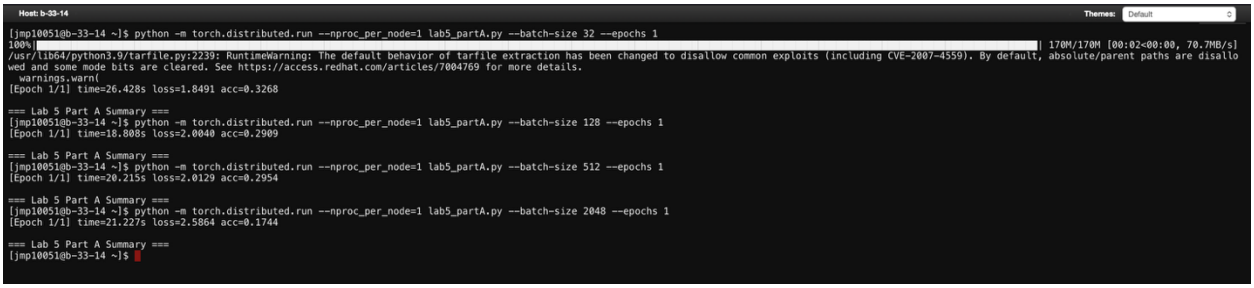
Homework Assignment 5

Part A

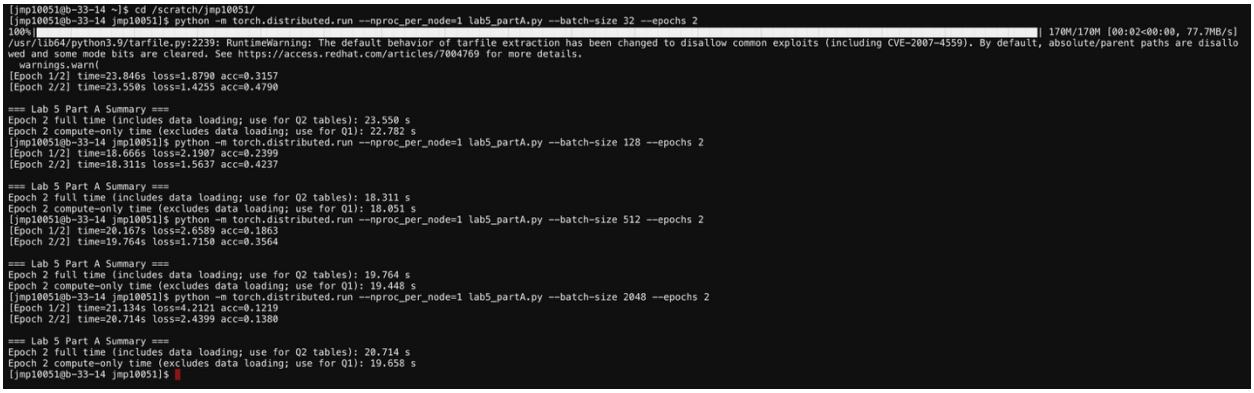
All Part A experiments (DDP training, timing, multi-GPU runs, and communication analysis) were implemented in the submitted file: Jmp10051_Lab5_partA.py

Q1

Epoch 1



Epoch 2



The epoch-2 compute-only training time was measured on a single GPU while increasing the batch size by a factor of four (32 → 128 → 512 → 2048). Batch size 8192 resulted in a CUDA out-of-memory error, so 2048 is the largest batch size that fits on one GPU.
Summary Table : (excludes data loading)

Batch Size	Compute-Only Epoch 2 Time (s)
32	22.782
128	18.051
512	19.448
2048	19.658

Batch Size	Compute-Only Epoch 2 Time (s)
8192	Out-of-memory (does not fit)

On a single NVIDIA A40 GPU (g2-standard-48 node on Greene), it takes 22.G6 seconds to train one epoch (excluding data-loading) using mini-batch size 32, 18.05 seconds using batch size 128, 19.4 seconds using batch size 512, and 19.6 seconds using batch size 2048. Increasing the batch size from 32 → 128 improves throughput because the GPU becomes better utilized and kernel overhead is amortized.
For larger batches (512 and 2048), the epoch time flattens around ~19 seconds, indicating that the GPU reaches a compute/memory bandwidth saturation point. Batch size 81G2 results in a CUDA out-of-memory error, so 2048 is the largest batch size that fits on a single GPU.

