Embedded Speech Recognition System on STM32F407G-DISC1 Board

1. Introduction:

1.1 Overview of Our Project

The project, proposed by ACTIA Engineering Services Tunisia as a two-month internship, aims to integrate AI-based solutions into embedded systems with limited resources and strict memory constraints. Since there is no direct toolchain for deploying and analyzing such models in embedded systems environments, especially in STM32, this application example can be useful for some.

The project is an embedded speech recognition system on the STM32F407 microcontroller, which is made up of the following software parts: audio acquisition, spectrogram generation by using audio processing techniques, USB transmission for debugging information, and deployment and execution of deep learning models and layers on the STM32 by using Cube.AI.

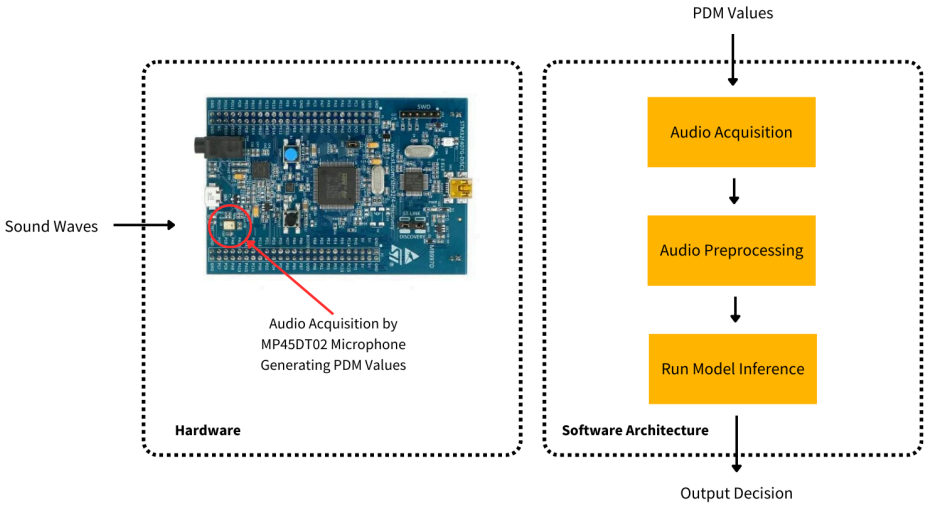

Figure 1.1: Embedded Speech Recognition System Diagram

1.2 Challenges

With the abundance of speech recognition tutorials available online, implementing this technology in an operating system environment has become relatively straightforward—just run TensorFlow's predict function, and you're done! However, bringing AI into the world of embedded systems presents unique challenges. First, our system operates without an OS, so we must rely on bare-metal programming. Second, our system has limited memory, requiring us to fit the entire project—including 1 second of audio snapshots—into just **112KB** of space. Furthermore, there isn't a direct toolchain

for integrating AI solutions seamlessly. Although Cube.AI can deploy the model and generate the necessary API for inference, it lacks preprocessing tools. Since raw data cannot be fed directly to the model, especially in sound classification tasks where the model expects a 2D matrix or "spectrogram," these preprocessing algorithms and tools must be implemented manually or sourced from external libraries.

2. Audio Aqcuistion:

Audio acquisition in an embedded system can be quite challenging because much of the abstraction is removed. Unlike running a simple Python script to capture voice commands or pressing a button to record, the process is more complex. When the integrated microphone of the STM32F407G-DISC1 board captures audio, it generates PDM (pulse-density modulation) signals.

Pulse-density modulation is a method used to represent an analog signal with a binary signal. The output of a 1-bit DAC (Digital-to-Analog Converter) mirrors the PDM encoding, where a '1' represents a positive pulse and a '0' represents a negative pulse.

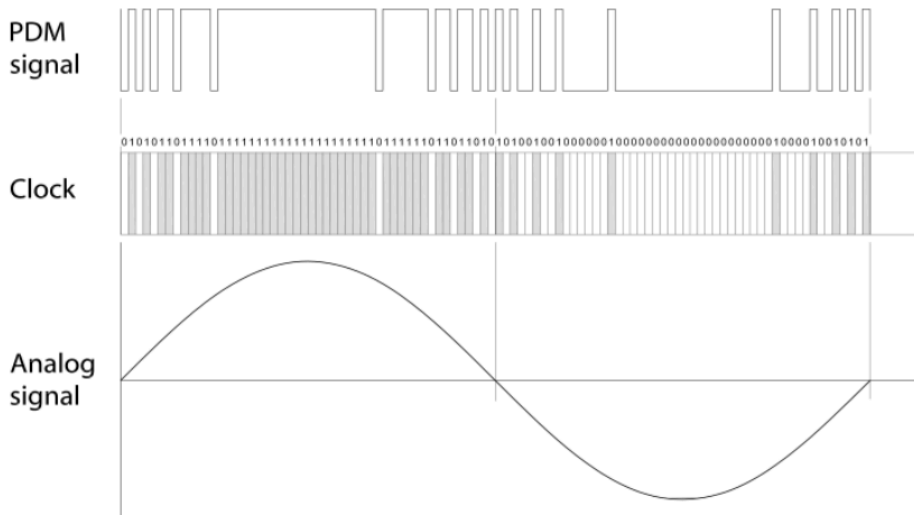

Figure 2.1: PDM Signal Example

PDM data will be received from the microphone into the SPI block via the I2S protocol; the data will be transferred to the RAM via DMA; when the CPU takes the PDM data from RAM, it will do some signal processing to generate PCM. Finally, PCM will be stored in memory.

