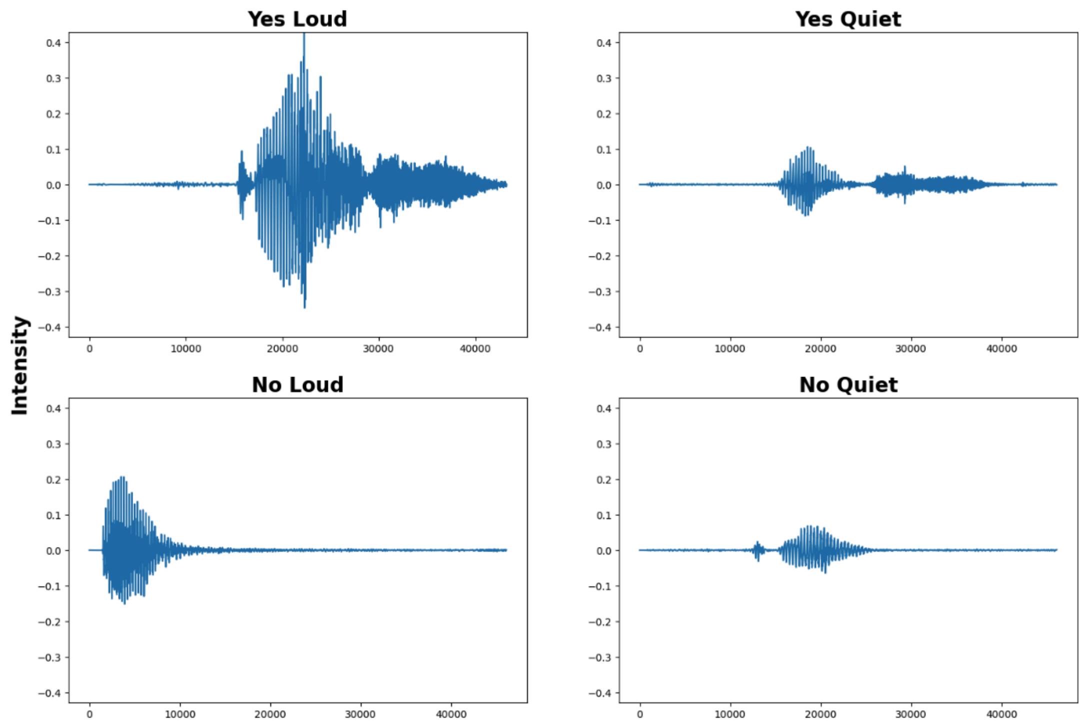
Aneesh Pandoh

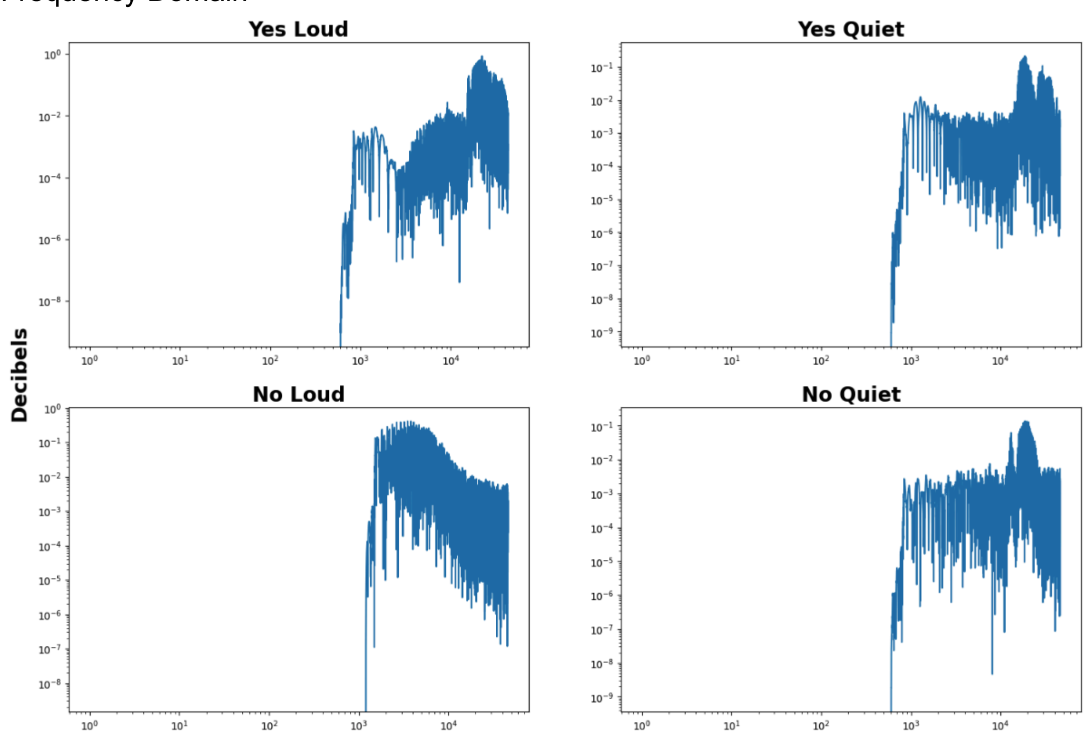
A2

Part 1

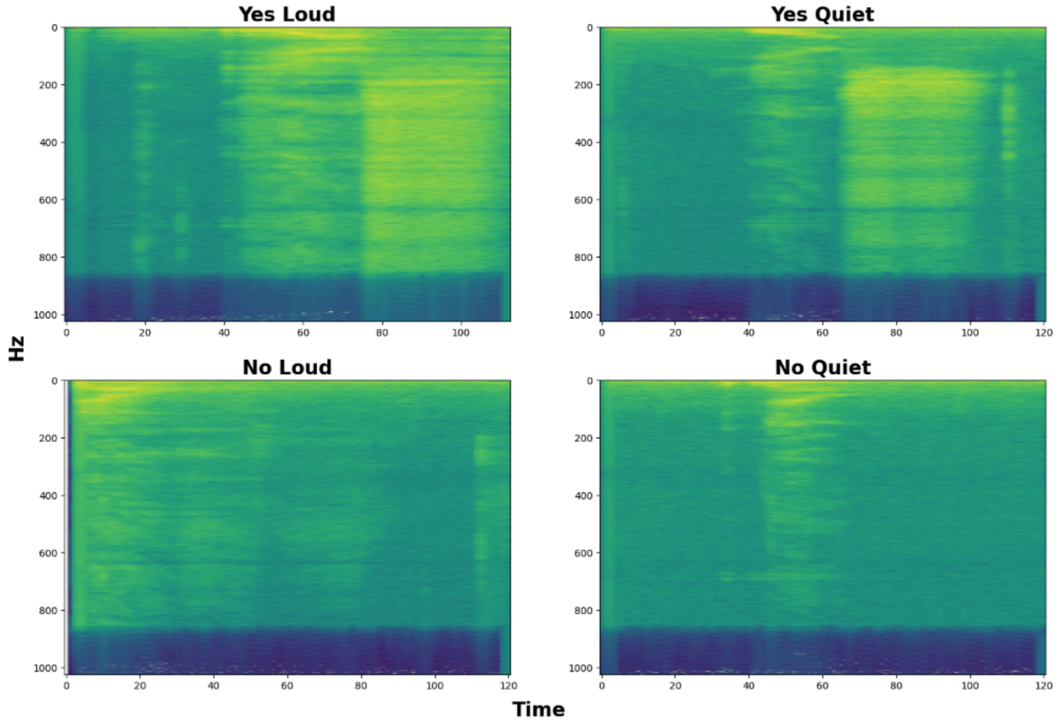
Time Domain



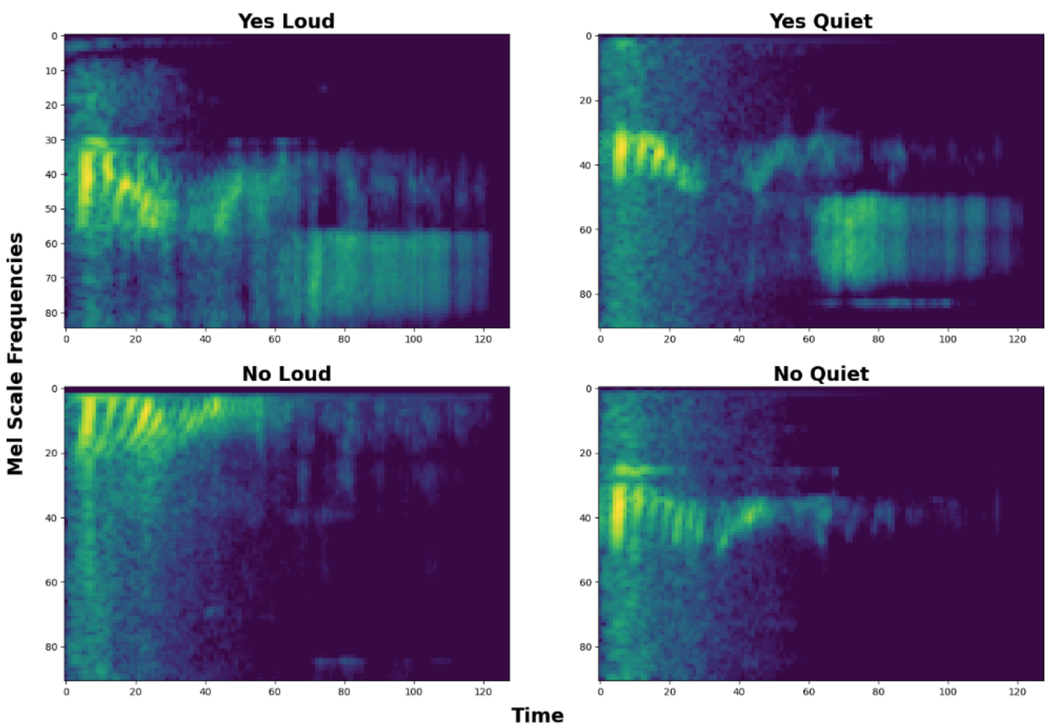
Frequency Domain



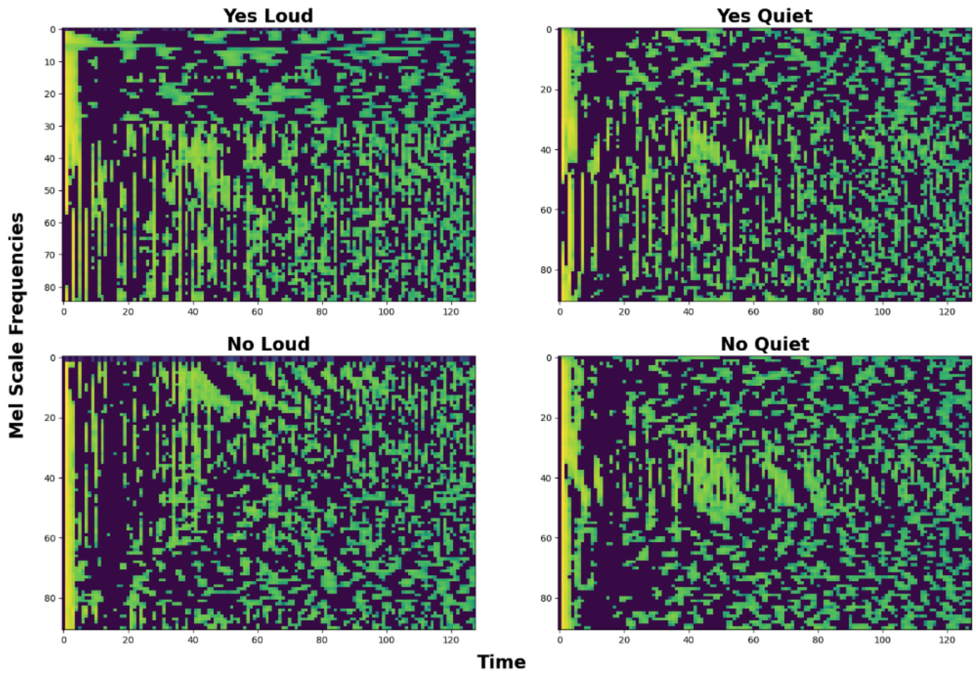
Spectrogram



Mel Spectrogram



MFCC Spectrogram



2.
Audio signals are complex, continuous, and unstructured so we can extract relevant features by preprocessing using graphs like MFCCs and spectrograms which can compress the audio representation and can easily be run through neural networks. It also helps in deciphering between noise from the raw audio data.

