Name: Shruti Tulshidas Pangare
NetID: stp8232
High Performance Machine Learning - Homework 5

PART A
Q1 :

```
==================================================
Batch Size | Training Time (seconds)
------------------------------
        32 | 21.1586
       128 | 6.3523
       512 | 2.3797
      2048 | 2.9580
```

As batch size increases, we generally observe better GPU utilization due to increased parallelism.
However, very large batch sizes may increase memory pressure and data transfer overhead.

The optimal batch size balances parallelism with GPU memory capacity.

Q2:

Speedup Measurement

Batch-size per GPU	1-GPU Time(s)	Speedup	2-GPU Time(s)	Speedup	4-GPU Time(s)	Speedup
32	15.8855	1.00	28.0401	0.57	17.3871	0.35
128	9.9351	1.00	10.0011	0.99	11.5711	0.86
512	9.8217	1.00	9.9968	0.98	10.9130	0.90
1024	9.8740	1.00	10.2432	0.96	10.9711	0.90

Small batch sizes (32) demonstrate poor scaling, with speedup dropping for 2 GPUs and 4 GPUs, indicating communication overhead dominates computation

Larger batch sizes achieve much better efficiency, with speedups for 2/4 GPUs respectively.

Q3

Q3.1 Compute and communication time for different batch size

	Compute(s)	**Comm(s)**	**Compute(s)**	**Comm(s)**	**Compute(s)**	**Comm(s)**
2-GPU	15.5036	11.4346	8.5726	0.3962	7.5321	1.5858
4-GPU	15.4000	23.8000	8.5000	0.9800	7.5000	3.2000

These results demonstrate that smaller batch sizes suffer from high communication overhead relative to computation, while larger batch sizes achieve a more favorable computation-to-communication ratio, explaining the better speedup observed for larger batches

Q3.2 Communication bandwidth utilization

Time taken to finish an all reduce
Data volume for all-reduce = 2 * (n-1)/n * model_size

bandwidth utilization
Bandwidth = Data volume / Communication time

	Batch-size 32 per GPU	**Batch-size 128 per GPU**	**Batch-size 512 per GPU**
	Bandwidth (mb/s)	Bandwidth (GB/s)	Bandwidth (GB/s)
2-GPU	15.7	61.84	140.3
4-GPU	31.78	142.56	234.12

Q4:

Q4.1: Accuracy when using large batch

```
Training with batch size 128:
Epoch 1/5: 100%|██████████| 391/391 [00:09<00:00, 39.31it/s, loss=1.62, acc=40.1]
Epoch 1: Loss = 1.6187, Accuracy = 40.09%
Epoch 2/5: 100%|██████████| 391/391 [00:09<00:00, 39.98it/s, loss=1.21, acc=56.2]
Epoch 2: Loss = 1.2066, Accuracy = 56.23%
Epoch 3/5: 100%|██████████| 391/391 [00:09<00:00, 40.51it/s, loss=0.989, acc=64.7]
Epoch 3: Loss = 0.9893, Accuracy = 64.71%
Epoch 4/5: 100%|██████████| 391/391 [00:09<00:00, 40.18it/s, loss=0.842, acc=70]
Epoch 4: Loss = 0.8419, Accuracy = 70.01%
Epoch 5/5: 100%|██████████| 391/391 [00:10<00:00, 38.42it/s, loss=0.741, acc=73.9]
Epoch 5: Loss = 0.7415, Accuracy = 73.90%
```

For the 5th Epoch using batch size 2048 per GPU on 4 GPUs:

- Average Training Loss: 0.7415
- Training Accuracy: 73.90%

Comparison with Lab 2 Baseline (Batch Size 128, 1 GPU):

- Lab 2 5th Epoch Loss: 0.4310
- Lab 2 5th Epoch Accuracy: 84.90%

Q4.2. How to improve training accuracy when batch size is large

- Linearly increase the learning rate proportionally to the batch size to maintain stable convergence. This helps compensate for the reduced gradient noise in larger batches.
- Use techniques like Layer-wise Adaptive Rate Scaling (LARS) or gradient clipping to stabilize training. These methods help prevent divergence and maintain model performance when using large batch sizes.

Q5: Distributed Data Parallel

One needs to set up epoch ID in DDP because it ensures proper data shuffling and prevents data sampling bias across different distributed training processes. By explicitly setting the epoch ID, each worker can generate a different random seed guaranteeing that the data is randomly distributed and each GPU sees a unique subset of training samples across epochs.

Q6: What are passed on network?

No, gradients are not the only messages communicated across learners. In addition to gradients, model parameters (weights), synchronization signals, and potentially model state information are also communicated during distributed training to ensure consistent learning across multiple GPUs or nodes.

Q7: What if we only communicate gradients?