Q2

2 GPUs

[jmp10051@b-33-14 jmp10051]$ python -m torch.distributed.run --nproc_per_node=2 lab5_partA.py --batch-size 32 --epochs 2
[Epoch 1/2] time=16.704s loss=1.8432 acc=0.3226
[Epoch 2/2] time=16.164s loss=1.3433 acc=0.5115
=== Lab 5 Part A Summary ===
Epoch 2 full time (includes data loading; use for Q2 tables): 16.164 s
Epoch 2 compute-only time (excludes data loading; use for Q1): 15.775 s
[jmp10051@b-33-14 jmp10051]$ python -m torch.distributed.run --nproc_per_node=2 lab5_partA.py --batch-size 128 --epochs 2
[Epoch 1/2] time=10.360s loss=1.9076 acc=0.3232
[Epoch 2/2] time=9.849s loss=1.4073 acc=0.4828
=== Lab 5 Part A Summary ===
Epoch 2 full time (includes data loading; use for Q2 tables): 9.849 s
Epoch 2 compute-only time (excludes data loading; use for Q1): 9.637 s
[jmp10051@b-33-14 jmp10051]$ python -m torch.distributed.run --nproc_per_node=2 lab5_partA.py --batch-size 512 --epochs 2
[Epoch 1/2] time=10.457s loss=2.9319 acc=0.1794
[Epoch 2/2] time=10.012s loss=1.8745 acc=0.3021
=== Lab 5 Part A Summary ===
Epoch 2 full time (includes data loading; use for Q2 tables): 10.012 s
Epoch 2 compute-only time (excludes data loading; use for Q1): 9.708 s

4 GPUs

[jmp10051@b-33-14 jmp10051]$ python -m torch.distributed.run --nproc_per_node=4 lab5_partA.py --batch-size 32 --epochs 2
[Epoch 1/2] time=9.623s loss=1.9499 acc=0.2972

[Epoch 2/2] time=9.060s loss=1.4588 acc=0.4652
=== Lab 5 Part A Summary ===
Epoch 2 full time (includes data loading; use for Q2 tables): 9.060 s
Epoch 2 compute-only time (excludes data loading; use for Q1): 8.811 s
[jmp10051@b-33-14 jmp10051]$ python -m torch.distributed.run --nproc_per_node=4 lab5_partA.py --batch-size 128 --epochs 2
[Epoch 1/2] time=5.586s loss=2.0906 acc=0.2740
[Epoch 2/2] time=5.023s loss=1.5321 acc=0.4318
=== Lab 5 Part A Summary ===
Epoch 2 full time (includes data loading; use for Q2 tables): 5.023 s
Epoch 2 compute-only time (excludes data loading; use for Q1): 4.852 s
[jmp10051@b-33-14 jmp10051]$ python -m torch.distributed.run --nproc_per_node=4 lab5_partA.py --batch-size 512 --epochs 2
[Epoch 1/2] time=5.707s loss=2.6404 acc=0.1935
[Epoch 2/2] time=5.160s loss=1.7537 acc=0.3244
=== Lab 5 Part A Summary ===
Epoch 2 full time (includes data loading; use for Q2 tables): 5.160 s
Epoch 2 compute-only time (excludes data loading; use for Q1): 4.861 s

Table 1: Training Time and Speedup (2 vs 4 GPUs)

Batch Size (per GPU)	2 GPUs Time (s)	4 GPUs Time (s)	Speedup (2→4 GPUs)
32	16.164	9.060	1.78×
128	9.849	5.023	1.96×
512	10.012	5.160	1.94×

Times used are "Epoch 2 full time." Speedup is computed as T_2/T_4.

Scaling Analysis

1. Type of scaling measured
This is **strong scaling**, because the global dataset size for an epoch is fixed while the number of GPUs increases.

2. Whether weak scaling would be better or worse
Weak scaling would show **better** speedup because each GPU would maintain the same amount of work, keeping computation dominant relative to communication.

3. Data point supporting the argument
Speedup improves when each GPU has more work.
Example: batch size 32 achieves 1.78× speedup, while batch size 128 achieves 1.96×.
The larger workload per GPU (128) maintains efficiency closer to ideal scaling, which is what weak scaling preserves.

Relevant Screenshots:

Q3

Q3.1: How much time spent in computation and communication

Methodology:
To distinguish between computation and communication time, we apply the Strong Scaling assumption (fixed total problem size).

