|  |  |
| --- | --- |
| *Przemysław Musiał 242473*  *Marcin Giska 242390* | Rok akademicki *2022/23*  *Poniedziałek, 15:45*  *Prowadzący: dr inż. Krzysztof Lichy* |

Data oddania: Ocena:

**Sztuczna inteligencja i systemy ekspertowe**

Zadanie 2: Poprawa lokalizacji UWB przy pomocy sieci neuronowych

1. **Cel**

Celem części programistycznej było zaprojektowanie i zaimplementowanie sieci neuronowej, która pozwoli na korygowanie błędów uzyskanych z systemu pomiarowego.

1. **Opis architektury sieci neuronowej**

Do zaimplementowania sieci neuronowej został wykorzystany interfejs Keras z biblioteki TensorFlow.  
- Warstwa 1: 64 neurony, funkcja aktywacyjna - ReLU  
- Warstwa 2: 32 neurony, funkcja aktywacyjna - ReLU  
- Warstwa 3: 16 neurony, funkcja aktywacyjna – ReLU

- Warstwa 4: 8 neurony, funkcja aktywacyjna – ReLU  
- Warstwa 5: 2 neurony, funkcja aktywacyjna – Sigmoid

Wagi neuronów:

[[ 0.07507 -0.26235402 -0.1638467 -0.1601508 0.48403364 -0.07333209

0.21995789 -0.0956281 0.2745356 0.24061441 -0.29373372 -0.03077607

-0.1977357 0.01242744 0.05264459 0.11933134 -0.00287846 0.19915308

0.3095348 0.38257024 -0.03710815 0.12965171 0.02358457 -0.28553492

-0.33381432 0.2784158 0.01161292 0.09475357 0.30181593 0.12476623

-0.17363063 -0.20485881 -0.17686622 0.34103662 -0.07889722 -0.01879697

-0.18436083 -0.11603399 0.3131766 -0.19976558 0.2679308 -0.29946834

0.19686161 0.0967225 -0.21379521 -0.25173596 0.1743734 0.2483384

-0.24663335 0.3174123 0.21197374 0.3404229 -0.2944067 -0.03425765

-0.32171416 0.31075624 -0.1802159 0.34315053 -0.00928134 -0.2777361

-0.17197196 0.05470686 0.2836668 -0.16871239]

