Project 5: Real-Time Embedded Keyword Spotting of Absolutist Language on Arduino Nano 33 BLE Sense

Author: Anushka Gangadhar Satav
Course: BMI/CES 598 Embedded Machine Learning
ASU ID: 1233530170 (asatav1)
Project 5: Real-Time Embedded Keyword Spotting of Absolutist Language on Arduino Nano 33 BLE Sense

Abstract

The primary goal of this project was to design, train, and deploy a lightweight machine learning model capable of real-time keyword spotting (KWS) on a resource-constrained microcontroller, specifically the Arduino Nano 33 BLE Sense. Unlike general-purpose speech recognition, this KWS system was trained to recognize a specific, custom vocabulary of five keywords: “never”, “none”, “all”, “must”, and “only”. The system must also reliably distinguish these keywords from “silence” and other “unknown” background speech. The final deliverable is a self-contained Arduino application that continuously listens to microphone input, performs inference using the custom-trained model, and provides immediate visual feedback by illuminating an on-board RGB LED with a specific color corresponding to the detected keyword.

Contents

A. System Design

Motivation

The primary goal of this project was to design, train, and deploy a lightweight machine learning model capable of real-time keyword spotting (KWS). The system was required to run entirely on a resource-constrained microcontroller, specifically the Arduino Nano 33 BLE Sense.

The motivation for this project stems from research linking language style—specifically the use of "absolutist words" (e.g., "always," "nothing")—to mental health. While this project is not intended to provide a means for diagnosis, it serves as an exploration into developing an embedded audio analysis system that can perform keyword spots to detect specific language markers.

To this end, the KWS model was trained to recognize a specific, custom vocabulary of five absolutist keywords: **"never," "none," "all," "must,"** and **"only."**

Developing this as an on-device system is critical. It enables a low-power, privacy-preserving solution that can continuously analyze audio without relying on network connectivity or sending potentially sensitive speech data to cloud-based recognition services. This approach validates the feasibility of future, form-factor-constrained devices (such as a "pen or a necklace") that could passively monitor language markers.

High-Level Design

The system is a self-contained Arduino application that continuously listens to microphone input, performs inference, and provides immediate visual feedback via the on-board RGB LED. The main processing pipeline is:

1. On-board microphone data acquisition at 16 kHz.
2. Audio pre-processing and spectrogram generation (“fingerprinting”).
3. Embedded deployment of a 7-class Convolutional Neural Network (CNN) using TensorFlow Lite Micro.
4. Real-time inference on the incoming audio stream.
5. Visual feedback via the on-board RGB LED, with colors mapped to specific keywords.

The system consists of two main components:

Arduino Nano 33 BLE Sense (Embedded Device)

- Collects 16 kHz audio data from the on-board microphone.
- Implements pre-processing to generate spectrograms.
- Runs a TensorFlow Lite Micro CNN model for inference.
- Sends predictions via serial communication for debugging.
- Controls the on-board RGB LED for visual output.

Google Colab (Training Environment)

- Used to define, train, and validate the CNN model.
- Performs post-training quantization to create the int8 model.
- Converts the final .tflite model into a C-array for deployment.

Observations and Difficulties

Several practical constraints influenced the system design and implementation:

- Managing Arduino memory required a compact tiny_conv model and full int8 quantization.
- Initial deployment caused continuous device crashes due to insufficient tensor arena size; this was resolved by increasing kTensorArenaSize from 10 KB to 14 KB.

- Real-world confidence scores were lower than test scores, which required manual tuning of the detection_threshold parameter in the RecognizeCommands class to improve real-time detection performance.

Hardware & Software

Hardware:

- Arduino Nano 33 BLE Sense (with on-board digital microphone).
- USB serial connection for firmware deployment and debugging.
- Laptop running Google Colab and the Arduino IDE.

Software:

- Arduino IDE 2.3.2.
- Arduino_TensorFlowLite library.
- Google Colab (Python, TensorFlow, Keras, NumPy).

