Multi Layer Perceptron-sprawozdanie

Szymon Gut, 313361

April 2023

Spis treści

1 Wprowadzenie

Celem laboratoriów było zapoznanie się z budową oraz implementacją sieci neuronowych MLP. W ramach kolejnych prac domowych zaimplementowano klasę sieci neuronowej, zaimplementowano propagację wsteczną wraz z usprawnieniami jak uczenie z momentem oraz normalizację RMSProp, rozwiązano zadania klasyfikacji, zapoznano się oraz zaimplementowano rózne funkcje aktywacji: Relu, Sigmoid, Tanh, Softmax, Linear oraz zapoznano się z zjawiskiem przeuczania (ang. overfitting) i zaimplementowano metody do radzenia sobie z takimi sytuacjami: wczesne zatrzymywanie oraz regularyzacja.

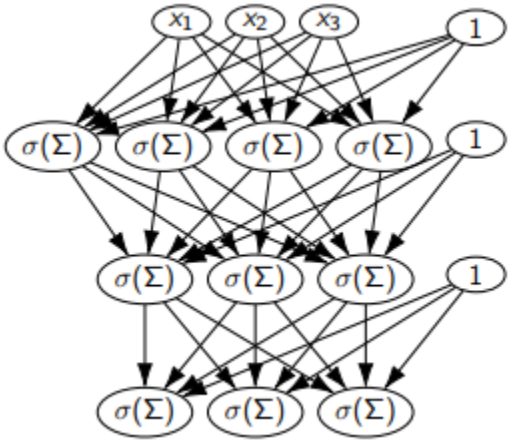
2 Implementacja klasy MLP

2.1 Część teoretyczna

Multi Layer Perceptron (MLP) to rodzaj sztucznej sieci neuronowej, która składa się z jednej lub więcej warstw ukrytych neuronów, a także z warstwy wejściowej i wyjściowej. Każda warstwa składa się z wielu neuronów, które przetwarzają informacje wejściowe i przekazują wyniki do kolejnej warstwy.

Neurony w warstwach ukrytych MLP przetwarzają informacje, które są dostarczane przez warstwę wejściową, wykorzystując wagi, które są dostosowywane podczas procesu uczenia. Każdy neuron wykonuje pewne obliczenia na danych wejściowych, a następnie przesyła wyniki do innych neuronów w kolejnej warstwie.

Ostatecznie, wynik działania sieci jest generowany przez neurony w warstwie wyjściowej, które przetwarzają informacje z ostatniej warstwy ukrytej.



Rysunek 1: Wizualizacja Perceptronu wielowarstwowego

3 Implementacja

Klasa NN:

```
class NN:
    def __init__(self, input_shape, neurons_num, activations, seed=123):
        self.batches = None
        self.y_test = None
        self.x_test = None
        self.y_train = None
        self.x_train = None
        self.delta_weights = None
        self.input_shape = input_shape
        self.layers_num = len(neurons_num)
        self.neurons_num = neurons_num
        self.activations = activations
```

```
        self.history = {'train': [], 'test': []}
        np.random.seed(seed)
        self._build()

    def _build(self):
        self.layers = []

        layer = Layer(shape=(self.input_shape[1], self.neurons_num[0]),
                      activation=self.activations[0])
        self.layers.append(layer)

        for i in range(1, self.layers_num):
            layer = Layer(
                shape=(self.layers[i - 1].shape[1], self.neurons_num[i]),
                activation=self.activations[i])
            self.layers.append(layer)
    def propagate_forward(self, x):
        for i in range(0, self.layers_num):
            x = self.layers[i].calculate(x)
            x = self.layers[i].activate(x)

        return x
```

Klasa Layer:

```
class Layer:
    def __init__(self, shape, activation=Sigmoid()):
        self.shape = shape
        self.activation = activation
        self._initialize_weights()
        self.weighted_input = None
        self.output = None

    def _initialize_weights(self, min_val=-1, max_val=1):
        self.weights = np.random.uniform(min_val, max_val, size=self.shape)
        self.biases = np.random.uniform(min_val, max_val, size=(self.shape[1], 1))

    def calculate(self, x):
        self.weighted_input = (self.weights.T @ x) + self.biases

        return self.weighted_input

    def activate(self, x):
        self.output = self.activation.calculate(x)
        return self.output
```

Klasa NN zawiera takie atrybuty jak wymiar danych treningowych, liczbę neuronów w warstwach ukrytych, funkcje aktywacji oraz random.seed umożliwiający replikowalność doświadczeń.

Klasa Layer natomiast odzwierciedla warstwy ukryte i jest wyposażona w funkcje do obliczania wartości wejściowych oraz wartości wyjściowych po wywołaniu funkcji aktywacji.

4 Backpropagacja

W ramach jednej z prac domowych dodano algorytm do backpropagacji w celu uczenia sieci neuronowej. Opiera się on na minimalizacji funkcji kosztu z wykorzystaniem optymalizacyjnej metody największego spadku.

4.1 Opis algorytmu

Oznaczenia:

- w_{ij}^k - waga pomiędzy węzłem j w warstwie ukrytej l^k, a węzłem i w warstwie ukrytej l^{k-1}
- b_i^k - bias dla węzła i w warstwie l^k
- $E(X, \theta)$ - funkcja błędu, definiująca błąd pomiędzy prawdziwą wartością naszej zmiennej objaśnianej $\vec{y_i}$, a wartością zwracaną przez sieć $\hat{\vec{y_i}}$

Trenowanie sieci przy użyciu algorytmu backpropagacji wymaga policzenia gradientu funkcji błędu $E(X, \theta)$ ze względu na wagi w_{ij}^k i bias b_i^k. Następnie każda iteracja algorytmu uaktualnia parametry naszej sieci (wagi oraz bias) zgodnie z formułą:

$$\theta^{t+1} = \theta^t - \alpha \frac{\partial E(X, \theta^t)}{\partial \theta},$$

gdzie α to parametr learning rate.

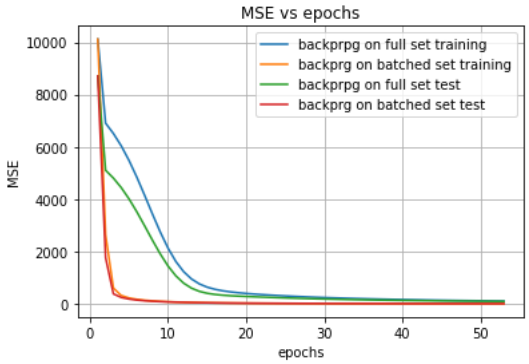
4.2 Doświadczenia

Sieć neuronowa z podstawowym algorytmem propagacji wstecznej została wytrenowana na kilku zbiorach treningowych. Na każdym zbiorze sieci były trenowane dwiema metodami: mini-batch oraz uaktualnianie wag po prezentacji wszystkich wzorców. Wyniki doświadczeń widoczne w tabeli.

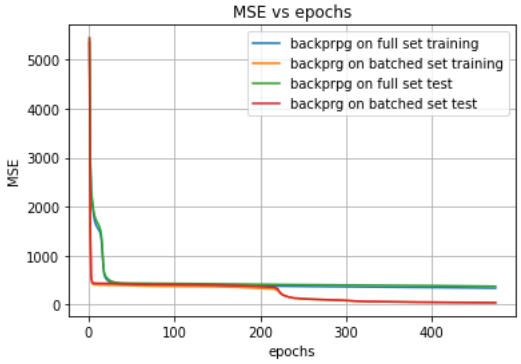
Zbiór danych	MSE na zbiorze treningowym	MSE na zbiorze testowym	Liczba epok
Square-Simple	3.3	3.94	52 000
Mutlimodal-Large	39.97	37.76	473 000
Steps-Small	0.21	4.1	365 000

A krzywe uczenia dla każdego ze zbiorów widoczne będą poniżej:

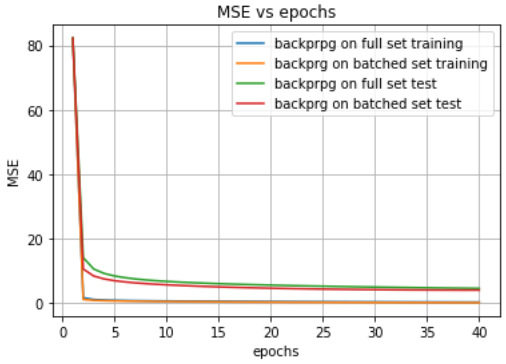
Square-Simple:



Multimodal-Large:



Steps-Small:



5 Uczenie z momentem oraz RMSProp

W kolejnym etapie pracy dodano dwie metody opymalizacji w celu przyśpieszenia uczenia, a następnie sprawdzono ich skuteczność. Metody zauważalnie przyśpieszyły proces uczenia, a rezultaty zostały przedstawione w dalszej częsci tej sekcji.

5.1 Opis Algorytmu Momentum

Uczenie z momentem polega na wprowadzeniu do procesu aktualizacji wag momentum, czyli pamięci poprzednich zmian wag. W każdej iteracji, momentum bierze pod uwagę poprzednią zmianę wag i dodaje ją do bieżącej zmiany wag.

Dzięki temu, w przypadku kiedy gradient funkcji kosztu ma stały kierunek przez wiele iteracji, momentum pozwala przyspieszyć proces uczenia poprzez kontynuowanie zmian wag w tym samym kierunku.