3.
A spectrogram represents the frequency content of an audio signal over time using a linear frequency scale. A Mel spectrogram maps the frequency axis to the Mel scale, which mimics human auditory perception, making it more suited for tasks like speech and music analysis. MFCCs are derived from the Mel spectrogram by compressing the frequency information into a set of coefficients, providing a compact representation that is widely used in speech recognition and audio classification.

Part 2

1. Total number of trainable parameters: 0.016652 MB out of 1 MB Flash (1.6% of Flash)
2. Forward memory: 0.159392 MB out of 0.25 MB RAM (63.6% of RAM)
3. Flop Count: 676004 FLOPs

This is about 3x more computationally demanding compared to the [Lightweight Dynamic Convolution Model] which has ~220K FLOPs which is an adaptive deep neural network architecture that generates filter parameters by neural networks. It is a smaller model however, with only 2k trainable parameters compared to the 16k of our network.

4. CPU: 16.344ms | GPU: 327.076us

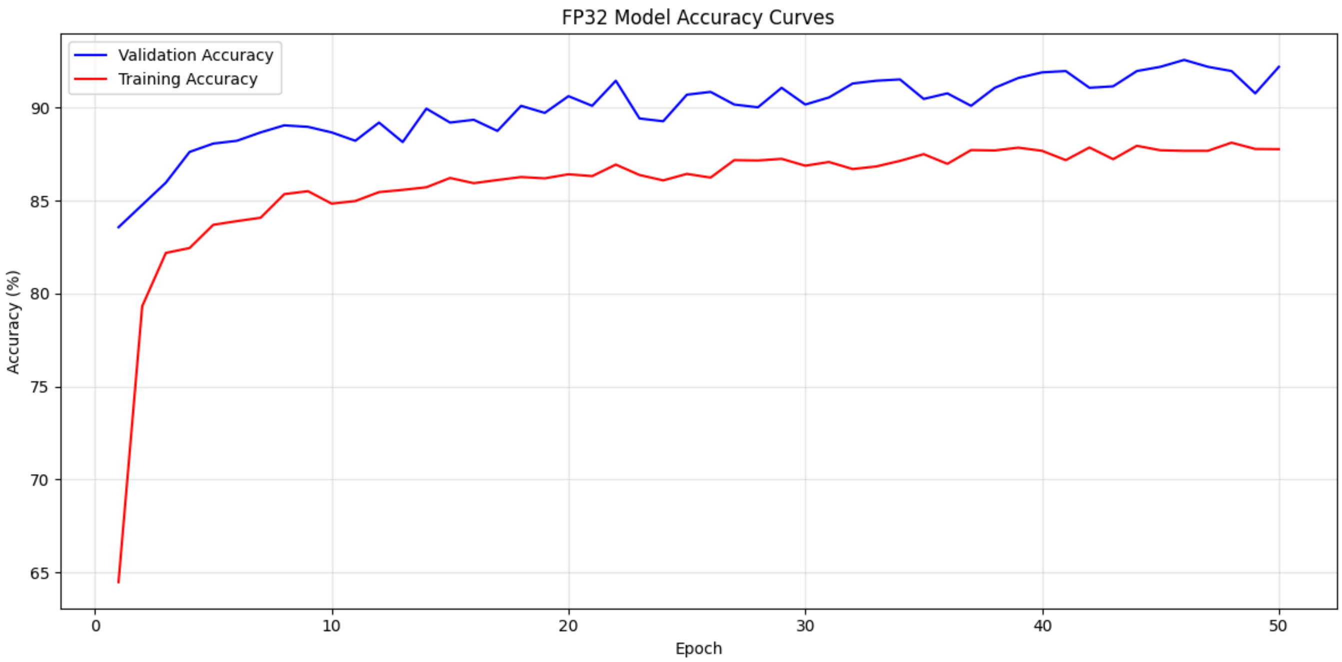
Part 3

1.

