Digital Signal Processing Project Report

Development and Analysis of an Audio Signal Processing Model for Speech Command Recognition

Introduction

This project is all about building and exploring a system to process digital audio signals and recognize simple speech commands like "yes" and "no." We've put together a full pipeline: transforming raw audio into spectrograms, training a neural network, and digging into how key settings—like the number of mel-frequency filters (`n_mels`) and grouped convolutions (`groups`)—affect accuracy, computational cost, and performance. In this report, I'll walk you through what we did, how it works, and what we learned along the way.

1. Implementing the LogMelFilterBanks Class

What's It For?

The `LogMelFilterBanks` class is the backbone of our audio processing system. It turns raw audio signals into something more useful—logarithmic mel-spectrograms. Why mel-spectrograms? Because they mimic how humans hear: we're more tuned to differences in lower frequencies than higher ones. This class is a crucial step in preparing data for our neural network.

How Does It Work?

1. Step 1: Time-to-Frequency Transformation

We use Short-Time Fourier Transform (STFT) to break the time-based audio signal into a frequency-based picture. Think of it as splitting a song into the notes playing at each moment.

2. Step 2: Mel Scale

We convert linear frequencies (in Hertz) into the mel scale, which is nonlinear and aligns with human perception. For example, the jump from 100 Hz to 200 Hz feels bigger to us than from 5000 Hz to 5100 Hz.

3. Step 3: Log Compression

We apply a logarithm to the results. This squashes the dynamic range (from super quiet to super loud) and makes the data friendlier for machine learning.

What Did We Add?

- In `__init__`, we added the `f_max_hz` variable. If the user doesn't specify a max frequency, it defaults to the Nyquist frequency (`samplerate // 2`). That's half the sampling rate—beyond this point, there's no useful info in the signal. For example, with a 16000 Hz audio sample rate, the max is 8000 Hz.

- In `_init_melscale_fbanks`, we finished the logic for generating mel filters and made it return the result. It now calculates the right number of frequency bins (`n_freqs = n_fft // 2 + 1`). Why this formula? FFT gives a symmetric spectrum, so we only need half the coefficients plus the zero-frequency (DC) component.

2. Data Prep: The SubsetSC Class

What's This?

The `SubsetSC` class is a wrapper around the `SPEECHCOMMANDS` dataset from `torchaudio`. We filtered it down to just "yes" and "no" commands to keep our classification task simple.

What Does It Do?

- Filters the dataset to include only "yes" and "no."

- Splits it into three parts: training, validation, and test sets.

- Converts audio into mel-spectrograms with a customizable number of filters (`n_mels`).

- Standardizes all spectrograms to 100 frames—short ones get padded with zeros, long ones get trimmed.

- Turns labels into numbers: "no" = 0, "yes" = 1.

This class tidied up our data and got it ready for training.

3. The Model: SpeechCommandsModel Class

Architecture

We built a convolutional neural network (CNN) that looks at mel-spectrograms and figures out if someone said "yes" or "no." Here's the setup:

- Three convolutional blocks:

- First: 32 filters,

- Second: 64 filters,

- Third: 128 filters.

- Each block includes: convolution, batch normalization (BatchNorm), ReLU activation, and size reduction (MaxPooling).

- Support for grouped convolutions (`groups`)—splitting filters into groups to cut down on computation.

- Adaptive pooling to handle varying input lengths.

- A final fully connected layer for the two-class prediction.

Handy Features

- `count_parameters()` tallies up the trainable parameters.

- `calculate_flops()` estimates computational complexity (FLOPs).

4. Training and Validation: The train_model Function

How It Works

- Loads data via `DataLoader`.

- Sets up the model, optimizer (Adam), and loss function (CrossEntropyLoss).

- Trains the model over a set number of epochs (passes through the data).

- Checks accuracy on the validation set after each epoch.

- Finishes with a test on the test set.

What We Track

- Training loss.

- Validation and test accuracy.

- Training time.

- Number of parameters and FLOPs.

5. Experiments

Experiment 1: Impact of n_mels

We tested different `n_mels` values (20, 40, 80):

- Trained a model for each value.

- Plotted loss and accuracy over epochs.

- Saved the results in `n_mels_experiment.png`.

Takeaway: More mel filters help the model pick up sound details, but there's a sweet spot—too many just add complexity without much accuracy gain.

Experiment 2: Impact of groups

We tried different `groups` values (1, 2, 4, 8, 16):

- Trained models and compared training time, parameters, FLOPs, and accuracy.

- Results went into `groups_experiment.png`.

Takeaway: More groups cut down on computation and memory, but push it too far, and the model struggles to spot complex patterns.

6. Main Block (main)

- Verified that `LogMelFilterBanks` matches `torchaudio.transforms.MelSpectrogram`.

