AKADEMIA GÓRNICZO-HUTNICZA IM. STANISŁAWA STASZICA W KRAKOWIE

WYDZIAŁ FIZYKI I INFORMATYKI

WYDZIAŁ INFORMATYKI STOSOWANEJ

Praca dyplomowa inżynierska

Wykorzystanie algorytmu sztucznej inteligencji do rozwiązania problemu optymalnego wyboru ścieżki lub wyjścia ewakuacyjnego

Autor: *Amadeusz Hercog*
Kierunek studiów: *Informatyka stosowana*
Opiekun pracy: *mgr. inż. Robert Lubaś*

Kraków, 21 listopada 2018

Uprzedzony o odpowiedzialności karnej na podstawie art. 115 ust. 1 i 2 ustawy z dnia 4 lutego 1994 r. o prawie autorskim i prawach pokrewnych (t.j. Dz.U. z 2006 r. Nr 90, poz. 631 z późn. zm.): „Kto przywłaszcza sobie autorstwo albo wprowadza w błąd co do autorstwa całości lub części cudzego utworu albo artystycznego wykonania, podlega grzywnie, karze ograniczenia wolności albo pozbawienia wolności do lat 3. Tej samej karze podlega, kto rozpowszechnia bez podania nazwiska lub pseudonimu twórcy cudzy utwór w wersji oryginalnej albo w postaci opracowania, artystycznego wykonania albo publicznie zniekształca taki utwór, artystyczne wykonanie, fonogram, wideogram lub nadanie.", a także uprzedzony o odpowiedzialności dyscyplinarnej na podstawie art. 211 ust. 1 ustawy z dnia 27 lipca 2005 r. Prawo o szkolnictwie wyższym (t.j. Dz. U. z 2012 r. poz. 572, z późn. zm.): „Za naruszenie przepisów obowiązujących w uczelni oraz za czyny uchybiające godności studenta student ponosi odpowiedzialność dyscyplinarną przed komisją dyscyplinarną albo przed sądem koleżeńskim samorządu studenckiego, zwanym dalej «sądem koleżeńskim».", oświadczam, że niniejszą pracę dyplomową wykonałem(-am) osobiście i samodzielnie i że nie korzystałem(-am) ze źródeł innych niż wymienione w pracy.

1. Wstęp

W tym rozdziale zostanie opisane podłoże historyczne, cel i jego założenia projektu. Zaprezentowane zostanie proponowane rozwiązanie postawionego problemu oraz technologie do tego użyte.

1.1. Geneza projektu

W dzisiejszym świecie kładziony jest nacisk na projektowanie budynków użytku publicznego tak, aby zapewnić możliwie największe bezpieczeństwo ludzi, którzy będą z nich korzystali. Jednakże, jednocześnie wymagane jest aby takie budynki były także funkcjonalne - na przykład zaplanowanie odpowiednie podział przestrzeni na pomieszczenia. Z tych powodów konieczne jest wyznaczenie ścieżek oraz wyjść ewakuacyjnych przewidzianych w taki sposób, aby ewakuacja ludzi odbyła się w sposób sprawny a jednocześnie, aby zachować wysoką funkcjonalność budynku.

Dziedzina informatyki, jaką jest uczenie maszynowe przez kilka ostatnich lat zyskała sporą popularność. Metody z tej dziedziny potrafią rozwiązywać problemy, które dotychczas były nierozwiązywalne za pomocą innych metod. Jednym z takich problemów jest uczenie się wzorca na podstawie zbioru danych oraz jego generalizacja. Daje to duże możliwości w takich zadaniach, gdzie potrzebujemy narzędzia potrafiącego symulować ruch ludzi na podstawie wczesniej nauczonych wzorców.

Rozwiązanie problemu optymalnego wyboru ścieżki lub wyjścia ewakuacyjnego dałoby odpowiedź jaki jest najkrótszy czas ewakuacji całego budynku, co daje informacje o tym, czy budynek jest dobrze zaprojektowany.

1.2. Cel projektu

W ramach pracy postawiłem sobie następujący problem - rozwiązanie problemu optymalizacji wyboru ścieżki lub wyjścia ewakuacyjnego ludzi w budynku. Co więcej, w związku z moim wcześniejszym doświadczeniem z tematem uczenia maszynowego, postanowiłem rozwiązać ten problem za pomocą technik machine learning.

1.3. Użyte technologie oraz programy

Wszystkie obliczenia oraz rysowanie wykresów zostało zrealizowane w języku programowania **Python** w wersji 3.6 przy użyciu bibliotek:

- **NumPy** - służąca do obliczeń numerycznych,
- **Pandas** - słuzaca łatwiejszemu zarządzaniu duzą ilością danych,
- **Matplotlib** - służąca do rysowania wykresów,
- **Tensorflow** - używana do głębokiego uczenia maszynowego.

Aby mieć możliwość symulacji ruchu pieszych w budynku, użyłem symulatora **PSP** *(Pederastian Simulation Project)* autorstwa Roberta Lubaś oraz Wojciecha Myśliwiec - opisany szerzej w rozdziale 4.

2. Wstęp teoretyczny

W tym rozdziale rozróżniono typy algorytmów uczenia maszynowego. Omówiono koncepcję Perceptronu. Przedstawiono także jedną z metod optymalizacji jaką jest Gradient descent. Ukazano w jaki sposób za pomocą tej metody można zoptymalizować wyjścia modelu uczenia maszynowego na podstawie jej wejść.

2.1. Typy modeli uczenia maszynowego

Uczenie maszynowe jest to termin określający całą gamę sposobów rozwiązywania problemów, które przy tradycyjnych metodach byłyby niemożliwe do rozwiązania lub skrajnie niewydajne obliczeniowo. Wyróżniane są 4 typy algorytmów uczenia maszynowego ze względu na sposób rozwiązywania przez nie problemu [1]:

- **Supervised learning** - algorytmy, do których danymi wejściowymi są zestawy cech i ich etykiety oznaczające do jakiej kategorii przynależy dana próbka. Tak nauczony model potrafi przewidzieć etykiety dla nowych zestawów cech.

- **Unsupervised learning** - typ algorytmów, do których dostarczane są tylko dane w postaci zbioru cech. Algorytm ma za zadanie sklasteryzować dane na podstawie podobieństw między nimi.

- **Semi-supervised learning** - te algorytmy są połączeniem algorytmów *Supervised learning* oraz *Unsupervised learning*. Przyjmują one dane, których tylko część jest oznaczona etykietami - na nich uczą się klasyfikacji, a na pozostałych (nieoznaczonych) - klasteryzacji.

- **Reinforcement learning** - są zupełnie odmienne od pozostałych typów algorytmów, gdyż nie uczą się na podstawie wcześniej przygotowanych danych wejściowych. Służą one do takich zadań, gdzie na podstawie obserwacji interakcji z pewnym środowiskiem, algorytm się uczy podejmowania najbardziej

opłacalnych akcji. Algorytmy te zbierają one dane na bieżąco, cały czas się na nich ucząc.

2.2. Perceptron - model regresji

Głównymi zadaniami, do których używane są algorytmy *Supervised learning* są regresja i klasyfikacja - mając dane wejściowe i ich etykiety, algorytmy te dają możliwość nauczenia modelu powiązań pomiędzy nimi z czego regresja przewiduje etykiety, które są wartościami ciągłymi a klasyfikacja - etykiety binarne (*czy to jest pies?*, *czy to jest kot?* i tym podobne).

2.2.1. Opis Perceptronu

Najprostszym modelem pozwalającym na regresję oraz na klasyfikację jest **Perceptron** [2][3]. Jest on przykładem jednowarstwowej sieci neuronowej. Posiada on zestaw neuronów wejściowych, każdy odpowiadający pojedynczej danej wejsciowej. Składa się on także z jednego lub wielu niezależnych neuronów wyjściowych. Każdy z nich jest połączony ze wszystkimi neuronami wejściowymi za pomocą krawędzi posiadających wagi - definiują one jak duży wpływ na wyjście ma każda dana wejściowa. Jeśli wyrazimy:

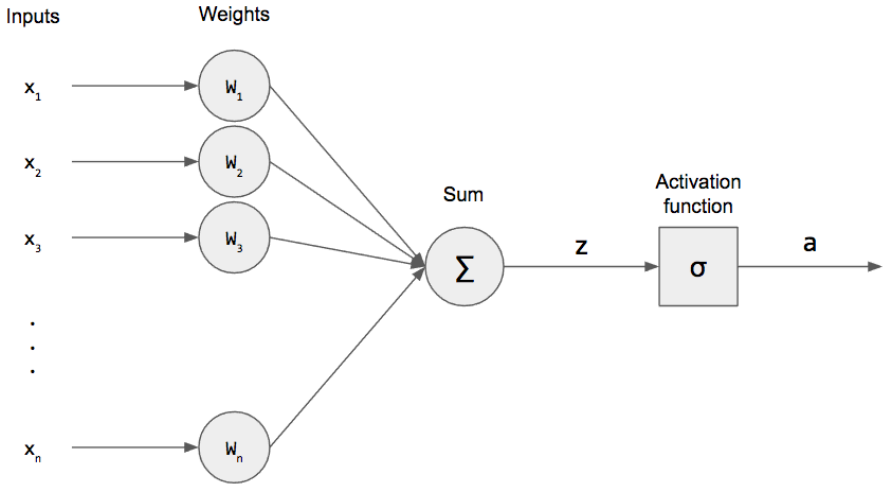
- wektor wejść jako *X*,
- wektor wyjść jako *Y*,
- macierz wag jako *N*,
- **funkcję aktywacji** jako σ - szerzej opisana w sekcji 2.2.2

to zachodzi równanie:

$$Y = \sigma(NX) \tag{2.1}$$

Budowa Perceptronu z jednym neuronem wyjściowym (oznaczonym jako *a*) jest widoczna na rysunku 5.1[1].

[1] *https://pythonmachinelearning.pro/perceptrons-the-first-neural-networks/*



Rys. 2.1. Budowa Perceptronu z jednym neuronem wyjściowym.

2.2.2. Parametry Perceptronu

Modele *Perceptron* posiadają pewien zestaw parametrów, który wpływa na proces uczenia w różny sposób. Nie ma uniwersalnego sposobu ich wyboru co oznacza, że proces ich doboru jest najczęściej serią testów z różnymi ich kombinacjami. Wszystkie parametry opisane w tej sekcji tyczą się modelu *Perceptron* jak i wielowarstwowych sieci neuronowych.

Pierwszym z parametrów jest funkcja aktywacji - wprowadza ona nieliniowość danych w sieci [4][5]. Służy ona także do ograniczania zakresu danych, najczęściej do przedziału (-1, 1). Wybrane z nich to:

– **Sigmoid** - stosowana często przy klasyfikacji, mająca wzór widoczny na równaniu 2.2,

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{2.2}$$

– **ReLU (Rectified linear unit)** - ze wzorem widocznym na równaniu 2.3,

$$\sigma(x) = \begin{cases} x, & \text{dla } x >= 0. \\ 0, & \text{dla } x < 0. \end{cases} \tag{2.3}$$

– **Softmax** - stosowana często w ostatniej warstwie modelu przy klasyfikacji ponieważ normalizuje normą *L1* wektor neuronów obarczonych tą funkcją do 1; ma ona wzór widoczny na równaniu 2.4, gdzie *K* to ilość neuronów w warstwie.

$$\sigma(x)_j = \frac{e^{x_j}}{\sum_{k=1}^{K} e^{x_k}} \text{ dla } j = 1, ..., K \tag{2.4}$$

Kolejnym ważnym parametrem modeli *Supervised learning* jest **funkcja kosztu** *(ang. loss function)* - metryka optymalizowana przez sieć [6]. Im ma ona mniejszą wartość, tym bardziej etykiety danych przewidziane przez sieć są bliższe etykietom rzeczywistym. Dla problemów regresji najczęściej uzywaną funkcją kosztu jest **MSE** (Mean Squared Error). Ma ona wzór widoczny na równaniu 2.5 gdzie:

- m - ilość zestawów **danych uczących** (na podstawie których odbywa się proces uczący sieci),
- h_θ - sieć neuronowa przedstawiona jako funkcja,
- $x^{(i)}$ - i-ty wektor danych wejściowych sieci,
- $y^{(i)}$ - i-ty wektor etykiet danych,
- θ - macierz wag.

$$J(\theta) = \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2 \tag{2.5}$$

Następnym parametrem sieci neuronowych jest parametr **learning rate**, który określa jak bardzo będzie się zmieniać macierz/wektor wag na podstawie gradientu podczas każdej iteracji uczącej [7]. Wzór przedstawiający udział tego współczynnika jest widoczny na równaniu 2.6 gdzie:

- θ_i - i-ta waga z macierzy wag,
- α - *learning rate*,
- $J(\theta_i)$ - funkcja kosztu.

$$\theta_i := \theta_i - \alpha\frac{\partial}{\partial\theta_i}J(\theta_i) \tag{2.6}$$

Zbyt duży współczynnik *learning rate* może spowodować, że zamiast optymalizować funkcję kosztu będzie wręcz przeciwnie - będzie ona osiągała coraz większe wartości. Gdy jest on zbyt mały, jest on mniej niebezpieczny gdyż nie doprowadzi on do wzrostu wartości funkcji kosztu; problemem może być jednak wolniejszy proces uczący sieci co wymaga więcej epok uczących.

Kolejnym parametrem mającym wpływ na sposób uczenia sieci jest **metoda minimalizacji**. Definiuje ona sposób, w jaki na podstawie *funkcji kosztu* będą dostrajane wagi w sieci podczas kazdej iteracji procesu uczenia. Najpopularniejszą

metodą minimalizacji jest **Metoda gradientów prostych** *(ang. Gradient descent)*, szerzej opisana w sekcji 2.3.

2.3. Optymalizacja wyjść modelu

Metoda minimalizacji jaką jest wspomniany wcześniej *Gradient descent* jest stosowana do znalezienia minimum funkcji [6]. Przyjmując, że sieć neuronowa jest funkcją oznaczoną $h(x)$, gdzie *x* to wektor danych wejściowych, to możliwe jest obliczenie pochodnej cząstkowej [8] tej funkcji po jej wektorze wejść w sposób pokazany w równaniu 2.7, gdzie:

– ϵ - pewien mały współczynnik, np. $1 * 10^{-6}$,

– *n* - długość wektora danych wejściowych.

$$\frac{\partial}{\partial x_i} h(x_0, ..., x_n) = \frac{h(x_0, ..., x_i + \epsilon, ..., x_n) - h(x_0, ..., x_i - \epsilon, ..., x_n)}{2\epsilon} \tag{2.7}$$

Tak obliczone pochodne cząstkowe funkcji następnie zostają pomnożone przez współczynnik λ (odpowiadający za siłę pochodnej) i odjęte od wektora wejściowego *x* w sposób pokazany na równaniu 2.8.

$$x_i := x_i - \lambda \frac{\partial}{\partial x_i} h(x_0, ..., x_n) \tag{2.8}$$

Proces jest powtarzany przez pewną liczbę iteracji, co prowadzi do optymalizacji funkcji $h(x)$.

Gdy metoda *Gradient descent* jest używana do minimalizacji *funkcji kosztu* sieci neuronowej, proces jest ten sam - tylko zamiast liczyć pochodne po wektorze wejściowym i później je od niego odejmować, są one liczone po macierzy wag, i to od niej są one odejmowane.

