20.05.2022
Sofiia Levchenko
Małgorzata Sikora

Sztuczne sieci neuronowe

1. Wstęp

Celem zadania była implementacja sieci perceptronu wielowarstwowego o wybranym algorytmie optymalizacji gradientowej z algorytmem propagacji wstecznej i przetestowanie jego działania na ogólnodostępnym zbiorze danych MNIST.

Zadanie zrealizowano w sposób ogólny tak, aby sieć dało się testować również na innych zbiorach danych.

2. Struktura projektu

Ze względu na specyfikę wykonywanego zadania, zdecydowałyśmy się, że struktura projektu będzie wyglądała w następujący sposób:

- ➔ *main.py* - zawiera wywołania eksperymentów przeprowadzonych w trakcie realizacji zadania
- ➔ *experiments.py* - funkcje z definicjami eksperymentów przeprowadzonych w trakcie zadania
- ➔ *multilayer_perceptron.py* - implementacja wielowarstwowego perceptronu (naszej sieci neuronowej) w postaci klasy MultilayerPerceptron.
- ➔ *neuron.py* - implementacja pojedynczego węzła warst ukrytych oraz warstwy wyjściowej sieci neuronowej w postaci klasy Neuron.
- ➔ *sgd.py* - implementacja funkcji pomocniczych, wykorzystywanych do obliczania gradientu na potrzeby zmiany wag węzłów sieci neuronowej.
- ➔ *activation_funcions.py* - reprezentacja używanych funkcji aktywacji przy pomocy biblioteki sympy (zostały zaimplementowane takie funkcje jak tnh, sigmoid oraz ReLU).
- ➔ *get_data.py* - funckje pomocnicze, wykorzystywane w celu załadowania oraz przetworzenia (normalizacji wartości oraz oddzielenia od zbioru danych nagłówka z nazwami kolumn) zbioru danych, na których następnie trenowaliśmy oraz sprawdzaliśmy sieć neuronową.
- ➔ *plot_data.py* - funkcije pomocnicze pozwalające na stworzenie wykresów - krzywej uczenia oraz macierzy pomyłek

W ramach projektu były wykorzystywane różne biblioteki zewnętrzne, ułatwiające wykonanie zadania:

- numpy, math - do działań na różnych strukturach danych oraz przeprowadzenia na nich obliczeń/działań matematycznych.
- sympy - do przedstawienia w czytelny sposób funkcji aktywacji oraz obliczanie ich gradientu
- random - do generowania liczb losowych na podstawie których wybierane są próbki do uczenia sieci neuronowej
- sklearn, pandas - do przetwarzania/załadowania zbiorów danych
- mlextend - biblioteka zawierająca narzędzie pozwalające w łatwy sposób automatycznie generować ilustrację macierz pomyłek.
- matplotlib - biblioteka ułatwiająca tworzenie wykresów

3. Decyzje projektowe

Główna część implementacji problemu zawarta jest w dwóch klasach - Neuron oraz MultilayerPerceptron.

Klasa Neuron stanowi reprezentację pojedynczej jednostki sieci przetwarzającej informację, jego główne atrybuty to:

- używana funkcja aktywacji
- wagi wejściowe (losowane podczas inicjalizacji z rozkładu jednostajnego)
- wartość bias

Również zostały wprowadzone dodatkowe atrybuty na potrzeby przeprowadzenia propagacji wstecznej:

- wartość wyjścia (po przejściu przez funkcję aktywacji)
- wartość wyjścia (przez przejściem przez funkcję aktywacji)

Wspomniana klasa Neuron również posiada również kilka metod (oprócz funkcji do inicjalizacji bias'u oraz wag), pozwalających na zmianę wartości wag oraz bias'u podczas przeprowadzenia propagacji wstecz. Do tych funkcji wprowadzany są obliczony gradient oraz wartość kroku (parametr beta), na podstawie których obliczane są nowe wartości wag oraz bias'u.

Klasa MultilayerPerceptron stanowi reprezentację całej sieci. Zawiera metody pozwalające na inicjalizację sieci, trenowanie sieci, czy uzyskiwanie przewidywań na podstawie próbki danych wejściowych. Dana klasa pozwala na stworzenia sieci neuronowej o dowolnej liczbie wejść, wyjść, warstw ukrytych oraz liczbie węzłów w środku takiej warstwy ukrytej.

