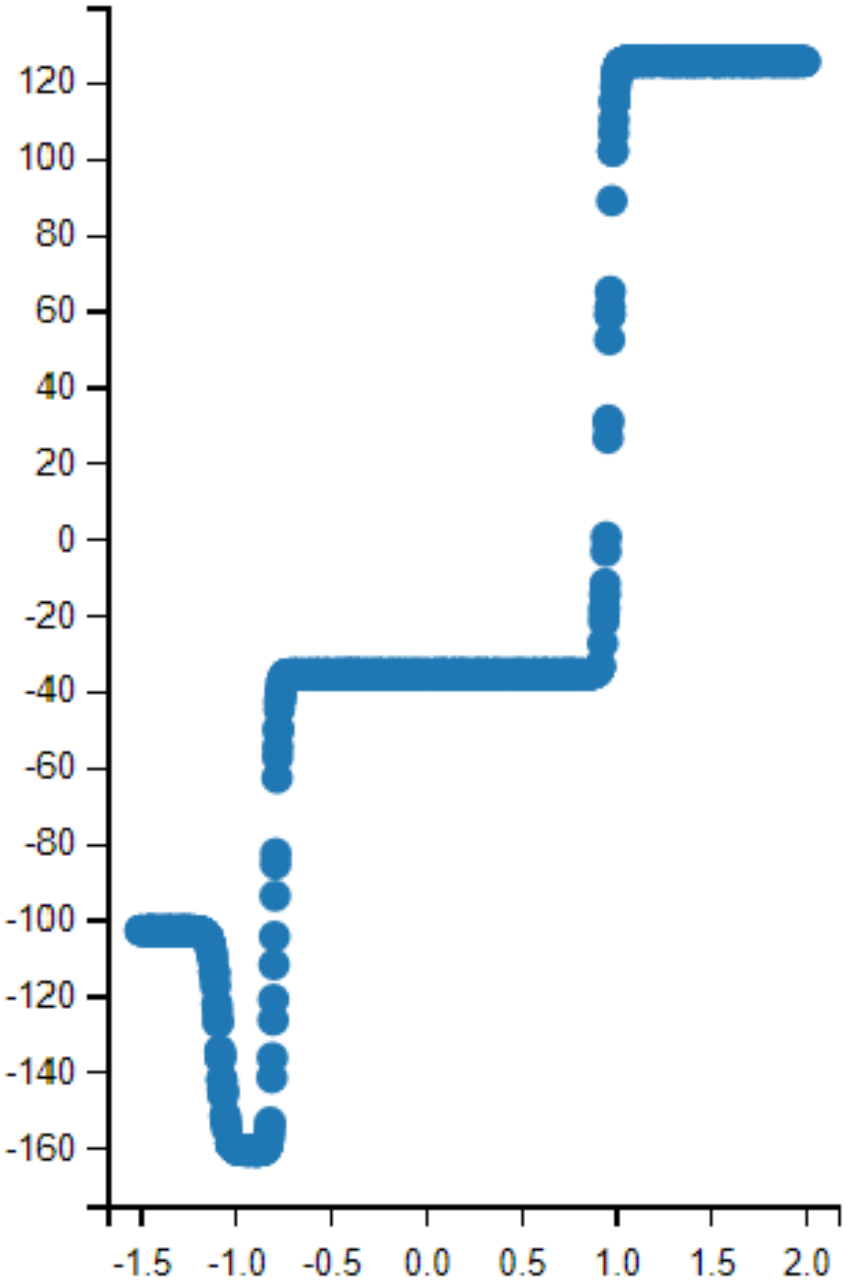


Figure 2: steps-large

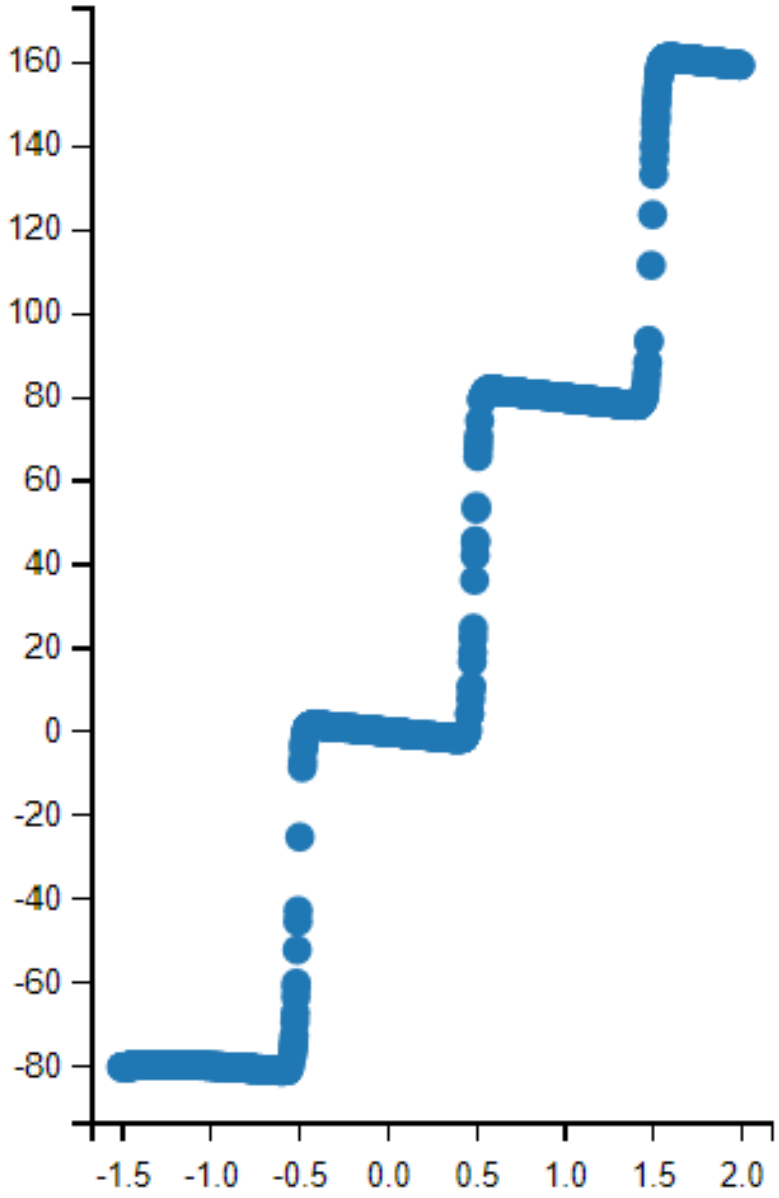


Figure 3: steps-large

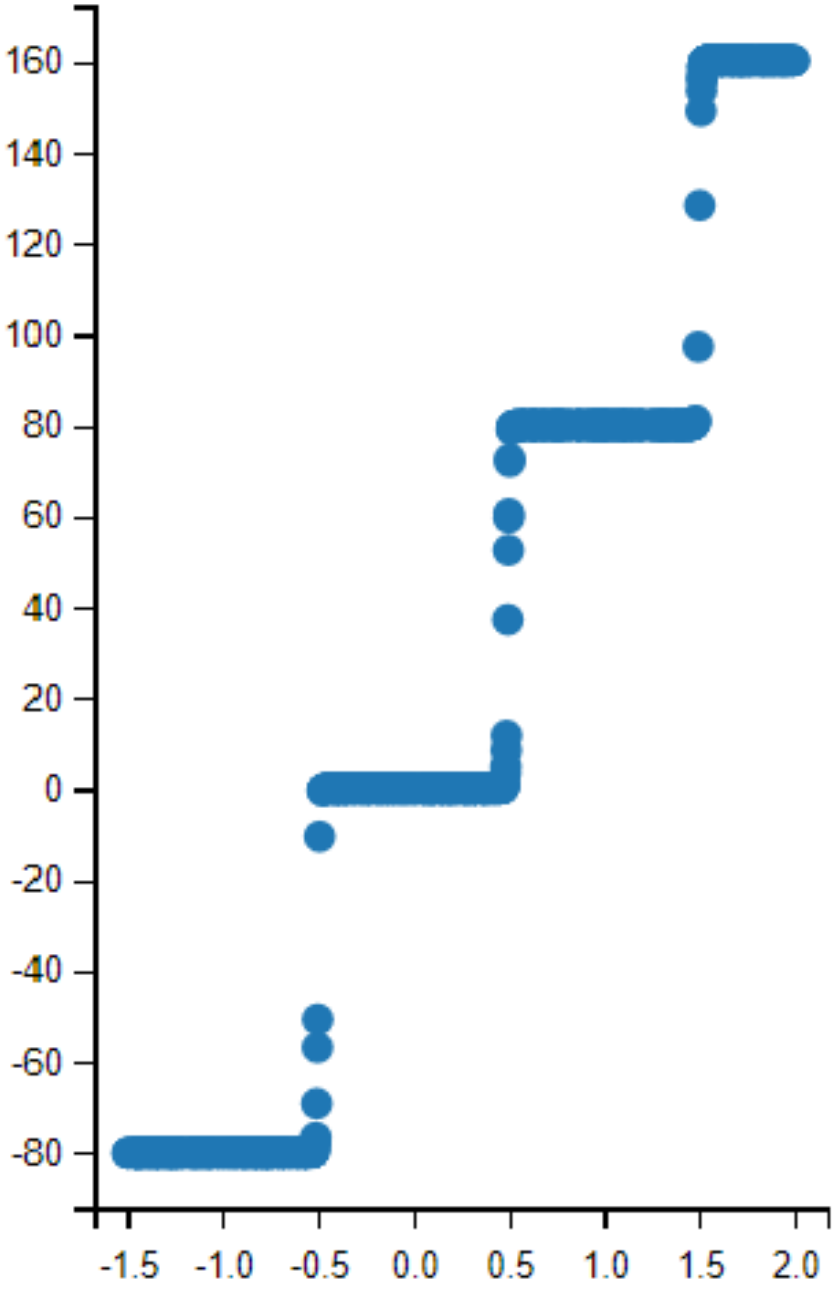


Figure 4: steps-large najlepsze

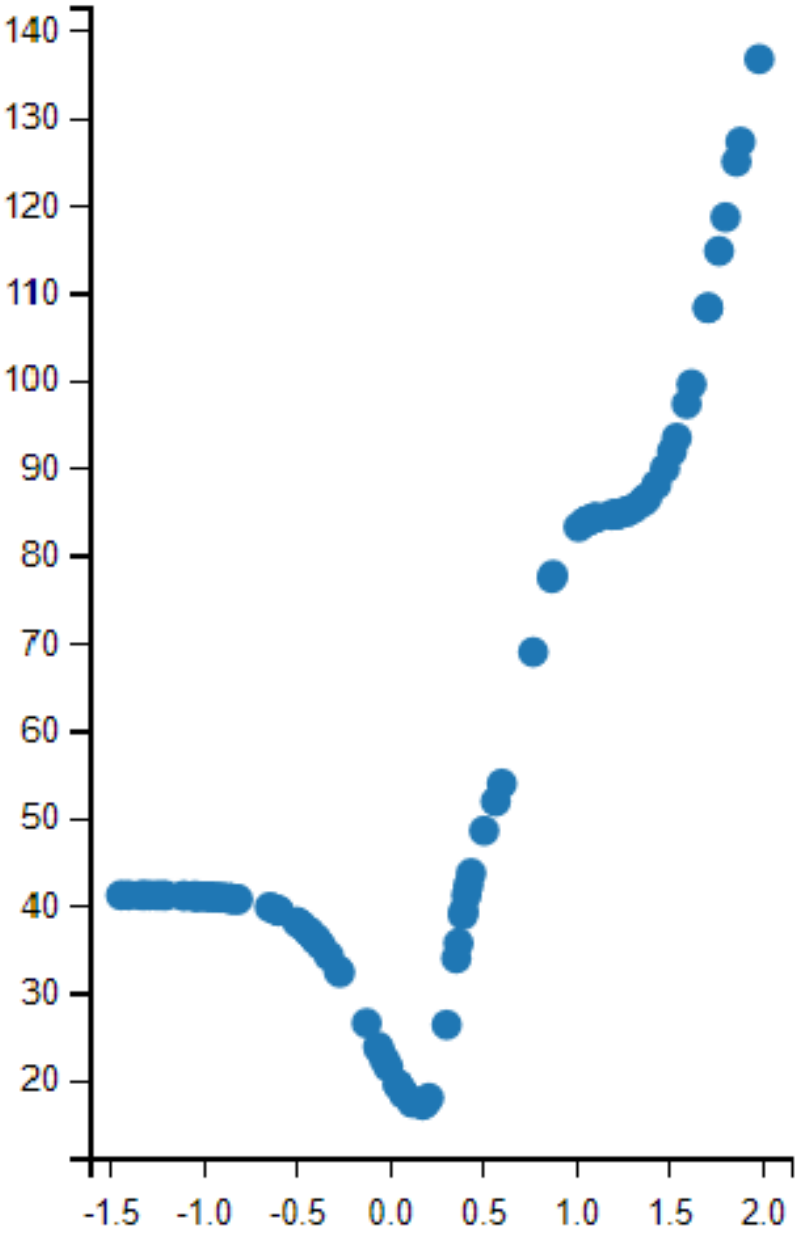


Figure 5: square-simple

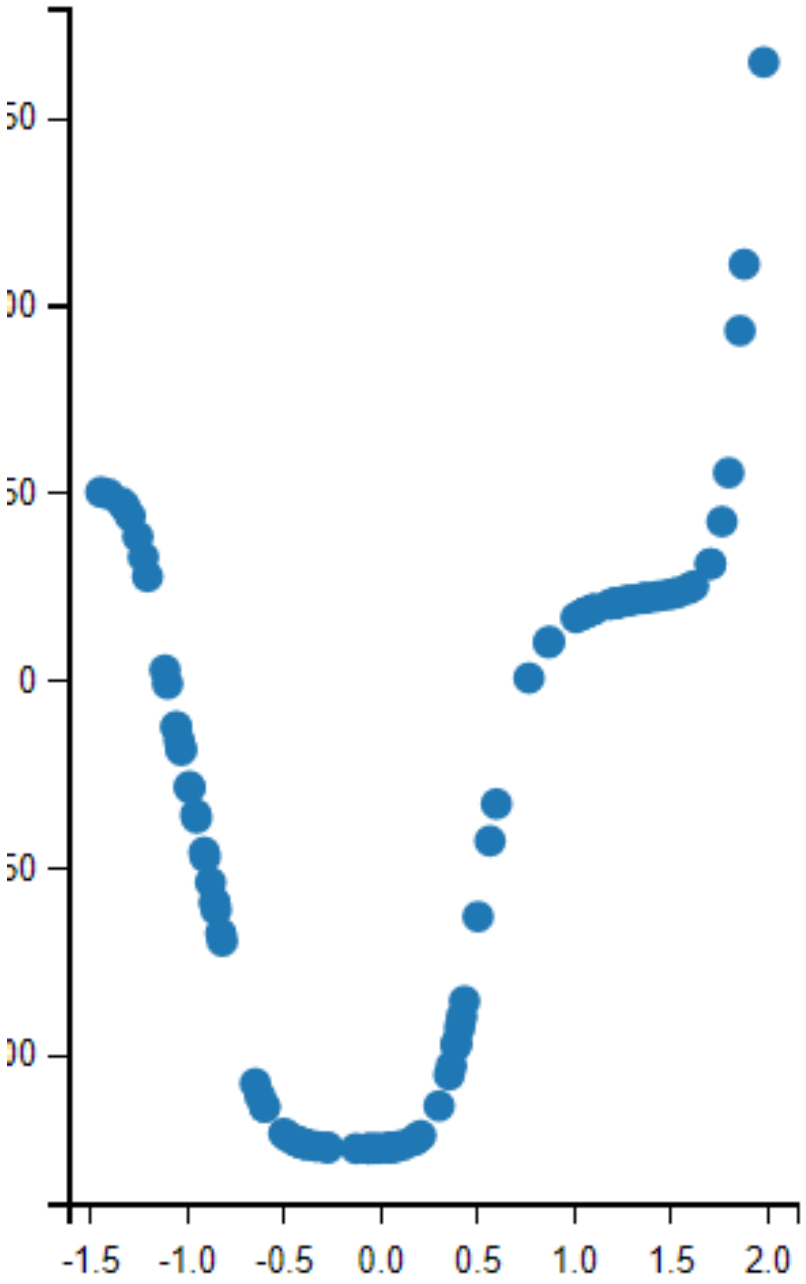


Figure 6: square-simple

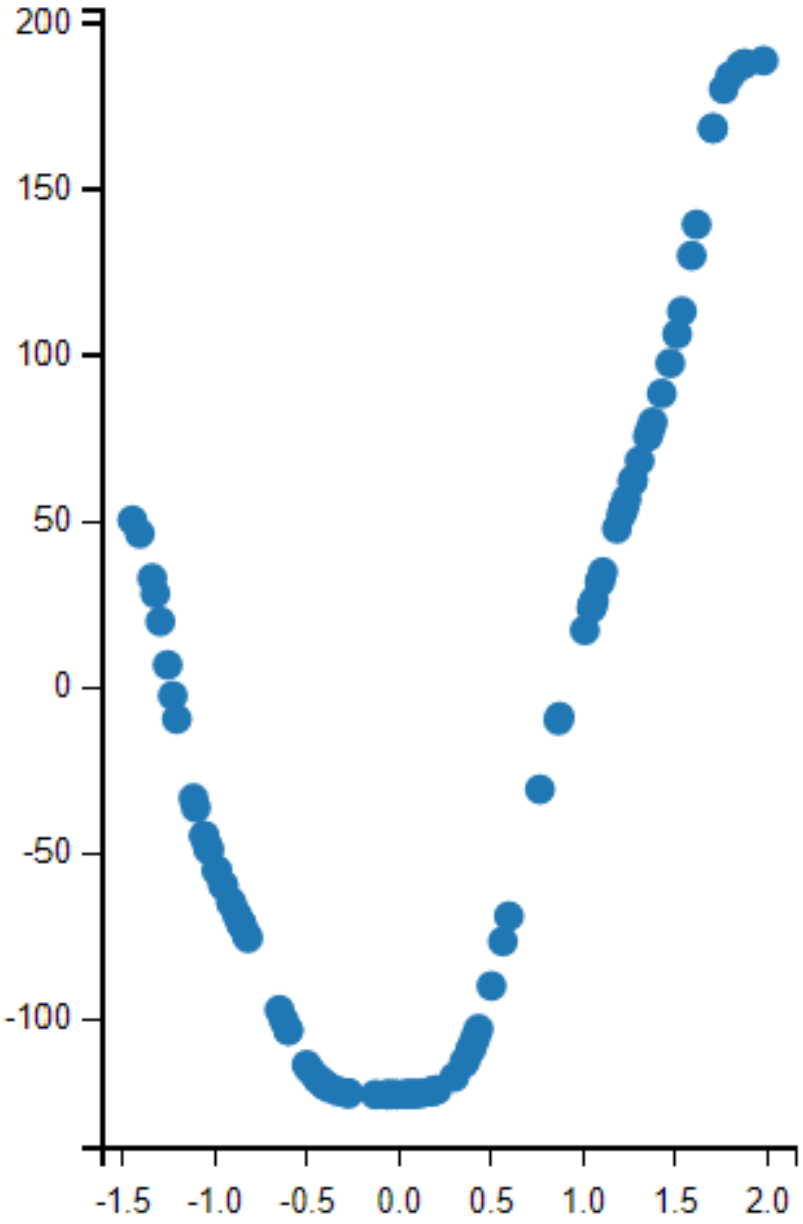


Figure 7: square-simple

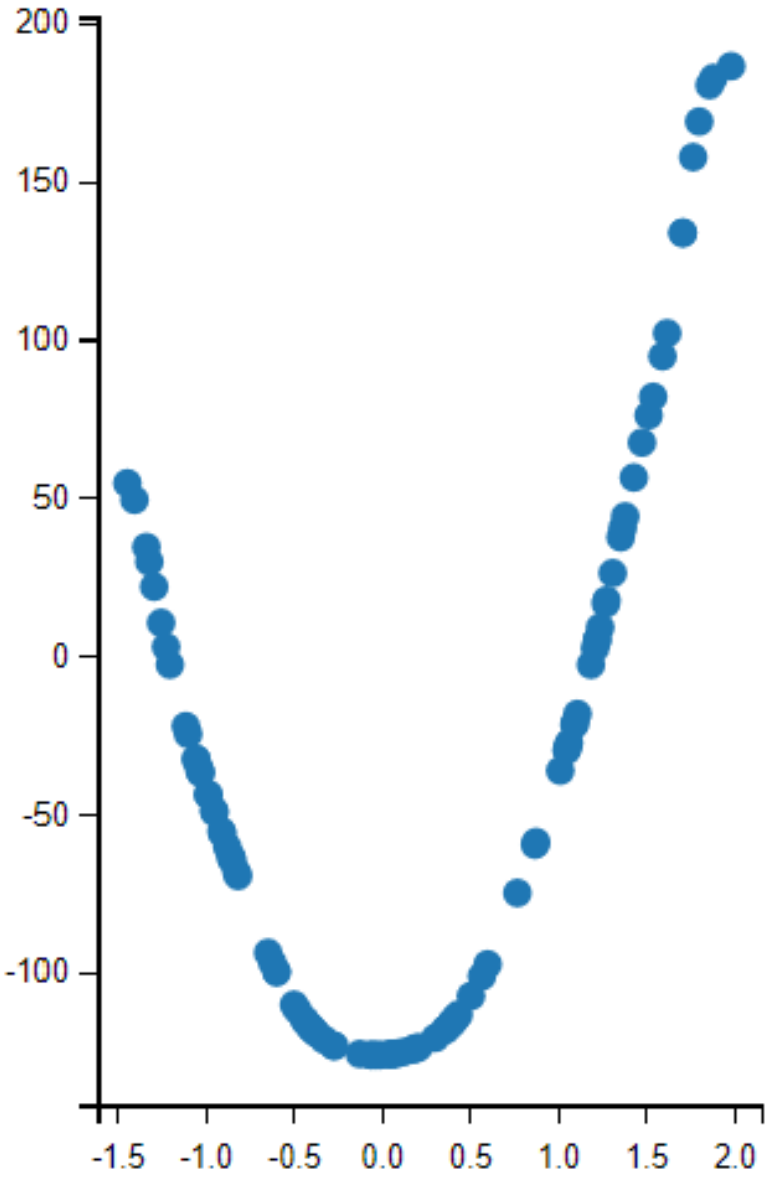


Figure 8: square-simple najlepsze

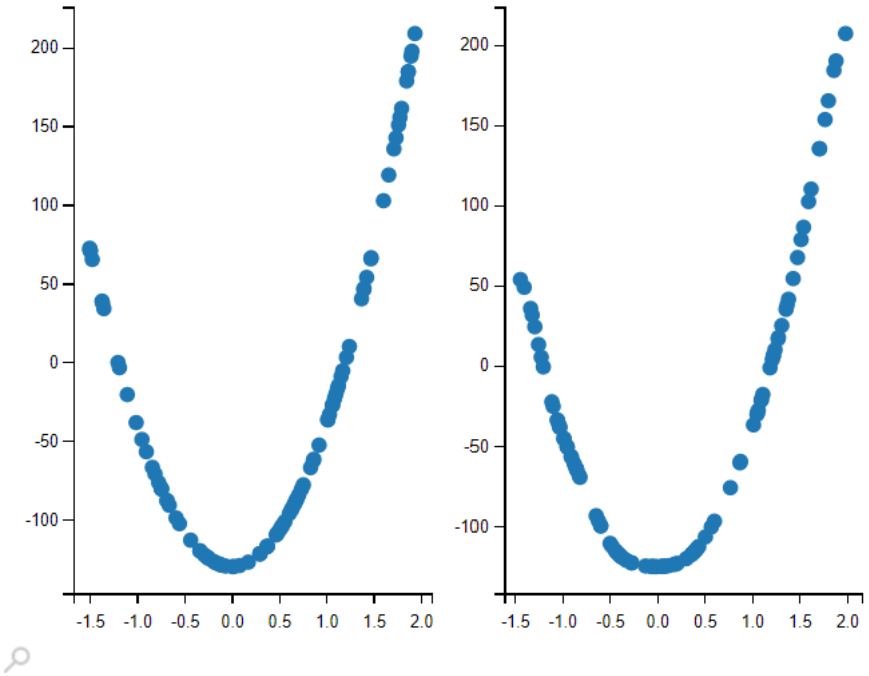


Figure 9: square-simple najlepsze z porównaniem oczekiwanej struktury

1.4 wyniki

O dziwo, większą trudność sprawiło mi znalezienie dobrych wag dla steps-large, niż dla square-simple. Tym więcej warstw i tym więcej neuronów, tym bardziej żmudne było stworzenie udanej strukruty. Używając wyżej opisanego algorytmu, którym się wspomagałem, mniejsza ilość neuronów dawała większe i bardziej widoczne zmiany, co pomagało w odpowiedniej modyfikacji wag. Oto wyniki:

steps-large	**square-simple**	
7.59	8.35	[5]
7.98	8.42	[10]
8.23	8.84	[5,5]

Table 1: MSE

2 NN2 - propagacja wsteczna

2.1 Opis zadania

Zadanie polegało na dodaniu do stworzonej przez nas sieci funkcję propagacji wstecznej wraz z mini batchami. Mieliśmy porównać jak zbiega mse przy różnej ilości mini batchy. Polecenie mówiło również, o wyświetlaniu wag w celu kontrolowania ich zachowania.

