

Sprawozdanie 1

Sieci Neuronowe

MIOwAD

Mateusz Nizwantowski
01161932@pw.edu.pl
313839

13 kwietnia 2024

Spis treści

1	Wstęp	3
2	Teoria	3
3	Eksperymenty	4
3.1	NN1	4
3.2	NN2	6
3.3	NN3	9
3.4	NN4	11
3.5	NN5	13
3.6	NN6	18
3.7	Dodatkowe	21
4	Podsumowanie	23

1 Wstęp

W ramach przedmiotu Metody Inżynierii Obliczeniowej w Analizie Danych poznaliśmy temat feedforward neural network. Dla wielu z nas było to pierwsze spotkanie z tą tematyką. Celem jest implementacja from scratch podstawowej sieci neuronowej. Co tydzień mieliśmy do wykonania różne etapy, które rozbudowywały dotychczasowe rozwiązanie. Dodatkowo testowaliśmy naszą sieć na różnorodnych problemach, które obrazowały pewne zjawiska takie jak przeuczenie. W ramach tych zadań zmagaliśmy się z takimi wyzwaniami jak:

- stworzenie szkieletu sieci
- implementacja backpropagacji
- różne algorytmy optymalizacji
- klasyfikacja
- różne funkcje aktywacji
- regularyzacja

Na końcu naszym finalnym zadaniem jest opisanie naszych doświadczeń w postaci raportu. Przedstawienie co udało nam się osiągnąć, jakie problemy napotkaliśmy, czego się nauczyliśmy i w jaki sposób planujemy rozwijać wiedzę zdobytą podczas tego etapu. Cały kod, który napisałem w ramach tego projektu znajduje się w [repozytorium](#) na GitHub.

2 Teoria

Sieci neuronowe stanowią potężne narzędzie w dziedzinie sztucznej inteligencji, inspirowane biologicznym funkcjonowaniem mózgu. Ich rozwój sięga lat 40. XX wieku, gdy Warren McCulloch i Walter Pitts stworzyli pierwszy model neuronu. Jednakże, prawdziwy przełom nastąpił w latach 80. i 90., kiedy to pojawiły się nowe algorytmy uczenia, takie jak algorytm wstępnej propagacji błędu, umożliwiające skuteczne trenowanie sieci neuronowych na dużą skalę.

Sieć neuronowa składa się z połączonych ze sobą sztucznych neuronów, które przetwarzają i analizują dane wejściowe, generując odpowiedź na podstawie nauczonej wiedzy. Podstawowym elementem jest neuron, który składa się z trzech głównych części: wejść, wag oraz funkcji aktywacji. Wejścia reprezentują dane wejściowe, wagi są parametrami, które modyfikują sygnały wejściowe, a funkcja aktywacji decyduje, czy neuron będzie aktywowany czy nie, w zależności od sumy ważonych wejść.

Sieci neuronowe znalazły zastosowanie w wielu dziedzinach, w tym w rozpoznawaniu obrazów, przetwarzaniu języka naturalnego, analizie danych, systemach rekommendacyjnych, predykcji rynkowej i wielu innych. Dzięki swojej zdolności do uczenia się na podstawie danych, sieci neuronowe są niezwykle elastyczne i mogą być dostosowywane do różnorodnych problemów.

Mimo ogromnego postępu, sieci neuronowe nadal stoją przed pewnymi wyzwaniami, takimi jak interpretowalność decyzji, ograniczenia w zasobach obliczeniowych i potrzeba dużej ilości danych do skutecznego uczenia. Jednakże, ciągłe badania i rozwój nowych technik, w tym głębokie uczenie, nadzorowane i nie nadzorowane uczenie, wraz z postępem w sprzęcie komputerowym, sugerują, że sieci neuronowe będą odgrywać coraz większą rolę w przyszłości, napędzając innowacje w dziedzinie sztucznej inteligencji.

3 Eksperymenty

3.1 NN1

Na pierwszych laboratoriach musieliśmy zaimplementować bazową sieć neuronową typu MLP, w której można ustawić liczbę warstw, liczbę neuronów w każdej z warstw i wagę poszczególnych połączeń (w tym biasów). Sieć miała używać sigmoidy jako funkcji aktywacji. Z racji, że wcześniej czytałem co nieco o sieciach neuronowych to już na tym etapie zaimplementowałem możliwość wyboru funkcji aktywacji ze zbioru tanh, sigmoida, relu i funkcja liniowa, na różnych warstwach można ustawać różne funkcje aktywacji. Dodatkowe wymagania co do tej części były następujące:

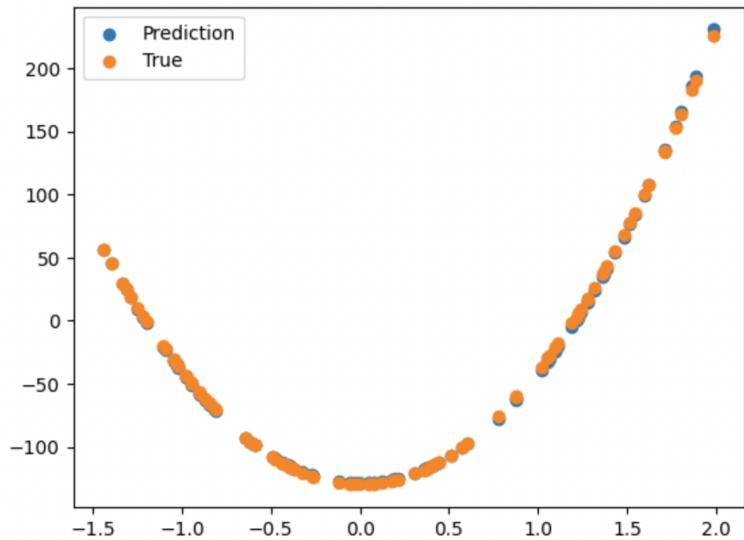
- prostota zmiany architektury, to znaczy liczby wejść, wyjść i neuronów w warstwach ukrytych,
- łatwość wymiany funkcji aktywacji.

Nie były to specjalnie trudne to zrealizowania wymagania i każda sensowna implementacja je spełniała. Ta część zadania była relatywnie prosta. Druga połowa zadania polegała na przetestowaniu naszego modelu sieci neuronowej na dwóch zbiorach *square-simple* i *steps-large*. Parametry do tych modeli należało dobrać ręcznie. Jak się okazało nie jest to najłatwiejsze zadanie. Podczas gdy do drugiego zbioru dało się wymyślić procedurę dobierania wag i biasów o tyle pierwszy zbiór okazał się znacznie bardziej problematyczny. Aby sobie z nim poradzić zaimplementowałem liczenie pochodnych w sposób numeryczny. Co to znaczy? Cofnałem się do samej definicji pochodnej, mianowicie liczyłem funkcję kosztu C następnie zwiększałem jeden parametr o małą liczbę h . Ponownie liczyłem funkcję kosztu C' tym razem ze zmienionym delikatnie parametrem. Jak wiadomo definicja pochodnej jest następująca:

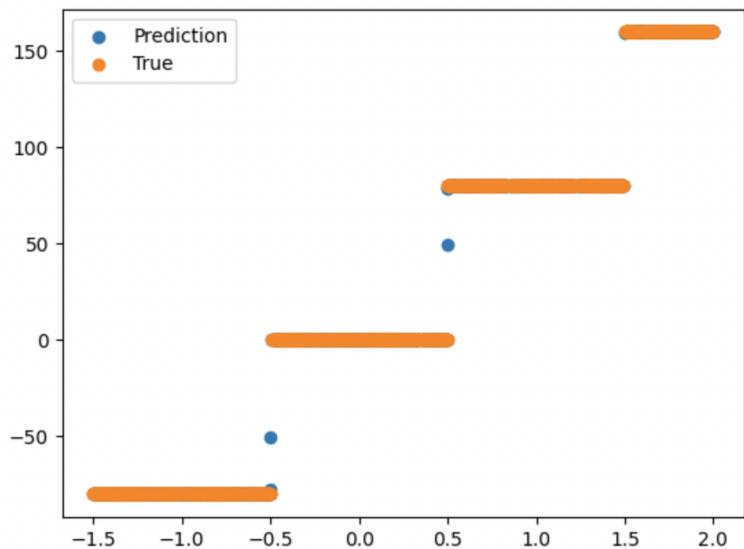
$$\lim_{h \rightarrow 0} \frac{C'(x + h) - C(x)}{h}$$

Na koniec odejmuję h od parametru do którego wcześniej go dodałem aby przywrócić go do pierwotnej wartości. Podczas gdy procedura ta okazała się bardzo wolna, dla małych sieci nie było to aż tak dużym wyzwaniem. Na początku napotkałem problemy ze zbieżnością. Jednak po dniu czytaniu o sieciach neuronowych postanowiłem znormalizować dane. Ten pomysł okazał się strzałem w dziesiątkę i rozwiązał wszelkie trudności, które dotyczyły napotkałem. Było to "odkrycie", które moi koledzy i koleżanki dokonają dopiero 2 tygodnie po moich zmaganiach.

Przy użyciu mojej prymitywnej metody liczenia pochodnych, wolno ale stabilnie udało mi się dojść do rozwiązania, które spełniało wymagania co do dokładności. Tutaj warto również wspomnieć o tym, że samo licznie gradientów nie wystarczyło, napisałem także prosty algorytm Stochastic Gradient Descent. Dlaczego akurat ten algorytm? Nie wymaga liczenia tak dużej ilości pochodnych w porównaniu do takich algorytmów jak Full Batch Gradient Descent, a miało to duże znaczenie, gdyż moje pochodne liczyły się bardzo wolno. Finalnie w drugim problemie użyłem ręcznej metody dobierania wag gdyż łatwo dało się je ustalić, 4 poziomy, trzeba było odpowiednio złożyć kolejne sigmoidy. Udało się osiągnąć zamierzane cele co do MSE na obu zbiorach.



