Model Pruning and Multi-Point Precision Optimization

Jinal Vyas and Pratik Agarwal

May 2025

Abstract

This project explores the trade-offs between different model optimization techniques—FP16 quantization, INT8 post-training quantization, mixed-precision quantization, and unstructured pruning—applied to a ResNet-18 model trained on ImageNet. We investigate their impact on inference time, model size, and top-5 accuracy. Through experimental analysis, we demonstrate how quantization combined with pruning achieves significant size and latency reductions with manageable accuracy loss, making it suitable for edge inference scenarios. While PyTorch provides built-in pruning utilities via the `torch.nn.utils.prune` module, we found that these functions are incompatible with quantized models, especially those using `torch.ao.quantization` modules. **As a result, we developed custom pruning routines tailored for post-training quantized models (INT8 and mixed-precision), enabling effective magnitude-based weight pruning while preserving the structural integrity of quantized layers.**

1 Introduction

Deep neural networks such as ResNet-18 deliver state-of-the-art performance but suffer from large model sizes and high computational demands. This hinders deployment on resource-constrained devices. To address this, we apply pruning and quantization methods to ResNet-18 and compare their effectiveness. Our objective is to balance accuracy and efficiency.

Motivation: Can we improve inference efficiency using quantization and pruning without drastically sacrificing performance?

Goal: Quantitatively compare model size, inference latency, and accuracy across FP32 baseline, FP16, INT8 quantization, mixed-precision, and pruning-only approaches.

2 Background and Related Work

2.1 Quantization

Quantization is a model compression technique that reduces the numerical precision of a neural network's parameters and activations, typically converting 32-bit floating-point representations (FP32) to lower bit-width formats such as 16-bit floating-point (FP16) or 8-bit integers (INT8). This reduction significantly decreases memory usage, accelerates inference speed, and lowers power consumption—critical advantages for deploying models on edge devices and embedded systems [1, 2].

There are two main categories of quantization:

- **Post-Training Quantization (PTQ):** PTQ is a non-invasive method applied after a model has been fully trained. It statically analyzes a few batches of calibration data to determine quantization parameters such as scale and zero-point, and then converts weights and activations to lower precision. PTQ is easy to implement and requires no additional retraining, making it highly suitable for quick deployment. However, in some cases, it may result in reduced accuracy, especially for low-bit representations like INT8.
- **Quantization-Aware Training (QAT):** QAT simulates quantization effects (such as rounding and clamping) during the forward and backward passes of training. This allows the model to adapt its

parameters in response to quantization-induced perturbations, leading to higher final accuracy, particularly when converting to INT8. However, QAT requires a complete retraining cycle and is computationally more expensive [4].

Why We Chose PTQ: In our project, the primary objective was to analyze and optimize the inference performance of the ResNet-18 architecture. Since our focus was on evaluating model behavior post-deployment—particularly in terms of latency, memory footprint, and parameter sparsity—we prioritized techniques that minimize engineering overhead and compute time. PTQ was therefore chosen over QAT because:

1. Our target was **inference-only optimization**, not improving model accuracy through re-training.
2. ResNet-18 is a relatively shallow network, and PTQ has shown acceptable accuracy drops in such architectures [3].
3. PTQ enabled us to explore different precision formats (INT8, FP16) and layer-specific quantization (mixed precision) without incurring additional training costs.

While QAT would likely yield higher accuracy, the additional training effort was outside the scope of our efficiency-focused objectives. Our use of PTQ across all quantization settings (INT8, FP16, and mixed-precision) provides a practical demonstration of how real-world deployment constraints can shape model compression choices.

2.2 Pruning

Pruning is another widely-used model compression technique that eliminates less significant parameters from a neural network. The primary goal of pruning is to reduce the number of computations and the memory footprint by removing weights or entire structures (such as neurons or filters) that contribute minimally to the model's output. Pruning often induces sparsity in the weight matrices, which can be further exploited by optimized libraries and hardware accelerators [5].

There are two major categories of pruning:

- **Structured Pruning:** In this approach, entire filters, channels, or layers are removed. This results in a smaller model with reduced computational graphs, making it easier to deploy on hardware. However, structured pruning is more aggressive and often results in a larger drop in accuracy, especially without retraining.
- **Unstructured Pruning:** This method removes individual weights based on some importance metric (commonly magnitude). Although the model size in memory might not reduce drastically unless specialized sparse matrix representations are used, unstructured pruning introduces fine-grained sparsity, which modern accelerators and libraries (e.g., NVIDIA's cuSparse or PyTorch sparse kernels) can leverage to improve inference speed [6].