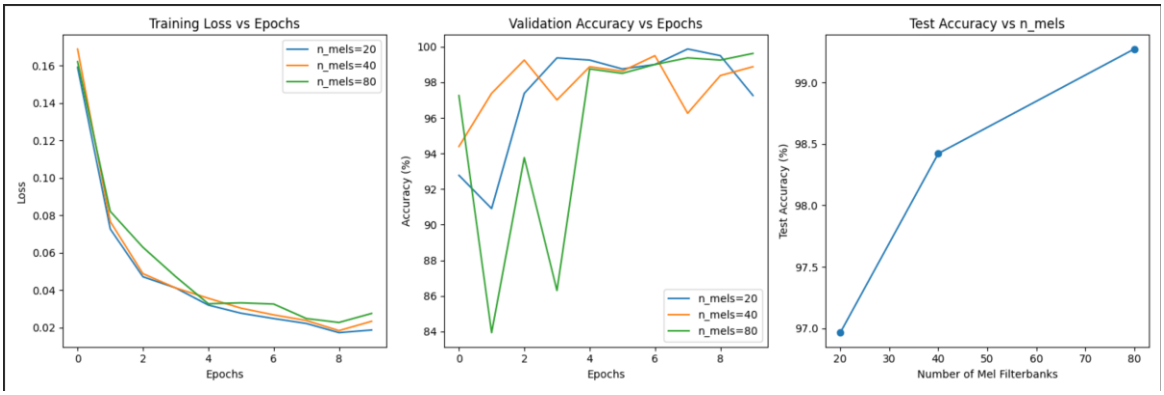
- Ran the `n_mels` experiment and picked the best value based on test accuracy.

- Used that value to run the `groups` experiment.

Goal: Strike a balance between accuracy, speed, and resource use.

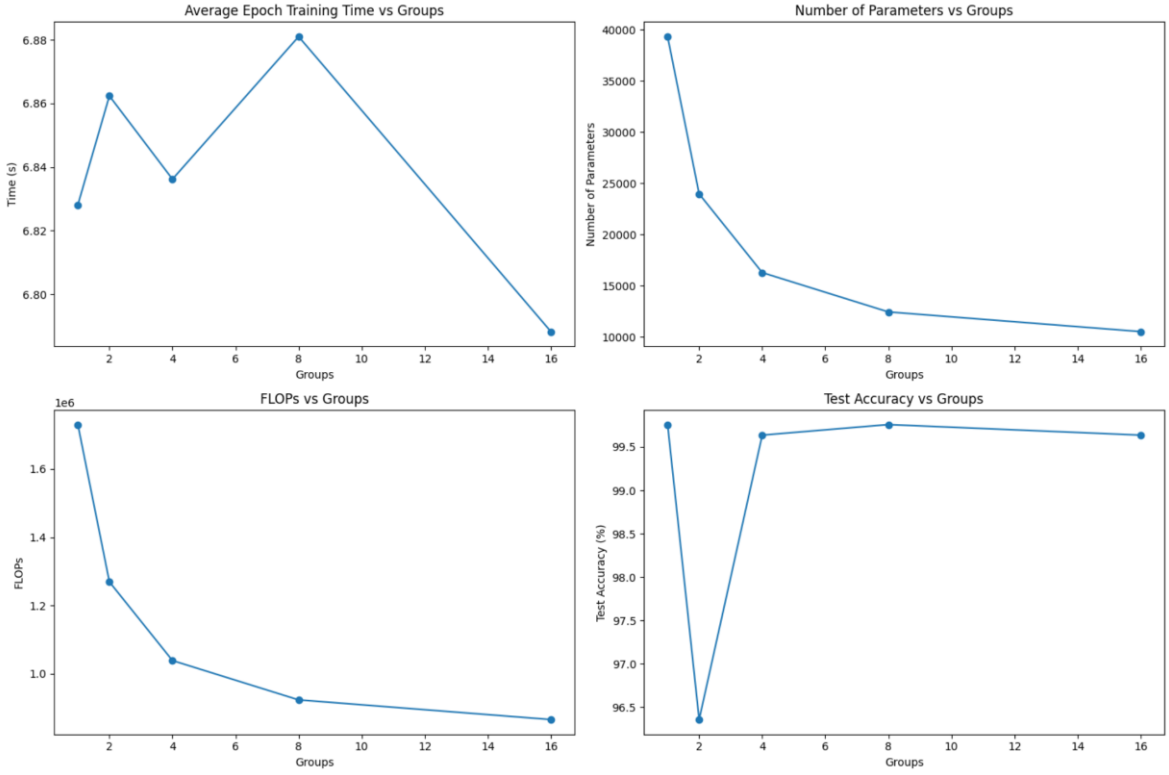
7. Results and Insights

From the n_mels Experiment



The graph showed accuracy improving as `n_mels` increased, but the gains tapered off after a point (say, 40 or 80). The best choice depends on your task and data—experimentation is key.

From the groups Experiment



- More groups = less computation and memory use.

- But there's a limit: too many groups hurt accuracy.

- Real-world speed-ups don't always match theoretical savings due to GPU quirks.

Bottom Line: Grouped convolutions are a powerful trick for optimization, especially on resource-tight devices like phones.