Rysunek 1: Na zbiorze *square-simple* udało się uzyskać MSE: 2.25



Rysunek 2: Na zbiorze *steps-large* udało się uzyskać MSE: 1.81

3.2 NN2

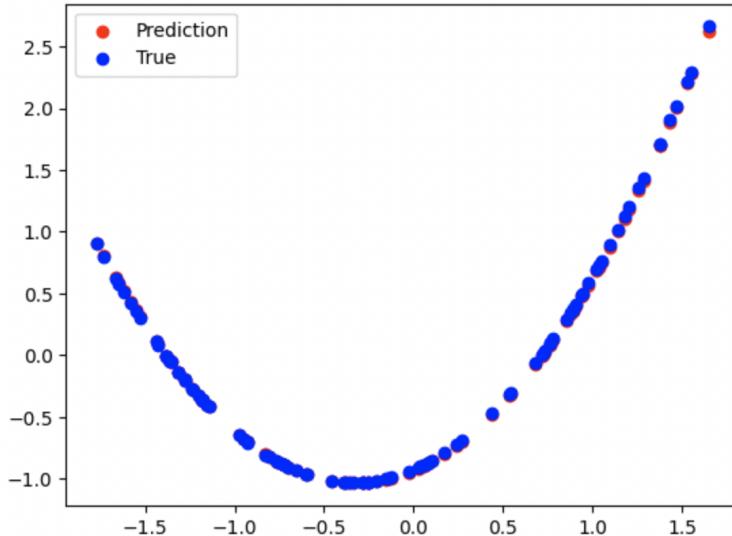
Na wykonanie tego etapu było wyjątkowo dużo czasu (2 tygodnie). Naszym celem było zaimplementowanie algorytmu wstecznej propagacji błędu, stąd więcej czasu. Założenie że potrzeba więcej czasu jest słuszne, gdyż w mojej opinii było to najcięższe zadanie. Razem ze znajomymi śmiały się, że tydzień zajmuje implementacja a drugi poświęca się na debugowanie. Ja miałem o tyle szczęścia, że zaimplementowałem liczenie pochodnych numerycznie. Miałem więc punkt odniesienia, pochodne zwarcane przez backpropagacje mogłem porównać z przybliżonymi - licznymi numerycznie. Okazało się to bardzo pomocne, szczególnie na etapie debugowania. W pewnym momencie otrzymywałem złe wyniki. Dzięki porównaniu byłem w stanie stwierdzić, że gradient dla biasów jest liczony poprawnie i szybko zlokalizować błąd. Sprawiło to, że cały ten proces był szybszy i mniej bolesny.

Co to backpropagacja? Jest to kluczowy algorytm wykorzystywany do uczenia się sieci neuronowych. Pozwala on na aktualizację wag połączeń między neuronami na podstawie błędu predykcji wyjściowej sieci, w celu minimalizacji tego błędu w trakcie procesu uczenia.

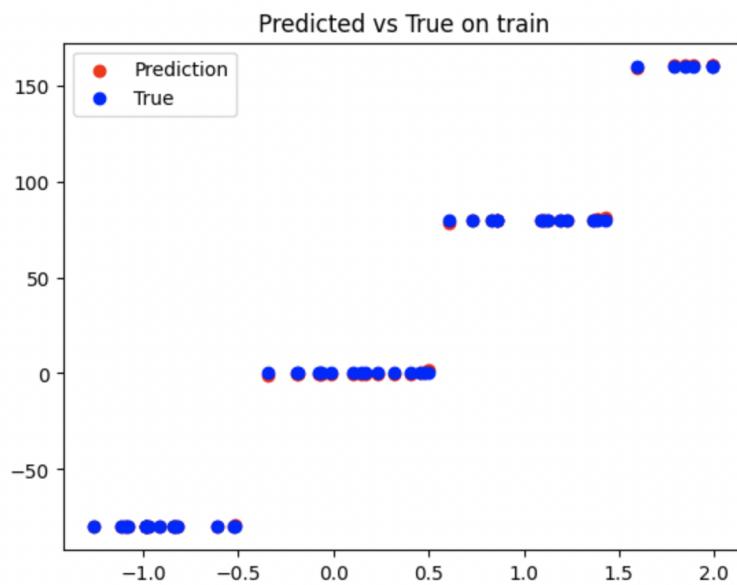
Dla zobrazowania założmy, że mamy sieć neuronową, która została już zainicjowana z losowymi wagami. Proces uczenia się sieci rozpoczyna się od podania danych treningowych na wejście sieci, a następnie generuje ona predykcję na wyjściu. Porównując tę predykcję z oczekiwany wynikiem (etykietą), możemy obliczyć błąd, czyli różnicę między predykcją a prawdziwym wyjściem. Algorytm backpropagacji polega na propagowaniu tego błędu wstecz przez sieć, tj. obliczeniu, jak bardzo każda waga przyczyniła się do ogólnego błędu. Proces ten składa się z dwóch głównych etapów:

1. Propagacja Wsteczna Błędu: Na początku, błąd jest obliczany na wyjściu sieci, a następnie propagowany wstecz przez warstwy sieci, obliczając gradient funkcji kosztu względem każdej wagi. To właśnie dlatego algorytm ten nazywany jest propagacją wsteczną.
2. Aktualizacja Wag: Po obliczeniu gradientów, wagi są aktualizowane w kierunku przeciwnym do gradientu, aby zminimalizować funkcję kosztu. Wagi są modyfikowane zgodnie z pewnym współczynnikiem uczenia, który kontroluje tempo uczenia się sieci.

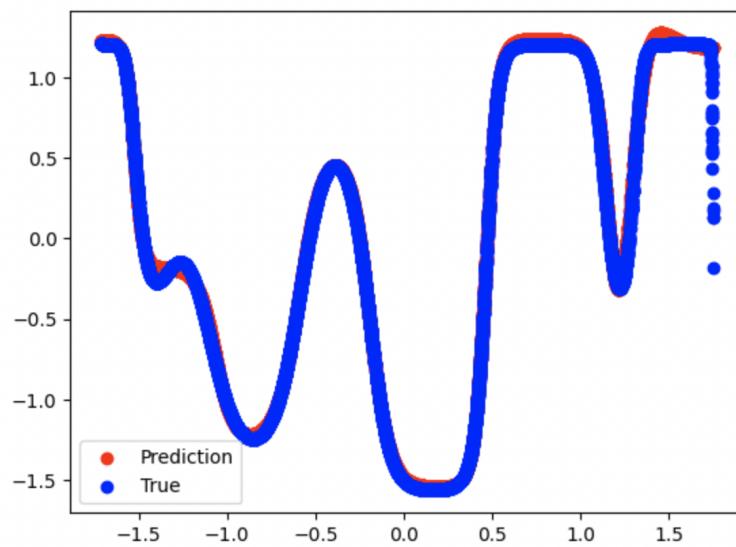
Po zaimplementowaniu należało dodatkowo przetestować algorytm na różnorodnych problemach i architekturach aby mieć pewność, że wszystko działa zgodnie z oczekiwaniemi. Tym razem sprawdzenie polegało na rozwiązyaniu 3 różnych zadań regresji. Udało się osiągnąć ustalone cele na dwóch z trzech zbiorów. Problem wystąpił ze zborem *steps-small* gdzie na zbiorze treningowym uzyskałem MSE 0.236 natomiast na zbiorze testowym MSE 91.53. Wynikało to z różnych rozkładów danych.



Rysunek 3: Na zbiorze *square-simple* udało się uzyskać MSE: 0.881

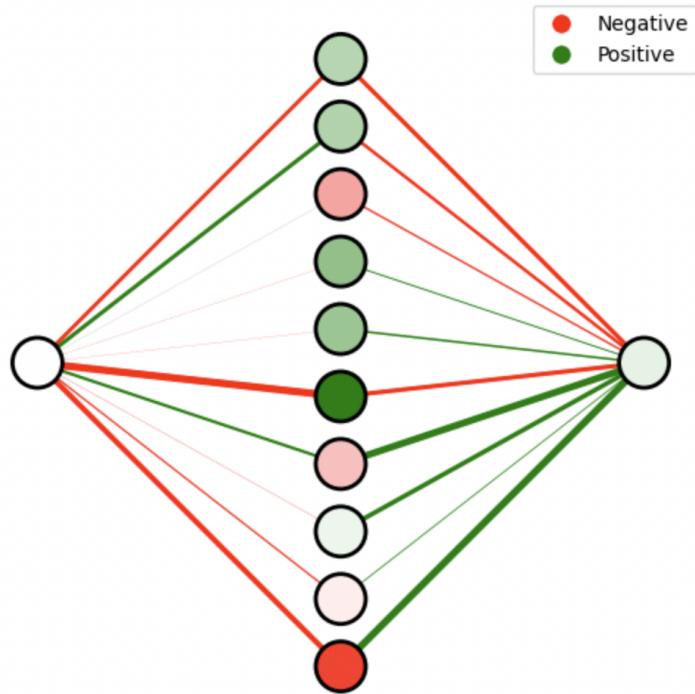


Rysunek 4: Na zbiorze treningowym *steps-small* udało się uzyskać MSE: 0.236

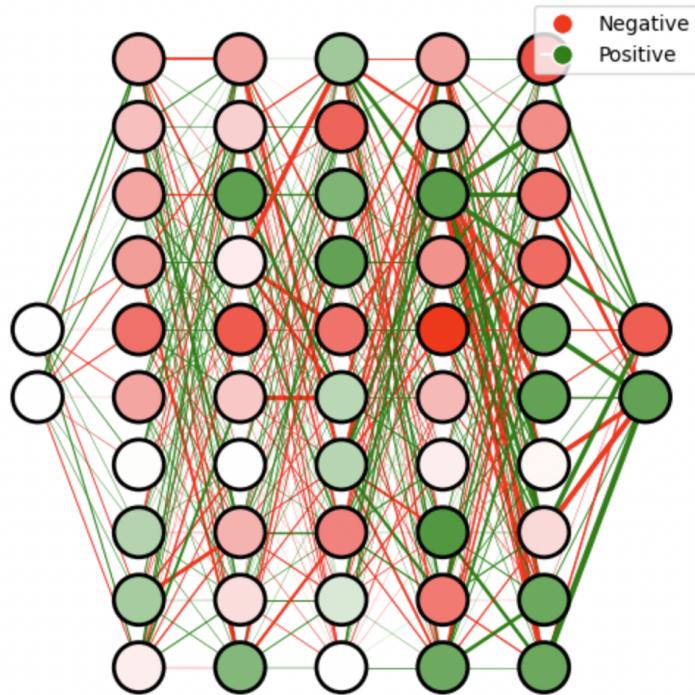


Rysunek 5: Na zbiorze *multimodal-large* udało się uzyskać MSE: 5.65

Dodatkowo należało napisać funkcjonalność wizualizacji architektury sieci i wag oraz biasów. Moja wizualizacja wygląda następująco:



