Tiny but Mighty: Leveraging Pruning and Quantization in Audio Neural Networks for Compact Device Efficiency

By Maxim Clouser, mmc276@cornell.edu

Preprocessing: Audio Recording and Visualization

The “Yes” and “No” audio recordings are visualized in Figures 1-5 for the time domain, frequency domain, Mel spectrogram, and MFCC spectrogram.

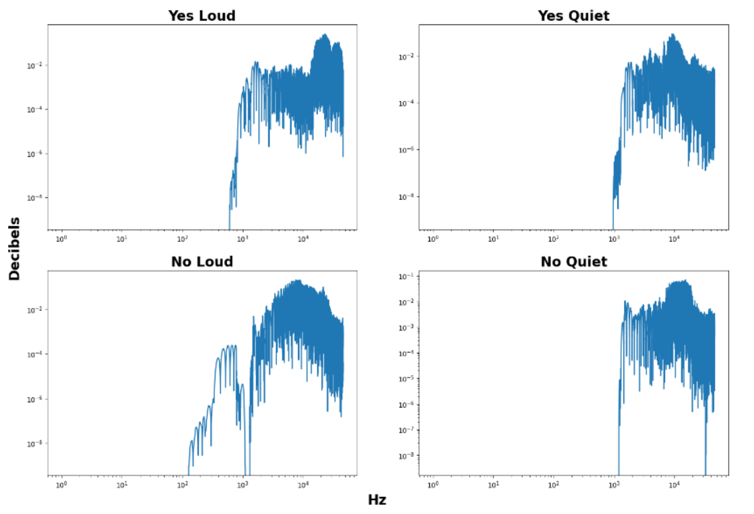


Figure 1- Time domain

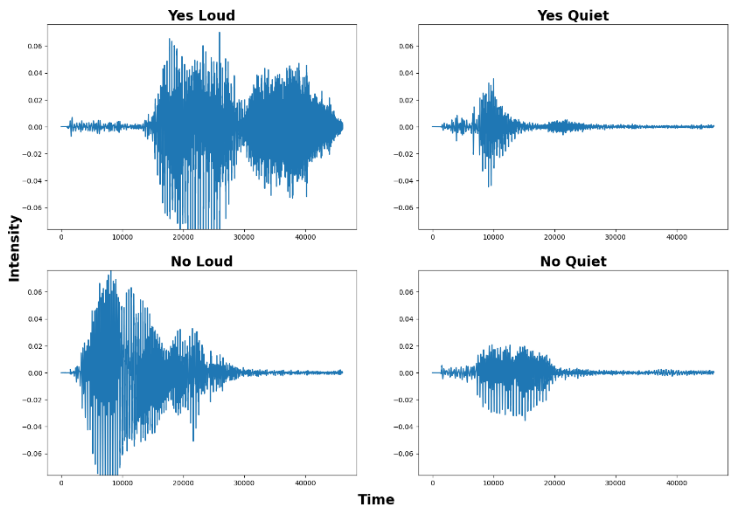


Figure 2- Frequency domain

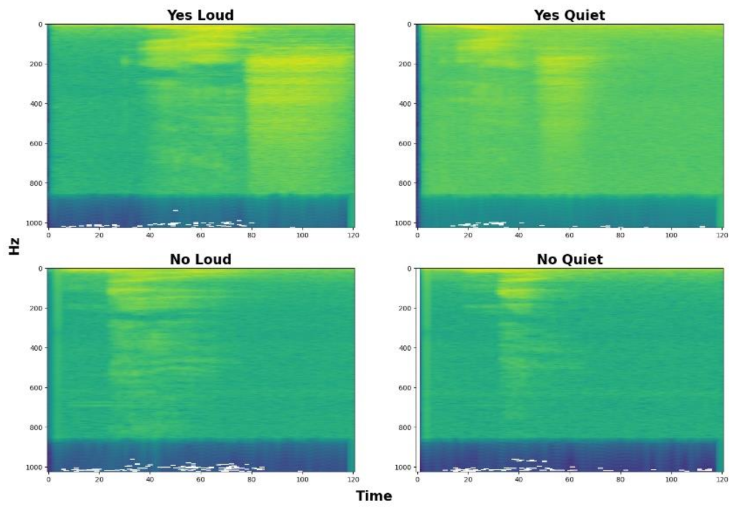


Figure 2- Spectrogram

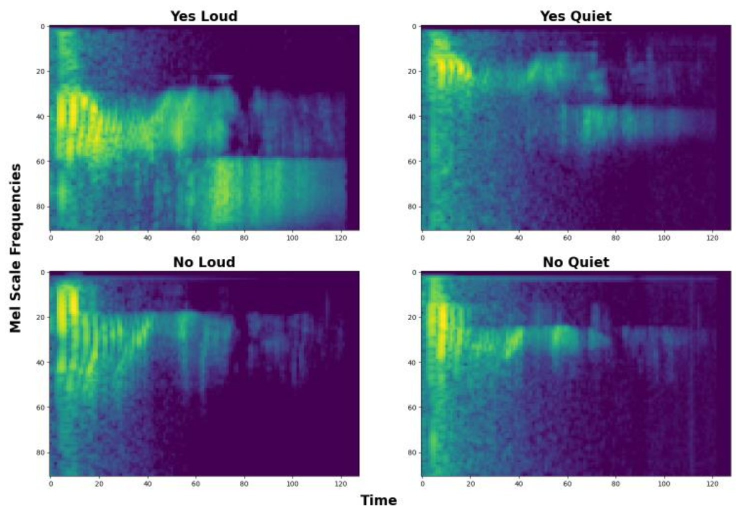


Figure 4- Mel Spectrogram

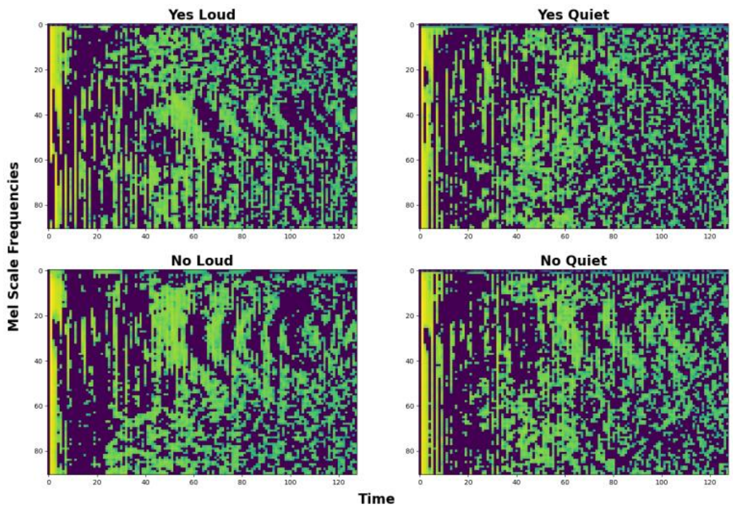


Figure 5- MFCC Spectrogram

Audio must be preprocessed before being passed into a neural network to reduce noise, extract features, and apply normalization. This process allows us to improve the accuracy and generalization ability of a neural network by providing it with a rich view of the audio.

The audio is first converted from analog to digital format, where the analog format of the audio is natural and continuous, and the digital format represents a sampling of the analog waves where the amplitudes are also quantized, resulting in a smaller number of discrete values that represent the audio wave which can be stored efficiently.

Before passing our digital audio to the neural network, we must extract features from the audio that can be learned. Applying a Fast Fourier Transform (FFT) to the time domain provides us with the frequency domain representation, which we can use to create a spectrogram. This spectrogram is a picture that includes an additional dimension to time and frequency, which is the amplitude (or energy) of a particular frequency at a particular time. This rich representation provides a neural network, especially a CNN, with a detailed view of the signal's properties, allowing it to recognize patterns corresponding to specific audio events.

In addition to the spectrogram, a Mel spectrogram uses the Mel scale to map frequencies that better approximate human hearing sensitivity. A Mel filter is applied to the magnitude of the FFT signal which partitions the frequencies into Mel-spaced bins which are then used to create a spectrogram. This Mel scale is created in a way such that sounds of equal distance from each other on this scale are also perceived as equal distance by humans. So, the difference between a spectrogram and Mel spectrogram is that a spectrogram uses a linear frequency scale, and a Mel spectrogram uses frequencies on the Mel scale. The Mel spectrogram is more effective for speech recognition than a linear frequency spectrogram because it emphasizes those frequency bands that are most important for human perception.

