

Projekt 1: klasyfikacja i regresja z użyciem perceptronu

Paweł Lewkowicz, Adam Skaskiewicz

4 marca 2024

Streszczenie

W raporcie "Projekt 1: klasyfikacja i regresja z użyciem perceptronu" przedstawiamy analizę sztucznych sieci neuronowych. Omawiamy czym są sieci neuronowe, czemu zawdzięczają taką nazwę oraz z czego są zbudowane. W prosty sposób staramy się przybliżyć podstawy matematyczne, aby móc dokładnie zrozumieć ich działanie. Następnie omawiamy wyniki naszych prac. Pokazujemy wiele wizualizacji, które ukazują zbiory na jakich działałyśmy oraz porównujemy różne sposoby konfiguracji stworzonej sieci.

Spis treści

1 Wstęp	2
1.1 Opis problemu	2
2 Opis sieci neuronowej	3
2.1 Warstwy sieci neuronowej	3
2.2 Pojedynczy perceptron	4
2.3 Funkcje aktywacji	4
2.4 Funkcje straty	5
2.5 Propagacja wsteczna	5
2.5.1 Obliczanie gradientu w ostatniej warstwie [4]	6
2.5.2 Obliczanie gradientu w warstwach ukrytych [4]	6
2.5.3 Połączenie funkcji aktywacyjnej softmax oraz entropii krzyżowej [5]	6
2.5.4 Współczynnik uczenia (ang. learning rate)	6
2.5.5 Aktualizacja wag	6
2.5.6 Problem zanikającego gradientu [6]	6
2.5.7 Metoda gradientu prostego	7
2.5.8 Heurystyka doboru wag	7
3 Nasza implementacja	7
3.1 Opis modułów sieci	7
3.2 Ziarno generatora liczb losowych	8
3.3 Konfiguracja sieci	8
3.4 Wizualizacja wartości wag w kolejnych iteracjach uczenia	9
3.5 Wizualizacja wartości gradientów w kolejnych iteracjach uczenia	10
4 Zbiory danych	11
4.0.1 Opis zbioru treningowego (uczącego) i testowego	11
4.0.2 Wizualizacja zbiorów	12
5 Eksperymenty	22
5.1 Wpływ funkcji aktywacji na skuteczność działania sieci	22
5.1.1 Porównanie funkcji sigmoidalnej oraz funkcji tangensa hiperbolicznego dla regresji na zbiorze 'activation'	22
5.1.2 Porównanie funkcji sigmoidalnej oraz funkcji tangensa hiperbolicznego dla regresji na zbiorze 'cube'	22

5.1.3	Porównanie funkcji sigmoidalnej oraz funkcji tangensa hiperbolicznego dla klasyfikacji na zbiorze 'simple'	23
5.1.4	Porównanie funkcji sigmoidalnej oraz funkcji tangensa hiperbolicznego dla klasyfikacji na zbiorze 'three gauss'	24
5.2	Wpływ liczby warstw ukrytych w sieci i ich liczności	25
5.3	Wpływ miary błędu na wyjściu sieci na skuteczność uczenia	26
5.4	Eksperyment na zbiorze MNIST	28
6	Podsumowanie	29

1 Wstęp

1.1 Opis problemu

Głównym zadaniem algorytmów uczenia maszynowego jest analiza predykcyjna. Istotną różnicą w porównaniu do wielu innych algorytmów optymalizacyjnych jest możliwość aktualizacji w czasie rzeczywistym po uzyskaniu większej ilości danych. Analiza predykcyjna działa zazwyczaj na statycznym zestawie danych, a aktualizacja jej wyników wymaga odświeżenia.

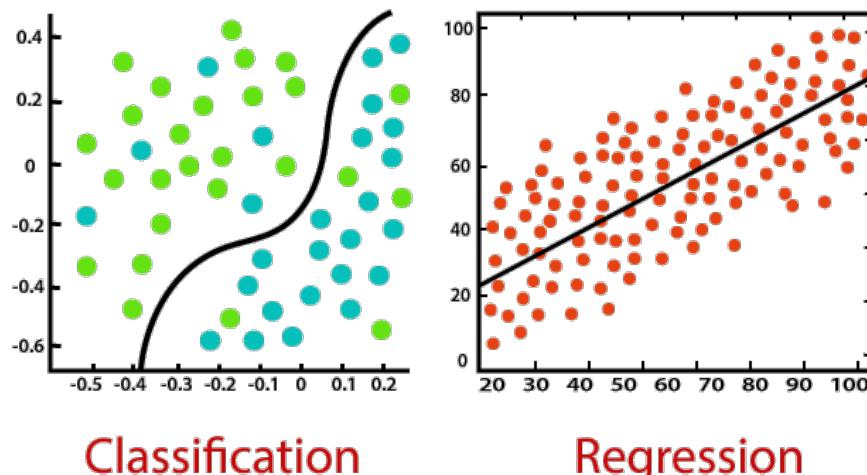
Wyróżniamy cztery rodzaje uczenia maszynowego:

- **Uczenie nadzorowane (Supervised Learning)**
- Uczenie częściowo nadzorowane (Semi – Supervised Learning)
- Uczenie nienadzorowane (Unsupervised Learning)
- Uczenie ze wzmacnieniem (Reinforcement Learning)

Do podstawowych problemów w uczeniu nadzorowanym należą:

- **regresja**
- **klasyfikacja**

Różnica w tych zadaniach polega na tym, że w regresji przypisujemy wartości każdemu obiekowi, natomiast w klasyfikacji przypisujemy obiektom określona klasę (etykietę).



Rysunek 1: Wizualizacja problemu klasyfikacji oraz regresji

Źródło: www.javatpoint.com/regression-vs-classification-in-machine-learning

Na rysunku 1 możemy zobaczyć linię podziału dla klasyfikacji oraz linię najlepszego dopasowania dla regresji. Zadaniem algorytmu regresji jest znalezienie funkcji mapującej, która mapuje zmienną wejściową (x) na ciągłą zmienną wyjściową (y), a w przypadku klasyfikacji na dyskretną zmienną wyjściową (y).

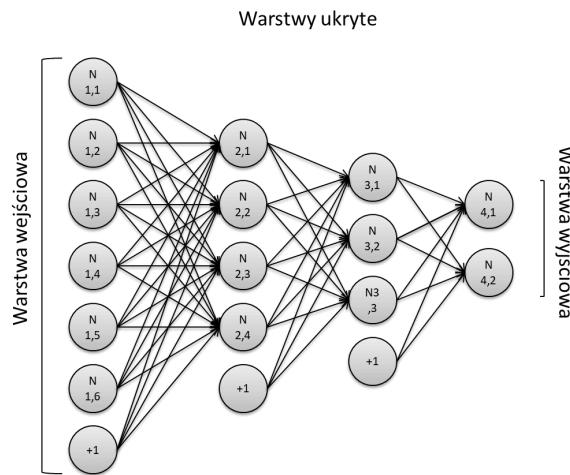
2 Opis sieci neuronowej

Sieć neuronowa (ang. Artificial Neural Network)[1][2] to model, który jest przeznaczony do przetwarzania informacji. Są one sztucznym odwzorowaniem biologicznych sieci neuronowych z których zdobudowane są mózgi zwierząt. Są to kolekcje połączonych ze sobą węzłów, które nazywamy perceptronami, czyli sztucznymi neuronami. Każdy połączenia działa podobnie do synaps, które transmitują sygnały do innych neuronów. Każdy perceptron otrzymawszy wiadomość przetwarza ją za pomocą ustalonej funkcji, a następnie przekazuje tak przetworzoną wiadomość przez synapsy do kolejnych neuronów. Zadaniem sieci neuronowych jest przetworzenie danych wejściowych na dane wyjściowe. W jaki sposób sieć neuronowa wie na jakie dane ma przetworzyć odpowiednie wejście? Dzieje się to poprzez proces uczenia sieci neuronowej, czyli wielokrotnego pokazywania danych i dostosowywania połączeń

2.1 Warstwy sieci neuronowej

W następnej części raportu, gdy będziemy mówić o sieciach neuronowych będziemy mieli na myśli jeden konkretny typ sztucznych sieci neuronowych - perceptron wielowarstwowy (ang. multilayer perceptron - MLP). Perceptron wielowarstwowy składa się z warstw neuronów, które możemy podzielić na trzy rodzaje:

1. *Warstwa wejściowa (ang. input layer)* - otrzymuje dane wejściowe, na podstawie których cała sieć ma obliczyć wynik. Ta warstwa nie przetwarza danych funkcjami aktywacyjnymi,
2. *Warstwa wyjściowa (ang. output layer)* - tworzy ostateczny wynik sieci neuronowej, który jest zwracany do użytkownika,
3. *Warstwy ukryte (ang. hidden layers)* - wszystkie warstwy, który znajdują się pomiędzy *warstwą wejściową*, a *warstwą wyjściową*



Rysunek 2: Model sieci neuronowej.

Źródło: <https://bulldogjob.pl/readme/czym-jest-deep-learning-i-sieci-neuronowe>

Każda warstwa składa się z określonej liczby perceptronów, która może być różna dla poszczególnych warstw. Perceptrony w danej warstwie są połączone z perceptronami z warstwy następnej. W naszej architekturze zdecydowaliśmy się na warstwy, które są ze sobą całkowicie połączone (ang. fully connected), co oznacza, że każdy perceptron z danej warstwy jest połączony ze wszystkimi perceptronami z następnej. Są one połączone tylko w jedną stronę, od warstwy wejściowej do warstwy wyjściowej, przez co nazywamy taką sieć *jednokierunkową siecią neuronową* (ang. *feedforward neural network*).

2.2 Pojedynczy perceptron

Pojedynczy perceptron ma połączenia z poprzedniej warstwy, z których otrzymuje dane. Każde połączenie ma również swoją wagę, która odpowiednio zmienia daną otrzymaną z tego połączenia. Jest to obliczane z wzoru:

$$a = \sum_{i=1}^m (w_i * x_i) + b, \quad (1)$$

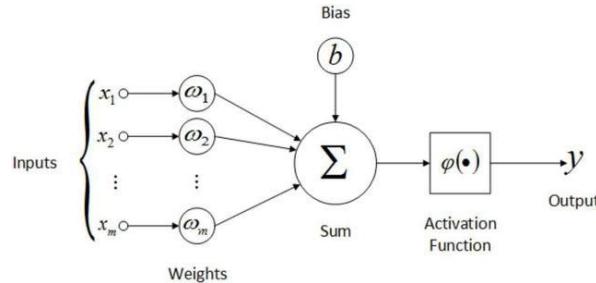
gdzie:

- x_i - wartość otrzymana z i-tego połączenia,
- w_i - waga i-tego połączenia,
- b - bias, stała dodawana do sumy w celu przesunięcia,
- a - wynik kombinacji liniowej danych wejściowych oraz wag z dodanym biasem,
- m to ilość połączeń wchodzących do danego perceptronu, równa ilości perceptronów z warstwy poprzedniej.

Następnie wynik a jest przetwarzany przez funkcję aktywacyjną ϕ , aby otrzymać wyjście z perceptronu:

$$y = \phi(a). \quad (2)$$

Wartość y jest przekazywana do wszystkich perceptronów z warstwy następnej.



Rysunek 3: Pojedynczy perceptron.

Źródło: <https://analyticsindiamag.com/perceptron-is-the-only-neural-network-without-any-hidden-layer/>

W postaci macierzowej będzie to wyglądać następująco:

$$Y_i = \phi_i(X_i \cdot W_i), \quad (3)$$

gdzie X_i to wektor wejściowy do warstwy i-tej, Y_i to wektor wyjściowy warstwy i-tej, W_i to macierz wag w warstwie i-tej. Zakładamy, że bias b^k jest tutaj oznaczony jako składnik w_0^k macierzy W .

