Assignment 2: Keyword Spotting with MicroControllers

Yuval Steimberg[1]
ys2335

ECE 5545: Machine Learning Hardware and Systems

Contents

1 Introduction

Embedded machine learning is an emerging field where deep learning models are optimized to run on low-power microcontrollers. This project focuses on deploying a speech recognition model on the Arduino TinyML Kit, utilizing quantization and pruning for efficient inference. The following key areas are covered:

- Preprocessing: Audio feature extraction and visualization.
- Model Size Estimation: Memory and computational requirements.
- Training and Analysis: Model performance evaluation.
- Model Conversion: Transition from PyTorch to TFLite.
- Quantization: Post-training and quantization-aware training.
- Pruning: Structured and unstructured pruning techniques.

2 Preprocessing: Audio Recording and Feature Extraction

2.1 Objective

Preprocessing input audio before feeding it into a neural network is an essential step to improve model performance and efficiency. Raw audio signals contain vast amounts of information, including background noise, amplitude variations, and redundant data that may not be directly relevant for classification or recognition tasks. To address these challenges, various preprocessing techniques are applied to extract meaningful features and ensure robust.

2.2 Preprocessing Input Audio

- **Normalization:** It is important to normalize the audio data since the audio samples input to a neural network can have varying amplitudes. If left unnormalized, large amplitude variations can lead to instability in model training and inaccurate feature extraction. Normalization ensures that all audio samples have a uniform scale, reducing discrepancies between different recordings. This helps in making the model more resilient to variations in speaker loudness, microphone sensitivity, and recording conditions.
- **Feature Extraction:** Preprocessing audio data helps us extract meaningful features from the audio waveform. Instead of feeding raw waveforms into a neural network, which can be highly variable and difficult to interpret, feature extraction techniques such as Mel Spectrograms and Mel Frequency Cepstral Coefficients (MFCCs) allow the model to focus on essential frequency and temporal characteristics. These representations are more compact and informative, enhancing the ability of neural networks to recognize patterns in speech and other audio signals efficiently. Additionally, feature extraction reduces computational complexity by transforming high-dimensional raw data into a lower-dimensional space with more relevant attributes.

2.3 Spectrogram Generation and Feature Extraction

Audio processing is performed using the `torchaudio` and `librosa` libraries. The key transformations applied include:

- **Spectrogram Computation:** Using `torchaudio.transforms.Spectrogram`, the raw waveform is converted into a frequency-time representation.
- **Mel Spectrogram Transformation:** The `torchaudio.transforms.MelSpectrogram` is applied to obtain a perceptually relevant feature set.
- **Log Power Conversion:** The spectrogram values are transformed into a logarithmic scale for better visualization and model training stability.

2.4 Difference Between Spectrograms

Differences between Mel Spectrograms, MFCCs, and Spectrograms:

- **Frequency Representation:**
  - Spectrograms use a **linear frequency scale**, directly representing frequency components as they appear in the signal.
  - Mel Spectrograms and MFCCs use the **Mel scale**, which is perceptually motivated and designed to match human hearing sensitivity, emphasizing lower frequencies.
- **Feature Representation and Processing:**
  - Spectrograms capture the full range of frequency content in a raw format, making them suitable for general audio analysis, music information retrieval, and speech synthesis.
  - Mel Spectrograms apply a Mel filter bank to group and smooth frequency components, making them robust for tasks like speech recognition and speaker identification.
  - MFCCs take Mel Spectrograms further by applying a Discrete Cosine Transform (DCT) to decorrelate frequency features and compress information, producing a compact feature set useful for machine learning models.
- **Computational Complexity:**
  - Spectrograms involve a direct Short-Time Fourier Transform (STFT), requiring high computational power due to the large matrix representation.
  - Mel Spectrograms reduce the computational burden by applying Mel-scaled filter banks, which smooth frequency content and reduce dimensionality.
  - MFCCs further reduce complexity by representing audio with a small set of coefficients (typically 12-40), making them efficient for real-time applications.
- **Usage in Audio Processing:**
  - MFCCs and Mel Spectrograms are commonly used in tasks requiring a high level of abstraction, such as speech recognition.
  - Spectrograms are often used in general audio processing tasks, including sound event detection and music analysis.
  - Both MFCCs and Mel Spectrograms are computationally less expensive than Spectrograms.

Differences between MFCCs and Mel Spectrograms:

- **Dimensionality and Representation:**
  - MFCCs are a **vector of coefficients** representing a compressed power spectrum, designed for efficient machine learning applications.
  - Mel Spectrograms are a **2D time-frequency representation** containing magnitude values, retaining more detailed spectral information.
- **Computational Efficiency:**
  - MFCCs are computationally less expensive than Mel Spectrograms, as they use a reduced feature set.
  - Mel Spectrograms consist of a 2D matrix containing the magnitude of frequency content, requiring more processing power than MFCCs.
- **Interpretability and Use Cases:**
  - Mel Spectrograms retain more information, making them useful for deep learning models that extract meaningful features automatically.
  - MFCCs provide a compact feature vector, which is advantageous for traditional machine learning models and low-power applications.

2.5 Spectrogram Visualizations

The following figures illustrate different types of spectrogram representations used in the processing pipeline:

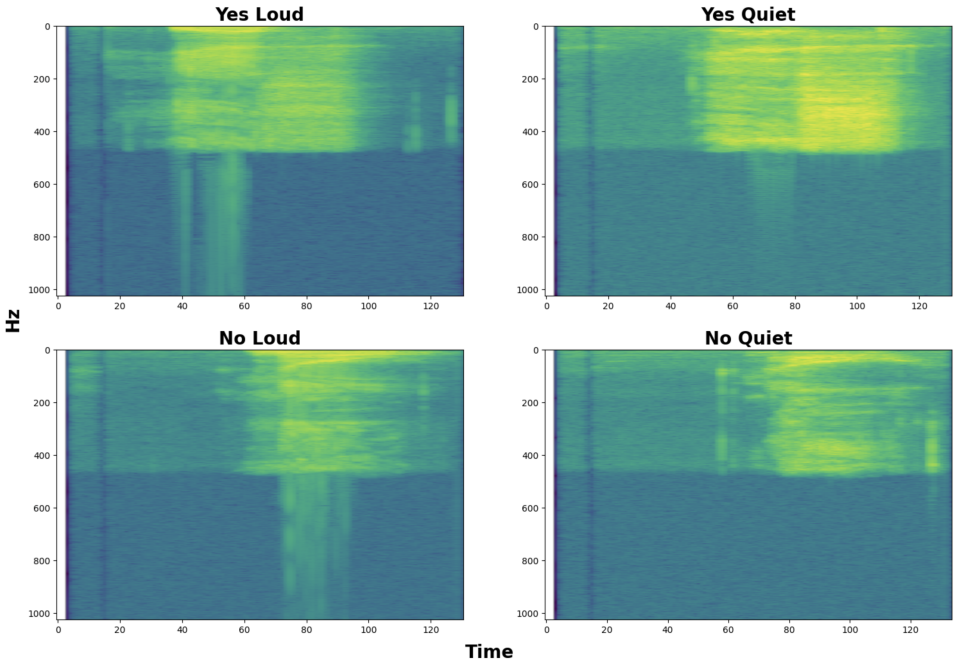

Figure 1: Spectrogram visualizations for different audio inputs.

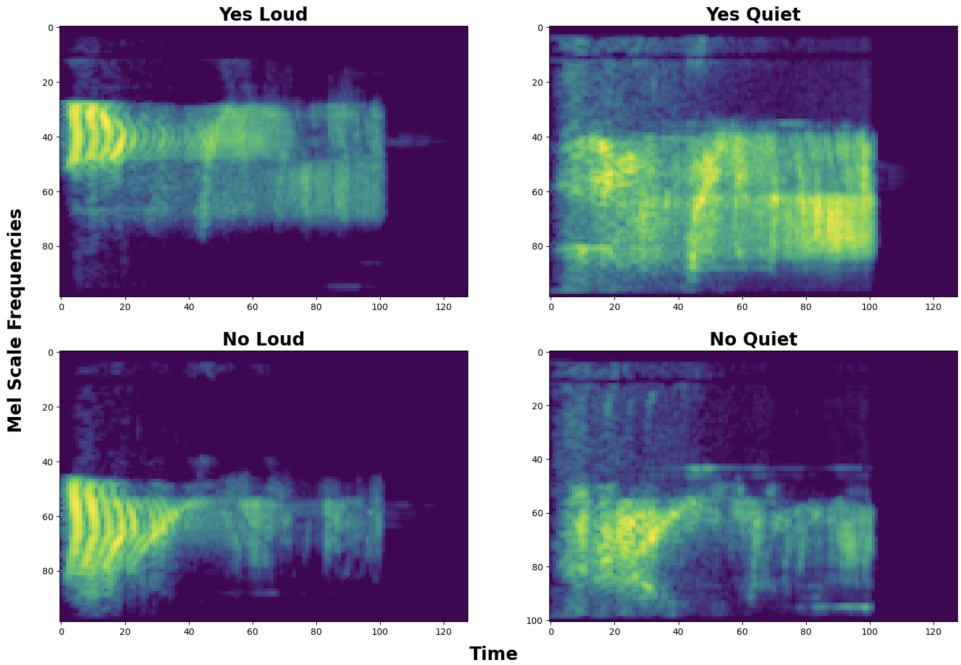

Figure 2: Mel Spectrogram representation.