The MFCC spectrogram is different in that the energies are logged when being passed through the Mel filter because humans perceive sound intensity logarithmically. These values then undergo a cosine transformation which decorrelates the filter banks. The result of this process is a more compact set of features that mimic the human perception more closely than the linear-frequency and Mel spectrograms. Additionally, timbral characteristics are introduced which would not otherwise be captured.

Model Size Estimation

The estimated flash usage of the TinyConv model is calculated by multiplying its total number of parameters by their size, resulting in an estimated flash usage of 0.07 MB (about 17k FP32 parameters). Therefore, the TinyConv model will only use 7% of the MCU's flash which is 1 MB.

The estimated RAM usage of the TinyConv model is calculated by capturing the input and output sizes of tensors for each layer which are multiplied by the number of bytes used per floating point value. This results in an estimated RAM usage of 117 KB which is 45.7% of the total available RAM on the MCU (256 KB).

The TinyConv model has roughly 17k parameters and 670k FLOPs during a forward pass. This was calculated using forward hooks and counting the number of FLOPs performed for each module type by multiplying the dimensions of tensors by the number of operations being performed on each element. For example, the number of FLOPs for the linear layer is calculated by multiplying the number of weights by two (one add operation and one multiply operation) and adding the number of biases (one add operation). The fully connected later accounted for 32k FLOPs and the conv2d layer accounted for 640k FLOPs.

Comparing the number of FLOPs of TinyConv to other models: This CRNN-based keyword spotting model has roughly 229k parameters and 30 MFLOPs during inference [1]. Furthermore, this paper introduces TC-ResNet, a CNN optimized for key word spotting on mobile devices, which has FLOPs that range between 3 – 13.4M depending on the layer size [2]. The paper compares the TC-ResNet to nine other models (which do not perform as well) using number of FLOPs and parameters, where the listed models have FLOPs that range anywhere from 1.5 – 1950M. The 2D-ResNet8 is a variant of TC-ResNet8 which utilizes 2D convolutions and has 35.8M FLOPs and 66k parameters [2]. The TinyConv model has significantly less FLOPs than the optimized TC-ResNet and the other models listed.

Using the CPU and T4 GPU on Google Colab, the inference runtime of the TinyConv model (batch size = 1) is 2.010ms for CPU and 34.000us on GPU. Note that running inference on the CPU is 59x slower than doing so on GPU.

Training and Analysis

An accuracy of 94% was achieved after training the TinyConv. Figure 6 plots the curves of the training loss and validation accuracy during training.

