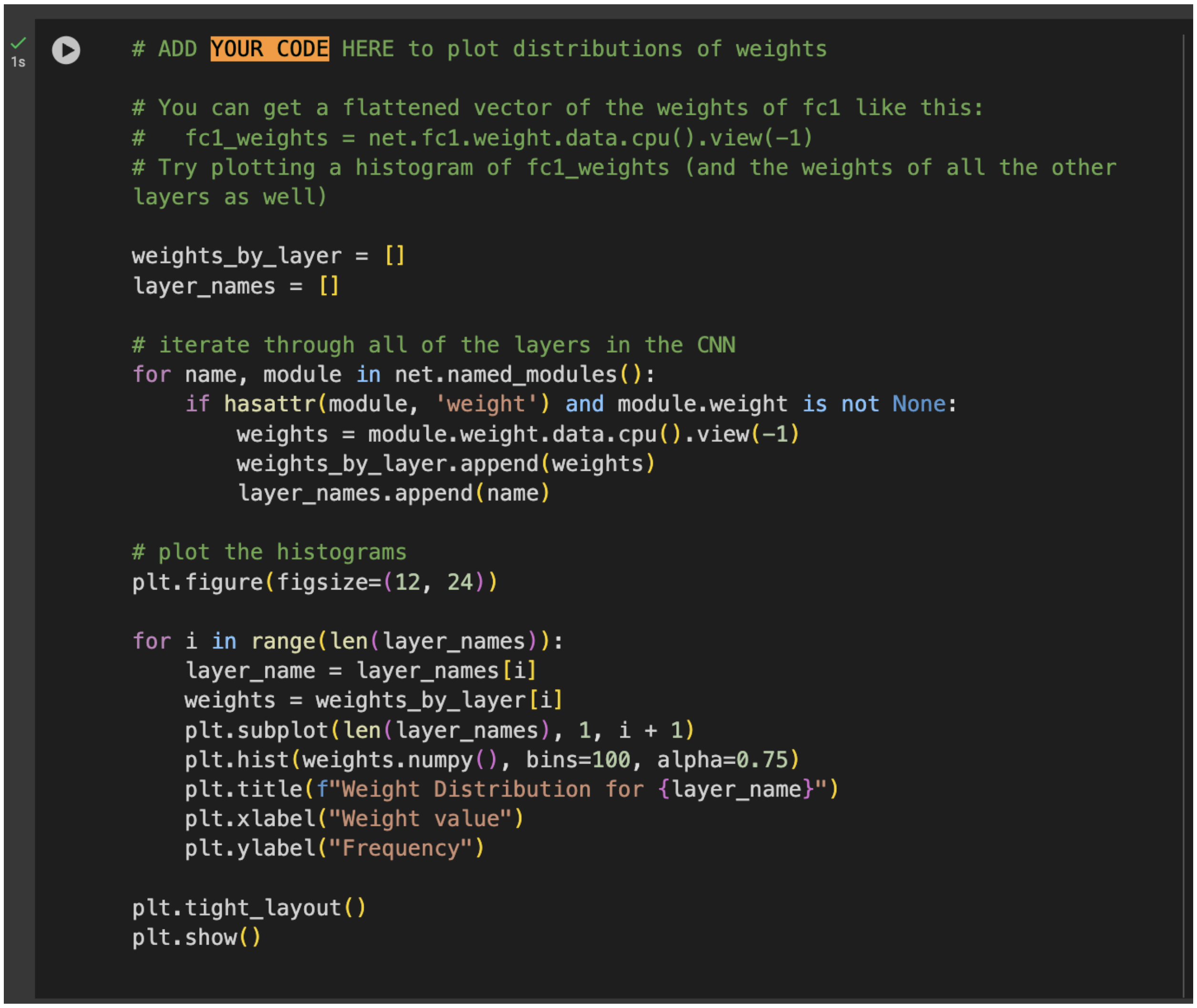
Name: Sriraman Iyengar

ID: si2468

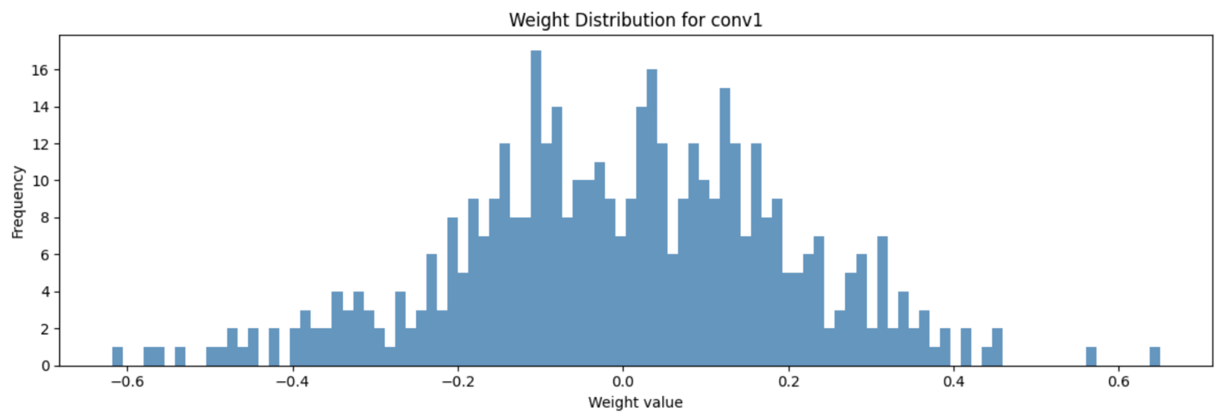
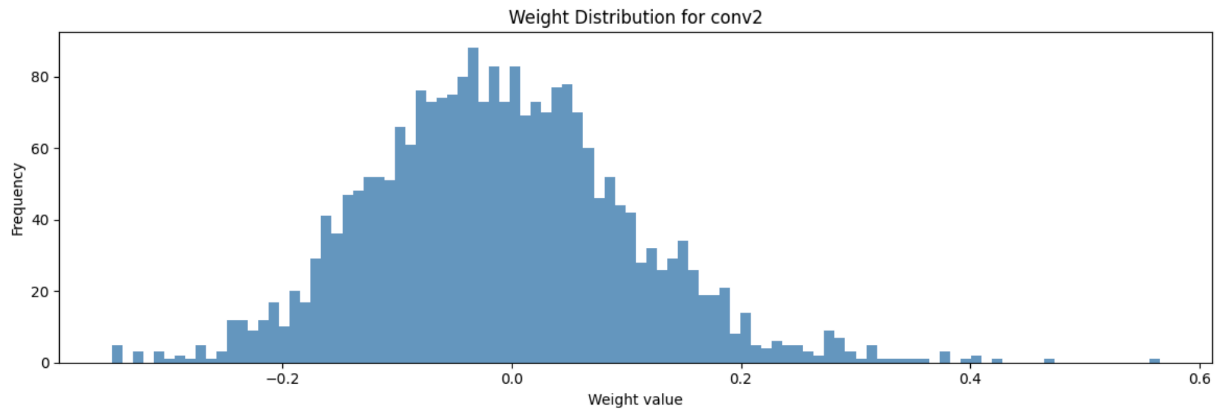
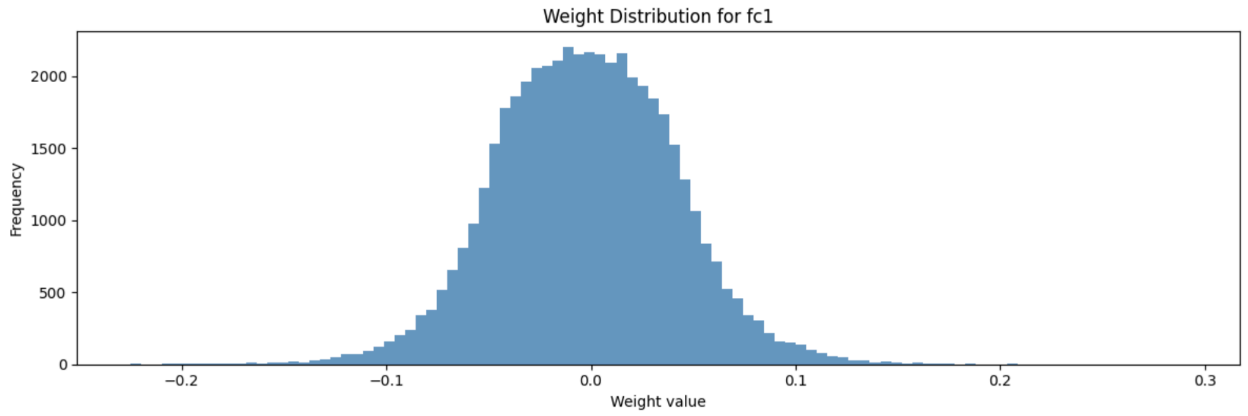
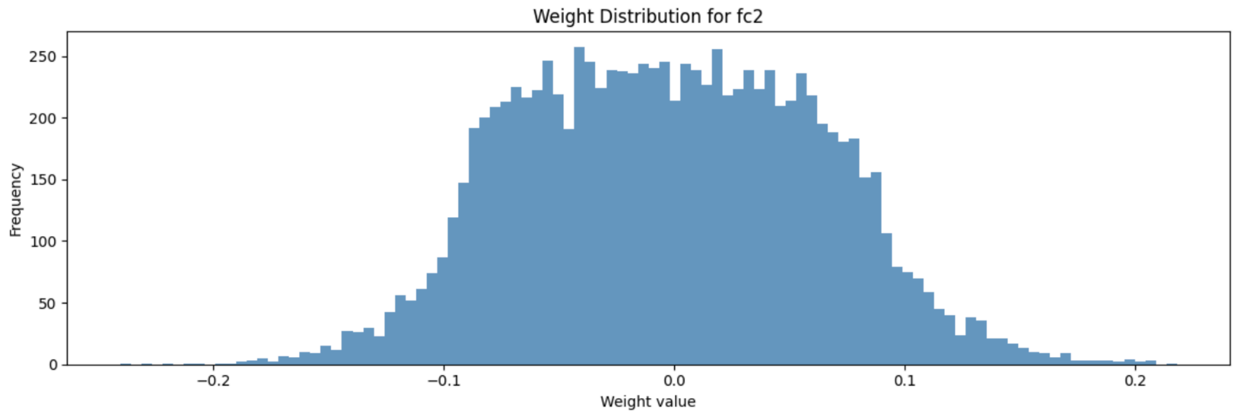
Question 1:

