Tiny Embedded AI Voice Recognition Application

DEPLOYMENT CUSTOM MFE AND DNN

Sources:

* <https://github.com/edgeimpulse/processing-blocks>: Processing blocks Python code of Edge Impulse
* <https://www.researchgate.net/publication/320733074_Novel_Phase_Encoded_Mel_Filterbank_Energies_for_Environmental_Sound_Classification>: MFE Block explaining process

1. Deployment in C code of MFE Block and DNN

Immagine che contiene testo, disegno, diagramma, schizzo

Descrizione generata automaticamente

* 1. Develop the Input Signal Capture

The input capture was done by using PortAudio library (<https://files.portaudio.com/download.html>). This is a portable cross-platform audio management, that with some predeveloped functions allows capturing or realizing raw data sound connecting devices. The inputs or the outputs must be connected to the platform and in my implementation the code will automatically get the default source. This library requires an OS to work and this code was only implemented to test the code without loading it. When deploying I will have to create a code that takes the data from the microphone manually, but I was focusing on develop a fine MFE Block and DNN, so this was more practical for testing and adjustments. Being only for testing purposes, I implemented 2 functions with that scope. The first one is test\_sampling and picks the data from a header file that I populated with a signal data if I use for previous model development in 1. I had available both the input raw data and the generated features/spectrogram, and I knew the expected result from the neural network, too. So, having the inputs and the expected outputs I would then reconstruct the process of MFE going from raw data to spectrogram and the DNN from spectrogram to output class. Regarding the live\_sampling, is already working for testing, but as of now is difficult to verify if the system works correctly because of sampling that seems to cut off relevant features, so as of now I verified with the raw data of model train and it output the corrected result.

Source code: <https://github.com/Gotta003/Tiny-Embedded-AI-Voice-Recognition-Application/blob/main/3.%20Custom%20C-pipeline/Complete%20Code/src/input_audio.c>

Header code: <https://github.com/Gotta003/Tiny-Embedded-AI-Voice-Recognition-Application/blob/main/3.%20Custom%20C-pipeline/Complete%20Code/include/input_audio.h>

The samples data instead used for testing of MFE block and DNN are here: <https://github.com/Gotta003/Tiny-Embedded-AI-Voice-Recognition-Application/blob/main/3.%20Custom%20C-pipeline/Complete%20Code/include/audio_samples.h>

This is the code where is there is the main, too. This will change when connecting the external device and will become a file that handles hardware connection.

* 1. MFE Block: <https://docs.edgeimpulse.com/docs/edge-impulse-studio/processing-blocks/audio-syntiant>

Immagine che contiene testo, diagramma, schermata, Carattere

Descrizione generata automaticamenteThe objective of this block is to process the raw data inputed in the microphone of Syntiant TinyML NDP101 or by an external microcontroller. All the reference code regarding this phase are:

* Source Code: <https://github.com/Gotta003/Tiny-Embedded-AI-Voice-Recognition-Application/blob/main/3.%20Custom%20C-pipeline/Complete%20Code/src/spectrogram.c>
* Header Code: <https://github.com/Gotta003/Tiny-Embedded-AI-Voice-Recognition-Application/blob/main/3.%20Custom%20C-pipeline/Complete%20Code/include/spectrogram.h>

1. Edge Impulse takes in input a raw data array of 968ms with a frequency of 16000 Hz, leading to 15478 samples with size of int16\_t, so going from -32768 to +32767. This consists in an implicit conversion from analog signal to the digital one. In the case of testing, the array given by Edge Impulse is already in Digital and the stream of portAudio allows getting that ADC as intended. In the case of an external microcontroller, the values will be passed already as digital ones.
2. Pre\_Emphasis – Filter that performs a high-pass that enhances high-frequency components, so the microphone will capture more low-frequency noise and increase high frequencies to make the speech clear. First, is performed a signal normalization dividing by , to have 15478 float values from -1 to 1. Then, is applied this mathematical function corresponds to:

The filter is applied from n=1, because depends on the previous result generated by this filter. α consists in a coefficient used by this filter that can be chosen and Edge Impulse in this Audio Syntiant processing uses 0.96875. The dimension for input and output array is the same, changes the type size, because the input is int16\_t instead the output is composed by float values.

