TinyML Keyword Spotting: Quantization and Pruning on MCU Hardware

Alif Ilham Madani

April 18, 2026

1 Part 1: Preprocessing: Audio Recording and Visualization

In this section, we transform the raw audio signals into representations suitable for machine learning algorithms. Raw audio in the time domain is highly dimensional, meaning that a single second of audio can contain tens of thousands of individual sample points. Also, raw waveforms contain redundant data, noise, and phase variations that make it incredibly difficult for a neural network to recognize underlying patterns. By preprocessing audio into frequency-based representations, we convert long one-dimensional sequences into two-dimensional matrices. This transformation drastically reduces dimensionality while highlighting the most important discriminatory features—such as pitch and energy concentrations across different frequency bands.

To illustrate these transformations, we visualize recorded sound of the words "Yes" and "No" spoken at varying volumes (loud and quiet) across multiple domains. Figure 1 displays the raw time-domain waveforms, while Figure 2 shows the corresponding frequency-domain representations. While these plots provide basic information, they lack the temporal-frequency dynamics necessary for classification tasks.

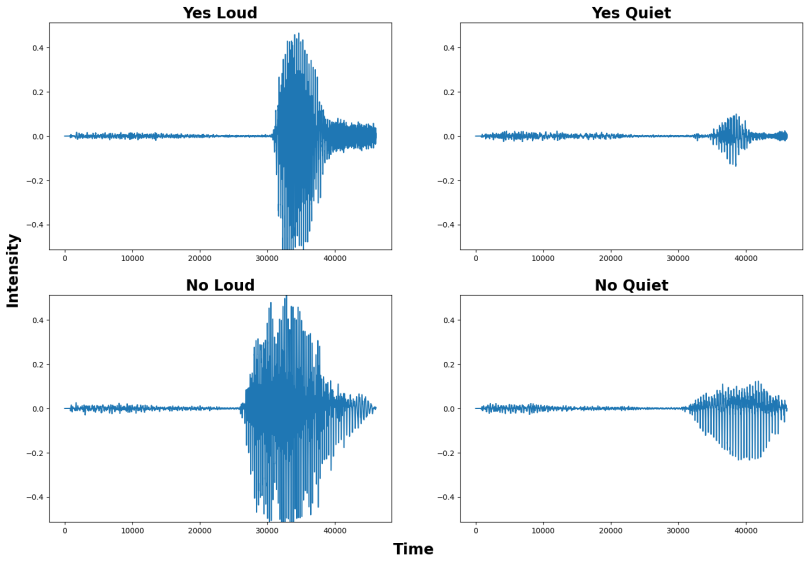

Figure 1: Time domain representation for 'Yes' and 'No' sounds (Loud vs. Quiet).

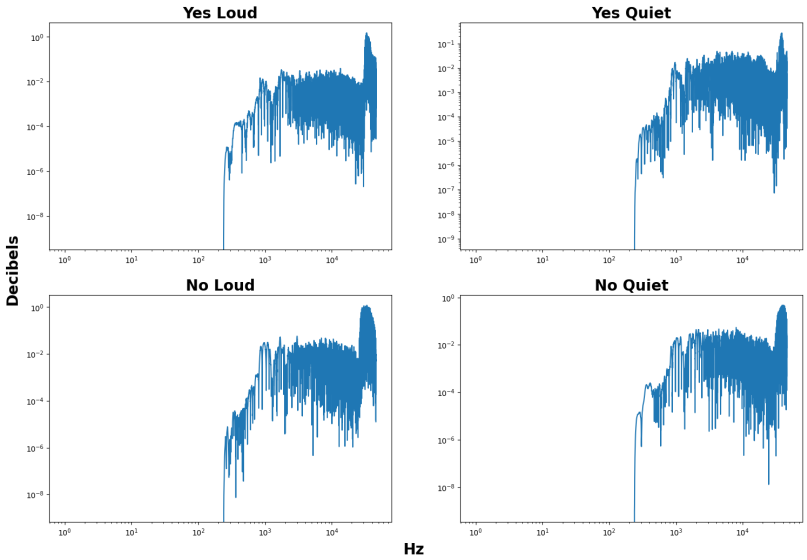

Figure 2: Frequency domain representations of the sounds.

Figure 3 shows a standard spectrogram, which plots frequency against time using a linear frequency scale. However, this mathematical representation treats all frequencies equally. Human hearing is non-linear and much more sensitive to lower frequencies. Therefore, the Mel spectrogram (Figure 4) improves the standard spectrogram by scaling the frequency axis to the logarithmic Mel scale [3, 4]. Figure 5 illustrates the Mel Frequency Cepstral Coefficients (MFCCs), generated by taking the discrete cosine transform of the Mel log powers. This process decorrelates the features and compresses the audio envelope into a highly informative, compact set of coefficients.

In conclusion, representations based on the Mel scale (Mel spectrograms and MFCCs) are significantly more effective for speech and audio classification because they selectively focus computational power on acoustically relevant frequencies. In the context of modern deep learning, the Mel spectrogram is typically the preferred representation. Convolutional Neural Networks (CNNs) excel at processing correlated spatial data and can extract richer, higher-level hierarchical features directly from the two-dimensional visual layout of a Mel spectrogram compared to the heavily compressed MFCCs.