[-0.57346004 -0.12161392 0.28223845 0.62589425 -0.40059966 -0.18827164

-0.50831336 -0.26395556 -0.04879035 0.38234937 -0.20031281 0.33642682

0.07226059 -0.32757416 0.35302764 0.37238008 -0.08771144 0.2780216

-0.5044546 0.257925 -0.13625844 0.10850492 0.532528 -0.02261394

0.22942573 -0.40678704 0.00417479 -0.2196061 -0.6922334 -0.49436668

0.03248739 -0.08206646 -0.13025118 0.5889908 -0.08971477 0.3389687

0.05594277 -0.29608253 -0.4489692 0.32792336 -0.03504704 -0.03459731

0.39134958 0.5285083 0.19795412 -0.17327206 0.07658665 -0.10601222

-0.21438894 -0.31970745 0.25158668 0.40115193 0.09562522 -0.18760137

-0.14319159 -0.49833855 0.00227317 0.15590037 -0.24587668 0.02352029

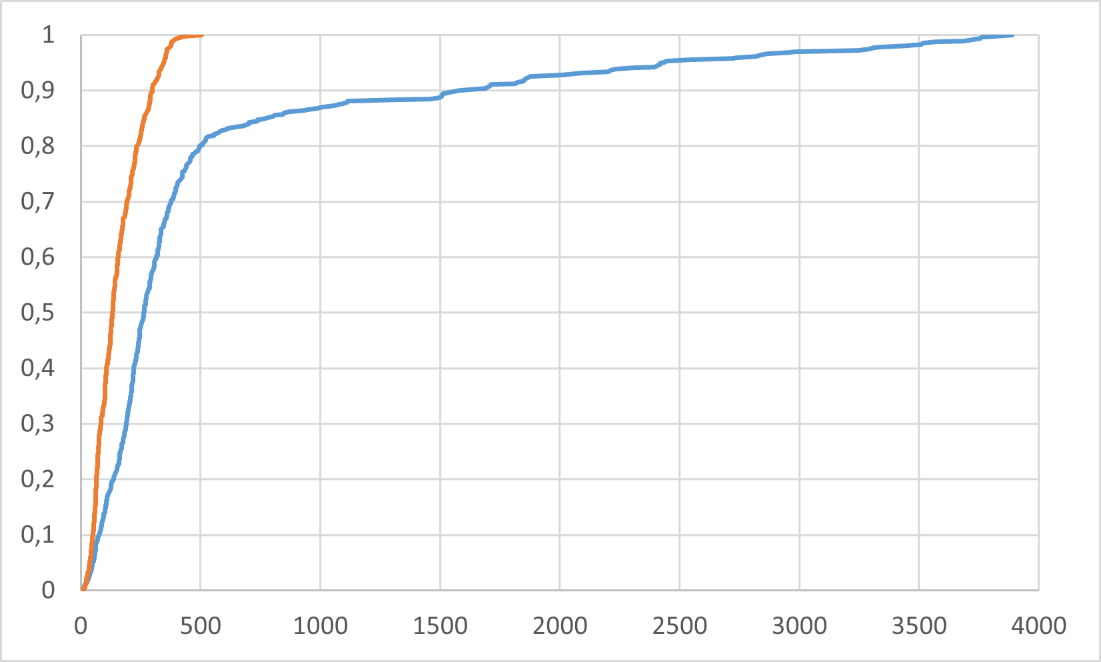
0.47551036 0.00459053 -0.11542412 0.32865402]]

Trenowanie sieci neuronowej zajmowało 150 epok.

1. **Opis algorytmu uczenia sieci neuronowej**

* Inicjalizacja danych: W pierwszym kroku inicjalizowane są niezbędne biblioteki oraz wczytywane są dane wejściowe z plików Excel.
* Przygotowanie danych: Następnie dane wejściowe są przetwarzane w celu przygotowania ich do uczenia sieci. Proces ten obejmuje usunięcie niektórych kolumn, wypełnienie wartości NaN zerami oraz normalizację wartości danych.
* Podział danych treningowych: Kolejnym krokiem jest podział danych treningowych na dane treningowe oraz dane walidacyjne. Dane treningowe stanowią około 90% zbioru, natomiast dane walidacyjne to pozostałe 10%.
* Utworzenie modelu sieci neuronowej: Algorytm tworzy sekwencyjny model sieci neuronowej przy użyciu biblioteki TensorFlow. Model składa się z warstw gęstych (Dense), gdzie każda warstwa zawiera różną liczbę neuronów oraz funkcję aktywacji.
* Kompilacja modelu: Następnie model sieci neuronowej jest kompilowany, co obejmuje określenie optymalizatora (Adam), funkcji straty (MeanSquaredError) oraz metryk (dokładność).
* Uczenie modelu: Algorytm rozpoczyna proces uczenia modelu na danych treningowych przez określoną liczbę epok. W każdej epoce, dane są podzielone na paczki o określonym rozmiarze (batch\_size), a model jest aktualizowany w celu minimalizacji funkcji straty. Model jest również oceniany na danych walidacyjnych w celu monitorowania postępu uczenia.
* Ocena modelu: Po zakończeniu procesu uczenia, model jest oceniany na danych weryfikacyjnych w celu określenia jego dokładności. Dodatkowo, waga pierwszej warstwy neuronowej jest wypisywana w celu analizy modelu.
* Przetwarzanie danych: Nauczoną sieć można użyć do przewidywania wyników na danych weryfikacyjnych. Wyniki są przekształcane do pierwotnego zakresu danych, aby umożliwić ich interpretację.
* Obliczenie błędu: Na podstawie wyników przewidywanych i prawdziwych danych, obliczany jest błąd pomiarowy, zarówno dla wyników przefiltrowanych, jak i nieprzefiltrowanych.
* Eksport danych: Na koniec wyniki są eksportowane do pliku Excel w celu dalszej analizy. Wyniki zawierają przewidziane wartości, błędy pomiarowe oraz inne informacje, które mogą być istotne dla dalszych działań.

1. **Porównanie dystrybuant błędu pomiaru**

****

Rys. 1 Dystrybuant błędu pomiaru

Linia o kolorze niebieskim przedstawia dystrybuantę błędu z danych przefiltrowanych.