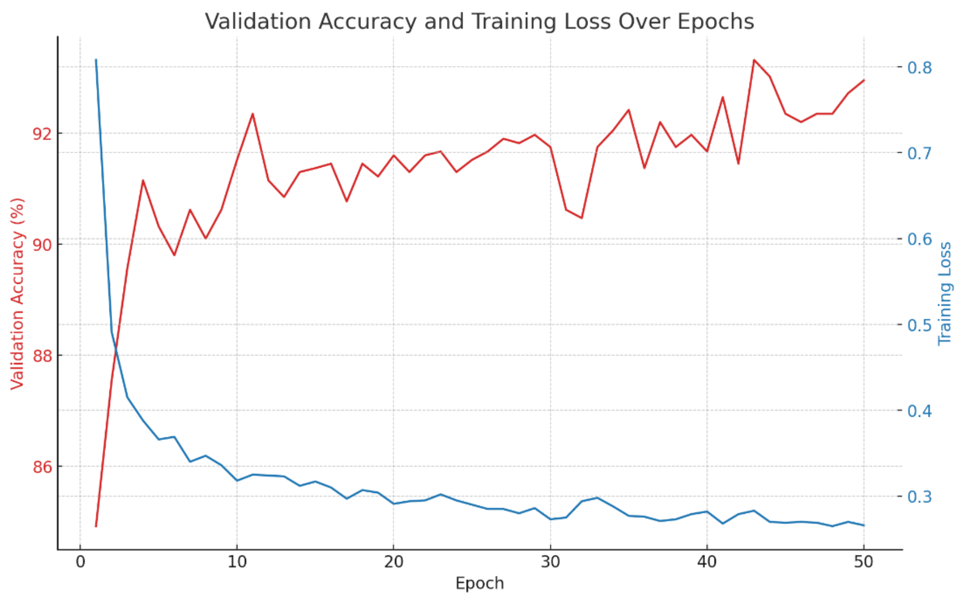
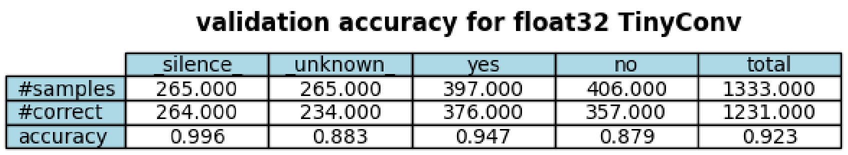


Figure 6- Training loss and validation accuracy when training the TinyConv model to achieve 94% accuracy with 50 epochs and a learning rate of 1e-3.

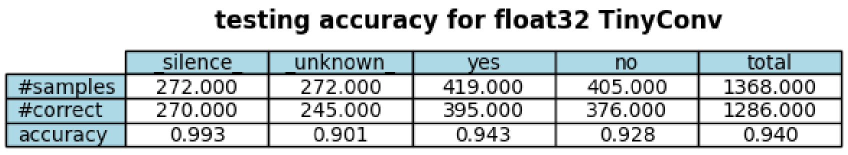
The speech commands dataset created by Google is a collection of one second audio files designed for use in speech recognition tasks. This dataset includes 35 classes of keywords and a collection of files for background noise. Version .02 includes 105,829 audio files and the number of samples for train/validation/test is 84,848 / 9,982 / 4,890 respectively [3].

The TinyConv model was trained using the following four classes: Silence, Unknown, Yes, No. The training of this model used 10,556 samples for the train set, 1,333 for the validation set, and 1,368 for the test set.



validation accuracy for float32 TinyConv

	silence	_unknown_	yes	no	total
#samples	265.000	265.000	397.000	406.000	1333.000
#correct	264.000	234.000	376.000	357.000	1231.000
accuracy	0.996	0.883	0.947	0.879	0.923



testing accuracy for float32 TinyConv

	silence	_unknown_	yes	no	total
#samples	272.000	272.000	419.000	405.000	1368.000
#correct	270.000	245.000	395.000	376.000	1286.000
accuracy	0.993	0.901	0.943	0.928	0.940

Model Conversion and Deployment

Profiling

The total running time of the TinyConv model on the Arduino MCU is 109ms, which is 55x slower than the CPU runtime (2ms) and 3,205x slower than GPU runtime (34us). The breakdown of the preprocessing, neural network, and post-processing runtimes on the Arduino MCU is plotted in Figure 7.

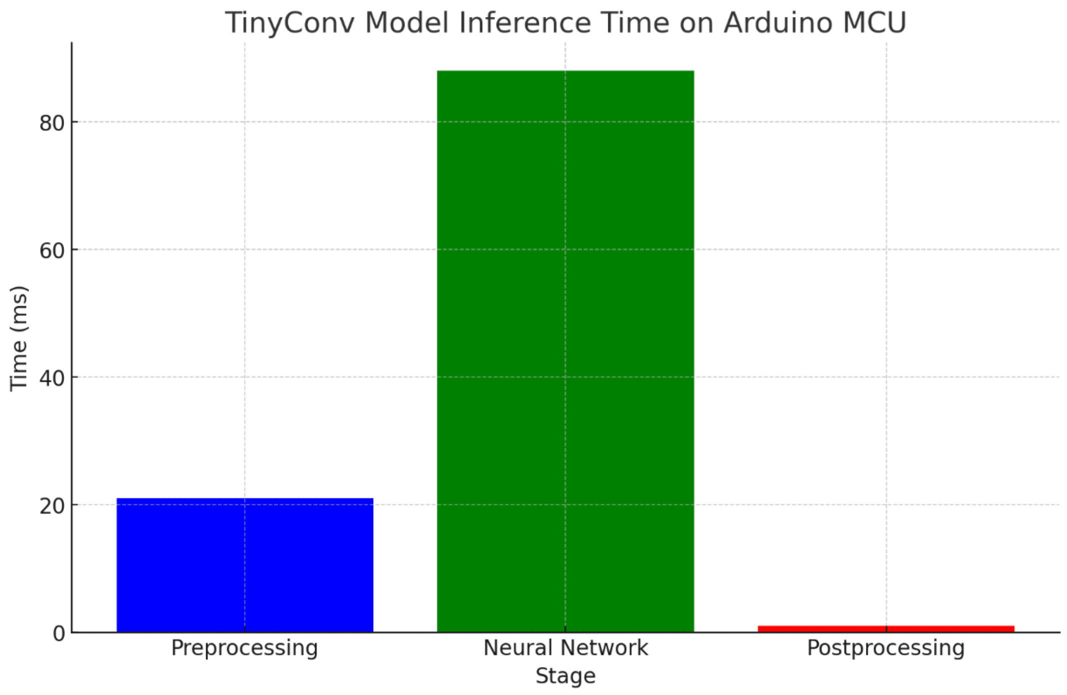


Figure 7- Preprocessing (21ms), neural network (88ms), and post-processing (<1ms) runtimes on the Arduino Nano 33 BLE Sense MCU

More than 50% of the running time is spent in the neural network stage, and roughly 0% of the time is spent during the postprocessing stage.