2.2 Implementacja

Aby podejść do tego problemu, musiałem wprowadzić kilka zmian w kodzie. Musiałem dodać inicjalizację wag, a nie wstawianie wag jako argumenty. Użyłem do tego inicjalizacji Xaviera:

$$w_{ij} \sim U\left[-\frac{\sqrt{6}}{\sqrt{n_{\text{in}} + n_{\text{out}}}}, \frac{\sqrt{6}}{\sqrt{n_{\text{in}} + n_{\text{out}}}}\right]$$

Gdzie:

- w_{ij} to waga między neuronem wejściowym i a neuronem wyjściowym j,
- $U[a, b]$ oznacza rozkład jednostajny na przedziale od a do b,
- n_{in} to liczba neuronów w poprzedniej warstwie,
- n_{out} to liczba neuronów w następnej warstwie.

Aby patrzeć jak zachowują się wagi podczas funkcji backpropagacji, dodałem funkcję wyświetlającą gradienty wag i biasów co kilka epochs.

Dodałem funkcję backpropagacji, która jako argumenty przyjmuje batch size oraz learning rate.

1. **Propagacja w przód (Forward Propagation):**

   - Na początku, dla danej próbki danych wejściowych X, obliczane są wyniki predykcji $\hat{Y}$ poprzez propagację w przód przez sieć neuronową.
   - Dla każdej warstwy l (łącznie z warstwą wyjściową), wartości wejściowe $Z^{(l)}$ oraz aktywacje $A^{(l)}$ są obliczane według równań:

   $$Z^{(l)} = W^{(l)} A^{(l-1)} + B^{(l)}$$

   $$A^{(l)} = g(Z^{(l)})$$

   gdzie $W^{(l)}$ to macierz wag, $B^{(l)}$ to wektor biasów, $g(\cdot)$ to funkcja aktywacji, a $A^{(0)} = X$.

2. **Obliczanie błędów (Error Computation):**

   - Po przeprowadzeniu propagacji w przód, obliczany jest błąd sieci neuronowej dla danej próbki. Ja zastosowałem funkcję Mean Squared Error (MSE):

   $$J = \frac{1}{m} \sum_{i=1}^{m} (Y_i - \hat{Y}_i)^2$$

   gdzie m to liczba próbek danych, Y to rzeczywiste etykiety, a $\hat{Y}$ to predykcje sieci.

3. **Propagacja wsteczna (Backward Propagation):**

   - Następnie, błąd $\delta^{(L)}$ dla warstwy wyjściowej L jest obliczany jako gradient funkcji kosztu J względem aktywacji warstwy wyjściowej, pomnożony przez pochodną funkcji aktywacji $g'(\cdot)$:

   $$\delta^{(L)} = \nabla_{A^{(L)}} J \odot g'(Z^{(L)})$$

   - Następnie, wykorzystując $\delta^{(L)}$, obliczane są błędy dla poprzednich warstw za pomocą równania:

   $$\delta^{(l)} = ((W^{(l+1)})^T \delta^{(l+1)}) \odot g'(Z^{(l)})$$

4. **Aktualizacja wag i biasów (Weights and Biases Update):**

   - Na koniec, po obliczeniu błędów dla wszystkich warstw, wagi i biasy sieci są aktualizowane za pomocą gradientu funkcji kosztu względem tych parametrów:

   $$W^{(l)} = W^{(l)} - \eta \frac{\partial J}{\partial W^{(l)}}$$

   $$B^{(l)} = B^{(l)} - \eta \frac{\partial J}{\partial B^{(l)}}$$

   gdzie η to współczynnik uczenia, a $\frac{\partial J}{\partial W^{(l)}}$ oraz $\frac{\partial J}{\partial B^{(l)}}$ to gradienty funkcji kosztu względem wag i biasów warstwy l.

Dodałem również mini batche. W moim kodzie, wygląda to tak, że co jedną epoch, wybierany jest mini batch o określonej wcześniej przeze mnie wielkości. Ten mini batch jest akutalnym zbiorem treningowym i to na nim wykonywana jest propagacja wsteczna. W następnej epoch ten zbiór treningowy się zmienia. W ten sposób co kilka iteracji sieć zostanie uczona przez cały początkowy zbiór treningowy. Na wykresach poniżej, będzie można zobaczyć prędkość uczenia w zależności od rozmiaru mini batchy.

Dzięki tak zaimplementowanemu kodowi, jestem w stanie nauczyć sieć wag i biasów potrzebnych do dokładnej predykcji. Teraz żmudna praca dobierania tych wag ręcznie jest zamieniona na uruchomienie algorytmu, który po pewnym czasie daje mi upragnione wyniki.

2.3 Eksperymenty

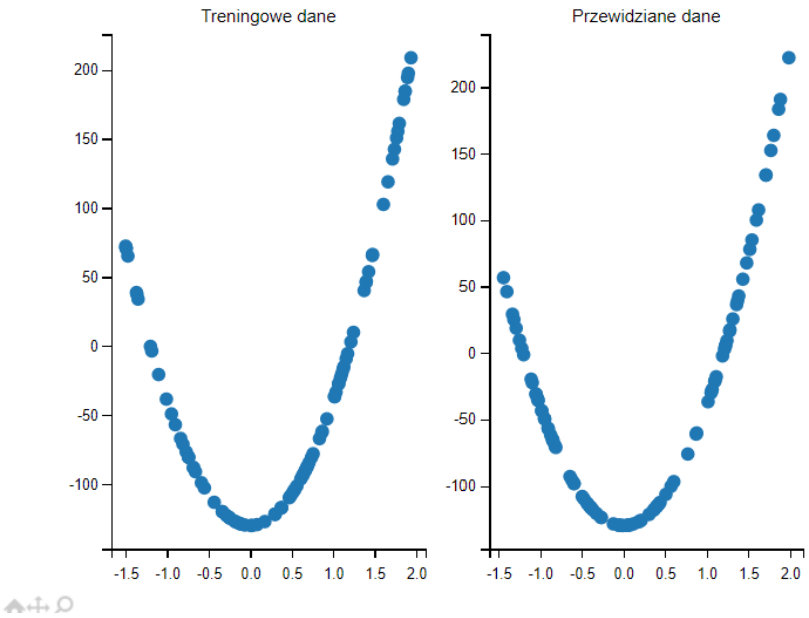
Dla każdego zbioru badałem szybkość zbieżności dla różnych rozmiarów batchy. Oto najlepsze wyniki jakie udało mi się uzyskać.

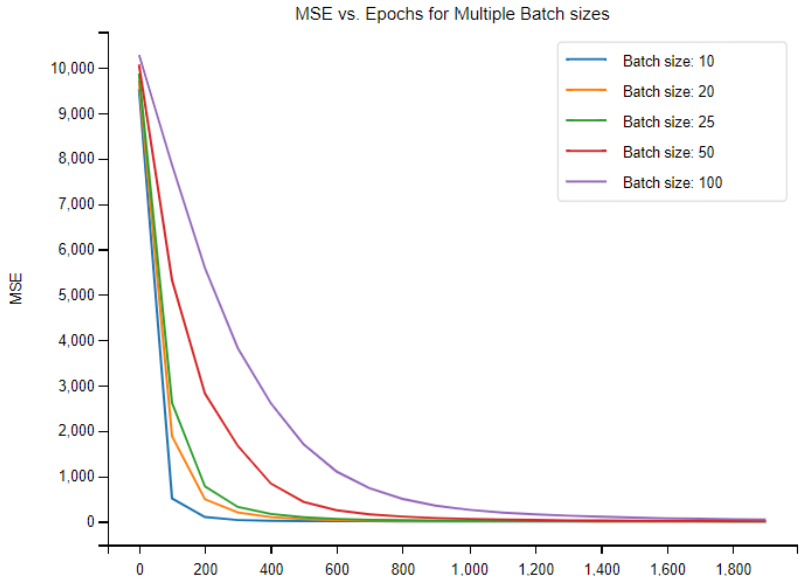
2.3.1 Square-simple

MSE na zbiorze treningowym dla 30000 epochs, learnign rate 0.01 i jednej warstwy ukrytej z 20 neuronami:

Batch size:	**10**	**20**	**25**	**50**	**100**
MSE	0.15	0.28	0.39	0.81	1.10

Table 2: MSE vs. Batch size: square-simple


Treningowe dane
Przewidziane dane
200
150
100
50
0
-50
-100
-1.5
-1.0
-0.5
0.0
0.5
1.0
1.5
2.0

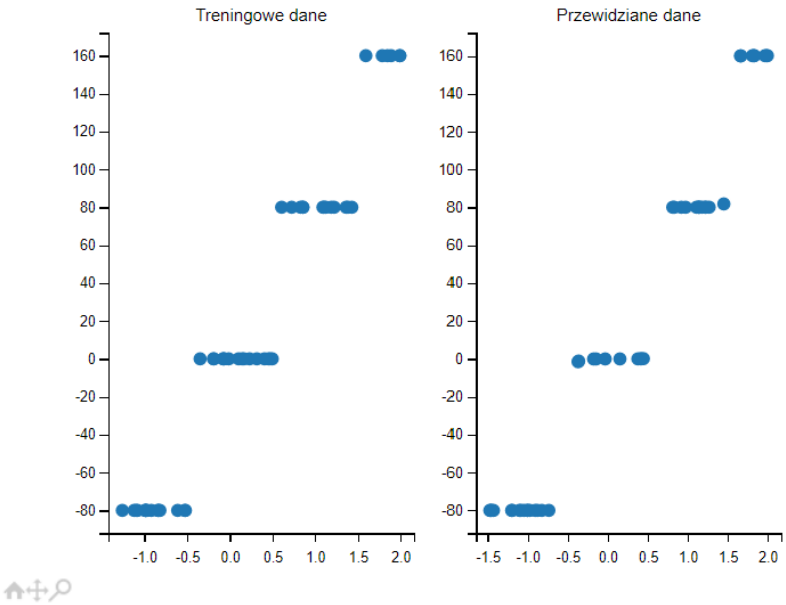

MSE vs. Epochs for Multiple Batch sizes
MSE
Batch size: 10
Batch size: 20
Batch size: 25
Batch size: 50
Batch size: 100
10,000
9,000
8,000
7,000
6,000
5,000
4,000
3,000
2,000
1,000
0
0
200
400
600
800
1.000
1.200
1.400
1.600
1.800

