# Raport końcowy

Sieci Neuronowe

Regresja i klasyfikacja za pomocą perceptronu wielowarstwowego

Bartłomiej Teodorczuk

Stanisław Wasilkowski

# Wstęp

## Zadanie

Zbudować program pozwalający na rozwiązywanie problemów regresji i klasyfikacji przy użyciu perceptronu wielowarstwowego uczonego algorytmem wstecznej propagacji błędu. Program pozwalać na wizualizację procesu uczenia sieci, jak i wyników względem zbioru uczącego: w przypadku regresji dla funkcji $R \rightarrow R$, a klasyfikacji dla wektorów $R^2$.
Wymagane jest, aby proces uczenia był powtarzalny, a sieć w pełni konfigurowalna (liczba warstw w sieci oraz neuronów w warstwie).

## Realizacja

W celu wykonania powyższego zadania stworzony został program w technologii Python 3.6.5. Jako perceptron wielowarstwowy została wykorzystana implementacja z biblioteki scikit-learn 0.20.0.
Aplikacja jest konfigurowalna z poziomu kodu. W pliku main.py, w metodzie main znajdują się zmienne zawierające: ścieżki do plików z zestawami uczącym (*trainFilePath*) i testowym (*testFilePath*), maksymalną liczbę iteracji (*max_iterations*), wartość mówiąca co ile iteracji ma być wizualizowana zmiana perceptronu w procesie uczenia (*visualize_every_iteration*), nazwę funkcji aktywującej neuron (*activation*) oraz liczby neuronów w poszczególnych warstwach ukrytych (*hidden_layer_sizes*). Za pomocą zmiennej *regression* można wybrać jakie zadanie jest rozwiązywane: regresja (*true*) czy klasyfikacja (*false*). Z kolei zmienne *perceptron_visualization*, *regression_visualization* oraz *classification_visualization* określają, czy włączona jest wizualizacja, odpowiednio procesu uczenia perceptronu, regresji i klasyfikacji.

# Testy

Na wstępie należy zaznaczyć, że wszystkie testy odbywają się na stałym, ustalonym ziarnie generatora pseudolosowego. Domyślną funkcją aktywacji neuronu jest funkcja sigmoid

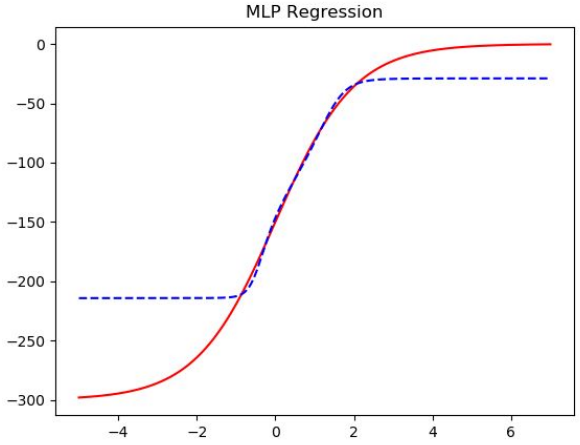
## Regresja

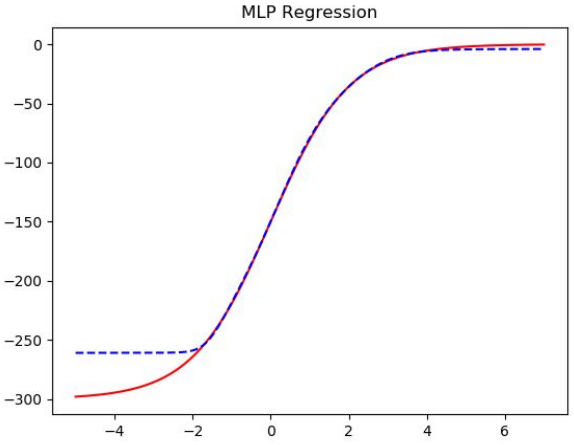
Regresja jest metodą pozwalającą na estymowanie wartości pewnej nieznanej nam funkcji na podstawie zbioru znanych wartości tej funkcji. W tym przypadku będą to funkcje $R \rightarrow R$.

Sprawdźmy skuteczność perceptronu wielowarstwowego w problemie na dostarczonych danych testowych. Testowane są dwa zbiory, dla dwóch różnych funkcji - *activation* oraz *cube*. Porównamy skuteczność i dokładność wyników w zależności od ilości punktów funkcji.

Przyjmijmy stałą ilość maksymalną iteracji, domyślną dla implementacji z scikit-learn - tj. 200 oraz dwie warstwy ukryte z 5 neuronami każda.
Porównywać będziemy błąd średni liczony dla wszystkich punktów podanych w zbiorach.