Rysunek 6: Wizualizacja prostej sieci po wytrenowaniu

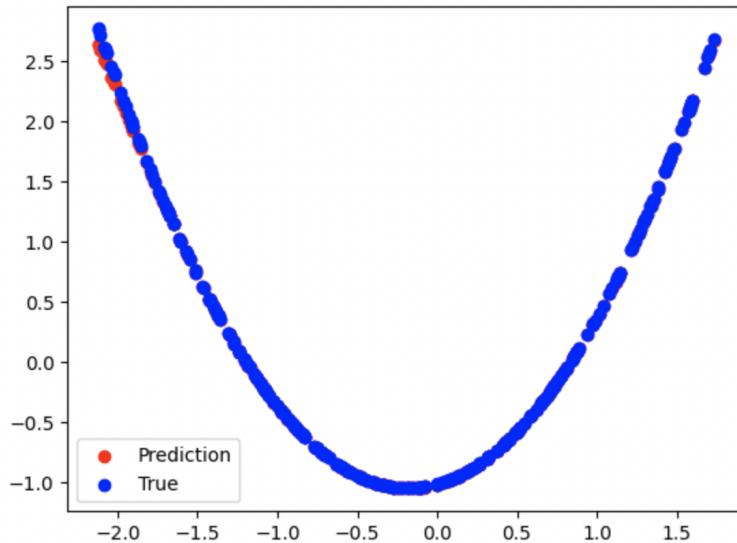


Rysunek 7: Wizualizacja bardziej skomplikowanej sieci również po wytrenowaniu

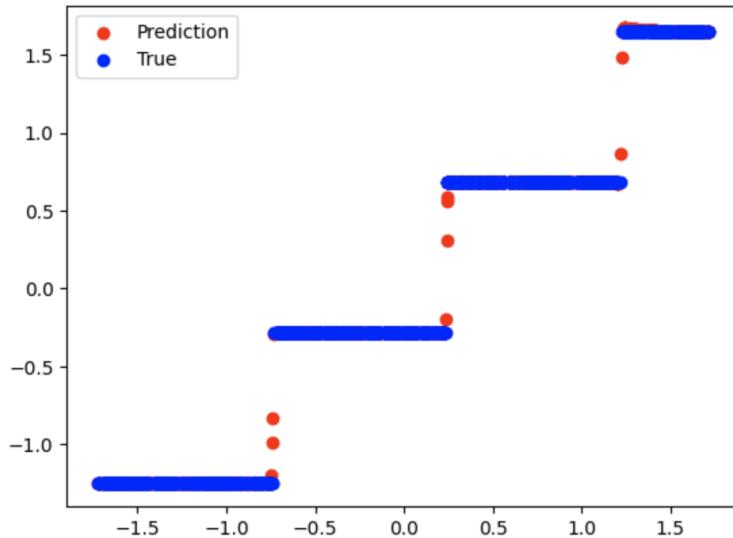
3.3 NN3

Te zadanie według mnie było zdecydowanie prostsze od poprzedniego. Polegało na usprawnieniu implementacji sieci neuronowej o dodanie 2 nowych algorytmów optymalizacji - Gradient Descent with momentum i normalizację gradientu RMSProp. Ja posunąłem się o krok dalej i oprócz 2 wyżej wymienionych zaimplementowałem Mini Batch Gradient Descent, Full Batch Gradient Descent, Adagrad i ADAM. Prawdę mówiąc nie było to znacznie bardziej skomplikowane, wymagało tylko trochę więcej pracy. Finalnie wszystkie algorytmy działały, jednak od tego momentu jedynym, którego używałem był ADAM. Dlaczego? Jest on niesprzecznie najpopularniejszym algorymem optymalizacji gradientowej używanym w machine learningu. Z moich eksperymentów wynika, że podczas gdy nie zawsze zgłębia najszybciej, jest najmniej wrażliwy na dobór hiperparametrów takich jak *learning_rate*. Dodatkowo jest on najbardziej stabilny i przewidywalny. Bardzo dobrze się z nim pracuje. W tym tygodniu tak samo jak w poprzednim należało przetestować 3 problemy regresji.

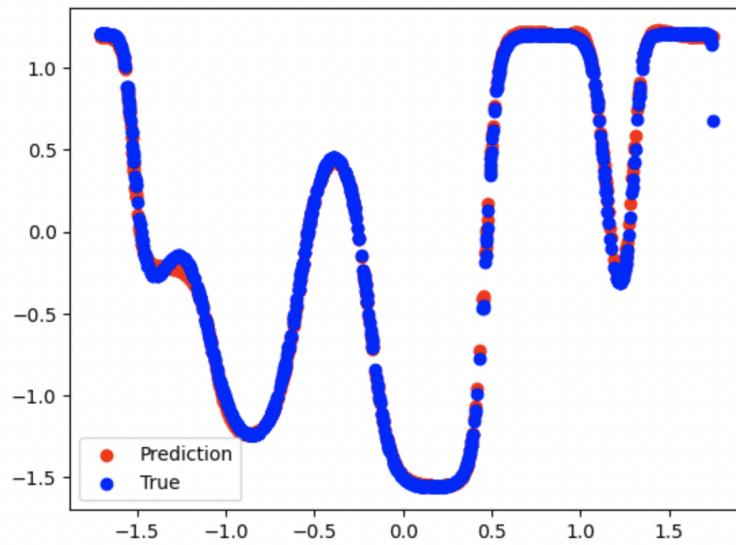
- *square-large* - zbiór testowy znacznie różnił się od zbioru treningowego dlatego niemożliwe było osiągnięcie wymaganego MSE 1, na zbiorze testowym osiągnąłem MSE 3.63 natomiast na zbiorze treningowym MSE 0.01
- *steps-large* - bardziej "gęsta" wersja zbioru z poprzedniego tygodnia, pierwotnie ćwicząc model tylko raz udało mi się osiągnąć MSE 12.81 jednak stosując technikę wielokrotnego trenowania z mniejszym parametrem *learning_rate* udało się zjeść do wartości MSE 3.37
- *multimodal-large* - identyczny zbiór jak w zeszłym tygodniu



Rysunek 8: Na zbiorze *square-large* udało się uzyskać MSE: 3.63, widać że predykcje rozjeżdżają się z prawdziwymi wartościami w lewej części, która nie była obecna w zbiorze treningowym

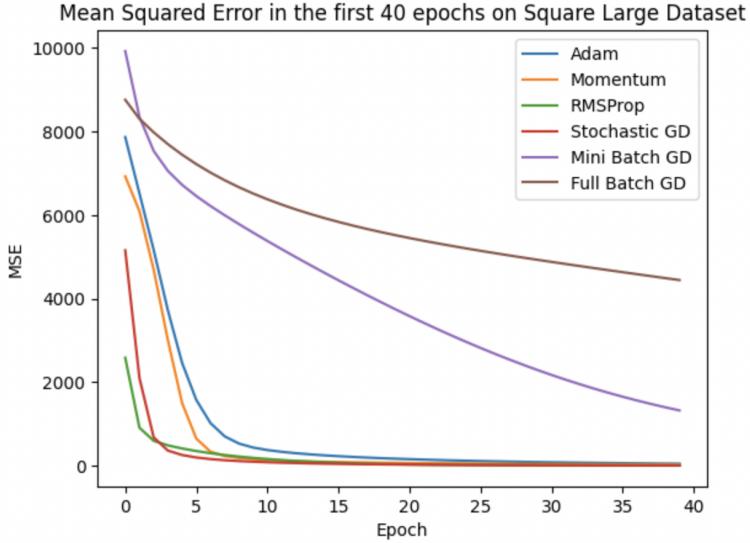


Rysunek 9: Na zbiorze *steps-large* udało się uzyskać MSE: 3.37



Rysunek 10: Na zbiorze *multimodal-large* udało się uzyskać MSE: 5.06

Na tych laboratoriach należało również porównać zaimplementowane algorytmy optymalizacji. Z moich eksperymentów wynika że ogromne znaczenie ma batch size, jest to zgodne z teorią. Z racji doboru stosunkowo prostego zbioru wszystkie optymizatory poradziły sobie dobrze. Te w których batch był większy zbiegają wolniej gdyż w takiej samej ilości epochów dochodzi do mniejszej ilości aktualizacji wag.



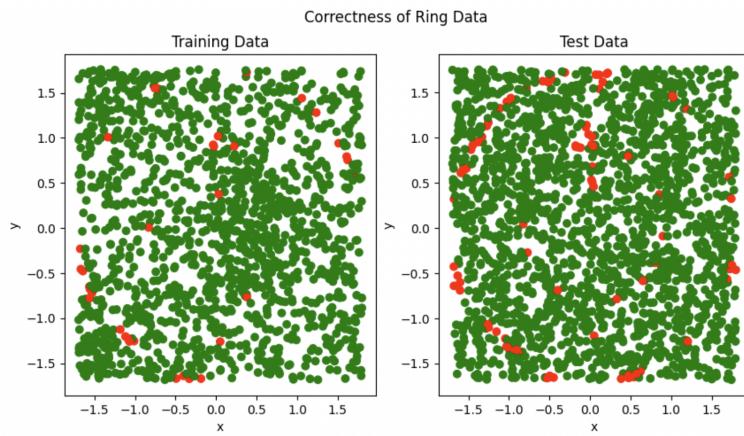
Rysunek 11: Porównanie szybkości zbieżności na zbiorze *square-large*

3.4 NN4

Te laboratoria również nie były ciękie w mojej opinii. Celem było przystosowanie sieci do problemów klasyfikacji. Z racji, że wiedziałem co nas czeka w przyszłości zaimplementowałem moją sieć neuronową w taki sposób aby była modularna i aby można było ją w łatwy sposób rozbudowywać. Kod był podzielony na wiele plików co ułatwiało jego zarządzanie.

Co należało zmodyfikować? Trzeba było dodać funkcję softmax oraz inną funkcję kosztu - cross entropy. Softmax jest to funkcja która zamienia wektor $w \in R^N$ w wektor prawdopodobieństwa czyli taki którego elementy są nieujemne i sumują się do 1. A cross-entropy to miara używana w uczeniu maszynowym do oceny różnicy między dwiema rozkładami prawdopodobieństwa. Jest szczególnie popularna w kontekście klasyfikacji, gdzie jest używana do porównywania przewidywanych rozkładów prawdopodobieństwa modelu z rzeczywistymi etykietami. Im niższa wartość entropii krzyżowej, tym lepsze dopasowanie modelu do danych. Oczywiście powyższe zmiany musiały zostać odzwierciedlone w algorytmie backpropagacji.

