

Human Activity Recognition with Wi-Fi Sensing

Opis: Celem pracy jest dokonanie przeglądu dostępnych rozwiązań do rozpoznawania aktywności osób z użyciem analizy stanu kanału Wi-Fi (*channel state information*). Należy wybrać najbardziej obiecujące rozwiązanie i go przetestować z użyciem rzeczywistych urządzeń (np. ESP32). W pracy należy się skupić przede wszystkim na porównaniu modeli uczenia maszynowego wykorzystywanych w tych zastosowaniach oraz podjęcie próby stworzenia własnego modelu, przeznaczonego do realizacji wybranej przez siebie aplikacji (np. monitoring osób starszych).

1: Analiza repozytoriów

Adres repozytorium	Zakres repozytorium	CSI?	ML?	Rok	Aplikacja
https://github.com/xyanchen/WiFi-CSI-Sensing-Benchmark	Analiza danych CSI przy użyciu uczenia maszynowego	Nie	Tak	2023	Rozpoznawanie aktywności człowieka na podstawie danych CSI
https://github.com/StevenMHernandez/ESP32-CSI-Tool	Zbieranie danych CSI z ESP32 i obsługa w Pythonie	Tak	Nie	2022	Zbieranie i analiza danych CSI w Pythonie
https://github.com/winwinashwin/CSI-Sense-Zero/tree/main	Zbieranie CSI z ESP32	Tak	Nie	2022	Wykrywanie i klasyfikacja ruchu ludzi na podstawie CSI
https://github.com/thu4n/ESP32-WiFi-Sensing/tree/master	Zbieranie CSI z ESP32 + ML do rozpoznawania aktywności	Tak	Tak	2022	Rozpoznawanie aktywności człowieka na podstawie CSI
https://github.com/RikeshMMM/ESP32-CSI-Python-Parser	Parser danych CSI z ESP32 w Pythonie	Nie	Nie	2022	Ekstrakcja i przetwarzanie amplitudy oraz fazy CSI
https://github.com/IMDEANetworksWNG/UbiLocate	Zbieranie danych CSI z routerów 802.11ac i lokalizacja urządzeń	Tak	Nie	2021	Lokalizacja wewnętrz budynkowa na podstawie CSI

Pełna analiza repozytoriów

WiFi-CSI-Sensing-Benchmark (<https://github.com/xyanchen/WiFi-CSI-Sensing-Benchmark>)

Framework do testowania i porównywania algorytmów uczenia maszynowego w zadaniach związanych z analizą informacji o stanie kanału (CSI). Zawiera gotowe zbiory danych, narzędzia do przetwarzania i ekstrakcji cech, oraz szeroki zestaw modeli ML – od klasycznych (Random Forest, SVM, XGBoost), po bardziej zaawansowane sieci neuronowe.

- obsługuje m.in. klasyfikację obecności (presence detection), czyli detekcję ruchu.
- oferuje pełny proces: od przetwarzania danych, przez ekstrakcję cech (np. wariancja, rozstęp), po trening i walidację modeli klasyfikacyjnych.
- pozwala na ocenę skuteczności różnych modeli (m.in. RF, XGB, Voting) w różnych warunkach (LoS, NLoS), co jest kluczowe przy projektach dotyczących detekcji obecności w środowiskach rzeczywistych.
- zgodne z założeniami artykułu (lekki ML, binarna klasyfikacja)

ESP32-CSI-Tool (<https://github.com/StevenMHernandez/ESP32-CSI-Tool>)

Narzędzie do pasywnego i aktywnego zbierania informacji o stanie kanału (CSI) przy użyciu mikrokontrolerów ESP32. Oferuje pełne wsparcie dla różnych trybów pracy urządzenia (Station, Access Point, Promiscuous).

- kompatybilność sprzętowa z ESP32, działa bezpośrednio na układzie ESP32 bez potrzeby modyfikowania firmware, co znaczco ułatwia integrację ze środowiskiem testowym.
- umożliwia dostęp do niskopoziomowych danych CSI w formie amplitudy i fazy dla 64 subcarrierów.
- pozwala na zbieranie danych CSI w trybie Passive i Active — umożliwia symulację różnych scenariuszy z artykułu.
- dane CSI są automatycznie zapisywane do plików CSV.

CSI-Sense-Zero (<https://github.com/winwinashwin/CSI-Sense-Zero/tree/main>)

Projekt wykorzystujący dane CSI z ESP32 do detekcji obecności człowieka bez potrzeby trenowania modelu ML.

- bazuje na danych zbieranych przy użyciu ESP32-CSI-Tool.
- używa statystycznych cech (entropii i odchylenia standardowego) do wykrywania zmian w sygnale.
- nie zawiera klasyfikatora ML,
- nie pozwala trenować modelu na etykietowanych danych,
- nie realizuje podejścia opisanego w artykule IEEE.

ESP32-WiFi-Sensing (<https://github.com/thu4n/ESP32-WiFi-Sensing/tree/master>)
Framework do zbierania danych CSI z ESP32, przesyłania ich do komputera (lub Jetson Nano) i klasyfikowania ruchu człowieka przy użyciu prostych algorytmów ML (głównie kNN).

- zgodność z ESP32-CSI-Tool
- zawiera klasyfikację obecności - ma prosty klasyfikator binarny (np. kNN)
- gotowy pipeline ML – zbieranie danych, obróbka CSI, predykcja obecności
- zaprojektowane z myślą o edge computing (np. Jetson Nano)
- zawiera narzędzia do etykietowania danych
- brak dokumentacji i opisu danych

ESP32-CSI-Python-Parser (<https://github.com/RikeshMMM/ESP32-CSI-Python-Parser>)

Służy wyłącznie do parsowania danych CSI zebranych za pomocą ESP32-CSI-Tool.

- odczytuje amplitudy, fazy, subcarriery, timestamps itp.
- ułatwia feature engineering przed trenowaniem klasyfikatora.
- integracja z WiFi-CSI-Sensing-Benchmark lub własnym pipeline'em ML.
- nie tworzy etykiet, nie robi agregacji danych – dane są w formie pakietów.

UbiLocate (<https://github.com/IMDEANetworksWNG/UbiLocate>)

Dotyczy pasywnego systemu lokalizacji wewnętrz budynkowej z użyciem CSI i routerów 802.11ac (5 GHz).

- różne etapy przetwarzania CSI: ekstrakcja, kalibracja, analiza
- przydatne funkcje do pracy z CSI: filtrowanie szumów, wykrywanie szczytów, analizy korelacyjne
- dobrze udokumentowana struktura projektu
- celem repozytorium jest lokalizacja urządzenia Wi-Fi, a nie detekcja obecności
- projekt dla routerów 802.11ac (np. ASUS RT-AC86U), a nie ESP32

Wybór repozytoriów

Zdecydowałam się oprzeć swoje działania na dwóch repozytoriach – **ESP32-CSI-Tool** oraz **WiFi-CSI-Sensing-Benchmark**. Oba uzupełniają się funkcjonalnie, a ich połączenie pozwala zrealizować pełny pipeline – od zbierania danych CSI, przez ekstrakcję cech, aż po trenowanie i ocenę modeli klasyfikacyjnych.

- **ESP32-CSI-Tool – zbieranie danych**

Stanowi niezawodne narzędzie do zbierania danych CSI z układów ESP32, w trybie pasywnym lub aktywnym. Jego największą zaletą jest:

- pełna zgodność z platformą ESP32,
- gotowy eksport danych w formacie CSV,
- dane w postaci amplitudy i fazy dla wielu subcarrierów,
- możliwość symulacji warunków LoS/NLoS.

- **WiFi-CSI-Sensing-Benchmark – przetwarzanie i klasyfikacja**

Stanowi kompletny framework do analizy danych CSI z wykorzystaniem uczenia maszynowego. Repozytorium zawiera:

- funkcje do przetwarzania danych (np. wariancja, IQR, MAD),
- zestaw modeli klasyfikacyjnych (Random Forest, XGBoost, SVM, Voting),
- narzędzia do walidacji i porównywania skuteczności modeli.

2. Dostępne modele i ich działanie

0.1 1. MLP – Multilayer Perceptron

Klasyczna architektura sieci neuronowej typu *feedforward*, składająca się z:

- **Warstwy wejściowej:** przyjmującej dane wejściowe.
- **Jednej lub więcej warstw ukrytych:** gdzie każdy neuron jest połączony z każdym neuronem w poprzedniej i następnej warstwie (pełne połączenia).
- **Warstwy wyjściowej:** generującej wynik modelu.

Każdy neuron w warstwach ukrytych i wyjściowej stosuje nieliniową funkcję aktywacji (np. ReLU, sigmoidę) do przetworzenia sumy ważonej swoich wejść.

1.1. Przetwarzanie danych CSI

Dane CSI to zazwyczaj macierze o wymiarach: *Czas* \times *Liczba anten* \times *Liczba podnośnych*. Aby dostosować dane do MLP, które oczekuje wektora 1D, macierz CSI jest spłaszczana (*flattening*) do jednego wektora.

MLP traktuje ten wektor jako zbiór cech i przetwarza go przez kolejne warstwy w pełni połączone:

- **Warstwa wejściowa:** przyjmuje wektor cech.
- **Warstwy ukryte:** każdy neuron oblicza sumę ważoną swoich wejść, stosuje funkcję aktywacji i przekazuje wynik do następnej warstwy.
- **Warstwa wyjściowa:** generuje wynik, np. prawdopodobieństwa przynależności do poszczególnych klas aktywności.

1.2. Podsumowanie modelu

MLP uczy się mapowania wysokowymiarowego wektora cech (spłaszczone dane CSI) na etykiety klas aktywności.

Zalety:

- Prosta architektura i implementacja.
- Niskie wymagania obliczeniowe.
- Szybki czas treningu.

Ograniczenia:

- Ignoruje strukturę przestrzenną i czasową danych CSI.
- Może mieć trudności z uchwyceniem złożonych wzorców w danych.
- Wrażliwy na skalowanie cech i wymaga starannej normalizacji danych wejściowych.

```
=====
Optymizacja zakończona.
```

Najlepsza próba:

- > Dokładność walidacji: 0.6630
 - > Najlepsze hiperparametry:
 - filters1: 32
 - filters2: 96
 - filters3: 160
 - dropout1: 0.3528335913842677
 - dropout2: 0.41942175880020394
 - dropout3: 0.32735903477883843
 - dropout_flatten: 0.4061878623180358
 - dense_units: 256
 - learning_rate: 0.0008853855849360852
- ```
=====
```

Rysunek 1: Schemat MLP

### 1.3. Co otrzymujemy na wyjściu?

W zadaniu klasyfikacji aktywności, MLP generuje wektor prawdopodobieństw dla każdej klasy. Wybierana jest klasa z najwyższym prawdopodobieństwem jako przewidywana aktywność.

## 0.2 2. LeNet

Jedna z pierwszych skutecznych konwolucyjnych sieci neuronowych, zaprojektowana przez Yanna LeCuna do rozpoznawania cyfr (MNIST). Jej prosta architektura jest nadal wykorzystywana jako punkt wyjścia do uczenia modeli przetwarzających dane przestrzenne – w tym CSI.

Składa się z następujących elementów:

- **Warstwy konwolucyjne:** uczą się lokalnych cech z danych wejściowych (np. krawędzi, wzorów).
- **Warstwy poolingowe:** redukują rozmiar reprezentacji (np. max pooling), zwiększaając odporność na przesunięcia.
- **Warstwy w pełni połączone (FC):** klasyfikator na końcu sieci.

### 2.1. Przetwarzanie danych CSI

Dane CSI mają postać:  $czas \times liczba\ anten \times liczba\ podno\acute{s}nych$ .

W przypadku LeNet, dane traktowane są jako *obraz 2D*: [antena  $\times$  subnośna] lub [czas  $\times$  subnośna].

Dane są często traktowane jako obraz o jednym kanale (1-channel grayscale-like).

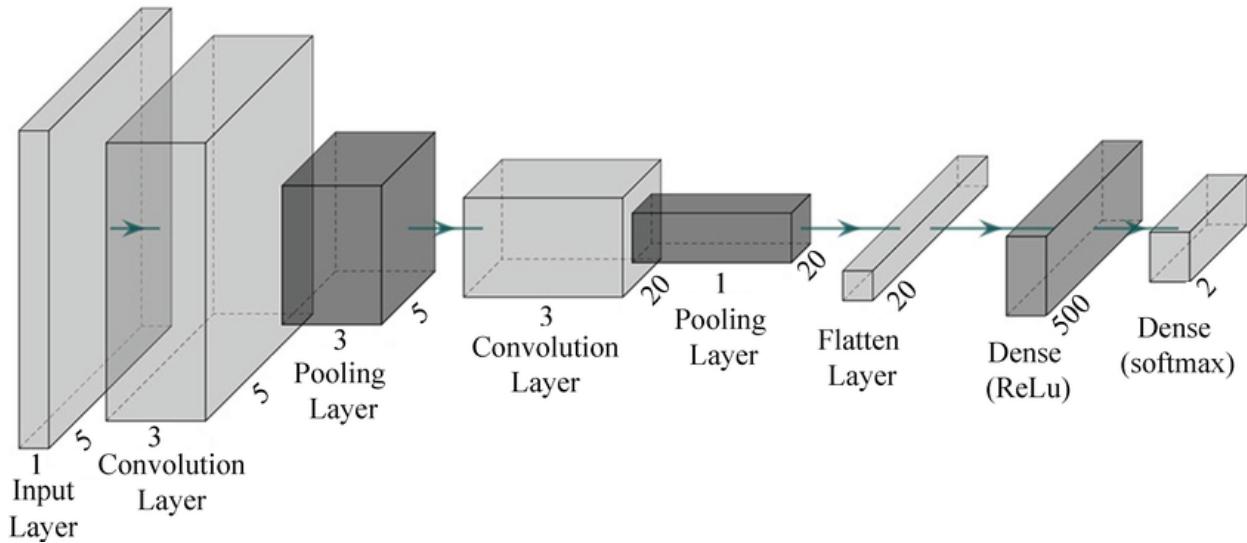
- Warstwa konwolucyjna uczy się lokalnych wzorców w CSI, np. pasm zakłóceń czy wzrostów amplitudy w określonych subnośnych.
- Pooling redukuje rozmiar danych i wzmacnia odporność na szum.
- Flattening: koniec sieci spłaszcza dane i przekazuje je do klasyfikatora (warstw FC).

### 2.2. Podsumowanie modelu

LeNet uczy się cech lokalnych – np. wzorców w ramach określonych anten i subnośnych. Rozpoznaje np. zakłócenia charakterystyczne dla danej aktywności (np. ruch w prawo może zakłócać tylko część pasma). W przeciwieństwie do MLP – zachowuje strukturę przestrzenną danych.

**Zalety:**

- Wydobywa lokalne cechy przestrzenne z CSI.
- Prosta, lekka architektura – dobra na początek.
- Mało parametrów – mniejsze ryzyko przeuczenia.



Rysunek 2: Schemat LeNet

Źródło: [https://www.researchgate.net/figure/The-structure-of-the-proposed-P-LeNet-model\\_fig6\\_353166963](https://www.researchgate.net/figure/The-structure-of-the-proposed-P-LeNet-model_fig6_353166963)

### Ograniczenia:

- Nie przetwarza sekwencji czasowej.
- Mniejsza zdolność uchwycenia globalnych zależności (całościowy kontekst).
- Może nie wystarczyć przy złożonych zadaniach z długimi sekwencjami.

### 2.3. Co otrzymujemy na wyjściu?

Tak jak w MLP, ostatnia warstwa to klasyfikator — zwraca wektor prawdopodobieństw dla klas aktywności.

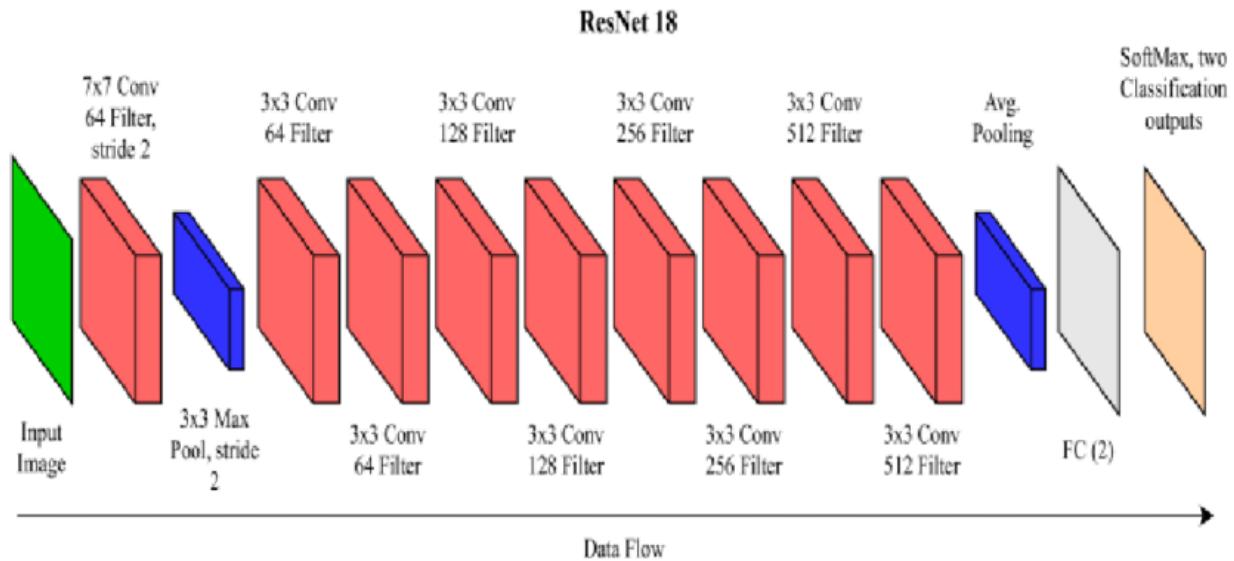
Wybierana jest klasa z najwyższym prawdopodobieństwem jako przewidywana aktywność.

## 0.3 ResNet – Residual Network (Sieć Resztkowa)

Gęboka sieć konwolucyjna wprowadzająca tzw. *połączenia resztkowe* (ang. *residual connections*), które umożliwiają trenowanie bardzo głębokich modeli bez problemu zanikania gradientu. Dzięki temu ResNet może efektywnie uczyć się złożonych reprezentacji danych, co jest szczególnie przydatne w analizie danych CSI.

### Architektura:

- **Bloki resztkowe:** zawierają konwolucje i połączenie skrótowe (skip connection), uczą się przekształcenia:  $F(x) = H(x) - x$ , czyli uczymy się tylko „zmiany” względem wejścia.
- **Połączenia resztkowe:** umożliwiają propagację gradientu nawet w bardzo głębokich sieciach.



Rysunek 3: Schemat ResNet-18

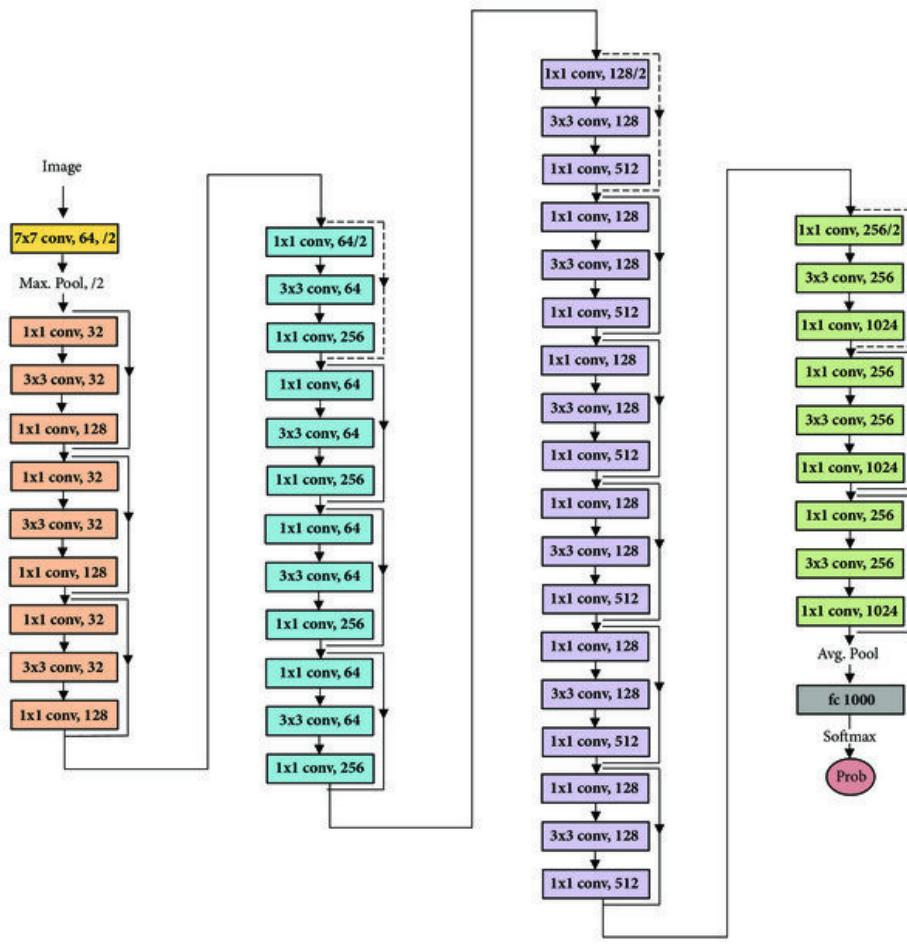
Źródło: [https://www.researchgate.net/figure/ResNet-18-architecture-20-The-numbers-added-to-the-end-of-ResNet-represent-the\\_fig2\\_349241995](https://www.researchgate.net/figure/ResNet-18-architecture-20-The-numbers-added-to-the-end-of-ResNet-represent-the_fig2_349241995)

- **Końcowe warstwy Fully Connected (FC):** służą do klasyfikacji aktywności po ekstrakcji cech.

### 3.1. Warianty ResNet dostępne w repozytorium

Repozytorium *WiFi-CSI-Sensing-Benchmark* implementuje następujące wersje modelu ResNet:

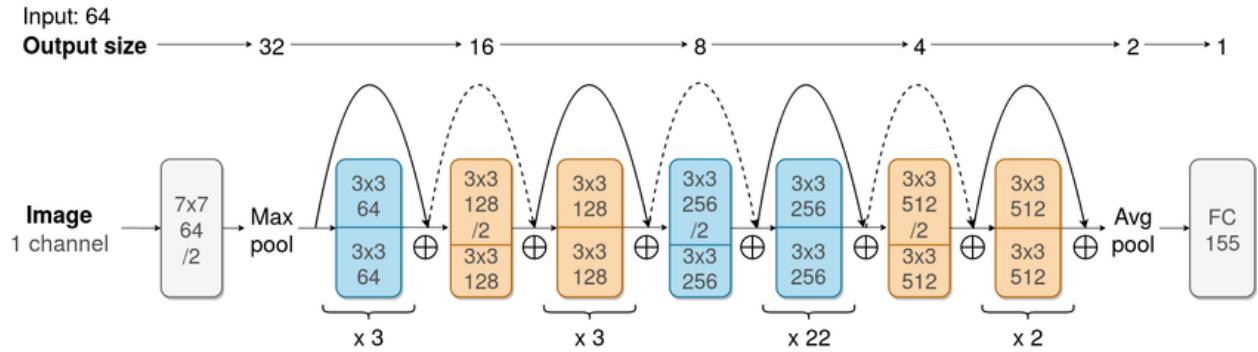
- **ResNet-18 – model do zastosowań na ograniczonym sprzęcie**
  - Odpowiedni do eksperymentów wstępnych oraz wdrożeń na urządzeniach typu edge, np. ESP32 z obsługą AI.
  - Prosta struktura bloków (*BasicBlock*) sprawia, że sieć jest odporna na przeuczenie przy małych zbiorach danych CSI.
  - Nadaje się do detekcji prostych aktywności (np. poruszanie się, stanie, upadek) przy niższej rozdzielczości cech.
  - Dzięki niewielkiej liczbie parametrów model może być łatwo konwertowany do formatu TensorFlow Lite lub ESP-DL.
  - Ograniczona głębokość powoduje, że sieć może nie uchwycić złożonych zależności między subnośnymi i antenami – sprawdza się lepiej przy wyraźnych, jednoznacznych aktywnościach.



