Spoken Language Recognition - Tensorflow Lite for Microcontrollers

Computer Science Engineering Project - Hardware Architectures for Embedded and Edge AI

Academic Year 2022 - 2023

Authors:

Simone Giampà - 10659184

Claudio Galimberti - 10610720

Github Project Repository: Spoken_Language_Recognition_Tensorflow_Embedded

Project Objective

The purpose of the project (in both hardware and software aspects) is to develop a system that is able to recognize in which language a person is speaking by recording a short voice audio.

Many tests conducted to test the limited memory availability of the Arduino showed that the maximum amount of time for the audio recording for inference is 5.6 seconds.

The project is meant to be run on the Arduino Nano 33 BLE Sense Lite, having 256KB of RAM and 1M of flash memory.

Our approach

Our approach to the project consists in defining a series of steps to follow for successfully implementing the algorithm to be run on the Arduino. The first step is the creation of the hardware device where the data collection process and the inference happen. The definition of which languages to include in our dataset and model capability was left as the last thing in the project since this choice would have been limited by the quantity of data to be acquired for training the model.

The next steps include the programming of the code for the data collection part, the model training and evaluation, the Python scripts for the dataset creation and splitting, and the C++ code for computing the features.

The development of the code was made difficult especially because of the lack of documentation of the TFMicro library and the scarcity of examples available online for what we were trying to achieve.

Our task was nonetheless difficult and proved to be very challenging, especially because probably no one published open-source code on an algorithm of this kind.

Eventually, we decided to keep just 3 languages as a demonstration of the potentiality of our project and also because our objective is not to maximize the accuracy of the model.

1. Hardware setup and utility

We developed a hardware device that we used to acquire the data, and in order to have the most practical experience with this project.

This represents the first step in our project since acquiring data for our model to be trained is the next important step. The hardware device proved useful throughout the entirety of the project.

We leveraged our ability in soldering, acquired in a previous course to solder the wires and all the electronic components to a circuit board, powered using a portable battery.
The electronic elements that we decided to use are:

1. A micro SD reader is used to store the audio acquired with Arduino.
2. An LCD to display the result of the model prediction at inference time and to display the recording status during the dataset creation.
3. A functional button is used to start the recording of both the acquisition of the dataset and the audio to be analyzed at inference time.
4. Pins connectors hold the Arduino in place and provide wire connections to the other components
5. The portable batteries power all the components of the device, making it easy to use in every environment

2. Dataset creation

The **acquisition of audio data** served for the dataset creation. All audio samples were collected using Arduino. We asked people we know to provide us a 1 minute audio recording of any topic, in the languages they felt most comfortable with. The audio samples were recorded from people physically in front of the device and also from some YouTube videos. The dataset was augmented using some audio of conversations found on YouTube.

The hardware device allowed us to save the audio files on the SD card, which were then imported into the workspace folder for the actual dataset creation. This was important in order to collect audio samples without relying completely on the computer, and instead having a portable device for recording.

The second step is the **transformation of the audio WAV files** into a CSV dataset using a Python script. This CSV file is further elaborated in order to process the relative MFCC features from each raw audio data in the starting CSV file.

Training - Validation - Test splitting criteria

The dataset is created by using overlapping sliding windows in order to create many chunks of audio from a single file. Applying overlapping windows and splitting the dataset in a traditional way would have led to the problem where the same window could be present in both the training set and the validation set, potentially causing inexact accuracy measurements. Therefore, it was decided to use a more complex splitting approach in the dataset creation script to avoid this error.

In order to have zero overlap between frames in training and validation, we divided each audio sample into 2 separate non-overlapping chunks, where the frames are computed. All the speakers will have their data split 75% - 25% between training and validation. The algorithm takes each audio sample and splits it into a piece of 45 seconds (used for training) and another separate piece of 15 seconds (used for validation). From each slice of audio created, many chunks of 5.6 seconds are extracted, from which the features are computed.

Test Sets

The test sets are created ad hoc. They were set up only when a lot of data was collected. They include audio samples placed in specific folders and separated from the audio samples used for training and validation.