1. Computation Time (T_{comp}):
   We estimate the ideal computation time by taking the measured Single-GPU (Epoch 2) time and dividing it by the number of GPUs.

$$T_{comp} = \frac{T_{1GPU}}{N}$$

2. Communication Time:
   We subtract the estimated computation time from the actual measured execution time.

$$T_{comm} = T_{actual} - T_{comp}$$

Table 2: Computation vs Communication Time

Batch Size 32

Setup	Total Time	Computation	Communication
2 GPUs	16.164 s	11.775 s	4.389 s
4 GPUs	9.060 s	5.888 s	3.172 s

Batch Size 128

Setup	Total Time	Computation	Communication
2 GPUs	9.849 s	9.156 s	0.693 s
4 GPUs	5.023 s	4.578 s	0.445 s

Batch Size 512

Setup	Total Time	Computation	Communication
2 GPUs	10.012 s	9.882 s	0.130 s
4 GPUs	5.160 s	4.941 s	0.219 s

(Single-GPU times used: 23.550 s for B32, 18.311 s for B128, 19.764 s for B512.)

Q3.2: Communication Bandwidth Utilization

1. AllReduce Time Formula (Ring):

$$T_{AllReduce} = \frac{2M(N-1)}{NB}$$

Where:

- $M = 44{,}695{,}848$ bytes
- N: GPU count
- $B = 0.9 \times 10^9$ bytes/s

2. Bandwidth Utilization Formula:

$$Utilization = \frac{\left(\frac{2M(N-1)}{NT_{comm}}\right)}{B}$$

3. Table 3: Bandwidth Utilization

Batch Size 32

Setup	Comm Time	Effective BW	Utilization
2 GPUs	4.389 s	0.010 GB/s	1.13%
4 GPUs	3.172 s	0.021 GB/s	2.35%

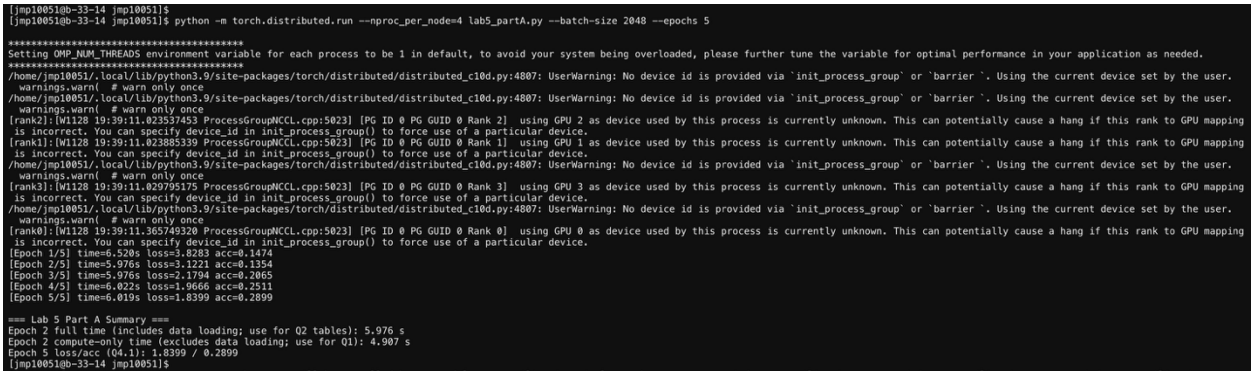
Batch Size 128

Setup	Comm Time	Effective BW	Utilization
2 GPUs	0.693 s	0.064 GB/s	7.17%
4 GPUs	0.445 s	0.151 GB/s	16.74%

Batch Size 512

Setup	Comm Time	Effective BW	Utilization
2 GPUs	0.130 s	0.344 GB/s	38.20%
4 GPUs	0.219 s	0.306 GB/s	34.02%

Q4



Q4.1: Accuracy When Using a Large Batch

Experimental Setup (4 GPUs, Batch Size 2048 per GPU):

- Batch Size per GPU: 2048
- Total Effective Batch Size: 8192
- Epoch 5 Average Training Loss: 1.8399
- Epoch 5 Training Accuracy: 0.2899 (28.99%)

Comparison with Lab 2 (Batch Size 128):
The training accuracy with this large-batch configuration (28.99% at Epoch 5) is much lower than the accuracy typically observed in Lab 2 with batch size 128 (often above 45% by Epoch 5).

Reason:
A very large batch reduces the number of parameter updates per epoch. With no hyperparameter adjustments, the model converges more slowly and is more likely to fall into sharper minima, creating a generalization gap and resulting in lower accuracy.

Q4.2: How to Improve Training Accuracy with a Large Batch

According to *Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour* [4], the following two methods address the degradation observed in Q4.1.

Remedy 1: Linear scaling of the learning rate
When increasing the minibatch by factor k, scale the learning rate by the same factor:

$$\mathrm{LR}_{\mathrm{new}} = \mathrm{LR}_{\mathrm{base}} \times \frac{\mathrm{BatchSize}_{\mathrm{new}}}{\mathrm{BatchSize}_{\mathrm{base}}}$$

Concrete example: base total batch = 128, new total batch = 8192 $\Rightarrow k = 64$.
If $\mathrm{LR}_{\mathrm{base}} = 0.1$, set $\mathrm{LR}_{\mathrm{new}} = 0.1 \times 64 = 6.4$.

