EE 5545 Assignment 2

Ethan Glaser (eg492)

I. Problem 1

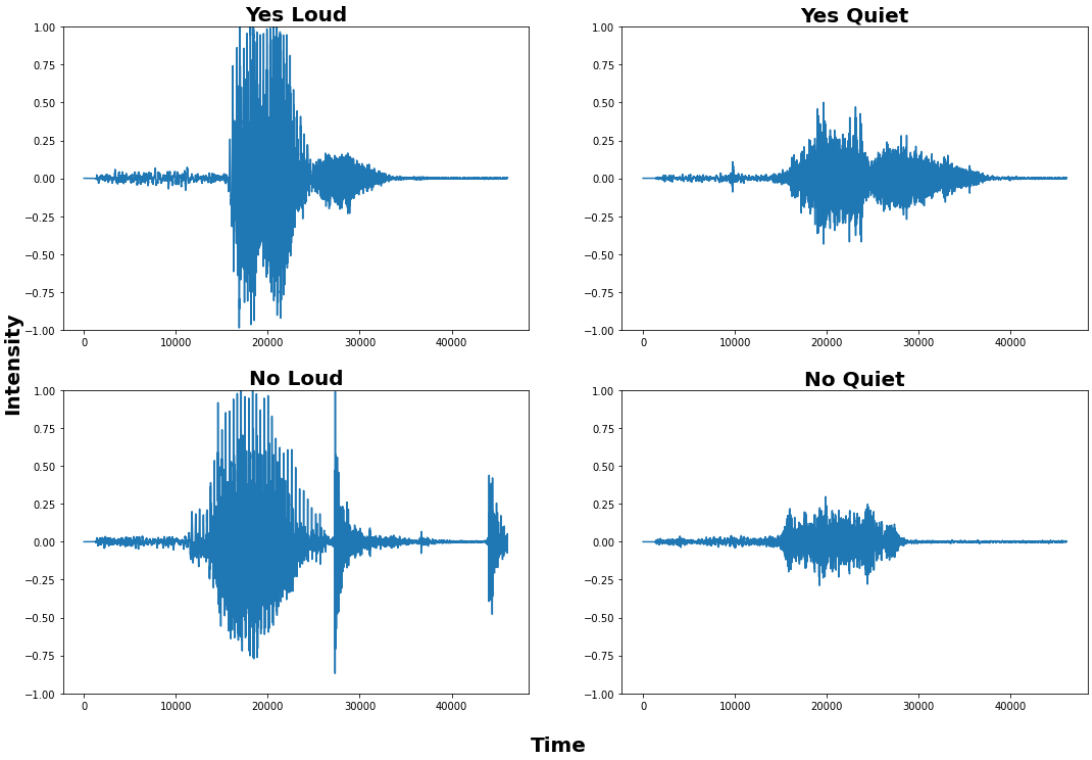

Figure 1: Time domain of recorded audio

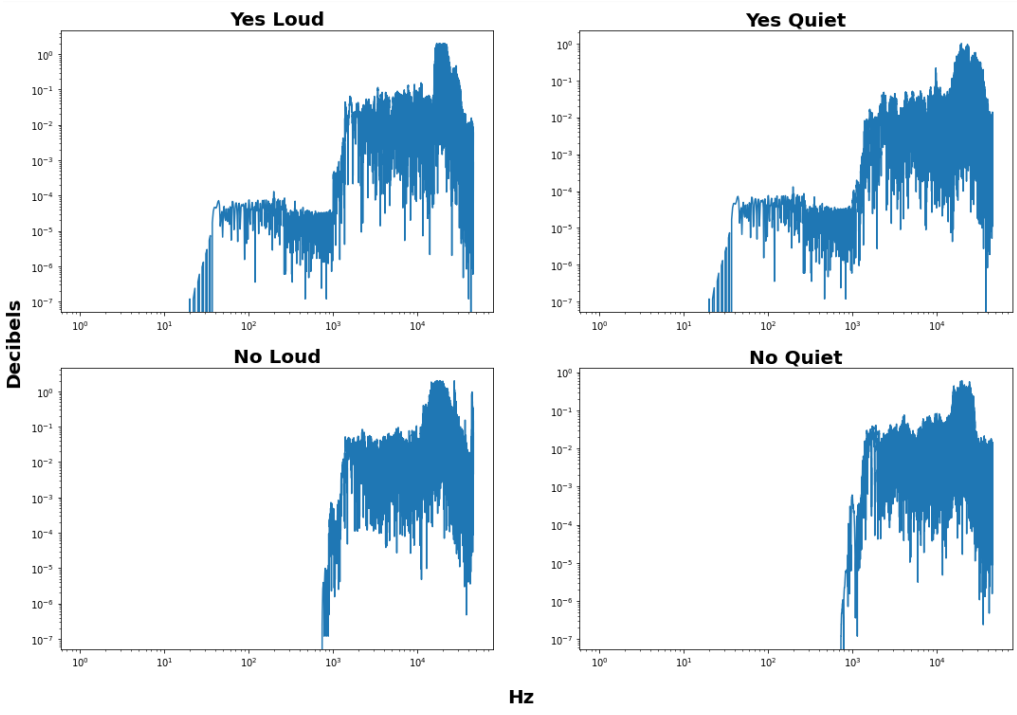

Figure 2: Frequency domain of recorded audio

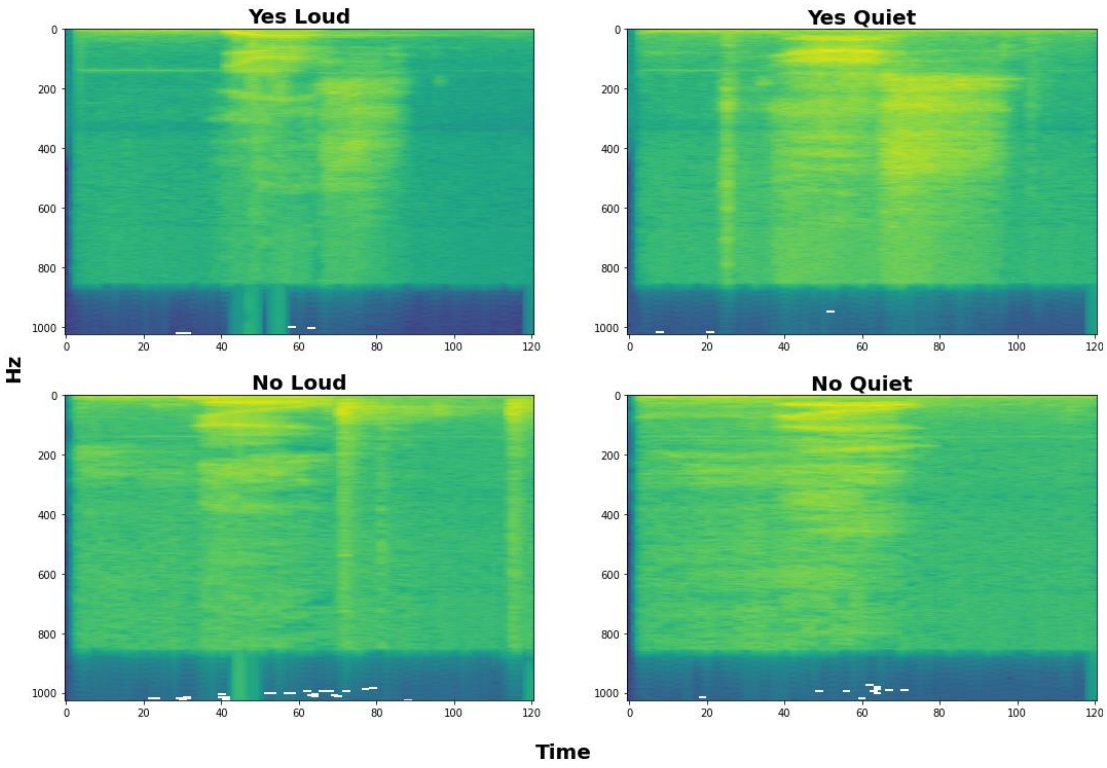

Figure 3: Spectrogram

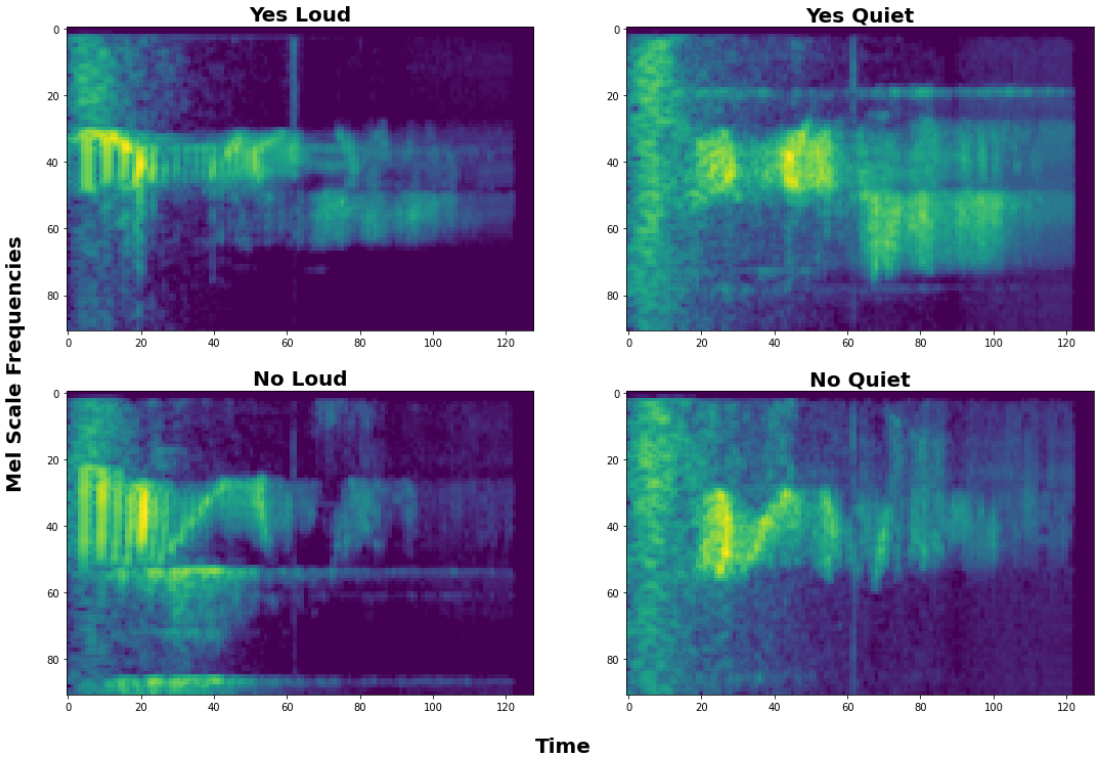

Figure 4: Mel Spectrogram

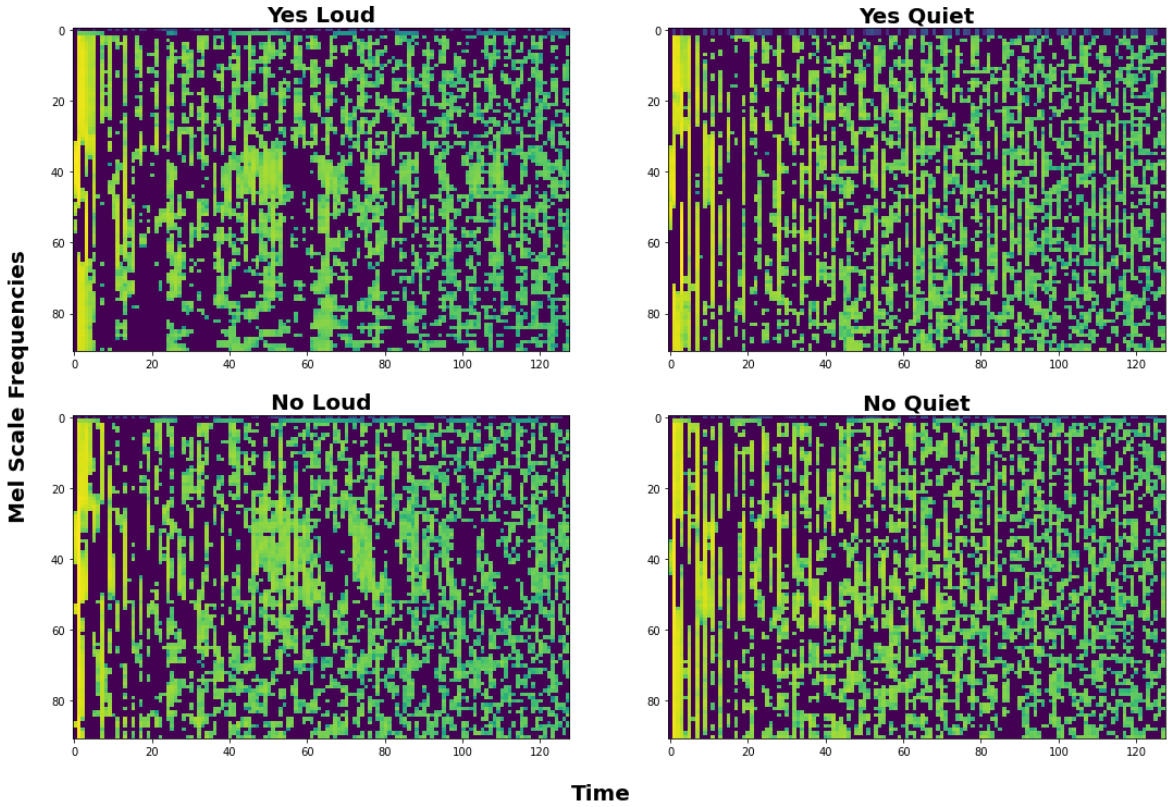

Figure 5: MFCC

We must preprocess the audio into a numeric format that can be utilized by a neural network. Frequencies must be isolated from the raw audio over time to create a spectrogram, which is like a picture of the audio, and is in a format that is compatible with neural networks, specifically convolutional layers, like a typical image where