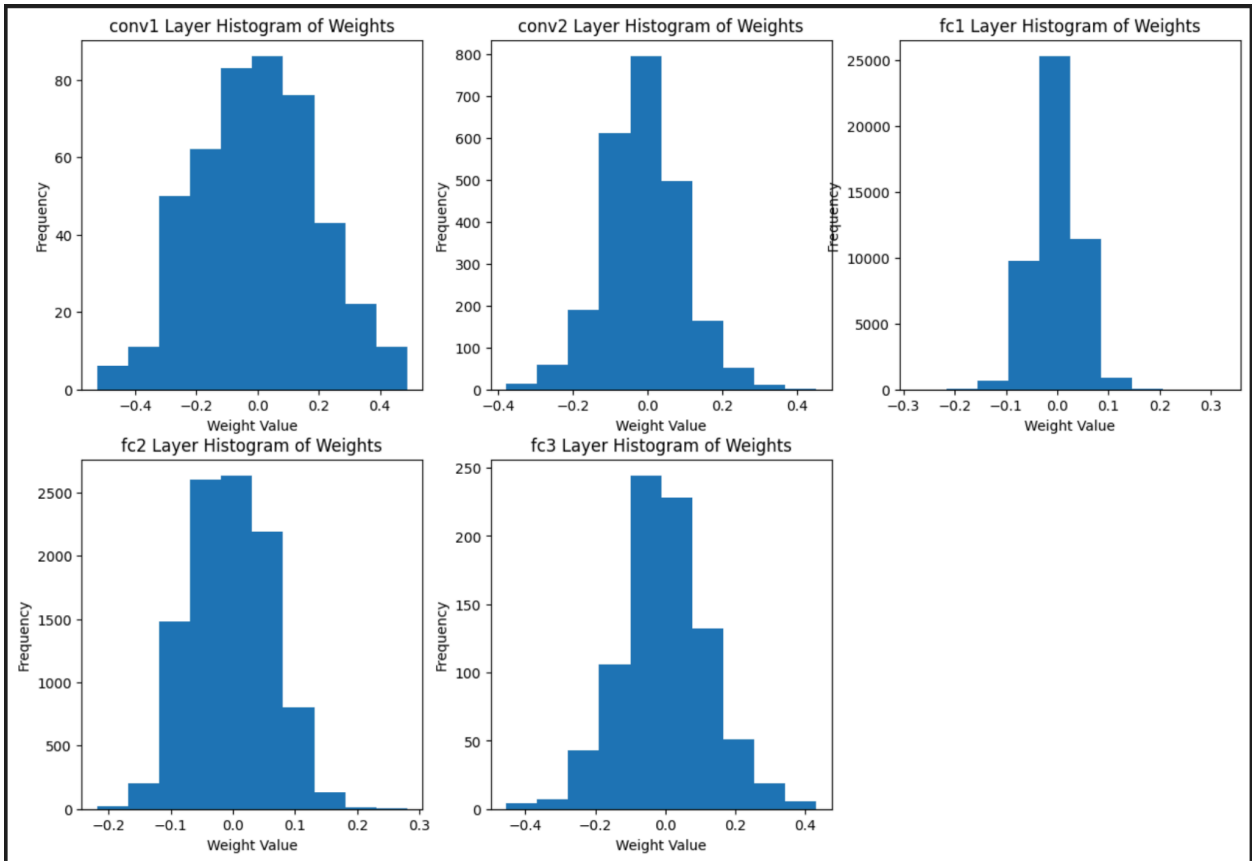
Name: Pranav Hariharane
CUID: pkh2120

Question 1: Visualize Weights
Code Snippet

```
# 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)

# Get a flattened vector of the weights for each layer
layers = {
    'conv1': net.conv1.weight.data.cpu().view(-1),
    'conv2': net.conv2.weight.data.cpu().view(-1),
    'fc1': net.fc1.weight.data.cpu().view(-1),
    'fc2': net.fc2.weight.data.cpu().view(-1),
    'fc3': net.fc3.weight.data.cpu().view(-1)
}

# Create subplots for histograms
fig, axes = plt.subplots(2, 3, figsize=(15, 10))

# Convert multi-dimensional subplot axes array into a 1D array
axes = axes.flatten()
i = 0
for key in layers:
    # Get layer name and weights
    layer_name = key
    layer_weights = layers[key]