Opis algorytmu

- α - krok uczenia
- $\lambda \in (0, 1)$ - współczynnik wygaszania momentu

$\theta \longleftarrow$ Inicjuj Losowo
Momentum $\longleftarrow$ [0, 0, ... 0]
while ¬StopCondition **do**
    $\Delta\theta \longleftarrow$ [0, 0, ... 0]
    for(X,Y) $\in$ Zbiór uczący **do**
        $\hat{Y} \longleftarrow$ Network(θ, X)
        $\Delta\theta \longleftarrow \Delta\theta - \Delta$Network($\theta$, X)
    Momentum $\longleftarrow \Delta\theta$ + Momentum * λ
    $\theta \longleftarrow \theta + \alpha$ * Momentum

5.2 RMSProp

Normalizacja RMSProp polega na zastosowaniu dodatkowego etapu normalizacji gradientów. W metodzie tej obliczamy średnią kwadratową poprzednich gradientów oraz aktualny gradient, a następnie dzielimy aktualny gradient przez pierwiastek z sumy średniej kwadratowej gradientów. W ten sposób, jeśli gradienty mają bardzo duże wartości, to normalizacja powoduje zmniejszenie ich wpływu na aktualizację wag.

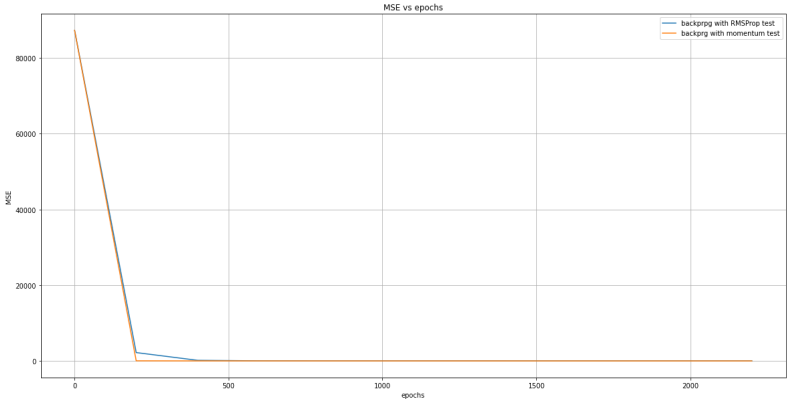
Normalizacja RMSProp może przyspieszyć proces uczenia, ponieważ poprawia stabilność procesu aktualizacji wag, zmniejszając ryzyko wystąpienia "wybuchającego gradientu" (ang. exploding gradient) lub "zanikającego gradientu" (ang. vanishing gradient).

Opis algorytmu

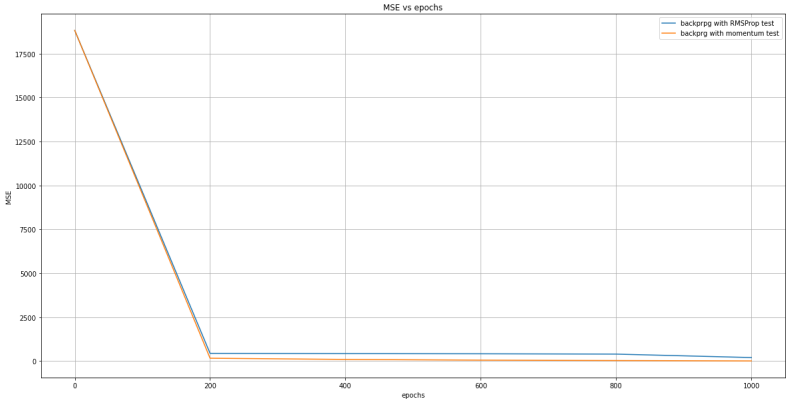
- α - krok uczenia
- β - współczynnik wygaszania

$\theta \longleftarrow$ Inicjuj Losowo
$E[g^2] \longleftarrow$ [0,0,... 0]
while ¬Stop Condition **do**
    g $\longleftarrow$0
    for(X,Y) $\in$ Zbiór uczący **do**
        $\hat{Y} \longleftarrow$ Network(θ, X)
        g $\longleftarrow$g + ΔNetwork(θ,X)
    $E[g^2] \longleftarrow \beta E[g^2] + (1-\beta)g^2$
    $\theta \longleftarrow \theta - \alpha[\frac{g_i}{\sqrt{E[g^2]_i}}]_{\forall i}$

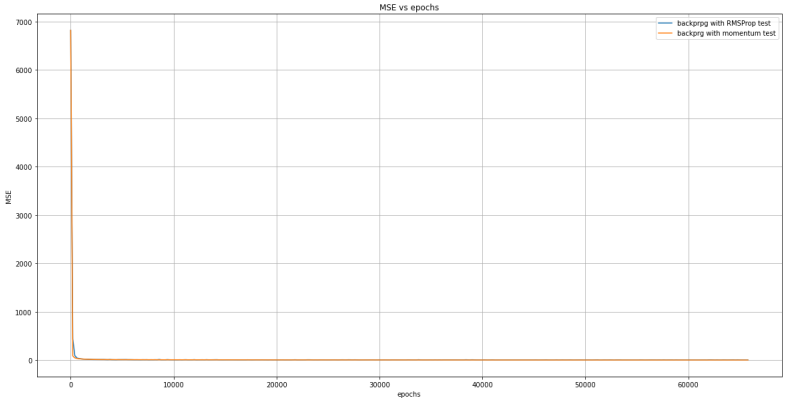
Square-Simple



Multimodal-Large



Steps-Small



5.3 Wyniki eksperymentu

Wyniki na przetestowanych zbiorach widoczne są w poniższej tabeli. Zostały tam zapisane najlepsze rezultaty biorąc pod uwagę różne architektury oraz optymalizacje RMSProp oraz z momentem.

Zbiór danych	MSE na zbiorze treningowym	MSE na zbiorze testowym	Liczba epok	Rodzaj optymalizacji
Square-Simple	0.68	0.98	2200	Momentum
Mutlimodal-Large	7.57	3.03	1000	Momentum
Steps-Small	3.0	3.0	65 800	Momentum

5.4 Porównanie wyników z uczeniem bez optymalizacji

Wyniki zestawione w poniższej tabeli pokazują różnice w szybkości zbieżności krzywej uczenia do oczekiwanego optimum między zwykłym algorytmem propagacji wstecz oraz propagacji wstecz z momentem.

Zbiór danych	MSE dla uczenia z momentem	Liczba epok dla uczenia z momentem	MSE bez optymalizacji	Liczba epok bez optymalizacji
Square-Simple	0.98	2200	3.94	52 000
Multimodal-Large	3.03	1000	37.76	473 000
Steps-Small	3.0	65 800	4.1	365 000

Tabela 1: Miara MSE została podana ze względu na zbiór testowy

Otrzymane wyniki potwierdzają znaczne przyśpieszenie procedury uczenia sieci neuronowej. Dla zbioru Square Simple algorytm zbiegł około 23 krotnie szybciej, dla zbioru Multimodal 473 krotnie szybciej oraz dla zbioru Steps-Small 5.5 krotnie szybciej.

6 Rozwiązywanie zadania klasyfikacji

W dalszej części przetestowano sieć neuronową na zadaniach klasyfikacji. W tym celu zaimplementowano funkcję Softmax w celu dodania jej do warstwy wyjściowej. Funkcja Softmax to funkcja, której zadaniem jest zamiana wartości wyjściowych neuronów w formę rozkładu prawdopodobieństwa, gdzie każda wartość wyjściowa reprezentuje prawdopodobieństwo przynależności danego przykładu do danej klasy.

Jako, że funkcja softmax jest zależna od wszystkich wyjściowych neuronów, jej implementacja trochę się różniła od implementacji pochodnych dla pozostałych funkcji aktywacji. Implementacja jest widoczna poniżej.

```
class Softmax(ActivationFunction):

    def calculate(self, x):
        if x.shape[1] == 1:
            return np.exp(x) / np.sum(np.exp(x))
        if x.shape[1] > 1:
            return np.exp(x) / (np.sum(np.exp(x), axis=1).reshape(
                (x.shape[0], 1)) @ np.ones((1, x.shape[1])))

    def derivative(self, x):
        s = self.calculate(x).T
        a = np.eye(s.shape[-1])
        temp1 = np.zeros((s.shape[0], s.shape[1], s.shape[1]), dtype=np.float32)
        temp2 = np.zeros((s.shape[0], s.shape[1], s.shape[1]), dtype=np.float32)
        temp1 = np.einsum('ij,jk->ijk', s, a)
        temp2 = np.einsum('ij,ik->ijk', s, s)
        return (temp1 - temp2).T

    def error(self, error, derivative):
        derivative = derivative.reshape(derivative.shape[0], derivative.shape[1])
        return (derivative @ error).reshape((-1, 1))
```

6.1 Doświadczenia

W celu sprawdzenia skuteczności funkcji Softmax sprawdzono ją w porównaniu do funkcji liniowej w warstwie wyjściowej do zadania klasyfikacji. Eksperymenty przeprowadzono na trzech zbiorach, a dla każdego z nich zbudowano po dwie architektury: jedna z aktywacją Softmax, druga z funkcją liniową w warstwie wyjściowej.
Wyniki eksperymentów widoczne poniżej:

Zbiór danych	**F-score dla Softmax**	**Liczba epok dla Softmax**	**F-score dla Linear**	**Liczba epok dla Linear**
Rings3-Regular	0.93	340	0.89	350
Easy	0.99	19	0.99	99
Xor3	0.97	330	0.97	270

Tabela 2: Miara F-score została podana ze względu na zbiór testowy

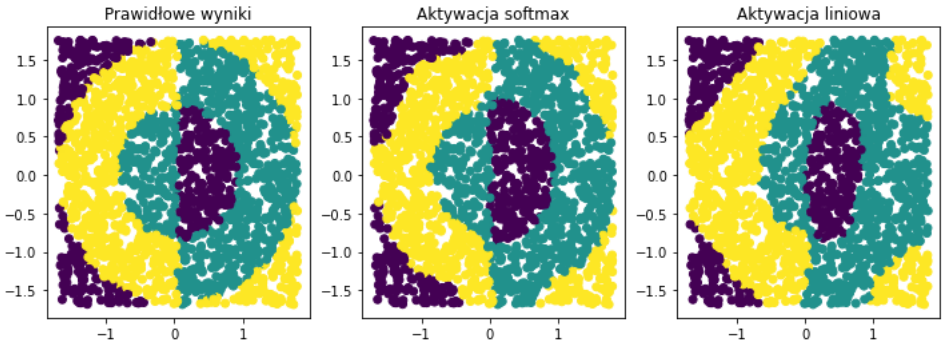
Porównania wizualne dla każdego ze zbiorów:

Rings3-regular:

(a) Rings3 data set with Softmax activation

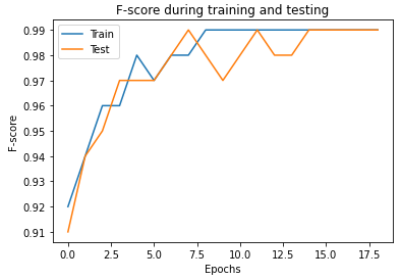
(b) Rings3 data set with Linear activation

Rysunek 2: Porównanie procesu uczenia

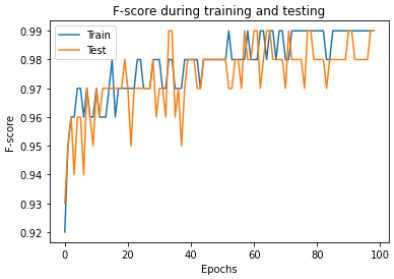


Rysunek 3: Wizualizacja efektów klasyfikacji

Easy:

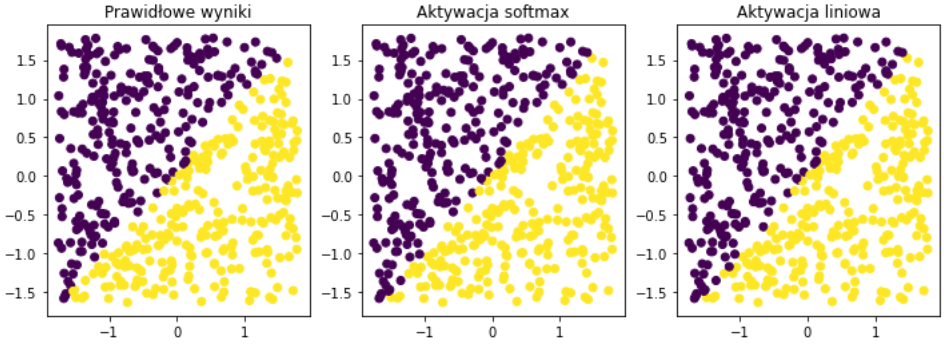


(a) Easy data set with Softmax activation



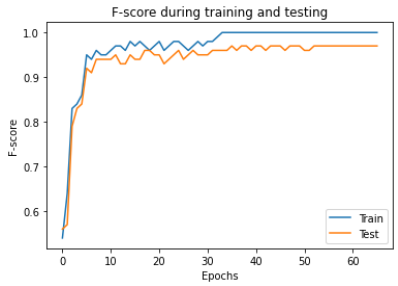
(b) Easy data set with Linear activation

Rysunek 4: Porównanie procesu uczenia



Rysunek 5: Wizualizacja efektów klasyfikacji

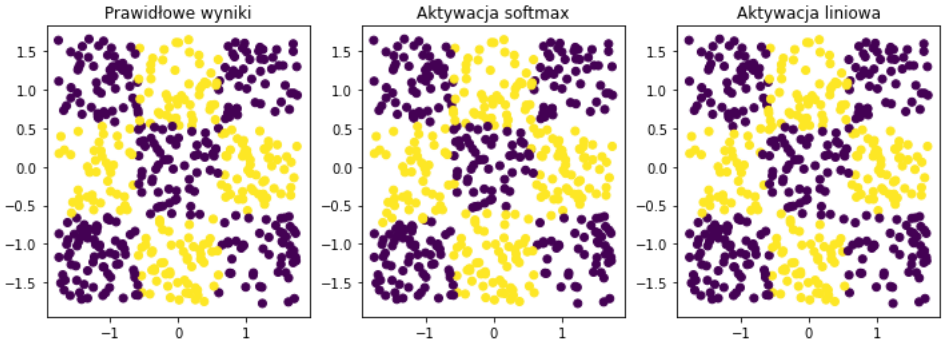
Xor3:



(a) Xor3 data set with Softmax activation

(b) Xor3 data set with Linear activation

Rysunek 6: Porównanie procesu uczenia



Rysunek 7: Wizualizacja efektów klasyfikacji

7 Dodanie innych funkcji aktywacji

W dalszej części rozwijania architektury zostały zaimplementowane pozostałe funkcje aktywacji tj. Tanh, Relu, LeakyRelu. Funkcja Softmax, Sigmoid oraz Liniowa były zaimplementowane już wcześniej.

7.1 Implementacja

```
class ActivationFunction:
    def calculate(self, x):
        pass

    def derivative(self, x):
        pass

    def error(self, error, derivative):
        pass


class Relu(ActivationFunction):

    def calculate(self, x):
        return np.where(x > 0, x, 0)

    def derivative(self, x):
        return np.where(x > 0, 1, 0)

    def error(self, error, derivative):
        return np.multiply(error, derivative)


class LeakyRelu(ActivationFunction):
    def calculate(self, x):
        return max(0.1 * x, x)

    def derivative(self, x, alpha=0.01):
        return alpha if x < 0 else 1

    def error(self, error, derivative):
        return np.multiply(error, derivative)


class Tanh(ActivationFunction):

    def calculate(self, x):
        counter = np.exp(x) - np.exp(-x)
        denominator = np.exp(x) + np.exp(-x)
        return counter / denominator

    def derivative(self, x):
        return 1 - np.square(self.calculate(x))

    def error(self, error, derivative):
        return np.multiply(error, derivative)
```

```
class Sigmoid(ActivationFunction):

    def calculate(self, x):
        return 1 / (1 + np.exp(-x))

    def derivative(self, x):
        return np.exp(-x) / np.square((1 + np.exp(-x)))

    def error(self, error, derivative):
        return np.multiply(error, derivative)


class Linear(ActivationFunction):

    def calculate(self, x):
        return x

    def derivative(self, x):
        return np.ones(shape=(x.shape[-1], 1))

    def error(self, error, derivative):
        return np.multiply(error, derivative)


class Softmax(ActivationFunction):

    def calculate(self, x):
        if x.shape[1] == 1:
            return np.exp(x) / np.sum(np.exp(x))
        if x.shape[1] > 1:
            return np.exp(x) / (np.sum(np.exp(x), axis=1).reshape(
                (x.shape[0], 1)) @ np.ones((1, x.shape[1])))

    def derivative(self, x):
        s = self.calculate(x).T
        a = np.eye(s.shape[-1])
        temp1 = np.zeros((s.shape[0], s.shape[1], s.shape[1]), dtype=np.float32)
        temp2 = np.zeros((s.shape[0], s.shape[1], s.shape[1]), dtype=np.float32)
        temp1 = np.einsum('ij,jk->ijk', s, a)
        temp2 = np.einsum('ij,ik->ijk', s, s)
        return (temp1 - temp2).T

    def error(self, error, derivative):
        derivative = derivative.reshape(derivative.shape[0], derivative.shape[1])
        return (derivative @ error).reshape((-1, 1))
```

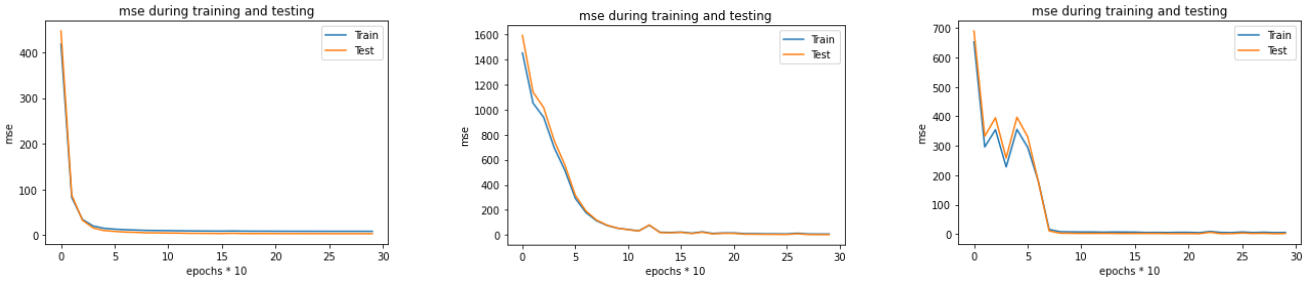
7.2 Doświadczenia

W pierwszym etapie doświadczeń zbudowano po 3 architektury dla każdej funkcji aktywacji (w każdej architekturze występowała tylko dana funkcja aktywacji) i przetestowano je na zbiorze **Multimodal-Large** z zadaniem regresji. Architektury budowane dla każdej funkcji aktywacji różniły się między sobą ilością warstw ukrytych oraz liczbą neuronów.