Najważniejsze z punktu widzenia działania programu atrybut tej klasy to:

- network - lista list reprezentująca sieć neuronową. Każdy jej wiersz odpowiada jednej warstwie. Elementy warstwy stanowią obiekty klasy Neuron, przykładowo dla 3 warstw i 2 neuronów w każdej warstwie:

  [[neuron_11, neuron_12], [neuron_21, neuron_22], [neuron_31, neuron_32]]

  Funkcja inicjalizująca pozwala wybrać dowolną liczbę warstw i ustawić różną liczbę neuronów na każdej z nich (poza ostatnią, w której liczba neuronów jest automatycznie dostosowywana do wymiaru wyjścia z sieci). Można też każdej warstwie ustawić inną funkcję aktywacji (identyczną dla wszystkich neuronów w tej warstwie).

Trenowanie sieci, reprezentowanej za pomocą klasy MultilayerPerceptron odbywa się na zasadzie propagacji wstecznej, gdzie najpierw jest obliczana pomyłka dla warstw wyjściowych, a następnie na jej podstawie jest obliczana odchyłka dla każdej z wag oraz biasów (którą uśredniamy dla N próbek z paczki) dla każdej kolejnej warstwy (od ostatniej do pierwszej) na zasadzie reakcji łańcuchowej.

4. Opis wykonanych eksperymentów

Ponieważ na chwilę obecną nie istnieją metody automatycznej generacji hiperparametrów sieci neuronowej dla wszystkich możliwych zadań (typu regresji czy klasyfikacji) i wszystkich możliwych danych, konfiguracja odbyła się metodą prób i błędów.

Zaczęłyśmy od testowania naszej sieci dla funkcji prostszych, wymagających prostszych konfiguracji sieci neuronowej (funkcji OR oraz AND), następnie dla bardziej skomplikowanego zbioru danych, gdzie sieć miałaby kilka wejść i jedno wyjście (klasyfikator zero-jedynkowy wykorzystywany do predykcji czy jakieś mieszkanie o określonych cechach ma cenę wyższą od średniej, czy jednak nie), na koniec ukształtowałyśmy naszą sieć w taki sposób aby mogła przetworzyć i przewidywać wartości dla docelowego zbioru MNIST.

Dla pierwszego typu eksperymentów, po przeprowadzeniu kilku prób, najlepszym zestawem parametrów był:

- Ustawienie 1 warstwy ukrytej o 3 węzłach
- Wykorzystanie funkcji aktywacji tanh dla warstwy ukrytej oraz logistic dla warstwy wyjściowej
- Wykorzystanie parametru beta równego 0.9

Dla drugiego typu eksperymentów, po przeprowadzeniu kilku prób, najlepszym zestawem parametrów był:

- Wykorzystanie 2 warstw ukrytych o 32 węzłach
- Wykorzystanie funkcji aktywacji tanh dla pierwszej warstwy ukrytej, ReLU dla drugie warstwy ukrytej oraz logistic dla warstwy wyjściowej
- Wykorzystanie parametru beta równego 0.4

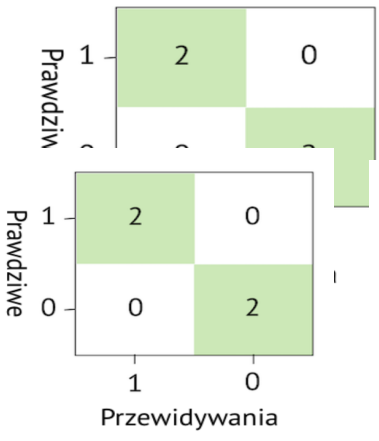
Dla trzeciego i ostatniego eksperymentu, po przeprowadzeniu kilku prób, najlepszym zestawem parametrów był:

- Wykorzystanie 2 warstw ukrytych o 192 węzłach
- Wykorzystanie funkcji aktywacji tanh dla pierwszej warstwy ukrytej, ReLU dla drugiej warstwy ukrytej oraz logistic dla warstwy wyjściowej
- Wykorzystanie parametru beta równego 0.4

5. Wyniki

Wyniki eksperymentu typu pierwszego (dla 40 epok, dla zbioru trenującego):

- Dla funkcji OR:

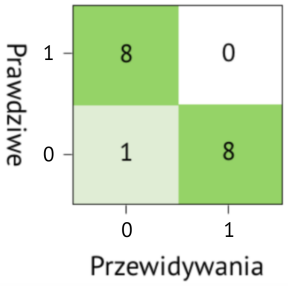


- Dla funkcji AND:

	Dane wejściowe w postaci [x1, x2]			
	[0, 0]	[0, 1]	[1, 0]	[1, 1]
Przewidziana klasa	0.139	0.919	0.906	0.979
Prawdziwa klasa	0	1	1	1

Wyniki eksperymenty typu drugiego (dla 80 epok):

- Dla zbioru trenującego:



LotArea	OverallQual	OverallCond	TotalBsmtSF	FullBath	HalfBath	BedroomAbvGr	TotRmsAbvGrd	Fireplaces	GarageArea	AboveMedianPrice	prediction
8450	7	5	856	2	1	3	8	0	548	1	0,894
9600	6	8	1262	2	0	3	6	1	460	1	0,887
11250	7	5	920	2	1	3	6	1	608	1	0,945
9550	7	5	756	1	0	3	7	1	642	0	0,228
14260	8	5	1145	2	1	4	9	1	836	1	0,976
14115	5	5	796	1	1	1	5	0	480	0	0,020
10084	8	5	1686	2	0	3	7	1	636	1	0,949
10382	7	6	1107	2	1	3	7	2	484	1	0,975
6120	7	5	952	2	0	2	8	2	468	0	0,949
7420	5	6	991	1	0	2	5	2	205	0	0,454
11200	5	5	1040	1	0	3	5	0	384	0	0,009
11924	9	5	1175	3	0	4	11	2	736	1	0,995
12968	5	6	912	1	0	2	4	0	352	0	0,005
10652	7	5	1494	2	0	3	7	1	840	1	0,946
10920	6	5	1253	1	1	2	5	1	352	0	0,448
6120	7	8	832	1	0	2	5	0	576	0	0,007
11241	6	7	1004	1	0	2	5	1	480	0	0,075

- Dla zbioru walidacyjnego:

LotArea	OverallQual	OverallCond	TotalBsmtSF	FullBath	HalfBath	BedroomAbvGr	TotRmsAbvGrd	Fireplaces	GarageArea	AboveMedianPrice	prediction
10552	5	5	1398	1	1	4	6	1	447	1	0,170
7313	9	5	1561	2	0	2	6	1	556	1	0,912
13418	8	5	1117	3	1	4	9	1	691	1	0,998
10859	5	5	1097	1	1	3	6	0	672	0	0,033
8532	5	6	1297	1	0	3	5	1	498	0	0,077
7922	5	7	1057	1	0	3	5	0	246	0	0,006
6040	4	5	0	2	0	2	6	0	0	0	0,025
8658	6	5	1088	2	0	3	6	1	440	0	0,373
16905	5	6	1350	1	1	2	5	2	308	1	0,799
9180	5	7	840	1	0	2	5	0	504	0	0,006
9200	5	6	938	1	0	3	5	0	308	0	0,006
7945	5	6	1150	1	0	3	6	0	300	0	0,008
7658	9	5	1752	2	0	2	6	1	576	1	0,926
12822	7	5	1434	1	1	1	6	1	670	1	0,171
11096	8	5	1656	2	0	3	7	0	826	1	0,977
4456	4	5	736	2	0	2	8	0	0	0	0,099
7742	5	7	955	1	0	3	6	0	386	0	0,007

Dla pierwszych dwóch typów przeprowadzanych eksperymentów nie tworzyłyśmy wykresów opisujących zależność poprawności wyniku/wartości straty od liczby epok. Celem tych eksperymentów było sprawdzenie, czy sieć potrafi się nauczyć na podanych do niej danych oraz, w razie potrzeby, poprawienie napisanego kodu, aby mieć pewność, że wyniki uzyskane dla zbioru MNIST będą wiarygodne.

5.1. Wyniki dla zbioru MNIST - pierwszy eksperyment

Pierwszy eksperyment stanowił wstępne badanie mające na celu określenie, czy wybrane wstępnie parametry mają szansę dać dobre wyniki na docelowym zbiorze MNIST po 30 epokach.

