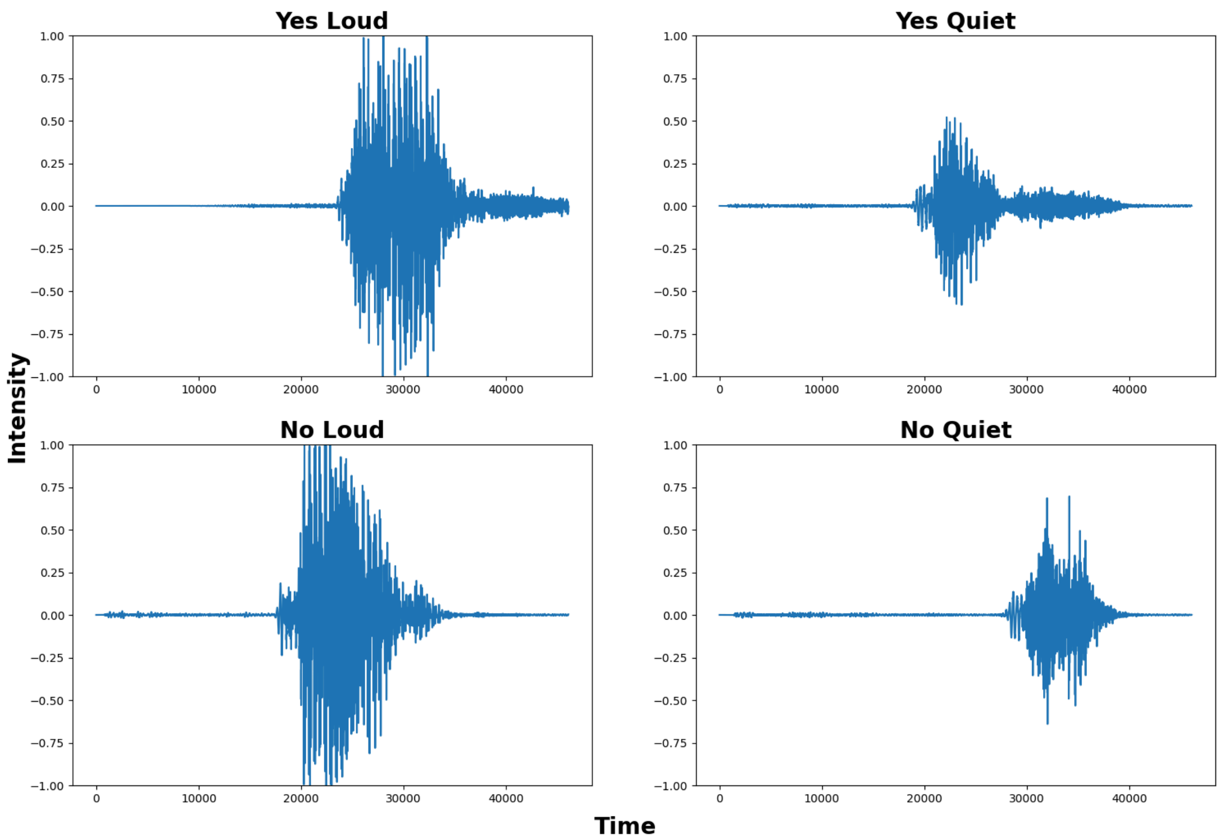
Kevin Pan (kp639)