7.2.1 Sigmoid

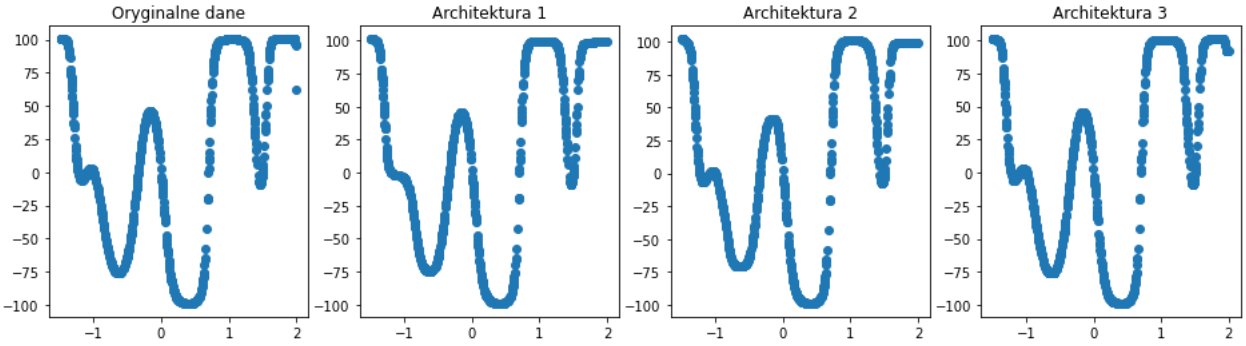
Rodzaj architektury	Funkcja aktywacji	Learning rate	Batch size	MSE test	Liczba Epok
[1, 32, 1]	Sigmoid	0.03	64	3.26	300
[1, 32, 32, 1]	Sigmoid	0.001	16	3.68	300
[1, 16, 32, 16, 1]	Sigmoid	0.003	6	1.86	330

Tabela 3: Miara MSE została podana ze względu na zbiór testowy



(a) Architektura 1, 32, 1 (b) Architektura 1, 32, 32, 1 (c) Architektura 1, 16, 32, 16, 1

Rysunek 8: Porównanie procesu uczenia dla 3 architektur

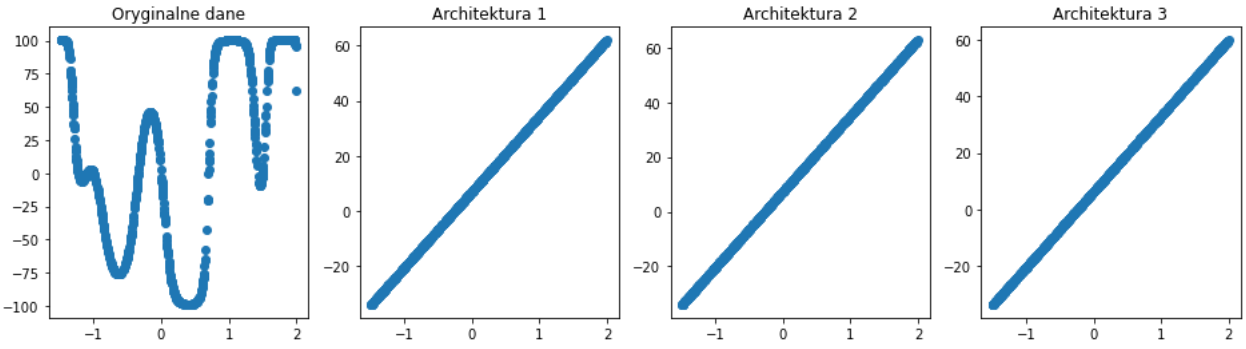


Rysunek 9: Wizualizacja efektów regresji

7.2.2 Linear

Rodzaj architektury	Funkcja aktywacji	Learning rate	Batch size	MSE test	Liczba Epok
[1, 32, 1]	Linear	0.00005	16	4433.7	200
[1, 32, 32, 1]	Linear	0.0005	16	4433.58	100
[1, 16, 32, 16, 1]	Linear	0.00001	32	4435.8	100

Tabela 4: Miara MSE została podana ze względu na zbiór testowy

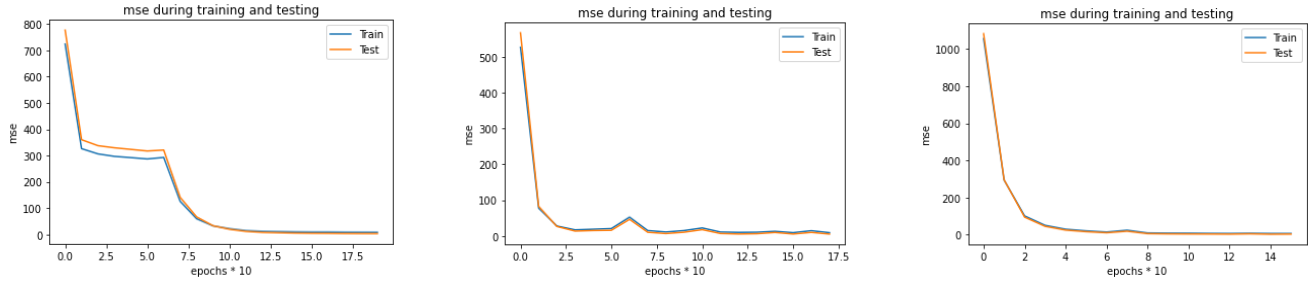


Rysunek 10: Wizualizacja efektów regresji

7.2.3 Tanh

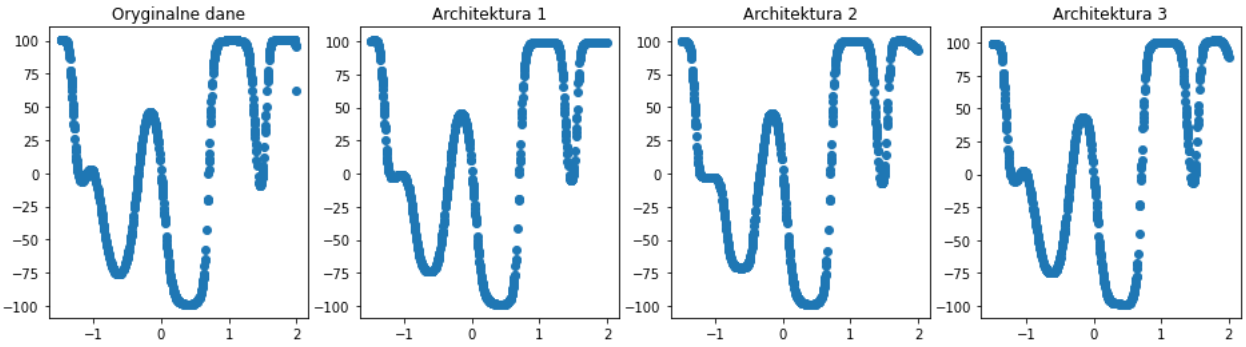
Rodzaj architektury	Funkcja aktywacji	Learning rate	Batch size	MSE test	Liczba Epok
[1, 32, 1]	Tanh	0.0005	6	3.26	200
[1, 32, 32, 1]	Tanh	0.001	32	4.98	180
[1, 16, 32, 16, 1]	Tanh	0.00003	4	2.64	160

Tabela 5: Miara MSE została podana ze względu na zbiór testowy



(a) Architektura 1, 32, 1 (b) Architektura 1, 32, 32, 1 (c) Architektura 1, 16, 32, 16, 1