Również i w tym tygodniu trzeba było przetestować nasze modyfikacje na 3 zbiorach. Tym razem jednak miara jakości naszych rozwiązań się zmieniła. Musieliśmy osiągnąć odpowiednie wartości F-measure. F-measure to metryka używana do oceny jakości klasyfikacji binarnej, która łączy precyzję (precision) i pełność (recall) w jedną liczbę. Jest wyrażana jako harmoniczna średnia tych dwóch wartości, co pozwala uwzględnić zarówno false positives jak i false negatives w ocenie modelu. F-measure jest szczególnie przydatna w przypadku niezbalansowanych danych, gdzie precyzja i pełność mogą być rozbieżne. Do policzenia F-measure wykorzystałem gotową implementację z biblioteki sklearn. Dla zbiorów ring i easy udało się, osiągnąć wymagane progi natomiast na zbiorze xor3 udało mi się otrzymać F-score 0.966 podczas gdy wymagany był 0.97 można więc powiedzieć, że byłem bardzo blisko. Poniższe wizualizacje ilustrują, w których miejscach się pomyliłem (punkty zaznaczone na czerwono). Jak można zauważyc największe problemy sprawiają punkty znajdujące się na granicy 2 obszarów. Dodatkowo nie pokażę wizualizacji dla zbioru easy, gdyż na zbiorze treningowym poprawnie sklasyfikowałem wszystkie obserwacje a na zbiorze testowym pomyliłem się tylko raz osiągając F-measure 0.997.

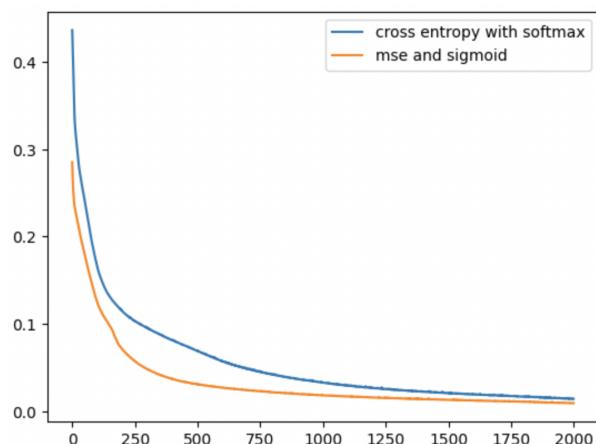


Rysunek 12: Na zbiorze *rings3-regular* udało się uzyskać F-measure: 0.954



Rysunek 13: Na zbiorze *xor3* udało się uzyskać F-measure: 0.966

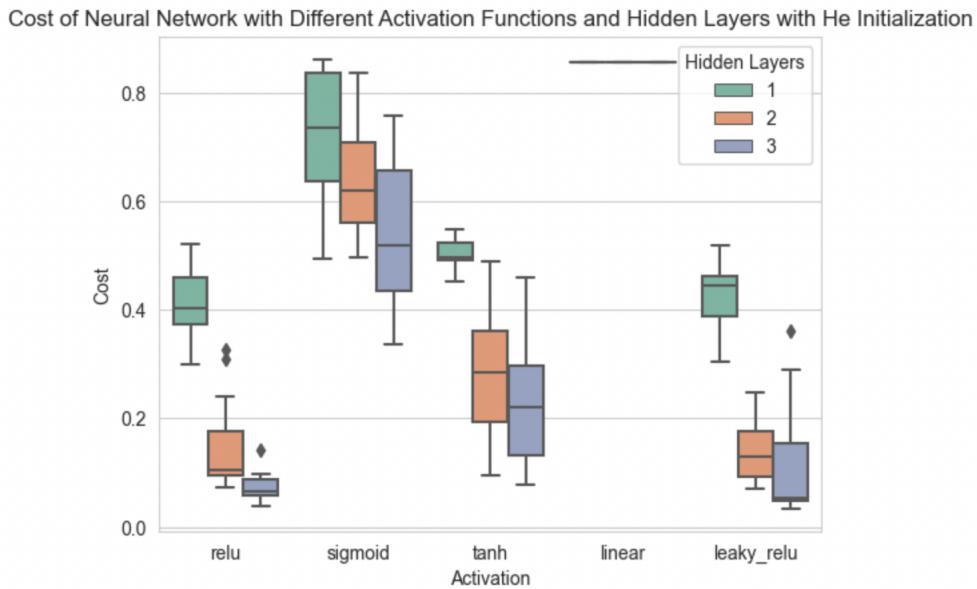
Dodatkowo należało porównać szybkość zbieżności używając jako funkcji straty MSE i cross entropy. Porównania dokonałem na zbiorze xor3. Wyniki okazały się nieoczekiwane, niezgodne z teorią. Wydaje mi się, że może być to spowodowane prostotą problemu.



Rysunek 14: Na zbiorze *xor3* MSE zbiega szybciej niż cross entropy

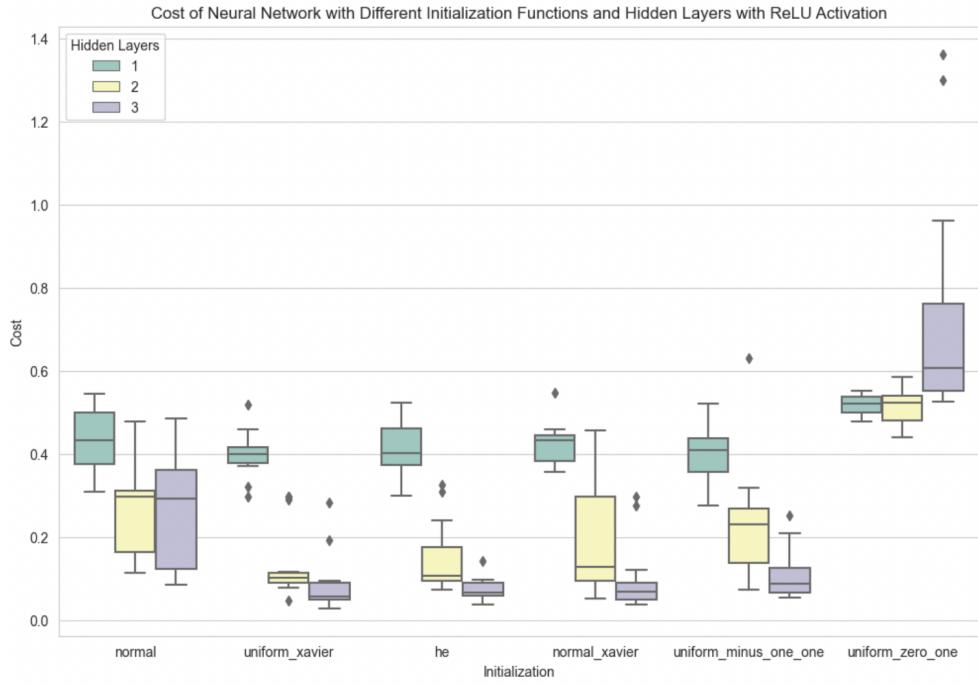
3.5 NN5

Domyślnie te laboratoria też nie należały do najtrudniejszych. Jeżeli ktoś zaprojektował swoją sieć w sensowny sposób, modyfikacje wymagane w NN5 sprowadzały się do krótkich 4-5 funkcji, łącznie 20 linijek kodu. Znaczna część czasu poświęcona na to ćwiczenie leżała w testowaniu. Ja zdecydowałem się jeszcze na więcej. Postanowiłem, że sprawdę jak mają się rzeczy gdy uwzględnimy różne funkcje inicjalizacji wag. Aby móc wyciągnąć jakieś wnioski należało wykonać więcej niż jeden trening i w jakiś sposób uśrednić wyniki. Ja zdecydowałem się każdą kombinację wykonać 15 razy. Łącznie musiałem stworzyć i wytrenować 1350 modeli. Aby nie obciążać zanadto mojego komputera, postanowiłem, że obliczenia do tej części będę wykonywać w Google Colab. Jest to chmurowe rozwiązywanie umożliwiające wykonywanie Jupyter Notebook na serwerach googla. Po wykonaniu obliczeń zapisałem wyniki w postaci obiektu pickle i pliku .csv a następnie zaimportowałem je do swojego lokalnego środowiska. Powyższe testy przeprowadziłem dla problemu *multimodal-large*. Następnie wybrałem dwie najlepsze kombinacje i użyłem ich do przetestowania *steps-large*, *rings5-regular* i *rings3-regular*. Ale najpierw przejdźmy do rezultatów uzyskanych na *multimodal-large*. Na początek boxplot finalnych wyników po wytrenowaniu z podziałem na różne funkcje aktywacji.



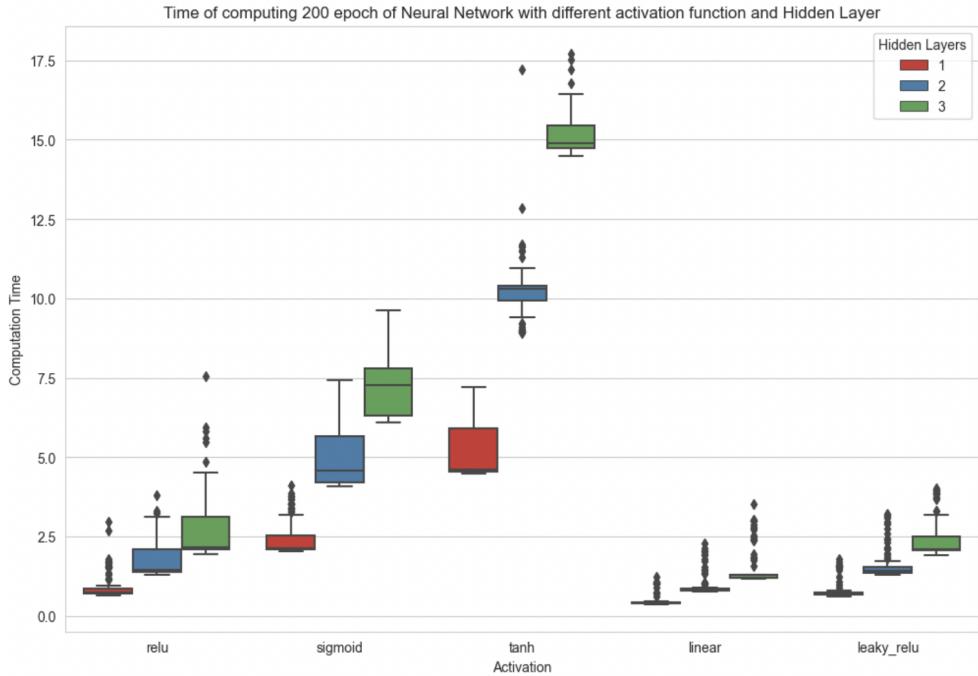
Rysunek 15: Jak jakość finalnej predykcji zależy od funkcji aktywacji

Jak widać im głębsza sieć tym lepsze rezultaty, każda z warstw ukrytych miała po 10 neuronów. Najgorzej poradziła sobie funkcja liniowa. Jest to zgodne z oczekiwaniami. Na wykładzie, pojawiło się twierdzenie, że dowolnie głęboka sieć z liniową funkcją aktywacji da się przedstawić jako sieć z jedną warstwą. Co ciekawe sigmoida, której używaliśmy dotychczas również nie poradziła sobie najlepiej w tym zadaniu i w porównaniu do relu daje znaczco gorsze wyniki. Z przeprowadzonego testu wynika, że relu jest najlepszą w konfiguracji, którą testowałem. Nie ma innej architektury, która może się równać z 3 warstwową siecią używającą relu. Kolejnym wykresem jest boxplot gdzie porównujemy różne funkcje inicjalizacji wag dla ustalonej funkcji aktywacji - relu.