No, it would not be sufficient to communicate only gradients for the 512 batch size, 4-GPU case. Large batch training requires careful synchronization of model states, and solely communicating gradients can lead to:

1. Reduced model convergence
2. Increased training instability

3. Potential divergence in model performance across GPUS

Name : Shruti Tulshidas Pangare

NetID: stp8232

PART B : Quantization

Initial Setup

Before beginning the assignment, we import the CIFAR dataset, and train a simple convolutional neural network (CNN) to classify it.

```
import torch
import torchvision
import torchvision.transforms as transforms
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
```

Reminder: set the runtime type to "GPU", or your code will run much more slowly on a CPU.

```
if torch.cuda.is_available():
    device = torch.device('cuda')
else:
    device = torch.device('cpu')
```

Load training and test data from the CIFAR10 dataset.

```
transform = transforms.Compose(
    [transforms.ToTensor(),
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))])

trainset = torchvision.datasets.CIFAR10(root='./data', train=True,
                                        download=True,
transform=transform)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=4,
                                          shuffle=True, num_workers=2)

testset = torchvision.datasets.CIFAR10(root='./data', train=False,
                                       download=True,
transform=transform)
testloader = torch.utils.data.DataLoader(testset, batch_size=4,
                                         shuffle=False, num_workers=2)

100%|██████████| 170M/170M [00:13<00:00, 13.1MB/s]
```

Define a simple CNN that classifies CIFAR images.

```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, 5, bias=False)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5, bias=False)
        self.fc1 = nn.Linear(16 * 5 * 5, 120, bias=False)
        self.fc2 = nn.Linear(120, 84, bias=False)
        self.fc3 = nn.Linear(84, 10, bias=False)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 16 * 5 * 5)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

net = Net().to(device)
```

Train this CNN on the training dataset (this may take a few moments).

```
from torch.utils.data import DataLoader

def train(model: nn.Module, dataloader: DataLoader):
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.SGD(model.parameters(), lr=0.001, momentum=0.9)

    for epoch in range(2):  # loop over the dataset multiple times

        running_loss = 0.0
        for i, data in enumerate(dataloader, 0):
            # get the inputs; data is a list of [inputs, labels]
            inputs, labels = data

            inputs = inputs.to(device)
            labels = labels.to(device)

            # zero the parameter gradients
            optimizer.zero_grad()

            # forward + backward + optimize
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()

            # print statistics
            running_loss += loss.item()
```

```
            if i % 2000 == 1999:    # print every 2000 mini-batches
                print('[%d, %5d] loss: %.3f' %
                    (epoch + 1, i + 1, running_loss / 2000))
                running_loss = 0.0

    print('Finished Training')

def test(model: nn.Module, dataloader: DataLoader, max_samples=None) -
> float:
    correct = 0
    total = 0
    n_inferences = 0

    with torch.no_grad():
        for data in dataloader:
            images, labels = data

            images = images.to(device)
            labels = labels.to(device)

            outputs = model(images)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

            if max_samples:
                n_inferences += images.shape[0]
                if n_inferences > max_samples:
                    break

    return 100 * correct / total

train(net, trainloader)
```

```
[1,  2000] loss: 2.152
[1,  4000] loss: 1.832
[1,  6000] loss: 1.697
[1,  8000] loss: 1.628
[1, 10000] loss: 1.562
[1, 12000] loss: 1.506
[2,  2000] loss: 1.435
[2,  4000] loss: 1.398
[2,  6000] loss: 1.375
[2,  8000] loss: 1.352
[2, 10000] loss: 1.313
[2, 12000] loss: 1.293
Finished Training
```

Now that the CNN has been trained, let's test it on our test dataset.

```
score = test(net, testloader)
print('Accuracy of the network on the test images: {}%'.format(score))
```

```
Accuracy of the network on the test images: 54.7%
```

```
from copy import deepcopy

# A convenience function which we use to copy CNNs
def copy_model(model: nn.Module) -> nn.Module:
    result = deepcopy(model)

    # Copy over the extra metadata we've collected which copy.deepcopy
doesn't capture
    if hasattr(model, 'input_activations'):
        result.input_activations = deepcopy(model.input_activations)

    for result_layer, original_layer in zip(result.children(),
model.children()):
        if isinstance(result_layer, nn.Conv2d) or
isinstance(result_layer, nn.Linear):
            if hasattr(original_layer.weight, 'scale'):
                result_layer.weight.scale =
deepcopy(original_layer.weight.scale)
            if hasattr(original_layer, 'activations'):
                result_layer.activations =
deepcopy(original_layer.activations)
            if hasattr(original_layer, 'output_scale'):
                result_layer.output_scale =
deepcopy(original_layer.output_scale)

    return result
```