2.3 Funkcje aktywacji

W projekcie używamy następujących funkcji aktywacji:

- funkcje aktywacji dla regresji oraz klasyfikacji:
 - sigmoid: $\text{sigmoid}(x) = \frac{1}{1+e^{-x}}$
 - tangens hiperboliczny: $\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$
- funkcje aktywacji dla klasyfikacji:
 - krok binarny: $\text{binary step}(x) = 1$ jeśli $x > 0$ w.p.p. 0
 - leaky relu (REctified Linear Unit): $\text{relu}(x) = x$ jeśli $x > 0$ w.p.p. β , gdzie β to mała liczba np. 0.01

- funkcje aktywacji dla klasyfikacji w ostatniej warstwie:
 - softmax: $softmax(x)_i = \frac{e^x_i}{\sum_{j=1}^n e^x_j}$ - oblicza prawdopodobieństwo wystąpienia x_i w stosunku do wszystkich x , stąd $\sum_{i=1}^n softmax(x)_i = 1$ [3]
- funkcje aktywacji dla regresji w ostatniej warstwie:
 - liniowa: $linear(x) = x$

2.4 Funkcje straty

Po pojedynczym przejściu w przód (ang. feedforward pass) przez funkcję straty obliczana jest wartość straty, która określa "koszt". Koszt jest tym większy im wynik obliczony przez sieć neuronową jest dalszy od wyniku, który sieć powinna osiągnąć dla określonych danych wejściowych. Przez Y będziemy oznaczać wektor wartości prawdziwych, natomiast przez \hat{Y} będziemy oznaczać wektor wartości obliczonych przez sieć, a przez N liczbę obserwacji. W projekcie zostały zaimplementowane następujące funkcje straty, która będzie oznaczana jako L :

- błędy używane w regresji:
 - błąd średniokwadratowy (ang. mean squared error - MSE):

$$\frac{1}{N} \sum_1^N (Y - \hat{Y})^2, \quad (4)$$
 - błąd średnio absolutny (ang. mean absolute error - MAE):

$$\frac{1}{N} \sum_1^N |Y - \hat{Y}|, \quad (5)$$
 - błąd średniokwadratowy logarytmiczny (ang. mean squared logarithmic error - MSLE):

$$\frac{1}{N} \sum_1^N (\log(1 + Y) - \log(1 + \hat{Y})), \quad (6)$$

- błędy używane w klasyfikacji:
 - strata entropii krzyżowej (ang. cross entropy loss):

$$-\sum_{i=1}^N (y_i * \log(\hat{y}_i)), \quad (7)$$
 - strata hinge (ang. hinge loss):

$$\frac{1}{N} \sum_{i=1}^N \max(1 - y_i * \hat{y}_i, 0). \quad (8)$$

2.5 Propagacja wsteczna

Propagacja wsteczna (ang. backpropagation) jest to mechanizm, dzięki któremu możemy dostosować wagę w celu minimalizacji propagowanej straty. Odbywa to się w następujących krokach:

1. obliczenie delty na aktualnej warstwie
2. obliczenie zmiany wartości (gradientu) wag w aktualnej warstwie
3. zaktualizowanie wag w aktualnej warstwie

Powyższe kroki realizujemy począwszy od ostatniej i przechodząc wstecz po kolej do pierwszej warstwy ukrytej.

2.5.1 Obliczanie gradientu w ostatniej warstwie [4]

W pierwszym kroku należy obliczyć deltę na ostatniej warstwie. Deltę możemy zapisać jako:

$$\delta_{out} = \frac{dL}{dA_{out}} * \frac{d\phi_{out}}{dY_{out}}. \quad (9)$$

Następnie należy obliczyć gradient dla aktualnej warstwy, co można zrobić poprzez obliczenie iloczynu skalarnego transponowanej macierzy wejścia danej warstwy X oraz δ :

$$grad_{out} = X^T \cdot \delta_{out}. \quad (10)$$

2.5.2 Obliczanie gradientu w warstwach ukrytych [4]

W warstwach ukrytych obliczanie delty będzie się odbywało inaczej niż w warstwie ostatniej. Jako δ_i oraz $grad_i$ oznaczamy deltę oraz gradient w warstwie i -tej. Jako δ_{i+1} oznaczamy deltę w warstwie następnej po i -tej. Jako W_i^T oznaczamy transponowaną macierz wag w warstwie i -tej. Wzór na δ_i wygląda następująco:

$$\delta_i = \delta_{i+1} \cdot W_i^T, \quad (11)$$

następnie na podstawie delty można policzyć gradient podobnie jak w warstwie ostatniej:

$$grad_i = X^T \cdot \delta_i \quad (12)$$

2.5.3 Połączenie funkcji aktywacyjnej softmax oraz entropii krzyżowej [5]

Propagacja wsteczna, wymaga policzenia pochodnych funkcji aktywacji oraz funkcji straty. Funkcji softmax będziemy używać razem z funkcją entropii krzyżowej. Jest dogodna optymalizacja pod względem łatwości zapisu oraz ominięcia możliwych problemów matematycznych, gdybyśmy liczyli pochodne każdej z tych funkcji osobno. Połączenie to jest wyrażone wzorem:

$$\delta_{out} = \frac{dL}{dY_{out}} = \hat{Y} - Y \quad (13)$$

2.5.4 Współczynnik uczenia (ang. learning rate)

Współczynnik uczenia α to parametr, który pozwala na dostosowanie szybkości zmiany wag. Wybór współczynnika uczenia jest bardzo ważny. Wybór zbyt małego współczynnika poskutkuje wydłużonym okresem uczenia, ponieważ bardzo powoli będziemy zmierać do optymalnego rozwiązania. Natomiast zbyt duży współczynnik może przyczynić się do "przeskoczenia" optymalnego rozwiązania, przez co algorytm może nie zmniejszać swojego błędu.

Współczynnik uczenia może być stały przez cały proces uczenia, bądź może się zmieniać wraz z jego trwaniem. Można zauważyć, że błąd w początkowych epokach uczenia jest stosunkowo duży, a błąd w późniejszych jest coraz mniejszy. Jedną z technik dostosowania współczynnika uczenia jest następująca:

$$\alpha_{i+1} = \frac{\alpha_i}{1 + d * i}, \quad (14)$$

gdzie d to parametr zaniku (ang. decay). Pozwala to na początkowo szybkie zmniejszenie błędu oraz późniejsze dokładne dostosowanie do rozwiązania optymalnego.

2.5.5 Aktualizacja wag

Aktualizację wag w warstwie i -tej można zapisać jako:

$$W_i = W_i - \alpha * grad_i. \quad (15)$$

2.5.6 Problem zanikającego gradientu [6]

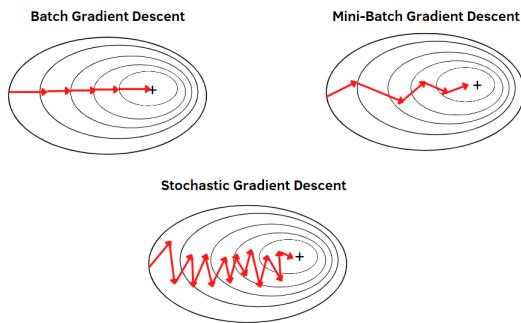
Ważną kwestią jest również problem zanikającego gradientu (ang. vanishing gradient problem). Polega on na normalizacji danych ze względu na funkcje aktywacji, których używamy wewnątrz sieci.

2.5.7 Metoda gradientu prostego

Metoda gradientu prostego w sieciach neuronowych ma na celu znalezienie optymalnego rozwiązania dla konkretnego zadania. W sieciach neuronowych najczęściej używa się trzech rodzajów optymalizacji:

- Metoda stochastycznego gradientu prostego (ang. Stochastic Gradient Descent - SGD) - aktualizacja wag następuje każdorazowo po przejściu przez jedną daną treningową,
- Metoda gradientu prostego dla całego zbioru (ang. Batch Gradient Descent) - aktualizacja wag następuje po każdym przejściu całego zbioru treningowego,
- Metoda gradientu prostego dla małych zbiorów (ang. Mini-Batch Gradient Descent) - aktualizacja wag następuje po przejściu podzbioru całego zbioru treningowego.

Różnice w działaniu są dobrze widoczne na poniższym grafice 4



Rysunek 4: Metoda gradientu prostego.

Źródło: <https://www.analyticsvidhya.com/blog/2022/07/gradient-descent-and-its-types/>

2.5.8 Heurystyka doboru wag

Heurystyka doboru wag pozwala na inicjalizację wag, które są bardziej skuteczne w procesie uczenia sieci neuronowych. Zastosowane heurystyki:

- He: $\sqrt{\frac{2}{n}}$ - n ilość wejść do warstwy, używana dla funkcji aktywacji RELU oraz liniowej,
- Xavier: $\sqrt{\frac{6}{n+m}}$ - n ilość wejść do warstwy, m ilość wyjść z warstwy, używana dla funkcji sigmoid, tanh, kroku binarnego oraz softmax.

3 Nasza implementacja

3.1 Opis modułów sieci

Moduły, z których składa się sieć:

- `neural.net`
- `nn.functions`
- `nn.serializer`
- `dataset`
- `visualizer`

`neural.net` odpowiada za całą logikę sieci neuronowej oraz proces treningu. W `nn.functions` występują wszystkie wykorzystywane funkcje aktywacji, ich pochodne, funkcje straty oraz heurystyki do inicjalizowania początkowych wag. `nn.serializer` odpowiada za proces zapisywania oraz wczytywania konfigów sieci. W `dataset` odbywa się proces wczytywania pliku .csv oraz przetwarzanie tych danych. `visualizer` służy do rysowania wszystkich rodzajów wykresów oraz grafów.

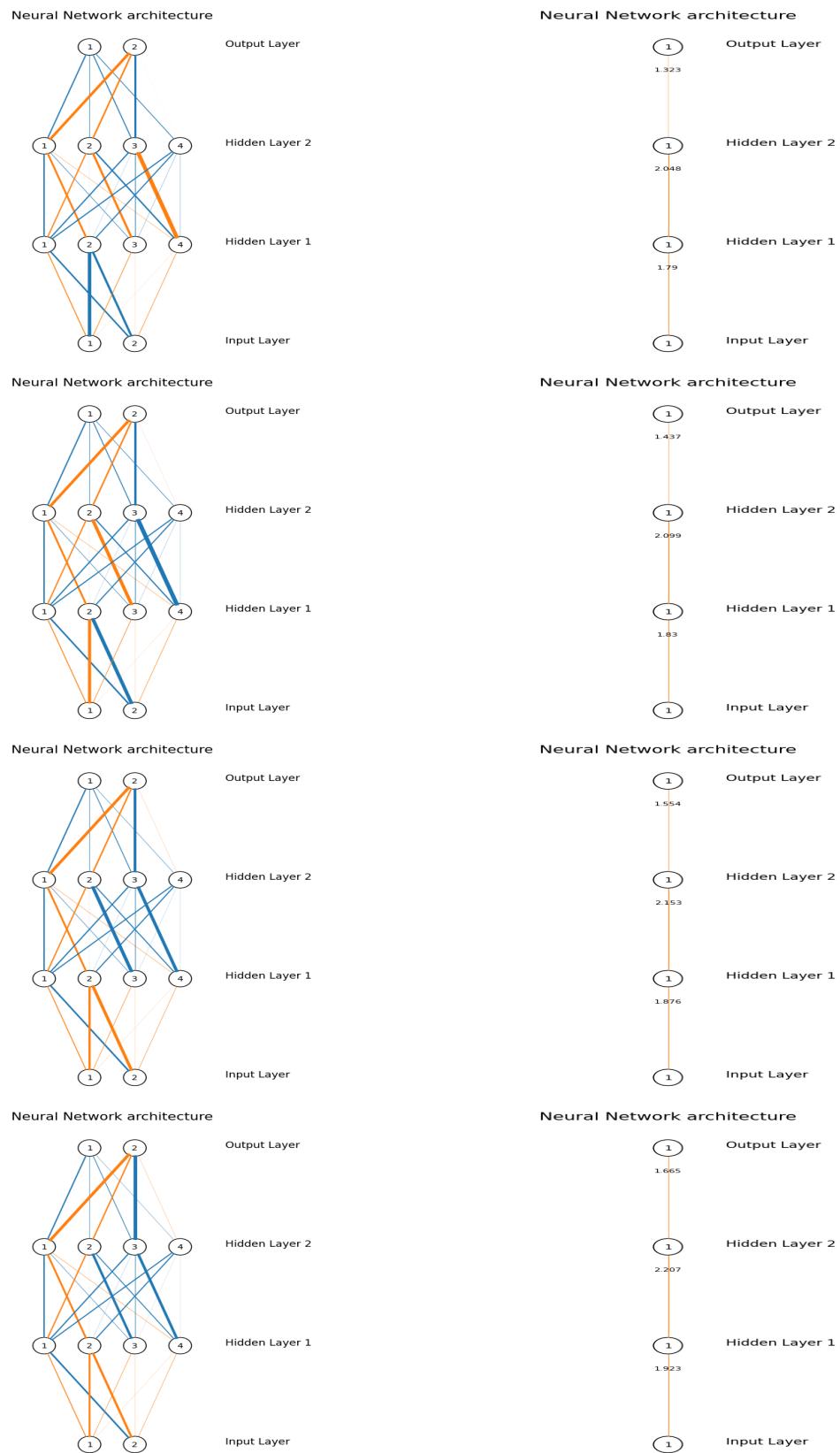
3.2 Ziarno generatora liczb losowych

Zaimplementowaliśmy możliwość inicjowania powtarzalnego procesu uczenia z zadanym ziarnem generatora liczb losowych. Zastosowaliśmy funkcje z pakietu numpy: *np.random.seed()* oraz *np.random.randn()*. Funkcje te pozwalają na ustawienie ziarna oraz wygenerowanie liczb pseudolosowych. Ustawienie ziarna pozwala na przeprowadzenie powtarzalnego procesu uczenia, w dla zadanych zmiennych początkowych wynik uczenia oraz predykcji będzie zawsze taki sam.

