Klasyfikator ruchów czujnika IMU na podstawie rekurencyjnej sieci neuronowej

Mateusz Woźniak
wozniakmat@student.agh.edu.pl

Maciej Pawłowski
maciekp@student.agh.edu.pl

1 Abstrakt

Przedmiotem tego artykułu jest omówienie realizacji zadania klasyfikacji ruchów z urządzenia IMU (Inertial Measurement Unit). Czujnik IMU określa przyśpieszenia postępowe i kątowe używając żyroskopu, akcelerometru i magnetometru. IMU jest powszechnie stosowane w lotnictwie, robotyce, wirtualnej rzeczywistości i medycynie. W lotnictwie umożliwia precyzyjne sterowanie statkami powietrznymi, a w robotyce wspomaga autonomiczne poruszanie się robotów. Jednym z zastosowań klasyfikatora ruchów może być detekcja przeciągnięcia samolotu. Z kolei w motoryzacji, IMU znajduje użycie w systemach kontroli stabilności pojazdów oraz w zaawansowanych systemach wspomagania kierowcy, które poprawiają bezpieczeństwo. My chcemy zaproponować realizację klasyfikatora 5 z góry ustalonych ruchów używając rekurencyjną sieć neuronowej.

Do pomiarów wykorzystaliśmy smartfon Samsung Galaxy S10e 2019 wyposażony w czujnik IMU. Implementację modelu wykonaliśmy we frameworku PyTorch. Trening sieci neuronowej była wykonywany na Apple Macbook M1 16GB.

2 Zbiór danych

Dane zostały zebrane z urządzenia Samsung Galaxy S10e 2019. Każdy ruch został zebrany ręcznie a seria danych z czujników była zapisywana do pliku csv o ilości wierszy takiej jak ilość kroków czasowych. Interwał samplowania pomiaru z czujnika ustaliliśmy na $50ms$. To znaczy, że w ciągu każdej sekundy trwania ruchu następowało 20 odczytów.

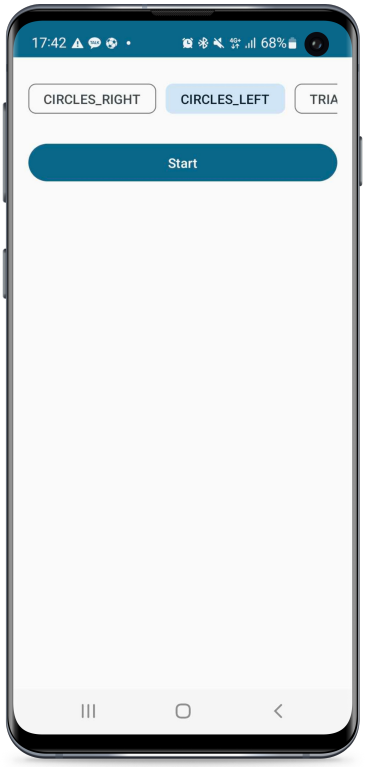
Ruchy były wykonywane przy włączonej aplikacji mobilnej napisanej w Kotlinie (rys. 1).

Dane były wysyłane na zdalny serwer na którym był uruchomiony mikroserwis HTTP napisany w Go. Mikroserwis zbierał dane i zapisywał je na dysk. Dzięki temu zbieranie danych odbywało się sprawnie, a mikroserwis dawał nam informacje o ilości ruchów dla każdej z klas.

Przykładowy plik csv:

```
gyro_x;gyro_y;gyro_z;magnetometer_x;...
-0.062384613;-0.15294538;-0.32650748;-43.62;...
-0.87361366;0.41210496;-0.49510628;-43.5;...
-1.7758616;0.20685424;-1.4266758;-43.92;...
-1.8662697;-0.46876273;-1.4040737;-44.399998;...
-0.7575493;-0.8077929;-1.488984;-44.28;...
0.06895141;-1.5170075;-2.1273382;-46.14;...
```

Zebraliśmy dane dla następujących klas:



Rysunek 1: Aplikacja mobilna do zbierania danych z czujnika IMU

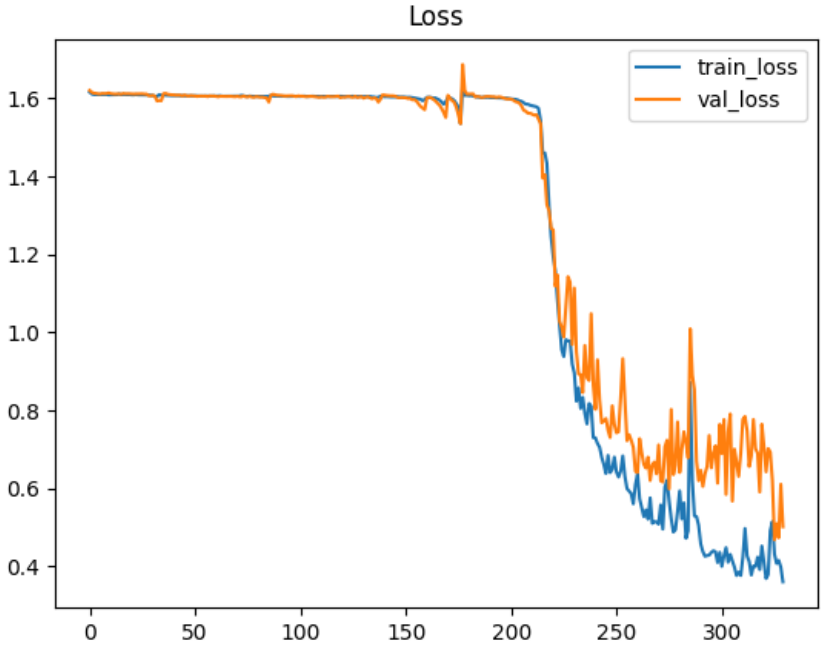
Klasa	Opis ruchu	Ilość sampli
SQUARE	Ruch imitujący rysowanie kwadratu	203
TRIANGLE	Ruch imitujący rysowanie trójkąta	179
CIRCLES_LEFT	Rysowanie okręgu przeciwnie z ruchem wskazówek zegara	193
CIRCLES_RIGHT	Rysowanie okręgu zgodnie z ruchem wskazówek zegara	191
FORWARD_BACK	Ruch "od siebie - do siebie"	187