Why We Chose Unstructured Pruning: Our objective was to examine the effect of sparsity on pre-trained models without significant retraining effort. Unstructured pruning offered the following benefits:

1. It is simple to implement using PyTorch's built-in pruning API [7].
2. It allows precise control over sparsity levels (e.g., 10
3. When combined with quantization, unstructured sparsity improves memory access efficiency and may benefit hardware that supports sparse matrix operations.
4. ResNet-18, being a compact network, responds more gracefully to unstructured pruning than structured pruning, which might otherwise overly degrade performance.

By applying magnitude-based L1 pruning to convolutional and fully connected layers, we were able to induce sparsity while maintaining reasonable accuracy across INT8, FP16, and mixed-precision variants.

2.3 Quantization-Then-Pruning Strategy

A key design choice in our methodology was the order of applying compression techniques. Specifically, we chose to perform **quantization first, followed by pruning**. This decision was based on both empirical evidence and practical deployment considerations.

Rationale for Quantization First:

1. Quantization alters the data representation and numerical characteristics of the weights (e.g., introducing zero-points and scaling factors). Pruning these post-quantization weights reflects their true contribution under deployment conditions.

2. Pruning after quantization allows us to exploit quantized inference backends (e.g., QNNPACK) and directly observe the impact of sparsity on quantized operators.

3. Attempting to prune *before* quantization risks pruning weights that may seem unimportant in FP32 but contribute more significantly when quantized due to value rounding and clamping effects [8].

4. PyTorch's quantization pipeline is more robust when operating on dense, full-precision models—performing pruning first could break assumptions within the fake-quantization modules.

This **quantize-then-prune** order ensured better stability, easier calibration, and better compatibility with PyTorch's quantization API. It also allowed us to analyze sparsity and accuracy trade-offs in a deployment-oriented context, where the final model is already quantized.

3 Project Description

3.1 System Overview

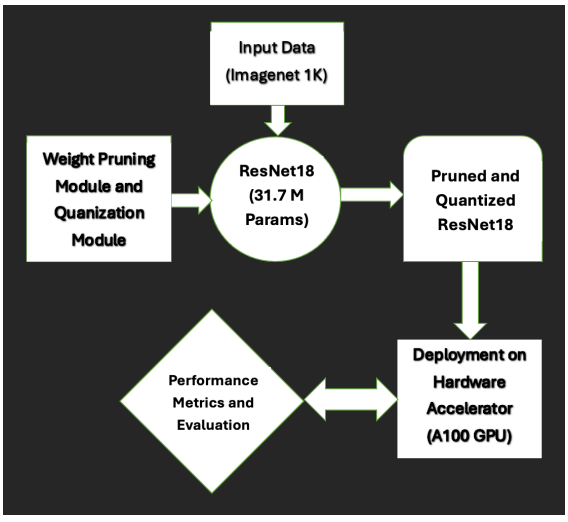

Figure 1: Pipeline for applying quantization and pruning

3.2 Setup

- **Base Model:** torchvision's ResNet-18 pretrained on ImageNet
- **Dataset:** ImageNet validation subset (1000 samples)
- **Tools:** PyTorch, Torchvision, PIL, Matplotlib
- **Hardware:** [MacBook Pro (M2), CPU-based evaluation], [T4 GPU]

4 Methodology

4.1 Baseline (FP32)

The default pretrained ResNet-18 is used without modification. Evaluation is done using:

- Top-5 accuracy
- Inference time (avg over 50 runs)
- Model size on disk

4.2 INT8 Quantization

1. Custom Fusion Strategy

To enable module fusion compatible with PyTorch quantization APIs, the following methods were added to the `BasicBlock` and `ResNet18` classes:

Listing 1: modules_to_fuse for BasicBlock

```
def modules_to_fuse(self, prefix):
    modules_to_fuse_ = []
    modules_to_fuse_.append([f'{prefix}.conv1', f'{prefix}.bn1', f'{prefix}.relu1'
        ])
    modules_to_fuse_.append([f'{prefix}.conv2', f'{prefix}.bn2'])
    if self.downsample:
        modules_to_fuse_.append([f'{prefix}.downsample.0', f'{prefix}.downsample.1
            '])
    return modules_to_fuse_
```