similar frequencies are close together and can be processed together. The Mel scale converts the original spectrogram information into a format that is based on the scale of human hearing using a logarithmic transformation, a more intuitive scale for use in a model centered around human hearing. The MFCC is a more compressed version of the log of the Mel spectrogram, with an additional discrete cosine transformation, which is effective at representing timbre which makes it useful in audio processing.

II. Problem 2

Flash = number of trained parameters * size of parameters = 0.016552* 1e6 * 4 bytes = 0.067 MB, or 6.7% of the MCU's flash memory.

RAM = forward size * size of parameters = 0.007856 * 1e6 * 4 bytes = 0.031 MB, or 3.1% of the MCU's flash memory.

```
total number of floating operations: 660004
Number of FLOPs by layer and parameters:
Conv:  {'bias': 4000, 'weight': 640000}
FC:    {'bias': 4, 'weight': 16000}
```

Figure 6: Number of FLOPs in model

Name	Self CPU %	Self CPU	CPU total %	CPU total	CPU time avg	# of Calls	Total KFLOPs
aten::zeros	0.15%	218.000us	0.16%	227.000us	227.000us	1	--
aten::empty	0.02%	22.000us	0.02%	22.000us	5.500us	4	--
aten::zero_	0.00%	2.000us	0.00%	2.000us	2.000us	1	--
model_inference	0.41%	587.000us	99.84%	142.694ms	142.694ms	1	--
aten::reshape	0.05%	72.000us	0.07%	94.000us	47.000us	2	--
aten::_reshape_alias	0.02%	22.000us	0.02%	22.000us	11.000us	2	--
aten::conv2d	0.03%	37.000us	85.66%	122.428ms	122.428ms	1	640.000
aten::convolution	0.03%	47.000us	85.64%	122.391ms	122.391ms	1	--
aten::_convolution	0.04%	52.000us	85.60%	122.344ms	122.344ms	1	--
aten::mkldnn_convolution	84.00%	120.053ms	85.57%	122.292ms	122.292ms	1	--
aten::as_strided_	1.56%	2.229ms	1.56%	2.229ms	2.229ms	1	--
aten::relu	0.04%	53.000us	0.27%	388.000us	388.000us	1	--
aten::clamp_min	0.23%	332.000us	0.41%	584.000us	292.000us	2	--
aten::linear	0.08%	116.000us	10.93%	15.618ms	15.618ms	1	--
aten::t	0.03%	40.000us	0.05%	74.000us	74.000us	1	--
aten::transpose	0.02%	32.000us	0.02%	34.000us	34.000us	1	--
aten::as_strided	0.00%	4.000us	0.00%	4.000us	2.000us	2	--
aten::addmm	7.17%	10.241ms	10.79%	15.428ms	15.428ms	1	32.000
aten::expand	0.03%	42.000us	0.03%	44.000us	44.000us	1	--
aten::copy_	3.60%	5.142ms	3.60%	5.142ms	5.142ms	1	--