Rysunek 4: Schemat ResNet-50

Źródło: [https://www.researchgate.net/figure/Block-diagram-of-Resnet-50-1-by-2-architecture\\_fig4\\_326198791](https://www.researchgate.net/figure/Block-diagram-of-Resnet-50-1-by-2-architecture_fig4_326198791)

- **ResNet-50 – zrównoważony wybór do analizy złożonych wzorców CSI**
  - Głębsza sieć z blokami typu *Bottleneck*, które umożliwiają modelowi uczenie się bardziej złożonych reprezentacji – przy zachowaniu rozsądniego rozmiaru.
  - Lepsze niż ResNet-18 rozpoznawanie subtelnych zmian w sygnale (np. różnica między skrętem ciała a uniesieniem ręki).
  - Możliwość trenowania na GPU w repozytorium benchmarku oraz dalszej konwersji do zoptymalizowanej wersji (np. przez ONNX + kwantyzację).
  - Kompromis między dokładnością a kosztem: nie wymaga tak wielu danych jak ResNet-101, a daje znaczaco lepsze wyniki niż modele płytkie.
  - Szczególnie przydatny w zastosowaniach takich jak wykrywanie mikroruchów, sekwencji aktywności oraz nieregularnych gestów.



Rysunek 5: Schemat ResNet-101

Źródło: [https://www.researchgate.net/figure/The-ResNet101-network-architecture-used-for-the-convolutional-engine-of-the-BNN-The-size\\_fig1\\_346555609](https://www.researchgate.net/figure/The-ResNet101-network-architecture-used-for-the-convolutional-engine-of-the-BNN-The-size_fig1_346555609)

- **ResNet-101 – model referencyjny do głębszej ekstrakcji cech**

- Zdolny do uchwycenia bardzo drobnych i złożonych różnic w danych CSI dzięki dużej liczbie warstw i parametrów.
- Przydatny do budowy modeli referencyjnych i benchmarków jakości – np. oceny, czy głębsze sieci rzeczywiście poprawiają skuteczność klasyfikacji aktywności.
- Nadaje się do analizy danych zawierających niewielkie różnice między klasami (np. przejścia między fazami ruchu: podnoszenie się, siadanie, zmiana pozycji).
- Wysokie ryzyko przeuczenia przy ograniczonej liczbie próbek – wymaga dużych zbiorów danych i zaawansowanych metod regularyzacji (dropout, augmentacje, normalizacja).
- Zbyt duży i wymagający obliczeniowo, by można go było wdrożyć na ESP32 – dlatego powinien być traktowany jako narzędzie offline do porównań jakości modeli.

### 3.2. Przetwarzanie danych CSI

Typowa próbka CSI to tensor o wymiarach:  $czas \times liczba\ anten \times liczba\ podno\acute{snych}$ . Dane są przekształcane do formy 2D, traktowanej jako obraz (lub obraz RGB: kanały = anteny, wysokość = czas, szerokość = subnośne).

- Dane przechodzą normalizację, zmianę rozmiaru (resize) oraz konwersję do tensora.
- Konwolucje i skip connection wyciągają złożone cechy przestrzenne z „obrazu” CSI.
- Sieć nie przetwarza danych czasowo – przetwarza pojedynczą migawkę.

Tabela 1: Rodzaje bloków resztkowych w modelach ResNet

| Typ bloku       | Używany w                | Struktura<br>(warstwy konw.)                                                | Zalety                                                                                       |
|-----------------|--------------------------|-----------------------------------------------------------------------------|----------------------------------------------------------------------------------------------|
| BasicBlock      | ResNet-18,<br>ResNet-34  | 2 konwolucje ( $3 \times 3 \rightarrow 3 \times 3$ )                        | Łatwy w implementacji,<br>prostszy, mniej<br>parametrów, dobry dla<br>CPU                    |
| BottleneckBlock | ResNet-50,<br>ResNet-101 | 3 konwolucje ( $1 \times 1 \rightarrow 3 \times 3 \rightarrow 1 \times 1$ ) | Lepsza wydajność w<br>głębokich sieciach,<br>oszczędność pamięci,<br>zoptymalizowany dla GPU |

Tabela 2: Porównanie wariantów modelu ResNet

| Wersja     | Liczba warstw | Typ bloku       | Parametry | Zastosowanie                         | Zasoby/potrzeby         |
|------------|---------------|-----------------|-----------|--------------------------------------|-------------------------|
| ResNet-18  | 18            | BasicBlock      | ~11 mln   | Szybkie eksperymenty,<br>małe zbiory | CPU / mały zbiór danych |
| ResNet-50  | 50            | BottleneckBlock | ~25 mln   | Lepsza dokładność                    | GPU / większy zbiór     |
| ResNet-101 | 101           | BottleneckBlock | ~44 mln   | Złożone zadania, duże zbiory         | Duży GPU / dużo danych  |

### **3.3. Podsumowanie modelu**

**Zalety:**

- Bardzo skuteczna ekstrakcja złożonych cech przestrzennych.
- Możliwość trenowania bardzo głębokich sieci bez utraty gradientu.
- Lepsze wyniki niż klasyczne CNN (np. LeNet) na dużych zbiorach danych.

**Ograniczenia:**

- Brak pamięci sekwencyjnej.
- Wysokie wymagania obliczeniowe w wersjach 50 i 101.
- Może wymagać więcej danych, by uniknąć przeuczenia.

### **3.4. Co otrzymujemy na wyjściu?**

- Ostatnia warstwa FC daje wektor prawdopodobieństw
- Najwyższa wartość decyduje o przewidywanej klasie aktywności (np. „wstawanie”).

## **0.4 RNN – Recurrent Neural Network**

Architektura zaprojektowana specjalnie do przetwarzania danych sekwencyjnych.

W przeciwieństwie do MLP czy CNN, RNN posiada pamięć – przetwarza dane krok po kroku w czasie i „pamięta” poprzednie stany.

W kontekście CSI jest to szczególnie przydatne, ponieważ dane pochodzą z transmisji Wi-Fi, a zmiany sygnału w czasie często świadczą o aktywności człowieka (np. wstawanie, chodzenie).

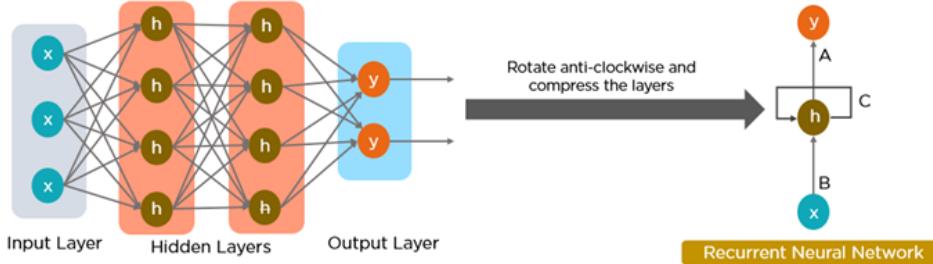
**Architektura:**

**• Warstwa RNN:**

- przyjmuje sekwencję wektorów jako wejście (np. CSI w czasie),
- przetwarza każdy krok czasowy osobno,
- stan ukryty (hidden state) jest przekazywany między krokami czasowymi,
- może występować w wersji:
  - \* jednokierunkowej (czyta tylko w przód),
  - \* dwukierunkowej (czyta w przód i wstecz – BiRNN).

**• Warstwa wyjściowa (FC):**

- po przejściu przez wszystkie kroki sekwencji, ostatni (lub pośredni) stan jest używany do klasyfikacji.



Rysunek 6: Schemat RNN. W przypadku danych CSI:  $x_t$  oznacza kolejną próbki CSI (input),  $h_t$  – stan ukryty (hidden state), a  $y_t$  – przewidywaną klasę aktywności (output).

Źródło: <https://blog.nashtechglobal.com/a-complete-guide-to-recurrent-neural-network/>

#### 4.1. Przetwarzanie danych CSI

Typowe wejście do RNN: tensor  $czas \times cechy$ , np. po spłaszczeniu jednej ramki CSI. RNN „czyta” dane ramka po ramce, utrzymując stan pamięci.

Dzięki temu może uczyć się, jak zmienia się sygnał w czasie – np. że amplituda rośnie przez 5 ramek, co może oznaczać ruch.

#### 4.2. Podsumowanie modelu

Uczy się zależności czasowych – np. jak zachowanie CSI zmienia się w czasie podczas ruchu. Model potrafi „zrozumieć sekwencję”, a nie tylko pojedynczą ramkę.

Dzięki pamięci wewnętrznej, działa lepiej niż ResNet/LeNet w kontekście ciągów ruchu.

#### Zalety:

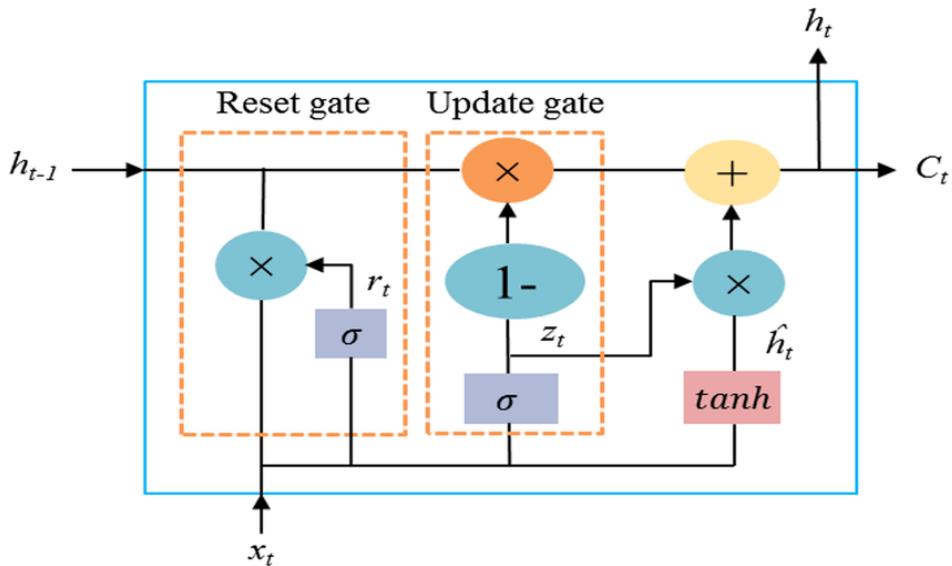
- Potrafi modelować czasowe zależności (czego nie umie MLP / LeNet / ResNet).
- Dobry wybór do zadań z dynamiką, np. gesty, ruch, chodzenie.
- Lekka architektura – prostsza niż LSTM.

#### Ograniczenia:

- Problem zanikającego gradientu przy długich sekwencjach (trudność w uczeniu bardzo długiej pamięci).
- Trudności w uczeniu skomplikowanych, długoterminowych zależności (rozwiązuje to LSTM i GRU).
- Wrażliwy na przeskalowanie danych i jakość sekwencji.

#### 4.3. Co otrzymujemy na wyjściu?

Ostatni stan ukryty (lub agregat wszystkich stanów) przechodzi do warstwy FC. Otrzymujemy wektor prawdopodobieństw dla klas aktywności.



Rysunek 7: Schemat GRU

Źródło: [https://www.researchgate.net/figure/The-Architecture-of-the-gated-recurrent-unit-GRU-cell\\_fig2\\_370683092](https://www.researchgate.net/figure/The-Architecture-of-the-gated-recurrent-unit-GRU-cell_fig2_370683092)

## 0.5 GRU – Gated Recurrent Unit

Ulepszony wariant klasycznego RNN, który wprowadza mechanizm bramek (ang. *gates*) – dzięki temu potrafi lepiej zarządzać przepływem informacji w sekwencji i zapamiętywać ważne stany na dłużej, a ignorować nieistotne.

GRU został zaprojektowany tak, by być lżejszy niż LSTM, ale skuteczniejszy niż klasyczne RNN.

### Architektura:

- **Bramka aktualizacji (update gate):** decyduje, ile z poprzedniego stanu ukrytego należy zachować.
- **Bramka resetująca (reset gate):** decyduje, ile nowej informacji ma wpłynąć na bieżący stan.
- **Nowy stan ukryty:** powstaje na podstawie poprzedniego stanu i aktualnego wejścia, przefiltrowanych przez te bramki.
- **Schemat działania:**  $\text{input\_}_t + \text{hidden\_}(t-1) \rightarrow \text{bramki (update, reset)} \rightarrow \text{hidden\_}_t$

### 5.1. Przetwarzanie danych CSI

Dane wejściowe to tensor  $czas \times cechy$ .

GRU analizuje dane ramka po ramce, ucząc się wzorców zmian sygnału w czasie.

Dzięki bramkom potrafi „zapamiętać” momenty, w których sygnał znacząco się zmienia (np. podczas ruchu osoby).

## 5.2. Podsumowanie modelu

Skutecznie modeluje zależności czasowe, ignorując mniej istotne zmiany.  
Działa efektywniej niż klasyczne RNN – lepiej radzi sobie z dłuższymi sekwencjami.  
GRU może wykryć typowe „sygnatury” ruchów w danych CSI – np. regularne kroki, upadek, szybki obrót.

### Zalety:

- Mniejsze ryzyko zanikania gradientu niż w klasycznym RNN.
- Lżejszy i szybszy niż LSTM.
- Dobry wybór, gdy masz ograniczone zasoby obliczeniowe (np. CPU, ESP32).
- Dobrze radzi sobie z sekwencjami o zmiennej długości.

### Ograniczenia:

- Mniej elastyczny niż LSTM (bo nie ma osobnego stanu komórkowego).
- W niektórych przypadkach (np. bardzo długie zależności) może być mniej skuteczny niż LSTM.
- Nadal potrzebuje starannego przygotowania sekwencji (np. padding, normalizacja).

## 5.3. Co otrzymujemy na wyjściu?

Ostatni stan ukryty (lub agregacja stanów) przechodzi do warstwy FC.  
Model zwraca wektor prawdopodobieństw dla klas aktywności.

## 0.6 LSTM – Long Short-Term Memory

Specjalny rodzaj sieci rekurencyjnej, który został zaprojektowany, aby radzić sobie z problemem zanikającego gradientu.

W odróżnieniu od RNN i GRU, posiada dwa rodzaje pamięci:

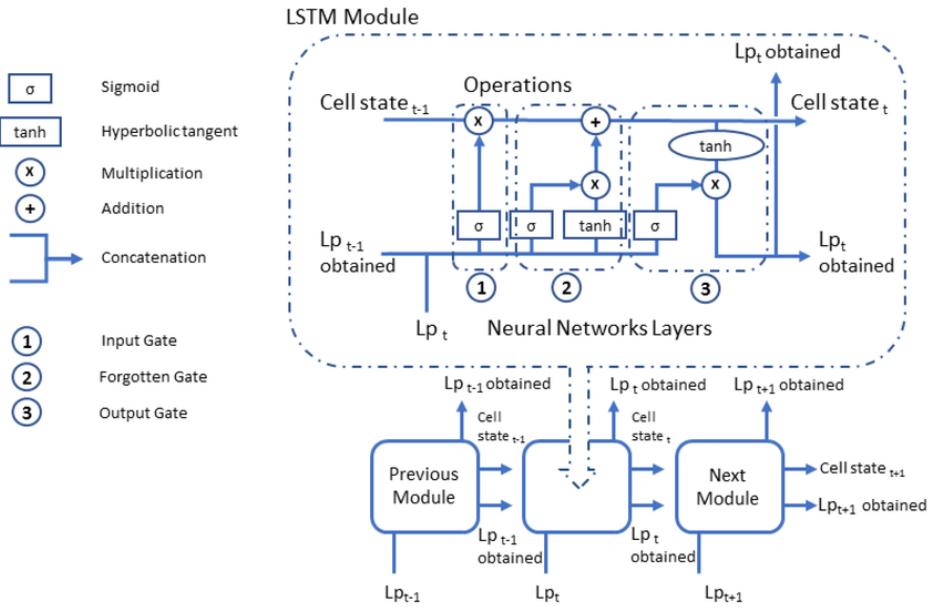
- **Stan ukryty (hidden state)** – krótkoterminowy,
- **Stan komórkowy (cell state)** – długoterminowy.

Dzięki mechanizmowi bramek, LSTM może decydować:

- co zapamiętać,
- co zaktualizować,
- co zapomnieć.

### Architektura:

- **Bramka zapominania (forgotten gate)** – decyduje, które informacje z poprzedniego stanu komórkowego usunąć.



Rysunek 8: Schmeat LSTM

Źródło: [https://www.researchgate.net/figure/General-scheme-of-an-Long-Short-Term-Memory-neural-networks-LSTM-for-Lp-The\\_fig1\\_339120709](https://www.researchgate.net/figure/General-scheme-of-an-Long-Short-Term-Memory-neural-networks-LSTM-for-Lp-The_fig1_339120709)

- **Bramka wejścia (input gate)** – decyduje, ile nowej informacji zapisać do pamięci.
- **Bramka wyjścia (output gate)** – określa, jaka część informacji z pamięci ma zostać przesłana dalej.

## 6.1. Przetwarzanie danych CSI

Wejście to sekwencja:  $czas \times cechy$ .

LSTM uczy się zmian w czasie — np. jak zmienia się amplituda lub faza na przestrzeni wielu ramek.

Dzięki stanowi komórkowemu potrafi uchwycić długie zależności, np. wcześnie ruch, który prowadzi do późniejszego efektu (np. osoba sięga po coś i potem wstaje).

## 6.2. Podsumowanie modelu

Rozpoznaje długie ciągi zmian w czasie – np. sekwencje gestów, powolne ruchy, przejścia między stanami.

Działa dobrze w przypadku złożonych aktywności, które trwają przez wiele ramek.

Zachowuje „świadomość kontekstu”, czego nie potrafi ResNet ani zwykły RNN.

### Zalety:

- Najlepsze właściwości pamięci długoterminowej.
- Idealny do złożonych aktywności z długimi sekwencjami danych.
- Lepsza kontrola nad tym, co „zapamiętać”, a co „zapomnieć”.

### Ograniczenia:

- Cięższy i wolniejszy niż GRU i RNN.
- Więcej parametrów – większe zużycie RAM i GPU.
- Może być „overkill” przy krótkich sekwencjach lub prostych zadaniach.

### 6.3. Co otrzymujemy na wyjściu?

Ostatni stan ukryty (lub sekwencja stanów) → przekazywany do warstwy FC.  
Finalnie otrzymujemy: wektor prawdopodobieństw klas aktywności.

## 0.7 BiLSTM – Bidirectional Long Short-Term Memory

To rozszerzenie klasycznego LSTM, które przetwarza dane w obu kierunkach czasowych:

- „do przodu” – tak jak standardowe LSTM,
- „do tyłu” – od końca sekwencji do początku.

Dzięki temu model ma dostęp zarówno do przeszłości, jak i przyszłości, co pozwala mu lepiej uchwycić kontekst czasowy.

### Architektura:

- dwie warstwy:
  - jedna „czyta” sekwencję od początku,
  - druga „czyta” ją od końca.
- Wyjścia obu warstw są łączone (najczęściej konkatenowane)
- Wewnątrz każdej z tych warstw działa klasyczny LSTM (z forget/input/output gate i cell state).

### 7.1. Przetwarzanie danych CSI

Dane CSI nadal mają formę: *czas × cechy*.

BiLSTM analizuje całą sekwencję od razu, korzystając z informacji o tym, co się już wydażyło i co się dopiero wydarzy.

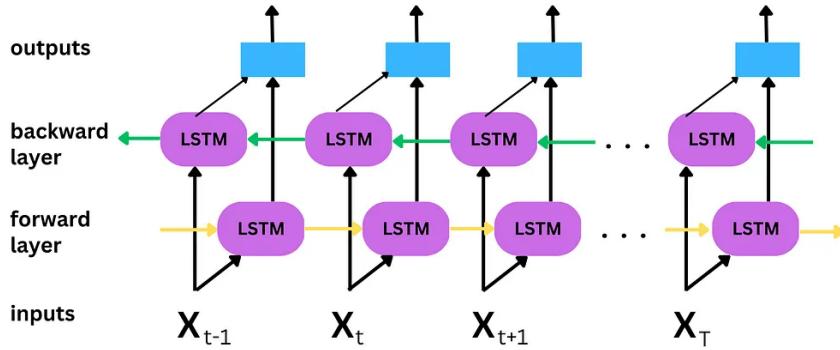
To może być pomocne np. przy detekcji upadków, gdzie nagły ruch na końcu sekwencji może zmienić interpretację tego, co było wcześniej.

### 7.2. Podsumowanie modelu

Model rozpoznaje symetrię ruchów i ich kontekst dwukierunkowy.

Dzięki temu potrafi poprawnie zinterpretować nawet częściowe lub niepełne sekwencje (np. rozpoczęte ruchy).

Działa lepiej niż zwykłe LSTM w przypadkach, gdy:



Rysunek 9: Schemat BiLSTM

Źródło: <https://medium.com/@sourav400.nath/why-is-bilstm-better-than-lstm-a7eb0090c1e4>

- aktywność nie jest jasno odcięta w czasie,
- dane są szumne i kontekst z przeszłości pomaga w klasyfikacji.

#### Zalety:

- Lepsza reprezentacja kontekstu czasowego niż LSTM.
- Wysoka skuteczność przy klasyfikacji sekwencji, w których znaczenie ma cały przebieg sygnału.
- Szczególnie dobre przy rozmytych granicach aktywności (np. przechodzenie z siedzenia do stania).

#### Ograniczenia:

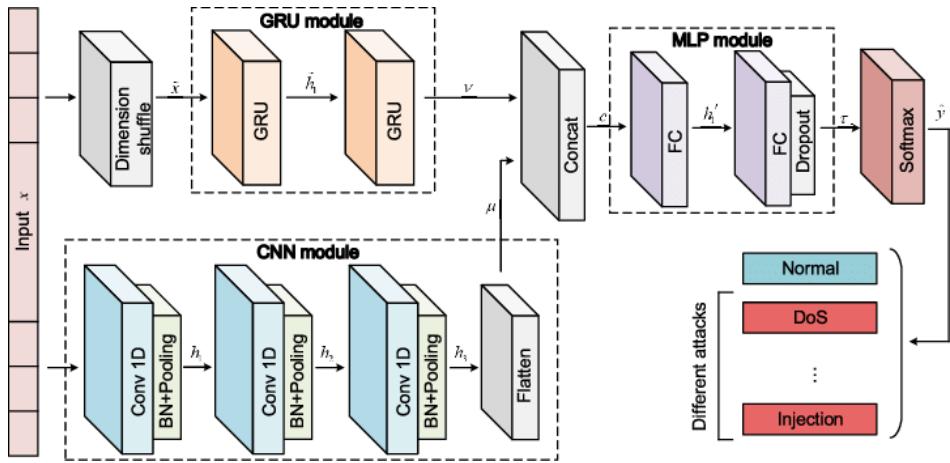
- Jeszcze większa złożoność niż LSTM (dwa razy więcej wag).
- Nie nadaje się do przetwarzania w czasie rzeczywistym – bo wymaga znajomości całej sekwencji z góry.
- Dłuższy czas treningu, więcej pamięci.

### 7.3. Co otrzymujemy na wyjściu?

Ostatni stan ukryty z obu kierunków jest łączony: [hidden\_forward\_T ; hidden\_backward\_1] Przekazywany do warstwy FC → uzyskujemy wektor prawdopodobieństw klas aktywności.

## 0.8 CNN+GRU – model hybrydowy (konwolucyjno-rekurencyjny)

To architektura, która najpierw przetwarza dane CSI przestrzennie (jak obraz) przy pomocy CNN, a następnie przekształca je w sekwencję cech i analizuje tę sekwencję w czasie za



Rysunek 10: Schemat CNN+GRU

Źródło:

[https://www.researchgate.net/figure/Architecture-of-designed-CNN-GRU-model\\_fig3\\_351360140](https://www.researchgate.net/figure/Architecture-of-designed-CNN-GRU-model_fig3_351360140)

pomocą GRU.

### Architektura:

- **Blok CNN:**
  - przetwarza dane w formie 2D,
  - wydobywa cechy lokalne (np. wzory zakłóceń, zakłócenia w określonych częstotliwościach).
- **Warstwa Flatten + Reshape:**
  - dane z CNN są przekształcane do sekwencji (czasowej) → czas × cechy.
- **Blok GRU:**
  - analizuje tę sekwencję cech w czasie (np. jak wzorce z CNN zmieniają się w kolejnych ramkach).
- **Warstwa FC:**
  - klasyfikuje aktywność na podstawie wyjścia z GRU.