Linia o kolorze pomarańczowym przedstawia dystrybuantę błędu z danych nieprzefiltrowanych.

Można zauważyć, że występuje poprawa jakości pomiaru, więcej pomiarów ma mniejsze błędy.

1. **Kod programu**

import glob  
import os  
import pandas as pd  
import tensorflow as tf  
import numpy as np  
  
columns = [  
 'Unnamed: 0',  
 'version',  
 'alive',  
 'tagId',  
 'success',  
 'timestamp',  
 'data\_\_tagData\_\_gyro\_\_x',  
 'data\_\_tagData\_\_gyro\_\_y',  
 'data\_\_tagData\_\_gyro\_\_z',  
 'data\_\_tagData\_\_magnetic\_\_x',  
 'data\_\_tagData\_\_magnetic\_\_y',  
 'data\_\_tagData\_\_magnetic\_\_z',  
 'data\_\_tagData\_\_quaternion\_\_x',  
 'data\_\_tagData\_\_quaternion\_\_y',  
 'data\_\_tagData\_\_quaternion\_\_z',  
 'data\_\_tagData\_\_quaternion\_\_w',  
 'data\_\_tagData\_\_linearAcceleration\_\_x',  
 'data\_\_tagData\_\_linearAcceleration\_\_y',  
 'data\_\_tagData\_\_linearAcceleration\_\_z',  
 'data\_\_tagData\_\_pressure',  
 'data\_\_tagData\_\_maxLinearAcceleration',  
 'data\_\_anchorData',  
 'data\_\_acceleration\_\_x',  
 'data\_\_acceleration\_\_y',  
 'data\_\_acceleration\_\_z',  
 'data\_\_orientation\_\_yaw',  
 'data\_\_orientation\_\_roll',  
 'data\_\_orientation\_\_pitch',  
 'data\_\_metrics\_\_latency',  
 'data\_\_metrics\_\_rates\_\_update',  
 'data\_\_metrics\_\_rates\_\_success',  
 'data\_\_coordinates\_\_x',  
 'data\_\_coordinates\_\_y',  
 'data\_\_coordinates\_\_z',  
 'reference\_\_x',  
 'reference\_\_y']  
  
if \_\_name\_\_ == '\_\_main\_\_':  
  
 # Odczytanie danych do późniejszego przefiltrowania  
 verif\_data = pd.read\_excel('./dane/pomiary/F10/f10\_random\_1p.xlsx')  
 verif\_measured\_x = verif\_data.pop('data\_\_coordinates\_\_x')  
 verif\_measured\_y = verif\_data.pop('data\_\_coordinates\_\_y')  
 verif\_reference\_x = verif\_data.pop('reference\_\_x')  
 verif\_reference\_y = verif\_data.pop('reference\_\_y')  
 verif\_measured = pd.concat([verif\_measured\_x, verif\_measured\_y], axis=1)  
 verif\_reference = pd.concat([verif\_reference\_x, verif\_reference\_y], axis=1)  
  
 # Odczytanie danych potrzebnych do uczenia sieci neuronowej  
 all\_files = glob.glob(os.path.join('./dane/pomiary/F10/f10\_stat\_\*.xlsx'))  
 df\_from\_each\_file = (pd.read\_excel(f, names=columns) for f in all\_files)  
 concatenated\_df = pd.concat(df\_from\_each\_file, ignore\_index=True)  
  
 measured\_x = concatenated\_df.pop('data\_\_coordinates\_\_x')  
 measured\_y = concatenated\_df.pop('data\_\_coordinates\_\_y')  
 training\_data = pd.concat([measured\_x, measured\_y], axis=1)  
  
 reference\_x = concatenated\_df.pop('reference\_\_x')  
 reference\_y = concatenated\_df.pop('reference\_\_y')  
 reference\_data = pd.concat([reference\_x, reference\_y], axis=1)  
  
 # Zmiana wszystkich wartosci "nan" na zera  
 verif\_measured.fillna(0, inplace=True)  
 verif\_reference.fillna(0, inplace=True)  
 training\_data.fillna(0, inplace=True)  
 reference\_data.fillna(0, inplace=True)  
  
 # Dodanie offsetu dla lepszego odczytu  
 training\_data = (training\_data.astype('float32') + 2000) / 10000  
 reference\_data = (reference\_data.astype('float32') + 2000) / 10000  
  