Dla funkcji *activation*:

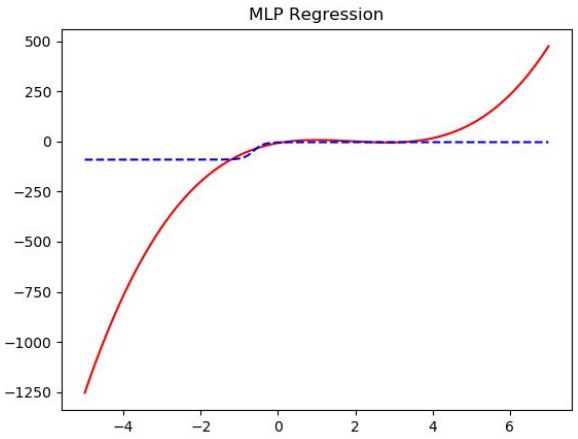


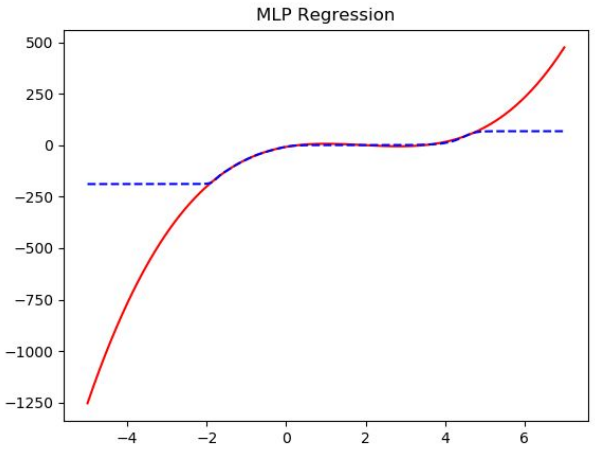




Ilość punktów = 100 Ilość punktów = 10 000
(czerwona ciągła linia - dane trenujące, niebieska przerywana linia - wartości przewidziane)

Na pierwszy rzut oka widać wzrost dokładności ze wzrostem ilości punktów w zbiorze uczącym. Warto dodać, że w przypadku zbioru uczącego o liczności 10 000 punktów zbieżność została osiągnięta po 66 iteracjach. Dokładne wyniki widać na poniższym wykresie:

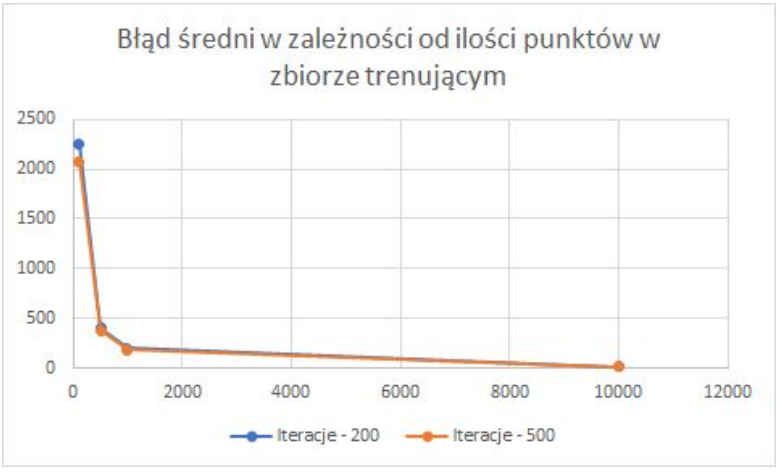
Dla funkcji *cube*:




Liczba punktów = 100

liczba punktów = 10 000

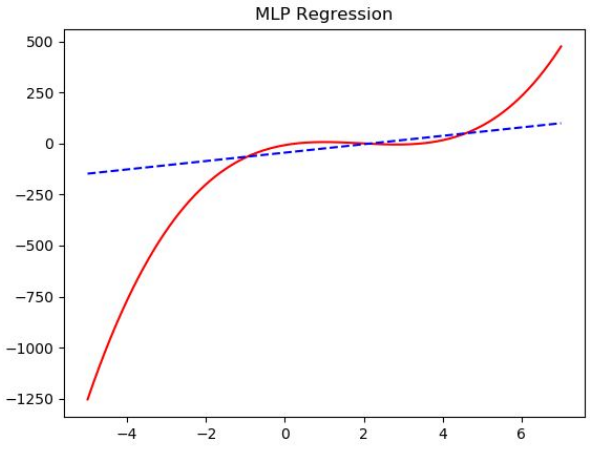



Tym razem również obserwujemy poprawę, jednak nie tak wyraźną, jak w przypadku poprzedniej funkcji. Naszym przypuszczeniem było to, że być może sieć nie osiąga zbieżności w określonej liczbie iteracji. Porównaliśmy zatem wyniki dla liczby iteracji 200 oraz 500. Na wykresie widać, iż nie jest to przyczyną średnich wyników pracy sieci - błędy są porównywalne (takie same w przypadku próby 10 000 - zbieżność po 164 iteracjach).

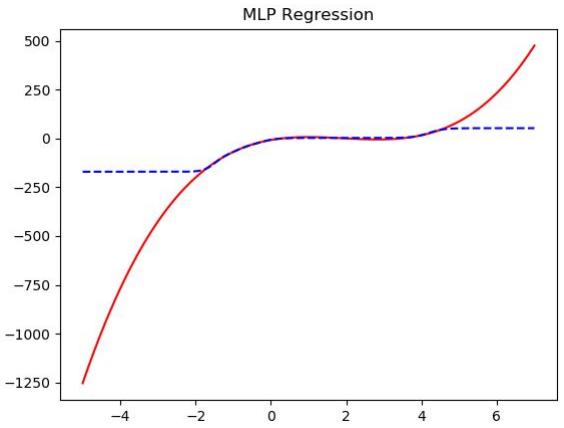
Widać, że perceptron dość dobrze spisuje się jeśli chodzi o zadanie regresji.