Accuracy

While testing the TinyConv model with my own voice, a high accuracy (95%) is achieved if I only say the words “yes” or “no”. However, when I test the model with words that sound like “yes” or “no”, such as “finesse” or “elbow” the model almost always incorrectly classifies the words as “yes” or “no” (50% accuracy). Therefore, this model performs very well when distinguishing between these two words (which sound very different from each other) but is not able to distinguish similar sounding words.

Comparing these results with the training and validation accuracy, the results are about the same. The training and validation accuracies were about 93% and I found that the model achieved 95% accuracy in practice (where spoken words are distinguishable).

Quantization-Aware Training

Symmetric and asymmetric quantization were implemented in this section to reduce the size of the model while maintaining as much accuracy as possible. The accuracies of post-training quantization and quantization-aware tuning are compared to understand the impact of reducing the precision of parameters after training versus simulating the precision difference during the training process.

In symmetric quantization, the quantized range is symmetrical around zero. Each value to be quantized is divided by a scale, which is determined using the maximum value and the number of bits. This value is then rounded and clamped to be within the range of bits specified to obtain the final quantized value [4].

In asymmetric quantization, the range does not necessarily need to be symmetric around zero, rather it can be tailored to the minimum and maximum values of the data. After each value is divided by the scale, a zero point is added because the real zero value within this quantization range might not be at the midpoint. The scale is calculated based on the range of the data and the number of bits. Finally, the range is clamped between zero and the number of bits [4].

A straight through estimator (STE) is used for rounding and allows gradients to pass through during the backward pass even though rounding is non-differentiable. A class is implemented for symmetric and asymmetric quantization that applies quantization to a given tensor for a given scale and zero-point. Additionally, a quantization configuration class is used to calculate the scale and zero-point for each type of quantization.

To quantize a given layer, the weights, biases, and activations are each quantized according to the configuration and inserted into a new module before being returned. During quantization-aware tuning, the model is quantized at the beginning of the forward pass, it performs computation using these quantized values, and then the values are converted back (dequantized). The goal of this is to simulate the effects of quantization during training which results in a model that is more robust to the reduced precision when deployed.

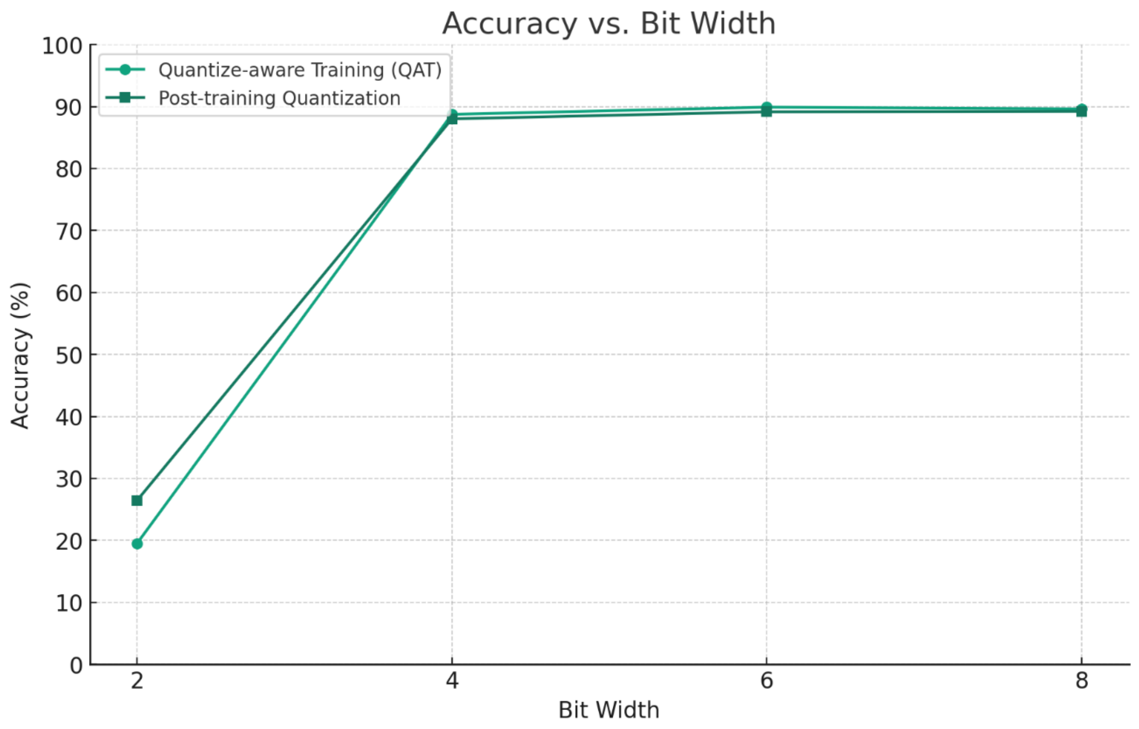


Figure 8- The accuracies for quantize-aware training (QAT) and post-training quantization are compared for bit widths [2, 4, 6, 8].

The impact of the two quantization techniques using various bit widths for precision is shown in Figure 8. For each examined bit width, the QAT model consistently outperforms the post-training quantization model in terms of accuracy. The largest gap in accuracy between these models exists at a bit width of 2. As the bit width increases, this gap shrinks.

