Assignment A2: Keyword Spotting with MicroControllers

1st William Daugherty-Miller
M Eng. Operations Research and Information Engineering
Cornell Tech
New York City, United States
wld37@cornell.edu

BACKGROUND

In this lab, you will train and deploy an audio processing neural network on your Arduino Tiny Machine Learning Kit. These are six sections in this assignment as outlined below. In addition to the write-up, make sure to submit all of your code in Github Classroom. Note that there are some automated test cases written to check the functionality of your code and they will be used for grading as well. If you fail a test case but think that your answer is also correct, this is okay, just please provide an explanation in your write-up. Note that the first 4 sections provide a simple walkthrough and some short-answer questions, however, most of the marks will be on sections 5 and 6 where you will implement and investigate quantization and pruning in more depth.

1 Preprocessing
2 Model Size Estimation
3 Training & Analysis
4 Model Conversion
5 Quantization
6 Pruning

Note that Colab Pro or Colab Pro+ are highly recommended for this assignment.

ASSIGNMENT

A. Preprocessing: Audio Recording and Visualization

The goal of Part 1 is to become familiar with common audio preprocessing techniques. In this module, you will have an opportunity to record and visualize your own sound bites. Your audio will be plotted in the time domain and frequency domain and as a spectrogram, a mel spectrogram, and lastly a mel frequency cepstral coefficients (MFCC) spectrogram. You will not need to write any code for this part, but do take the time to appreciate how signal transformations enable machine learning algorithms to better extract and utilize features.

Hand in:

- Plot the time domain, frequency domain, spectrogram, mel spectrogram, and MFCC spectrogram of your audio.
- Comment on why we preprocess input audio sending it through a neural network.
- What is the difference between the different spectograms that you plotted? Why does one work better than the other?

B. Model Size Estimation

MCUs have limited memory and processing power. Before thinking of deploying a model to the MCU we need to check its estimated RAM and flash usage. Furthermore, we will benchmark the speed of the DNN on desktop or server-grade CPU and GPU to compare to the MCU.

Hand in:

- Find the estimated flash usage of your DNN. Explain how you arrived at your answer. What percent of your MCU's flash will this model use?
- Find the estimated RAM usage of your DNN when operating in batch size = 1. Explain how you arrived at your answer. What percent of your MCU's flash will this model use?
- Find the number of FLOPS of your model during a forward pass (or inference). Compare this number to another speech model that you find in the literature. Search with the words "keyword spotting" or "wake word detection" and you should be able to find DNNs that implement similar functionality.
- Find the inference runtime of your DNN (batch size = 1) on your Colab cpu and gpu to later compare to the MCU.

C. Training & Analysis

Now that we have verified that the model could fit on our MCU, we need to train the model. Please go through the training notebook and understand each step. There is no code that you need to write in this step—it's all there. The result of this notebook is a DNN model checkpoint that you will use in the following notebooks.

Hand in:

- Report the accuracy that you get from your model.
- Plot curves of the training and validation accuracy during training.
- Comment on the speech commands dataset. How many classes of keywords are supported, and how many train/test/validation samples are there?

D. Model Conversion and Deployment

Our model has been trained using the PyTorch framework, but the MCU only supports TFLite Micro. While converting to a TFLite model, you will quantize the model to reduce its memory and computation. You will use the output of this notebook to deploy the model to your Ardino Nano 33 BLE. Find detailed instructions and notes on how to install the Ardunio SDK and libraries by following this tutorial. You do not need to worry about running the "Blink" example but feel free to do so to verify your installation and board connection. Next, deploy your model onto your TinyML Kit by following these instructions.

Hand in:

- Profile running time and plot the breakdown between preprocessing, neural network, and post-processing on arduino. Compare to the CPU and GPU numbers you got in part 2. How much slower is the MCU?
- Record accuracy (out of 10 or more trials) when you try the model with your own voice. Comment on any discrepancy between training accuracy, validation accuracy, and in-field test.
- Extra Credit (1 bonus mark): Repeat the above question when training with your own keyword. State what keyword you used and how you trained your model with that extra keyword. What accuracy do you achieve for that keyword. Submit a notebook that performs training with this extra keyword.