Listing 2: modules_to_fuse for ResNet18

```
def modules_to_fuse(self):
    modules_to_fuse_ = []
    modules_to_fuse_.append(['conv1', 'bn1', 'relu'])
    for layer_str in ['layer1', 'layer2', 'layer3', 'layer4']:
        layer = eval(f'self.{layer_str}')
        for block_nb in range(len(layer)):
            prefix = f'{layer_str}.{block_nb}'
            modules_to_fuse_layer = layer[block_nb].modules_to_fuse(prefix)
            modules_to_fuse_.extend(modules_to_fuse_layer)
    return modules_to_fuse_
```

Note: A line was also added in the `BasicBlock`'s `__init__()` method to support FloatFunctional operations:

```
self.add_relu_FF = torch.ao.nn.quantized.FloatFunctional()
```

Note: A line was also added in the `BasicBlock`'s `forward()` method to support FloatFunctional operations:

```
out = self.add_relu_FF.add_relu(out, identity)
```

2. Quantization Flow

2.1 Module Fusion

```
modules_to_list = model.modules_to_fuse()
model.eval()
fused_model = torch.ao.quantization.fuse_modules(model, modules_to_list)
```

2.2 QConfig for QAT

```
from torch.ao.quantization.fake_quantize import FakeQuantize

activation_qconfig = FakeQuantize.with_args(
    observer=torch.ao.quantization.observer.HistogramObserver.with_args(
        quant_min=0,
        quant_max=255,
        dtype=torch.quint8,
        qscheme=torch.per_tensor_affine,
    )
)

weight_qconfig = FakeQuantize.with_args(
    observer=torch.ao.quantization.observer.PerChannelMinMaxObserver.with_args(
        quant_min=-128,
        quant_max=127,
        dtype=torch.qint8,
        qscheme=torch.per_channel_symmetric,
    )
)

qconfig = torch.quantization.QConfig(activation=activation_qconfig,
                                     weight=weight_qconfig)
fused_model.qconfig = qconfig
```

2.3 Prepare Fake Quantization Model

```
fused_model.train()
fake_quant_model = torch.ao.quantization.prepare_qat(fused_model)
```

2.4 Evaluate Pre-Conversion Model

```
print("\nFake quant - PTQ")
evaluate(fake_quant_model, 'cpu')

# Freeze quantization parameters
fake_quant_model.apply(torch.ao.quantization.fake_quantize.disable_observer)

print("\nFake quant - post-PTQ")
evaluate(fake_quant_model, 'cpu')
```

2.5 Convert to Int8 Model

```
converted_model = torch.ao.quantization.convert(fake_quant_model)
print("\nConverted model")
evaluate(converted_model, 'cpu')
```

4.3 Pruning of Quantized (INT8) Layers

After applying post-training quantization (PTQ) to ResNet-18, we perform **custom unstructured pruning** on the resulting INT8 model. This involves directly modifying the quantized weights of `QuantizedConv2d`, `QuantizedConvReLU2d`, and `QuantizedLinear` layers.

Approach

1. **Access Quantized Parameters:**
   - For each quantized convolutional or fully connected layer, we retrieve the quantized weight tensor, scale, and zero point.
   - For layers like `QuantizedLinear`, weights and biases are extracted using the `_weight_bias()` method.
2. **Dequantization for Magnitude Comparison:**
   - Since weights are stored as `qint8`, we must dequantize them to apply magnitude-based pruning.
   - A helper function, `custom_abs()`, is used to compute:

$$w_{\text{dequant}} = \text{scale} \times (\text{int8\_weight} - \text{zero\_point})$$

   - Absolute values of dequantized weights are computed and re-quantized to remain in INT8 format.
3. **Magnitude-Based Pruning:**
   - We flatten the weight tensor and sort it by absolute magnitude.
   - A fixed proportion (30%) of the smallest weights for convolutional layers and (10%) for fully connected layers are set to zero, implementing unstructured sparsity.
   - The pruned tensor is reshaped to its original shape.
4. **Reassignment to the Module:**
   - The pruned weights are assigned back to the layer using `set_weight_bias()`.