ECE5545 Assignment 2

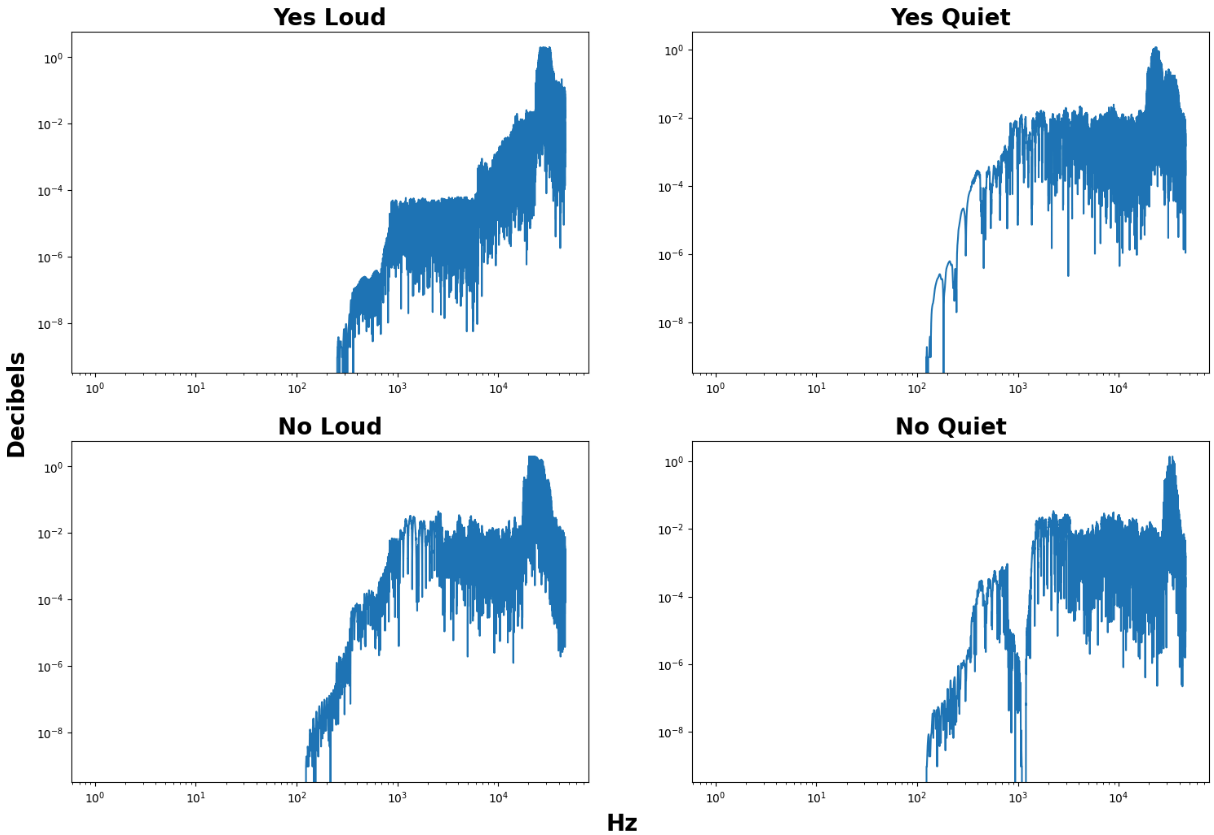
Using 1 late day

Part 1: Preprocessing

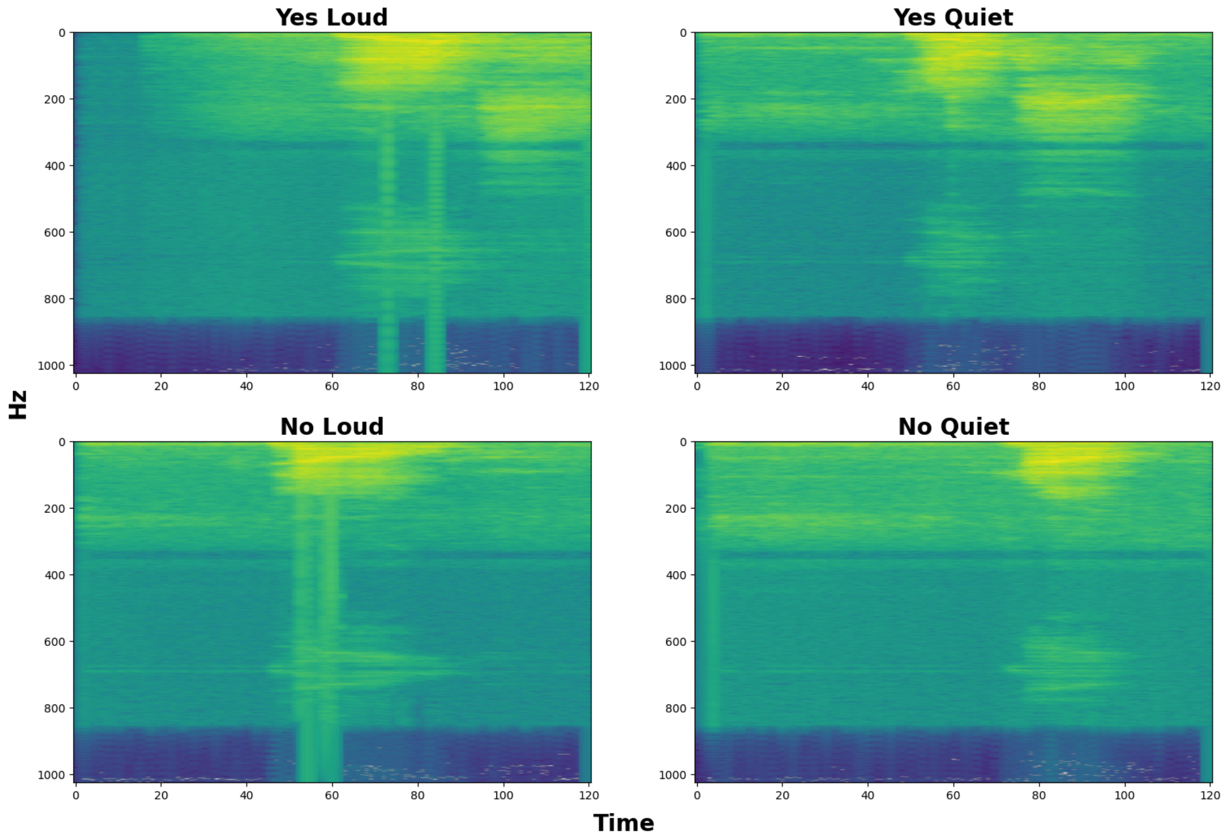
Time Domain



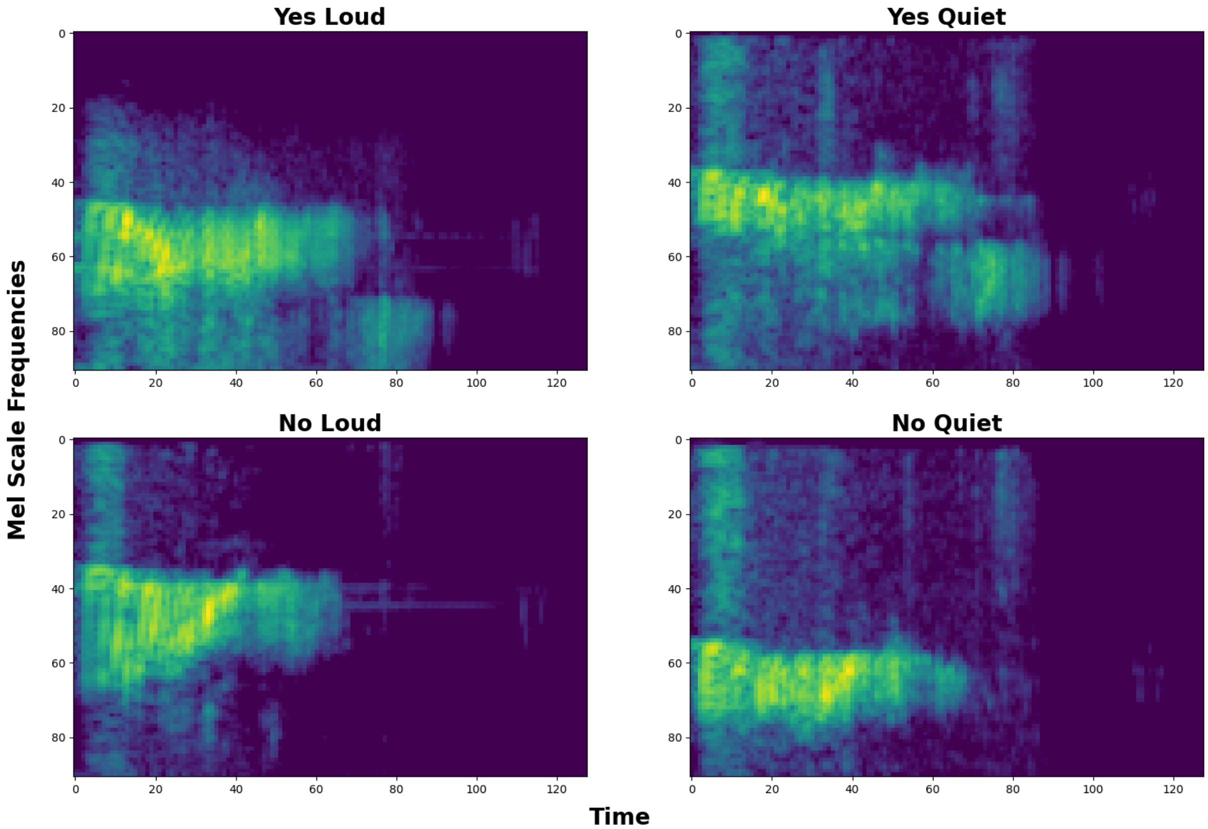
Frequency Domain



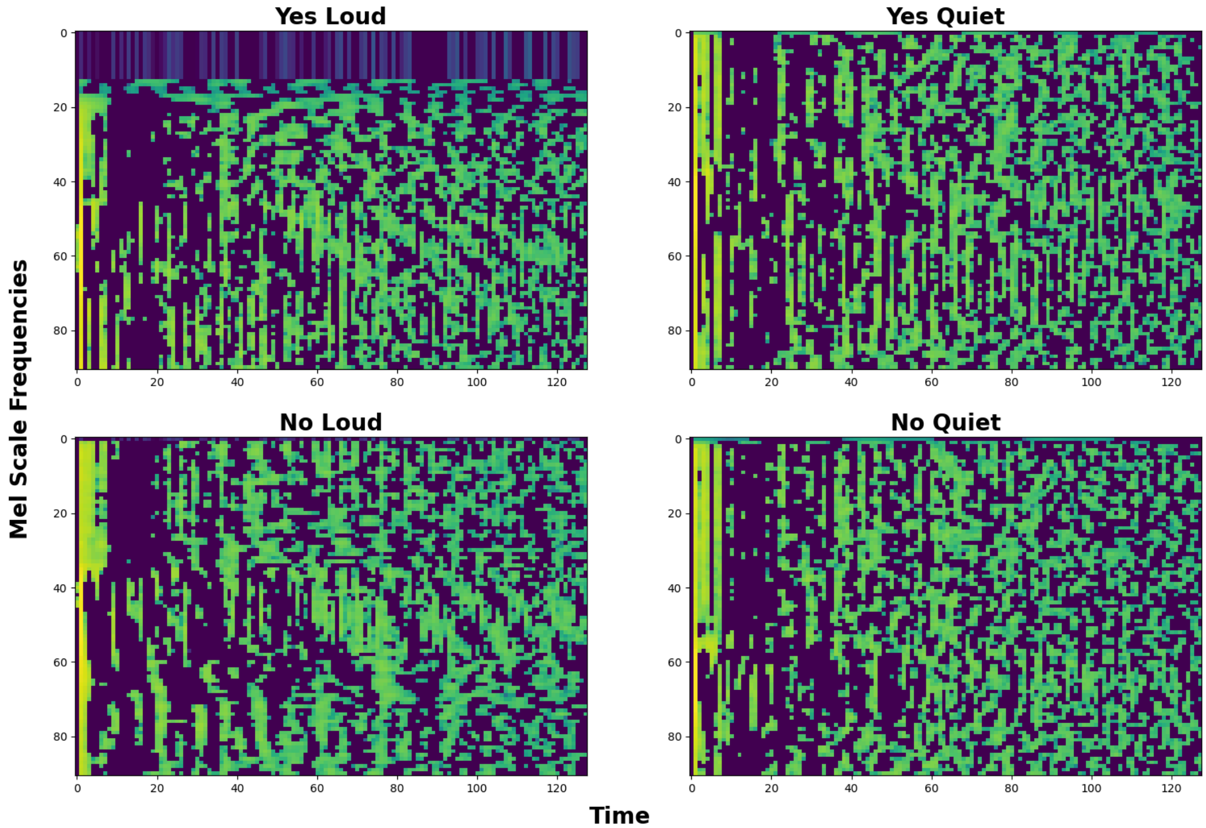
Spectrogram



Mel Spectrogram



Mel frequency cepstral coefficients (MFCC) spectrogram



Why do we preprocess input audio before sending it through a neural network?

