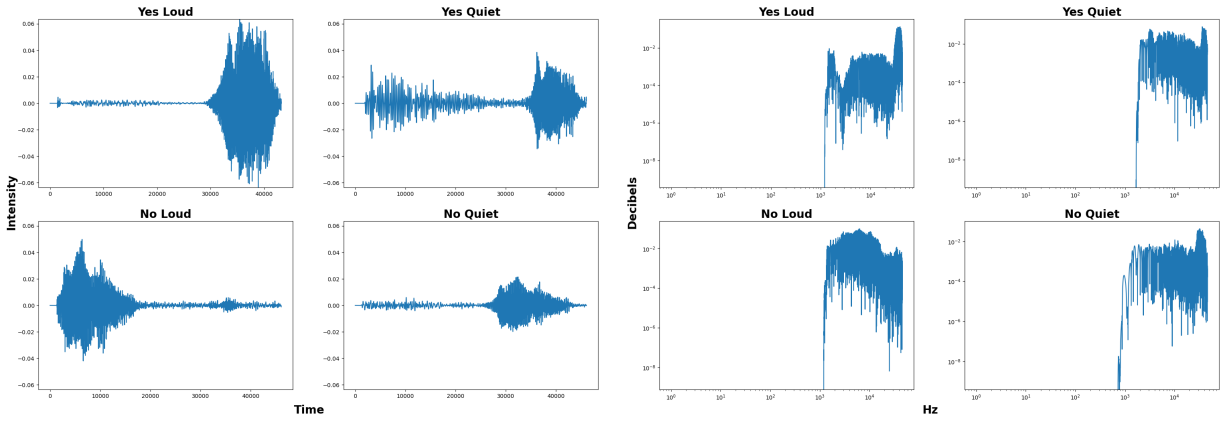
Machine Learning Hardware and Systems

ECE 5545 A2

Beichen Ma (bm685)

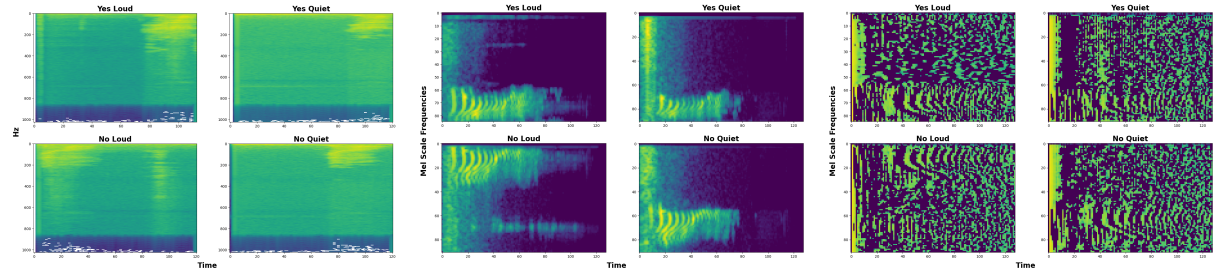
Cornell Tech

Mar 2024



(a) Time domain (b) Frequency domain

Figure 1: Time domain and frequency domain of the audio



(a) Spectogram (b) Mel Spectrogram (c) MFCC

Figure 2: Different spectograms (e.g. spectogram, mel spectogoram, and mel frequency cepstral coefficients (MFCC)) of the audio

1 Preprocessing & Visualization

The plots of time domain and frequency domain of the audio are shown in Figure 1. We preprocess input audio sending it through a neural neural network for the following reasons:

- Normalization ensures consistent audio levels across recordings, despite variations in microphone sensitivity, sound source distance, and recording environments. By adjusting the amplitude of the audio signal to a standard level, normalization enhances the neural network's capacity for data processing and analysis.
- Neural networks often work with features that represent the audio signal in a more informative way, such as Mel-frequency cepstral coefficients (MFCCs), spectrograms, or chromagrams. These features can capture important aspects of the sound, such as pitch, tone, and rhythm, making it easier for the network to understand and classify audio signals.

From Figure 2, MFCC and Mel spectrograms provide a clearer distinction between the "Yes" and "No" utterances, as well as between "Loud" and "Quiet" variations. MFCCs are often preferred in voice recognition systems because they capture the important characteristics of human speech while being compact and efficient. Mel spectrograms might be better for general audio processing tasks where the emphasis on lower

Environment	Self CPU Time	Self CUDA Time
Colab CPU	14.761 ms	-
Colab GPU	-	28.000 us

Table 1: Inference runtime of DNN on Colab CPU and GPU.