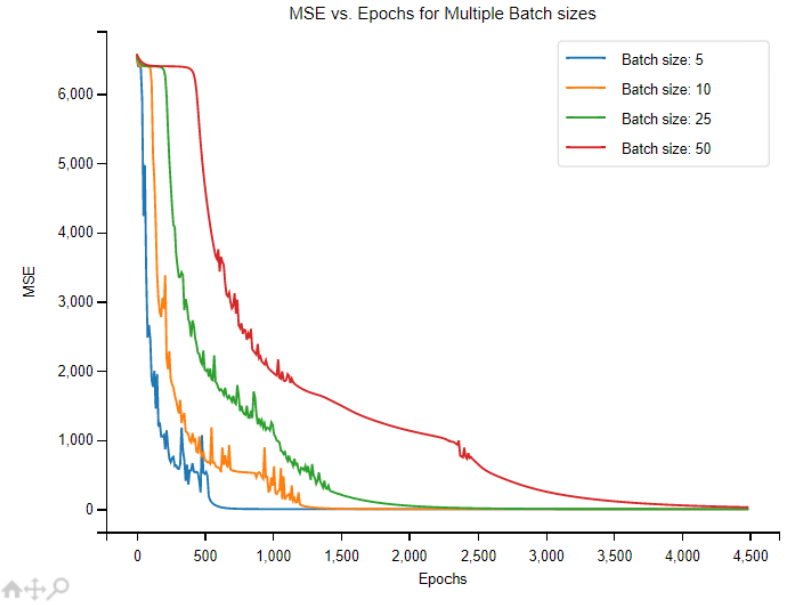
2.3.2 Steps small

MSE na zbiorze treningowym dla 9000 epochs, learning rate 0.023 i trzech warstw ukrytcyh z 5 neuronami:

Batch size:	**5**	**10**	**25**	**50**
MSE	0.04	0.08	0.23	0.91

Table 3: MSE vs. Batch size: steps-small


Treningowe dane
Przewidziane dane
160
140
120
100
80
60
40
20
0
-20
-40
-60
-80
-1.5
-1.0
-0.5
0.0
0.5
1.0
1.5
2.0


MSE vs. Epochs for Multiple Batch sizes
Batch size: 5
Batch size: 10
Batch size: 25
Batch size: 50
MSE
6,000
5,000
4,000
3,000
2,000
1,000
0
0
500
1,000
1,500
2,000
2,500
3,000
3,500
4,000
4,500
Epochs

2.3.3 Multimodal-large

MSE na zbiorze treningowym dla 5000 epochs, learnign rate 0.05 i jednej warstwy ukrytej z 50 neuronami:

Batch size:	**1000**	**2000**	**2500**	**5000**	**1000**
MSE	3	7	11	66	179

Table 4: MSE vs. Batch size: multimodal -large

Zacząłem od pełnego minibatcha, więc 10000 dla tego zbioru. Niestety można zobaczyć, że dla tego zbioru uczącego mse nie chce zejść poniżej wymaganego 40. Jednak gdy mini batch jest ustawiony na mniej niz 2500 sieć uczy się z łatwością.

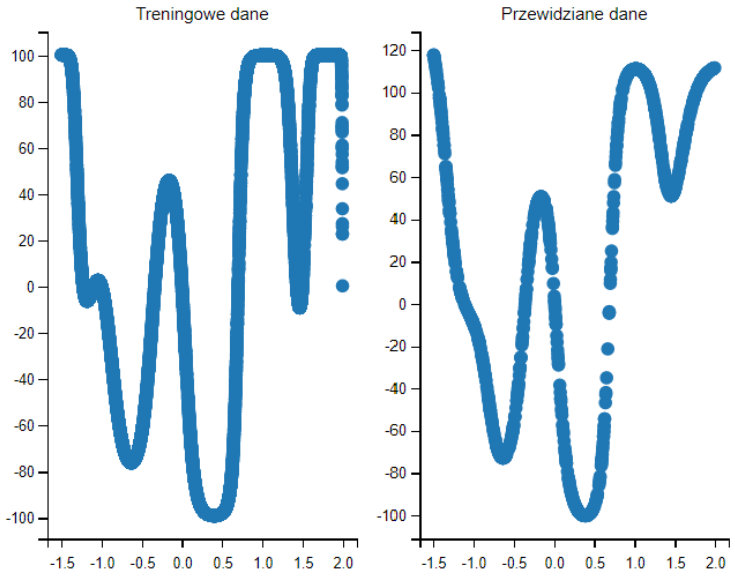


Figure 10: Wyniki dla pełnego batcha

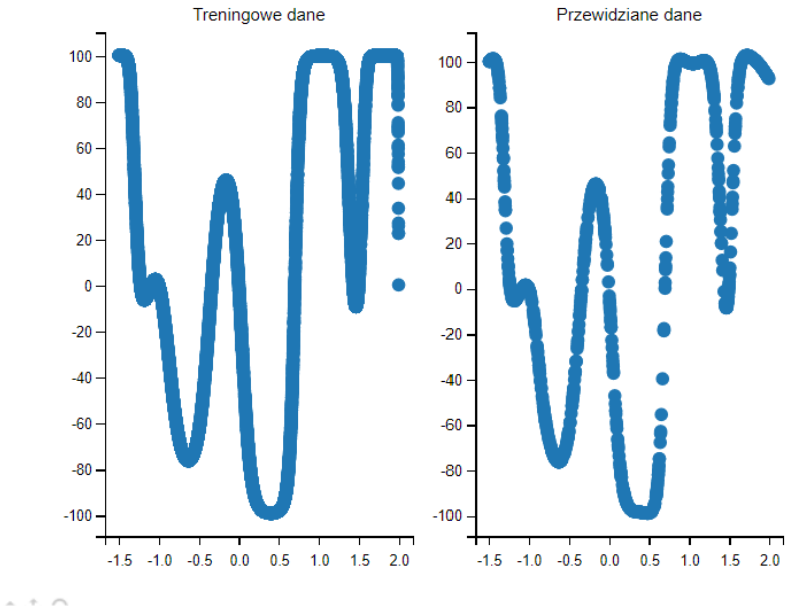
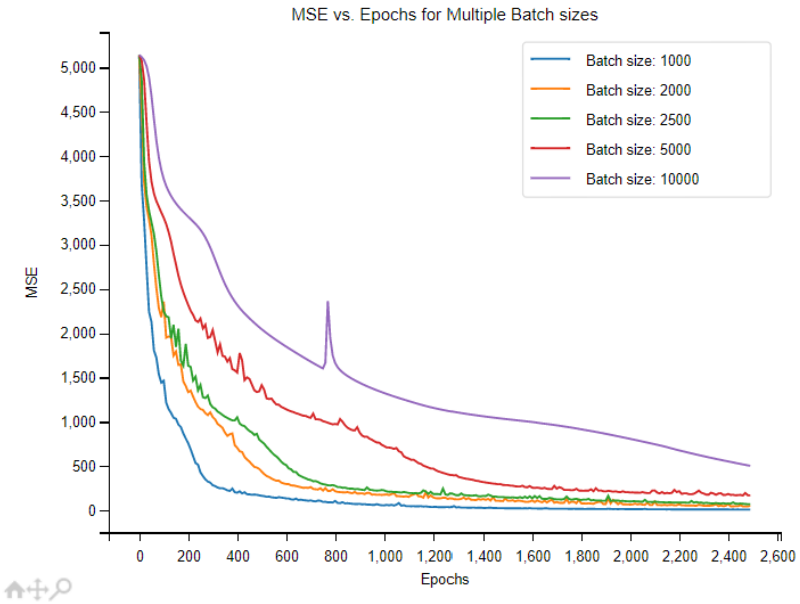


Figure 11: Wyniki dla batcha o rozmiarze 2500


MSE vs. Epochs for Multiple Batch sizes
Batch size: 1000
Batch size: 2000
Batch size: 2500
Batch size: 5000
Batch size: 10000
MSE
5,000
4,500
4,000
3,500
3,000
2,500
2,000
1,500
1,000
500
0
0
200
400
600
800
1,000
1,200
1,400
1,600
1,800
2,000
2,200
2,400
2,600
Epochs

2.4 Wyniki

Jak widać backpropagacja i jest to świetny punkt wyjścia do rozwijania uczenia maszynowego przy pomocji sieci neuronowych. W następnych framgentach spawrozdania, można zobaczyć jak ta sieć stopniowo rozrasta się o kolejne funkcjonalności. Backpropagacja działa dla prostych funkcji jak i tych bardziej skomplikowanych.

3 NN3 - moment i regularyzacja gradientu

3.1 Opis zadania

Do istniejącego kodu mieliśmy zaimplementować usprawnienie uczenia wykorzystując metodę momentu oraz normalizację gradinetu RMSProp

3.2 Implementacja

Metoda momentu to technika optymalizacji używana w algorytmach uczenia maszynowego, takich jak algorytm propagacji wstecznej. Jej celem jest przyspieszenie procesu uczenia poprzez uwzględnienie historii poprzednich aktualizacji wag.

W przypadku algorytmu propagacji wstecznej, metoda momentu modyfikuje standardową aktualizację wag poprzez dodanie współczynnika momentum, który odpowiada za akumulację poprzednich gradientów. Dzięki temu, wagi są aktualizowane nie tylko na podstawie bieżącego gradientu, ale również na podstawie kierunku i wielkości poprzednich zmian.

Poniżej przedstawiam główne kroki działania metody momentu w kontekście algorytmu propagacji wstecznej:

1. Inicjalizacja prędkości wag:

$$\text{velocity} = 0$$

2. Dla każdej epoch i każdego mini batcha: - Obliczamy gradient funkcji kosztu względem wag. - Aktualizujemy prędkość wag, uwzględniając poprzednie zmiany:

$$\text{velocity} = \text{momentum} \times \text{velocity} - \text{learning\_rate} \times \text{gradient}$$

- Aktualizujemy wagi:

$$\text{wagi} = \text{wagi} + \text{velocity}$$

W powyższych równaniach: - momentum to współczynnik momentum, zwykle wybierany z przedziału $[0, 1]$.(Ja używam 0.9) - learning_rate to współczynnik uczenia, kontrolujący szybkość uczenia. - velocity to prędkość wag, czyli akumulowany gradient. - gradient to gradient funkcji kosztu względem wag.

3.3 Eksperymenty

3.3.1 square-large

MSE na zbiorze treningowym dla 5000 epochs, learnign rate 0.01 i jednej warstwy ukrytej z 50 neuronami:

Moment:	**Tak**	**Nie**
MSE	0.015	0.47

Table 5: MSE vs. moment: square -large

Dla zbioru testowego dostaję bardzo duże wyniki. Moim zdaniem, zbiór testowy jest źle dobrany, ponieważ zawiera inny zakres wartości x.Oto wykres z tym zbiorem testowym Jeśli odetnę wartości do takich, które znajdują się tylo w zbiorze treningowym dostanę bardzo dobre wyniki. Widać to na nizszych wykresach

