Sprawozdanie 1 - Sieci neuronowe, metody inteligencji obliczeniowej

Jan Pogłód

April 2024

1 Wstęp

Celem pracy było opisanie sprawozdania na podstawie wyników skontruowanych własnoręcznie sieci neuronowych. Do osiągnięcia poniższych wyników niezbędne były poszczególne kroki, bez których implementacja skutecznych sztucznych sieci neuronowych, do problemu regresji jak i klasyfikacji, nie byłaby możliwa. Z kroków tych podkreślić można implementacje ruchu forward, backward, normalizację gradientu, zastosowanie momentu, algorytm RMSprop oraz stosowanie różnych funkcji aktywacji. W rezultacie wszystkich powyższych kroków otrzymaliśmy sieci neuronowe typu Multi Layer Perceptron, które skutecznie potrafiły dokonywać predykcji na danych zbiorach.

2 Wprowadzenie do teorii sieci neuronowej

Aby zrozumieć w jaki sposób skontruować wielo-neuronową sieć konieczne jest zdefiniowanie matematycznego modelu pojedyńczego neuronu. Pierwszy matematyczny model neuronu opracowany został w 1943 roku przez naukowców na których cześć nadano mu nazwę McCulloch-Pitts neuron, co dało korzenie do implementacji współczesnych sieci neuronowych. Neuron ten przyjmuje na wejściu dwa sygnały, 0 albo 1 i oblicza sumę ważoną (1) wszystkich tych wejść. Założeniami modelu jest, że jeżeli suma ta przekroczy pewną progową wartość, neuron aktywuje się, w przeciwnym razie pozostanie nieaktywny. Sumę tę można zapisać za pomocą symboli x_i - wartość sygnału na *i-tym* wejściu, w_i - waga przypisana na *i-tym* wejściu oraz b - próg aktywacji powszechnie nazywany jako bias. W rezultacie, informacją o wszystkich wejściach neuronu będzie poniższa suma:

$$\sum_{i=1}^{n}(x_i \cdot w_i) + b \tag{1}$$

Następnie na wyjście neuronu kładziemy powyższą sumę obłążoną funkcją aktywacji. Na jej podstawie neuron ten decyduje czy aktywować siebie czy nie. W modelu McCulloch'a i Pitts'a na sumę ważoną nakłada się funkcję skokową, która zwraca 1, jeżeli neuron przekroczy pewien próg aktywacji θ, a w przeciwnym razie zwraca 0.

$$y = \begin{cases} 1 & \text{gdy } \sum_{i=1}^{n}(x_i \cdot w_i) + b > \theta \\ 0 & \text{w przeciwnym razie} \end{cases}$$

Powyższy model neuronu jest stosowany współcześnie, a za sprawą rozwoju coraz to lepszych komputerów zaczęto rozważać różne połączenia, architektury ułożenia neuronów jak i funkcje aktywacji, aby osiągnąć najbardziej skuteczne sieci neuronowe dla konkretnego problemu. Model sieci, w którym jest wiele neuronów nazywamy modelem Multi Layer Perceptron. Implementacje takiego modelu przedstawiłem w poniższych krokach.

3 Implementacja Multi Layer Perceptron

We współczesnej specyfikacji rozbudowanej sieci neuronowej może być umieszczona duża ilość neuronów oraz warstw ukrytych. Wówczas każdy neuron jest najczęściej połączony z każdym neuronem z kolejnej warstwy ukrytej/wyjścia tworząc w ten sposób sekwencję funkcji, które mogą się aktywować, lub być nieaktywne w zależności od wejść i wag konkretnego neuronu. Przejście przez całą sieć neuronową, od wejścia, po wszystkich jej warstwach ukrytych i otrzymanie wyliczonego z nich wyjścia nazywamy ruchem forward.

3.1 Ruch Forward

Przy danym wejściu $x = x_1, x_2, ..., x_n$ w każdym neuronie obliczane są wyjścia z ustalonymi wagami - początkowo w losowy sposób na jedną z metod, na przykład z rozkładu normalnego. Przykładowo, przy czterech neuronach w pierwszej warstwie ukrytej dla każdego z *i-tego*, *i=1,2,3,4* neuronu losujemy wagi: $w_1^i, w_2^i, ..., w_n^i$, oraz każdy z tych neuronów ma przypisany unikalny bias - b^i. Z powyższych wartości układamy równanie macierzowe jak w (2), dzięki któremu obliczyć możemy wyjście dla pierwszej warstwy ukrytej. Na to wyjście możemy nałożyć następnie funkcję aktywacji (przykładowo tangens hiperboliczny), aby aktywować poszczególne neurony. Przykładowy ruch forward został ukazany poniżej.

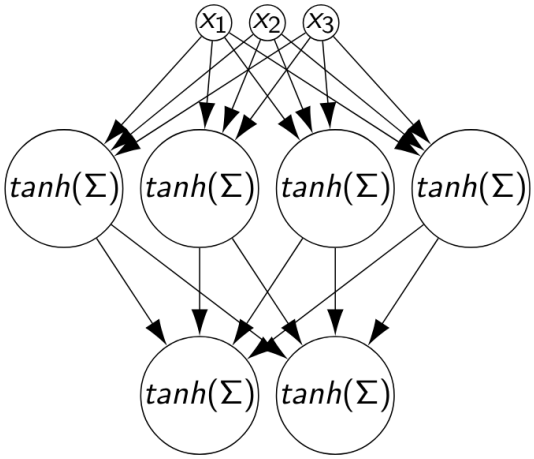


Figure 1: Przykład dwu-warstwowej sieci neuronowej z funkcją aktywacji tanh

Wyjście dla pierwszej warstwy ukrytej (przy ustalonej i=1,2,...,k liczbie neuronów), przy danym wejściu x, macierzy wag W oraz wektorze biasów b, możemy obliczyć w następujący sposób:

$$y^1 = x \cdot W + b = \begin{bmatrix} x_1 & x_2 & \cdots & x_n \end{bmatrix} \begin{bmatrix} w_1^1 & w_1^2 & \cdots & w_1^k \\ w_2^1 & w_2^2 & \cdots & w_2^k \\ \vdots & \vdots & \vdots & \vdots \\ w_n^1 & w_n^2 & \cdots & w_n^k \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} \tag{2}$$

W ten sposób dostaniemy wyjście z warstwy ukrytej, k-elementowy wektor y, na który możemy nałożyć funkcję aktywacji, otrzymując w ten sposób decyzje o aktywacji poszczególnych neuronów.

$$Z^1 = tanh(y^l) = tanh(\begin{bmatrix} y_1^1 & y_2^1 & \cdots & y_k^1 \end{bmatrix}^T) \tag{3}$$

W testach rozważać będziemy również inne funkcje aktywacji takie jak sigmoidalną, czy ReLU:

- $\sigma(x) = \frac{1}{1+e^{-x}}$
- $\text{ReLU}(x) = \max(0, x)$

Możemy postępować analogicznie jak w (2-3), aby stworzyć kolejną warstwę ukrytą, tym razem przyjmując Z^1 jako wejście do drugiej warstwy ukrytej. Również w tym celu stworzyć należy nowe wagi, oraz biasy dla neuronów w kolejnej warstwie ukrytej. W ten sposób otrzymujemy kolejno: Z^2 - wyjście z drugiej warstwy ukrytej, Z^3 - wyjście z trzeciej warstwy ukrytej. Przypuśćmy, że wektor y^l jest wyjściem z ostatniej warstwy ukrytej, bez nałożonej na niego funkcji aktywacji. Wówczas wektor ten jest predykcją naszej sieci neuronowej oraz zakończyliśmy działanie ruchu Forward. W przypadku problemu regresji istotnie nie będziemy nakładać funkcji aktywacji na wyjście z sieci, posłużymy więc w tym celu zwykłą funkcję liniową, która zwraca sama siebie. W przypadku problemu klasyfikacji wykorzystywać będziemy na wyjściu funkcję softmax, która dla wektora $y = (y_1, y_2, \ldots, y_n)$ jest zdefiniowana jako:

$$\text{softmax}(y_i) = \frac{e^{y_i}}{\sum_{j=1}^{n} e^{y_j}} \quad \text{dla} \quad i = 1, 2, \ldots, n$$