3. Proponowane rozwiązanie

Rozwiązanie proponowane przeze mnie zostało podzielone na kilka kroków:

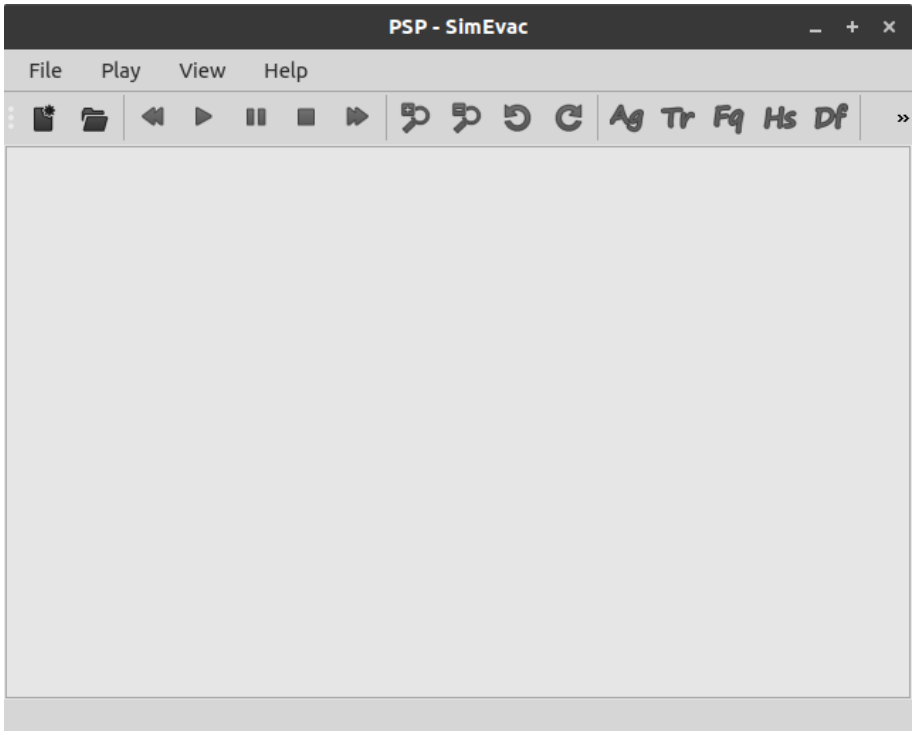
1. Wygenerowanie *n* zestawów parametrów wejściowych do symulatora *PSP* w celu pozyskania jego danych wyjściowych.
2. Stworzenie modelu uczenia maszynowego na podstawie wygenerowanych zestawów danych wejściowych oraz wyjściowych symulatora i nauczenie go na danych z punktu 1. Daje to możliwość generowania nowych danych wyjściowych bez używania symulatora.
3. Optymalizacja zestawu danych wyjściowych modelu na podstawie jego danych wejściowych używając metod optymalizacji *Gradient descent*.

4. PSP - Pederastian Simulator Project

W tym rozdziale opisany zostanie symutalor ruchu pieszych użyty w projekcie i jego sposób użycia wraz z potrzebnymi dodatkowymi plikami. Opisane zostaną użyte w projekcie parametry wejściowe do symulatora oraz metryki stworzone na podstawie jego danych wyjściowych.

4.1. Opis

Symulator *Pederastian Simulator Project* jest programem napisanym w języku C++ autorstwa Roberta Lubaś oraz Wojciecha Myśliwiec. Służy on do modelowania ruchu agentów (ludzi) podczas ewakuacji w danej przestrzeni. Symulowana przestrzeń jest dwuwymiarowa, z możliwością dodania wielu poziomów odpowiadających kolejnym piętrom budynku. Główne okno programu jest widoczne na rysunku 4.1.



Rys. 4.1. Główne okno symulatora.

4.2. Sposób użycia

Do przeprowadzenia symulacji potrzebne są następujące rzeczy:

– plik *.xml* zawierający parametry wejściowe symulacji,

– pliki *.jpg* i *.bmp* zwierające przestrzeń użytą do symulacji.

W oknie głównym programu należy wybrać opcje otwarcia pliku *.xml* po czym od razu zaczyna się symulacja w czasie rzeczywistym. Po symulacji, we wskazanym w pliku z parametrami folderze zostaną wygenerowane pliki *.csv* z danymi wyjściowymi symulacji.

4.3. Dane wejściowe i wyjściowe

Dane, na których miałem możliwość przeprowadzenia badań znajdowały się w:

– dane wejściowe - plik *.xml*,

– dane wyjściowe - pliki *.csv*.

Danymi wejściowymi, które wybrałem - na podstawie łatwości modyfikacji - do dalszej analizy były:

- **Panic spread factor** - współczynnik, który decyduje jak duzy wpływ ma **tryb paniki** (tryb, w którym agenci zwracają mniejszą uwagę na otoczenie oraz charakteryzują się bardziej chaotycznym ruchem) na agentów.
- **Panic cancel zone** - współczynnik określający odległość od wyjść, w obrębie jakiej agenci mają szansę na deaktywację trybu paniki.
- **Cancel panic chance**- podczas testów na anulowanie trybu paniki określa procent szans na powodzenie.
- **Choosing evacuation path mode** - tryb wyboru drogi ewakuacyjnej przez agentów. Dostępne są 4 tryby:
  - odległości,
  - gęstości przy wyjściu,
  - odległości oraz gęstości przy wyjściu,
  - odległości, gęstości przy wyjściu oraz popularności wyboru wyjścia.
- **Number of pederastians** - liczba agentów biorąca udział w symulacji.
- **Chaos level** - szansa na aktywację trybu paniki u agentów.
- **Density factor** - określa wpływ współczynnika gęstości wokół agenta na funkcję kary.
- **Frequency factor** - określa wpływ częstości wyboru danego pola na funkcję kary.
- **Panic factor** - potęguje część współczynników biorących udział w obliczaniu funkcji kary.
- **Distance factor** - wpływa na wybór agentów ruchu po skosie lub na wprost.
- **Randomness factor** - czynnik losowości dla wartości kary.
- **Pre-movement time mean value** oraz **Pre-movement time standard deviation** - średnia i odchylenie czasu *Pre-movement*, który ma wpływ na częstotliwość wykonywania testu na zmianę obranego wyjścia przez agentów.

– **Speed distribution mean value** oraz **Speed distribution standard deviation** - średnia i odchylenie prędkości agentów.

Dane wyjściowe wybrane wybrane do analizy przeze mnie były następujące:

– czas ewakuacji osatatniego agenta (odpowiadający czasowi ewakuacji wszystkich agentów),

– błąd średniokwadratowy średnich prędkości wszystkich agentów w stosunku do średniej prędkości najszybszego agenta.

Dodatkowe parametry wejściowe, które były mi potrzebne (lecz nie wpływały na sam proces badawczy) to:

– **Repeat number** - ilość powtórzeń symulacji wykonanej na pojedynczym pliku *.xml*.

– **Sim directory** - określa ścieżkę do folderu zawierającego pliki symulowanej przestrzeni.

– **Sim stat directory** - określa ścieżkę do folderu, do którego zostaną zapisane dane wyjściowe symulacji (pliki *.csv*).

5. Budowa modelu uczenia maszynowego

W tym rozdziale opisany będzie proces zbierania danych za pomocą symulatora PSP, tworzenie modelu uczenia maszynowego oraz optymalizacji jego wartości wyjściowych.

5.1. Zebranie danych

W celu zebrania danych, które miałyby trafić później do modelu uczenia maszynowego, stworzyłem skrypt pythonowy, który wygenerował 200 różnych zestawów parametrów wejściowych do symulatora. Rodzaje danych, które wygenerowałem znajdują się w rozdziale 4.3. Dla każdego zestawu danych ustawiłem parametrowi *Repeat number* wartość **10**, dzięki czemu dla każdego zestawu danych wejściowych generowane było 10 zestawów danych wyjściowych. Było to niezwykle cenne z 2 powodów:

– Do symulatora trzeba było ręcznie ładować plik *.xml*, a dzięki temu z jedngo pliku otrzymywałem od razu 10 zestawów plików wyjściowych.

– Ruchy agentów, nawet z takim samym zestawem parametrów, rożnią się między symulacjami, a dzięki powtórzeniom łatwiej było wyeliminować te różnice.

Dla części symulacji, symulator niespodziewanie wyłączył się podczas pracy, przez co niektórym zestawom danych wejściwych odpowiada mniej niż 10 zestawów danych wyjściowych. Ostatecznie z 200 zestawów parametrów wejściowych uzyskałem 1888 zestawów parametrów wyjsciowych. Oznacza to, że 5,6% z symulacji, które miały zostać przeprowadzone nie udały się.

W pierwszym podejściu, aby móc przetestować model w prostych warunkach, wybrałem z kazdego zestawu danych: wszystkie dane wejściowe, jedną daną wyjściową. Wybraną daną wyjściową był czas ewakuacji ostatniego agenta. Każdy zestaw danych zawierał więc **15** parametrów wejściowych oraz **1** parametr wyjściowy. Z tak przygotowanymi danymi miałem możliwość stworzenia pierwszego modelu.

5.2. Opis modelu Perceptron

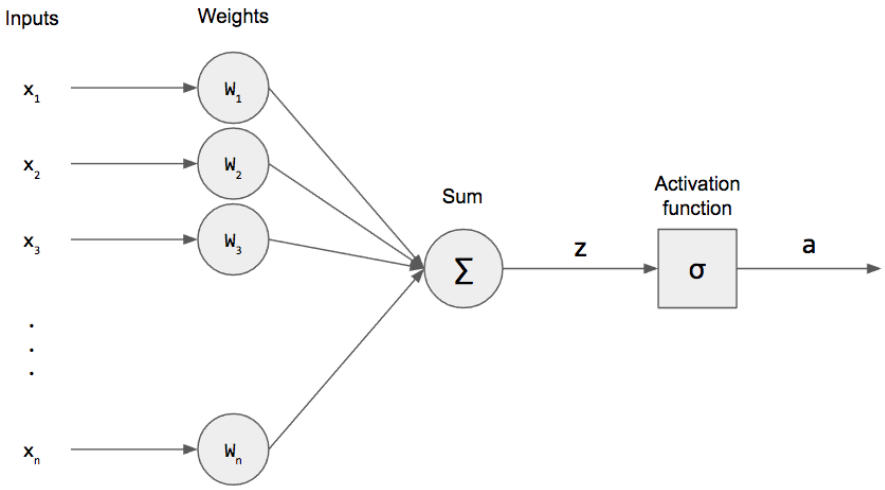
Na poczatek, wybrałem prosty model uczenia maszynowego - regresję liniową. Stworzyłem go wykorzystując **Perceptron**. Jest to prosta sieć neuronowa, która posiada zestaw neuronów wejściowych, kazdy odpowiadający pojedynczej danej wejsciowej. Składa się on także z jednego lub wielu niezależnych neuronów wyjściowych. Każdy z nich jest połączony ze wszystkimi neuronami wejściowymi za pomocą krawędzi posiadających wagi - definiują one jak duży wpływ na wyjście ma każda dana wejściowa. Jeśli wyrazimy:

- wejście jako wektor *X*,
- wyjście pojedyczego Perceptronu jako *a*,
- neuron jako wektor wag *N*,
- funkcję aktywacji jako σ - jest to funkcja wprowadzająca nieliniowość danych, najpopularniejsze z nich to *Sigmoid*, *ReLU (Rectified linear unit)* czy też *Softmax*

to zachodzi równanie:

$$Y = \sigma(NX) \tag{5.1}$$

Budowa Perceptronu z jednym neuronem wyjściowym jest widoczna na rysunku 5.1[1].



Rys. 5.1. Budowa Perceptronu z jednym neuronem wyjściowym.

[1] *https://pythonmachinelearning.pro/perceptrons-the-first-neural-networks/*

Podczas procesu uczenia *Perceptronu* ważne jest aby określić **loss function** - metrykę którą chcemy optymalizować. Dla problemów regresji najczęściej uzywaną funkcją kosztu jest **MSE** (Mean Squared Error). Kolejnym parametrem sieci *Perceptron* jest parametr **learning rate**, który określa jak bardzo będzie się zmieniać macierz/wektor wag na podstawie gradientu podczas każdej iteracji uczącej. Zbyt duży współczynnik *learning rate* może spowodować, że zamiast optymalizować funkcję kosztu będzie wręcz przeciwnie - będzie ona osiągała coraz większe wartości. Gdy jest on zbyt mały, jest on mniej niebezpieczny gdyż nie doprowadzi on do wzrostu wartości funkcji kosztu; problemem jest jednak wolniejszy proces uczący sieci co wymaga więcej epok uczących. Nie ma jednak uniwersalnej metody regulującej sposób dobierania *learning rate* - trzeba to robić eksperymentalnie.

5.3. Pierwszy model - tworzenie i analiza

Dla powyższych zastosowań stworzyłem więc *Perceptron* posiadający pojdeyncze wyjście oraz nie posiadający funkcji aktywacji - jako że model miał służyć do regresji a nie klasyfikacji. Zestawy danych podzieliłem w stosunku 80% : 20% na **dane treningowe** i **dane testowe**, pozwala to na trenowanie sieci na danych treningowych oraz sprawdzanie jej efektywności na danych testowych co daje możliwość walidacji tego, jak dobrze sieć radzi sobie z danymi, z którymi nie miała styczności.

Podsumowując, parametry pierwszego modelu były następujące:

- pojedyncza warstwa składająca się z jednego neuronu,
- **funkcja aktywacji** - brak,
- **learning rate** - 0.1,
- **metryka do optymalizacji** - *MSE*,
- **liczba epok uczacych** - 300,
- podział na **dane treningowe** i **dane testowe** - 80% : 20%.

Proces uczący został wykonany 10 razy od nowa na nienauczonym modelu, za kazdym razem wykonując permutację danych. Średnie *MSE* danych wyglądają następująco:

- **średni MSE danych treningowych** - 147,77 ($\pm$16,04),

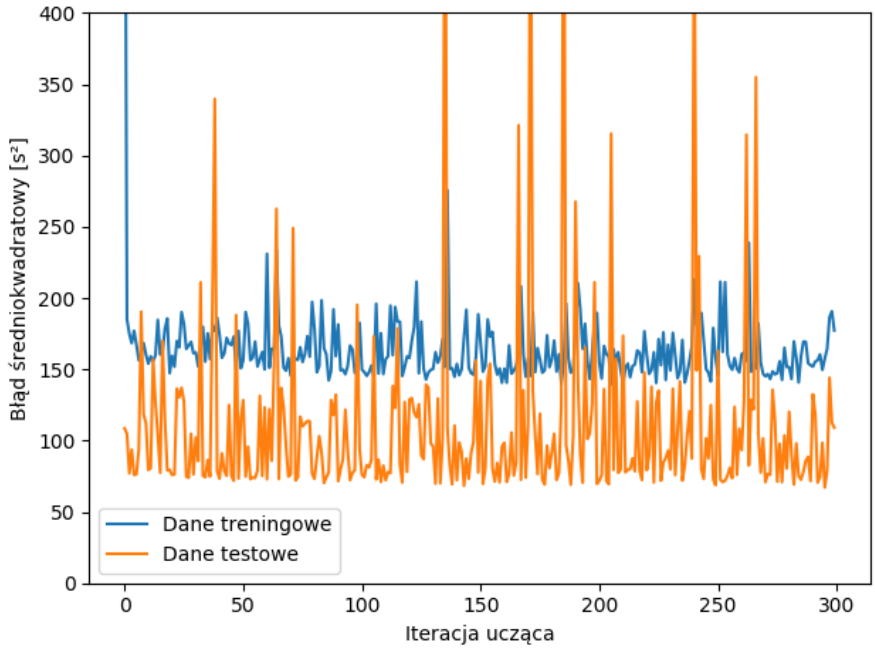
– **średni MSE danych testowych** - 166,18 (±50,77).

Wykres zależności *MSE* danych od iteracji uczacej najlepszego przypadku (biorąc pod uwagę *MSE* danych testowych) jest widoczna na rysunku 5.2. Taki rodzaj wykresów będzie nazywany dalej **wykresami krzywych uczących dla najlepszego przypadku**. Analizując go, doszedłem do następujących wniosków:

– *MSE* danych treningowych jak i testowych jest na wysokim poziomie co oznacza, że model ma **wysoki bias**. Oznacza to, że użyty model jest zbyt prosty i nie ważne jak dużo danych zostanie do niego podanych i jak długo będzie się on uczył - *MSE* danych nie obniży się poniżej pewnego poziomu.

– Podczas uczenia *MSE* danych jest obarczony duzym szumem, co oznacza niestabilność danych lub modelu.