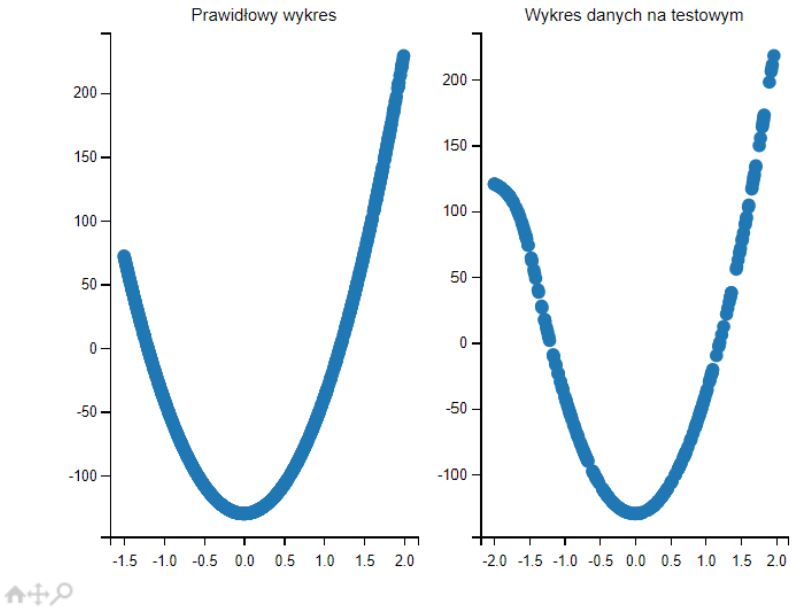


Figure 12: Wyniki na pełnym zbiorze testowym

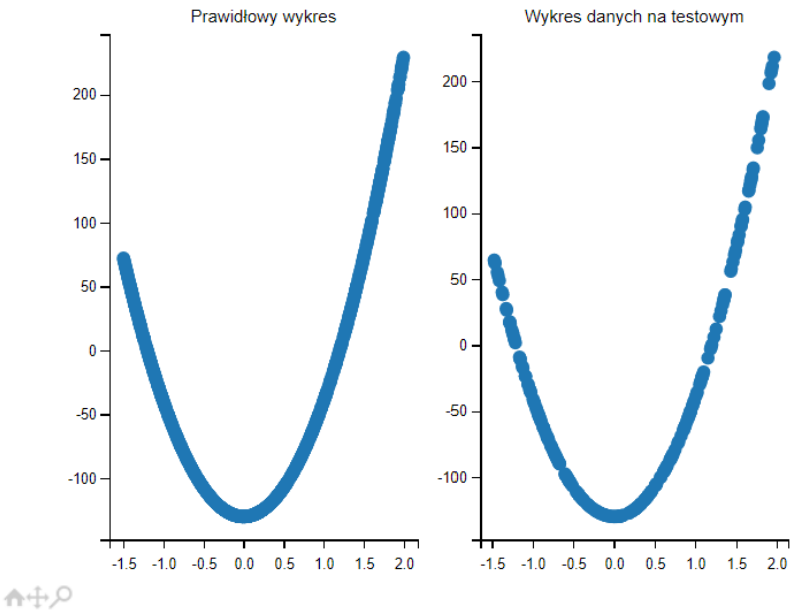
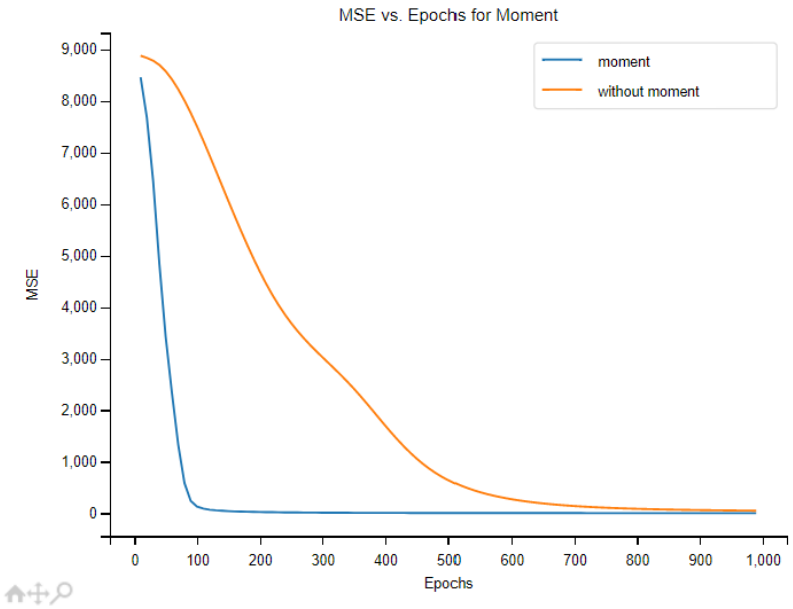


Figure 13: Wyniki na obciętym zbiorze testowym


MSE vs. Epochs for Moment
moment
without moment
MSE
9,000
8,000
7,000
6,000
5,000
4,000
3,000
2,000
1,000
0
0
100
200
300
400
500
600
700
800
900
1,000
Epochs

3.3.2 steps-large

MSE na zbiorze treningowym dla 10000 epochs, learnign rate 0.01 i dwóch warstw ukrytych z 5 neuronami:

Moment:	**Tak**	**Nie**
MSE	2.37	6.23

Table 6: MSE vs. moment: steps -large

Bez momentu, trudno w małej liczbie epoch dostać dobry wynik. Widać to na gorszym wykresie: A tutaj już lepszy wykres z użyciem momentu

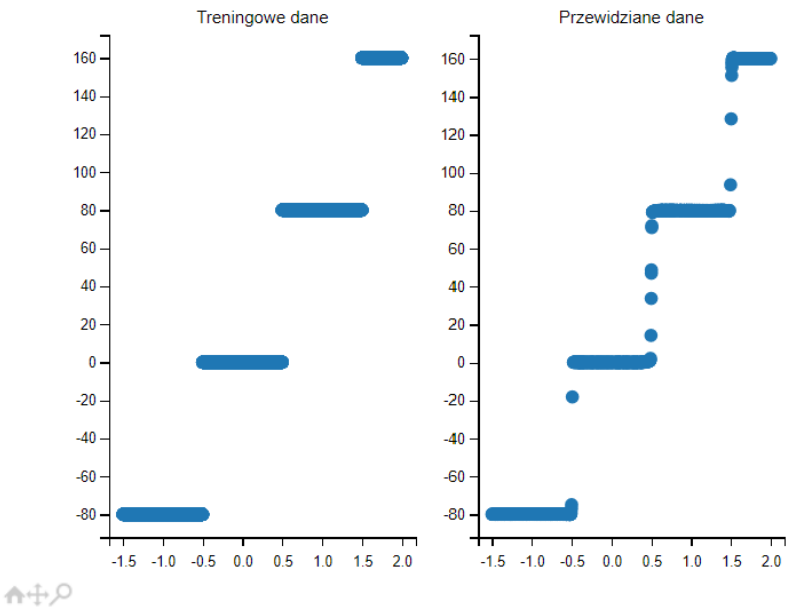


Figure 14: Gorsze wyniki bez momentu

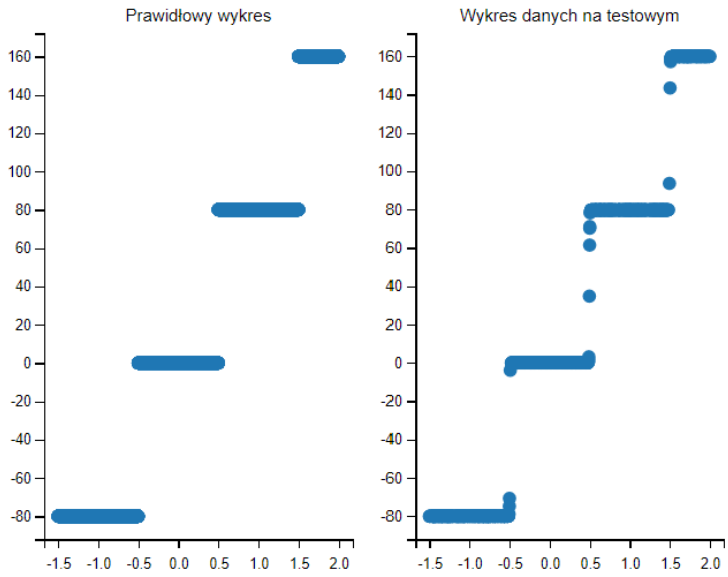
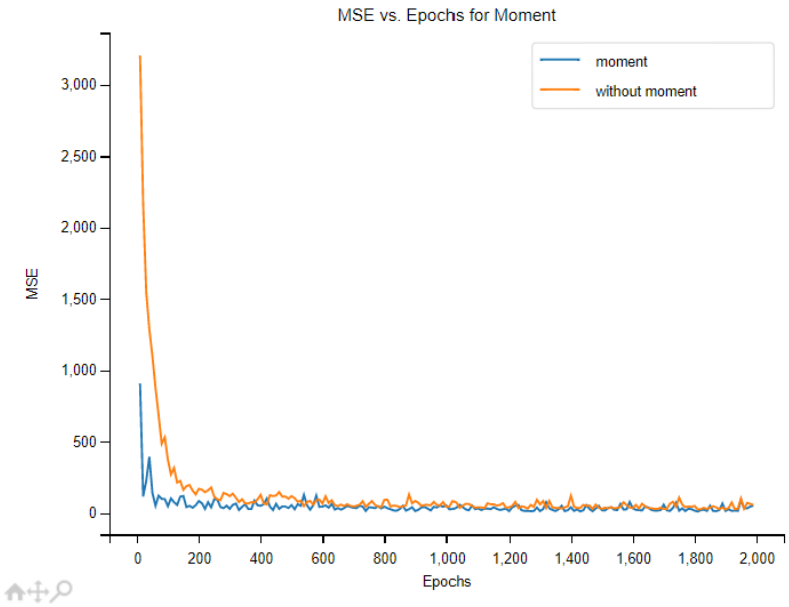


Figure 15: lepsze wyniki z momentem

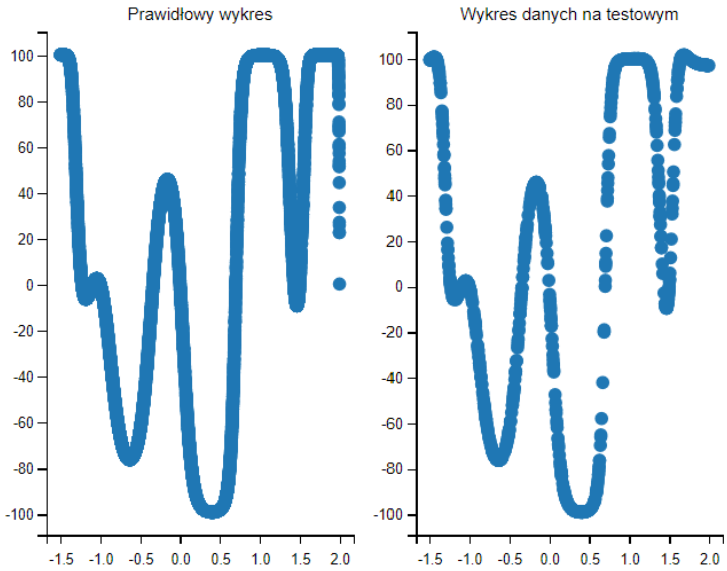

MSE vs. Epochs for Moment
moment
without moment
MSE
3,000
2,500
2,000
1,500
1,000
500
0
0
200
400
600
800
1,000
1,200
1,400
1,600
1,800
2,000
Epochs

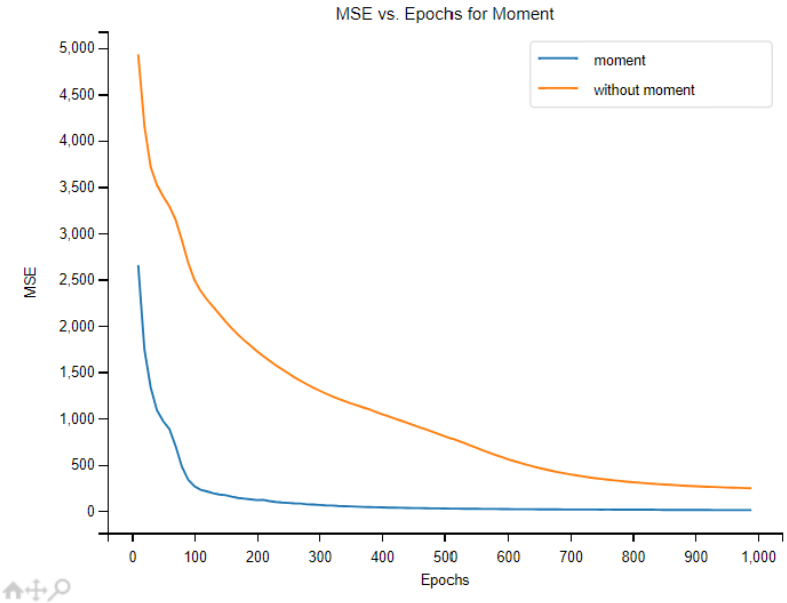
3.3.3 multimodal-large

MSE na zbiorze treningowym dla 10000 epochs, learnign rate 0.01 i jednej warstyw ukrytej z 50 neuronami

Moment:	**Tak**	**Nie**
MSE	1.69	4,29

Table 7: MSE vs. moment: multimodal -large


Prawidłowy wykres
Wykres danych na testowym
100
80
60
40
20
0
-20
-40
-60
-80
-100
-1.5
-1.0
-0.5
0.0
0.5
1.0
1.5
2.0


MSE vs. Epochs for Moment
moment
without moment
MSE
5,000
4,500
4,000
3,500
3,000
2,500
2,000
1,500
1,000
500
0
0
100
200
300
400
500
600
700
800
900
1,000
Epochs

3.4 Wyniki

Jak widać moment bardzo przyśpiesz uczenie. Widać to na wszystkich zbiorach, na których go testowałem. Dzięki temu praca w daleszych zadaniach była dużo łatwiejsza i szybsza.