### 8.1. Przetwarzanie danych CSI

Dane wejściowe: zazwyczaj tensor  $czas \times anteny \times subnośne$ .

CNN przetwarza dane pojedynczych ramek CSI jako "obrazy", wyciągając cechy z przestrzennego rozkładu sygnału.

Dla każdej ramki uzyskujemy wektor cech. Następnie GRU analizuje te wektory w czasie, ucząc się ich zmian (np. ruch, gesty, przejścia między stanami).

## 8.2. Podsumowanie modelu

- CNN rozpoznaje charakterystyczne wzorce przestrzenne – np. układy zmian między antenami i subnośnymi.
- GRU analizuje dynamikę tych wzorców w czasie – czyli np. czy pasmo przesuwa się, rozszerza, pojawia i znika.
- W połączeniu model potrafi wykrywać ruchy i aktywności, które mają zarówno strukturę w przestrzeni, jak i ciągłość w czasie.

### Zalety:

- Łączy zalety CNN (przestrzeń) i GRU (czas).
- Skuteczny przy analizie złożonych aktywności (np. ruch w przestrzeni, gesty, zmiany pozycji).
- Nadaje się do CSI o dużych rozmiarach – przekształca dane w sekwencje cech o niższej wymiarowości.

### Ograniczenia:

- Większa złożoność niż samodzielne CNN lub GRU.
- Wymaga starannego przygotowania danych (formatowanie ramek, normalizacja).
- Trudniejszy do debugowania – ciężej ocenić, która część (CNN czy GRU) odpowiada za błędy.

## 8.3. Co otrzymujemy na wyjściu?

GRU generuje ostatni stan ukryty (lub agregację stanów).

Ten stan trafia do warstwy FC, która daje wektor prawdopodobieństw klas aktywności.

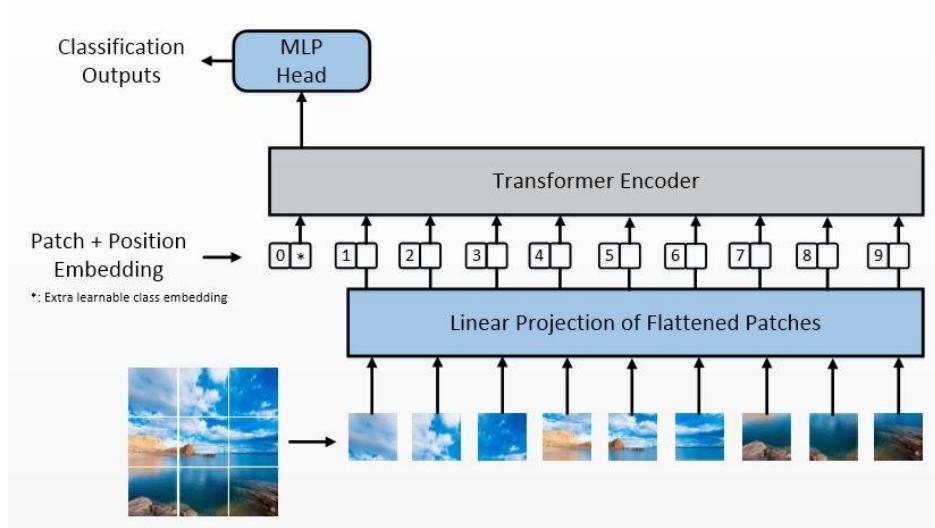
## 0.9 ViT – Vision Transformer

Model oparty na architekturze Transformerów, zaprojektowany pierwotnie do przetwarzania tekstu (np. w NLP), ale zaadaptowany do analizy danych wizualnych – obrazów, a w tym przypadku: danych CSI przekształconych do formy obrazu.

Główna cecha ViT to mechanizm samouwagi (self-attention), który uczestniczy w rozpoznawaniu globalnych zależności – model nie tylko patrzy lokalnie jak CNN, ale też analizuje, jak różne fragmenty danych „wpływają na siebie”.

### Etapy przetwarzania:

- Podział obrazu wejściowego na patch'e (kafelki),
- Każdy patch zamieniany jest na wektor (embedding) przez warstwę liniową,
- Dodawane są informacje o pozycji patcha (positional encoding),



Rysunek 11: Schemat ViT

Źródło: [https://www.researchgate.net/figure/System-Diagram-of-Vision-Transformer-ViT\\_fig2\\_354094782](https://www.researchgate.net/figure/System-Diagram-of-Vision-Transformer-ViT_fig2_354094782)

- **Transformer encoder:**

- składa się z warstw self-attention + feed-forward,
- uczy się zależności między wszystkimi patchami jednocześnie.
- Token klasyfikacyjny (CLS token) jest używany jako reprezentacja całego obrazu,
- Warstwa FC przewiduje klasę aktywności.

## 9.1. Przetwarzanie danych CSI

Dane CSI są traktowane jako obraz 2D.

Ten „obraz CSI” jest dzielony na kafelki (patch'e).

Model analizuje te kafelki, uczestnicząc w globalnym dopasowaniu cech – np. zmiana sygnału w jednym miejscu może być powiązana z odległą zmianą w innym miejscu.

Świętne sprawdza się przy globalnych wzorcach w danych CSI, niekoniecznie lokalnych (jak w CNN).

## 9.2. Podsumowanie modelu

Uczy się globalnych zależności: np. związek między zmianą w sygnale z jednej anteny a zmianą 200 ms później.

Nie potrzebuje konwolucji – rozumie dane jako całość.

Świętne działa, jeśli dane mają złożony i niejednolity układ wzorców, a nie tylko lokalne „plamki”.

**Zalety:**

- Bardzo dobra reprezentacja globalnych zależności w danych CSI.

- Brak założeń lokalności – model „widzi” wszystko naraz.
- Działa świetnie na dużych zbiorach danych z bogatymi cechami.

#### **Ograniczenia:**

- Wysokie wymagania obliczeniowe – dużo pamięci, długi czas treningu.
- Słabsze działanie na małych zbiorach (potrzebuje dużych danych do pełnej skuteczności).
- Trudniejszy w interpretacji (tzw. „czarna skrzynka”).

#### **9.3. Co otrzymujemy na wyjściu?**

Po przetworzeniu wszystkich patchy i warstw self-attention:

Token CLS zawiera reprezentację całego wejścia.

Token ten trafia do warstwy FC → wynik to wektor prawdopodobieństw klas aktywności.

### 3. Kursy

#### 1. MLP

- <https://www.youtube.com/watch?v=u5GAVdLQvIg>
- <https://www.youtube.com/watch?v=llmNhFxre0w>
- <https://cs231n.github.io/neural-networks-1/>

#### 2. LeNet

- <https://www.youtube.com/watch?v=PcGCpxstTCg>
- <https://www.youtube.com/watch?v=m3BrTjo2zUA>

#### 3. ResNet

- <https://www.youtube.com/watch?v=w1UsKanMatM>
- <https://www.youtube.com/watch?v=Q1JCrG1bJ-A>
- <https://www.youtube.com/watch?v=o3mboe1jYI>

#### 4. RNN

- <https://www.youtube.com/watch?v=Gafik7w1i8>

#### 5. GRU

- <https://www.youtube.com/watch?v=8HyCNIVRbSU>
- <https://www.youtube.com/watch?v=hkJNjxIEO-s>
- <https://www.youtube.com/watch?v=rdz0UqQz5Sw>

#### 6. LSTM

- <https://www.youtube.com/watch?v=rmxogwljOhE>
- <https://www.youtube.com/watch?v=b61DPVFX03I>
- <https://www.youtube.com/watch?v=guqgmVqcy2c>
- [https://www.youtube.com/watch?v=xW\\_SRRHZLAU](https://www.youtube.com/watch?v=xW_SRRHZLAU)

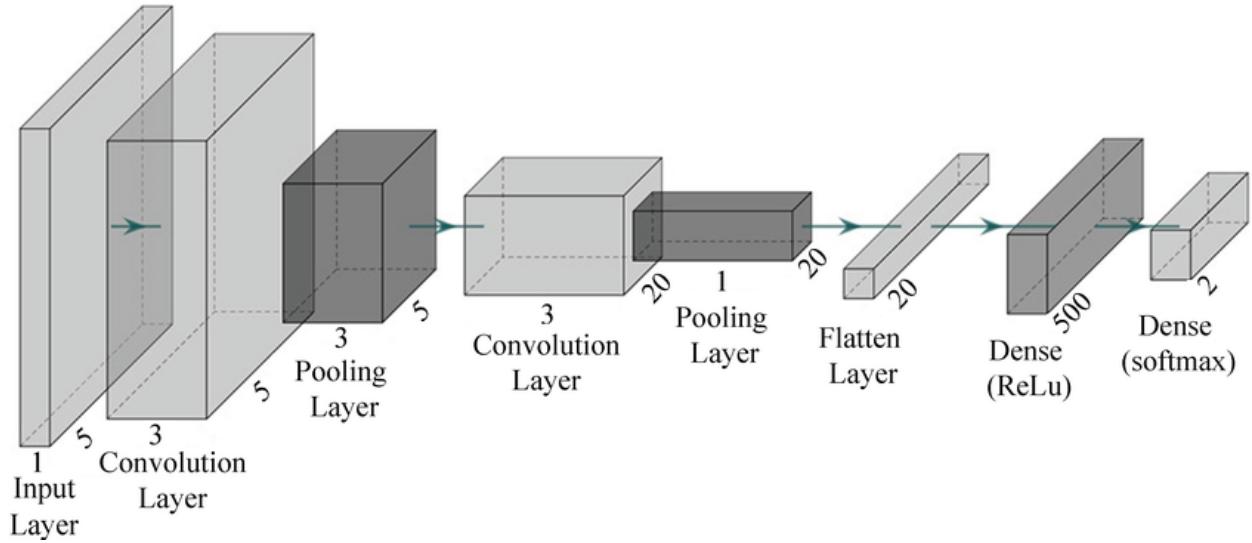
#### 7. BiLSTM

- <https://www.youtube.com/watch?v=k2NSm3MNdyg>

#### 8. ViT

- <https://www.youtube.com/watch?v=vsqKGZT8Qn8>
- <https://www.youtube.com/watch?v=TrdevFKam4t640s>

# Uruchamianie LeNet na ESP32-WiFi-Sensing



Rysunek 12: Ogólny schemat architektury sieci konwolucyjnej (CNN).

Poniżej znajduje się kod analizowanego modelu LeNet, dostosowanego do danych wejściowych.

```
[] import torch.nn as nn

class UT_HAR_LeNet(nn.Module):
 def __init__(self):
 super(UT_HAR_LeNet, self).__init__()
 # input size: (1, 200, 66)
 self.encoder = nn.Sequential(
 nn.Conv2d(1, 32, kernel_size=(7, 3), stride=(3, 1)), # → (32, 65, 64)
 nn.ReLU(True),
 nn.MaxPool2d(2), # → (32, 32, 32)
 nn.Conv2d(32, 64, kernel_size=(5, 4), stride=(2, 2), padding=(1, 0)), # → (64, 15, 15)
 nn.ReLU(True),
 nn.MaxPool2d(2), # → (64, 7, 7)
 nn.Conv2d(64, 96, kernel_size=(3, 3), stride=1), # → (96, 5, 5)
 nn.ReLU(True),
 nn.MaxPool2d(2) # → (96, 2, 2)
)
 self.fc = nn.Sequential(
 nn.Linear(96 * 2 * 2, 128),
 nn.ReLU(),
 nn.Linear(128, 7) # 7 klas
)

 def forward(self, x):
 x = self.encoder(x)
 x = x.view(-1, 96 * 2 * 2) # flatten
 x = self.fc(x)
 return x
```

Rysunek 13: Definicja klasy modelu UT\_HAR\_LeNet w PyTorch.

## Jak dostosować kod do nowych danych?

Stan początkowy: dane to "obrazek" o wymiarach 200 pikseli wysokości i 66 pikseli szerokości, z jednym kanałem (amplituda).

### 1. Pierwsza warstwa Conv2d: `nn.Conv2d(1, 32, kernel_size=(7,3), stride=(3,1))`

Sieć bierze fragmenty  $7 \times 3$  pikseli (`kernel_size`) i przesuwa się co 3 piksele w dół i 1 w bok (`stride`).

#### Obliczanie wymiaru wyjściowego:

- Nowa wysokość:  $(200 - 7)/3 + 1 \approx 65$
- Nowa szerokość:  $(66 - 3)/1 + 1 = 64$

**Wynik:** Oryginalna informacja została przetworzona na 32 nowe, mniejsze (65x64) zestawy danych, z których każdy reprezentuje inną cechę wykrytą przez jeden z 32 filtrów.

### 2. Pierwsza warstwa MaxPool2d: `nn.MaxPool2d(2)`

Zapis `nn.MaxPool2d(2)` to skrót, który oznacza, że sieć analizuje dane w oknach o rozmiarze  $2 \times 2$  (`kernel_size=2`) i z każdego takiego okna wybiera jedną, największą wartość - wynika to z domyślnego zachowania biblioteki PyTorch. Jeśli ręcznie nie ustawimy kroku (`stride`), PyTorch automatycznie przyjmuje, że krok jest równy wielkości okna. W naszym przypadku `stride` wynosi 2. Połączenie okna  $2 \times 2$  i kroku 2 (przesuwanie okna co 2 piksele, bez nakładania się) jest tym, co powoduje zmniejszenie wymiarów o połowę.

**Obliczenia:** Zamiast prostego "zaokrąglania w dół", PyTorch oblicza nowy wymiar według wzoru  $nowy\_wymiar = \lfloor (stary\_wymiar - kernel\_size) / stride + 1 \rfloor$ . Dlatego, gdy na wejściu mamy wymiary 65x64:

- Nowa wysokość:  $\lfloor (65 - 2)/2 + 1 \rfloor = \lfloor 32.5 \rfloor = 32$
- Nowa szerokość:  $\lfloor (64 - 2)/2 + 1 \rfloor = \lfloor 32 \rfloor = 32$

### 3. ...i tak dalej przez kolejne warstwy konwolucyjne i pooling, każda z nich dalej zmniejsza wymiary wysokości $\times$ szerokość.

### 4. Hiperparametry, które będziemy musieli dostroić do naszych danych przed trenowaniem modelu

- **Liczba filtrów (out\_channels):** Decyduje o "głębokości" analizy - ile różnych cech sieć ma szukać na danym etapie.
- **Rozmiar filtra (kernel\_size):** Określa wielkość "okna", przez które sieć patrzy na dane. Większe filtry wychwytują bardziej globalne wzorce, mniejsze - lokalne detale.
- **Stride:** Kontroluje, jak szczegółowo sieć skanuje dane i jak bardzo zmniejsza ich wymiary.

- **Padding:** Dodaje "margines" wokół danych, co pozwala lepiej kontrolować wymiary wyjściowe i analizować piksele na krawędziach.
- **Liczba neuronów w warstwie ukrytej:** Określa "pojemność" warstwy, która łączy cechy w całość przed podjęciem ostatecznej decyzji. Większa liczba neuronów pozwala na naukę bardziej złożonych kombinacji cech, ale zwiększa też ryzyko przeuczenia.

## 5. Ostatnia warstwa MaxPool2d

Po ostatniej warstwie MaxPool2d, dane mają wymiary wysokość=2, szerokość=2 i 96 kanałów.

## 6. Spłaszczenie i Warstwa Liniowa (nn.Linear)

Stary kod: `nn.Linear(96*4*4, 128)`

Nowy kod: `nn.Linear(96*2*2, 128)`

Zmiana była konieczna, ponieważ warstwa `nn.Linear` nie rozumie "obrazków". Oczekuje jednowymiarowej listy cech. Musimy więc wziąć nasz trójwymiarowy klocek o wymiarach  $96 \times 2 \times 2$  i "spłaszczyć" go do wektora.

**Operacja `x.view(-1, 96*2*2)`:** Jest to operacja flatteningu. Bierzemy tensor x. Zmieniamy jego kształt (`.view`) na dwuwymiarowy. Drugi wymiar ma mieć rozmiar dokładnie  $96 \times 2 \times 2$ , czyli 384. Pierwszy wymiar jako -1 jest sygnałem dla PyTorch, aby został obliczony automatycznie tak, żeby wszystko się zgadzało.

## 7. Wnioski

Dane wejściowe (200 x 66) po przejściu przez sieć dały na końcu "obraz cech" o wymiarach  $96 \times 2 \times 2$ , więc po spłaszczeniu otrzymaliśmy 384 cechy.

## 8. Warstwa Wyjściowa i Liczba Klas

Ostatnia warstwa modelu, `nn.Linear(128, 7)`, ma na wyjściu 7 neuronów. Wynika to z tego, że nasz zbiór danych zawiera 7 różnych aktywności do rozpoznania, więc model musi mieć na wyjściu dokładnie 7 miejsc na odpowiedzi.

# Badanie wydajności na różnych kodach

## 1. Kod ze statycznego repozytorium z danymi z tego repozytorium (<https://github.com/xyanchen/WiFi-CSI-Sensing-Benchmark>)

Zbiór danych: UT\_HAR\_dataset -> link do dysku:

[https://drive.google.com/drive/folders/1R0R8S1VbLI1iUFQCzh\\_mH90H\\_4CW2iwt](https://drive.google.com/drive/folders/1R0R8S1VbLI1iUFQCzh_mH90H_4CW2iwt)

Struktura pliku UT\_HAR:

```
UT_HAR/
 .ipynb_checkpoints/ (folder systemowy Colab, można ignorować)

 data/ (folder zawierający dane wejściowe)
 X_train.csv
 X_test.csv
 X_val.csv (zbiór walidacyjny, którego na razie nie używamy)

 label/ (folder zawierający odpowiadające etykiety)
 y_train.csv
 y_test.csv
 y_val.csv
```

**Format plików:** Mimo rozszerzenia .csv, pliki te nie są plikami tekstowymi. Są to pliki binarne w formacie NumPy. Dlatego do ich wczytania musimy używać funkcji np.load(), a nie pd.read\_csv() czy np.loadtxt().

**Wymiary danych (x):**

- Każdy wiersz w plikach X\_... to spłaszczony "obraz" CSI.
- Po wczytaniu i przekształceniu, każda pojedyncza próbka ma wymiar 250x90 (250 próbek w czasie, 90 podnośnych).
- Dla modelu Keras/TensorFlow musimy dodać wymiar kanału, więc ostateczny kształt jednej próbki to (250, 90, 1).

**Etykiety (y):**

- Pliki y\_... zawierają etykiety dla odpowiadających im danych X\_....
- Są to liczby całkowite w zakresie od 1 do 7, gdzie każda liczba reprezentuje jedną z 7 aktywności (lie down, fall, pick up, run, sit down, stand up, walk).
- Przed podaniem ich do modelu musimy je przekonwertować na zakres od 0 do 6, ponieważ tak działają funkcje straty w bibliotekach uczenia maszynowego.

Pozy: lie down, fall, walk, pickup, run, sit down, stand up

Wydajność: 98,60%

Link do Colaba:

```

Epoka 25/50 zakończona. Dokładność na teście: 97.40%
Epoka 26/50 zakończona. Dokładność na teście: 97.60%
Epoka 27/50 zakończona. Dokładność na teście: 98.60%
Epoka 28/50 zakończona. Dokładność na teście: 98.40%
Epoka 29/50 zakończona. Dokładność na teście: 98.60%
Epoka 30/50 zakończona. Dokładność na teście: 98.60%
Epoka 31/50 zakończona. Dokładność na teście: 98.60%
-
Epoka 32/50 zakończona. Dokładność na teście: 98.60%
Epoka 33/50 zakończona. Dokładność na teście: 98.60%
Epoka 34/50 zakończona. Dokładność na teście: 98.60%
Epoka 35/50 zakończona. Dokładność na teście: 98.60%
Epoka 36/50 zakończona. Dokładność na teście: 98.60%
Epoka 37/50 zakończona. Dokładność na teście: 98.60%
Epoka 38/50 zakończona. Dokładność na teście: 98.60%
Epoka 39/50 zakończona. Dokładność na teście: 98.60%
Epoka 40/50 zakończona. Dokładność na teście: 98.60%
Epoka 41/50 zakończona. Dokładność na teście: 98.60%
Epoka 42/50 zakończona. Dokładność na teście: 98.60%
Epoka 43/50 zakończona. Dokładność na teście: 98.60%
Epoka 44/50 zakończona. Dokładność na teście: 98.60%
Epoka 45/50 zakończona. Dokładność na teście: 98.60%
Epoka 46/50 zakończona. Dokładność na teście: 98.60%
Epoka 47/50 zakończona. Dokładność na teście: 98.60%
Epoka 48/50 zakończona. Dokładność na teście: 98.60%
Epoka 49/50 zakończona. Dokładność na teście: 98.60%
Epoka 50/50 zakończona. Dokładność na teście: 98.60%
-
Trening zakończony!

OSTATECZNA DOKŁADNOŚĆ NA ZBIORZE TESTOWYM: 98.60 %

```

Rysunek 14: Wydajność dla modelu z repozytorium statycznego z danymi z tego repozytorium

[https://colab.research.google.com/drive/1RkP5ylXoX9AV4a2p6bNTDcmwSZu\\_gsB6#scrollTo=E1BN9HB1p3Bx](https://colab.research.google.com/drive/1RkP5ylXoX9AV4a2p6bNTDcmwSZu_gsB6#scrollTo=E1BN9HB1p3Bx)

## 2. Kod z dynamicznego repozytorium (<https://github.com/thu4n/ESP32-WiFi-Sensing>) z danymi ze statycznego repozytorium (<https://github.com/xyanchen/WiFi-CSI-Sensing-Benchmark>)

Zbiór danych: UT\_HAR\_dataset

Pozy: lie down, fall, walk, pickup, run, sit down, stand up

Prosty model CNN:

### Działanie:

- `keras.layers.Conv2D(32, 7, activation='relu', input_shape=(250, 90, 1))`
  - pierwsza warstwa splotowa
  - używamy 32 równych filtrów
  - każdy filtr ma rozmiar 7x7
  - funkcja aktywacji “ReLU”:  $\text{ReLU}(x) = \max(0, x)$ 
    - \* Dla każdej wartości wejściowej  $x$ , ReLU zwraca 0, jeśli  $x < 0$ ,

```

import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.layers import Dense, Dropout, Attention
from tensorflow.keras.losses import SparseCategoricalCrossentropy

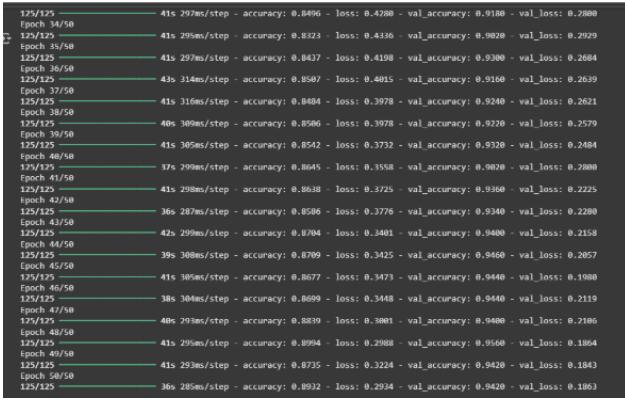
Define the CNN model
model = keras.Sequential([
 keras.layers.Conv2D(32, 7, activation='relu', input_shape=(200, 53,1)),
 keras.layers.MaxPooling2D(2),
 keras.layers.Conv2D(96, 5, activation='relu'),
 keras.layers.MaxPooling2D(2),
 keras.layers.Flatten(),
 keras.layers.Dense(128, activation='relu'),
 keras.layers.Dropout(0.5),
 keras.layers.Dense(64, activation='relu'),
 keras.layers.Dropout(0.25),
 keras.layers.Dense(7, activation='softmax')
])

Compile the model
model.compile(optimizer= tf.keras.optimizers.Adam(learning_rate=0.0001),
 loss='sparse_categorical_crossentropy',
 metrics=['accuracy'])
model.summary()

```

Rysunek 15: Kod prostego modelu CNN

- \* Jeśli  $x > 0$ , zwraca po prostu  $x$ .
- Wynik: 32 "mapy cech" (feature maps), które pokazują, gdzie na obrazie wejściowym zostały znalezione poszczególne cechy.
- `keras.layers.MaxPooling2D(2)`
  - warstwa poolingowa, zmniejsza wysokość i szerokość map cech, przy zachowaniu najważniejszych informacji → Dzieli każdą mapę na małe kwadraty  $2 \times 2$  i z każdego kwadratu wybiera tylko największą wartość.
- `keras.layers.Conv2D(96, 5, activation='relu')`
  - druga warstwa splotowa
  - Używa 96 filtrów. Są one bardziej zaawansowane i uczą się łączyć proste cechy z pierwszej warstwy w bardziej skomplikowane wzorce.
  - Filtry mają rozmiar  $5 \times 5$ .
- `keras.layers.Flatten()`
  - "Spłaszcza" wszystkie mapy cech do postaci jednego, długiego wektora liczb.
- `keras.layers.Dense(128, activation='relu')`
  - Pierwsza warstwa w pełni połączona. Każdy z 128 neuronów w tej warstwie jest połączony z każdą wartością ze spłaszczonego wektora.
  - Uczy się jeszcze bardziej złożonych kombinacji cech z całego "obrazu". To tutaj model zaczyna "rozumieć", jak różne cechy razem tworzą konkretną aktywność.
- `keras.layers.Dropout(0.5)`



Rysunek 16: Wydajność dla modelu z dynamicznego repozytorium z danymi ze statycznego repozytorium

- Podczas każdej iteracji treningu, losowo "wyłącza" 50% neuronów z poprzedniej warstwy.
- Zapobiega overfittingowi.
- `keras.layers.Dense(64, activation='relu')` i `Dropout(0.25)`
  - Kolejna, mniejsza warstwa w pełni połączona i mniejszy dropout.
- `keras.layers.Dense(7, activation='softmax')`
  - Warstwa wyjściowa. Ma 7 neuronów - po jednym dla każdej klasy.
  - Funkcja aktywacji Softmax - zamienia wyjścia neuronów na prawdopodobieństwa. Suma wszystkich 7 wartości zawsze będzie wynosić 1. Neuron z najwyższą wartością wskazuje na najbardziej prawdopodobną klasę.

