Przetwarzanie danych z wykorzystaniem GPU

Projekt Detekcja Mowy

Styczeń 2026

Mikołaj Rosowski
165313

1. Wstęp

Celem projektu były przygotowanie aplikacji, która na podstawie wejściowego sygnału audio wykrywać predefiniowane słowa.

Podstawowymi elementami projektu było:

- Przygotowanie modelu z wykorzystaniem biblioteki `PyTorch` pozwalającego na wykrywanie mowy
- Skwantyzowanie modelu i porównanie wyników
- Przygotowanie aplikacji w C++, która wykorzystuje przygotowany model za pomocą biblioteki `ExecuTorch`

Projekt dostępny jest na platformie GitHub pod adresem https://github.com/rosowskimik/subcommands-put

2. Zbiór Danych

Do treningu modelu został wykorzystany zbiór `Speech Commands` (https://arxiv.org/abs/1804.03209) w wersji `v0.02`. Zbiór ten zawiera pliki audio w formacie `.wav`. Pliki te są podzielone na zbiór konkertnych komend. Każdy plik zawiera 1 sekundę nagrania wymowy jednego wyrazu w języku angielskim lub szum tła. Słowa są wymawiane przez różne osoby. Dodatkowo do każdego nagrania przypisany jest numer identyfikacyjny mówiącego `speaker_id`, indeks wymowy `utterance_id` na każdego mówiącego, etykieta z wymawianym słowem `label`, wartość logiczna czy wypowiadane słowo jest uznawane za prostą komendę `is_unknown` oraz częstotliwość próbkowania pliku audio `sampling_rate` (Program 1).

```
{
  "file": "no/7846fd85_nohash_0.wav",
  "audio": {
    "path": "no/7846fd85_nohash_0.wav",
    "array": array([ -0.00021362, -0.00027466, -0.00036621, ...,  0.00079346,
          0.00091553,  0.00079346]),
    "sampling_rate": 16000
    },
  "label": 1,  # "no"
  "is_unknown": False,
  "speaker_id": "7846fd85",
  "utterance_id": 0
}
```

Program 1: Przykład danych dla jednej z wymów słowa *„no”*

Cały zbiór danych (w wersji `0.02`) zawiera 105829 plików audio, podzielonych na 36 etykiet:

- „Backward”
- „Bed”
- „Bird”
- „Cat”
- „Dog”
- „Down”
- „Eight”
- „Five”
- „Follow”
- „Forward”
- „Four”
- „Go”
- „Happy”
- „House”
- „Learn”
- „Left”
- „Marvin”
- „Nine”
- „No”
- „Off”
- „On”
- „One”
- „Right”
- „Seven”
- „Sheila”
- „Six”
- „Stop”
- „Three”
- „Tree”
- „Two”
- „Up”
- „Visual”
- „Wow”
- „Yes”
- „Zero”

Dodatkowo istnieje także specjalna etykieta `_silence_`. Nagrania oznaczone tą etykietą są albo nagraniami albo matematyczną symulacją szumu.

Zbiór `Speech Commands v0.02` jest dostarczany z podziałem na trzy rozłączne podzbiory: treningowy (`training`), walidacyjny (`validation`) oraz testowy (`testing`). W projekcie wykorzystano dokładnie ten podział udostępniany przez dataset, bez ręcznego mieszania przykładów pomiędzy subsetami. Liczność przykładów w poszczególnych częściach wynosiła odpowiednio:

- `train`: 84848
- `validation`: 9982
- `testing`: 4890

3. Transformacja danych

Zbiór `Speech Commands` zawiera surowe nagrania w formacie `.wav`, których nie da się bezpośrednio podać na wejście klasycznego modelu konwolucyjnego. Z tego powodu każdy przykład został przekształcony

do postaci dwuwymiarowej reprezentacji czas–częstotliwość (spektrogram Mel), o stałym rozmiarze (Rysunek 1).

