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.

Serdecznie dziękuję . . . tu ciąg dalszych podziękowań np. dla promotora, żony, sąsiada itp.

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. Cel projektu

W wielu miastach miastach, gdzie istnieją duże skupiska ludzi prędzej czy później pojawia się problem alokacji przestrzeni na zabudowę. W tym celu nowoczesne budynki uzytku publicznego są projektowane pod kątem wykorzystania jak największej przestrzeni dla celów użytkowych, ale i z drugiej strony - dla zapewnienia jak największego bezpieczeństwa ludziom w nich przebywających. Jednym z czynników określających wchodzącym w skład szeroko pojętego bezpieczeństwa, jest zapewnienie ludziom ścieżek ewakuacyjnych w razie pożaru lub innych wypadków. Z tego powodu celu postawiłem 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.

1.2. 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 2.

1.3. 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.

2. 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.

2.1. 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 2.1.

2.2. Sposób użycia

Do przeprowadzenia symulacji potrzebne są następujące rzeczy:

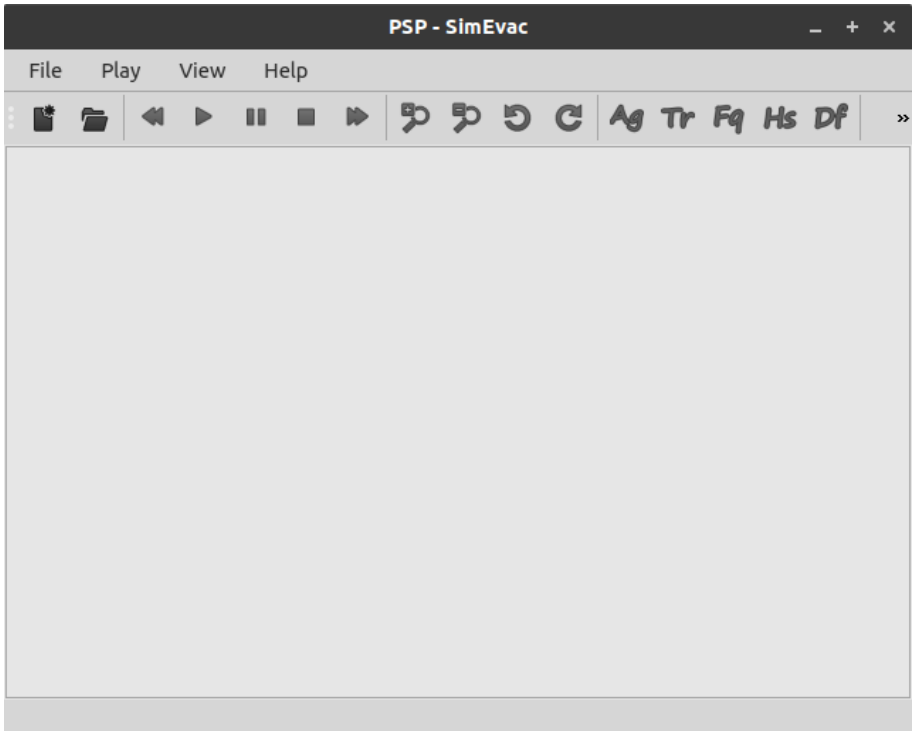
– plik *.xml* zawierający parametry wejściowe symulacji,

– 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.

2.3. Dane wejściowe i wyjściowe

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



Rys. 2.1. Główne okno symulatora.

– 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*).

3. 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.

3.1. 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 2.3. 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. Daną wyjściową, którą wybrałem był czas ewakuacji ostatniego agenta. Każdy zestaw danych zawierał więc **15** parametrów wejściowych oraz **1** parametr

wyjściowy. Z tak przygotowanymi danymi miałem możliwość wybrania pierwszego modelu.