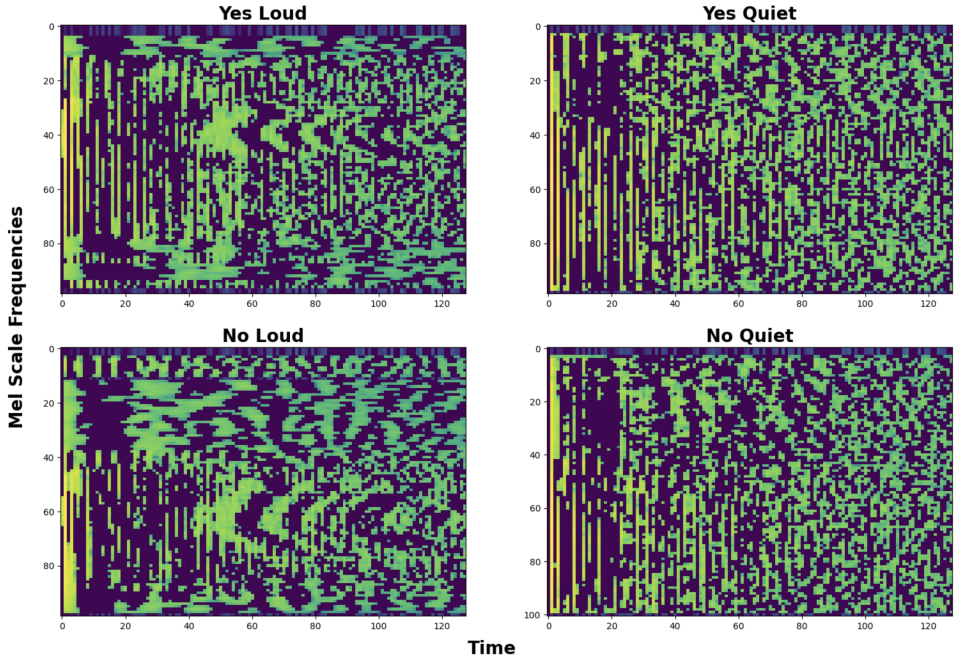

Figure 3: MFCC Spectrogram representation.

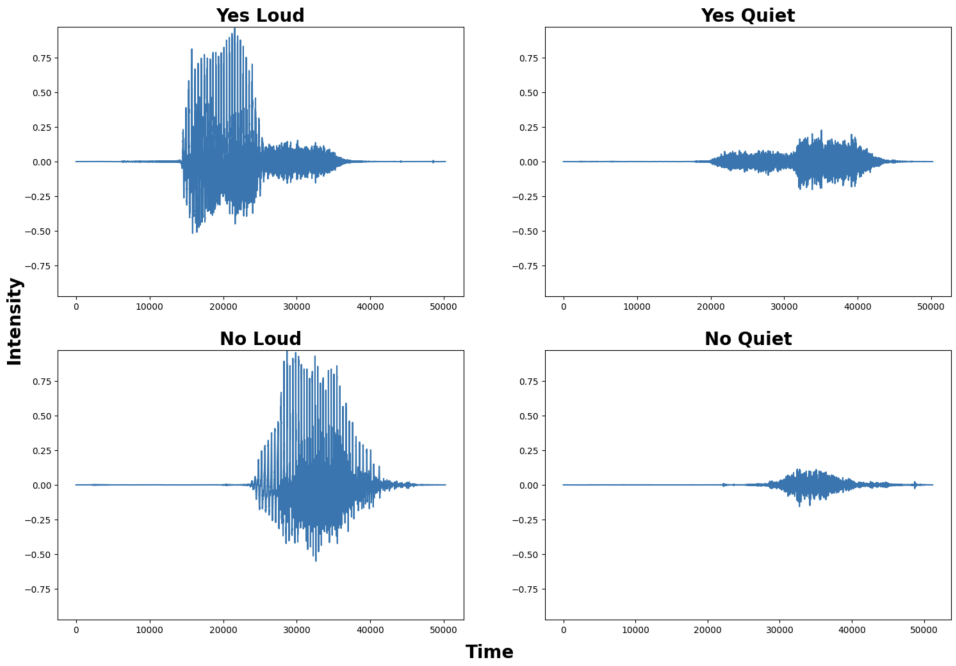

Figure 4: Time-domain waveform representations for different audio inputs.

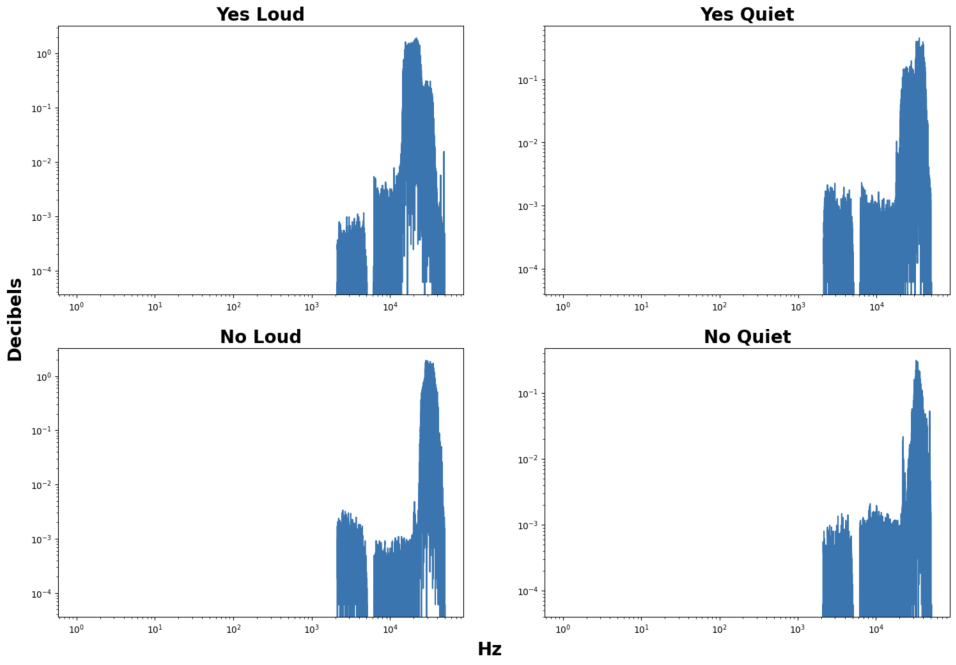

Figure 5: Frequency-domain waveform representations for different audio inputs.

3 Model Size Estimation

3.1 Memory Footprint

Estimating the memory footprint of a Deep Neural Network (DNN) is crucial for deployment feasibility, particularly in resource-constrained environments such as microcontrollers (MCUs). Below are the key considerations:

- **Flash and RAM estimation for deployment feasibility.** Ensuring the model fits within the available memory constraints while maintaining efficiency.
- **Percentage utilization of MCU resources.** This helps understand how much of the total available memory is consumed by the model.

3.1.1 Estimated Flash Usage of DNN

To estimate the flash memory usage of the TinyConv model on the MCU, we consider the storage required for both the model's trainable parameters and any additional code necessary for execution. The total number of trainable parameters in the model is given as:

$$0.016652M = 16,652 \text{ parameters} \tag{1}$$

Since the model uses **32-bit floating point (FP32)** representation, each parameter requires 4 bytes of storage. For the given number of parameters:

$$S_{\text{flash}} = 0.016652 \times 10^6 \times 4 \text{ bytes/parameter} = 0.016652 \times 4 \text{ MB} = 0.066608 \text{ MB} \tag{2}$$

To convert megabytes to kilobytes:

$$S_{\text{flash}} = 0.066608 \text{ MB} \times 1024 \text{ KB/MB} = 68.2 \text{ KB} \tag{3}$$

Thus, the estimated flash usage for the model Given the MCU's total flash memory of **1MB** (1024 KB), we compute the fraction of flash memory occupied by the model as:

$$\frac{\text{Model Size in KB}}{\text{Total Flash Memory in KB}} \times 100 \tag{4}$$

$$\frac{68.2 \text{ KB}}{1024 \text{ KB}} \times 100 \approx 6.6\% \tag{5}$$

This estimation considers only the model weights. Additional flash memory usage may be required for storing metadata, framework dependencies (e.g., TFLite Micro runtime), and any lookup tables used during inference. Despite this, the model remains compact and efficient, making it well-suited for deployment on resource-constrained devices.

3.1.2 RAM Usage of DNN with Batch Size = 1

The forward pass RAM usage of the TinyConv model was determined by computing the storage space required for the input and output tensors of the model. Specifically, we used a helper function to measure memory allocation during inference, summing the sizes of activations, intermediate feature maps, and temporary buffers needed for computation. The forward RAM of the TinyConv model was computed as:

$$\text{Forward RAM} = 0.007856 \text{ MB} \tag{6}$$

This memory usage accounts for:

$$\text{Percentage of Flash Memory Used} = 0.78\% \tag{7}$$

Thus, the TinyConv model utilizes a minimal portion of the MCU's RAM memory, ensuring efficient deployment within the device's constraints.

3.2 Computational Complexity

Understanding the computational complexity of a model is important for estimating processing latency, energy consumption, and feasibility for real-time applications.

- **FLOPS estimation and benchmarking.** The number of floating-point operations per second (FLOPS) provides an estimate of the computational requirements of the model.
- **Comparison with existing keyword spotting models.** Evaluating the model against state-of-the-art architectures helps assess its efficiency.

3.2.1 FLOPS of the DNN

The TinyConv model achieves a remarkable reduction in computational complexity while maintaining effective performance. It requires a total of 676,004 floating-point operations (FLOPS), where the Conv2D layer accounts for 644,000 FLOPS, and the Linear layer contributes 32,004 FLOPS. This results in a total computational burden of 0.676M FLOPS, which is a dramatic reduction compared to the 30M FLOPS required by Convolutional Recurrent Neural Networks (CRNNs) as referenced in [1]. The significantly lower FLOPS count suggests a substantial decrease in both power consumption and processing latency, making it highly suitable for deployment on low-power edge devices. Additionally, this reduction in computational complexity allows TinyConv to run on microcontrollers or embedded systems with limited memory and processing power, whereas CRNNs typically require more robust hardware due to their recurrent connections and deeper architectures. While models like DeepSpeech2 are designed for high accuracy and complexity, leveraging substantial computational resources during both training and inference, TinyConv prioritizes efficiency. Its lightweight design makes it well-suited for real-time applications in environments with strict resource constraints, such as embedded systems and edge devices. Unlike DeepSpeech2, which benefits from large-scale GPUs or cloud-based computation, TinyConv can operate effectively on low-power hardware without sacrificing usability in keyword spotting tasks. The efficiency of TinyConv is further emphasized when considering memory footprint. Given its lower number of parameters and reduced floating-point operations, TinyConv not only requires less storage but also leads to faster inference times, reducing real-time processing constraints.