W obu przypadkach interesujący jest fakt, że sieć nie radzi sobie z wartościami funkcji odbiegającymi od 0 - wykres funkcji zaczyna się prostować i zbiegać do pewnej wartości.

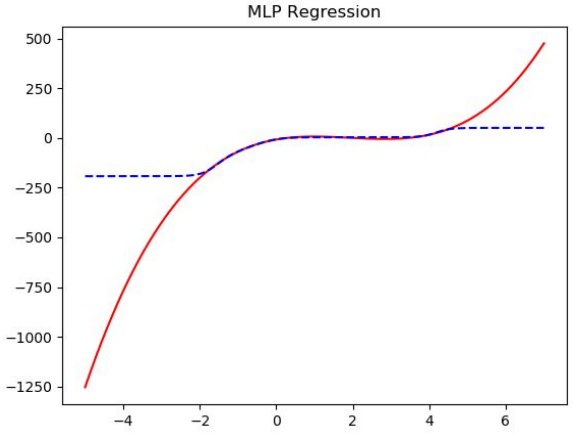
Sprawdźmy więc czy analogicznie zachowa się w przypadku zmiany konfiguracji sieci - ilości warstw ukrytych oraz liczby neuronów w sieci. Dla ułatwienia zajmijmy się tylko zbiorem trenującym o liczności 1000 punktów.



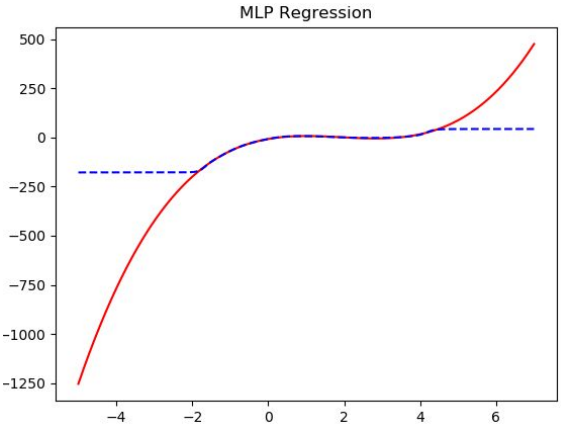

Brak warstw ukrytych, błąd: 206.01




Jedna warstwa ukryta 5 neuronów, błąd: 180.38



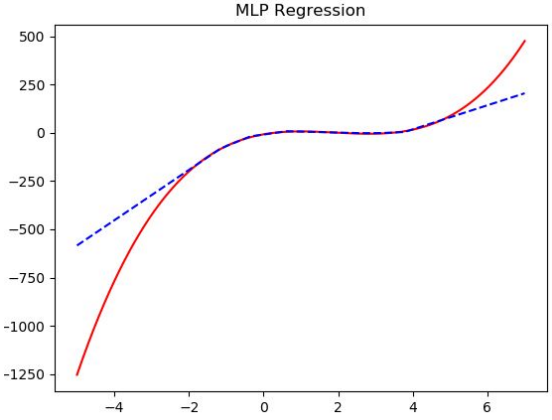

Jedna warstwa, 10 neuronów, błąd: 174.34




Dwie warstwy po 10 neuronów, błąd: 179.14

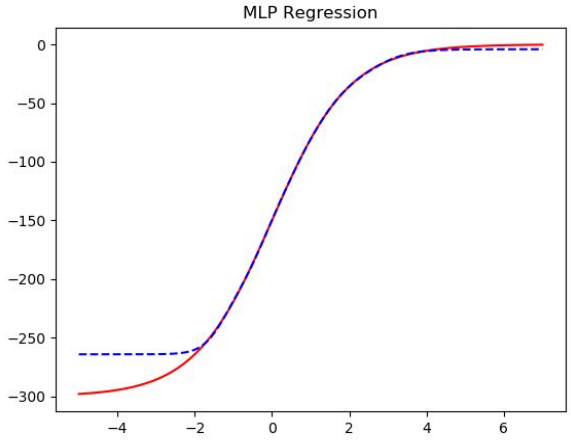
Widać wyraźnie, że poza przypadkiem bez warstw ukrytych wyniki są porównywalne i powtarzają się dla podobnych przypadków - powtarzając eksperyment dla jeszcze większych ilości warstw i neuronów w warstwie można nawet zaobserwować wzrost wartości błędu. Oznacza to, iż algorytm wstecznej propagacji raczej nie radzi sobie ze skutecznym uczeniem perceptronu o wielu warstwach.

Test nie wskazał również rozwiązania zaobserwowanego wcześniej problemu. Spróbujemy w takim razie zmienić funkcję aktywacji z funkcji logistic (sigmoid) na inną dostępną w bibliotece scikit-learn, np “relu”, czyli f(x) = max (0,x). Przyjmijmy parametry takie jak poprzednio oraz jedną warstwę ukrytą z 10 neuronami.