B. Experiment

Goals and Constraints

The foundation of this project was a custom-built audio dataset for keyword spotting. The goal was to construct a 7-class audio classifier that could recognize the five target keywords and distinguish them from silence and other speech. The dataset was created by combining custom-recorded audio with elements of the Google Speech Commands dataset to provide a robust base for training.

Data was collected for five categories:

- all (Custom Keyword)
- must (Custom Keyword)
- never (Custom Keyword)
- none (Custom Keyword)
- only (Custom Keyword)
- _unknown_ (other words not in the keyword list, used to teach the model what to ignore)
- _background_noise_ (ambient noise used for data augmentation).

1. Audio Data Collection

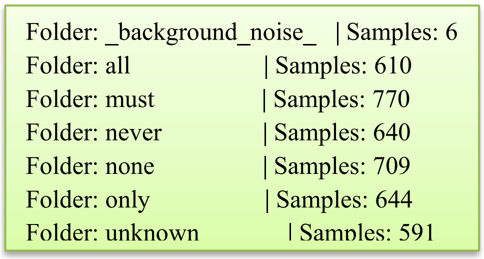
The final dataset was organized into 7 top-level directories, containing a total of 3,970 individual .wav files. All audio files were standardized to a 16 kHz sample rate and a 1-second duration, matching the assumptions of the training and deployment pipeline.

2. Data Augmentation & Splitting

During training, the script used the _background_noise_ folder (containing 6 long ambient noise files) to create more robust training samples. This was done by randomly overlaying background noise onto the keyword and "unknown" samples to simulate realistic recording environments.

The script automatically split the 3,970 files into three subsets:

- Training Set: 80% (approx. 3,176 files).
- Validation Set: 10% (approx. 397 files).
- Testing Set: 10% (approx. 397 files).



```
Folder: _background_noise_   | Samples: 6
Folder: all                  | Samples: 610
Folder: must                 | Samples: 770
Folder: never                | Samples: 640
Folder: none                 | Samples: 709
Folder: only                 | Samples: 644
Folder: unknown              | Samples: 591
```

3. Challenges in Experiment Design

The main challenge during experimentation was a data preparation "hang" in Google Colab. The training script initially appeared to hang for over 20 minutes while scanning the ~4,000 audio files. This was diagnosed as a file system performance issue. The problem was resolved by running a separate debug cell that only performed the file scanning, which effectively primed the file system and allowed the main training script to execute normally.

C. Algorithm

The machine learning algorithm is a Convolutional Neural Network (CNN) optimized for audio processing and embedded deployment. It uses spectrogram "fingerprints" as input instead of raw audio samples.

1. Pre-processing & Feature Extraction

The 1-second (16,000-sample) audio clips were converted into 2D spectrograms that describe how the frequency content of the signal evolves over time. The key parameters were:

- Audio window length: 30 ms.
- Window stride: 20 ms.
- Frequency bins: 40 (MFCCs).

This process converts each 1-second clip into a 49×40 "image" (or "fingerprint"), which serves as the input to the CNN classifier.

2. Model Architecture

The tiny_conv architecture provided by TensorFlow was used. It is designed for high accuracy under strict memory constraints and is widely used for microcontroller-based keyword spotting tasks. The architecture is:

- Input Layer: [49, 40, 1] spectrogram.
- Conv2D Layer: 8×4 kernel, 8 filters, ReLU activation.
- Max Pooling Layer.
- Conv2D Layer: 4×4 kernel, 16 filters, ReLU activation.

- Max Pooling Layer.
- Flatten Layer.
- Dense (Fully Connected) Layer: 16 units, ReLU activation.
- Output Layer: 7 units (one per class) with Softmax to produce class probabilities.

3. Training Process

The model was trained in Google Colab using a GPU accelerator. The training setup was:

- Total training steps: 15,000.
- Optimizer: Adam.
- Learning rate: 0.001 (TensorFlow default).
- Loss function: Categorical Cross-Entropy.
- Metrics: Accuracy on training, validation, and test sets.