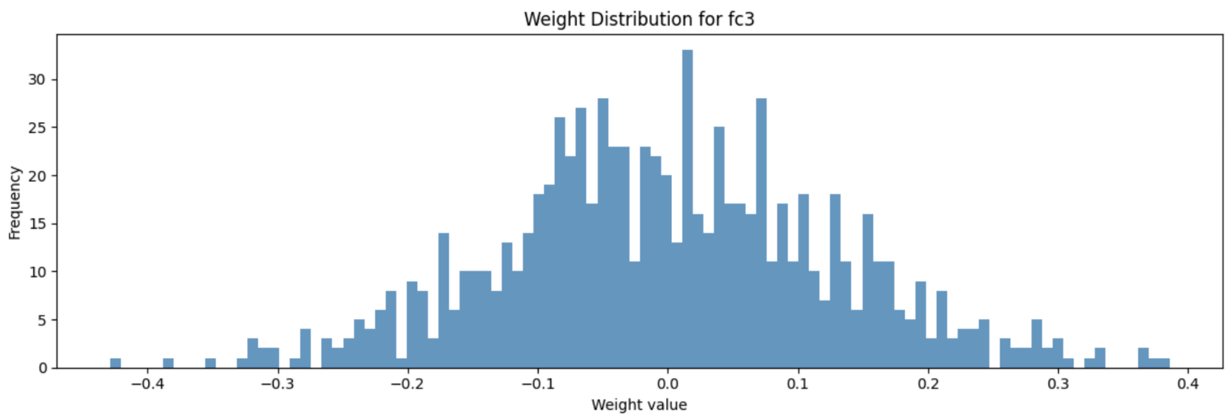
Visualizing the different layers



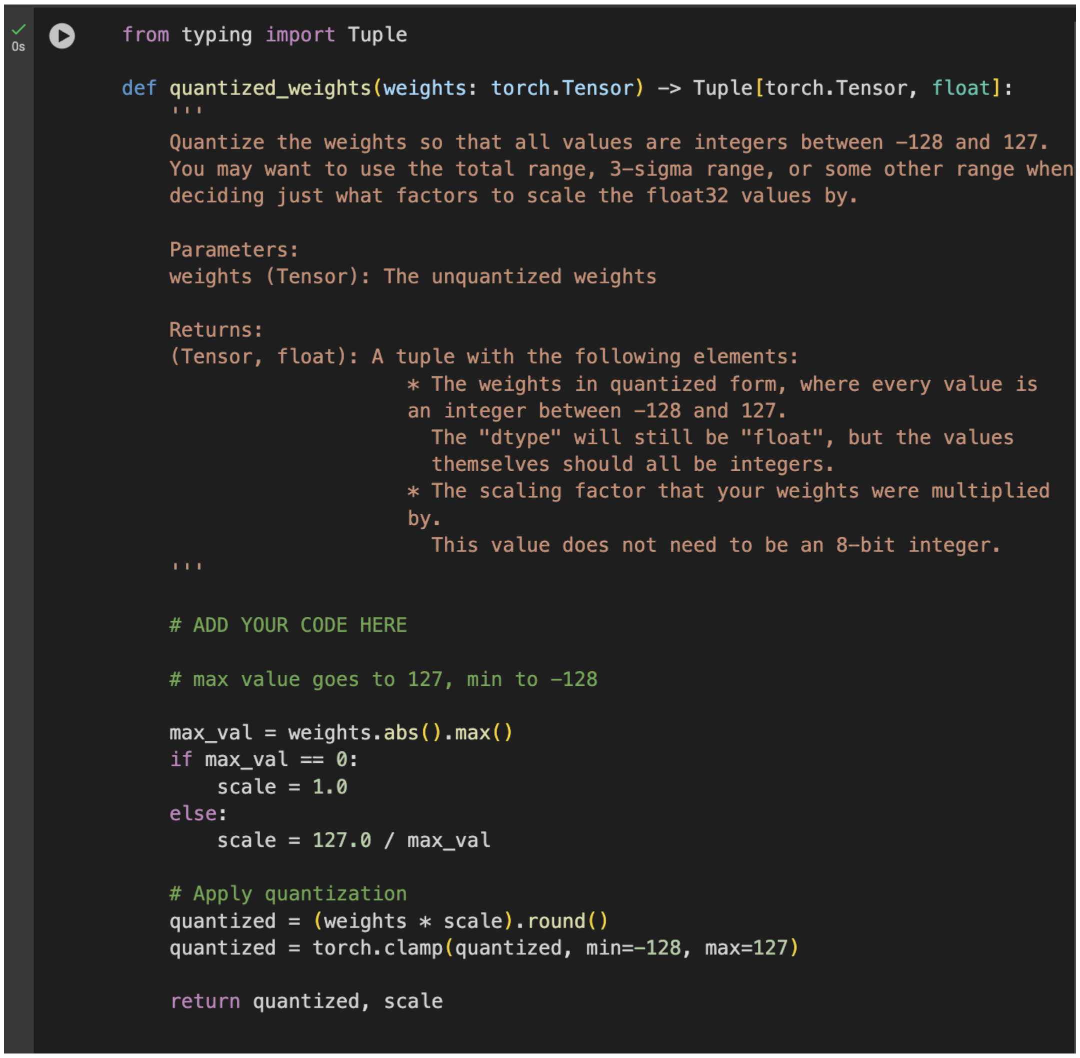
```
# ADD YOUR CODE HERE to plot distributions of weights

# 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)

weights_by_layer = []
layer_names = []

# iterate through all of the layers in the CNN
for name, module in net.named_modules():
    if hasattr(module, 'weight') and module.weight is not None:
        weights = module.weight.data.cpu().view(-1)
        weights_by_layer.append(weights)
        layer_names.append(name)

# plot the histograms
plt.figure(figsize=(12, 24))

for i in range(len(layer_names)):
    layer_name = layer_names[i]
    weights = weights_by_layer[i]
    plt.subplot(len(layer_names), 1, i + 1)
    plt.hist(weights.numpy(), bins=100, alpha=0.75)
    plt.title(f"Weight Distribution for {layer_name}")
    plt.xlabel("Weight value")
    plt.ylabel("Frequency")

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

Plots for visualizing layers


Weight Distribution for conv1
Frequency
Weight value

Weight Distribution for conv2
Frequency
Weight value

Weight Distribution for fc1
Frequency
Weight value

Weight Distribution for fc2
Frequency
Weight value



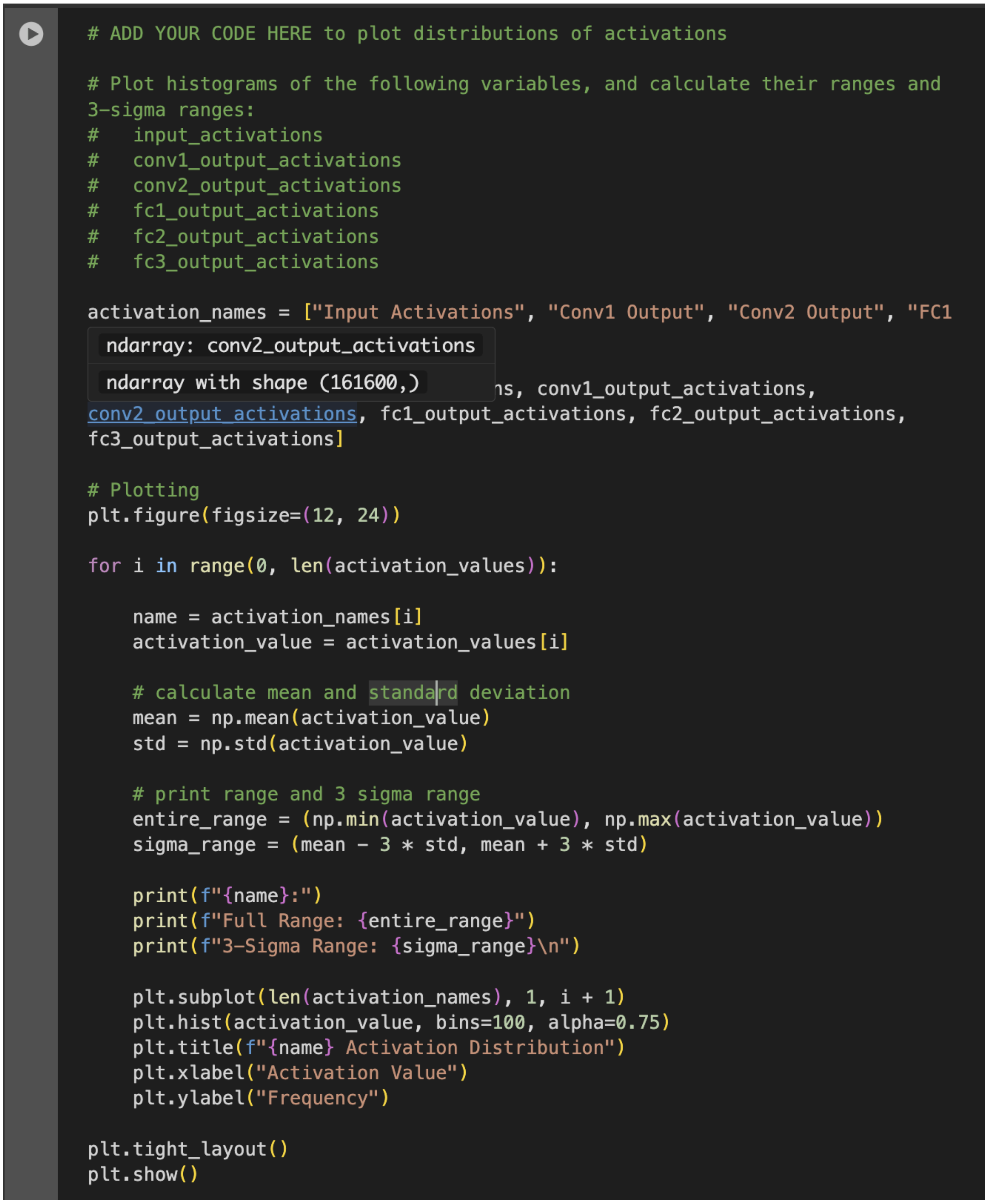
Question 2:



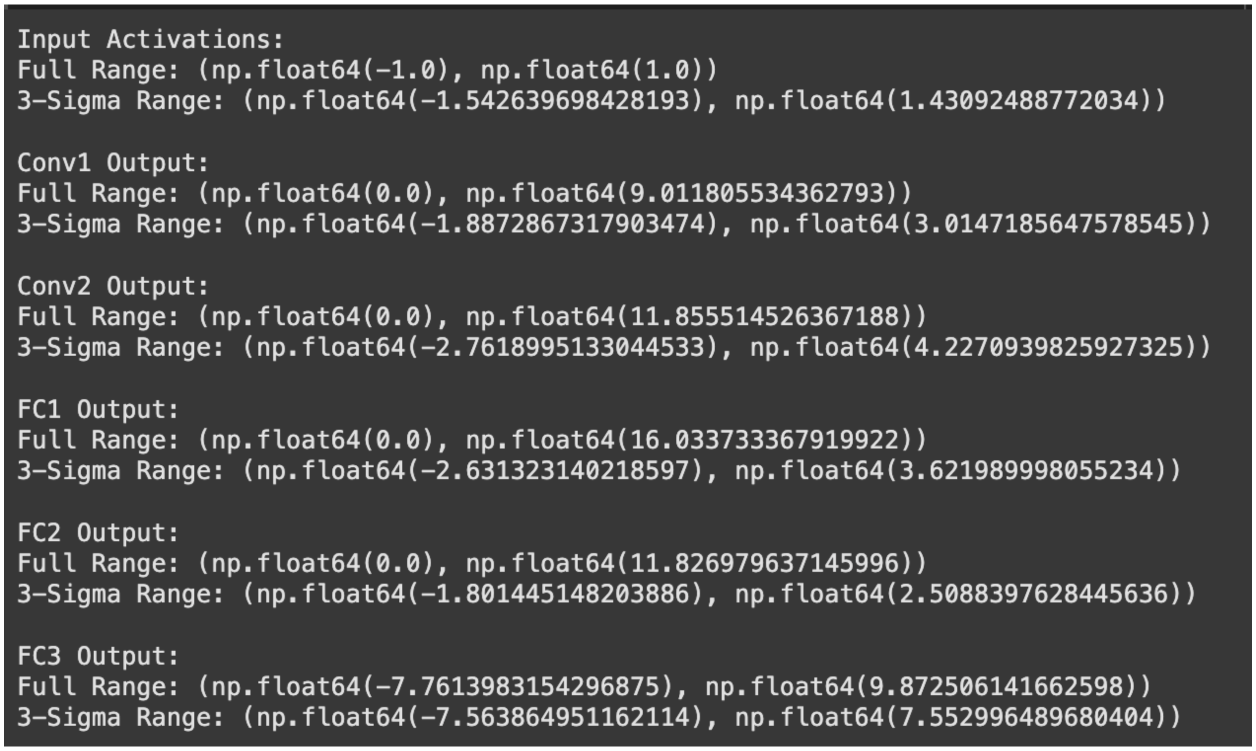
```python
from typing import Tuple