Link do Colaba:

[https://colab.research.google.com/drive/1xuEMFTTxW8znFa9MBiI8Ttni3\\_Xn2wTX#scrollTo=LX6Df0Z7uvZu](https://colab.research.google.com/drive/1xuEMFTTxW8znFa9MBiI8Ttni3_Xn2wTX#scrollTo=LX6Df0Z7uvZu)

Najlepszy wynik: 95,6%

### 3. Użycie modelu z przypadku statycznego przepisanego na potrzeby przypadku dynamicznego z danymi z repozytorium dynamicznego

Zbiór danych: 01-tvat-raw -> link:

<https://github.com/thu4n/ESP32-WiFi-Sensing/tree/master/datasets>

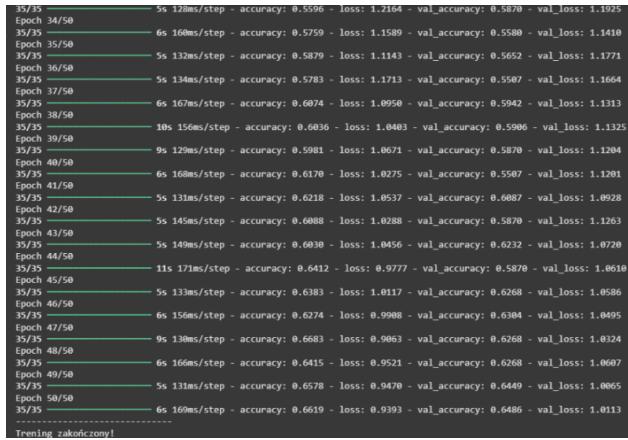
Pozy: JJ - Jumping Jacks, LA - Left Arm, LL - Left Leg, NA - No Activity, RA - Right Arm, RL - Right Leg, SO - Standing Object

Link do Colaba:

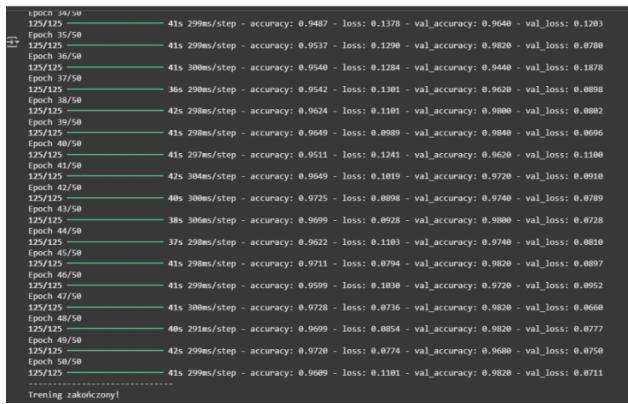
<https://colab.research.google.com/drive/1KZ06KwnIRcpaIzlsNDaC7yD716pbB70t#scrollTo=FFiYntGPzQzQ>

Najlepszy wynik: 64,86%

### 4. Użycie modelu z przypadku statycznego przepisanego na potrzeby przypadku



Rysunek 17: Wydajność dla modelu z przypadku statycznego przepisanego na potrzeby przypadku dynamicznego z danymi z repozytorium dynamicznego



Rysunek 18: Wydajność dla modelu z przypadku statycznego przepisanego na potrzeby przypadku dynamicznego z danymi z repozytorium statycznego

## dynamicznego z danymi z repozytorium statycznego

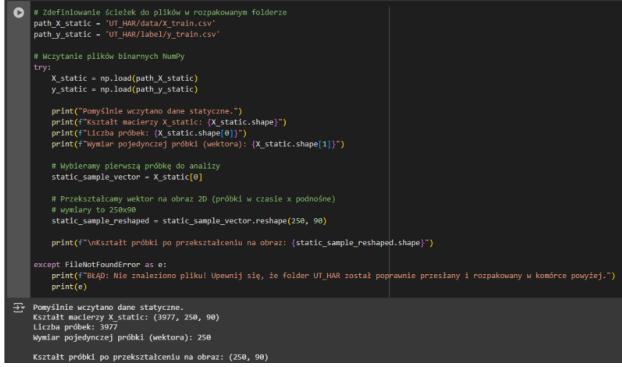
Zbiór danych: UT\_HAR\_dataset

Pozy: lie down, fall, walk, pickup, run, sit down, stand up

Najlepszy wynik: 98,4%

Link do Colaba:

<https://colab.research.google.com/drive/1S0QqTH9iqeqI0l0AeXthXv6YzeBUWcjJ#scrollTo=FFiYntGPzQzQ>



```

Zdefiniowanie ścieżek do plików w rozpakowanym folderze
path_x_static = 'Uf_HAR/data/x_train.csv'
path_y_static = 'Uf_HAR/label/y_train.csv'

Wczytanie plików binarnych NumPy
try:
 X_static = np.loadtxt(path_x_static)
 y_static = np.loadtxt(path_y_static)

 print("Pomyślnie wczytano dane statyczne.")
 print("Kształt macierzy X_static: (X_static.shape[0], X_static.shape[1])")
 print("Liczba próbki: (X_static.shape[0])")
 print("Wysokość pojedynczej próbki (wektora): (X_static.shape[1])")

 # Wyberemy pierwszą próbkę do analizy
 static_sample_vector = X_static[0]

 # Przekształcamy wektor na obraz 2D (próbki w czasie x podnośnie)
 # wymiary to 250x90
 static_sample_reshaped = static_sample_vector.reshape(250, 90)

 print("Unkształtuję próbki po przekształceniu na obraz: (static_sample_reshaped.shape)")

except FileNotFoundError as e:
 print("UfHAR: Nie znaleziono pliku! Upewnij się, że folder Uf_HAR został poprawnie przesunięty i rozpakowany w komórce powyżej.")
 print(e)

print("Pomyślnie wczytano dane statyczne.
Kształt macierzy X_static: (3997, 250, 90)
Liczba próbki: 3997
Wysokość pojedynczej próbki (wektora): 250
Kształt próbki po przekształceniu na obraz: (250, 90)
")

```

Rysunek 19: Dane statyczne

## Porównanie zbiorów danych

Link do Colaba analitycznego:

<https://colab.research.google.com/drive/1nCZ9pBFpvAtBoyHwMSVssIiVD9Mfx091#scrollTo=AX6NKAfyhEAF>

## 1. Format danych i przetwarzanie wstępne

### Zbiór statyczny:

- Pliki z tego repozytorium pomimo rozszerzenia .csv, są plikami w formacie NumPy.
- Dane są wstępnie przetworzone i zapisane jako gotowe macierze. Jak pokazała analiza, każda próbka jest już ukształtowana w formę obrazu o wymiarach 250 próbek w czasie na 90 podnośnych.

### Zbiór dynamiczny:

- Dane to surowe pliki tekstowe .csv.
- Każdy wiersz reprezentuje pojedynczy pakiet WiFi i zawiera 26 kolumn z metadanymi.
- Kluczowe informacje o kanale (CSI) znajdują się w kolumnie CSI\_DATA jako string, który zawiera przeplatane wartości urojone i rzeczywiste na 64 podnośnych.
- Aby te dane stały się użyteczne, wymagają następującego procesu:
  1. Parsowanie tekstu w celu wyodrębnienia liczb.
  2. Obliczenie amplitudy z wartości urojonych i rzeczywistych.
  3. Zastosowanie filtrów w celu usunięcia szumu.
  4. Segmentację długich serii czasowych na mniejsze próbki.

| --- Surowe dane (pierwsze 5 wierszy) --- |                                                                   |              |                   |               |                   |                 |         |           |   |  |
|------------------------------------------|-------------------------------------------------------------------|--------------|-------------------|---------------|-------------------|-----------------|---------|-----------|---|--|
|                                          | type                                                              | role         | mac               | rssi          | rate              | sig_mode        | mcs     | bandwidth | \ |  |
| 0                                        | CSI_DATA                                                          | AP           | C8:F0:9E:F2:C2:EC | -72           | 11                | 0               | 0       | 0         |   |  |
| 1                                        | CSI_DATA                                                          | AP           | C8:F0:9E:F2:C2:EC | -72           | 11                | 0               | 0       | 0         |   |  |
| 2                                        | CSI_DATA                                                          | AP           | C8:F0:9E:F2:C2:EC | -72           | 11                | 0               | 0       | 0         |   |  |
| 3                                        | CSI_DATA                                                          | AP           | C8:F0:9E:F2:C2:EC | -75           | 11                | 0               | 0       | 0         |   |  |
| 4                                        | CSI_DATA                                                          | AP           | C8:F0:9E:F2:C2:EC | -77           | 11                | 0               | 0       | 0         |   |  |
|                                          | smoothing                                                         | not_sounding | ...               | channel       | secondary_channel | local_timestamp |         |           | \ |  |
| 0                                        | 0                                                                 | 0            | ...               | 6             |                   | 1               | 8346021 |           |   |  |
| 1                                        | 0                                                                 | 0            | ...               | 6             |                   | 1               | 8346426 |           |   |  |
| 2                                        | 0                                                                 | 0            | ...               | 6             |                   | 1               | 8375326 |           |   |  |
| 3                                        | 0                                                                 | 0            | ...               | 6             |                   | 1               | 8397104 |           |   |  |
| 4                                        | 0                                                                 | 0            | ...               | 6             |                   | 1               | 8434918 |           |   |  |
|                                          | ant                                                               | sig_len      | rx_state          | real_time_set | real_timestamp    | len             |         |           | \ |  |
| 0                                        | 0                                                                 | 28           | 0                 | 0             | 8.50623           | 128             |         |           |   |  |
| 1                                        | 0                                                                 | 157          | 0                 | 0             | 8.51344           | 128             |         |           |   |  |
| 2                                        | 0                                                                 | 135          | 0                 | 0             | 8.53536           | 128             |         |           |   |  |
| 3                                        | 0                                                                 | 390          | 0                 | 0             | 8.55748           | 128             |         |           |   |  |
| 4                                        | 0                                                                 | 390          | 0                 | 0             | 8.59521           | 128             |         |           |   |  |
|                                          | CSI_DATA                                                          |              |                   |               |                   |                 |         |           |   |  |
| 0                                        | [28 -64 1 0 0 0 0 0 0 0 0 0 -9 -9 -9 -13 -8 -12 -7 -15 -7 -12 -10 |              |                   |               |                   |                 |         |           |   |  |
| 1                                        | [-99 -48 9 0 0 0 0 0 0 0 0 0 -14 -1 -16 -7 -18 ...                |              |                   |               |                   |                 |         |           |   |  |
| 2                                        | [-121 112 8 0 0 0 0 0 0 0 0 -1 7 0 8 -2 10 - ...                  |              |                   |               |                   |                 |         |           |   |  |
| 3                                        | [-122 -31 23 0 0 0 0 0 0 0 0 2 -14 6 -14 4 - ...                  |              |                   |               |                   |                 |         |           |   |  |
| 4                                        | [-122 -31 23 0 0 0 0 0 0 0 0 8 27 12 31 8 31 ...                  |              |                   |               |                   |                 |         |           |   |  |

[5 rows x 26 columns]

Rysunek 20: Dane dynamiczne.

— Przykładowy string *CSI\_DATA* —

```
[28 -64 1 0 0 0 0 0 0 0 0 0 -9 -9 -9 -13 -8 -12 -7 -15 -7 -12 -10
-12 -8 -13 -12 -13 -11 -10 -11 -13 -12 -13 -15 -12 -14 -12 -13 -11 -13
-11 -16 -14 -16 -11 -15 -10 -15 -13 -16 -8 -13 -12 -17 -8 -16 -11 -18
-10 -15 -9 -16 0 0 -10 -17 -10 -16 -8 -15 -12 -14 -8 -13 -10 -14 -6 -14
-8 -13 -9 -13 -7 -14 -9 -12 -5 -11 -8 -12 -9 -11 -5 -10 -6 -10 -6 -9 -6
-10 -9 -8 -7 -8 -9 -9 -8 -6 -7 -5 -8 -6 -10 -5 -10 -6 0 0 0 0 0 0 0 0 0]
```

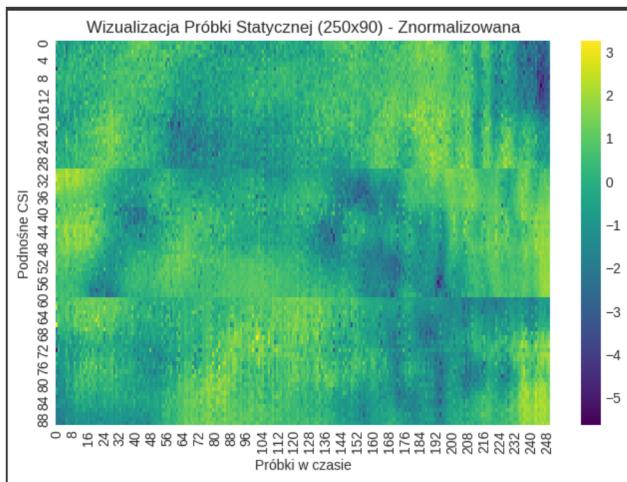
## 2. Różnica w charakterystyce i poziomie szumu sygnału

Zbiór statyczny:

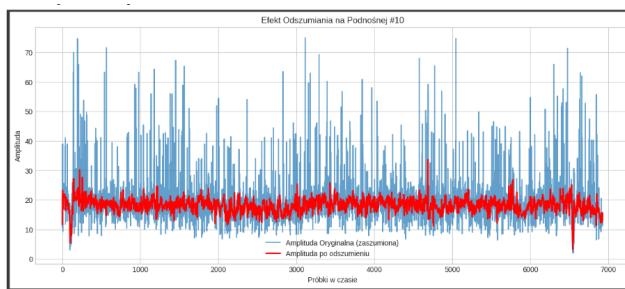
- Wizualizacja próbki w postaci mapy ciepła pokazuje sygnał stosunkowo gładki, o niskiej wariancji, co sugeruje, że dane zostały zebrane w kontrolowanych warunkach.

Zbiór dynamiczny:

- Oryginalny sygnał amplitudy charakteryzuje się dużą ilością pików oraz wysoką wariancją (niebieska linia rozciąga się w pionie od wartości bliskich 0 aż do ponad 70), dlatego zastosowanie filtrów (Hampela i Savitzky'ego-Golaya) jest krokiem niezbędnym do usunięcia tych zakłóceń (czerwona linia jest znacznie bardziej ściśnięta i oscyluje głównie w zakresie od 10 do 25).



Rysunek 21: Wizualizacja próbki statycznej.



Rysunek 22: Efekt odszumiania na podnośnej.

```

[12] # Bierzymy pierwszy segment o długości 200 z odszumionych danych
segment_length = 200
dynamic_segment_raw = denoised_df.iloc[:segment_length]

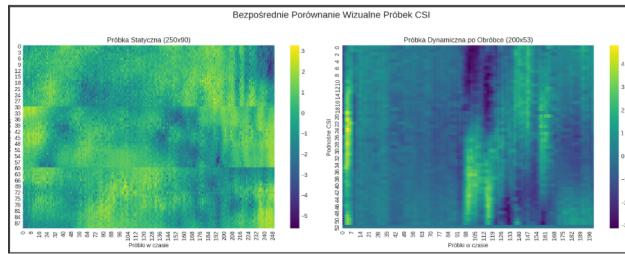
Kolumny do usunięcia (z notebooka)
columns_to_drop = [0, 1, 2, 3, 4, 5, 32, 59, 60, 61, 62]
dynamic_sample = dynamic_segment_raw.drop(columns=columns_to_drop)

print(f"Kształt próbki dynamicznej po segmentacji i usunięciu kolumn: {dynamic_sample.shape}")

→ Kształt próbki dynamicznej po segmentacji i usunięciu kolumn: (200, 53)

```

Rysunek 23: Próbki ze zbioru dynamicznego.



Rysunek 24: Bezpośrednie porównanie wizualne próbek CSI

### 3. Różnica w wymiarach i strukturze próbek

#### Wymiary:

- Próbki ze zbioru statycznego mają wymiary 250x90 (250 próbek w czasie na 90 podnośnych).
- Próbki ze zbioru dynamicznego po przetworzeniu mają wymiary 200x53 (200 próbek w czasie na 53 podnośne).

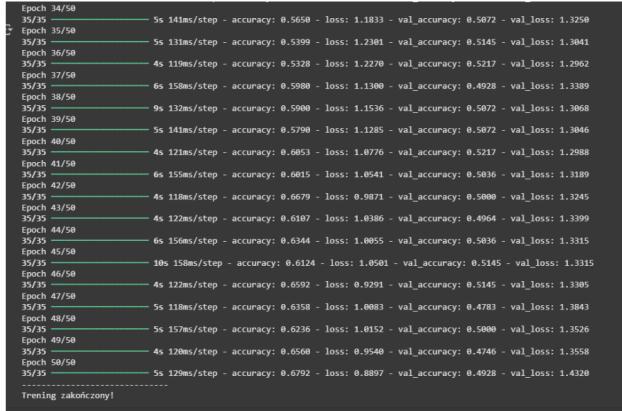
#### Struktura:

- **Próbka statyczna:** Wzorce w postaci poziomych pasm, co sugeruje, że pewne podnośne mają naturalnie inną charakterystykę energetyczną. Zmiany związane z aktywnością będą subtelnymi wariancjami wewnętrz tych pasm.
- **Próbka dynamiczna:** Widoczne bardzo wyraźnie pionowe pasma, co oznacza, że wykonywana czynność generuje silny sygnał w konkretnym momencie, który jest widoczny na większości podnośnych równocześnie.

### 4. Podsumowanie wniosków

Model uczony na danych statycznych uczy się rozpoznawać niewielkie, delikatne zakłócenia pojawiające się na tle regularnych, poziomych linii. To właśnie rozmieszczenie tych subtelnich zmian pozwala mu określić, jaka aktywność została zarejestrowana.

W przypadku danych dynamicznych model powinien skupić się na dużych, nagłych zmianach, które pojawiają się jako pionowe bloki. Ich kształt i długość trwania niosą kluczowe informacje o tym, co się dzieje.



Rysunek 25: Wyniki bez odszumiania.

## Wnioski końcowe:

- Głównym problemem nie jest sam szum. Chociaż odszumianie jest kluczowe, nie jest jedynym i najważniejszym czynnikiem wpływającym na niską skuteczność.
- Architektura sieci splotowej jest wrażliwa na kształt danych wejściowych, ponieważ rozmiar filtrów (`kernel_size`) i operacji poolingowych (`MaxPooling`) jest zdefiniowany w sposób absolutny. Model referencyjny, który osiągał wysoką skuteczność, był zoptymalizowany dla większych obrazów (250x90). Próba zastosowania podobnej architektury do mniejszych próbek (200x53) bez odpowiedniego przeskalowania i dostosowania hiperparametrów jest nieefektywna, ponieważ filtry uczą się wzorców o innej skali przestrzennej.
- Próba wykorzystania gotowego, przefiltrowanego zbioru danych (`02-tvat-filtered`) okazała się niemożliwa ze względu na jego niekompletność.

## Propozycja dalszych kroków:

- **Opcja 1:** Próba dostosowania danych do obecnej architektury – zmiana długości segmentu i dodanie paddingu, aby próbki miały rozmiar 250x90.
- **Opcja 2:** Znalezienie optymalnej architektury i hiperparametrów dla formatu 200x53 – użycie `Optuny`, aby przetestować różne kombinacje rozmiarów filtrów i `stride'u`, liczby filtrów, `learning rate'u` i `dropout'u`.

### Summary of LeNet-5 Architecture

| Layer  |                 | Feature Map | Size  | Kernel Size | Stride | Activation |
|--------|-----------------|-------------|-------|-------------|--------|------------|
| Input  | Image           | 1           | 32x32 | -           | -      | -          |
| 1      | Convolution     | 6           | 28x28 | 5x5         | 1      | tanh       |
| 2      | Average Pooling | 6           | 14x14 | 2x2         | 2      | tanh       |
| 3      | Convolution     | 16          | 10x10 | 5x5         | 1      | tanh       |
| 4      | Average Pooling | 16          | 5x5   | 2x2         | 2      | tanh       |
| 5      | Convolution     | 120         | 1x1   | 5x5         | 1      | tanh       |
| 6      | FC              | -           | 84    | -           | -      | tanh       |
| Output | FC              | -           | 10    | -           | -      | softmax    |

Summarized table for LeNet 5 Architecture

Rysunek 26: Podsumowanie architektury LeNet-5

```
class UT_HAR_LeNet(nn.Module):
 def __init__(self):
 super(UT_HAR_LeNet, self).__init__()
 self.encoder = nn.Sequential(
 #input size: (1,250,90)
 nn.Conv2d(1,32,7,stride=(3,1)),
 nn.ReLU(True),
 nn.MaxPool2d(2),
 nn.Conv2d(32,64,(5,4),stride=(2,2),padding=(1,0)),
 nn.ReLU(True),
 nn.MaxPool2d(2),
 nn.Conv2d(64,96,(3,3),stride=1),
 nn.ReLU(True),
 nn.MaxPool2d(2)
)
 self.fc = nn.Sequential(
 nn.Linear(96*4*4,128),
 nn.ReLU(),
 nn.Linear(128,7)
)
```

Rysunek 27: Zrzut kodu modelu LeNet z repozytorium statycznego

## Analiza modelu LeNet

### Architektura LeNet-5

<https://medium.com/@siddheshb008/lenet-5-architecture-explained-3b559cb2d52b>

### Model LeNet z repozytorium statycznego:

- nn.Linear → warstwa w pełni połączona
- (128,7) → 128 wejść, 7 wyjść (nasze klasy) – w Keras mamy po prostu Dense(num\_class), bo tam input size jest ustalany automatycznie

Model w PyTorch to zmodyfikowana wersja klasycznego LeNet-5, dostosowana do danych o wymiarach (1, 250, 90) i klasyfikacji na 7 klas. Nie jest to oryginalny LeNet-5, ale jego

inspirowana wersja, znana również jako „LeNet-like”.