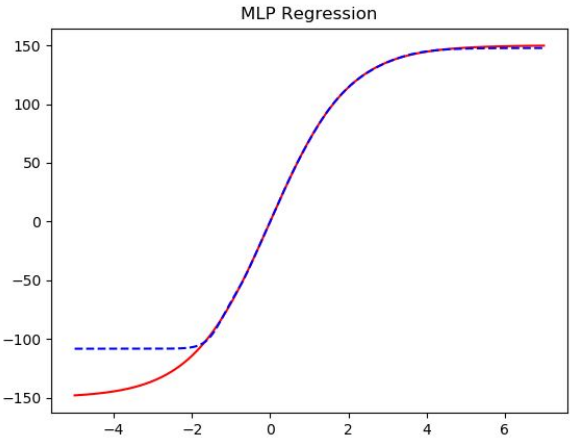



Już przy pierwszej próbie widać wyraźną poprawę. Błąd średni wynosi 95.42 co jest znaczącą poprawą w porównaniu z wynikami oscylującymi poniżej 200 dla podobnych parametrów dla funkcji aktywacji sigmoid. Oznacza to, że wina leży po stronie funkcji aktywacji. Jaka jest tego przyczyna?

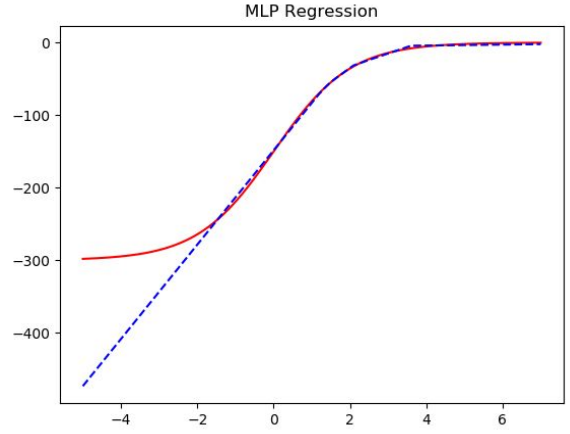
Zwróciliśmy uwagę na to, że w przypadku pierwszej funkcji błąd jest znacząco większy dla końca wykresu dążącego do wartości -300 niż końca dążącego do 0. Biorąc pod uwagę przedział wartości funkcji sigmoid, może być to spowodowane potencjalnie większą dokładnością dla wartości bliskich zera. Spróbujmy więc przesunąć wykres funkcji tak, aby jego środek znajdował się w okolicach 0 (parametry bez zmian).



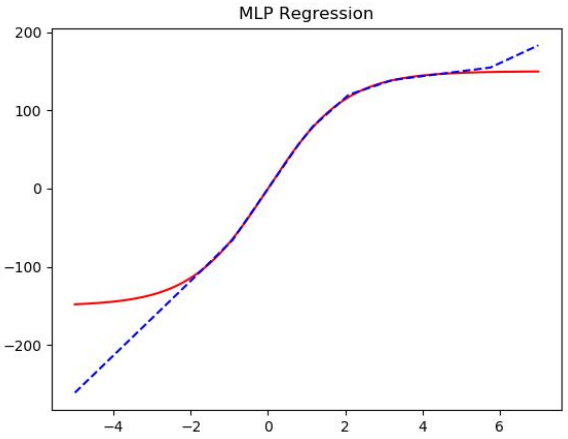

sigmoid, brak przesunięcia




sigmoid, z przesunięciem +150

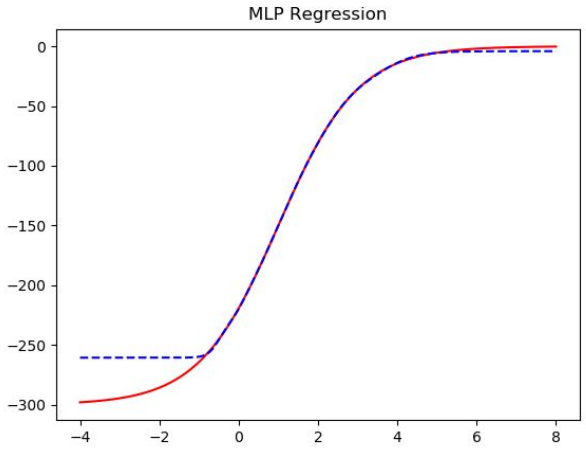



relu, brak przesunięcia




relu, z przesunięciem +150

Widzimy, że nie przyniosło to oczekiwanego rezultatu. Podobne zachowanie można zaobserwować przesuwając wykres powyżej 0, o +300 - tj. koniec w okolicach zera “urywa się” wcześniej niż przeciwny. Oznacza to, że nie wartości ujemne bądź oddalone od osi zera nie są winne temu problemowi i że ten problem również nie zależy od funkcji aktywacji - zachowanie jest podobne.
Przy okazji widzimy, że funkcja aktywacji relu gorzej poradziła sobie z względnie łatwą funkcją ze zbioru *activation.*




Uwagę zwraca fakt, że niezależnie od parametrów oraz przesunięć “wypłaszczenie” następuje w okolicy argumentu -2. Sprawdziliśmy więc co stanie się w przypadku przesunięcia wartości na osi X o +1. Jak widać