E. Quantization-Aware Training

As we explored in the last two sections, the size of your model is important when dealing with on-device applications. Part 3 implemented a full precision (float 32) model which was quantized after training. In this section, we will use quantization-aware training to try and improve model accuracy during training. Note that we will perform this investigation in PyTorch and not in TFLite.

Hand in:

- Code used to finish implementing quantization-aware training (QAT). Provide a very brief explanation of your implementations of the missing functions in notebook 5.
- Plot accuracy vs. bit-width for:
  - post-training quantization between 2 and 8 bits.
  - quantization-aware training between 2 and 8 bits.
- Comment on the impact of post-training quantization versus quantization-aware training.
- Extra Credit (1 bonus mark): Repeat the above steps with minifloat quantization. You can select the exponent and mantissa bit widths as you see fit. Explain your choices and submit your code.

F. Pruning

Pruning is another machine learning tactic that can help reduce model size and increase computational efficieny by removing unused parameters in neural networks. Pruning can remove groups of weights in structured pruning or individual weights in unstructured pruning. For this section, you will implement both structured and unstructured pruning, and measure the impact on accuracy.

Hand in:

- Code used to implement unstructured and structured pruning, including a fine-tuning step after pruning to regain accuracy. Note that you may use PyTorch's native pruning library or implement your own.
- For unstructured pruning:
  - Plot the accuracy vs. number of parameters at different pruning thresholds. Please choose at least 5 pruning thresholds and plot two curves, one with finetuning and one without. You want to aim to plot the "cliff", after which accuracy drops.
  - Comment on how we can utilize unstructured pruning to speed up computation.
  - What is the difference between L1 norm, L2 norm and L-infinity norm. Which one works best with pruning?
- For structured pruning (channel pruning) plot the following:
  - Accuracy vs. parameters. Please choose at least 5 pruning thresholds and plot two curves, one with finetuning and one without.
  - Accuracy vs. FLOPs. Note that you need to elliminate the pruned channel from the model to compute FLOPS correctly and to perform the following two measurements.
  - Accuracy vs. runtime on desktop CPU.
  - Accuracy vs. runtime on MCU (Yes, you need to deploy your pruned model onto the MCU for this step).

