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


import math
```

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)
```

```
    Files already downloaded and verified
    Files already downloaded and verified
```

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: 2.259
[1,  4000] loss: 1.974
[1,  6000] loss: 1.773
[1,  8000] loss: 1.651
[1, 10000] loss: 1.597
[1, 12000] loss: 1.515
[2,  2000] loss: 1.478
[2,  4000] loss: 1.433
[2,  6000] loss: 1.410
[2,  8000] loss: 1.366
[2, 10000] loss: 1.347
[2, 12000] loss: 1.326
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: 52.99%
```

```
from copy import deepcopy
```

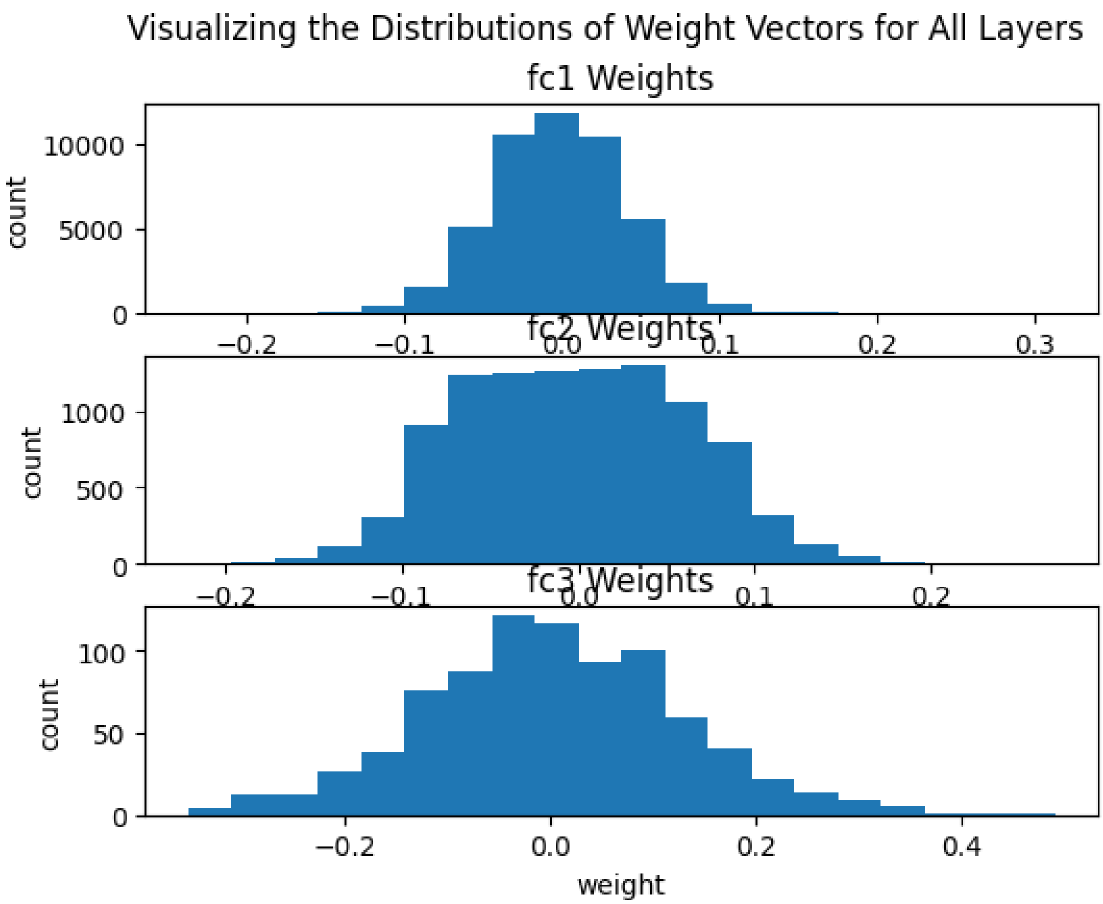
```
# A convenience function which we use to copy CNNs
def copy_model(model: nn.Module) -> nn.Module:
    result = deepcopy(model)