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 value goes to 127, min to −128

    max_val = weights.abs().max()
    if max_val == 0:
        scale = 1.0
    else:
        scale = 127.0 / max_val

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

    return quantized, scale
```

Question 3:

Code:

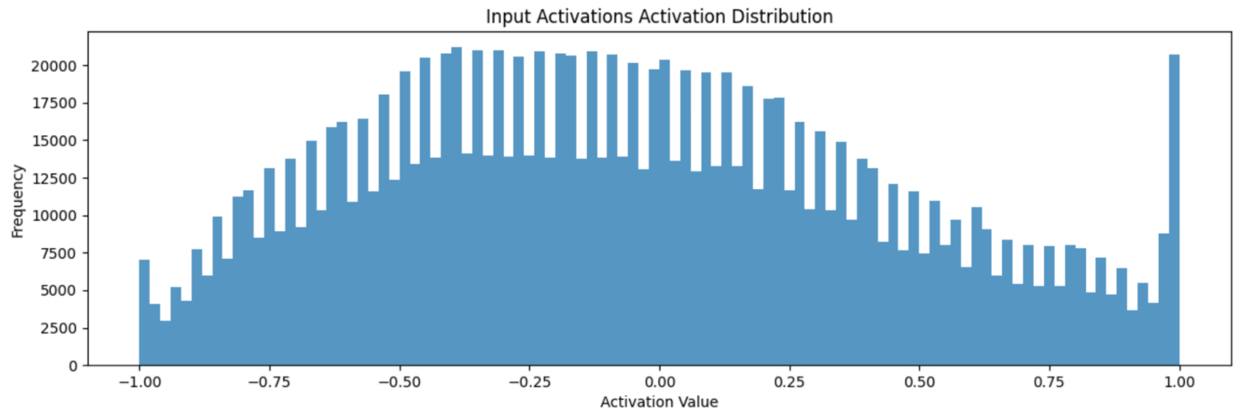


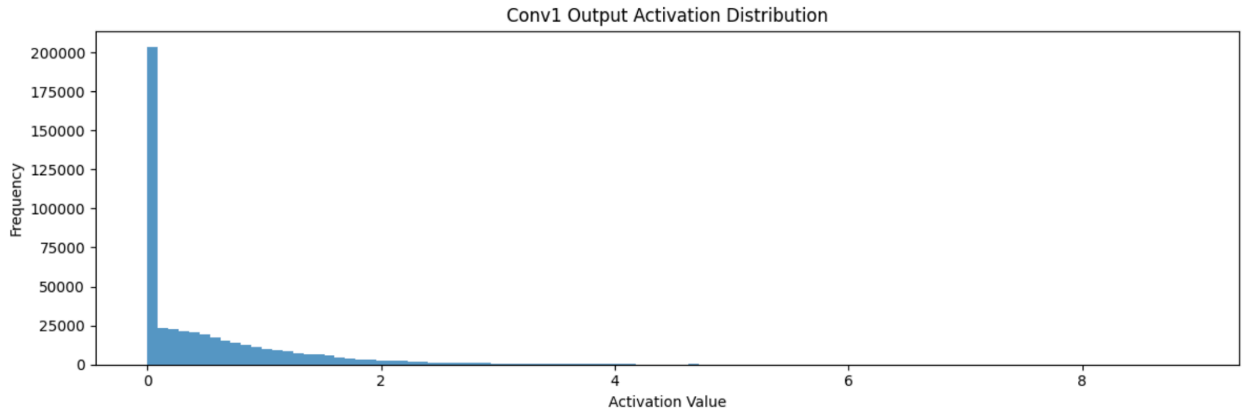
```
# ADD YOUR CODE HERE to plot distributions of activations