The test sets aim to evaluate the model's generalization capability and understanding in a more accurate way compared to training and validation. They are needed to evaluate the performance of the model in different scenarios:

1. known speakers in already heard languages —> evaluate model performance in similar scenarios and with noisier audio recordings
2. known speakers in un-heard languages —> evaluate the performance of recognizing language characteristics instead of the speaker vocal traits
3. unknown speakers —> evaluate performance for recognizing language from an unseen speaker in a variety of conditions

The test sets are more representative of the model's ability to generalize over different speakers and languages and are meant to show whether the model is understanding the characteristics of similar languages, without emphasizing the speaker's vocal characteristics and traits.

Our forecast for the test evaluations was that the performance in case 3 would be lower than in case 2. Having separate test sets is useful in order to have an unbiased estimate of the model's performance on different tasks.

3. Features engineering: MFCC

MFCC stands for Mel-Frequency-Cepstral-Coefficient and they are a feature extraction technique in speech and audio processing.

The term “Mel” refers to the Mel scale, which is a perceptual scale of pitches that is more aligned with the way humans hear sounds. It is used to convert the linear frequency scale of audio signals into a logarithmic scale that approximates human auditory perception.

Cepstral Coefficients: Cepstral analysis is a mathematical technique used to represent the short-term power spectrum of a signal. Cepstral coefficients capture the envelope of the spectrum, highlighting the characteristics of the audio signal, such as its timbre and pitch.

Computation of the MFCCs from a chunk of raw audio signal:

1. Compute pre-emphasis filter on the entire audio signal to boost higher frequencies
2. Apply hopping window on signal computing a series of frames with some overlap (hop)
3. Apply the hamming windowing function to each frame
4. Apply STFT to obtain the power spectrum
    a. apply the Discrete Fourier Transform (DFT) to each windowed frame.
    b. compute the magnitude spectrum (or squared magnitude) for each frame.
5. Compute the mel-scale filter bank and apply the filter to the magnitude spectrum obtained
    a. Design a Mel-filterbank consisting of a set of triangular filters, uniformly spaced on the Mel-frequency scale.
    b. Apply each triangular filter to the magnitude spectrum to obtain the energy in each Mel-frequency band.
6. Take the logarithm of the filterbank energies to compress the dynamic range.
7. Apply the Discrete Cosine Transform (DCT) to the log-filterbank energies to obtain the MFCC coefficients.
8. Cepstral Mean-Variance Normalization (column-wise): Perform mean-variance normalization by subtracting the mean value and dividing by the standard deviation of each MFCC coefficient across all frames. This step helps to minimize speaker-specific characteristics.
9. 1-byte min-max column-wise quantization of the floating point values of the coefficients

4. Data preprocessing pipeline: main challenge

The data preprocessing part was the most challenging one due to the scarcity of material available online regarding this particular topic. Initially, the project's implementation was carried out by following different approaches to determine which one was the most suitable in terms of feasibility and efficiency. Three main paths were pursued:

1. Complete code implementation using Tensorflow libraries for both the preprocessing phase and the neural network part, while including all the code inside a customized Model architecture.
2. Utilization of Edge Impulse for the entirety of the project, from the data preprocessing to the compilation of the Arduino code for inference
3. Mix of C++ libraries for the preprocessing phase and Python scripts for the functional part and model architecture.

The preprocessing of the dataset represented the part that took the most time to be implemented and is the main reason for which we tried the three different approaches described before. The main problem to face was the necessity of the same pre-processing code to be executed at both testing and inference time.

Approach 1: Custom model including the preprocessing

The conversions to Tensorflow Lite and Tensorflow Lite for Microcontrollers limit the functions that can be imported and used, so after a lot of tests and searches for trying to use pre-built ones we decided to move to other directions. We understood that implementing the preprocessing in tfmicro would require a lot of effort in research for the functions that would be needed. In fact, due to the lack of documentation for the TFMicro library, the search was quite difficult. In the end, after understanding well the TFMicro kernel functions, we understood that implementing C++ code for the spectrogram computation and the equivalent Python wrappers would have been quite challenging, Furthermore, the very limited availability of functions for the micro-operations resolver, would have made the implementation probably unfeasible.