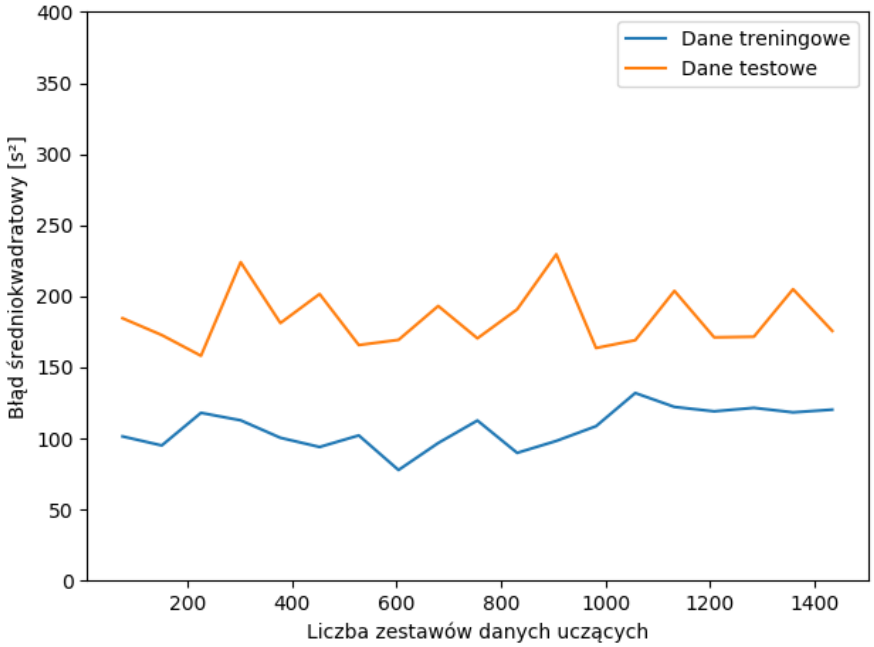
– *MSE* danych testowych jest przez większość iteracji mniejsze niż treningowych co oznacza, że dla danych których sieć nie widziała, przewiduje ona wartości bliższe prawdzie niż na danych na których się uczyła. Jest to niecodzienne zachowanie co również dowodzi niestabilności modelu lub danych.



Rys. 5.2. Krzywe uczące dla najlepszego przypadku w pierwszym modelu.

Aby mieć szerszy pogląd na to, jaki wpływ ma ilość danych na których się uczy sieć na *MSE* tych danych - postanowiłem uruchomić proces uczacy kolejne 20 razy

dla pojedynczej permutacji danych wejściowych, lecz przy zwiększającej się ilości danych treningowych. Wykres obrazujący wynik tych badań jest widoczny na rysunku 5.3. Ten rodzaj wykresów dalej nazywany będzie **wykresami krzywych uczących dla zmiennej ilości danych**. Informacją jaką można z niego wyciągnąć jest ilość danych potrzebna do tego aby osiągnąć dany pułap *MSE* danych. Widać że podczas gdy przy danych treningowych, większa ich ilość spowodowała spadek ich *MSE* to przy danych testowych nie spowodowało to większej różnicy.



Rys. 5.3. Krzywe uczące dla zmiennej ilości danych w pierwszym modelu.

Podsumowując, obecny model ma wysoki bias - jest zbyt prosty dla danych, przez co większa ich ilość oraz dłuższy proces uczący nie dają żadnych efektów. Na wykresach widać niestabilność modelu lub danych, co powoduje bardzo duże szumy na wykresach *MSE*. Co więcej - niepokojący jest fakt, że *MSE* danych testowych był często niższy niż *MSE* danych treningowych.

5.4. Normalizacja danych

Mając na uwadze problemy stworzonego modelu, w pierwszej kolejności postanowiłem przejrzeć dane wejściowe do modelu. Ze wszystkich zestawów danych obliczyłem więc średnią każdego z parametrów wejściowych do sieci neuronowej.

Tabela 5.1. Porównanie średnich wartości parametrów wejściowych do modelu.

Nazwa parametru	Średnia wartość
Panic spread factor	1.74
Panic cancel zone	0.53
Cancel panic chance	51.59
Choosing evacuation path mode	2.53
Number of pederastian	271.8
Chaos level	53.9
Density factor	5.26
Frequency factor	5.33
Panic factor	1.45
Distance factor	2.51
Randomness factor	0.49
Pre-movement time mean value	5.44
Pre-movement time standard deviation	0.51
Speed distribution mean value	5.62
Speed distribution standard deviation	0.54

Wyniki są widoczne w tabeli 5.1. Widać w niej, że niektóre parametry mają wartości 2 rzędy wielkości większe niż inne. Takie różnice pomiędzy różnymi parametrami wejściowymi powodują faworyzowanie przez sieć jednych parametrów ponad drugie. Postanowiłem więc zastosować normalizację danych według wzoru:

$$X_{new} = \frac{X_{old} - \mu}{\sigma} \tag{5.2}$$

Gdzie:

– μ - wektor średniej parametrów,

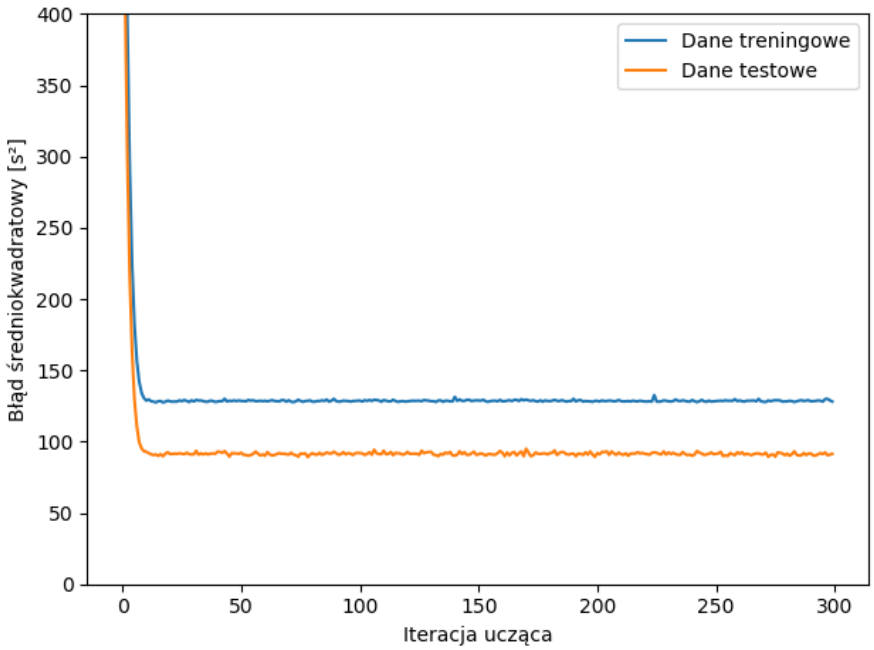
– σ - wektor odchylenia parametrów.

Posiadając tak przygotowane dane wykonałem - tak jak w punkcie 5.3 - 10 cykli nauczania z różnymi permutacjami danych oraz 20 cykli z jedną permutacją, lecz zmieniającym się rozmiarem zestawu danych treningowych. Średnie *MSE* danych są następujące:

– **średni MSE danych treningowych** - 118,64 ($\pm$6,07),

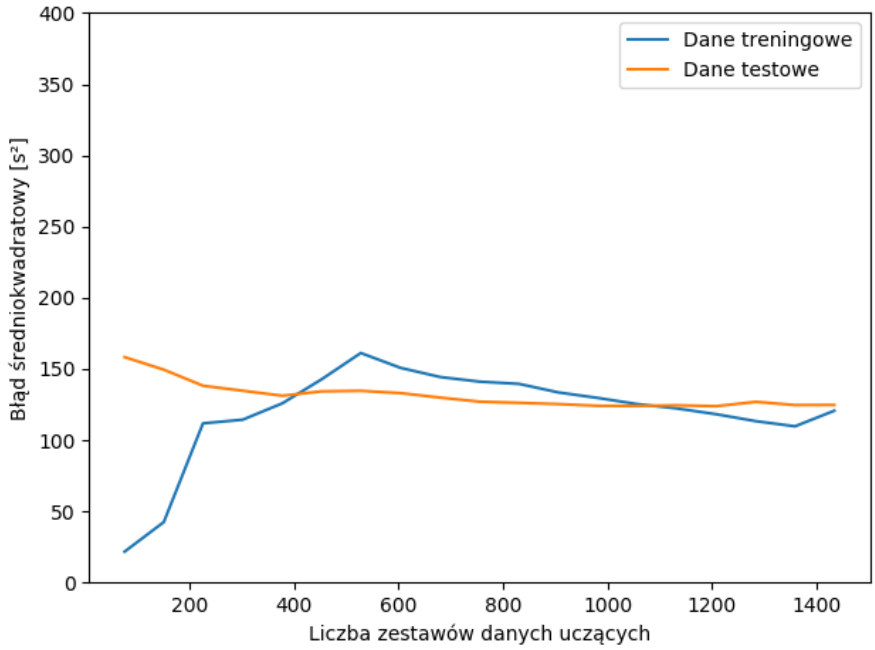
– **średni MSE danych testowych** - 130,70 (±23,47).

Wykres krzywych uczących dla najlepszego przypadku widoczny jest na rysunku 5.4. Dodatkowo patrząc na wykres krzywych uczących dla zmiennej ilości danych z rysunku 5.5 - widać zależność *MSE* danych od liczby danych treningowych.



Rys. 5.4. Krzywe uczące dla najlepszego przypadku w pierwszym modelu ze znormalizowanymi danymi.

Podsumowując, pierwszą rzeczą rzucającą się w oczy jest duża stabilność wartości na wykresach - eliminuje to sytuacje w których wynik uczenia był winą przypadku, jako że kilka iteracji wcześniej lub później *Perceptron* dawał zupełnie inny rezultat. Dzięki większej stabilności sieci łatwiej jest także wyciągać wnioski na podstawie wykresów. Lepiej widoczny jest problem zauważony wcześniej - wysoki bias modelu. Objawia się on tym, że na początku *MSE* danych maleje, aż do pewnego momentu gdzie jest stały i żadna ilość danych albo długość uczenia nie spowoduje jego zmniejszenia. Ostatnią rzeczą, która jest taka sama jak w rozdziale 5.3 - *MSE* danych testowych często jest mniejsze niż danych treningowych.



Rys. 5.5. Krzywe uczące dla zmniennej ilości danych w pierwszym modelu ze znormalizowanymi danymi.

5.5. Drugi model - tworzenie i analiza

Chcąc rozwiązać główny problem poprzedniego modelu - wysoki bias - potrzebowałem modelu, który potrafi nauczyć się bardziej skomplikowanych funkcji. Takim modelem była sieć neuronowa z kilkoma warstwami. Dzięki każdej następnej warstwie ma ona możliwość uczenia się coraz bardziej skomplikowanych funkcji. Sieć jaką stworzyłem miała 1 **warstwę ukrytą** - warstwę neuronów pomiędzy wejściami a warstwą wyjść - z 30 neuronami. Jako funkcję aktywacji wybrałem funkcję *ReLU*. Podsumowując, parametry drugiego modelu były następujące:

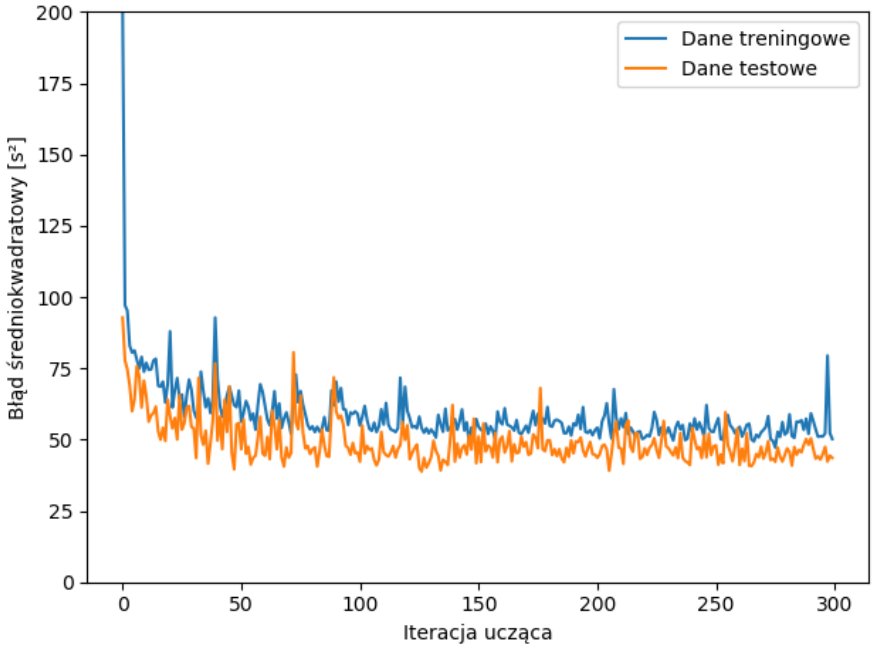
- 1 warstwa ukryta z 30 neuronami i funkcją aktywacji ReLU,
- warstwa wyjściowa składająca się z jednego neuronu bez funkcji aktywacji,
- **learning rate** - 0.1,
- **metryka do optymalizacji** - MSE,
- **liczba epok uczacych** - 300,
- podział na **dane treningowe** i **dane testowe** - 80% : 20%.

Proces uczący przeprowadziłem identycznie jak poprzednio - wykonałem 10 osobnych cykli uczących z różnymi permutacjami danych oraz 20 osobnych cykli uczących na zmieniającej się wielkości zestawu danych traningowych. Średnie *MSE* danych z 10 cykli są następujące:

– **średni MSE danych treningowych** - 47,47 (±6,08),

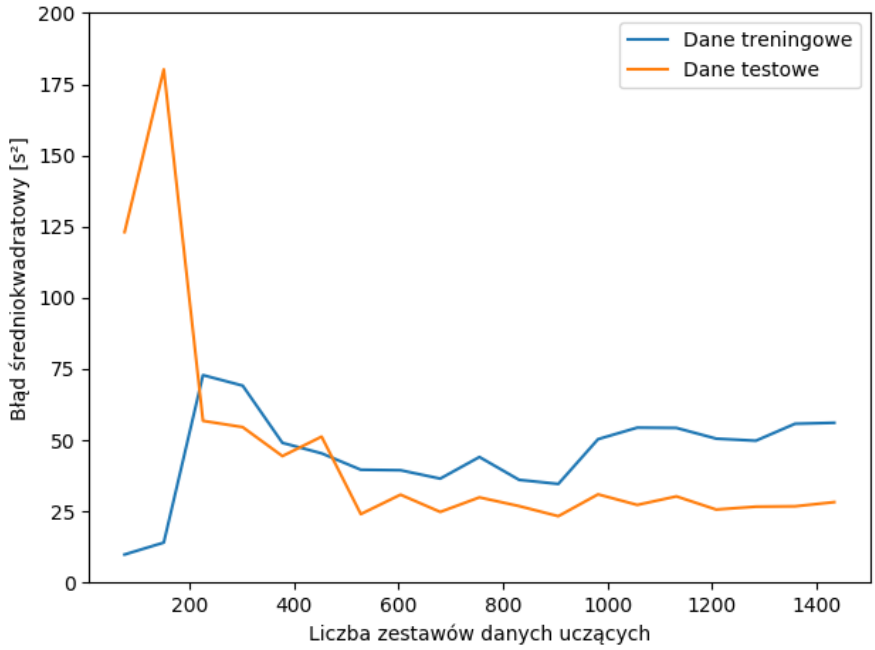
– **średni MSE danych testowych** - 73,66 (±25,77).

Wykres krzywych uczacych dla najlepszego przypadku widoczny jest na rysunku 5.6 a krzywych uczących dla zmiennej ilości danych - na rysunku 5.7.



Rys. 5.6. Krzywe uczące dla najlepszego przypadku w drugim modelu.

Porównując te dane z tymi z sekcji 5.4 widać znaczącą poprawę modelu - trzykrotny spadek średnich *MSE* danych treningowych oraz testowych. Mimo tego widać, *MSE* po osiągnięciu pewnego pułapu przestaje maleć - oznacza to, że model dalej ma duży bias.



Rys. 5.7. Krzywe uczące dla zmiennej ilości danych w drugim modelu.

5.6. Zwiększenie stopnia skomplikowania modelu

Próbując zmniejszyć bias sieci neuronowej, postanowiłem przeprowadzić eksperymenty z innymi parametrami sieci, a mianowicie:

– liczba warstw ukrytych - **1**, liczba neuronów w warstwie - **40**;

– liczba warstw ukrytych - **2**, liczba neuronów w każdej warstwie - **20**;

– liczba warstw ukrytych - **2**, liczba neuronów w każdej warstwie - **40**;

Zestawienie wynikow z sekcji 5.5 oraz powyższych kombinacji parametrów jest następujące:

– krzywe uczące dla najlepszych przypadków są widoczne na rysunku 5.8,

– krzywe uczące dla zmiennej ilości danych pokazane są na rysunku 5.9.