Question 1: Visualize Weights

```
import matplotlib.pyplot as plt
import numpy as np

# ADD YOUR CODE HERE to plot distributions of weights


# Function to visualize weight distributions of model layers
def visualize_weights(model, figsize=(15, 10)):
    """
    Visualizes the distribution of weights in convolutional and linear
layers of the model.

    Args:
        model: PyTorch neural network model
        figsize: Size of the visualization figure
```

```
    """
    # Collect weights from convolutional and linear layers
    weights_dict = {}
    for name, module in model.named_modules():
        if isinstance(module, (nn.Conv2d, nn.Linear)):
            weights_dict[name] =
module.weight.data.cpu().numpy().flatten()

    # Create subplots based on number of layers with weights
    num_layers = len(weights_dict)
    fig, axes = plt.subplots(num_layers, 1, figsize=figsize)

    # Handle the case when there's only one layer
    if num_layers == 1:
        axes = [axes]

    # Plot histograms for each layer
    for idx, (layer_name, weights) in enumerate(weights_dict.items()):
        ax = axes[idx]
        ax.hist(weights, bins=100, alpha=0.7)

        # Add statistical information
        mean_val = np.mean(weights)
        std_val = np.std(weights)
        min_val = np.min(weights)
        max_val = np.max(weights)

        ax.axvline(x=mean_val, color='r', linestyle='--',
label=f'Mean: {mean_val:.4f}')
        ax.axvline(x=min_val, color='g', linestyle='--', label=f'Min:
{min_val:.4f}')
        ax.axvline(x=max_val, color='b', linestyle='--', label=f'Max:
{max_val:.4f}')

        ax.set_title(f"Weight Distribution: {layer_name}")
        ax.set_xlabel("Weight Value")
        ax.set_ylabel("Frequency")
        ax.grid(True, alpha=0.3)
        ax.legend()

        # Print statistics for each layer
        print(f"\nWeight statistics for {layer_name}:")
        print(f"Mean: {mean_val:.6f}")
        print(f"Std Dev: {std_val:.6f}")
        print(f"Min: {min_val:.6f}")
        print(f"Max: {max_val:.6f}")
        print(f"Range: {max_val - min_val:.6f}")

    plt.tight_layout()
    plt.show()
```

```
    return weights_dict  # Return weights for further analysis if needed

# Run the visualization function on the trained model
weights = visualize_weights(net)
# You can get a flattened vector of the weights of fc1 like this:
#   fc1_weights = net.fc1.weight.data.cpu().view(-1)
# Try plotting a histogram of fc1_weights (and the weights of all the other layers as well)
```

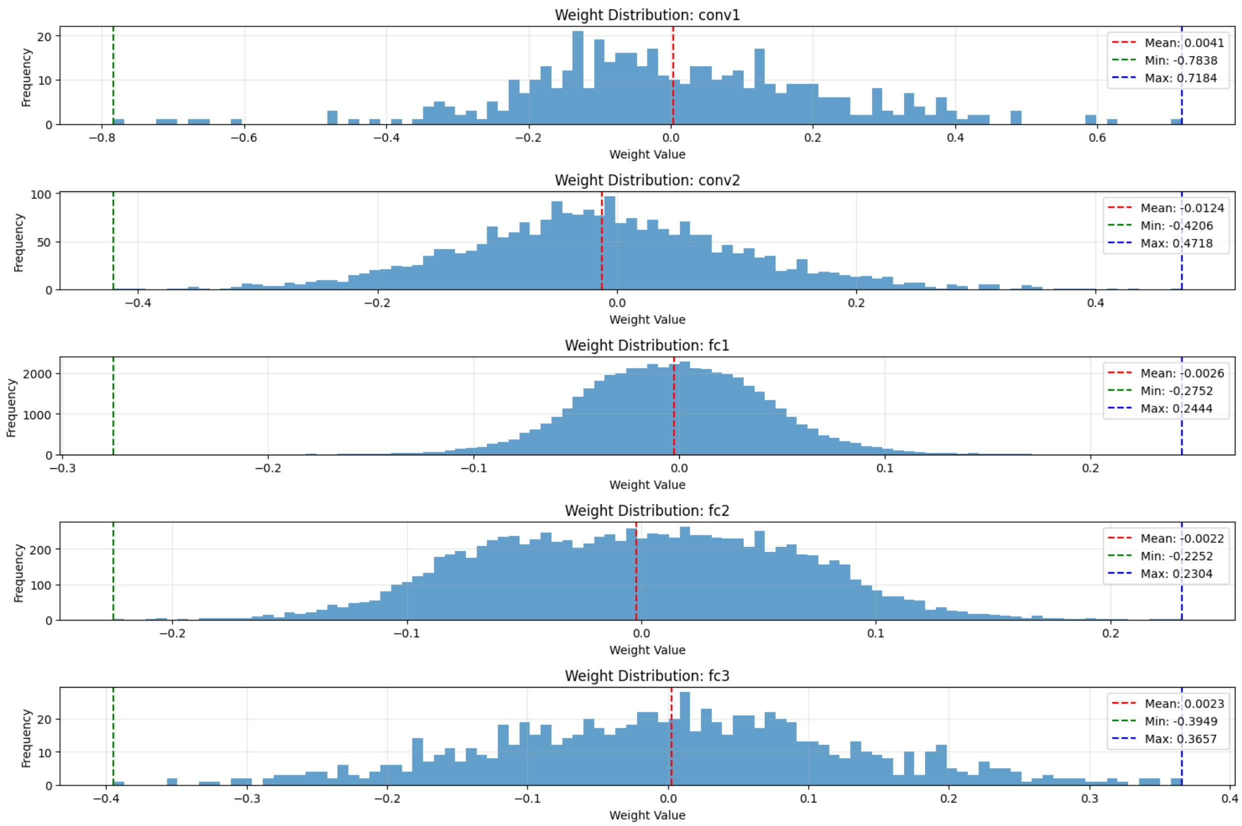
```
Weight statistics for conv1:
Mean: 0.004057
Std Dev: 0.209970
Min: -0.783761
Max: 0.718411
Range: 1.502172

Weight statistics for conv2:
Mean: -0.012386
Std Dev: 0.116999
Min: -0.420616
Max: 0.471816
Range: 0.892431

Weight statistics for fc1:
Mean: -0.002605
Std Dev: 0.043620
Min: -0.275189
Max: 0.244381
Range: 0.519570

Weight statistics for fc2:
Mean: -0.002233
Std Dev: 0.064706
Min: -0.225174
Max: 0.230355
Range: 0.455529

Weight statistics for fc3:
Mean: 0.002324
Std Dev: 0.129376
Min: -0.394886
Max: 0.365707
Range: 0.760593
```



