Keyword Spotting System

Technical Documentation

Arduino Nicla Voice Deployment

October 27, 2025

Contents

List of Tables

1 Introduction

1.1 Background and Motivation

Keyword spotting is a foundational technology enabling voice-activated interfaces in modern devices. Unlike full automatic speech recognition (ASR) systems that transcribe complete utterances, KWS systems detect specific predefined keywords or wake words. This focused task makes KWS particularly suitable for embedded deployment where computational resources are limited.

The proliferation of IoT devices and edge computing has created demand for on-device voice interfaces that operate without cloud connectivity. Benefits include:

- **Privacy**: Audio never leaves the device
- **Latency**: Sub-100ms response times
- **Reliability**: Functions without internet connection
- **Cost**: No cloud API fees

1.2 Phase Objectives

This Phase aims to develop a complete end-to-end keyword spotting system with the following objectives:

1. Design and implement CNN architectures optimized for embedded deployment
2. Achieve >85% accuracy on 8-keyword classification task
3. Reduce model size for microcontroller deployment
4. Provide multiple architecture variants for different resource constraints
5. Generate deployment-ready artifacts including TensorFlow Lite models

1.3 Target Hardware: Arduino Nicla Voice

Specification	**Value**
Microcontroller	STM32H747AII6 (Dual Core)
Processor	ARM Cortex-M7 @ 480MHz
	ARM Cortex-M4 @ 240MHz
RAM	512 KB (Cortex-M7)
	288 KB (Cortex-M4)
Flash Memory	2 MB
Microphone	Digital MP34DT06JTR (MEMS)
Sample Rate	16 kHz, 16-bit
Power Consumption	10-80 mW (operation dependent)
Operating Voltage	3.3V

Table 1: Arduino Nicla Voice hardware specifications

2 Dataset and Preprocessing

2.1 Google Speech Commands Dataset

2.1.1 Dataset Overview

The Google Speech Commands Dataset v0.02 [?] comprises:

- 105,829 utterances from 2,618 speakers
- 35 word categories (v2) including digits, directions, and common commands
- Each audio clip is 1 second at 16 kHz sample rate (16,000 samples)
- Additional background noise samples for data augmentation
- Standardized validation and test splits provided

2.1.2 Dataset Statistics

Category	**Training**	**Val/Test**
Core vocabulary (12 words)	2,000-3,000	200-300
Auxiliary vocabulary (23 words)	2,000-3,000	200-300
Background noise (6 files)	-	-
Total utterances	84,843	10,493 (each)

Table 2: Dataset composition

2.2 Class Definition and Selection

The system recognizes 10 classes total:

$$\mathcal{C} = \{\text{yes, no, up, down, left, right, on, off, \_silence\_, \_unknown\_}\} \tag{1}$$

Keywords (8): User-selected target words for detection. For this implementation: {`yes, no, up, down, left, right, on, off`}

silence: Ambient noise, background sounds, or absence of speech. Generated synthetically using background noise files.

unknown: All words not in the keyword set (27 other words from dataset). This handles out-of-vocabulary words gracefully.

2.3 Class Imbalance Problem

Original Distribution:

$$N_{\text{keywords}} \approx 3,000 \text{ samples each} \times 8 = 24,000 \tag{2}$$
$$N_{\text{unknown}} \approx 27 \text{ words} \times 3,000 = 81,000 \tag{3}$$
$$N_{\text{silence}}(\text{generated during training}) \tag{4}$$

This creates severe imbalance where unknown comprises 62% of the dataset, causing the model to develop a bias toward predicting unknown class.