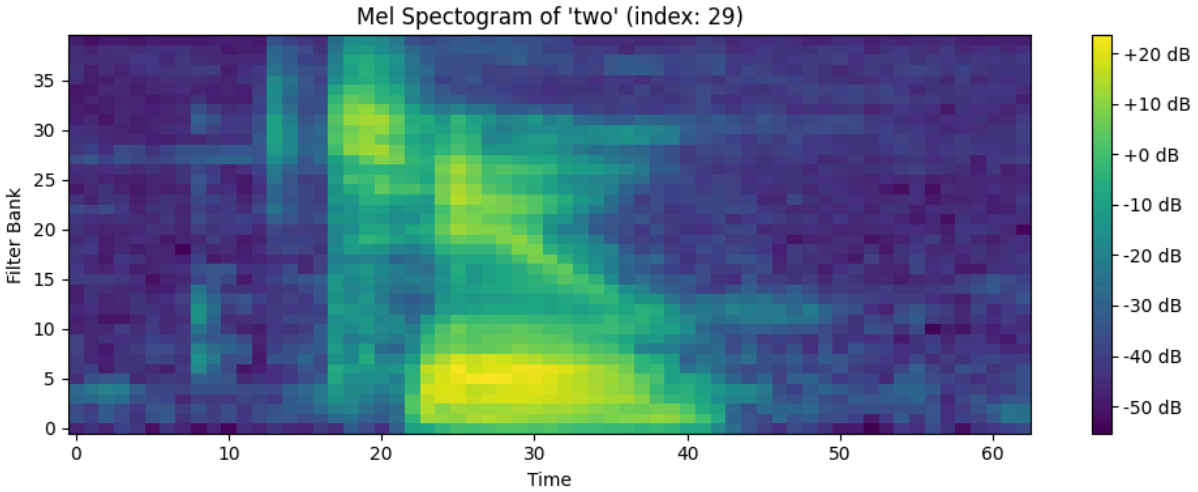
Dane zostały wczytane nie bezpośrednio z surowego archiwum pobranego ze strony źródłowej, lecz z wykorzystaniem klasy `torchaudio.datasets.SPEECHCOMMANDS`, czyli przygotowanego pod PyTorcha wrappera na ten sam zbiór danych. Zapewnia on automatyczne pobranie i rozpakowanie zbioru oraz zwraca próbki w postaci tensora `waveform` wraz z metadanymi (m.in. `sample_rate`, `label`, `speaker_id`, `utterance_number`), co upraszcza dalsze przetwarzanie i trening.

Dla każdego pliku audio wykonywane były następujące kroki:

- **Resampling:** jeżeli częstotliwość próbkowania różniła się od docelowej, sygnał był przeliczany do `16 kHz`.
- **Ujednolicenie długości:** nagrania były przycinane lub dopełniane zerami do dokładnie 1 sekundy (`16000` próbek). Zapewnia to stały rozmiar wejścia niezależnie od konkretnego pliku.
- **Wyznaczenie spektrogramu Mel:** z tak przygotowanego sygnału wyliczany był spektrogram Mel z parametrami: `n_mels = 40`, `n_fft = 512`, `hop_length = 256` (co dla 1 sekundy daje `63` ramek czasowych).
- **Skala logarytmiczna:** wartości mocy zostały przekształcone do skali decybelowej (logarytmicznej) za pomocą `AmplitudeToDB`, co poprawia interpretowalność i zwykle ułatwia uczenie (duże różnice amplitud są „spłaszczane").
- **Etykietowanie:** etykieta tekstowa słowa była mapowana na indeks klasy (liczbę całkowitą) zgodnie z listą `labels`.

Aby przyspieszyć kolejne uruchomienia i uniknąć wielokrotnego przeliczania tych samych transformacji, przetworzone dane były zapisywane do pamięci podręcznej (cache) w postaci plików `.pt`. Każdy plik zawierał parę: `mel_spec` o rozmiarze `(40, 63)` oraz `label` jako indeks klasy. Następnie podczas treningu modelu wykorzystywany był własny `Dataset`, który wczytywał gotowe spektrogramy z cache, a `DataLoader` łączył je w batch o rozmiarze `(B, 40, 63)`.

Dodatkowo ze zbioru treningowego wydzielona została niewielka część danych kalibracyjnych (ok. 10%), która została wykorzystana na etapie kalibracji podczas kwantyzacji post-training (PTQ).



Rysunek 1: Wizualizacja spektogramu Mel po transformacji

4. Architektura modelu

W projekcie wykorzystano niewielką sieć konwolucyjną typu CNN przeznaczoną do klasyfikacji słów na podstawie reprezentacji czas–częstotliwość (spektrogramu Mel). Architektura była inspirowana modelem do klasyfikacji audio prezentowanymi w dokumentacji MathWorks (Deep Learning, przykłady dla rozpoznawania komend głosowych).