Remedy 2: Gradual warmup of the learning rate
Avoid starting with the full scaled LR immediately. Linearly increase the LR from a small initial value to the scaled LR over a short warmup period (e.g., first 3–5 epochs).

Linear warmup per epoch (or per step):

$$\mathrm{LR}(t) = \mathrm{LR}_{\mathrm{start}} + \frac{t}{T_{\mathrm{warmup}}}(\mathrm{LR}_{\mathrm{new}} - \mathrm{LR}_{\mathrm{start}}), 0 \le t \le T_{\mathrm{warmup}}$$

Choose $\mathrm{LR}_{\mathrm{start}}$ small (e.g., $\mathrm{LR}_{\mathrm{base}}$ or $\mathrm{LR}_{\mathrm{base}}/10$), and $T_{\mathrm{warmup}} = 3$–5 epochs.

Q5

No, in addition to gradients, DistributedDataParallel also communicates model buffers such as BatchNorm running mean and running variance.

These buffers appear in C7 of Lab 2 (e.g., running_mean, running_var) and are synchronized across GPUs so that all replicas use consistent normalization statistics. This buffer broadcast happens every iteration (unless explicitly disabled), so gradients are not the only tensors sent over the network.

Q6

Yes, for batch size per GPU 512 it is generally sufficient to communicate only gradients across the 4 GPUs, because each GPU already sees a very large local mini-batch, so its BatchNorm statistics are well estimated from local data.

As discussed in [4], large-minibatch ImageNet training can keep BatchNorm local (no cross-GPU BN synchronization) and still match small-batch accuracy, so gradient all-reduce is the only communication that is really required for correct SSGD in this regime.

Not synchronizing BN statistics in this setting mainly saves communication overhead; the difference between local BN statistics on each GPU becomes small when each GPU's batch size is as large as 512.

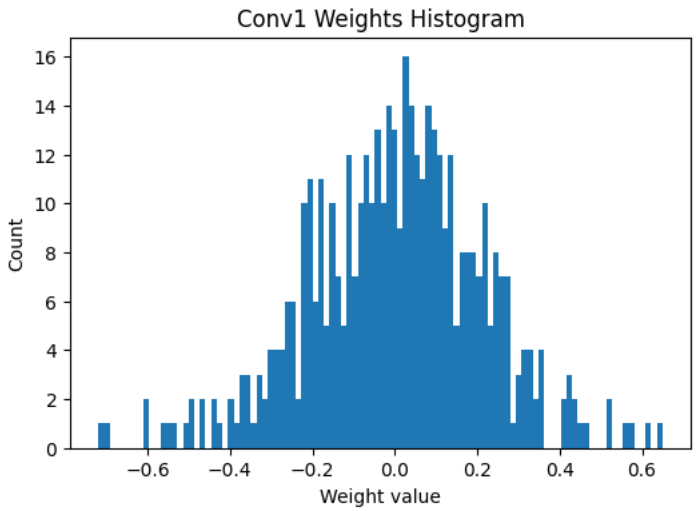
Part-B

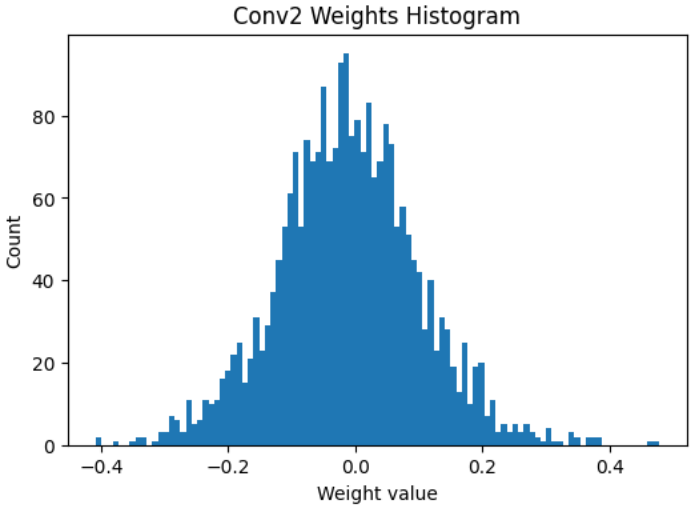
All quantization steps, histograms, and evaluation results for Part B were generated in the submitted notebook: Jmp10051_HW5_PartB.ipynb

