

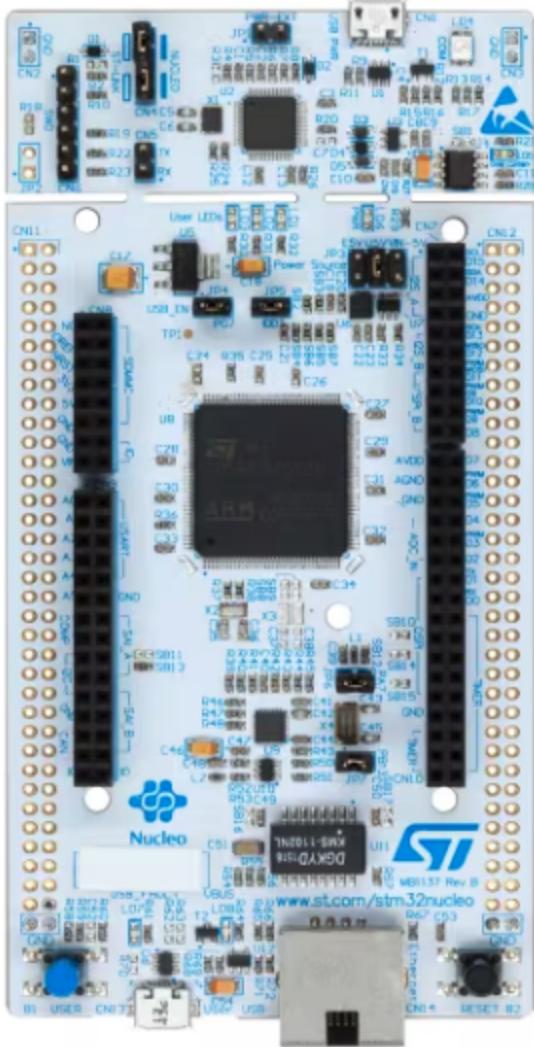
Introduction

TinyML is an exciting new field. It is a fast-paced study of adapting newer machine learning models on low-power, typically in the mW range. In other words, intelligent decisions can potentially be made at the edge compute level, slowly diminishing the importance of the cloud, GPUs, or other centralized locations for AI inference. This matters quite a bit for many reasons:

- Low Latency: Inference happens locally, so there is no delay.
- Energy Efficiency: MCUs use a fraction of a watt compared to the ~500W of a GPU.
- Privacy: Data stays on device. This is one of the more compelling arguments.
- Reduced Cost: Lower electricity translates to controlled and much lower costs.

Of course, these benefits come at a trade-off, primarily in terms of compute power. These devices have limited RAM and Flash, so models must not only fit, but it is practically impossible to train a model on an MCU. Instead, the state of the current field relies on training on a CPU and exporting the model weights to the MCU to be used for inference-only. Additionally, many modern datasets will not work outright and will be required to be pruned, quantized or specifically designed for various chips. Nonetheless, the applications in the IoT space are fascinating and include voice recognition, gesture recognition, environmental monitoring, home security, healthcare devices, and so much more. This project will explore the realm of voice recognition on an ultra low-power STM32 MCU yet high-performance ARM Cortex M7 chip. The introductory goal is to realize a Sine Wave Neural Network classifier to predict the function $y = \sin(x)$ on the STM32 MCU to examine the validity of the main thesis. Once established, the primary goal was to generate my own proprietary dataset of 3 keywords on my voice samples, train the dataset using a Neural Network with MFCC in Keras, and export the weights of the model as raw C code to be used as inference material on the [STM32Cube.AI](#) software.

Data/Technological Sources



- [STM32Cube.AI](#): This is the IDE made for the STM32 series. I made extensive use of this fascinatingly confusing tool while brushing up on my C code.
- X-CUBE-AI: Software that enables tinyML to be run in the [STM32Cube.AI](#) Ide. This takes in a pre-trained model, such as Keras as I used, in order to produce multiple C files for the user to manually change and adapt to their inference needs. I found the header

- The Nucleo-F767ZI Development

Board. A nice feature that enabled easy development include a Floating-Point Unit (FPU), which helped greatly in not forcing integer conversions on model weights. Another nice feature that enabled quick iteration was the ST_Link debugger. On the top right, the USB port enabled a GDB session that made debugging raw C code much easier.

files for the network weights to be the most useful and appreciated many of their built-in functions that saved a lot of time determining things like input and output buffer size.

- Keras: Training primarily occurred in a Python/Keras environment. I have tried to adapt a raw TensorFlow Lite environment, but given the time constraints, found the low-level implementations harder to maintain on-chip.
- AirPods Pro Generation 4: I used the microphone on these to record all portions of the training dataset, which was composed of 3 classes for different keywords: [“Hey”, “Hi”, and “Hello.”]. Each folder contained 30 examples of 1-2 seconds of audio. I recorded the entire dataset as .m4a and later converted these to .wav to make it easier on the MFCC.
- Logitech Webcam 520E: I used the microphone on this to record a separate validation set of 5 samples of each keyword because I was curious to see how the model would perform on data it had never seen before.
- Librosa: For MFCC purposes.

Methods Employed

The primary method of capturing voice patterns came through an MFCC interpretation to be used in a dense Neural Network. MFCC, or Mel-Frequency Cepstral Coefficients, which came through multiple research papers I had read that adapted this as their respective techniques. A common citation across the board highlighted that they are incredibly efficient while maintaining perceptual relevance. I first began by standardizing all audio samples under the same sampling rate (16 Mhz) then following up by converting audio into a mono channel and normalizing the amplitude of volume for variations in recording volume. I purposely included examples of myself speaking loudly and softly across random iterations to hopefully capture more variance.