Na koniec powyższych obliczeń możemy obliczyć jak bardzo nasza sieć neuronowa się myli. W tym celu, jeżeli dysponujemy zbiorem treningowym w którym znamy rzeczywiste wartości wektora wyjścia, oznaczonego jako y_{rzec}, możemy obliczyć różnicę między nim, a przewidzianym wektorem y^l. Jest to informacja o tym jak dużym błędem dysponujemy przy ruchu forward, i możemy ją obliczać poprzez:

$$error^l = y^l - y_{rzec}$$

3.2 Ruch Backward

Celem ruchu backward, a więc od wyjścia sieci neuronowej do jej wejścia, jest poprawienie wag oraz biasów wszystkich neuronów znajdujących się w architekturze sieci, tak aby zminimalizować błąd *error*. Ważną metodą w propagacji wstecznej jest 2. reguła perceptronu, która mówi, że jeśli neuron nie jest aktywowany, a powinien być, to zmieniamy wagę tak, żeby zwiększyć jego szansę aktywacji. Pierwszym krokiem jest policzenie błędu na wyjściu z l-tej warstwy ukrytej. Dokonujemy tego za pomocą nas†epującego iloczynu skalarnego dla wektora error oraz macierzy wag dla neuronów w tej warstwie ukrytej, przy ustalonej liczbie k neuronów oraz m - elementach wektora wyjściowego y_{rzec}. Następnie wyrażenie to mnożymy przez pochodną funkcji aktywacji:

$$delta^l = error^l \begin{bmatrix} w_1^1 & w_1^2 & \cdots & w_1^k \\ w_2^1 & w_2^2 & \cdots & w_2^k \\ \vdots & \vdots & \vdots & \vdots \\ w_m^1 & w_m^2 & \cdots & w_m^k \end{bmatrix}^T \frac{d}{dZ^l}\text{activation}(Z^l) \tag{4}$$

Kolejnym krokiem jest obliczenie gradientów dla *l-tej* warstwy ukrytej. Poniżej kolejno przedstawione są rezultaty dla wag oraz biasów dla rozpatrywanej, l-tej warstwy ukrytej.

$$grad^w =< (Z^l)^T, error^l > \tag{5}$$

$$grad^b = \sum_{i=1}^{m} error_i^l \tag{6}$$

Otrzymane w ten sposób gradienty na l-tej warstwie ukrytej możemy zapisać jako:

grad = ($grad^w$, $grad^b$)

Kolejnym krokiem jest zaktualizowanie wag i biasów dla tej warstwy uktrytej. W tym celu wprowadzamy krok uczenia, oznaczany jako η i najczęściej przyjmujemy go na poziomie $\eta = 0.001$. Dalej dokonujemy aktualizacji wag i biasów za pomocą grad i η:

$\Theta = (W^l, b^l) = \eta$ grad

Następnie przeprowadzamy tę operację dla kolejnych *l-1, l-2, ... , 1* wastw ukrytych. W momencie gdy zakończymy na wejściu do pierwszej warstwy ukrytej kończone jest działanie algorytmu backpropagacji oraz od nowa przeprowadzany jest ruch forward, dla zaktualizowanych wag i biasów. Kolejne iteracje ruchu forward-backward nazywamy epochami. Zadaniem każdej z epoch jest obliczenie powyższych gradientów i zminimalizowanie ostatecznego błędu $error^l$. Algorytm uczenia przetwarzamy do momentu osięgnięcia pewnego warunku stopu. Możemy go zapisać poprzez poniższy pseudokod.

Algorithm 1 Algorytm uczenia sieci neuronowej

$\Theta \leftarrow$ InicjujLosowo;
while $\neg$StopCondition **do**
  for all $(X, Y) \in ZbiorUczacy$ **do**
    $\hat{Y} \leftarrow$ FeedForward(Θ, X);
    $\Delta\Theta \leftarrow$ Backpropagate$(\hat{Y}, Y)$;
    $\Theta \leftarrow \Theta + \Delta\Theta$;
  end for
end while

Warunkiem stopu w powyższym pseudokodzie może być na przykład osiągnięcie zamierzonego błędu. Błąd możemy mierzyć na kilka sposobów, a najpopularniejsze z nich przedstawiono poniżej.

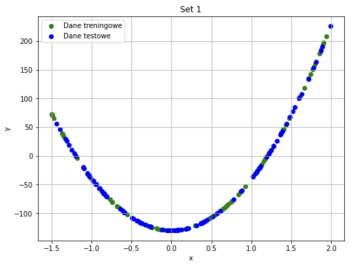
- Średni Błąd Bezwzględny: $\frac{1}{N}\sum_{i=1}^{N}|\hat{y}_i - y_i|$
- Średni Błąd Kwadratowy: $\frac{1}{N}\sum_{i=1}^{N}(\hat{y}_i - y_i)^2$
- Lub w przypadku klasyfikacji: Krosentropia: $\frac{1}{N}\sum_i\sum_j y_{ij}\log\hat{y}_{ij}$

W zależności od stosowanej wyżej funkcji, nazywanej funkcją kosztu, możemy otrzymywać w bardziej efektywny sposób błąd $error^l$. Dla funkcji kosztu wybranej jako średni błąd kwadratowy, wyrażenie $errol^l$ otrzymujemy za pomocą jej pochodnej, danej jako:

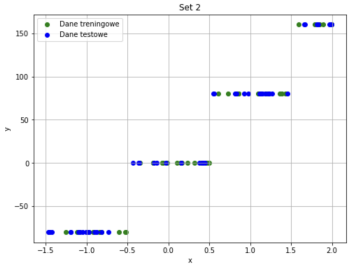
- $errol^l = \frac{dLoss}{dy^l} = 2(y^l - y_{\text{rzec}})/n$

4 Zaprezentowanie wyników na podstawie powyższych implementacji

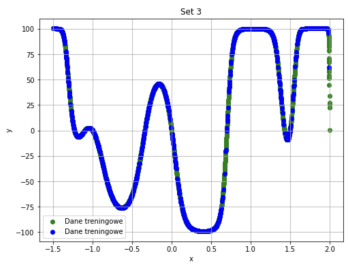
Aby sprawdzić skuteczność uczenia się sieci neuronowej wykorzystane zostały w tym celu różne zbiory danych. Początkowo rozpatrywany był problem regresji, a jako miarę skuteczności użyto średniego błędu kwadratowego. Poniżej zademonstrowano kilka z funkcji umieszczonych w tych zbiorach z podziałem na dane treningowe i testowe. Jako podpisy podane są nazwy zbiorów.