## Porównanie UT\_HAR\_LeNet z oryginalnym LeNet-5

|                            | <b>LeNet-5</b>                      | <b>UT_HAR_LeNet</b>                          |
|----------------------------|-------------------------------------|----------------------------------------------|
| Dane wejściowe             | (1, 32, 32)                         | (1, 250, 90)                                 |
| Warstwy Conv               | 3 (Conv1, Conv3, Conv5)             | 3 (Conv2d bez nieregularnych połączeń)       |
| Liczba filtrów             | 6 → 16 → 120                        | 32 → 64 → 96                                 |
| Rozmiar kernela Conv       | 5x5 przez cały model                | 7x7, 5x4, 3x3                                |
| Pooling                    | AvgPooling2D, kernel=2x2            | MaxPooling2D, kernel=2x2                     |
| Aktywacje                  | tanh → tanh → tanh → tanh → softmax | ReLU → ReLU → ReLU → brak aktywacji (logity) |
| Funkcja wyjściowa (output) | Softmax                             | CrossEntropyLoss                             |
| Warstwy w pełni połączone  | 2                                   | 2                                            |

## Co z softmax?

W modelu PyTorch ostatnią warstwą jest `nn.Linear(128, 7)`, co oznacza, że sieć zwraca logity – surowe wartości output z warstwy w pełni połączonej, bez zastosowania funkcji aktywacji. Są to dowolne liczby rzeczywiste (mogą być dodatnie, ujemne lub równe zero), które nie są jeszcze interpretowalne jako prawdopodobieństwa. Aby przekształcić logity na rozkład prawdopodobieństwa, stosuje się funkcję aktywacji softmax, która przekształca wektor wartości rzeczywistych w wektor dodatnich wartości w przedziale (0, 1), sumujących się do 1. W tym modelu softmax nie został dodany ręcznie, ponieważ mamy funkcję strat `nn.CrossEntropyLoss()`.

Funkcja strat `nn.CrossEntropyLoss()` automatycznie łączy w sobie dwie operacje:

1. Softmax – przekształca logity na prawdopodobieństwa klas,
2. Log loss (entropię krzyżową) – mierzy błąd predykcji względem rzeczywistej etykiety.

Dzięki temu logity mogą być bezpośrednim wyjściem modelu, a `CrossEntropyLoss` zadba o prawidłowe przekształcenie i porównanie. Jest to standardowa i zalecana praktyka w PyTorchu, pozwalająca uniknąć niestabilności numerycznej i uprościć strukturę sieci.

## Model w Keras

Ostatnia warstwa `Dense(num_classes)` zwraca logity, a funkcja strat `SparseCategoricalCrossentropy(from_logits=True)` wewnętrznie przekształca je przez softmax, dzięki czemu nie trzeba (i nie powinno się) dodawać softmaxa ręcznie.

|                       | Zbiór statyczny          | Zbiór dynamiczny                                     |
|-----------------------|--------------------------|------------------------------------------------------|
| <b>Format pliku</b>   | Binarny Numpy            | Tekstowy                                             |
| <b>Przetwarzanie</b>  | Gotowe do użycia         | Wymaga parsowania, obliczania amplitudy, odszumiania |
| <b>Poziom szumu</b>   | Niski, dane czyste       | Wysoki, dane surowe, zaszumione                      |
| <b>Wymiary próbki</b> | (250, 90)                | (200, 53) po przetworzeniu                           |
| <b>Struktura</b>      | Dominujące pasma poziome | Dominujące pasma pionowe (eventy)                    |

Tabela 3: Tabela porównawcza zbiorów danych

```
#Definicja modelu LeNet w Keras
from tensorflow.keras.layers import Input

model = Sequential([
 Input(shape=(200, 53, 1)),

 Conv2D(32, kernel_size=(7, 7), strides=(3, 1), activation='relu'),
 MaxPooling2D(pool_size=(2, 2)),
 Dropout(0.25),
 Conv2D(64, kernel_size=(5, 4), strides=(2, 2), activation='relu'),
 MaxPooling2D(pool_size=(2, 2)),
 Dropout(0.25),
 Conv2D(96, kernel_size=(3, 3), activation='relu'),
 MaxPooling2D(pool_size=(2, 2)),
 Dropout(0.25),
 Flatten(),
 Dropout(0.5),
 Dense(128, activation='relu'),
 Dense(num_classes) # Wyjście to logity
])

Komplikacja modelu z loss='...' (from_logits=True)
model.compile(
 optimizer=tf.keras.optimizers.Adam(learning_rate=0.0005),
 loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
 metrics=['accuracy']
)

model.summary()
```

Rysunek 28: Zrzut kodu modelu w Keras

```
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.layers import Dense, Dropout, Conv2D, MaxPooling2D, Flatten
from tensorflow.keras.losses import SparseCategoricalCrossentropy

Define the CNN model
model = keras.Sequential([
 # Używamy parametrów z LeNet, żeby wymiary się zgadzały
 keras.layers.Conv2D(32, kernel_size=(7,7), strides=(3,1), activation='relu', input_shape=(250, 90, 1)),
 keras.layers.MaxPooling2D(2),

 # Druga warstwa również z parametrymi inspirowanymi LeNet
 keras.layers.Conv2D(64, kernel_size=(5,4), strides=(2,2), activation='relu'),
 keras.layers.MaxPooling2D(2),

 keras.layers.Flatten(),

 # Reszta Twojej oryginalnej architektury zostaje
 keras.layers.Dense(128, activation='relu'),
 keras.layers.Dropout(0.5),
 keras.layers.Dense(64, activation='relu'),
 keras.layers.Dropout(0.25),
 keras.layers.Dense(7, activation='softmax')
])

Compile the model
model.compile(optimizer=tf.keras.optimizers.Adam(learning_rate=0.0001),
 loss='sparse_categorical_crossentropy',
 metrics=['accuracy'])

model.summary()
```

Rysunek 29: Zrzut kodu prostej sieci CNN z repozytorium dynamicznego

## Analiza prostej sieci CNN z repozytorium dynamicznego

|                           | <b>LeNet-5</b> | <b>prosty CNN</b>  |
|---------------------------|----------------|--------------------|
| Ilość Conv layers         | 3              | 2                  |
| Pooling                   | AveragePooling | MaxPooling         |
| Warstwy w pełni połączone | 2              | 3                  |
| Aktywacje                 | Softmax        | Softmax            |
| Dropout                   | brak           | obecny (0.5, 0.25) |

```
=====
Optymalizacja zakończona.
Liczba zakończonych prób: 25

Najlepsza próba:
 > Dokładność walidacji: 0.6775
 > Najlepsze hiperparametry:
 - filters1: 32
 - filters2: 96
 - filters3: 160
 - dropout1: 0.2368661935399467
 - dropout2: 0.2991304309686125
 - dropout3: 0.2922253286580787
 - dropout_flatten: 0.43336967902257334
 - dense_units: 256
 - learning_rate: 0.0007476146586326026
=====
```

Rysunek 30: Parametry Optuny dla pierwszych testów

## Dostrajanie modelu LeNet

**Dodanie dodatkowej warstwy w pełni połączonej:**

Najlepszy wynik: 57,61%

**3 warstwy w pełni połączone, usunięcie zbędnego dropoutu i dodanie early stoppingu:**

Najlepszy wynik: 64,86%

**3 warstwy w pełni połączone, usunięcie zbędnego dropoutu, dodanie early stoppingu i zwiększenie epok do 100:**

Najlepszy wynik: 65,58%

**2 warstwy w pełni połączone (tak jak na początku), early stopping, 50 epok:**

Najlepszy wynik: 64,86%

**Optuna:**

Najlepszy wynik: 68,12%

**Zmiana pierwszego kernel size na 5x5 i użycie stride 2,1 zamiast 3,1:**

Najlepszy wynik: 72,46%

```

#Definicja modelu LeNet w Keras
from tensorflow.keras.layers import Input

model = Sequential([
 Input(shape=(200, 53, 1)),

 Conv2D(32, kernel_size=(5, 5), strides=(2, 1), activation='relu'),
 MaxPooling2D(pool_size=(2, 2)),
 Dropout(0.24),
 Conv2D(64, kernel_size=(5, 4), strides=(2, 2), activation='relu'),
 MaxPooling2D(pool_size=(2, 2)),
 Dropout(0.29),
 Conv2D(128, kernel_size=(3, 3), activation='relu'),
 MaxPooling2D(pool_size=(2, 2)),
 Dropout(0.29),
 Flatten(),
 Dropout(0.43),
 Dense(256, activation='relu'),
 Dense(num_classes) # Wyjście to logity
])

Kompilacja modelu z loss='...' (from_logits=True)
model.compile(
 optimizer=tf.keras.optimizers.Adam(learning_rate=0.00075),
 loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
 metrics=['accuracy']
)

model.summary()

```

Rysunek 31: Zrzut kodu modelu dla początkowej zmieny kernel size przybliżonej do struktury LeNet-5

## Usuwanie podnośnych

Schemat:

Źródło: <https://www.extremenetworks.com/resources/blogs/ofdm-and-ofdma-subcarriers-what-are-the>

Efekt usunięcia podnośnych [2, 3, 4, 5, 32, 59, 60, 61, 62, 63] (mamy wtedy 54 podnośne “niosące informacje”):

Najlepszy wynik to 75%.

<https://www.extremenetworks.com/resources/blogs/ofdm-and-ofdma-subcarriers-what-are-the>

Efekt usunięcia podnośnych [1, 2, 3, 4, 5, 32, 59, 60, 61, 62, 63] (mamy wtedy 53 podnośne “niosące informacje”): Najlepszy wynik to 69,93%.

Efekt usunięcia podnośnych [0, 1, 2, 3, 4, 5, 32, 59, 60, 61, 62, 63] (mamy wtedy 52 podnośne niosące informacje – wtedy usuwam wszystkie podnośne pilotowe, co nie powinno być poprawne): Najlepszy wynik to 70,65%.

Model, na którym działałam:

Usunięcia podnośnych [2, 3, 4, 5, 32, 59, 60, 61, 62, 63] (mamy wtedy 54 podnośne “niosące informacje”) dla starego modelu (przed użyciem optuny):

```
[99] #Definicja modelu LeNet w Keras
from tensorflow.keras.layers import Input

model = Sequential([
 Input(shape=(28, 28, 1)),

 Conv2D(48, kernel_size=(5, 5), strides=(2, 1), activation='relu'),
 MaxPooling2D(pool_size=(2, 2)),
 Dropout(0.38),
 Conv2D(96, kernel_size=(5, 4), strides=(2, 2), activation='relu'),
 MaxPooling2D(pool_size=(2, 2)),
 Dropout(0.3),
 Conv2D(160, kernel_size=(3, 3), activation='relu'),
 MaxPooling2D(pool_size=(2, 2)),
 Dropout(0.29),
 Flatten(),
 Dropout(0.34),
 Dense(256, activation='relu'),
 Dense(num_classes) # Wyjście to logity
])

Komplikacja modelu z loss='...' (from_logits=True)
model.compile(
 optimizer=tf.keras.optimizers.Adam(learning_rate=0.00088),
 loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
 metrics=['accuracy']
)

model.summary()
```

Rysunek 32: Zrzut modelu, na którym zostały przeprowadzone eksperymenty

```
=====
Optymalizacja zakończona.

Najlepsza próba:
> Dokładność walidacji: 0.6630
> Najlepsze hiperparametry:
- filters1: 32
- filters2: 96
- filters3: 160
- dropout1: 0.3528335913842677
- dropout2: 0.41942175880020394
- dropout3: 0.32735903477883843
- dropout_flatten: 0.4061878623180358
- dense_units: 256
- learning_rate: 0.0008853855849360852
=====
```

Rysunek 33: Nowe parametry Optuny dla 54 podnośnych

Najlepszy wynik: 66,30%.

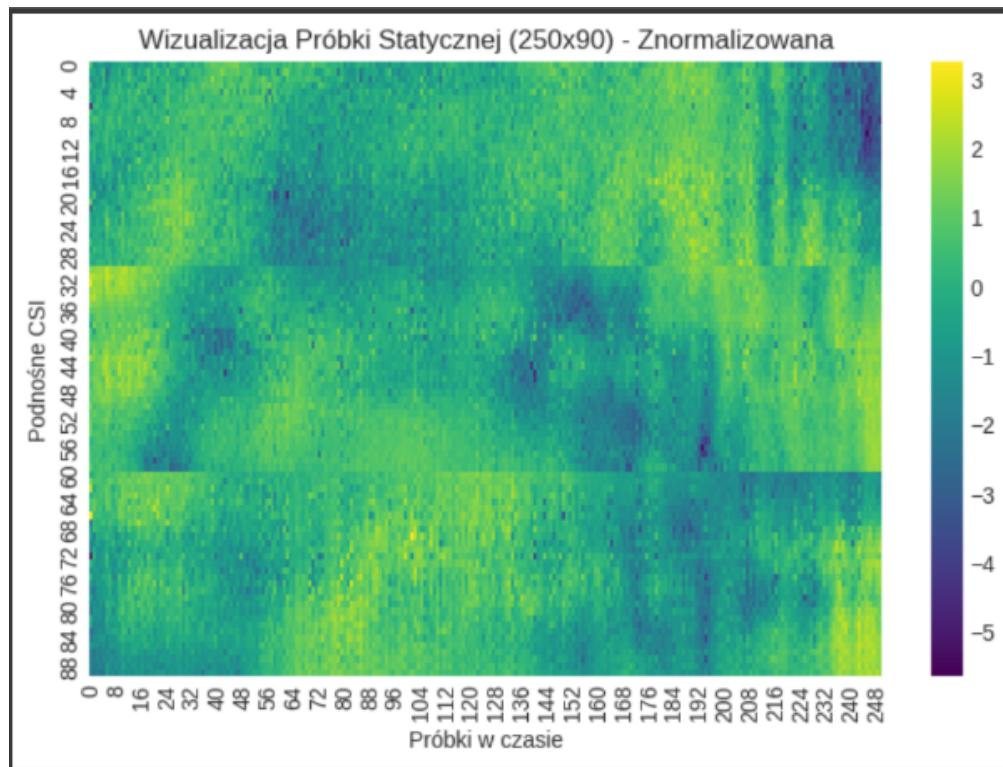
**Usunięcia podnośnych [2, 3, 4, 5, 32, 59, 60, 61, 62, 63]** (mamy wtedy **54** podnośne “niosące informacje”) z **najlepszymi parametrami optuny**: Najlepszy wynik to 69,57%.

**Usunięcia podnośnych [2, 3, 4, 5, 32, 59, 60, 61, 62, 63]** (mamy wtedy **54** podnośne “niosące informacje”) dla starego modelu (przed użyciem optuny) – 100 epok:

Najlepszy wynik: 73,91%.

**Nowe parametry optuny dla 54 podnośnych:**

Najlepszy wynik: 68,48%.

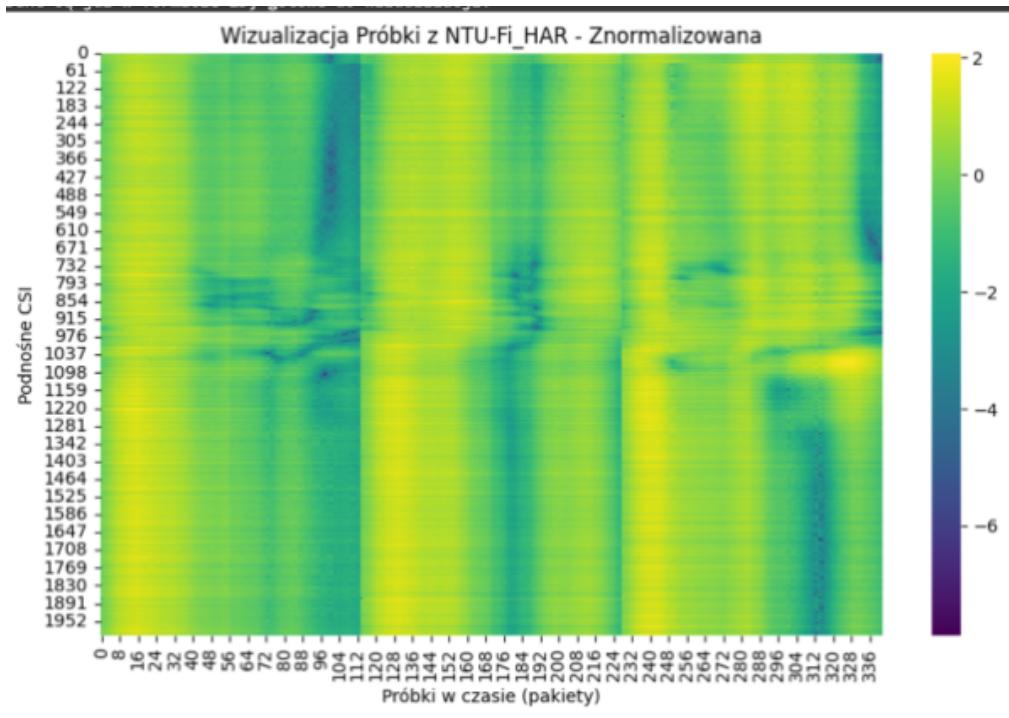


Rysunek 34: Przykładowa wizualizacja ze zbioru UT\_HAR.

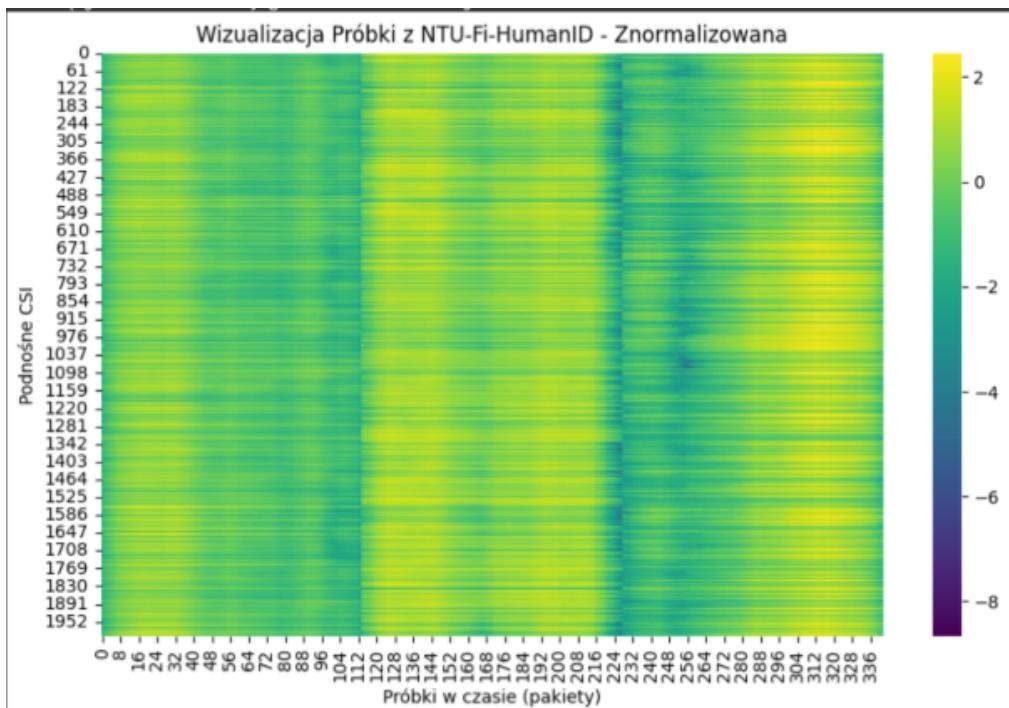
## Analiza zbiorów danych z repozytorium statycznego

Zwizualizowałam przykładowe próbki ze zbiorów podanych w repozytorium statycznym, aby zweryfikować, czy któryś z modeli jest dostosowany do danych dla jednej anteny. Wykonałam to za pomocą notatnika dostępnego pod adresem:

[https://colab.research.google.com/drive/1pRBZd-A\\_o6RQ5EQ7DIQ3tNZMZlA4tK5w](https://colab.research.google.com/drive/1pRBZd-A_o6RQ5EQ7DIQ3tNZMZlA4tK5w)



Rysunek 35: Przykładowa wizualizacja ze zbioru NTU-Fi\_HAR.



Rysunek 36: Przykładowa wizualizacja ze zbioru NTU-Fi-HumanID.

## **Analiza modeli**

Analiza modeli z repozytorium statycznego na danych dynamicznych bez optymalizacji. Modele zostały przepisane z pliku `UT_HAR_model.py` na bibliotekę Keras, ponieważ pomimo 3 anten właśnie te dane najlepiej pasują do danych z repozytorium dynamicznego (1 kanał, 7 klas), zatem to ten model najlepiej będzie się nadawał do dalszej optymalizacji.

Link do notatnika: [https://colab.research.google.com/drive/1M2w9csy4rqN0zr68exdxKJmtZ5rnwrki\(scrollTo=y\\_0GMJxdg7rw](https://colab.research.google.com/drive/1M2w9csy4rqN0zr68exdxKJmtZ5rnwrki(scrollTo=y_0GMJxdg7rw)

## **Wyniki wstępnych testów modeli**

1. LeNet: ok. 65%
2. ResNet18: 70–80%
3. ResNet50: 70–80%
4. ResNet101: 70–80%
5. RNN: ok. 50 %
6. GRU: ok. 70 %
7. LSTM: ok. 65 %
8. BiLSTM: ok. 70 %
9. CNN+GRU: ok. 80 %
10. ViT (Transformer): ok. 65 %

|            | Dokładność | Szybkość    | Kluczowe Komponenty                                                                                          | Ocena                                                                                                                                                         |
|------------|------------|-------------|--------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------|
| LeNet      | ok. 65%    | ok. 6 min   | Conv2D, MaxPooling2D, Dropout, Flatten, Dense                                                                | - niższa dokładność względem innych modeli                                                                                                                    |
| ResNet-18  | 70-80%     | ok. 1 h     | Conv2D, BatchNorm, ReLU, MaxPool, Add, AvgPool, Dense                                                        | - dobra dokładność kosztem długiego czasu treningu<br>- może być zbyt złożone                                                                                 |
| ReNet-50   | 70-80%     | ok. 2 h     | Conv2D (Bottleneck), BatchNorm, ReLU, MaxPool, Add, AvgPool, Dense                                           | - dobra dokładność kosztem długiego czasu treningu<br>- może być zbyt złożone                                                                                 |
| ResNet-100 | 70-80%     | ok. 3 h     | Conv2D (Bottleneck), BatchNorm, ReLU, MaxPool, Add, AvgPool, Dense                                           | - dobra dokładność kosztem długiego czasu treningu<br>- może być zbyt złożone                                                                                 |
| RNN        | ok. 50%    | ok. 1,5 min | Reshape, SimpleRNN, Dense                                                                                    | - najniższa dokładność pomimo szybkości i prostoty                                                                                                            |
| GRU        | ok. 70%    | ok. 5 min   | Reshape, GRU, Dense                                                                                          | - szybkie i bardzo lekkie                                                                                                                                     |
| LSTM       | ok. 65%    | ok. 3,5 min | Reshape, LSTM, Dense                                                                                         | - niższa dokładność względem innych modeli                                                                                                                    |
| BiLSTM     | ok. 70%    | ok. 7 min   | Reshape, Bidirectional(LSTM), Dense                                                                          | - nie przyniosł znaczącej poprawy względem GRU                                                                                                                |
| CNN+GRU    | ok. 80%    | ok. 3,5 min | Reshape, Conv1D, GRU, Dropout, Dense                                                                         | - połączenie najwyższej wydajności z dobrą szybkością architektura hybrydowa dobrze pasuje do danych takich jak CSI                                           |
| VIT        | ok. 65%    | ok. 15 min  | Patches, PatchEncoder, LayerNorm, <del>MultiHeadAttention</del> , <del>Add</del> , Dense, <del>AvgPool</del> | - niższa dokładność względem innych modeli<br>- mimo nowoczesnej architektury, nie pokazał swojej przewagi (może wymaga więcej danych lub dłuższego treningu) |

Rysunek 37: Tabela porównawcza modeli

```
zuzia@zuzakomp:/mnt/c/Users/zrota/Desktop/studia/praca_inżynierska/faza testowa$
```

Rysunek 38: Pierwszy test środowiska

## Implementacja środowiska

1. Podłączenie obu płyt do portów USB
2. Uruchomienie w PowerShellu środowiska IDF-ESP v. 5.5
3. Identyfikacja portów szeregowych, do których podłączone są poszczególne płytki
4. Wejście do folderu z softwarem poszczególnej płytki (active\_sta/active\_ap)
5. idf.py bulid na każdej płytce
6. Konfiguracja środowiska idf wg zaleceń na repozytorium (idf.py menuconfig)
7. Flashowanie software'u na każdej płytce przy pomocy zwarcia pinu G23 z uziemieniem GND, jednocześnie wciskając przycisk RESET
8. idf.py -p <numer portu szeregowego> - wejście na konkretną płytke, aby wgrać software
9. Aby uruchomić program przekierowujący dane CSI i zainportować potrzebne biblioteki do modelu należy użyć wsl/linuxa
10. Należy wykonać attach portów szeregowych z Windowsa do WSL
11. Uruchomienie programy komendą python test.py

## Porównanie działania 2 modeli na własnych danych

**Scenariusz testowy:** 2 razy wykrycie chodzenia, 10 min na wykrycie ok. 5 braków aktywności, 3 razy wykrycie chodzenia

**Wyniki:**

### BiLSTM

Dane “chodzenie”/”brak aktywności” po ok. 2300 próbek

- Czas treningu: ok. 4 min
- Wyniki scenariusza testowego:
  - Chodzenie 2/2
  - Brak aktywności 4/5
  - Chodzenie 3/3

### CNN+GRU

Dane “chodzenie”/”brak aktywności” po ok. 2300 próbek

- Czas treningu: 1 min
- Wyniki scenariusza testowego:
  - Chodzenie 2/2
  - Brak aktywności 5/6
  - Chodzenie 2/3

**Uwagi:** Aktywności wykrywane są w czasie 2–3min. BiLSTM wydaje się być bardziej precyzyjny (zazwyczaj 1 błąd na cały test). Zauważono, że kiedy BiLSTM już popełni błąd, to nigdy nie jest pewny swojej odpowiedzi (wyświetla pewność 50%). Jeśli chodzi o CNN+GRU pierwszy błąd popełnił z pewnością 64%, a drugi z pewnością 94%.