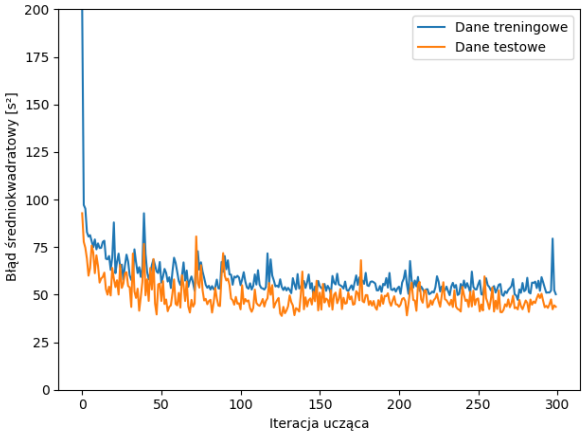
Dodatkowo średnie *MSE* z 10 różnych permutacji danych zestawiłem w tabeli 5.2.

Mimo zwiększenia stopnia skomplikowania sieci, *MSE* danych treningowych i testowych nie wykazywały tendencji spadkowej. To co powinno się stać w takim przypadku, to *MSE* danych treningowych powinien osiągać coraz niższe wartości wraz ze skomplikowaniem modelu. Na wykresach również widac, że dane testowe miały

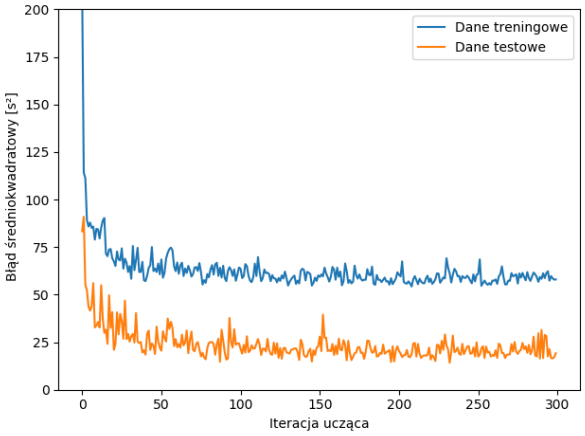
Tabela 5.2. Porównanie średnich wartości *MSE* danych dla różnych kombinacji parametrów sieci.

Ilość warstw	Ilość neuronów w warstwie	MSE danych treningowych	MSE danych testowych
1	20	47,47 ($\pm$6,08)	73,66 ($\pm$25,77)
1	40	53,16 ($\pm$4,93)	47,87 ($\pm$20,13)
2	20	59,49 ($\pm$10,81)	89,52 ($\pm$22,04)
2	40	64,13 ($\pm$7,30)	70,68 ($\pm$20,42)

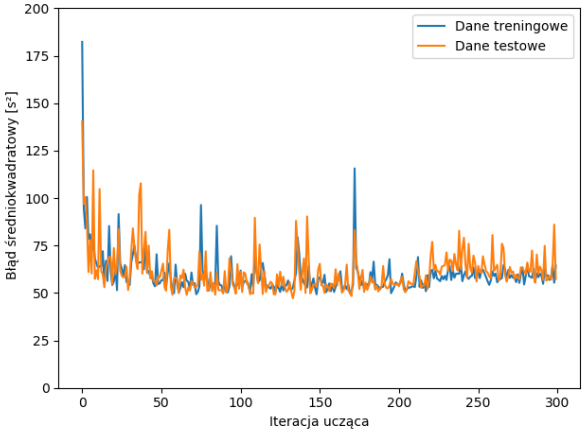
czasem *MSE* mniejszych od zestawu treningowego. Wnioskiem tej analizy jest to, że problem z uczeniem sieci leży nie w modelu, a w danych.



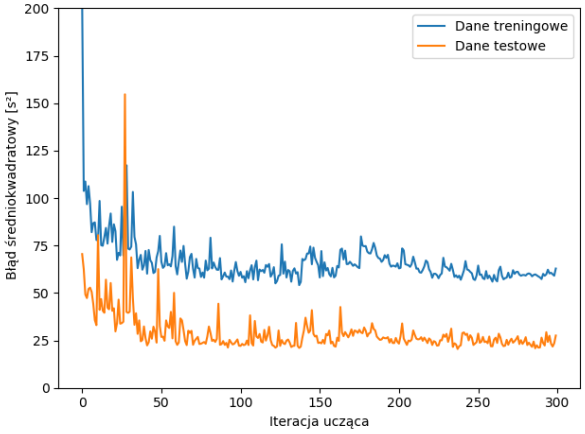
(a) 1 warstwa ukryta, 20 neuronow w każdej warstwie



(b) 1 warstwa ukryta, 40 neuronow w każdej warstwie

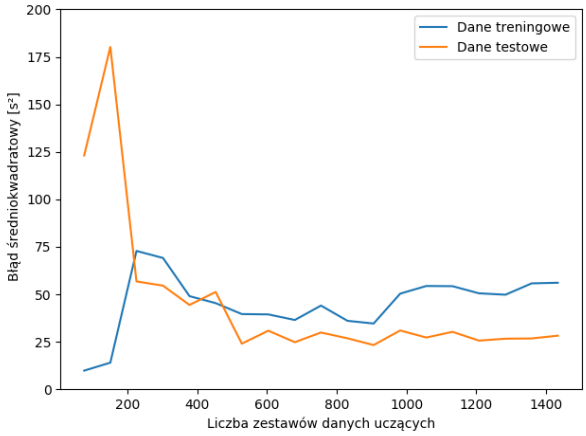


(c) 2 warstwy ukryte, 20 neuronow w każdej warstwie

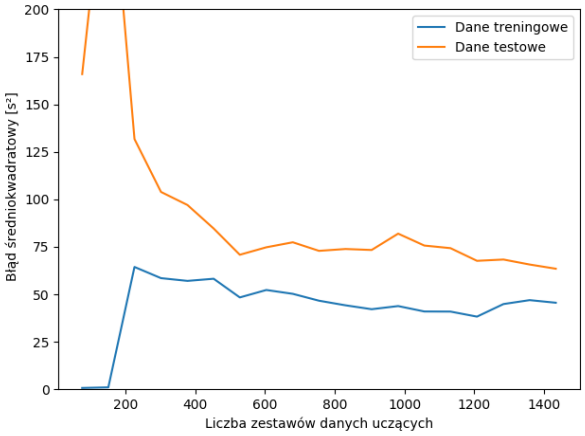


(d) 2 warstwy ukryte, 40 neuronow w każdej warstwie

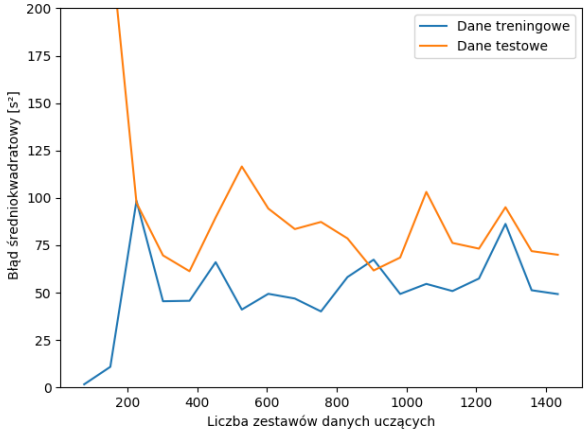
Rys. 5.8. Krzywe uczące dla najlepszych przypadków przy użyciu różnych kombinacji parametrów w drugim modelu.



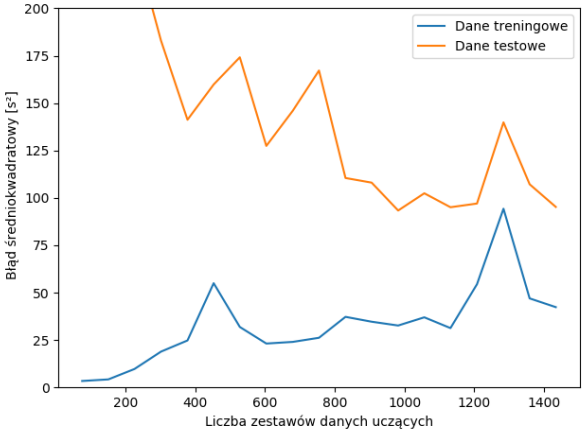
(a) 1 warstwa ukryta, 20 neuronow w każdej warstwie



(b) 1 warstwa ukryta, 40 neuronow w każdej warstwie



(c) 2 warstwy ukryte, 20 neuronow w każdej warstwie



(d) 2 warstwy ukryte, 40 neuronow w każdej warstwie

Rys. 5.9. Krzywe uczące dla zmiennej ilosci danych przy różnych kombinacjach parametrów w drugim modelu.

5.7. Uśrednianie danych

Wiedząc że problem leży w danych postanowiłem ulepszyć istniejące dane, uśredniając je. W sekcji 5.1 napisałem, że dla każdego zbioru parametrów wejściowych wygenerowałem do 10 zbiorów parametrów wyjściowych. Dla każdego zestawu wejściowego obliczyłem średnią zestawów parametrów wyjściowych co zmniejszyło ilość danych prawie 10-krotnie. Ostatecznie mój nowy zbiór zawierał 200 zestawów danych (przy czym, podczas procesu uczenia zostały one podzielone na dane treningowe i testowe).

Po przetworzeniu danych postanowiłem powtórzyć badanie z sekcji 5.6, używając kombinacji parametrów:

– ilość warstw ukrytych - **1** i **2**,

– ilość neuronów w każdej warstwie - **20** i **40**.

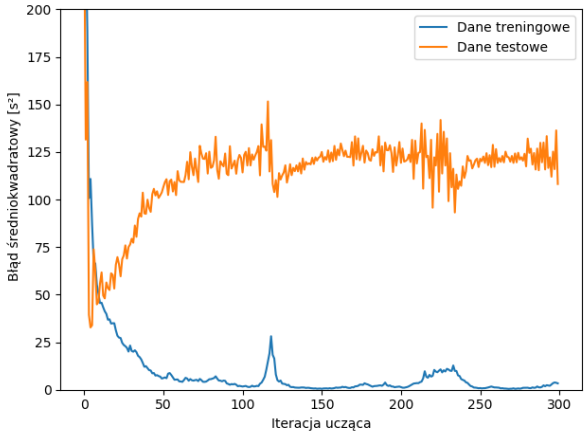
Po powtórzeniu poprzedniego badania na uśrednionych danych, wyniki widać na rysunkach 5.10 i 5.11. Średnie *MSE* danych dla kombinacji parametrów jest umieszczone w tabeli 5.3.

Analizując dane, można zauważyć że po uśrednieniu zachowują się one zgodnie z przewidywaniami - większe skomplikowanie sieci neuronowej zmniejszyło dramatycznie *MSE* danych treningowych. Co było również spodziewane - *MSE* danych testowych pozostał wysoki; jednakże na wykresach 5.11 (c) i (d) widać ciekawą tendencję - wygląda na to, że przy większych rozmiarach zbioru danych nawet *MSE* zbioru testowego zaczyna się obniżać. Podobną tendencję, lecz słabszą można też zauważyć na wykresach 5.11 (a) i (b) po przekroczeniu pułapu 140 zestawów danych uczących. Gdy model jest zbyt dobrze przyuczony do danych treningowych i źle generalizuje na danych testowych, mówi się że posiada on **wysoką wariancję**.

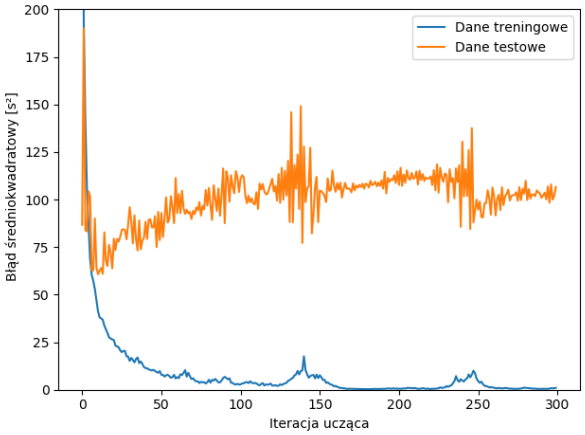
Ze względu na duży czas oczekiwania oraz potrzebę ciągłego nadzoru nad symulatorem, postanowiłem w pirewszej kolejności poszukać innych sposobów na zwiększenie efektywności modelu.

Tabela 5.3. Porównanie średnich wartości *MSE* danych przy uśrednionych danych.

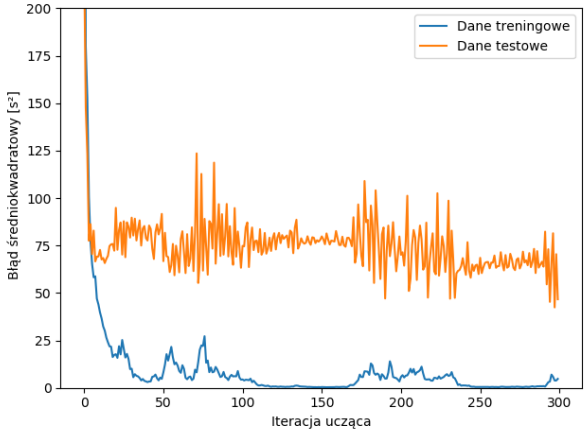
Ilość warstw	Ilość neuronów w warstwie	MSE danych treningowych	MSE danych testowych
1	20	1,38 (±0,81)	189,06 (±60,12)
1	40	2,21 (±2,02)	182,91 (±50,58)
2	20	6,06 (±8,93)	113,56 (±55,77)
2	40	2,02 (±4,38)	108,03 (±40,39)



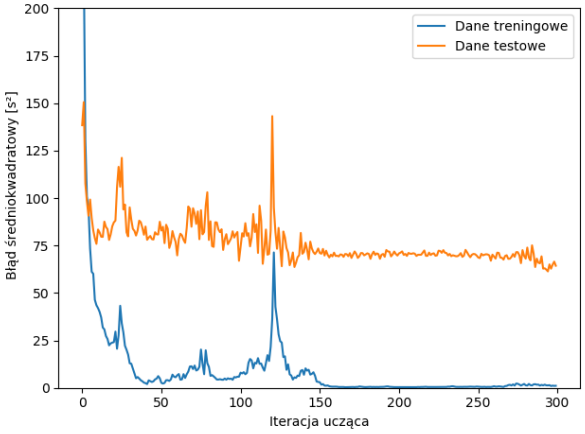
(a) 1 warstwa ukryta, 20 neuronow w każdej warstwie



(b) 1 warstwa ukryta, 40 neuronow w każdej warstwie

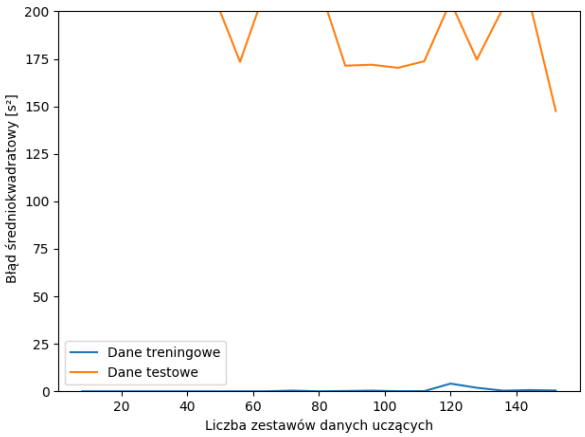


(c) 2 warstwy ukryte, 20 neuronow w każdej warstwie

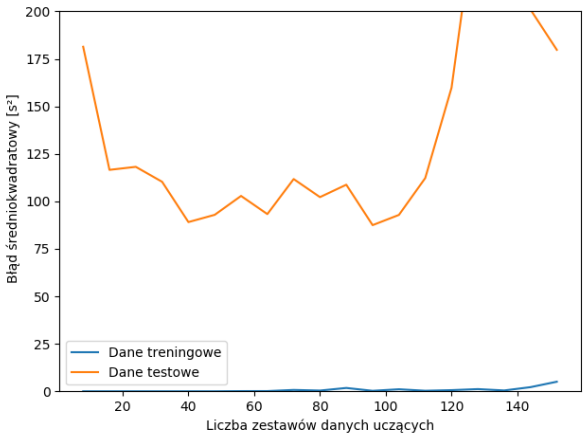


(d) 2 warstwy ukryte, 40 neuronow w każdej warstwie

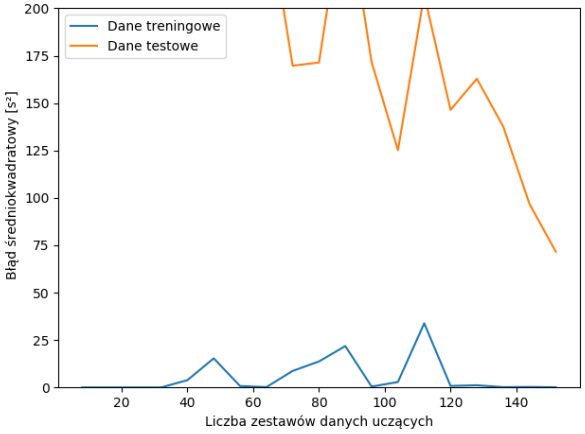
Rys. 5.10. Krzywe uczące dla najlepszych przypadków przy uśrednionych danych w drugim modelu.