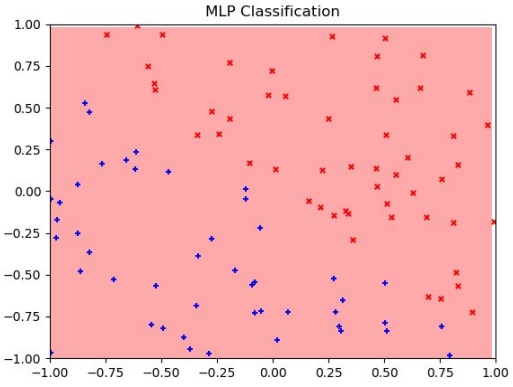
zmieniło to tylko argument w okolicy którego pojawia się problem na argument -1.
Widać, że opisany problem zależy od funkcji aktywacji i wykres funkcji przewidywanej wygląda podobnie do funkcji aktywacji. Pozostaje pytanie dlaczego jeden koniec wykresu zbiega inaczej niż drugi?
Niestety, tutaj kończą się nasze pomysły skąd bierze się problem. Żadne z przeprowadzonych testów nie pokazały istotnej zmiany i nie wskazały przyczyny problemu. Domysły iż może być to spowodowane funkcją aktywacji, bądź jej wartościowaniem są prawidłowe, ale nie dają dostatecznej informacji i przyczynie. Dane również wydają się być prawidłowe. Nie byliśmy w stanie stwierdzić, co jest przyczyną takiego zachowania.

# Klasyfikacja

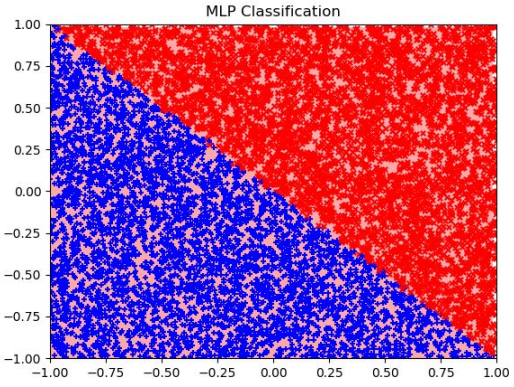
Problem klasyfikacji polega na przypisaniu wektora cech do jednej ze wzajemnie rozłącznych klas na podstawie znanych nam przynależności wektorów ze zbioru uczącego. W tym przypadku rozpatrywać będziemy wektory z przestrzeni $R^2$ .

Na początku przetestujemy skuteczność perceptronu dla zadania klasyfikacji w zależności od wielkości zbiorów uczących i testowych dla dwóch przykładów: *simple* i *three_gauss*. Maksymalna liczba iteracji będzie wynosić 200, a sieć będzie mieć dwie warstwy ukryte po 50 neuronów. Funkcją aktywacji będzie funkcja sigmoidalna.

Przykład *simple*:



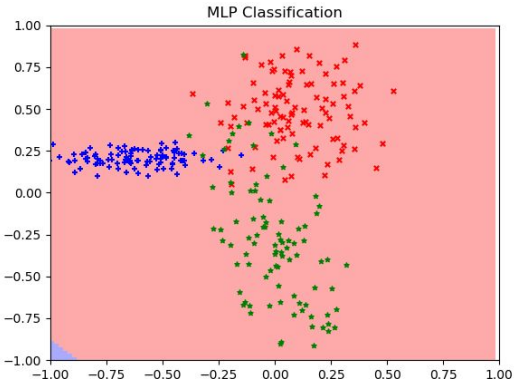

100 punktów, skuteczność 53%

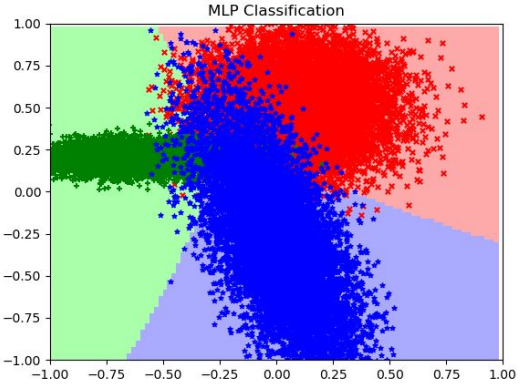



10 000 punktów, skuteczność 50.42%

Dla tego przykładu w obu przypadkach skuteczność jest porównywalna, niestety jest to skuteczność na poziomie klasyfikacji losowej.

Przykład *three_gauss*:



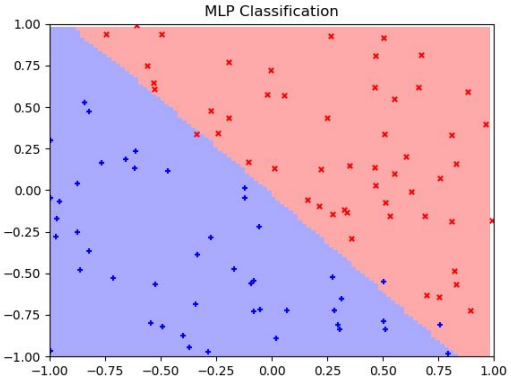



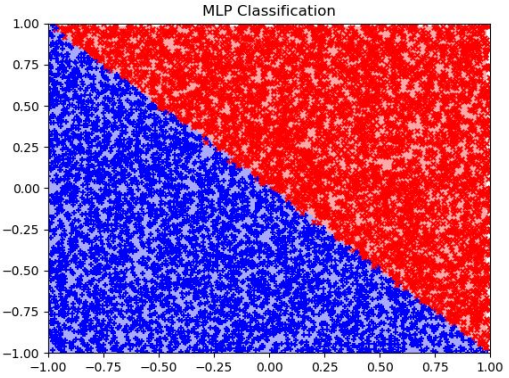

100 punktów, skuteczność 33,33%

10 000 punktów, skuteczność 92.76%

Tutaj przy większej ilości punktów osiągnięta skuteczność jest już na całkiem niezłym poziomie, przy 100 jest, tak jak dla pierwszego przykładu, beznadziejna.
Ponieważ wyniki tego eksperymentu okazały się niezadowalające, powtórzymy go dla innej funkcji aktywacji, na przykład funkcji tangens hiperboliczny.