Rysunek 16: Jak jakość finalnej predykcji zależy od funkcji inicjalizującej wagi

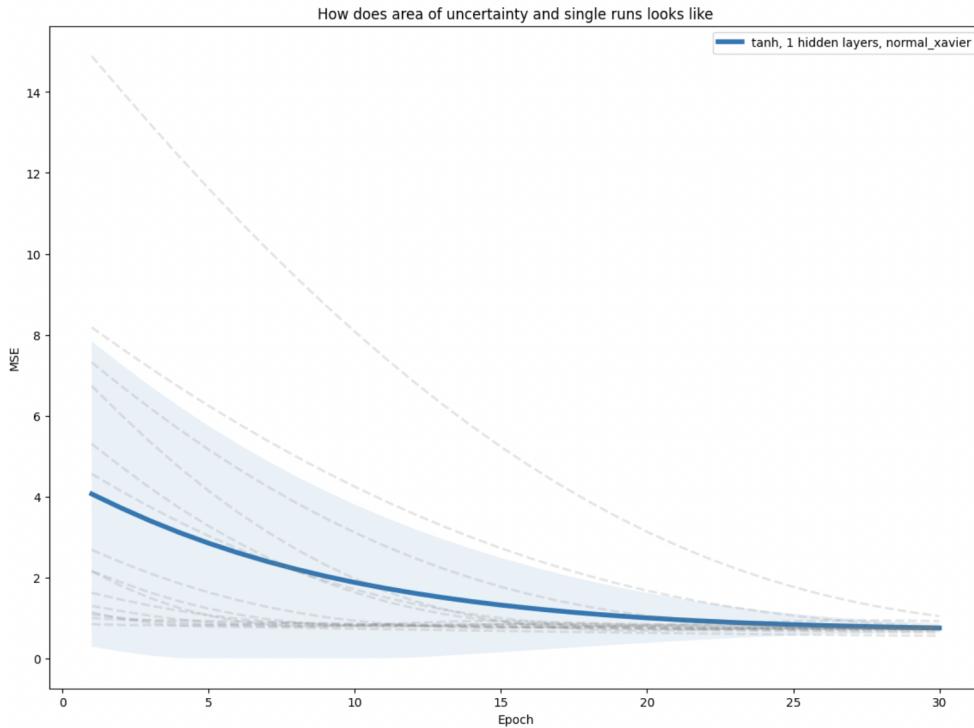
Ciekawym spostrzeżeniem jest to, że gdy inicjalizujemy wagi ze rozkładu $U[0,1]$, sieć nie zbiega i dostajemy tak zwane wybuchające wagi. Zwykłe inicjalizowanie z rozkładu $N(0,1)$ również nie daje najlepszych wyników. Metody, które spisują się najlepiej to warianty xaviera i he, są to funkcje, które omawialiśmy na wykładzie i które zostały nam polecone na NN2, jako te które warto zaimplementować. Teraz przyjrzymy się czasowi wykonywania z podziałem na różne funkcje aktywacji. Pierwotnie nie spodziewałem się dużych różnic w tym obszarze, jednak wyniki zaskoczyły mnie znacząco.



Rysunek 17: Jak jakość czas treningu zależy od funkcji aktywacji

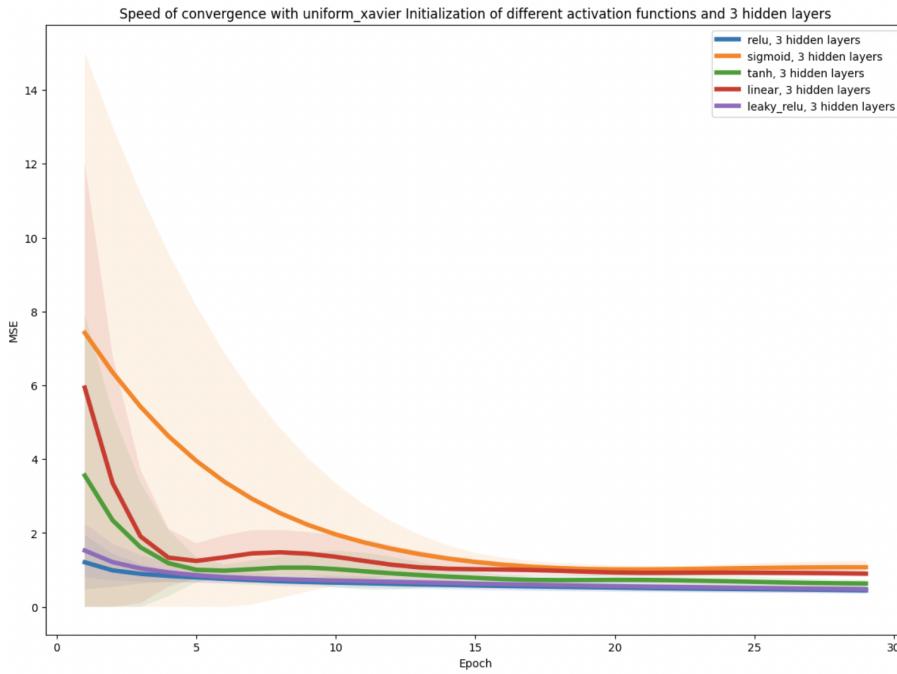
Oczywistym wnioskiem, łatwym do przewidzenia jest to, że im głębsza jest nasza sieć tym dłużej się

liczy. Najszybciej liczy się wariant z liniową funkcją aktywacji co nie powinno nikogo dziwić. Jednakże co dziwi to fakt jak długo liczyła się sigmoida w porównaniu do relu, prawie 3 razy dłużej. Co wprawia w osłupienie to fakt jak wolno liczy się wariant z tanh. Jest to o tyle dziwne, że tanh jest liniowym przekształceniem sigmoidy. Teraz przejdźmy do wykresów pokazujących szybkość zbieżności. Najpierw pokaże jak powinno się taki wykres interpretować.



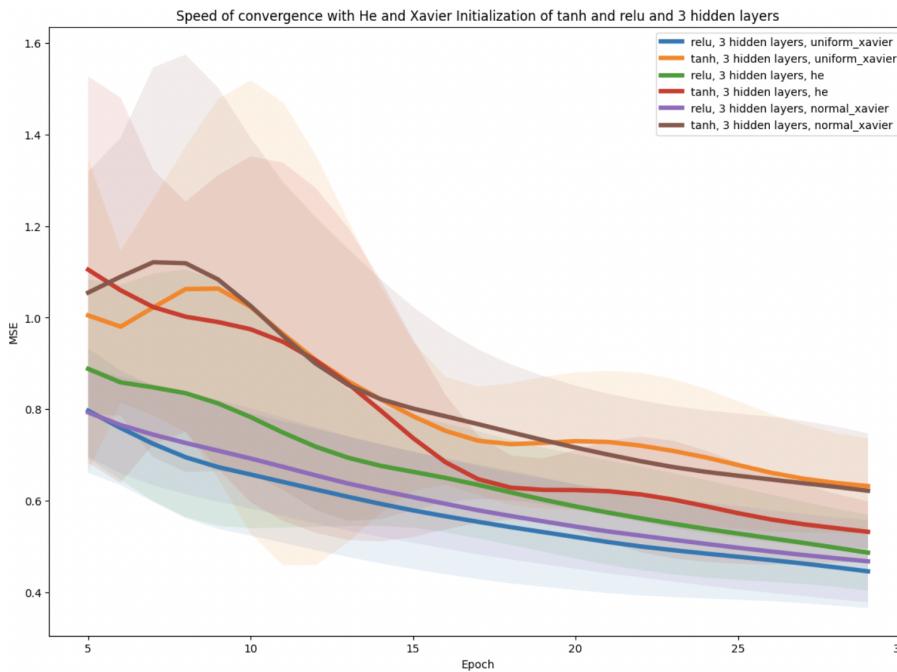
Rysunek 18: Interpretacja wykresu zbieżności

Linia niebieska jest to średnia z wszystkich 15 treningów, niebieski obszar to odchylenie standarde liczone osobno dla każdego epocha z 15 "obserwacji". Aby nie dawać, głupich rezultatów, obszar niepewności jest obcinany jeżeli dolna granica jest ujemna. Z perspektywy czasu ta wizualizacja nie jest idealna. Sugeruje, że błąd może być równy zero praktycznie od początku trenowania, jednakże widać patrząc na przerywane szare linie, które symbolizują MSE dla poszczególnych treningów, że tak nie jest. Może lepszym wyborem byłoby jako granice obszaru niepewności wzięcie 2 najgorszej i 2 najlepszej wartości, lub też mówiąc bardziej konkretnie wzięcie odpowiednio 10 i 90 centyla. Nie mniej jednak, wszystkie kolejne wykresy używają metody opisanej wcześniej. Przyjrzyjmy się, jak szybkość zbiegania zależy od doboru funkcji aktywacji.