Dataset Split	Number of Samples
Training Set	10,556
Validation Set	1,333
Test Set	1,368
Total Classes of Keywords	4

Table 2: Summary of the Speech Commands Dataset

frequencies, which are more critical in human speech perception, is beneficial. Linear frequency spectrograms could be more suitable when the precise distribution of energy across the entire frequency spectrum is important, such as in certain types of music analysis or signal processing applications where one needs to capture high-frequency details.

2 Model Size Estimation

To estimate the flash usage of the DNN on MCU, we consider the space required to store the model's weights and any other code necessary for the model to run. From the Code Snippet 7, the total number of trainable parameters is 0.016652 million. Given the total number of trainable parameters (N) in millions (M), and assuming each parameter is stored as a 32-bit floating point number (FP32), which occupies 4 bytes, we can calculate the estimated flash usage (S_{flash}) as follows:

$$S_{flash} = N \times 10^6 \times 4 \text{ bytes/parameter}$$

For the given number of parameters:

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

To convert megabytes to kilobytes:

$$S_{flash} = 0.066608 \text{ MB} \times 1024 \text{ KB/MB} = 68.24 \text{ KB}$$

Thus, the estimated flash usage for the model weights is approximately 68.24 KB. The percentage of flash used is calculated as follows:

training accuracy for float32 TinyConv

	silence	_unknown_	yes	no	total
#samples	2099.000	2099.000	3228.000	3130.000	10556.000
#correct	2077.000	1548.000	3056.000	2846.000	9527.000
accuracy	0.990	0.737	0.947	0.909	0.903

testing accuracy for float32 TinyConv

	silence_	_unknown_	yes	no	total
#samples	272.000	272.000	419.000	405.000	1368.000
#correct	270.000	206.000	395.000	365.000	1236.000
accuracy	0.993	0.757	0.943	0.901	0.904

validation accuracy for float32 TinyConv

	silence	_unknown_	yes	no	total
#samples	265.000	265.000	397.000	406.000	1333.000
#correct	265.000	210.000	383.000	356.000	1214.000
accuracy	1.000	0.792	0.965	0.877	0.911

Figure 3: Accuracy report

$$\begin{aligned}\text{Percentage of Flash Used} &= \left(\frac{\text{Model Size in KB}}{\text{Total Flash Memory in KB}}\right) \times 100 \\ &= \left(\frac{68.24\,\text{KB}}{1024\,\text{KB}}\right) \times 100 \approx 6.66\%\end{aligned}$$

To estimate the RAM usage of a DNN, we consider the memory required for the forward pass. From the Code Snippet 12, the forward memory is 0.007856 million parameters. The RAM usage could be obtained as follows:

$$\text{Forward Memory in KB} = \frac{0.007856 \times 10^6\,\text{parameters} \times 4\text{bytes/parameters}}{1024\,\text{bytes/KB}} \approx 8.045\,\text{KB}$$

Given the MCU's total RAM of 256 KB, the percentage of RAM used by the model is calculated as follows:

$$\text{Percentage of RAM Used} = \left(\frac{\text{Forward Memory in KB}}{256\,\text{KB}}\right) \times 100 \approx 3.14\%$$

From the Code Snippet 13, we obtain that given a forward pass of our model, we have computed the total number of floating-point operations (FLOPs) as 672,024, with the FLOPs broken down by layer: Convolutional layer: 640,008 FLOPs, Fully Connected layer: 32,004 FLOPs

Comparing this to a more complex model such as DeepSpeech2 [1], which employs recurrent neural networks (RNNs) and bidirectional layers, we expect that the computational complexity would be significantly

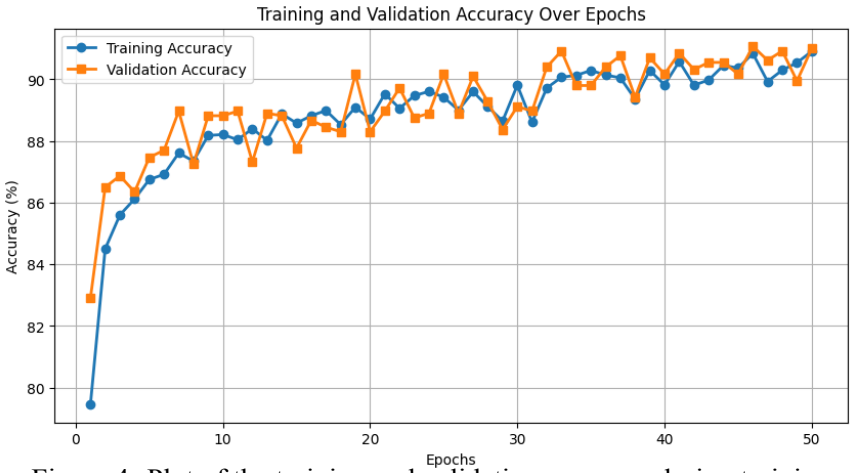


Figure 4: Plot of the training and validation accuracy during training

Process Stage	Min Time	Max Time	Avg Time
Preprocessing	19 ms	23 ms	21 ms
Inference	88 ms	88 ms	86 ms
Post-processing	0 ms	1 ms	0 ms