2 Part 2: Model Size Estimation

Deploying neural networks on Microcontroller Units (MCUs) such as the Arduino Nano 33 BLE Sense requires strict hardware requirements, specifically regarding non-volatile flash memory and volatile Random Access Memory (RAM). We profile the TinyConv model which contains 16,652 trainable parameters. Assuming these parameters are stored as 32-bit floating-point numbers (4 bytes per parameter), the weight data occupies approximately 66.6 KB. Given the MCU's 1 MB flash memory capacity, the model uses roughly 6.5% of the available flash space, leaving enough room for the inference framework and application code. During inference, the device's RAM is utilized to store temporary input buffers and intermediate layer activations. The size estimator calculates a forward pass memory requirement of 0.079696 MB (approximately 79.7 KB) for a batch size of 1. This peak memory utilization consumes about 31.1% of the MCU's 256 KB RAM limit, still within constraints without risking out-of-memory errors during real-time execution.

Computational complexity is a major limiting factor for edge devices. The model profiler calculates a total of

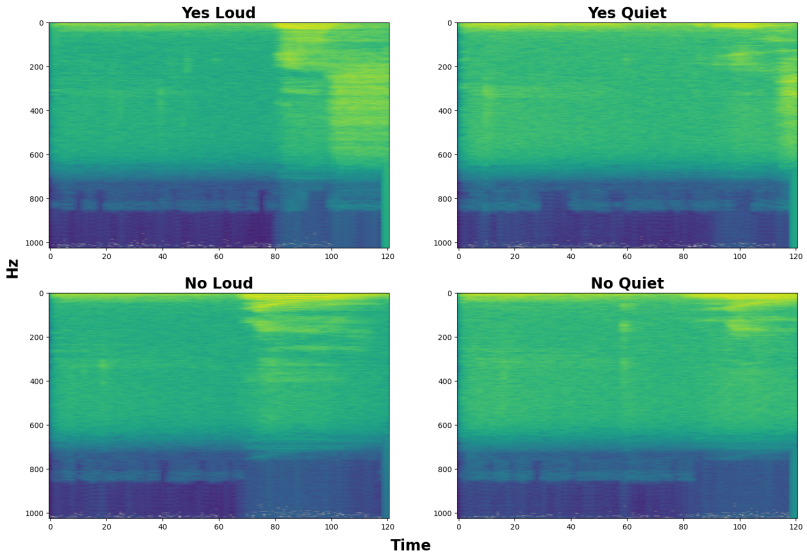

Figure 3: Standard spectrograms mapping Hz over time.

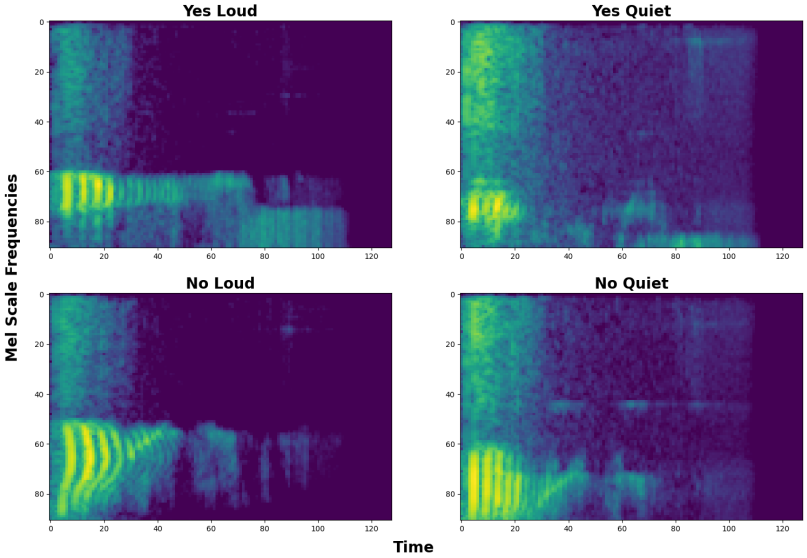

Figure 4: Mel spectrograms utilizing the perceptually-based Mel scale.

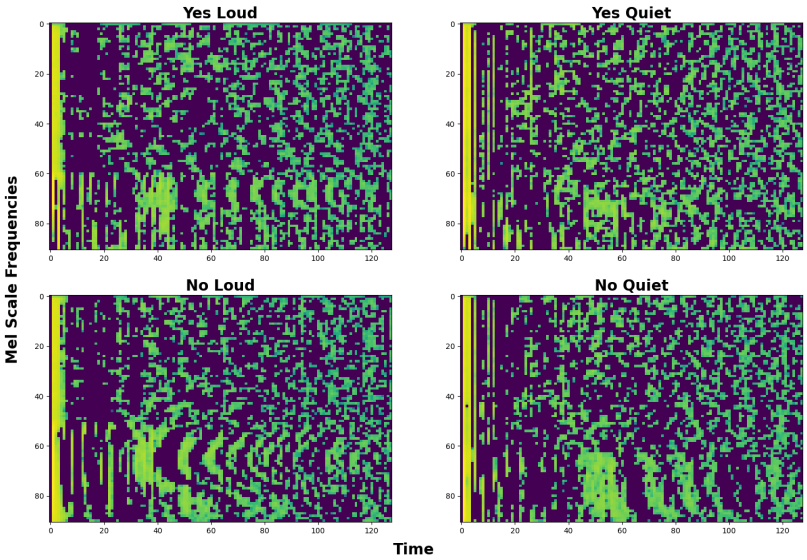

Figure 5: MFCC (Mel Frequency Cepstral Coefficients) representations.