testing accuracy for float32 TinyConv

	silence	_unknown_	yes	no	total
#samples	272.000	272.000	419.000	405.000	1368.000
#correct	271.000	215.000	390.000	359.000	1235.000
accuracy	0.996	0.790	0.931	0.886	0.903

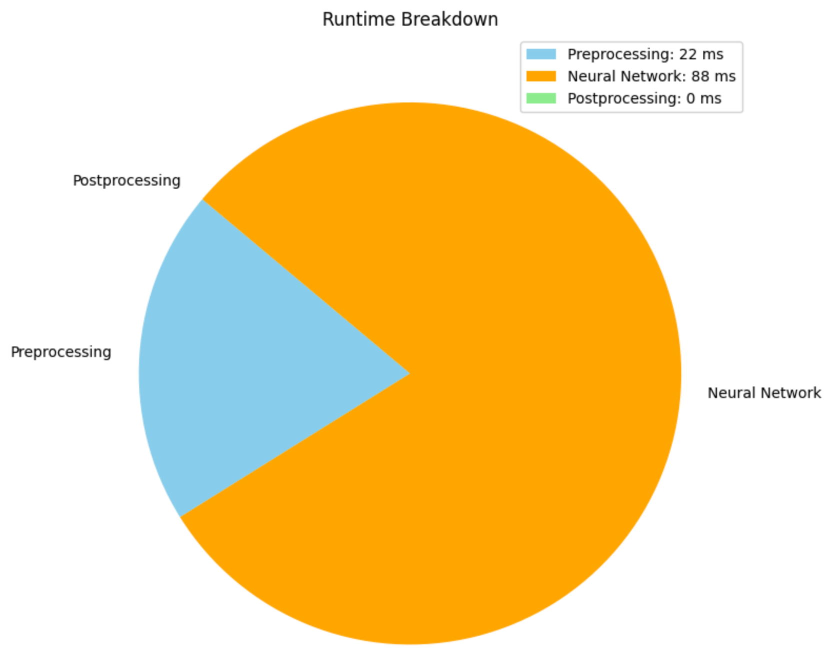
2.



3. Dataset includes 20 core command words with:
   Train size: 10556 Samples | Val size: 1333 Samples | Test size: 1368 Samples
   The dataset contains fairly common english words and also includes background noise information.

Part 4

1.Total time: min 107ms max 113ms avg 109ms
Preprocessing: min 19ms max 24ms avg 22ms
Neural Network: min 88ms max 88ms avg 88ms
Postprocessing: min 0ms max 1ms avg 0ms



The MCU seems about 5.5x times slower than the CPU and ~150x slower than the GPU.

2. Training Accuracy: 90.05% Validation Accuracy: 91.10% In-Field test Accuracy 80%

Actual Word	**Classification**
Yes	Heard Yes (157) @330112ms
No	Heard no (159) @330112ms
Dog	Heard no (162) @332480ms
Cat	Heard unknown (152) @334544ms
Yeast	Heard unknown (162) @336448ms
Neighbor	Heard unknown (151) @338576ms
Never	Heard unknown (153) @341104ms
Bless	Heard Yes (156) @343248ms
Train	Heard unknown (160) @345312ms
Flight	Heard unknown (157) @347600ms

We can see that the real life test shows the model performs worse than expected, this could be due to a combination of factors including sample size, background noise as well as the fact that some of the words tested sound similar to the keywords.

Part 5

1. **[See Github for Full Code]**

```
class ste_round(torch.autograd.Function):
    """
    Straight-through Estimator(STE) for torch.round()
    """
...
    @staticmethod
    def backward(ctx, grad_output):
        # TODO: fill-in (start)
        return grad_output.clone()
        # TODO: fill-in (end)
```

Backwards pass of round just passes through the already calculated gradient

```
def linear_quantize(input, scale, zero_point):
    """
    Quantize floating point input tensor to integers with the given
scaling
    factor and zeropoint.
    Parameters:
    ----------
    input: floating point input tensor to be quantized
    scale: scaling factor for quantization
    zero_pint: shift for quantization
    """
    # TODO: fill-in (start)
    output = input / scale + zero_point
    output = ste_round.apply(output)
    # TODO: fill-in (end)
    return output
```

Quantize using the formula given in class, use the rounding just implemented

```
class SymmetricQuantFunction(torch.autograd.Function):
    """
    Class to quantize the given floating-point values using
quantization with given range and bitwidth.
    """
    @staticmethod
    def forward(ctx, x, k,specified_scale=None, specified_zero_point
      = None):
...
        # TODO: fill-in (start)
        qmin = -(2 ** (k - 1))
        qmax = (2 ** (k - 1)) - 1
        output = ste_round.apply(x / scale).clamp(qmin, qmax)
        # TODO: fill-in (end)
        return output
```

Scale values down and clamp values between ranges using formula given in class to calculate max and min values from bitwidth.

```
class AsymmetricQuantFunction(torch.autograd.Function):
    """
    Class to quantize the given floating-point values using
quantization with given range and bitwidth.
    """

    @staticmethod
    def forward(ctx, x, k,specified_scale=None, specified_zero_point
      = None):
...
        # TODO: fill-in (start)
        qmin = 0
        qmax = (2 ** k) - 1
        output = ste_round.apply(x / scale + zero_point).clamp(qmin,
qmax)
        # TODO: fill-in (end)
        return output
```