Table 3: Timing Breakdown for Different Stages of the Model Execution on MCU

higher. The DeepSpeech2 model is designed to process longer sequences of data and achieve higher accuracy, which typically results in a much larger number of FLOPs. DeepSpeech2 sustains approximately 50 teraFLOP/second when training on 16 GPUs, which translates to 3.125 teraFLOP/second/GPU. For our model, which requires $672,024$ FLOPs per forward pass, we can calculate the number of forward passes possible per second on one GPU as:

$$\frac{3.125 \times 10^{12} \text{ FLOP/second/GPU}}{672,024 \text{ FLOP/pass}} \approx 4650 \text{ passes/second/GPU}$$

This rough estimation highlights the significant difference in computational complexity between the two models. While DeepSpeech2 is designed for accuracy and complexity, capable of utilizing substantial computational resources during training, our model is much more computationally efficient, suitable for environments with strict resource constraints such as edge devices. The inference time of the model is shown in Table 1.

3 Training & Analysis

The accuracy result and plots are shown in Figure 3 and 4.

From the Table 2, there are 4 classes of keywords are supported (e.g. yes, no, unknown, and silence). The dataset consists of 10,556 training samples, 1,333 validation samples and 1,368 test samples.

Testset	Yes	No	Go	Yes	S	Q	No	yes	Start	Help
First Trial	yes	no	no	Yes	Yes	no	no	Yes	unknown	no
Second Trial	yes	no	no	Yes	Yes	no detection	No	Yes	unknown	no
Third Trial	yes	no	no	Yes	no detection	no detection	no	Yes	unknown	no

Table 4: Model prediction results across three trials

4 Model Conversion and Deployment

To profile running time and plot the breakdown between preprocessing, neural network, and post-processing on arduino, we use 'millis()' function defined in 'PROFILE_MICRO_SPEECH'. The profile logic is shown in Figure 14. The profile's running time is obtained by averaging the results of 1000 trials to minimize randomness.

The profile result is shown in Table 3. To compare the inference time on MCU with that of the Colab CPU and GPU, we consider the average time taken for inference on each platform. The MCU has an average inference time of 86 ms, whereas the Colab CPU and GPU times are 14.761 ms and 28.000 us respectively. To facilitate a direct comparison, the GPU time is converted to milliseconds, resulting in 0.028 ms. The slowdown factor for each platform is calculated as the ratio of the MCU inference time to the respective platform's time:

$$\text{Slower percentage (CPU)} = \frac{T_{MCU} - T_{CPU}}{T_{CPU}} = \frac{86 - 14.761}{14.761} \approx 4.83$$

$$\text{Slower percentage (GPU)} = \frac{T_{MCU} - T_{GPU}}{T_{GPU}} = \frac{86 - 0.028}{0.028} \approx 3,070.43$$

These calculations indicate that the MCU is approximately 4.83 times slower than the Colab CPU and 3070.43 times slower than the Colab GPU during the inference phase of a deep neural network operation.

The model was tested with a set of 10 keywords, and the predictions were recorded over three trials. Table 4 summarize the testing results.

5 Quantization-Aware Training

The result of accuracies of 4-bit and 8-bit quantization and post-quantized TinyConv are shown in Figure 5 and 6. From the plot of accuracy v.s. bit-width for both post-training quantization and quantization-aware training shown in Figure 7, post-training quantization yields slightly higher accuracy than quantized-aware training. In PTQ, the model is first trained using floating-point arithmetic and then quantized, which means that the original model's parameters are approximated to reduce the model size and computational requirements. This approach seems to retain more of the original model's predictive capabilities in this case.