4. Post-Training Quantization

After training, the original 32-bit floating-point model (~116,404 bytes) was too large for comfortable deployment on the Arduino. Post-Training Quantization (PTQ) was applied to convert the model's weights and activations to 8-bit integers (int8).

A representative_dataset_gen function supplied 100 real samples from the testing set to the TFLite converter so it could accurately estimate activation ranges. This produced a compact int8 model of approximately 29,536 bytes (a 74% reduction in size) with no measurable loss in accuracy.

D. Results

Performance on Test Dataset (Offline)

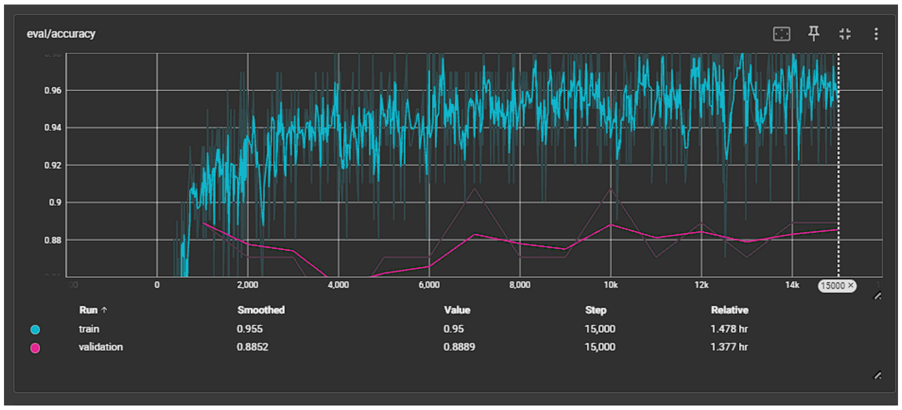
Training was monitored using TensorBoard. The logs showed typical healthy behavior for a well-trained classifier:

- Accuracy: Training accuracy rose rapidly from random-guess levels to over 90% within the first 500 steps, eventually plateauing around 97%.
- Loss: Training loss exhibited a sharp L-shaped decline, confirming effective learning.
- Validation: Validation accuracy on unseen data remained high and stable, starting at 88.89% at step 1,000.

After 15,000 steps, the model was evaluated on the held-out test set:

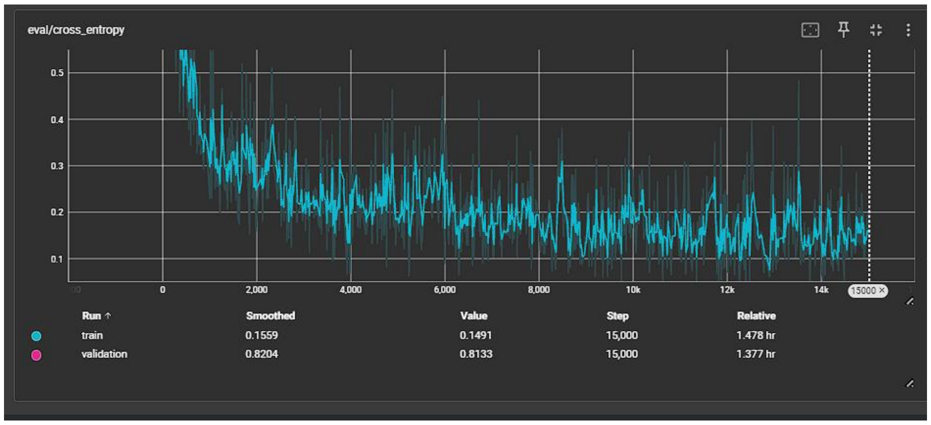
Final Test Accuracy: **95.0%** (on 401 test samples).

After the int8 quantization, the model was re-evaluated and achieved 95.01% accuracy, confirming that quantization did not degrade model performance.