Similar implementation to symmetric but make sure values stay positive and instead shift from 0 to 256 instead of -128 to 128 for 8 bit.

```
def get_quantization_params(self, saturation_min, saturation_max):
        """
        Calculate scale and zero point given saturation_min and
saturation_max
        """
        # TODO: fill-in (start)
        epsilon = 1e-8
        with torch.no_grad():
            if self.is_symmetric:
                max_abs = max(abs(saturation_min),
abs(saturation_max))
                # Calculate the scale factor
                n = 2 ** (self.quant_bits - 1) - 1
                scale = (max_abs + epsilon) / n
                # For symmetric quantization, zero point is 0
                zero_point = torch.tensor(0)
            else:
                # For asymmetric quantization
                n = 2 ** self.quant_bits - 1
                # Calculate scale factor
                scale = (saturation_max - saturation_min+epsilon)/ n
                # Calculate zero point
                zero_point = torch.tensor(-ste_round.apply(
                            saturation_min / scale).item())
       # TODO: fill-in (end)
        return scale, zero_point
```

Calculates the underlying scale and zero point values by taking a variation 2^{bit_width} depending on the type of quantization (also add epsilon to avoid NaN error for 2-bit quantization)

```
def quantize_weights_bias(w, qconfig, fake_quantize=False):
    """
    Return quantized weights calculated using given qconfig.
    """
    # TODO: fill-in (start)
    # Get the min and max values from the tensor
    w_min = w.data.min()
    w_max = w.data.max()
    scale, zero_point = qconfig.get_quantization_params(w_min,
w_max)
    # Update qconfig parameters
    qconfig.prev_min = w_min
    qconfig.prev_max = w_max
    qconfig.prev_scale = scale
    qconfig.prev_zeropoint = zero_point
    # Quantize the weights
    w_q = qconfig.quantize_with_params(w, scale, zero_point,
fake_quantize=fake_quantize)
    # TODO: fill-in (end)
    return w_q
```

Basically just calls implementations of functions already created and updates the values of Qconfig class.

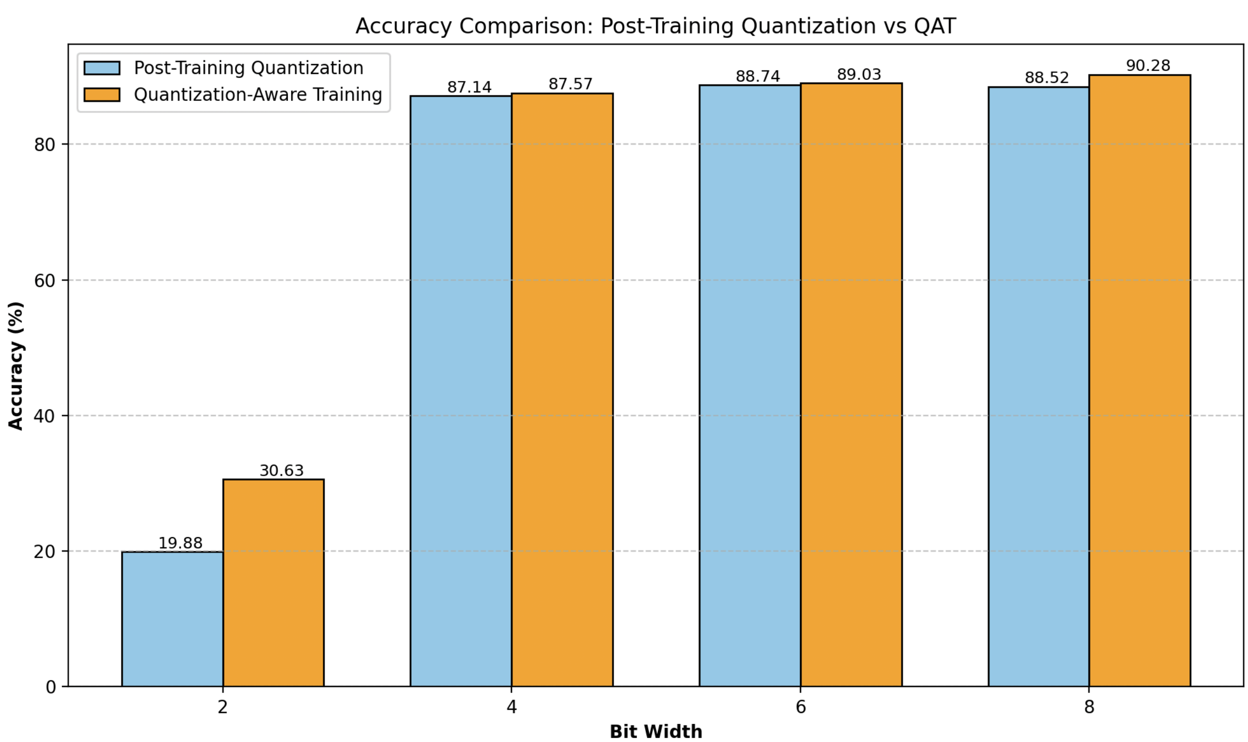
```
def conv2d_linear_quantized(
        module, x, a_qconfig=None, w_qconfig=None, b_qconfig=None):
    """
    Calculate fake quantized output of conv2d or linear layer given
the float module, input tensor, and quantization configurations for
activation, weight, and bias.
    """
 ...
    # TODO: fill-in (start)
    # Quantize input activations
    x_q = quantize_activations(x, a_qconfig, is_moving_avg=True,
fake_quantize=True)
    # Quantize weights
    weight_q = quantize_weights_bias(module.weight, w_qconfig,
fake_quantize=True)
    # Perform computation with quantized values
    if isinstance(module, nn.Conv2d):
        if module.bias is not None:
            # Quantize bias
            bias_q = quantize_weights_bias(module.bias, b_qconfig,
fake_quantize=True)
            # Perform convolution with bias
            y = F.conv2d(
                x_q, weight_q, bias_q, stride=module.stride,
                padding=module.padding,dilation=module.dilation,
```

```
                    groups=module.groups)
            else:
                # Perform convolution without bias
                y = F.conv2d(
                    x_q, weight_q, None, stride=module.stride,
                    padding=module.padding,dilation=module.dilation,
                    groups=module.groups)
        else:  # nn.Linear
            if module.bias is not None:
                # Quantize bias
                bias_q = quantize_weights_bias(module.bias, b_qconfig,
    fake_quantize=True)
                # Perform linear operation with bias
                y = F.linear(x_q, weight_q, bias_q)
            else:
                # Perform linear operation without bias
                y = F.linear(x_q, weight_q)
        # TODO: fill-in (end)
        return y
```