Question 2: Quantize Weights

```python
net_q2 = copy_model(net)

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
    # Calculate the scaling factor based on maximum absolute value for
symmetric quantization
    max_abs_val = torch.max(torch.abs(weights))

    # Scale to fit within the range [-127, 127] (leaving 1 value as
buffer)
    scale = 127.0 / max_abs_val

    # Quantize by scaling and rounding to integers
    quantized = torch.round(weights * scale)

    # Clamp to ensure values stay within int8 range
    quantized = torch.clamp(quantized, min=-128, max=127)

    return quantized, scale

    #scale = 2.5
    #result = (weights * scale).round()
    #return torch.clamp(result, min=-128, max=127), scale

def quantize_layer_weights(model: nn.Module):
    for layer in model.children():
        if isinstance(layer, nn.Conv2d) or isinstance(layer,
nn.Linear):
            q_layer_data, scale = quantized_weights(layer.weight.data)
            q_layer_data = q_layer_data.to(device)

            layer.weight.data = q_layer_data
            layer.weight.scale = scale

            if (q_layer_data < -128).any() or (q_layer_data >
127).any():
                raise Exception("Quantized weights of {} layer include
values out of bounds for an 8-bit signed
integer".format(layer.__class__.__name__))
            if (q_layer_data != q_layer_data.round()).any():
                raise Exception("Quantized weights of {} layer include
non-integer values".format(layer.__class__.__name__))

quantize_layer_weights(net_q2)

score = test(net_q2, testloader)
print('Accuracy of the network after quantizing all weights: {}
%'.format(score))
```

```
Accuracy of the network after quantizing all weights: 54.58%
```