This divergence can be attributed to the differences in how each technique approaches quantization. QAT integrates quantization directly into the training process, allowing the model to adapt to the lower precision constraints. This approach enables the model to learn and adjust to the information loss due to the lower precision, resulting in a higher accuracy. In contrast, post-training quantization applies quantization after the model has been trained and does not provide the model the opportunity to adapt to the reduction in precision. Consequently, this leads to a more significant drop in performance, particularly at lower bit widths where precision loss is greater.

Lower bit widths imply less precision and a higher likelihood of information loss. As bit width increases, allowing for more precise representations, the gap in accuracy between the QAT and post-training quantization shrinks. This suggests that while QAT is more robust to lower precision levels, this advantage diminishes as precision increases.

Pruning

In this section, structured and unstructured pruning are applied to the TinyConv model to observe their effects on accuracy. Under both pruning techniques, weights are removed based on a criterion such as the L1 Norm or L2 Norm. Unstructured pruning involves removing individual weights based on their magnitudes resulting in sparse matrices. Structured pruning involves removing entire units of the network, such as channels in a CNN module, based on an aggregate measure which results in a change in the network architecture.

Norms

Criteria for deciding which weights to remove include L1 Norm, L2 Norm, and L-infinity Norm. L1 Norm, also known as Manhattan distance, is the sum of the absolute values of all elements. Using L1 Norm for pruning means to remove the weights with the smallest absolute value which can be beneficial when many of the weights are close to zero and the goal is to enforce sparsity. This technique is less sensitive to outliers compared to L2 Norm.

$$|\mathbf{x}|_1 = \sum_{r=1}^{n} |x_r|.$$

Equation 1- L1 Norm

L2 Norm, also known as the Euclidean Norm, is the square root of the sum of the squares of all elements. This is Norm is more sensitive to outliers compared to the L1 Norm, and using this criterion means to remove weights that have less impact on the output, since larger weights contribute more to the L2 Norm.

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

Equation 2- L2 Norm

Finally, the L-infinity Norm is the maximum absolute value of all elements. This Norm is focused on the largest outlier in terms of its absolute value and results in selecting the weight which has the largest magnitude.

$$|\mathbf{x}|_\infty = \max_i |x_i|.$$

Equation 3- L-infinity Norm

When choosing a Norm to use during pruning, L1 Norm is best for achieving sparse matrices, L2 Norm is more suitable if the goal is to minimize the impact on the output of the network, and L-infinity is useful in scenarios where the largest weights are specifically target for some reason. Therefore, for the purpose of pruning and achieving sparsity, L1 Norm is the best match and is used for the following experiments.

Structured Pruning

Structured pruning is the process of removing groups of weights in the network. Entire units of the network are removed such as channels, resulting in reductions in computation during inference since entire units of the network no longer need to be computed resulting in a simplified network architecture. This pruning technique allows for optimizations at the hardware or software level.

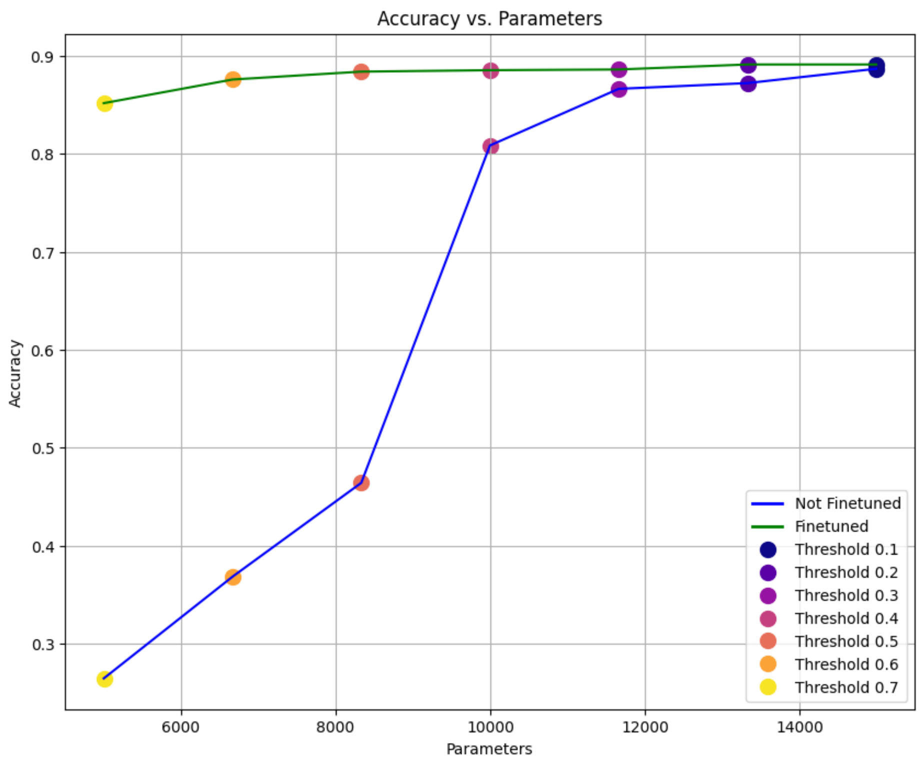


Figure 9- Accuracy vs number of parameters of a structured pruned model using 5 thresholds before and after finetuning with 10 epochs and a learning rate of 1e-4.

The TinyConv model was pruned across convolutional channels using structured pruning from the PyTorch framework [5]. While pruning a model using Torch, the framework keeps track of the original weights and uses a mask and forward hooks to simulate the pruning during the forward pass. Using the "remove" method from the pruning library removes the mask and hooks so all is left is pruned values (where channels that were pruned are set to zero). After pruning the channels, the zeroed channels were removed from the model to reduce size. This simplification of the model is achieved by choosing the nonzero channels from the convolutional layer, calculating the new output size of a convolutional layer with only these nonzero channels, and adjusting the input size of the following fully connected linear layer. The result of this process is a new TinyConv model with a convolutional module that has a smaller number of channels, and the rest of the old model is kept the same (besides the reshaping of the linear layer).