(a) Zbiór 1 - "square-simple"



(b) Zbiór 2 - "steps-small"

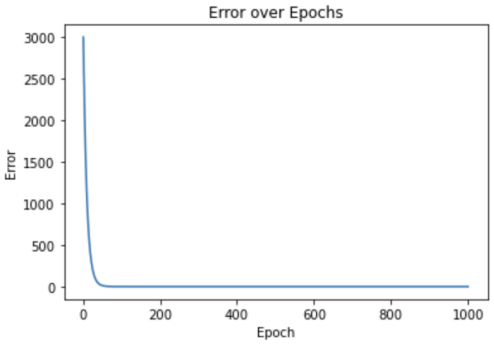


(c) Zbiór 3 - "multimodal-small"

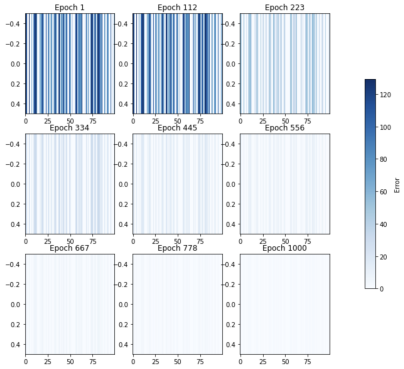
Testowanie opierać się będzie na następujących zmiennych:

- architektura sieci (liczba warstw ukrytych i neuronów w poszczególnych warstwach ukrytych)
- metoda losowania wag spośród: rozkład normalny, rozkład jednostajny na przedziale [0,1], Xavier oraz he.
- uczenie z normalizacją min-max lub bez
- uczenie z zastosowaniem momentu i normalizacji gradientu lub bez
- funkcje aktywacji

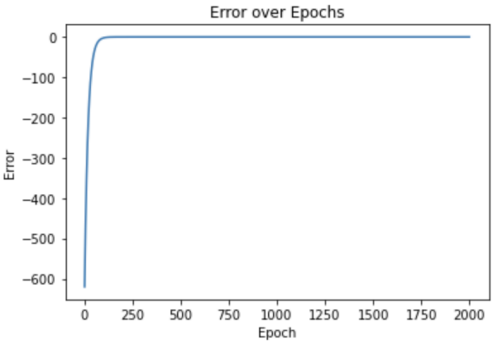
Dla powyższych zbiorów mierzyliśmy jak zmienia się błąd sieci neuronowej na przestrzeni kolejnych epoch oraz jak wygląda $delta^l$ wag dla poszczegolnych epoch. Wszystkie wizualizacje o zbiorach 1-3 umieszczone są poniżej.



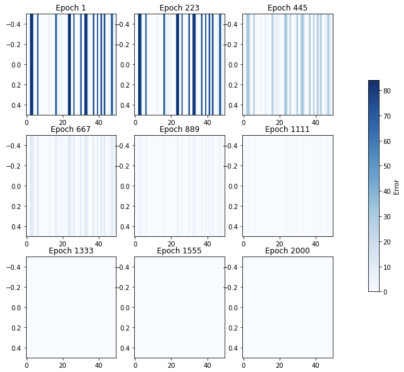
(a) "square-simple" - error w zależności od epoch *MSE train: 0.41*



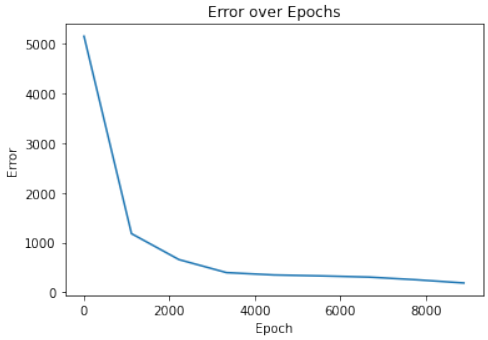
(b) "square-simple" - zmiana macierzy $delta^l$, *MSE test: 21.35*



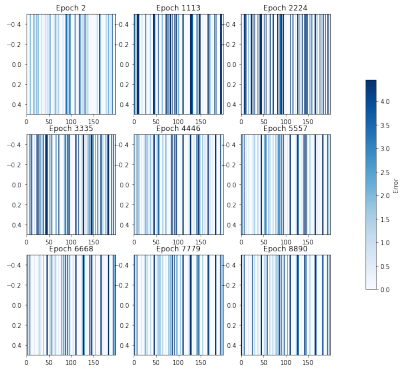
(a) "steps-small" - error w zależności od epoch *MSE train: 1.28*



(b) "steps-small" - zmiana macierzy $delta^l$ *MSE test: 127.99*



(a) "multimodal-small" - error w zależności od epoch *MSE train: 188.83*



(b) "multimodal-small" - zmiana macierzy $delta^l$ *MSE test: 1180.78*

Na powyższych wizualizacjach widzimy, że implementacja z części 3 istotnie pozwala sieci neuronowej uczyć się danych, minimalizować funkcję kosztu oraz przewidywać wyniki na nowych danych z pewną dokładnością. Każda z powyższych sieci została wytrenowana poprzez sigmoidalną funkcję aktywacji w warstwie ukrytej oraz liniową w warstwie wyjściowej, z parametrem learning rate$=0.001$ oraz jedną warstwą ukrytą. Liczba nerunów dla każdego zbioru w jednej warstwie ukrytej wyniosła:

Zbiór	square-simple	steps-small	multimodal-small
Liczba neuronów:	50	50	80
Czas wykonania:	0.12	0.15	4.31

Choć udało nam się osiągnąć efekt uczenia i optymalizowania błędu, to wciąż sieć nie daje zadawalających wyników - szczególnie w przypadku skomplikowanej funkcji multimodal. W celu poprawy rezulatów w dalszych krokach badać będziemy wpływ metod normalizacji gradientu, momentów, normalizacji danych treningowych, zwiększenia liczby warstw, aby poprawić nasze wyniki. Testowanie w sekcji 5 przeprowadzimy dla zbioru "square-simple" przy stałej liczbie 50 neuronów w jednej warstwie.

5 Dodanie ulepszeń i rozpatrywanie nowych metod

5.1 Normalizacja danych

Normalizacja danych ma na celu przekształcenie wartości w zbiorze danych w taki sposób, aby mieć ustalony zakres wartości, przykładowo od 0 do 1. Jest to przydatne w przypadku, gdy cechy danych mają różne zakresy wartości, co może wpłynąć na wydajność zaimplementowanej sieci neuronowej. Normalizacja pozwala na lepsze porównanie wartości w zbiorze treningowym. Wyróżnimy dwie metody normalizacji: min-max oraz standaryzację. Normalizacja Min-Max:

$$\text{Normalizacja}(X) = \frac{X - \min(X)}{\max(X) - \min(X)}$$

Denormalizacja Min-Max:

$$\text{Denormalizacja}(X) = X(\max(X) - \min(X)) + \min(X)$$

Standaryzacja:

$$\text{Standaryzacja}(x) = \frac{x - \mu}{\sigma}$$

Destandaryzacja:

$$\text{Destandaryzacja}(x) = x\sigma + \mu$$

Do testu rozpatrywany będzie zbiór "square-simple" z jedną warstwą ukrytą i 50 neuronami.