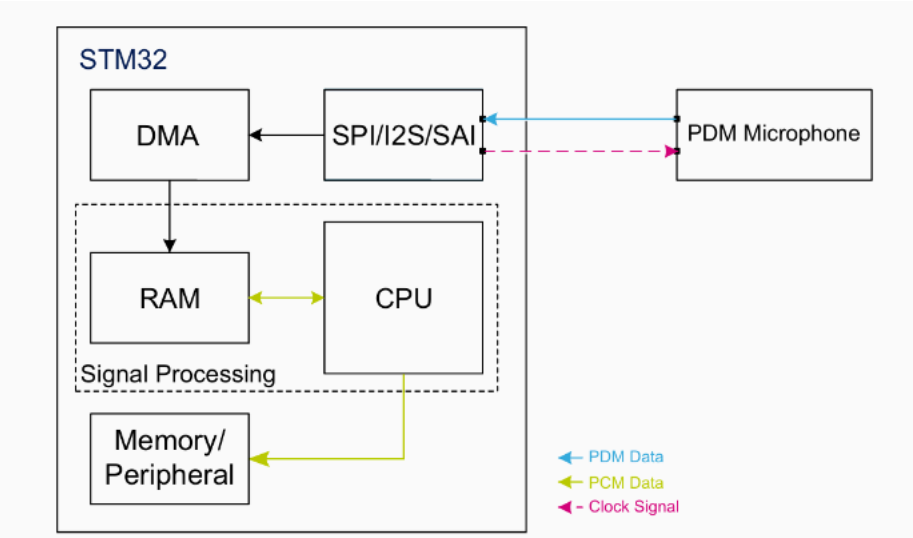

Figure 2.2: Digital Data Acquisition Block Diagram

The PDM signals are typically passed by digital filters to convert them into PCM (pulse-code modulation) signals that contain better information about the changes of pitch and content in them.

In pulse code modulation, the analog message signal is first sampled, and then the amplitude of the sample is approximated to the nearest set of quantization levels. This allows the representation of time and amplitude in a discrete manner. Thereby, generating a discrete signal.

But we need a way to convert the PDM signals to PCM. There are some PCM signal construction filters that take a stream of PDM values and convert it into PCM values. One of them is called decimation filters; the general signal processing technique. is called downsampling. The process can be described as a bandwidth reduction and data compression, where we take multiple inputs and find a way to generate one output (in our case, taking 1024 PDM bits and converting them into 16 pcm values).

Downsampling by an integer factor involves reducing the sample rate of a signal. This process can be broken down into two main steps:

1. **Lowpass Filtering:** a digital lowpass filter is applied to the signal to reduce or eliminate high-frequency components.

2. **Decimation:** The filtered signal is the ndownsampled by keeping only every Mth sample, where M is the downsampling factor.

Note: If you skip the filtering step, high-frequency components might cause aliasing, where they interfere with lower frequencies and create errors. The lowpass filter, also known as an anti-aliasing filter, helps prevent this issue.

With FIR filtering, it is an easy matter to compute only every Mth output. The calculation performed by a decimating Fir filter for the nth output sample is a dot product.

$$y[n] = b_0x[n] + b_1x[n-1] + \cdots + b_Nx[n-N] = \sum_{i=0}^{N} b_i x[n-i]$$

Where:
$x[n]$ is the input signal,
$y[n]$ is the output signal,
N is the filter order
b_i is the value of the impulse responses; these values are filter coefficients.

For embedded devices, there is a famous DSP library that has some different implementations of the FIR decimation filter. The famous CMSIS-DSP provides functions that can be used to perform such tasks.

Moving to its documentation, we can find **arm_fir_decimate_f32**.

The software for our project, which implements this library, is located in the Microphone BSP within the drivers folder.

One key challenge is managing PDM sampling and filtering to produce PCM values. If the processor is busy converting PCM values, it may miss incoming PDM data, leading to information loss. To avoid this, using DMA is essential. DMA will handle data transfer independently, ensuring a continuous stream of PDM data is captured without overloading the processor.

We'll also use double buffering techniques. PDM values will be stored in a buffer, and while one buffer is being processed, DMA will write new data to a second buffer. This prevents the DMA from overwriting the buffer currently being processed.

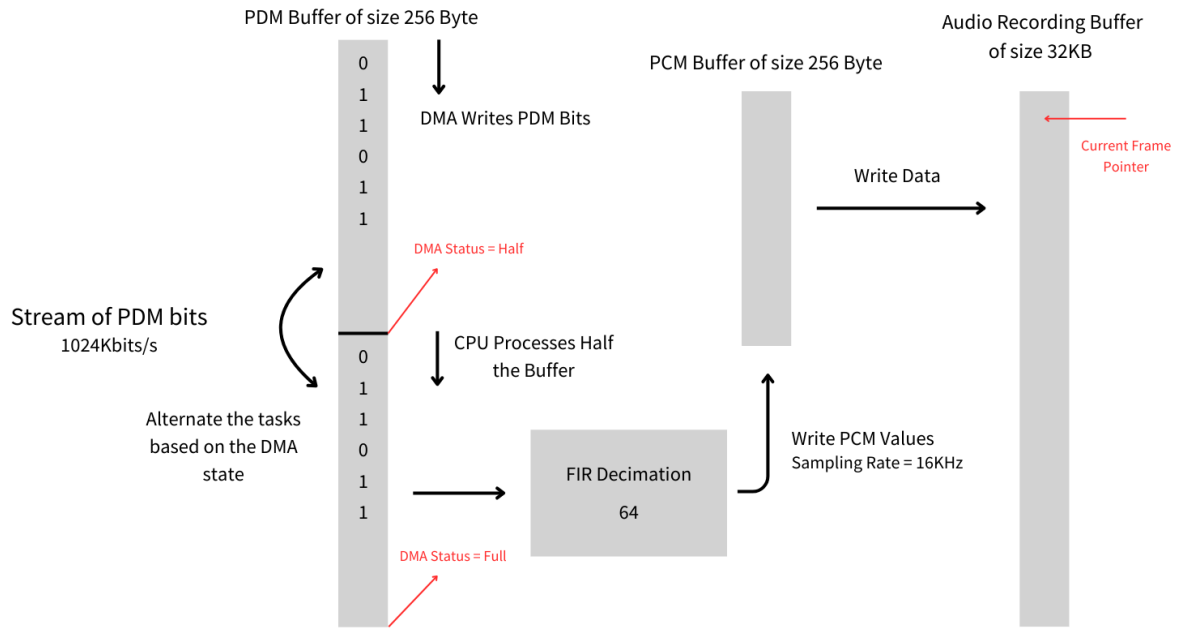

Figure 2.3: Audio Acquisition Software Architecture

3. Audio preprocessing and spectrograms

Preprocessing audio before feeding it into a deep learning model is essential. The problem is that raw audio data can be noisy, unstructured, and contain irrelevant information that may hinder the model's performance.