Wejściem modelu jest macierz spektrogramu o rozmiarze `(40, 63)` (liczba banków Mel × liczba ramek czasowych). Ponieważ warstwy konwolucyjne 2D oczekują kanału, dane są traktowane jako obraz jednokanałowy (kanał o rozmiarze 1).

Model składa się z pięciu bloków konwolucyjnych. Każdy blok ma postać: `Conv2d(3×3) → BatchNorm2d → ReLU`, po którym wykonywane jest uśredniające próbkowanie w dół `AvgPool2d(2)`. Liczba kanałów rośnie w pierwszych etapach (1→12→24→32), a następnie pozostaje stała na poziomie 32. Taka konstrukcja pozwala stopniowo redukować wymiar przestrzenny (czas i częstotliwość) aż do reprezentacji `1×1`, którą następnie można spłaszczyć do wektora cech i podać na warstwę klasyfikującą.

Ostatnią warstwą jest `Linear`, która mapuje 32 cechy na 37 klas (liczba etykiet w zbiorze danych). Cały model jest bardzo lekki (ok. 29.5 tys. parametrów), co ułatwia późniejszą kwantyzację oraz uruchamianie na urządzeniach o ograniczonych zasobach.

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1          [-1, 12, 40, 63]             108
       BatchNorm2d-2          [-1, 12, 40, 63]              24
              ReLU-3          [-1, 12, 40, 63]               0
        ConvBnReLU-4          [-1, 12, 40, 63]               0
         AvgPool2d-5          [-1, 12, 20, 31]               0
            Conv2d-6          [-1, 24, 20, 31]           2,592
       BatchNorm2d-7          [-1, 24, 20, 31]              48
              ReLU-8          [-1, 24, 20, 31]               0
        ConvBnReLU-9          [-1, 24, 20, 31]               0
        AvgPool2d-10          [-1, 24, 10, 15]               0
           Conv2d-11          [-1, 32, 10, 15]           6,912
      BatchNorm2d-12          [-1, 32, 10, 15]              64
             ReLU-13          [-1, 32, 10, 15]               0
       ConvBnReLU-14          [-1, 32, 10, 15]               0
        AvgPool2d-15            [-1, 32, 5, 7]               0
           Conv2d-16            [-1, 32, 5, 7]           9,216
      BatchNorm2d-17            [-1, 32, 5, 7]              64
             ReLU-18            [-1, 32, 5, 7]               0
       ConvBnReLU-19            [-1, 32, 5, 7]               0
        AvgPool2d-20            [-1, 32, 2, 3]               0
           Conv2d-21            [-1, 32, 2, 3]           9,216