Handling the Fully Connected (FC) Layer

- The FC layer often uses *per-channel quantization*, providing a different scale and zero point per output channel.
- To simplify pruning, we approximate this by taking the mean of all per-channel scales and zero points, enabling a consistent pruning strategy.

Summary

This approach allows us to apply unstructured magnitude-based pruning directly to quantized weights while preserving the INT8 format. It avoids the need for retraining or recalibration and offers memory and potential inference time benefits, especially on sparsity-aware hardware.

```
for name, module in converted_model.named_modules():
    if "conv1" in name or "conv2" in name:
        print(module)
        weight = module.weight()
        scale = module.scale
        zero_point = module.zero_point

        # Perform pruning: Setting a fraction of weights to zero (e.g., 30%)
        pruning_amount = 0.3
        num_weights = weight.numel()
        num_pruned_weights = int(pruning_amount * num_weights)

        # Flatten the weights, sort by absolute value, and zero out the smallest
        zero_point = max(min(zero_point, 127), -128)
        weight = custom_abs(weight, zero_point, scale)
        flattened_weights = weight.reshape(-1)
```

```
        sorted_indices = torch.argsort(flattened_weights)

        # # Prune the smallest weights
        pruned_weights = flattened_weights.clone()
        pruned_weights[sorted_indices[:num_pruned_weights]] = 0

        # # Reshape pruned weights back to the original shape
        pruned_weights = pruned_weights.view_as(weight)

        # print(module.weight())
        # print(module.bias())
        bias = module.bias()
        module.set_weight_bias(pruned_weights, bias)

    elif "fc" in name:
        weight, bias = module._weight_bias()
        try:
            scale = weight.q_scale()
            zero_point = weight.q_zero_point()
        except:
            scale = weight.q_per_channel_scales()
            zero_point = weight.q_per_channel_zero_points()
            scale = float(scale.mean())
            zero_point = int(zero_point.float().mean().round())

        # Perform pruning: Setting a fraction of weights to zero (e.g., 30%)
        pruning_amount = 0.1
        num_weights = weight.numel()
        num_pruned_weights = int(pruning_amount * num_weights)

        # Flatten the weights, sort by absolute value, and zero out the smallest
        zero_point = max(min(zero_point, 127), -128)
        weight = custom_abs(weight, zero_point, scale)
        flattened_weights = weight.reshape(-1)
        sorted_indices = torch.argsort(flattened_weights)

        # # Prune the smallest weights
        pruned_weights = flattened_weights.clone()
        pruned_weights[sorted_indices[:num_pruned_weights]] = 0

        # # Reshape pruned weights back to the original shape
        pruned_weights = pruned_weights.view_as(weight)

        # print(module.weight())
        # print(module.bias())
        module.set_weight_bias(pruned_weights, bias)
```

4.4 FP16 Quantization

All weights and activations are cast to FP16 using .half(). Pruning is applied post-quantization using custom threshold-based zeroing.

4.5 Pruning Strategy for FP16 Model

In the case of the FP16 quantized model, we applied **magnitude-based unstructured pruning** directly to the convolutional and fully connected layers. The process is as follows:

- We iterate through each module in the model and select layers whose names include `conv1`, `conv2`, or `fc`, which correspond to convolutional and fully connected layers in ResNet-18.
- For each selected layer, the weights are first converted to their absolute values using `torch.abs()`, so that pruning is based purely on magnitude, without regard to sign.
- These absolute weight values are flattened into a 1D tensor and sorted in ascending order.
- A fixed pruning ratio (30% in our case) is then applied by identifying the lowest-magnitude weights and setting them to zero.
- The pruned weight vector is reshaped to match the original weight tensor's shape and reassigned to the layer using `torch.nn.Parameter`.

This method effectively introduces sparsity in the network without relying on structured patterns or masks. Unlike PyTorch's built-in pruning modules, this manual approach directly rewrites the weight tensors, making it lightweight and compatible with mixed-precision models such as FP16.