Quantize weights then determines if the layer is a convolution or a fc layer then performs the forward pass of the layer with quantized weights and bias (if exists)

(Keep in mind that this is all fake quantization as the backwards pass is not affected).

2.



We can see that we can see a significant impact for QAT when going to lower bit widths as after an additional. We can see from the MSE values that it makes some of the jumps to lower precision accuracy less drastic.

Part 6

1. **[See Colab for Full Code]**

```
def apply_ln_structured_pruning(model, amount=0.5, n=1):
   """
   Apply Ln structured pruning to the model's Conv2d layers

   Args:
       model: The model to prune
       amount: Percentage of channels to prune (between 0 and 1)
       n: The L-norm to use (1 for L1 norm, 2 for L2 norm)

   Returns:
       model_copy: Copy of the model with pruning masks applied
   """
   model_copy = copy.deepcopy(model)

   # Apply structured pruning to all Conv2d layers
   for name, module in model_copy.named_modules():
       if isinstance(module, nn.Conv2d):
           prune.ln_structured(module,name='weight',amount=amount,n=n,
            dim=0)  # Prune output channels (filters)
       if isinstance(module, nn.Linear):
           prune.ln_structured(
               module, name='weight',amount=amount,n=n,dim=1)  # neurons

   return model_copy
```

Applying ln_structured to convolution and fc layer along dim 0 (channels) and dim 1 (4000 neurons) respectively.

```
def create_reconstructed_model(original_model, pruned_model, amount):
   # Get reference to the original layers and their configurations
   orig_conv1 = original_model.conv
   orig_fc = original_model.fc

   # Extract masks from pruned model (Assumed to be a boolean mask, adjust
as needed)
```

```
    conv1_mask = pruned_model.conv.weight_mask  # Assuming pruned_model has
this attribute
    fc_mask = pruned_model.fc.weight_mask  # Assuming pruned_model has this
attribute

    # Identify which filters to keep in conv1 (output channels)
    conv1_kept_filters = torch.any(conv1_mask != 0, dim=(1, 2, 3))
    conv1_kept_indices =
torch.nonzero(conv1_kept_filters).squeeze().tolist()
    if isinstance(conv1_kept_indices, int):  # Handle single element case
        conv1_kept_indices = [conv1_kept_indices]
    # Number of filters kept in conv1
    new_conv1_out_channels = len(conv1_kept_indices)
    # Identify which neurons to keep in fc (input features)
    fc_kept_neurons = torch.any(fc_mask != 0, dim=0)  # Check which neurons
(columns) remain
    fc_kept_indices = torch.nonzero(fc_kept_neurons).squeeze().tolist()
    if isinstance(fc_kept_indices, int):  # Handle single element case
        fc_kept_indices = [fc_kept_indices]

    # Number of neurons kept in fc
    new_fc_in_features = len(fc_kept_indices)
      new_model = PrunedTinyConv(
            original_model.model_settings,
            1,orig_fc.out_features,  orig_conv1.in_channels
           new_conv1_out_channels).to(device)

    # Copy the unpruned weights from the pruned model to the new model
    with torch.no_grad():
        # Copy conv1 weights for the kept filters
        new_model.conv.weight.data = pruned_model.conv.weight.data
[conv1_kept_indices, :, :, :]
        new_model.conv.bias.data = pruned_model.conv.bias.data
[conv1_kept_indices]
    return new_model
```