Training accuracy for 4-bit Quantized TinyConv

	silence	unknown	yes	no	total
#samples	2099.000	2099.000	3228.000	3130.000	10556.000
#correct	1981.000	1574.000	3012.000	2800.000	9367.000
accuracy	0.944	0.750	0.933	0.895	0.887

(a)

Training accuracy for 8-bit Quantized TinyConv

	silence	unknown	yes	no	total
#samples	1283.000	1283.000	3228.000	658.000	6452.000
#correct	1274.000	1157.000	3138.000	581.000	6150.000
accuracy	0.993	0.902	0.972	0.883	0.953

(b)

Validation accuracy for 4-bit Quantized TinyConv

	silence	unknown	yes	no	total
#samples	265.000	265.000	397.000	406.000	1333.000
#correct	251.000	206.000	373.000	346.000	1176.000
accuracy	0.947	0.777	0.940	0.852	0.882

(c)

Validation accuracy for 8-bit Quantized TinyConv

	silence	unknown	yes	no	total
#samples	158.000	158.000	397.000	79.000	792.000
#correct	158.000	151.000	390.000	66.000	765.000
accuracy	1.000	0.956	0.982	0.835	0.966

(d)

Testing accuracy for 4-bit Quantized TinyConv

	silence	unknown	yes	no	total
#samples	272.000	272.000	419.000	405.000	1368.000
#correct	255.000	208.000	388.000	351.000	1202.000
accuracy	0.938	0.765	0.926	0.867	0.879

(e)

Testing accuracy for 8-bit Quantized TinyConv

	silence	unknown	yes	no	total
#samples	171.000	171.000	419.000	98.000	859.000
#correct	170.000	154.000	405.000	82.000	811.000
accuracy	0.994	0.901	0.967	0.837	0.944

(f)

Figure 5: Accuracy of 4-bit and 8-bit quantized TinyConv across training, validation, and testing.

Training accuracy for Post Quantized 4-bit TinyConv

	silence	unknown	yes	no	total
#samples	2099.000	2099.000	3228.000	3130.000	10556.000
#correct	2012.000	1717.000	2901.000	2853.000	9483.000
accuracy	0.959	0.818	0.899	0.912	0.898

(a)

Training accuracy for Post Quantized 8-bit TinyConv

	silence	unknown	yes	no	total
#samples	1283.000	1283.000	3228.000	658.000	6452.000
#correct	1259.000	1182.000	3137.000	574.000	6152.000
accuracy	0.981	0.921	0.972	0.872	0.954

(b)

Validation accuracy for Post Quantized 4-bit TinyConv

	silence	unknown	yes	no	total
#samples	265.000	265.000	397.000	406.000	1333.000
#correct	251.000	218.000	361.000	357.000	1187.000
accuracy	0.947	0.823	0.909	0.879	0.890

(c)

Validation accuracy for Post Quantized 8-bit TinyConv

	silence	unknown	yes	no	total
#samples	158.000	158.000	397.000	79.000	792.000
#correct	154.000	151.000	388.000	65.000	758.000
accuracy	0.975	0.956	0.977	0.823	0.957

(d)

Testing accuracy for Post Quantized 4-bit TinyConv

	silence	unknown	yes	no	total
#samples	272.000	272.000	419.000	405.000	1368.000
#correct	253.000	226.000	366.000	361.000	1206.000
accuracy	0.930	0.831	0.874	0.891	0.882

(e)

Testing accuracy for Post Quantized 8-bit TinyConv

	silence	unknown	yes	no	total
#samples	171.000	171.000	419.000	98.000	859.000
#correct	167.000	155.000	401.000	83.000	806.000
accuracy	0.977	0.906	0.957	0.847	0.938

(f)

Figure 6: Accuracy of 4-bit and 8-bit post quantized TinyConv across training, validation, and testing.

On the other hand, QAT integrates the quantization process into the training loop, which allows the model to adjust its parameters in anticipation of the reduced precision. However, QAT might not always reach the same level of accuracy as PTQ because the training process needs to balance the learning of the task at hand with the added constraint of low-precision weights.

In summary, while QAT generally prepares the model better for the reduced precision due to quantization, PTQ in this case achieves marginally higher accuracy, suggesting that the models are robust to quantization and can maintain performance even when quantization is applied post-training.