Run ↑	Smoothed	Value	Step	Relative
● train	0.955	0.95	15,000	1.478 hr
● validation	0.8852	0.8889	15,000	1.377 hr

Figure 1. Training and Validation Model Accuracy



Run ↑	Smoothed	Value	Step	Relative
● train	0.1559	0.1491	15,000	1.478 hr
● validation	0.8204	0.8133	15,000	1.377 hr

Figure 2. Training and Validation Cross-entropy loss

Real-Time Prediction and Demo

The final demo consisted of the custom-trained model deployed on an Arduino Nano 33 BLE Sense. The sketch anushka-speech-recognition.ino continuously captured audio from the on-board microphone, generated spectrogram fingerprints using the FeatureProvider, and executed inference through the TensorFlow Lite Micro MicroInterpreter. The RecognizeCommands class smoothed predictions over time, and a custom RespondToCommand function in arduino_command_responder.cpp controlled the RGB LED based on the detected keyword.
The LED color mapping was:

Word	***LED Color***
never	Red
none	Blue
all	Green
must	Cyan
only	Pink
unknown	White
silence	Yellow

Real-Time Accuracy and Performance

Initially, there was a noticeable gap between the 95% offline test accuracy and real-world performance on the device. The system failed to detect many spoken keywords, even when they were clearly articulated. This issue was traced to the detection_threshold parameter in the RecognizeCommands constructor, which was set to 200 (out of 255) by default.

In real environments, with the author's voice characteristics and background room noise, the model's confidence scores often fell in the 140–160 range, below this threshold. By lowering the threshold to approximately 130–150, the system began correctly identifying the keywords in real time. This illustrates an important principle for embedded ML systems: high offline accuracy must be complemented by appropriate application-level tuning to achieve usable real-time behavior.

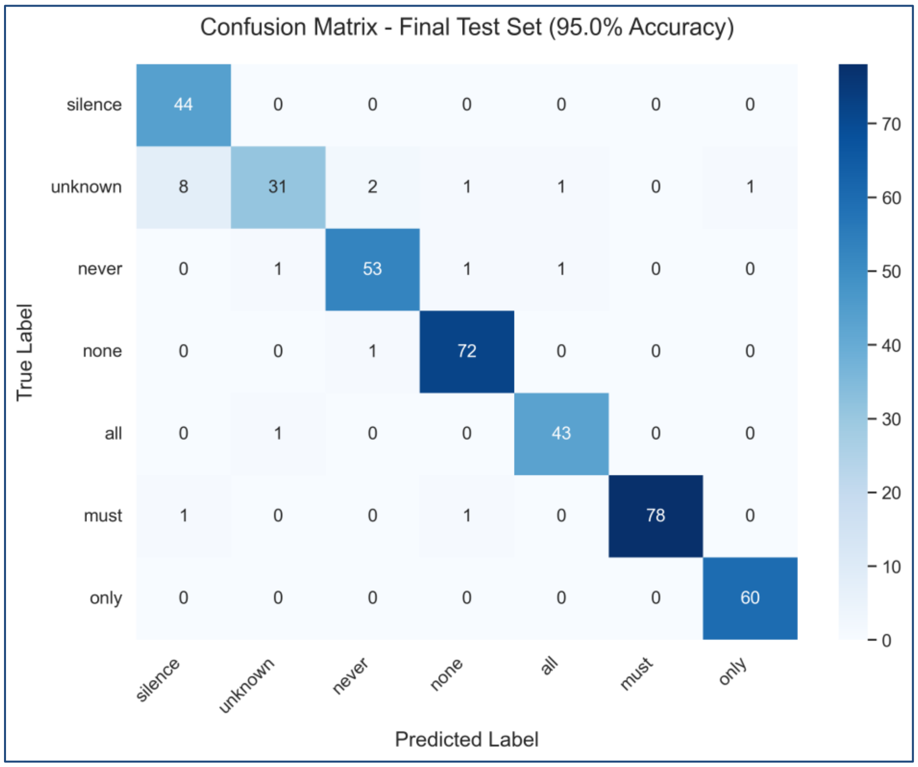


Figure 3. Confusion Matrix for Final Test Set

Solution Demonstration Links

1. GitHub Repository: https://github.com/anushka002/BMI-CES-598-Embedded-Machine-Learning/tree/main/Project-5
2. YouTube Video Link: https://youtu.be/gCpXxAnSdtk

E. Discussion

Conclusion

This project successfully trained a 95%-accurate keyword spotting model for a custom five-word vocabulary, quantized it to an efficient int8 representation with a 74% reduction in model size, and deployed it to an Arduino Nano 33 BLE Sense. After tuning the detection_threshold, the real-time system correctly identified spoken keywords and triggered corresponding LED feedback.