**Wnioski:** Oba modele mają bardzo podobne wyniki. Pomimo, że w przeprowadzonych badaniach przedstawionych w tabeli porównawczej modeli oraz krótkiemu czasu treningu, zdecydowałam się na użycie modelu BiLSTM w dalszej części projektu. Jest on minimalnie bardziej dokładny niż CNN+GRU, a także tego typu modele są wykorzystywane w podobnych typu projektach przedstawionych w literaturze.

## Model rozpoznający "siedzenie"/"leżenie"

**Badanie 1:** Zebrałam próbki kładzenia się i siadania w odstępach co 5s - 5s na położenie się, następnie 5s na podniesienie się do siadu. Rozdzielenie odpowiednich próbek do odpowiednich plików csv. było możliwe, dzięki dodaniu do programu do zbierania danych fragmentu umieszczonego poniżej.

```
FILE_LEZENIE = 'lezenie5s.csv'
FILE_SIEDZENIE = 'siedzenie5s.csv'
INTERVAL_SEC = 5.0

print(f"Łączenie z portem {COM_PORT} przy {BAUD_RATE} bps...")

try:
 ser = serial.Serial(COM_PORT, BAUD_RATE, timeout=1)

 with open(FILE_LEZENIE, 'a', newline='', encoding='utf-8') as f_lezenie,
 open(FILE_SIEDZENIE, 'a', newline='', encoding='utf-8') as f_siedzenie:

 writer_lezenie = csv.writer(f_lezenie)
 writer_siedzenie = csv.writer(f_siedzenie)

 print(f"Zapisywanie danych 'leżenie' do: {FILE_LEZENIE}")
 print(f"Zapisywanie danych 'siedzenie' do: {FILE_SIEDZENIE}")
 print(f"Przełączanie plików co {INTERVAL_SEC} sekund.")
 print("Rozpoczęto nasłuchiwanie... (Naciśnij CTRL+C, aby zakończyć)")

 print_line = False
```

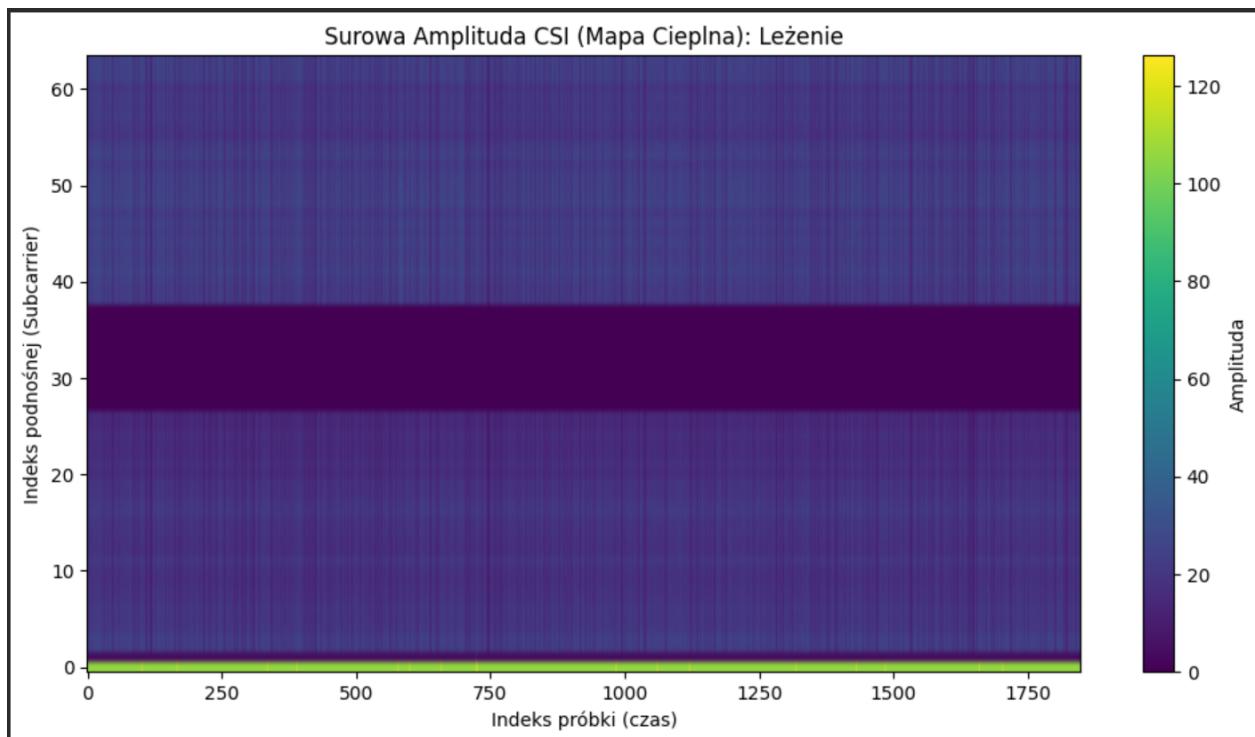
Dodałam również wyświetlanie na ekranie, kiedy kłaść się, a kiedy wstawać do siedzenia, aby nie musieć polegać na stoperze w ręce, a jedynie sygnałach na wyświetlaczu komputera.

## Segmentacja

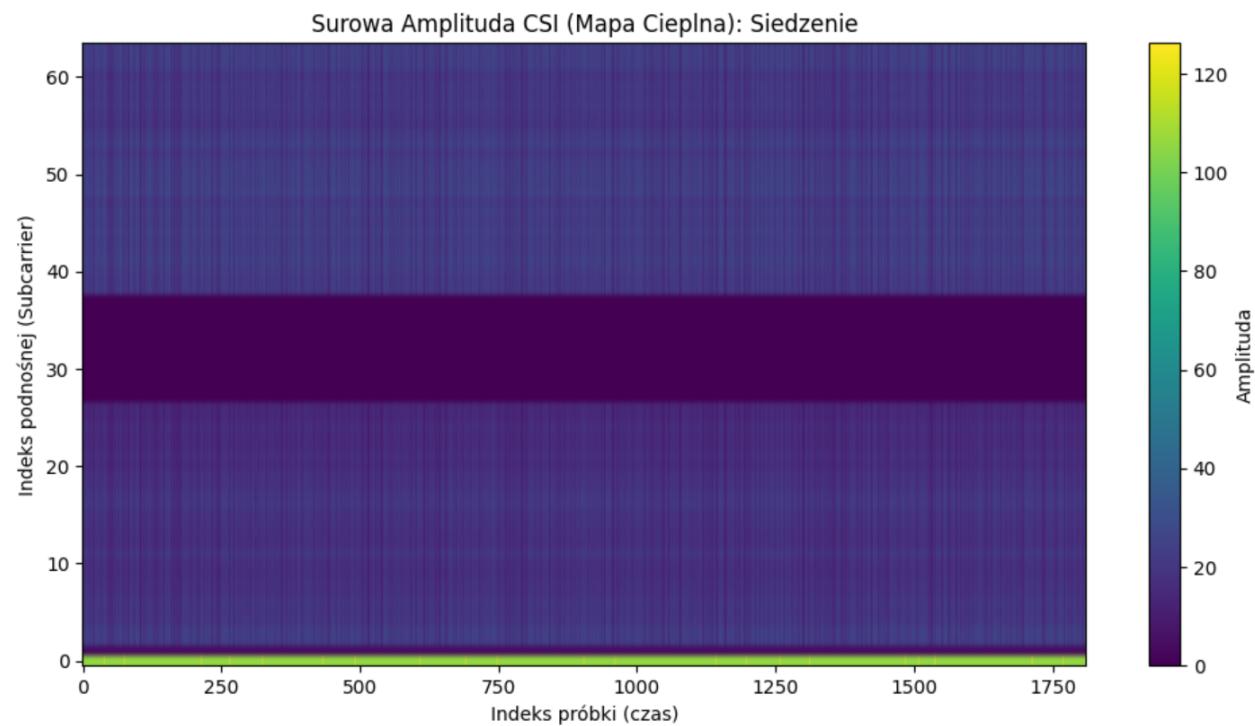
W poprzednich badaniach (chodzenie/brak aktywności) zaobserwowałam, że zbieranie próbek do wykrycia aktywności trwa 2–3 min. Tak długi czas może się nie sprawdzić w tym projekcie ze względu na to, że np. podnoszenie się do siadu trwa zaledwie kilka sekund. Problemem oczywiście nie był użyty model, a długość segmentów, które model następnie przetwarza na wejściu. Skróciłam więc długość zbieranych segmentów z 200 do 20 i w ten sposób zbieranie próbek do wykrycia trwa ok. 10s.

## Trening

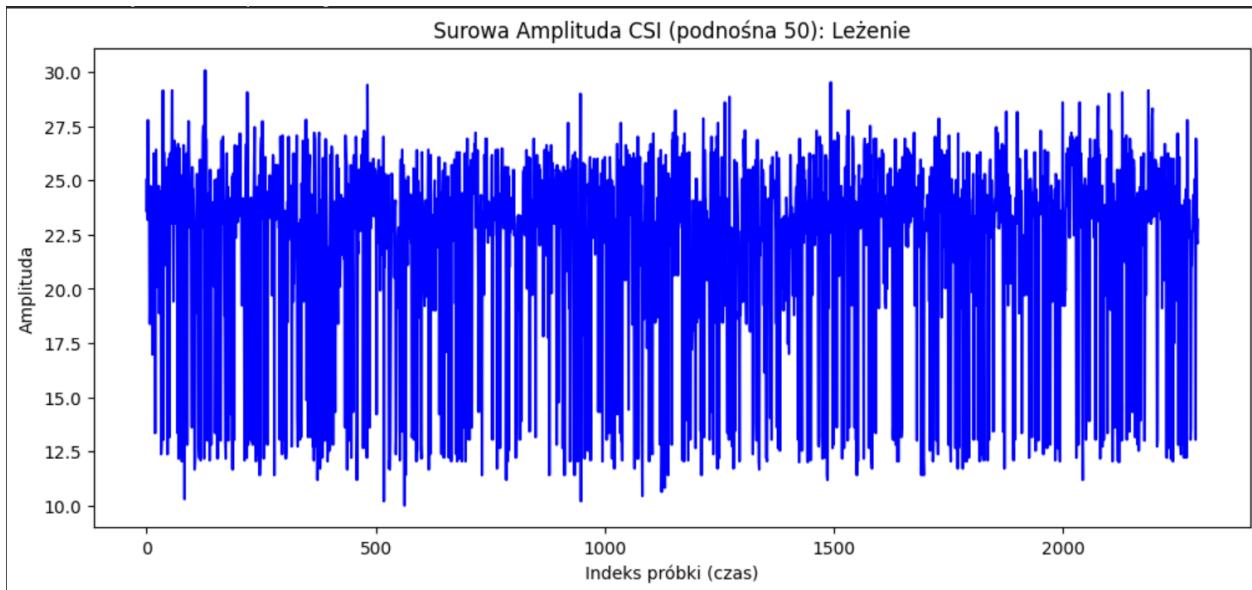
W treningu wykorzystałam ok. 3600 próbek (po 1800 na każdą aktywność). Po segmentacji łącznie wygenerowano 725 próbek treningowych (366 dla leżenia i 359 dla siedzenia). Zbiór treningowy wynosił 580 próbek, a testowy 145. Zaobserwowałam, że model BiLSTM osiągnął



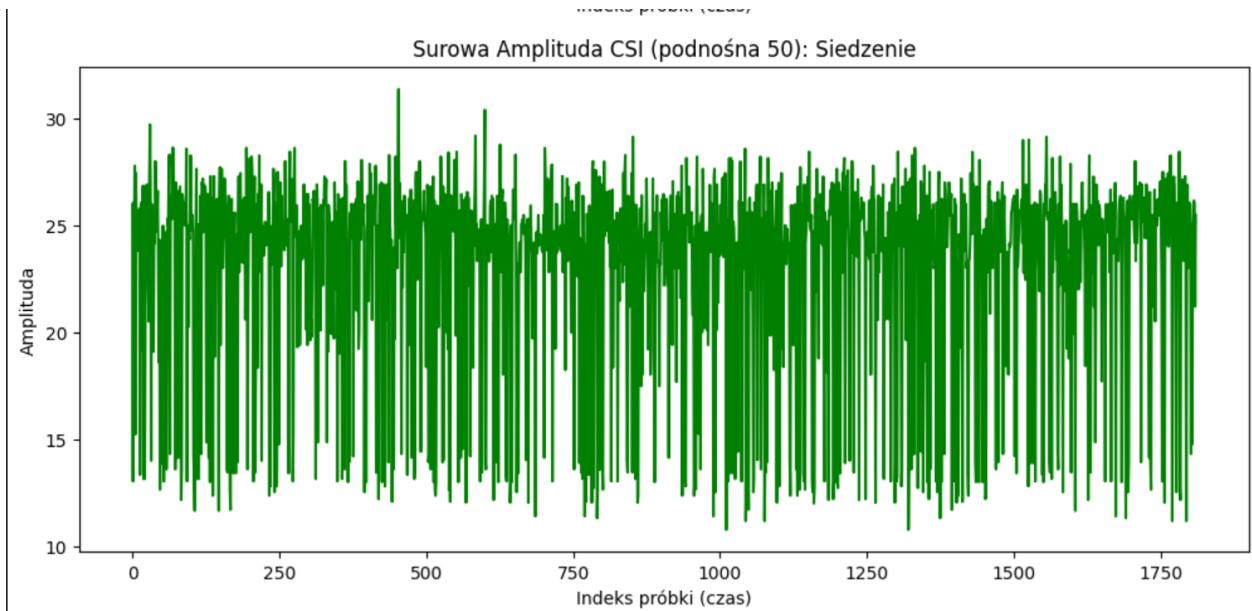
Rysunek 39: Wykres: Surowa amplituda, Aktywność: leżenie, Badanie nr 1



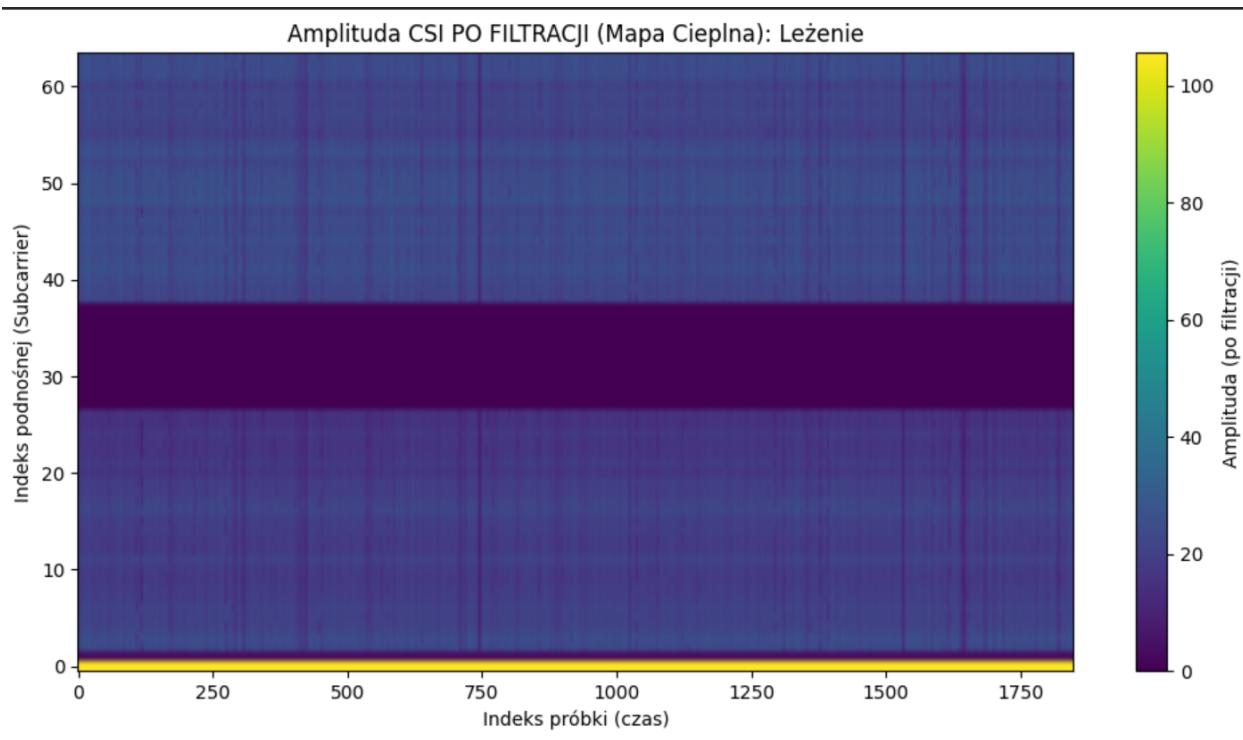
Rysunek 40: Wykres: Surowa amplituda, Aktywność: siedzenie, Badanie nr 1



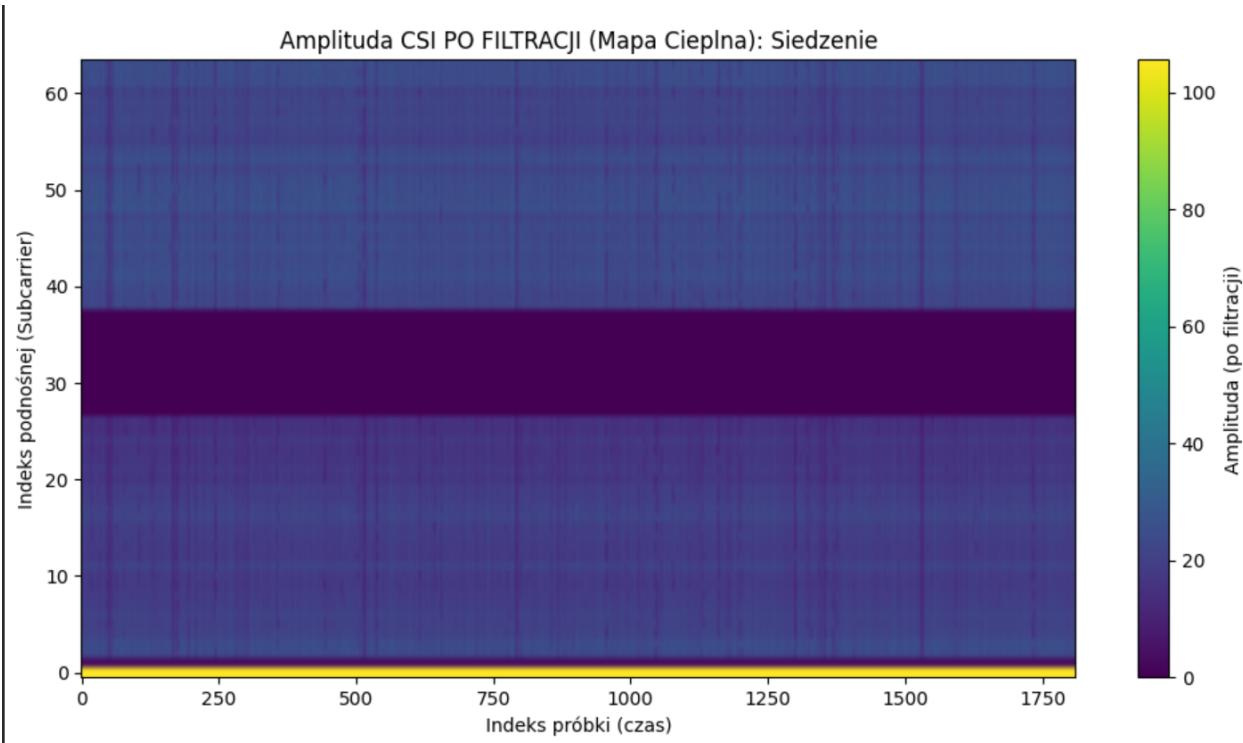
Rysunek 41: Wykres: Podnośna nr 50 (surowa), Aktywność: leżenie, Badanie nr 1



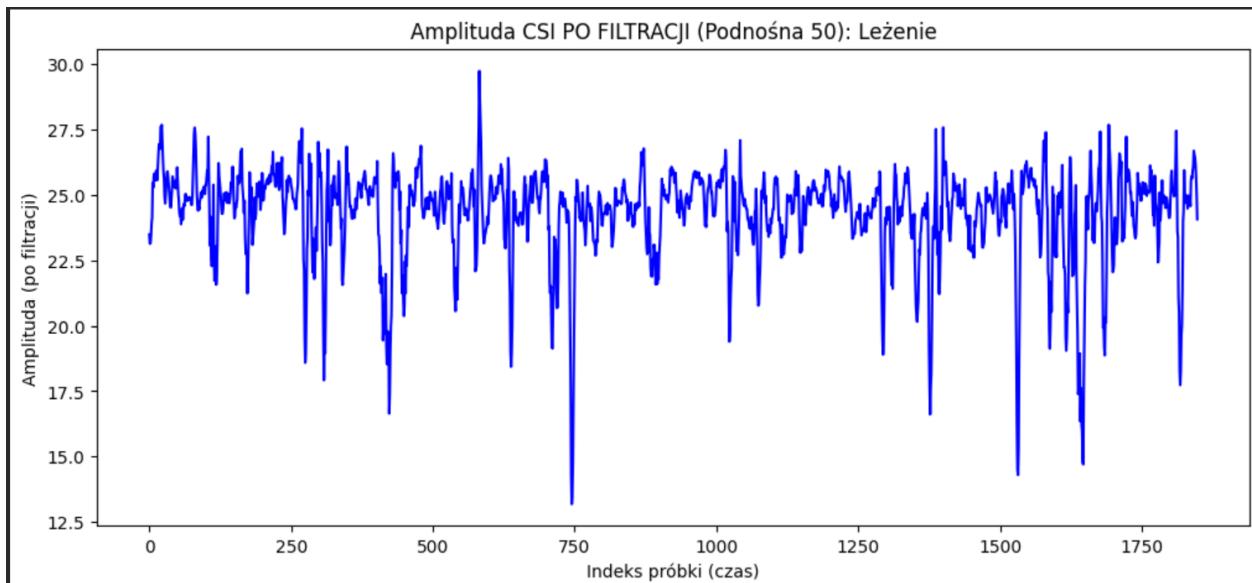
Rysunek 42: Wykres: Podnośna nr 50 (surowa), Aktywność: siedzenie, Badanie nr 1