PREPROCESSING: AUDIO RECORDING AND VISUALIZATION

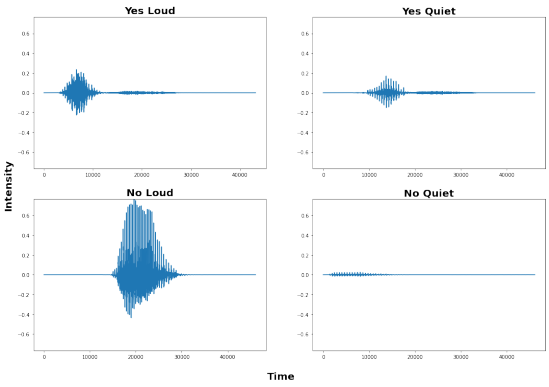

Fig. 1: Time Domain Plots

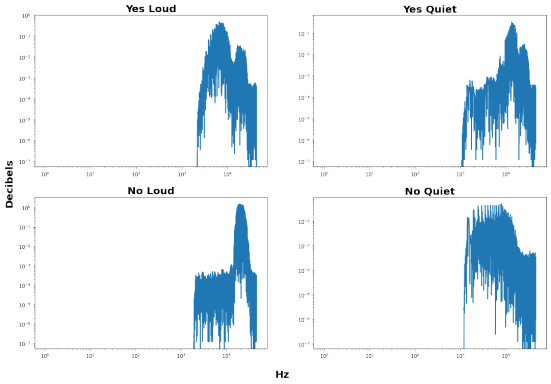

Fig. 2: Frequency Domain Plots

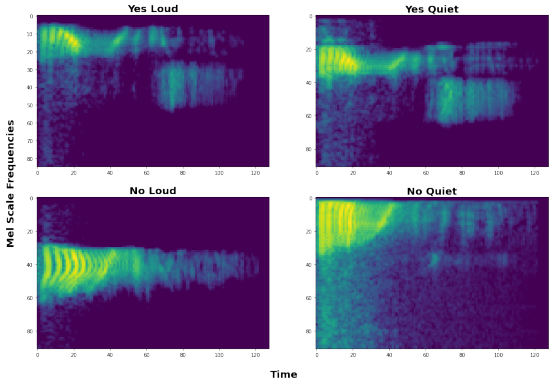

Fig. 4: Mel Spectrogram Plots

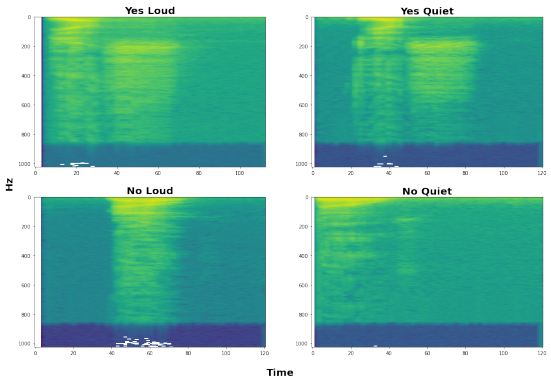

Fig. 3: Spectrogram Plots

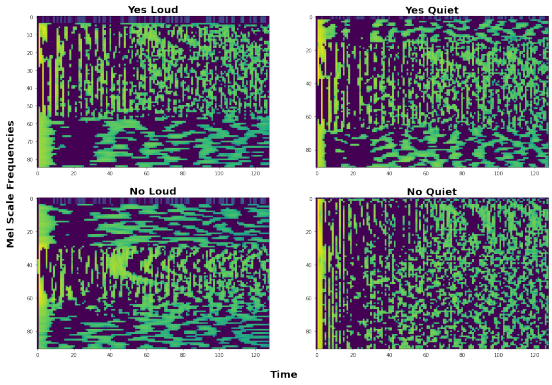

Fig. 5: MFCC Plots

As stated in the notebooks, one of the first steps in preparing the data for analysis is to preprocess it into a numeric format that can be utilized by a neural network. This involves transforming the raw audio data into a form that can be easily processed by machine learning algorithms. Once this is completed, we can then look at a variety of plotting techniques.

One common technique for preprocessing audio data is to convert it into a spectrogram, which is a visual representation of the frequencies present in the audio over time. This is accomplished by applying a mathematical transformation called the Fourier transform, which separates the audio signal into its component frequencies. The resulting spectrogram can be thought of as a picture of the audio, with similar frequencies represented as contiguous regions of the image.

In order to make the spectrogram compatible with neural networks, particularly convolutional layers that are commonly used for image processing, it is important to rearrange the data so that similar frequencies are close together and can be processed together. This is typically done by rearranging the rows of the spectrogram so that they are in ascending order of frequency, and then dividing the resulting matrix into smaller blocks or patches that can be processed in parallel.

Another important step in preprocessing audio data is to convert the spectrogram into a format that is based on the scale of human hearing. The Mel scale is a logarithmic transformation that maps the frequency spectrum into a more intuitive scale for use in a model centered around human hearing. This allows us to more accurately capture the perceptual characteristics of the sound, as our hearing system is more sensitive to certain frequency ranges than others.

Finally, the Mel spectrogram can be further compressed and represented using Mel-frequency cepstral coefficients (MFCCs), which are derived from a discrete cosine transform of the log Mel spectrum. This technique is particularly effective at capturing the timbre of the sound, which is a critical aspect of many audio processing applications such as speech recognition or music classification.

In summary, preprocessing audio data involves several steps, including transforming the raw audio into a spectrogram, converting it into a format based on the scale of human hearing, and compressing it using MFCCs to effectively represent timbre (think 2D matrix). These techniques are essential for effectively analyzing and processing audio data using neural networks.

Model Size Estimation

In order to implement such a model, it is important to consider the computational resources required for its deployment. The number of trained parameters and size of the parameters will determine the amount of flash memory required, while the forward size and size of parameters will determine the amount of RAM needed.

We observe that the estimated flash usage is roughly 6.7% of the MCU's flash and 3.1% of the RAM. This model has a total of 0.016552* 1e6 trained parameters, with each parameter taking up 4 bytes of memory. The number of FLOPs required for this model is 660004, or 0.66 MFLOPs.

Compared to similar models found in literature, such as the 30M in Convolutional Recurrent Neural Networks for Small-Footprint Keyword Spotting and the 4.1 M to 57 M in A New Lightweight CRNN Model for Keyword Spotting with Edge Computing Devices, this model requires significantly fewer resources.

We also note that the inference time for CPU is 209.000us and for the GPU it is 43.000us. Next steps for this comparison would be to map the time to the following categories preprocesing, NN and postprocessing.