3.3 Konfiguracja sieci

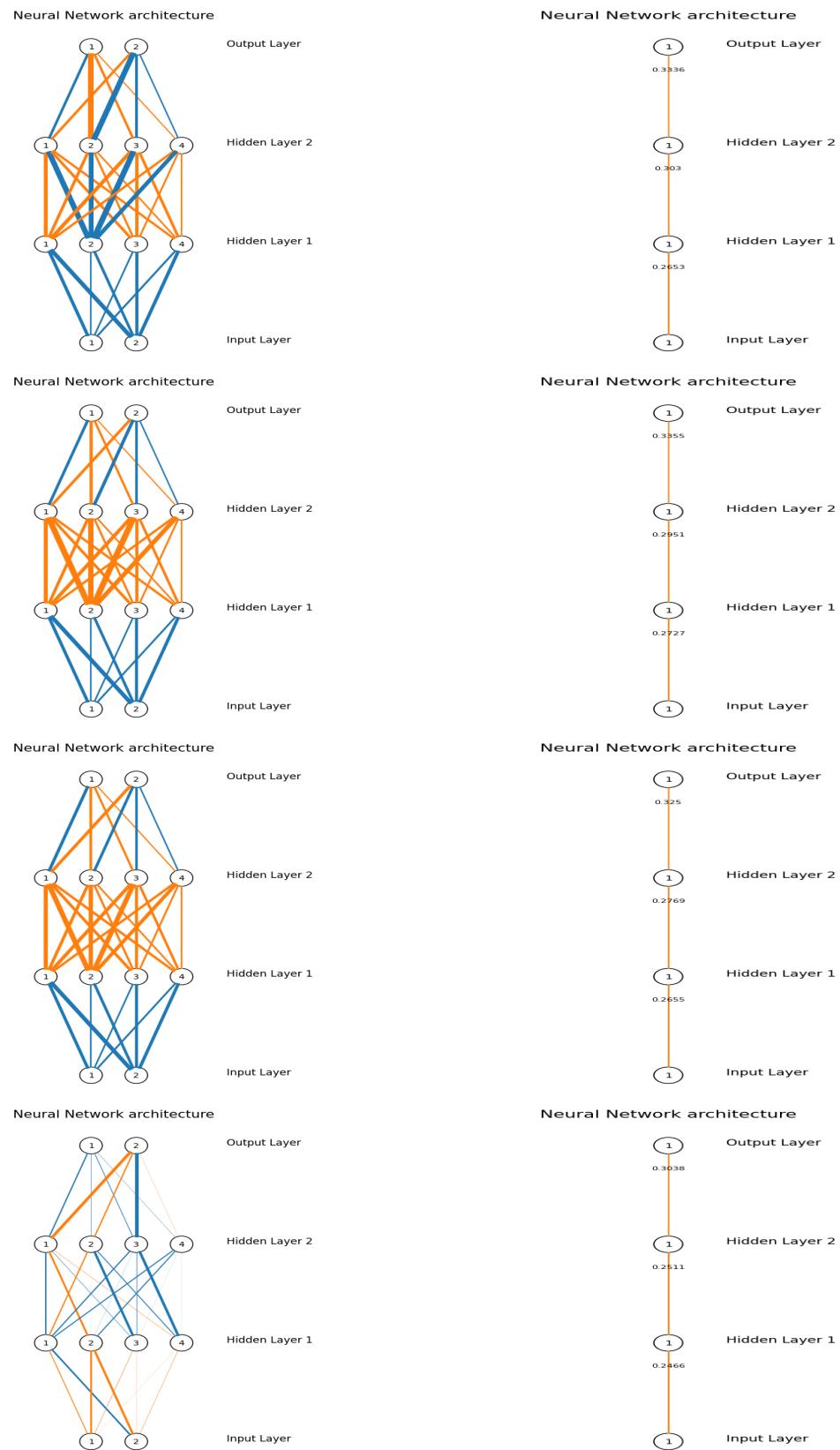
Łatwa konfiguracja liczby warstw w sieci i neuronów w warstwie, obecności biasów (w dniu oddania będzie trzeba szybko dostosować architekturę sieci)

3.4 Wizualizacja wartości wag w kolejnych iteracjach uczenia



Rysunek 5: Wizualizacja wag w 4 kolejnych iteracjach

3.5 Wizualizacja wartości gradientów w kolejnych iteracjach uczenia



Rysunek 6: Wizualizacja wag w 4 kolejnych iteracjach

Przy implementowaniu wizualizacji wartości wag oraz gradientów bazowano na kodzie [7]. Została do tego dodana funkcjonalność, która spłaszcza wartości wag oraz gradientów w poszczególnych warstwach do skalara przy użyciu normy L2:

$$\|\vec{r}\| = \sqrt{w_1^2 + w_2^2 + \dots}$$

4 Zbiory danych

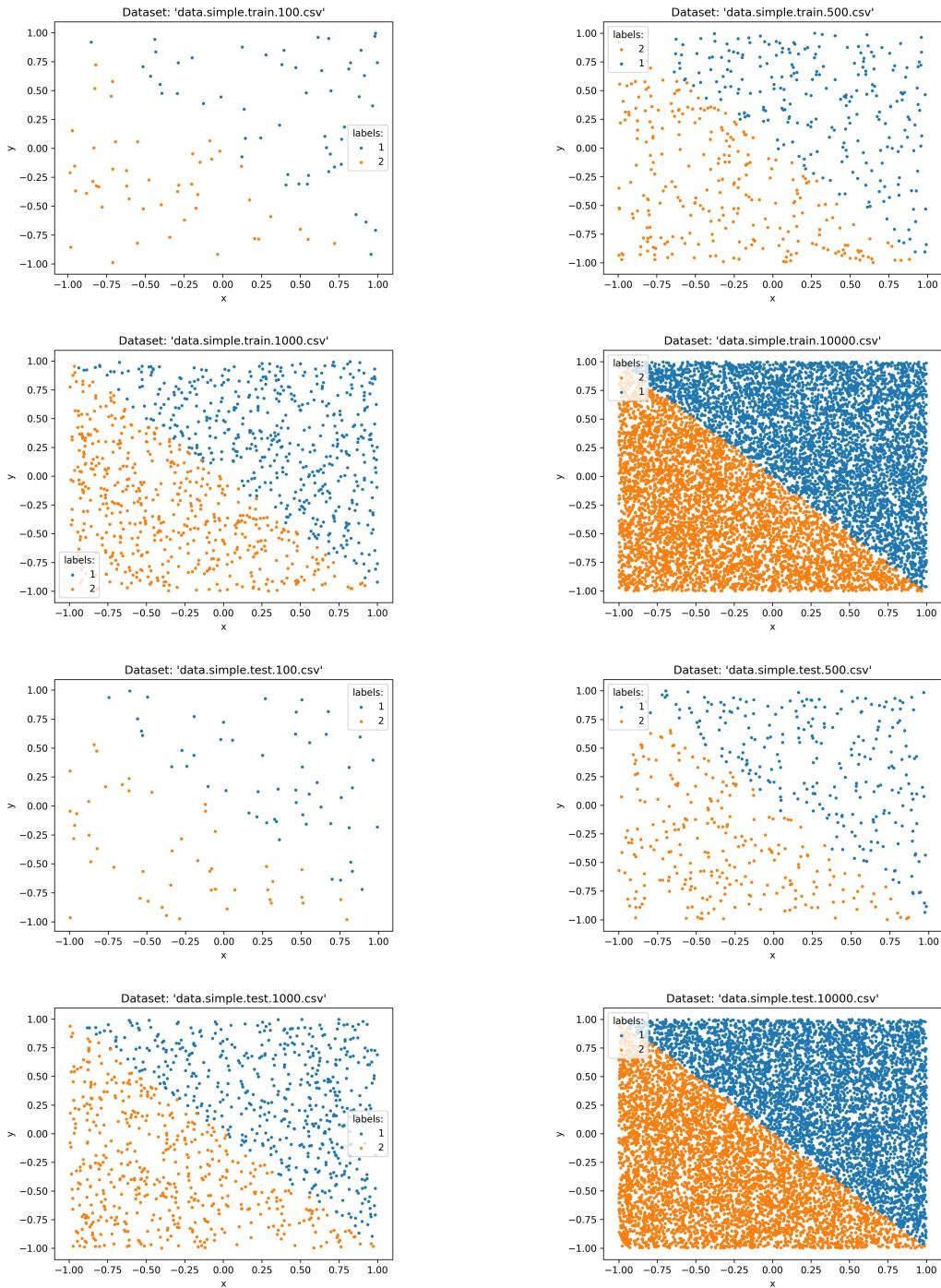
4.0.1 Opis zbioru treningowego (uczącego) i testowego

Dane zostały udostępnione w formacie plików csv z atrybutami [x, y] dla problemu regresji oraz [x, y, cls] dla klasyfikacji. W zbiorach danych została utworzona następująca hierarchia:

- projekt1
 - classification
 - * simple
 - * three_gauss
 - regression
 - * activation
 - * cube
- projekt1-oddanie
 - classification
 - * circles
 - * noisyXOR
 - * XOR
 - regression
 - * linear
 - * multimodal
 - * square

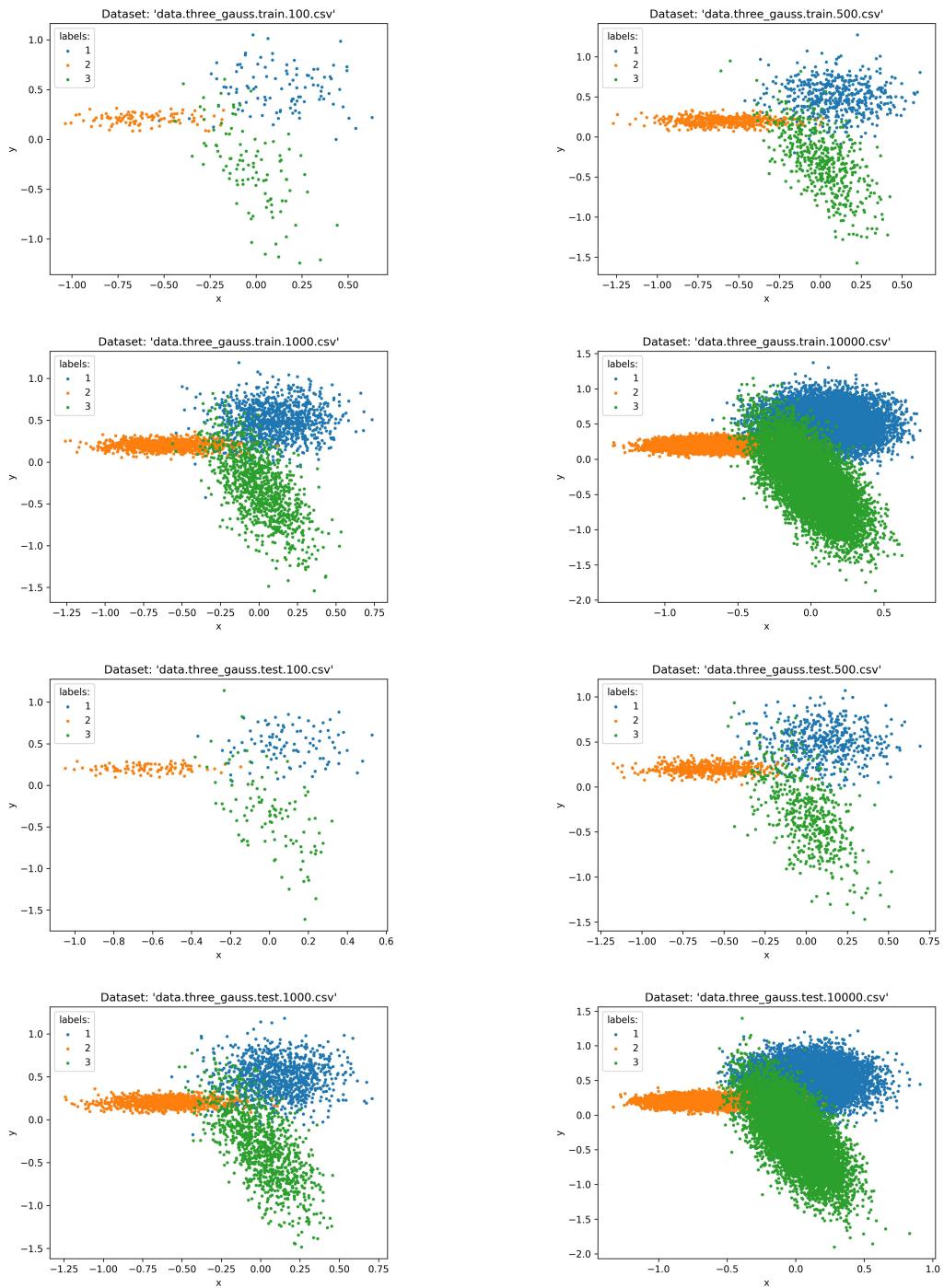
W każdej grupie, czyli na najniższym poziomie w hierarchii występuje podział na zbiór treningowy **train** oraz testowy **test**, a w każdym z nich zbiory po 100, 500, 1000 oraz 10000 rekordów.