    # Convert weights into a numpy array
    weights = layer_weights.numpy()

    # Plot histogram of weights for layer
    axes[i].hist(weights)
    axes[i].set_title(layer_name + " Layer Histogram of Weights")
    axes[i].set_xlabel('Weight Value')
    axes[i].set_ylabel('Frequency')
    i += 1

# Remove the extra subplot
axes[-1].axis('off')
plt.show()
```

Plots



Question 2: Quantize Weights

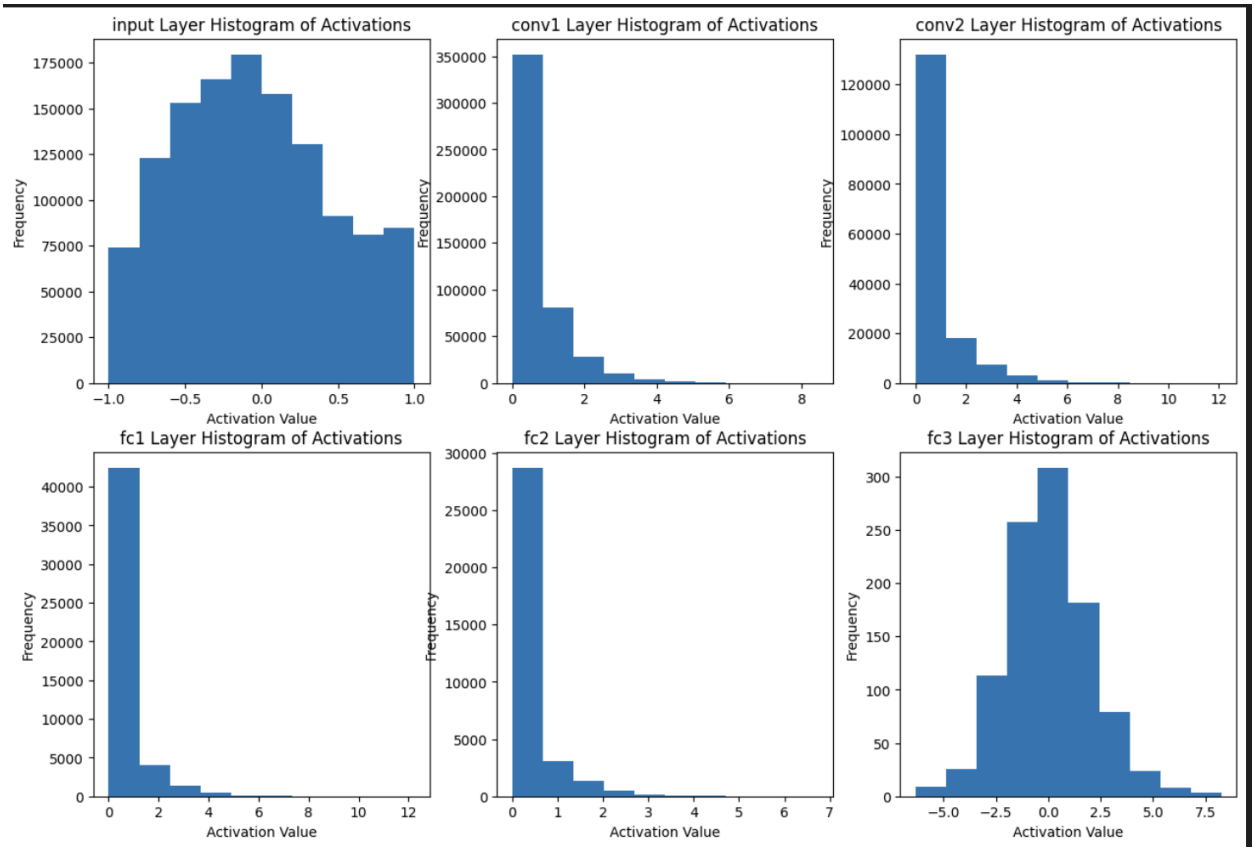
Code Snippet

```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

    # Compute scaling factor using max absolute value
    max_val = weights.abs().max()
    scale = 0.0
    if max_val == 0:
      scale = 1.0
    else:
      scale = 127.0 / max_val

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

Question 3: Visualize Activations
Code Snippet