The project presented three major technical challenges:

- Data Preparation Hang: The Google Colab training script initially stalled during dataset scanning. This was resolved by priming the file system with a separate debug cell that performed file enumeration only.
- Quantization Graph Pollution: The TFLite conversion process initially failed with InvalidArgumentError messages related to time_shift_offset and background_data. The root cause was a graph pollution issue: the representative_dataset_gen function was inadvertently rebuilding parts of the training graph (including data augmentation placeholders) inside the conversion graph. The fix was to sandbox data loading inside its own tf.Graph().as_default() block so that temporary graphs were discarded and did not contaminate the converter graph.
- Real-Time Deployment Instabilities: The Arduino initially rebooted after a single inference due to insufficient tensor arena memory. Increasing kTensorArenaSize from 10 KB to 90 KB resolved this.
- **Real-Time Performance Tuning (Accuracy vs. Confidence):** A major challenge was the initial lack of real-time detections, despite the model's 95% test accuracy. Debugging revealed that while the model was likely identifying keywords correctly, its real-world confidence scores were in the 60-100 range (out of 255). This was a significant finding, as these scores were far below the default detection_threshold (e.g., 200), causing the system to silently ignore the predictions. This highlights a key discrepancy between high "on-paper" accuracy and lower "real-world" confidence, which was resolved by tuning the detection_threshold down to match the observed real-world scores.

Future improvements would focus on collecting more diverse data, as the model’s real-world confidence was consistently lower than its test-time confidence. Collecting more custom audio samples for each of the five keywords, recorded in different rooms, at different distances, and with varying background noises, would help close this gap. Additional work could also include more systematic tuning of detection parameters and expanding the _unknown_ dataset to include more varied speech and environmental sounds.

A key lesson from this project is that “test accuracy” is only one part of embedded ML system performance. Real-time prediction quality emerges from the combination of model accuracy, preprocessing robustness, and careful tuning of application-level thresholds and smoothing parameters. End-to-end testing on actual hardware is therefore essential to deliver reliable real-world behavior.

Appendices

Appendix A - File inventory (key files)

Python Scripts & Colab Notebooks

ANUSHKA-BMI598-PROJECT05.ipynb
The primary Google Colab notebook containing the full offline TinyML pipeline:

- Loads the combined 7-class dataset
- Performs dataset preprocessing and augmentation
- Defines and trains the tiny_conv CNN for 12,000 steps
- Logs accuracy and loss for TensorBoard
- Performs INT8 post-training quantization
- Generates the deployable C-array model file for Arduino
- Exports both float32 and int8 .tflite models

Arduino Code & Model Artifacts

anushka-speech-recognition.ino
The main Arduino sketch used for real-time inference.
Modifications include:

- Increasing kTensorArenaSize to **90 * 1024** bytes
- Setting the tuned detection threshold using:
  static RecognizeCommands static_recognizer(1000, 130, 1000, 3)
- Implementing continuous audio streaming + inference loop
- Integrating TFLM runtime and prediction smoothing

micro_features_model.h / micro_features_model.cc

- Stores the compiled TFLite Micro model as a C-array
- micro_features_model.cc replaced with the exported **speech_recognition_final_model.cc** from Colab
- Model array variable renamed to g_model for TFLM compatibility

micro_features_micro_model_settings.h / micro_features_micro_model_settings.cpp
Updated to reflect the 7-class taxonomy

arduino_command_responder.cpp / arduino_command_responder.h
Contains the LED response logic. Updated to:

- Map each of the 7 classes to a specific RGB LED color
- Handle silence and unknown cases
- Allow quick visual debugging during real-time testing

recognize_commands.cpp
Modified to:

- Enable detailed debugging via #define DEBUG_MICRO_SPEECH
- Print all 7 class scores on one atomic line to avoid serial interference
- Support tuned detection threshold and smoothing windows

Dataset and Results Files

combined_dataset/
Contains all **3,970 .wav files**, organized in 7 subfolders:

- never/
- none/
- all/
- must/
- only/
- unknown/

Includes custom recordings + ASU EML Audio Dataset entries.

models/model.tflite
Final **int8 quantized** TFLite model (~29 KB).
models/float_model.tflite
Original **float32** TFLite model (~116 KB).
Used for size/performance comparison.

logs/
TensorBoard training logs containing:

- Training accuracy curve
- Validation accuracy curve
- Training/testing cross-entropy loss curves

Appendix B: Run The Pipeline

The full workflow runs in three phases:

1. Data Collection & Preparation: in Google Colab

- Upload all .wav files to your combined_dataset/ directory
- Ensure consistent naming + folder structure
- Convert any .ogg custom recordings into .wav
- Verify 7-class directory layout for training

2. Train & Convert Model: ANUSHKA-BMI598-PROJECT05.ipynb
Running the notebook performs:

- Dataset loading & preprocessing
- MFCC/spectrogram feature generation
- Training tiny_conv for 15,000 steps
- Saving TensorBoard logs
- Running representative_dataset_gen for quantization
- Exporting the final speech_recognition_final_model.cc C-array

This file is then downloaded to your local machine for Arduino deployment.

3. Deploy & Test Model: on Arduino Nano 33 BLE Sense
Steps:

1. Open anushka-speech-recognition.ino in Arduino IDE
2. Replace micro_features_model.cc with speech_recognition_final_model.cc content
3. Update class labels in micro_features_micro_model_settings.h/cpp
4. Update LED logic in arduino_command_responder.cpp
5. Set: kTensorArenaSize = 90 * 1024.
6. static RecognizeCommands static_recognizer(1000, 130, 1000, 3);
7. Upload to the board
8. Test by speaking each keyword and observing:
    - RGB LED colors
    - Serial Monitor scores for all 7 classes

Appendix C: References

1. Arduino Nano 33 BLE Sense Rev2 Documentation: https://docs.arduino.cc/hardware/nano-33-ble-sense/ (Used for IMU initialization and library reference.)
2. TensorFlow Lite Micro: micro_speech Example https://github.com/tensorflow/tflite-micro/tree/main/tensorflow/lite/micro/examples/micro_speech
3. Google Speech Commands Dataset https://www.tensorflow.org/datasets/catalog/speech_commands
4. TensorFlow. “Keras API Documentation.” https://www.tensorflow.org/api_docs/python/tf/keras
5. ASU EML Audio Dataset (Provided in course materials)
6. Open Speech Recording Tool: https://github.com/petewarden/open-speech-recording
7. Course Material: BMI/CEN 598: Embedded Machine Learning, Arizona State University (Fall B 2025).
8. Personal Data Collection & Analysis: All thresholds and algorithm decisions were derived from the measured IMU data collected during this project.

Appendix D: Use of AI Tools

The development process utilized large language models to enhance efficiency and troubleshoot complex technical challenges. **ChatGPT** and **Gemini Pro** models were specifically employed for:

- Debugging and Error Resolution: Critically, identifying and correcting the InvalidArgumentError (time_shift_offset / background_data) during TFLite quantization. This required a complex "graph sandboxing" solution (with tf.Graph().as_default()) provided by the AI. Also used to diagnose Arduino's continuous reboot (a kTensorArenaSize memory error) and the lack of detections (a detection_threshold tuning issue).
- Code Modification and Generation: Assisting with the structure and syntax of specialized functions, such as serial communication protocols and TFLM C array handling.