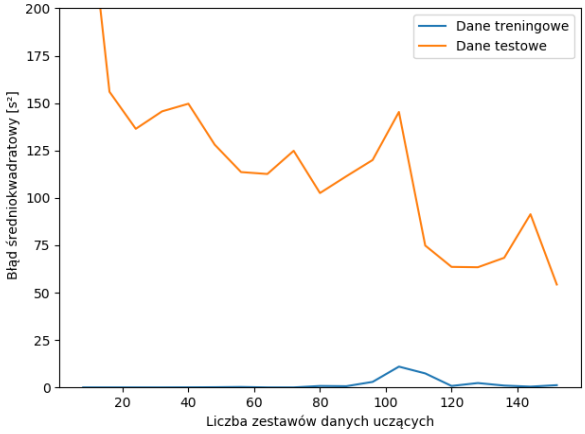
(a) 1 warstwa ukryta, 20 neuronow w każdej warstwie



(b) 1 warstwa ukryta, 40 neuronow w każdej warstwie



(c) 2 warstwy ukryte, 20 neuronow w każdej warstwie



(d) 2 warstwy ukryte, 40 neuronow w każdej warstwie

Rys. 5.11. Krzywe uczące dla zmniennej ilości uśrednionych danych w drugim modelu.

5.8. Regularyzacja modelu

Chcąc obnizyć wariancję modelu uczenia maszynowego stosuje się techniki zwane regularyzacją. Najprostsze z nich czyli regularyzacja **L1** i **L2** polegają na dodaniu współczynnika kary do funkcji kosztu. Karze ona model za posiadanie wysokich wag na połączeniach pomiędzy neuronami. Dla porównania mając funkcję kosztu 5.3, używając regularyzacji *L1* będzie ona wyglądać jak na równaniu 5.4 a regularyzacji *L2* - tak jak na równaniu 5.5. Współczynnik λ to tak zwany **współczynnik regularyzacji** - odpowiada on za siłę, z jaką wagi będa zmniejszane.

$$S = \sum_{i=1}^{n} (y_i - f(x_i))^2 \tag{5.3}$$

$$S = \sum_{i=1}^{n} (y_i - f(x_i))^2 + \lambda \sum_{i=1}^{k} |w_i| \tag{5.4}$$

$$S = \sum_{i=1}^{n} (y_i - f(x_i))^2 + \lambda \sum_{i=1}^{k} w_i^2 \tag{5.5}$$

Bazując na badaniach z poprzedniego rozdziału, wybrałem model z jedną 20-neuronową warstwą ukrytą aby przeprowadzić na nim eksperymenty z regularyzacją. Mój wybór był umotywowany tym, że był to najprostszy z 4 modeli, a jednocześnie miał możliwość prawie idealnego nauczenia się danych treningowych. Parametry regularyzacji jakie wybrałem do badań to:

- funkcje regularyzujące - **L1** i **L2**,
- parametr regularyzacji λ - **0,1**; **0,3**; **1**; **3**.

Po przeprowadzeniu eksperymentów ze wszystkimi powyższymi kombinacjami parametrów wysrysowałem wykresy krzywych uczących dla najlepszych przypadków na rysunkach 5.12 i 5.13 oraz wykresy krzywych uczących dla zmiennej ilości danych na rysunkach 5.14 i 5.15. Dodatkowo zestawienie średnich *MSE* dla owych kombinacji umieściłem w tabeli 5.4.

Po przeanalizowaniu wszystkich wykresów oraz wartości z tabeli nasunęły się mi następujące spostrzeżenia:

- Bazując na tabeli, najlepsza kombinacja parametrów to funkcja regularyzacji *L1* wraz z parametrem λ równym 1. Przy tej kombinacji średni *MSE* danych testowych jest najniższy oraz ma najniższe odchylenie.

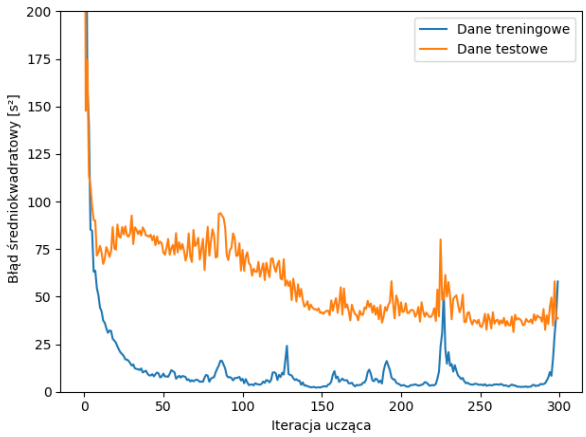
Tabela 5.4. Porównanie średnich wartości *MSE* danych przy regularyzacji.

Rodzaj regularyzacji	Ilość neuronów w warstwie	MSE danych treningowych	MSE danych testowych
L1	0,1	11,65 ($\pm$15,70)	99,61 ($\pm$43,27)
L1	0,3	7,68 ($\pm$1,04)	82,24 ($\pm$39,32)
L1	1	15,89 ($\pm$3,44)	36,21 ($\pm$24,35)
L1	3	40,44 ($\pm$21,46)	64,26 ($\pm$39,62)
L2	0,1	7,42 ($\pm$1,74)	102,93 ($\pm$47,46)
L2	0,3	20,04 ($\pm$3,16)	76,49 ($\pm$36,76)
L2	1	53,44 ($\pm$4,77)	81,18 ($\pm$43,06)
L2	3	103,04 ($\pm$7,29)	62,86 ($\pm$34,93)

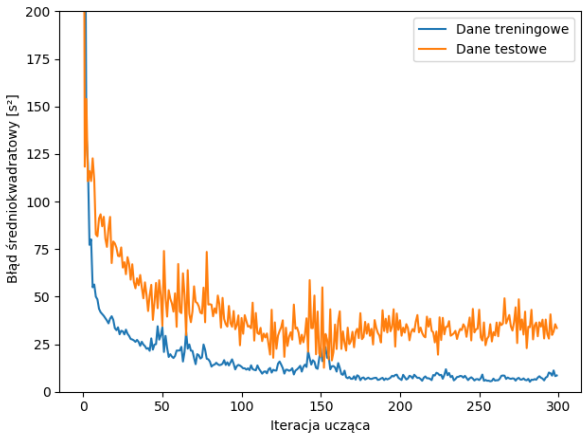
– Patrząc na krzywe uczące dla najlepszych przypadków można zauważyć, że im wiekszy współczynnik λ, tym bardziej krzywa *MSE* danych testowych ma tendencję do schodzenia poniżej krzywej *MSE* danych treningowych. Oznacza to, że problem lepszego przewidywania zbioru testującego niż trenującego nie został wyeliminowany w całości. Widać także na nich, że dla λ, który w powyższym punkcie wydaje się najlepiej rokującą opcją, zdarzają się takie przypadki.

– Porównując krzywe uczące dla zmiennej ilości danych L1 z krzywymi L2, wśród tych pierwszych widać większą tendencję na zmniejszenie *MSE* danych testowych przy większej ilości danych.

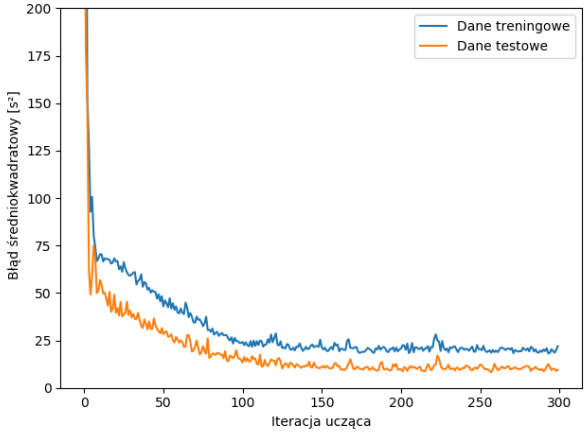
Podsumowując, badając powyższe dane okazuje się, że uśrednianie danych nie wyeliminowało do końca problemu gorszego przyuczenia sieci do danych treningowych niż do testowych. Jednocześnie wstępnym wnioskiem na obecnych danych jest to, że regularyzacja *L1* pomaga bardziej niż *L2*. Jednakże aby mieć tego pewność, trzeba dokładniej sprawdzić dane oraz ewentualnie zebrać ich więcej, po czym zrobić powyższe testy jeszcze raz.



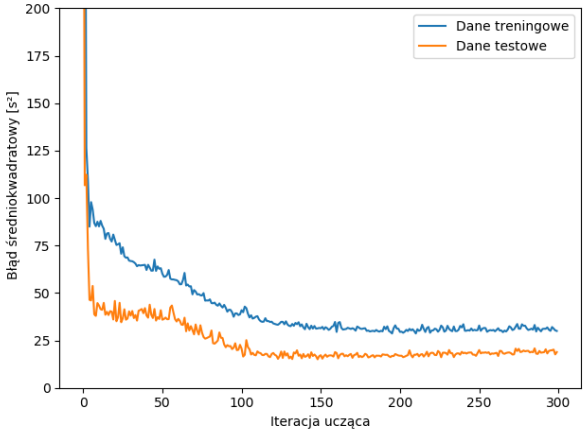
(a) parametr λ - 0,1



(b) parametr λ - 0,3

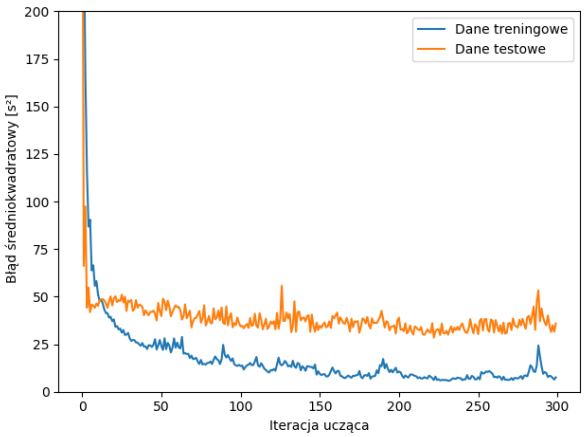


(c) parametr λ - 1

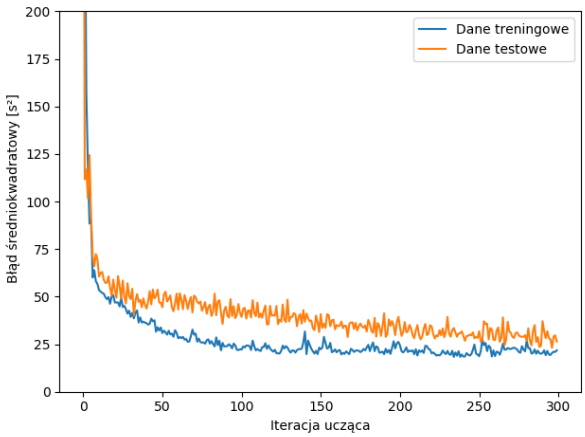


(d) parametr λ - 3

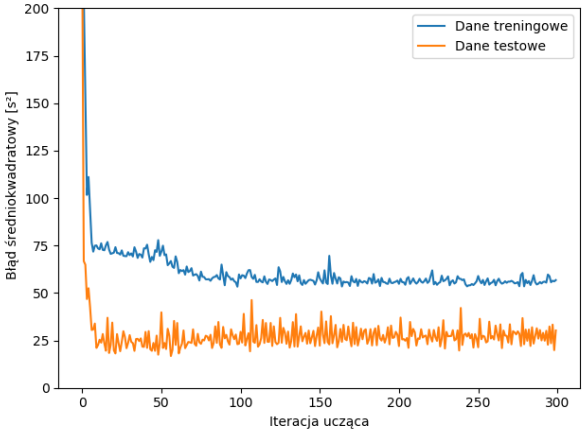
Rys. 5.12. Krzywe uczące dla najlepszych przypadków używając regularyzacji L1 w drugim modelu.



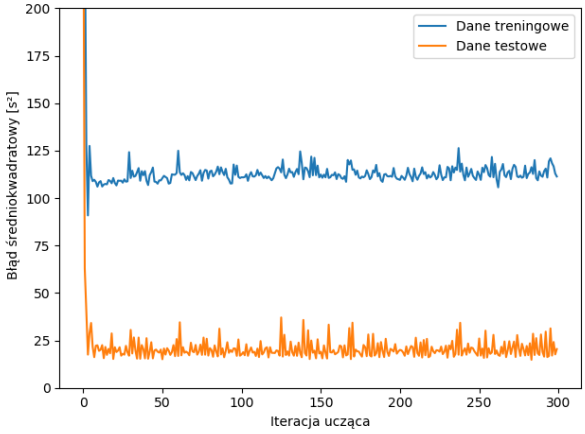
(a) parametr λ - 0,1



(b) parametr λ - 0,3

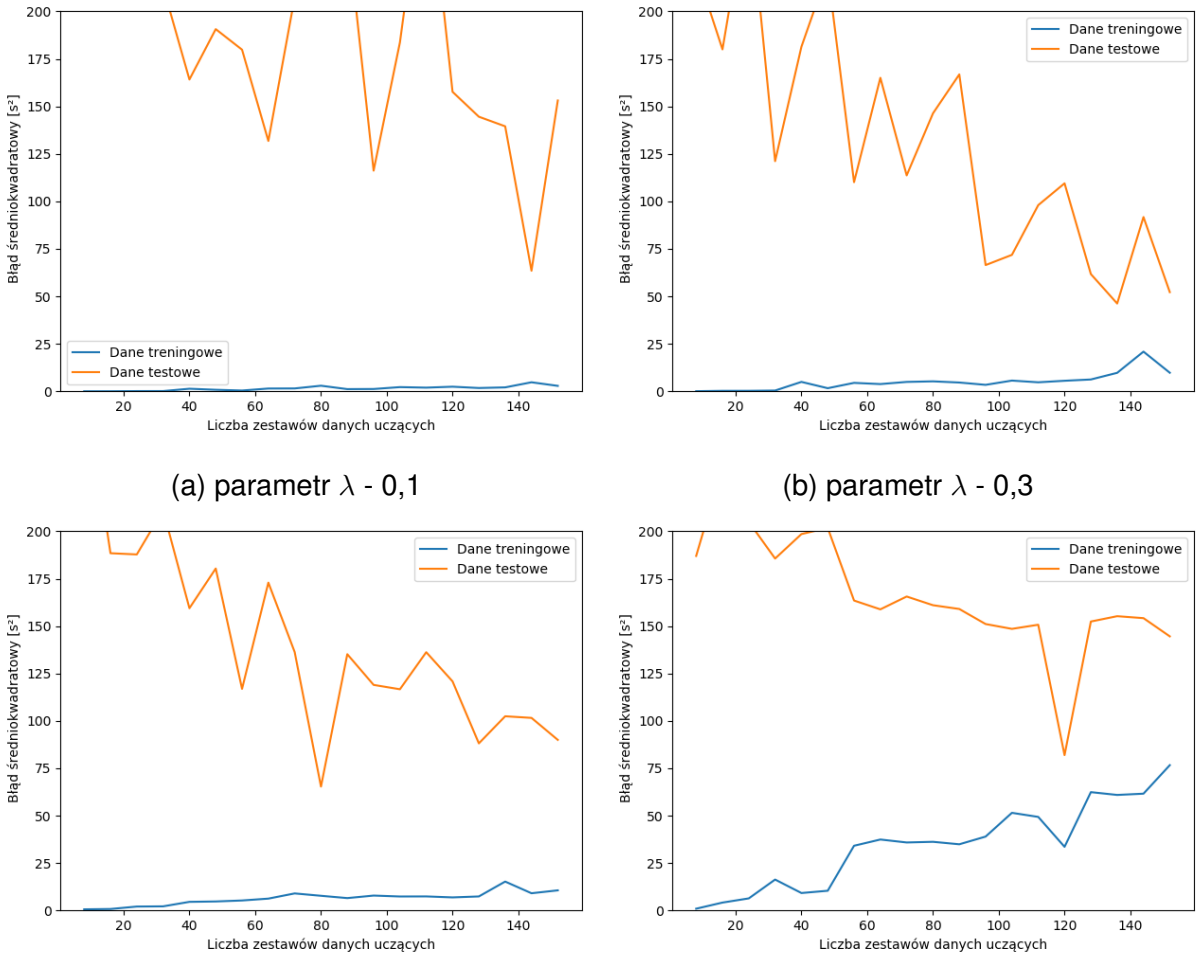


(c) parametr λ - 1



(d) parametr λ - 3

Rys. 5.13. Krzywe uczące dla najlepszych przypadków używając regularyzacji L2 w drugim modelu.