# 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_names = ["Input Activations", "Conv1 Output", "Conv2 Output", "FC1
 ndarray: conv2_output_activations

 ndarray with shape (161600,)                ns, conv1_output_activations,
conv2_output_activations, fc1_output_activations, fc2_output_activations,
fc3_output_activations]

# Plotting
plt.figure(figsize=(12, 24))

for i in range(0, len(activation_values)):

    name = activation_names[i]
    activation_value = activation_values[i]

    # calculate mean and standard deviation
    mean = np.mean(activation_value)
    std = np.std(activation_value)

    # print range and 3 sigma range
    entire_range = (np.min(activation_value), np.max(activation_value))
    sigma_range = (mean - 3 * std, mean + 3 * std)

    print(f"{name}:")
    print(f"Full Range: {entire_range}")
    print(f"3-Sigma Range: {sigma_range}\n")

    plt.subplot(len(activation_names), 1, i + 1)
    plt.hist(activation_value, bins=100, alpha=0.75)
    plt.title(f"{name} Activation Distribution")
    plt.xlabel("Activation Value")
    plt.ylabel("Frequency")

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

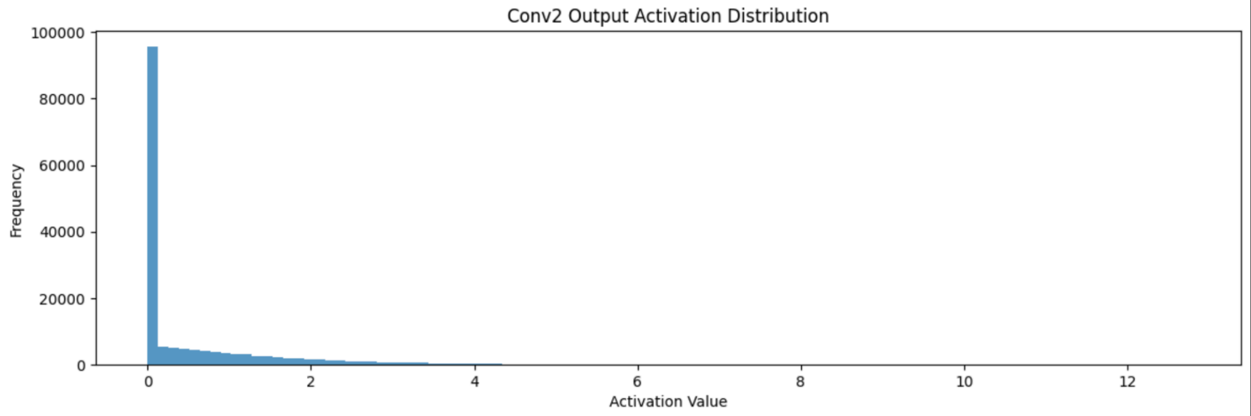


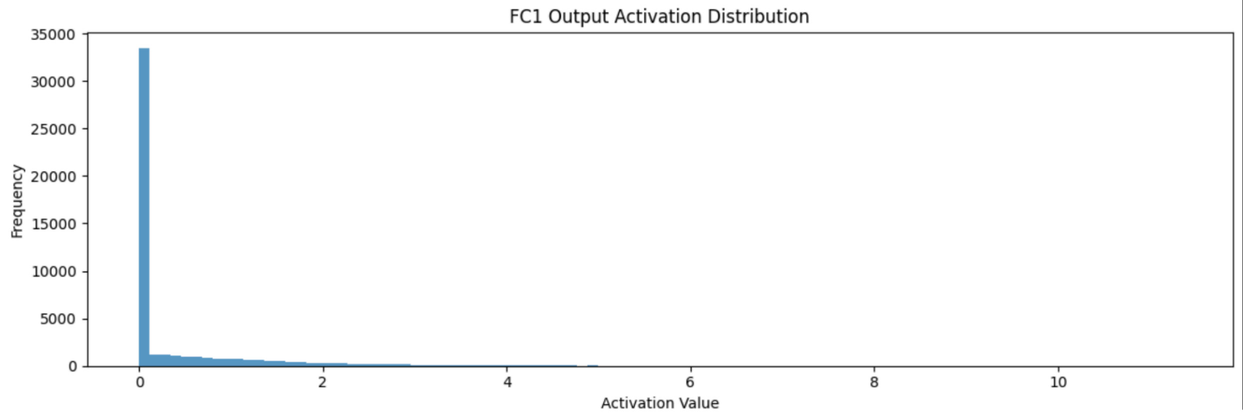
```
Input Activations:
Full Range: (np.float64(-1.0), np.float64(1.0))
3-Sigma Range: (np.float64(-1.542639698428193), np.float64(1.43092488772034))

Conv1 Output:
Full Range: (np.float64(0.0), np.float64(9.011805534362793))
3-Sigma Range: (np.float64(-1.8872867317903474), np.float64(3.0147185647578545))

Conv2 Output:
Full Range: (np.float64(0.0), np.float64(11.855514526367188))
3-Sigma Range: (np.float64(-2.7618995133044533), np.float64(4.2270939825927325))

FC1 Output:
Full Range: (np.float64(0.0), np.float64(16.033733367919922))
3-Sigma Range: (np.float64(-2.631323140218597), np.float64(3.621989998055234))

FC2 Output:
Full Range: (np.float64(0.0), np.float64(11.826979637145996))
3-Sigma Range: (np.float64(-1.801445148203886), np.float64(2.5088397628445636))

FC3 Output:
Full Range: (np.float64(-7.7613983154296875), np.float64(9.872506141662598))
3-Sigma Range: (np.float64(-7.563864951162114), np.float64(7.552996489680404))
```