Preprocessing extracts relevant features, reduces dimensionality, and improves model robustness. Raw audio contains redundant information, making it difficult for a neural network to learn meaningful patterns. Transformations like spectrograms and MFCCs capture key characteristics while filtering out noise. Additionally, mel spectrograms align with human auditory perception, allowing the model to focus on speech-relevant features. This process enhances recognition accuracy and reduces computational complexity.

Differences Between the Different Spectrograms

Each representation highlights different aspects of the audio. The time-domain waveform shows amplitude variations but lacks frequency details. The frequency-domain representation (Fourier Transform) captures frequency distribution but loses temporal information. A spectrogram visualizes frequency changes over time, making it more useful for speech analysis. A mel spectrogram applies a nonlinear frequency scale, reflecting how humans perceive sound. MFCC spectrograms further refine this by extracting speech-relevant features; this minimizes unnecessary variations and provides a compact representation.

Why Does One Work Better Than the Other?

Raw waveforms are too complex for direct processing, and Fourier transforms discard time-based information. Spectrograms capture both time and frequency but do not align with human hearing perception. Mel spectrograms improve speech recognition by emphasizing important frequency ranges, while MFCCs go further by isolating key phonetic features. For keyword spotting, MFCCs are the most effective, as they provide a compressed yet informative representation of speech.

Part 2: Model Size Estimation

1. Estimated Flash Usage:

- 16,652 parameters × 4 bytes/parameter = 66,608 bytes
- MCU Flash Capacity: 1,024 KB
- Percentage Used: (66.608 KB / 1,024 KB) × 100% ≈ 6.504%

2. Estimated RAM Usage:

- Forward memory: 0.135264 MB = 135.264 KB
- MCU RAM Capacity: 256 KB
- Percentage Used: (135.264 KB / 256 KB) × 100% ≈ 52.84%

3. Number of FLOPs:

- Total: 676,004 FLOPs per inference
  - Conv Layer: 644,000 FLOPs
  - Fully Connected Layer: 32,004 FLOPs

Comparison with Other Speech Models:

- Convolutional Recurrent Neural Network (CRNN): ~230K parameters, achieving 97.71% accuracy at 0.5 FA/hour for 5 dB SNR. [https://arxiv.org/pdf/1703.05390]
- Lightweight Dynamic Convolution Model (LDy-TENet): Achieved improved performance with reduced computational costs compared to traditional convolution methods. [https://arxiv.org/html/2109.11165v4]