For various thresholds, two models were pruned and simplified, and only one was finetuned (10 epochs with a learning rate of 1e-4). As the threshold increases, more parameters are removed, and removing more parameters reduces accuracy. Here we have a trade-off between size and accuracy, which is displayed in Figure 9 and 10 where accuracy plotted against the number of parameters and flops. Note the similarity in the trend between accuracy vs. flops and accuracy vs. number of parameters. The finetuned model handles the reduction in parameters well, as a small amount of accuracy is sacrificed as the threshold increases and parameters are reduced. The model that was not finetuned has a significant reduction in accuracy starting when 35% - 40% of its channels are pruned. With most of its parameters removed, the pruned and simplified model without fine tuning struggles to accurately predict keywords.

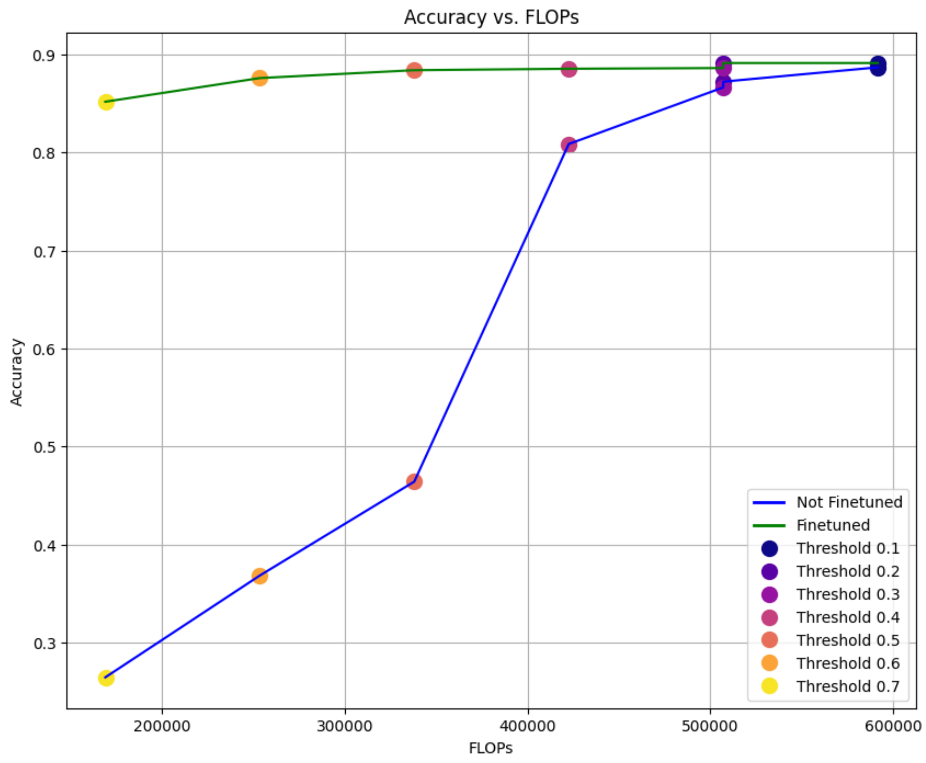


Figure 10- Accuracy vs FLOPs of structured pruned models using 5 thresholds before and after finetuning with 10 epochs and a learning rate of 1e-4.

Increasing the threshold reduces the number of parameters, which causes a reduction in accuracy by a factor that depends on whether the model was provided additional training time after being pruned to adapt to the loss in parameters. Although reducing the number of parameters reduces accuracy, doing so also reduces runtime which can be crucial depending on the use case of the model. Critical machine learning applications need to return outputs as fast as possible with limited memory, in which case a reduction in accuracy may be worth the faster runtime. Figures 11 and 12 show the effect of structured pruning on runtime for CPU and the Arduino Nano 33 BLE Sense MCU.

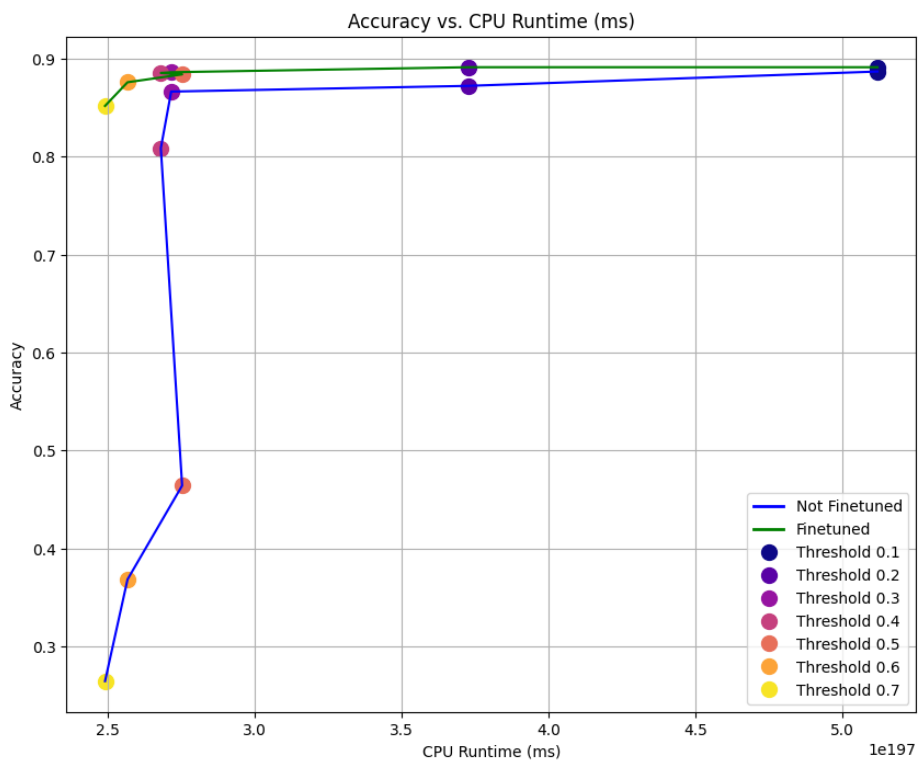


Figure 11- Accuracy vs runtime on the Google Colab CPU for structured pruned models using 5 thresholds before and after finetuning with 10 epochs and a learning rate of 1e-4.