4.0.2 Wizualizacja zbiorów



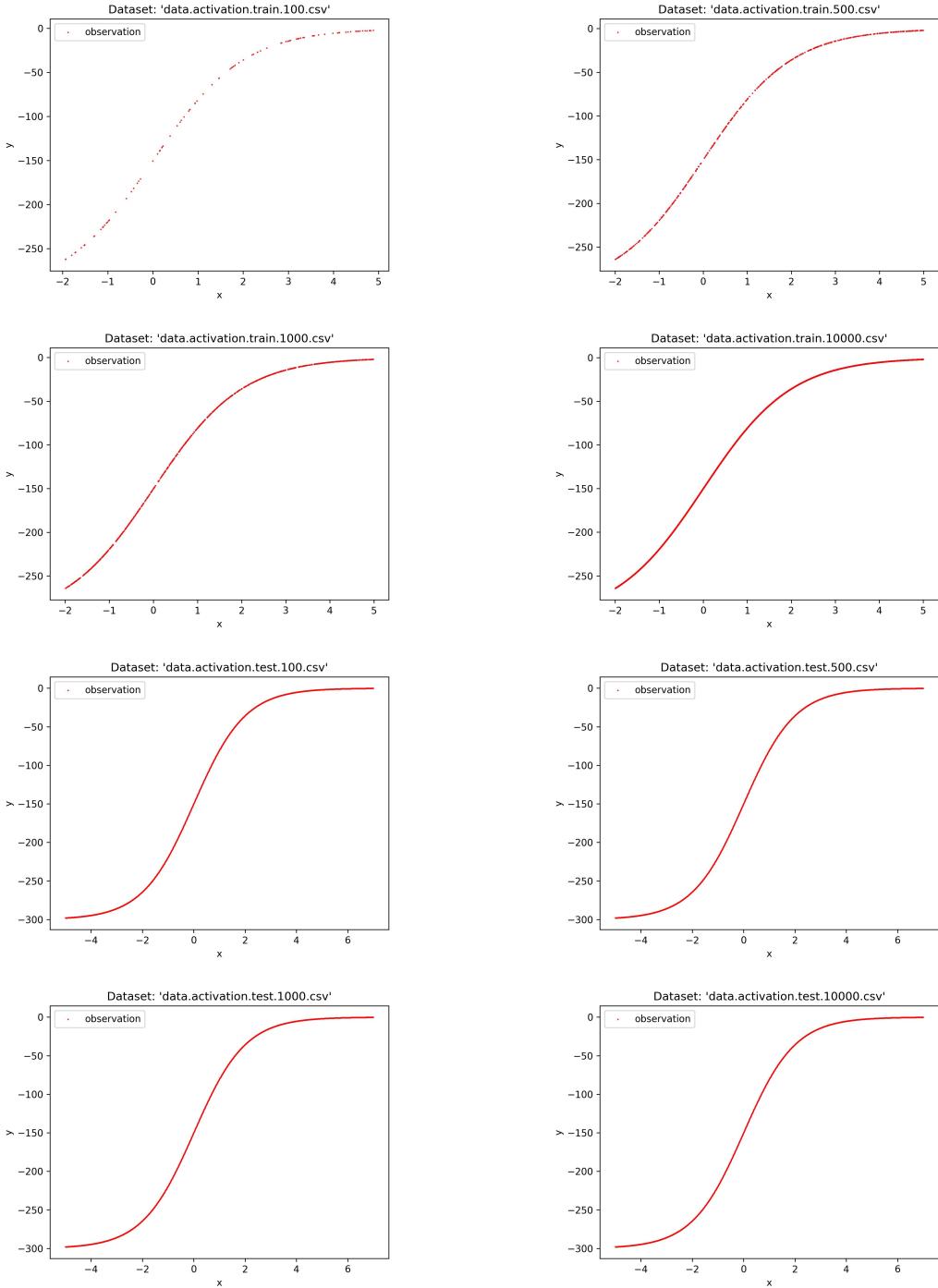
Rysunek 7: Wizualizacja zbioru simple

Na rysunku 7 zbiory `train` oraz `test` wyglądają bardzo podobnie, prawdopodobnie zostały wygenerowane tym samym generatorem. Po rozmiarach 1000 oraz 10000 możemy z dużą dozą pewności przypuszczać, że jest to rozkład równomierny.



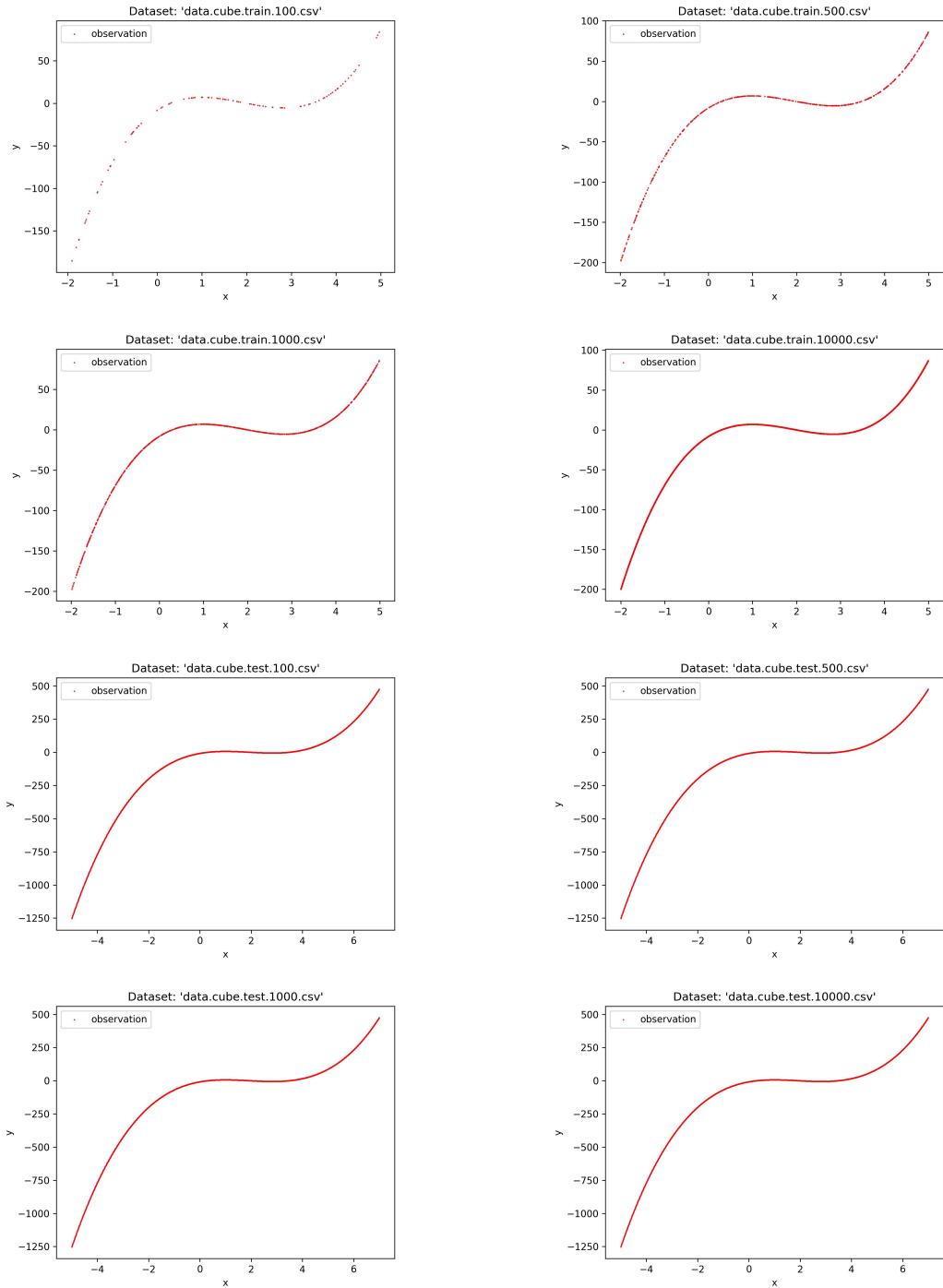
Rysunek 8: Wizualizacja zbioru `three_gauss`

Na rysunku 8 zbiory `train` oraz `test` wyglądają bardzo podobnie, prawdopodobnie zostały wygenerowane tym samym generatorem. Przedstawiają 3 dwuwymiarowe rozkłady normalne z różnymi parametrami wektorów wartości oczekiwanych μ oraz macierzy kowariancji Σ .



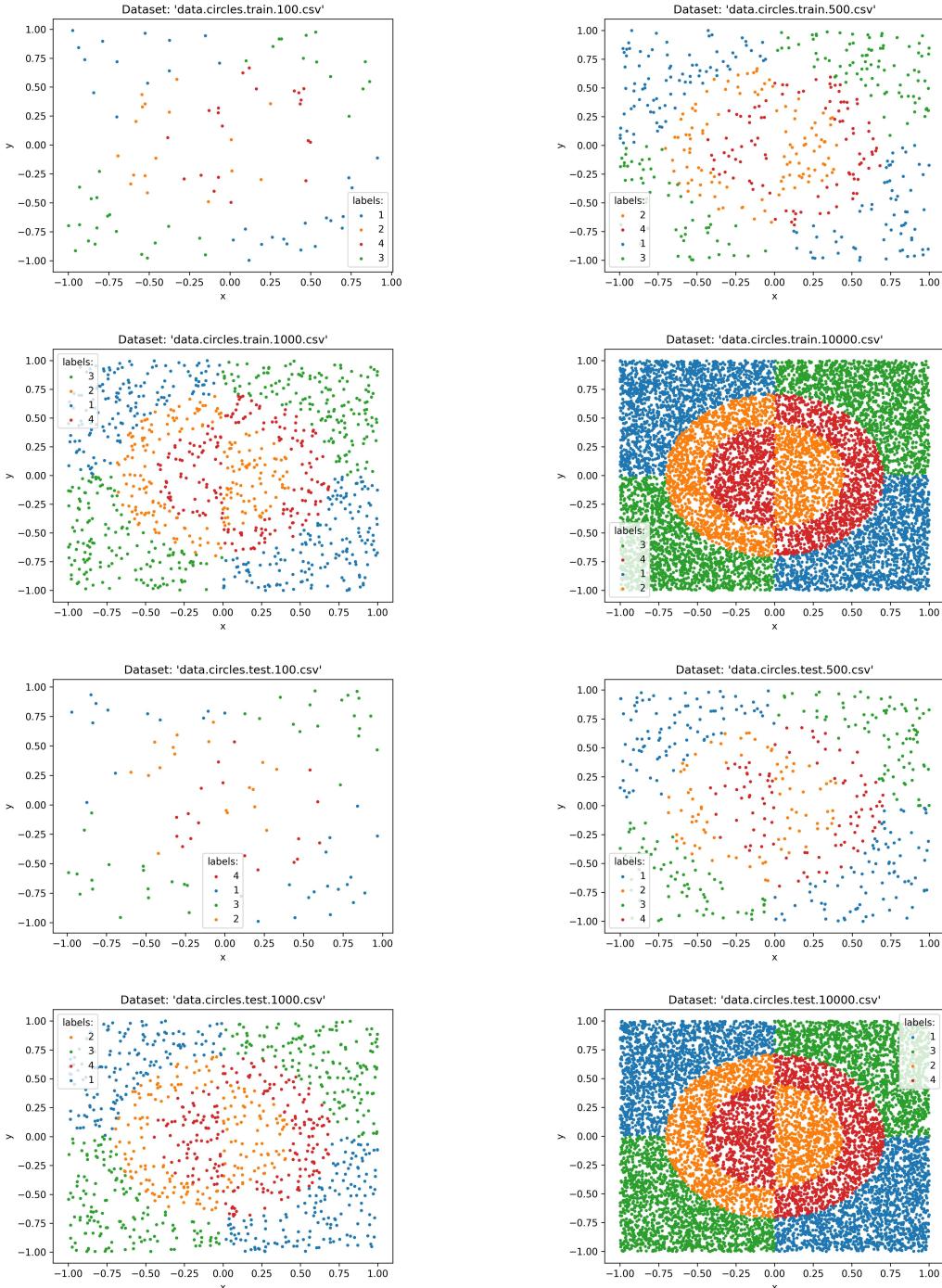
Rysunek 9: Wizualizacja zbioru **activation**