    # 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


# You can get a flattened vector of the weights of fc1 like this:
# Try plotting a histogram of fc1_weights (and the weights of all the other layers as wel

fc1_weights = net.fc1.weight.data.cpu().view(-1)
fc2_weights = net.fc2.weight.data.cpu().view(-1)
fc3_weights = net.fc3.weight.data.cpu().view(-1)
```

```
fig, axs = plt.subplots(3,1)
fig.suptitle("Visualizing the Distributions of Weight Vectors for All Layers")
plt.figure(figsize=(50,50))
axs[0].hist(fc1_weights, bins=20)
axs[0].set_title("fc1 Weights")
axs[0].set_xlabel("weight")
axs[0].set_ylabel("count")
axs[1].hist(fc2_weights, bins=20)
axs[1].set_title("fc2 Weights")
axs[1].set_xlabel("weight")
axs[1].set_ylabel("count")
axs[2].hist(fc3_weights, bins=20)
axs[2].set_title("fc3 Weights")
axs[2].set_xlabel("weight")
axs[2].set_ylabel("count")
```

```
# set the spacing between subplots
plt.subplots_adjust(left=0.1,
                    bottom=0.1,
                    right=0.9,
                    top=0.9,
                    wspace=0.4,
                    hspace=0.4)
plt.show()
```




```
<Figure size 5000x5000 with 0 Axes>
```

Question 2: Quantize Weights

```
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 i
                          The "dtype" will still be "float", but the values themselves sh
                        * The scaling factor that your weights were multiplied by.
                          This value does not need to be an 8-bit integer.
    '''
    # Flatten the weights tensor into a 1D tensor
    weights_flat = weights.flatten()

    # Calculate the mean and standard deviation of the weights
    mean = weights_flat.mean()
    std = weights_flat.std()

    # Calculate the 3 sigma range
    three_sigma_range = 3.0 * std

    # Determine the minimum and maximum values for the quantization levels
    min_val = mean - three_sigma_range
    max_val = mean + three_sigma_range

    # Calculate the interval size between quantization levels
    interval_size = 255 / (max_val - min_val)

    # calculate the scaling factor to map the range to the desired range of [-128, 127]
    scale =  255.0 / (max_val - min_val)

    # quantize the weights and return the result
    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 boun
            if (q_layer_data != q_layer_data.round()).any():
                raise Exception("Quantized weights of {} layer include non-integer values

quantize_layer_weights(net_q2)


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

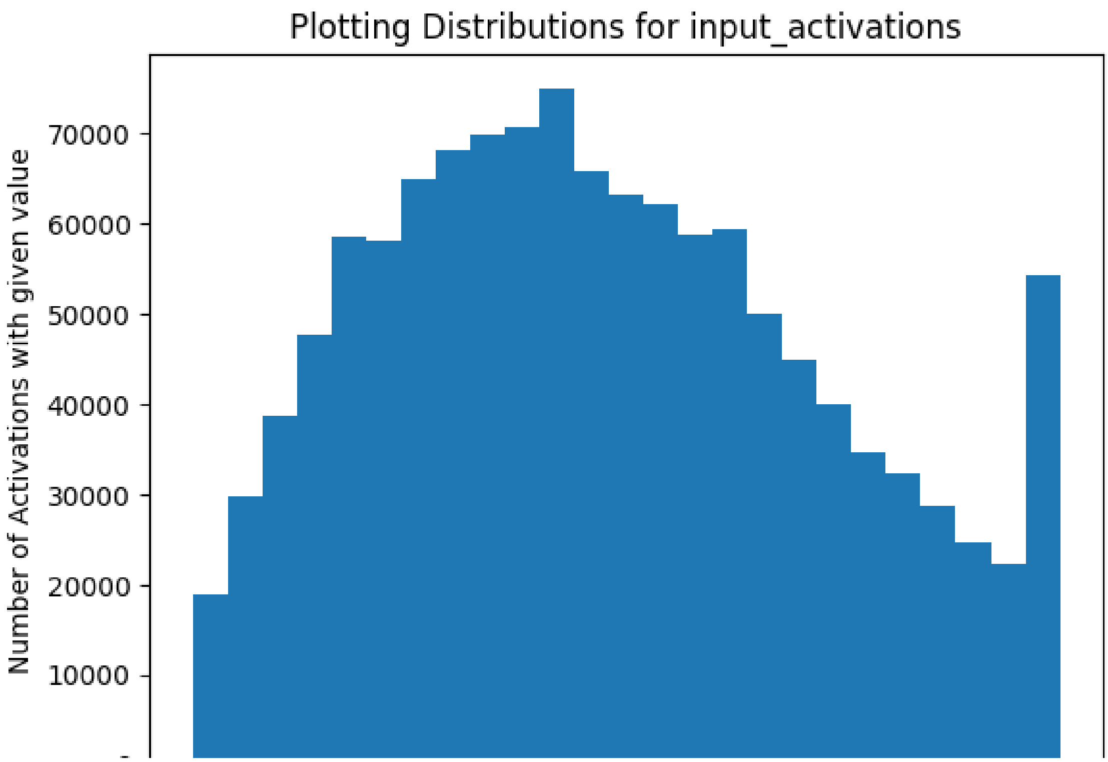
```
Accuracy of the network after quantizing all weights: 52.2%
```