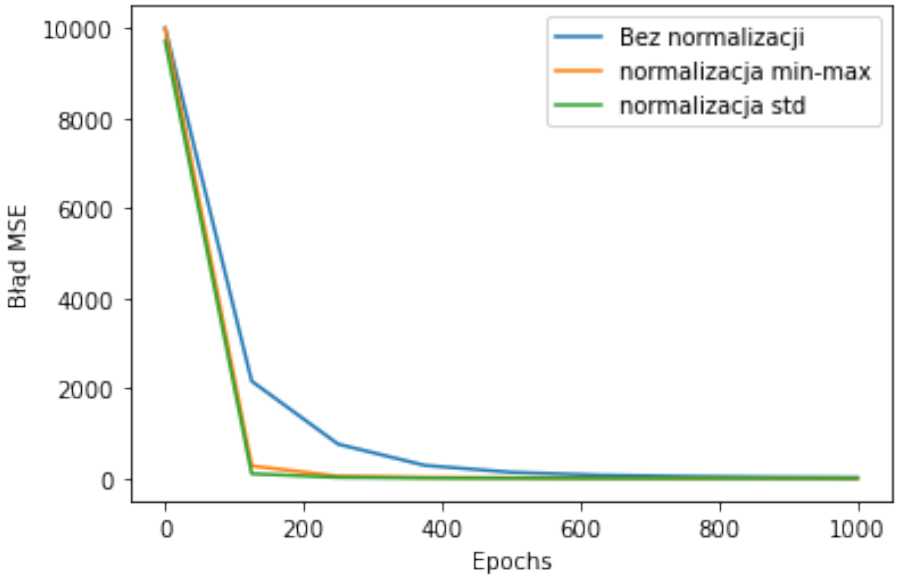


Figure 6: Wpływ normalizacji danych na proces uczenia - zbiór "square-simple"

Metoda normalizacji	Brak	Min-Max	Std
Wynik MSE na zbiorze treningowym:	23.64	5.45	1.91
Wynik MSE na zbiorze testowym:	41.11	26.92	23.79

Na powyższej ilustracji widzimy, że w problemie regresji normalizacja danych ma duży wpływ na proces uczenia. W przypadku metody min-max lub standaryzacji minimalizacja funkcji kosztu nastepuję szybciej, i w przypadku takich samych pozostałych parametrów, sieci neuronowej udało się zbiec do znacznie niższego błędu już poniżej 200 epoch. Na wykresie widzimy również nieznaczną, lecz dostrzegalną przewagę normalizacji min-max nad standaryzacją w szybkości zbieżności. Winiki końcowe wskazują jednak nad przewagą standaryzacji. W dalszych krokach będziemy rozpatrywać obie te metody normalizacji.

5.2 Dodanie momentu

Kolejnym sposobem poprawienia uczenia jest zaimplementowanie metody uczenia z momentem. Sposób ten ma za zadanie przyspieszyć proces uczenia poprzez szybsze zbieganie w kierunku minimum funkcji kosztu. Moment zapamiętuje kierunek, w którym w poprzednich krokach były zmiany wag, co pozwala na szybsze poruszanie się wzdłuż gradientu. Ponadto dzięki temu sieć neuronowa może zapobiegać sytuacje gdzie gradient jest bliski zeru, a uczenie bez momentu może utknąć w lokalnym minimum. W celu zaimplementowania uczenia z momentem definiujemy nowy parametr λ - współczynnik wygaszania momentu. Zapamiętywania kierunków zmian wag w poprzednich krokach dokonujemy za pomocą tablicy Momentum. Poniżej można zaobserwować skuteczność dodania momentu do procesu uczenia.

Algorithm 2 Uczenie z momentem

$\Theta \leftarrow$ InicjujLosowo;
Momentum $\leftarrow [0, 0, \ldots, 0]$;
while $\neg$StopCondition **do**
    $\Delta\Theta \leftarrow [0, 0, \ldots, 0]$;
    for all $(X, Y) \in$ *Zbior uczacy* **do**
        $\hat{Y} \leftarrow Network(\Theta, X)$;
        $\Delta\Theta \leftarrow \Delta\Theta - \nabla Network(\Theta, X)$;
    end for
    Momentum $\leftarrow \Delta\Theta + \text{Momentum} \cdot \lambda$;
    $\Theta \leftarrow \Theta + \eta \cdot \text{Momentum}$;
end while

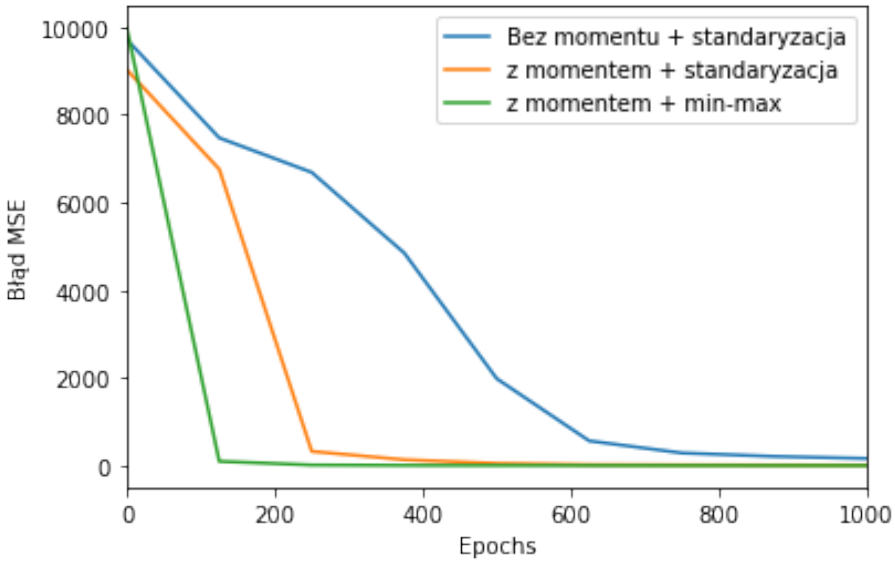


Figure 7: Wpływ dodawania momentów na proces uczenia - zbiór "square-simple"

Sposób uczenia	Bez momentu std	Z momentem std	Z momentem min-max
MSE na zbiorze treningowym:	2.46	0.64	0.13
MSE na zbiorze testowym:	25.01	22.18	22.21

Możemy zauważyć, że w przypadku uczenia z momentem oraz normalizacji sieć neuronowa uczy się danych znacznie szybciej i na poziomie epoch = 200 skutecznie minimalizuje funkcję kosztu. Oznacza to, że dzięki tej metodzie możemy skrócić czas oczekiwania na uczenie, co może mieć znaczny wpływ przy większych zbiorach. Ponadto otrzymujemy jeszcze niższy błąd MSE na zbiorze treningowym, tj. na poziomie 0.13.

5.3 Normalizacja gradientu - RMS prop

Główną zaletą implementowania normalizacju gradientu jest zapobieganie problemowi zanikającego lub wybuchającego gradientu. Przykładem metody normalizacji gradientu jest algorytm RMS prop. Algorytm utrzymuje normę gradientu w określonym zakresie, dzięki temu unikamy problemów związanych z bardzo małymi lub bardzo dużymi wartościami gradientu, które mogą utrudnić lub spowolnić proces uczenia. Aby zaimplementować RMS prop wprowadzamy parametr β – współczynnik wygaszania. Skuteczną wartością współczynnika β jest wartość *0.9*. Normalizacji gradientu dokonujemy poprzez kombinację liniową drugiego momentu funkcji g.

Algorithm 3 RMSProp

$\Theta \leftarrow$ InicjujLosowo;
$Eg^2 \leftarrow [0, 0, \ldots, 0]$;
while ¬StopCondition **do**
    $g \leftarrow 0$;
    for all $(X, Y) \in$ *Zbior uczacy* **do**
        $\hat{Y} \leftarrow Network(\Theta, X)$;
        $g \leftarrow g + \nabla Network(\Theta, X)$;
    end for
    $Eg^2 \leftarrow \beta Eg^2 + (1 - \beta)Eg^2$;
    $\Theta \leftarrow \Theta - \eta \cdot \frac{g}{\sqrt{Eg^2 + \epsilon}}$;
end while

Następnie możemy połączyć implementację RMSProp oraz uczenia z momentem, otrzymując w ten sposób algorytm Adam (Adaptive moment estimation). Dzięki implementacji tego algorytmu w naszej sieci neuronowej unikamy wrażliwych sytuacji, które mogą wpłynąć na proces uczenia: eksplozji wag, utknięcia w minimum lokalnym, spowolnienie procesu uczenia oraz przyspieszymy proces minimalizacji funkcji kosztu.