Self CPU time total: 142.921ms

Name	Self CPU %	Self CPU	CPU total %	CPU total	CPU time avg	Self CUDA	Self CUDA %	CUDA total	CUDA time avg	# of Calls
void gemv2T_kernel_val<int, int, float, float, float...	0.00%	0.000us	0.00%	0.000us	0.000us	37.000us	63.79%	37.000us	37.000us	1
void implicit_convolve_sgemm<float, float, 1024, 5, ...	0.00%	0.000us	0.00%	0.000us	0.000us	12.000us	20.69%	12.000us	12.000us	1
void at::native::unrolled_elementwise_kernel<at::nat...	0.00%	0.000us	0.00%	0.000us	0.000us	3.000us	5.17%	3.000us	3.000us	1
void at::native::vectorized_elementwise_kernel<4, at...	0.00%	0.000us	0.00%	0.000us	0.000us	2.000us	3.45%	2.000us	2.000us	1
Memcpy DtoD (Device -> Device)	0.00%	0.000us	0.00%	0.000us	0.000us	2.000us	3.45%	2.000us	2.000us	1
void (anonymous namespace)::softmax_warp_forward<flo...	0.00%	0.000us	0.00%	0.000us	0.000us	2.000us	3.45%	2.000us	2.000us	1
cudaEventRecord	0.93%	15.000us	0.93%	15.000us	15.000us	0.000us	0.00%	0.000us	0.000us	1
cudaLaunchKernel	96.96%	1.564ms	96.96%	1.564ms	312.800us	0.000us	0.00%	0.000us	0.000us	5
cudaMemcpyAsync	1.43%	23.000us	1.43%	23.000us	23.000us	0.000us	0.00%	0.000us	0.000us	1
cudaOccupancyMaxActiveBlocksPerMultiprocessorWithFla...	0.19%	3.000us	0.19%	3.000us	3.000us	0.000us	0.00%	0.000us	0.000us	1
cudaDeviceSynchronize	0.50%	8.000us	0.50%	8.000us	8.000us	0.000us	0.00%	0.000us	0.000us	1

Self CPU time total: 1.613ms
Self CUDA time total: 58.000us

Figures 7 & 8: Runtime breakdown of CPU and GPU

The model will require 6.7% of the MCU's flash and 3.1% of the RAM, as shown in the above equations. The number of FLOPs is 660004, or 0.66 MFLOPs, with the breakdown shown in Figure 6. This is significantly fewer than similar models, such as the 30M in *Convolutional Recurrent Neural Networks for Small-Footprint Keyword Spotting* (https://arxiv.org/ftp/arxiv/papers/1703/1703.05390.pdf) and the 4.1 M to 57 M in *A New Lightweight CRNN Model for Keyword Spotting with Edge Computing Devices* (https://easychair.org/publications/preprint_open/r92V). The CPU inference runtime of the model is approximately 143 milliseconds compared to just 58 microseconds for the GPU.