3.2.2 Inference Time of the Model on Different Hardware Platforms

To further assess real-time feasibility, the inference time of the TinyConv model was measured on different hardware platforms:

- On a **Colab CPU**, the inference time was 1.314ms. This reflects the latency experienced on general-purpose processors without hardware acceleration.
- On a **Colab GPU**, the inference time was significantly reduced to 41.6µs, demonstrating the efficiency of GPU acceleration for deep learning inference.

These results highlight the efficiency of TinyConv for real-time applications while maintaining a minimal computational footprint. The inference times indicate that while CPU inference is feasible for some applications, leveraging GPU acceleration can drastically improve performance for more demanding real-time use cases.

4 Training and Analysis

4.1 Training Procedure

The training procedure of the TinyConv model was designed to optimize its performance for keyword spotting while ensuring generalization across different datasets. The following aspects were considered:

- **Dataset Description and Class Distribution:** The dataset used for training consists of 4 keyword classes: *yes*, *no*, *unknown*, and *silence*. The dataset is structured as follows:
  - **Training Set:** 10,556 samples
  - **Validation Set:** 1,333 samples
  - **Test Set:** 1,368 samples

  The class distribution ensures a balanced representation of different keyword categories, preventing bias towards a specific class.
- **Model Architecture and Hyperparameter Tuning:** The TinyConv model is a lightweight convolutional neural network (CNN) optimized for small-footprint keyword spotting. The architecture consists of multiple convolutional layers followed by fully connected layers to extract and classify audio features efficiently. The hyperparameters were tuned to achieve optimal performance:
  - Learning Rate: 0.001
  - Number of Epochs: 50
  - Loss Function: Cross-Entropy Loss
  - Regularization: weight decay of $1e^{-4}$

  Hyperparameter tuning involved experimenting with different learning rates and batch sizes to find the best trade-off between convergence speed and model performance.
- **Training and Validation Accuracy Curves:** The model was trained for 50 epochs, during which training and validation accuracy were recorded. **Figure 6** presents the accuracy curves, showcasing the learning progression of the model. The curves indicate that the model effectively learned the patterns in the dataset while avoiding overfitting.

4.2 Accuracy of the Model on the Training, Validation, and Test Sets

The TinyConv model achieved high accuracy across all datasets, demonstrating its ability to generalize well:

- **Training Accuracy:** 90.5%
- **Validation Accuracy:** 90.8%
- **Test Accuracy:** 91.2%

These results highlight the consistency of the model's performance across different datasets, ensuring robustness in recognizing keywords in various conditions.

4.3 Training and Validation Accuracy Trends

Figure 6 provides the accuracy curves for both training and validation phases. The trends indicate:

- A steady increase in accuracy over epochs, signifying proper learning without abrupt fluctuations.
- Minimal overfitting, as indicated by the close alignment between training and validation accuracy curves.
- A plateau towards the final epochs, suggesting model convergence and optimal performance.

These results confirm that the TinyConv model is well-optimized for the keyword spotting task, striking a balance between accuracy and computational efficiency.

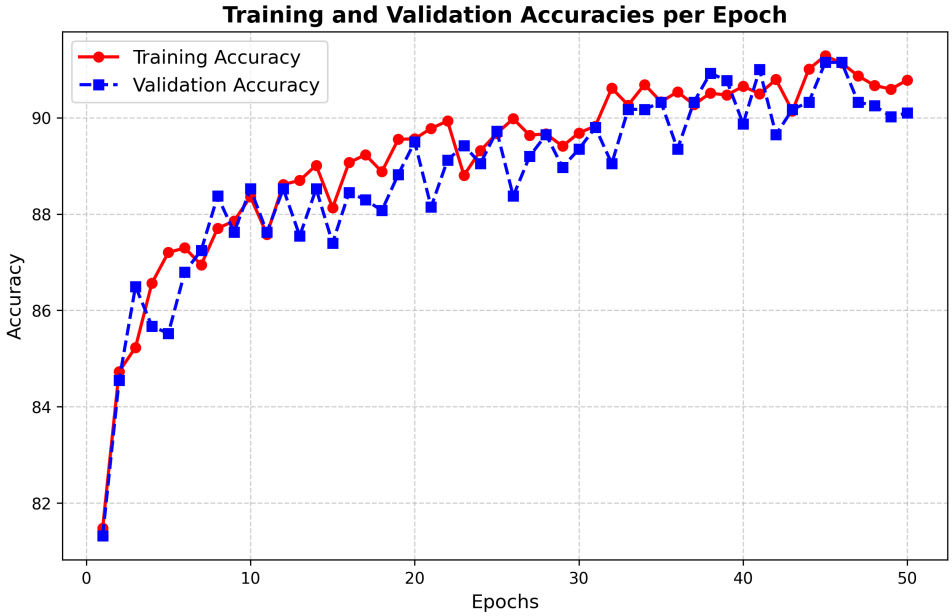

Figure 6: Training and Validation Accuracy Curve

5 Model Conversion and Deployment

5.1 Transition to TFLite

The model was converted to TensorFlow Lite (TFLite) Micro for deployment on an MCU. This process involved the following steps:

1. Model quantization: Converted the model from 32-bit floating-point to 8-bit integer to reduce memory footprint.
2. TFLite conversion: Used TensorFlow's `TFLiteConverter` to transform the model into an MCU-compatible format.
3. Testing on simulated environments: Verified model performance before deployment using the TFLite Interpreter.
4. Deployment on the Arduino MCU: Finalized testing and profiling on actual hardware.

5.2 Deployment on Arduino

The model was deployed on an Arduino-compatible MCU, and inference profiling was performed to measure different execution stages:

5.2.1 MCU Inference Profiling

To ensure accurate measurements, each timing value was recorded over 300 iterations, and the minimum, maximum, and average times were computed. The timing breakdown for different stages of the model execution on the MCU is as follows:

Stage	Min Time (ms)	Max Time (ms)	Avg Time (ms)
Preprocessing	20	23	21.76
Inference	86.5	88	87.88
Post-processing	0	1	0.06

Table 1: Execution Time Breakdown (Min, Max, and Avg)

5.2.2 Comparison with CPU and GPU Inference Times

The MCU's average inference time was measured at 87.8 ms, while the Colab CPU and GPU achieved inference times of 1.314 ms and 41.6 μs, respectively. For direct comparison, the GPU time is converted to milliseconds, resulting in 0.0416 ms. The following profiling results were observed:

- **MCU Model Execution Time:** 88 ms
- **CPU Model Execution Time:** 1.314 ms
- **GPU Model Execution Time:** 0.0416 ms

Comparison of Execution Speed: To quantify how much slower the MCU is compared to the CPU and GPU:

$$\text{Slowdown Factor (MCU vs. CPU)} = \frac{88}{1.314} \approx 67 \tag{8}$$

$$\text{Slowdown Factor (MCU vs. GPU)} = \frac{88}{0.0416} \approx 2115 \tag{9}$$

Thus, the MCU is approximately **67 times slower** than the CPU and about **2115 times slower** than the GPU for inference. This highlights the efficiency trade-offs when deploying deep learning models on low-power embedded systems.

- **Computational Resources:** The MCU has limited computational power and lacks parallel processing capabilities, whereas modern CPUs and GPUs feature advanced architectures with multiple cores and hardware accelerators optimized for matrix operations.

- **Memory Bandwidth and Cache Efficiency:** CPUs and GPUs benefit from high-speed memory hierarchies, including L1/L2/L3 caches, which significantly reduce memory access latency. In contrast, MCUs often rely on slower external memory, increasing data access times.
- **Parallelism:** GPUs are designed for highly parallel workloads, enabling thousands of concurrent computations. CPUs, though not as parallel as GPUs, still leverage multi-threading and SIMD instructions to accelerate inference. MCUs, however, execute instructions sequentially or with minimal parallelization, limiting their ability to process deep learning models efficiently.
- **Instruction Set and Optimization:** CPUs and GPUs are optimized for deep learning inference through specialized instruction sets such as AVX (Advanced Vector Extensions) and Tensor Cores. MCUs, on the other hand, typically lack such specialized hardware, requiring models to be quantized and optimized for integer-based operations, which can still result in significant performance overhead.
- **Power and Thermal Constraints:** MCUs are designed for low-power applications and prioritize energy efficiency over raw computational performance. This constraint affects their ability to execute deep learning models at high speeds compared to power-hungry CPUs and GPUs.

These factors collectively explain the large gap in inference times between MCUs, CPUs, and GPUs. While MCUs provide a viable solution for embedded deep learning applications, they require careful model optimization and quantization to mitigate their performance limitations.