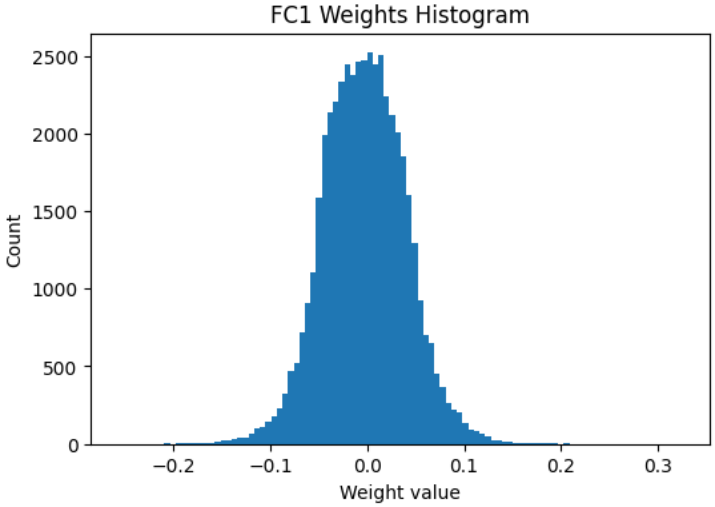
Q1

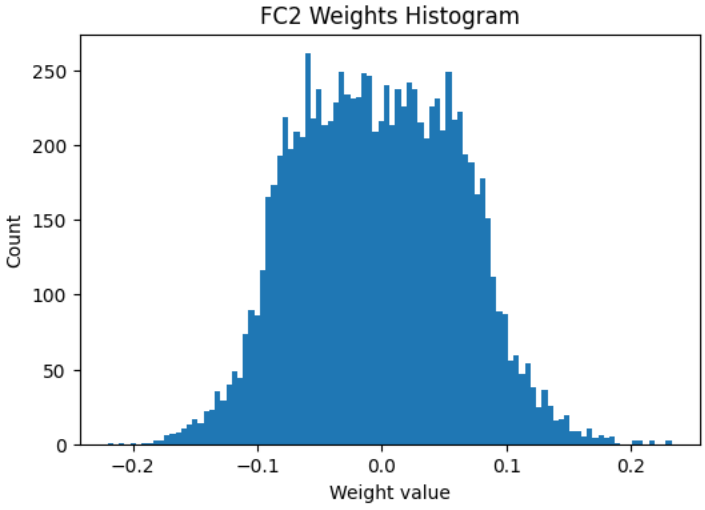
```python
# ---- Conv1 ----
conv1_w = net.conv1.weight.detach().cpu().numpy().flatten()
plt.figure(figsize=(6,4))
plt.hist(conv1_w, bins=100)
plt.title("Conv1 Weights Histogram")
plt.xlabel("Weight value")
plt.ylabel("Count")
plt.show()

# ---- Conv2 ----
conv2_w = net.conv2.weight.detach().cpu().numpy().flatten()
plt.figure(figsize=(6,4))
plt.hist(conv2_w, bins=100)
plt.title("Conv2 Weights Histogram")
plt.xlabel("Weight value")
plt.ylabel("Count")
plt.show()

# ---- FC1 ----
fc1_w = net.fc1.weight.detach().cpu().numpy().flatten()
plt.figure(figsize=(6,4))
plt.hist(fc1_w, bins=100)
plt.title("FC1 Weights Histogram")
plt.xlabel("Weight value")
plt.ylabel("Count")
plt.show()

# ---- FC2 ----
fc2_w = net.fc2.weight.detach().cpu().numpy().flatten()
plt.figure(figsize=(6,4))
plt.hist(fc2_w, bins=100)
plt.title("FC2 Weights Histogram")
plt.xlabel("Weight value")
plt.ylabel("Count")
plt.show()

# ---- FC3 ----
fc3_w = net.fc3.weight.detach().cpu().numpy().flatten()
plt.figure(figsize=(6,4))
plt.hist(fc3_w, bins=100)
plt.title("FC3 Weights Histogram")
plt.xlabel("Weight value")
plt.ylabel("Count")
plt.show()
```