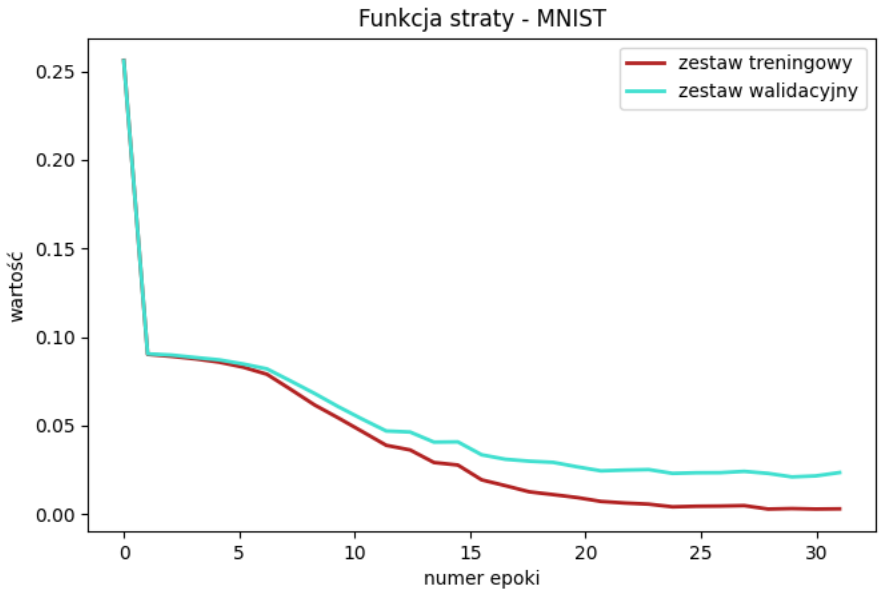
Z tego powodu w eksperymencie tym posłużono się bardzo ograniczoną próbką danych podzieloną na dwie części - zbiór treningowy o liczebności 99 próbek oraz zbiór walidacyjny o liczebności równej 51. Mały rozmiar zbioru pozwolił szybko uzyskać wstępne wyniki działania sieci.

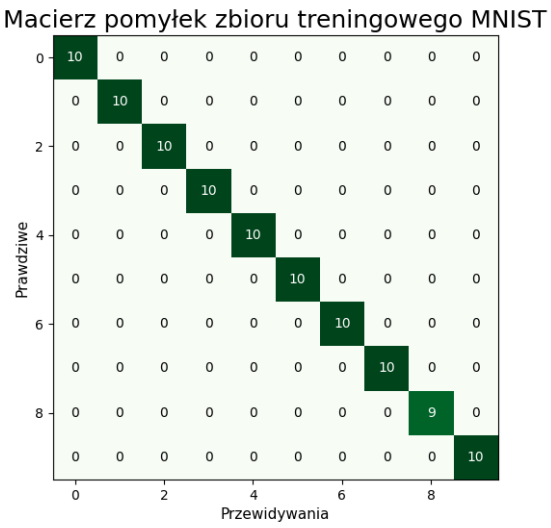
Hiperparametry dla tego treningu wynosiły:

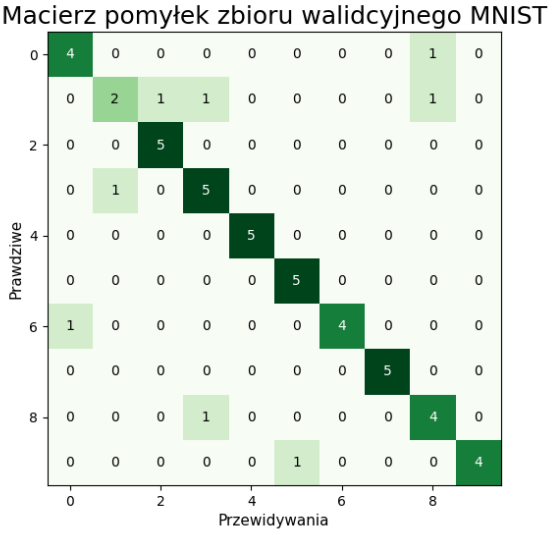
- 3 warstwy
  - funkcje aktywacji dla każdej to: [TANH, RELU, LOGISTIC]
  - liczebność każdej z warstw to: [192, 192, 10]

- 30 epok
- 40 batchy trenujących
- wartość beta uwzględniania poprawki równa 0,4

Wyniki przedstawiono na poniższych wykresach:







Jak widać, dla tak małego zbioru sieć nauczyła się go bez problemu. Nawet wyniki na zbiorze walidacyjnym są niezłe, chociaż pod koniec treningu wartość funkcji straty dla niego zmniejsza się wolniej. Z tego powodu kolejny eksperyment, dla większej próbki, przeprowadzono zmieniając ustawienia tylko nieznacznie

5.2. Wyniki dla zbioru MNIST - drugi eksperyment

Tym razem przeprowadzony został docelowy eksperyment na dużej próbce danych. Ze względu na bardzo długi czas obliczeń dla tej liczby neuronów, nie wykorzystano całego zbioru MNIST, a jedynie 400 próbek z niego pochodzących. Przy podziale upewniono się, że wybrany podzbiór zawiera równomierny rozkład wszystkich klas. Najmniej liczna klasa w tym zbiorze stanowiła 9,75 %, natomiast najbardziej liczna: 10,50%, co oznacza że był on zrównoważony.