Question 3: Visualize Activations

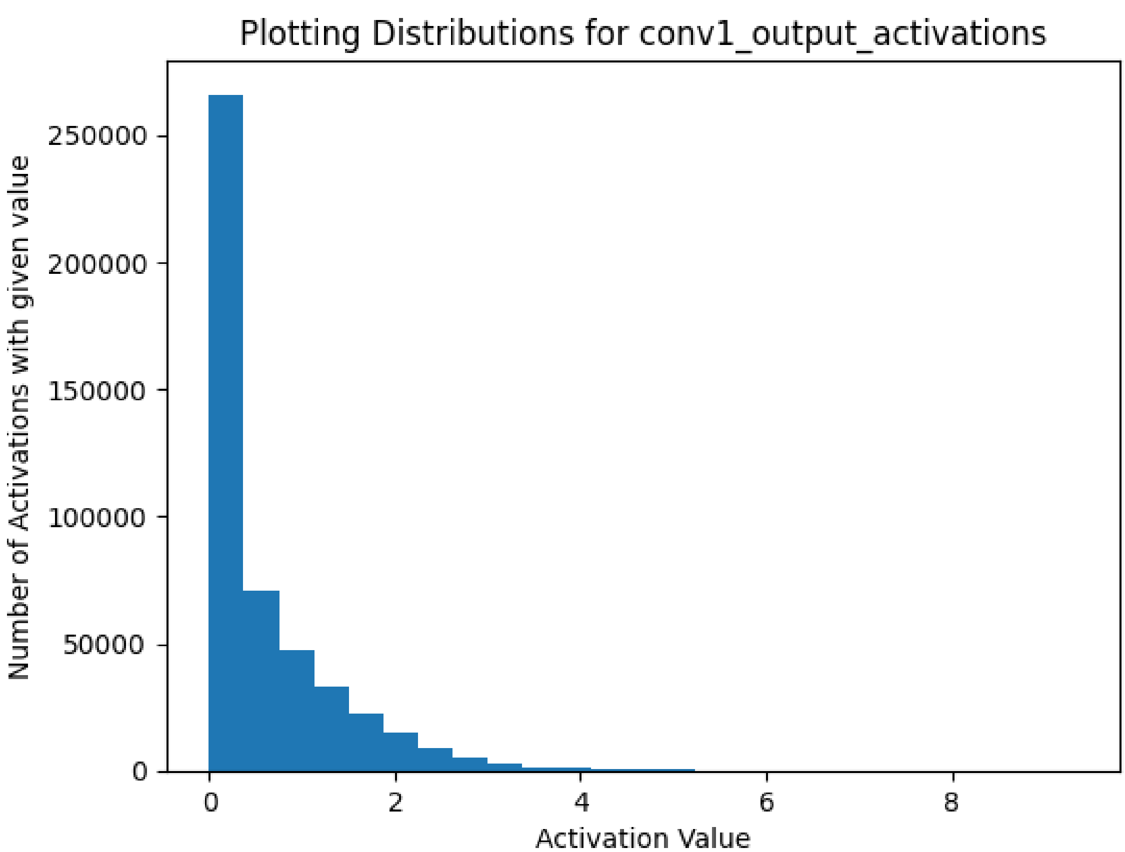
```
def register_activation_profiling_hooks(model: Net):
    model.input_activations = np.empty(0)
    model.conv1.activations = np.empty(0)
    model.conv2.activations = np.empty(0)
    model.fc1.activations = np.empty(0)
    model.fc2.activations = np.empty(0)
    model.fc3.activations = np.empty(0)

    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(
    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(
    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(