Spectrograms are important for audio classification because they provide a visual representation of how the frequency content of an audio signal varies over time. Unlike raw audio waveforms, which represent sound as a function of amplitude versus time, spectrograms display the intensity of different frequency components over time, making it easier to identify patterns that are relevant to classification tasks.

For instance, different sounds or spoken words have distinct frequency patterns that can be recognized. This makes spectrograms particularly useful for tasks like voice command recognition between various spoken words or sounds.

Here is an example of how the same words have similar patterns, while different words can have other shapes and patterns.

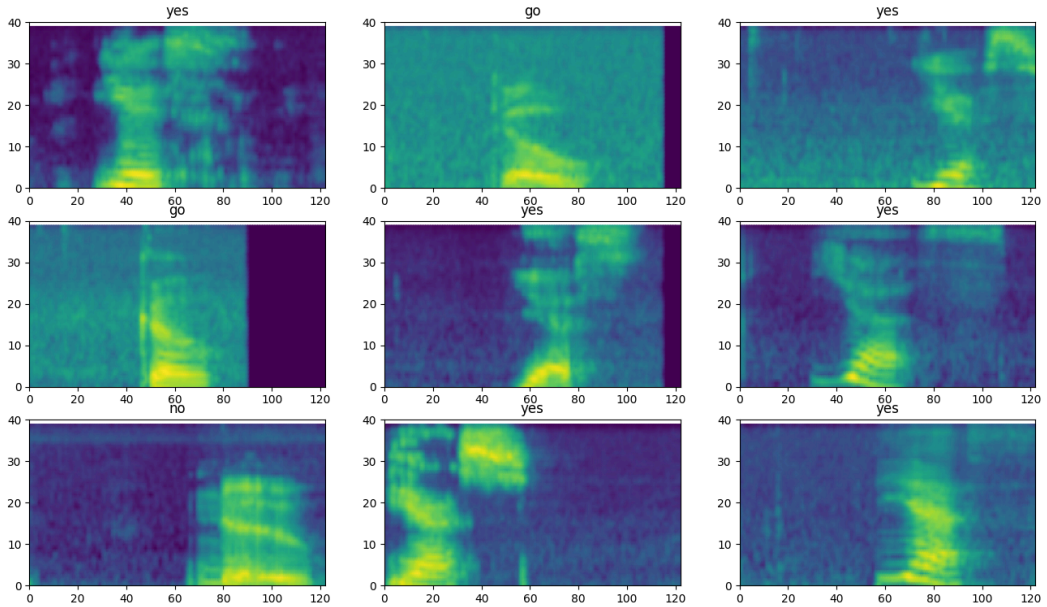

Figure 3.1: Different Spectrograms of Different Spoken Words.

A normal spectrogram is created by applying the Short-Time Fourier Transform (STFT) to an audio signal. The process involves dividing the signal into overlapping segments, applying a window function, and then computing the Fourier transform for each segment.

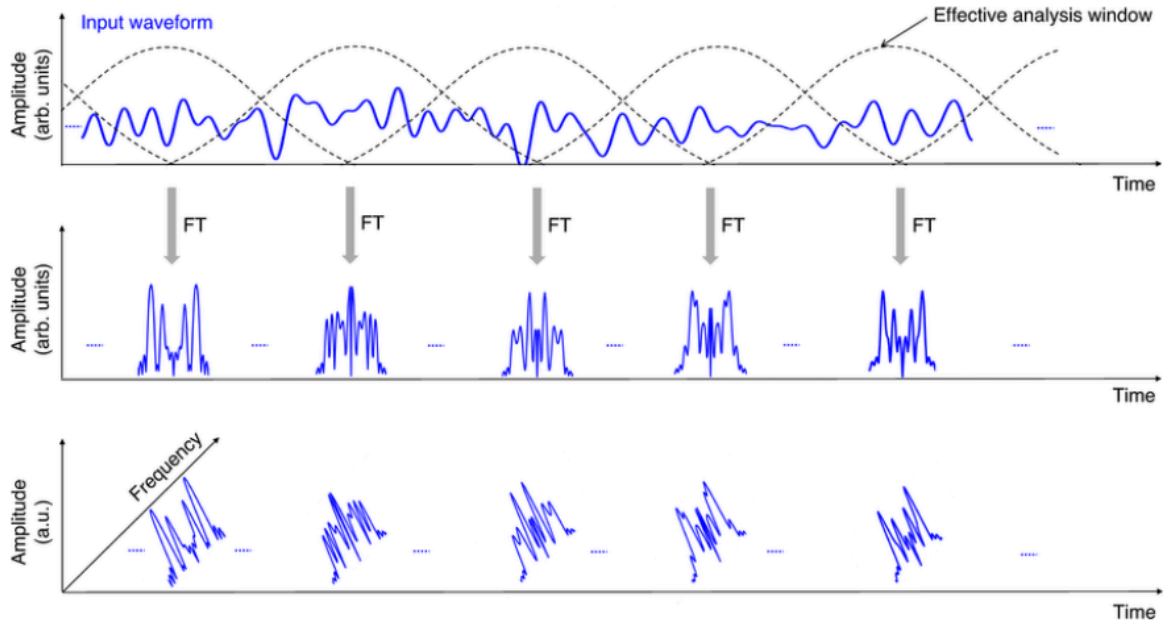

Figure 3.2: Short-Time Fourier Transform

The resulting spectrogram displays frequency on a linear scale. Which means that each frequency bin corresponds to a specific frequency.