676,004 floating-point operations (FLOPs) per forward pass. To put this into perspective, Zhang et al. demonstrated that a basic CNN constrained to an 80 KB memory footprint requires 5.0 million operations, while an optimized Depthwise Separable CNN (DS-CNN) requires 5.4 million operations to achieve 94.4% accuracy on the Google Speech Commands dataset [6]. At roughly 0.68 MFLOPs, the TinyConv architecture is aggressively downscaled, operating at a fraction of the computational cost of standard baseline models. This trades a potential fraction of accuracy for the computational efficiency required of an always-on MCU.

Finally, benchmarking this lightweight architecture on Colab hardware establishes a baseline for expected inference speeds. On a standard CPU, the model's inference runtime is 1.019 ms. When accelerated on a Colab T4 GPU, the core CUDA execution time drops drastically to just 35.968 microseconds. While the MCU will lack the parallel processing power of a server GPU and the clock speed of a desktop CPU, these benchmarks confirm that the TinyConv's minimal parameter count and low FLOP footprint will translate to low-latency, real-time predictions once deployed on the edge hardware.

3 Part 3: Training & Analysis

After verifying that the TinyConv architecture adheres to the hardware constraints of the microcontroller, the next step is to train the model on our target data. For this keyword spotting task, we utilize the Google Speech Commands dataset (v0.02). While the full dataset supports 35 distinct spoken keywords, our task focuses on a subset mapped to four specific classification categories: the target keywords "yes" and "no", an "unknown" class, and a "silence" class (representing background noise) [5]. Based on the execution of our data preprocessing pipeline, the dataset is divided into three splits: 10,556 samples for training, 1,333 samples for validation, and 1,368 samples for final testing.

The standard 32-bit floating-point (float32) TinyConv model was trained over 50 epochs. As illustrated in Figure 6, the training and validation accuracy curves show a steady and stable convergence. The validation accuracy closely tracks and occasionally exceeds the training accuracy. This indicates that the model generalizes well to unseen data and avoids significant overfitting, which is a common risk when utilizing heavily constrained architectures.

Following the 50 epochs of training, the model's performance was evaluated across all data splits. As detailed in

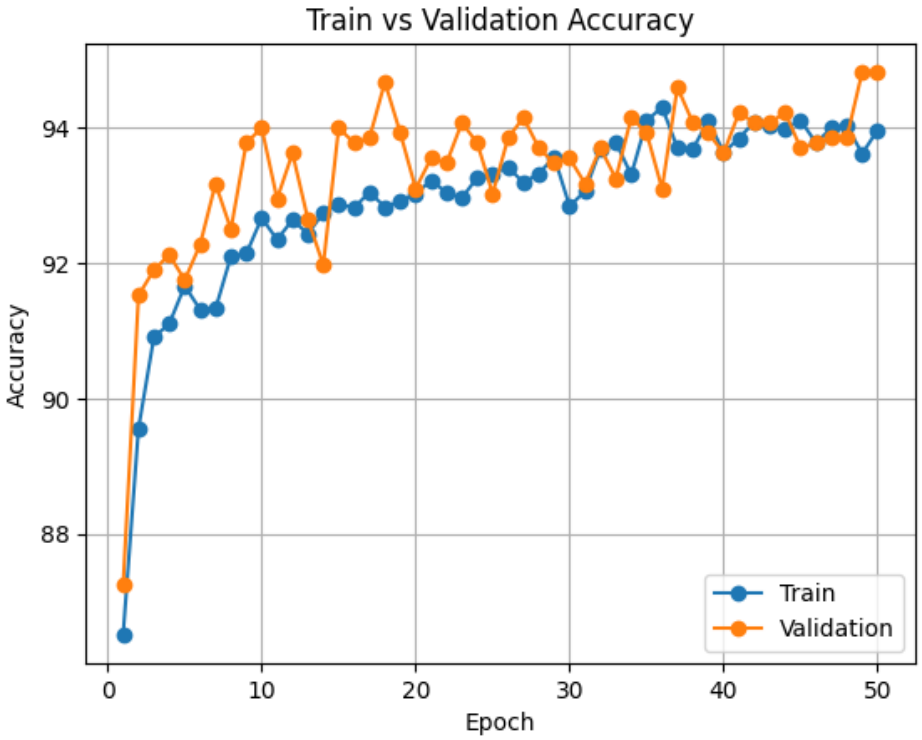

Figure 6: Training and validation accuracy curves over 50 epochs.