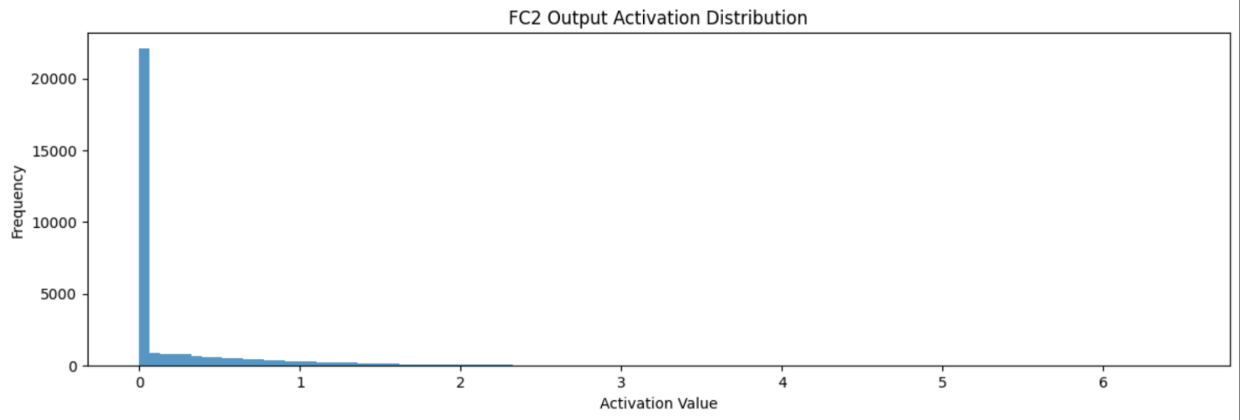
Visualizations:

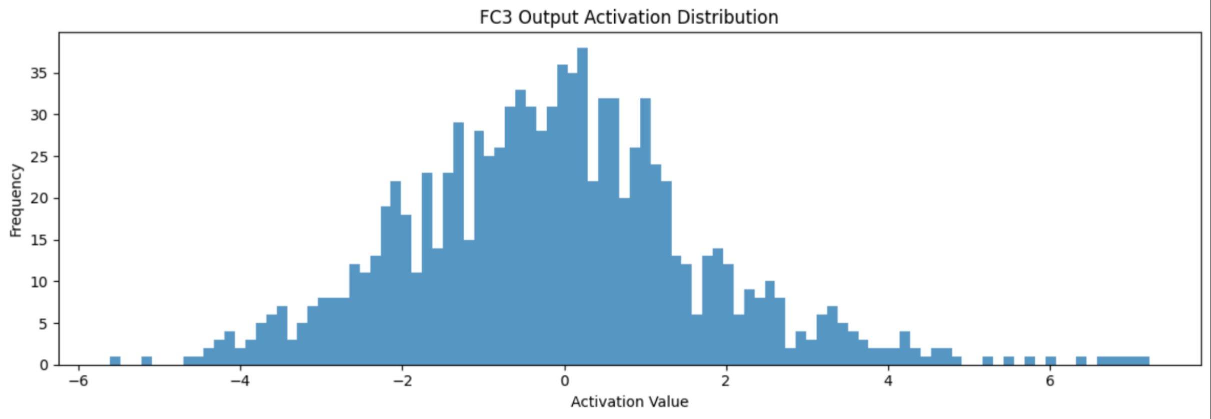

Input Activations Activation Distribution
Frequency
Activation Value


Conv1 Output Activation Distribution
Frequency
Activation Value

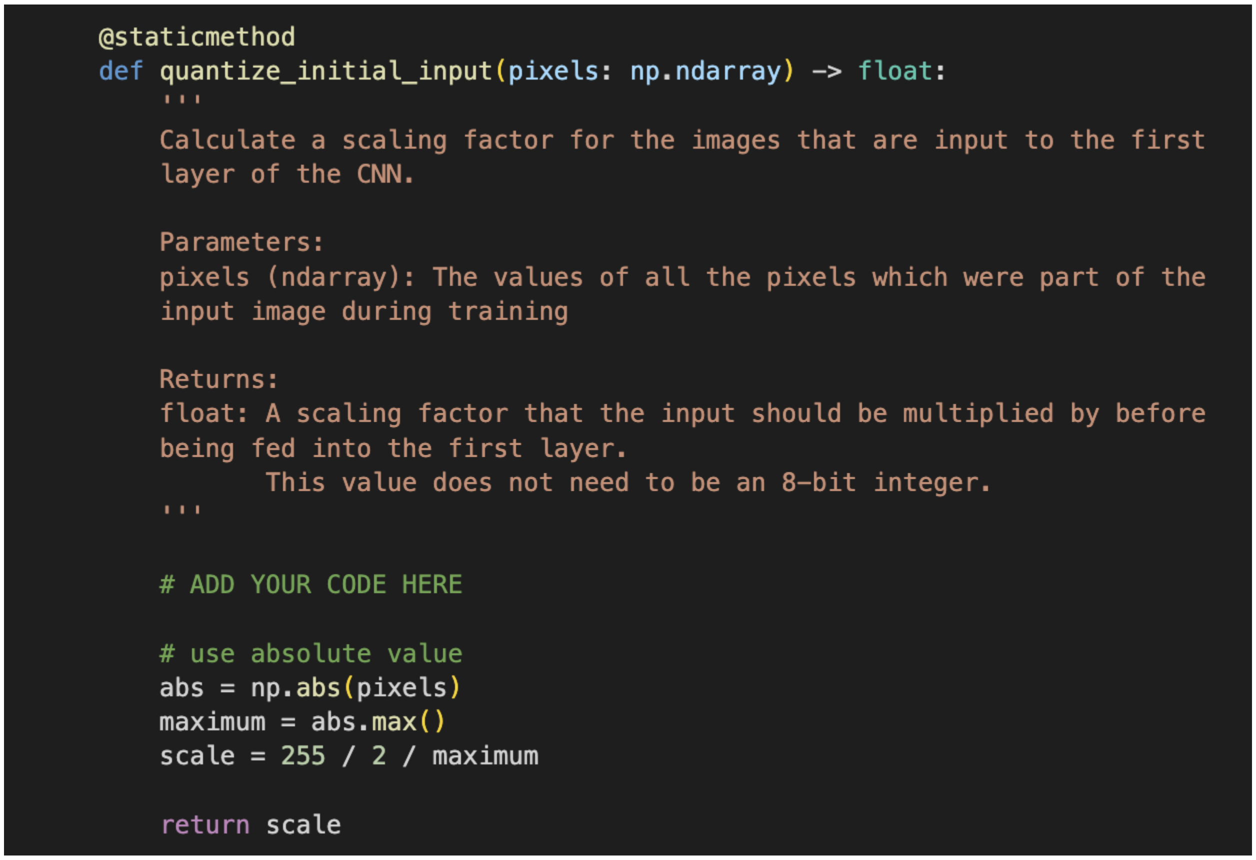

Conv2 Output Activation Distribution
Frequency
Activation Value


FC1 Output Activation Distribution
Frequency
Activation Value

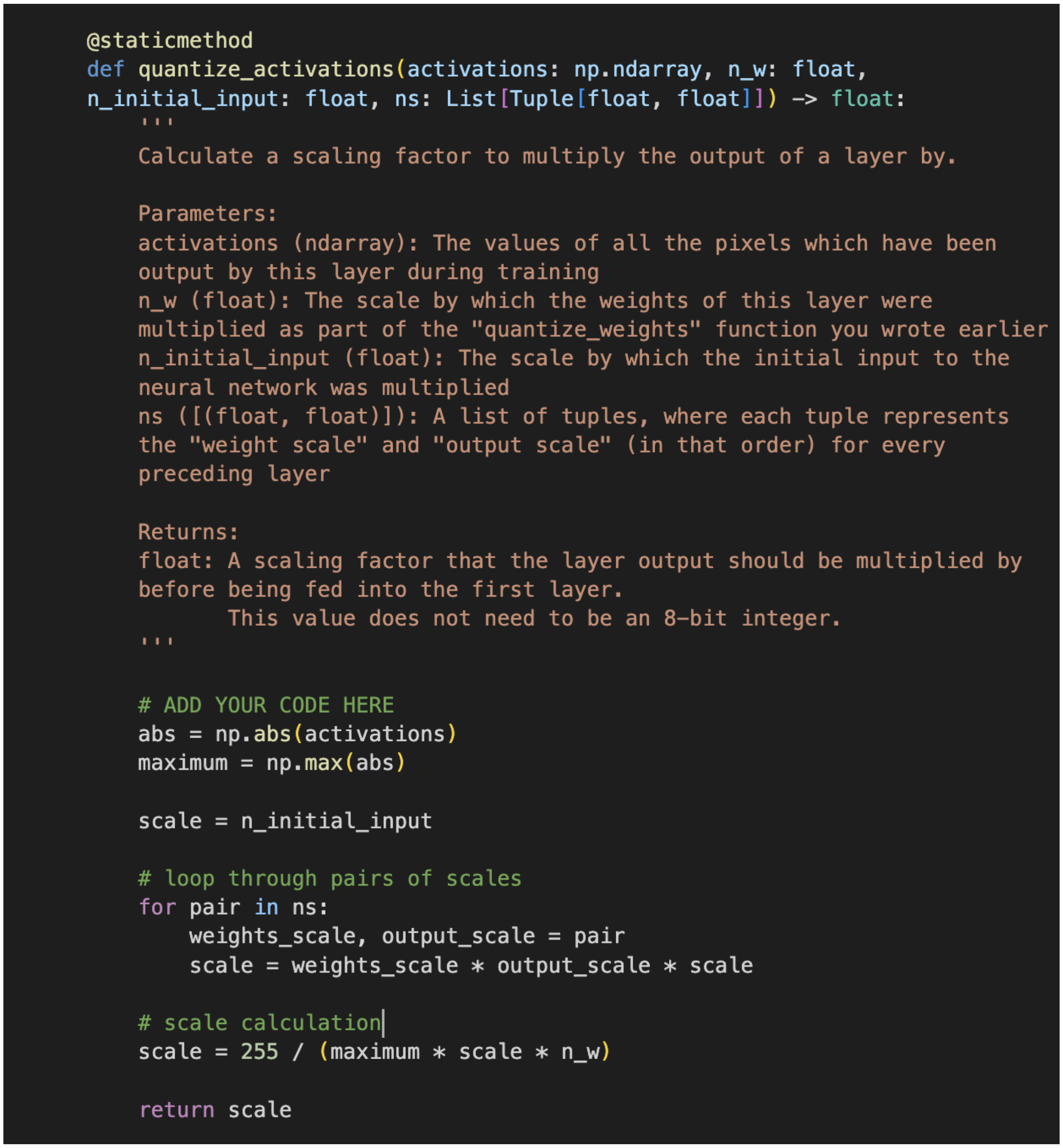

FC2 Output Activation Distribution