Training & Analysis

We see that a final recorded training accuracy of 89.94% and a final validation accuracy of 91.30%. Interestingly, we observe that most of the recorded accuracies do fall in this range as we can observe by the plotted curves.

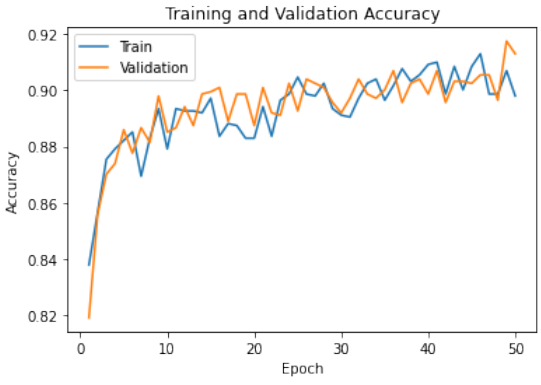

Fig. 6: Train vs Validation Accuracy

Additionally, we can observe a breakdown of the speech commands dataset (broken up into train, test and validation). We note even though there are roughly 30 keywords supported, our experiments support only 4 ('Yes','No','Unknown' and 'Silence'). We also observe that there the following amounts for the sample sets: Train size: 10556 Val size: 1333 Test size: 1368.

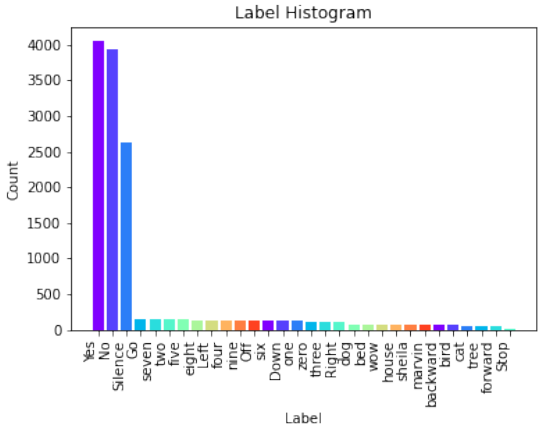

Fig. 7: Train vs Validation Accuracy

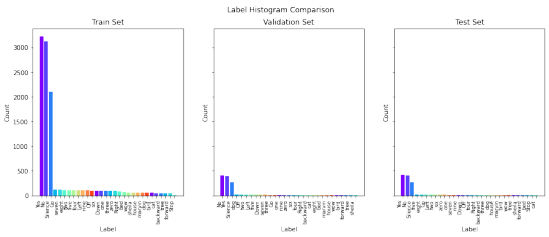

Fig. 8: Train vs Validation Accuracy

[**bonus**] We can then look at what happens when we support another keyword ("'left' since it is one of the most common words found in all three sample sets). Shown below is the results of this which drops the accuracy by 1 percent however if this was scaled to include all the keywords this may results in a large drop in accuracy.

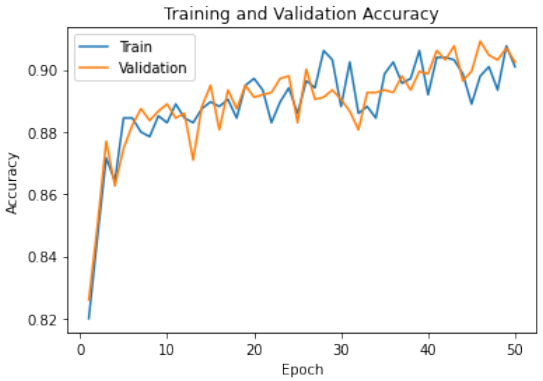

Fig. 9: Train vs Validation Accuracy

A final note about these curves is that there may be an overfitting error however this would need a further examination to confirm.

Model Conversion and Deployment

When we run the model on the MCU, we observe a significant decrease in speed when compared to GPU (43 us)

and the CPU (209 us). When timestamps were printed out, we observed that on average the model took 109 ms which we can see is quite significant.

Below is a breakdown of breakdown between preprocessing, neural network, and post-processing on the arduino. It is interesting to note that post-processing takes no time at all.