Figures 7, 8, and 9, the model achieved a final overall training accuracy of 93.7% and a validation accuracy of 95.1%. On the hold-out test set, the model achieved overall accuracy of 93.2%. A closer look at the test set class breakdown reveals that the model is exceptionally proficient at identifying background "silence" (99.6% accuracy), while the "unknown" class proved to be the most challenging (87.5% accuracy), likely due to the different words encompassed within that category. Overall, these metrics confirm that the baseline float32 model is successfully trained and ready for subsequent quantization and deployment.

training accuracy for float32 TinyConv

	silence_	unknown_	yes	no	total
#samples	2099.000	2099.000	3228.000	3130.000	10556.000
#correct	2089.000	1888.000	3009.000	2901.000	9887.000
accuracy	0.995	0.899	0.932	0.927	0.937

Figure 7: Confusion matrix summary and overall training accuracy for the float32 TinyConv.

validation accuracy for float32 TinyConv

	silence_	unknown_	yes	no	total
#samples	265.000	265.000	397.000	406.000	1333.000
#correct	264.000	247.000	382.000	375.000	1268.000
accuracy	0.996	0.932	0.962	0.924	0.951

Figure 8: Confusion matrix summary and overall validation accuracy.

testing accuracy for float32 TinyConv

	silence_	unknown_	yes	no	total
#samples	272.000	272.000	419.000	405.000	1368.000
#correct	271.000	238.000	387.000	379.000	1275.000
accuracy	0.996	0.875	0.924	0.936	0.932

Figure 9: Confusion matrix summary and overall test accuracy on the hold-out set.

4 Part 4: Model Conversion and Deployment

The PyTorch-trained TinyConv model was exported to TensorFlow Lite (TFLite) and subjected to 8-bit integer quantization to dramatically reduce its memory footprint. This quantized TFLite Micro model was then compiled into a C++ byte array and deployed onto the microcontroller to evaluate its real-world performance.

To profile the computational efficiency of the edge device, the internal hardware timer (`micros()`) was utilized to measure the execution time of a single audio classification cycle. The inference pipeline consists of three distinct phases: acoustic feature extraction (preprocessing), the neural network forward pass, and the interpretation of the output tensor (post-processing). Based on the captured serial logs, the preprocessing phase takes an average of 22.52 ms to convert a raw audio buffer into a valid spectrogram tensor. The neural network execution itself consumes the

vast majority of the cycle, averaging 87.91 ms per inference. Finally, post-processing requires 0.23 ms to parse the output probabilities and determine if a command was recognized.

When comparing the core neural network execution time on the microcontroller to the server-grade hardware benchmarked in Part 2, the difference in computational power is substantial. The Arduino's 87.91 ms inference latency is approximately 86 times slower than the Colab CPU, which executed the exact same forward pass in just 1.019 ms. Furthermore, the MCU is over 2,400 times slower than the Colab GPU's core CUDA execution time of 0.036 ms. Despite this stark contrast, an 88 ms inference latency translates to processing more than 11 audio frames per second. This speed is more than sufficient for a continuous, real-time keyword spotting application operating at the edge.

To evaluate the model's robustness, an in-field test was conducted where the deployed model was prompted with live voice commands in a standard room environment. Out of 20 spoken trials (5 instances each of the "yes", "no", "unknown", and "silence" classes), the model correctly classified 13 instances, yielding an in-field accuracy of 65%. A closer breakdown of the results, illustrated in the confusion matrix in Figure 10, reveals that the model perfectly classified the "no" and "silence" classes (5/5 each) but struggled with "yes" (3/5) and failed entirely on the "unknown" class (0/5), repeatedly misclassifying diverse spoken words as background "silence".

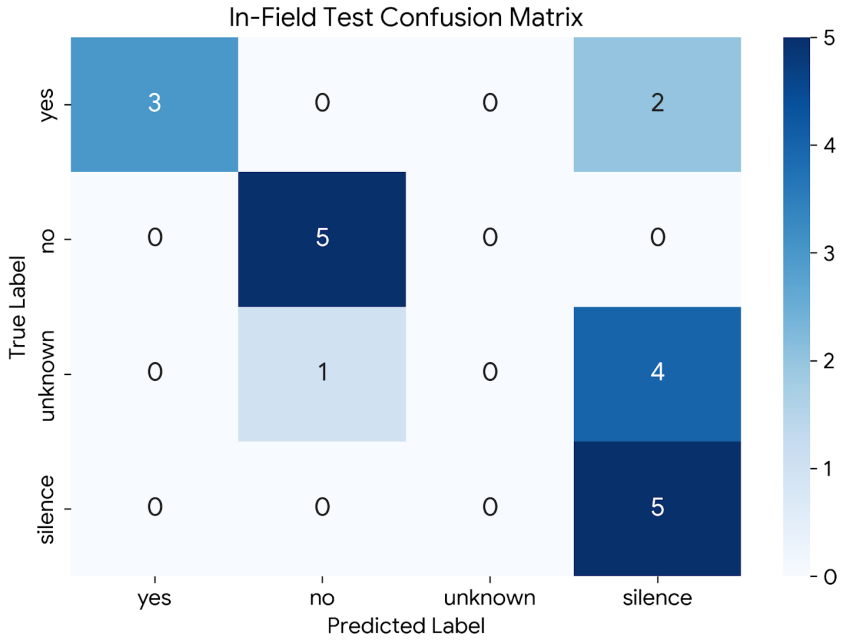

Figure 10: Confusion Matrix for the 20 live in-field inference trials on the Arduino Nano 33 BLE.