4 NN4 - softmax

4.1 Opis zadania

Zadaniem było zaaplikowanie funkcji softmax do uczenia w problemach klasyfikacji. Mieliśmy porównać prędkość zbiegania dla softmaxa i zwykłej funkcji aktywacji.

4.2 Implementacja

Musiałem zmienić dotychczasową sieć. na wejściu po raz pierwszy użyte były dwie zmienne. Zasada działania funkcji feedforward była podobna z wyjątkiem tego, że na wyjściu zamiast funkcji liniowej była użyta funkcja softmax. Również tym razem na wyjściu zamiast wyników otrzymywaliśmy wektor prawdopodobieństw, który był używany do przydzielenia wyniku do konkretnej klasy. Jako argument do budowy sieci musiałem dodać liczbę klas. Dzięki temu mogłem dostosować rozmiar wektora wyjściowego dla każdego zbioru. Dodatkwo w preprocesingu danych musiałem zadbać o to, aby w zbiorach z wartościami boolean użyć one-hot-encoding.

Pamiętałem również o odpowiednim zaimplementowaniu pochodnej ostatniej funkcji - czyli softmaxa. Softmax i jego pochodna wyglądały tak:

Funkcja softmax dla wektora x o długości n:

$$\text{softmax}(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}$$

Gdzie:

- x_i oznacza i-ty element wejściowego wektora x,
- e oznacza podstawę logarytmu naturalnego (liczbę Eulera).

Pochodna funkcji softmax (względem wejścia x_i):

$$\frac{\partial \text{softmax}(x_i)}{\partial x_i} = \text{softmax}(x_i) \cdot (1 - \text{softmax}(x_i))$$

$$\frac{\partial \text{softmax}(x_i)}{\partial x_j} = \begin{cases} \text{softmax}(x_i) \cdot (1 - \text{softmax}(x_i)), & \text{jeśli } i = j \\ -\text{softmax}(x_i) \cdot \text{softmax}(x_j), & \text{jeśli } i \neq j \end{cases}$$

Gdzie:

- softmax(x_i) oznacza wartość funkcji softmax dla i-tego elementu wejściowego x,
- i i j są indeksami elementów wejściowych x.

Dodatkowo użyłem innej funkcji straty niż MSE: Funkcja obliczająca stratę (*loss*) w zadaniach klasyfikacji wieloklasowej może być zapisana jako:

$$\text{loss}(\text{real\_y}, \text{predicted\_y}) = -\frac{1}{m}\sum_{i=1}^{m}\log(p_{i,y_i})$$

Gdzie:

- real_y to rzeczywiste etykiety klas (wektor kolumnowy),
- predicted_y to przewidywane etykiety klas (wektor kolumnowy),
- m to liczba próbek,
- p_{i,y_i} to prawdopodobieństwo przypisane do prawidłowej klasy dla i-tej próbki.

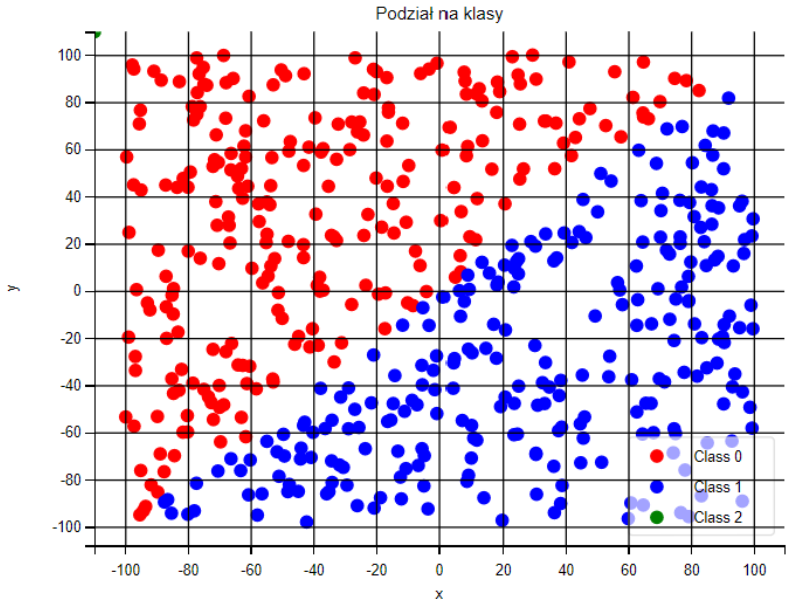
W tej funkcji log oznacza funkcję logarytmiczną naturalną (o podstawie e). Funkcja ta mierzy, jak dobrze przewidywane etykiety klasy odpowiadają rzeczywistym etykietom klas. Im niższa wartość straty, tym lepsze dopasowanie modelu do danych rzeczywistych.