Frequency
Activation Value
0
5000
10000
15000
20000
0
1
2
3
4
5
6


FC3 Output Activation Distribution
Frequency
Activation Value
0
5
10
15
20
25
30
35
−6
−4
−2
0
2
4
6

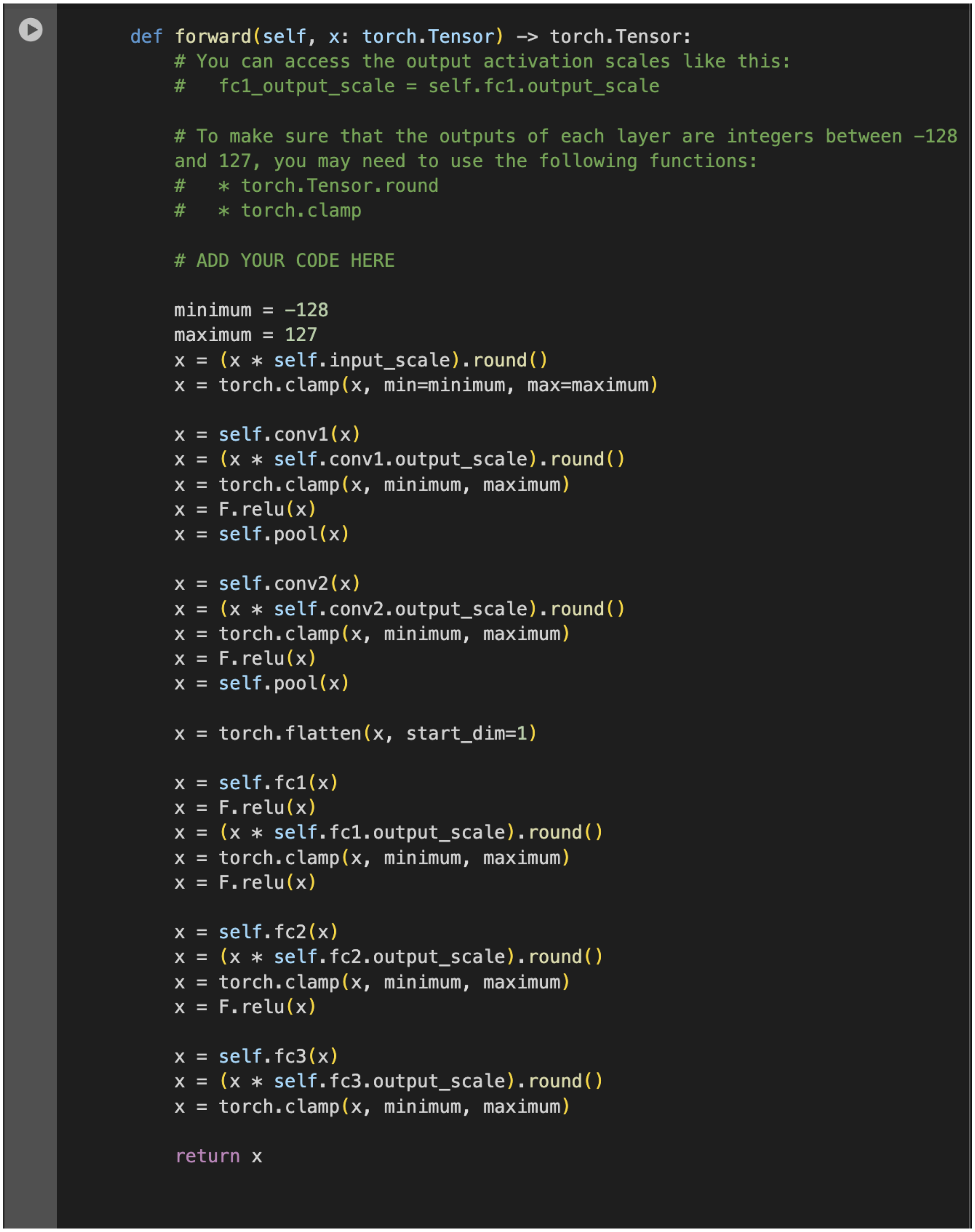
Question 4:



```python
@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

    # use absolute value
    abs = np.abs(pixels)
    maximum = abs.max()
    scale = 255 / 2 / maximum

    return scale
```