Na rysunku 9 zbiory **train** oraz **test** nieco inaczej. Wynika to z tego, że w zbiorze treningowym mamy zakres argumentów funkcji sigmoidalnej $[-2, 5]$, natomiast w testowym zakres $[-5, 7]$. Dodatkowo można przypuszczać, że **train** jest rozkładu równomiernego. Z kolei **test** ma równoodległe wartości dziedziny x .



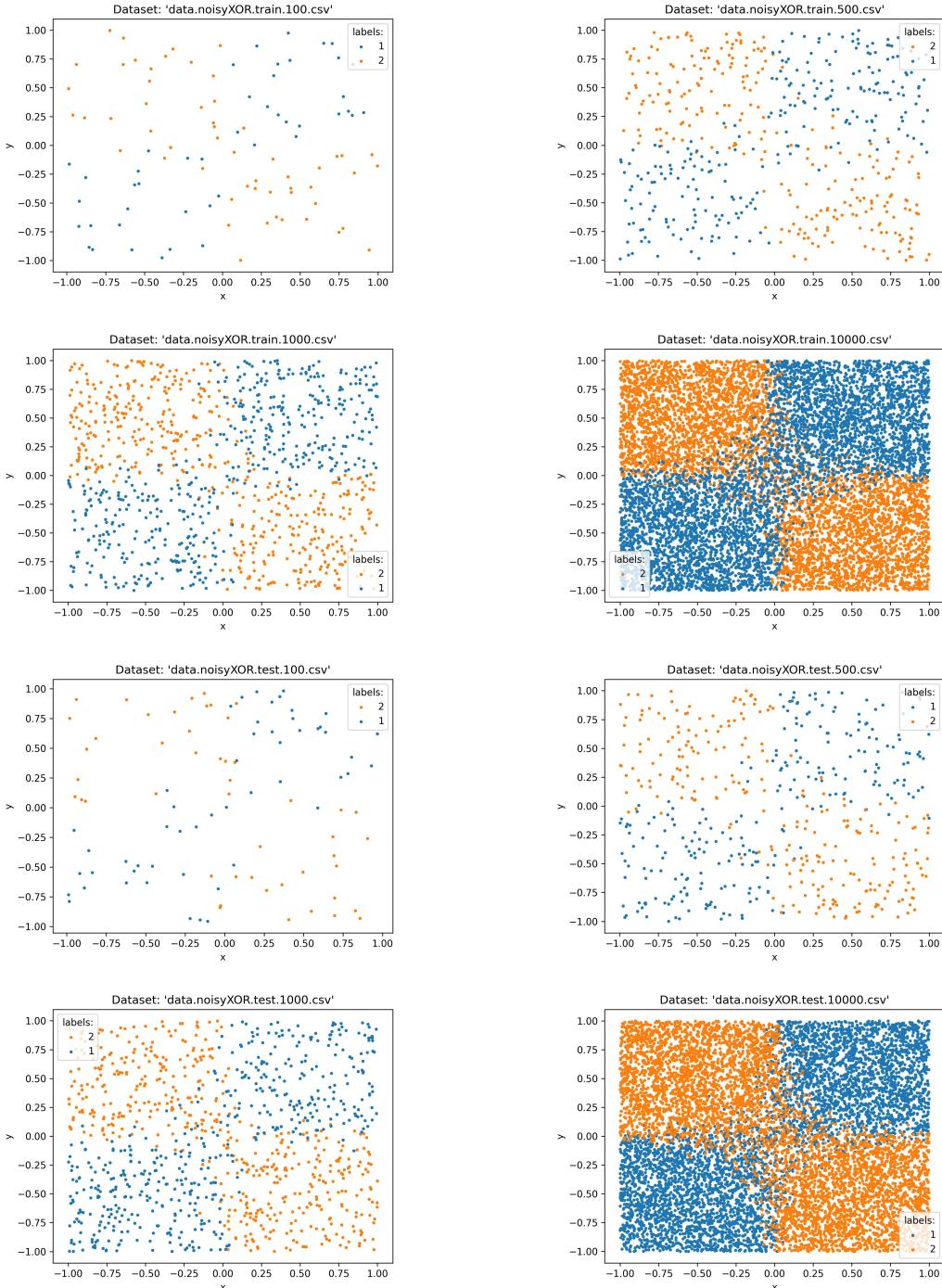
Rysunek 10: Wizualizacja zbioru **cube**

Rysunek 10 można opisać analogicznie jak 9, z jedyną różnicą funkcji, która jest sześciem zamiast sigmoidy.



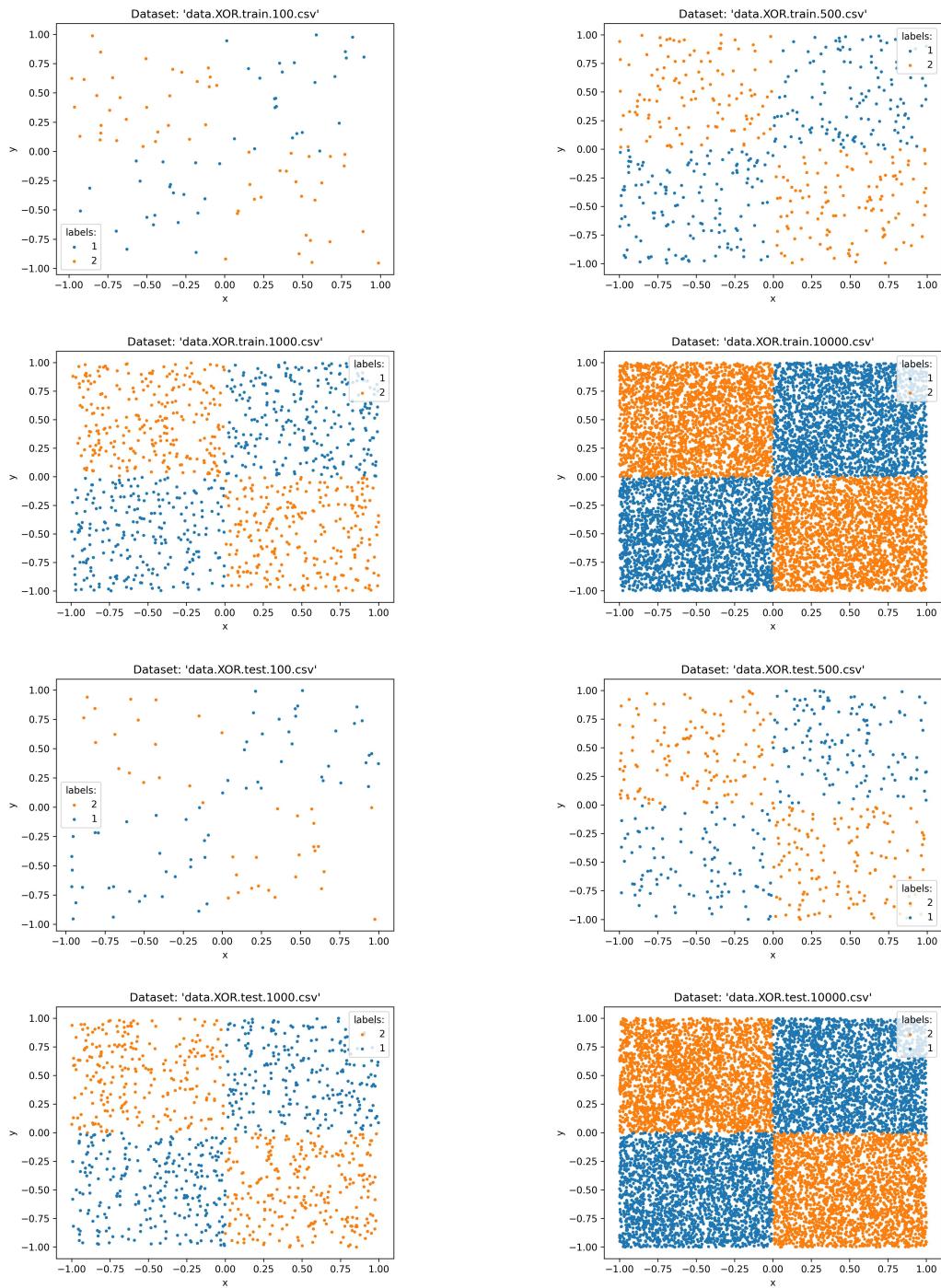
Rysunek 11: Wizualizacja zbioru **circles**

Na rysunku 11 zbiory **train** oraz **test** wyglądają bardzo podobnie, prawdopodobnie zostały wygenerowane tym samym generatorem. Po rozmiarach 1000 oraz 10000 możemy z dużą dozą pewności przypuszczać, że jest to dwuwymiarowy rozkład równomierny. Grupy obiektów układają się w charakterystyczne półokręgi na prostokątnych płaszczyznach.



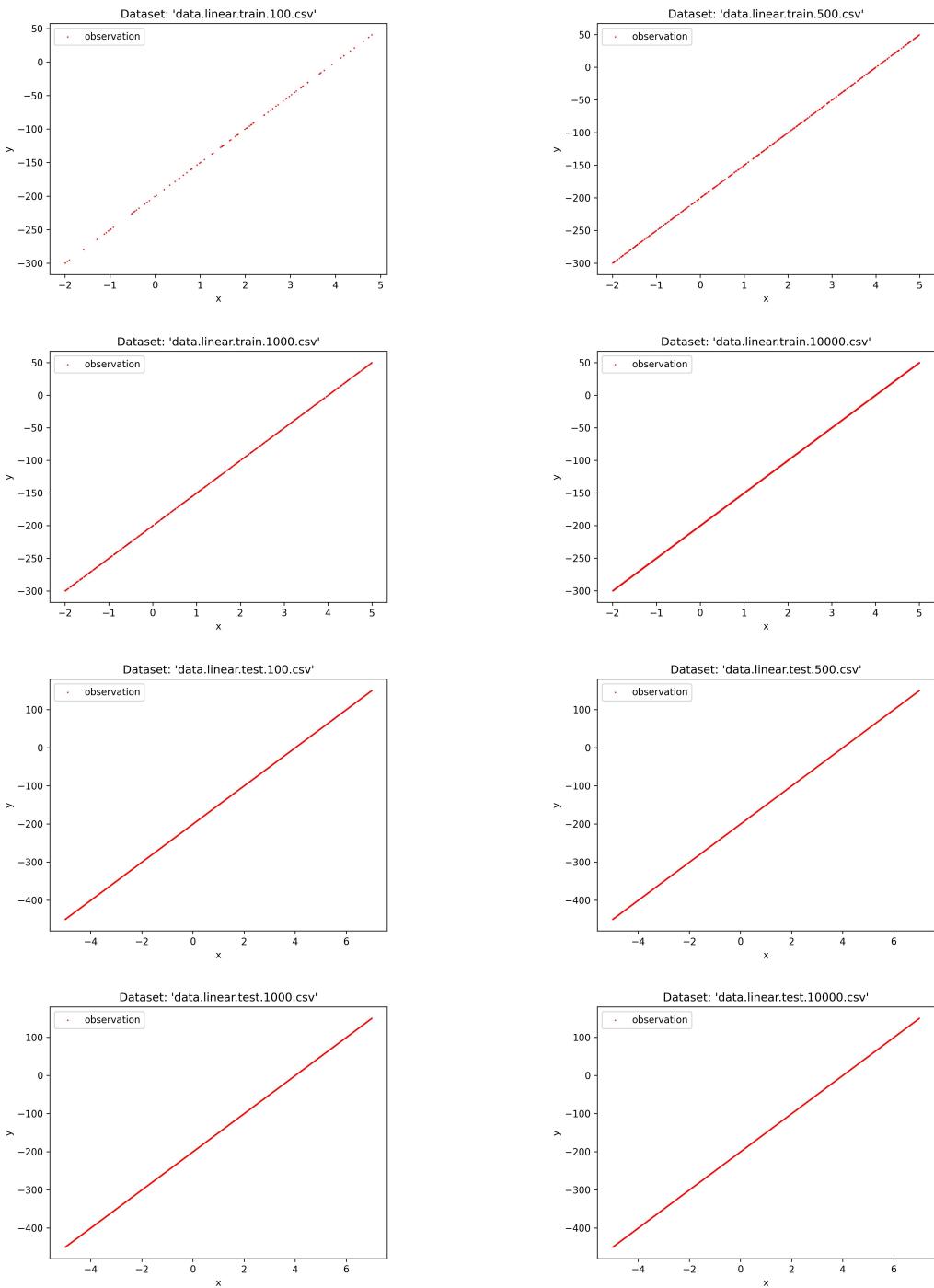
Rysunek 12: Wizualizacja zbioru `noisyXOR`