While this pruning technique may cause slight degradation in accuracy, it significantly reduces the number of active parameters and can lead to improved inference speed and lower memory footprint on supported hardware accelerators.

```
for name, module in converted_model.named_modules():
    if "conv1" in name or "conv2" in name or "fc" in name:
        weight = module.weight
        pruning_amount = 0.3
        num_weights = weight.numel()
        num_pruned_weights = int(pruning_amount * num_weights)
        weight = torch.abs(weight)
        flattened_weights = weight.reshape(-1)
        sorted_indices = torch.argsort(flattened_weights)
        pruned_weights = flattened_weights.clone()
        pruned_weights[sorted_indices[:num_pruned_weights]] = 0
        pruned_weights = pruned_weights.view_as(weight)
        module.weight = torch.nn.Parameter(pruned_weights)
```

4.6 Mixed Precision Quantization

INT8 quantization is applied to all layers except the final fully connected (FC) layer, which is converted back to FP32. This enables accuracy recovery while retaining low inference cost.

```
fc_q = converted_model.fc
fc_fp32 = nn.Linear(fc_q.in_features, fc_q.out_features)
fc_fp32.weight.data = fc_q._weight_bias()[0].dequantize()
fc_fp32.bias.data = fc_q._weight_bias()[1]
converted_model.fc = nn.Sequential(CustomDeQuantize(), fc_fp32)
```

4.7 Pruning Strategy for Mixed Precision Model (INT8 + FP32)

In the mixed precision setup, the model is quantized such that all convolutional layers are in INT8 format while the final fully connected (FC) layer is retained in full precision (FP32). This hybrid design allows the model to benefit from the memory and speed advantages of INT8 quantization in most layers, while preserving accuracy-critical computations in the FC layer using higher precision.

To further reduce the model's redundancy, we apply pruning separately to the INT8 and FP32 components:

- **Pruning INT8 Layers:** For quantized convolutional layers (e.g., `conv1`, `conv2`), we first dequantize the weight tensor using the provided scale and zero-point values. We then compute the absolute values of the dequantized weights and flatten them to identify low-magnitude weights. The bottom 30% of weights (in terms of absolute value) are pruned (set to zero), and the tensor is reshaped to its original form before being reassigned using the `set_weight_bias()` method. This approach ensures that the sparsity pattern is directly embedded into the quantized model.
- **Pruning the FP32 FC Layer:** The final `fc` layer, which remains in FP32 and is wrapped in a `nn.Sequential` block, is pruned using PyTorch's `l1_unstructured` method. This technique removes 10% of weights with the lowest L1 norm, effectively reducing parameter count while preserving the high-precision behavior in the critical classification head. The pruning reparameterization is made permanent using `prune.remove()`.

This mixed strategy combines the compression benefits of INT8 inference with the stability of FP32 outputs, and the sparsity introduced by pruning further enhances model compactness and potentially inference speed on sparse-aware hardware.

```
for name, module in converted_model.named_modules():
    if "conv1" in name or "conv2" in name:
        weight = module.weight()
        scale = module.scale
        zero_point = max(min(module.zero_point, 127), -128)
        weight = custom_abs(weight, zero_point, scale)
        flattened = weight.reshape(-1)
        num_to_prune = int(0.3 * flattened.numel())
        flattened[torch.argsort(flattened)[:num_to_prune]] = 0
        pruned_weight = flattened.view_as(weight)
        module.set_weight_bias(pruned_weight, module.bias())

    elif "fc" in name and isinstance(module, nn.Sequential) and isinstance(module
        [1], nn.Linear):
        prune.l1_unstructured(module[1], name="weight", amount=0.1)
        prune.remove(module[1], "weight")
```

4.8 Unstructured Pruning Only

Pruning is applied using PyTorch's prune.l1_unstructured() on all Conv2D layers:

```
prune.l1_unstructured(module, name='weight', amount=0.3)
```

5 Results and Discussion

Table 1: Evaluation Metrics for FP32 Model

Device	Inference Time (ms)	Model Size (MB)	Top-5 Accuracy (%)
CPU	14.36 ms	46.83 MB	89%
GPU	6.82 ms	46.83 MB	89.21%

Table 2: Evaluation Metrics for INT8 Model - Custom Quantization and Pruning

Device	Inference Time (ms)	Model Size (MB)	Top-5 Accuracy (%)
CPU	782 ms	11.83 MB	76.45%
GPU	3.33 ms	11.81 MB	77.89%

Table 3: Evaluation Metrics for FP16 Model - Custom Quantization and Pruning

Device	Inference Time (ms)	Model Size (MB)	Top-5 Accuracy (%)
CPU	16.55 ms	23.43 MB	81.98%
GPU	4.86 ms	23.22 MB	81.11%