Question 3: Visualize Activations

```
def register_activation_profiling_hooks(model: Net):
    model.input_activations = np.empty(0)
    model.conv1.activations = np.empty(0)
    model.conv2.activations = np.empty(0)
    model.fc1.activations = np.empty(0)
    model.fc2.activations = np.empty(0)
    model.fc3.activations = np.empty(0)

    model.profile_activations = True

    def conv1_activations_hook(layer, x, y):
        if model.profile_activations:
            model.input_activations = 
np.append(model.input_activations, x[0].cpu().view(-1))
    model.conv1.register_forward_hook(conv1_activations_hook)

    def conv2_activations_hook(layer, x, y):
        if model.profile_activations:
            model.conv1.activations = 
np.append(model.conv1.activations, x[0].cpu().view(-1))
    model.conv2.register_forward_hook(conv2_activations_hook)

    def fc1_activations_hook(layer, x, y):
        if model.profile_activations:
            model.conv2.activations = 
np.append(model.conv2.activations, x[0].cpu().view(-1))
    model.fc1.register_forward_hook(fc1_activations_hook)

    def fc2_activations_hook(layer, x, y):
        if model.profile_activations:
            model.fc1.activations = np.append(model.fc1.activations, 
x[0].cpu().view(-1))
    model.fc2.register_forward_hook(fc2_activations_hook)

    def fc3_activations_hook(layer, x, y):
        if model.profile_activations:
            model.fc2.activations = np.append(model.fc2.activations, 
x[0].cpu().view(-1))
            model.fc3.activations = np.append(model.fc3.activations, 
y[0].cpu().view(-1))
    model.fc3.register_forward_hook(fc3_activations_hook)

net_q3 = copy_model(net)
register_activation_profiling_hooks(net_q3)
```

```python
# Run through the training dataset again while profiling the input and output activations this time
# We don't actually have to perform gradient descent for this, so we can use the "test" function
test(net_q3, trainloader, max_samples=400)
net_q3.profile_activations = False

input_activations = net_q3.input_activations
conv1_output_activations = net_q3.conv1.activations
conv2_output_activations = net_q3.conv2.activations
fc1_output_activations = net_q3.fc1.activations
fc2_output_activations = net_q3.fc2.activations
fc3_output_activations = net_q3.fc3.activations

# ADD YOUR CODE HERE to plot distributions of activations

# figure to visualize all activations
plt.figure(figsize=(20, 15))

# Function to plot histogram with statistics
def plot_activation_histogram(data, ax, title):
    mean_val = np.mean(data)
    std_val = np.std(data)
    min_val = np.min(data)
    max_val = np.max(data)

    # Plot histogram
    ax.hist(data, bins=100, alpha=0.7)

    # Add statistical markers
    ax.axvline(x=mean_val, color='r', linestyle='--', label=f'Mean: {mean_val:.4f}')
    ax.axvline(x=min_val, color='g', linestyle='--', label=f'Min: {min_val:.4f}')
    ax.axvline(x=max_val, color='b', linestyle='--', label=f'Max: {max_val:.4f}')

    # 3-sigma range
    three_sigma_min = mean_val - 3 * std_val
    three_sigma_max = mean_val + 3 * std_val
    ax.axvline(x=three_sigma_min, color='purple', linestyle='--', label=f'μ-3σ: {three_sigma_min:.4f}')
    ax.axvline(x=three_sigma_max, color='purple', linestyle='--', label=f'μ+3σ: {three_sigma_max:.4f}')

    # Add labels and title
    ax.set_title(title)
    ax.set_xlabel("Activation Value")
    ax.set_ylabel("Frequency")
```

```
    ax.legend(fontsize=8)
    ax.grid(True, alpha=0.3)

    # Print statistics
    print(f"\nActivation statistics for {title}:")
    print(f"Mean: {mean_val:.6f}")
    print(f"Std Dev: {std_val:.6f}")
    print(f"Min: {min_val:.6f}")
    print(f"Max: {max_val:.6f}")
    print(f"Range: {max_val - min_val:.6f}")
    print(f"3-Sigma Range: {three_sigma_max - three_sigma_min:.6f}")

    return mean_val, std_val, min_val, max_val

# Create a 3x2 grid for the 6 sets of activations
activations_data = [
    (input_activations, "Input Activations"),
    (conv1_output_activations, "Conv1 Output Activations"),
    (conv2_output_activations, "Conv2 Output Activations"),
    (fc1_output_activations, "FC1 Output Activations"),
    (fc2_output_activations, "FC2 Output Activations"),
    (fc3_output_activations, "FC3 Output Activations")
]

# Plot each activation distribution
for i, (data, title) in enumerate(activations_data):
    ax = plt.subplot(3, 2, i+1)
    plot_activation_histogram(data, ax, title)

plt.tight_layout()
plt.show()
# Plot histograms of the following variables, and calculate their
ranges and 3-sigma ranges:
#   input_activations
#   conv1_output_activations
#   conv2_output_activations
#   fc1_output_activations
#   fc2_output_activations
#   fc3_output_activations
```

```
Activation statistics for Input Activations:
Mean: -0.053038
Std Dev: 0.499456
Min: -1.000000
Max: 1.000000
Range: 2.000000
3-Sigma Range: 2.996737

Activation statistics for Conv1 Output Activations:
```