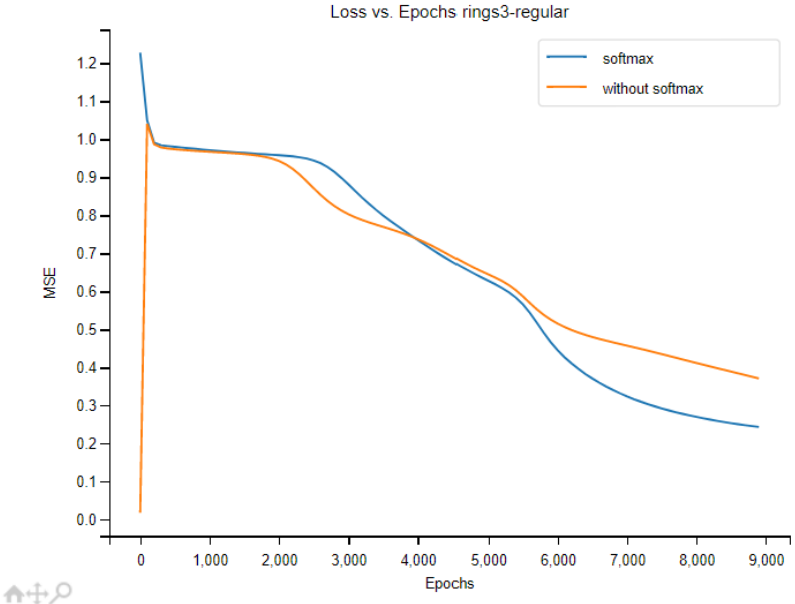
4.3 Eksperymenty

4.3.1 rings3-regular

softmax:	**Tak**	**Nie**
MSE	0.95	0.94

Table 8: F-measur: rings3-regular

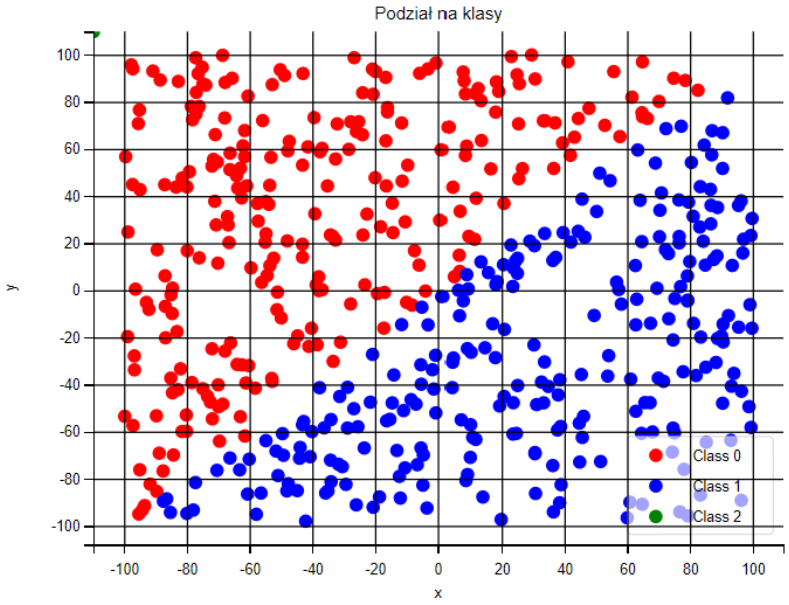

Podział na klasy
y
x
Class 0
Class 1
Class 2

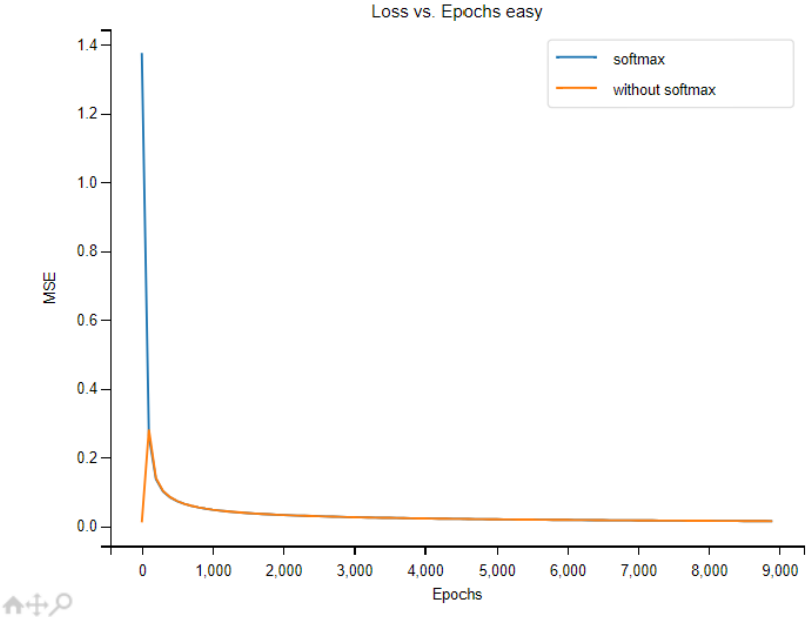

Loss vs. Epochs rings3-regular
softmax
without softmax
MSE
Epochs

4.3.2 easy

softmax:	**Tak**	**Nie**
MSE	0.998	0.998

Table 9: F-measur: easy

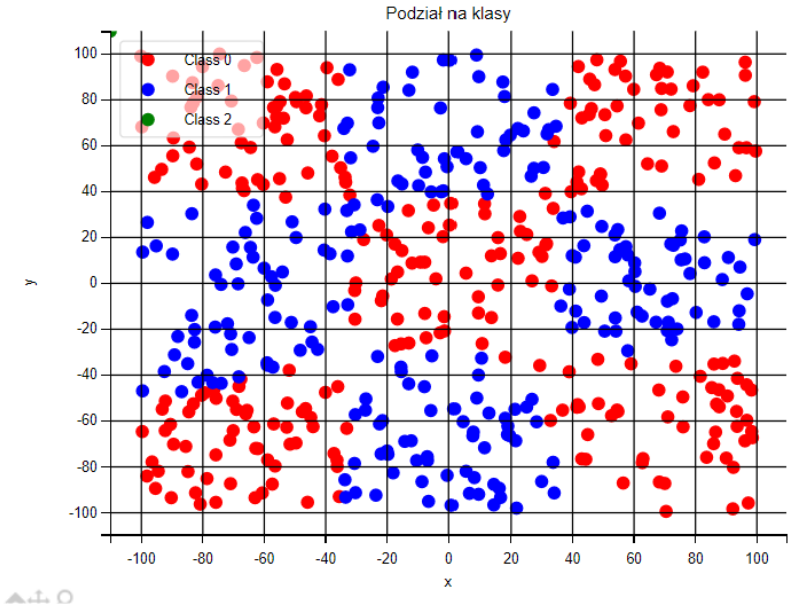

Podział na klasy
y
x
Class 0
Class 1
Class 2

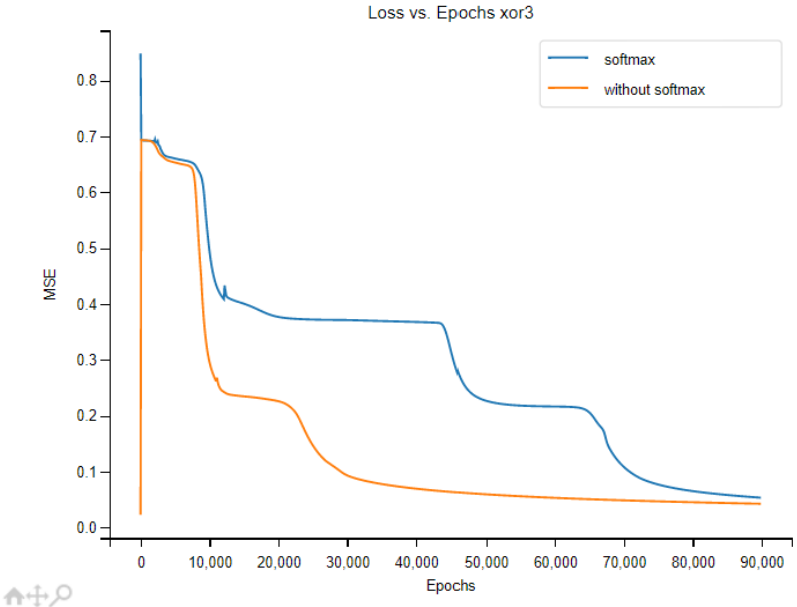

Loss vs. Epochs easy
softmax
without softmax
MSE
Epochs

4.3.3 xor3

softmax:	**Tak**	**Nie**
MSE	0.944	0.976

Table 10: F-measur: xor3


Podział na klasy
Class 0
Class 1
Class 2
100
80
60
40
20
0
-20
-40
-60
-80
-100
y
-100
-80
-60
-40
-20
0
20
40
60
80
100
x


Loss vs. Epochs xor3
softmax
without softmax
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0.0
MSE
0
10,000
20,000
30,000
40,000
50,000
60,000
70,000
80,000
90,000
Epochs

4.4 Wyniki

Zazwyczaj używając softmaxa loss zbiega szybciej. Jednak możemy zauważyć, że dla xor3 funkcja softmax nie była w stanie przekroczyć wymaganego progu Fmeasure. Jednak w tym przypadku standradowa funkcja aktywacji zbiegała szybciej. Można na wykresach zobaczyć, że klasa są odpowiednio przydzielane. Uważam więc zadanie klasyfikacji za udane.

5 NN5 - funkcje aktywacji

5.1 Opis zadania

Zadaniem było napisanie różnych funkcji aktywacji i użycie ich do badania zbiorów. Były to:

- Sigmoid
- Liniowa
- Tangens hiperboliczny (tanh)
- ReLU (Rectified Linear Unit)

5.2 Implementacja

W implementacji musialem zmienić pochodną w zależności od używanej funkcji

1. Sigmoid:

$$f(x) = \frac{1}{1 + e^{-x}}$$
$$f'(x) = f(x) \cdot (1 - f(x))$$

2. Liniowa:

$$f(x) = x$$
$$f'(x) = 1$$

3. Tangens hiperboliczny (tanh):

$$f(x) = \tanh(x)$$
$$f'(x) = 1 - \tanh^2(x)$$

4. ReLU (Rectified Linear Unit):

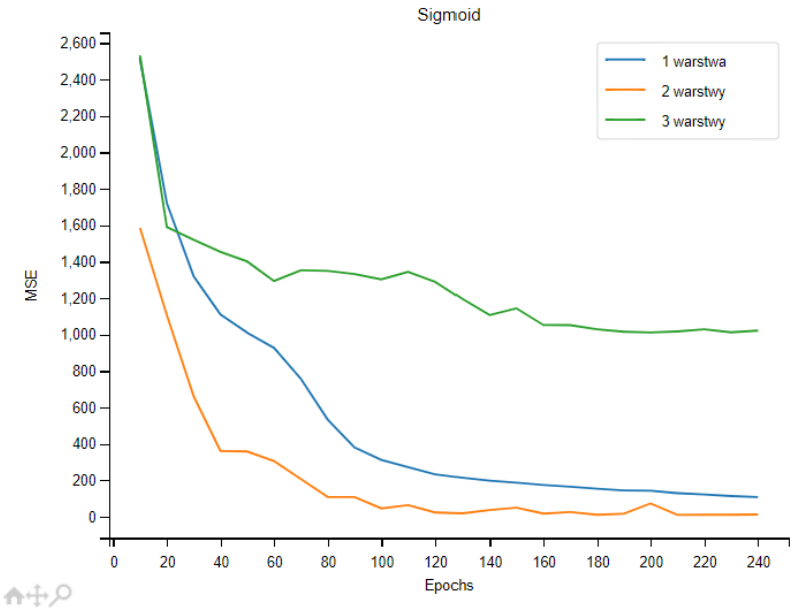
$$f(x) = \max(0, x)$$
$$f'(x) = \begin{cases} 1, & \text{dla } x > 0 \\ 0, & \text{dla } x \leq 0 \end{cases}$$

5.3 Eksperymenty

5.3.1 Multimodal-large

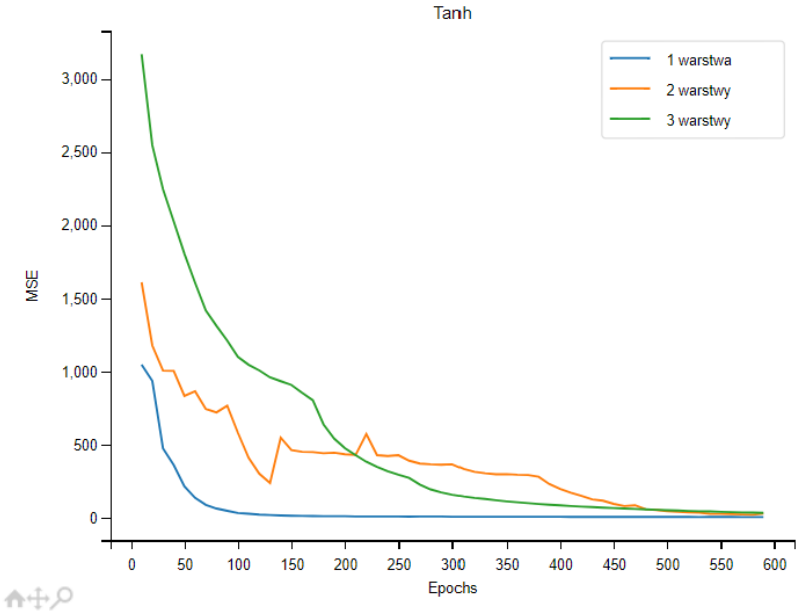
Warstwy	**learning rate**	**MSE**
[50]	0.01	1.95
[25,25]	0.01	1.6
[20,20,20]	0.01	4.14

Table 11: Sigmoid


Sigmoid
1 warstwa
2 warstwy
3 warstwy
MSE
2,600
2,400
2,200
2,000
1,800
1,600
1,400
1,200
1,000
800
600
400
200
0
0
20
40
60
80
100
120
140
160
180
200
220
240
Epochs

Warstwy	**learning rate**	**MSE**
[50]	0.01	2.14
[25,25]	0.001	2.82
[20,20,20]	0.0001	4.14

Table 12: Tanh

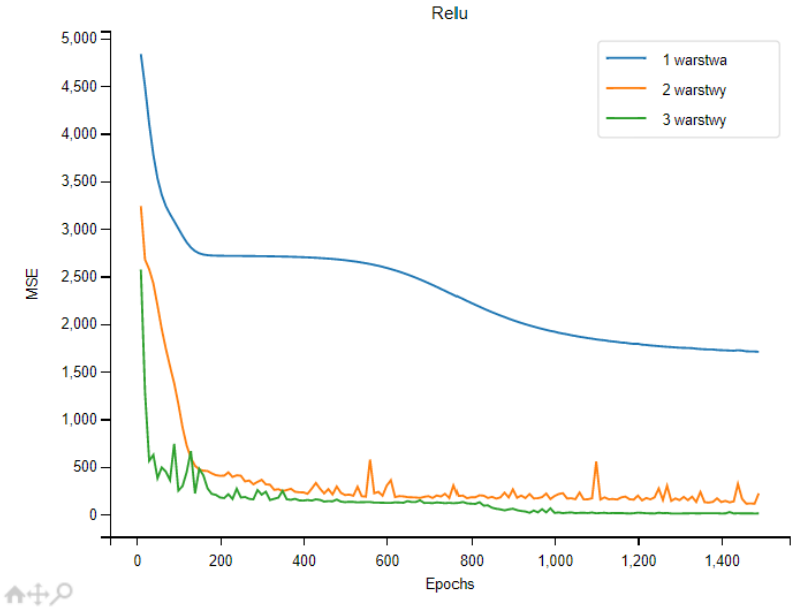

Tanh
1 warstwa
2 warstwy
3 warstwy
MSE
3,000
2,500
2,000
1,500
1,000
500
0
0
50
100
150
200
250
300
350
400
450
500
550
600
Epochs

Warstwy	**learning rate**	**MSE**
[50]	0.01	4434
[25,25]	0.001	4434
[20,20,20]	0.0001	4433

Table 13: Linear

Warstwy	**learning rate**	**MSE**
[50]	0.01	1689
[25,25]	0.001	54
[20,20,20]	0.0001	7.78

Table 14: Relu


Relu
1 warstwa
2 warstwy
3 warstwy
MSE
5,000
4,500
4,000
3,500
3,000
2,500
2,000
1,500
1,000
500
0
0
200
400
600
800
1,000
1,200
1,400
Epochs

5.4 Wyniki

Chciałbym w wynikach omówić kilka rzeczy. Po pierwsze zauważymy, że dla sigmidy oraz tanh wyniki msa są bardzo niskie i dość sprawnie zbiegają, widać to na poniższym wykresie, który praktycznie się nie różni dla simgoidy i tangensa hiperbolicznego: (w tej kolejności)
Widać, że jest wiele funkcji aktywacji, które w zależności od problemu i architekruty sieci, mogą dać różne rezultaty

Spójrzmy jednak teraz na funkcję liniową. Powinna ona działać w przypadku przewidywania funkcji liniowych. i faktycznie dla wielomianu nie działa:

Lecz najbardziej specyficznie zachowuje się funkcja Relu. Tym więcej warstw dam, tym lepsze wyniki uzyskam. Jednak dla małej ilości warstw przyjmuje ona taką postać. Stopniowo na następnych wykresach widać poprawę dla większej ilosci warstw.