2.4 Class Balancing Strategy

To mitigate imbalance, a subsampling strategy is employed:

Algorithm 1 Class Balancing for Unknown Category

Input: File list with labels
Parameters: $\alpha = 3$ (unknown multiplier)
Separate files: $F_{\text{keywords}}, F_{\text{unknown}}$
Compute: $\bar{N}_{\text{keyword}} = |F_{\text{keywords}}|/8$
Set: $N_{\text{unknown}}^{\max} = \alpha \cdot \bar{N}_{\text{keyword}}$
if $|F_{\text{unknown}}| > N_{\text{unknown}}^{\max}$ **then**
  $F_{\text{unknown}} \leftarrow$ Random sample of size $N_{\text{unknown}}^{\max}$
end if
Return: $F_{\text{keywords}} \cup F_{\text{unknown}}$

Result: Unknown class reduced from 62% to 28% of dataset.

Class Type	Before	After	Change
Keywords (8 total)	24,000 (23%)	24,000 (67%)	-
silence	8,484 (8%)	8,484 (24%)	-
unknown	81,000 (62%)	9,000 (25%)	-89%
Total	113,484	35,884	-68%

Table 3: Dataset distribution before and after balancing

2.5 Audio Preprocessing Pipeline

2.5.1 Loading and Normalization

Each audio file undergoes standardized preprocessing:

$$x[n] = \begin{cases} \text{pad}(x[n], 16000 - N, \text{mode} = \text{'constant'}) & \text{if } N < 16000 \\ x[n][0:16000] & \text{if } N \geq 16000 \end{cases} \tag{5}$$

where N is the original sample count. Padding ensures all inputs are exactly 1 second (16,000 samples).

2.5.2 Data Augmentation

1. Time Shifting
Randomly shifts audio in time to improve temporal invariance:

$$x_{\text{shift}}[n] = \begin{cases} 0 & \text{if } n < \Delta t \\ x[n - \Delta t] & \text{if } \Delta t \leq n < N \\ 0 & \text{if } n \geq N \end{cases} \tag{6}$$

where $\Delta t \sim \mathcal{U}(-1600, 1600)$ ($\pm$100ms at 16kHz)

2. Background Noise Mixing
Adds ambient noise to improve robustness:

$$x_{\text{aug}}[n] = x[n] + \alpha \cdot n_{\text{bg}}[n] \tag{7}$$

where:

- $n_{\text{bg}}[n]$ is randomly sampled background noise
- $\alpha \sim \mathcal{U}(0.05, 0.15)$ controls noise intensity
- Applied to 80% of training samples

3. Volume Scaling
Simulates varying microphone distances and volumes:

$$x_{\text{scale}}[n] = \beta \cdot x[n], \quad \beta \sim \mathcal{U}(0.7, 1.3) \tag{8}$$

4. Final Clipping

$$x_{\text{final}}[n] = \max(-1.0, \min(1.0, x_{\text{aug}}[n])) \tag{9}$$

3 Model Architectures

3.1 Design Principles

The architectures follow these principles for embedded deployment:

1. **Depthwise Separable Convolutions**: Reduce parameters by factorizing standard convolutions
2. **Global Average Pooling**: Eliminate large dense layers before output
3. **Batch Normalization**: Enable stable training with higher learning rates
4. **Dropout**: Prevent overfitting on small dataset
5. **Progressive Channel Expansion**: Gradually increase feature maps

3.2 Baseline Architecture

The baseline hybrid CNN combines standard and depthwise separable convolutions.

3.2.1 Architecture Specification

Layer	Type	Output Shape	Params	FLOPs
Input	-	$(100, 40)$	0	-
Conv1D-1	$k = 3, f = 32$	$(50, 32)$	3,872	387K
BatchNorm-1	-	$(50, 32)$	128	6.4K
MaxPool1D-1	$p = 2$	$(25, 32)$	0	-
SepConv1D-1	$k = 3, f = 64$	$(25, 64)$	2,336	175K
BatchNorm-2	-	$(25, 64)$	256	6.4K
MaxPool1D-2	$p = 2$	$(12, 64)$	0	-
Dropout-1	$p = 0.25$	$(12, 64)$	0	-
SepConv1D-2	$k = 3, f = 128$	$(12, 128)$	8,896	320K
BatchNorm-3	-	$(12, 128)$	512	6.1K
GlobalAvgPool	-	(128)	0	-
Dense-1	$n = 128$	(128)	16,512	16.5K
Dropout-2	$p = 0.35$	(128)	0	-
Dense-2	$n = 10$	(10)	1,290	1.3K
Total			**33,802**	**919K**