This 65% real-world performance represents a significant discrepancy from the 95.1% validation accuracy and 93.2% test accuracy observed during the PyTorch training phase in Part 3. This drop is a well-documented phenomenon in TinyML applications and can be attributed to several factors. First, the model exhibits a distinct bias toward the "silence" class when uncertain. Because the "unknown" category encompasses a highly complex and wide distribution of different words, the aggressive 8-bit post-training quantization likely degraded the model's ability to map these diverse features. This precision loss pushed the activation scores for "unknown" words below the recognition threshold, causing the network to default to predicting background silence.

Furthermore, the model suffers from severe acoustic domain shift. The training dataset consisted of thousands of

tightly cropped audio clips recorded on various high-quality microphones, whereas the live test utilized the specific hardware profile of the Nano 33 BLE's single MP34DT05 MEMS microphone. The unconstrained environmental background noise, combined with the specific vocal characteristic and distance, fell outside the distribution of the original training data, further reducing the confidence of the edge predictions.

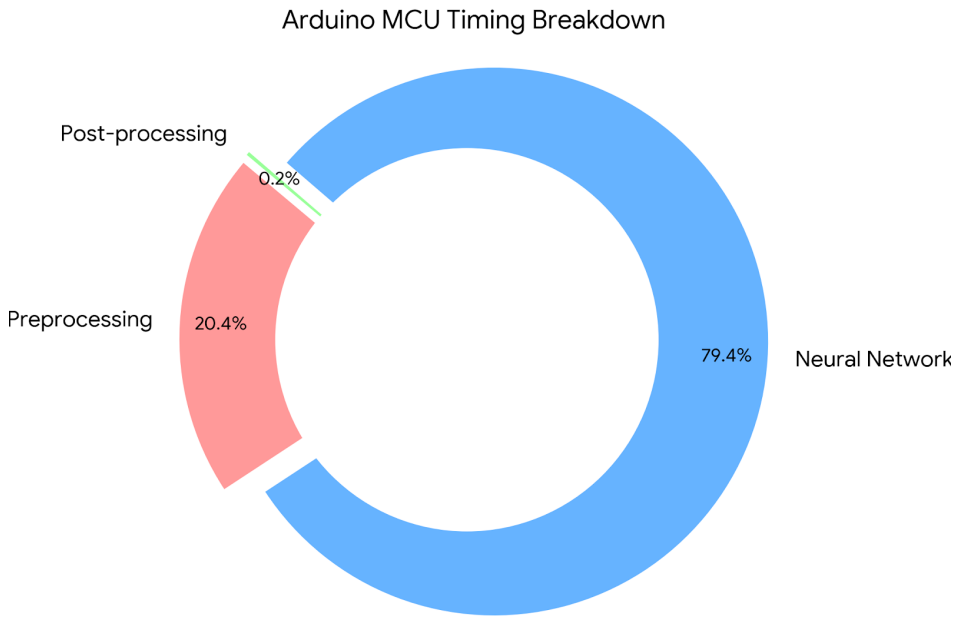

Figure 11: Arduino MCU operational timing breakdown: Preprocessing (22.52 ms), Neural Network (87.91 ms), and Post-processing (0.23 ms).

5 Part 5: Quantization-Aware Training

While Part 3 evaluated a full-precision (float32) model, floating-point representations require 32 bits per parameter, which is highly inefficient for a microcontroller. To solve this, models must be quantized to lower bit-width integers. However, naively rounding weights after training often results in degraded performance. In this section, we implement and evaluate Quantization-Aware Training (QAT) to mitigate precision loss by simulating quantization during the training process itself.

First, we implemented the Straight-Through Estimator (STE) to handle the backpropagation of the rounding function. Because standard rounding operations have zero gradients almost everywhere (which would halt backpropagation), the STE bypasses this non-differentiability by passing the gradients as they are. This allows the model to update its weights while being "aware" of the rounding that occurs in the forward pass [1].

We then implemented the core linear mapping using the scaling factor and zero-point shift formula. To handle different scaling modes, both symmetric and asymmetric quantization forward functions were developed. The symmetric function dynamically clamps rounded values to a centralized range spanning from -2^{k-1} to $2^{k-1} - 1$ with a zero-point of 0. Conversely, the asymmetric function clamps values strictly to positive integers spanning from 0 to $2^k - 1$, incorporating a non-zero origin shift [2]. These parameters are dynamically determined by calculating the

bounds based on the empirical saturation minimum and maximum of the activation or weight tensors. Finally, these mechanisms were unified in a wrapper module that applies "fake quantization" to both activations and pre-trained weights prior to executing the standard linear or convolutional operations, simulating low-precision limits.