    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)

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

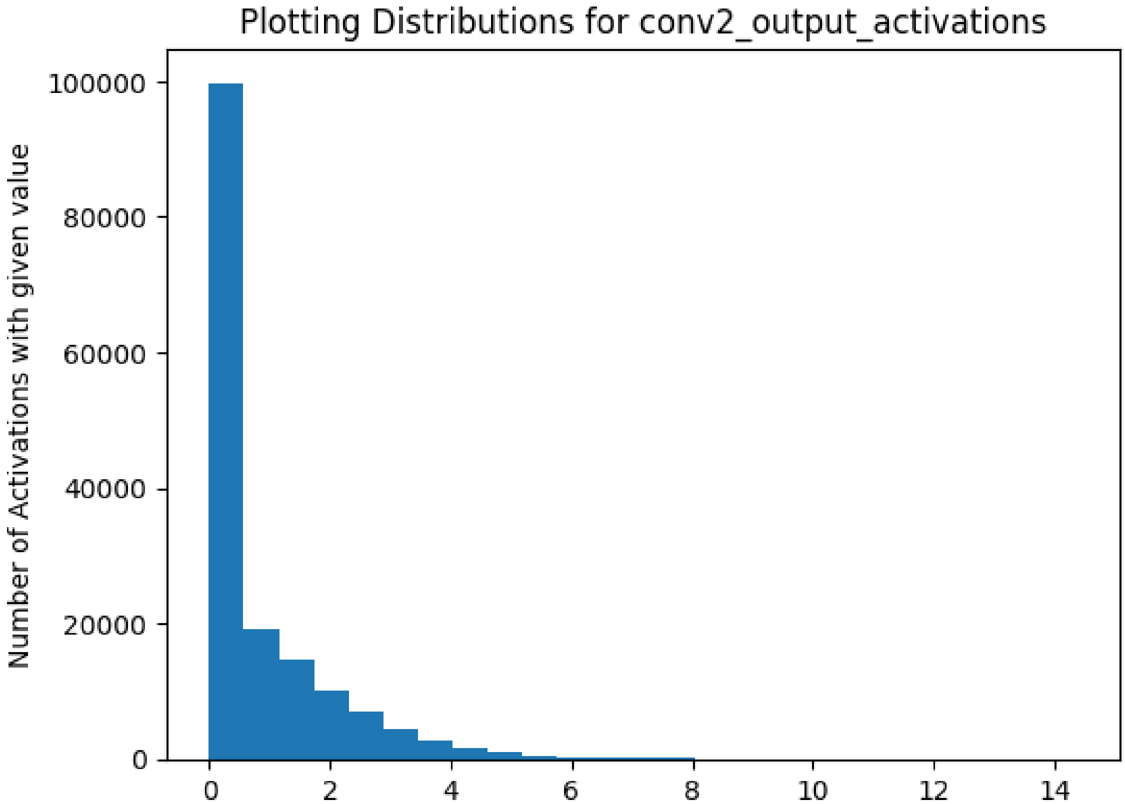
```
input_activations = net_q3.input_activations
conv1_output_activations = net_q3.conv1.activations
conv2_output_activations = net_q3.conv2.activations
fc1_output_activations = net_q3.fc1.activations
fc2_output_activations = net_q3.fc2.activations
fc3_output_activations = net_q3.fc3.activations


def compute_range(activations, name):
    # Flatten the weights tensor into a 1D tensor
    activations_flat = activations.flatten()