3.2. 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*

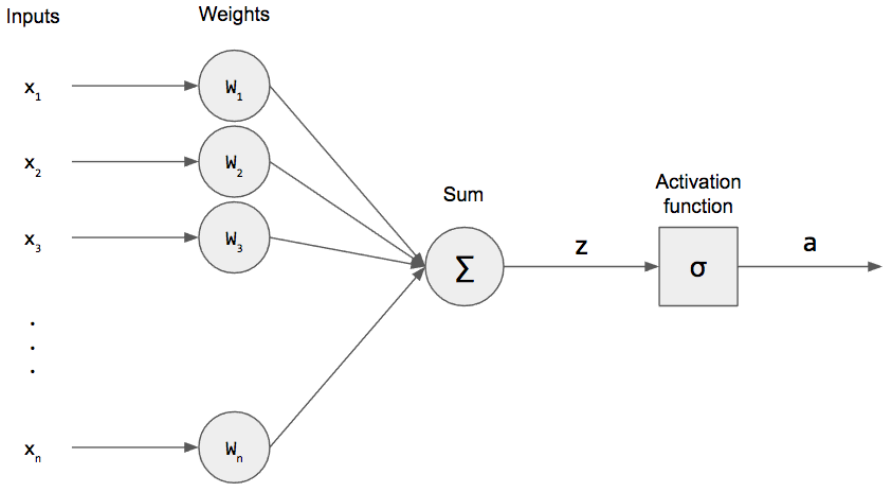
to zachodzi równanie:

$$Y = \sigma(NX) \tag{3.1}$$

Budowa Perceptronu z jednym neuronem wyjściowym jest widoczna na rysunku 3.1[1].

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.

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



Rys. 3.1. Budowa Perceptronu z jednym neuronem wyjściowym.

3.3. 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 efektowności na tanych 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:

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

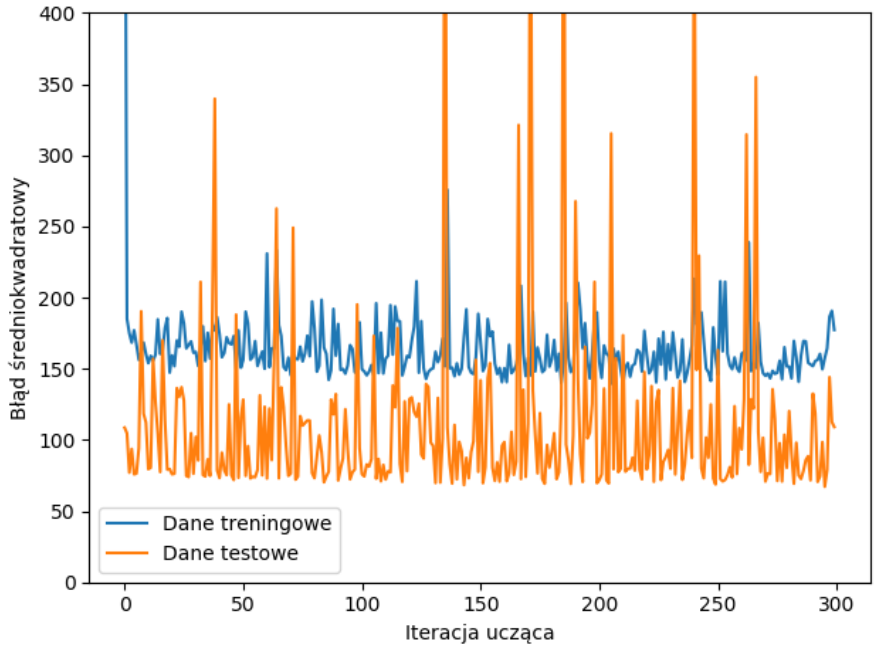
– **średnie MSE danych testowych** - 166,18 (±50,77).

Zależność *MSE* danych od iteracji uczacej najlepszego przypadku (biorąc pod uwagę *MSE* danych testowych) jest widoczna na wykresie 3.2. 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 bardzo niestabilny, co oznacza niestabilność danych lub modelu.