      BatchNorm2d-22            [-1, 32, 2, 3]              64
             ReLU-23            [-1, 32, 2, 3]               0
       ConvBnReLU-24            [-1, 32, 2, 3]               0
        AvgPool2d-25            [-1, 32, 1, 1]               0
           Linear-26                  [-1, 37]           1,221
================================================================
Total params: 29,529
Trainable params: 29,529
Non-trainable params: 0
```

Program 2: Struktura modelu

5. Proces kwantyzacji

6. Opis procesu kwantyzacji

Celem kwantyzacji było zmniejszenie rozmiaru modelu oraz przygotowanie go do uruchomienia w aplikacji C++ z wykorzystaniem `ExecuTorch` i backendu `XNNPACK`. W projekcie zastosowano kwantyzację typu post-training (PTQ).

Proces kwantyzacji składał się z kilku etapów:

- **Przygotowanie modelu do kwantyzacji (tryb inference):** model został przełączony w tryb `eval()`. Dodatkowo wykonano "folding" warstw `BatchNorm2d` do poprzedzających je `Conv2d` (fuzja `Conv+BN`), co upraszcza graf obliczeń i jest korzystne w kontekście kwantyzacji.
- **Eksport grafu:** model FP32 został wyeksportowany za pomocą `torch.export` do postaci modułu o stałym grafie (GraphModule). Wykorzystano przykładowe wejście o rozmiarze `(B, 40, 63)` oraz dopuszczono dynamiczny rozmiar batcha w ustalonym zakresie (1..`batch_size`).
- **Konfiguracja kwantyzatora:** do kwantyzacji użyto narzędzi z `torchao` (pipeline PT2E) oraz kwantyzatora `XNNPACKQuantizer` z `executorch`. Zastosowano globalną konfigurację symetrycznej kwantyzacji (`get_symmetric_quantization_config()`), ukierunkowaną na generowanie modelu kompatybilnego z backendem XNNPACK.
- **Wstawienie obserwatorów i kalibracja:** w kroku `prepare_*` do grafu wstawiane są elementy zbierające statystyki (obserwatory), które pozwalają wyznaczyć parametry kwantyzacji (m.in. skale). Następnie wykonano kalibrację, na wydzielonym zbiorze kalibracyjnym (ok. 10% zbioru treningowego).
- **Konwersja do modelu INT8:** po zakończeniu kalibracji wykonano `convert_pt2e(...)`, co zamienia odpowiednie operacje w grafie na ich odpowiedniki działające na reprezentacji skwantyzowanej (INT8), zgodnie z wcześniej wyznaczonymi parametrami.

Otrzymany model INT8 został następnie wykorzystany w dalszym etapie eksportu do formatu `.pte`.