Rysunek 11: Porównanie procesu uczenia dla 3 architektur

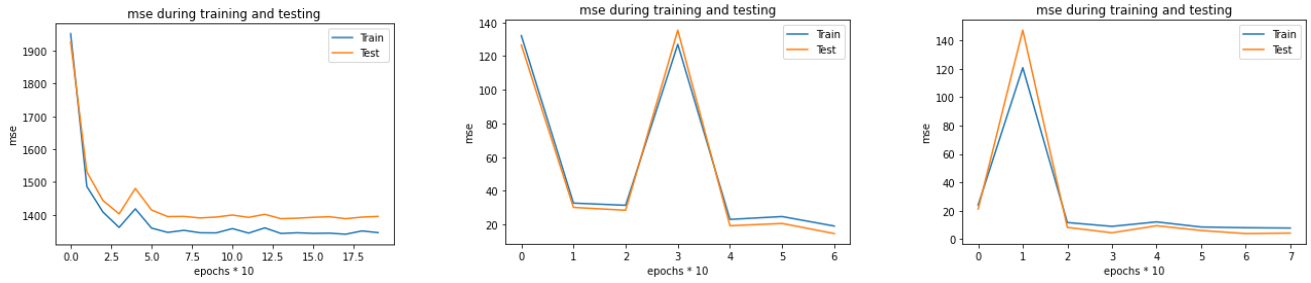


Rysunek 12: Wizualizacja efektów regresji

7.2.4 Relu

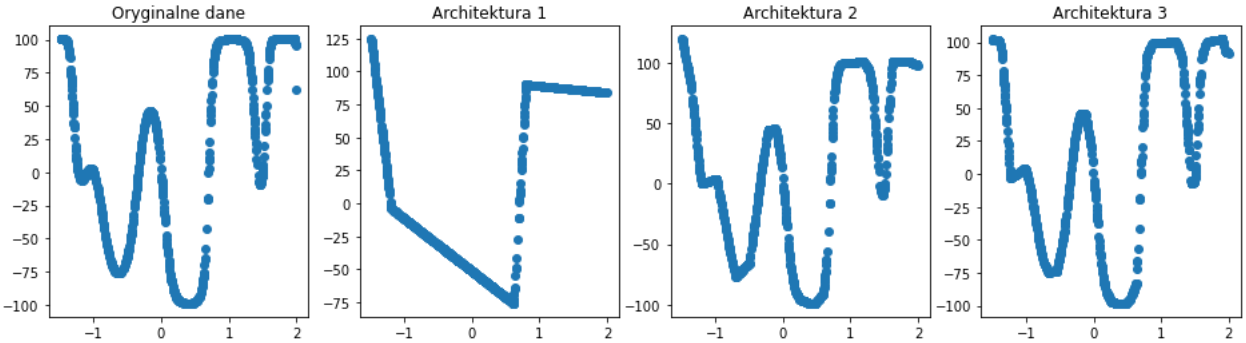
Rodzaj architektury	Funkcja aktywacji	Learning rate	Batch size	MSE test	Liczba Epok
[1, 32, 1]	Relu	0.0001	4	1395.43	200
[1, 32, 32, 1]	Relu	0.0003	4	14.66	70
[1, 16, 32, 16, 1]	Relu	0.0001	4	4.31	150

Tabela 6: Miara MSE została podana ze względu na zbiór testowy



(a) Architektura 1, 32, 1 (b) Architektura 1, 32, 32, 1 (c) Architektura 1, 16, 32, 16, 1

Rysunek 13: Porównanie procesu uczenia dla 3 architektur



Rysunek 14: Wizualizacja efektów regresji

7.2.5 Rezultaty

Funkcja aktywacji	MSE Architektura1	MSE Architektura2	MSE Architektura3
Sigmoid	3.256278	3.682077	1.857315
Linear	4433.696630	4433.581323	4435.800171
Tanh	3.264200	4.979507	2.635586
Relu	1395.430509	14.657854	4.307024

Tabela 7: Miara MSE została podana ze względu na zbiór testowy

Na zbiorze Multimodal-Large najlepszą funkcją okazała się funkcja Sigmoid oraz Tanh. Funkcja Relu też przyniosła pozytywne rezultaty jednak zauważalne jest, że przynosi ona efekty dopiero w większych architekturach (przyniosła zdecydowanie słabsze rezultaty dla architektury1, to znaczy architektury składającej się z jednej warstwy ukrytej). Funkcja liniowa zdecydowanie nie powinna być używana do regresji nieliniowej, gdyż jak można zaobserwować w wizualizacjach oraz patrząc na otrzymane wyniki architektura estymuje prostą minimalizującą MSE, co jest użyteczne tylko w regresji z zależnością liniową między parametrami.

7.3 Testowanie

Następnie przetestowaliśmy Architekturę3 na 3 zbiorach danych:

regresja:

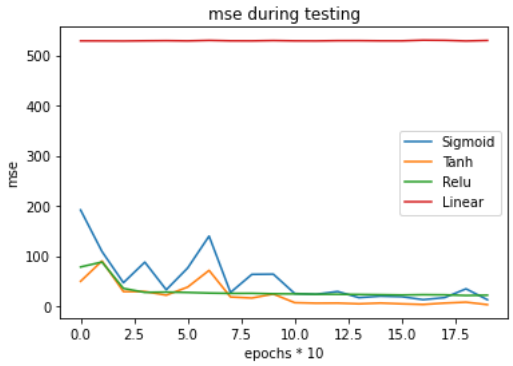
- Steps-Large

klasyfikacja:

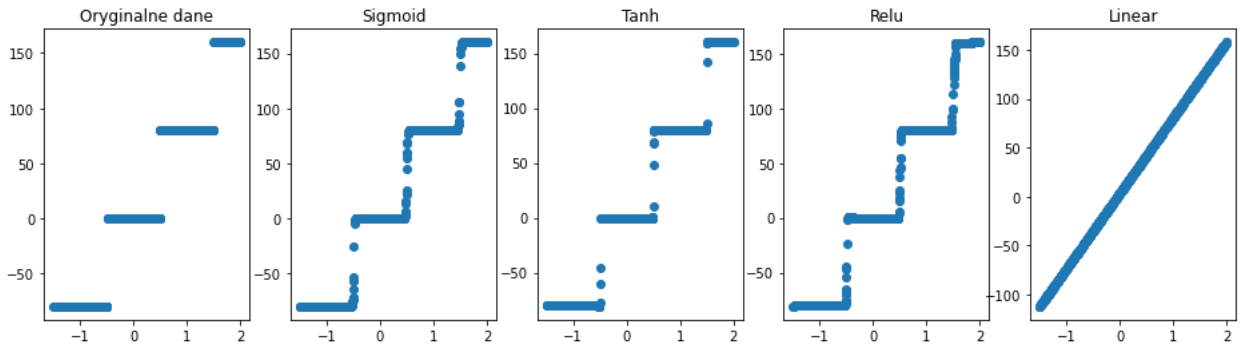
- Rings5-regular
- Rings3-regular

W architekturze 3 zmieniano funkcje aktywacji we wszystkich warstwach i patrzono jak wpływa to na efektywność uczenia. Wyniki eksperymentu przedstawione poniżej.

7.3.1 Zbiór Steps-Large



Rysunek 15: Wizualizacja krzywej uczenia

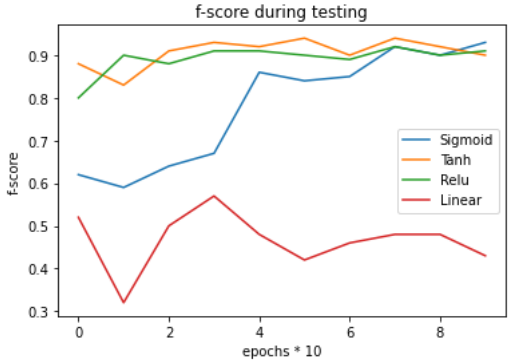


Rysunek 16: Wizualizacja efektów

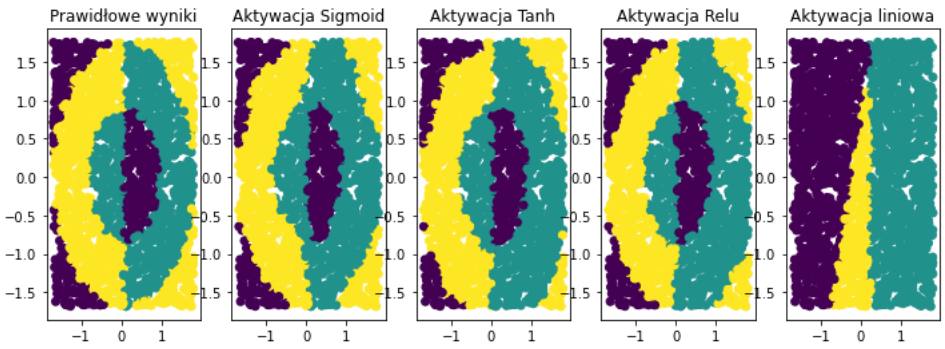
Funkcja aktywacji	Sigmoid	Linear	Tanh	Relu
MSE train	25.78	518.77	3.74	24.57
MSE test	13.1	529.45	3.32	22.18

Tabela 8: Uzyskane metryki

7.3.2 Zbiór rings3-regular



Rysunek 17: Wizualizacja krzywej uczenia

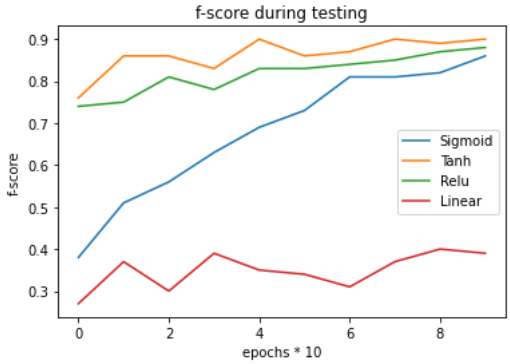


Rysunek 18: Wizualizacja efektów

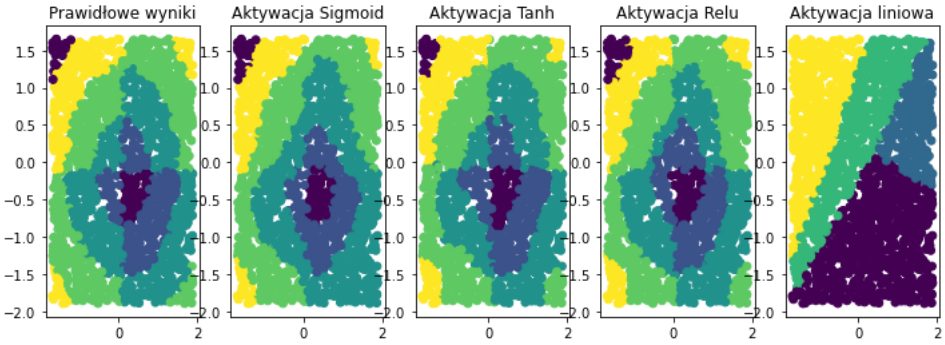
Funkcja aktywacji	Sigmoid	Linear	Tanh	Relu
F-score train	0.94	0.43	0.92	0.93
F-score test	0.93	0.43	0.90	0.91