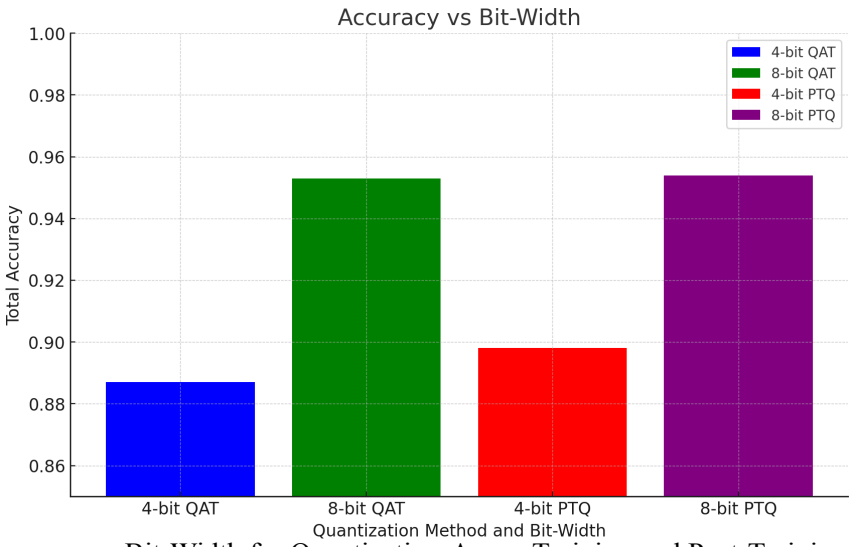


Figure 7: Accuracy vs Bit-Width for Quantization-Aware Training and Post-Training Quantization

6 Pruning

6.1 Unstructured pruning

The L1 norm, L2 norm, and L-infinity norm are mathematical tools used to measure the size or length of vectors in different ways. These norms have various applications, including regularization, optimization, and model pruning in machine learning.

L1 Norm

- Definition: The L1 norm of a vector is the sum of the absolute values of its components, also known as the Manhattan distance or taxicab norm.
- Formula: For a vector $x = [x_1, x_2, \ldots, x_n]$, the L1 norm is defined as $\|x\|_1 = |x_1| + |x_2| + \ldots + |x_n|$.
- Characteristics: The L1 norm encourages sparsity in the vector it is applied to, making it useful for feature selection and model pruning in machine learning.

L2 Norm

- Definition: The L2 norm of a vector is the square root of the sum of the squares of its components, known as the Euclidean norm.
- Formula: For a vector x, the L2 norm is $\|x\|_2 = \sqrt{x_1^2 + x_2^2 + \ldots + x_n^2}$.
- Characteristics: The L2 norm tends to distribute errors among all terms equally and is used in machine learning to prevent overfitting by discouraging large weights.

L-infinity Norm

- Definition: The L-infinity norm of a vector is the maximum absolute value of its components.