Przykład *simple*:

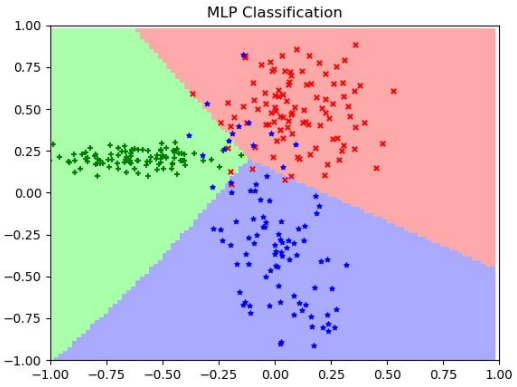


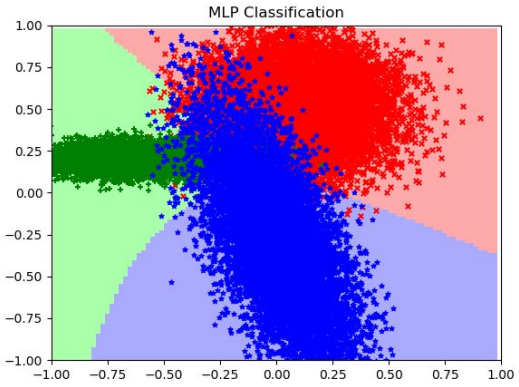

100 punktów, skuteczność 97%

10 000 punktów, skuteczność 99,67%

Przykład *three_gauss*:



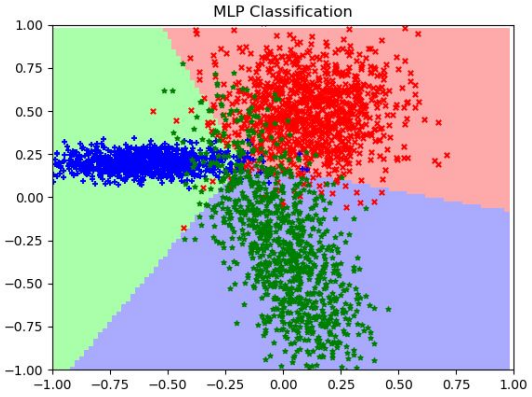

100 punktów, skuteczność 97%



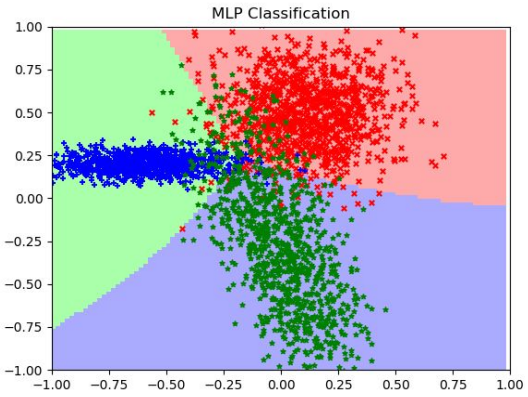

10 000 punktów, skuteczność 99,67%

Jak widać zmiana funkcji aktywacji znacząco poprawiła skuteczność, szczególnie przy małej ilości punktów w zbiorze uczącym.

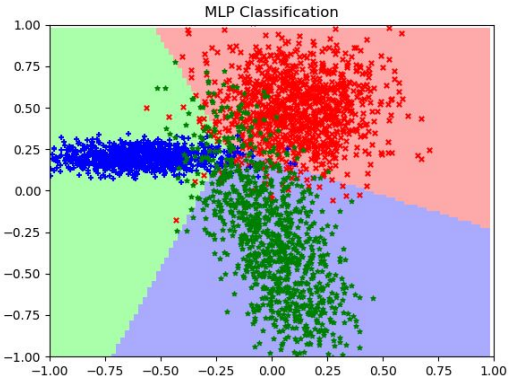
Następnym krokiem będzie sprawdzenie wpływu liczby warstw ukrytych oraz neuronów w warstwie na skuteczność klasyfikacji. Funkcją aktywacji będzie nadal tangens hiperboliczny, maksymalna liczba iteracji: 5000. Skorzystamy z przykładu *three_gauss* o wielkości 1000 wektorów.