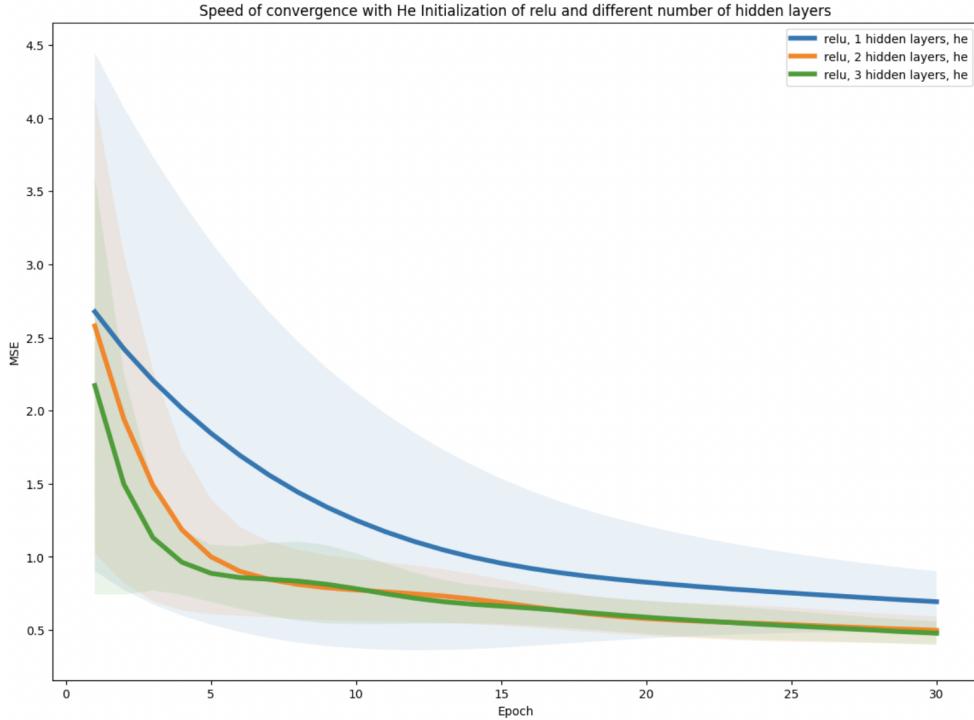
Rysunek 19: Szybkość zbiegania a funkcja aktywacji

Wszystkie wykresy zostały stworzone dla sieci z 3 warstwami ukrytymi. Widać, że sigmoida zbiega naj wolniej. Co ciekawe dla funkcji liniowej wartość MSE jest nie monotoniczna. Najlepiej wydają się działać relu i leaky relu. W późniejszym etapie trenowania, który dla przejrzystości wizualizacji nie został pokazany na wykresie, najlepiej sprawuje się leaky relu, funkcja liniowa staje się stała i ma wariancję równą 0, to znaczy, że dla wszystkich treningów zaobserwowano identyczne zachowanie, w rezultacie sigmoida prześciga funkcję liniową, a tanh jest trzeci.



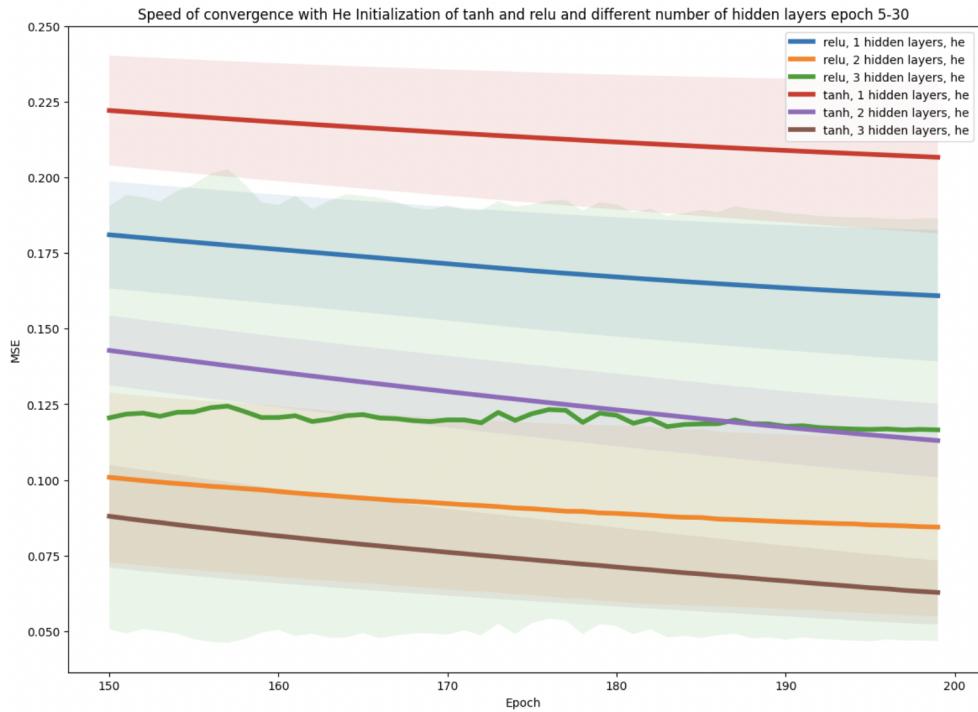
Rysunek 20: Szybkość zbiegania dla relu i tanh przy różnych funkcjach inicjalizacji

Na tym wykresie widzimy epoch 5-30. Najlepsze wyniki wydają się dawać kombinacje relu + warianty Xaviera. Trochę gorsze jest relu z He, następnie są funkcje tanh, jednak szybkości zbiegania są do siebie bardzo zbliżone i trzeba być ostrożnym z wyciąganiem wniosków, gdyż mogą być one nieprawdziwe. Ostatnim wykresem, który chce pokazać w tej części jest szybkość zbiegania dla relu z He dla różnych architektur. Możemy zobaczyć, że najgorzej wypada prosta sieć a im bardziej skomplikowana tym radzi sobie lepiej. Najbardziej nieprzewidywalna jest sieć z jedną warstwą ukrytą.



Rysunek 21: Szybkość zbiegania dla różnych architektur relu

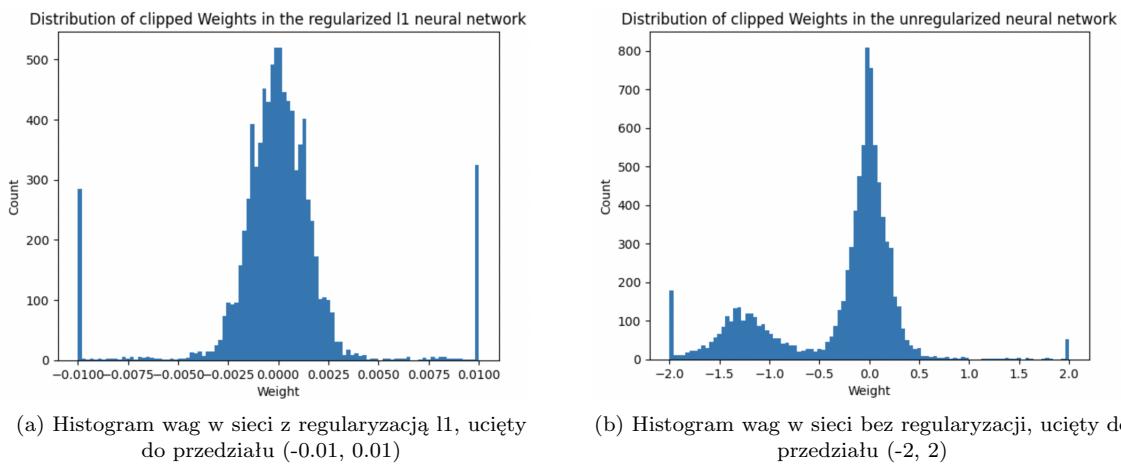
Do dalszych testów wybrałem relu z 3 warstwami ukrytymi i tanh również z 3 warstwami. Przetestowałem pozostałe zbiory przy ich użyciu, jednak pokażę wykres tylko dla rings5, gdyż ten rozdział jest już wystarczająco długi a wyciągnięte wnioski są podobne. Po znacznie więcej wykresów, dotyczących zarówno pierwszej części gdzie skupiłem się na zbiorze *multimodal*, jak i drugiej gdzie badam rezultaty na innych problemach odsyłam do [notebooka z NN5](#), który znajduje się na moim repozytorium. W zadaniach klasyfikacji tanh wypada znacznie lepiej niż w regresji, 3-warstwowa sieć relu ma w późniejszych etapach treningu problemy ze stabilnością, jednak bardziej skomplikowane sieci nadal radzą sobie lepiej.



Rysunek 22: Końcowa faza trenowania na zbiorze rigns5

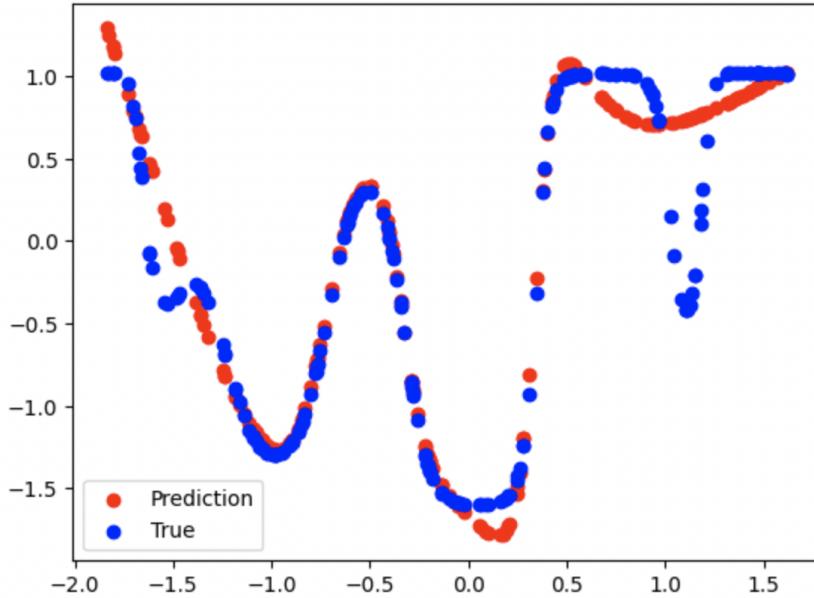
3.6 NN6

NN6 było według mnie najgorsze, nie najtrudniejsze, najgorsze. Implementacja wymaganych funkcjonalności była stosunkowo łatwa, jednak mój kod rozrosł się za bardzo i jego zarządzanie zaczyna być uciążliwe. Doświadczyłem podobnej sytuacji w okolicach NN3 i przepisałem z nabytym doświadczeniem kod od nowa. Okazało się to świetną decyzją, aktualnie czuję, że aby dalej rozwijać moją sieć, ten zabieg należało by powtórzyć. Dodałem 2 opcje regularyzacji: "l1" i "l2". Moje rozwiązanie nie jest najładniejsze jednak jest stosunkowo krótkie, około 20 linijek. Z early stopping był większy problem gdyż, każdy algorytm optymalizacji jest napisany oddziennie i występuje tam dużo powtórzeń kodu. Aby dodać funkcjonalność musiałem ją dodać osobno do każdej funkcji optymalizującej co było uciążliwe. Z testowaniem nie było lepiej. Dodając regularyzacje zauważałem, że mój estymator miał większe obciążenie, jednak niekoniecznie mniejszą wariancję. Sprawdziłem nawet empirycznie czy działa wykonując histogram wag w mojej sieci, gdy używam regularyzacji i gdy jej nie ma.