In feature extraction, an MFCC uses a short-time Fourier Transform (FFT), then maps the powers of the spectrum onto a scale. A fun fact I learned through this project was that mel-frequency bands, which are frequency ranges defined by a nonlinear transformation, are modeled after the voice system of a human! Meaning, future implementations of this project could not only just detect keywords, but even who is speaking. **That said, MFCC values are not robust in the presence of additive noise.** To combat this, researchers often propose adjustments to their own MFCC models, and they sometimes end up building their own proprietary system in which they change the mel-amplitudes before taking the Fourier Transform. The end result I had the MFCC produce was a numerical array that averaged over time and I specifically found that yielding a 13-dimensional vector gave me the best results from quick iteration. This 13 dimensional vector was fed in a Neural Network of 13 Nodes with two dense layers with 32 neurons each. I opted for the ReLU activation function, and made the last layer a softmax with 3 output classes, one for each keyword.

Results:

In training, both Neural Networks (Sine Wave and Voice) worked. The Sine Wave function was not a classification problem, but as a regression it performed quite well. The average Mean Squared Error was below 0.001 and respectively the Mean Absolute Error was 0.02, indicating a very strong fit on accuracy as well as fitting close to a true sine wave. I was not concerned with overfitting here, as this is not only a testing function for validation, but also because the sine wave is well-defined (fully-defined). Meaning the model will naturally generalize to unseen inputs. After exporting onto [STM32Cube.AI](#), I was able to get the model up and running on the

STM32 with various configuration issues along the way. Some of the most common and frustrating ones to debug included:

- C Build Errors without a debugger (this later was changed after connecting to a GDB session)
- Input/Output Buffer Size. This was the hardest to debug. Because the network's weights are adapted to one certain size, the buffer you allocate in the main function that holds both the input and outputs to the model has to be perfectly aligned. When I miscalculated the buffer size as more than it should be, the model returned malformed data. Likewise, a buffer too small led to inaccurate output, which was difficult to distinguish from a model training error, STM32 proprietary software error, or a buffer error. For the sine wave, it was almost always a buffer error.

The voice model was significantly more challenging to adapt properly on the STM32. This comes from the unique fact that the [STM32Cube.AI](#) IDE and extensions are not open-source. Meaning, under the hood quantization, compression, but more importantly, the softmax and other output functions are slightly different than a Python environment. There was this fascinating error that occurred on some of the MFCC, where I would get the correct classification in the Python environment, but get misclassified in the STM32. In the raw Python environment, the voice keyword spotter achieved metrics from both speakers of [80, 77, 75] for accuracy, F1, and recall, respectively. That alone is not ideal, and suggests the model does a good job predicting the keywords, but does not perform to industry or well-acknowledged standards. Some reasons for this include the poor microphone quality on the AirPods as well as the combined fact that the

keywords generally sound alike. With regards to “hi” vs “hey,” especially considering they phonetically are similar, the MFCC had trouble distinguishing both inputs.

More interesting is the fact that the STM32 metrics on the same metrics performed subpar to this standard, achieving [61, 65, 64] on the same criteria of accuracy, F1, and recall. It took me a while to understand the source of this error until I did a deeper dive into the numerical data.

Take this MFCC array for my voice saying “hi” →

```
float hi1[13] = { -578.3177, 43.5092, -14.5059, 5.1716, -14.7966,  
-2.1161, -4.4688, -4.5388, 1.6780, 3.7076,  
-0.5514, -3.7969, -4.8062 };
```

I put this array both in the STM32 and copied it exactly for the Python version →

```
hi1_array = np.array([-578.3177, 43.5092, -14.5059, 5.1716, -14.7966, -2.1161, -4.4688,  
-4.5388, 1.6780, 3.7076, -0.5514, -3.7969, -4.8062], dtype=np.float32)  
  
pred = model.predict(hi1_array)  
  
print("Raw model outputs:", pred)
```

Yet the raw outputs were different!

The STM32 outputs **[0.053274 0.023260 0.923466]**

While the Python outputs **[0.2952013 0.04858567 0.6562131]**

This is because the MFCC is prone to even the smallest fluctuations. Even though in this example, the answer was the same (still “hi” was accurately predicted), there’s no question that the source of error lies on the [STM32Cube.AI](#) software, which is known for slightly altering the

weights. Both models were fed the exact same raw input and received the same preprocessing techniques, so the only logical conclusion is somewhere along the lines of exporting the weights and the software used to interpret the AI network caused a slight discrepancy.

This discrepancy is not always a bad thing, however. The software I'm using, after reading the user manual, found that it automatically optimizes for not just space, but real-time performance. This is the trade-off a researcher can expect when building tinyML models; accuracy is the first metric to drop during optimizations. Although, for this particular dataset, I did not need to quantize, so even after turning off auto-quantization, I found a slight accuracy increase, but not to where I assumed it would lie. Post-quantization yielded a new output matrix where the predicted value for "hi" was [0.84652707], which is closer to the expected output of [0.6562131], but still, not enough to say that was the only problem.

Works Cited

Duy, Anh, and Pham. "Bachelorthesis Implementation of a Speech-Command-Interface on Microcontroller with TinyML." 21 June 2021.

Malche, Timothy, et al. "Voice-Activated Home Automation System for IoT Edge Devices Using TinyML." *Discover Internet of Things*, vol. 5, no. 1, 6 June 2025,
<https://doi.org/10.1007/s43926-025-00165-x>.

Kadir, Ahmad Dziaul Islam Abdul, Ahmed Al-Haiqi, and Norashidah Md Din. "A dataset and TinyML model for coarse age classification based on voice commands." 2021 IEEE 15th Malaysia International Conference on Communication (MICC). IEEE, 2021.

****I found this paper to be of most particular use, and future work will look at how they were able to make their dataset more robust.****