A Machine Learning Audio Pipeline

Beltrán Castro Gómez

November 3, 2025

Abstract

This brief report presents a machine learning audio pipeline developed in the context of a hiring process at Auphonic.

1 Introduction

The aim of this project is to implement a complete machine learning pipeline, including preprocessing, training, and evaluation. The project considers a simple audio classification task based on the AudioMNIST Dataset [1]. The said task consists in predicting a spoken digit from 0 to 9, given an audio input file.

2 Modules

Given the (a priori) easiness of the classification task, I decided to focus on implementing a high-quality pipeline, meaning that the code is readable, the modules are extensible, I tried to avoid hard-coding, and allow for the machine learning experiments to be highly parametrizable and reproducible.

The user is able to interact with the pipeline via a simple command-line interface (CLI) consisting of Python command-line arguments. The pipeline provides three main functionalities:

1. Generate and save a dataset to disk by processing the raw data.
2. Train a neural network for the spoken digit classification task. The model's parameters and the desired optimization algorithm can be defined via a YAML configuration file, as well as some training options.
3. Perform inference on some test data, given a trained model.

The remainder of this section aims to present the structure of the pipeline, i.e., the different directories in the project and their purpose.

2.1 Source code

This module contains the source code and it is therefore the backbone of the pipeline. It is divided into three submodules: data, models, and pipeline. All three submodules can be invoked separately and support parametrization via command-line arguments. Additionally, all submodules have an optional command-line logging functionality, that informs the user about the progress in the execution of tasks.

2.1.1 Data submodule

This submodule allows the user to read the raw data, process it to generate a dataset suitable for the classification task and save such dataset to disk. The dataset needs little preprocessing, since all audio files have the same sampling rate, they all come in the same format (WAV), and they all seem to have already gone though a denoising step. Regarding the processing, I decided to allow for the extraction of Mel-frequency cepstral coefficients (MFCCs), which are then aggregated over time. Even though these features are less informative than other options, say Mel-spectrograms, they are more compact and should be good enough for the task at issue.

2.1.2 Modelling submodule

This submodule allows the user to define model architectures. I focused on the definition of Fully-Connected Neural Networks (FCNN), which can be parametrized via the Configuration module (see Section 2.3.2). In addition to the user definition of the number and size of linear layers, batch normalization is applied to all of them. The output layer is activated with the Log-softmax function.

Besides the FCNNs, I was curious to compare their performance with Convolutional Neural Networks (CNNs), so I decided to hardcode a simple CNN architecture to compare them.

2.1.3 Main pipeline submodule

This submodule allows the user to execute training and inference tasks. For training, the user must give the path to an already processed dataset in .npz format. The user can also provide the path to its desired model and training configuration. Finally, the user can use the flags -r, -p and -v to generate a report of the training results, save the trained model, and add verbosity, respectively. Similarly, for inference, the user must provide the path to a trained model, and to the test data. The user can also provide the path to its desired data processing configuration, and also use the flags -r, and -v.

2.2 Data

This module contains both raw (before processing) and processed (ready for training) data. This directory contains three subdirectories:

- raw/ : Contains the original dataset from [1].
- processed/ : Contains the processed datasets, in .npz format.
- test/ : Contains test data, recorded and denoised by me with Audacity.

The format of the original dataset divides the audio files into subdirectories corresponding to all the different speakers, and the audio files are formatted according to the following schema: "*label_speaker_sample*". I followed the same schema for the test recordings.

2.3 Configuration

This module contains configuration files in YAML format. I decided to add this module, as opposed to hard-coding the model's architecture and other parameters, for the sake of code-configuration separation. This approach improves reproducibility and experiment tracking, and providing more flexibility to the pipeline user. The module supports two types of configuration files: data configuration, and model configuration.

2.3.1 Data configuration

Simple configuration file that allows the user to experiment with different parameters to generate the MFC coefficients. The options are:

- n_mfcc: Number of MFC coefficients to extract.
- resampling_rate: Resampling rate for the raw audio file. If not specified, the original sampling rate is used.
- n_fft: Lenght of the Fast Fourier Transform (FFT) window.
- hop_length: Number of samples between successive frames.
- aggregate_over_t: Option to aggregate the extracted coefficients over time. If True, the resulting shape will be (n_mfcc,).
- padding: If aggregate_over_t is set to False, then a padding width can be specified to keep consistency between samples. The resulting shape will be (n_mfcc, padded_t).