III. Problem 3

Test Acc = 89.91% Test Acc = 89.18%

Figure 9: Test accuracies obtained from two different instances of training the model

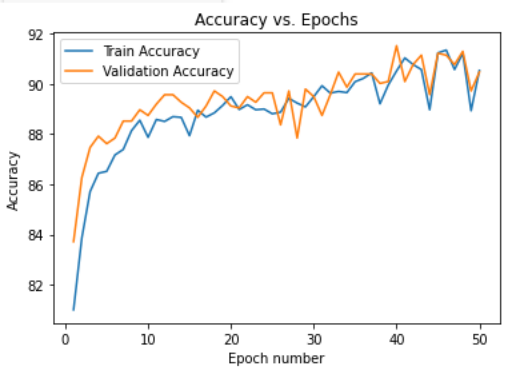

Figure 10: Train and validation accuracies as a function of epochs

```
Train size: 10556 Val size: 1333 Test size: 1368
```

Figure 11: Size of validation, train, and test sets

Figure 9 shows 2 examples of models trained, suggesting that the accuracy is in the 89-90% range. Figure 10 shows the train and validation accuracy during training over the course of 50 epochs. Figure 11 gives some detail on the speech commands dataset, with a breakdown of the size of the train, test, and validation sets. The number of classes of keyword is 4, as the labels range from 0 to 3, likely aligning with the “Yes”, “No”, unknown, and silence.

IV. Problem 4

Total Runtime	Pre-processing	Neural Network	Post-processing
117 ms	22 ms	94 ms	0 ms

Table 1: Breakdown of MCU runtime

	# of Trials	# Correct	% Correct
“Yes”	10	8	80%
“No”	10	10	100%
Total	20	18	90%

Table 2: Experimentation accuracy results

The MCU runtime is significantly slower than the GPU but actually runs faster than the CPU. Additionally, the MCU performs well when hearing “No” with a bit lower performance when hearing “Yes”, recording an unknown and silence reading among the 10 trials. Overall, this performance closely matches the 89-90% accuracy indicated in part 3.

V. Problem 5

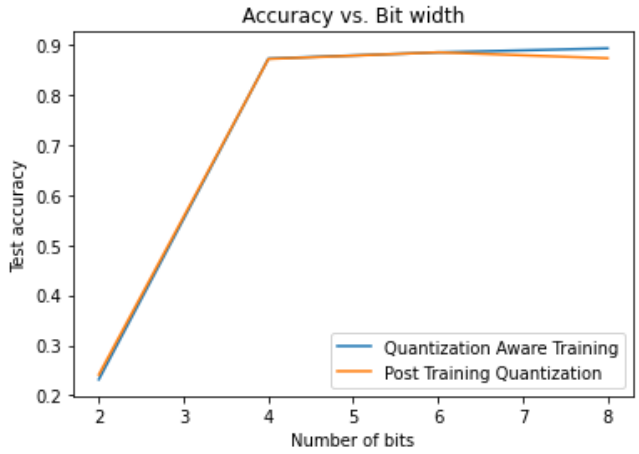

Figure 12: Test accuracy vs. bit width of several models trained in part 5

Several python functions were implemented to help with the quantization process. The first is a simple linear quantization function that takes in an input, scale, and zero point and then the output is the shifted values. Symmetric and asymmetric quantization are also implemented, going beyond the basic linear quantization to also include a bit scale that is achieved using the torch clamp function. Additional helper functions were then added to the quantization configuration class to then be able to take in an input, get the quantization parameters based on the input, and then go through the entire quantization process based on the number of bits and symmetry that the class is initialized with.

In this experimentation, the post train quantization matched the performance of the quantization aware training, which is a bit surprising considering the quantization aware training was fined tuned after quantization occurred and the post training quantization was accomplished by simply taking the model and quantizing it. If these results are consistent with how the two perform in general then post training quantization would be a very ideal option since it is very simple and doesn't require finetuning yet still achieves comparable performance.