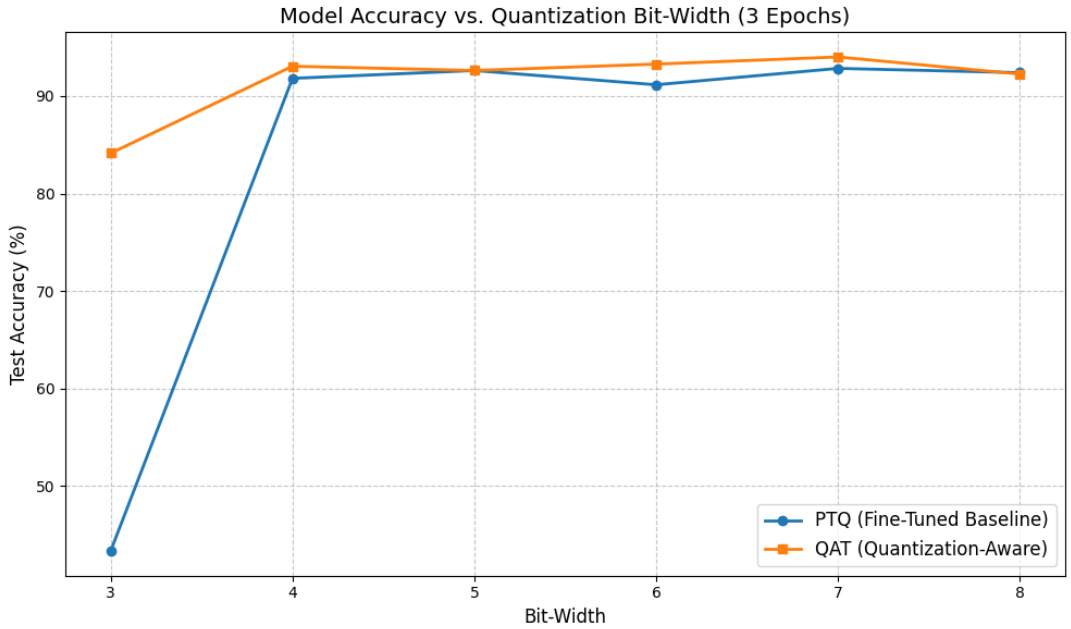

Figure 12: Model Accuracy vs. Quantization Bit-Width (3 to 8 bits) comparing Post-Training Quantization (PTQ) and Quantization-Aware Training (QAT).

To evaluate the effectiveness of this pipeline, we compared the accuracy of standard Post-Training Quantization (PTQ) against Quantization-Aware Training (QAT) across a spectrum of bit-widths from 3 to 8 bits. We did not perform 2-bit quantization, as we encountered numerical instability during the training. As illustrated in Figure 12, both methods perform comparably well at higher bit-widths (5 to 8 bits), maintaining accuracies above 92% with negligible loss compared to the full-precision baseline. At 4 bits, both models still hold up reasonably well, though QAT maintains a slight edge.

However, as the bit-width drops to 3 bits, their trajectories diverge drastically. The PTQ model suffers a catastrophic accuracy degradation, plummeting to approximately 43%. This occurs because PTQ blindly truncates and rounds the trained weights, resulting in high quantization noise. Conversely, the QAT model is quite stable, only dropping to roughly 84% accuracy at 3 bits. By incorporating the fake quantization operations and the Straight-Through Estimator into the training loop, QAT allows the network to iteratively adjust its unquantized weights to account for the severe truncation noise. Ultimately, the model learns how to perform efficiently within a heavily constrained state, proving QAT to be a vastly superior strategy for ultra-low memory edge deployments.

6 Part 6: Pruning

Pruning is a model compression method that reduces model size and increases computational efficiency by removing unused or less important parameters in neural networks. This section explores two distinct methodologies: unstructured pruning, which zeros out individual weights across the entire model based on their magnitude, and structured

pruning, which removes entire groups of weights (such as convolutional channels) to physically shrink the network architecture.

Unstructured Pruning

Unstructured pruning was applied globally to both the convolutional and fully connected layers. By utilizing the L_1 norm as our pruning method, we targeted and zeroed out the weights with the absolute lowest magnitudes across the network.

To evaluate the impact, we measured the test accuracy against the number of non-zero parameters across several pruning thresholds ranging from 0% to 99%. As illustrated in Figure 13, the model shows a "pruning cliff." Without fine-tuning, accuracy drops precipitously once 50% of the weights are removed, falling to roughly 38.6% at a 75% sparsity level. However, incorporating a brief fine-tuning step allows the remaining weights to adjust to the truncated network. With fine-tuning, the model robustly maintains near-baseline accuracy (above 87%) even when 90% to 95% of the parameters are removed. The cliff for the fine-tuned model only appears at 99% sparsity, where the model is too constrained to recover the full accuracy.