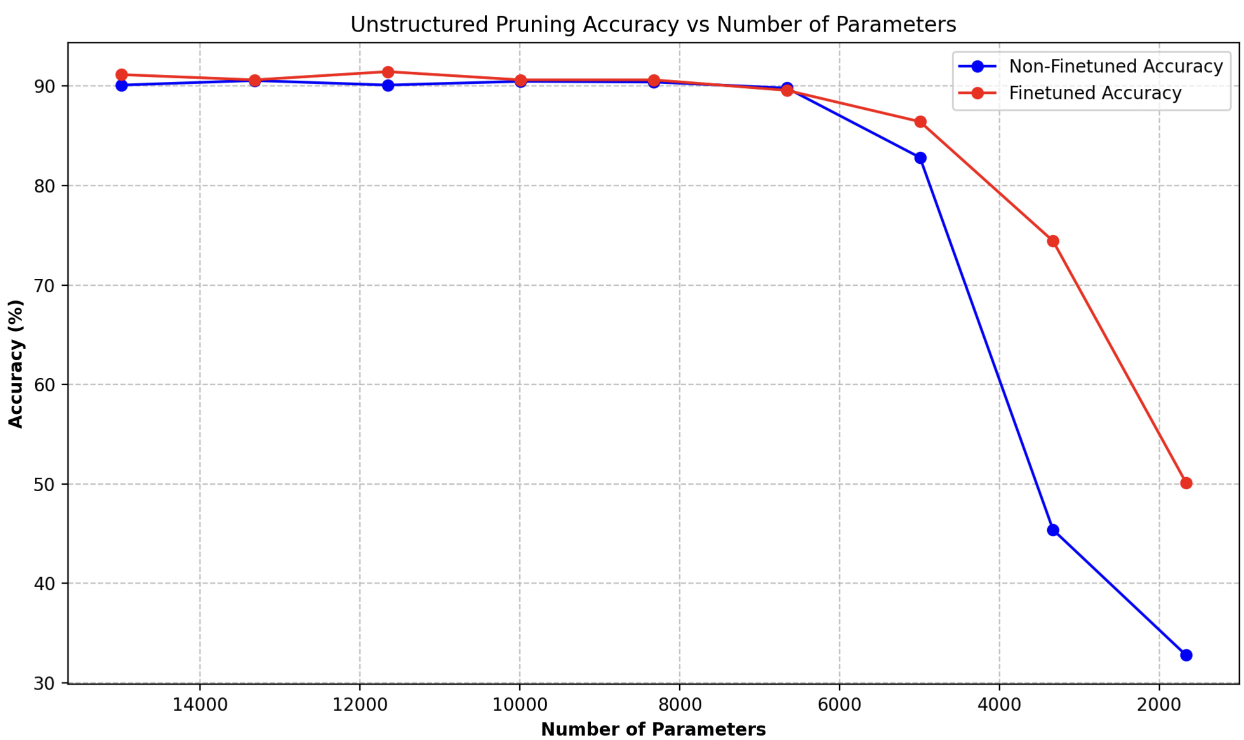
Basically just determines what was not pruned by looking at the pruning mask, then remakes the TinyConv model (Also created new model definition to support this [see colab])

```python
def pruner(tensor, amount):
      # Compute norm of each weight
      l2_norm = tensor.view(tensor.size(0), -1).norm(p=2, dim=1)
      threshold = torch.quantile(l2_norm, amount)
      mask = l2_norm < threshold
      with torch.no_grad():
          tensor = tensor.clone()
          tensor[mask] = 0
      return tensor

parameters_to_prune = (
   (model_fp32.conv, 'weight'),
   (model_fp32.fc, 'weight'),
   (model_fp32.fc, 'bias'))
for module, param_name in parameters_to_prune:
    param = getattr(module, param_name)
    pruner(param, amount)
```

Calculates the norm of the weights and creates a mask based off of it and globally prunes. (Adjusted for L1, L2, L-Inf)

2.