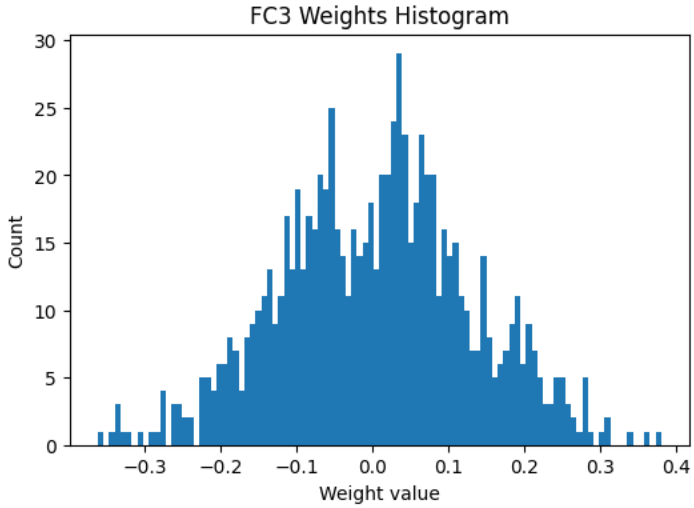
Plots:




Conv2 Weights Histogram
Count
Weight value
0
20
40
60
80
−0.4
−0.2
0.0
0.2
0.4


FC1 Weights Histogram
Count
Weight value
0
500
1000
1500
2000
2500
−0.2
−0.1
0.0
0.1
0.2
0.3


FC2 Weights Histogram
Count
Weight value
0
50
100
150
200
250
−0.2
−0.1
0.0
0.1
0.2



After plotting the weight histograms for all layers, I noticed that the weights are centered around zero and follow a smooth, bell-shaped distribution. Both convolutional and fully connected layers stay within a small range, with only a few outliers. This kind of symmetric, narrow distribution suggests that the network does not depend on extremely large weights and should tolerate uniform int8 quantization quite well. Overall, the plots show that a simple per-layer symmetric scale would be appropriate.

Q2

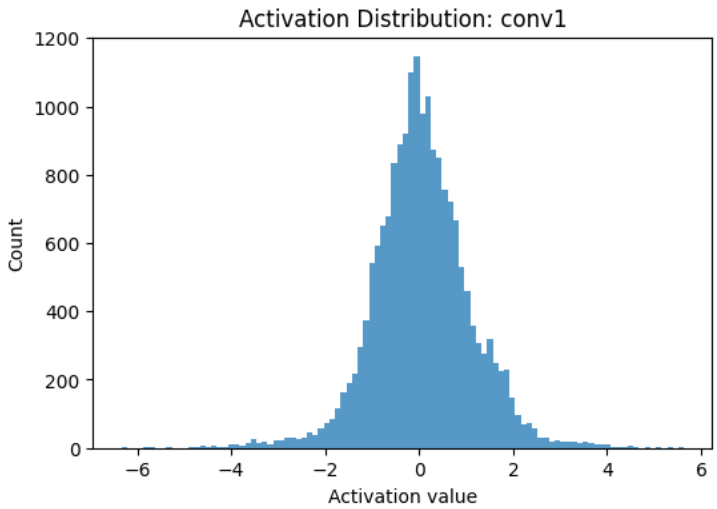
```python
from typing import Tuple

def quantized_weights(weights: torch.Tensor) -> Tuple[torch.Tensor, float]:
    '''
    Quantize the weights so that all values are integers between -128 and 127.
    You may want to use the total range, 3-sigma range, or some other range when
    deciding just what factors to scale the float32 values by.

    Parameters:
    weights (Tensor): The unquantized weights

    Returns:
    (Tensor, float): A tuple with the following elements:
                     * The weights in quantized form, where every value is an integer between -128 and 127.
                       The "dtype" will still be "float", but the values themselves should all be integers.
                     * The scaling factor that your weights were multiplied by.
                       This value does not need to be an 8-bit integer.
    '''

    # ADD YOUR CODE HERE
    #max : 127, min : -128
    max_val = weights.abs().max()
    if max_val == 0:
        scale = 1.0
    else:
        scale = 127.0 / max_val

    quantized = (weights * scale).round()
    quantized = torch.clamp(quantized, min=-128, max=127)

    return quantized, scale
```

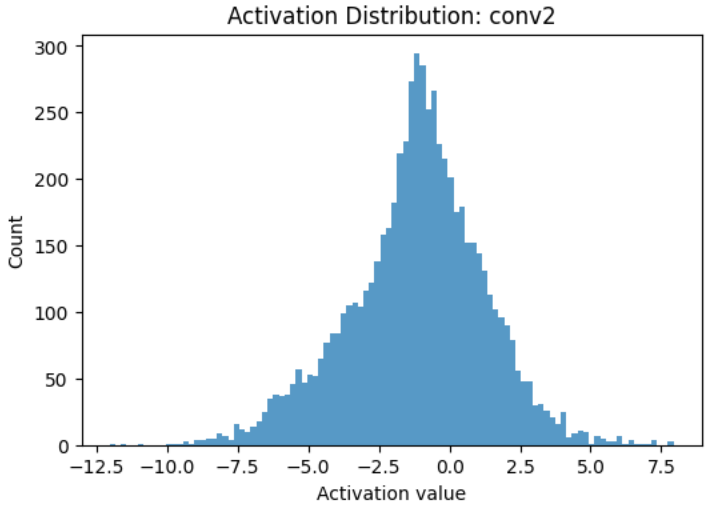
I quantized all convolutional and fully connected weights by scaling them so that the largest absolute weight maps to 127, then rounding and clamping everything into the int8 range. After applying this quantization to a copy of the model, the accuracy became 54.1%, slightly below the original model's accuracy. The model still performs reasonably well, which shows that it can tolerate 8-bit weight precision without significant degradation.