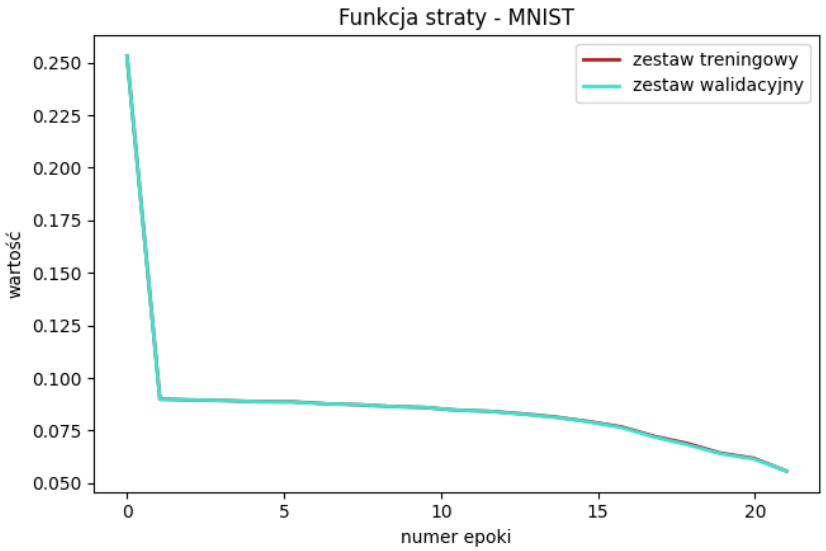
Zestaw był podzielony na trzy zbiory: Treningowy o liczebności 320, walidacyjny o liczebności 64 oraz testowy o liczebności 80.

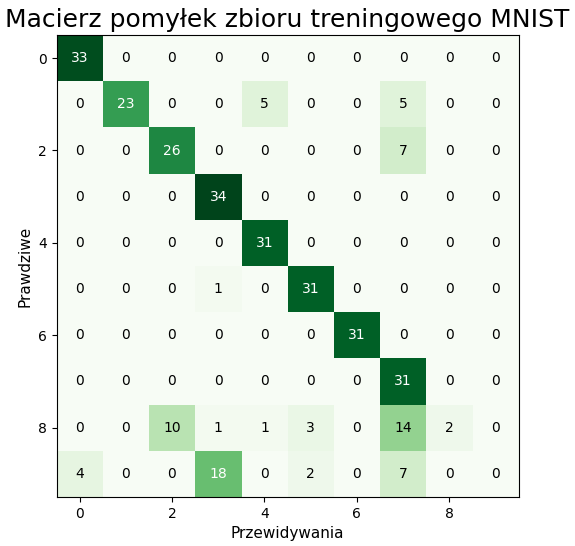
Hiperparametry dla tego treningu wynosiły:

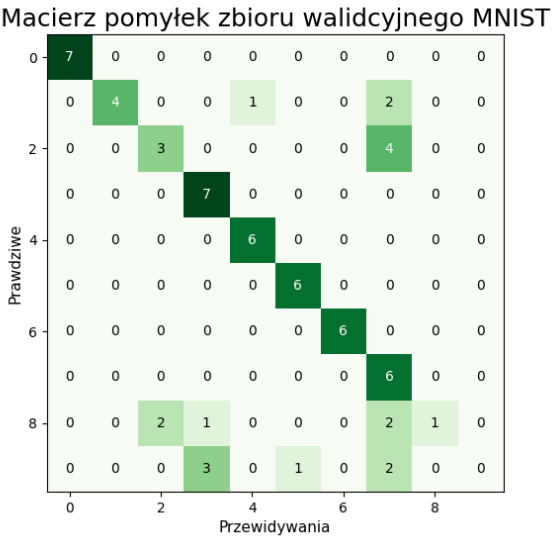
- 3 warstwy
  - funkcje aktywacji dla każdej z nich to: [TANH, RELU, LOGISTIC]
  - liczebność każdej z warstw to: [128, 128, 10]