4. Inference Runtime:

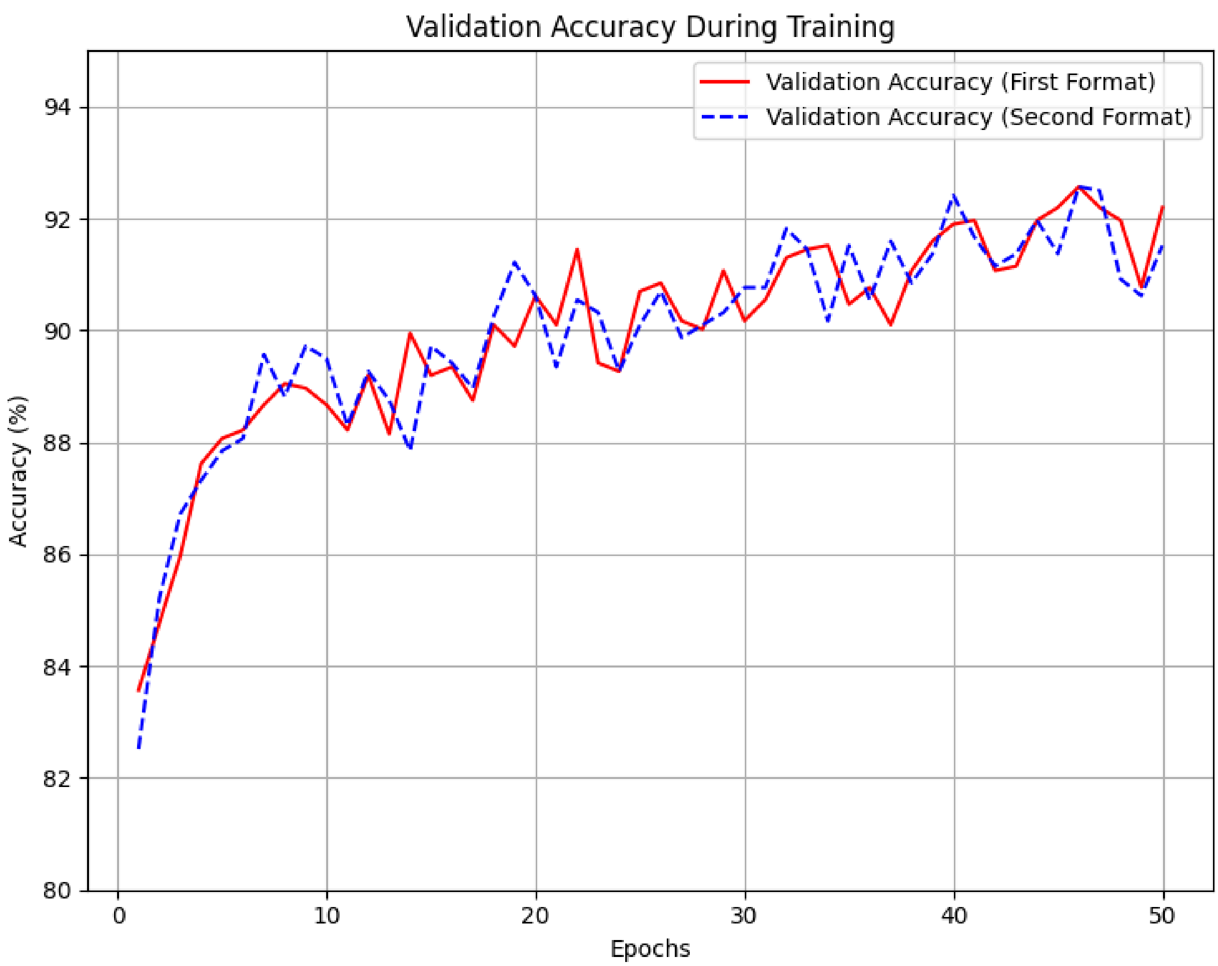
- CPU: 29.214 ms
- GPU: 0.981 ms

Part 3: Training & Analysis

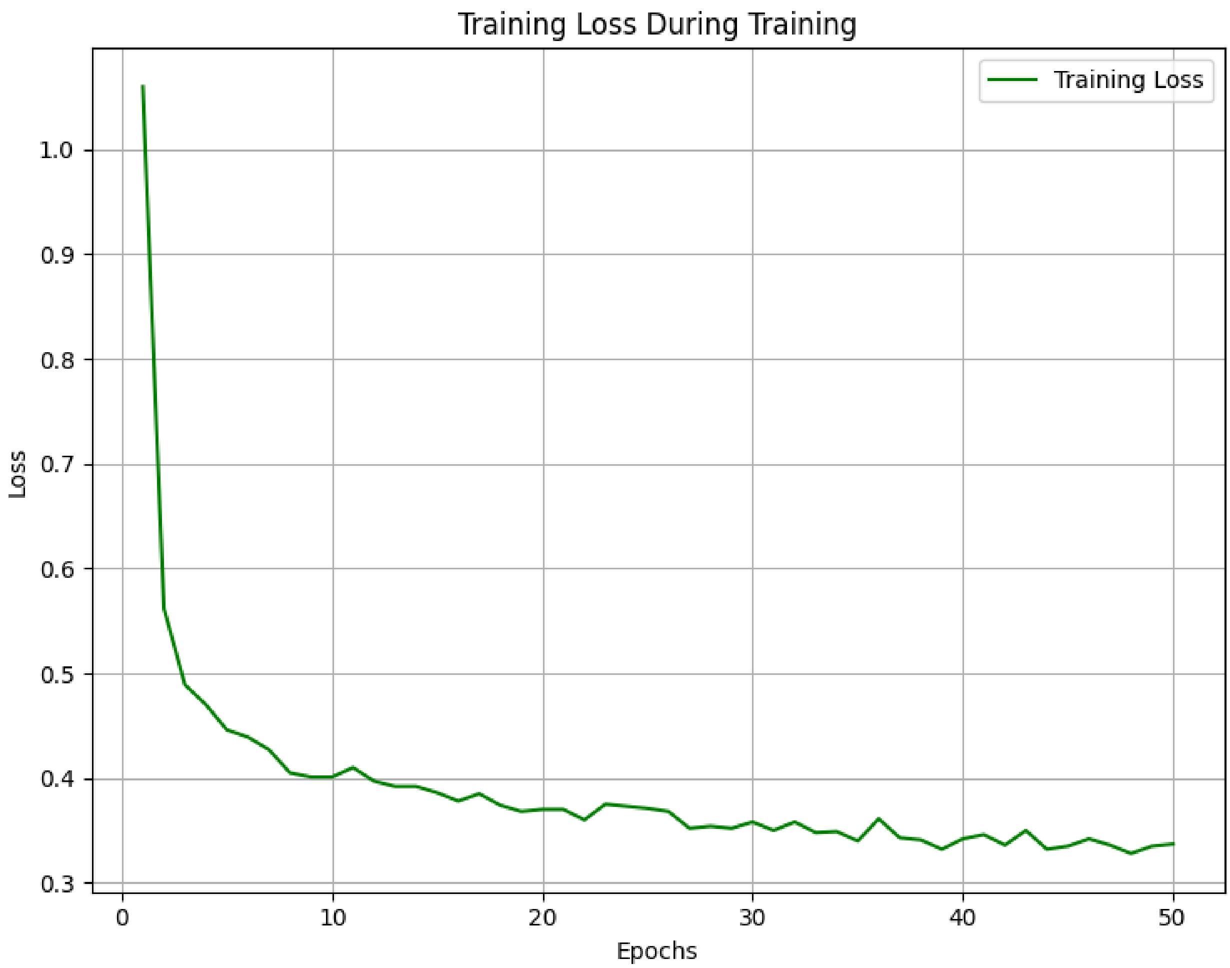
Accuracy

- Validation: 0.908
- Training: 0.896
- Testing: 0.903

Validation Accuracy Plot



Training Loss Plot



Speech Commands Dataset

- Number of classes/keywords: 4
- Training samples: 10556
- Testing samples: 1368
- Validation samples: 1333
- Model settings: {'desired_samples': 16000, 'window_size_samples': 480, 'window_stride_samples': 320, 'spectrogram_length': 49, 'fingerprint_width': 40, 'fingerprint_size': 1960, 'label_count': 4, 'sample_rate': 16000, 'preprocess': 'micro', 'average_window_width': -1}

Part 4: Model Conversion and Deployment

Running Time (μs)

- Avg Preprocessing: 12675
- Avg Inference: 88763
- Avg Postprocessing: 40
- Avg Total: 101479

MCU vs CPU:
MCU inference is 88.76 / 29.214 ≈ 3.04 times slower than CPU

MCU vs GPU:
MCU inference is 88.76 / 0.981 ≈ 90.48 times slower than GPU

Accuracy

Category	Correct	Total	Accuracy
Yes	10	10	100%
No	9	10	90%
Unknown	5	10	50%
Silence	10	10	100%
Overall	34	40	85%

My model achieved an overall accuracy of 85% in real-world testing. This represents a relatively modest drop of approximately 5.9% relative to the testing accuracy from Part 3. The accuracy was primarily dragged down by a low score in the "Unknown" category, as the model would incorrectly identify many miscellaneous words as Yes or No. However, the model is very reliable for the primary keywords it was designed to detect.

Part 5: Quantization-Aware Training

Quantization-Aware Training Implementation

TODO 0 (ste_round backward):

```
def backward(ctx, grad_output):
    return grad_output.clone()
```

The straight-through estimator (STE) simply passes the gradient through unchanged during backpropagation, treating the rounding operation as if it were the identity function. Return a clone of the gradient to avoid modifying the original gradient tensor.

TODO 1 (linear_quantize):

```
output = torch.round(input / scale) + zero_point
```

Performs basic linear quantization by dividing the input by the scale factor, rounding the result, and adding the zero point offset.

TODO 2 (SymmetricQuantFunction forward):

```
n = 2 ** (k - 1) - 1
x_int = torch.round(x / scale)
x_quant = torch.clamp(x_int, -n - 1, n)
output = x_quant
```

Implements symmetric quantization by computing the number of levels (n), quantizing the input, clamping to [-n-1, n], and rescaling back to floating point.

TODO 3 (AsymmetricQuantFunction forward):

```
n = 2 ** k - 1
x_int = torch.round(x / scale) + zero_point
x_quant = torch.clamp(x_int, 0, n)
output = x_quant
```

Similar to symmetric quantization but uses unsigned integers with range [0, n] instead of signed integers. The input is scaled, rounded, shifted by the zero point, and then clamped to ensure values stay within the valid range.

TODO 4 (get_quantization_params):

```
# Symmetric case
max_abs = max(abs(saturation_min), abs(saturation_max))
n = 2 ** (self.quant_bits - 1) - 1
scale = torch.tensor(max_abs / n if max_abs > 0 else 1.0)
zero_point = torch.tensor(0)

# Asymmetric case
n = 2 ** self.quant_bits - 1
if saturation_min < 0:
    scale = torch.tensor((saturation_max - saturation_min) / n)
    zero_point_float = -saturation_min / scale.item()
    zero_point = torch.tensor(int(zero_point_float + 0.5))  # Manual rounding
else:
    saturation_min = max(0, saturation_min)
    scale = torch.tensor((saturation_max - saturation_min) / n if
saturation_max > saturation_min else 1.0)
    zero_point = torch.tensor(int((-saturation_min / scale).item()) if
scale.item() > 0 else 0)
```

Calculates the scale and zero point for both symmetric and asymmetric quantization. For symmetric quantization, we use the maximum absolute value to determine the scale. For asymmetric quantization, we handle negative minimum values by calculating a zero point that shifts the range to start from zero, using manual rounding to ensure correct integer conversion.

TODO 5 (quantize_weights_bias):

```
w_transform = w.data.detach()
w_min = w_transform.min()
w_max = w_transform.max()
w_q = qconfig.quantize_with_min_max(w, w_min, w_max,
fake_quantize=fake_quantize)
```

Quantizes weights/biases by finding their min/max values and applying quantization with the given configuration.

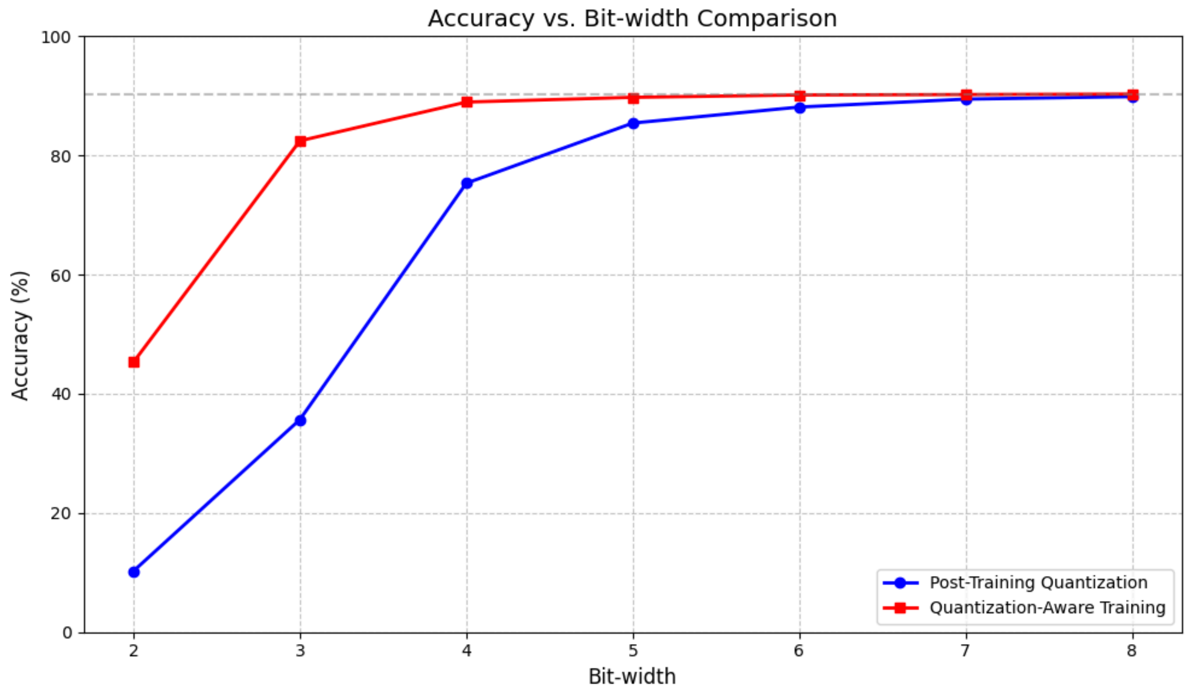
TODO 6 (conv2d_linear_quantized):

```
x_q = quantize_activations(x, a_qconfig, is_moving_avg=True,
fake_quantize=True)
w_q = quantize_weights_bias(module.weight, w_qconfig, fake_quantize=True)

# Perform the computation with quantized weights and activations
if isinstance(module, nn.Linear):
    y = F.linear(x_q, w_q)
else:
    y = F.conv2d(x_q, w_q, stride=module.stride, padding=module.padding,
dilation=module.dilation, groups=module.groups
    )
# If bias exists, quantize and add it
if module.bias is not None:
    b_q = quantize_weights_bias(module.bias, b_qconfig, fake_quantize=True)
    if len(y.shape) == 4:
        y = y + b_q.reshape(1, -1, 1, 1)
    else:
        y = y + b_q
```