    # Calculate the true range
    max = activations_flat.max()
    min = activations_flat.min()

    # Calculate the mean and standard deviation of the weights
    mean = activations_flat.mean()
    std = activations_flat.std()

    # Calculate the 3 sigma range
    three_sigma_range = 3.0 * std

    # Determine the minimum and maximum values for the quantization levels
    min_val = mean - three_sigma_range
    max_val = mean + three_sigma_range

    print(f"{name}")
    print(f"range: ({min}, {max}) = {max-min:.4f}")
    print(f"3-sigma range: ({min_val:.4f}, {max_val:.4f}) = {max_val-min_val:.4f}")
    print("=========")


def plot_distribution(activations, name):
    activations_flat = activations.flatten()
    plt.hist(activations_flat, bins=25)
    plt.title(f"Plotting Distributions for {name}")
    plt.xlabel("Activation Value")
    plt.ylabel("Number of Activations with given value")


# Printing the range and 3-sigma range for each layer
plot_distribution(input_activations, "input_activations")
# Plot histograms of the following variables, and calculate their ranges and 3-sigma rang
#   input_activations
#   conv1_output_activations
#   conv2_output_activations
#   fc1_output_activations
#   fc2_output_activations
#   fc3_output_activations
```



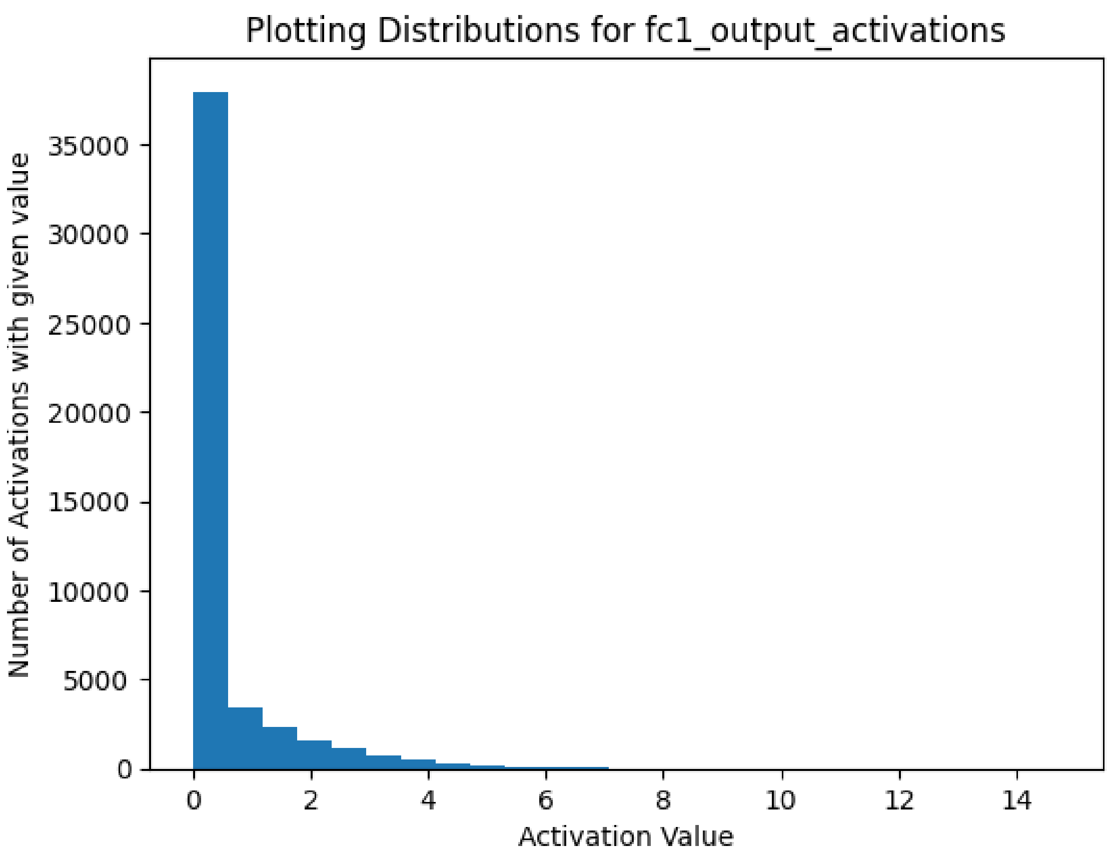