Even after generating spectrograms, since our application is a voice command classifier, generally voice commands will be coming from human beings, and what we want to do is simulate the way a human being can hear sound. That's why we will transform the spectrogram into a **mel-scaled spectrogram**.

The mel spectrogram is a representation of sound that mimics the human auditory system's perception of pitch and frequency by mapping frequencies to the mel scale. This is achieved by applying a series of filters called mel filters, which compress higher frequencies and expand lower ones to align with how humans perceive sound.

Moreover, employing such a filter can reduce the dimensionality of the data, which can improve the performance of the model and significantly reduce the memory size needed to process such a model.

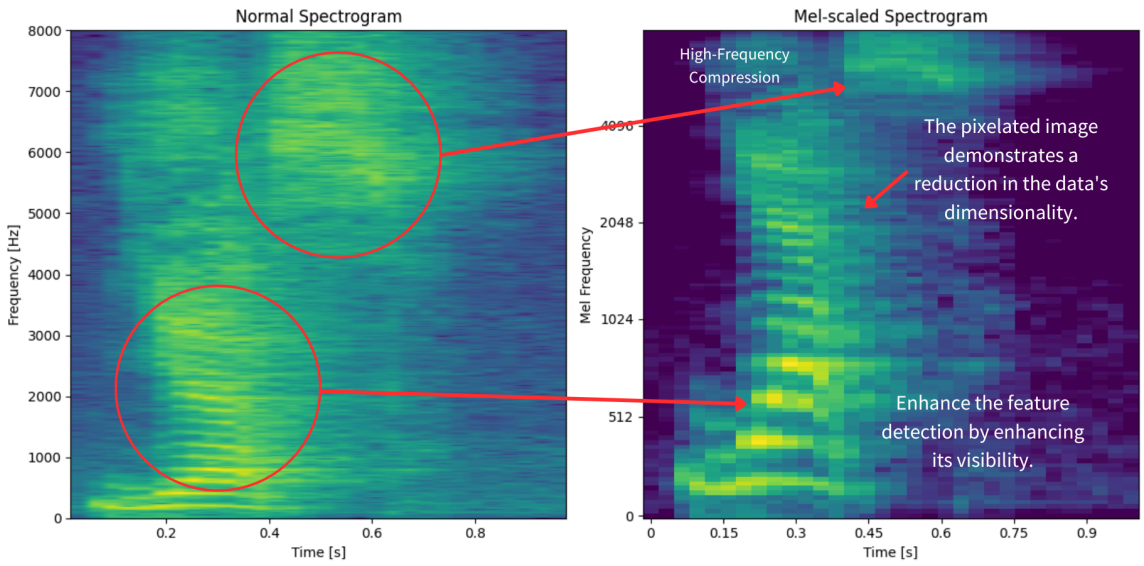

Figure 3.3: Comparison between Normal Spectrogram and Mel-scaled one of a "yes" audio file.

The question that arises now is, how can we implement such a technique in our discovery board?

For **STM32**, I have used the **STM32_AI_AudioPreprocessing_Library middleware** library that is being used for the **FP-AI-SENSING1 v3.0.0**. The library provides the building blocks for spectral analysis and feature extraction, is based on the **CMSIS-DSP** functionalities, is open-source, and can generate the computation of the spectrogram, **Mel-scaled** and **LogMel-scaled** spectrogram computation, and the computation of the **Mel-Frequency Cepstral Coefficients (MFCCs).**

In our project, you can find the library in the drivers folder; it contains source code for different audio processing techniques.

I have interfaced with this library to generate the logmel-spectrogram. The source code for this implementation can be found inside the **AUDIO_PROCESSING_UTLS**, where you can find the correct configuration of the different filters (which is an important aspect of the project).

4. Speech Recognition Deep Learning Model

In embedded systems, deep learning and machine learning models have specific requirements. They need to be lightweight in memory and have fast inference times. These types of models are often referred to as **TinyML** models.

The problem with our system is that the audio recording and **Mel-Scaled Spectrogram** generation consume a significant portion of the available memory. This constraint requires our model to be less than **40 KB** in size.

To address this limitation, we need to optimize the model to predict the correct output with the minimum number of weights possible. However, before we start, it's advantageous that we have already helped ease the design of our model by implementing a solid preprocessing phase. This phase will feed the model with data that has high feature visibility.

4.1 Deep Learning Model Architecture

Here is a general overview of our model:

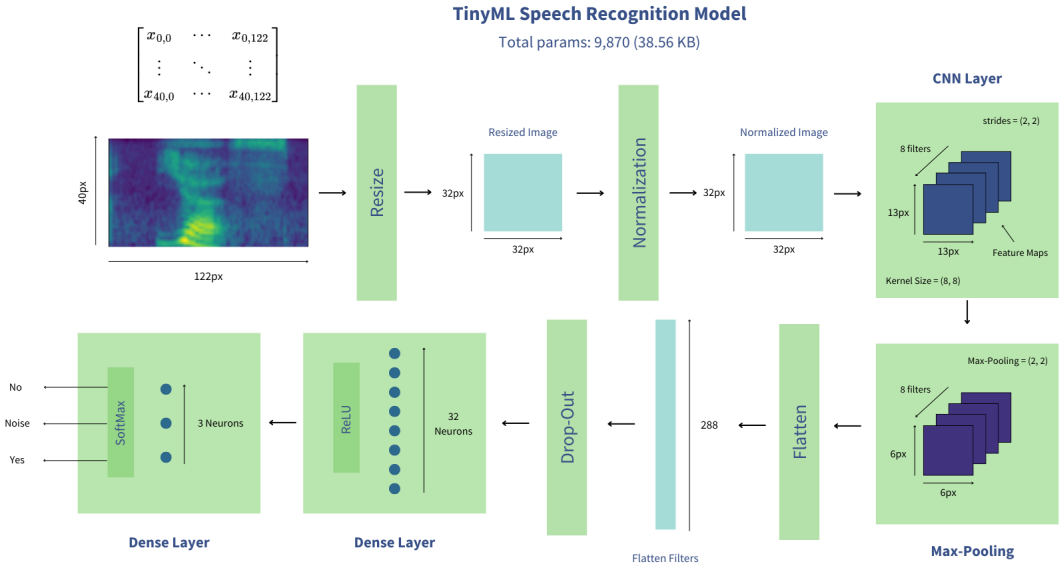

Figure 4.1: TinyML Speech Recognition Model

Preprocessing the input data:

1. Downsize the Mel-Scaled Spectrogram: We can reduce the size of the Mel-Scaled Spectrogram from the original **122x40 2D** grid to a **32x32 2D** grid.
2. Normalize the Values: After downsizing, we must normalize our values and map all of them to the range of [-1, 1].

These two preprocessing steps can be considered part of the preprocessing phase, but they will be integrated into the model itself.

Here is an example of a **"no"** spectrogram that is being preprocessed.

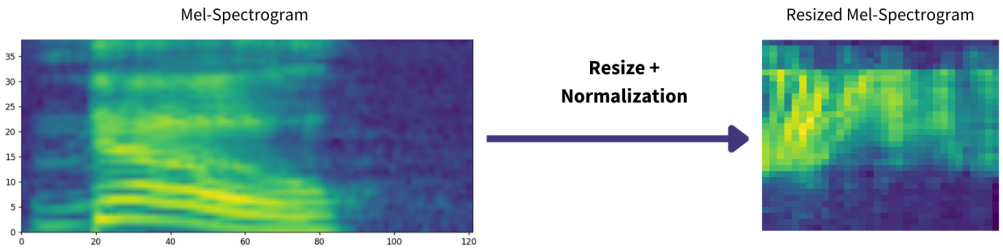

Figure 4.2: Resize and Normalization Layer Output

Feature Extraction using Convolutional Neural Networks

After normalization, it's time to extract the features from the image. In computer vision, the tool used to extract features from a specific image is called a **Convolutional Neural Network (CNN)**. These types of networks have a remarkable ability to learn which features the model should extract to provide the best possible input to the **MLP (multilayer perceptron)** .

The convolutional layer is the core building block of a CNN. It applies a set of filters (or kernels) to the input image to detect various features. Each filter is a small matrix that moves across the image, performing a mathematical operation called convolution. This process involves calculating the dot product between the filter and the portion of the image it covers, resulting in a feature map that highlights the presence of specific features, such as edges or textures.

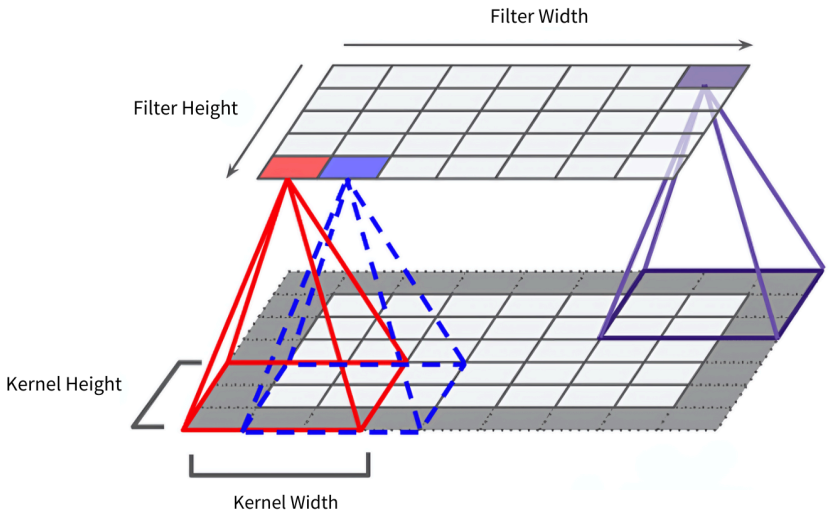

Figure 4.3: Convolutional Layer

This size of the convolutional neural network is important to consider, as a small increase in the kernel size or number of filters can increase the size of the model significantly. In our case, our model contains only 1 convolutional layer with 8 filters, so we will be getting 8 feature maps for this layer of size (13x13) because the kernel size is (8x8).

To calculate the number of parameters to be trained, which means the weights and the biases of the convolutional neural network, we can use the following formula:

$$N_{param} = (K_{width} \cdot K_{height} + 1) \cdot N_{filters}$$

Where:

N_{param} is the number of pramaetres,

K_{width} is the kernel width,

K_{height} is the kernel height,

$N_{filters}$ is the number of filters.

In our case we will be having **520** parameters, and if we suppose that these parameters are of type **float32,** we will be needing $520 \times 4Bytes = 2080Bytes$.

Here is an output example of the Convolutional Layer Output, which consists of 8 feature maps.