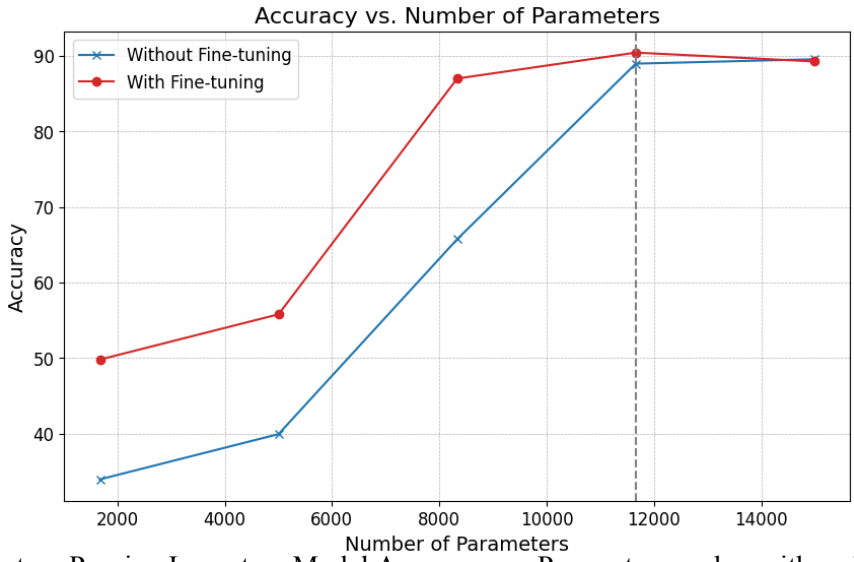


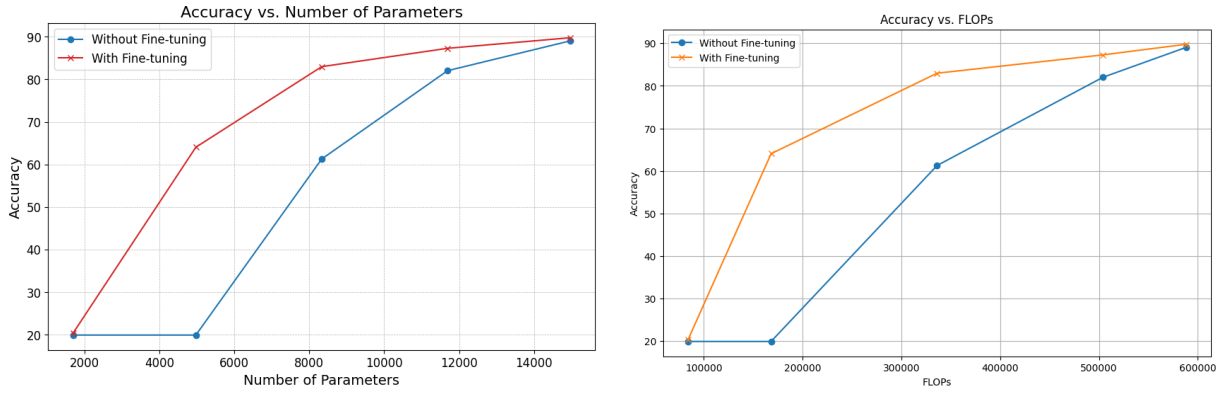
Figure 8: Unstructure Pruning Impact on Model Accuracy vs. Parameter number with and without Finetuning

- Formula: For a vector x, the L-infinity norm is $\|x\|_\infty = \max(|x_1|, |x_2|, \ldots, |x_n|)$.
- Characteristics: The L-infinity norm focuses on the single largest term, ignoring other components' magnitudes. It is less common in machine learning compared to the L1 and L2 norms.

The L1 norm is generally considered the best for pruning because it directly induces sparsity in the model's parameters. This leads to simpler, more interpretable models that are efficient to store and compute. While the L2 norm is effective for regularization, it does not naturally induce sparsity. The L-infinity norm is less commonly used for pruning purposes.

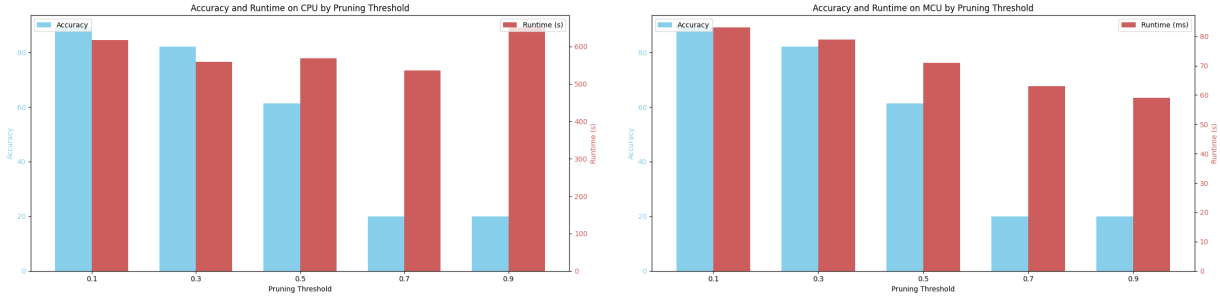
The plot for unstructured pruning result is shown in Figure 8. The plot depicts the impact of unstructured pruning on model accuracy as a function of the number of parameters. As the pruning threshold increases (0.1 to 0.9), we observe that the accuracy improves with a larger number of parameters retained. However, there exists a pronounced "cliff" in both curves, particularly 11660, where the accuracy sharply declines. This suggests that beyond a certain pruning threshold, the removal of parameters significantly deteriorates the model's predictive capabilities.

The curve with fine-tuning demonstrates a consistent and pronounced advantage over the one without fine-tuning. Fine-tuning post-pruning helps in recovering some of the lost accuracy, underscoring its importance in the pruning process. It is noteworthy that the accuracy of the fine-tuned model plateaus, indicating a point of diminishing returns where increasing the number of parameters further does not yield significant improvements in accuracy. Regarding the utilization for Speeding Up Computation, unstructured pruning can be leveraged to accelerate computation by reducing the number of parameters within a neural network. By pruning the parameters that contribute the least to the output of the model, we can create a sparser network.



(a) Parameters (b) FLOPs

Figure 9: Structure Pruning Impact on Model Accuracy vs. Parameter number and FLOPs with and without Finetuning



(a) CPU runtime (b) MCU runtime

Figure 10: Structure Pruning Impact on Model Accuracy vs. runtime on CPU and MCU with and without Finetuning

This sparsity translates into fewer mathematical operations, thereby reducing the computational load during both training and inference.

In conclusion, to maximize computational efficiency while maintaining accuracy, we should prune up to the point just before the ”cliff” in the accuracy curve. Fine-tuning is critical after pruning to restore, as much as possible, any lost accuracy. The trade-off between computation and accuracy must be carefully managed to ensure the pruned model meets the performance requirements of the specific application it is intended for.