Table 4: Evaluation Metrics for Mixed Precision Model (INT8 + FP32 FC Layer) - Custom Quantization and Pruning

Device	Inference Time (ms)	Model Size (MB)	Top-5 Accuracy (%)
CPU	80.82 ms	14.29 MB	79.234%
GPU	3.84	14.3	79.9%

Table 5: Evaluation Metrics for Just Pruned Model (Using PyTorch Function)

Device	Inference Time (ms)	Model Size (MB)	Top-5 Accuracy (%)
CPU	12.11 ms	44.96 MB	89.54%
GPU	3.33 ms	45.1 MB	89.11%

This section provides a comparative evaluation of different quantization and pruning strategies applied to the ResNet-18 architecture. Each configuration was assessed across three primary metrics: inference time, model size, and Top-5 accuracy. Both CPU and GPU performance are reported for a comprehensive view.

5.1 Evaluation Summary

Table 5 through Table 4 provide a breakdown of results across FP32 (baseline), INT8 quantized and pruned, FP16 quantized and pruned, mixed precision (INT8 + FP32 FC), and a pruned-only model using PyTorch's built-in unstructured pruning.

5.2 Inference Time

Inference time showed significant variance between CPU and GPU:

- **FP32:** GPU inference (6.82 ms) was more than twice as fast as CPU (14.36 ms).
- **INT8:** Surprisingly, INT8 on CPU was significantly slower (782 ms), possibly due to inefficient CPU kernel support or batch dequantization overhead. GPU inference, however, was the fastest (3.33 ms).
- **FP16:** As expected, FP16 inference was faster than FP32, especially on GPU (4.86 ms), leveraging half-precision compute units.
- **Mixed Precision:** With only the FC layer in FP32 and rest quantized to INT8, the model achieves GPU inference time close to pure INT8 (3.84 ms), while being significantly faster than CPU (80.82 ms).
- **Pruned-Only:** Despite retaining FP32 precision, pruning alone achieves competitive speeds (12.11 ms on CPU, 3.33 ms on GPU).

5.3 Model Size

- The **baseline FP32 model** consumes 46.83 MB.
- **INT8 quantization and pruning** reduces model size to **11.8 MB**, a 75% compression.
- **FP16 model** is approximately 50% smaller than FP32.

- **Mixed precision** (INT8 + FP32 for FC layer) occupies 14.3 MB, slightly larger than INT8 but smaller than FP16.
- **Pruned-only model** shows a minor reduction, indicating pruning alone doesn't affect model size without serialization-aware techniques.

5.4 Top-5 Accuracy

- **FP32 and pruned-only models** maintained the highest accuracy, 89%.
- **FP16** yielded 81.9%, indicating a minor trade-off.
- **Mixed precision** achieved 79.9%, suggesting preserving FP32 in the FC layer helps regain some lost accuracy compared to full INT8.
- **INT8** showed the most degradation (76.5% on CPU), though still acceptable in many latency-sensitive applications.

5.5 INT8 Fully Connected Layer Weight Distribution Before and After Pruning

To visualize the effect of pruning on the quantized fully connected (FC) layer, we present weight distribution histograms before and after pruning.

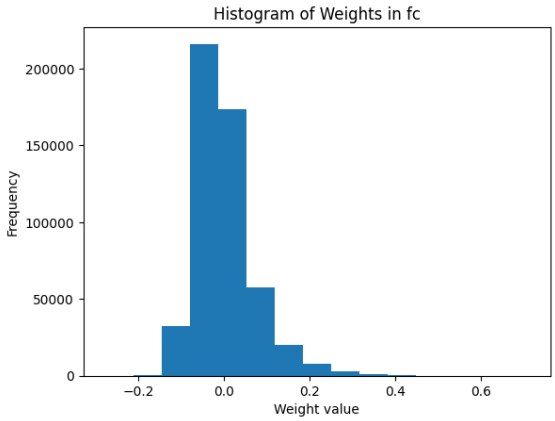

Figure 2: Histogram of FC Layer Weights (INT8, Post Quantization, Pre-Pruning)