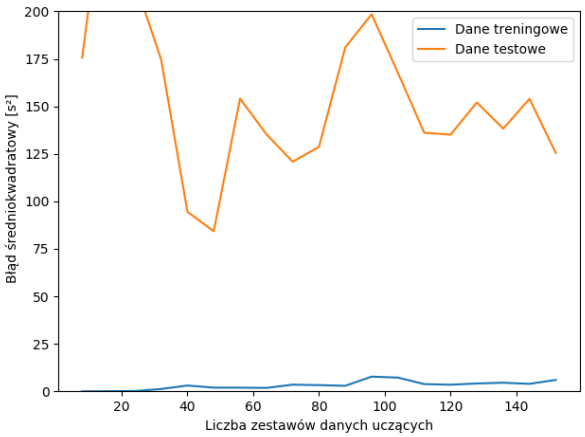
(a) parametr λ - 0,1

(b) parametr λ - 0,3

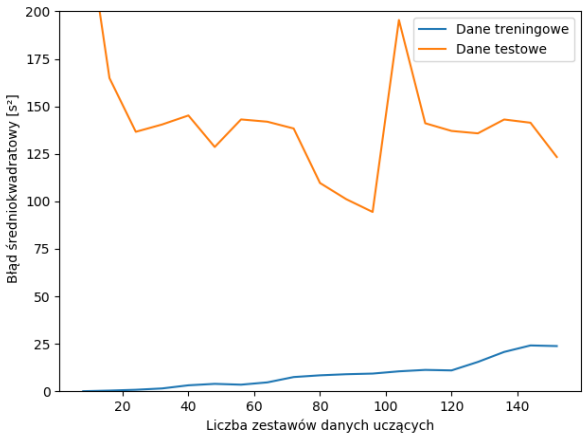
(c) parametr λ - 1

(d) parametr λ - 3

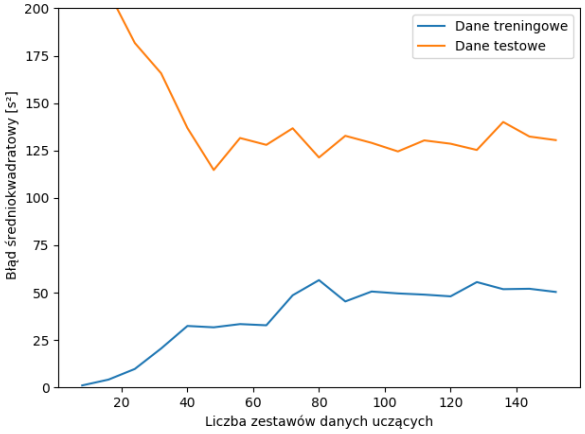
Rys. 5.14. Krzywe uczące dla zmiennej ilości danych używając regularyzacji L1 w drugim modelu.



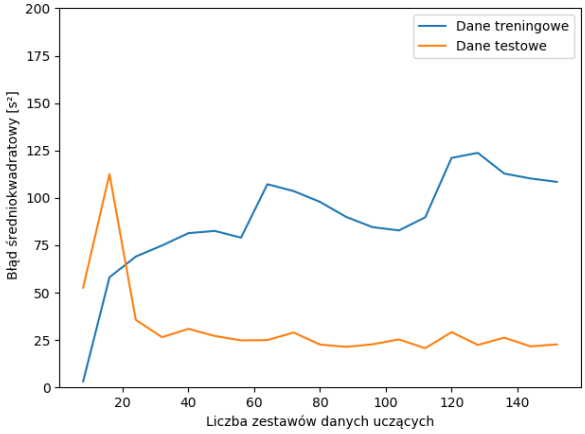
(a) parametr λ - 0,1



(b) parametr λ - 0,3



(c) parametr λ - 1

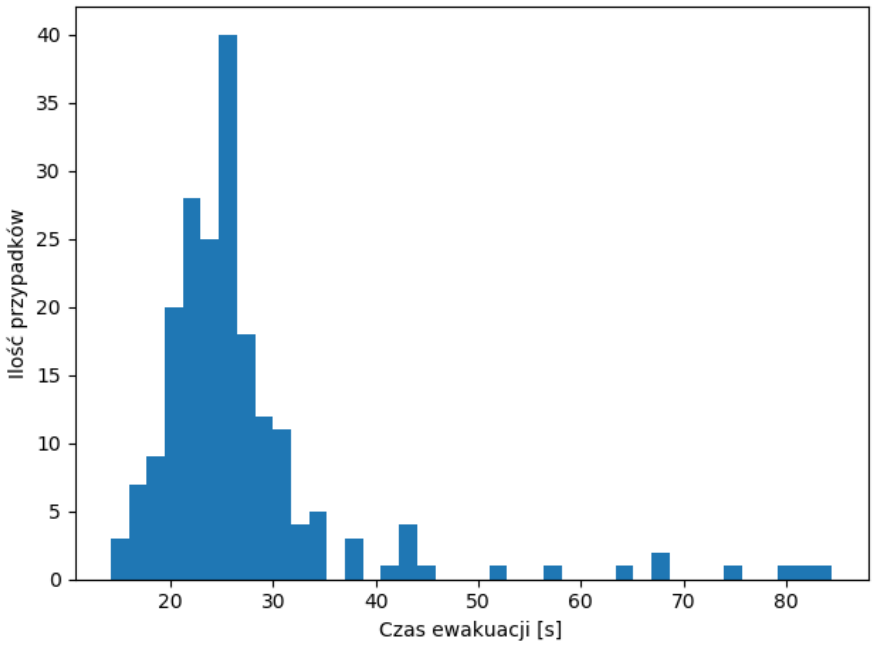


(d) parametr λ - 3

Rys. 5.15. Krzywe uczące dla zmiennej ilości danych używając regularyzacji L2 w drugim modelu.

5.9. Badanie danych

Zastanawiającym problemem pojawiającym się w poprzednich rozdziałach było to, że często po nauczeniu sieci neuronowej, przewiduje ona dane testujące lepiej niż treningowe. Zwiększając poziom skomplikowania sieci neuronowej sytuacja się nie zmieniała, co dowiodło że problem leży nie po stronie modelu, a po stronie danych. Z tego powodu postanowiłem sprawdzić jakie wartości przyjmują dane wyjściowe w moim zbiorze danych. Wyrysowałem więc histogram obrazujący ilość danych dla danego czasu ewakuacji widoczny na rysunku 5.16.



Rys. 5.16. Zależność ilości danych od czasu ewakuacji.

Analizując powyższy histogram widać, że większośc danych wyjściowych jest skupiona poniżej 40 sekund czasu ewakuacji, a jedynie nieliczne wyniki przekraczają górną granicę. Taki rozrzut części danych może powodować, że będą one trudne do nauczenia przez model - postanowiłem więc odrzucić wszystkie zestawy danych, których dana wyjściowa - czas ewakuacji - jest równe lub większe 40 sekund. Po takiej operacji zostało wektorów danych wejściowych i wyjsciowych. Na nich wykonałem badanie identyczne jak w sekcji 5.8. Średnie wyniki umieściłem w tabeli 5.5 oraz w tabelach, krzywe uczenia dla najlepszych przypadków na rysunkach 5.17 i 5.18 oraz krzywe uczenia przy zmiennej ilości danych - na rysunkach 5.19 i 5.20.

Badając powyższe wykresy nasuwają się następujące wnioski:

Tabela 5.5. Porównanie średnich wartości *MSE* danych po odrzuceniu części danych.

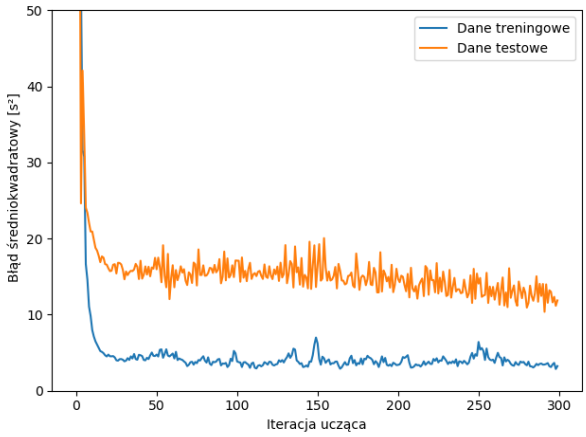
Rodzaj regularyzacji	Ilość neuronów w warstwie	MSE danych treningowych	MSE danych testowych
L1	0,1	4,13 (±0,65)	15,63 (±3,32)
L1	0,3	7,21 (±0,64)	10,82 (±3,16)
L1	1	10,45 (±0,49)	9,35 (±2,73)
L1	3	12,87 (±0,92)	13,87 (±2,46)
L2	0,1	4,31 (±1,58)	12,97 (±3,42)
L2	0,3	8,08 (±0,72)	12,63 (±2,36)
L2	1	11,63 (±0,76)	12,97 (±3,12)
L2	3	16,90 (±1,17)	16,43 (±2,89)

– Odrzucenie przypadków skrajnie odstających od reszty obniżyło *MSE* danych co najmniej 3-krotnie (a w dotychczzas najgorszych przypadkach nawet 10-krotnie), co daje jednoznaczną odpowiedź gdzie leżał problem z lepszym przyuczeniem sieci do danych testowych.

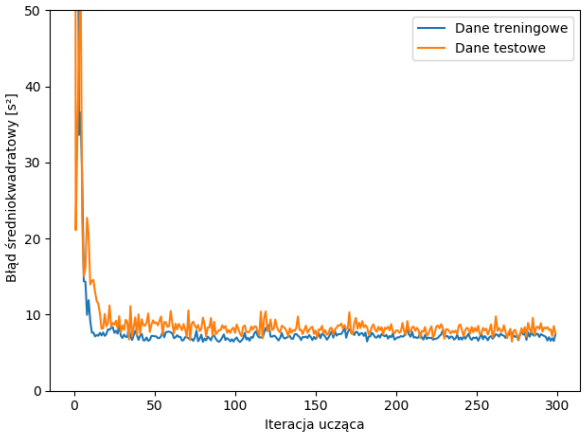
– Analizując tabelę 5.5 wygląda na to, że tak jak poprzednio - funkcja regularyzacji *L1* z parametrem λ równym **1** daje najlepsze rezultaty. Jeznakże nie wiele gorszym zestawem parametrów, ze zbliżonym *MSE* danych testowych odznacza się ta sama funkcja z parametrem λ równym **0,3**.

– Patrząc na wykresy krzywych uczących dla najlepszych przypadków widać, że niezależnie czy użyta została funkcja regularyzacji *L1* czy *L2*, przy parametrze λ równym **1** sieć przyucza się minimalnie lepiej do danych testowych. Pamiętając co było źródłem dotychczasowego problemu można wywnioskować, że takie zachowanie może być spowodowane przez szumy zbioru danych, a sposobem ich wyeliminowania było by zebranie danych z większej ilości symulacji przeprowadzonych z wyższym parametrem *Repeat number*.

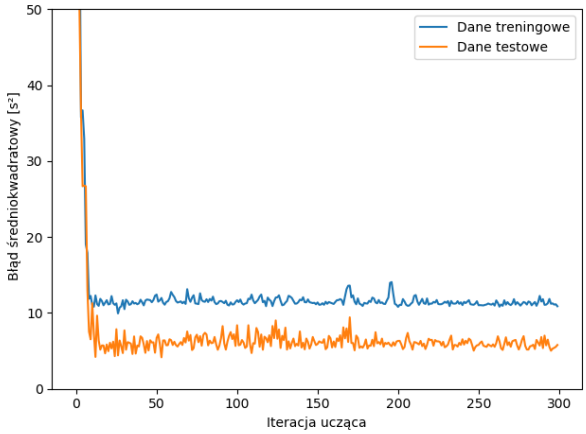
Podsumowując, w wyniku przeprowadzonych badań została znaleziony najlepsze kombinacje parametrów regularyzacji dla obecnych danych - funkcja *L1* z parametrem λ równym **0,3** lub **1**. Jednakże istnieje podejrzenie, że przez niską ilość powtórzeń symulacji dla danego zestawu parametrów wejściowych, w zbiorze danych



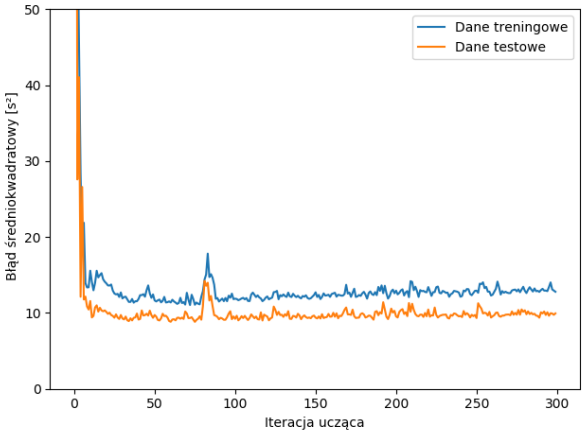
(a) parametr λ - 0,1



(b) parametr λ - 0,3



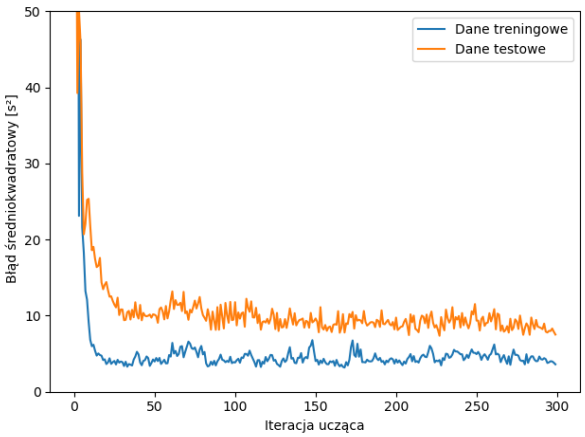
(c) parametr λ - 1



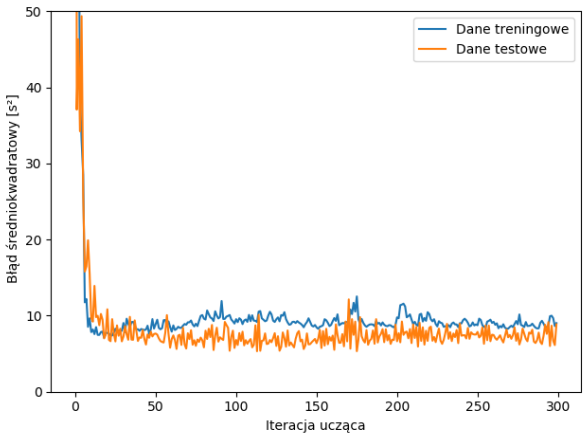
(d) parametr λ - 3

Rys. 5.17. Krzywe uczące dla najlepszych przypadków używając regularyzacji L1 w drugim modelu po odrzuceniu częsci danych.

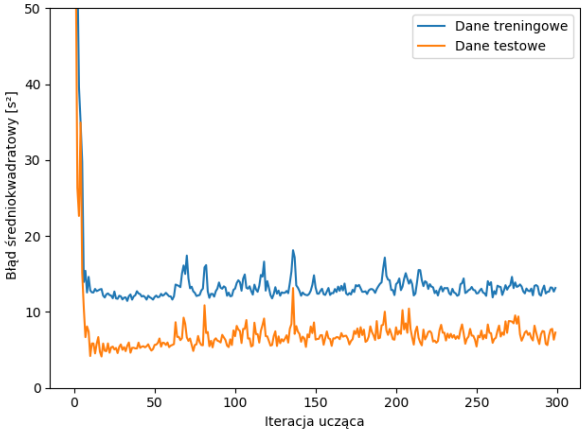
wyjściowych istnieją szumy, których eliminacja podniosłaby efektywność sieci przy parametrze λ równym **1**.