```
plot_distribution(conv1_output_activations, "conv1_output_activations")
```



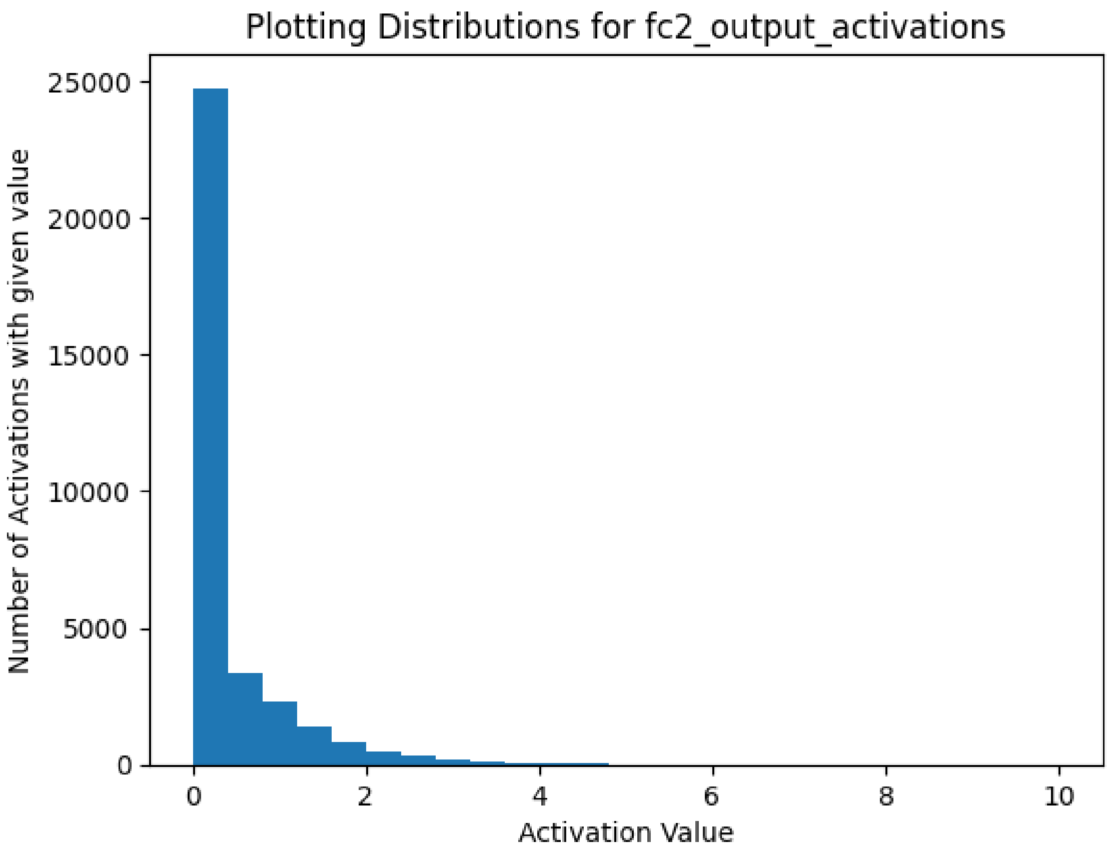