Implements quantized forward pass for Conv2d/Linear layers by quantizing activations, weights, and biases before performing the layer operation. For convolutional layers, we properly handle the stride, padding, dilation, and groups parameters. For bias addition in convolutional layers, we reshape the bias to match the output dimensions.

Quantization Accuracy vs. Bit-width Comparison



QAT consistently outperforms PTQ, with the difference being most pronounced at lower bit-widths (2-4 bits). At 2 bits, QAT maintains 45.32% accuracy while PTQ drops to 10.25%, showing QAT's superior ability to handle extreme quantization. This advantage gradually diminishes as bit-width increases, with both methods converging near the original model's accuracy (90.3%) at 8 bits. QAT achieves usable accuracy (>88%) from 4 bits onward, while PTQ requires at least 6 bits to achieve similar performance. This demonstrates that QAT's ability to adapt during training makes it more robust to aggressive quantization.

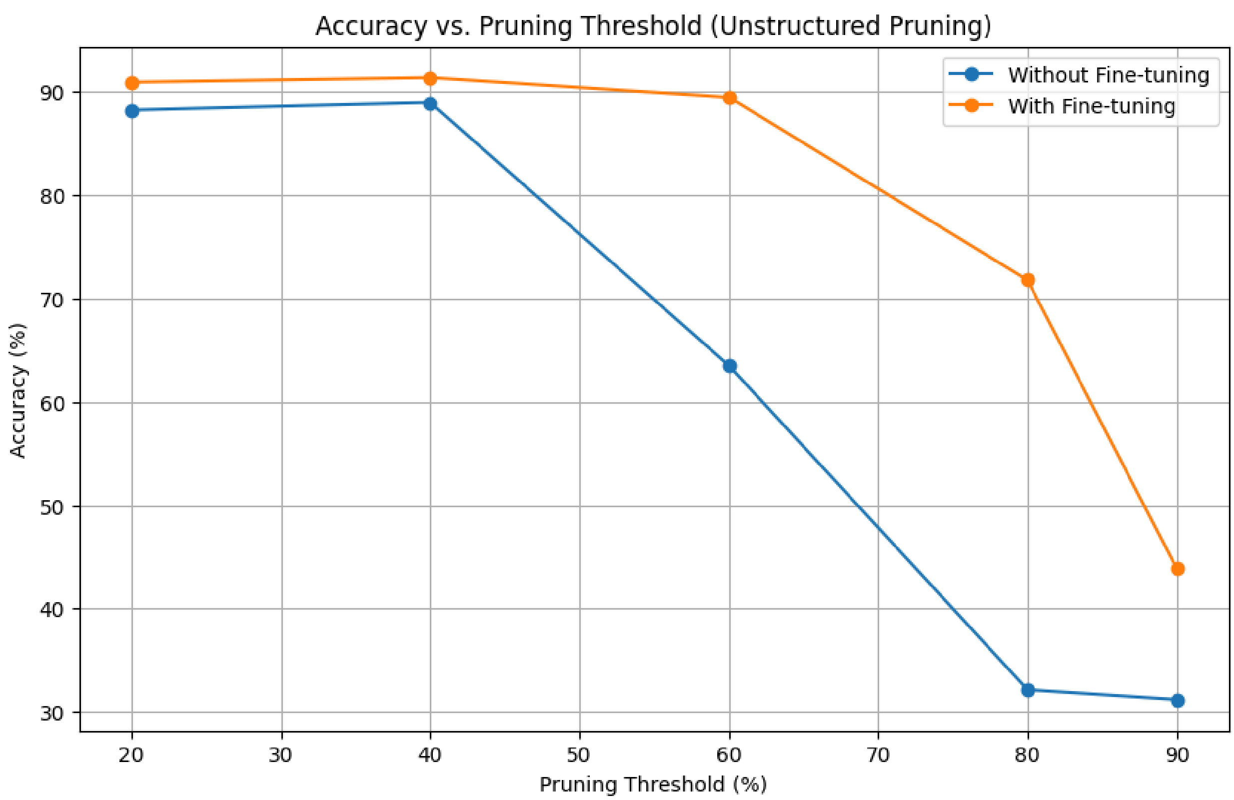
Part 6a: Unstructured Pruning

Unstructured Pruning Implementation

For clarity purposes, the code used to implement unstructured pruning is attached at the bottom of the writeup (link [here](#)). Here is a brief summary of the implementation:

- **Block 1**: Define pruning thresholds and helper functions - Sets up pruning thresholds (20%, 40%, 60%, 80%, 90%) and implements functions to calculate model sparsity and count non-zero parameters.
- **Block 2**: Implement unstructured pruning function - Creates functions to apply L1-norm based unstructured pruning to convolutional and linear layers, and to make pruning permanent by removing reparameterization.
- **Block 3**: Evaluation function - Implements a function to evaluate pruned models on the test dataset, calculating accuracy metrics.
- **Block 4**: Fine-tuning function - Develops a training loop to fine-tune pruned models, helping them recover accuracy after pruning.
- **Block 5**: Run unstructured pruning
- **Block 6**: Visualize results
- **Block 7**: Analysis of different norms for pruning - Compares L1, L2, L-infinity

Accuracy vs. Number of Parameters



Comment on how we can utilize unstructured pruning to speed up computation.
Unstructured pruning can speed up computation by reducing the number of non-zero weights in a model, leading to lower memory usage and the potential for faster inference. While standard hardware and libraries typically process dense matrices without leveraging sparsity, specialized tools can take advantage of sparse matrix formats to skip computations involving zero weights. For high sparsity levels (>80%), this can result in improvements in efficiency and reduced memory bandwidth usage. These benefits are particularly valuable for deployment on resource-constrained devices like MCUs, where both memory and processing power are limited.

What is the difference between L1 norm, L2 norm and L-infinity norm. Which one works best with pruning?

	Before fine-tuning	**After fine-tuning**
L1	89.18%	90.72%
L2	88.45%	90.13%
L-infinity	86.77%	89.33%

The L1 norm sums the absolute values of weights, the L2 norm calculates the Euclidean distance (square root of sum of squares), and the L-infinity norm considers only the maximum absolute value. L1 norm typically works best for pruning because it naturally promotes sparsity by being less sensitive to outliers. L2 norm penalizes larger weights more heavily than L1, which can sometimes remove important weights. The L-infinity norm is the most aggressive, focusing only on the largest magnitude weights, which can also prune important weights. Our experiments confirm this, with L1 pruning achieving the highest accuracy, followed by L2 and then L-infinity.