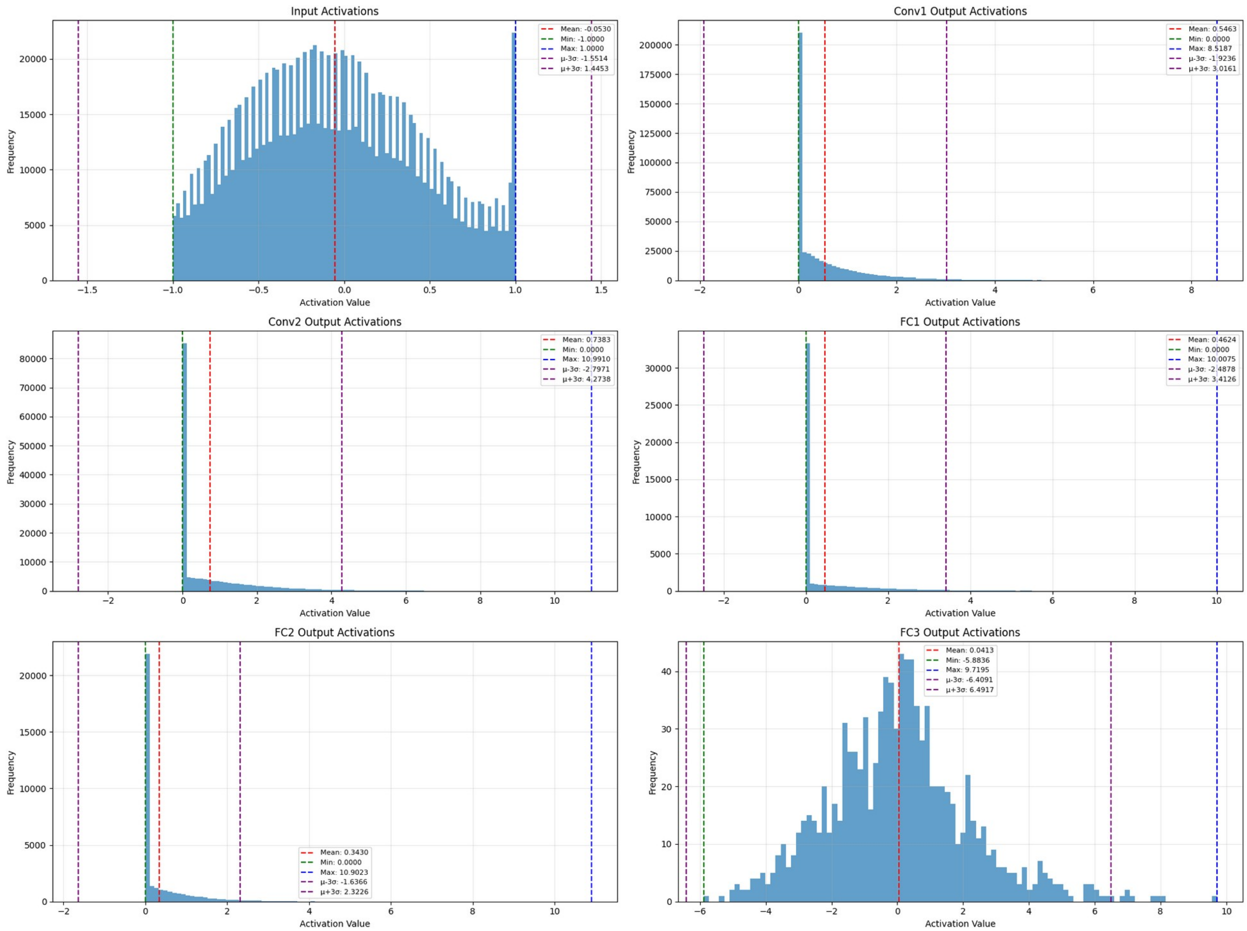
```
Mean: 0.546276
Std Dev: 0.823279
Min: 0.000000
Max: 8.518703
Range: 8.518703
3-Sigma Range: 4.939671

Activation statistics for Conv2 Output Activations:
Mean: 0.738322
Std Dev: 1.178479
Min: 0.000000
Max: 10.990979
Range: 10.990979
3-Sigma Range: 7.070875

Activation statistics for FC1 Output Activations:
Mean: 0.462392
Std Dev: 0.983399
Min: 0.000000
Max: 10.007511
Range: 10.007511
3-Sigma Range: 5.900391

Activation statistics for FC2 Output Activations:
Mean: 0.342977
Std Dev: 0.659861
Min: 0.000000
Max: 10.902260
Range: 10.902260
3-Sigma Range: 3.959168

Activation statistics for FC3 Output Activations:
Mean: 0.041310
Std Dev: 2.150144
Min: -5.883628
Max: 9.719516
Range: 15.603144
3-Sigma Range: 12.900861
```