1. Framing – Audio signal is split and segmentate into small window frames. Of each frame. FRAME\_STRIDE determines how much the window moves between frames and in the case the frame size exceeds the available data, it pads with zero values. The number of frames is determined by a macro and in this case considering the input array size of 15478, the FRAME\_SIZE of 512, the FRAME\_STRIDE of 384 (all parameters used by the model trained by Edge Impulse, is possible to obtain that the frames will be 30. For each frame will be performed 3 operations:
   1. Windowing – Windowing helps in reducing spectral leakage in FFT. The window used is a sinusoidal function which is , consider k an incremental value identifying the position in the frame from 0 to 512 and the size will be the frame size so 512. This window is applied by multiplication to the starting frame for the windowing purpose.
   2. Fourier Transform – Converts the signal from time-domain to frequency-domain, extracting that spectrum of audio signal. To do this phase I used an external library called kiss\_fft. The function used is kiss\_fftr because the sample comes from reality, in fact if we used kiss\_fft we will require complex values, but being in the reality is sufficient the usage of a floating-value as input, and as fact the windowing is performed with floating-values, instead the output of this function is a complex number. This phase requires allocation for FFT computation and is representing in the code by cfg which reserves a size of 512 (FRAME\_SIZE). A peculiarity is the output in real-valued valued input processing produces symmetric output, meanwhile that the second half of FFT output will be a mirror of the first half, corresponding to the complex conjugate, so we require only to store half of values to avoid redundant data.
   3. Spectrogram Population – Extracts the magnitude of frequency components and stores them in spectrogram array that will be used in advance. The purpose is to transform these complex values to floating ones again and reuniting this magnitudes in a unique array that will be big the number of frames (30) multiplied by the half of the FRAME\_SIZE, so 256, with a total size of 7680.

After these operation the cfg, the memory allocated for FFT computation is deallocated.

1. Filterbank – After having this array with the various frames magnitudes, all will be passed to a filterbank to extract meaning full features, while reducing irrelevant information. There are two phases:
   1. Creation of the filter – This function tries to construct a set of triangular filters to map frequencies from the linear scale (FFT output) to mel scale, so this will create that mel scale. The scale is divided into equally space points, that depends by the lowest acceptable frequency (0 Hz) and the maximum (8000 Hz) converted in Mel. The various mel points are computing with this function logic: min\_mel+array position \* mel\_spacing. Then, for a better understanding is all reverted back in frequency. Using these 42 values, because of FILTER\_SIZE of 40 +2 because we would like to apply a triangular filter, leading to 40 operations.

This filtering will ensure smooth frequency weighting. This will form a matrix having on rows the filters (40) and on columns a few values equal to 256.

* 1. Application of the filter – Applies filter bank to spectrogram to get mel spectrogram features, because we know that spectrogram consists of multiple frames processing over time. Each frame is computed separately. Each mel filter is applied to sum up the weighted energy from corresponding FFT bins previously calculated. In the end the output is scaled using a logarithmic compression. Is added to a little size of , to avoid the log(0). The results will be given in decibels and they will be a 40 x 40 matrix.

1. Noise floor – I noticed that the feature generated by Edge Impulse were not only in float value from 0 to 1, but values minor of a certain threshold will correspond to a suppression. So, I reconverted the decibel values first in float value, so the energy level of each frame will be normalized according to a noise floor. Then to be even sure I quantized the values to uint8\_t to clamping them staying between [0, 255]. Then they are reverted in float-point representation and then to these are applied noise suppression to leave only strong signals and eliminating low-energy components, like noise. This will output an array of 1600 features that can be given in input in dense neural network.

Immagine che contiene linea, testo, diagramma, Carattere

Descrizione generata automaticamente

To verify a visual correspondence of the generated features with the ones generated by Edge Impulse, I created this script in Python that is responsible to draw the result: <https://github.com/Gotta003/Tiny-Embedded-AI-Voice-Recognition-Application/blob/main/3.%20Custom%20C-pipeline/Complete%20Code/scripts/spectrogram_draw.py>

Immagine che contiene testo, schermata, Policromia, Blu elettrico

Descrizione generata automaticamenteImmagine che contiene schermata, Policromia, diagramma, Elementi grafici

Descrizione generata automaticamente

* 1. DNN Code
* Source Code: [https://github.com/Gotta003/Tiny-Embedded-AI-Voice-Recognition-Application/blob/main/3. Custom C-pipeline/Complete Code/src/dense\_neural\_network.c](https://github.com/Gotta003/Tiny-Embedded-AI-Voice-Recognition-Application/blob/main/3.%20Custom%20C-pipeline/Complete%20Code/src/dense_neural_network.c)
* Header Code: <https://github.com/Gotta003/Tiny-Embedded-AI-Voice-Recognition-Application/blob/main/3.%20Custom%20C-pipeline/Complete%20Code/include/dense_neural_network.h>

In this phase, I took the 1600 features generated and will be process in DNN, which will be composed by 3 FC layers and an output layer. To verify the correctness of the result I used the same header of MFE block testing, but using the feature generated by Edge Impulse seeing if it outputs the exact same class. The components of a FC layer are composed by:

* Neurons - Basic units that receive inputs from all neurons of the previous layer and send outputs to all neurons of the subsequent one.
* Weights - Each connection between neurons has an associated weight, indicating the strength and influence of one neuron on another.
* Biases - Bias term for each neuron helps adjust the output along with the weighted sum of inputs.
* Activation Function - (sed for non-linearity to the model, enabling complex patterns

So, FC takes in input neurons and applies to it the weights and biases that must be extracted by the model deployed. At first I used a binary approach, but to be sure that I extracted the corrected ones, I saved the values in the following header: <https://github.com/Gotta003/Tiny-Embedded-AI-Voice-Recognition-Application/blob/main/3.%20Custom%20C-pipeline/Complete%20Code/include/model_weights.h>

The next step will be making the code work with binary code, but as of now I used the following scripts.