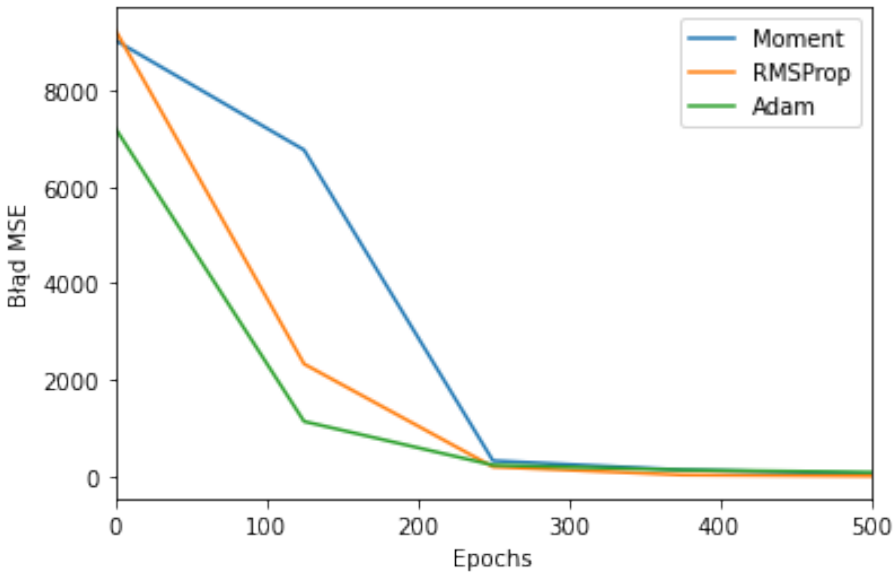


Figure 8: Wpływ metod moment, rmsprop oraz adam na proces uczenia - zbiór "square-simple"

5.4 Batching

Kolejną implementacją przydatną do testowania rezulatów sieci neuronowej jest metoda batching. Polega ona na stopniowym uczeniu sieci poprzez dawkowanie mniejszej ilości danych w przeciwieństwie do trenowania wszystkich danych naraz. Dzięki temu, w każdej z epoch, możemy uczyć sieć neuronową poprzez podział danych na p częsci. Wówczas konieczne jest, aby w każdej z p z podziałów indeksy danych były wymieszane między sobą. Następnie trenujemy sieć neuronową nadając jej stopniowo w treningu 1-szą,2-gą,...p-tą część danych.

$$X_{k\in 1,..,p} = X[k]$$

Takie podejście może być szczególnie pomocne w przypadku dużych zbiorów danych ponieważ efektywnie zarządzamy pamięcią. Ponadto możemy ustabilizować proces uczenia, ponieważ uśrednianie gradientów dla wielu przykładów w batchu pomaga w redukcji zmienności gradientów i może prowadzić do bardziej stabilnego procesu uczenia, co z kolei może pomóc w uniknięciu oscylacji funkcji kosztu. Dodatkowo zapobiegamy przetrenowania sieci, ponieważ model ma szansę zobaczyć różne przykłady z różnych części zestawu danych w każdym kroku uczenia, co może pomóc w lepszym uchwyceniu różnorodności danych

W tym ekperymencie uchwycone zostały trzy sposoby podziału danych:

- bez batchingu - 100% danych w jednym batchu
- podział na 10 batchów - 10% danych w jednym batchu
- podział na 5 batchów - 5% danych w jednym batchu

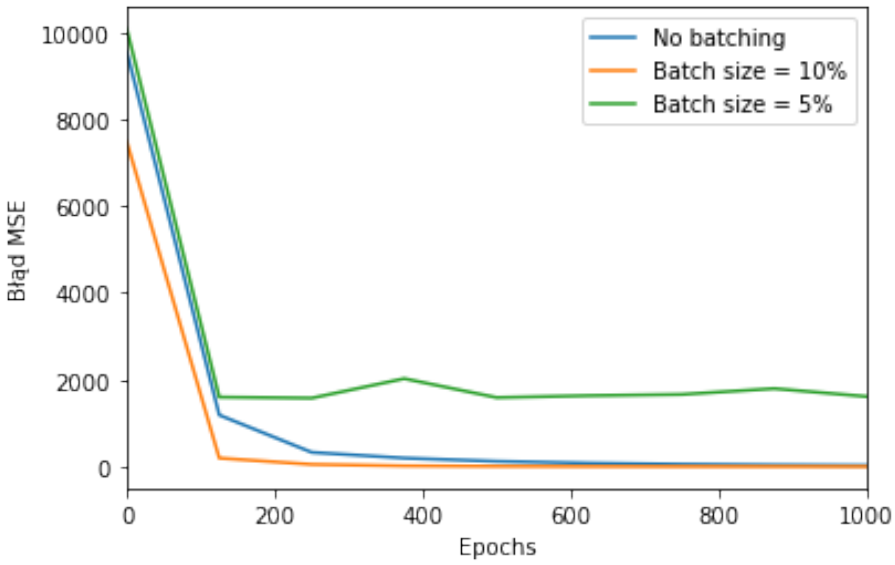


Figure 9: Wpływ implementacji metody batching na proces uczenia - zbiór "square-simple"

Zbiór	Bez batchingu	Batching 10%	Batching 5%
MSE na zbiorze treningowym:	0.019	0.0025	1837.27
MSE na zbiorze testowym:	21.164	21.07	1248.46

Na podstawie powyższej tabeli i wykresu można zauważyć, że istotne jest odpowiednie zarządzanie pojemnością batcha. W przypadku batchów zawierających jedynie 5% danych sieć neuronowa nie dała rady zminimalizować funkcji kosztu zatrzymując się na poziomie MSE = 1800. Natomiast podział zbioru na 10 części pozwolił sieci neuronowej szybciej zbiec oraz zmniejszyć MSE na zbiorze treningowym jak i testowym.

5.5 Sposób inicjowania wag

Testować będziemy również sposoby początkowego inicjowania wag dla każdej warstwy ukrytej W^l. Metoda inicjowania wag wpływa na szybkość zbieżności oraz czas uczenia, a także na końcowe MSE. Wyróżnimy 4 sposoby inicjowania wag:

- Rozkład normalny na przedziale [0,1]: $W_{ij} \sim \mathcal{N}(0,1)$
- Rozkład jednostajny na przedziale [0,1]: $W_{ij} \sim U(0,1)$
- Inicjacja Xavier: $W_{ij} \sim \mathcal{N}\left(0, \sqrt{\frac{2}{n+m}}\right)$
- Inicjacja He: $W_{ij} \sim \mathcal{N}\left(0, \sqrt{\frac{6}{n+m}}\right)$

5.6 Różne funkcje aktywacji

Wspomnieliśmy, że testować będziemy równiez różne funkcje aktywacji spośród tangens hiperboliczny, ReLU oraz sigmoidalną. Wówczas uwzględniać będziemy, we wzorze na $delta^l$ różne pochodne tych funkcji, które po wyprowadzeniu przyjmują postać:

- Dla funkcji sigmoidalnej σ: $\frac{d}{dZ^l}\text{activation}(Z^l) = \sigma(x) \cdot (1 - \sigma(x))$
- Dla funkcji tanh: $\frac{d}{dZ^l}\text{activation}(Z^l) = 1 - \tanh^2(x)$
- Dla funkcji ReLU: $\frac{d}{dZ^l}\text{activation}(Z^l) = \begin{cases} 1 & \text{jeśli } x > 0 \\ 0 & \text{jeśli } x \leq 0 \end{cases}$

Na poniższym wykresie widzimy, że dla problemu regresji rozsądniejszym narzędziem od funkcji sigmoidalnej jest funkcja tanh lub relu. Znacznie szybciej udało im się zminimalizować funkcję kosztu oraz osiągnąć niższe MSE.