6.2 Structured pruning

Structured pruning, specifically channel pruning, focuses on reducing the complexity of a neural network by eliminating entire channels, as opposed to individual weights. This approach has implications for the model’s accuracy, computational efficiency, and deployability on hardware with limited resources, such as microcontrollers (MCUs).

Our empirical results demonstrate a clear trend: as the pruning threshold increases, the number of parameters decreases. However, the relationship between pruning and accuracy is not linear. The accuracy remains relatively stable until reaching a critical threshold, beyond which it begins to drop significantly. This effect

Pruning Threshold	Parameters	FLOPs	Actual Channels	Actual Neurons	Runtime CPU (us)	Runtime MCU (ms)
0.1	14972	588011	7	3600	618.04	83
0.3	11692	504010	6	2800	559.21	79
0.5	8332	336008	4	2000	569.21	71
0.7	4972	168006	2	1200	536.18	63
0.9	1692	84005	1	400	650.79	59

Table 5: Structured Pruning: Impact on Parameters, FLOPs, Runtime, and Model Size

is mitigated by fine-tuning, which helps recover some of the lost accuracy (see Figure 9(a)). Fine-tuning adjusts the remaining weights in the network to compensate for the loss of pruned channels, underscoring its necessity in structured pruning strategies.

Figure 9(b) shows the plot of accuracy versus FLOPs, FLOPs is a direct measure of computational complexity. As channels are pruned and eliminated from the network, the FLOPs required for both forward and backward passes are reduced, resulting in a more computationally efficient model. Our findings show a reduction in FLOPs correlates with a decrease in accuracy, yet fine-tuning can again reduce this negative impact. This balance between computational cost and model performance is crucial for deploying models in real-world applications where resources are constrained.

In our experiment, we inspect the effective number of channels and neurons of the model after pruning, parameter numbers, FLOPs, runtime on CPU and runtime on MCU. The details can be found in the Table 5

The plots for accuracy versus runtime are shown in Figure 10. The impact of structured pruning on a desktop CPU was assessed by measuring both the accuracy and runtime at various pruning thresholds, which are depicted in Figure 10(a). 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 extreme pruning thresholds, accuracy declined notably, suggesting the existence of an optimal pruning level where the model preserves a balance between accuracy and speed.

The model was also deployed on MCU to evaluate its performance in a resource-constrained environment, with the findings presented in Figure 10(b). The runtime result is reported over the average of 1000 times. Similar to the CPU results, increased pruning resulted in faster runtimes on the MCU. Notably, the decline in accuracy was less steep on the MCU compared to the CPU, potentially indicating that MCUs may tolerate higher pruning thresholds before experiencing a significant drop in performance. This could be attributed to the different processing capabilities and optimizations present in MCU environments.

We also observe a phenomenon where CPU runtime does not decrease monotonically with increased pruning threshold, while MCU runtime does. We discuss the potential factors can be attributed to several architectural and operational characteristics of modern CPUs. Unlike MCUs, CPUs possess advanced execution features such as out-of-order execution, speculative execution, and sophisticated branch prediction, which may mitigate the expected runtime gains from reduced model complexity [3]. Moreover, the presence of SIMD

instructions and the ability to execute multiple instructions per cycle can result in diminished returns from pruning when it comes to runtime [5]. Memory access patterns and caching mechanisms further complicate this relationship, as the reduction in the number of operations due to pruning does not linearly translate to a decrease in runtime [4]. On the other hand, MCUs, with their simpler architectures and constrained computational resources, exhibit more direct correlations between model simplification and runtime improvements [2].

References

[1] Dario Amodei, Sundaram Ananthanarayanan, Rishita Anubhai, Jingliang Bai, Eric Battenberg, Carl Case, Jared Casper, Bryan Catanzaro, Qiang Cheng, Guoliang Chen, Jie Chen, Jingdong Chen, Zhijie Chen, Mike Chrzanowski, Adam Coates, Greg Diamos, Ke Ding, Niandong Du, Erich Elsen, Jesse Engel, Weiwei Fang, Linxi Fan, Christopher Fougner, Liang Gao, Caixia Gong, Awni Hannun, Tony Han, Lappi Johannes, Bing Jiang, Cai Ju, Billy Jun, Patrick LeGresley, Libby Lin, Junjie Liu, Yang Liu, Weigao Li, Xiangang Li, Dongpeng Ma, Sharan Narang, Andrew Ng, Sherjil Ozair, Yiping Peng, Ryan Prenger, Sheng Qian, Zongfeng Quan, Jonathan Raiman, Vinay Rao, Sanjeev Satheesh, David Seetapun, Shubho Sengupta, Kavya Srinet, Anuroop Sriram, Haiyuan Tang, Liliang Tang, Chong Wang, Jidong Wang, Kaifu Wang, Yi Wang, Zhijian Wang, Zhiqian Wang, Shuang Wu, Likai Wei, Bo Xiao, Wen Xie, Yan Xie, Dani Yogatama, Bin Yuan, Jun Zhan, and Zhenyao Zhu. 2016. Deep Speech 2 : End-to-End Speech Recognition in English and Mandarin. In *Proceedings of The 33rd International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 48)*, Maria Florina Balcan and Kilian Q. Weinberger (Eds.). PMLR, New York, New York, USA, 173–182. `https://proceedings.mlr.press/v48/amodei16.html`