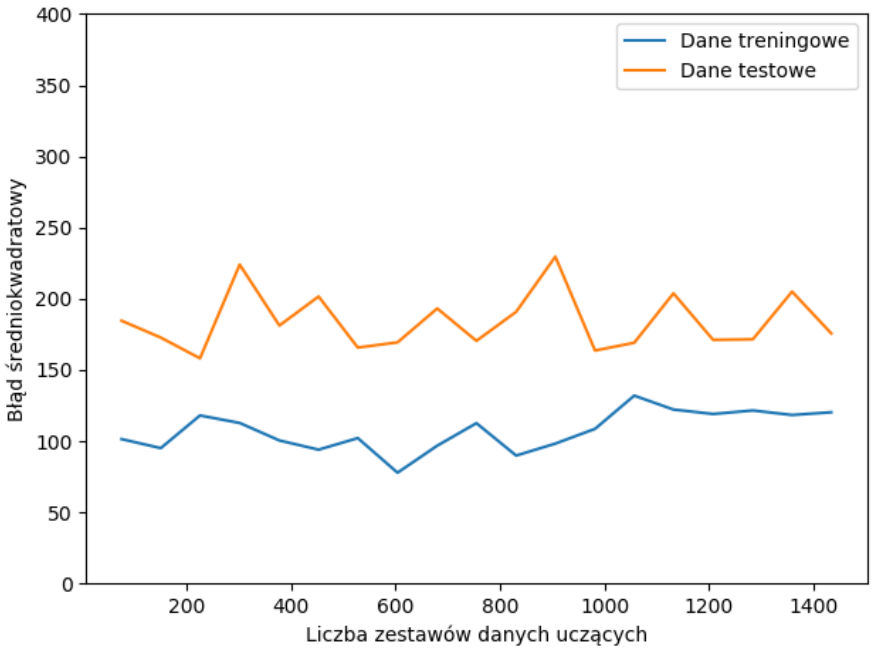
– *MSE* danych testowych jest przez większośc iteracji mniejsze niż treningowych co oznacza, że dla danych których sieć nie widziała, przewiduje ona wartości bliższe prawdzie niż na danych na których się uczyła. Jest to niecodzienne zachowanie co również dowodzi niestabilności modelu lub danych.



Rys. 3.2. Zależność *MSE* danych od iteracji uczącej 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. Wynik tego badania jest widoczny na wykresie 3.3. 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. 3.3. Zależność *MSE* danych od ich ilości 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.

3.4. 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. Wyniki są widoczne w tabeli 3.1. Widać w niej, że niektóre parametry mają wartości 2 rzędy wielkości większe niż inne. Takie różnice pomiędzy różnymi parametrami

Tabela 3.1. Średnie 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

wejściowymi powodują faworyzowanie przez sieć jednych parametrów ponad drugie. Postanowiłem więc zastosować normalizację danych według wzoru:

$$X_{new} = \frac{X_{old} - \mu}{\sigma} \tag{3.2}$$

Gdzie:

– μ - wektor średniej parametrów,

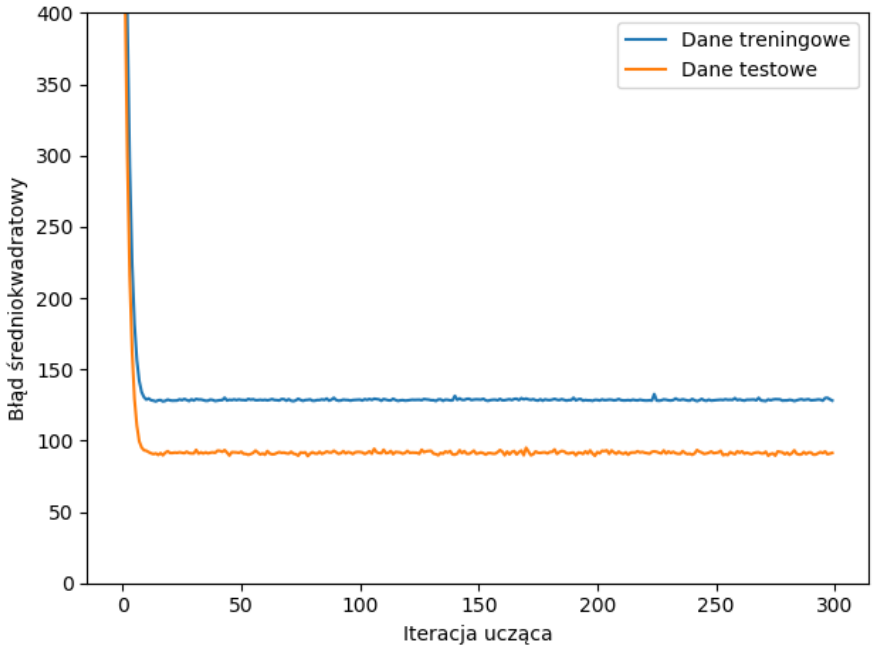
– σ - wektor odchylenia parametrów.