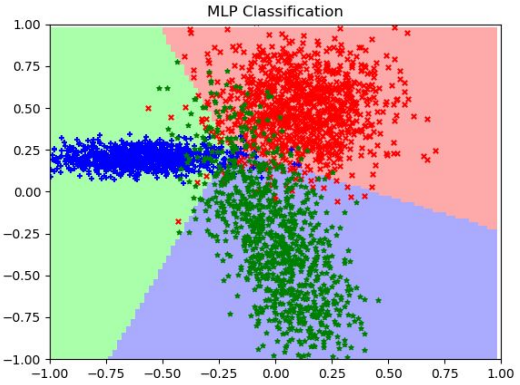

Bez warstw ukrytych, sk. 92,56%, 747 iteracji



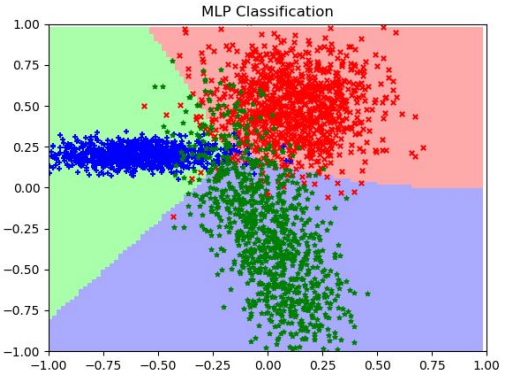

10 neuronów, 1 w., sk. 92,66%, 337 iteracji



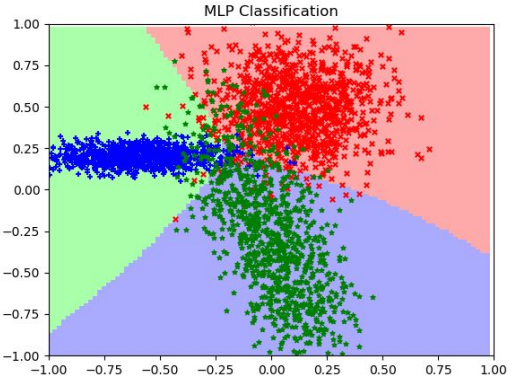

50 neuronów, 1 w., sk. 92,65%, 310 iteracji




150 neronów, 1 w., sk. 92,6%, 309 iteracji




2 w. po 10 neuronów, sk. 92,8%, 305 iteracji



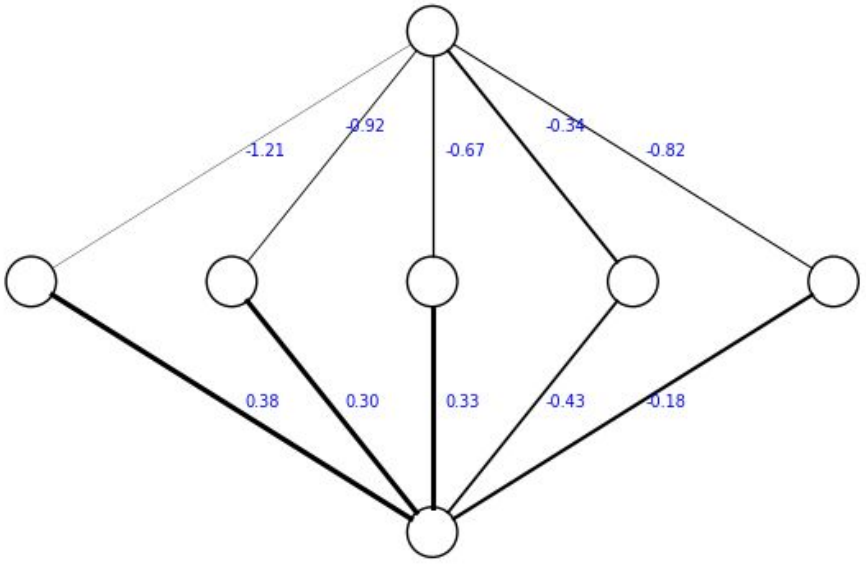

3 w. po 10 neuronów, sk. 92, 77%, 352 iteracji

Jak widać w każdym przypadku poprawność działania klasyfikatora jest dosyć wysoka, jedyne, co możemy zaobserwować, to ponad dwukrotny spadek liczby iteracji przy zastosowaniu przynajmniej jednej warstwy wewnętrznej.