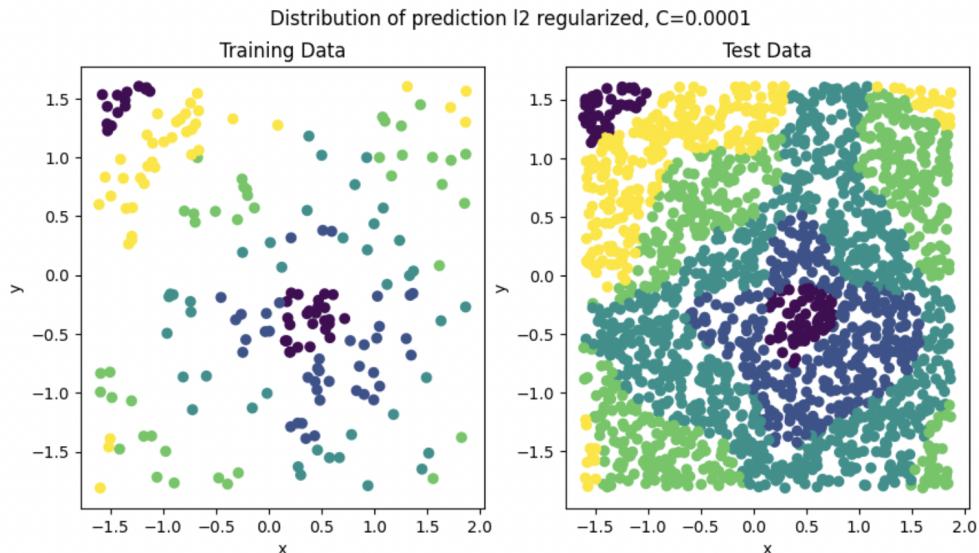
Jak widać wartość wag drastycznie maleje, na tyle że aby zobaczyć coś innego niż pojedynczy słupek

muszę "przybliżyć" o 2 rzędy wielkości. Z innych ciekawych rzeczy w mojej implementacji, oczywiście, dodanie regularyzacji powoduje, że trenowanie trwa dłużej, jednak co ciekawsze l1 wykonuje się około 20% dłużej niż l2. Teraz kiedy wiemy już, że regularyzacja działa przejdźmy do wyników. Dla sieci bez regularyzacji na *multimodal-sparse* uzyskałem MSE 175, z regularyzacją l2 MSE wynosiło 1809 a dla l1 MSE 728.

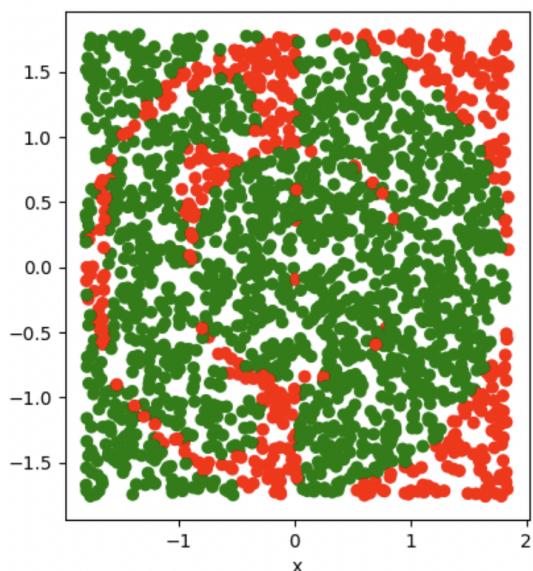


Rysunek 23: Predykcje dla modelu z regularyzacją l1

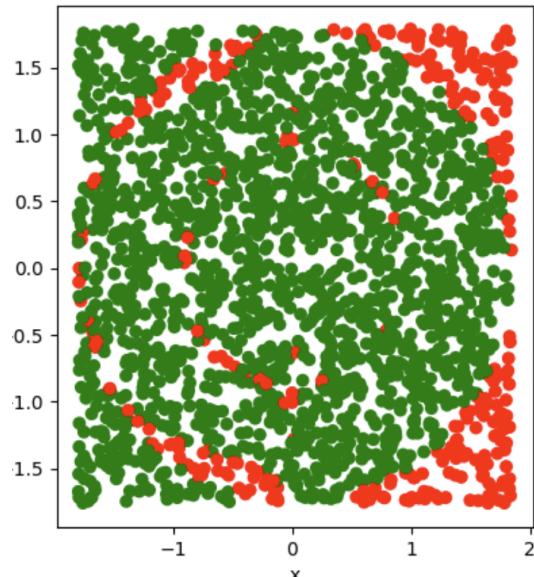
Widać, że wyjściowa funkcja ma mniejsze skłonności do "falowania". Dla tych danych nie testowalem metody early stopping. Dane *rings5-sparse* okazały się również wymagające jak *multimodal-sparse*. Widać, że po dodaniu regularyzacji model tworzy mniej skomplikowane kształty, bez ostrzych "kolców", które są powodowane przez jedną obserwację. Normalnie o to nam chodzi jednak tutaj wydaje mi się, że dane są zbyt proste, ciężko mówić tutaj o przeuczeniu.



Rysunek 24: Jak przewiduje model z regularyzacją l2



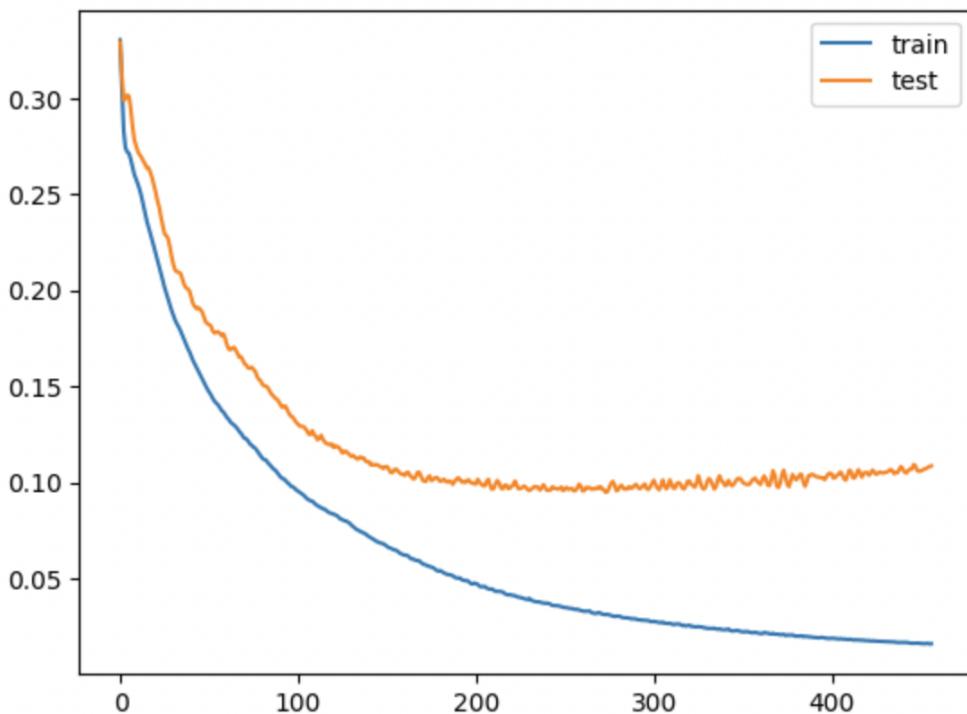
(a) Bez regularyzacji



(b) Z regularyzacją l1

Rysunek 25: Porównanie predykcji dla danych *ring3-balance*

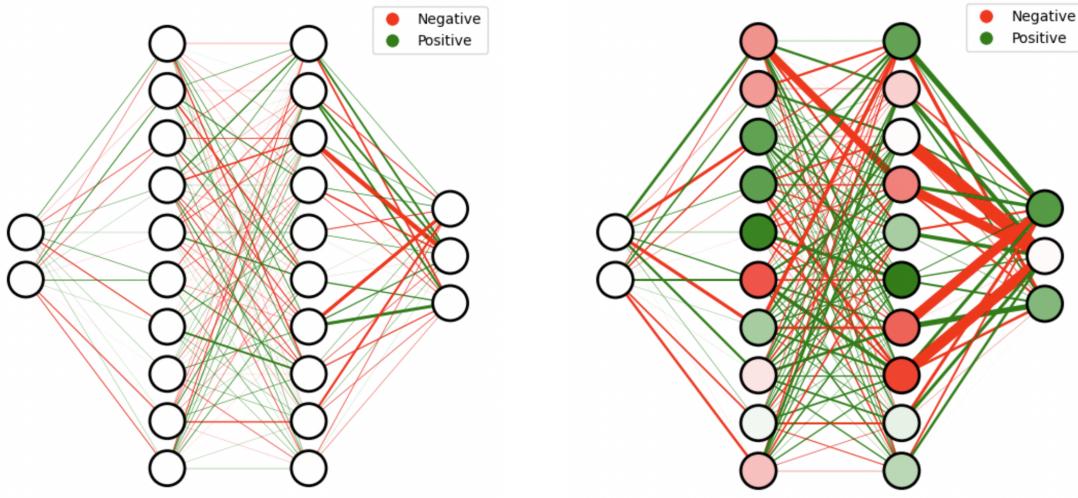
Na danych *ring3-balance*, regularyzacja zadziałała najlepiej. Możemy zobaczyć to na wykresie, po lewej model bez regularyzacji, po prawej z l1. Mało wspominam w tym rozdziale o metodzie early stopping. Zaimplementowałem ją w taki sposób, że przestaje uczyć kiedy przez 5 kolejnych batchy funkcja straty na niezależnym zbiorze się nie zmniejsza. Problem jaki napotkałem polega na tym, że funkcja straty na niezależnym zbiorze oscyluje przez co, nie kończę uczenia wystarczająco wcześnie. Dobry przykład tego zachowania znajduje się na wykresie poniżej.