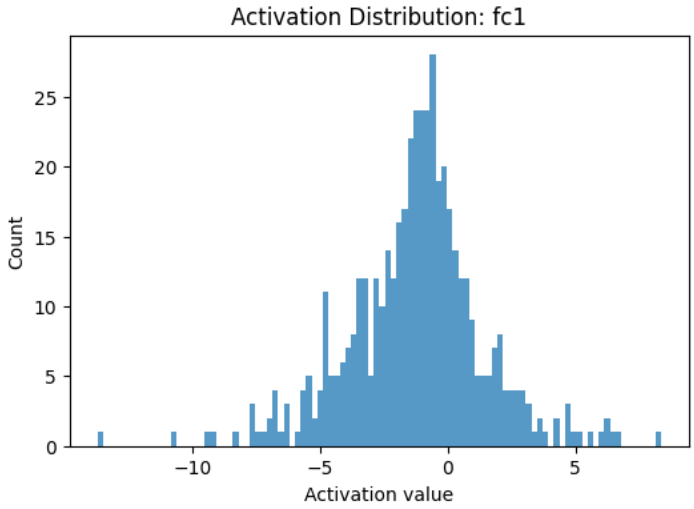
Q3

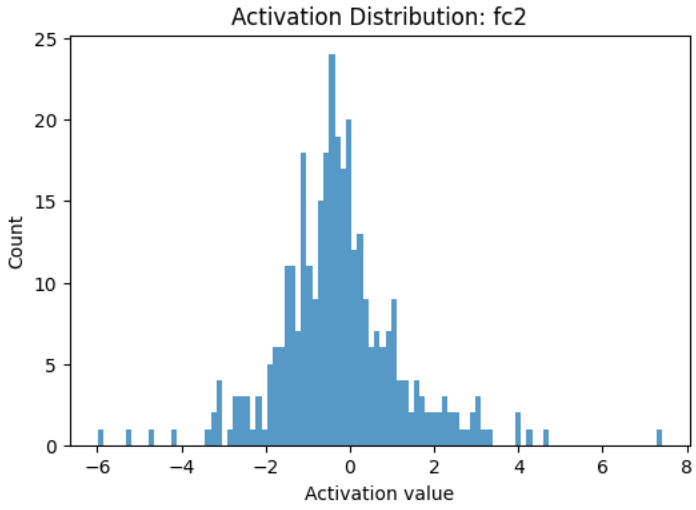
```python
# Dictionary to hold activations
activations = {}
# Hook function
def save_activation(name):
    def hook(module, input, output):
        activations[name] = output.detach().cpu().view(-1).numpy()
    return hook

# Attach hooks to a few key layers
handles = []
handles.append(net.conv1.register_forward_hook(save_activation("conv1")))
handles.append(net.conv2.register_forward_hook(save_activation("conv2")))
handles.append(net.fc1.register_forward_hook(save_activation("fc1")))
handles.append(net.fc2.register_forward_hook(save_activation("fc2")))

# Run one batch to collect activations
images, labels = next(iter(testloader))
images = images.to(device)

with torch.no_grad():
    _ = net(images)

# Plot histograms
for name, act in activations.items():
    plt.figure(figsize=(6,4))
    plt.hist(act, bins=100, alpha=0.75)
    plt.title(f"Activation Distribution: {name}")
    plt.xlabel("Activation value")
    plt.ylabel("Count")
    plt.show()

# Remove hooks
for h in handles:
    h.remove()

```

Plots:









I added forward hooks to several layers and plotted the resulting histograms. The activations before ReLU were centered near zero with a roughly Gaussian shape, while the post-ReLU activations became strictly non-negative and heavily concentrated near zero.
Overall, the activation ranges were moderate, which is helpful for choosing scaling factors during activation quantization later.