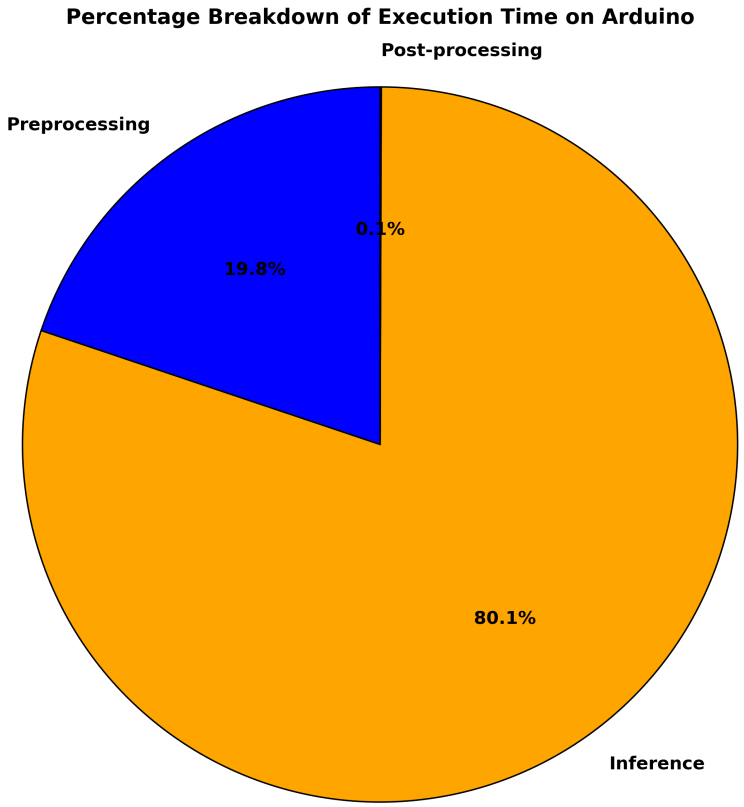

Figure 7: Percentage Breakdown of Execution Time on MCU: Preprocessing, Inference, and Post-processing

5.3 Recorded Test Set and Accuracy

A real-time test was conducted using an MCU to assess model performance. Below are the test results:

- Ground truth labels: 'yes', 'no', 's', 'y', 'n', 'p', 'no', 'no', 'yes', 'no', 'p', 'no'.
- MCU predictions: 'yes', 'no', 'yes', #unknown, 'no-detection', #unknown, 'no', 'yes', 'no', 'no-detection','no' , 'no'.e

Out of the 12 predictions made by the MCU, 9 were correct, resulting in an accuracy of 75% for this test. This demonstrates a reasonably high recognition performance, though some misclassifications and non-detections indicate areas for further refinement. After analyzing the breakdown, the model was evaluated on various combinations of "yes" and "no" inputs to assess its classification performance. In total, 20 different samples were tested, as shown in Table 2.

Keyword	**Trial Count**	**Correct Labels Percentage**
yes	10	80%
no	10	85%

Table 2: Model performance breakdown on different keywords

5.3.1 Observations

The discrepancy between real-world accuracy and the training/validation accuracy can be attributed to multiple factors, including hardware limitations, data representation challenges, and quantization effects:

- **Noisy MCU sensor inputs**: In real-world deployment, the microcontroller's microphone may introduce noise, distortion, or artifacts due to environmental factors such as background noise, varying distances from the speaker, or differences in pronunciation. Words like *no* might be particularly susceptible to misclassification if their acoustic features are not consistently captured across different conditions, leading to detection failures.
- **Spectrogram similarity**: Some words, such as *s* and *yes*, may generate spectrograms with highly overlapping features, making it difficult for the model to distinguish between them. This issue arises because the model primarily relies on frequency patterns, and similar phonetic structures can result in nearly identical time-frequency representations, increasing the likelihood of confusion and misclassification.
- **Quantization artifacts**: Transitioning from a high-precision floating-point model to an 8-bit quantized TensorFlow Lite Micro version can introduce small numerical inaccuracies. While quantization reduces memory and computational requirements, it also leads to a loss of precision in weights and activations, potentially degrading the model's ability to differentiate between subtly different speech patterns. This effect may be particularly noticeable in edge cases where classification decisions rely on fine-grained details.

Further improvements could involve better preprocessing, enhanced noise filtering, or retraining the model with augmented data that better represents in-field conditions. Additionally, optimizing the model's architecture for robustness against environmental variations can help mitigate performance drops in real-world scenarios.

6 Quantization-Aware Training

6.1 Impact of Quantization

Quantization is a fundamental technique in deep learning that aims to reduce computational complexity and memory requirements, particularly for deployment on resource-constrained edge devices. The two principal methods employed are post-training quantization (PTQ) and quantization-aware training (QAT). PTQ involves applying quantization to a pre-trained floating-point model without further optimization, whereas QAT incorporates quantization effects during training, enabling the model to adapt to reduced precision constraints. This section provides a comparative analysis of PTQ and QAT in terms of accuracy and efficiency.

6.2 Experiments and Results

To systematically evaluate the impact of quantization, experiments were conducted using bit-widths ranging from 2 to 8 bits. The following analyses were performed:

- Accuracy as a function of bit-width (2 to 8 bits) for both PTQ and QAT, illustrating the effect of reduced numerical precision on model performance.
- The impact of minifloat quantization, which balances precision and computational efficiency.

6.3 Accuracy vs. Bit-Width for Post-Training Quantization

Post-training quantization was applied to a trained floating-point model, and accuracy was assessed across different bit-widths. As shown in Figure 8, while accuracy remains relatively stable at higher bit-widths, it deteriorates significantly at 2-bit quantization due to the amplified impact of quantization noise. This degradation can be attributed to the aggressive reduction in numerical precision, which limits the model's ability to represent fine-grained features in the data. Consequently, the weight and activation quantization errors become more pronounced, leading to a loss of critical information necessary for accurate predictions. In contrast, higher bit-widths, such as 6-bit and 8-bit, preserve most of the original model's accuracy, indicating that a moderate level of quantization can be implemented without severe performance loss. These findings highlight the limitations of PTQ at extremely low bit-widths and emphasize the importance of selecting an optimal precision level to balance computational efficiency with model accuracy.

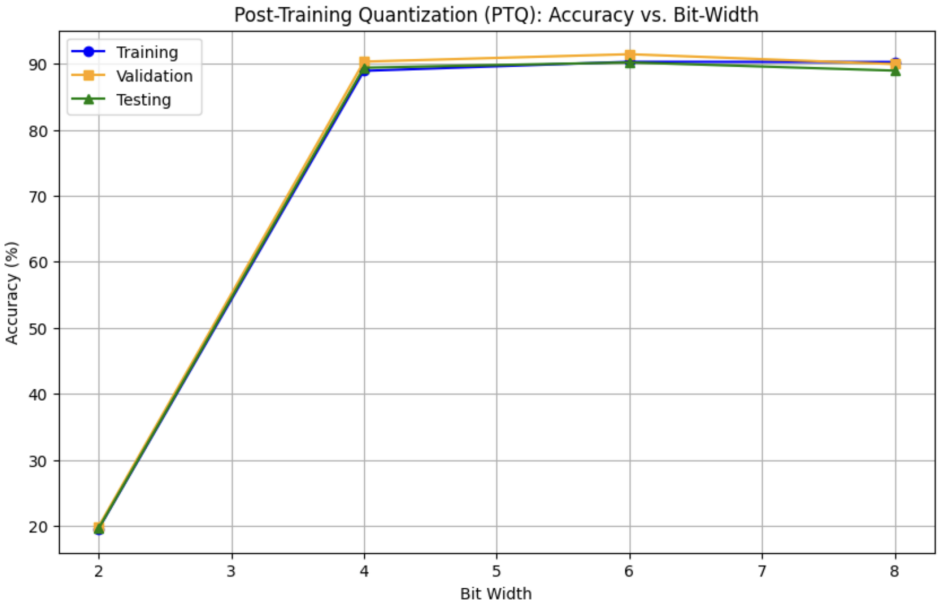

Figure 8: Post-Training Quantization (PTQ): Accuracy vs. Bit-Width. Lower bit-widths induce significant accuracy degradation due to quantization noise.

6.4 Accuracy vs. Bit-Width for Quantization-Aware Training

For QAT, the model was trained while simulating quantization effects, allowing it to learn to mitigate the adverse effects of reduced numerical precision. The accuracy results across different bit-widths are presented in Figure 9. In general, QAT demonstrates greater stability in performance across bit-widths when compared to PTQ, underscoring the advantages of integrating quantization during the training process.

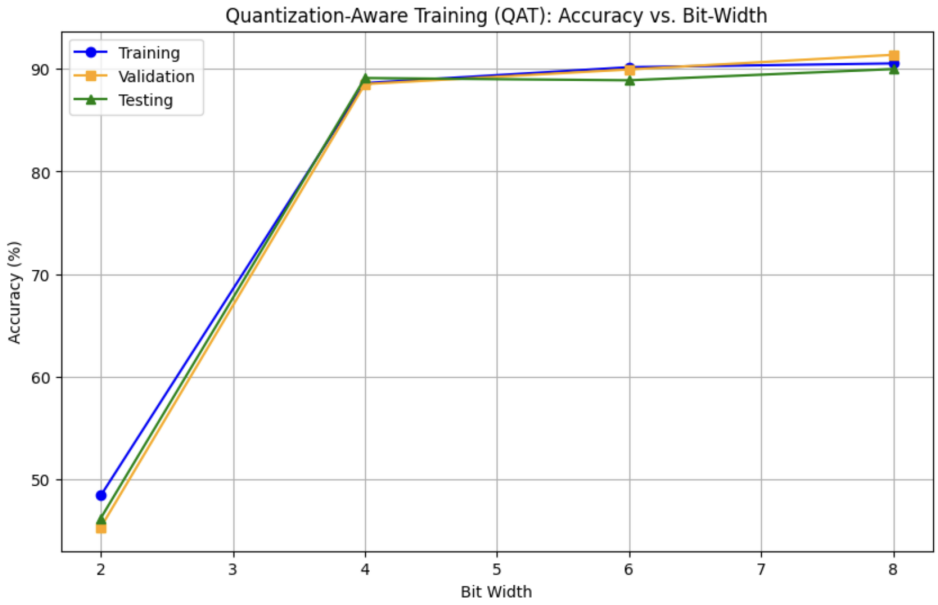

Figure 9: Quantization-Aware Training (QAT): Accuracy vs. Bit-Width. Incorporating quantization effects during training mitigates accuracy degradation.