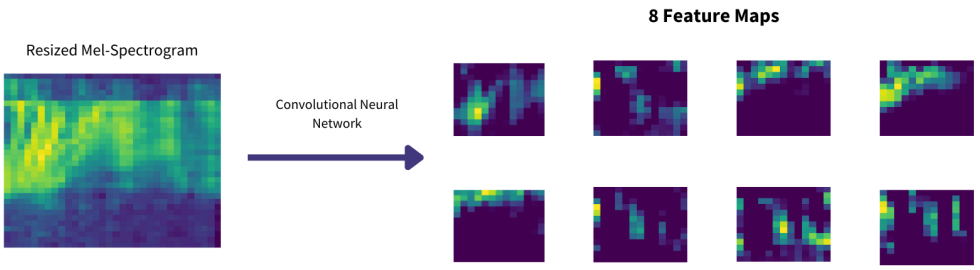

Figure 4.4: Convolutional Layer Output

Reducing computational load and memory usage using pooling layers

A pooling layer is a crucial component of convolutional neural networks (CNNs) that helps to reduce the spatial dimensions (width and height) of feature maps while retaining the most important information. This reduction in size helps to decrease computational complexity and mitigate the risk of overfitting.

Pooling layers operate on the output of convolutional layers, which produce feature maps. These feature maps contain various features detected by the convolutional filters. The pooling layer applies a specific operation to these feature maps based on the pooling method used.

We will be using max-pooling. It works by sliding a window (or kernel) over the feature map and selecting the maximum value within that window.

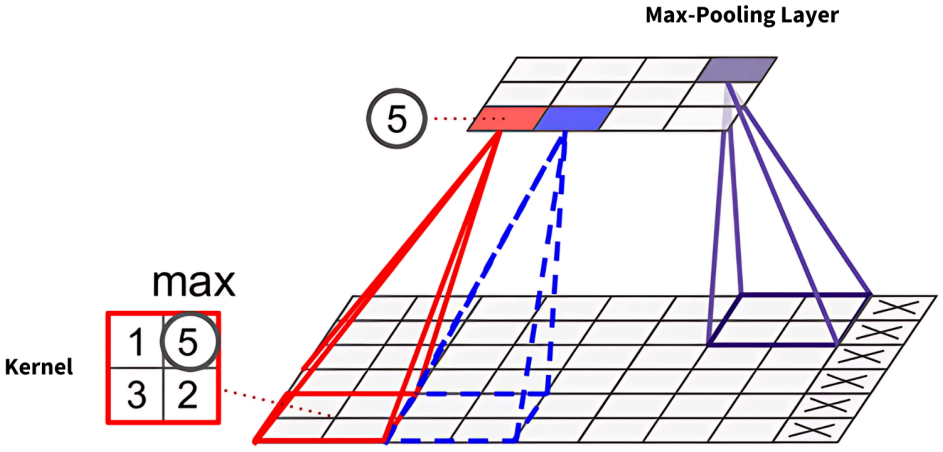

Figure 4.5: Max pooling layer (2x2 pooling kernel, stride 2, no padding)

After we have extracted important features from the input data using convolution and pooling layers, we can flatten our 2D grid of features into a 1D array. This step is necessary to prepare the data for the next layers of the neural network. Since the key information has already been captured by the filters, flattening does not lose any important details about the input.

Here is an output example of the Max-Pooling Layer Output, which consists of 8 feature maps.

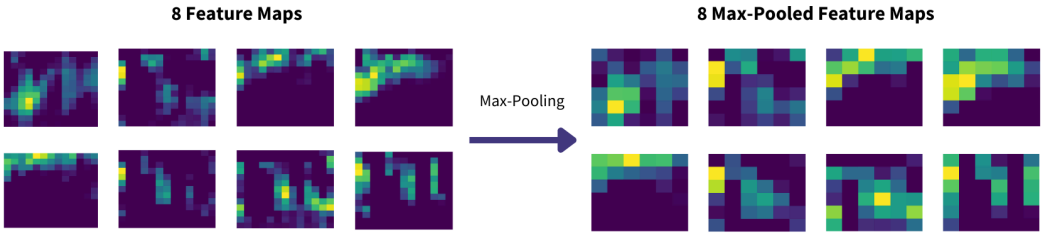

Figure 4.6: Max-Pooling Layer Output 8 Feature Maps

Adding Dropout for Regularization

To help prevent overfitting—where the model learns the training data too well but performs poorly on new data—we add a dropout layer after flattening. Dropout randomly sets a portion of the values in the array to zero during training.

Here is an output example of the Flatten and Drop-out Layer Output, which consists of 1 vector of size **288.**

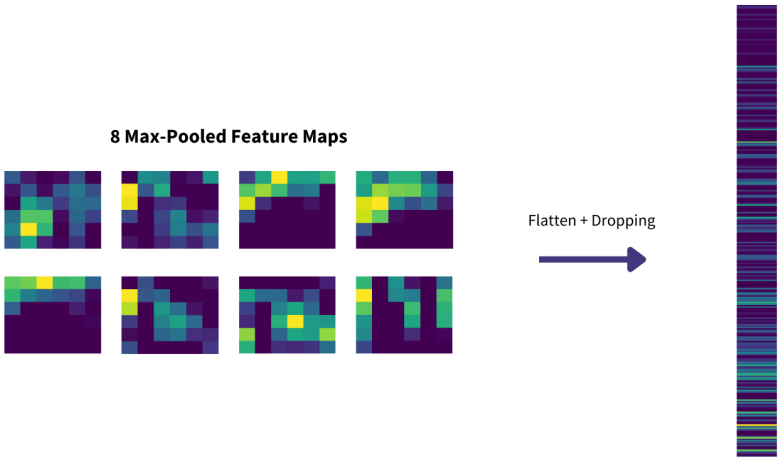

Figure 4.7: Flatten and Drop-out Layer Output

Adding multilayer perceptron for classification

Now, we will use a feedforward neural network to predict whether the input is a "yes," "no," or just "noise." This type of network processes information in a straightforward way, moving from input to output without any cycles or loops.

In our application, the input size is 288, meaning we have 288 features to work with. To effectively process this input, we need a suitable number of neurons in the network.