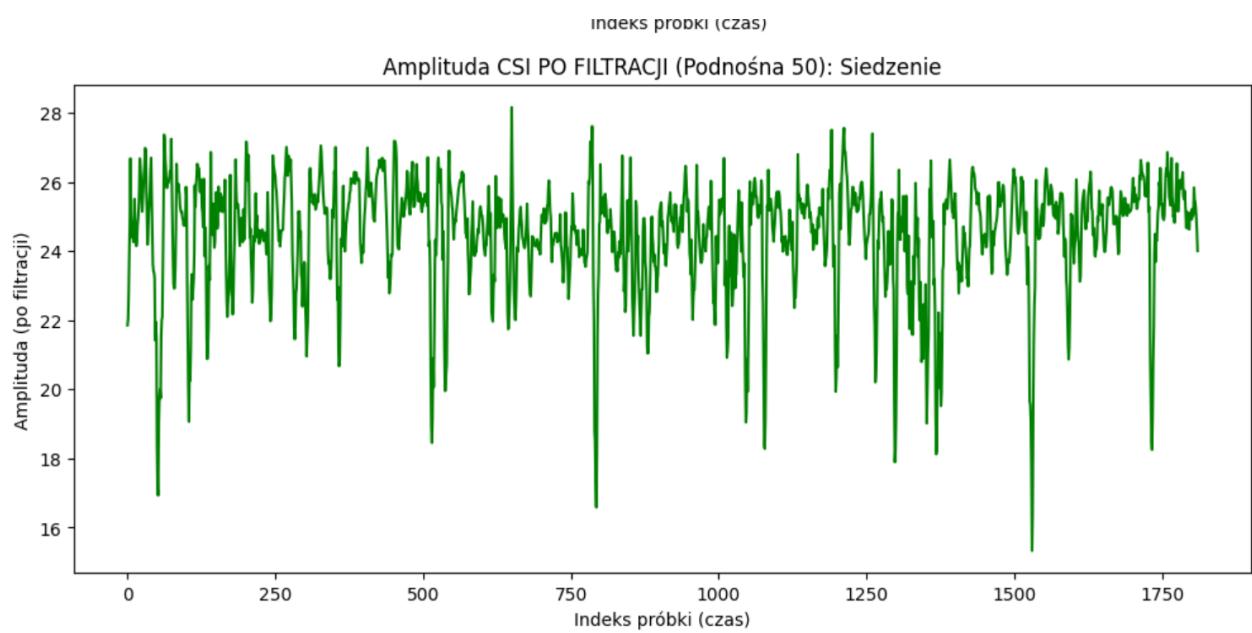
Rysunek 43: Wykres: Po filtracji, Aktywność: leżenie, Badanie nr 1



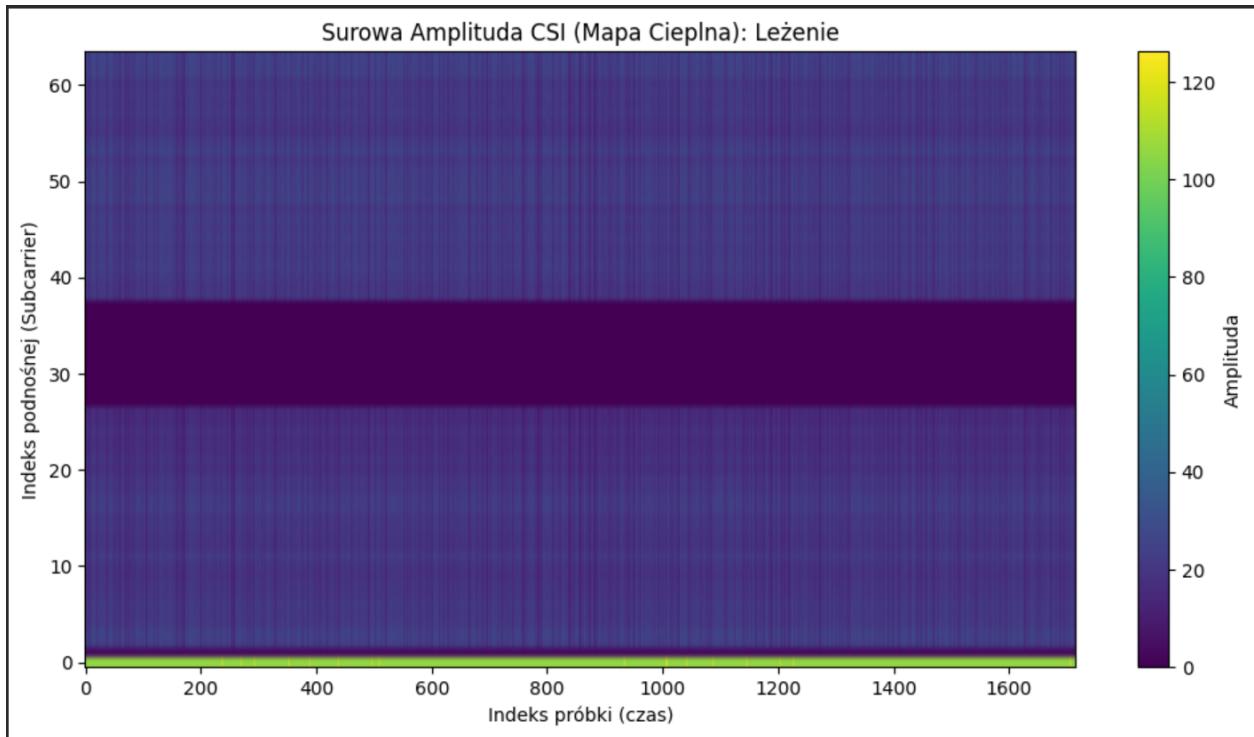
Rysunek 44: Wykres: Po filtracji, Aktywność: siedzenie, Badanie nr 1



Rysunek 45: Wykres: Podnośna nr 50 (po filtracji), Aktywność: leżenie, Badanie nr 1



Rysunek 46: Wykres: Podnośna nr 50 (po filtracji), Aktywność: siedzenie, Badanie nr 1



Rysunek 47: Wykres: Surowa amplituda, Aktywność: leżenie, Badanie nr 2

ok. 90% dokładności i zakończył trening na 18 epoce dzięki zastosowaniu early stoppingu.

## Wyniki

Niestety model nie wykrył żadnej aktywności.

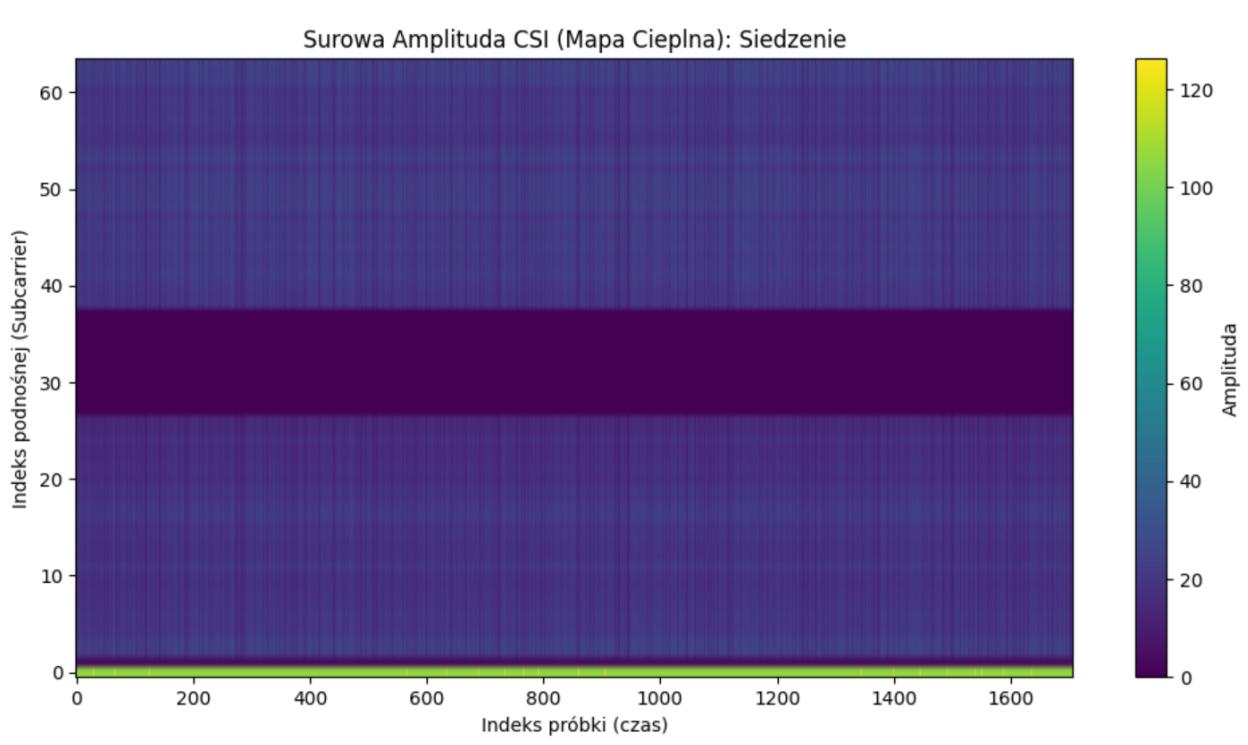
**Badanie 2:** Podejrzewam, że problemem może być fakt, że 5s to za mało, aby rozróżnić kładzenie się i siadanie (zbyt podobne dla siebie w tak krótkim czasie). Wydłużyłam zatem czas zbierania próbek dla jednej aktywności do 15s.

## Trening

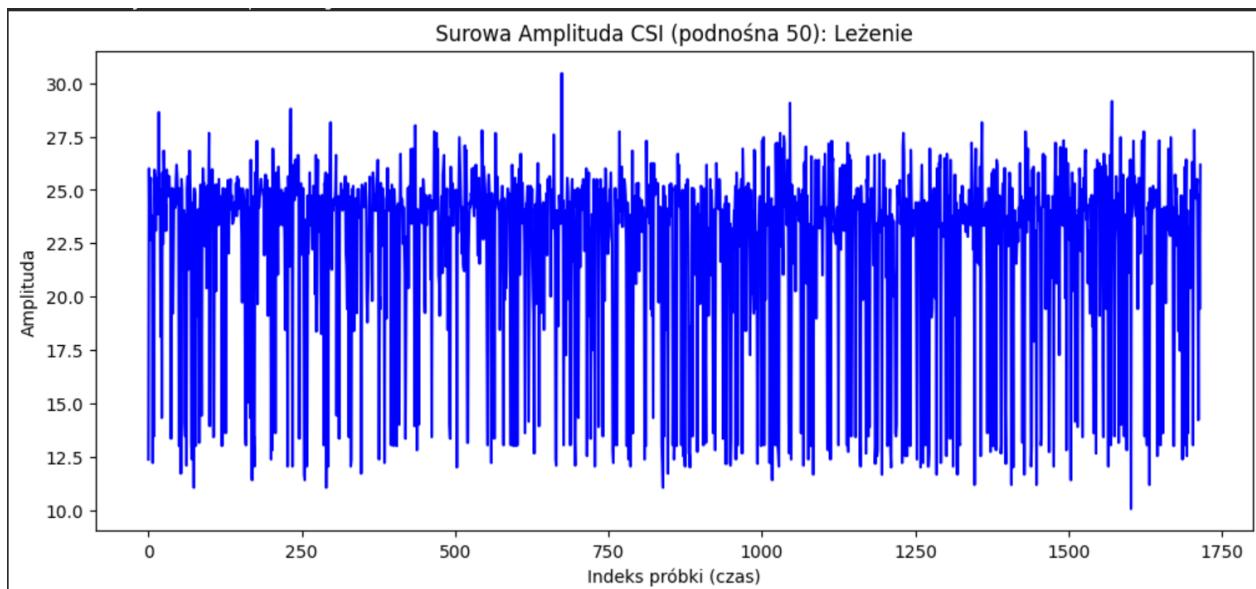
W treningu wykorzystałam ok. 3400 próbek (po 1700 na każdą aktywność). Po segmentacji łącznie wygenerowano 678 próbek treningowych (340 dla leżenia i 338 dla siedzenia). Zbiór treningowy wynosił 542 próbek, a testowy 136. Zaobserwowałam, że model BiLSTM osiągnął 88% dokładności i zakończył trening na 11 epoce dzięki zastosowaniu early stoppingu.

## Wyniki

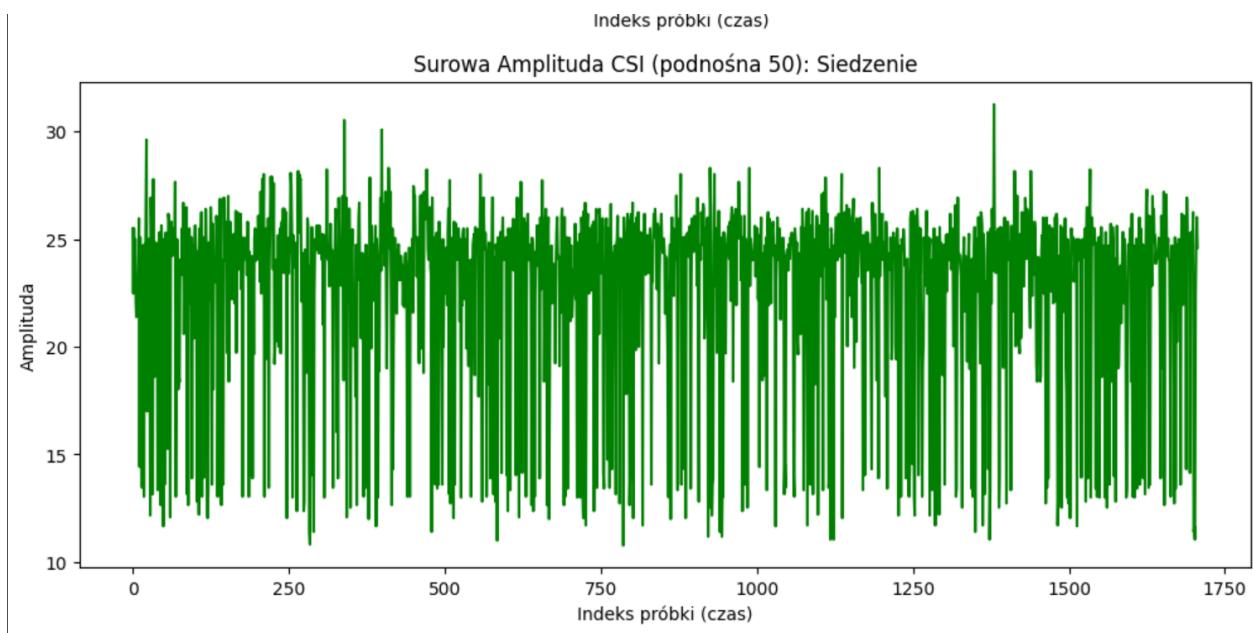
Ciekawym zjawiskiem było zaobserwowanie, że model nie rozpoznawał poprawnie żadnych z aktywności, jednak kiedy znajdowałam się pomiędzy antenami rozpoznawała “leżenie”, natomiast kiedy nie było mnie na linii pomiędzy antenami wykrywane było “siedzenie”.



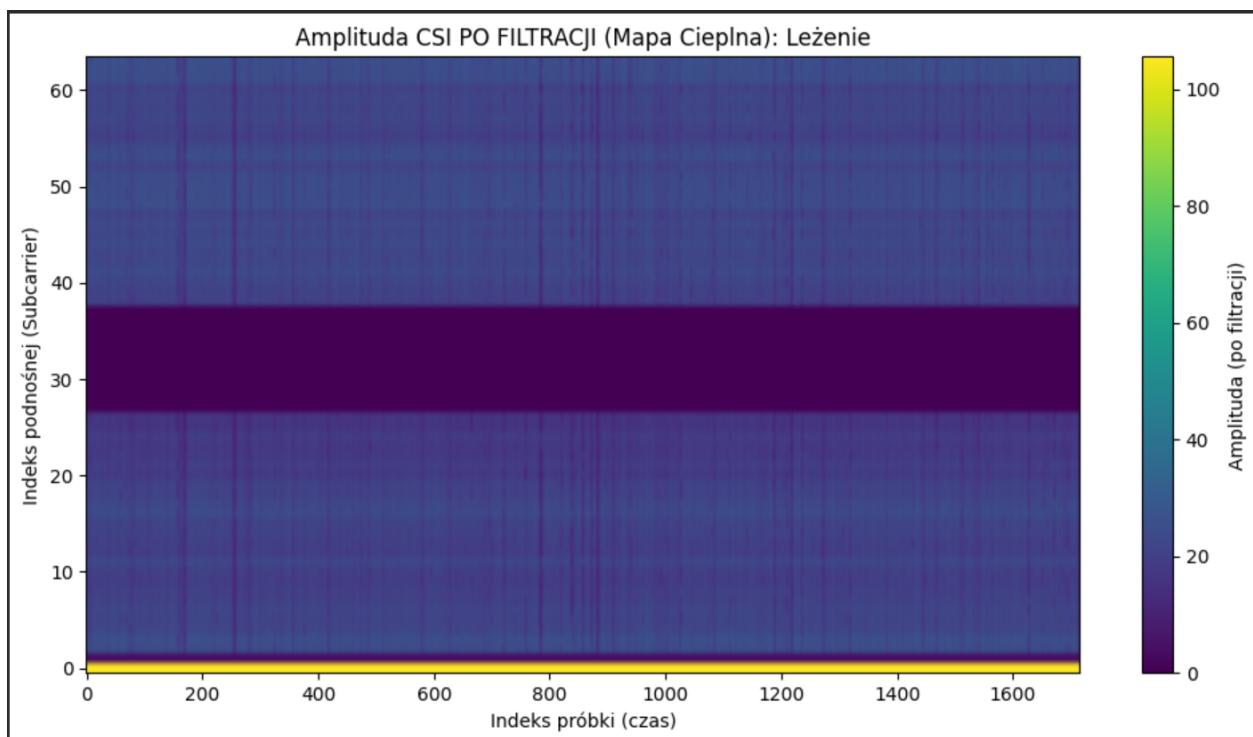
Rysunek 48: Wykres: Surowa amplituda, Aktywność: siedzenie, Badanie nr 2



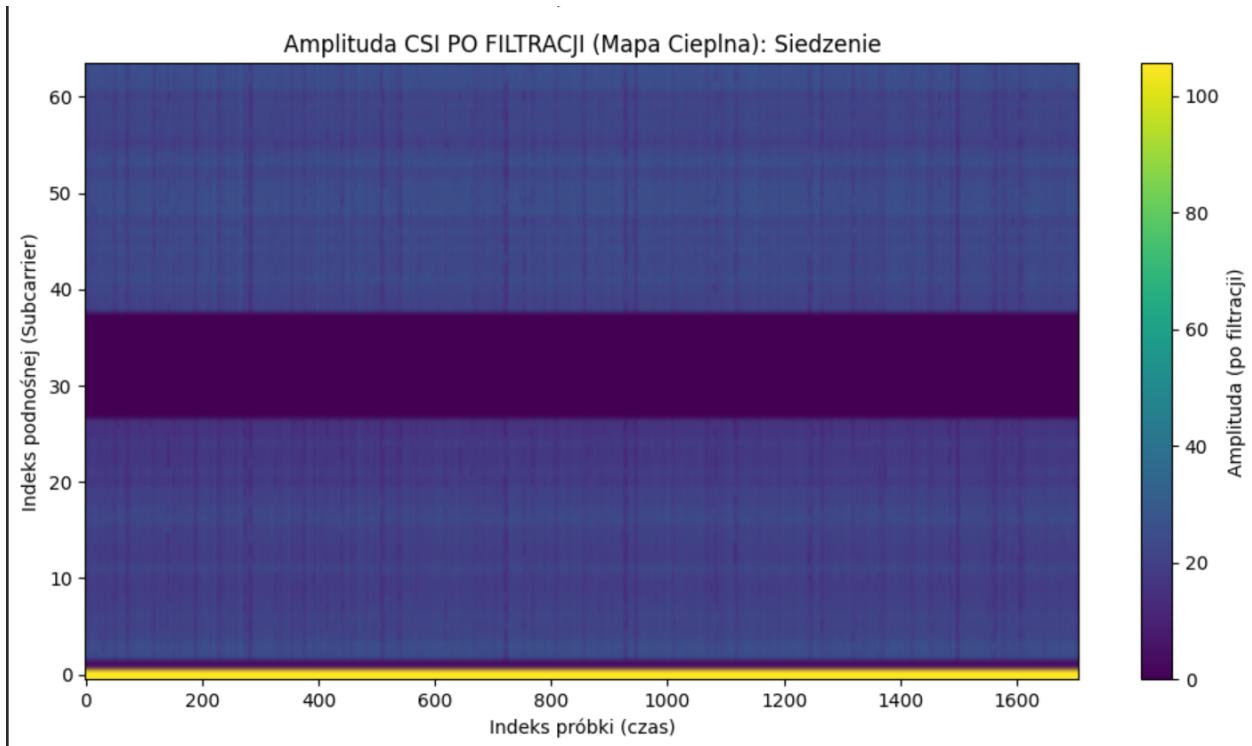
Rysunek 49: Wykres: Podnośna nr 50 (surowa), Aktywność: leżenie, Badanie nr 2



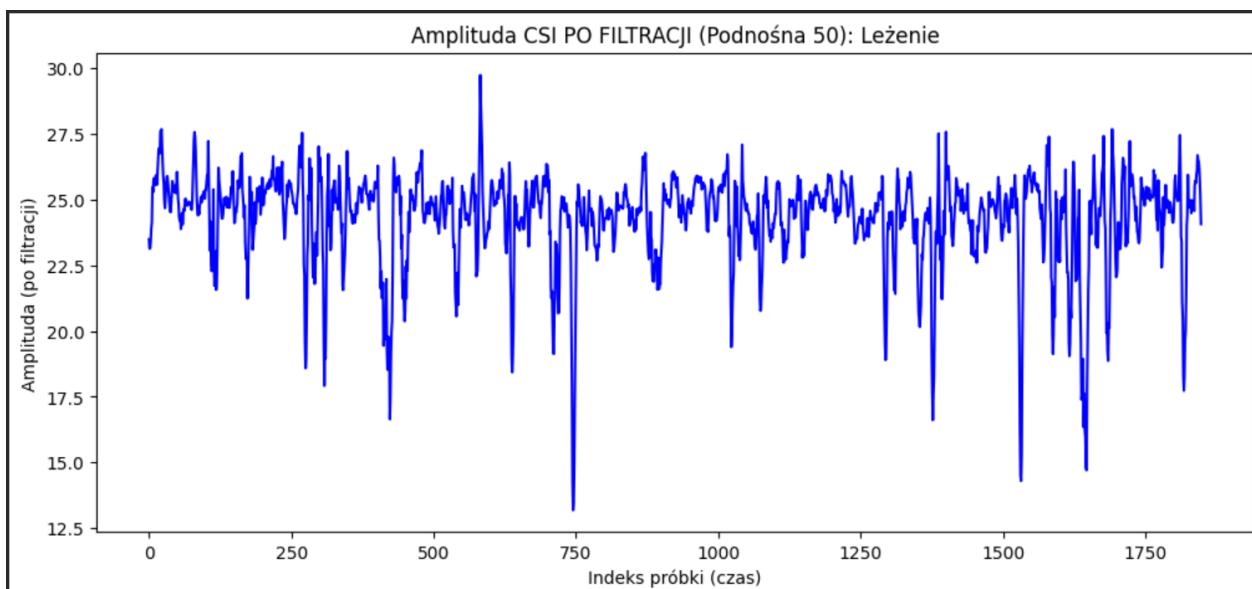
Rysunek 50: Wykres: Podnośna nr 50 (surowa), Aktywność: siedzenie, Badanie nr 2



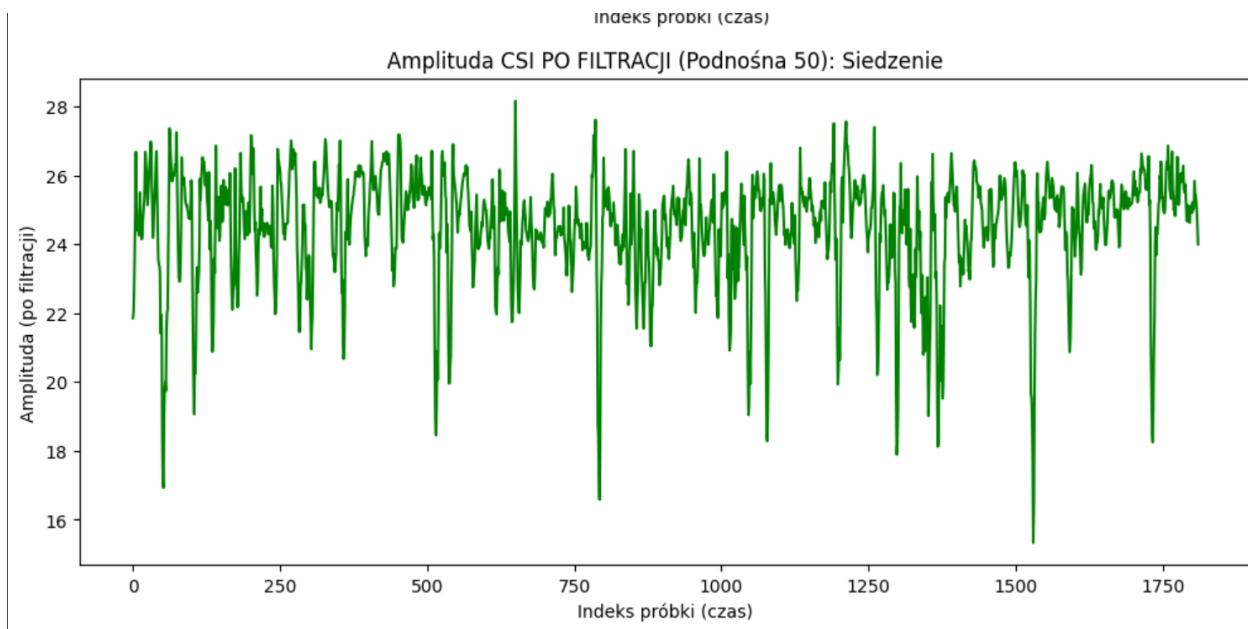
Rysunek 51: Wykres: Po filtracji, Aktywność: leżenie, Badanie nr 2



Rysunek 52: Wykres: Po filtracji, Aktywność: siedzenie, Badanie nr 2



Rysunek 53: Wykres: Podnośna nr 50 (po filtracji), Aktywność: leżenie, Badanie nr 2



Rysunek 54: Wykres: Podnośna nr 50 (po filtracji), Aktywność: siedzenie, Badanie nr 2

**Badanie 3:** Po dwóch nieudanych badaniach, gdzie fragmentowanie kładzenie się i siadanie uznałam, że spróbuję po prostu osobno pobrać próbki, gdy tylko leżę i osobno pobrać próbki, gdy tylko siedzę. Doszłam do wniosku, że w projekcie ma zostać wykryte, że pacjent usiadł i to jest ważniejsze do rozpoznania niż sama czynność siadania lub kładzenia się.