Posiadając tak przygotowane dane wykonałem - tak jak w punkcie 3.3 - 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:

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

– **średnia MSE danych testowych** - 130,70 ($\pm$23,47).

Proces uczenia najlepszego przypadku (biorąc pod uwagę *MSE* danych testowych) jest widoczny na wykresie 3.4. Dodatkowo na wykresie 3.5 widać zależność *MSE* danych od liczby danych treningowych.

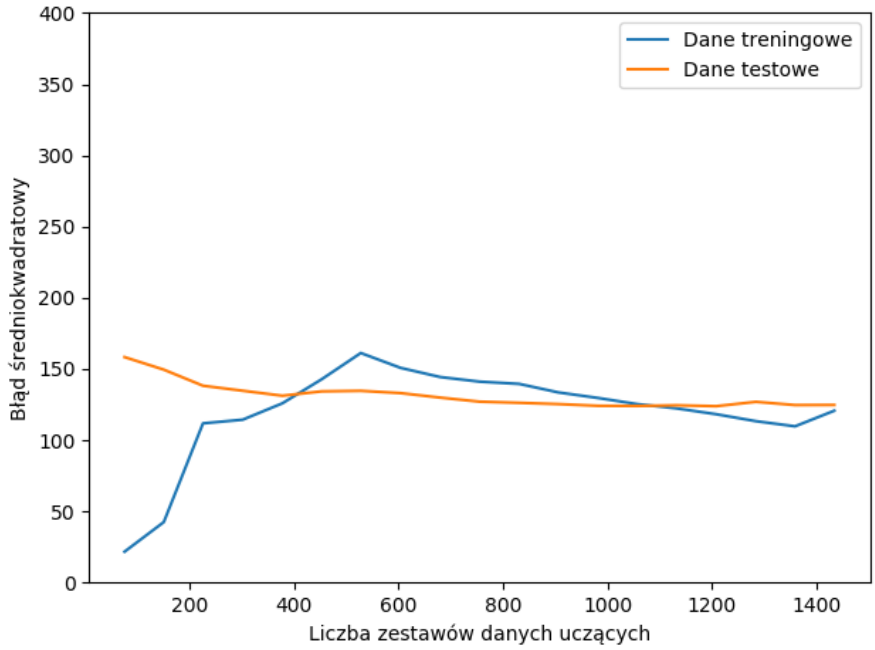


Rys. 3.4. Zależność *MSE* danych od iteracji uczącej dla najlepszego przypadku w pierwszym modelu ze znormalizowanymi danymi.

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 3.3 - *MSE* danych testowych często jest mniejsze niż danych treningowych.

3.5. 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



Rys. 3.5. Zależność *MSE* danych od ich ilości w pierwszym modelu ze znormalizowanymi danymi.

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:

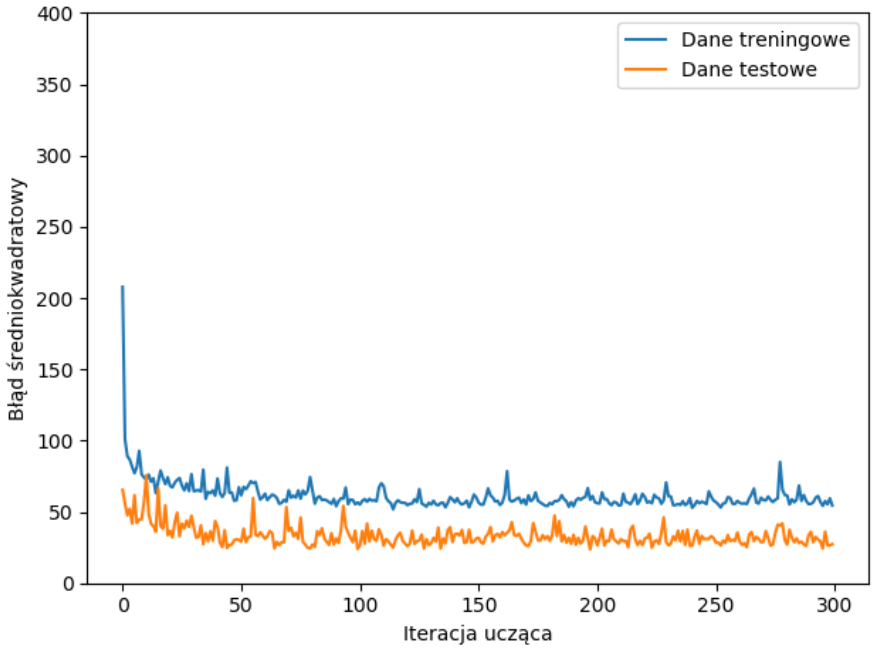
- 1 warstwa ukryta z 30 neuronami i funkcją aktywacji ReLU,
- warstwa wyjściowa składająca się z jednego neuronu bez funkcji aktywacji,
- **learning rate** - 0.1,
- **metryka do optymalizacji** - MSE,
- **liczba epok uczacych** - 300,
- podział na **dane treningowe** i **dane testowe** - 80% : 20%.

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:

- **średnia MSE danych treningowych** - 54,07 ($\pm$3,11),

– **średnia MSE danych testowych** - 48,82 ($\pm$14,69).

Proces uczący najlepszego przypadku (biorąc pod uwagę *MSE* danych testowych) jest widoczny na wykresie 3.6. Dane zebrane z cykli ze zmnieniającą się ilością danych treningowych umieszczone są na wykresie 3.7.



Rys. 3.6. Zależność *MSE* danych od iteracji uczącej dla najlepszego przypadku w drugim modelu.

Porównując te dane z tymi z sekcji 3.4 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.

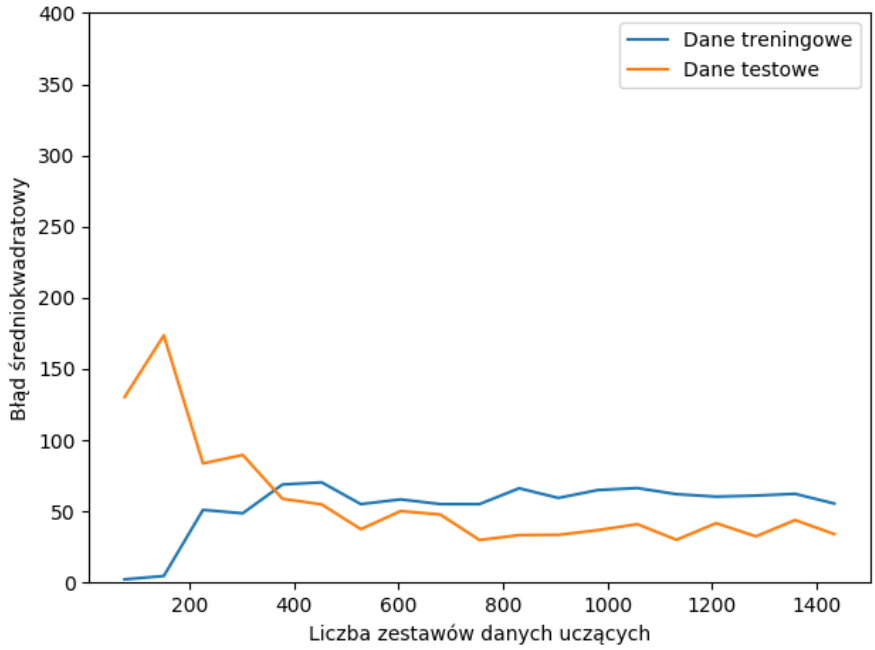
3.6. Zwiększenie stopnia skomplikowania modelu