6.5 Comparison of Quantization-Aware Training and Post-Training Quantization

A comparative analysis of Post-Training Quantization (PTQ) and Quantization-Aware Training (QAT) reveals distinct accuracy trends across different bit-widths, highlighting the trade-offs between efficiency and performance. At 8-bit quantization, QAT consistently achieves slightly higher accuracy than PTQ, underscoring the benefits of training with quantization constraints. However, at 4-bit and 6-bit quantization, both methods yield nearly identical accuracy, suggesting that the pre-trained floating-point model retains robust feature representations even after quantization. This indicates that for moderate bit-width reductions, PTQ can be an effective and computationally efficient alternative to QAT. At extremely low-bit quantization (e.g., 2-bit), both PTQ and QAT experience significant performance degradation, but QAT retains a slight advantage. This suggests that while QAT can mitigate some accuracy loss through learned adaptations to reduced precision, standard QAT techniques alone may be insufficient for ultra-low-bit quantization, which often requires specialized methods such as adaptive quantization or mixed-precision approaches [3]. The fundamental difference between PTQ and QAT lies in their adaptation mechanisms. QAT integrates quantization effects directly into the training process, allowing the model to adjust to precision loss and compensate for quantization-induced errors. This results in more stable performance across varying bit-widths, particularly at extreme quantization levels. In contrast, PTQ applies quantization as a post-processing step without modifying the original model weights, making it computationally cheaper and easier to deploy. However, because PTQ does not incorporate quantization constraints during training, it is more susceptible to accuracy degradation at lower bit-widths, where quantization noise becomes overwhelmingly destructive [2]. From a practical standpoint, PTQ is well-suited for scenarios where retraining is infeasible or computationally expensive, offering strong performance at bit-widths of 4 and above while maintaining a simple and efficient deployment process [5]. However, its sharp accuracy drop at extreme quantization levels limits its applicability in ultra-low precision settings. In contrast, QAT provides greater robustness across all bit-widths, making it a more viable option for applications with stringent computational and memory constraints that

necessitate aggressive quantization [3]. Ultimately, the choice between PTQ and QAT depends on the target application: PTQ is preferable when fast deployment with minimal computational overhead is required, whereas QAT is advantageous in performance-critical settings where maintaining accuracy under aggressive quantization is essential.
In summary, while QAT generally better prepares the model for reduced precision by allowing it to learn and adapt to quantization constraints, the marginally higher accuracy achieved by PTQ at certain bit-widths suggests that the models used in this study are inherently robust to quantization. This indicates that, in some cases, PTQ can effectively maintain performance despite being applied post-training, reducing the need for extensive retraining while still delivering competitive accuracy.

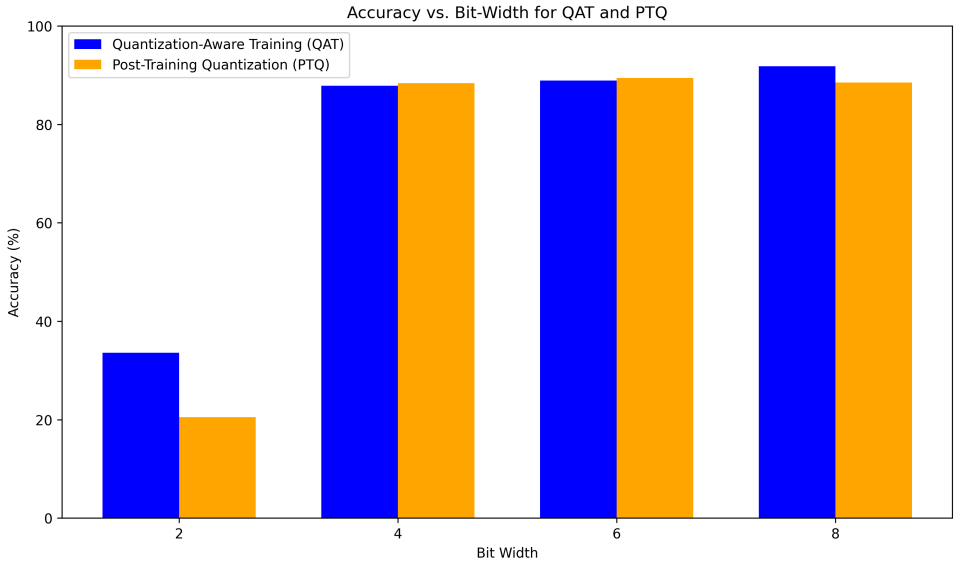

Figure 10: Comparison of accuracy across different bit-widths for Quantization-Aware Training (QAT) and Post-Training Quantization (PTQ).

6.6 Extra Credit: Minifloat Quantization & QAT vs PTQ Analysis

6.6.1 Explanation of Minifloat Choices

Minifloat formats are defined by a 1-bit sign, a varying exponent (1-4 bits), and a mantissa (1-3 bits), leading to total bit-widths of 3, 4, 6, and 8. These formats explore trade-offs between dynamic range (exponent) and precision (mantissa):

- **3-bit (exp=1, mant=1):**
  - 2 exponent levels, 2 mantissa values.
  - Minimal precision but improved over 2-bit formats.
  - Suitable for extreme quantization in ultra-low-power applications.
- **4-bit (exp=2, mant=1):**
  - 4 exponent levels, 2 mantissa values.
  - Balances dynamic range and detail while maintaining efficiency.
  - A practical trade-off for tiny ML models with limited resources.
- **6-bit (exp=3, mant=2):**
  - 8 exponent levels, 4 mantissa values.
  - Moderate precision, improving numerical representation.
  - More accurate while remaining lightweight for embedded AI applications.

- **8-bit (exp=4, mant=3):**
  - 16 exponent levels, 8 mantissa values.
  - Maximizes range and precision within 8-bit constraints.
  - Well-suited for models requiring higher accuracy with quantization benefits.

6.6.2 QAT vs. PTQ: Accuracy vs. Bit-Width

The following figure illustrates the accuracy of Post-Training Quantization (PTQ) and Quantization-Aware Training (QAT) across different minifloat bit-widths.

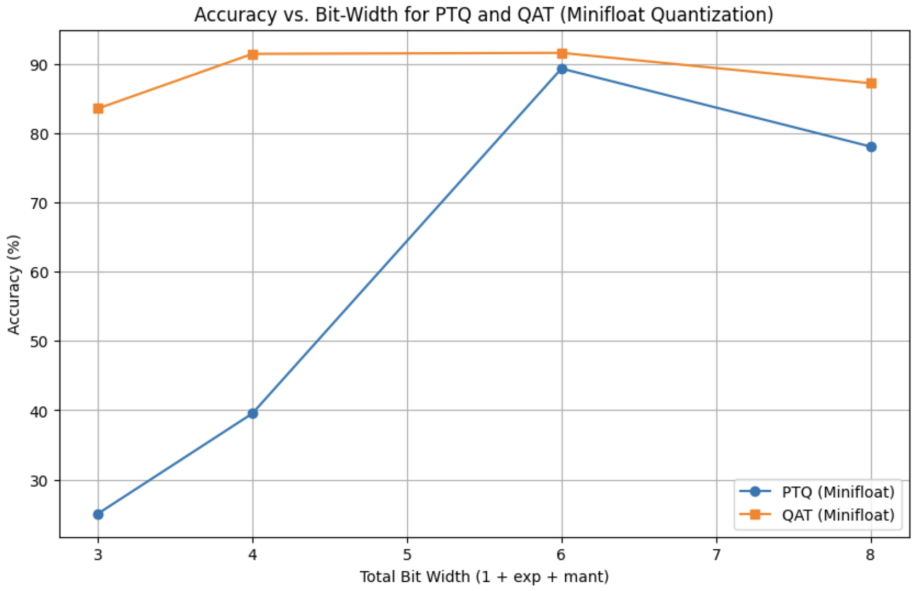

Figure 11: Accuracy vs. Bit-Width for PTQ and QAT (Minifloat Quantization)

The results show that:

- QAT consistently outperforms PTQ, especially at lower bit-widths.
- PTQ suffers from significant accuracy drops at 3-bit and 4-bit settings.
- QAT maintains high accuracy even with aggressive quantization.
- The optimal trade-off appears at 6-bit quantization, where PTQ and QAT accuracy levels converge.

These findings reinforce the benefits of QAT in maintaining model performance under aggressive quantization constraints.

7 Pruning for Model Optimization

7.1 Unstructured Pruning

Unstructured pruning aims to reduce the number of parameters in a model by selectively removing weights based on a predefined criterion, such as their L1 norm magnitude. This process decreases the computational complexity of the model while preserving accuracy within an acceptable range.

7.1.1 Speeding Up Computation with Unstructured Pruning

Unstructured pruning eliminates specific weights and connections within a model, leading to a reduction in the number of floating-point operations (FLOPs) required for both training and inference. The benefits include:

- **Faster computation due to a reduced number of active parameters:** Removing redundant parameters decreases the number of required operations, leading to lower inference latency. On optimized hardware and sparse-aware libraries, this can significantly improve execution speed by skipping zero-weight calculations.
- **Lower memory requirements for model storage:** Pruned models have a smaller memory footprint, making them ideal for deployment on devices with limited storage, such as embedded systems and mobile platforms.
- **Reduced power consumption, making it suitable for deployment on edge devices:** By decreasing the number of computations, pruned models require less power during inference. This is particularly beneficial for battery-powered devices, where energy efficiency directly impacts device longevity and operational stability.

Pruning was implemented using `torch.nn.utils.prune.l1_unstructured`, which removes connections based on the L1 norm of the weight values. Following the comparison in Section 7.1.3, we decided to implement L1 unstructured pruning. While unstructured pruning reduces the number of model parameters and provides some speedup, higher speed improvements can be achieved by optimizing the computation pipeline, leveraging sparse matrix representations, and utilizing hardware accelerators designed for sparse operations. The following strategies can further enhance efficiency:

- **Sparse Matrix Operations**: Utilizing libraries optimized for sparse computations (e.g., cuSPARSE on NVIDIA GPUs or sparse BLAS on CPUs) enables skipping multiplications involving zero weights, reducing computation time in proportion to the model's sparsity level [7].
- **Specialized Hardware**: Certain hardware accelerators, such as TPUs, FPGAs, or MCUs optimized for TensorFlow Lite Micro, can natively exploit sparsity by executing only non-zero weight operations, leading to substantial runtime reductions [8, 9].