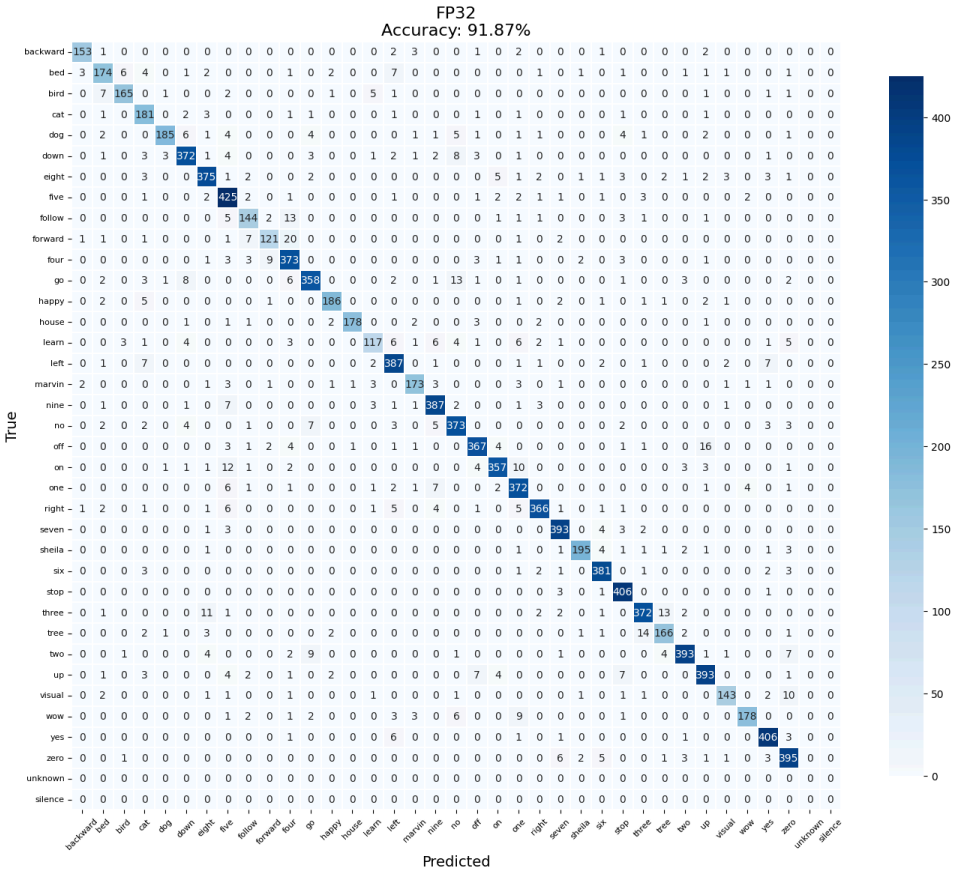
7. Porównanie modeli

Porównanie wykonano w trzech obszarach: skuteczność klasyfikacji, rozmiar modelu oraz wydajność inferencji.

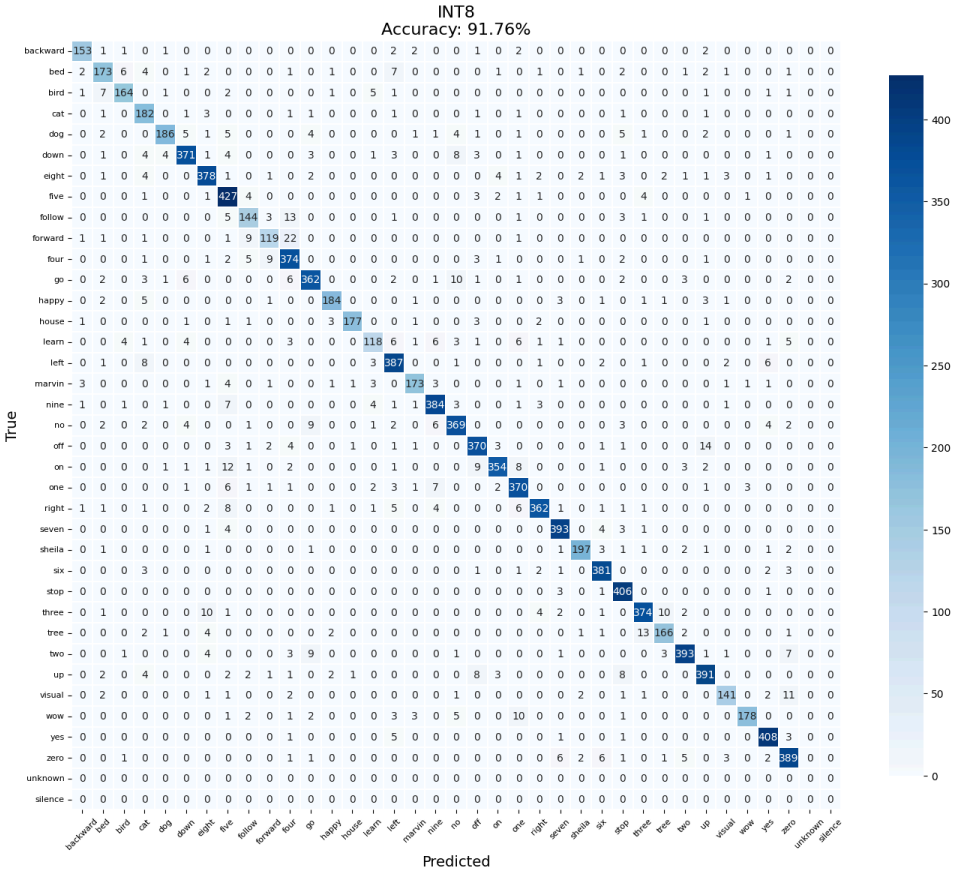
7.1. Skuteczność (accuracy)

Dla zbioru testowego oba modele osiągają bardzo zbliżone wyniki - 91.87% dla FP32 i 91.76% dla INT8. Model INT8 (Rysunek 3) ma nieznacznie niższą średnią skuteczność niż FP32 (Rysunek 2) (różnica jest mała i nie zmienia ogólnej jakości działania modelu).

Najczęściej występujące pomyłki powielają się dla modelu FP32 (Tabela 1) i INT8 (Tabela 2). Kwantyzacja wywołała niewielki zmiany wystąpień błędów (Rysunek 4, Tabela 3)



Rysunek 2: Macierz pomyłek modelu FP32 (zbiór testowy)



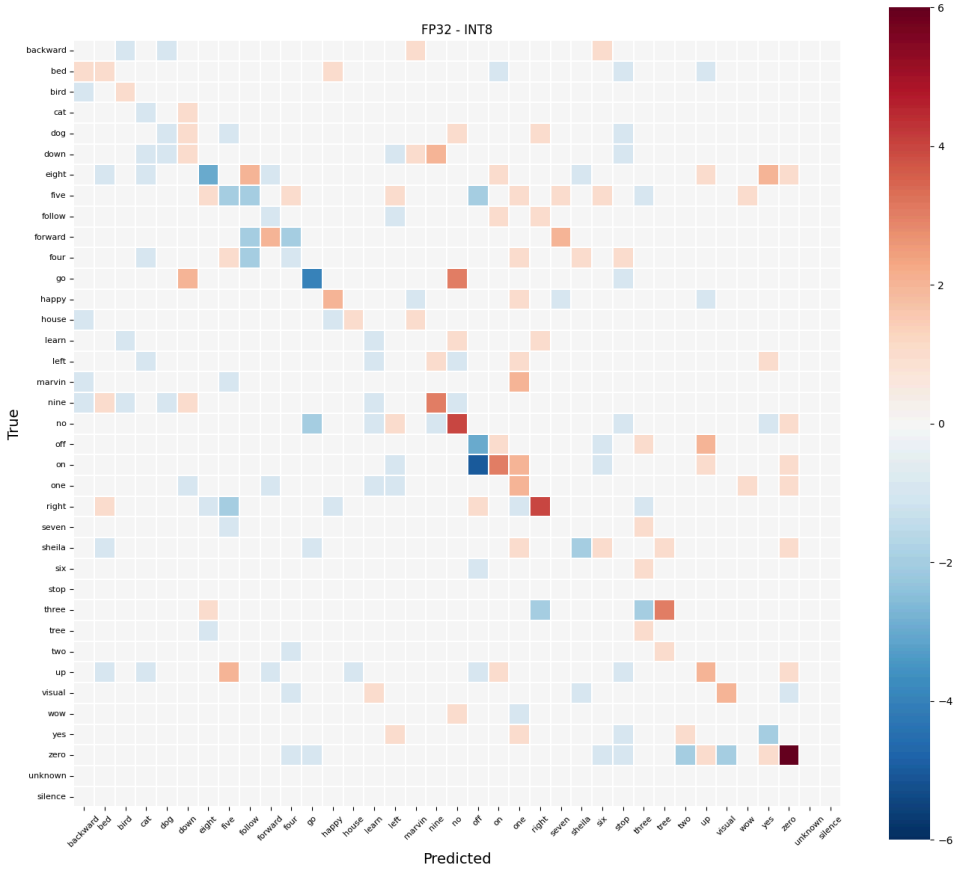
Rysunek 3: Macierz pomyłek modelu INT8 (zbiór testowy)

Liczba wystąpień	Prawdziwa Klasa	Przewidziana Klasa
20	forward	four
16	off	up
14	tree	three
13	follow	four
13	go	no
13	three	tree
12	on	five
11	three	eight
10	on	one
10	visual	zero
9	four	forward
9	two	go
9	wow	one
8	down	no
8	go	down

Tabela 1: Najczęstsze błędy modelu FP32

Liczba wystąpień	Prawdziwa Klasa	Przewidziana Klasa
22	forward	four
14	off	up
13	follow	four
13	tree	three
12	on	five
11	visual	zero
10	go	no
10	three	eight
10	three	tree
10	wow	one
9	forward	follow
9	four	forward
9	no	go
9	on	off
9	two	go

Tabela 2: Najczęstsze błędy modelu INT8



Rysunek 4: Macierz różnic (INT8 - FP32) dla zliczeń błędów; wartości dodatnie oznaczają częstsze występowanie pomyłki po kwantyzacji

Zmiana wystąpień błędów	**Prawdziwa Klasa**	**Przewidziana Klasa**	**Wystąpienia FP32**	**Wystąpienia INT8**
5	on	off	4	9
2	five	follow	2	4
2	five	off	1	3
2	forward	follow	7	9
2	forward	four	20	22
2	four	follow	3	5
2	no	go	7	9
2	right	five	6	8
2	three	right	2	4
2	zero	two	3	5
2	zero	visual	1	3
1	backward	bird	0	1
1	backward	dog	0	1
1	bed	on	0	1
1	bed	stop	1	2

Tabela 3: Zmiana występowań błędów po kwantyzacji

Klasa	Liczba sampli	Poprawnych (FP32)	%Poprawnych (FP32)	Liczba sampli	Poprawnych (INT8)	%Poprawnych (INT8)	%Różnica
zero	418	395	94.5	418	389	93.06	-1.44
forward	155	121	78.06	155	119	76.77	-1.29
visual	165	143	86.67	165	141	85.45	-1.21
right	396	366	92.42	396	362	91.41	-1.01
no	405	373	92.1	405	369	91.11	-0.99
happy	203	186	91.63	203	184	90.64	-0.99