Q4

```python
@staticmethod
def quantize_activations(activations: np.ndarray, n_w: float, n_initial_input: float, ns: List[Tuple[float, float]]) -> float:
    '''
    Calculate a scaling factor to multiply the output of a layer by.

    Parameters:
    activations (ndarray): The values of all the pixels which have been output by this layer during training
    n_w (float): The scale by which the weights of this layer were multiplied as part of the "quantize_weights" function you wrote earlier
    n_initial_input (float): The scale by which the initial input to the neural network was multiplied
    ns ([(float, float)]): A list of tuples, where each tuple represents the "weight scale" and "output scale" (in that order) for every preceding layer

    Returns:
    float: A scaling factor that the layer output should be multiplied by before being fed into the first layer.
           This value does not need to be an 8-bit integer.
    '''
    abs_vals = np.abs(activations)
    maximum = np.max(abs_vals)
    # start from the initial input scale
    scale = n_initial_input
    # loop through pairs of scales for previous layers
    for weights_scale, output_scale in ns:
        scale = weights_scale * output_scale * scale
    # scale calculation
    scale = 255.0 / (maximum * scale * n_w)
    return scale
```

```python
def forward(self, x: torch.Tensor) -> torch.Tensor:
    # You can access the output activation scales like this:
    #   fc1_output_scale = self.fc1.output_scale

    # To make sure that the outputs of each layer are integers between -128 and 127, you may need to use the following functions:
    #   * torch.Tensor.round
    #   * torch.clamp

    # ADD YOUR CODE HERE
    minimum = -128
    maximum = 127

    # quantize initial input
    x = (x * self.input_scale).round()
    x = torch.clamp(x, min=minimum, max=maximum)

    # ----- Conv1 -----
    x = self.conv1(x)
    x = (x * self.conv1.output_scale).round()
    x = torch.clamp(x, minimum, maximum)
    x = F.relu(x)
    x = self.pool(x)

    # ----- Conv2 -----
    x = self.conv2(x)
    x = (x * self.conv2.output_scale).round()
    x = torch.clamp(x, minimum, maximum)
        x = F.relu(x)
        x = self.pool(x)

        # flatten
        x = torch.flatten(x, start_dim=1)
⌘L to chat, ⌘K to generate
        # ----- FC1 -----
        x = self.fc1(x)
        x = F.relu(x)
        x = (x * self.fc1.output_scale).round()
        x = torch.clamp(x, minimum, maximum)

        # ----- FC2 -----
        x = self.fc2(x)
        x = F.relu(x)
        x = (x * self.fc2.output_scale).round()
        x = torch.clamp(x, minimum, maximum)

        # ----- FC3 (logits) -----
        x = self.fc3(x)
        x = (x * self.fc3.output_scale).round()
        x = torch.clamp(x, minimum, maximum)

        return x
```

I quantized the activations at every layer using the scaling factors computed during training. After each convolution, fully connected layer, and ReLU, the outputs were scaled, rounded, and clamped to the int8 range.

With both weights and activations quantized, the model reached 53.17% accuracy. This is a small drop from the weight-only case, which is expected since activations are quantized repeatedly throughout the forward pass. Even so, the network remains stable under full activation quantization.

Q5

```python
@staticmethod
def quantized_bias(bias: torch.Tensor, n_w: float, n_initial_input: float, ns: List[Tuple[float, float]]) -> torch.Tensor:
    '''
    Quantize the bias so that all values are integers between -2147483648 and 2147483647.

    Parameters:
    bias (Tensor): The floating point values of the bias
    n_w (float): The scale by which the weights of this layer were multiplied
    n_initial_input (float): The scale by which the initial input to the neural network was multiplied
    ns ([(float, float)]): A list of tuples, where each tuple represents the "weight scale" and "output scale" (in that order) for every preceding layer

    Returns:
    Tensor: The bias in quantized form, where every value is an integer between -2147483648 and 2147483647.
            The "dtype" will still be "float", but the values themselves should all be integers.
    '''

    # ADD YOUR CODE HERE
    # Start from the scale used for the very first input
    scale = n_initial_input
    for i in range(0, len(ns)):
        weights_scale, outputs_scale = ns[i]
        scale = weights_scale * outputs_scale * scale

    # include this layer's weight scale
    scale = scale * n_w

    return torch.clamp((bias * scale).round(), min=-2147483648, max=2147483647)
```

I quantized the biases using the combined scaling from the input, previous layers' output scales, and the current layer's weight scale.

- The full-precision model reached 54.66% accuracy.
- After quantizing only the weights, the accuracy dropped to 51.4%.
- Once I also quantized the biases, the accuracy improved to 54.49% almost matching the original model.

This shows that properly scaling and quantizing the biases helps recover some of the accuracy lost from weight quantization and is important for stable fully quantized inference.