Tabela 9: Uzyskane metryki

7.3.3 Zbiór rings5-regular



Rysunek 19: Wizualizacja krzywej uczenia



Rysunek 20: Wizualizacja efektów

Funkcja aktywacji	Sigmoid	Linear	Tanh	Relu
F-score train	0.89	0.42	0.94	0.89
F-score test	0.86	0.39	0.90	0.88

Tabela 10: Uzyskane metryki

7.4 Posumowanie

Funkcja Sigmoid i Tanh najlepiej sprawdziły się zarówno w zadaniach regresji, jak i klasyfikacji. Funkcja Relu dawała minimalnie gorsze rezultaty, jednak z zwiększaniem liczby neuronów oraz warstw ukrytych otrzymywaliśmy

coraz lepsze wyniki. Funkcja liniowa nie sprawdziła się we wszystkich eksperymentach oraz w fazie testowania. Dzieje się tak, gdyż stosując funkcję liniową jako funkcję aktywacji w warstwach ukrytych powodujemy zależność liniową pomiędzy kolejnymi warstwami, przez co nie jesteśmy w stanie estymować zależności nieliniowych.

8 Zjawisko przeuczenia

Przeuczenie (ang. overfitting) to zjawisko, w którym model uczenia maszynowego osiąga wysoką skuteczność w dopasowywaniu się do danych treningowych, ale niską skuteczność w generalizacji na nowe dane. Jest to częsty problem w uczeniu maszynowym i wymaga stosowania odpowiednich technik regularyzacji i walidacji modelu. W tym celu stosuję się różne formy regularyzacji takie jak regularyzacja L1, czy L2. Aby kontrolować procedurę uczenia stosuje się również algorytmy early stop'u tj. wczesnego zatrzymywania polegające na zatrzymaniu procedury uczenia, gdy błąd na danych testowych wzrasta lub utrzymuje się przez dłuższy czas na tym samym poziomie.

8.1 Regularyzacja L1

Regularyzacja L1 to technika regularyzacji stosowana w sieciach neuronowych, która pomaga zapobiegać nadmiernemu dopasowaniu modelu do danych treningowych poprzez wprowadzenie kary za duże wartości wag.

W przypadku regularyzacji L1, dodawana jest dodatkowa kara do funkcji kosztu, równa sumie wartości bezwzględnych wszystkich wag modelu. To oznacza, że im większa wartość wag, tym większa kara zostanie nałożona. Dzięki temu, wagi, które nie są potrzebne do predykcji, ale mają dużą wartość, będą karane i stopniowo zanikną. Termin regularyzacji wyraża się następująco:

$$\Omega(W) = \|W\|_1 = \sum_i \sum_j w_{ij} \qquad \text{(Termin regularyzacji)}$$

$$\hat{L}(W) = \alpha\|W\|_1 + L(W) \qquad \text{(Funkcja straty przy regularyzacji L1)}$$

$$\nabla_w \hat{L}(W) = \alpha sign(W) + \nabla_w L(W) \qquad \text{(Gradient funkcji straty przy regularyzacji L1)}$$

Zastosowanie regularyzacji L1 ma kilka korzyści. Po pierwsze, pomaga zmniejszyć rozmiar modelu, co może przyspieszyć uczenie się i zmniejszyć ryzyko nadmiernego dopasowania. Po drugie, pozwala na identyfikację ważnych funkcji wejściowych, które mają wpływ na wyniki predykcji. Ponadto, regularyzacja L1 może być przydatna do odkrywania ciekawych wzorców w danych, ponieważ wymusza rzadkość wagi.

Jednym z wad regularyzacji L1 jest to, że w niektórych przypadkach może ona prowadzić do zbyt agresywnego wyzerowania wag, co może prowadzić do utraty informacji. Dlatego też, często stosuje się bardziej złożone techniki regularyzacji, które uwzględniają różne rodzaje kar za duże wagi i jednocześnie minimalizują stratę informacji.

8.2 Regularyzacja L2

Regularyzacja L2 (inaczej znana jako "ridge" lub "Tikhonov regularization") polega na dodaniu do funkcji kosztu sieci neuronowej kary proporcjonalnej do kwadratu normy L2 wag modelu. Innymi słowy, w każdej iteracji treningowej, funkcja kosztu jest zwiększana przez sumę kwadratów wszystkich wag w modelu.

$$\Omega(W) = \|W\|_2^2 = \sum_i \sum_j w_{ij}^2 \qquad \text{(Termin regularyzacji)}$$

$$\hat{L}(W) = \frac{\alpha}{2}\|W\|_2^2 + L(W) \qquad \text{(Funkcja straty przy regularyzacji L1)}$$

$$\nabla_w \hat{L}(W) = \alpha W + \nabla_w L(W) \qquad \text{(Gradient funkcji straty przy regularyzacji L1)}$$

$$W_{new} = W_{old} - \epsilon(\alpha W_{old} + \nabla_w L(W_{old})) \qquad \text{(Uaktualnianie wag w propagacji wstecz)}$$

Regularyzacja L2 działa poprzez penalizację dużych wartości wag, co zmniejsza ich znaczenie i pomaga uniknąć nadmiernej złożoności modelu. W efekcie, model staje się bardziej ogólny i lepiej generalizuje dla nowych danych.

8.3 Doświadczenia

W ramach jednej z prac domowych została zaimplementowana regularyzacja L1, L2 oraz mechanizm wczesnego zatrzymywania. Następnie dla wybranej architektury przetestowano jej skuteczność na 4 zbiorach:

regresja:

- multimodal-sparse

klasyfikacja:

- rings5-sparse
- rings3-balance
- xor3-balance

Wyniki porównano dla architektury bez żadnej regularyzacji oraz mechanizmu wczesnego zatrzymywania oraz dla tej, która te mechanizmy posiada. Dla każdej architektury stworzono po 5 sieci z tymi samymi parametrami, zmieniając tylko random.seed, aby w fazie końcowej uśrednić wyniki dla każdej z architektur.
Implementacja funkcji wykonującej tą cross-validację:

```
def cv_network(seeds=[123, 1, 2, 23, 42], build_args=None, fit_args=None):
    scores_test = []
    scores_train = []
    nns  = []
    for s in seeds:
        nn = NN(**build_args, seed=s)
        last_fa = None
        for fa in fit_args:
            nn.fit(**fa)
            last_fa = fa
            nns.append(nn)
            scores_test.append(last_fa['metric'].calculate(last_fa['y_test'],
            nn.predict(last_fa['x_test'])))
            scores_train.append(last_fa['metric'].calculate(last_fa['y_train'],
            nn.predict(last_fa['x_train'])))

    return scores_train, scores_test, nns
```

8.3.1 Zbiór Multimodal-Sparse

Na tym zbiorze danych rozważono dwie architektury.

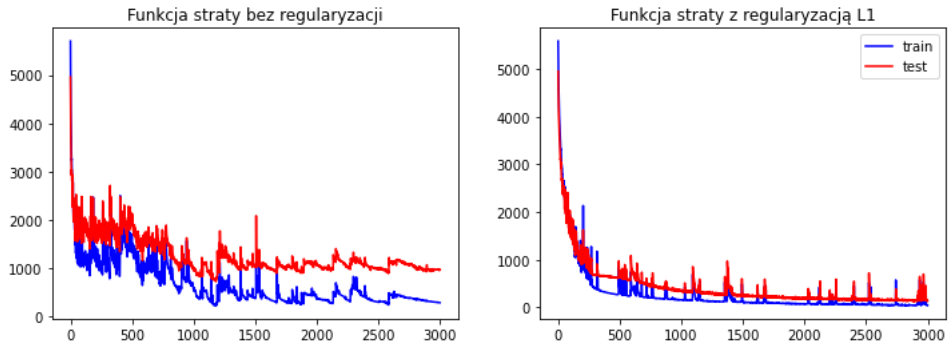
Bez regularyzacji oraz algorytmu wczesnego zatrzymywania:

```
ms_no_reg_build = {'input_shape': ms_x_train.shape, 'neurons_num': [128, 128, 128, 1],
                   'activations': [Tanh(), Tanh(), Tanh(), Linear()]}
ms_no_reg_fit = [{'x_train': ms_x_train, 'y_train': ms_y_train, 'batch_size': 32,
                  'n_epochs': 3000, 'learning_rate': 0.003, 'x_test': ms_x_test,
                  'y_test': ms_y_test, 'loss': Mse(), 'metric': Mse(), 'verbose_step': 1,
                  'regularization_rate': 0, 'stop_action' : False}]

results_ms_train, results_ms_test, ms_nns = cv_network(build_args=ms_no_reg_build,
fit_args=ms_no_reg_fit)
```