```
plot_distribution(conv2_output_activations, "conv2_output_activations")
```



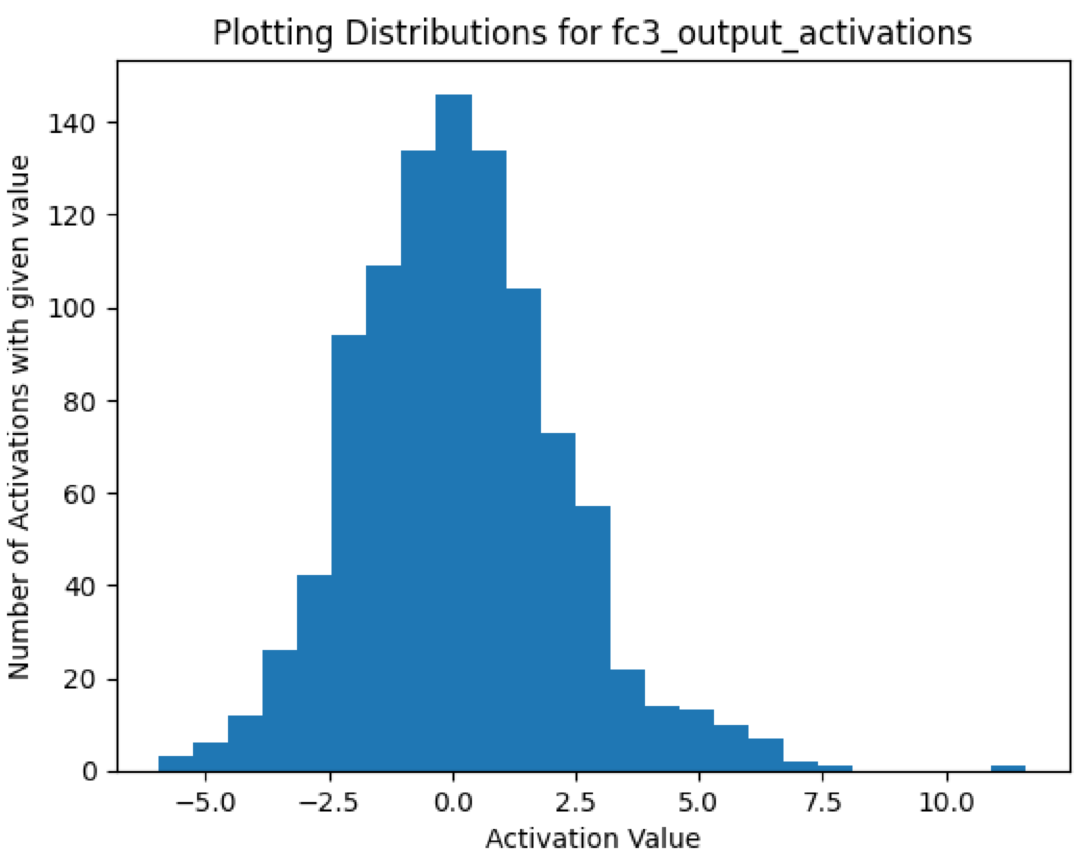

```
plot_distribution(fc1_output_activations, "fc1_output_activations")
```




```
plot_distribution(fc2_output_activations, "fc2_output_activations")
```




```
plot_distribution(fc3_output_activations, "fc3_output_activations")
```



```
# Printing the range and 3-sigma range for each layer
compute_range(input_activations, "input_activations")
compute_range(conv1_output_activations, "conv1_output_activations")
compute_range(conv2_output_activations, "conv2_output_activations")
compute_range(fc1_output_activations, "fc1_output_activations")
compute_range(fc2_output_activations, "fc2_output_activations")
compute_range(fc3_output_activations, "fc3_output_activations")
```