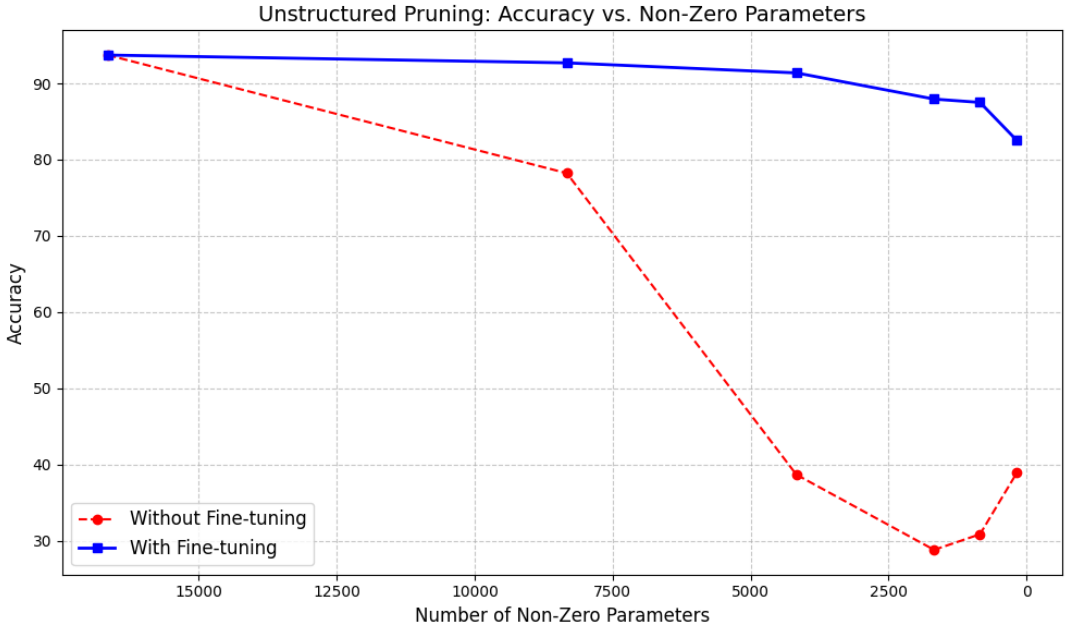

Figure 13: Unstructured Pruning: Accuracy vs. Non-Zero Parameters, demonstrating the pruning cliff with and without fine-tuning.

While unstructured pruning aggressively reduces the number of non-zero weights, it does not naturally speed up computation on standard hardware. Standard dense matrix multiplication operations still perform the mathematical operations for the zeroed-out weights, costing the same amount of time and memory bandwidth. To actually realize computational speedups, the pruned model must be executed using sparse matrix formats (such as compressed sparse row or column formats) and run on specialized software libraries or hardware accelerators (such as sparse Tensor Cores) that are explicitly optimized to skip zero-value multiplications.

Pruning relies on a mathematical norm to determine the importance of a weight:

- L_1 **Norm:** The sum of the absolute values of the weights. It promotes sparsity and is highly effective at identifying the absolute smallest magnitude weights to remove.
- L_2 **Norm:** The Euclidean distance, or the square root of the sum of the squared weights. It penalizes larger weights more heavily but is less aggressive at pushing a wide distribution of smaller weights exactly to zero.
- L_∞ **(L-infinity) Norm:** The maximum absolute value in the tensor. It only evaluates the single largest element, making it unsuitable for evaluating the importance of a wide distribution of weights.

In conclusion, the L_1 **norm** works best for unstructured pruning because it directly targets and eliminates the smallest, least impactful weights while actively driving sparsity.

Structured Pruning

Unlike unstructured pruning, structured pruning removes entire filters or channels. This physically reduces the tensor dimensions, immediately translating to lower RAM usage, fewer Floating-Point Operations (FLOPs), and faster inference times on standard hardware like desktop CPUs and microcontrollers.

As shown in the structured pruning evaluation (Figure 14), physically removing convolutional channels linearly decreases the number of parameters and FLOPs (dropping from roughly 0.67 MFLOPs down to 0.08 MFLOPs at a 90% threshold). The desktop CPU runtime tightly tracks this reduction, falling from the baseline 1.01 ms down to roughly 0.46 ms at 0-% threshold. Because structured pruning removes entire feature maps rather than isolated weights, the accuracy drops more aggressively than in unstructured pruning. Fine-tuning successfully restores reasonable accuracy (around 87.6%) up to a 50% pruning threshold. Beyond the 50% threshold, the model capacity becomes too narrow to accurately classify the speech commands.

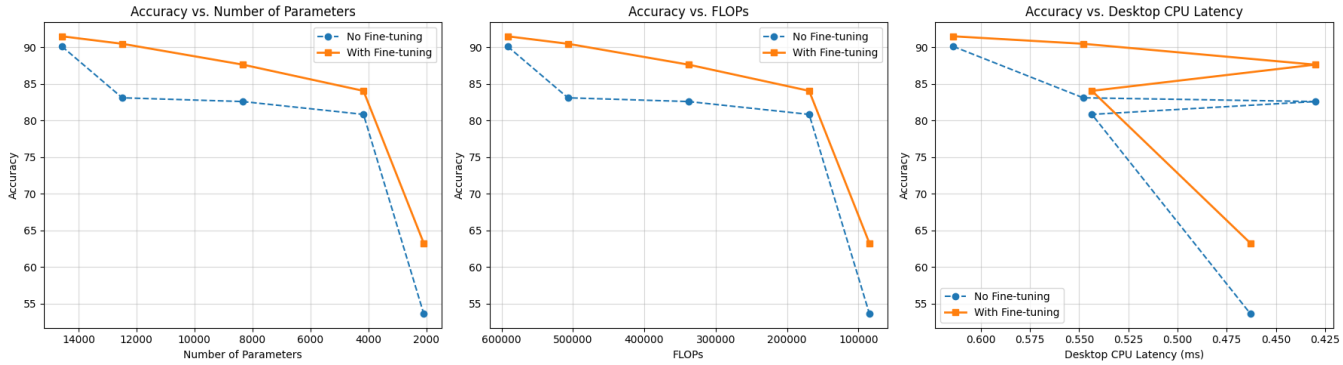

Figure 14: Structured Pruning Metrics: Accuracy vs. Parameters, Accuracy vs. FLOPs, and Accuracy vs. Desktop CPU Latency.