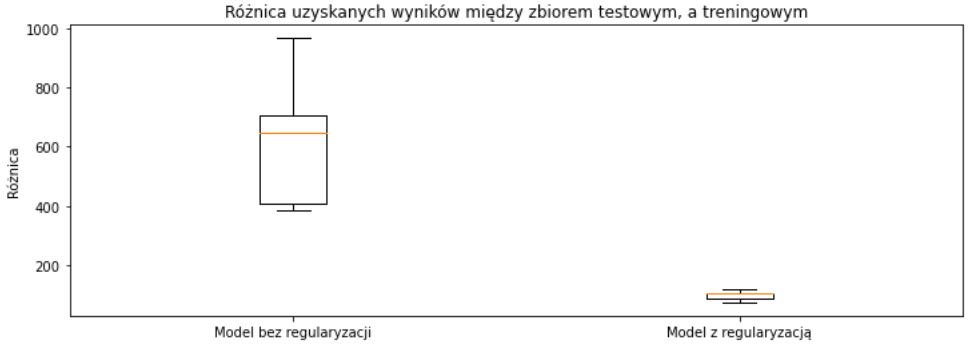
Z regularyzacją L1 oraz algorytmem wczesnego zatrzymywania:

```
ms_l1_build = {'input_shape': ms_x_train.shape, 'neurons_num': [128, 128, 128, 1],
               'activations': [Tanh(), Tanh(), Tanh(), Linear()]}
ms_l1_fit = [{'x_train': ms_x_train, 'y_train': ms_y_train, 'batch_size': 32,
              'n_epochs': 3000, 'learning_rate': 0.0003, 'x_test': ms_x_test,
              'y_test': ms_y_test, 'loss': Mse(), 'metric': Mse(), 'verbose_step': 1,
              'regularization_rate': 0.2, 'stop_action' : True, 'stop_treshold' : 50,
              'patience':100}]

results_ms_train_l1, results_ms_test_l1, ms_nns_l1 = cv_network(build_args=ms_l1_build,
fit_args=ms_l1_fit)
```

Po wytrenowaniu dla każdej architektury po 5 modeli metodą cross-validacji i uśrednieniu wyników sporządzono wykres porównawczy.



Rysunek 21: Wizualizacja funkcji straty dla modelu z regularyzacją oraz bez regularyzacji



Rysunek 22: Różnica pomiędzy MSE uzyskanym na zbiorze testowym oraz treningowym dla dwóch architektur

8.3.2 Zbiór Rings5-Sparse

Na tym zbiorze rozważono dwie architektury.

Bez regularyzacji oraz algorytmu wczesnego zatrzymywania:

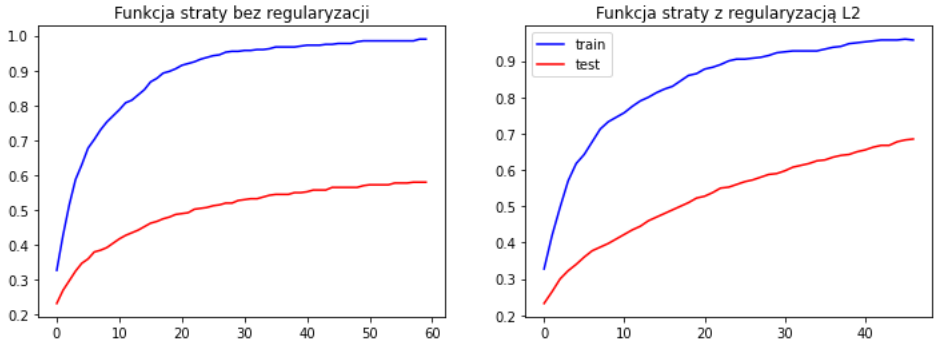
```
r5_no_reg_build = {'input_shape': r5_x_train.shape, 'neurons_num': [180,180,180,180,5],
                   'activations': [Tanh(), Tanh(), Tanh(), Tanh(), Softmax()]}
r5_no_reg_fit = [{'x_train': r5_x_train, 'y_train': r5_y_train, 'batch_size': 32,
                  'n_epochs': 600, 'learning_rate': 0.0001, 'x_test': r5_x_test,
                  'y_test': r5_y_test, 'loss': Cross_entropy(), 'metric': F_score(),
                  'verbose_step': 10, 'regularization_rate': 0, 'stop_action' : False}]
```

```
results_r5_train , results_r5_test , r5_nns = cv_network(build_args=r5_no_reg_build ,
fit_args=r5_no_reg_fit)
```

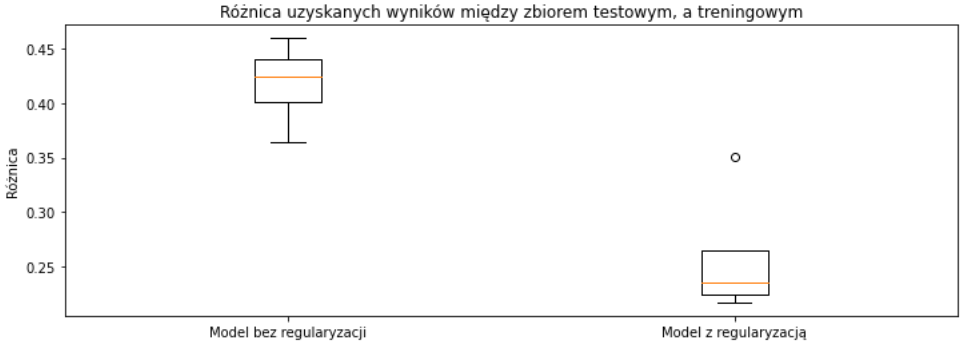
Z regularyzacją L2 oraz algorytmem wczesnego zatrzymywania:

```
r5_l2_build = {'input_shape': r5_x_train.shape, 'neurons_num': [180,180,180,180,5],
               'activations': [Tanh(), Tanh(), Tanh(), Tanh(), Softmax()]}
r5_l2_fit = [{'x_train': r5_x_train , 'y_train': r5_y_train , 'batch_size': 32,
               'n_epochs': 600, 'learning_rate': 0.0001, 'x_test': r5_x_test ,
               'y_test': r5_y_test , 'loss': Cross_entropy(), 'metric': F_score(),
               'verbose_step': 10, 'regularization_rate': 1.2, 'regularization_type':"l2",
               'stop_action' : True, 'stop_treshold' : 0.05, 'patience':15}]

results_r5_train_l2 , results_r5_test_l2 , r5_nns_l2 = cv_network(build_args=r5_l2_build ,
fit_args=r5_l2_fit)
```

Po wytrenowaniu dla każdej architektury po 5 modeli metodą cross-validacji i uśrednieniu wyników sporządzono wykres porównawczy.



Rysunek 23: Wizualizacja funkcji straty dla modelu z regularyzacją oraz bez regularyzacji



Rysunek 24: Różnica pomiędzy F-score uzyskanym na zbiorze testowym oraz treningowym dla dwóch architektur

8.3.3 Zbiór Rings3-Balance

Na tym zbiorze rozważono dwie architektury.

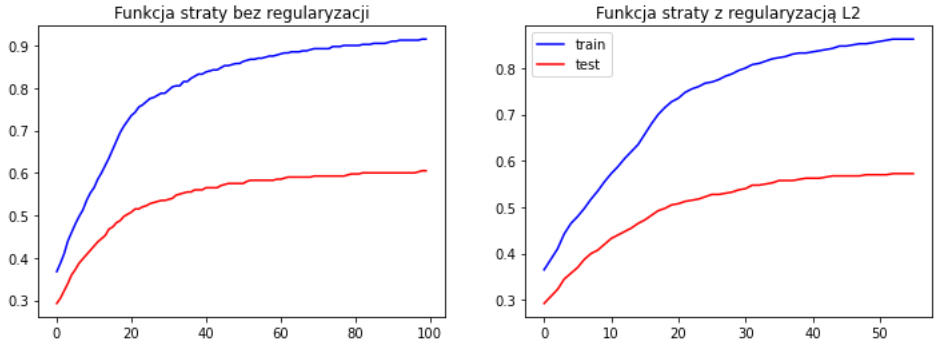
Bez regularyzacji oraz algorytmu wczesnego zatrzymywania:

```
r3_no_reg_build = {'input_shape': r3_x_train.shape, 'neurons_num': [50, 20, 20, 50, 50, 3],
                   'activations': [Tanh(), Tanh(), Tanh(), Tanh(), Tanh(), Softmax()]}
r3_no_reg_fit = [{'x_train': r3_x_train, 'y_train': r3_y_train, 'batch_size': 64,
                  'n_epochs': 100, 'learning_rate': 0.0001, 'x_test': r3_x_test,
                  'y_test': r3_y_test, 'loss': Cross_entropy(), 'metric': F_score(),
                  'verbose_step': 1, 'regularization_rate': 0, 'stop_action' : False}]
results_r3_train, results_r3_test, r3_nns = cv_network(build_args=r3_no_reg_build,
fit_args=r3_no_reg_fit)
```