[2] Song Han, Jeff Pool, John Tran, and William J. Dally. 2015. Learning both Weights and Connections for Efficient Neural Networks. arXiv:1506.02626 [cs.NE]

[3] John L. Hennessy and David A. Patterson. 2012. *Computer Architecture: A Quantitative Approach* (5 ed.). Morgan Kaufmann, Amsterdam.

[4] Sparsh Mittal. 2014. A Survey of Techniques For Improving Energy Efficiency in Embedded Computing Systems. arXiv:1401.0765 [cs.AR]

[5] Samuel Williams, Andrew Waterman, and David A. Patterson. 2009. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM* 52, 4 (2009), 65–76. `http://dblp.uni-trier.de/db/journals/cacm/cacm52.html#WilliamsWP09`

7 Appendix

```
# Sends model weights to the GPU if tensors are on GPU
if torch.cuda.is_available():
    model_fp32.cuda()

from src.size_estimate import count_trainable_parameters
num_params = count_trainable_parameters(model_fp32)
print("Total number of trainable parameters: ", num_params / float(1e6), "M")
# Output: Total number of trainable parameters: 0.016652 M
```

Figure 11: Code to count the number of trainable parameters in a model.

```
# Sends model weights to the GPU if tensors are on GPU
if torch.cuda.is_available():
    model_fp32.cuda()

from src.size_estimate import compute_forward_memory
frd_memory = compute_forward_memory(
    model_fp32,
    (1, model_fp32.model_settings['fingerprint_width'],
        model_fp32.model_settings['spectrogram_length']),
    device
)
print("Forward memory: ", frd_memory / float(1e6), "M")
# Output: Forward memory: 0.007856 M
```

Figure 12: Code to compute the memory needed for a forward pass.

```
from pprint import pprint
from src.size_estimate import flop

if torch.cuda.is_available():
    model_fp32.cuda()

# The total number of floating point operations
flop_by_layers = flop(
    model=model_fp32,
    input_shape=(
        1,
        model_fp32.model_settings['fingerprint_width'],
        model_fp32.model_settings['spectrogram_length']
    ),
    device=device)
total_param_flops = sum([sum(val.values()) for val in flop_by_layers.values()])

print(f'total number of floating operations: {total_param_flops}')
print('Number of FLOPs by layer and parameters:')
print("Conv: ", flop_by_layers['conv'])
print("FC: ", flop_by_layers['fc'])
# Output:
total number of floating operations: 672024
Number of FLOPs by layer and parameters:
Conv: {Conv2d(1, 8, kernel_size=(10, 8), stride=(2, 2), padding=(5, 3)): 640008}
FC:   {Linear(in_features=4000, out_features=4, bias=True): 32004}
```

Figure 13: Code to compute the total FLOPS in a forward pass.

```
void loop() {
#ifdef PROFILE_MICRO_SPEECH
  const uint32_t prof_start = millis();
  static uint32_t prof_count = 0;
  static uint32_t prof_sum = 0;
  static uint32_t prof_min = std::numeric_limits<uint32_t>::max();
  static uint32_t prof_max = 0;
#endif  // PROFILE_MICRO_SPEECH

% ------------------------------
% code for ...
% ------------------------------

#ifdef PROFILE_MICRO_SPEECH
  const uint32_t prof_end = millis();
  if (++prof_count > 10) {
    uint32_t elapsed = prof_end - prof_start;
    prof_sum += elapsed;
    if (elapsed < prof_min) {
      prof_min = elapsed;
    }
    if (elapsed > prof_max) {
      prof_max = elapsed;
    }
    if (prof_count % 100 == 0) {
      MicroPrintf("## time: min %dms  max %dms  avg %dms",
                  prof_min, prof_max, prof_sum / prof_count);
    }
  }
#endif  // PROFILE_MICRO_SPEECH

}
```

Figure 14: Profile running time on MCU pseudo-code