VI. Problem 6

```python
from torch.nn.utils import prune
from src.train_val_test_utils import plot_acc
import torch.nn as nn
from src.size_estimate import *
import torch_pruning as tp
from time import time


def prune(model, layer_type, num_channels_remove, structured=True):
    strategy = tp.strategy.L1Strategy() # or tp.strategy.RandomStrategy()
    DG = tp.DependencyGraph()
    DG.build_dependency(model, example_inputs=torch.randn([[1, model_fp32.model_settings['fingerprint_width'], model_fp32.model_settings['spectrogram_length']]]))
    pruning_idxs = strategy(model.conv.weight, amount=num_channels_remove/8.) # or manually selected pruning_idxs=[2, 6, 9, ...]
    print(pruning_idxs)
    pruning_plan = DG.get_pruning_plan( model.conv, tp.prune_conv, idxs=pruning_idxs )
    pruning_plan.exec()
    return model

test_time_data_loaders = make_data_loaders(
    audio_processor, device,
    test_batch_size=1, valid_batch_size=1,
    num_workers=0
)

runtimes = []
baseline_accuracy = []
finetune_accuracy = []
flops = []
parameters = []

for proportion in range(8):
    current_model = copy.deepcopy(model_fp32)
    print([layer.size() for layer in list(model_fp32.parameters())])
    pruned_model = prune(current_model, nn.Conv2d, proportion)
    flop_by_layers = flop(
        model=pruned_model,
        input_shape=(
            1,
            pruned_model.model_settings['fingerprint_width'],
            pruned_model.model_settings['spectrogram_length']
        ),
        device=device)
    print(flop_by_layers)
    print("proportion", proportion, "params", count_trainable_parameters(pruned_model))
    print("flops", sum([sum(val.values()) for val in flop_by_layers.values()]))
    flops.append(sum([sum(val.values()) for val in flop_by_layers.values()]))
    parameters.append(count_trainable_parameters(pruned_model))
    acc, runtime = plot_acc(
        test_time_data_loaders['testing'], pruned_model, audio_processor, device,
        'Testing', 'm-bit Quantized TinyConv', "Float")
    print("accuracy", acc)

    baseline_accuracy.append(acc)
    runtimes.append(runtime)

    # Create optimizer
    optimizer_fp32 = create_optimizer(model=pruned_model, learning_rate=0.0001)

    checkpoint_path = os.path.join(TORCH_DIR, "pruned_finetune_checkpoint" + str(proportion) + ".pt")
    pruned_model.to(device)
    verbose = False
    log_interval = 100
    num_batches = len(train_loader)
    n_epoch = 10
    run_training(
        model=pruned_model, data_loaders=data_loaders,
        n_epoch=n_epoch, log_interval=log_interval,
        optimizer=optimizer_fp32, scheduler=None,
        resume=False,
        checkpoint_path=checkpoint_path,
        verbose=verbose
    )
    ft_acc, _ = plot_acc(
        test_time_data_loaders['testing'], pruned_model, audio_processor, device,
        'Testing', 'm-bit Quantized TinyConv', "Float")
    print("Fine tuned accuracy", ft_acc)
    finetune_accuracy.append(ft_acc)

print("Accuracy", baseline_accuracy)
print("Finetuned accuracy", finetune_accuracy)
print("runtimes", runtimes)
print("Flops", flops)
print("params", parameters)
```

```python
from torch.nn.utils import prune
import torch.nn as nn

def prune_model_l1_unstructured(model, layer_type, proportion):
    for module in model.modules():
        if isinstance(module, layer_type):
            prune.l1_unstructured(module, 'weight', proportion)
            prune.remove(module, 'weight')
    return model

accs = []
ft_accs = []
counts = []
for ratio in range(8):
    actual_ratio = ratio / 8.
    current_model = copy.deepcopy(model_fp32)
    prune_model_l1_unstructured(current_model, nn.Conv2d, actual_ratio)
    acc, _ = plot_acc(
        test_time_data_loaders['testing'], current_model, audio_processor, device,
        'Testing', 'n-bit Quantized TinyConv', "float")
    print("accuracy", acc)

    optimizer_fp32 = create_optimizer(model=pruned_model, learning_rate=0.0001)

    checkpoint_path = os.path.join(TORCH_DIR, "pruned_unstructured_finetune_checkpoint" + str(ratio) + ".pt")
    pruned_model.to(device)
    verbose = False
    log_interval = 100
    num_batches = len(train_loader)
    n_epoch = 8
    run_training(
        model=current_model, data_loaders=data_loaders,
        n_epoch=n_epoch, log_interval=log_interval,
        optimizer=optimizer_fp32, scheduler=None,
        resume=False,
        checkpoint_path=checkpoint_path,
        verbose=verbose
    )
    tf_acc, _ = plot_acc(
        test_time_data_loaders['testing'], current_model, audio_processor, device,
        'Testing', 'n-bit Quantized TinyConv', "float")
    print("ft accuracy", ft_acc)
    print("proportion", proportion, "params", (1. - actual_ratio) * count_trainable_parameters(pruned_model))
    accs.append(acc)
    ft_accs.append(tf_acc)
    counts.append((1. - actual_ratio) * count_trainable_parameters(pruned_model))

print(accs, ft_accs, counts)
```