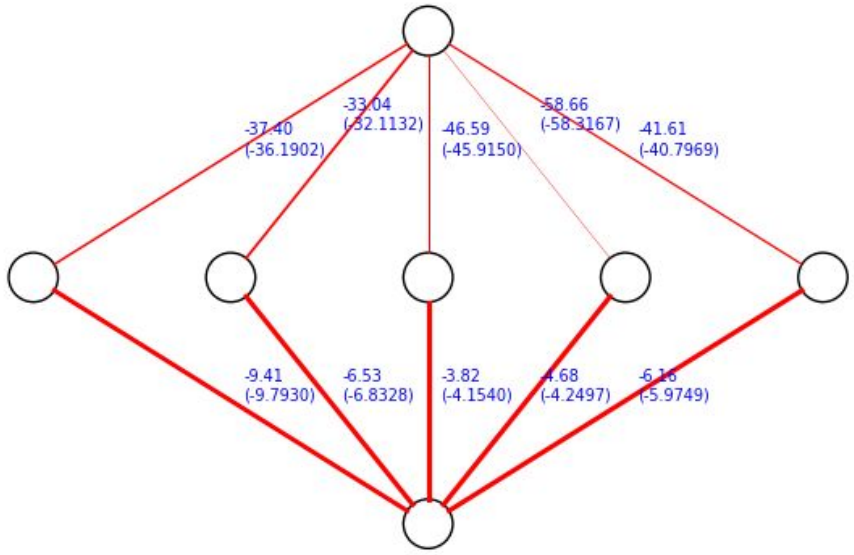
# Wizualizacja wag perceptronu

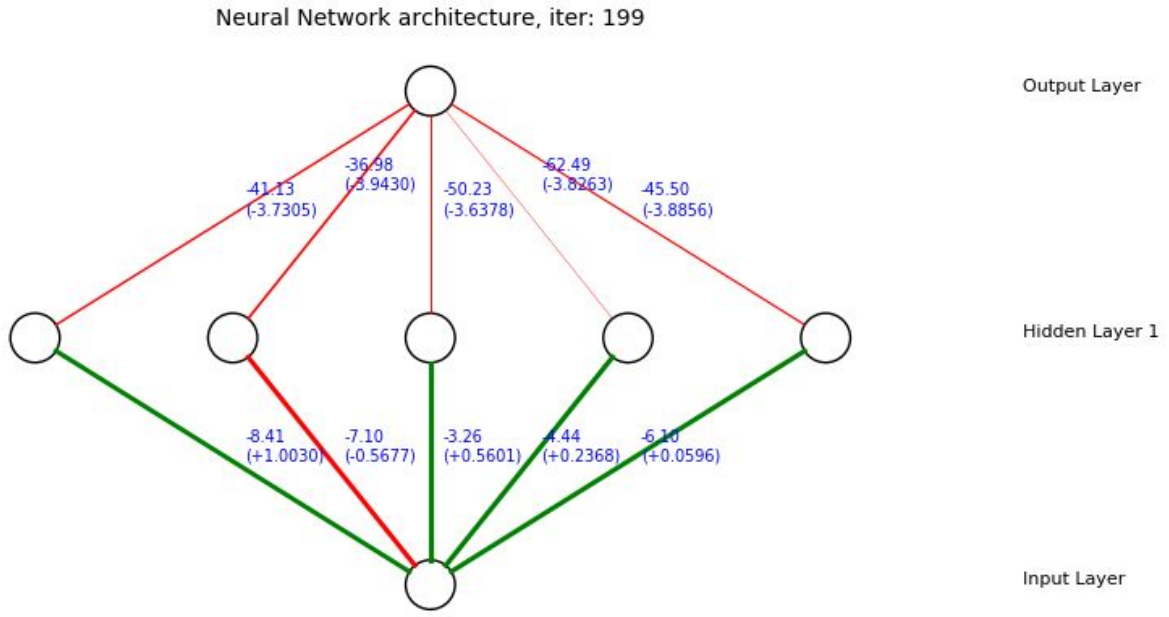
Jedną z wymaganych funkcjonalności programu niezaprezentowaną jeszcze w raporcie jest wizualizacja wag perceptronu oraz ich zmian pomiędzy kolejnymi iteracjami, wynikającymi z błędu propagowanego przez algorytm wstecznej propagacji błędu. Funkcjonalność pozwala na ustawienie co ile iteracji uczenia chcemy podejrzeć zmiany w wagach perceptronu. Dodatkowo zawsze zobaczymy stan początkowy oraz końcowy, po ostatniej iteracji.
Przykładowo, dla problemu regresji omawianego wcześniej dla 200 iteracji i wizualizacji co 100 iteracji z jedną warstwą ukrytą z 5 neuronami (dla czytelności wizualizacji) otrzymamy takie wizualizacje:

Neural Network architecture, iter: 0




Neural Network architecture, iter: 100






Grubość krawędzi symbolizuje wartość jej wagi - wartość jest opisana na niebiesko (w nawiasie zmiana od ostatniej wizualizacji). Grubość krawędzi jest znormalizowana ze względu na wartości wag, tj. krawędź o najmniejszej wadze jest najcieńsza, o największej - najgrubsza.
Początkowo wszystkie krawędzie są oznaczone kolorem czarnym oraz nie widac zmian ich wag.. Kolor czerwony oznacza spadek wagi w porównaniu z poprzednią wizualizacją, kolor zielony oznacza wzrost wartości wagi danej krawędzi.

# Podsumowanie

Z przeprowadzonych testów wynika, że perceptron wielowarstwowy oraz uczenie z wykorzystaniem algorytmu wstecznej propagacji błędu, mimo że jest niezbyt skomplikowane (w porównaniu z bardziej zaawansowanymi metodami sztucznej inteligencji wręcz prymitywne) dobrze sprawdza się jako narzędzie do rozwiązywania prostych problemów klasyfikacji oraz regresji. Mimo, że wystąpiły problemy, to przy odpowiedniej konfiguracji sieci można dojść do stanu, gdzie rozwiązanie osiąga zadowalające, a nawet dobre wyniki.

Zauważyliśmy dwa znaczące problemy:

a) dla zagadnienia regresji - problem z wykresem funkcji przewidywanej, który dopasowuje się do wykresu funkcji przybliżanej tylko w środku przedziału, a na brzegach przedziału odbiega od prawidłowych wartości.

b) dla zagadnienia klasyfikacji - problem z bardzo niską skutecznością funkcji aktywacji sigmoid.

Niestety, na podstawie testów przez nas przeprowadzonych, nie udało nam się wskazać jednoznacznych przyczyn i odpowiedzieć na pytanie dlaczego tak się dzieje.