7.1.2 Impact of Fine-Tuning on Accuracy

The effectiveness of pruning can be further enhanced through fine-tuning, which allows the model to recover some of the accuracy lost during pruning. As shown in Figure 12, fine-tuning consistently improves accuracy compared to models without fine-tuning. This underscores its importance in the pruning process, ensuring that the model retains predictive performance despite having fewer parameters. An interesting observation is that the accuracy of the fine-tuned model eventually plateaus, indicating a point of diminishing returns. Beyond this threshold, increasing the number of parameters does not yield significant improvements in accuracy, suggesting that fine-tuning primarily benefits models up to a certain level of sparsity. This behavior highlights the need for careful selection of pruning thresholds to balance accuracy and computational efficiency effectively. The plot shows the relationship between model accuracy and the number of retained parameters across different pruning thresholds. As expected, accuracy generally improves with a higher number of retained parameters. However, a notable "cliff" is observed, particularly at pruning threshold 0.5, where accuracy sharply declines. This suggests that beyond a critical threshold, excessive parameter removal significantly impairs the model's predictive capability. In summary, while unstructured pruning effectively reduces the number of active weights, fine-tuning ensures that the remaining parameters are optimized for accuracy. Combined with sparse computation techniques or specialized hardware, these methods enable efficient deployment of deep learning models

in resource-constrained environments. The findings emphasize the importance of selecting appropriate pruning thresholds to maintain a balance between model sparsity and performance.

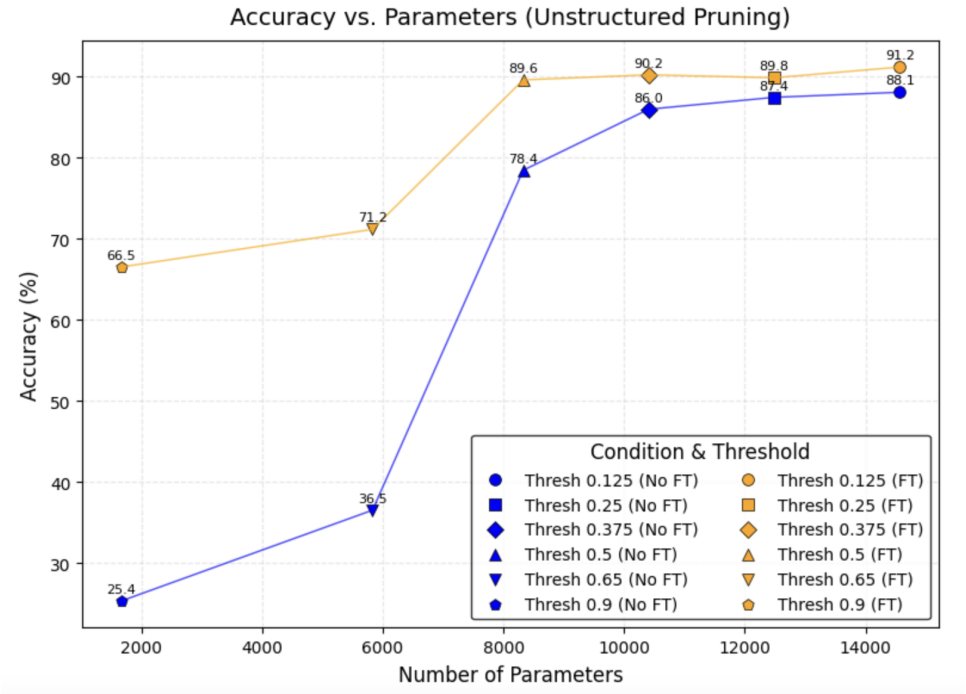

Figure 12: Accuracy vs Parameters for Unstructured Pruned Models (With and Without Finetuning)

7.1.3 Comparison of L1, L2, and L-Infinity Norms for Unstructured Pruning

Different norms can be used as criteria for selecting which weights to prune:

- **L1 Norm**: Defined as the sum of the absolute values of the weights:

$$\|x\|_1 = \sum_{i=1}^{n} |x_i| \tag{10}$$

  The L1 norm encourages sparsity by shrinking some weight values to zero. This is highly beneficial for pruning, as it effectively reduces model size while preserving essential connections. By prioritizing weights with smaller magnitudes for removal, the L1 norm allows for a more efficient and compact network without significantly affecting overall model performance. Unlike the L2 norm, which distributes shrinkage more evenly across all weights, the L1 norm selectively eliminates weaker connections, leading to a more interpretable and structured reduction in network complexity.

- **L2 Norm**: The square root of the sum of squared magnitudes:

$$\|x\|_2 = \sqrt{\sum_{i=1}^{n} x_i^2} \tag{11}$$

  Unlike the L1 norm, the L2 norm penalizes all weights more uniformly. It does not directly encourage sparsity, as smaller weights are not forced to shrink to zero. Instead, it results in weight decay, where all weights are reduced in magnitude proportionally. This makes the L2 norm less effective for pruning, as it does not selectively eliminate insignificant weights. Consequently, models trained with L2 regularization tend to retain a denser distribution of weights, requiring more aggressive thresholding during pruning to achieve meaningful sparsity. As a result, L2-based pruning can lead to a sharper drop in accuracy when higher pruning thresholds are applied, as important parameters are more likely to be affected.

- **L-Infinity Norm**: The L-Infinity norm, also known as the max norm, is defined as the maximum absolute value among all elements in a given vector:

$$|x|\infty = \max 1 \leq i \leq n|x_i| \tag{12}$$

This norm essentially captures the largest magnitude weight in the vector while completely disregarding the contributions of all other elements. Unlike other norms such as the L1 norm (which sums absolute values) or the L2 norm (which considers all values quadratically), the L-Infinity norm does not provide a measure of overall weight distribution. In the context of neural network pruning, where the goal is to remove less significant weights to reduce model complexity, the L-Infinity norm is not particularly useful. Since it focuses solely on the maximum value, it does not provide a structured way to identify and remove small-magnitude weights across the network. Instead, pruning typically relies on norms that consider the entire set of weights, such as the L1 norm which promotes sparsity or the L2 norm which considers the overall magnitude distribution.

Among these norms, **L1 norm is the most effective for pruning**. Its ability to drive some weights to exactly zero makes it particularly well-suited for reducing the number of parameters in a neural network. By selectively pruning less significant connections, it helps maintain model accuracy while achieving significant compression. The L2 norm, while useful for regularization, does not promote sparsity as effectively, and the L-Infinity norm is unsuitable for pruning since it focuses solely on the largest weights rather than a global reduction strategy.

7.1.4 Empirical Comparison of L1, L2, and L-infinity Norm Unstructured Pruning

Analyzing pruning methods is vital for optimizing neural network efficiency. This section evaluates unstructured pruning using L1, L2, and L∞ norms at thresholds from 0.125 to 0.9, with accuracy trends shown in Figure 13. L1 pruning removes weights with the smallest absolute values, enhancing sparsity. L2 pruning leverages local Euclidean norms across weight neighborhoods, while L∞ pruning uses maximum absolute values, each yielding unique sparsity-accuracy trade-offs. To improve this, we add L1 regularization to the training loss, promoting zero weights naturally, and retrain from scratch. This preconditions the model for pruning, which we then apply using the same L1, L2, and L∞ methods. Evaluating post-pruning accuracy reveals how regularization during training enhances pruning effectiveness.

Accuracy Trends: L1 pruning consistently outperforms L2 and L-infinity norm pruning across various pruning thresholds:

- At a low pruning threshold (0.125), all methods achieve similar accuracy: L1 at **92.03%**, L2 at **91.81%**, and L-infinity norm at **92.76%**.
- As the pruning threshold increases, accuracy degradation varies significantly. At 0.375, L1 maintains **89.69%** accuracy, while L2 drops to **89.40%**, and L-infinity norm slightly lower at **88.74%**.
- At a higher pruning threshold of 0.625, L1 pruning retains **81.07%** accuracy, whereas L2 declines more sharply to **58.63%**, and L-infinity norm drops further to **49.20%**.
- At the highest pruning threshold of 0.9, accuracy declines significantly across all methods: L1 pruning retains **25.58%**, while L2 drops to **27.78%**, and L-infinity norm to **26.17%**.

Key Observations:

1. **L1 pruning is the most resilient to higher pruning thresholds**, maintaining relatively high accuracy compared to L2 and L-infinity norm pruning, particularly in mid-range sparsity levels.
2. **L2 and L-infinity norm pruning suffer from more rapid accuracy degradation**, especially beyond a pruning threshold of 0.375, where their performance drops significantly.
3. **L-infinity norm pruning performs similarly to L2 but tends to degrade slightly faster at high pruning thresholds**, making it less effective in maintaining accuracy.
4. **L1 pruning exhibits a more gradual decline in accuracy**, suggesting it is better suited for balancing compression and model performance.

These findings suggest that **L1 pruning is the most effective unstructured pruning method**, as it enables substantial compression while minimizing accuracy loss. In contrast, L2 and L-infinity norm pruning result in more pronounced accuracy degradation, making them less suitable for maintaining model performance at higher sparsity levels.

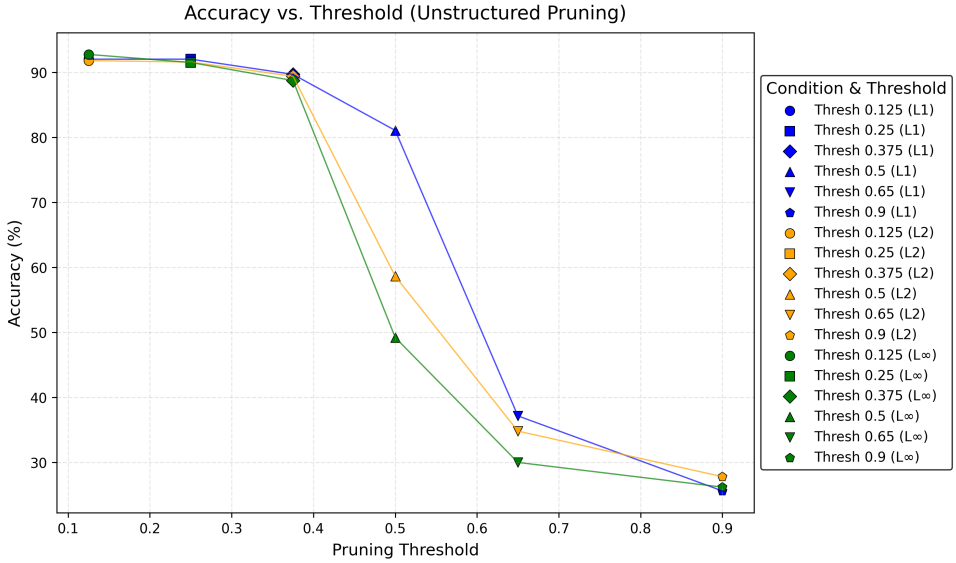

Figure 13: Accuracy vs. Pruning Threshold for L1, L2, and L-infinity norm Unstructured Pruning. The results illustrate the resilience of L1 pruning compared to the rapid accuracy decline in L2 and L-infinity norm pruning as the threshold increases.