Figures 13 & 14: Code used to implement structured and unstructured pruning

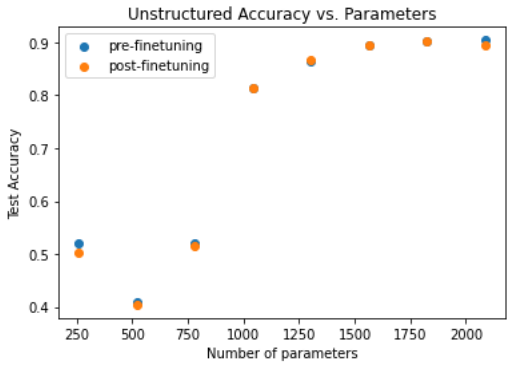

Figure 15: Accuracy vs. number of parameters at different pruning thresholds with and without finetuning

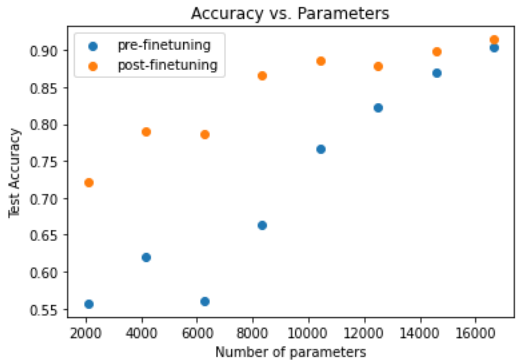

Figure 16: Accuracy vs. parameters for structured pruning

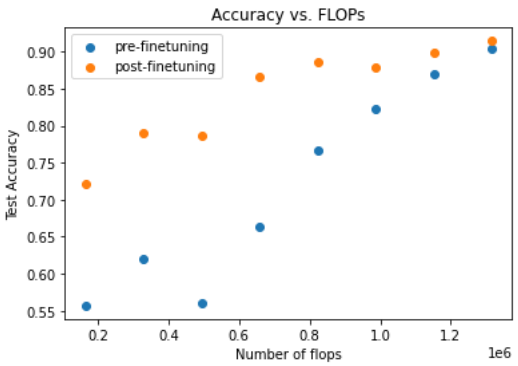

Figure 17: Accuracy vs. FLOPs for structured pruning

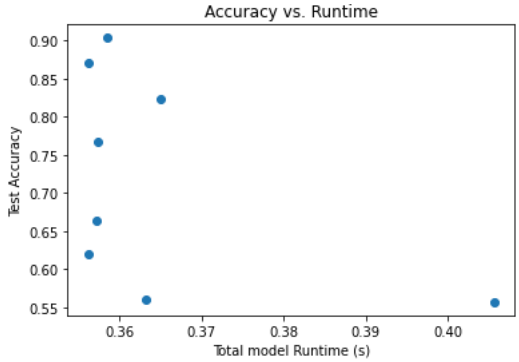

Figure 18: Accuracy vs. runtime for GPU

All of the code used to implement structured and unstructured pruning is shown in the screenshots above as well as included in the notebook for part 6. Pruning can be used to speed up and also reduce the size of the models – by reducing the number of channels the number of flops and number of parameters both decrease by the same scale factor, leading the faster runtimes from the decrease in flops and lower model size from the reduction in parameters. L1 norm finds the sum of the absolute values, L2 takes the square root of the sum of the squared magnitudes, and L infinity just finds the largest absolute magnitude. For pruning, L1 is used because a holistic view of the channel is desired instead of focusing just on the highest magnitude of elements. All additional requested plots are included, with performance generally decreasing with higher rates of pruning (and lower number of flops and parameters). The only plot that is not included is the Accuracy vs. Runtime on the MCU because even though the model was obtained, the implementation of Arduino ran into an error and it is 4 am so I am going to bed.