on	396	357	90.15	396	354	89.39	-0.76
nine	408	387	94.85	408	384	94.12	-0.74
bird	185	165	89.19	185	164	88.65	-0.54
house	191	178	93.19	191	177	92.67	-0.52
one	399	372	93.23	399	370	92.73	-0.5
bed	207	174	84.06	207	173	83.57	-0.48
up	425	393	92.47	425	391	92.0	-0.47
down	406	372	91.63	406	371	91.38	-0.25
follow	172	144	83.72	172	144	83.72	0.0
left	412	387	93.93	412	387	93.93	0.0
two	424	393	92.69	424	393	92.69	0.0
seven	406	393	96.8	406	393	96.8	0.0
six	394	381	96.7	394	381	96.7	0.0
stop	411	406	98.78	411	406	98.78	0.0
tree	193	166	86.01	193	166	86.01	0.0
marvin	195	173	88.72	195	173	88.72	0.0
wow	206	178	86.41	206	178	86.41	0.0
backward	165	153	92.73	165	153	92.73	0.0
four	400	373	93.25	400	374	93.5	0.25
five	445	425	95.51	445	427	95.96	0.45
dog	220	185	84.09	220	186	84.55	0.45
yes	419	406	96.9	419	408	97.37	0.48
three	405	372	91.85	405	374	92.35	0.49
cat	194	181	93.3	194	182	93.81	0.52
learn	161	117	72.67	161	118	73.29	0.62
eight	408	375	91.91	408	378	92.65	0.74
off	402	367	91.29	402	370	92.04	0.75
sheila	212	195	91.98	212	197	92.92	0.94
go	402	358	89.05	402	362	90.05	1.0

Tabela 4: Różnica skuteczności per-klasa

Na poziomie poszczególnych etykiet widać, że kwantyzacja nie wpływa jednakowo na wszystkie klasy (Tabela 4). Mimo minimalnego spadku średniej accuracy, model po kwantyzacji może lokalnie poprawiać rozpoznawanie niektórych etykiet, kosztem innyc.

7.2. Rozmiar modelu

Rozmiar plików programu ExecuTorch (`.pte`) wynosił:

- FP32: 130 580 bajtów,
- INT8: 45 972 bajtów.

Oznacza to zmniejszenie rozmiaru o ok. 64.8% (model INT8 jest ok. 2.84x mniejszy od FP32). Jest to istotne z punktu widzenia docelowego uruchamiania modelu w aplikacji C++ oraz potencjalnego wdrażania na urządzeniach o ograniczonej pamięci.

7.3. Wydajność inferencji

Pomiary wydajności wykonano w aplikacji C++ z użyciem modeli:
- `tiny.pte` (FP32),
- `tiny_qt.pte` (INT8).

Test polegał na wykonaniu 50 rund rozgrzewki oraz 100000 inferencji na tym samym wejściu (pojedynczy przykład danych) (Tabela 5).

Model	min [ns]	avg [ns]	max [ns]
FP32 (`tiny.pte`)	201 639	216 591	3 637 681
INT8 (`tiny_qt.pte`)	205 119	217 850	3 951 459

Tabela 5: Porównanie wydajności