 # Podzial danych treningowych na dane treningowe oraz walidacyjne  
 rowCount = int(len(training\_data))  
 training\_data\_1 = training\_data[:(9 \* (rowCount//10))]  
 reference\_data\_1 = reference\_data[:(9 \* (rowCount//10))]  
 val\_data\_measured = training\_data[(9 \* (rowCount//10)):]  
 val\_data\_reference = reference\_data[(9 \* (rowCount//10)):]  
 val\_data\_measured.reset\_index(drop=True, inplace=True)  
 val\_data\_reference.reset\_index(drop=True, inplace=True)  
  
 # Dodanie offsetu dla lepszego odczytu  
 verif\_measured = (verif\_measured.astype('float32') + 2000) / 10000  
 verif\_reference = (verif\_reference.astype('float32') + 2000) / 10000  
  
 # Utworzenie sieci  
 model = tf.keras.Sequential([ #liczba epok - 150  
 tf.keras.layers.Dense(64, activation=tf.nn.relu), #I warstwa ukryta - 64 neurony, funkcja aktywacyjna to ReLU  
 tf.keras.layers.Dense(32, activation=tf.nn.relu), #II warstwa ukryta - 32 neurony, funkcja aktywacyjna to ReLU  
 tf.keras.layers.Dense(16, activation=tf.nn.relu), #III warstwa ukryta - 16 neuronów, funkcja aktywacyjna to ReLU  
 tf.keras.layers.Dense(8, activation=tf.nn.relu), #IV warstwa ukryta - 8 neuronów, funkcja aktywacyjna to ReLU  
 tf.keras.layers.Dense(2, activation=tf.nn.sigmoid), #Warstwa wyjściowa - 2 neurony, funkcja aktywacyjna to funkcja sigmoidalna  
 ])  
  
 # Kompilacja sieci przy pomocy optymalizatora "Adam"  
 model.compile(optimizer=tf.keras.optimizers.Adam(), loss=tf.keras.losses.MeanSquaredError(), metrics=['accuracy'])  
  
 # Nauczanie sieci za pomoca danych statycznych  
 model.fit(np.asarray(training\_data\_1), np.asarray(reference\_data\_1), epochs=150, batch\_size=512,  
 validation\_data=(val\_data\_measured, val\_data\_reference))  
  
 # Wypisanie wag neuronow  
 model.evaluate(verif\_measured, verif\_reference, batch\_size=512)  
 weights = model.layers[0].get\_weights()[0]  
 print(weights)  
  
 # Uzycie nauczonej juz sieci w celu przefiltrowania danych  
 result = model.predict(verif\_measured)  
 result = result \* 10000 - 2000  
 result\_df = pd.DataFrame(result)  
  
 # Ulatwienie dostepu do danych poprzez przekazanie ich do list  
 result\_x\_array = []  
 result\_y\_array = []  
 for x, y in result:  
 result\_x\_array.append(x)  
 result\_y\_array.append(y)  
 reference\_x\_array = verif\_reference\_x.to\_list()  
 reference\_y\_array = verif\_reference\_y.to\_list()  
 measured\_x\_array = verif\_measured\_x.to\_list()  
 measured\_y\_array = verif\_measured\_y.to\_list()  
  
 # Blad pomiarowy  
 error\_filtered = []  
 error\_unfiltered = []  
 for i in range(len(result\_x\_array)):  
 buff\_filtered = (np.sqrt(np.power(result\_x\_array[i] - reference\_x\_array[i], 2) + np.power(result\_y\_array[i] - reference\_y\_array[i], 2)))  
 buff\_unfiltered = (np.sqrt(np.power(measured\_x\_array[i] - reference\_x\_array[i], 2) + np.power(measured\_y\_array[i] - reference\_y\_array[i], 2)))  
 error\_filtered.append(buff\_filtered)  
 error\_unfiltered.append(buff\_unfiltered)  
 result\_df['error\_filtered'] = error\_filtered  
 result\_df['error\_unfiltered'] = error\_unfiltered  
  
 # Wyeksportowanie danych do pliku excel  
 result\_df.to\_excel('wynik\_F10.xlsx', engine='xlsxwriter')  
 print(result)