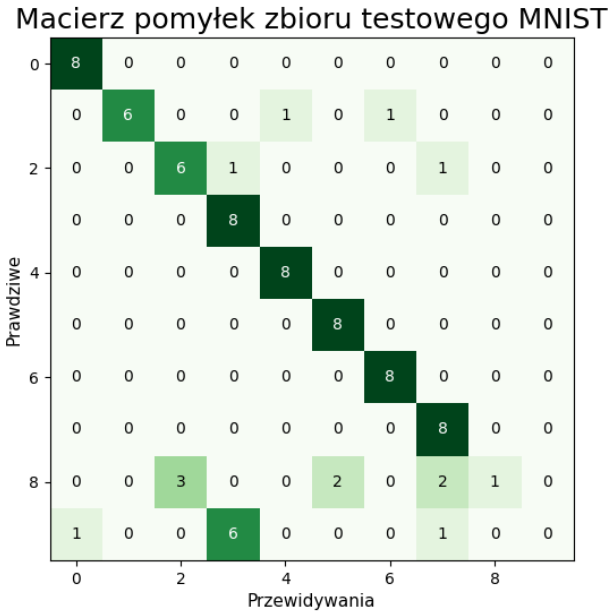
- 20 epok
- 40 batchy trenujących
- wartość beta uwzględniania poprawki równa 0,4

Wynikowe wykresy przedstawiono poniżej:









Z wykresu przedstawiającego krzywą uczenia wynika, że proces uczenia się sieci przebiegał prawidłowo i nie ulegała ona przeuczeniu ani niedouczeniu.

Macierze pomyłek przybliżają charakter błędów popełnianych przez klasyfikator. Jak widać ma on problem z rozpoznawaniem dwóch klas. Być może wynika to ze zbyt krótkiego czasu uczenia bądź zbyt małej liczby neuronów.

Pozostałe klasy perceptron rozpoznaje bez problemu, co zostało potwierdzone na zbiorze testowym (ostatnia macierz).

6. Wnioski

Stworzona implementacja pozwala w elastyczny sposób inicjalizować sieci o różnej liczbie warstw, neuronów, różnych funkcjach aktywacji i parametrach uczenia. Dzięki temu

można ją wykorzystywać dla zbiorów danych o różnej liczebności, stopniu skomplikowania, wymiarze danych wejściowych oraz różnej liczbie klas.

Po przeprowadzeniu wielu eksperymentów, mając na celu znaleźć odpowiednie parametry, dla których sieć będzie w stanie nauczyć się konkretnemu zbiorowi danych, zauważyliśmy kilka zależności między tymi parametrami i ich wpływem na jakość wytworzonej sieci:

- Dla mniejszych zbiorów danych treningowych nie jest efektywnym wykorzystywanie małych beta (wyjątkiem są sytuacje kiedy mamy proporcjonalnie większą liczbę epok) - wartości prawdopodobieństw dla różnych klas (w przypadku klasyfikatora) wtedy trzymają się na podobnym poziomie i widać tylko niewielkie odchyłki od tych wartości
- Również, dla małych zbiorów danych treningowych, istotną rolę gra liczebność batch'a - im mniejszy wtedy jest batch, tym jest to lepsze dla małych zbiorów danych, w przeciwnym przypadku uzyskujemy wynik uśredniony, nie odzwierciedlający rzeczywistość
- Nie zawsze wykorzystanie większej liczby warstw oznacza lepsze działanie sieci - czasami mniejsza liczba warstw z większą liczbą węzłów daje o wiele lepsze wyniki
- Wybór właściwej funkcji aktywacji dla każdej z warstw jest bardzo ważnym czynnikiem, mającym wpływ na działanie całej sieci (wykorzystanie niepasującej do problemu funkcji aktywacji w jednej z warstw może skutkować wielorazowym obniżeniem jakości przewidywanych wyników już nauczonej sieci)

Głównym problemem napotkanym podczas realizacji zadania był wyjątkowo długi czas obliczeń podczas treningu sieci. Częściowo wynika on z wyboru dużej liczby neuronów w każdej warstw a co za tym idzie - dużej liczby wag do aktualizacji w każdym kroku. (rzędu parunastu tysięcy dla warstwy).

Z pewnością implementacja, mogłaby być wydajniejsza, jednak wymagało by to kompletnej refaktoryzacji kodu oraz przemyślenia używanych w implementacji struktur danych.