As the threshold is increased, both models perform inference faster. Increasing the threshold increases the number of parameters that are pruned (and removed) which consequently reduces the computation required during a forward pass. The finetuned model performs inference just as fast as the model that was not fine-tuned (as they have the same number of parameters), yet it achieves a much higher accuracy at greater thresholds. Consider the case where 70% of the parameters are pruned. The fine-tuned model achieves an accuracy of about 85% and the model that was not fine-tuned achieves less than 30% accuracy.

Observe the "cliff" that occurs in Figures 9-12 at a threshold of 35% - 40% where the accuracy of the model that was not fine-tuned decreases dramatically. At this point, the CPU and MCU runtime is 2.7ms and 98ms respectively, and the model has 10k parameters with 400k flops. This is the limit of the TinyConv model without finetuning, where removing any more parameters results in significant reduction in accuracy. Therefore, if one were to use this model in practice and the ability to fine-tune is constrained, no more than 35% of the model should be pruned to maintain 90% of the accuracy that was achieved during training. In contrast, if fine-tuning is feasible, up to 65% of the TinyConv model can be pruned while still maintaining strong accuracy.

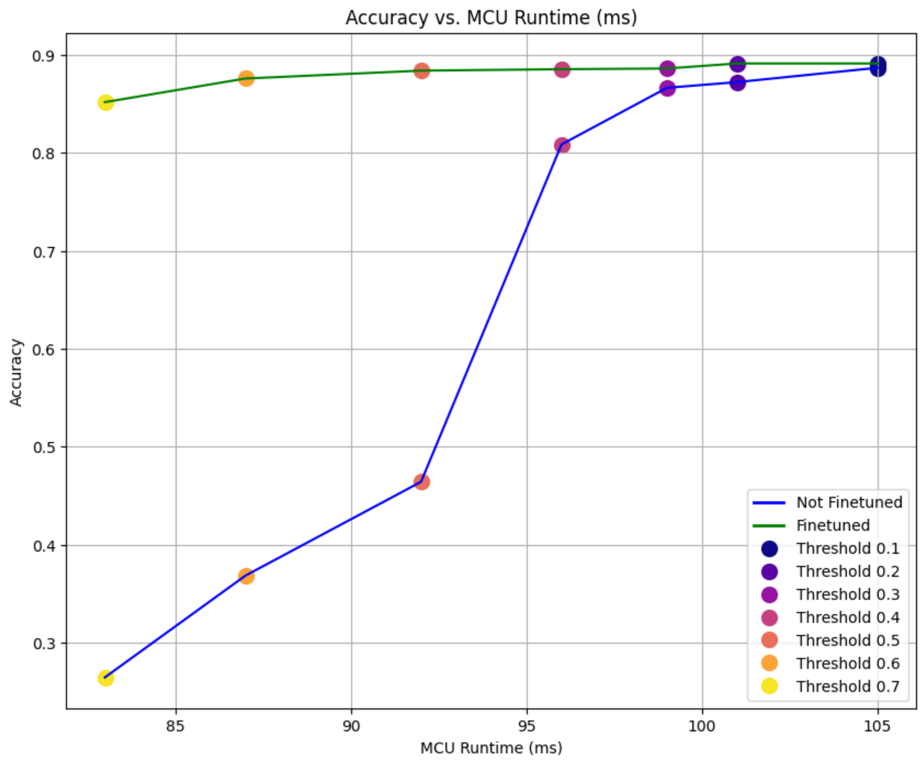


Figure 12- Accuracy vs runtime on the Arduino Nano 33 BLE Sense MCU for structured pruned models using 5 thresholds before and after finetuning with 10 epochs and a learning rate of 1e-4.

Unstructured Pruning

Unstructured pruning is the process of removing individual weights from the network. This can operate on the local level (within a single layer) or at the global level where individual weights are removed across all layers. The resulting network after unstructured pruning has the same architecture but with many weights set to zero, which creates a sparse matrix and reduces the model size for storage. This technique can be beneficial for hardware and software that is optimized for handling sparse matrix operations.

Pruning the TinyConv model using unstructured pruning was done using the L1 unstructured module from the PyTorch pruning library [6]. This process removes the values with the lowest L1 Norm from a given module. In this experiment, the L1 unstructured module was used to prune the convolutional and linear layers. Two TinyConv models are created and pruned using this module, and the “remove” method is not applied to the model that is to be fine-tuned. This effectively “freezes” the pruned weights to zero which prevents them from being updated during the fine-tuning process. Recall that the prune library in PyTorch achieves this by storing a mask and the original weights of the model which are applied during the forward pass using forward hooks stored in the buffer [6]. Removing these using the “remove” method would consequently remove this capability by removing the hooks and mask and leaving the original weights with the mask applied, effectively “unfreezing” the pruned (zeroed) weights.

Although the zeroed parameters are not entirely removed from the model as in structured pruning, unstructured pruning leads to a speed up in computation due to the introduction of sparsity. When the weights are pruned, they are set to zero which creates a sparse matrix. These sparce matrices require less computation since operations involving zero often do not need to be computed. To realize these computational benefits, specialized hardware and software is required which are designed to efficiently handle spare matrices by skipping over the zeros and focusing on non-zero elements. Similarly, sparse representations can reduce the amount of memory required to store the weights (since zeros do not need to be stored) and decrease memory bandwidth. In terms of software, there are libraries and frameworks that support sparse tensor operations to speedup computation such as PyTorch.