Starting with **32** neurons allows us to capture important patterns in the data while keeping memory consumption low. We can calculate the number of parameters, thus the size of the MLP, by the following formula.

$$N_{param} = N_{neurons} \cdot (N_{inputs} + 1)$$

Where:
N_{param} is the number of pramaetres,
$N_{neurons}$ is the number of neurons,
N_{inputs} is the number of inputs,

In our case we will be having **9248** parameters, and if we suppose that these parameters are of type **float32,** we will be needing $9248 \times 4Bytes = 36,992KB$.

Finally, the last layer is the classification layer that contains three neurons; each neuron represents a class "yes," "no," or "noise," and each neuron contains a softmax activation function to represent the output in the form of probabilities.

Here is an output example of the multilayer perception, which consists of the classification

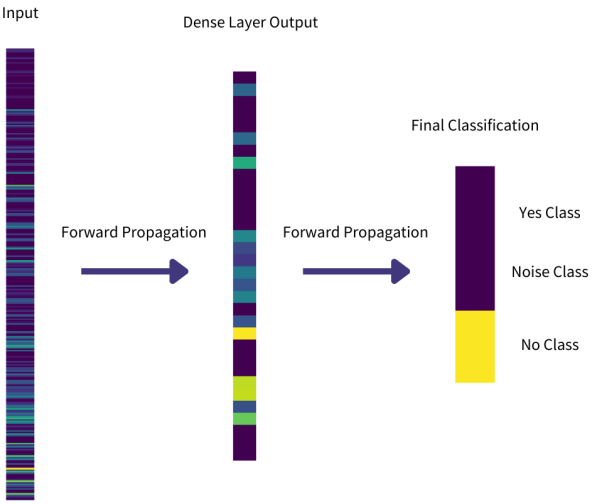

Figure 4.8: MLP Output

4.2 Model Training

To train deep learning models, we require clean and high-quality data. Our model has primarily been trained using the publicly available Speech Commands dataset provided by Google. This dataset consists of audio-labeled files featuring basic commands such as "yes," "no," and "go." We utilized both "yes" and "no" for training. Initially, we trained the model solely with this dataset, but the results were not satisfactory. Although the model performed well on the test set, it struggled with audio generated by the ST microphone. Consequently, I decided to incorporate some audio samples from the microphone, leading to a significant improvement in the model's accuracy.

To classify other sounds as "noise," I attempted to build my own noise dataset. This noise data was generated by the microphone when it was recording without any spoken commands like "yes" or "no," treating this type of audio as noise. I also included various random words and phrases, categorizing them as noise. Following this, the model was

able to predict noise in most cases. However, there are still some edge cases where the audio shares similar features with the keywords "no" and "yes," such as "go" and "slow." Since these keywords were not explicitly trained as "no," and they exhibit similarities in features with "no," the model occasionally predicts them as "no."

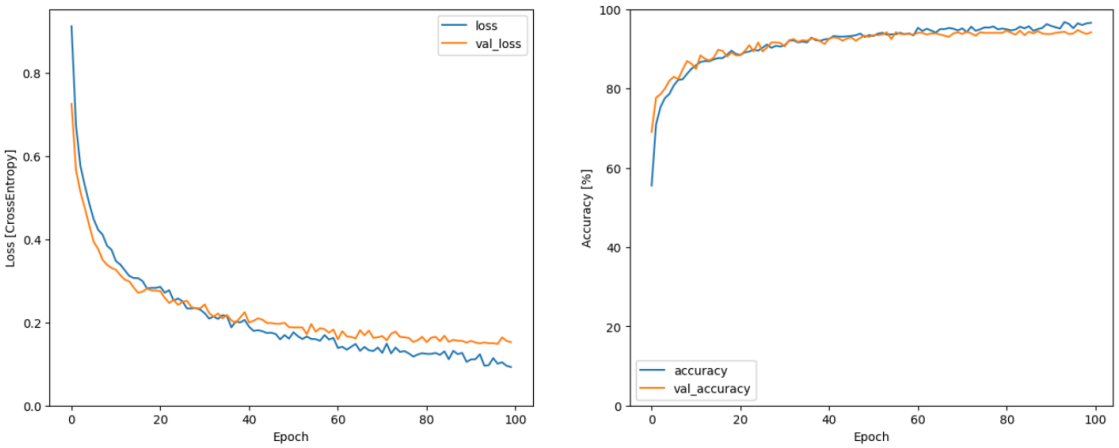

Figure 4.9: Training History of the Model.

4.3 Model Performance

In deep learning, there are common metrics used to evaluate model performance, with accuracy being one of the most prominent, alongside the confusion matrix.

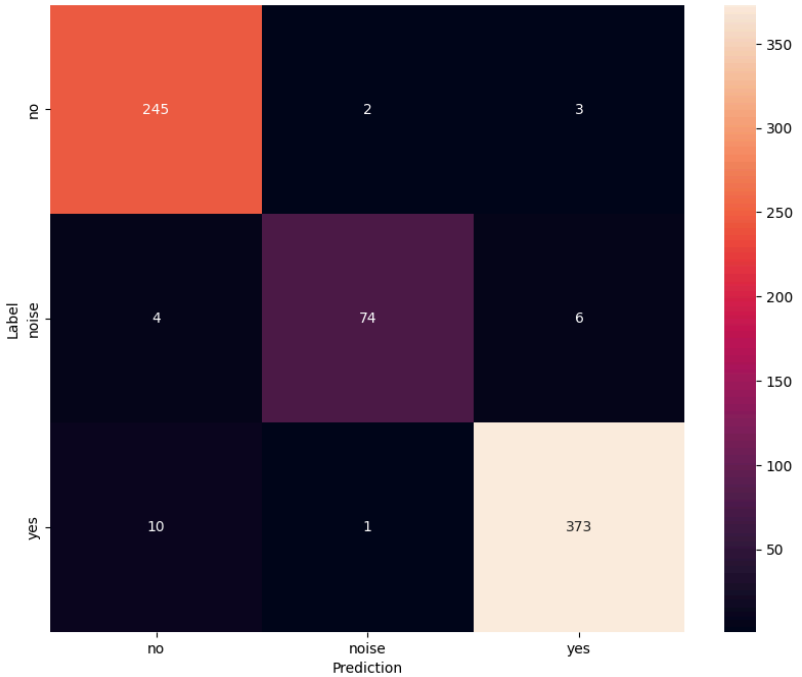

Figure 4.10: Confusion Matrix.