Table 4: Baseline model architecture details

Metric	Value
Trainable Parameters	30,250
INT8 Size	55.77 KB
Test Accuracy	89.96%

Table 5: Baseline model specifications

3.2.2 Mathematical Formulation

Let $\mathbf{x}^{(0)} \in \mathbb{R}^{T \times F}$ be the input MFCC features where $T = 100$, $F = 40$.

Block 1: Standard Convolution

$$\mathbf{h}^{(1)} = \text{ReLU}(\text{BN}(\mathbf{W}^{(1)} * \mathbf{x}^{(0)} + \mathbf{b}^{(1)})) \tag{10}$$

$$\mathbf{h}^{(1)} \in \mathbb{R}^{50 \times 32} \tag{11}$$

where $\mathbf{W}^{(1)} \in \mathbb{R}^{3 \times 40 \times 32}$ and stride=2 reduces temporal dimension.

Block 2: Depthwise Separable Convolution

Depthwise operation:

$$\mathbf{h}^{(2)}_{\text{dw}} = \mathbf{W}^{(2)}_{\text{dw}} \circledast \mathbf{h}^{(1)} \tag{12}$$

where $\circledast$ denotes depthwise convolution (separate filter per channel).

Pointwise operation:

$$\mathbf{h}^{(2)} = \text{ReLU}(\text{BN}(\mathbf{W}^{(2)}_{\text{pw}} \mathbf{h}^{(2)}_{\text{dw}})) \tag{13}$$

where $\mathbf{W}^{(2)}_{\text{pw}} \in \mathbb{R}^{1 \times 32 \times 64}$ is 1×1 convolution.

Global Average Pooling

$$\mathbf{h}_{\text{pool}} = \frac{1}{T'} \sum_{t=1}^{T'} \mathbf{h}^{(3)}[t, :] \tag{14}$$

reduces spatial dimensions to single vector $\mathbf{h}_{\text{pool}} \in \mathbb{R}^{128}$.

Output Layer

$$\mathbf{y} = \text{softmax}(\mathbf{W}_{\text{out}} \mathbf{h}_{\text{dense}} + \mathbf{b}_{\text{out}}) \tag{15}$$

produces probability distribution over 10 classes.

3.3 Simplified Architecture Variants

3.3.1 Medium Simplified Model

Targets optimal accuracy-size trade-off.

Modifications from Baseline:

- Filters: $32 \to 16$, $64 \to 32$, $128 \to 64$
- Dense layer: $128 \to 64$ neurons
- First convolution stride: $1 \to 2$ for faster dimension reduction

Metric	Medium
Trainable Parameters	8,474
INT8 Size	26.76 KB
Test Accuracy	85.70%

Table 6: Baseline vs Medium simplified comparison

3.3.2 Very High Simplified Model

Maximum compression for extreme constraints.

Modifications:

- Minimal filters: 8 → 16 → 32
- Remove dense layer (direct GlobalAvgPool → output)
- Aggressive striding: stride=2 in multiple layers

Metric	Very High
Trainable Parameters	1,522
INT8 Size	15.81 KB
Test Accuracy	78.13%

Table 7: Very high simplification metrics

3.3.3 Ultra Simplified Model

Extreme minimalism for proof-of-concept.