The trade-off between accuracy and size can be seen in Figure 13 where the accuracy is plotted against the number of the parameters for two unstructured pruned TinyConv models. One of these models was fine-tuned for 10 epochs with a learning rate of 1e-4, and one was not fine-tuned.

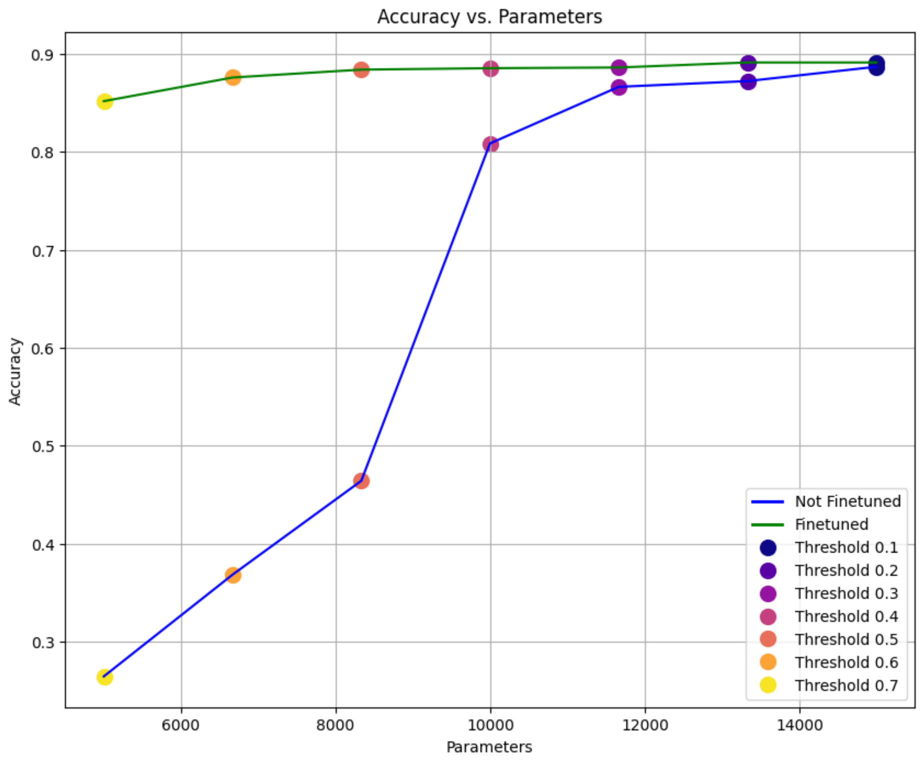


Figure 13- Accuracy vs number of parameters for pruned TinyConv models using unstructured pruning. One model was fine-tuned for 10 epochs with a learning rate of 1e-4 and the other was not fine-tuned. Five thresholds were used to prune each model

Like the results observed from structured pruning, increasing the threshold results in a decrease in accuracy. The model that was not fine-tuned has a significant decrease in accuracy at a threshold of 35% - 40%. This "cliff" phenomenon is consistent with what is seen in structured pruning. It corresponds with the understanding that reducing the number of parameters to a certain extent leads to significant drops in accuracy, making further pruning counterproductive.

The fine-tuned model performs significantly better than the model that was not fine-tuned after pruning as threshold increases. Fine-tuning the models post-pruning mitigates the impact of this accuracy drop. Models that undergo fine-tuning after pruning display a more graceful degradation in accuracy as the pruning threshold increases. This is because fine-tuning allows the network to readjust its remaining parameters, compensating for the loss of the pruned ones. The network essentially relearns to make accurate predictions with its reduced parameters, leveraging the remaining parameters more effectively.

Conclusion

This study on preprocessing audio for neural networks, size estimation, training, model conversion, quantization, and pruning techniques demonstrates the intricate balance between model performance, accuracy, and computational efficiency. The preprocessing stages, including the conversion from analog to digital and feature extraction through various spectrogram techniques, significantly enhance neural network performance for audio recognition tasks. Moreover, the impact of quantization and pruning on model efficiency is substantial. Quantization-aware training (QAT) and both structured and unstructured pruning methods show that model accuracy can be maintained to a considerable degree, even with substantial reductions in computational resources. These techniques explored through the TinyConv model allow for the deployment of more efficient, yet still powerful, models in resource-limited settings. The TinyConv model, with its minimal flash and RAM usage, showcases efficient use of resources while retaining high accuracy, making it ideal for constrained environments like microcontrollers.

References

1. Arık, Sercan Ö., et al. "Convolutional Recurrent Neural Networks for Small-Footprint Keyword Spotting." arXiv preprint arXiv:1703.05390, 2017
2. Choi, Seungwoo, et al. "Temporal Convolution for Real-Time Keyword Spotting on Mobile Devices." arXiv preprint arXiv:1904.03814v2, 2019
3. Hugging Face. (n.d.). Speech Commands dataset. Hugging Face Datasets. Retrieved from https://huggingface.co/datasets/speech_commands
4. Intel Labs. (n.d.). Quantization Algorithms. Distiller. Retrieved from https://intellabs.github.io/distiller/algo_quantization.html
5. "torch.nn.utils.prune.LnStructured — PyTorch 1.10 Documentation." PyTorch, PyTorch, https://pytorch.org/docs/stable/generated/torch.nn.utils.prune.LnStructured.html#torch.nn.utils.prune.LnStructured.prune
6. "torch.nn.utils.prune.l1_unstructured — PyTorch 1.10 Documentation." PyTorch, PyTorch,https://pytorch.org/docs/stable/generated/torch.nn.utils.prune.l1_unstructured.html#torch.nn.utils.prune.l1_unstructured