Na rysunku 11 zbiory `train` oraz `test` wyglądają bardzo podobnie, prawdopodobnie zostały wygenerowane tym samym generatorem. Po rozmiarach 1000 oraz 10000 możemy z dużą dozą pewności przypuszczać, że jest to dwuwymiarowy rozkład równomierny. Grupy obiektów układają się w charakterystyczne prostokąty z pewnym zaszumieniem.



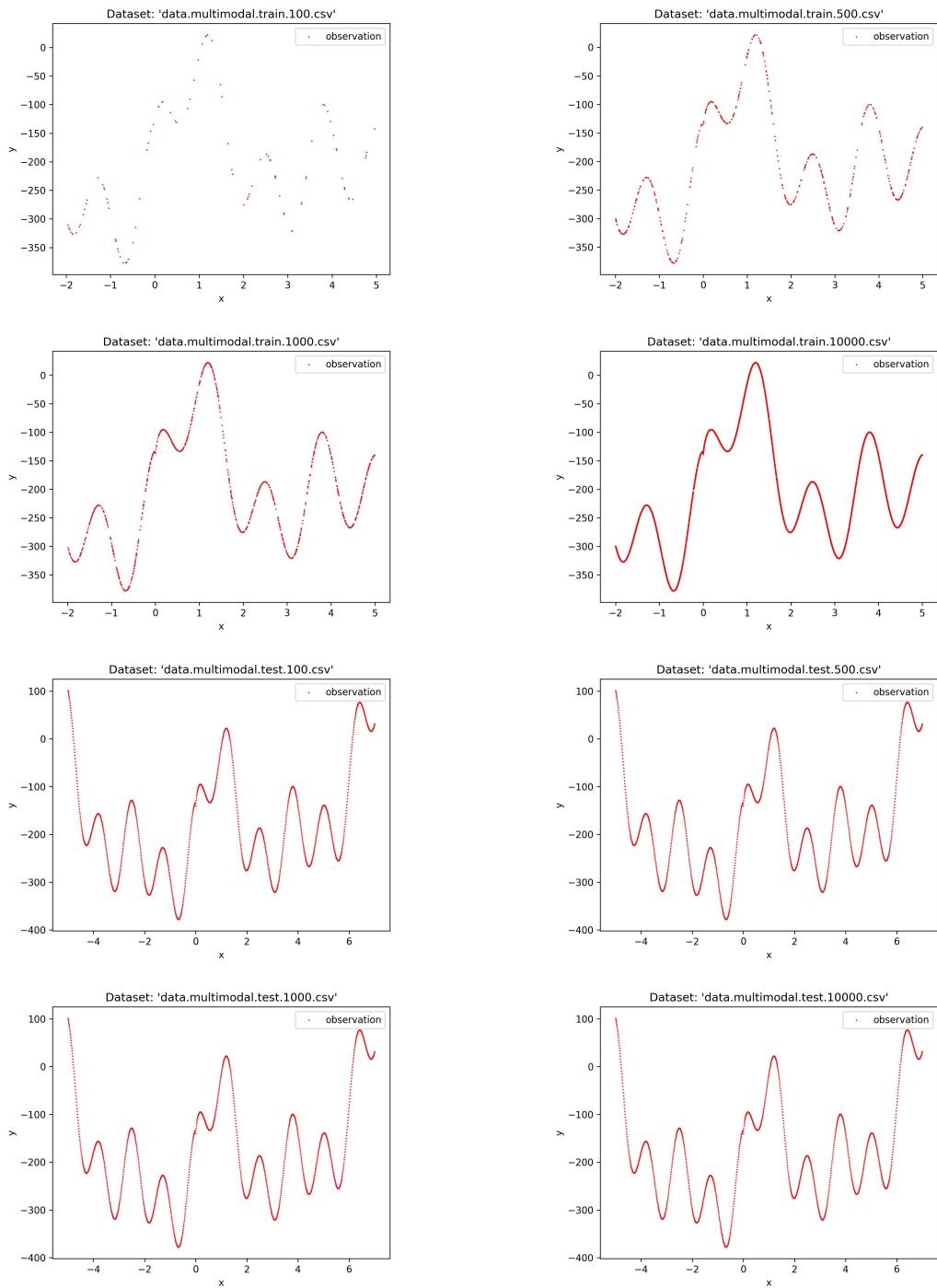
Rysunek 13: Wizualizacja zbioru XOR

Rysunek 13 przedstawia analogiczny zbiór jak 12, ale bez zaszumienia.



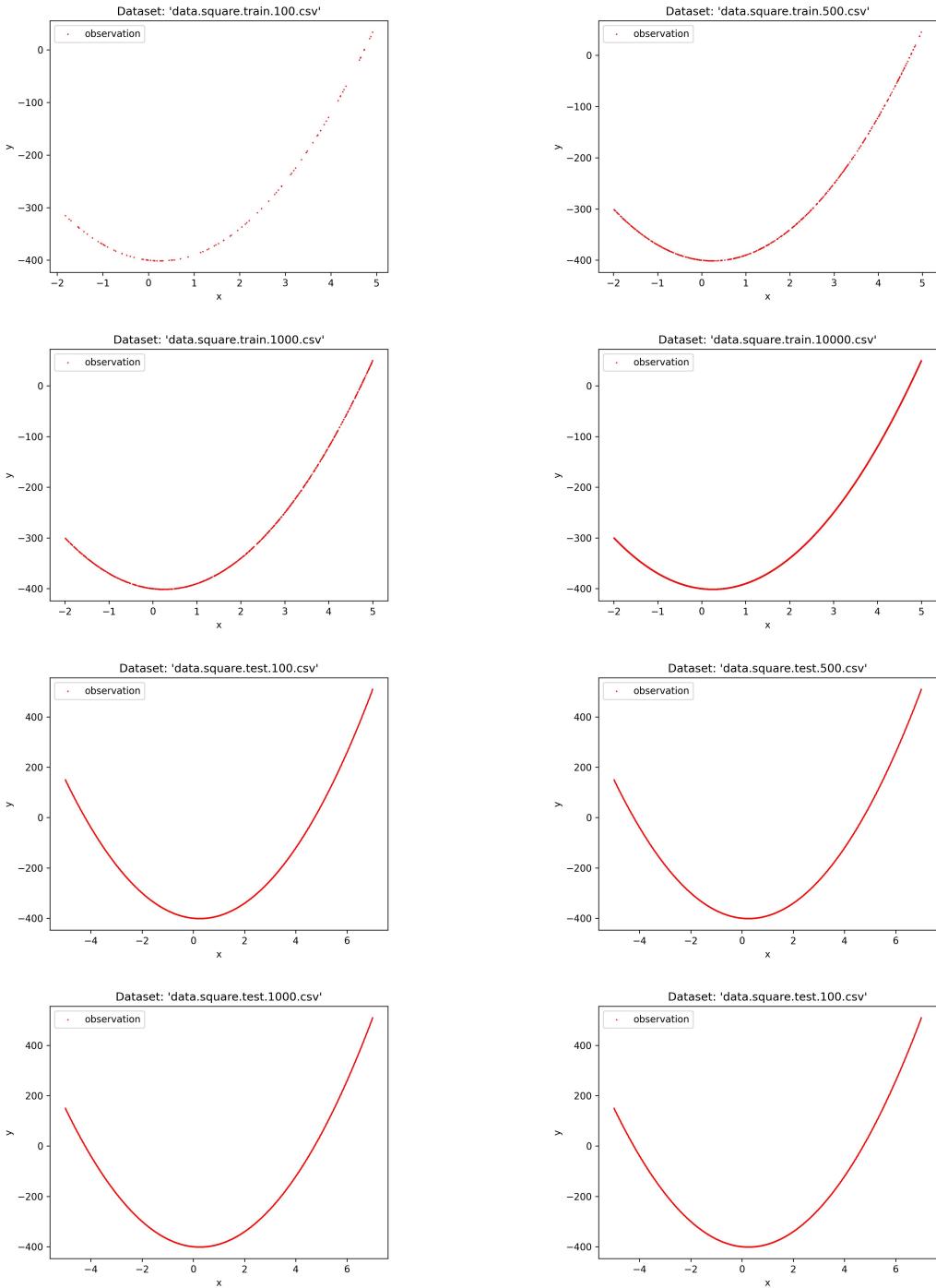
Rysunek 14: Wizualizacja zbioru `linear`

Rysunek 14 można opisać analogicznie jak 9, z jedyną różnicą funkcji, która jest funkcją liniową zamiast sigmoidy.



Rysunek 15: Wizualizacja zbioru **multimodal**

Rysunek 15 można opisać analogicznie jak 9, z jedyną różnicą funkcji, która jest pewną funkcją multimodalną (funkcją ciągłą, dla której w zadanym przedziale istnieje wiele ekstremów lokalnych) zamiast sigmoidy.



Rysunek 16: Wizualizacja zbioru **square**

Rysunek 15 można opisać analogicznie jak 9, z jedyną różnicą funkcji, która jest pewną funkcją kwadratową zamiast sigmoidy.

5 Eksperymenty

5.1 Wpływ funkcji aktywacji na skuteczność działania sieci

5.1.1 Porównanie funkcji sigmoidalnej oraz funkcji tangensa hiperbolicznego dla regresji na zbiorze 'activation'

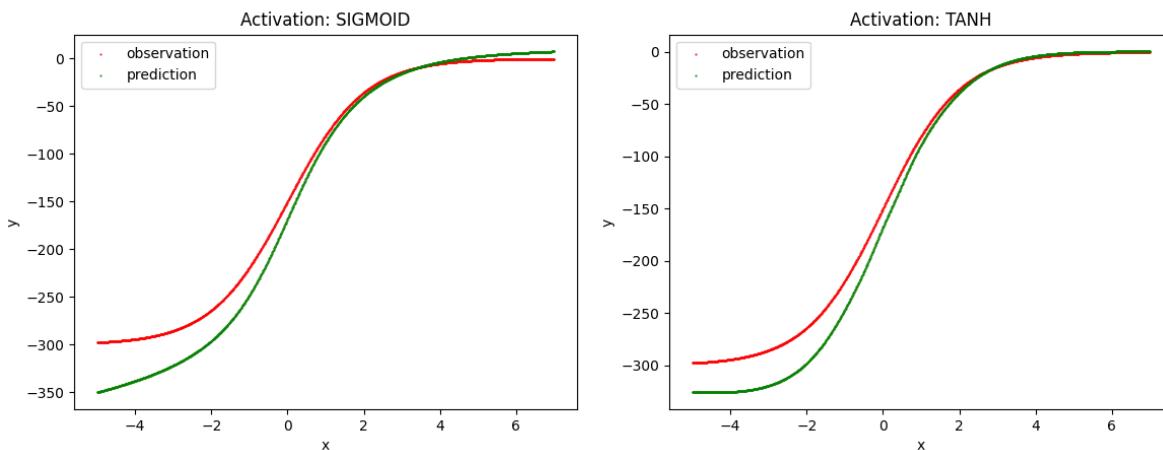
Parametry testu dla obu przypadków:

- zbiór treningowy: activation.train.500.csv
- zbiór testowy: activation.test.500.csv
- funkcja aktywacji: liniowa,
- liczba epoch: 300,
- współczynnik uczenia: 0.01
- warstwy ukryte: 2 x 8 perceptronów,
- funkcja straty: błąd średniokwadratowy,
- obecne biasy.