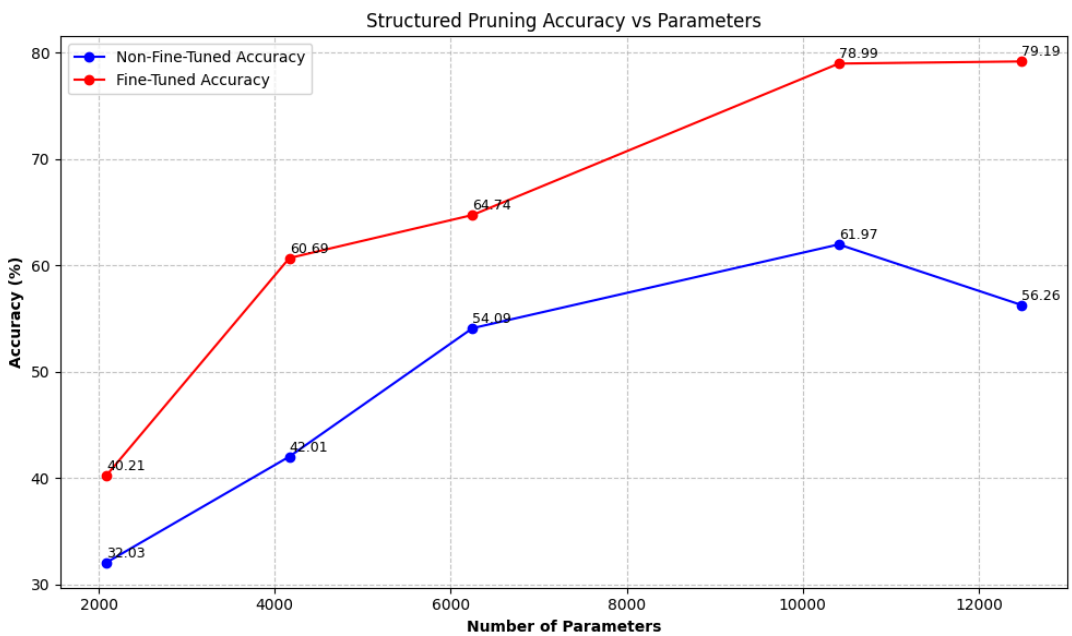


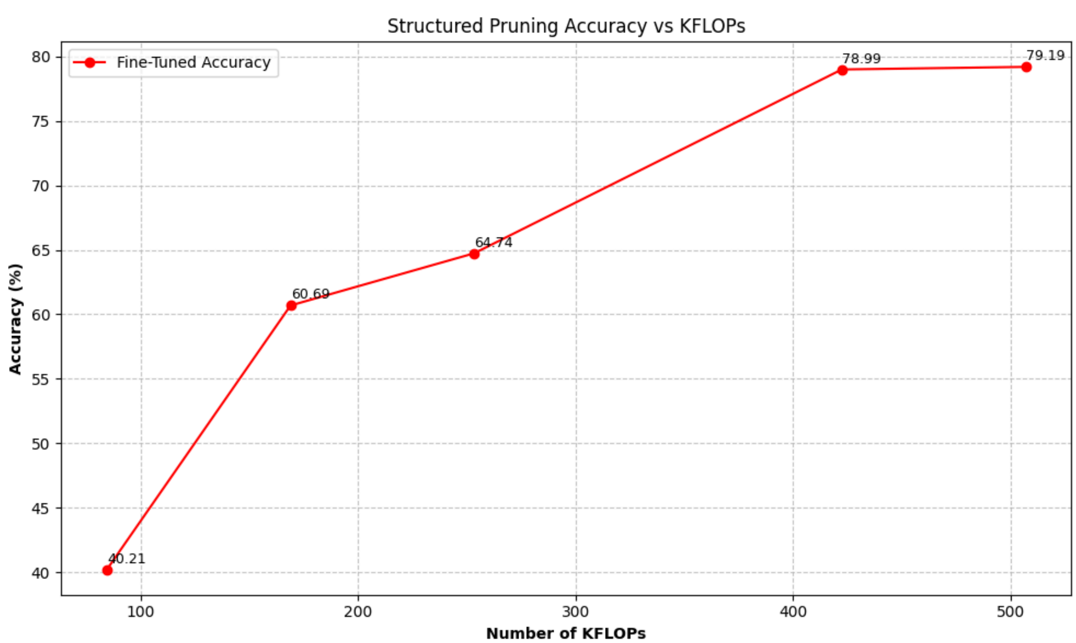
Pruned at increments of 10% from 0 to 90%.

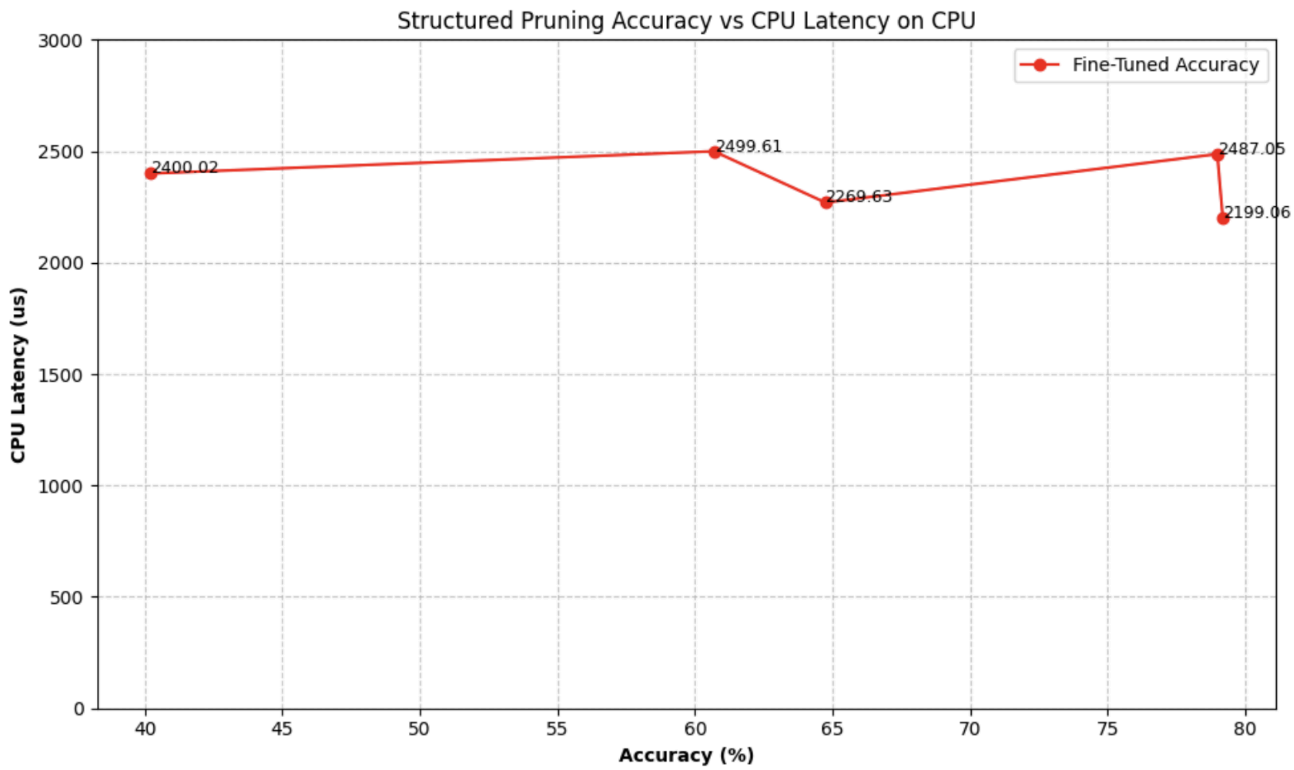
Unstructured pruning improves sparsity by removing the weights that are small in magnitude which allows us to use sparse matrix multiply algorithms as well take up less space. This leads to better cache utilization and less necessary computations.

The different normalizations, L1, L2, and L-Inf seemed to change where we see the accuracy drop off cliff around (70% instead). This is due to the fact that L2 and L-infinity norms are less effective as they dont target the weights with the least magnitude but rather either the highest magnitude weight or more evenly removing the weights.

3.







Latency was calculated over an average of 20 runs each.