This configuration file is useful to keep track of the parameters used to process raw data, whether it is to generate a dataset for training, or to process test data at inference time.

2.3.2 Model configuration

This configuration file allows the user to: 1) define the parameters of a FCNN, 2) define the parameters of an optimization algorithm, and 3) specify some training-related parameters. It is also possible to specify a name and version for the experiment.

Regarding the FCNN model architecture, the number and size of layers can be specified, as well as the type of activation function and its parameters. Note that, the same names as in the torch.nn package must be used for correct parsing (the same logic applies to the optimizer definition).

Concerning the additional training parameters, it is possible to specify the proportion of samples used for validation (eval_split), the number of epochs (epochs), the batch size (batchsize), and whether to do early stopping if the validation loss has increased with respect to the previous epoch (earlystop).

2.4 Models

This module contains the trained model instances, divided into subdirectories by experiment. To avoid serializing the whole model, we save both the model's parameters and its state. On the one hand, the model's parameters allow us to re-instantiate the model class in the future. On the other hand, the model state, i.e., its weight and biases, allows us to recover the state the model reached after training. This two-way approach avoids complicated and heavy file handling.

2.5 Reports

This module contains standard figures reporting the model's performance at train and inference time. At training time, the pipeline supports the reporting of the evolution of accuracy and loss for training and validation sets. At inference time, the pipeline supports the evaluation of the model's performance in the form of a confusion matrix.

3 Results

The data used to train the FCNN model was downsampled to 16kHz, used a FFT size of 512 and a length of hop between STFT windows of 16 to extract 20 MFC coefficients, aggregated over time. The network architecture consisted in three fully-connected layers of size 1024, activated with the ReLU function. The Adam optimizer was chosen, with a learning rate of 1e-6. The model reported a negative log likelihood loss of 0.2637 and an accuracy of 0.9292 in the validation set after 21 epochs, on a train-validation split of 80%:20%, with a batch size of 32. When evaluated on the data recorded by myself, the model reported an accuracy of 0.7321.

The data used to train the CNN was processed with the same parameters described for the FCNN, with exception of the aggregation over time. Instead, the MFC coefficients where padded to a maximum length of 200 for the time axis. The network architecture of the CNN is a simplified version of the AudioNet architecture described in [1]. The Stochastic Gradient Descent (SGD) optimizer was chosen, with a learning rate of 1e-5, and a momentum of 0.9. The model reported a negative log likelihood loss of 0.1419 and an accuracy of 0.9538 in the validation set after 23 epochs, on a train-validation split of 80%:20%, with a batch size of 100. When evaluated on the data recorded by myself, the model reported an accuracy of 0.50.

4 Conclusions

This implemented pipeline with allows for data preprocessing, and model training, and evaluation. The data processing is based on the librosa library, while the model definition and training is based on PyTorch. The FCNN model can be trained on a CPU in a very reasonable time, while the training of the CNN model becomes more computationally intensive (at least for my machine's CPU). An inference functionality, along with some pre-recorded test audio files are provided. The trained mdoels and processed datasets can be accesed in the following GitHub repository: AudioML-pipe.git.

The project presents a lot of areas of potential improvement, for example:

- Parallelize data reading and processing with multiprocessing library.

- Include data augmentation (adding noise, shifting pitch, altering speed, etc) to increase model robustness.
- Include normalization for the MFCCs [2].
- Add more exception handling for config reading and parsing.
- Include the option to extract Mel Spectograms instead of MFCCs from audio files for CNN.
- Automate hyperparameter tuning.
- Include cross-validation.
- Implement an interactive recording interface for the inference mode.
- Add persistent logging.
- Include a batch of unit tests to validate the expected behaviour of the different pipeline functionalities.

References

[1] S. Becker, J. Vielhaben, M. Ackermann, K.-R. Müller, S. Lapuschkin, and W. Samek, "Audiomnist: Exploring explainable artificial intelligence for audio analysis on a simple benchmark," *Journal of the Franklin Institute*, vol. 361, no. 1, pp. 418–428, 2024.

[2] N. V. Prasad and S. Umesh, "Improved cepstral mean and variance normalization using bayesian framework," in *2013 IEEE Workshop on Automatic Speech Recognition and Understanding*, pp. 156–161, 2013.