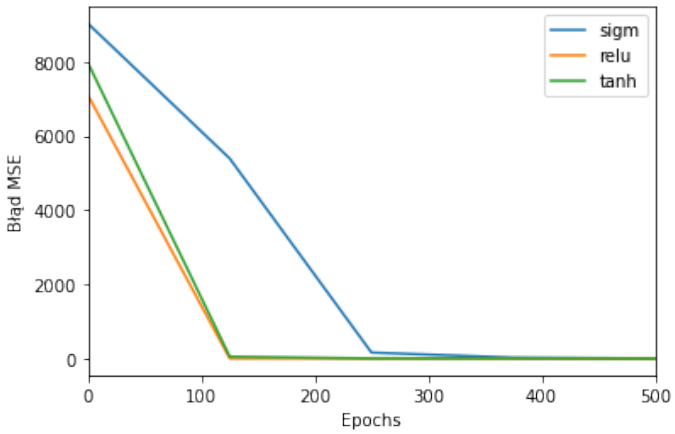
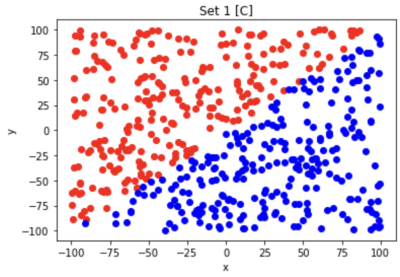


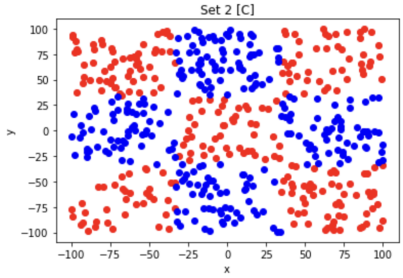
Figure 10: Wpływ różnych funkcji aktywacji na proces uczenia - zbiór "square-simple"

6 Testowanie ulepszeń w sieci neuronowej

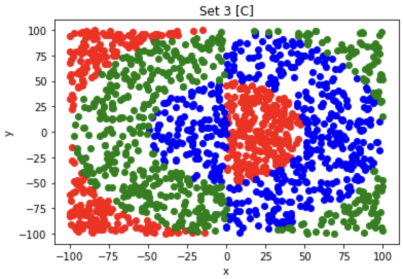
Po wprowadzeniu powyższych rozwiązań zobaczymy jakie metody są najlepsze w zależności od problemu regresji/ klasyfikacji oraz charekterystyki zbioru. Rozpatrywać będziemy również różną liczbę warstw ukrytych z różną liczbą neuronów na każdej warstwie. Zbiory klasyfikacji na których przeprowadzimy ostateczne testy oraz ich nazwy to:



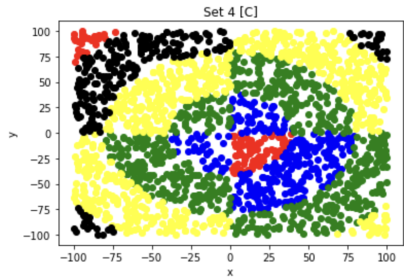
(a) Zbiór "square-simple"



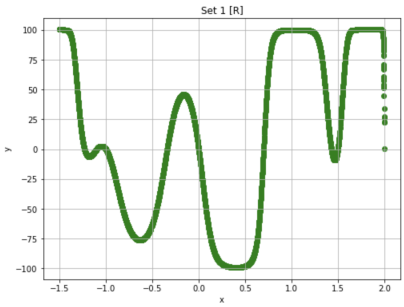
(b) Zbiór "xor"



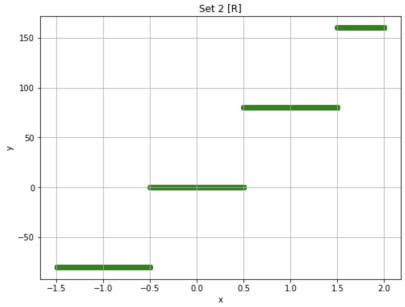
(a) Zbiór "ring3"



(b) Zbiór "ring5"



(a) Zbiór "multimodal large"



(b) Zbiór "steps-large"

6.1 Problem klasyfikacji

W przypadku zbioru "square-simple" testowaliśmy wpływ takich parametrów jak funkcja aktywacji, liczba neuronów, i liczba warstw ukrytych. Najlepszy rezultat dało użycie funkcji tanh, przy podziale zbioru treningowego na 10 części oraz z użyciem metody Adam do kierunków spadku gradientu osiągając wynik *f1 score = 0.98* na zbiorze testowym. Ponadto ustawiliśmy 1000 epoch, normalizację - standaryzację oraz jedną warstwę ukrytą z 50 neuronami.

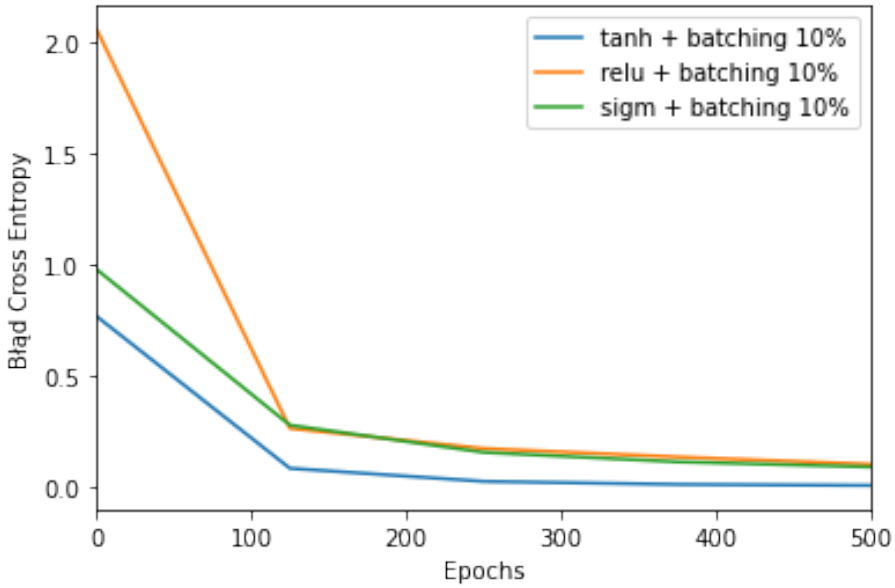


Figure 14: Porównanie wyników dla procesu uczenia zbioru "square-simple"

Sposób uczenia	tanh	sigmoidalna	relu
Cross entropy na zbiorze treningowym:	0.184	0.503	2.928
F1 score na zbiorze treningowym:	1.0	1.0	1.0
F1 score na zbiorze testowym:	0.998	0.998	0.996

Dla zbioru "xor" osiągnęliśmy najlepsze rezultaty dla dwóch warstw ukrytych, odpowiednio po liczbie neuronów 50 i 20 oraz liczbie epoch 1000. Również skorzystaliśmy z algorytmu Adam oraz funkcji tanh. Za inicjowanie wag posłużyła nam metoda "he" jako najlepsza z badanych.