```
# 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

# Plot distributions of activations for each layer

activations = {
    'input': input_activations,
    'conv1': conv1_output_activations,
    'conv2': conv2_output_activations,
    'fc1': fc1_output_activations,
    'fc2': fc2_output_activations,
    'fc3': fc3_output_activations
}

statistics = {
    'input': {},
    'conv1': {},
    'conv2': {},
    'fc1': {},
    'fc2': {},
    'fc3': {}
}

# Create subplots for histograms
fig, axes = plt.subplots(2, 3, figsize=(15, 10))

# Convert multi-dimensional subplot axes array into a 1D array
axes = axes.flatten()
i = 0
for key in activations:
    # Get layer name and activations
    layer_name = key
    layer_activations = activations[key]

    # Store statistics(full range, 3-sigma range) for layer
    activations_range = (layer_activations.min(), layer_activations.max())
    statistics[key]['full_range'] = activations_range
    activations_mean = layer_activations.mean()
    activations_std = layer_activations.std()
    statistics[key]['3-sigma_range'] = (activations_mean - 3 * activations_std, activations_mean + 3 * activations_std)

    # Plot histogram of activations for layer
    axes[i].hist(layer_activations)
    axes[i].set_title(layer_name + " Layer Histogram of Activations")
    axes[i].set_xlabel('Activation Value')
    axes[i].set_ylabel('Frequency')
    i += 1

plt.show()

for key in statistics:
    print(key + " Layer")
    print("Range: " + str(statistics[key]['full_range']))
    print("3-sigma range: " + str(statistics[key]['3-sigma_range']))
    print()
    print()
```

Plots



Print Output of Full Ranges and 3-Sigma Ranges for Layer Activations

```
input Layer
Range: (np.float64(-1.0), np.float64(1.0))
3-sigma range: (np.float64(-1.5743796559277579), np.float64(1.473554327933634))


conv1 Layer
Range: (np.float64(0.0), np.float64(8.451160430908203))
3-sigma range: (np.float64(-1.8142897196958079), np.float64(2.945701936618352))


conv2 Layer
Range: (np.float64(0.0), np.float64(12.093034744262695))
3-sigma range: (np.float64(-2.615422996719322), np.float64(3.7752142378726514))


fc1 Layer
Range: (np.float64(0.0), np.float64(12.227825164794922))
3-sigma range: (np.float64(-2.225321901042535), np.float64(3.0405650982243815))


fc2 Layer
Range: (np.float64(0.0), np.float64(6.738529205322266))
3-sigma range: (np.float64(-1.456991068222094), np.float64(2.012357180982864))

...
Range: (np.float64(-6.32207727432251), np.float64(8.258025169372559))
3-sigma range: (np.float64(-6.0121699164581175), np.float64(6.057465501364416))
```

Question 4: Quantize Activations
Code Snippets

```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
    # Compute scaling factor using max absolute value
    max_val = np.max(np.abs(pixels))
    scale = 0.0
    if max_val == 0:
      scale = 1.0
    else:
      scale = 127.0 / max_val

    return scale
```

```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.
    '''