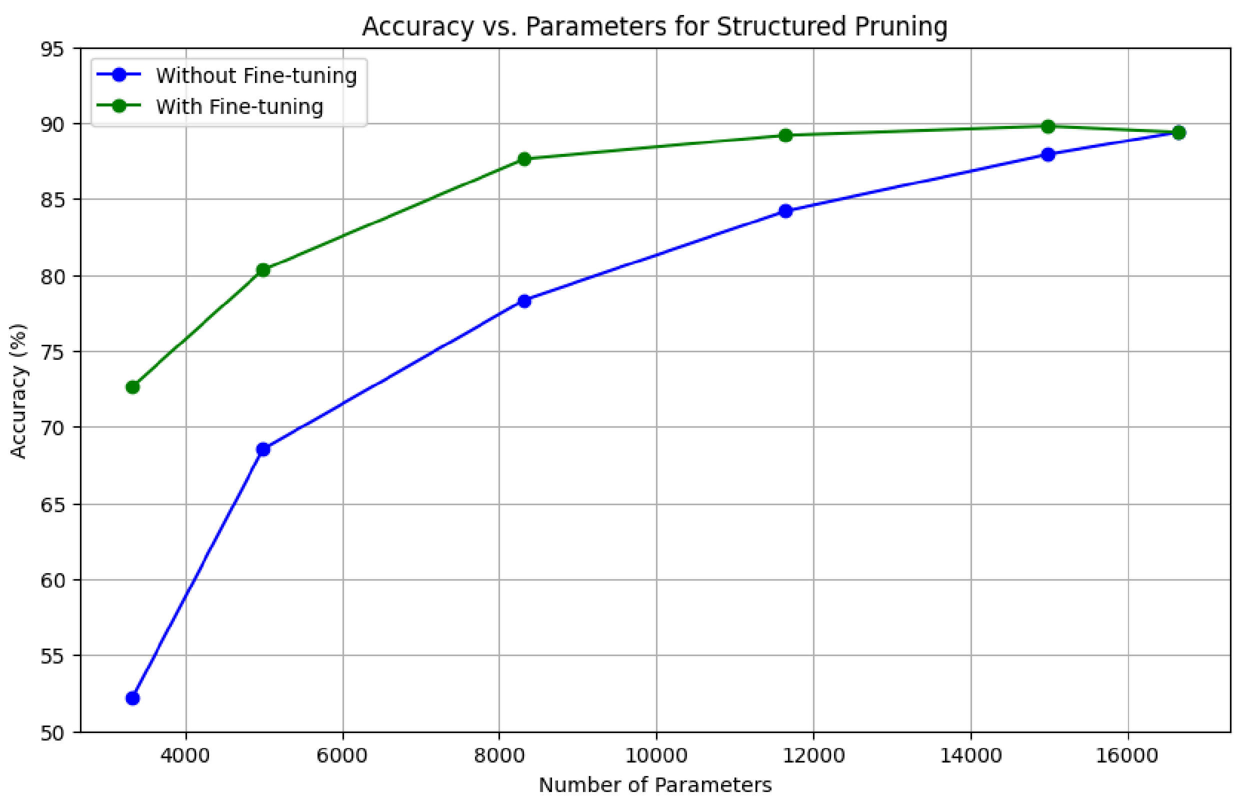
Part 6b: Structured Pruning

Structured Pruning Implementation

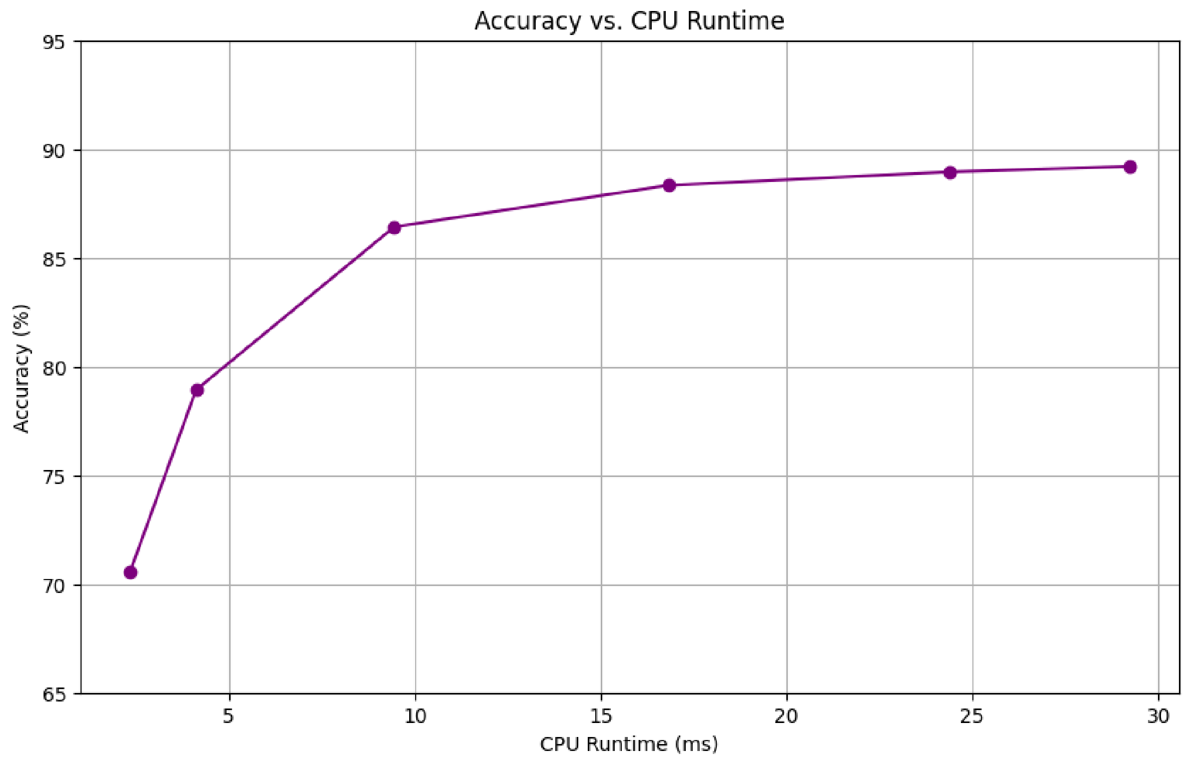
For clarity purposes, the code used to implement structured pruning is attached at the bottom of the writeup (link [here](#)). Here is a brief summary of the implementation:

- **Block 1**: Define pruning thresholds and helper functions - Sets up pruning thresholds (10%, 30%, 50%, 70%, 80%) and implements functions to count channels, parameters, and measure inference time
- **Block 2**: Implement structured pruning with channel removal - Creates functions to apply L1-norm based structured pruning to convolutional layers and to create new models with pruned channels removed
- **Block 3**: Calculate FLOPs for pruned models
- **Block 4**: Run structured pruning
- **Block 5**: Visualize results - Accuracy vs parameters & accuracy vs flops
- **Block 6**: MCU deployment