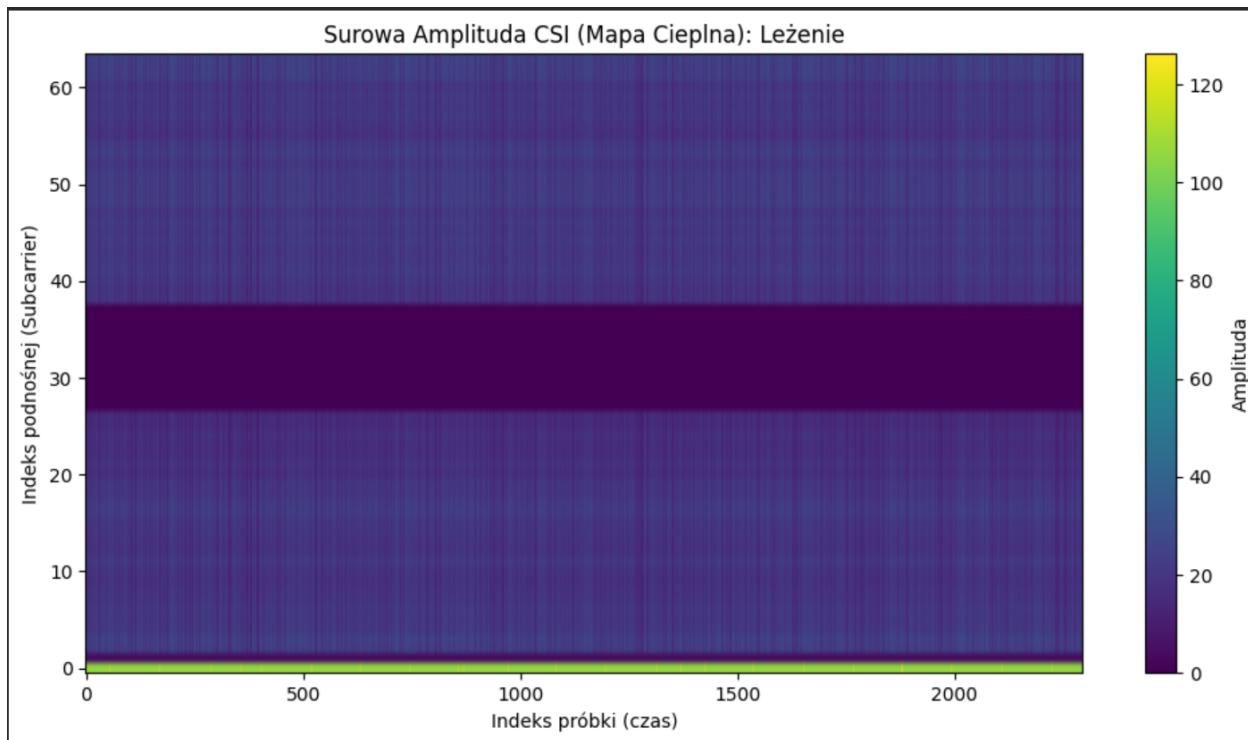
## Trening

W treningu wykorzystałam ok. 4500 próbek (po 2750 na każdą aktywność). Po segmentacji łącznie wygenerowano 907 próbek treningowych (456 dla leżenia i 451 dla siedzenia). Zbiór treningowy wynosił 725 próbek, a testowy 182. Zaobserwowałam, że model BiLSTM osiągnął 99% dokładności i zakończył trening na 14 epoce dzięki zastosowaniu early stoppingu.

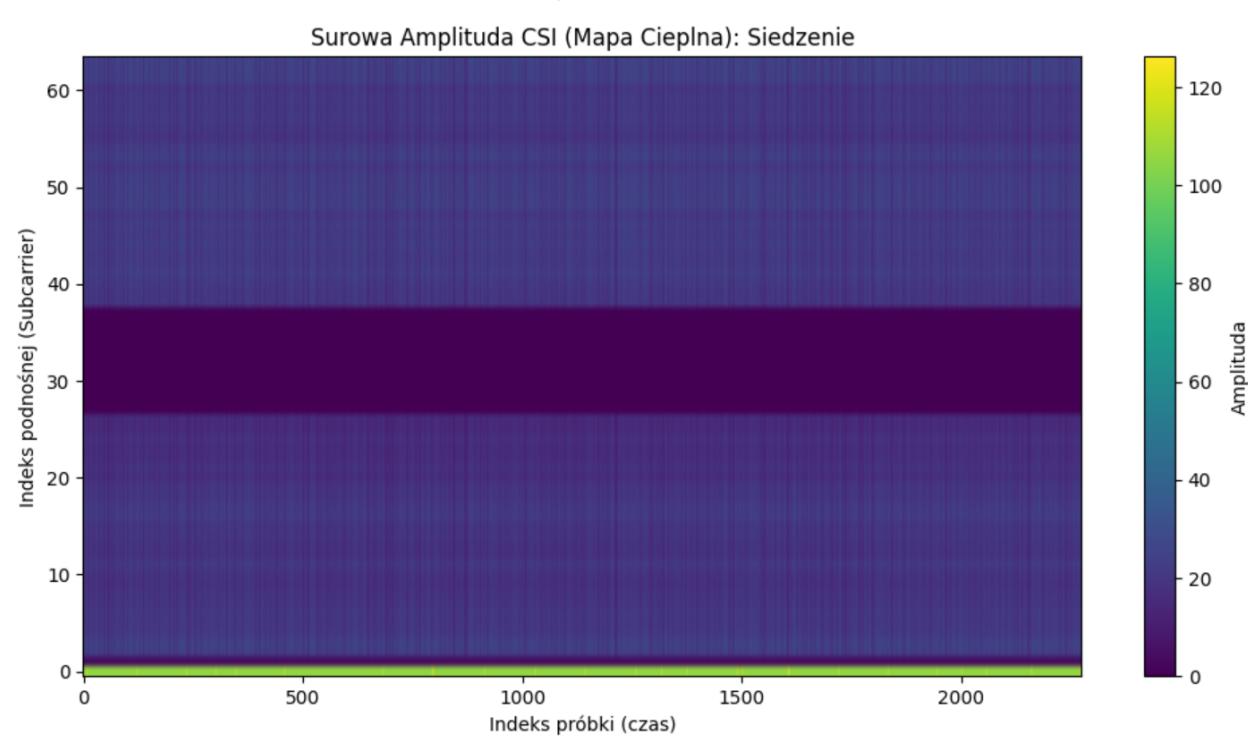
## Wyniki

Badanie zakończyło się sukcesem. W przeprowadzonym scenariuszu wykryto:

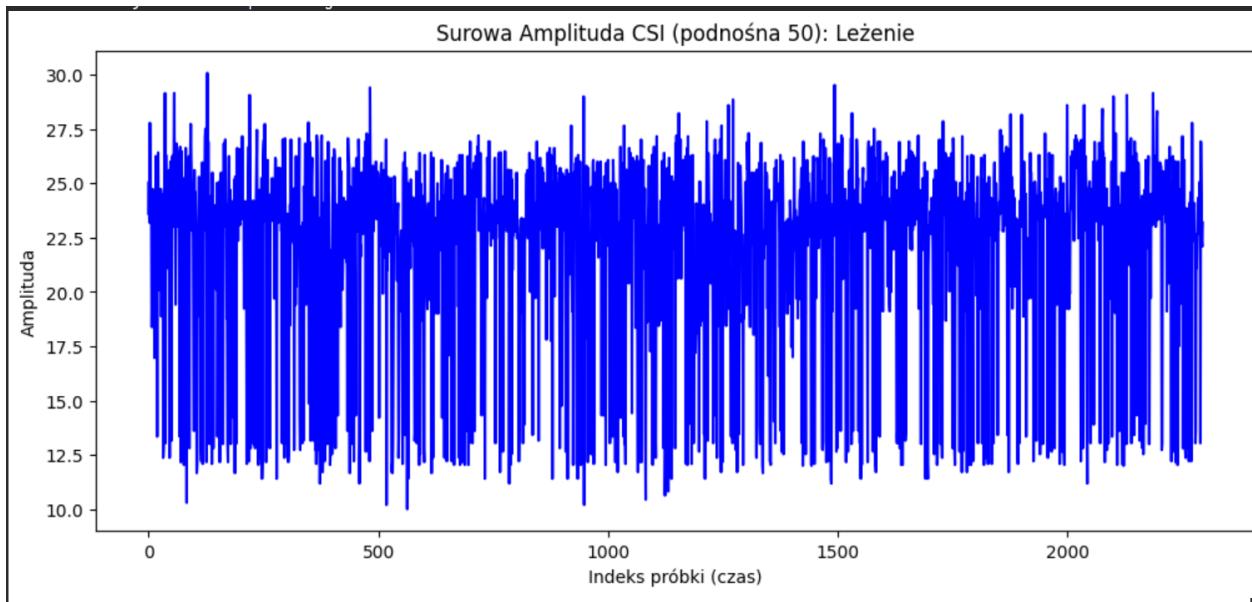
1. Siedzenie 2/2
2. Leżenie 3/3
3. Siedzenie 4/4
4. Leżenie 3/3



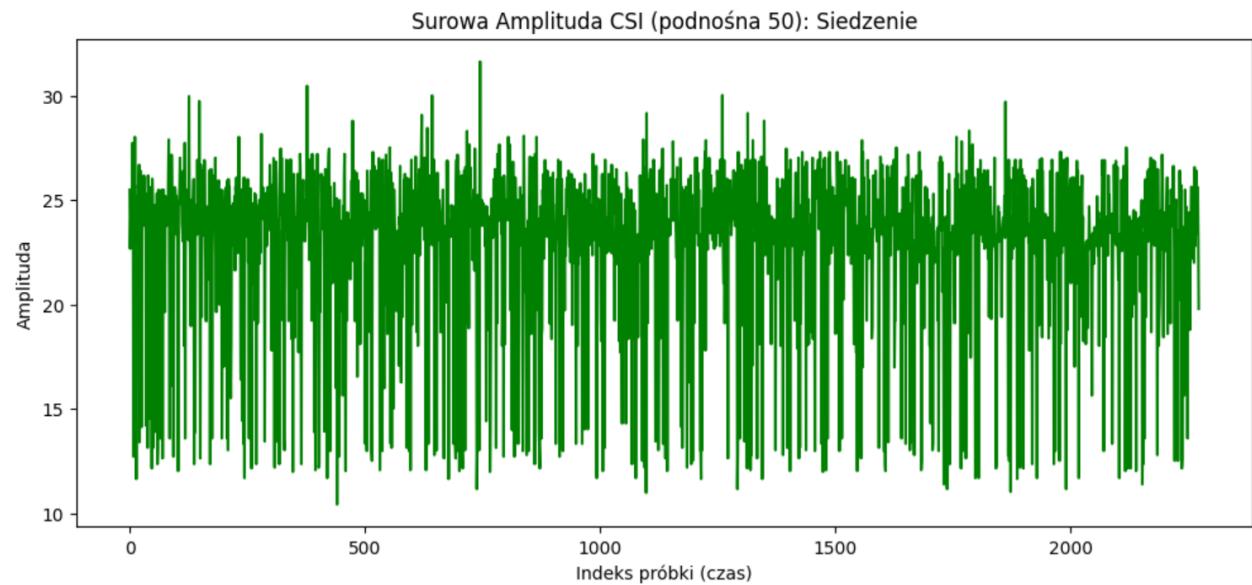
Rysunek 55: Wykres: Surowa amplituda, Aktywność: leżenie, Badanie nr 3



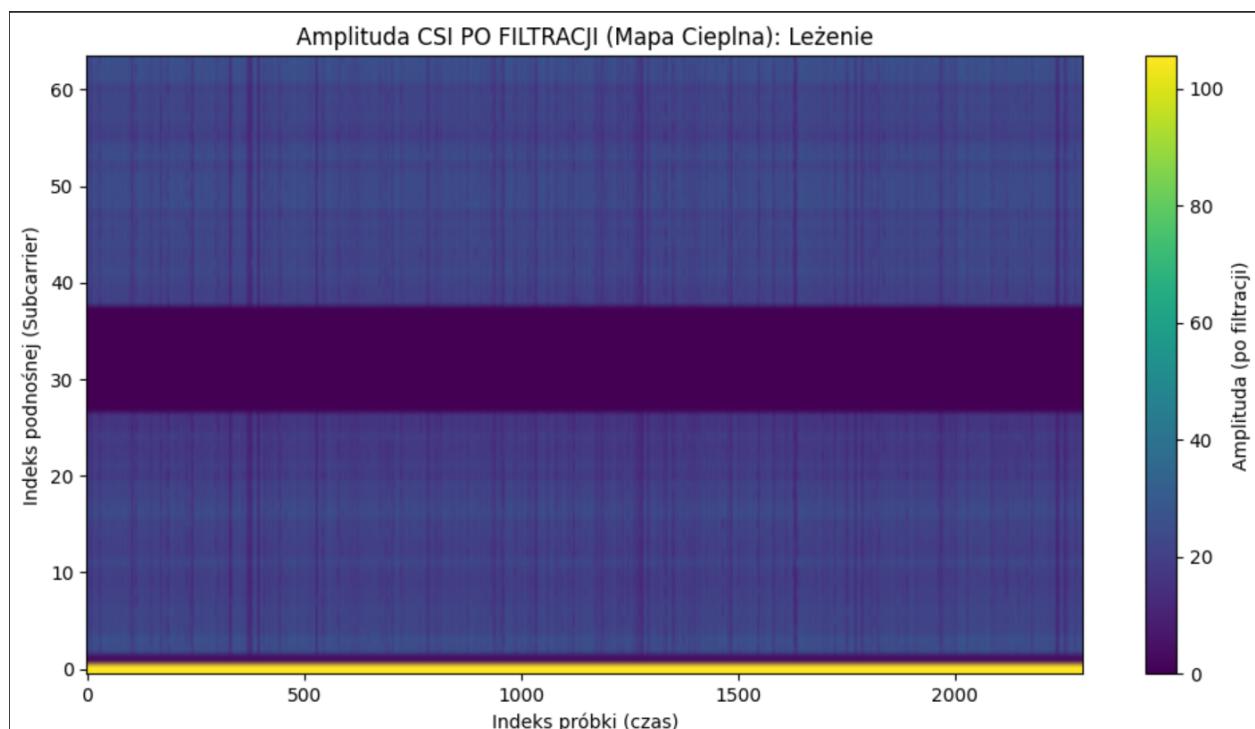
Rysunek 56: Wykres: Surowa amplituda, Aktywność: siedzenie, Badanie nr 3



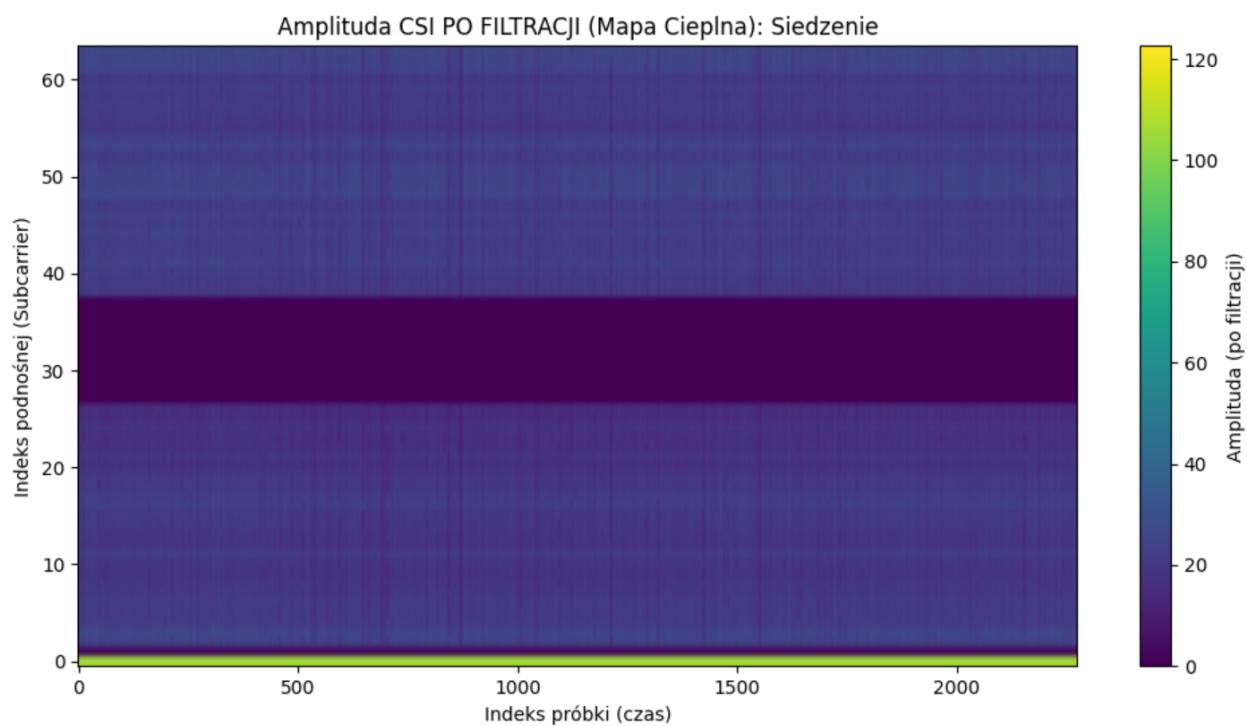
Rysunek 57: Wykres: Podnośna nr 50 (surowa), Aktywność: leżenie, Badanie nr 3



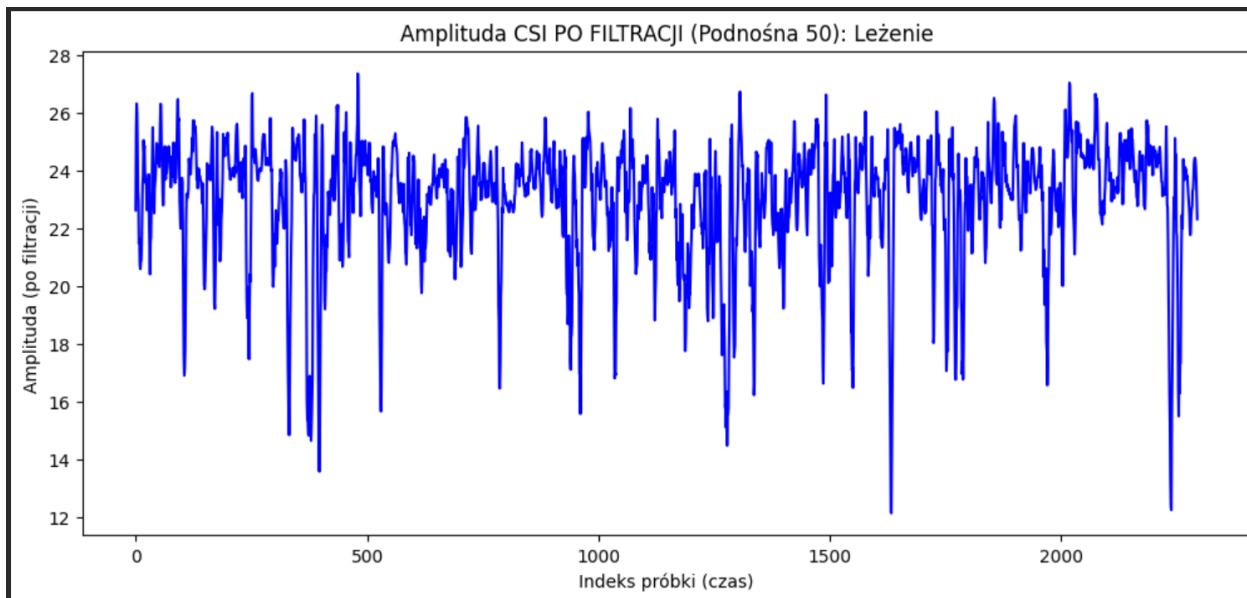
Rysunek 58: Wykres: Podnośna nr 50 (surowa), Aktywność: siedzenie, Badanie nr 3



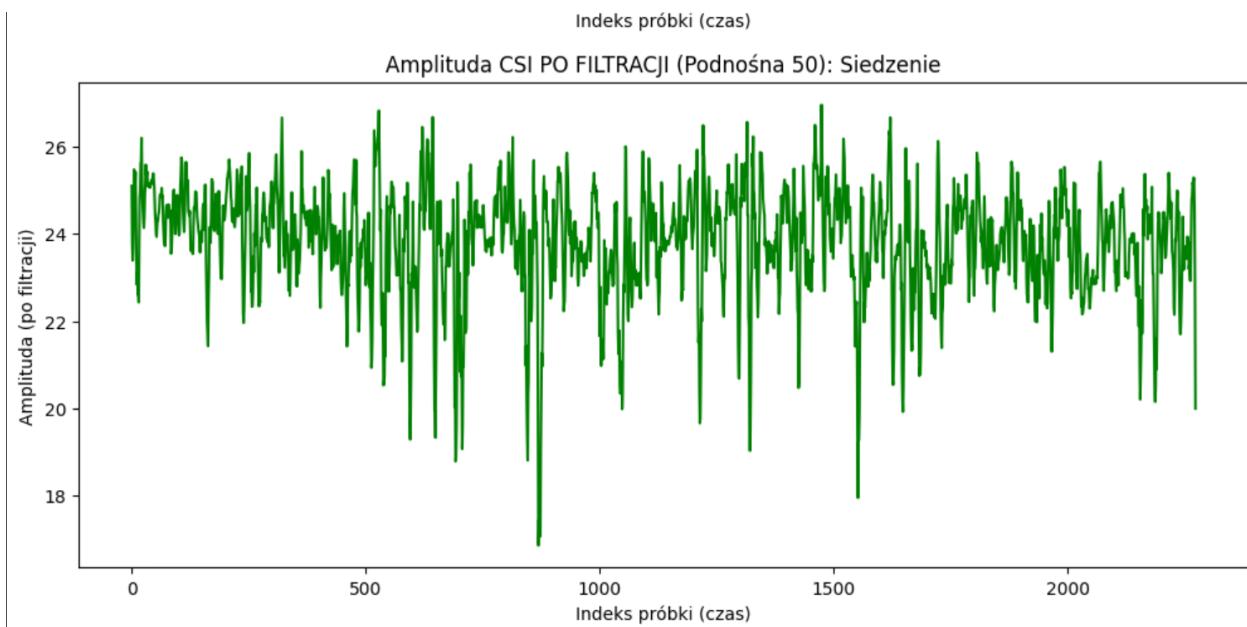
Rysunek 59: Wykres: Po filtracji, Aktywność: leżenie, Badanie nr 3



Rysunek 60: Wykres: Po filtracji, Aktywność: siedzenie, Badanie nr 3



Rysunek 61: Wykres: Podnośna nr 50 (po filtracji), Aktywność: leżenie, Badanie nr 3



Rysunek 62: Wykres: Podnośna nr 50 (po filtracji), Aktywność: siedzenie, Badanie nr 3