Wyniki testów:

	SIGMOID	TANH
Czas treningu	3.21s	3.38s
Błąd	0.00013	0.00041

Tabela 1: Porównanie funkcji sigmoid oraz tanh na zbiorze 'activation'.



Rysunek 17: Porównanie funkcji sigmoid oraz tanh na zbiorze activation

Na podstawie tabeli 1 oraz rysunku 17 obie sieci nie radzą sobie aż tak dobrze dla funkcji sześciennu. Prawdopodobnie wynika to z za małej liczby epok.

5.1.2 Porównanie funkcji sigmoidalnej oraz funkcji tangensa hiperbolicznego dla regresji na zbiorze 'cube'

Parametry testu dla obu przypadków:

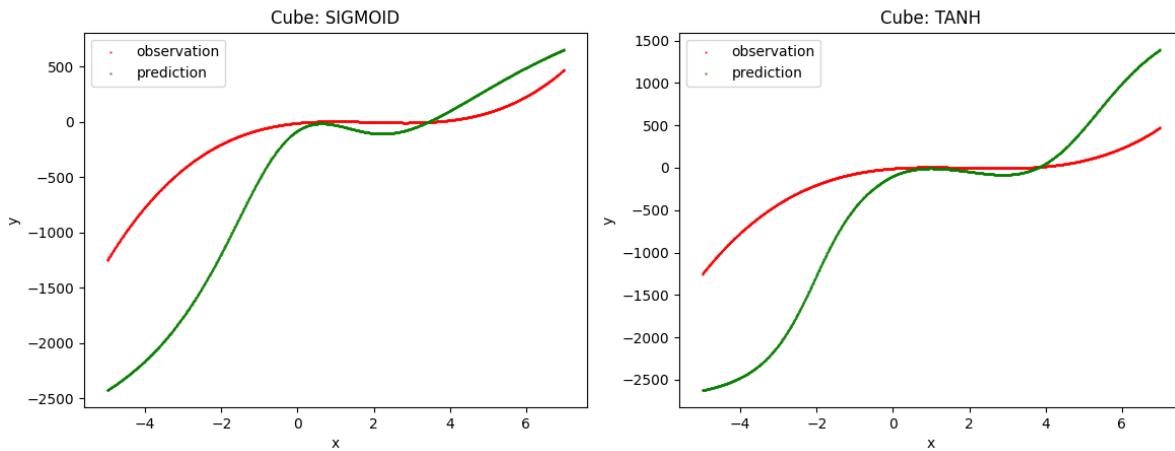
- zbiór treningowy: cube.train.500.csv

- zbiór testowy: cube.test.500.csv
- funkcja aktywacji: liniowa,
- liczba epoch: 300,
- współczynnik uczenia: 0.01
- warstwy ukryte: 1×32 perceptrony + 1×16 perceptronów
- funkcja straty: błąd średniokwadratowy,
- obecne biasy.

Wyniki testów:

Cecha	SIGMOID	TANH
Czas treningu	3.51s	3.76s
Błąd	0.588	1.818

Tabela 2: Porównanie funkcji sigmoid oraz tanh na zbiorze 'activation'.



Rysunek 18: Porównanie funkcji sigmoid oraz tanh na zbiorze cube

Na podstawie tabeli 2 oraz rysunku 18 obie sieci nie radzą sobie aż tak dobrze dla funkcji sześciianu. Prawdopodobnie wynika to z za małej liczby epok.

5.1.3 Porównanie funkcji sigmoidalnej oraz funkcji tangensa hiperbolicznego dla klasyfikacji na zbiorze 'simple'

Parametry testu dla obu przypadków:

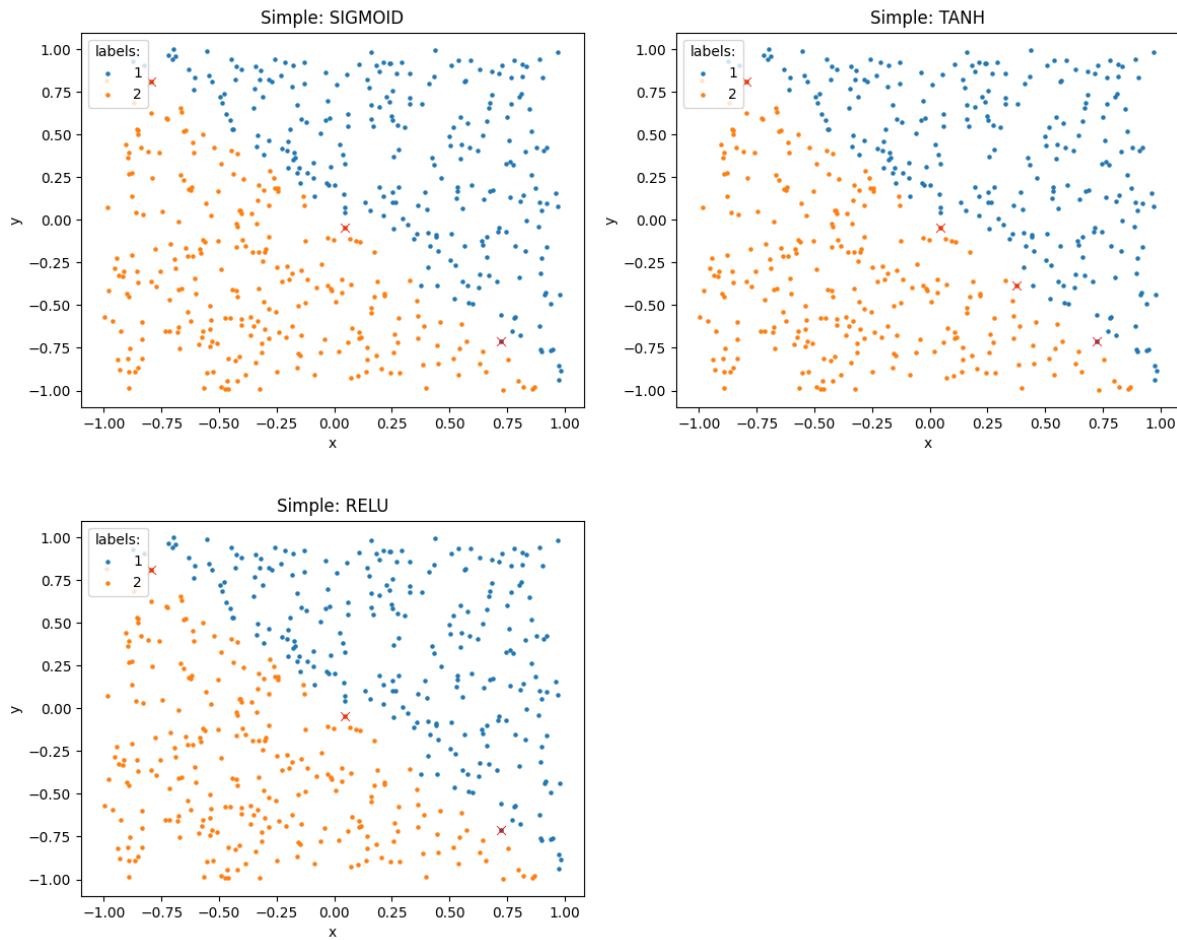
- zbiór treningowy: simple.train.500.csv
- zbiór testowy: simple.test.500.csv
- liczba epoch: 300,
- współczynnik uczenia: 0.01
- warstwy ukryte: 1×4 perceptrony,
- funkcja aktywacji: softmax,
- funkcja straty: entropia krzyżowa,

- obecne biasy.

Wyniki testów:

	SIGMOID	TANH	RELU
Czas treningu	3.096s	3.237s	3.372s
Błąd	0.0213	0.0148	0.0154

Tabela 3: Porównanie funkcji sigmoid oraz tanh na zbiorze 'simple'.



Rysunek 19: Porównanie funkcji sigmoid, tanh oraz relu na zbiorze 'simple'

Na podstawie tabeli 3 oraz rysunku 19 dla RELU oraz TANH osiągnęliśmy lepsze rezultaty niż dla SIGMOID.

5.1.4 Porównanie funkcji sigmoidalnej oraz funkcji tangensa hiperbolicznego dla klasyfikacji na zbiorze 'three gauss'

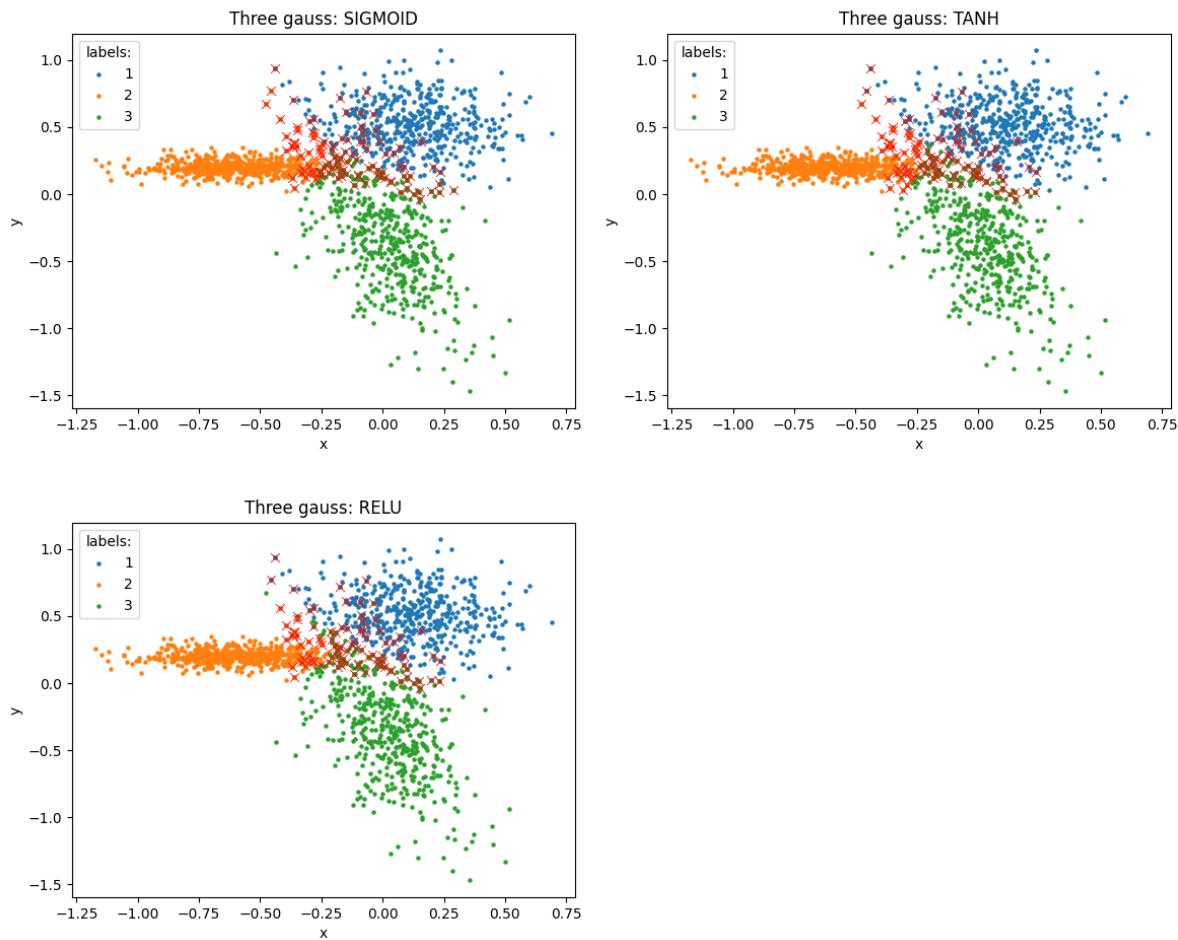
Parametry testu dla obu przypadków:

- zbiór treningowy: simple.train.500.csv
- zbiór testowy: simple.test.500.csv
- liczba epoch: 300,
- współczynnik uczenia: 0.001

- warstwy ukryte: 1 x 16 perceptronów,
- funkcja aktywacji: softmax,
- funkcja straty: entropia krzyżowa,
- obecne biasy.