Średni czas inferencji obu modeli jest bardzo zbliżony (ok. 0.217 ms na jedno uruchomienie). W tym pomiarze model INT8 okazał się minimalnie wolniejszy (około 1.26 µs różnicy w średniej), co może wynikać m.in. z narzutu backendu, formatu wag/aktywacji albo sposobu partycjonowania grafu.

Warto podkreślić, że ten benchmark mierzy czas samej inferencji modelu dla przygotowanego wejścia. W scenariuszu „na żywo" (mikrofon) całkowity koszt systemu obejmuje również wyznaczenie cech (np. spektrogramu Mel), buforowanie sygnału i obsługę I/O.

Mimo to, przy czasie rzędu 0.217 ms na inferencję, przetwarzanie w trybie zbliżonym do real-time (np. przesuwające się okno) jest jak najbardziej osiągalne na testowanej platformie: nawet dla kroku co 10 ms (100 inferencji/s) czas samej inferencji stanowi niewielką część dostępnego budżetu czasowego.

8. Problemy z Zephyrem i ExecuTorchem

Pirwotnym założeniem projektu było uruchomienie skwantyzowanego modelu na płytce Arduino Nano 33 BLE Sense z systemem Zephyr i ExecuTorchem. Płytka posiada mikrofon PDM, co pasowało do docelowego scenariusza „embedded". Po wciśnięciu przycisku (GPIO) urządzenie miało nagrać 1 sekundę audio, przekształcić sygnał do spektogramu, wykonać inferencję i wypisać wynik na uarcie. Przetwarzanie sygnału miało być zaimplementowane z pomocą biblioteki `cmsis-dsp`, co dało by znacznie lepsze rezultaty niż surowe C/C++ z powodu wykorzystania akceleracji sprzętowej / operacji wektorowych. Dodatkowo niektóre stałe (np. `hann_window`) miały być wyeksportowane z PyTorcha, co też przyśpieszyło by ten proces.

W praktyce wsparcie ExecuTorcha na Zephyrze okazało się być bardzo esperymentalne i w testowej konfiguracji nie udało się doprowadzić do poprawnej budowy aplikacji. Próby były wykonywane na ExecuTorchu w wersji `a577584f927fe082256e4b8be7e2b9ada27c10f4`.

Napotkane problemy to m.in:

- brak dobrej dokumentacji: jedyny dowód, że takie coś jest w ogóle możliwe to przykład arm z ExecuTorch'a, który nie buduje się out-of-the-box
- problem z symlinkami: symlinki w `executorch/src` powodowały błędy; obejściem było ręczne ich zastąpienie hard linkami.
- problemy z backendem arm (głównie TOSA)
- problemy z vendorowanymi zależnościami, np. próby linkowania dynamicznego ze strony build systemu

W ostateczności plan uruchomienia na płytce wraz z Zephyrem został odłożony z powodu braku czasu.

9. Wnioski