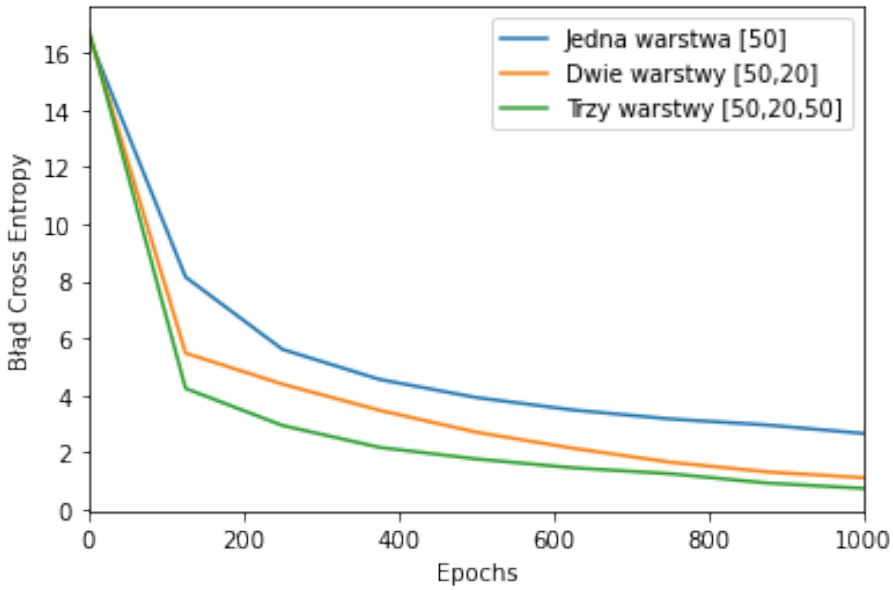


Figure 15: Porównanie wyników dla procesu uczenia zbioru "xor"

Ilość warstw i neuronów dla nich	Jedna [50]	Dwie [50,20]	Trzy [50,20,50]
Cross entropy na zbiorze treningowym:	2.90	2.584	2.005
F1 score na zbiorze treningowym:	0.974	0.99	0.988
F1 score na zbiorze testowym:	0.896	0.926	0.914

Na powyższym przykładzie możemy zauważyć, że dokładanie kolejnych warstw neuronowych niekoniecznie poprawia wyniki na zbiorze testowym. Choć widzimy, że powyżej 800 epoch sieć neuronowa z trzema warstwami zaczęła bardziej minimalizować funkcje kosztu od tej z dwoma warstwami, to ostatecznie lepsze wyniki na zbiorze testowym dała sieć z dwoma. Testowanie różnej liczby warstw ukrytych istotnie może przybliżyć nas do lepszych rezultatów.

Również w przypadku zbioru "ring3" najlepsze rezultaty osiągneliśmy dla dwóch warstw ukrytych, odpowiednio po liczbie neuronów 50 i 20. Zastowanie najbardziej optymalnych parametrów dla poprzednich zbiorów również okazało sie dawać najlepsze rezultaty. Jako liczbę epoch podaliśmy 1000.

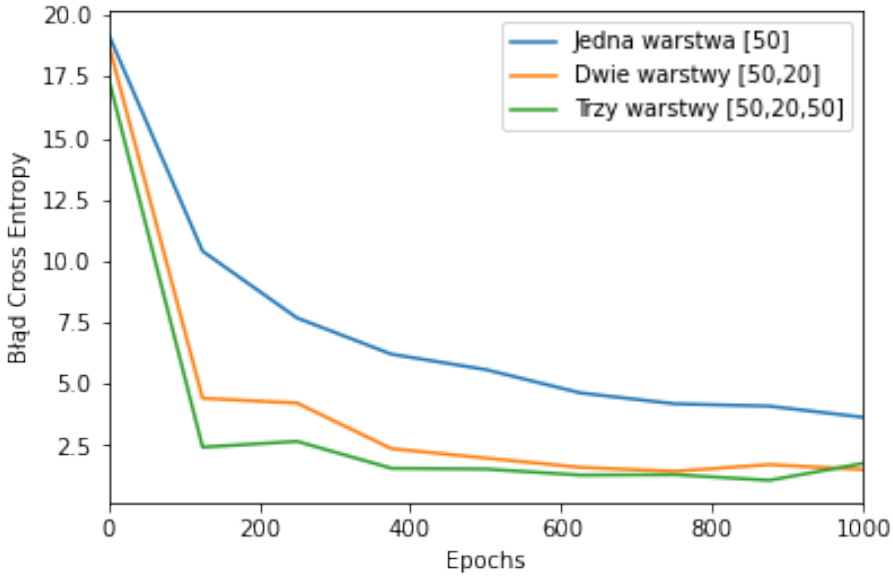


Figure 16: Porównanie wyników dla procesu uczenia zbioru "ring3"

Ilość warstw i neuronów dla nich	Jedna [50]	Dwie [50,20]	Trzy [50,20,50]
Cross entropy na zbiorze treningowym:	3.154	0.417	0.705
F1 score na zbiorze treningowym:	0.968	0.958	0.988
F1 score na zbiorze testowym:	0.945	0.952	0.986

Zbiór "ring5" był największym rozpatrywanym zbiorem, a w dodatku do predykcji było 5 klas: [0,1,2,3,4]. Również parametry, które były optymalne w poprzednich przypadkach wypadły naskuteczniej. Ponownie użyliśmy 1000 epoch do treningu

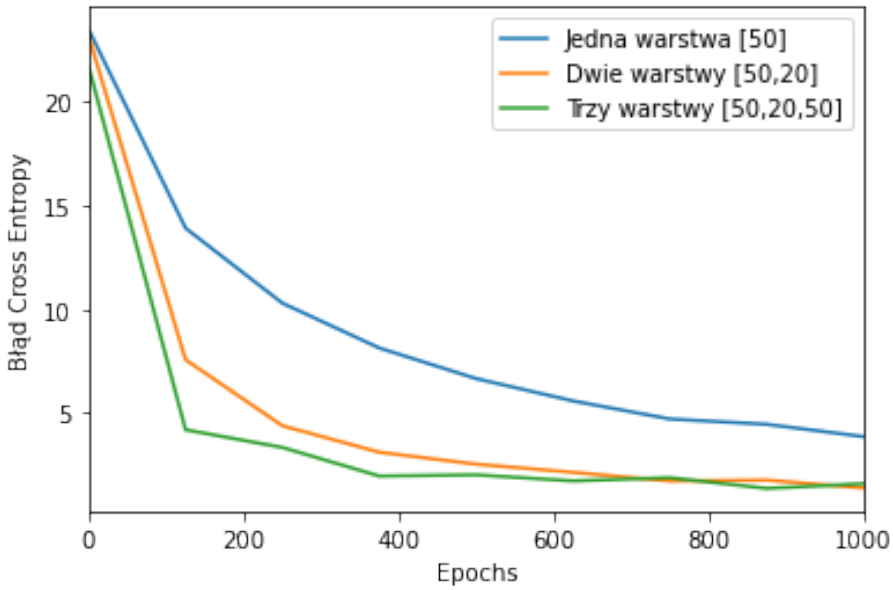


Figure 17: Porównanie wyników dla procesu uczenia zbioru "ring5"

Ilość warstw i neuronów dla nich	Jedna [50]	Dwie [50,20]	Trzy [50,20,50]
Cross entropy na zbiorze treningowym:	4.421	0.595	0.465
F1 score na zbiorze treningowym:	0.9648	0.9896	0.9344
F1 score na zbiorze testowym:	0.891	0.925	0.904