7.2 Structured Pruning and Conversion for MCU Deployment

Structured pruning removes entire structures such as channels, filters, or layers rather than individual weights. This approach leads to a more hardware-efficient model, reducing latency and improving performance, particularly for deployment on resource-constrained devices like microcontrollers (MCUs). In this study, we applied structured pruning to the TinyConv convolutional neural network designed for audio classification. The following sections describe the pruning methodology, conversion process, and experimental results.

7.2.1 Thresholds and Methodology

We employed L_1-norm-based structured pruning, where entire channels in convolutional layers and entire neurons in fully connected layers were pruned based on their L_1-norm magnitude. The pruning threshold (th) defines the proportion of channels or neurons removed, with six thresholds evaluated: 0.125, 0.25, 0.375, 0.5, 0.65, and 0.9. Each threshold determines the extent of structural reduction and its impact on model performance.

1. **Pruning Process**: Pruning was performed along the channel dimension (`dim=0`) for convolutional layers and along the neuron dimension for fully connected layers. After pruning, the pruned weights were permanently removed from the model. Additionally, the reshaped fully connected layer's input features were updated dynamically after pruning.

2. **Evaluation Metrics**:

   - **Accuracy (A_p)**: The pruned model was evaluated on the test dataset under different pruning thresholds, both before and after fine-tuning, using inference on a test set. Accuracy values were recorded for comparison.
   - **Parameters (P)**: The remaining number of trainable parameters was computed by summing the non-zero elements in the pruned convolutional and fully connected layers.
   - **FLOPs (F)**: The total floating point operations (FLOPs) were estimated for an input shape of $(1, 1, W, H)$, where W and H are the fingerprint width and spectrogram length. The pruned channels were fully removed from the model before computing FLOPs to ensure an accurate measurement.
   - **Runtime on CPU (T_{CPU})**: The inference time for a single forward pass was measured on a desktop CPU using a randomly generated input of shape $(1, 1, W, H)$. The model was transferred to the CPU for measurement and returned to its original device after testing. Accuracy was plotted against runtime on the CPU.
   - **Runtime on MCU (T_{MCU})**: The pruned models were deployed onto an MCU, and inference time was measured.

   The pruned models were prepared for MCU deployment through a multi-step pipeline. Each PyTorch model was first exported to ONNX using `torch.onnx.export`, then converted to TensorFlow's SavedModel format via `onnx-tf`. Next, the models were transformed into TensorFlow Lite (TFLite) format, producing both floating-point and quantized INT8 versions. For MCU compatibility, the quantized TFLite model was converted into a C array using `xxd` and standardized to `g_model` via `sed`. Finally, inference testing was conducted using a TFLite interpreter to validate accuracy, ensuring optimized deployment on embedded systems.

7.2.2 Channel Pruning and Impact on FLOPs and Latency

By removing entire filters or channels, structured pruning directly reduces the number of computations (FLOPs) required for inference, leading to:

- Lower FLOPs, proportional to the reduction in channels.
- Decreased latency, enhancing real-time performance on CPUs and MCUs.
- A compact model optimized for embedded systems.

The plots (Figures 14–17) utilize a scientific visualization approach, incorporating scatter points to represent different pruning thresholds, with accuracy percentages annotated for clarity. Two conditions are compared: **No Fine-tuning** (blue) and **Fine-tuning** (orange), illustrating the trade-offs between model complexity (measured in parameters and FLOPs) and inference efficiency (runtime) in relation to accuracy.

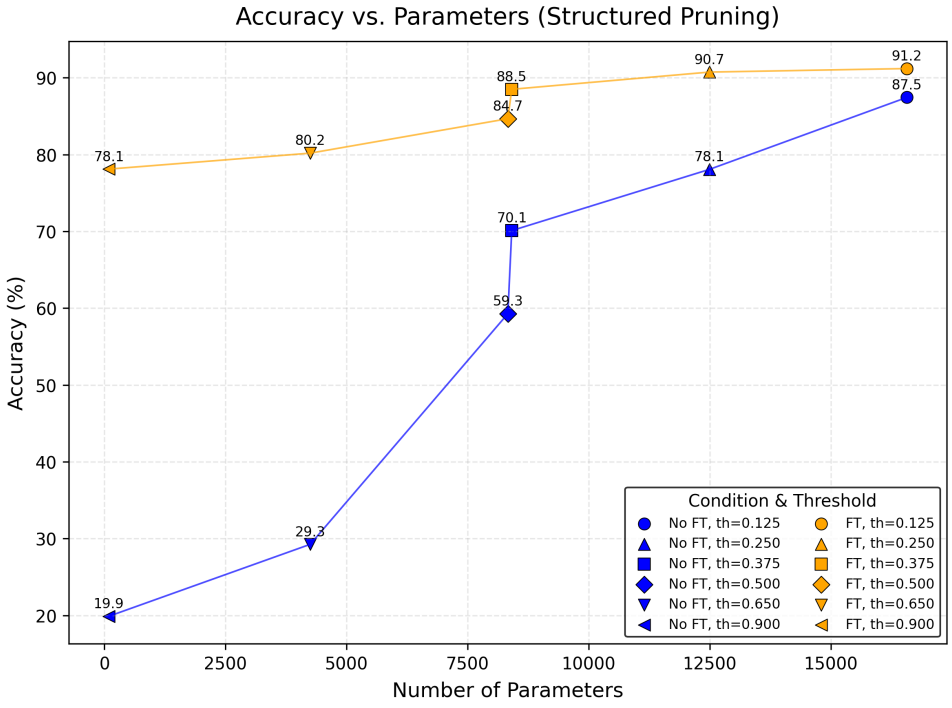

Figure 14: Accuracy vs Parameters for Structured Pruned Models

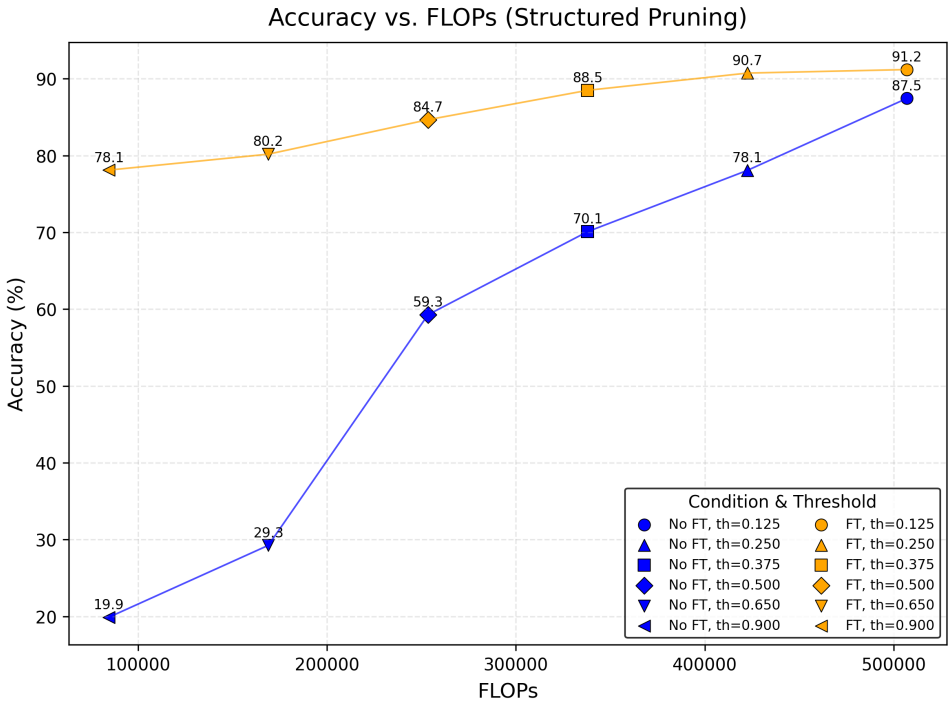

Figure 15: Accuracy vs FLOPs for Structured Pruned Models

Figure 15 illustrates the relationship between model accuracy and FLOPs, where FLOPs serve as a direct indicator of computational complexity. As channels are pruned from the network, both the forward and backward pass computations decrease, leading to a more efficient model. However, our results indicate that while reducing FLOPs also leads to an accuracy drop, fine-tuning mitigates this decline. The fine-tuning process allows the model to recover a significant portion of its lost accuracy, with improvements stabilizing over multiple epochs. Our experimental results demonstrate that fine-tuning effectiveness varies based on the pruning threshold. While models pruned at lower thresholds exhibit strong recovery, highly pruned models show diminishing returns, indicating that excessive pruning can lead to irreversible information loss. These findings highlight the importance of selecting appropriate pruning levels and leveraging fine-tuning to maximize efficiency without severely compromising model accuracy.

7.2.3 Inference Time for Pruned Models on Desktop CPU and MCU Device

Inference times on a desktop CPU (float32) ranged from 0.9 ms ($th = 0.125$) to 0.44 ms ($th = 0.9$), reflecting a reduction in computational load with increased pruning. Full results are:

- **CPU Times (ms)**: $[0.9, 0.56, 0.53, 0.48, 0.45, 0.445]$ for $th = [0.125, 0.25, 0.375, 0.5, 0.65, 0.9]$.