The challenge in evaluating our model lies in the randomness of real-world input, making it impossible to replicate all potential cases. This means that even if the model demonstrates high accuracy on a test dataset, that dataset may not adequately

represent the complexities of real-world scenarios, which include random speech, varying sound volumes, and different spectrogram generation algorithms. The model is trained on preprocessed audio created by the TensorFlow library, while the inference preprocessing is implemented on the STM32 using the **STM32_AI_AudioPreprocessing_Library**, which may have different implementations. Therefore, the true measure of the model's performance is to test it on the STM32. Another method I developed involves capturing audio, preprocessing the spectrogram on the STM32, and then sending the spectrogram matrix to a computer via USB. A Python script `(debug-model-inference.py)` reads the USB data, passes the matrix as input to the model, and runs the inference. Here is an example of the script running.

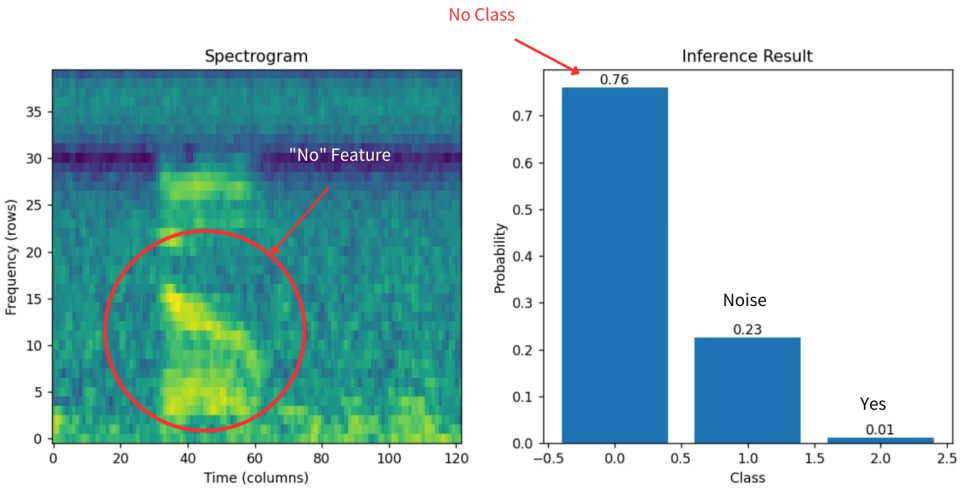

Figure 4.11: Model Debugger Python Script Output.

5. Merge different components

Merging these different components and making them to make seamless together what will define our system, we have 3 main states or functions: acquire 1s of audio, preprocess the audio, and finally run an inference. And then perform some output.

In this project, each section is abstracted entirely into 1 function.

We have discussed the working of the audio acquisition and the preprocessing, but we haven't discussed yet how to deploy the AI model that we have created into the stm32.

In some STM32 microcontrollers

For general embedded AI projects, generally Keras, Tensorflow, or Pytorch models can be exported in a specific format or extension; in the case of TensorFlow, the TensorFlow Lite extension is the standard ".tflite." Another extension, such as ".onnx," is used for non-Google-related tools.

The STM32-Cube. AI is a tool that helps integrate and deploy deep learning models in STM32 embedded devices.

It automatically converts famous model files into optimized C code defining model parameters such as weights, biases, and activation functions, and a static library that servers an API to the AI runtime library.

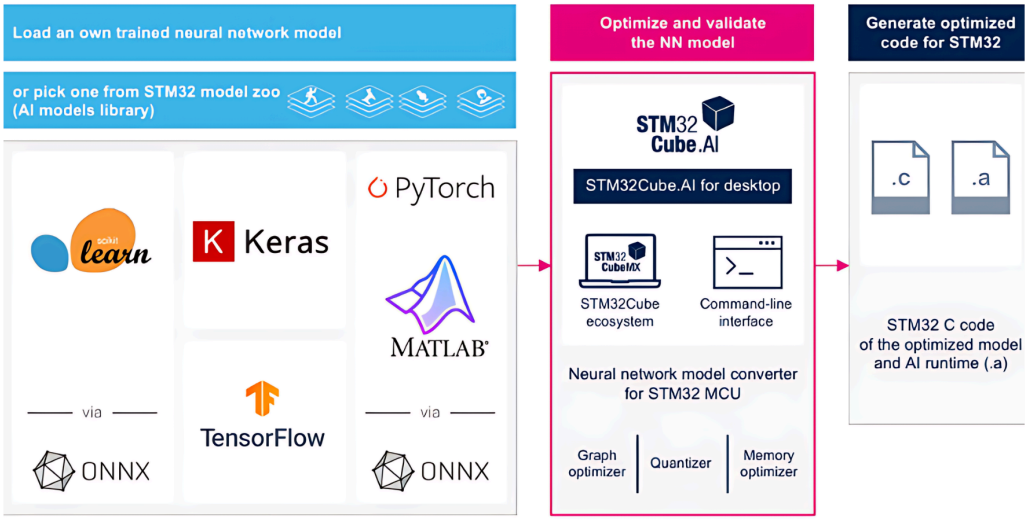

Figure 5.1: Cube.AI Illustration.