Approach 2: Edge impulse: simple but not free cost

The second approach tested was the usage of Edge Impulse but even this way wasn't the most suited for our purpose because of the lack of customizability that our project required and the lack of resources available without a business account. In fact, implementing the entire project would have been feasible with a business account, that does not have the limit of 20 minutes of training time on a slow CPU.

Approach 3: Complicated but effective

Finally, we moved to the last and currently implemented solution which consists of a pre-processing phase that is developed using complete C++ libraries. We implemented from scratch the code for computing the MFCCs using a pre-built FFT library. The code is extremely well optimized for minimizing memory consumption since this code is also meant to be executed on the Arduino platform.

This solution allows the use of the pre-processing phase both before the training and during inference time because it can be run on the Arduino without the use of the Tensorflow library but as a separate code to run before the inference.

This approach consists of developing a custom library that can be run both on a Linux machine and on an Arduino platform, without exceeding memory limits. This step is essential in order to ensure correct data preprocessing at training and inference. This code is run before the Python scripts for the model training, in order to create the CSV files containing the features, which are then used as input for the neural network.

5. Convolutional Neural Network Training

We trained different convolutional neural networks to find the one with the least number of parameters that were performing well on both training and validation sets.

We tested a CNN with skip connections of concatenative type, which proved to have more or less the same accuracy but required more parameters, so this type of architecture was eventually discarded.

We tested a CNN with skip connections of additive type, which proved to have a slightly decreased accuracy and longer training times, despite having the same number of parameters. So this network was discarded as well.

Eventually, the best-performing CNN was the one with 1D and 2D convolutions, max polling in between, a final global average pooling layer, and then a sequence of fully connected layers. The output is a softmax layer because of the categorical cross-entropy loss function (one hot encoded labels).

6. Test sets model evaluation

The test sets were set up to evaluate the quantized model performance. We proved with many tests that the post-training quantization worked really well. In fact, we saw an accuracy drop of 1-2% overall after quantizing the model. This loss in accuracy is totally negligible for our purposes.

```
evaluating accuracy of quantized model on test set: unheard speakers
x shape =  (1260, 349, 12, 1)
y shape = (1260, 3)
Accuracy in test set of quantized TFLite model: 0.6429
MSE loss in test set of quantized TFLite model: 0.0237
[[0.62619048 0.30714286 0.06666667]
 [0.23571429 0.58809524 0.17619048]
 [0.18571429 0.1        0.71428571]]

evaluating accuracy of quantized model on test set: known speakers with noisy
audio recordings
x shape =  (1170, 349, 12, 1)
y shape = (1170, 3)
Accuracy in test set of quantized TFLite model: 0.7752
MSE loss in test set of quantized TFLite model: 0.0198
[[0.7025641  0.10769231 0.18974359]
 [0.14615385 0.71538462 0.13846154]
 [0.06923077 0.02307692 0.90769231]]
```

7. Export to TFLite and TFMicro

The trained model was exported to Tensorflow Lite using the Converter. The option chosen for the conversion is the post-training full integer (INT8) quantization. This allows our model to have integer inputs and outputs. The weights and activations are also quantized to INT8 for efficient model storage.

Exporting the model to a C byte array for TFMicro resulted in a model of size 16.7KB. This compressed model contains the model weights and the internal functions used by the TF kernel to compute inference on the model. This C byte array is then used by the TFMicro interpreter, along with the micro mutable operations resolver to convert the operations in the model to actual executable code that works seamlessly with our trained model.

8. Inference on Arduino

We developed code for testing the model's capabilities at inference time. We use the LCD display to show the user the final prediction output. We use the push button to start recording the audio sample. At inference time the SD card reader is not used, since everything runs locally on the Arduino, without making use of external memory.

Fitting everything in the limited RAM of the Arduino was not an easy task.

After the audio sample is collected, the features are computed. When the features are computed, the audio raw data is freed from memory, making space for the Tensor Arena. The tensors are allocated, and the interpreter is called using the micro mutable operation resolver, equipped only with the essential operations.