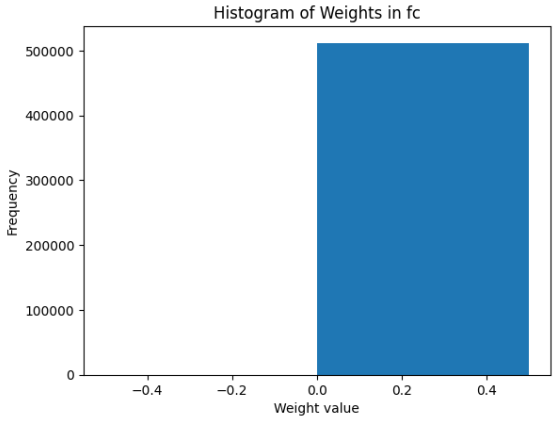

Figure 3: Histogram of FC Layer Weights (INT8, Post Quantization, Post-Pruning)

As shown in Figure 2, the weight distribution of the FC layer after quantization retains a full range of non-zero values. This implies that although quantization reduced numerical precision, the FC layer still preserved the representational richness of its floating-point counterpart.

In contrast, Figure 3 illustrates the post-pruning state of the same FC layer, where approximately 10% of the weights (those with the smallest absolute magnitudes) were set to zero. As observed, the resulting distribution is now heavily dominated by zero values, demonstrating the sparsification effect induced by unstructured magnitude-based pruning.

This comparison clearly highlights how pruning complements quantization by reducing the effective number of parameters, thus enabling better memory and computational efficiency while maintaining acceptable inference performance.

5.6 FC Layer Weight Distribution in FP16 Model: Pre- and Post-Pruning

Figure 4 shows the distribution of weights in the fully connected (FC) layer of the ResNet-18 model after converting all layers to FP16 precision but before applying any pruning. The weight values are centered around zero, with the majority lying within the range of $[-0.2, 0.4]$. This indicates a fairly typical spread seen in pretrained models, with a notable concentration of low-magnitude weights.

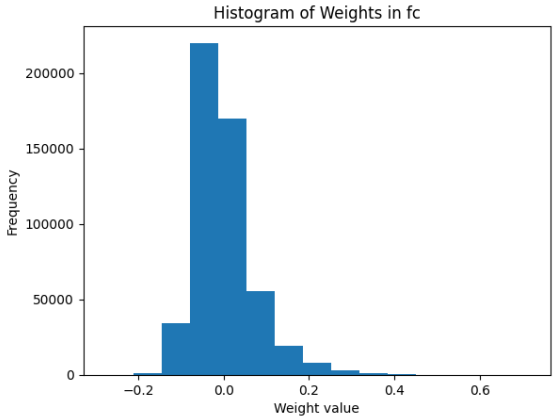

Figure 4: Histogram of FC Layer Weights (FP16, Post Quantization, Pre-Pruning)

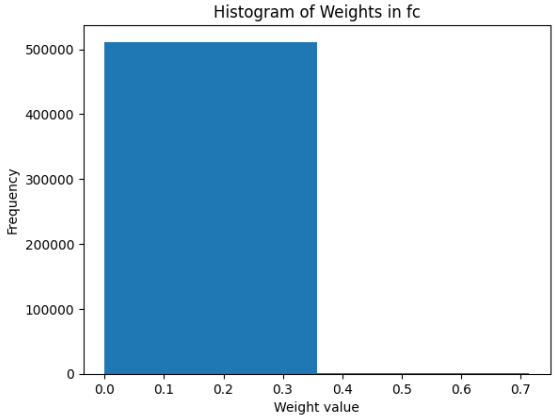

Figure 5: Histogram of FC Layer Weights (FP16, Post Quantization, Post-Pruning)

After applying unstructured magnitude-based pruning at a pruning ratio of 30% (i.e., zeroing out the lowest 30% of absolute weight values), we observe a significant shift in the histogram shown in Figure 5. The

distribution now displays a large spike at zero, reflecting the high degree of sparsity induced by pruning. This transformation highlights how pruning not only reduces the parameter count but also simplifies the distribution, which can be exploited for further compression or efficient inference strategies on hardware that supports sparse matrix operations.

This comparison validates the effectiveness of unstructured pruning in reducing weight redundancy while maintaining the essential signal in the model, particularly when applied to quantized formats like FP16.

5.7 Weight Distribution in Mixed Precision Model (FP32 FC + INT8 Rest)

In the mixed precision setup, the ResNet-18 model was quantized such that all layers were converted to INT8, except the final fully connected (FC) layer which was kept in full precision (FP32). This approach leverages the memory and speed efficiency of quantization while preserving critical output precision.