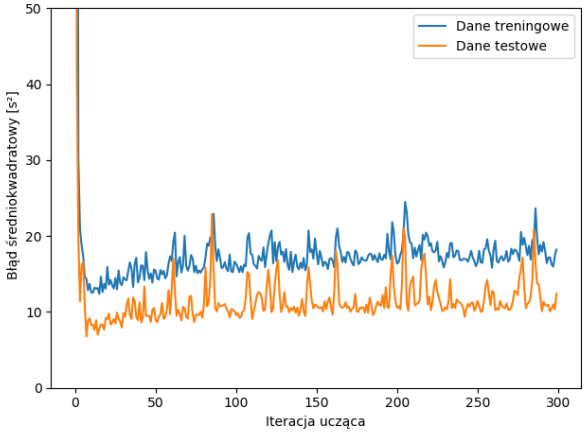
(a) parametr λ - 0,1



(b) parametr λ - 0,3

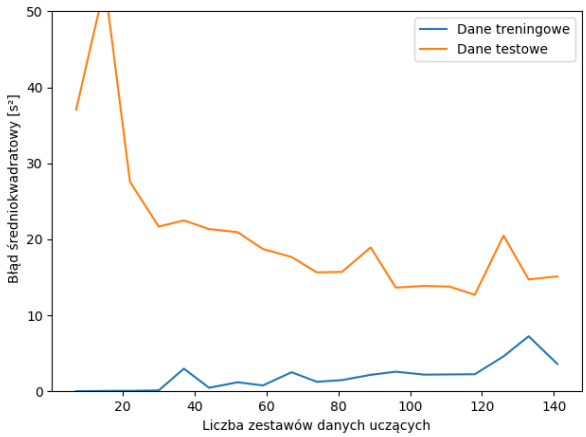


(c) parametr λ - 1

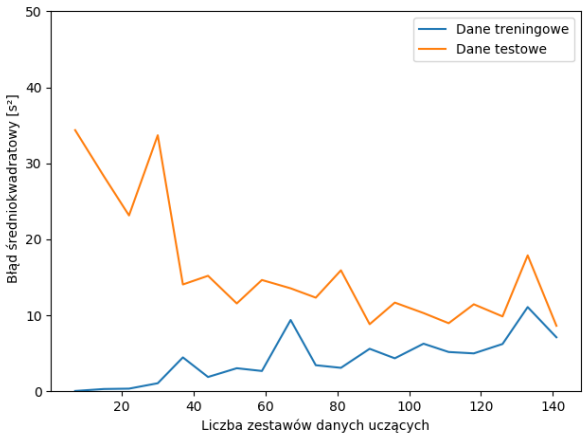


(d) parametr λ - 3

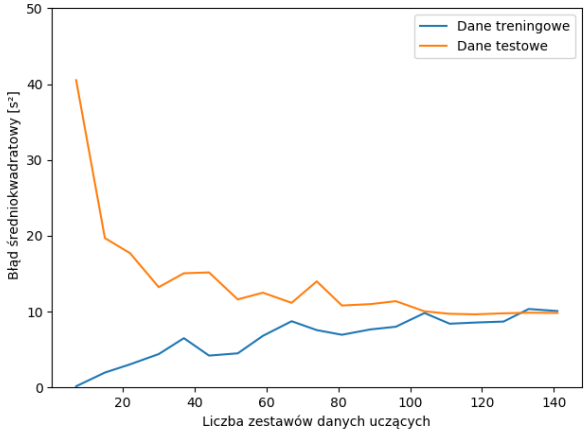
Rys. 5.18. Krzywe uczące dla najlepszych przypadków używając regularyzacji L2 w drugim modelu po odrzuceniu częsci danych.



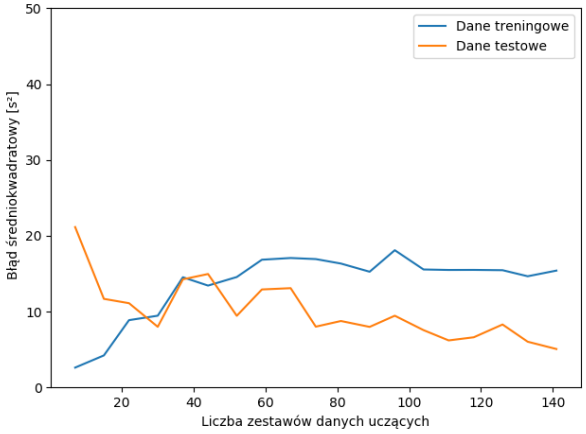
(a) parametr λ - 0,1



(b) parametr λ - 0,3

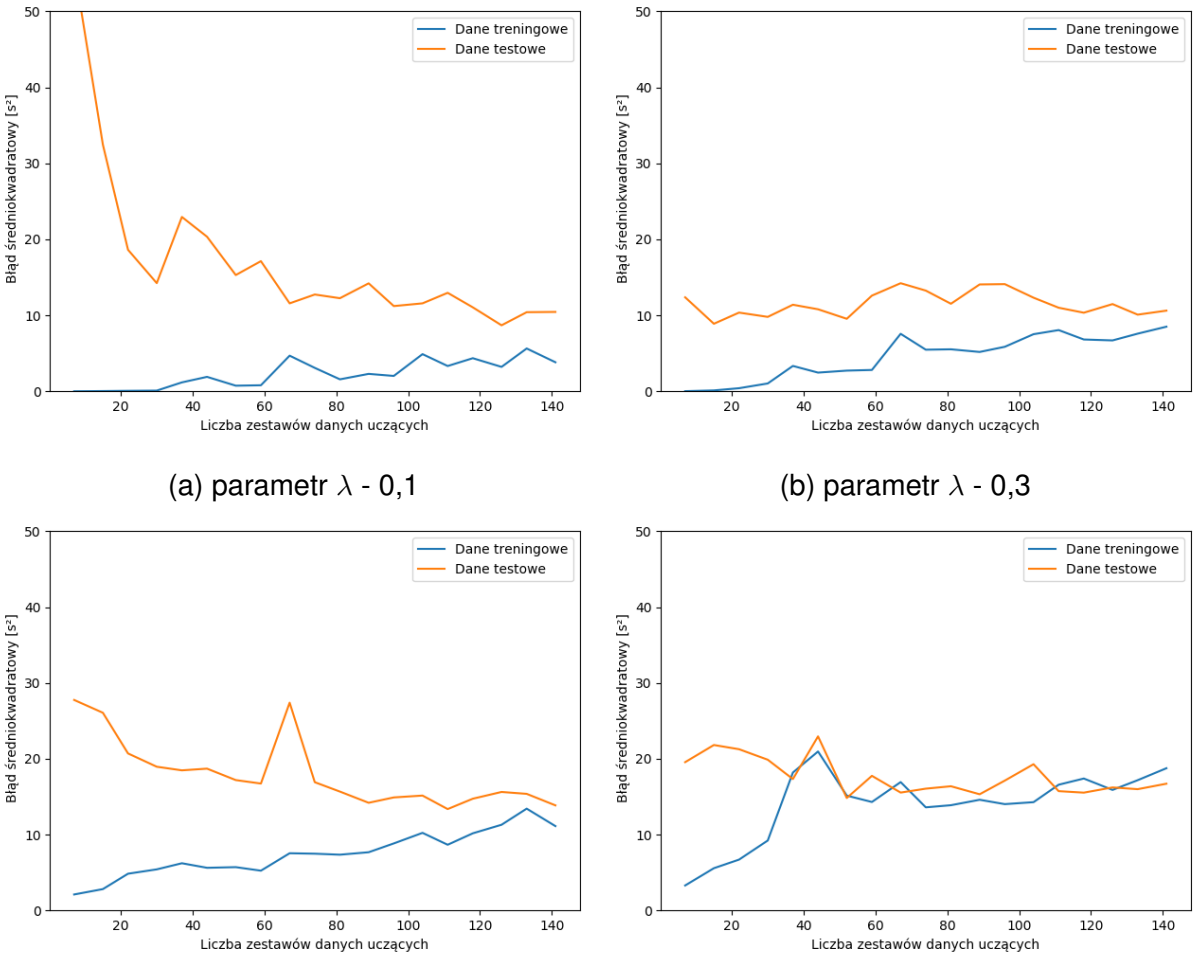


(c) parametr λ - 1



(d) parametr λ - 3

Rys. 5.19. Krzywe uczące dla zmiennej ilości danych używając regularyzacji L1 w drugim modelu po odrzuceniu częsci danych.



(a) parametr λ - 0,1

(b) parametr λ - 0,3

(c) parametr λ - 1

(d) parametr λ - 3

Rys. 5.20. Krzywe uczące dla zmiennej ilości danych używając regularyzacji L2 w drugim modelu po odrzuceniu częsci danych.

5.10. Zebranie większej ilości danych

Chcąc zbadać hipotezę, że przy parametrze regularyzacji λ równym 1 model będzie się uczył lepiej na wiekszej ilości danych, postanowiłem powtórzyć proces generacji danych symulacyjnych. Zrobilem to podobnie jak w sekcji 5.1 z tym, że parametr *Repeat number* w wygenerowanych plikach ustawiłem na 50. Dzięki temu każda dana wyjściowa powinna być stabilniejsza ze względu na obliczanie średniej na 5 razy wiekszej liczbie przypadków niż dotychczas.

Po zebraniu danych oraz uśrednieniu ich w taki sam sposób jak w sekcji 5.7 powiększyłem zbiór uczący o kolejne 200 przypadków. Postanowiłem przeprowadzić badanie, jak wpłynie na model nauka na:

– **starym zestawie danych** - danych zebranych w sekcji 5.1,

– **nowym zestawie danych** - danych zebranych w tej sekcji,

– **wszystkich zestawach danych** - połączonym starym i nowym zestawie danych.

W eksperymencie nauczyłem sieć osobno na każdej kombinacji wielkości zbioru danych uczących oraz parametrów uczenia λ - **0,3** i **1**.

Wyniki eksperymentów są następujące:

– porównanie średnich wartości *MSE* danych w tabeli 5.6,

– krzywe uczące dla najlepszych przypadków na rysunku 5.21,

– krzywe uczące dla zmiennej ilości danych na rysunku 5.22.

Po analizie danych doszedłem do następujących wniosków:

– Wbrew oczekiwaniom - na starym zbiorze danych sieć neuronowa dawała lepsze rezultaty, w najlepszym przypadku - mniejsze o około 5 sekund kwadratowych od wyników na pozostałych zbiorach.

– Co jest niespodziewane - na nowych, stabilniejszych danych sieć miała największe problemy z nauką osiągając nawet dwukrotnie wyższe średnie lub odchylenia *MSE* danych testowych w stosunku do najlepszych przypadków.

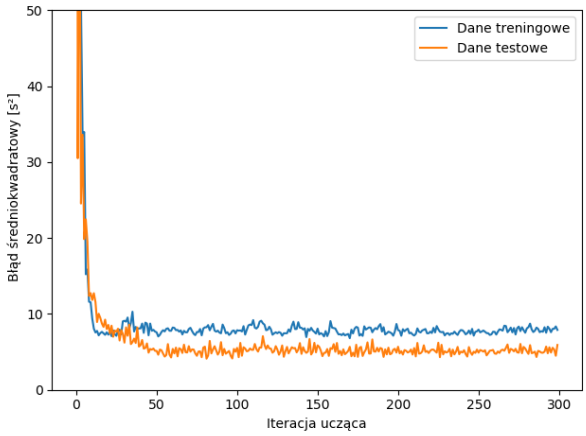
– Na wykresach krzywych uczących dla najlepszych przypadków widać, że nie ważne z jakiego zbioru danych sieć korzystała, to i tak czasami pojawiają się szumy powodujące lepsze przyuczenie sieci do danych testowych.

Tabela 5.6. Porównanie średnich wartości *MSE* danych po zebraniu większej ich ilości.

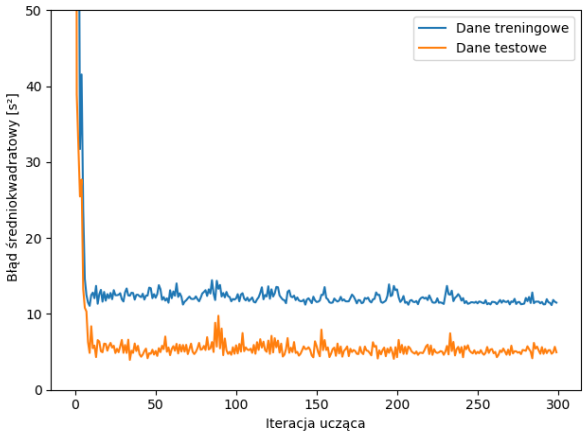
Zestaw danych	Parametr regularyzacji L1	MSE danych treningowych	MSE danych testowych
stary	0,3	7,02 ($\pm$0,56)	11,58 ($\pm$4,83)
stary	1	10,51 ($\pm$0,85)	9,50 ($\pm$2,87)
nowy	0,3	10,76 ($\pm$0,88)	19,48 ($\pm$3,95)
nowy	1	14,53 ($\pm$1,10)	14,90 ($\pm$5,51)
stary + nowy	0,3	14,72 ($\pm$0,65)	14,23 ($\pm$2,16)
stary + nowy	1	14,80 ($\pm$0,98)	15,94 ($\pm$3,00)

– Analiza wykresów krzywych uczących dla zmiennej ilości danych obrazują jak ilość danych wpłynęła na *MSE* danych - dopiero większa ilość danych sprawiła, że krzywe danych treningowych i testowych zetknęły się ze sobą - oznacza to większą stabilność przewidywania danych co jest cechą bardzo porządaną.

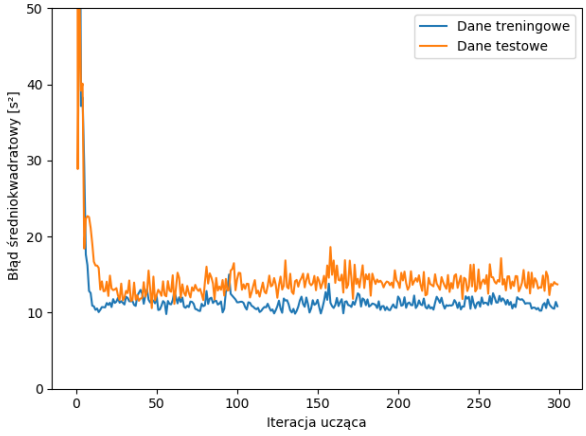
Zbiór nowych danych tylko pogorszył *MSE* modelu; podobny efekt dał zbiór połączonych danych. Obaliło to tezę, jako że większy zestaw danych miałby zmniejszyć ich *MSE*. Jednakże zestaw wszystkich danych wprowadził do modelu większą stabilność, co przełożyło się na bardziej do siebie zbliżony średni *MSE* danych testowych oraz danych treningowych. Szczególnie jest to prawdą dla parametru λ równego **0,3** - oznacza to, że do osiągnięcia największej stabilności modelu o akceptowalnym poziomie *MSE* właśnie on będzie dalej używany. Jeśli chodzi o zbiór uczący - do dalszego użytku zostanie wybrany pełny zbiór uczący, składający się z nowych oraz starych danych.