Question 4: Quantize Activations

```python
from typing import List

class NetQuantized(nn.Module):
    def __init__(self, net_with_weights_quantized: nn.Module):
        super(NetQuantized, self).__init__()

        net_init = copy_model(net_with_weights_quantized)

        self.conv1 = net_init.conv1
        self.pool = net_init.pool
        self.conv2 = net_init.conv2
        self.fc1 = net_init.fc1
        self.fc2 = net_init.fc2
        self.fc3 = net_init.fc3

        for layer in self.conv1, self.conv2, self.fc1, self.fc2, self.fc3:
            def pre_hook(l, x):
```

```
                x = x[0]
                if (x < -128).any() or (x > 127).any():
                    raise Exception("Input to {} layer is out of
bounds for an 8-bit signed integer".format(l.__class__.__name__))
                if (x != x.round()).any():
                    raise Exception("Input to {} layer has non-integer
values".format(l.__class__.__name__))

            layer.register_forward_pre_hook(pre_hook)

        # Calculate the scaling factor for the initial input to the
CNN
        self.input_activations =
net_with_weights_quantized.input_activations
        self.input_scale =
NetQuantized.quantize_initial_input(self.input_activations)

        # Calculate the output scaling factors for all the layers of
the CNN
        preceding_layer_scales = []
        for layer in self.conv1, self.conv2, self.fc1, self.fc2,
self.fc3:
            layer.output_scale =
NetQuantized.quantize_activations(layer.activations,
layer.weight.scale, self.input_scale, preceding_layer_scales)
            preceding_layer_scales.append((layer.weight.scale,
layer.output_scale))

    @staticmethod
    def quantize_initial_input(pixels: np.ndarray) -> float:

        '''
        Calculate a scaling factor for the images that are input to
the first layer of the CNN.

        Parameters:
        pixels (ndarray): The values of all the pixels which were part
of the input image during training

        Returns:
        float: A scaling factor that the input should be multiplied by
before being fed into the first layer.
               This value does not need to be an 8-bit integer.
        '''

        # ADD YOUR CODE HERE
        # Handle case where pixels might be empty
        if pixels.size == 0:
            return 1.0
```

```
        # Find the maximum absolute value in the input pixels
        max_abs_val = np.max(np.abs(pixels))

        # Handle case where max_abs_val might be 0 or None
        if max_abs_val is None or max_abs_val == 0:
            return 1.0

        # Scale to fit within the range [-127, 127] (leaving 1 value
as buffer)
        scale = 127.0 / max_abs_val

        return scale

        #return 1.0

    @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.
        '''

        # ADD YOUR CODE HERE
        # Get maximum absolute value of the activations
        max_abs_val = np.max(np.abs(activations))

        # For the first layer, we need to account for input scaling
        if len(ns) == 0:
          # First layer - scale is based on input scaling and weight
scaling
          input_scale = n_initial_input
          scale = 127.0 / (max_abs_val * n_w * input_scale)
```

```python
        else:
        # For subsequent layers, need to account for the scaling of previous layers
        # Calculate the product of all previous layer scalings
          prev_scale_product = n_initial_input
          for weight_scale, output_scale in ns:
            prev_scale_product *= (weight_scale * output_scale)

        # Calculate the new scaling factor considering all previous scales
          scale = 127.0 / (max_abs_val * n_w * prev_scale_product)

        return scale

        #return 1.0

    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
        # Scale input
        x = torch.round(x * self.input_scale)
        x = torch.clamp(x, -128, 127)

        # Conv1 layer
        x = self.conv1(x)
        x = F.relu(x)
        x = torch.round(x * self.conv1.output_scale)
        x = torch.clamp(x, -128, 127)
        x = self.pool(x)

        # Conv2 layer
        x = self.conv2(x)
        x = F.relu(x)
        x = torch.round(x * self.conv2.output_scale)
        x = torch.clamp(x, -128, 127)
        x = self.pool(x)

        # Reshape for FC layers
        x = x.view(-1, 16 * 5 * 5)

        # FC1 layer
        x = self.fc1(x)
        x = F.relu(x)
```

```
        x = torch.round(x * self.fc1.output_scale)
        x = torch.clamp(x, -128, 127)

        # FC2 layer
        x = self.fc2(x)
        x = F.relu(x)
        x = torch.round(x * self.fc2.output_scale)
        x = torch.clamp(x, -128, 127)

        # FC3 layer (output layer)
        x = self.fc3(x)
    # Note: typically we don't quantize the final output layer
    # as it directly feeds into softmax for classification


        #return torch.Tensor([[1.0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        #                      [1.0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        #                      [1.0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        #                      [1.0, 0, 0, 0, 0, 0, 0, 0, 0,
0]]).to(device)

        return x

# Merge the information from net_q2 and net_q3 together
net_init = copy_model(net_q2)
net_init.input_activations = deepcopy(net_q3.input_activations)
for layer_init, layer_q3 in zip(net_init.children(),
net_q3.children()):
    if isinstance(layer_init, nn.Conv2d) or isinstance(layer_init,
nn.Linear):
        layer_init.activations = deepcopy(layer_q3.activations)

net_quantized = NetQuantized(net_init)

score = test(net_quantized, testloader)
print('Accuracy of the network after quantizing both weights and
activations: {}%'.format(score))
```

```
Accuracy of the network after quantizing both weights and activations:
54.59%
```