Architecture:

- Only 2 convolutional blocks
- Filters: 8 → 16
- Direct to output, no dense layers
- Stride=4 and stride=2 for rapid dimension reduction

Metric	Value
Trainable Parameters	1,226
INT8 Size	9.62 KB
Test Accuracy	78.73%

Table 8: Ultra simplified model specifications

3.4 Depthwise Separable Convolution Analysis

3.4.1 Computational Complexity

Standard convolution:

$$\text{Cost}_{\text{std}} = D_K \cdot D_K \cdot M \cdot N \cdot D_F \cdot D_F \tag{16}$$

Depthwise separable convolution:

$$\text{Cost}_{\text{dw}} = D_K \cdot D_K \cdot M \cdot D_F \cdot D_F \quad \text{(depthwise)} \tag{17}$$
$$\text{Cost}_{\text{pw}} = M \cdot N \cdot D_F \cdot D_F \quad \text{(pointwise)} \tag{18}$$
$$\text{Cost}_{\text{sep}} = \text{Cost}_{\text{dw}} + \text{Cost}_{\text{pw}} \tag{19}$$

where:

- D_K: kernel size

- M: input channels
- N: output channels
- D_F: spatial feature map size

Reduction Factor:

$$\frac{\text{Cost}_{\text{sep}}}{\text{Cost}_{\text{std}}} = \frac{1}{N} + \frac{1}{D_K^2} \tag{20}$$

For typical values $D_K = 3$, $N = 64$:

$$\frac{\text{Cost}_{\text{sep}}}{\text{Cost}_{\text{std}}} = \frac{1}{64} + \frac{1}{9} \approx 0.127 \tag{21}$$

This yields approximately **8× reduction** in computational cost.

4 Training Methodology

4.1 Loss Function

Sparse categorical cross-entropy loss with class weights:

$$\mathcal{L} = -\frac{1}{N} \sum_{i=1}^{N} w_{y_i} \log(p(y_i|\mathbf{x}_i)) \tag{22}$$

where w_c are class weights computed as:

$$w_c = \frac{N_{\text{total}}}{K \cdot N_c} \tag{23}$$

K = number of classes, N_c = samples in class c.

4.2 Class Weights

To address remaining class imbalance:

Class	**Weight**
yes, no, up, down, left, right, on, off	1.12-1.22
silence	1.10
unknown	0.44

Table 9: Computed class weights

4.3 Optimization

Optimizer: Adam with initial learning rate $\alpha = 10^{-4}$

Learning Rate Schedule:

$$\alpha_t = \alpha_0 \cdot 0.5^{\lfloor t/5 \rfloor} \tag{24}$$

Applied when validation loss plateaus.

Regularization:

- L2 weight decay: $\lambda = 10^{-4}$ for baseline, $\lambda = 2 \times 10^{-3}$ for simplified
- Dropout: $p = 0.25$ (conv layers), $p = 0.35$ (dense layers)
- Batch normalization: momentum = 0.99, $\epsilon = 10^{-3}$

4.4 Training Hyperparameters

Parameter	Value
Batch Size	64
Epochs	50 (with early stopping)
Initial Learning Rate	10^{-4}
Learning Rate Decay	0.5 every 5 epochs
Early Stopping Patience	10 epochs
Validation Split	10%

Table 10: Training hyperparameters

5 Model Optimization Techniques

5.1 Quantization

Post-training INT8 quantization reduces model size by 4×:

Weight Quantization:

$$w_{\text{int8}} = \text{round}\left(\frac{w_{\text{float32}}}{s}\right) + z \tag{25}$$

where s is scale and z is zero-point.

Activation Quantization:

$$a_{\text{int8}} = \text{clip}\left(\text{round}\left(\frac{a_{\text{float32}}}{s_a}\right) + z_a, -128, 127\right) \tag{26}$$

5.2 TensorFlow Lite Conversion

The model is converted to TensorFlow Lite format for embedded deployment:

```
converter = tf.lite.TFLiteConverter.from_keras_model(model)
converter.optimizations = [tf.lite.Optimize.DEFAULT]
converter.representative_dataset = representative_dataset
converter.target_spec.supported_ops = [
    tf.lite.OpsSet.TFLITE_BUILTINS_INT8
]
converter.inference_input_type = tf.int8
converter.inference_output_type = tf.int8
tflite_model = converter.convert()
```

Listing 1: TFLite conversion code