	SIGMOID	TANH	RELU
Czas treningu	9.249s	8.425s	8.800s
Błąd	0.205	0.197	0.181

Tabela 4: Porównanie funkcji sigmoid oraz tanh na zbiorze 'three gauss'.



Rysunek 20: Porównanie funkcji sigmoid, tanh oraz relu na zbiorze 'three gauss'

Na podstawie tabeli 4 oraz rysunku 20 dla opisanych założeń oraz zbiorów wszystkie trzy funkcje aktywacyjne radzą sobie porównywalnie dobrze. Nie widać dużych różnic ani w czasach uczenia, ani w błędzie na zbiorze testowym.

5.2 Wpływ liczby warstw ukrytych w sieci i ich liczności

Parametry testu dla obu przypadków:

- zbiór treningowy: data.square.train.1000.csv

- zbiór testowy: data.square.test.1000.csv
- funkcja aktywacji: tanh,
- liczba epoch: 500,
- współczynnik uczenia: 0.01
- liczba warstwy ukrytej: 0, 1, 2, 3, 4,
- liczba perceptronów w warstwie ukrytej: 5, 10, 20
- funkcja straty: błąd średniokwadratowy,
- nieobecne biasy.

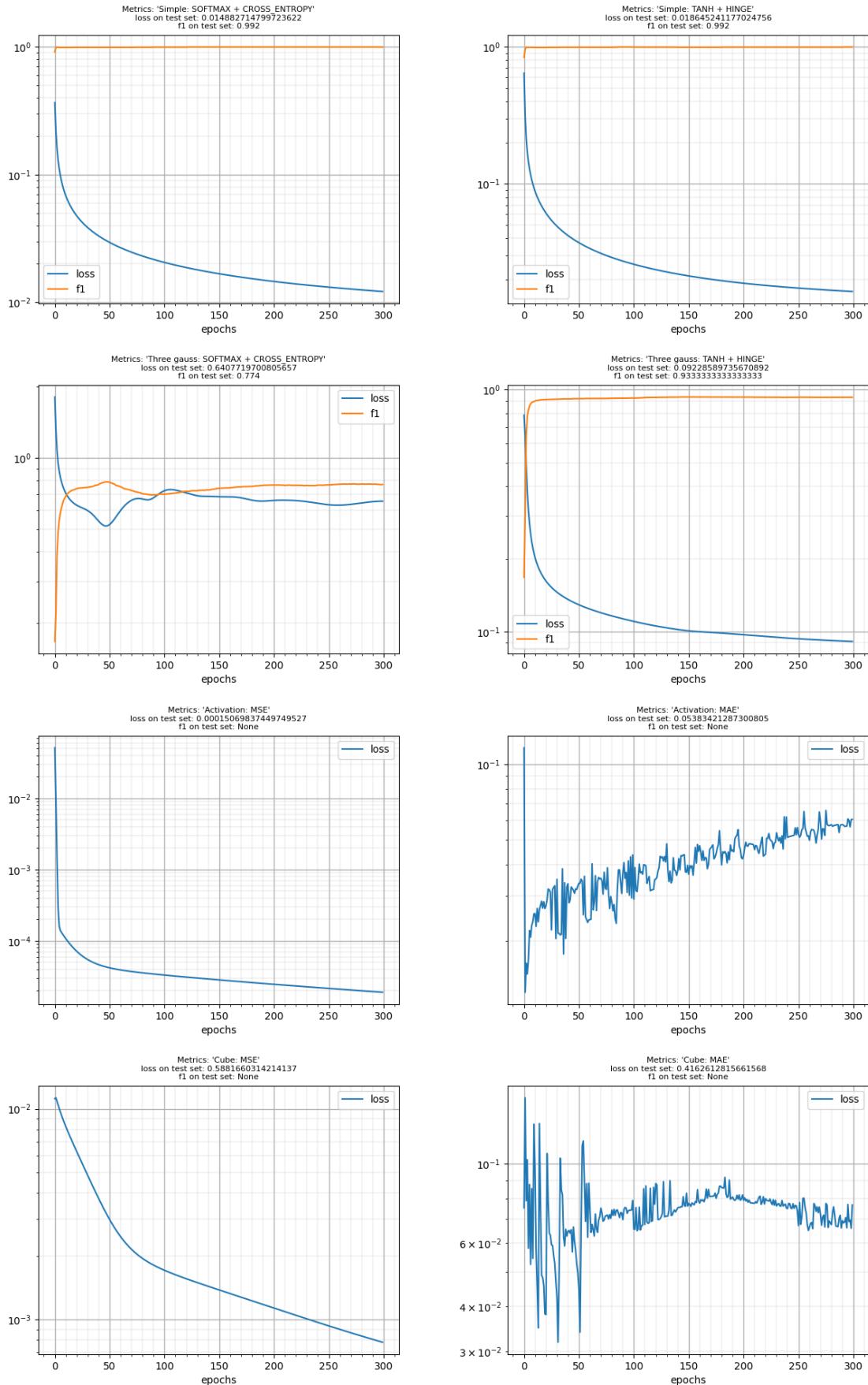
Liczba warstw ukrytych	Liczba perceptronów w warstwie ukrytej	Loss
0	-	1.0549
1	5	0.8776
	10	0.8360
	20	0.8101
2	5	0.7955
	10	0.7997
	20	0.7910
3	5	0.7907
	10	0.7907
	20	0.7885
4	5	0.7952
	10	0.8013
	20	0.7923

Tabela 5: Porównanie różnych architektur sieci

Z tabeli 5 możemy zauważyć, że im więcej warstw ukrytych, tym mniejszy loss. Wyjątkiem jest tu dodanie kolejnej czwartej warstwy, gdzie model zaczyna radzić sobie gorzej. Im więcej perceptronów w warstwie ukrytej, tym mniejszy jest loss. Model bez warstwy ukrytej ma znacząco słabszy rezultat niż reszta.

5.3 Wpływ miary błędu na wyjściu sieci na skuteczność uczenia

Przedstawiamy wykresy, na których można porównać wpływ miary błędu na wyjściu sieci na skuteczność uczenia. Dla klasyfikacji porównaliśmy funkcje Hinge (z funkcją aktywacji na ostatniej warstwie tanh) oraz Cross Entropy (z funkcją aktywacji na ostatniej warstwie softmax). Natomiast dla regresji zostały porównane funkcje straty MSE ora MAE. Dla odpowiadających sobie testów, klasyfikacji oraz regresji, wszystkie inne parametry były pozostawione bez zmian.

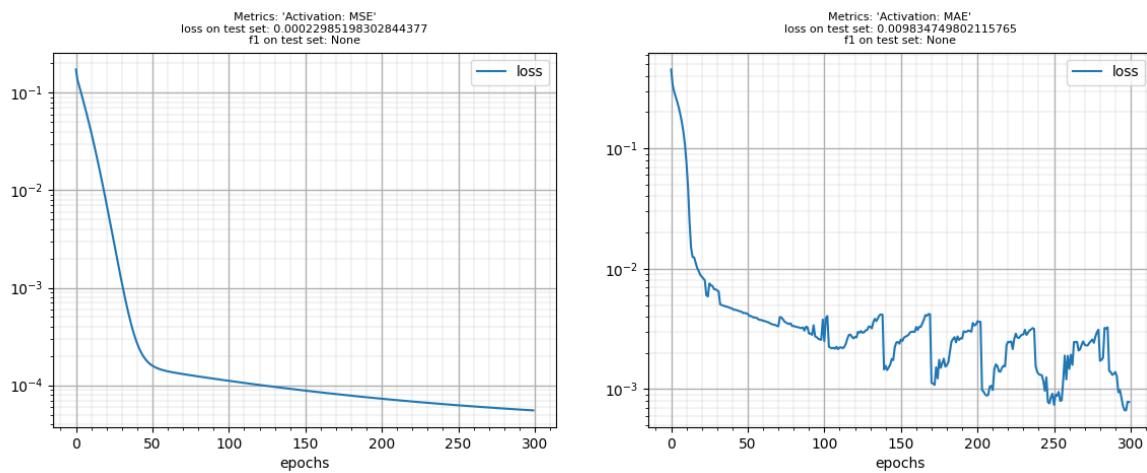


Rysunek 21: Wizualizacja błędów funkcji straty w kolejnych iteracjach uczenia

Na podstawie powyższych rysunków możemy wysnuć następujące wnioski:

- Dla zbioru 'simple' zarówno 'cross entropy' jak i 'hinge' poradziły sobie porównywalnie dobrze,
- Dla zbioru 'three gauss' połączenie funkcji 'tanh' we wszystkich warstwach wraz z funkcją aktywacji 'hinge' daje o wiele lepsze wyniki,
- 'MSE' daje dużo lepsze wyniki niż 'MAE' dla przyjętych danych. Może to być spowodowane zbyt dużym współczynnikiem learning rate, który skutkuje nie uczeniem się sieci z funkcją błędu 'MSE'.

Zostały również przeprowadzone eksperymenty porównujące 'MSE' oraz 'MAE' ze współczynnikiem uczenia 10 razy mniejszym niż na powyższych testach. Widać na nich, że błąd 'MAE' w miarę uczenia maleje, ale nadal jest o wiele gorszy niż błąd 'MSE'.



Rysunek 22: Porównanie błędu 'MSE' oraz 'MAE' przy obniżonym współczynniku uczenia

5.4 Eksperyment na zbiorze MNIST

Kolejnym eksperymentem było stworzenie architektury oraz dobranie takich hiperparametrów, aby osiągnąć powyżej 97% na zbiorze MNIST. Przeprowadziliśmy proces treningu na następujących parametrach oraz hiperparametrach:

- `net_structure`: [32, 16, 10],
- `learning_rate`: 0.01,
- `loss_name`: CROSS_ENTROPY,
- `activ_func_name`: SIGMOID,
- `epochs`: 2000

Do wczytania zbioru MNIST użyto biblioteki scikit-learn:

```
from sklearn import datasets
datasets.load_digits()
```

[INFO] evaluating network...				
	precision	recall	f1-score	support
0	1.0000	1.0000	1.0000	43
1	1.0000	1.0000	1.0000	37
2	0.9737	0.9737	0.9737	38
3	0.9362	0.9565	0.9462	46
4	0.9821	1.0000	0.9910	55
5	0.9667	0.9831	0.9748	59
6	0.9778	0.9778	0.9778	45
7	1.0000	0.9756	0.9877	41
8	0.9737	0.9737	0.9737	38
9	0.9783	0.9375	0.9574	48
accuracy			0.9778	450
macro avg		0.9788	0.9778	450
weighted avg		0.9779	0.9778	450

Rysunek 23: Wyniki na zbiorze MNIST

Jak widać na rysunku 23, udało się osiągnąć na tym zbiorze accuracy równe 0.9778.

Model sieci został zapisany w pliku `models/mnist/mnist.json`. Do powtórzenia testu wystarczy urochomić funkcję `main` bez żadnych argumentów.

6 Podsumowanie

Podczas projektu mieliśmy okazję do dogłębniego zrozumienia działania sieci neuronowej od środka. Na przykładzie zadania regresji oraz klasyfikacji zostały przedstawione rezultaty sieci napisanych od zera. W trakcie przygotowywania eksperymentów odkryliśmy ciekawe własności różnych funkcji aktywacji oraz funkcji strat. Dzięki wizualizacji różnych atrybutów sieci łatwiej było nam zrozumieć proces przebiegu treningu.

Literatura

- [1] Wikipedia - Artificial Neural Network, https://en.wikipedia.org/wiki/Artificial_neural_network.
- [2] Wikipedia - Sieć neuronowa, https://pl.wikipedia.org/wiki/Sieć_neuronowa.
- [3] Wikipedia - Softmax function, https://en.wikipedia.org/wiki/Softmax_function.
- [4] StackOverflow - gradient w sieci neuronowej, [StackOverflow derivative of softmax function](#).
- [5] Paras Dahal, Softmax and Cross Entropy Loss, <https://www.parasdahal.com/softmax-crossentropy>.
- [6] Gianluca Malato, Which models require normalized data, <https://towardsdatascience.com/which-models-require-normalized-data-d85ca3c85388>.
- [7] github.com/jzliu-100/visualize-neural-network, <https://github.com/jzliu-100/visualize-neural-network>