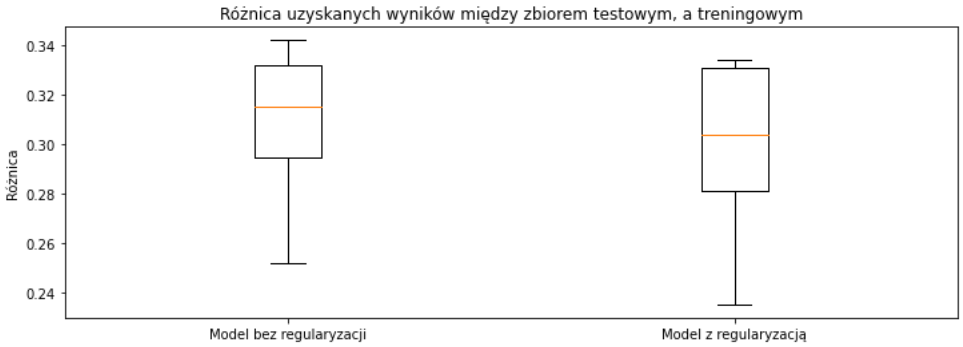
Z regularyzacją L2 oraz algorytmem wczesnego zatrzymywania:

```
r3_build_l2 = {'input_shape': r3_x_train.shape, 'neurons_num': [50, 20, 20, 50, 50, 3],
               'activations': [Tanh(), Tanh(), Tanh(), Tanh(), Tanh(), Softmax()]}
r3_l2_fit = [{'x_train': r3_x_train, 'y_train': r3_y_train, 'batch_size': 64,
              'n_epochs': 100, 'learning_rate': 0.0001, 'x_test': r3_x_test,
              'y_test': r3_y_test, 'loss': Cross_entropy(), 'metric': F_score(),
              'verbose_step': 1, 'regularization_rate': 0.5, 'regularization_type':"l2",
              'patience' : 40, 'stop_action' : True, 'stop_treshold' : 0.03}]
results_r3_train_l2, results_r3_test_l2, r3_nns_l2 = cv_network(build_args=r3_build_l2,
fit_args=r3_l2_fit)
```

Po wytrenowaniu dla każdej architektury po 5 modeli metodą cross-validacji i uśrednieniu wyników sporządzono wykres porównawczy.



Rysunek 25: Wizualizacja funkcji straty dla modelu z regularyzacją oraz bez regularyzacji



Rysunek 26: Różnica pomiędzy F-score uzyskanym na zbiorze testowym oraz treningowym dla dwóch architektur

8.3.4 Zbiór Xor3-Balance

Na tym zbiorze rozważono dwie architektury.

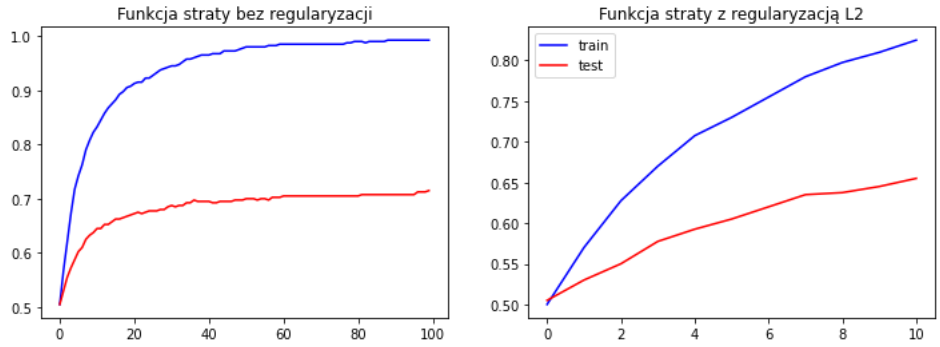
Bez regularyzacji oraz algorytmu wczesnego zatrzymywania:

```
xor_no_reg_build = {'input_shape': xor3_x_train.shape, 'neurons_num': [70, 300, 300, 300, 70,
                    'activations': [Tanh(), Tanh(), Tanh(), Tanh(), Tanh(), Softmax()]}
xor_no_reg_fit = [{'x_train': xor3_x_train, 'y_train': xor3_y_train, 'batch_size': 64,
                    'n_epochs': 100, 'learning_rate': 0.0001, 'x_test': xor3_x_test,
                    'y_test': xor3_y_test, 'loss': Cross_entropy(), 'metric': F_score(),
                    'verbose_step': 1,'regularization_rate': 0, 'stop_action' : False}]

results_xor_train, results_xor_test, xor_nns = cv_network(build_args=xor_no_reg_build,
fit_args=xor_no_reg_fit)
```

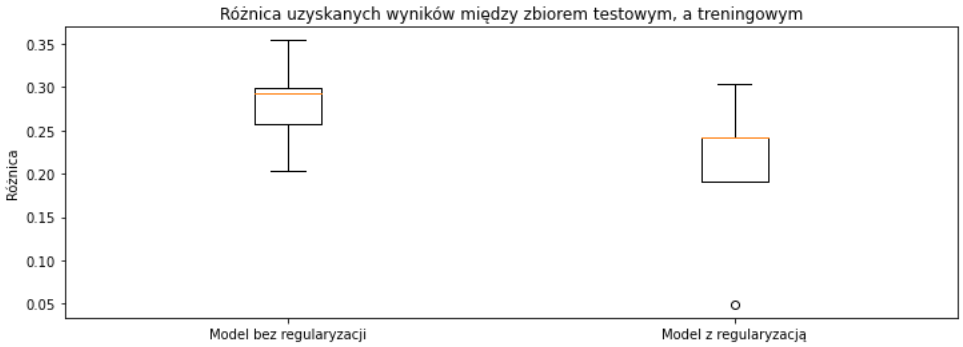
Z regularyzacją L2 oraz algorytmem wczesnego zatrzymywania:

```
xor_build_l2 = {'input_shape': xor3_x_train.shape,
                'neurons_num': [70, 300, 300, 300, 70, 2],
                'activations': [Tanh(), Tanh(), Tanh(), Tanh(), Tanh(), Softmax()]}
xor_l2_fit = [{'x_train': xor3_x_train, 'y_train': xor3_y_train, 'batch_size': 64,
                'n_epochs': 100, 'learning_rate': 0.0001, 'x_test': xor3_x_test,
                'y_test': xor3_y_test, 'loss': Cross_entropy(), 'metric': F_score(),
                'verbose_step': 1, 'regularization_rate': 3, 'regularization_type':"l2",
                'patience' : 10, 'stop_action' : True, 'stop_treshold' : 0.1}]

results_xor_train_l2, results_xor_test_l2, xor_nns_l2 = cv_network(build_args=xor_build_l2,
fit_args=xor_l2_fit)
```

Po wytrenowaniu dla każdej architektury po 5 modeli metodą cross-validacji i uśrednieniu wyników sporządzono wykres porównawczy.



Rysunek 27: Wizualizacja funkcji straty dla modelu z regularyzacją oraz bez regularyzacji



Rysunek 28: Różnica pomiędzy F-score uzyskanym na zbiorze testowym oraz treningowym dla dwóch architektur

8.4 Wnioski oraz podsumowanie

Regularyzacja wprowadzona w eksperymentach pozwoliła uzyskać wyższe wartości metryk na zbiorach testowych oraz minimalnie zmniejszyć dopasowanie modelu. Jak można było zobaczyć na wizualizacjach, zazwyczaj architektury wyposażone w mechanizmy regularyzacji, przyjmowały zbliżone wartości metryk na zbiorach testowym i treningowym (boxplot) oraz ich krzywe uczenia były bliżej względem siebie w porównaniu do architektury bez regularyzacji. Zauważalny jest więc fakt, iż regularyzacja poprzez nakładanie dodatkowej kary na wagi zmniejsza dopasowanie modelu, a dzięki temu pozbywamy się zjawiska "overfittingu".

Podsumowanie eksperymentów:

Dataset	regularization and stop	metric	mean metric train	mean metric test
Multimodal-sparse	No	Mse	292.805	968.465
Multimodal-sparse	Yes	Mse	42.718	137.008
Rings5-sparse	No	F-score	0.990	0.580
Rings5-sparse	Yes	F-score	0.958	0.685
Rings3-balance	No	F-score	0.915	0.605
Rings3-balance	Yes	F-score	0.862	0.572
Xor3-balance	No	F-score	0.992	0.715
Xor3-balance	Yes	F-score	0.825	0.655

Tabela 11: Wyniki przeprowadzonych eksperymentów

9 Podsumowanie całego projektu

W ramach tego projektu przybliżona została tematyka sieci neuronowych, ich implementacji oraz wizualizacji działania. W raporcie zostały przedstawione kluczowe wizualizacje wykonane podczas wykonywania kolejnych prac domowych, przedstawiające efekty działania dodawanych nowych mechanizmów.

Literatura

[1] Normalizacja gradientu: Wykład 6e. http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf.

[2] Towards data science. https://towardsdatascience.com.

[3] A. P. Engelbrecht. *Uczenie gradientowe z momentem: rozdział 3.2.2.*

[4] Jan Karwowski. Strona przedmiotowa. https://pages.mini.pw.edu.pl/ karwowskij/dydaktyka.