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

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

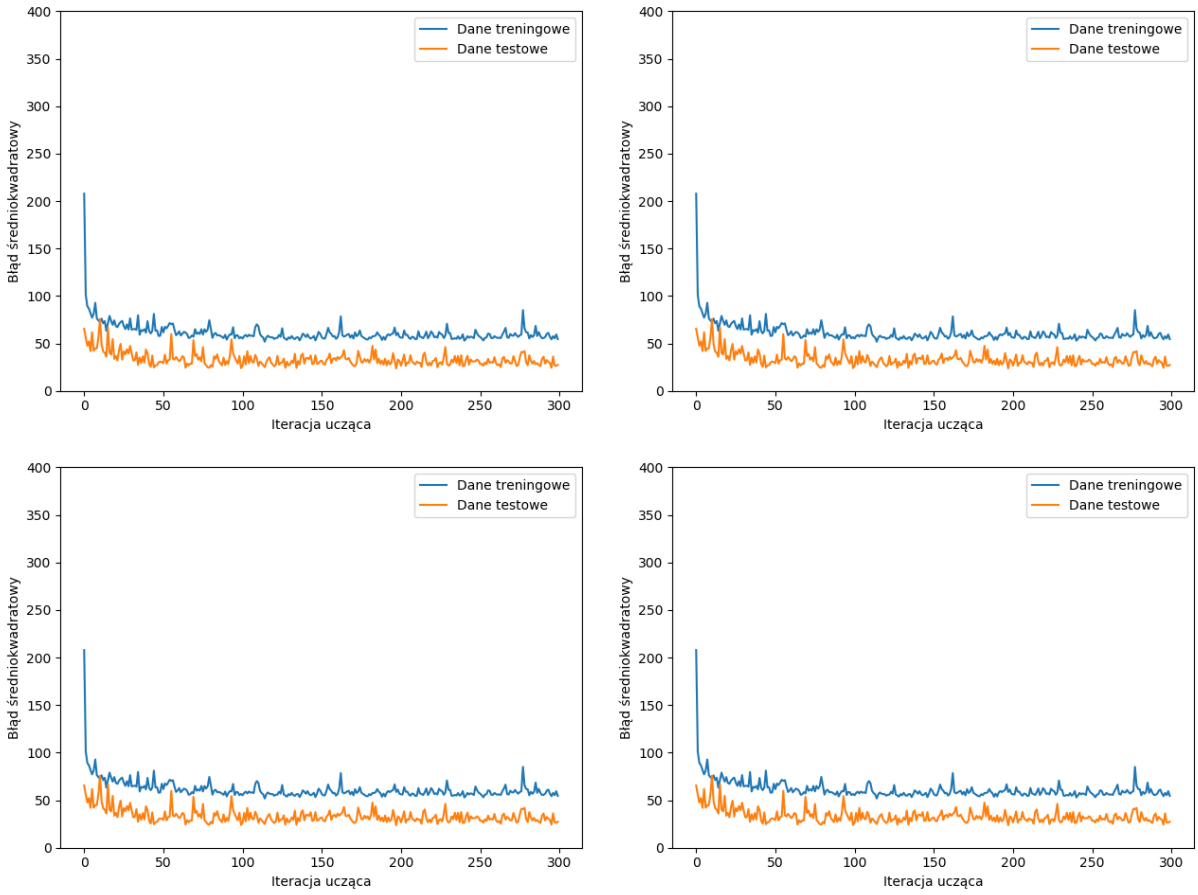
– liczba warstw ukrytych - **2**, liczba neuronów w każdej warstwie - **20**;