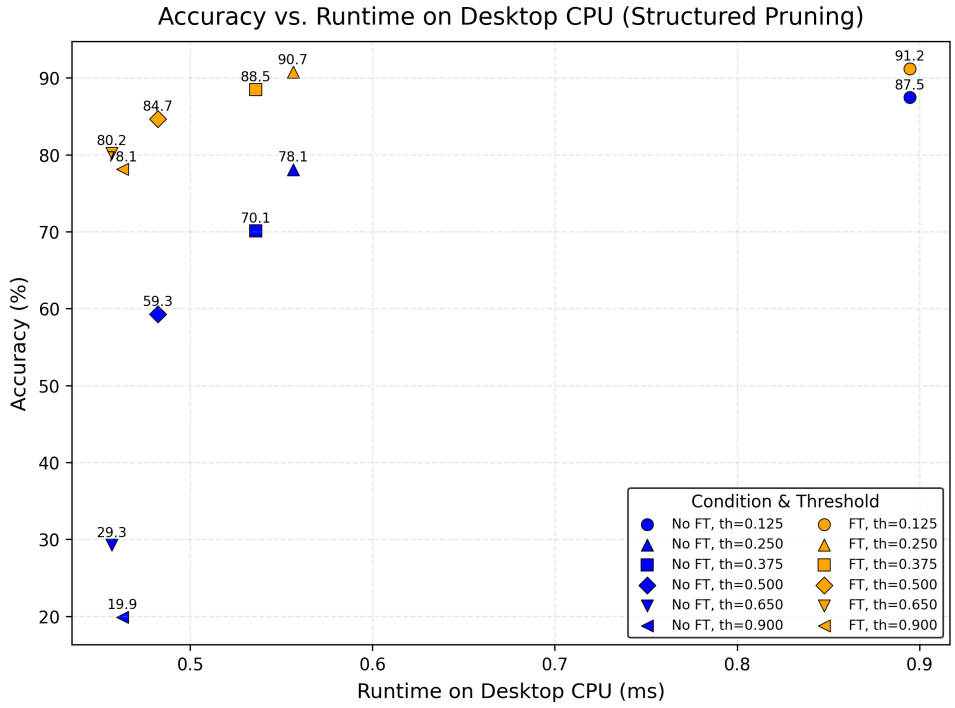

Figure 16: Accuracy vs Runtime for Pruned Models (Structured Pruning on Desktop CPU)

The impact of structured pruning on a desktop CPU was assessed by measuring both accuracy and runtime at various pruning thresholds, as shown in Figure 16. The results demonstrate a trade-off between accuracy and inference speed. As the pruning threshold increased, the runtime decreased, indicating a faster inference process. However, at high pruning thresholds, accuracy declined notably, suggesting the existence of an optimal pruning level where the model maintains a balance between accuracy and speed.

The model was also deployed on an MCU to evaluate its performance in a resource-constrained environment, with the findings presented in Figure 17. The reported runtime represents the average over 300 inference runs. Similar to the CPU results, increased pruning threshold resulted in faster runtimes on the MCU.

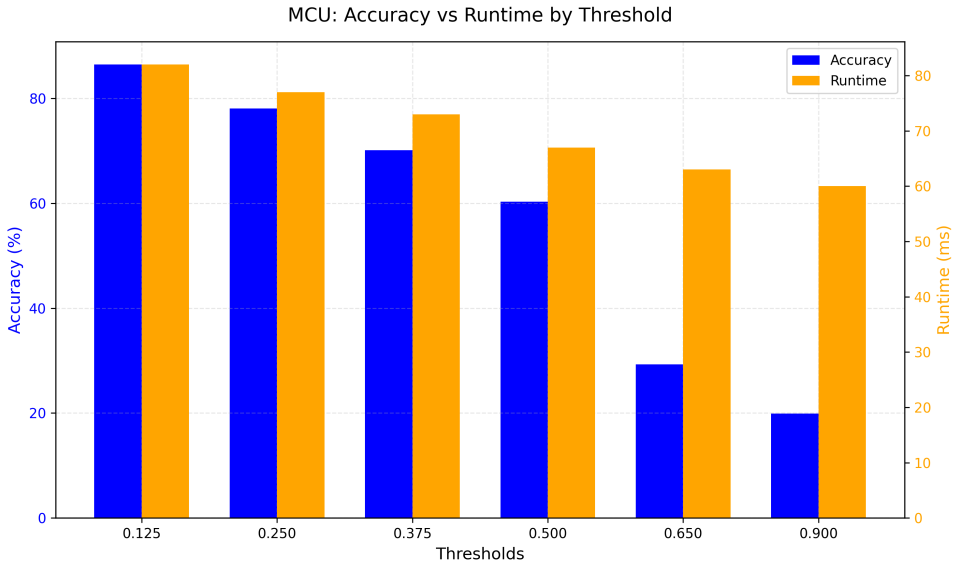

Figure 17: Accuracy vs Runtime for Pruned Models on MCU

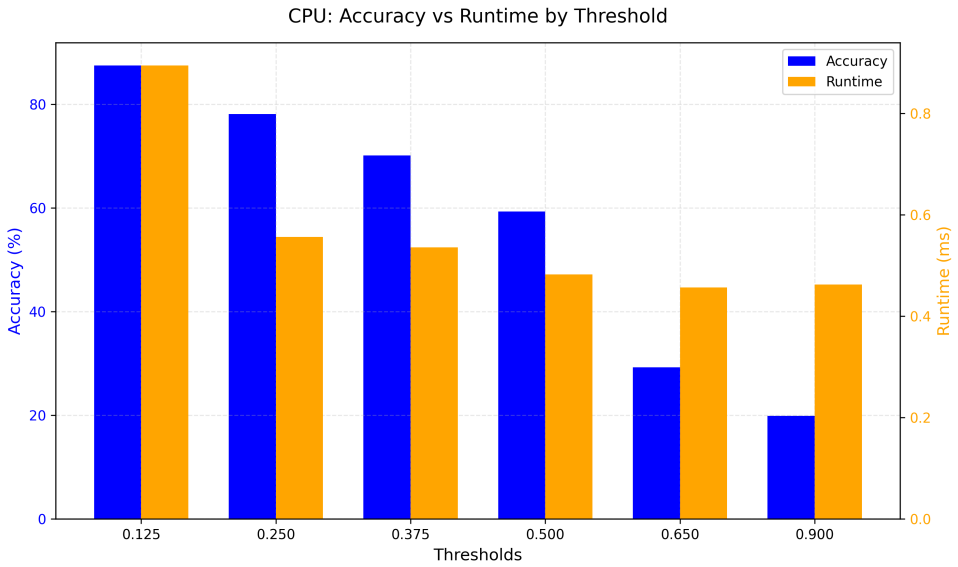

Figure 18: Accuracy vs Runtime for Pruned Models on MCU

A notable observation is that while MCU runtime consistently decreases with higher pruning thresholds, CPU runtime does not follow a strictly monotonic trend. This behavior can be attributed to several architectural and operational characteristics unique to modern CPUs. Unlike MCUs, CPUs incorporate advanced execution mechanisms such as out-of-order execution, speculative execution, and branch prediction, which can offset the expected reductions in runtime from model pruning. Furthermore, features like SIMD instructions and the ability to execute multiple instructions per cycle may limit the runtime benefits gained through pruning. Additionally, memory access patterns and caching strategies introduce further complexity, as a decrease in computational operations does not always directly translate to a proportional reduction in runtime. In contrast, MCUs, which have a simpler architecture and more limited computational resources, exhibit a more predictable relationship between model simplification and runtime reduction. This distinction underscores the need for platform-specific optimization strategies when deploying deep learning models on embedded systems.

Note: Please ignore the extra notebook (`Extra_credit_part_4.ipynb`) in my repository related to the extra credit. It is not part of the main submission.

References

[1] T. N. Sainath and C. Parada, "Convolutional Recurrent Neural Networks for Small-Footprint Keyword Spotting," *arXiv preprint*, arXiv:1703.05390, 2017.

[2] Y. Nahshan, B. Chmiel, C. Baskin, E. Zheltonozhskii, R. Banner, A. M. Bronstein, and A. Mendelson, "Loss aware post-training quantization," *Machine Learning*, vol. 110, pp. 3245–3262, 2021. [Online].

[3] C. Gernigon, S.-I. Filip, O. Sentieys, C. Coggiola, and M. Bruno, "AdaQAT: Adaptive Bit-Width Quantization-Aware Training," *arXiv preprint arXiv:2404.16876*, 2024.

[4] "Quantization aware training," TensorFlow Model Optimization.

[5] J. Liu, L. Niu, Z. Zhang, S. Liu, X. Li, and Y. Wang, "PD-Quant: Post-Training Quantization Based on Prediction Difference Metric," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2023, pp. 19631–19640.

[6] M. Gale, P. Elsen, and S. Hooker, "Sparse Neural Networks and Their Acceleration," *Proceedings of the IEEE*, vol. 109, no. 4, pp. 456-469, 2021.

[7] B. Moons and M. Verhelst, "An Energy-Efficient Precision-Scalable ConvNet Processor in 40-nm CMOS," *IEEE Journal of Solid-State Circuits*, vol. 52, no. 4, pp. 903-914, 2017.

[8] N. P. Jouppi et al., "In-Datacenter Performance Analysis of a Tensor Processing Unit," in *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA)*, 2017, pp. 1–12.

[9] D. Patterson et al., "The Carbon Footprint of Machine Learning Training Will Plateau, Then Shrink," *arXiv preprint arXiv:2104.10350*, 2021.

[10] P. Banbury et al., "Benchmarking TinyML Systems: Challenges and Direction," *Proceedings of the NeurIPS ML for Systems Workshop*, 2021.

[11] A. R. Kang, J. H. Kim, and H. S. Kim, "Dynamic Sparse Inference for Efficient Deep Learning Models," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 33, no. 7, pp. 3019-3031, 2022.