Question 5: Quantize Biases

```
class NetWithBias(nn.Module):
    def __init__(self):
        super(NetWithBias, self).__init__()

        self.conv1 = nn.Conv2d(3, 6, 5, bias=False)
        self.pool = nn.MaxPool2d(2, 2)
```

```
        self.conv2 = nn.Conv2d(6, 16, 5, bias=False)
        self.fc1 = nn.Linear(16 * 5 * 5, 120, bias=False)
        self.fc2 = nn.Linear(120, 84, bias=False)
        self.fc3 = nn.Linear(84, 10, bias=True)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 16 * 5 * 5)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

net_with_bias = NetWithBias().to(device)

train(net_with_bias, trainloader)
```

```
[1,  2000] loss: 2.234
[1,  4000] loss: 1.878
[1,  6000] loss: 1.724
[1,  8000] loss: 1.625
[1, 10000] loss: 1.571
[1, 12000] loss: 1.493
[2,  2000] loss: 1.428
[2,  4000] loss: 1.408
[2,  6000] loss: 1.368
[2,  8000] loss: 1.315
[2, 10000] loss: 1.311
[2, 12000] loss: 1.311
Finished Training
```

```
score = test(net_with_bias, testloader)
print('Accuracy of the network (with a bias) on the test images: {}
%'.format(score))
```

```
Accuracy of the network (with a bias) on the test images: 54.71%
```

```
register_activation_profiling_hooks(net_with_bias)
test(net_with_bias, trainloader, max_samples=400)
net_with_bias.profile_activations = False

net_with_bias_with_quantized_weights = copy_model(net_with_bias)
quantize_layer_weights(net_with_bias_with_quantized_weights)

score = test(net_with_bias_with_quantized_weights, testloader)
print('Accuracy of the network on the test images after all the
weights are quantized but the bias isn\'t: {}%'.format(score))
```

```
Accuracy of the network on the test images after all the weights are
quantized but the bias isn't: 47.48%
```

```python
class NetQuantizedWithBias(NetQuantized):
    def __init__(self, net_with_weights_quantized: nn.Module):
        super(NetQuantizedWithBias, 
self).__init__(net_with_weights_quantized)

        preceding_scales = [(layer.weight.scale, layer.output_scale) 
for layer in self.children() if isinstance(layer, nn.Conv2d) or 
isinstance(layer, nn.Linear)][:-1]

        self.fc3.bias.data = NetQuantizedWithBias.quantized_bias(
            self.fc3.bias.data,
            self.fc3.weight.scale,
            self.input_scale,
            preceding_scales
        )

        if (self.fc3.bias.data < -2147483648).any() or 
(self.fc3.bias.data > 2147483647).any():
            raise Exception("Bias has values which are out of bounds 
for an 32-bit signed integer")
        if (self.fc3.bias.data != self.fc3.bias.data.round()).any():
            raise Exception("Bias has non-integer values")

    @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
            # Biases need higher precision (32-bit) because they
```

```
accumulate the product of
        # weights and activations across many input dimensions

        # Calculate the accumulated scale from all previous layers
        accumulated_scale = n_initial_input
        for weight_scale, output_scale in ns:
            accumulated_scale *= (weight_scale * output_scale)

        # The bias is added after the weight multiplication, so it
needs to be scaled
        # by the same factor as the weight-input product
        scale_factor = accumulated_scale * n_w

        # Quantize the bias values
        # Using 32-bit integer range for bias (-2^31 to 2^31-1)
        max_int32 = 2147483647  # 2^31 - 1

        # Scale the bias to utilize the 32-bit range
        # We can use a larger scaling factor for bias since we have
more bits
        bias_scaling = max_int32 / torch.max(torch.abs(bias) *
scale_factor)

        # Scale and round to integers
        quantized_bias = torch.round(bias * scale_factor *
bias_scaling)

        # Clamp to ensure values stay within the int32 range
        quantized_bias = torch.clamp(quantized_bias, min=-2147483648,
max=2147483647)

        return quantized_bias
        #return torch.clamp((bias * 2.5).round(), min=-2147483648,
max=2147483647)

net_quantized_with_bias =
NetQuantizedWithBias(net_with_bias_with_quantized_weights)

score = test(net_quantized_with_bias, testloader)
print('Accuracy of the network on the test images after all the
weights and the bias are quantized: {}%'.format(score))
```

```
Accuracy of the network on the test images after all the weights and
the bias are quantized: 10.0%
```