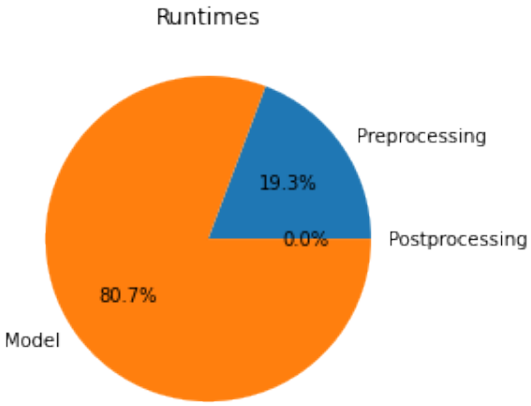

Fig. 10: Train vs Validation Accuracy

After looking at the breakdown, the model was utilized on 20 different samples

Keyword	Trial Count	Correct labels	Percentage
yes	10	8	80%
no	10	9	90%

TABLE I: In-Field Test

As we can see, the in-field test lines up with what we saw in the part 3. It should be noted that I tried adversarial data for one where I went 'sss' which the program classified as yes.(Note this technique was talked about with Naman)

Quantization-Aware Training

For filling in the missing functions, I will briefly explain the thought process.

The first function is the ste round function which just needed to return the grad output in tensor. The next was the linear quantization function that takes an input, scale, and zero point, and outputs shifted values. Next, we worked on the symmetric and asymmetric quantization functions, which incorporate a bit scale using the torch clamp function with a range(-(2 ** (k - 1)), 2 ** (k - 1) - 1) for symmetric function and (0, 2 ** k - 1) for the asymmetric. Additionally, some helper functions were modified to assist with the data input, extraction, quantization parameters based on that data, and execution of t the quantization process.

In the figure below, we see the difference between QAF and QAT. As we can see there is very little difference when the bit-width is lower, however when around bit-width 6 we start to see the QAT model perform better. This is slightly surprising and I am curious if there a reason for this since one is more fine-tuned.

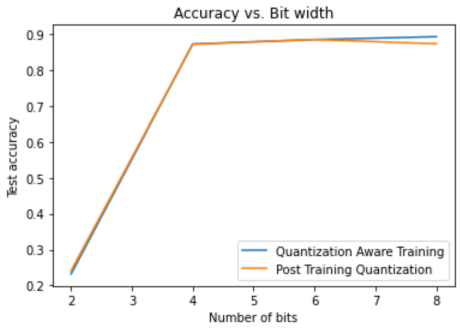

Fig. 11: Train vs Validation Accuracy

Pruning

To enhance the efficiency and compactness of a model, we can utilize pruning techniques, whether structured or unstructured. By reducing the number of channels, both the number of floating-point operations (FLOPs) and parameters decrease proportionally, which results in faster processing times and a more manageable model size. This strategy yields significant performance improvements by minimizing FLOPs and parameter count, thus optimizing the overall architecture of the model. Consequently, the reduced FLOPs lead to swifter run-times, while the decreased number of parameters leads to a lower model size, making it more feasible for deployment on resource-constrained systems.

Turning to unstructured pruning, we observe that there is very little difference before and after fine-tuning. This may indicate that a different implementation is needed or that structured offers better improvement.

After consulting various online resources and textbooks, we can establish the definitions of different norms with greater mathematical rigor:

The L1 norm is defined as the sum of the absolute values of the elements in a vector, given by:

$$|\boldsymbol{x}| \, 1 = \sum i = 1^n \, |x_i|$$

The L2 norm is the square root of the sum of the squared magnitudes of the vector elements, given by:

$$|\boldsymbol{x}| \, 2 = \sqrt{\sum i = 1^n x_i^2}$$

The L infinity norm is defined as the maximum absolute value of the elements in the vector, given by:

$$|\boldsymbol{x}| \, \infty = \max 1 \leq i \leq n \, |x_i|$$

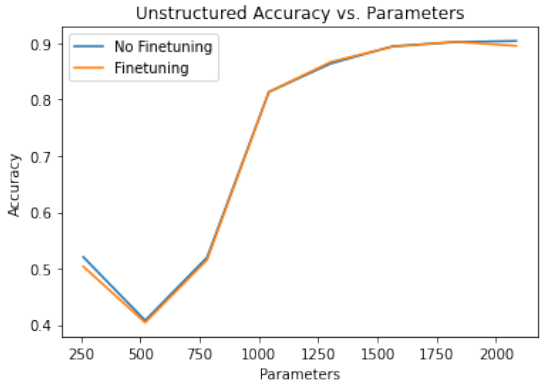

Fig. 12: Unstructured Pruning Accuracy vs Parameters

Mathematical Exaplantion

The L1 norm is widely used in pruning techniques due to several advantages it offers. One of the main benefits of using L1 norm is that it promotes sparsity, allowing for the reduction of model size. By applying the L1 norm to a set of weights or channels in a neural network, some of the weights or channels can be shrunk to zero, effectively removing them from the model. This results in a sparse model that can be faster and more memory-efficient to train and evaluate. Additionally, the L1 norm encourages weight sharing in neural networks, which can lead to better generalization and robustness of the model. By grouping together similar weights, the model becomes less sensitive to noise and can perform well even in the presence of perturbations.

For structured, we can observe the following graphs and note that as the pruning threshold is increased, the models perform worse (This is what I believe to be ”the cliff” mentioned in the assignment).

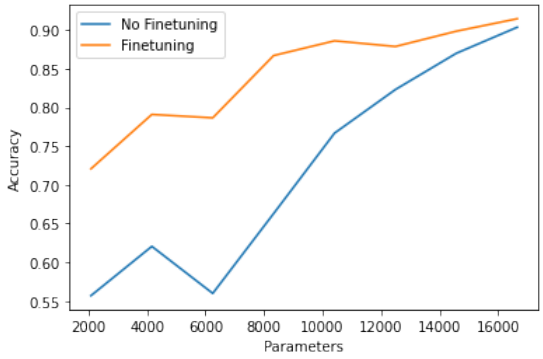

Fig. 13: Structured Pruning Accuracy vs Parameters

DICLAIMER: I am using my remaining late days for this assignment. In the spirit of transparency, I worked with, talked to and helped Ajan, Anastasia, Nick, Naman and Bill. Additionally, I used code from stack overflow

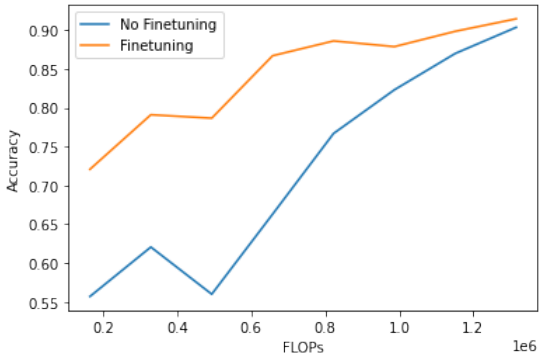

Fig. 14: Structured Pruning Accuracy vs FLOPs

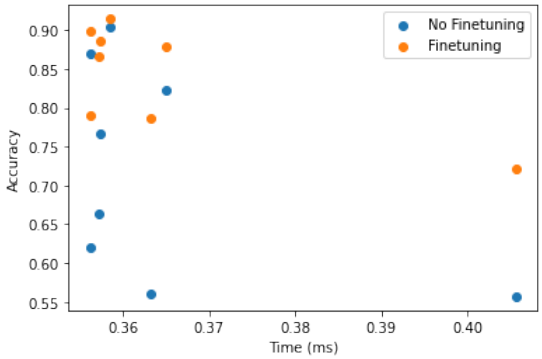

Fig. 15: Structured Pruning Accuracy vs Time (CPU)

and had improvements and suggestions given by Tabine.ai, Chatgpt, Github Copilot and balckbox. Additionally, I consluted the textbook and other articles for what I should approximately be seeing in my code.

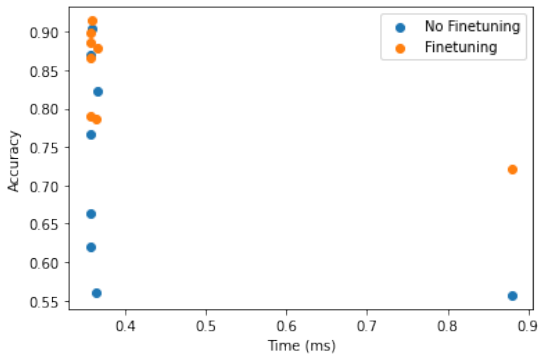

Fig. 16: Structured Pruning Accuracy vs Time (MCU)