Podsumowując wszystkie testy na zbiorach do problemu klasyifkacji widzimy, że zaimplementowana sieć neuronowa przy wszystkich ulepszeniach potrafi uczyć się danych w bardzo efektywny sposób. W przypadku łatwiejszych jak i trudniejszych zadań otrzymaliśmy F1 score na poziomie 0.89-0.99 na zbiorze testowym, zaś na zbiorze treningowym sieć neuronowa skutecznie minimalizowała funkcję kosztu Cross entropy. Możemy wywnioskować również, że średnio dla naszych zbiorów skuteczna jest architektura dwóch warstw ukrytych po liczbie neuronów 50,20. Ponadto użyteczne było również użycie metody batching, optymalizacji kierunków poprzez algorytm Adam oraz korzystanie z funkcji aktywacji tanh jak i metody inicjowania wag "he". Korzystanie z trzech warstw ukrytych doprowadzało do skuteczniejszej minimalizacji funkcji kosztu, lecz dawało gorsze wyniki na zbiorze testowym, co może wskazywać na pojawianie się przetrenowania zbioru. Reasumując, niezbędne jest zbadanie wszystkich parametrów dla konkretnego zbioru, aby osiągnąć najlepszy proces uczenia.

7 Problem regresji

W problemie regresji rozpatrywać będziemy błąd średnio kwadratowy jako funkcję kosztu. Ponadto dla warstw ukrytych stosować będziemy funkcję aktywacji relu, zaś na wyjściu funkcję liniową. Po przetesowaniu różnej liczby warstw ukrytych, dla dwóch rozaptrywanych zbiorów dostaliśmy następujące rezultaty dla zbioru "multimodal large".

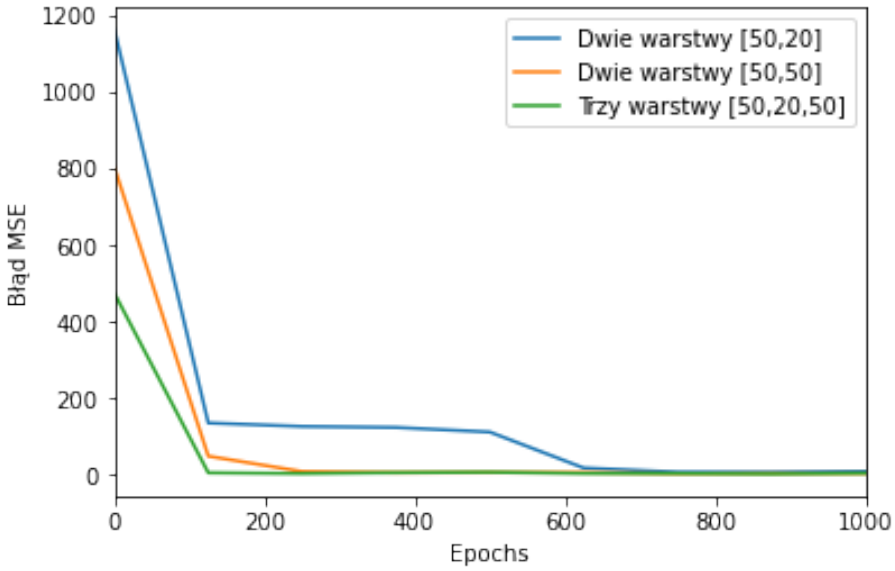


Figure 18: Porównanie wyników dla procesu uczenia zbioru "multimodal large"

Ilość warstw i neuronów dla nich	Dwie [50,20]	Dwie [50,50]	Trzy [50,20,50]
MSE na zbiorze treningowym:	9.501	4.202	4.473
MSE na zbiorze testowym:	4.61	9.29	17.5

Widzimy na powyższym wykresie, że sieć neuronowa z dwoma warstwami [50,20], mimo iż zaczęła minimalizować błąd znacznie później niż pozostałe sieci, to dała najlepszy wynik na zbiorze testowym. Jednak sieć z trzema warstwami osiągnała na zbiorze testowym nagorszy wynik MSE, a treningowym wynik na poziomie najlepszego. Możemy wysnuć na tej podstawie wniosek, że im większa liczba warstw ukrytych, oraz im więcej w nich neuronów, tym sieć neuronowa jest bardziej podatna na overfitting.

Ostatnim zbiorem do testów był "steps-large". Ponownie zastowaliśmy analogiczne parametry jak w poprzednim przykładzie. W tym przypadku wyniki okazały się bardzo podobne dla każdej sieci i najlepszą okazała się być ponownie ta, o najprostrzej strukturze.

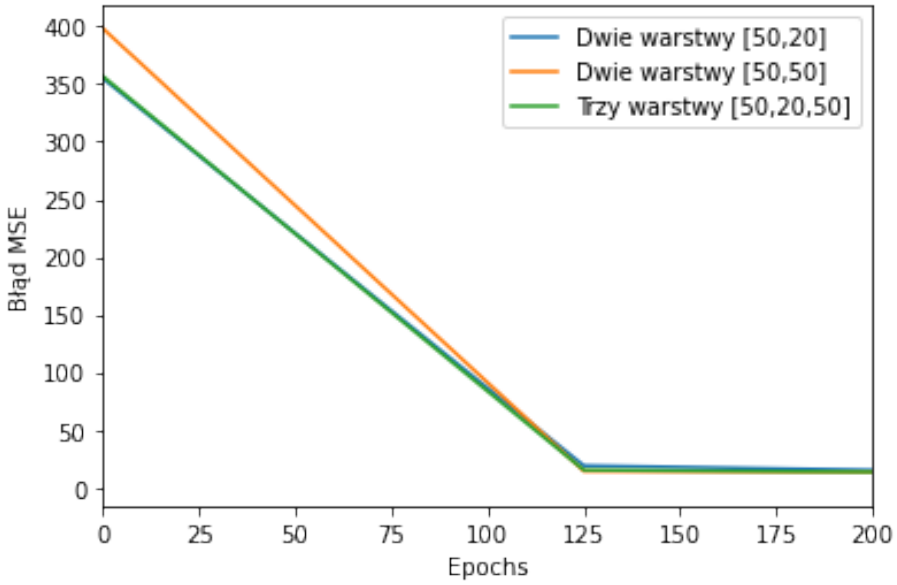


Figure 19: Porównanie wyników dla procesu uczenia zbioru "steps-large"

Ilość warstw i neuronów dla nich	Dwie [50,20]	Dwie [50,50]	Trzy [50,20,50]
MSE na zbiorze treningowym:	7.48	7.90	6.445
MSE na zbiorze testowym:	5.971	9.117	9.402

8 Podsumowanie

Na podstawie wszystkich testów widzimy, że dodanie ulepszeń takich jak normalizację, metody kierunków spadku gradientu i wiele innych do sztucznej sieci neuronowej jest obowiązkowym krokiem, aby otrzymać satysfakcjonujące rezultaty. Mimo to takie implementacje nie gwarantują najlepszych wyników, ponieważ dla każdego zbioru należy określić odpowiednią strukturę warstw ukrytych, liczbę neuronów dla nich, funkcję aktywacji i inne parametry, przez co nasuwa się wniosek, że dopiero przebadanie każdej z możliwości dałoby nam oczekiwaną efektywność sieci. Z optymistecznej strony jednak, przebadanie już kilku możliwości dało nam miarę F1-Score na poziomie 0.9-0.99 oraz miarę MSE na poziomie 0.5-10, co daje, że sieci neuronowe są bardzo skutecznymi metodami do problemów klasyfikacji i regresji.