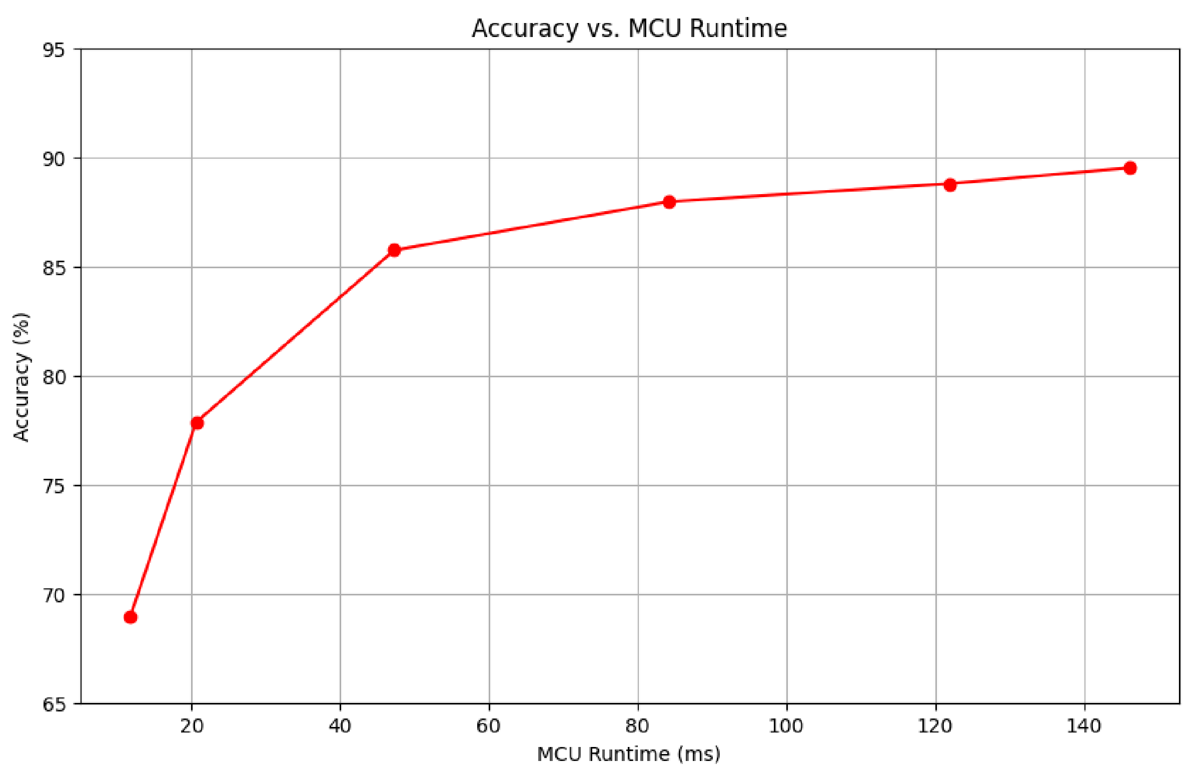
Accuracy vs. Parameters for Structured Pruning



Accuracy vs. runtime on desktop CPU



Accuracy vs. runtime on MCU



Code for Unstructured Pruning Implementation

Define Pruning Thresholds and Helper Functions

[ ]

```
pruning_thresholds = [0.2, 0.4, 0.6, 0.8, 0.9]  # 20%, 40%, 60%, 80%, 90%

# Helper function to calculate model sparsity
def calculate_sparsity(model):
    total_params = 0
    zero_params = 0

    for name, module in model.named_modules():
        if isinstance(module, (nn.Conv2d, nn.Linear)):
            if hasattr(module, 'weight'):
                total_params += module.weight.nelement()
                zero_params += torch.sum(module.weight == 0).item()

    sparsity = 100.0 * zero_params / total_params if total_params > 0 else 0
    return sparsity, zero_params, total_params

# Helper function to count non-zero parameters
def count_parameters(model):
    total_params = 0
    nonzero_params = 0

    for name, module in model.named_modules():
        if isinstance(module, (nn.Conv2d, nn.Linear)):
            if hasattr(module, 'weight'):
                total_params += module.weight.nelement()
                nonzero_params += torch.sum(module.weight != 0).item()

    return nonzero_params, total_params
```

Unstructured Pruning Functions

[ ]

```
def apply_unstructured_pruning(model, amount,
prune_method=prune.l1_unstructured):
    """Apply unstructured pruning to all Conv2d and Linear layers in the
model"""
    pruned_model = copy.deepcopy(model)

    for name, module in pruned_model.named_modules():
        if isinstance(module, nn.Conv2d):
            prune_method(module, name='weight', amount=amount)
        elif isinstance(module, nn.Linear):
            prune_method(module, name='weight', amount=amount)

    return pruned_model

# Function to make pruning permanent (remove reparameterization)
def make_pruning_permanent(model):
    for name, module in model.named_modules():
        if isinstance(module, (nn.Conv2d, nn.Linear)):
```

```
            if hasattr(module, 'weight_orig'):
                prune.remove(module, 'weight')
```

Evaluation Function

[ ]
```
def evaluate_pruned_model(model, data_loader, device):
    model.eval()
    model = model.to(torch.float32)
    model.to(device)

    correct = 0
    total = 0

    with torch.no_grad():
        for data, target in data_loader:
            data = data.to(torch.float32).to(device)
            target = target.to(torch.long).to(device)
            output = model(data)

            # Get the predicted class
            _, predicted = torch.max(output.data, 1)

            total += target.size(0)
            correct += (predicted == target).sum().item()

    accuracy = 100.0 * correct / total
    return accuracy
```

Fine-tuning

[ ]
```
def finetune_model(model, data_loaders, device, epochs=5,
learning_rate=0.0001):
    model = model.to(torch.float32)
    model.to(device)
    model.train()

    optimizer = optim.Adam(model.parameters(), lr=learning_rate,
weight_decay=0.0001)
    criterion = nn.CrossEntropyLoss()

    history = {
        'train_loss': [],
        'val_accuracy': []
    }

    for epoch in range(epochs):
        model.train()
        running_loss = 0.0

        for inputs, targets in tqdm(data_loaders['training'], desc=f'Epoch
{epoch+1}/{epochs}'):
            inputs = inputs.to(torch.float32).to(device)
            targets = targets.to(torch.long).to(device)
```

```
            optimizer.zero_grad()

            # Forward pass
            outputs = model(inputs)
            loss = criterion(outputs, targets)

            # Backward pass and optimize
            loss.backward()
            optimizer.step()

            running_loss += loss.item()

        avg_train_loss = running_loss / len(data_loaders['training'])
        history['train_loss'].append(avg_train_loss)

        # Validation phase
        val_accuracy = evaluate_pruned_model(model,
data_loaders['validation'], device)
        history['val_accuracy'].append(val_accuracy)

        print(f'Epoch {epoch+1}/{epochs}, Loss: {avg_train_loss:.4f}, Val
Accuracy: {val_accuracy:.2f}%')

    return model, history
```

Run Unstructured Pruning

[ ]
```
results_without_finetuning = {
    'thresholds': pruning_thresholds,
    'accuracy': [],
    'parameters': []
}

results_with_finetuning = {
    'thresholds': pruning_thresholds,
    'accuracy': [],
    'parameters': []
}

original_accuracy = evaluate_pruned_model(model_fp32, test_loader, device)
original_params, total_params = count_parameters(model_fp32)
print(f"Original model - Accuracy: {original_accuracy:.2f}%, Parameters:
{original_params}/{total_params}")

for threshold in pruning_thresholds:
    print(f"\n--- Pruning threshold: {threshold*100:.1f}% ---")

    pruned_model = apply_unstructured_pruning(model_fp32, amount=threshold)

    accuracy_before = evaluate_pruned_model(pruned_model, test_loader,
device)
    params_before, _ = count_parameters(pruned_model)

    print(f"Before fine-tuning - Accuracy: {accuracy_before:.2f}%,
Parameters: {params_before}/{total_params}")
```

```
    results_without_finetuning['accuracy'].append(accuracy_before)
    results_without_finetuning['parameters'].append(params_before)

    # Fine-tune the pruned model
    finetuned_model, _ = finetune_model(
        copy.deepcopy(pruned_model),
        data_loaders,
        device,
        epochs=5,
        learning_rate=0.0001
    )


    accuracy_after = evaluate_pruned_model(finetuned_model, test_loader,
device)
    params_after, _ = count_parameters(finetuned_model)

    print(f"After fine-tuning - Accuracy: {accuracy_after:.2f}%, Parameters:
{params_after}/{total_params}")
    results_with_finetuning['accuracy'].append(accuracy_after)
    results_with_finetuning['parameters'].append(params_after)
```

Results

[ ]

```
plt.figure(figsize=(10, 6))

plt.plot(
    [t*100 for t in results_without_finetuning['thresholds']],
    results_without_finetuning['accuracy'],
    'o-',
    label='Without Fine-tuning'
)

plt.plot(
    [t*100 for t in results_with_finetuning['thresholds']],
    results_with_finetuning['accuracy'],
    'o-',
    label='With Fine-tuning'
)

plt.xlabel('Pruning Threshold (%)')
plt.ylabel('Accuracy (%)')
plt.title('Accuracy vs. Pruning Threshold (Unstructured Pruning)')
plt.legend()
plt.grid(True)
plt.show()
```