- **Skuteczność (accuracy):** kwantyzacja spowodowała jedynie minimalny spadek średniej skuteczności na zbiorze testowym: 91.45% (FP32) → 91.16% (INT8), tj. różnica ok. (−0.29) pp. Jednocześnie wpływ kwantyzacji nie był równomierny dla wszystkich klas: dla części etykiet zanotowano niewielkie spadki (np. `zero`, `forward`, `visual`), ale dla innych pojawiły się poprawy (np. `go`, `sheila`, `off`, `eight`). Oznacza to, że INT8 nie jest „gorszy wszędzie", lecz zmienia rozkład błędów.
- **Charakter błędów:** analiza macierzy pomyłek pokazała, że najczęstsze pomyłki obu modeli są bardzo podobne (np. `forward -> four`, `off -> up`, `tree -> three`). Różnice dotyczą głównie liczności wybranych par błędów. Przykładowo, po kwantyzacji wzrosła liczba pomyłek `on -> off` (z 4 do 9). Takie przypadki są dobrym kandydatem do dalszej analizy (np. sprawdzenia, czy próbki są akustycznie podobne albo czy problem wynika z cech wejściowych).
- **Rozmiar modelu:** największą korzyścią kwantyzacji okazała się redukcja rozmiaru modelu `.pte`: 130 580 B (FP32) -> 45 972 B (INT8), czyli ok. 64.78% mniej (około 2.84×). Jest to istotne z punktu widzenia wdrożeń na urządzeniach o ograniczonej pamięci oraz dystrybucji modeli.
- **Możliwość przetwarzania w czasie rzeczywistym:** przy czasie inferencji rzędu 0.217 ms na pojedyncze uruchomienie modelu, przetwarzanie z przesuwającym się oknem jest realne (np. przy kroku co 10 ms jest to ok. 100 inferencji/s, co nadal zostawia duży budżet czasowy). Należy jednak pamiętać, że w praktycznej aplikacji istotniejszy koszt najpewniej stanowić będzie przygotowanie cech (Mel-spektrogram) oraz obsługa buforowania i wejścia audio.

Podsumowując: kwantyzacja INT8 w badanym przypadku dała bardzo dużą redukcję rozmiaru modelu przy niemal niezmienionej skuteczności, natomiast nie przyniosła zauważalnego zysku szybkości w wykonanym benchmarku C++.

10. Wykorzystane źródła / biblioteki

- SPEECHCOMMANDS dataset - https://huggingface.co/datasets/google/speech_commands
- Pierwotne źródło modelu - https://www.mathworks.com/help/deeplearning/ug/deep-learning-speech-recognition.html
- PyTorch (v2.9.0) - https://github.com/pytorch/pytorch
- TorchAudio (v2.9.0) - https://github.com/pytorch/audio
- ExecuTorch (v1.0.1) - https://github.com/pytorch/executorch
- cxxopts (v3.3.1) - https://github.com/jarro2783/cxxopts