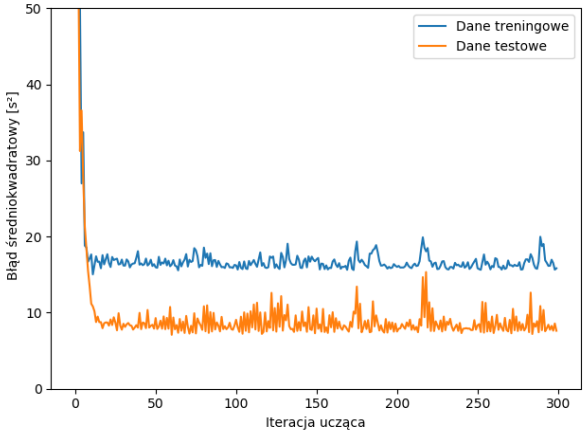
(a) stary zestaw danych, parametr λ - 0,3



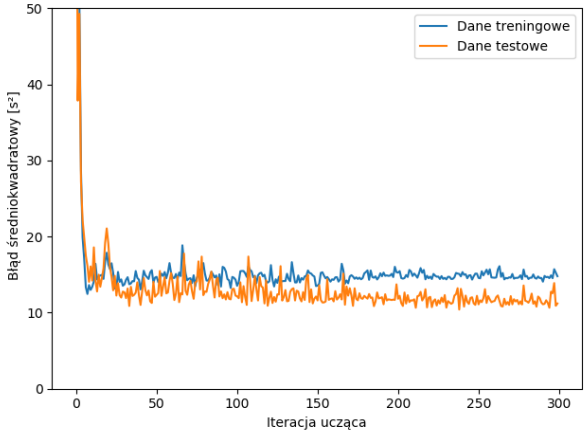
(b) stary zestaw danych, parametr λ - 1



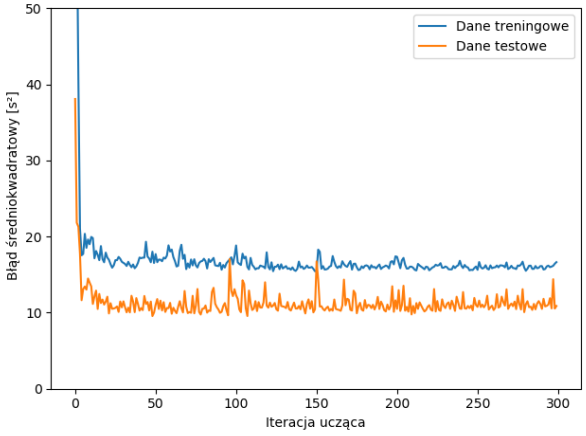
(c) nowy zestaw danych, parametr λ - 0,3



(d) nowy zestaw danych, parametr λ - 1

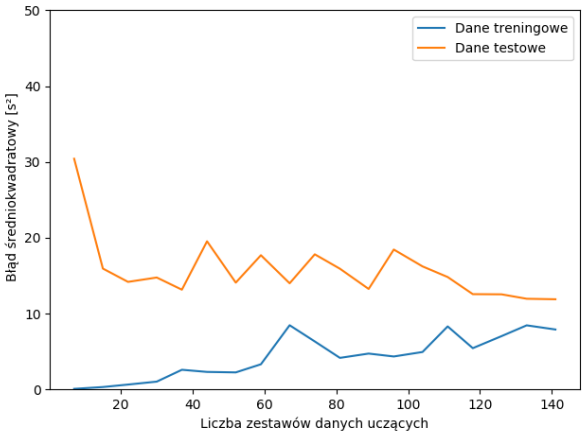


(e) oba zestawy danych, parametr λ - 0,3

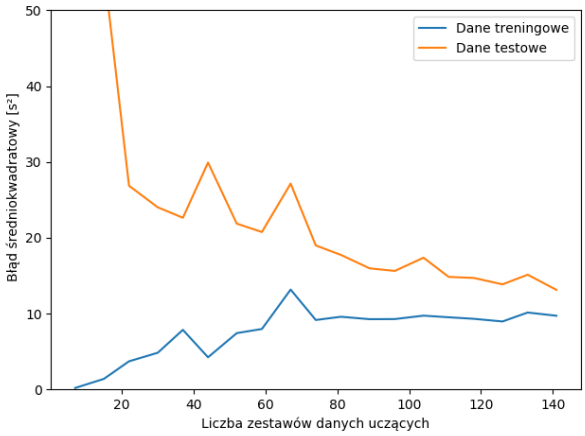


(f) oba zestawy danych, parametr λ - 1

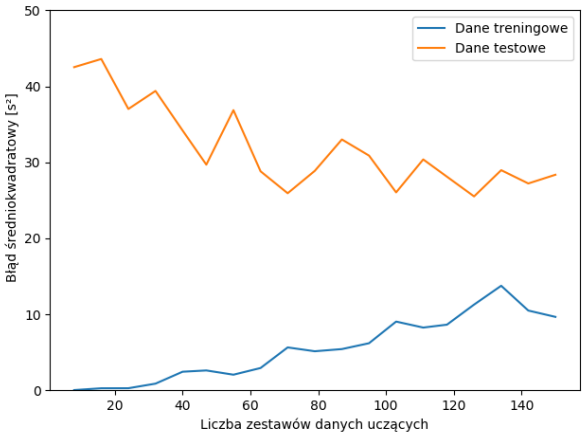
Rys. 5.21. Krzywe uczące dla najlepszych przypadków w drugim modelu po zebraniu większej ilości danych.



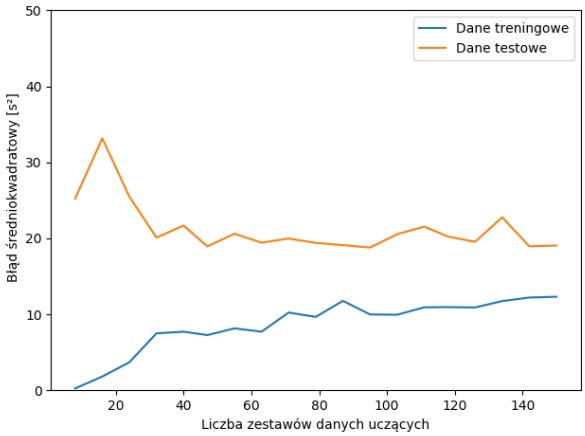
(a) stary zestaw danych, parametr λ - 0,3



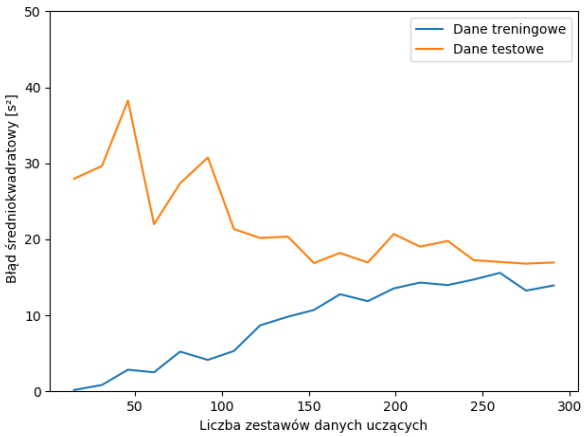
(b) stary zestaw danych, parametr λ - 1



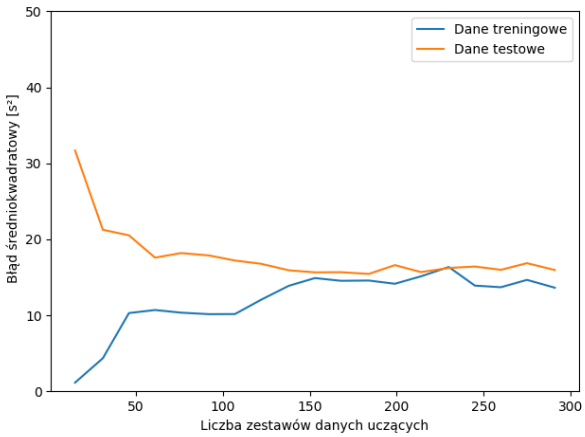
(c) nowy zestaw danych, parametr λ - 0,3



(d) nowy zestaw danych, parametr λ - 1



(e) oba zestawy danych, parametr λ - 0,3



(f) oba zestawy danych, parametr λ - 1

Rys. 5.22. Krzywe uczące dla zmiennej ilości danych w drugim modelu po zebraniu większej ich ilości.

5.11. Podsumowanie

Podsumowując, po procesie dostrajania parametrów sieci neuronowej udało się osiągnąć model, mający błąd średniokwadratowy około 15 s^2. Parametry ostatecznego modelu sieci neuronowej były następujące:

- 1 warstwa ukryta z 20 neuronami i funkcją aktywacji ReLU,
- warstwa wyjściowa składająca się z jednego neuronu z funkcją aktywacji ReLU,
- użyta **regularyacja L1** ze współczynnikiem λ równym **0,3**,
- **learning rate** - 0.1,
- **metryka do optymalizacji** - MSE,
- **liczba epok uczacych** - 300,
- podział na **dane treningowe** i **dane testowe** - 80% : 20%.

Operacje wykonane na danych uczących przed rozpoczęciem procesu uczącego były następujące:

1. uśrednienie wszystkich powtórzeń pojedynczej symulacji,
2. odrzucenie wszystkich zestawów danych, których dana wyjściowa (czas ewakuacji) była większa lub równa 40,
3. normalizacja danych opisana w sekcji 5.4.

Na koniec stworzyłem 20 modeli z powyższymi parametrami, które zostały osobno uczone. Po zakończeniu procesu nauki do dalszych badań wybrałem ten, który miał najmniejszą różnicę pomiędzy błędem na danych treningowych i na danych testowych.

6. Optymalizacja modelu

W tym rozdziale opisany zostanie proces optymalizacji wyjść modelu na podstawie danych wejściowych.

6.1. Proces optymalizacji modelu

Algorytmem optymalizacji którego użyłem był algorytm *Gradient descent* opisany w sekcji 2.3. Sam proces był następujący:

1. Weź jeden losowo wybrany zestaw danych ze zbioru.
2. Jeśli dana wyjściowa tego zestawu jest większa od 40, wróć do punktu 1.
3. Znormalizuj dane wejściowe.
4. Wykonaj 100 iteracji algorytmu *Gradient descent* w celu optymalizacji wyjścia modelu na podstawie jego wejść, przy czym wejścia nie mogą wykroczyć poza ustalony przez twórców symulatora *PSP* zakres (na przykład aby nie dostać negatywnej liczby pieszych w symulacji).
5. Wykonaj de-normalizacje danych wejściowych i zapisz je do pliku *.xml* z parametrem powtórzeń *Repeat number* ustawionym na 10.
6. Wykonaj symulacje na podstawie zapisanych danych.
7. Oblicz średnią daną wyjściową ze wszystkich symulacji.

Powyższy proces wykonałem 10 razy a jego wyniki są widoczne w tabeli 6.1. W pierwszej i drugiej kolumnie widać rzeczywiste i przewidziane przez model czasy ewakuacji - różnią się one nieznacznie, lecz w granicach błędu uczenia widocznego na wykresach 5.21 i 5.22 (dla obu zestawów danych i parametru λ równego 0,3). W trzeciej i czwartej kolumnie widoczne są czasy ewakuacji po procesie optymalizacji. Od

Tabela 6.1. Porównanie rzeczywistych i przewidzianych czasów ewakuacji przed i po optymalizacji.

Rzeczywisty czas ewakuacji przed optymalizacją [s]	Przewidywany czas ewakuacji przed optymalizacją [s]	Rzeczywisty czas ewakuacji po optymalizacji [s]	Przewidywany czas ewakuacji po optymalizacji [s]
24.22	26.37	15.925	19.58
23.27	22.11	18.125	19.57
19.70	23.69	18.125	19.57
26.12	24.43	17.475	19.57
27.06	27.04	17.725	19.57
22.39	27.72	17.45	19.57
24.38	26.45	18.125	19.57
23.98	27.48	17.875	19.57
32.77	27.84	17.1625	19.59
25.23	24.47	16.825	19.60

razu widać, że wartości przewidziane przez model są prawie identyczne - oznacza to, że algorytm *Gradient descent* prawidłowo znajduje minimum modelu, które nie zależy od tego z jakim zestawem danych rozpoczniemy optymalizację. Porównując kolumnę 3 i 4 można zauważyć, że dane rzeczywiste też mieszczą się w zakresie błędu uczącego; co więcej - w większości są one mniejsze od przewidywanego czasu ewakuacji o 1,5 - 2 sekund.

6.2. Podsumowanie

Podsumowując, udało się przeprowadzić optymalizację wyjść modelu za pomocą algorytmu *Gradient descent*. Przy każdej próbie czas ewakuacji był optymalizowany do prawie identycznych wartości co oznacza, że proces optymalizacji nie zależy od wartości początkowych. Porównując wartości wartości rzeczywiste i przewidywane przez model (przed jak i po optymalizacji), zdecydowana ich większość była w granicach błędu uczenia. Jak widać najniższy przewidywany przez sieć czas ewakuacji

jest to 19,57 sekund - co po weryfikacji w symulatorze *PSP* daje czasy w okolicach 17 - 18 sekund.

7. Podsumowanie projektu

W tym rozdziale zostały podsumowane prace związane ze zbieraniem danych, budowaniem modelu i dostrajaniem jego parametrów. Znajduje się tu także podsumowanie procesu optymalizacji.

7.1. Podsumowanie procesu zbierania danych

Jako że w dziedzinie uczenia maszynowego operacje na danych są tak samo ważne jak budowa odpowiedniego modelu to i tym razem dane, które zebrałem z symulatora *PSP* wymagały bliższego poznania i ewentualnej obróbki. Zbiór danych został poddany następującym operacjom:

1. uśrednienie wszystkich powtórzeń pojedynczej symulacji,
2. odrzucenie wszystkich zestawów danych, których dana wyjściowa była większa lub równa 40,
3. normalizacja danych.

Zbiór za pomocą którego uczyłem model miał 400 zestawów danych; dało to większą stabilność przewidywanych wyników w stosunku do zbioru składającego się z 200 zestawów danych. Jednakże mimo tego, nie wyeliminowało to zupełnie szumu związanego z czynnikami losowymi w symulatorze *PSP*.

7.2. Podsumowanie budowy modelu

Budowa modelu była najdłuższą częścią projektu - wymagała ona dużej ilości testów z różnymi kombinacjami parametrów (gdzie niektóre z nich wykonywały się po kilka godzin). Ostatecznie udało się osiągnąć model, którego błąd średniokwadratowy był oscylował w granicach 15 s^2. Wybranym modelem została sieć neuronowa z następującymi parametrami:

– 1 warstwa ukryta z 20 neuronami i funkcją aktywacji ReLU,

– warstwa wyjściowa składająca się z jednego neuronu z funkcją aktywacji ReLU,

– użyta **regularyacja L1** ze współczynnikiem λ równym **0,3**,

– **learning rate** - 0.1,

– **metryka do optymalizacji** - MSE,

– **liczba epok uczacych** - 300,

– podział na **dane treningowe** i **dane testowe** - 80% : 20%.

Budowa takiego modelu pozwoliła uniezależnić się od symulatora *PSP* podczas procesu symulacji - co było dużym zyskiem, gdyż symulacje trzeba było w nim włączać ręcznie.

7.3. Optymalizacja wyjść modelu

Ostatnią częścią projektu była optymalizacja danyc wyjściowych modelu na podstawie jego wejść. Użyłem do tego algorytmu *Gradient descent* z 10 różnymi zestawami danych do optymalizacji, co miało dać szerszy pogląd na to czy proces optymalizacji będzie generował duże szumy. Jak się okazało - model przewidywał daną wyjściową na podstawie danych wejściowych z wypracowaną wcześniej dokładnością co było dobrym znakiem. Co więcej algorytm potrafił zoptymalizować wejścia modelu tak, żeby za każdym zwracał bardzo zbliżone wyniki. Daje to także ostateczne rozwiązanie postawionego problemu optymalnego wyboru ścieżki lub wyjścia ewakuacyjnego - dzięki odpowiedniemu dostrojeniu parametrów pieszych - szybkości, sposobu wyboru ścieżki, dozwolonej gęstości na metr kwadratowy i tym podobnch - udało się zminimalizować czas potrzebny na ich ewakuację. Czas jaki się udało osiągnąć było to przewidywane przez sieć - 19,57 sekund, a przetestowane w symulatorze - 17 - 18 sekund.

Bibliografia

[1] David Fumo. *Types of Machine Learning Algorithms You Should Know*. *https://towardsdatascience.com/types-of-machine-learning-algorithms-you-should-know-953a08248861*.

[2] Sagar Sharma. *What the Hell is Perceptron?* *https://towardsdatascience.com/what-the-hell-is-perceptron-626217814f53*.

[3] Brian Dolhansky. *Artificial Neural Networks: Linear Regression (Part 1)*. *http://www.briandolhansky.com/blog/artificial-neural-networks-linear-regression-part-1*.

[4] Avinash Sharma. *Understanding Activation Functions in Neural Networks*. *https://medium.com/the-theory-of-everything/understanding-activation-functions-in-neural-networks-9491262884e0*.

[5] Saimadhu Polamuri. *Difference between softmax function and sigmoid function*. *http://dataaspirant.com/2017/03/07/difference-between-softmax-function-and-sigmoid-function*.

[6] Lachlan Miller. *Machine Learning week 1: Cost Function, Gradient Descent and Univariate Linear Regression*. *https://medium.com/@lachlanmiller_52885/machine-learning-week-1-cost-function-gradient-descent-and-univariate-linear-regression-8f5fe69815fd*.

[7] Hafidz Zulkifli. *Understanding Learning Rates and How It Improves Performance in Deep Learning*. *https://towardsdatascience.com/understanding-learning-rates-and-how-it-improves-performance-in-deep-learning-d0d4059c1c10*.

[8] *Partial derivative*. *https://en.wikipedia.org/wiki/Partial_derivative*.