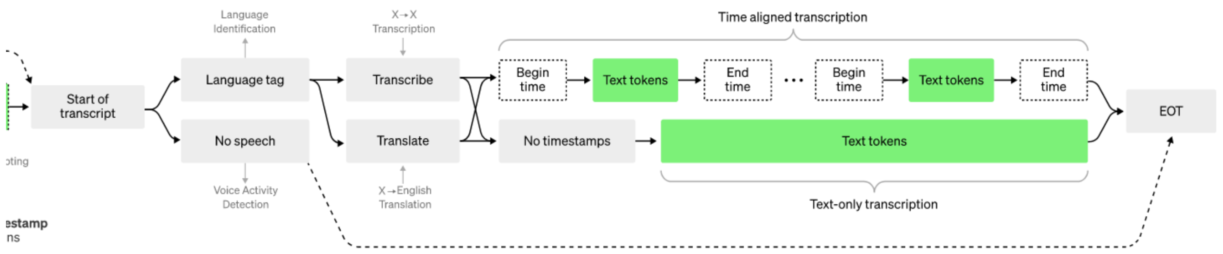
Then the model is fed with the features as input, the interpreter is invoked and the output result is shown as the predicted class, from the softmax of the output values.

Market perspective: Opportunities and usability

A neural network application designed to recognize spoken languages on the Arduino platform holds considerable market perspective across different sectors.

- In the field of voice assistants and smart devices, this technology can enhance language recognition capabilities by allowing the assistants to analyze the audio with this system before producing the translation. This is useful in the case of environments where there are many people speaking multiple languages (such as airports or train stations) in which it would be useful to have these sorts of assistants and devices being able to respond to users without any prior language configuration. In this case, the device would understand the language and answer accordingly.
- The customer support and call center industry can leverage language recognition to optimize operations, the calls can be analyzed with the model and passed to the appropriate operator that is able to communicate with that specific language. In general, in this type of situation, the language has to be identified and so a human interpreter of that specific language can be called.
- In the IoT and home automation fields, this system can be included to allow users to interact more easily with their smart devices using their preferred language without troubleshooting in changing settings. For example, our device would be useful in households where multiple languages are spoken on a daily basis.
- A spoken language recognition application can be used even for cultural purposes allowing users to understand and preserve local dialects. However, this approach would require more extensive neural network training and an increased dataset of audio recordings.
- OpenAI's Whisper model uses a language recognition module before translating audio data. Our device implements the same mechanism as in Whisper but is able to execute it in an embedded device such as a microcontroller. Other similar systems, such as Google Translate provide a feature to understand the language

from a piece of text or an image. A similar task can be achieved by our device, using the audio and not relying on powerful servers for the computation.



- Considering the translation field, a spoken language recognition model can be used to categorize and analyze customer interactions based on the language they spoke, providing businesses with valuable data and improving the service and strategies.

Ethical perspective: Data privacy concerns during the audio dataset collection

The process of collecting and storing audio data for language recognition purposes raises significant privacy questions, especially when people are unaware of being recorded. Without specific informed consent this practice can lead to unauthorized access to sensitive information. The necessity of obtaining informed consent is a fundamental ethical principle in deploying language recognition systems. Individuals must be fully aware that their spoken language may be recorded and analyzed and they should have the option to opt out of the presence in such databases.

In our approach, we inform all our speakers that their recordings would have been used only for our specific application and would never be shared with anyone else. They were informed about the fact that they're recording would have been stored only in our devices until the computation of the features would have been completed. They were also informed about the scientific purpose of the application.

To avoid the usage of sensitive data, every speaker has been asked to produce audio recordings without talking about themselves (and mentioning private information) and without using any information that can be attributable to them.

The speakers were suggested to read from some public sources of data on a topic that they like. Therefore, many of our recordings contain information about news articles, Wikipedia pages, books, recipes, tales and stories, etc.

All of the speakers adopted our recommendations. We also checked all of the recordings and the ones containing some sensitive data have been deleted, in which case we asked the speakers to provide a new audio recording.