3 Model

Ze względu na to, że zadanie klasyfikacji ruchów ma być niezależne od czasu, tzn. że ruch może zawierać dowolną ilość sampli zdecydowaliśmy się użyć rekurencyjnej sieci neuronowej z komórką LSTM. Long Short-Term Memory (LSTM) to rodzaj architektury sieci neuronowej rekurencyjnej (RNN), zaprojektowanej w celu przezwyciężenia ograniczeń tradycyjnych RNN w przechwytywaniu i uczeniu się długoterminowych zależności w danych sekwencyjnych. LSTMy zostały wprowadzone przez Seppa Hochreitera i Jürgena Schmidhubera w 1997 roku. Chcemy, aby sieć neuronowa nauczyła się zaleności w czasie pomiędzy zmianami w przyśpieszeniach czujnika. W związku z tym w pierwszym podejściu zdecydowaliśmy sprawdzić architekturę składającą się z 32 komórek LSTM oraz dwóch transformacji liniowych $nn.Linear()$ o wielkości 24 i 5. Na ostatniej warstwie została zastosowana funkcja aktywacji $softmax$ by uzyskać dystrybucję prawdopodobieństwa klas (realizuje to $nn.CrossEntropyLoss()$)

Po kilku próbach znaleźlimy hiperparametry modelu, które dają najlepsze zachowanie sieci. LSTM musi mieć 22 komórki, a warstwa gęsta musi mieć 32 neurony. Przeszukiwanie przestrzeni hiperparametrów zostało zrealizowane metodyką gridsearch. Finalny model ma następującą architekturę:

```
import torch
import torch.nn as nn

class Net(nn.Module):
```



Rysunek 2: Wykres funkcji straty

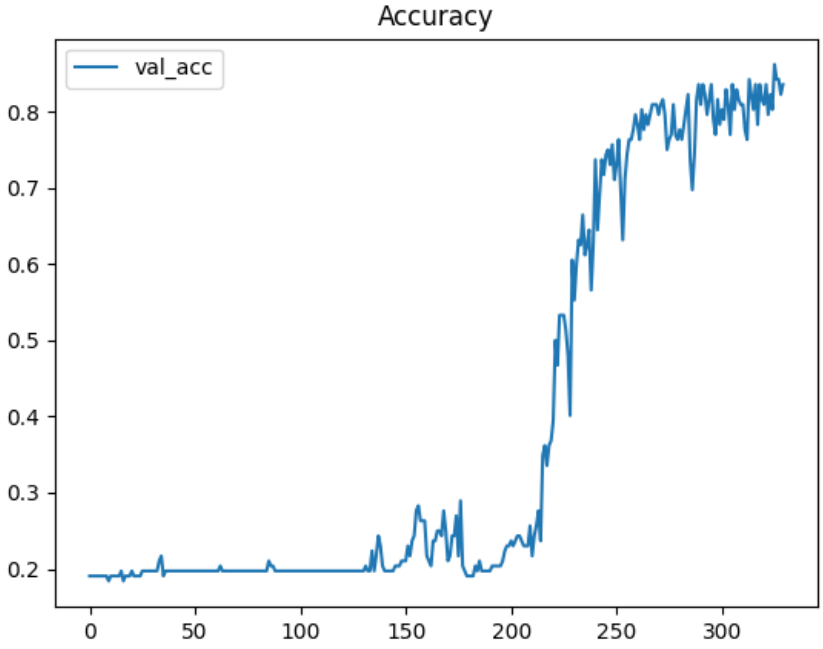
```
def __init__(self, input_size):
    super(Net, self).__init__()
    self.lstm = nn.LSTM(input_size, 22, batch_first=True)
    self.fc1 = nn.Linear(22, 32)
    self.fc2 = nn.Linear(32, 5)

def forward(self, x):
    _, (h_n, _) = self.lstm(x)
    x = h_n[-1, :, :]
    x = self.fc1(x)
    x = self.fc2(x)
    return x
```

4 Obserwacje

Model osiąga zbieżność dostarczając przy tym jakościowe predykcje. Zadanie znajdowania zależności między odczytami dalego od siebie nie jest trywialne, aczkolwiek optymalizator jest w stanie znaleźć odpowiedni kierunek by dostosować wagi sieci (rys. 2). Użyty optymalizator to $Adam$ ze współczynnikiem uczenia 0.001 Uzyskana precyzja to ok. 90% (rys. 3).

5 Wnioski

c



Rysunek 3: Wykres precyzji na zbiorze walidacyjnym