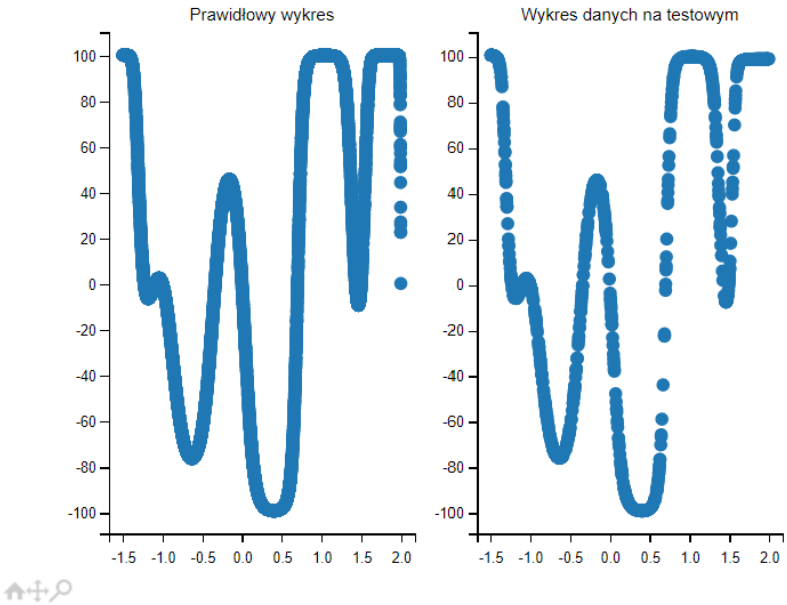


Figure 16: Wynik dla sigmoidy

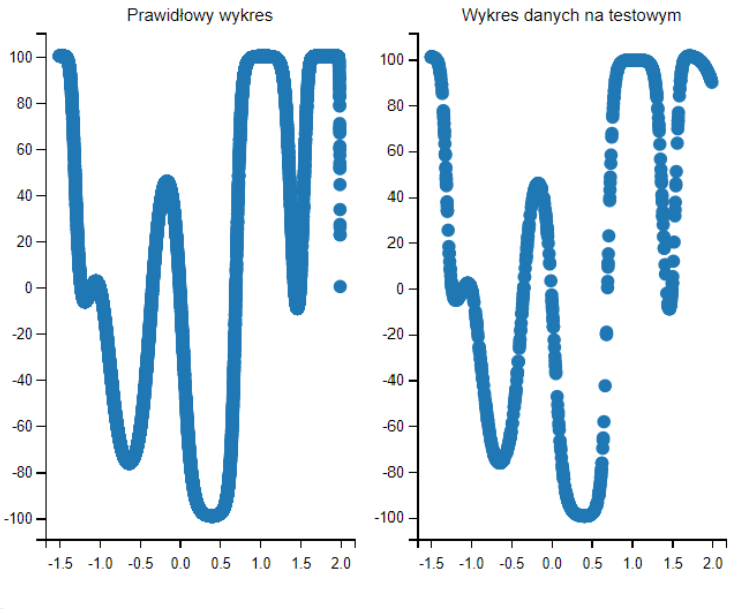


Figure 17: wynik dla tangensa hiperbolicznego

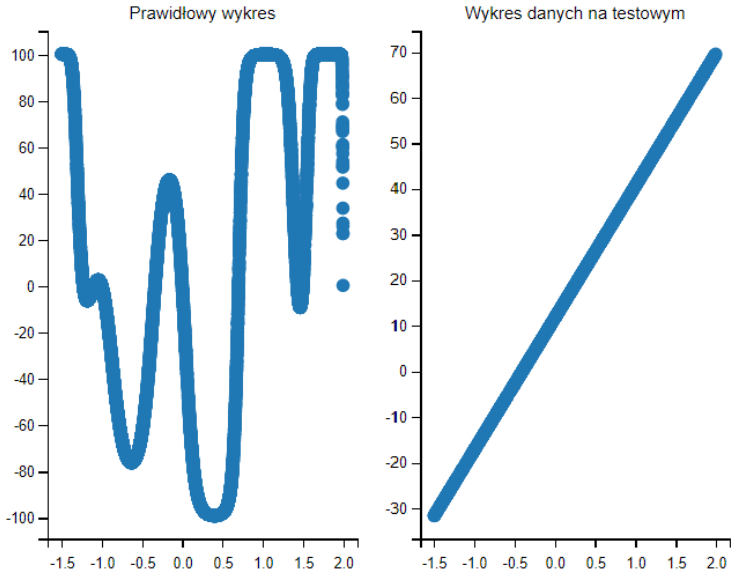


Figure 18: Spójrzmy jednak teraz na funkcję liniową. Powinna ona działać w przypadku przewidywania funkcji liniowych. i faktycznie dla wielomianu nie działa:

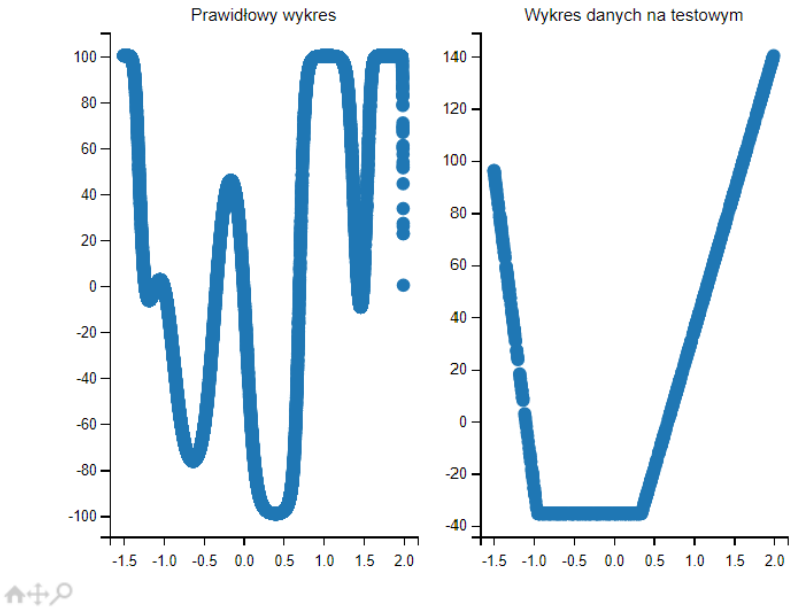


Figure 19: funkcja relu dla 1 warstwy

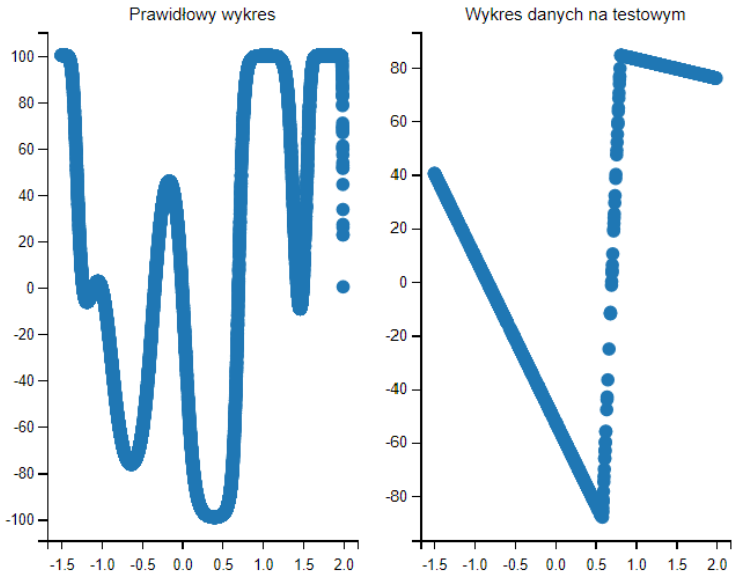


Figure 20: funkcja relu dla 2 warstw

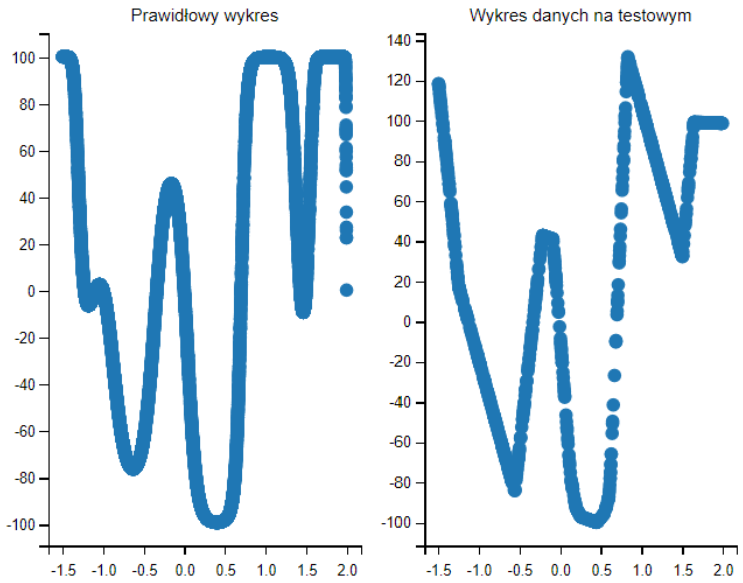


Figure 21: funkcja relu dla 3 warstw

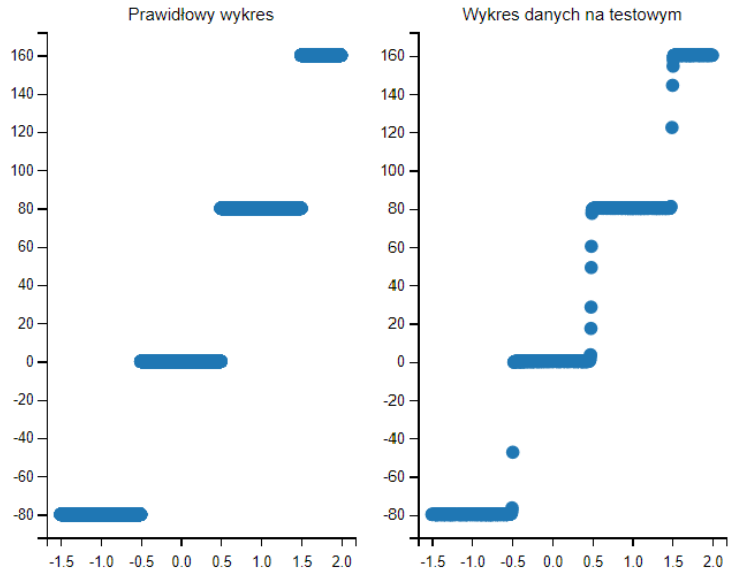


Figure 22: Najlepszymi zbiorami treningowymi jest sigmoida z kolejno warstwą [50] oraz [25,25]. Dla nich dostajemy kolejno niestety niskie wyniki na steps-large 16 mse dla jednej warstwy oraz 17 dla dwóch warstw - wykres dla jednej warswty

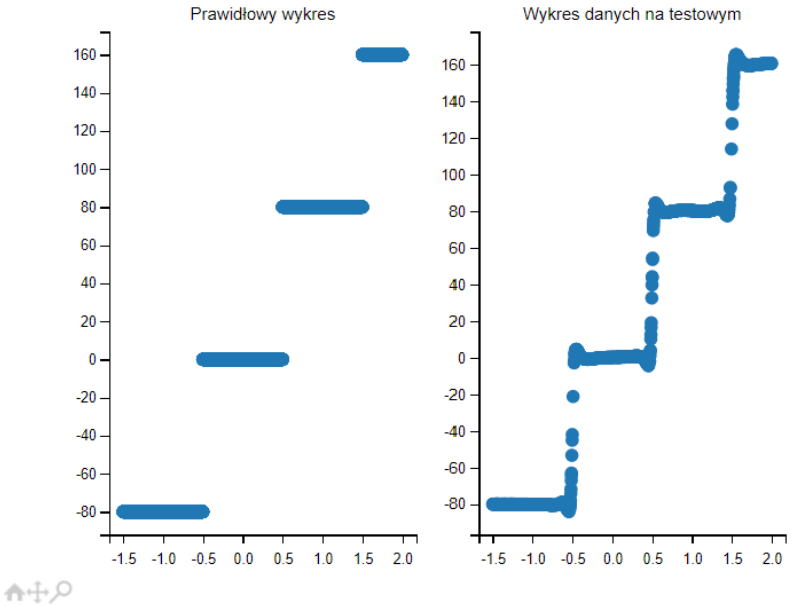


Figure 23: Najlepszymi zbiorami treningowymi jest sigmoida z kolejno warstwą [50] oraz [25,25]. Dla nich dostajemy kolejno niestety niskie wyniki na steps-large 16 mse dla jednej warstwy oraz 17 dla dwóch warstw - wykres dla dwóch warstw

5.5 Podsumowanie

Pracując nad tym wszystkim jestem w stanie stwierdzić, że sieć można nieustannie rozwijać w celu poprawy jej wydajności. Wszyskite wymagania mse zostały spełnione co pokazuje, że moja sieć stopniowo się rozwijała i sprawnie działa. Z czasem zacząłem sobie uświadamiać, jak potężne może to być narzędzie. Mam nadzieję, że pokazałem swój stopniowy rozwój w kwestii sieci neuronowych.