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

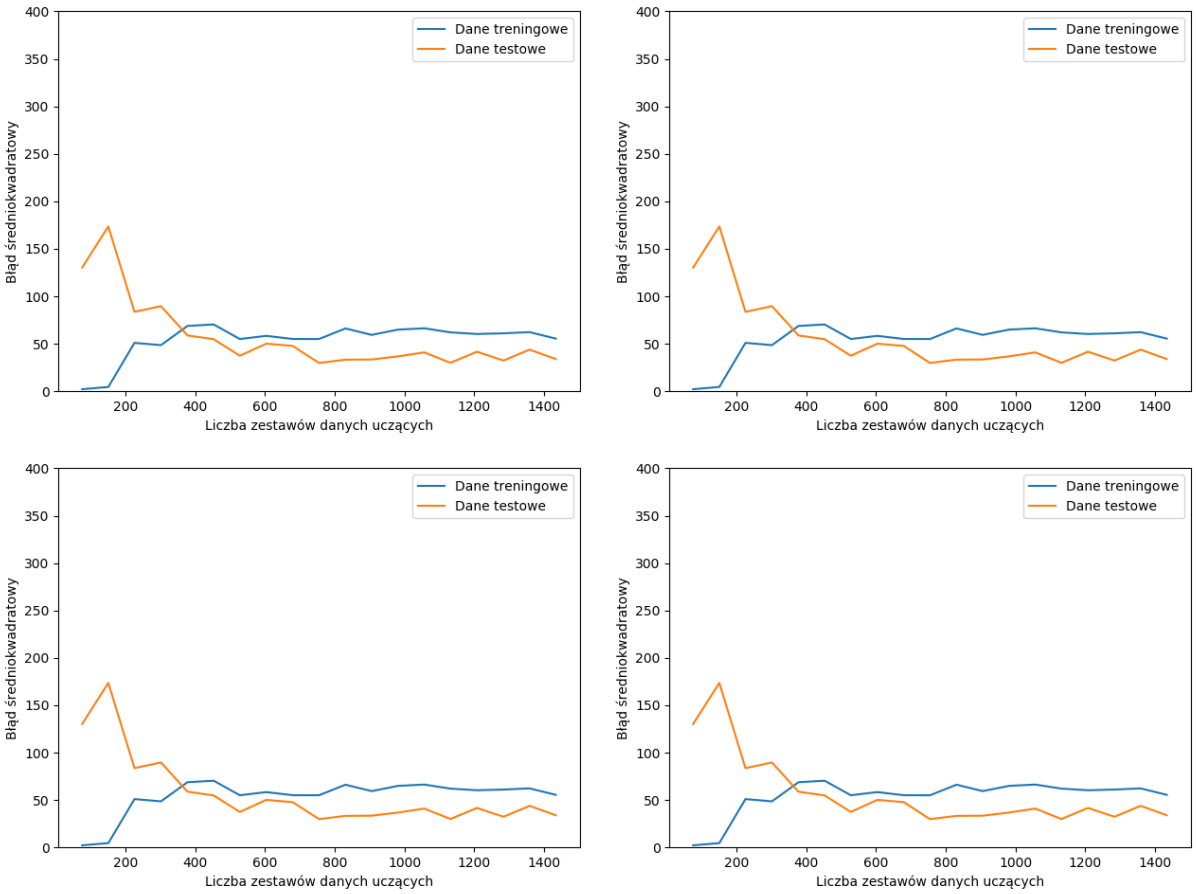


Rys. 3.7. Zależność *MSE* danych od ich ilości w drugim modelu.

Zestawienie wynikow z sekcji 3.5 oraz powyższych kombinacji parametrów jest widoczne na wykresach 3.8 i 3.9.



Rys. 3.8. Zależność *MSE* danych od iteracji uczącej dla najlepszych przypadków przy użyciu różnych kombinacji parametrów w drugim modelu.



Rys. 3.9. Zależność *MSE* danych od iteracji uczącej dla różnych kombinacji parametrów w drugim modelu.

Bibliografia

[1] L. Lamport. *LaTeX system przygotowywania dokumentów*. Kraków: Wydawnictwo Ariel, 1992.

[2] *Ada Reference Manual ISO/IEC 8652:200y(E) Ed. 3*. Ada Europe. 2006.

[3] A. Burns i B. Dobbing. „The Ravenscar Profile for Real–Time and High Integrity Systems". W: *CrossTalk* 16.11 (2003), s. 9–12.