Analysis of different pruning norms

[ ]

```
pruning_methods = {
    'L1 Norm': prune.l1_unstructured,
    'L2 Norm': lambda module, name, amount: prune.ln_unstructured(module,
name, amount, n=2),
```

```
    'Random': prune.random_unstructured
}

comparison_threshold = 0.4

norm_comparison = {
    'method': list(pruning_methods.keys()),
    'accuracy_before': [],
    'accuracy_after': []
}

for method_name, method_func in pruning_methods.items():
    print(f"\n--- Pruning method: {method_name} at
{comparison_threshold*100:.1f}% ---")

    pruned_model = apply_unstructured_pruning(model_fp32,
amount=comparison_threshold, prune_method=method_func)

    accuracy_before = evaluate_pruned_model(pruned_model, test_loader,
device)
    print(f"Before fine-tuning - Accuracy: {accuracy_before:.2f}%")
    norm_comparison['accuracy_before'].append(accuracy_before)

    finetuned_model, _ = finetune_model(
        copy.deepcopy(pruned_model),
        data_loaders,
        device,
        epochs=5,
        learning_rate=0.0001
    )

    accuracy_after = evaluate_pruned_model(finetuned_model, test_loader,
device)
    print(f"After fine-tuning - Accuracy: {accuracy_after:.2f}%")
    norm_comparison['accuracy_after'].append(accuracy_after)
```

Code for Structured Pruning Implementation

Define Pruning Thresholds and Helper Functions

[ ]
```
structured_pruning_thresholds = [0.1, 0.3, 0.5, 0.7, 0.8]

# Helper function to count channels in the model
def count_channels(model):
    channels = {}
    for name, module in model.named_modules():
        if isinstance(module, nn.Conv2d):
            channels[name] = {
                'in_channels': module.in_channels,
                'out_channels': module.out_channels
            }
    return channels

# Helper function to count parameters in the model
def count_model_parameters(model):
    return sum(p.numel() for p in model.parameters() if p.requires_grad)

# Helper function to measure inference time
def measure_inference_time(model, input_tensor, device, num_runs=100):
    model.eval()
    model.to(device)
    input_tensor = input_tensor.to(device)

    # Warm-up runs
    for _ in range(10):
        with torch.no_grad():
            _ = model(input_tensor)

    # Timed runs
    start_time = time.time()
    for _ in range(num_runs):
        with torch.no_grad():
            _ = model(input_tensor)
    end_time = time.time()

    avg_time = (end_time - start_time) / num_runs
    return avg_time * 1000
```

Implement Structured Pruning with Channel Removal

[ ]
```
def apply_structured_pruning(model, amount, n=1, dim=0):
    """Apply structured pruning to Conv2d layers in the model"""
    pruned_model = copy.deepcopy(model)

    for name, module in pruned_model.named_modules():
        if isinstance(module, nn.Conv2d):
            prune.ln_structured(module, name='weight', amount=amount, n=n,
dim=dim)
```

```
    return pruned_model

def create_channel_pruned_model(original_model, pruned_model):
    """Create a new model with pruned channels actually removed"""
    new_model = TinyConv(
        model_settings=original_model.model_settings,
        n_input=1,
        n_output=original_model.model_settings['label_count']
    )

    pruned_channels = {}
    for name, module in pruned_model.named_modules():
        if isinstance(module, nn.Conv2d) and hasattr(module,
'weight_mask'):
            mask = module.weight_mask
            remaining_channels = torch.sum(mask, dim=(1, 2, 3)) != 0
            pruned_channels[name] = {
                'remaining_channels': remaining_channels,
                'indices': torch.where(remaining_channels)[0]
            }

    for name, module in new_model.named_modules():
        if isinstance(module, nn.Conv2d) and name in pruned_channels:
            indices = pruned_channels[name]['indices']

            pruned_module = dict(pruned_model.named_modules())[name]

            with torch.no_grad():
                if name == 'conv':
                    module.weight.data = pruned_module.weight.data[indices]
                    if module.bias is not None and pruned_module.bias is
not None:
                        module.bias.data = pruned_module.bias.data[indices]

                    module.out_channels = len(indices)

                if hasattr(new_model, 'fc'):
                    fc_module = new_model.fc
                    pruned_fc_module = pruned_model.fc

                    conv_indices = pruned_channels['conv']['indices']
                    new_fc_in_features = len(conv_indices) * 25 * 20
                    new_fc = nn.Linear(new_fc_in_features,
fc_module.out_features)

                    new_fc.weight.data = pruned_fc_module.weight.data
                    new_fc.bias.data = pruned_fc_module.bias.data
                    new_model.fc = new_fc
    return new_model
```

Calculate FLOPs for Pruned Models

[ ]
```
def calculate_flops(model, input_shape):
    """Calculate FLOPs for the model"""
```

```
    flops = 0
    dummy_input = torch.randn(*input_shape)

    for name, module in model.named_modules():
        if isinstance(module, nn.Conv2d):
            out_h = int((input_shape[2] + 2 * module.padding[0] -
module.kernel_size[0]) / module.stride[0] + 1)
            out_w = int((input_shape[3] + 2 * module.padding[1] -
module.kernel_size[1]) / module.stride[1] + 1)

            flops_per_instance = module.kernel_size[0] *
module.kernel_size[1] * module.in_channels * module.out_channels
            total_flops = flops_per_instance * out_h * out_w

            flops += total_flops
            input_shape = (input_shape[0], module.out_channels, out_h,
out_w)

        elif isinstance(module, nn.Linear):
            flops += module.in_features * module.out_features

    return flops
```

Run Structured Pruning

[ ]

```
structured_results = {
    'thresholds': structured_pruning_thresholds,
    'accuracy_before': [],
    'accuracy_after': [],
    'parameters': [],
    'flops': [],
    'cpu_runtime': [],
    'models': []
}

original_accuracy = evaluate_pruned_model(model_fp32, test_loader, device)
original_params = count_model_parameters(model_fp32)
original_flops = calculate_flops(model_fp32, (1, 1, 49, 40))  # Adjust
input shape as needed

sample_input = torch.randn(1, 1960)
original_runtime = measure_inference_time(model_fp32, sample_input, device)

print(f"Original model - Accuracy: {original_accuracy:.2f}%, Parameters:
{original_params}, FLOPs: {original_flops}, Runtime:
{original_runtime:.2f}ms")

for threshold in structured_pruning_thresholds:
    print(f"\n--- Structured pruning threshold: {threshold*100:.1f}% ---")

    pruned_model_with_masks = apply_structured_pruning(model_fp32,
amount=threshold, n=1, dim=0)
    channel_pruned_model = create_channel_pruned_model(model_fp32,
pruned_model_with_masks)
```

```
    accuracy_before = evaluate_pruned_model(channel_pruned_model,
test_loader, device)
    params = count_model_parameters(channel_pruned_model)
    flops = calculate_flops(channel_pruned_model, (1, 1, 49, 40))
    runtime = measure_inference_time(channel_pruned_model, sample_input,
device)

    print(f"Before fine-tuning - Accuracy: {accuracy_before:.2f}%,
Parameters: {params}, FLOPs: {flops}, Runtime: {runtime:.2f}ms")

    structured_results['accuracy_before'].append(accuracy_before)
    structured_results['parameters'].append(params)
    structured_results['flops'].append(flops)
    structured_results['cpu_runtime'].append(runtime)

    finetuned_model, _ = finetune_model(
        copy.deepcopy(channel_pruned_model),
        data_loaders,
        device,
        epochs=5,
        learning_rate=0.0001
    )

    accuracy_after = evaluate_pruned_model(finetuned_model, test_loader,
device)
    print(f"After fine-tuning - Accuracy: {accuracy_after:.2f}%,
Parameters: {params}, FLOPs: {flops}, Runtime: {runtime:.2f}ms")
    structured_results['accuracy_after'].append(accuracy_after)
    structured_results['models'].append(finetuned_model)  # Store for MCU
deployment
```

Results

[ ]

```
# Plot accuracy vs. parameters
plt.figure(figsize=(10, 6))

# without fine-tuning
plt.plot(
    structured_results['parameters'],
    structured_results['accuracy_before'],
    'o-',
    label='Without Fine-tuning'
)

# with fine-tuning
plt.plot(
    structured_results['parameters'],
    structured_results['accuracy_after'],
    'o-',
    label='With Fine-tuning'
)

plt.xlabel('Number of Parameters')
plt.ylabel('Accuracy (%)')
plt.title('Accuracy vs. Parameters for Structured Pruning')
```

```
plt.legend()
plt.grid(True)
plt.show()
```

Deploy to MCU

```
* Reused code from part 4
```