Rysunek 26: Niepożądane zachowanie funkcji straty na zbiorze niezależnym

3.7 Dodatkowe

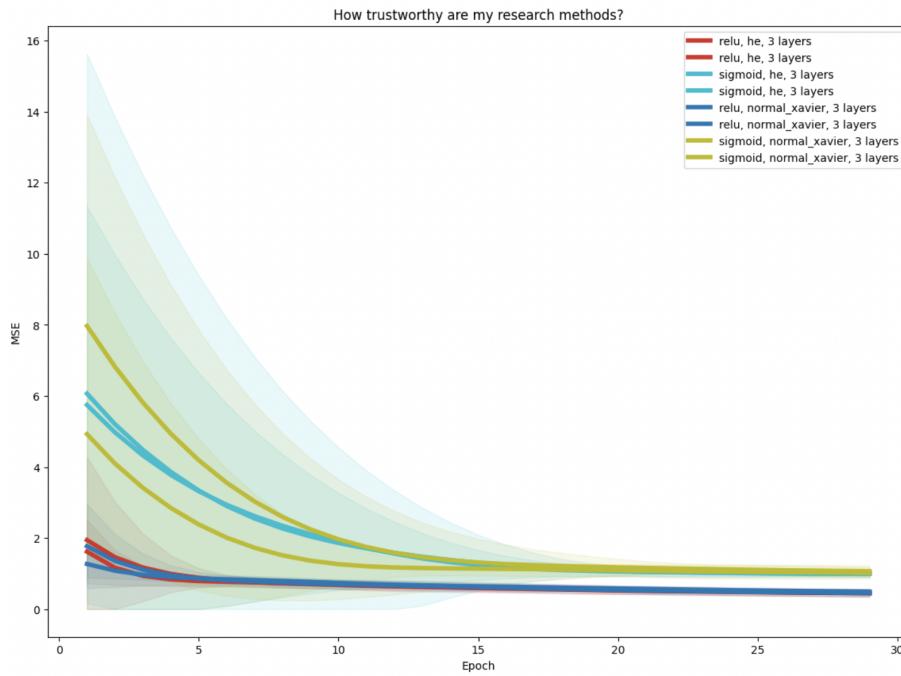
W trakcie tych wszystkich zadań wykonalem też kilka eksperymentów które nie pasują tematycznie do polecen dla danego tygodnia. Postanowiłem, że dwa najciekawsze załączę w tej części. Na początku chciałem się podzielić prostą wizualizacją. Po lewej widać nie wytrenowaną sieć, z biasami ustawnionymi na zero i z relatywnie małymi wagami. Natomiast po prawej widać wytrenowaną sieć ze znacznie bardziej zróżnicowanymi co do rzędów wielkości wartościami. Może to być tylko przypadek i nie chce wyciągać pochopnych wniosków, ale bardzo duża część wag zachowała swój pierwotny znak. Aby zdeterminować czy faktycznie tak jest należało by przeprowadzić bardziej rygorystyczne testy.



(a) Wizualizacja wag w sieci przed trenowaniem

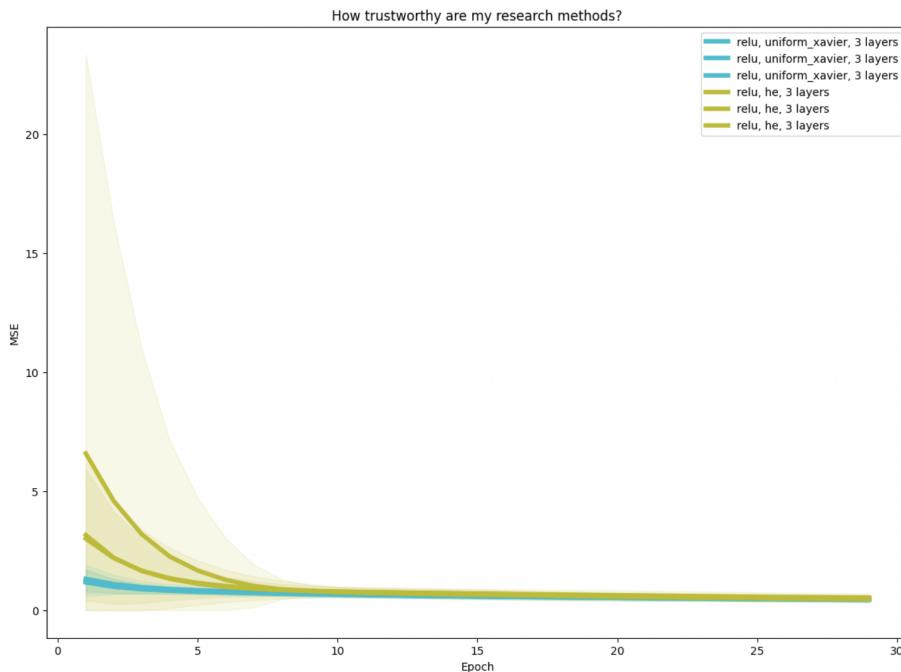
(b) Wizualizacja wag w sieci po trenowaniu

Chciałem również sprawdzić jak wiarygodne są moje pomiary szybkości zbiegania z NN5. W tym celu powtórzyłem część testów, wykonując 2 lub 3 identyczne próby po 15 treningów ka da. Nast epnie stworzyłem wykresy jak wcze niej z t  różnic ,  e pr by z tymi samymi kombinacjami oznaczone s  tymi samymi kolorami.



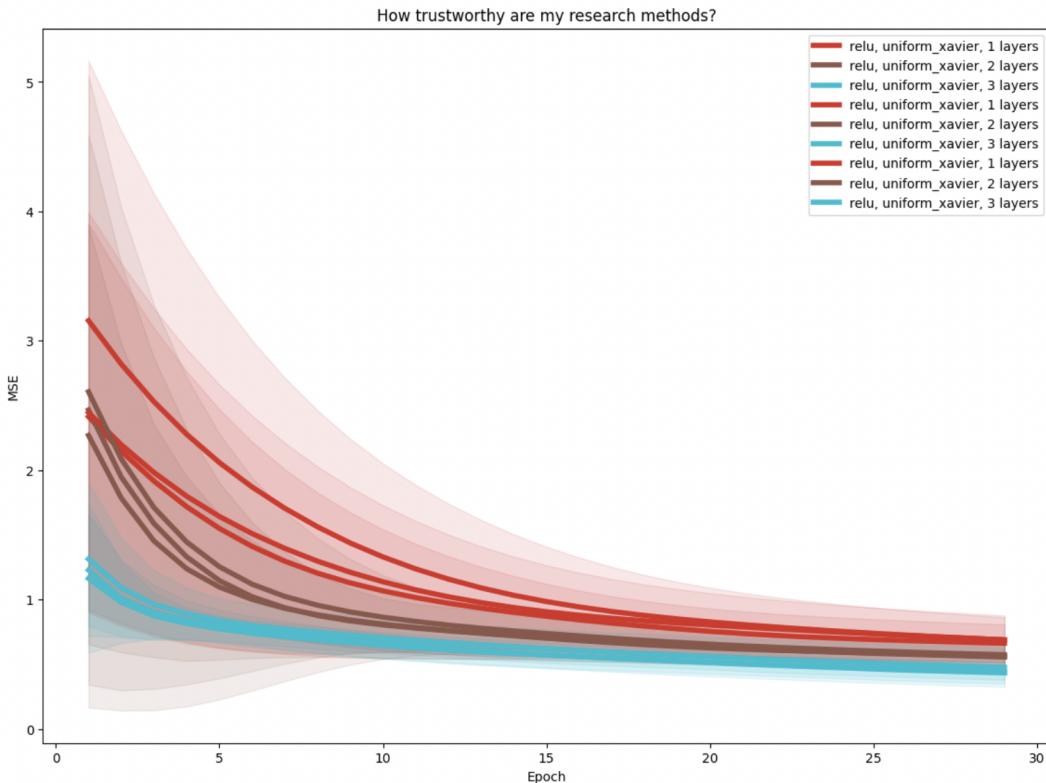
Rysunek 27: Jak wiarygodne są wyniki z NN5, 15 treningów w próbie, taka sama ilość jak w NN5

Jak widać wydaje się, że możemy wyciągnąć wnioski, że relu zbiega znacznie szybciej niż sigmoida, jednak losowość jest zbyt duża, żeby porównywać ze sobą funkcje aktywacji. Co również ciekawe niepewność związana z losością początkowych wag w późniejszym etapie trenowania przestaje odgrywać znaczenie. Następnie zwiększyłem dwukrotnie ilość treningów w próbie z 15 do 30 oraz zamiast wykonywać tą samą kombinację dwa razy zwiększyłem to do 3 razy a następnie ponownie przeprowadziłem analizę.



Rysunek 28: Jak wiarygodne są wyniki z NN5, 30 treningów, 2 razy większa ilość niż w NN5

Tym razem widać różnicę w doborze funkcji, wnioski dla tego przykładu są sprzeczne z teorią, że dla relu warto używać He jako funkcji inicjalizacji. Poszczególne trening z inicjalizacją Xavier zbiegają szybciej, dodatkowo nie mają tak dużej wariancji jak te z He. Kolejnym ciekawym wykresem jest porównanie różnych architektur, tutaj też widzimy trend, i możemy stwierdzić, że jest znacząca różnica. Najgorzej radzą sobie sieci z jedną warstwą ukrytą, lepsze od nich są te próby w których użyta była sieć z 2 warstwami ukrytymi a najlepiej spisują się sieci z 3 warstwami ukrytymi. Również z ilością warstw zmniejsza się wariancja poszczególnych treningów, ma to sens, mając mniej wag losowość ma większy wpływ, gdy mamy ich bardzo dużo wszystko się "uśrednia".



Rysunek 29: Jak wiarygodne są wyniki z NN5, 30 treningów, 2 razy większa ilość niż w NN5

4 Podsumowanie

Projekt oceniam za udany. Mówiąc prawdę był on bardzo czasochłonny, jednak czas ten nie poszedł na marne. Nauczyłem się niezwykle dużo i z czuję, że zbudowałem solidne fundamenty pod dalszą wiedzę z zakresu Deep Learning. Wszystkie zadania uważam za wykonane poprawne, otrzymywałem zadowalające wyniki, może pomijając NN6, gdzie regularyzacja nie przyniosła oczekiwanych rezultatów. Mam wrażenie, że ciężko jest przedstawić to zjawisko na tak syntetycznych danych, nie mniej jednak implementacja działa, co empirycznie zostało udowodnione histogramem wag w sieci. Jestem zadowolony z tego co zrobilem, a robiąc to, dobrze się bawiłem, jednak przyznam, że były momenty gdzie czułem się sfrustrowany. Jest to zdecydowanie jeden z lepszych projektów jakie miałem dotychczas na studiach i przykro mi, że kończymy temat sieci neuronowych, wolał bym aby został on bardziej rozwinięty, gdyż z pewnością jest jeszcze dużo do nauczenia się.