```
input_activations
range: (-1.0, 1.0) = 2.0000
3-sigma range: (-1.5687, 1.4720) = 3.0407
=========
conv1_output_activations
range: (0.0, 9.342909812927246) = 9.3429
3-sigma range: (-1.7463, 2.8820) = 4.6283
=========
conv2_output_activations
range: (0.0, 14.358724594116211) = 14.3587
3-sigma range: (-2.7615, 4.3134) = 7.0749
=========
fc1_output_activations
range: (0.0, 14.740952491760254) = 14.7410
3-sigma range: (-2.5942, 3.5153) = 6.1095
=========
fc2_output_activations
range: (0.0, 10.012285232543945) = 10.0123
3-sigma range: (-1.6809, 2.3931) = 4.0740
=========
fc3_output_activations
range: (-5.945919513702393, 11.618141174316406) = 17.5641
3-sigma range: (-6.1796, 6.5012) = 12.6808
=========
```

Question 4: Quantize Activations

```
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 sign
            if (x != x.round()).any():
                raise Exception("Input to {} layer has non-integer values".format(l._

        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, lay
        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 th

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

    Returns:
    float: A scaling factor that the input should be multiplied by before being fed i
           This value does not need to be an 8-bit integer.
    '''
    return 255/(np.max(pixels) - np.min(pixels))

@staticmethod
def quantize_activations(activations: np.ndarray, n_w: float, n_initial_input: 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 thi
    n_w (float): The scale by which the weights of this layer were multiplied as part
    n_initial_input (float): The scale by which the initial input to the neural netwo
    ns ([(float, float)]): A list of tuples, where each tuple represents the "weight 

    Returns:
    float: A scaling factor that the layer output should be multiplied by before bein
           This value does not need to be an 8-bit integer.
    '''

    activations = activations* n_w.item()
    activations = activations * n_initial_input.item()


    for pair in ns:
      activations = activations* pair[0].item() * pair[1].item()
```

```
        scale = 255/(np.max(activations) - np.min(activations))

        return scale

    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, 
        #   * torch.Tensor.round
        #   * torch.clamp

        x = torch.clamp(torch.Tensor.round(x*self.input_scale), min = -128, max = 127)
        x = self.pool(F.relu(self.conv1(x)))

        x = torch.clamp(torch.Tensor.round(x * self.conv1.output_scale), min=-128, max=12

        x = self.pool(F.relu(self.conv2(x)))


        x = torch.clamp(torch.Tensor.round(x * self.conv2.output_scale), min=-128, max=12

        x = x.view(-1, 16 * 5 * 5)
        x = F.relu(self.fc1(x))

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

        x = F.relu(self.fc2(x))

        x = torch.clamp(torch.Tensor.round(x * self.fc2.output_scale), min=-128, max=127)

        x = self.fc3(x)

        x = torch.clamp(torch.Tensor.round(x * self.fc3.output_scale), min=-128, max=127)


        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
```

```
Accuracy of the network after quantizing both weights and activations: 52.2%
```

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)
```

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

```
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
```

```
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.ch

        self.fc3.bias.data = NetQuantizedWithBias.quantized_bias(
            self.fc3.bias.data,
            self.fc3.weight.scale
```

```
            self.fc3.weight.scale,
            self.input_scale,
            preceding_scales
        )

        if (self.fc3.bias.data < -2147483648).any() or (self.fc3.bias.data > 2147483647).
            raise Exception("Bias has values which are out of bounds for an 32-bit signed
        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[T
        '''
        Quantize the bias so that all values are integers between -2147483648 and 2147483

        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 netwo
        ns ([(float, float)]): A list of tuples, where each tuple represents the "weight

        Returns:
        Tensor: The bias in quantized form, where every value is an integer between -2147
                The "dtype" will still be "float", but the values themselves should all b
        '''

        bias *= n_initial_input.item()
        bias *= n_w.item()
        for (first, second) in ns:
          bias *= first.item()
          bias *= second.item()

        scale = 255/(torch.max(bias) - torch.min(bias))
        return torch.clamp((bias * scale).round(), min=-127, max=128)


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
```

```
e network on the test images after all the weights and the bias are quantized: 49.39%
```

✓ 0s completed at 1:28 AM