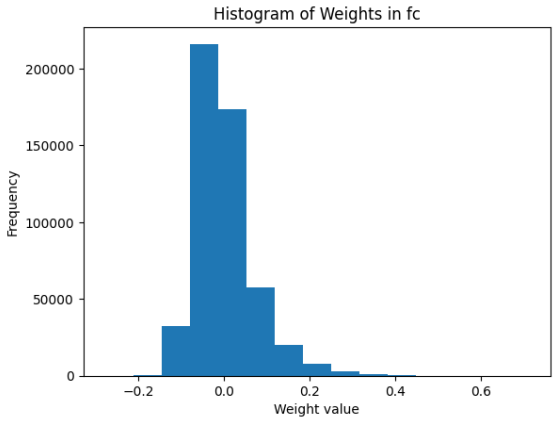

Figure 6: Histogram of FC Layer Weights (Mixed Precision, Post Quantization, Pre-Pruning)

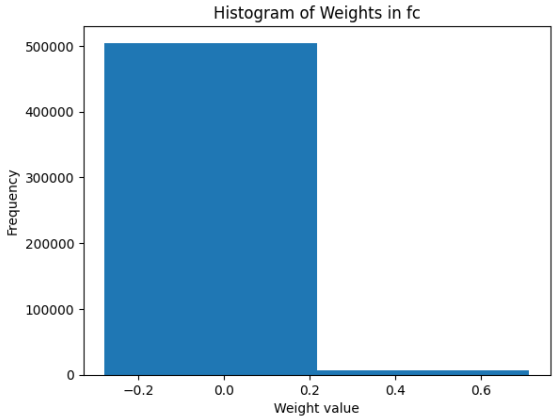

Figure 7: Histogram of FC Layer Weights (Mixed Precision, Post Quantization, Post-Pruning)

This shows the weight distribution of the fully connected layer in the mixed precision model before and after pruning. After quantization, the weight distribution is dense with most values clustered near zero. Once 10% of the weights with the smallest L1 magnitude are pruned, we observe a significant increase in the count of zero-valued weights, confirming successful unstructured pruning. The pruning was applied only to the FP32 FC layer using PyTorch's `l1_unstructured()` function, while the rest of the network remained quantized.

This setup allows us to retain relatively high prediction accuracy while significantly reducing inference time and memory usage.

5.8 Conclusion

The comparative analysis reveals that while **pruning alone** preserves most of the model's original accuracy, it has a limited impact on reducing model size or improving inference time. In contrast, **quantization combined with pruning**—especially using INT8 or mixed precision strategies—results in significant improvements in both **model compression** and **inference efficiency**, particularly when deployed on GPU hardware.

Although there is a noticeable drop in Top-5 accuracy for INT8 and mixed-precision models compared to full-precision FP32, this trade-off is often acceptable in latency-critical or edge applications. Furthermore, the drop in accuracy observed post-quantization and pruning can be **effectively mitigated through fine-tuning**, which was not performed in this project but is a well-documented technique in quantization-aware training (QAT) workflows.

Overall, we find that **quantization + pruning offers a more holistic and impactful optimization** for model deployment than pruning alone, striking a favorable balance between accuracy, speed, and memory footprint.

References

[1] Jacob, Benoit, et al. "Quantization and training of neural networks for efficient integer-arithmetic-only inference." CVPR. 2018.

[2] Krishnamoorthi, Raghuraman. "Quantizing deep convolutional networks for efficient inference: A whitepaper." arXiv preprint arXiv:1806.08342 (2018).

[3] Banner, Ron, et al. "Post-training 4-bit quantization of convolutional networks for rapid-deployment." NeurIPS. 2018.

[4] Esser, Steven K., et al. "Learned step size quantization." ICLR. 2020.

[5] Han, Song, et al. "Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding." ICLR. 2016.

[6] Zhu, Michael, and Suyog Gupta. "To prune, or not to prune: exploring the efficacy of pruning for model compression." arXiv preprint arXiv:1710.01878 (2017).

[7] Gale, Trevor, et al. "The state of sparsity in deep neural networks." arXiv preprint arXiv:1902.09574 (2019).

[8] Li, Hao, et al. "Pruning filters for efficient convnets." ICLR. 2017.