To observe these theoretical speedups, the baseline (0% pruning), 10% pruned, and 50% pruned models were converted to TFLite Micro C++ byte arrays and sequentially flashed to the Arduino Nano 33 BLE. The internal `micros()` timer was utilized to capture the execution time of the neural network forward pass, and live voice trials were conducted to measure real-world accuracy like in Part 4.

As shown in Figure 15, physical channel removal directly accelerates MCU inference. The unpruned baseline executed the forward pass in an average of 87.85 ms and achieved an in-field accuracy of 65%. When 10% of the channels were removed, the inference time dropped to 84.08 ms at the cost of a slight accuracy degradation (60%). At the extreme 50% pruning threshold, the MCU inference time rapidly decreased to just 72.16 ms—representing an impressive 18% speedup over the baseline model.

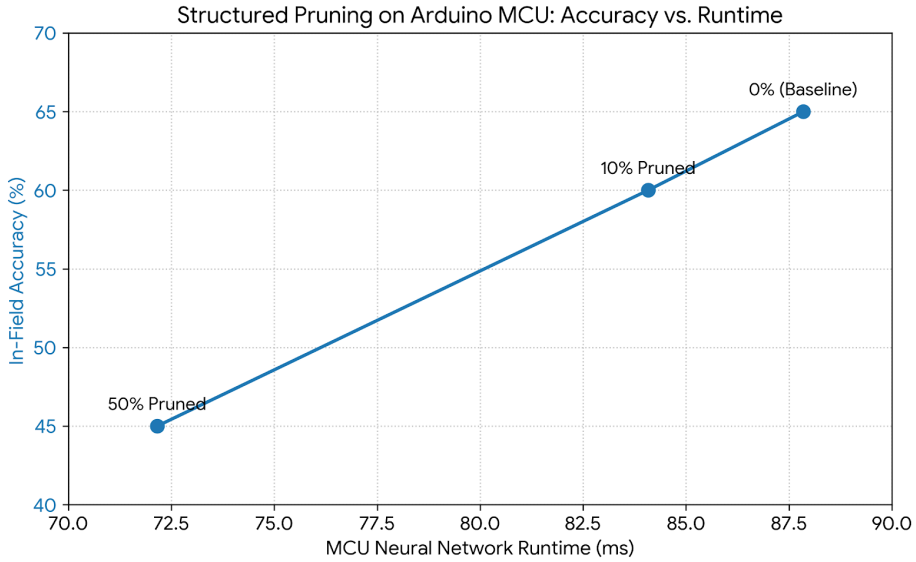

Figure 15: Structured Pruning on Arduino MCU: In-Field Accuracy vs. Neural Network Runtime.

However, because structured pruning removes entire feature maps rather than isolated weights, it severely constrains the network's representational capacity. As the runtime becomes faster, the accuracy drops aggressively, plummeting to 45% in-field accuracy at the 50% pruning threshold. This clearly highlights the core TinyML engineering trade-off: aggressive structured pruning yields highly desirable reductions in power consumption and latency at the direct expense of classification reliability.

References

[1] ApX Machine Learning. *Straight-Through Estimator (STE) - Practical LLM Quantization, Chapter 4*. https://apxml.com/courses/practical-llm-quantization/chapter-4-quantization-aware-training-qat/straight-through-estimator-ste. Accessed: 2026-03-16.

[2] ApX Machine Learning. *Symmetric vs. Asymmetric Quantization - Practical LLM Quantization, Chapter 1*. https://apxml.com/courses/practical-llm-quantization/chapter-1-foundations-model-quantization/symmetric-asymmetric-quantization. Accessed: 2026-03-16.

[3] Ketan Doshi. *Audio Deep Learning Made Simple (Part 1): State-of-the-Art Techniques*. Feb. 2021. URL: https://towardsdatascience.com/audio-deep-learning-made-simple-part-1-state-of-the-art-techniques-da1d3dff2504/ (visited on 03/16/2026).

[4] Ketan Doshi. *Audio Deep Learning Made Simple (Part 2): Why Mel Spectrograms perform better*. Feb. 2021. URL: https://towardsdatascience.com/audio-deep-learning-made-simple-part-2-why-mel-spectrograms-perform-better-aad889a93505/ (visited on 03/16/2026).

[5] Pete Warden. "Speech commands: A dataset for limited-vocabulary speech recognition". In: *arXiv preprint arXiv:1804.03209* (2018). URL: https://arxiv.org/abs/1804.03209.

[6] Yundong Zhang et al. *Hello Edge: Keyword Spotting on Microcontrollers*. 2017. arXiv: 1711.07128 [cs.SD].