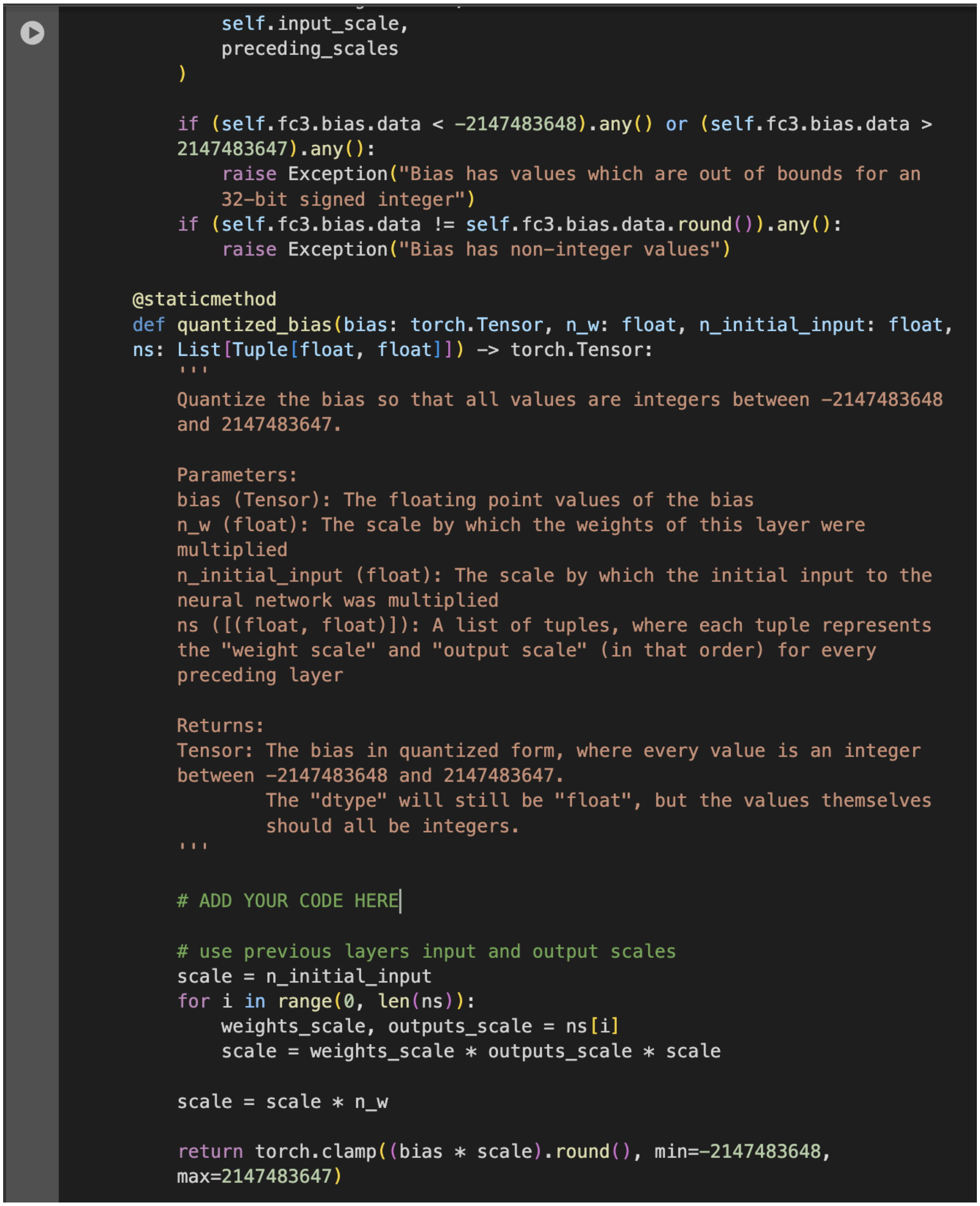


```
@staticmethod
def quantize_activations(activations: np.ndarray, n_w: float,
n_initial_input: float, ns: List[Tuple[float, float]]) -> float:
    '''
    Calculate a scaling factor to multiply the output of a layer by.

    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
    abs = np.abs(activations)
    maximum = np.max(abs)

    scale = n_initial_input

    # loop through pairs of scales
    for pair in ns:
        weights_scale, output_scale = pair
        scale = weights_scale * output_scale * scale

    # scale calculation
    scale = 255 / (maximum * scale * n_w)

    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, you may need to use the following functions:
        #   * torch.Tensor.round
        #   * torch.clamp

        # ADD YOUR CODE HERE

        minimum = -128
        maximum = 127
        x = (x * self.input_scale).round()
        x = torch.clamp(x, min=minimum, max=maximum)

        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)

        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)

        x = torch.flatten(x, start_dim=1)

        x = self.fc1(x)
        x = F.relu(x)
        x = (x * self.fc1.output_scale).round()
        x = torch.clamp(x, minimum, maximum)
        x = F.relu(x)

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

        x = self.fc3(x)
        x = (x * self.fc3.output_scale).round()
        x = torch.clamp(x, minimum, maximum)

        return x
```

Question 5:



```
            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

        # use previous layers input and output scales
        scale = n_initial_input
        for i in range(0, len(ns)):
            weights_scale, outputs_scale = ns[i]
            scale = weights_scale * outputs_scale * scale

        scale = scale * n_w

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

Question 6:

1. GPTQ’s main innovations that allow for efficient quantizations of large models are arbitrary order insight, lazy batch updates, Cholesky Reformulation, and Approximate second order quantization.
2. There is a runtime complexity reduction in the way that we move from a row-wise greedy update to a shared column update. The new technique also addresses the memory throughput bottleneck by batching updates within blocks with lazy batching. Additionally, the Cholesky reformulation allows for increased numerical stability for large models.
3. One discussed limitation is that the method does not increase the speed of matrix multiplication. A possible solution for this in future work would be to develop the hardware support necessary for mixed precision multiplication.