* Weight\_extractor.py uses tensorflow library taking the .tflite file, to print all the outputs present in it and saves the results of each file having “MatMul” (weights) or “BiasAdd” (biases) in the name and these will be saved in weights.npy and biases.npy files respectively.

Code: <https://github.com/Gotta003/Tiny-Embedded-AI-Voice-Recognition-Application/blob/main/3.%20Custom%20C-pipeline/Complete%20Code/scripts/weights_extractor.py>

* Npy\_verifier.py is used only to verify if the correct files were inserted correctly in weights and biases .npy file.

Code: <https://github.com/Gotta003/Tiny-Embedded-AI-Voice-Recognition-Application/blob/main/3.%20Custom%20C-pipeline/Complete%20Code/scripts/npy_verifier.py>

* Convertion\_in\_c\_bin.py is responsible for header creation extracting values from .npy files and creating global static const uint8\_t in the case of weight and float for biases.

Code: <https://github.com/Gotta003/Tiny-Embedded-AI-Voice-Recognition-Application/blob/main/3.%20Custom%20C-pipeline/Complete%20Code/scripts/convertion_in_c_bin.py>

In FC layer the function that is computed is the following:

W consists in weights array, x is input array, b the biases. After, this computation the value is passed through ReLU actiovation function that is used for hidden layers and prevents negative values from propagating and if a negative value is encountered it is set to 0.

The exception is with output layer, because has another activation function which is softmax.

Softmax converts raw scores into probabilities, subtracting max value from inputs for numerical stability. This is an exponential function ensuring that there are non-negative values. The whole is then divided by sum to normalize values between 0 and 1, so it will look like:

This will output each class belonging probability and the sum of all classes probability must be 1.

As of now the structure of the network is:

* 1. First hidden layer – Input: 1600 features, Weights[1600][256], Biases [256], activation function ReLU
  2. Second hidden layer – Input: 256 features, Weights[256][256], Biases [256], activation function ReLU
  3. Third hidden layer – Input: 256 features, Weights[256][256], Biases [256], activation function ReLU
  4. Output Layer - Input: 256 features, Weights[256][OUTPUT\_CLASSES(4)], Biases [OUTPUT\_CLASSES(4)], activation function ReLU and softmax

For the code is missing the deployment on Syntiant NDP101 that will require not in having the files in an header, but in a binary implementation and there must be a parameters optimization, because NDP101 supports only features with 4-bit.

1. Sound detection – Another theme is that we would like to use an external device to activate Syntiant NDP101. A possible solution is using ESP32 with a simple code for classification if there is noise or not and then, first turn on Syntiant NDP101 and passing through I2C the raw data captured and then given that in the algorithm code, developed above and then return the response to ESP32.
2. Speaker Verification

Source: <https://docs.edgeimpulse.com/docs/tips-and-tricks/combine-impulses>

<https://www.mdpi.com/2076-3417/11/8/3603>

Technically, we can take two ways. Use 2 NDP101, one for KWS and the other for SV, but we will have drawback in power consumption, but we will have a better management, or we can combine the two models the one which will have our keyword spotting database and the one that rely on speech detection. This would be a multi-model approach because the DSP block will be practically the same, but the SV will require another step, because KWS uses only MFE feature generated, instead SV uses MFCC.

Immagine che contiene testo, diagramma, Carattere, linea

Descrizione generata automaticamente

Immagine che contiene testo, schermata, Carattere

Descrizione generata automaticamente

4) Size of DNN and the maximum number of layers

The Deep Neural Network Core of Syntiant NDP101 can host at most 589k 4-bit parameters, so 294.500 bytes, with a clock frequency lower than 16MHz and 4 Fully Connected Layers, we can’t implement like a convolution layer, because that support was introduced from NDP115. The memory in the MCU instead is as saw before a Cortex-M0 with 112Kb and the same frequency of the DNN. It will support at most 64 output classifications. So, the internal RAM on the NDP101 is exactly 294.5 KB and is completely dedicated to DNN weights and biases, so is not required an external memory for storing the network.

The NDP101 supports:

* 4 Fully Connected Layers
* No Convolution support natively, but I think that it is implementable
* Maximum 64 output classes

Source: <https://www.syntiant.com/hardware>  
Considering that Edge Impulse when performing the classification starts with 1600 features as neurons input, we can’t use 512 as output, because 1600\*512=819.200 parameters, which are higher that the network support, going down to 256, the first layer will be of 1600\*256=409.600 parameters, then we can do for 2 more layers of 256 neurons (256\*256=65.536 parameters), and then the output theoretically should be at most 64 classes, so the final layer will be 256\*64=16.384. So, the total parameters will be 409.600+65.536\*2+16.384=557.056 parameters, so there is a space of almost 32.000 parameters left Technically, we could go up to 188 classes, because (589.000-540.672)/256=188,78.