    # ADD YOUR CODE HERE
    # Compute scaling factor using max absolute value
    max_val = np.max(np.abs(activations))
    if max_val == 0:
      return 1.0

    # Calculate the cumulative input scale to this layer
    cumulative_input_scale = n_initial_input
    for weight_scale, output_scale in ns:
        # Each layer multiplies by weight_scale and by output_scale
        cumulative_input_scale *= (weight_scale * output_scale)

    # The scale of the output would be: cumulative_input_scale * n_w
    cumulative_input_scale *= n_w

    # Compute scaling factor using max absolute value of activations
    output_scale = 127.0 / (max_val * cumulative_input_scale)

    return output_scale
```

```
    return output_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
    # scale input and clamp input to [-128, 127]
    x = x * self.input_scale
    x = torch.clamp(x.round(), min=-128, max=127)

    # Multiply output of conv1 layer by its output scale and clamp output to [-128, 127]
    x = self.conv1(x)
    x = x * self.conv1.output_scale
    x = torch.clamp(x.round(), min=-128, max=127)
    # ReLU
    x = F.relu(x)
    # Pooling
    x = self.pool(x)

    # Multiply output of conv2 layer by its output scale and clamp output to [-128, 127]
    x = self.conv2(x)
    x = x * self.conv2.output_scale
    x = torch.clamp(x.round(), min=-128, max=127)
    # ReLU
    x = F.relu(x)
    # Pooling
    x = self.pool(x)

    # Flatten
    x = torch.flatten(x, 1)

    # Multiply output of fc1 layer by its output scale and clamp output to [-128, 127]
    x = self.fc1(x)
    x = x * self.fc1.output_scale
    x = torch.clamp(x.round(), min=-128, max=127)
    # ReLU
    x = F.relu(x)

    # Multiply output of fc2 layer by its output scale and clamp output to [-128, 127]
    x = self.fc2(x)
    x = x * self.fc2.output_scale
    x = torch.clamp(x.round(), min=-128, max=127)
    # ReLU
    x = F.relu(x)

    # Return output of fc3 layer
    x = self.fc3(x)
    x = x * self.fc3.output_scale
    x = torch.clamp(x.round(), min=-128, max=127)
    # ReLU
    x = F.relu(x)

    return x
```

Question 5: Quantize Biases
Code Snippet

```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
        # Calculate the cumulative input scale to this layer
        cumulative_input_scale = n_initial_input
        for weight_scale, output_scale in ns:
            # Each layer multiplies by weight_scale and by output_scale
            cumulative_input_scale *= (weight_scale * output_scale)

        cumulative_input_scale *= n_w

        # Round and clamp the quantized_bias to range [-2147483648, 2147483647]
        quantized_bias = (bias * cumulative_input_scale).round()
        return torch.clamp(quantized_bias, min=-2147483648, max=2147483647)
```

Question 6: Grouped Pruning and Quantization (GPTQ)

1. The main innovations of GPTQ that enable efficient quantization of models with hundreds of billions of parameters are Arbitrary Order Insight, Lazy Batch Updates, and Cholesky Reformulation.
2. The reduction from the cubic input dependency runtime of $O(d_{row} \times d^3_{col})$ to $O(\max(d_{row} \times d^2_{col}, d^3_{col}))$ enables GPTQ's approach to be much more scalable than prior methods, especially for large models. Its use of approximate second-order information(shared Hessian across rows) addresses challenges in layer-wise quantization by quantizing weights one-by-one and using the inverse Hessian to calculate how to adjust the remaining unquantized weights to minimize the resulting error. Because the Hessian depends only on layer inputs, it's the same for all rows and GPTQ computes it once per column rather than once per weight. This maintains accuracy while also reducing computation efforts drastically.
3. One limitation of GPTQ discussed in the paper is that it currently doesn't provide speedups for the actual matrix multiplications. One possible way to address this in future work is to create hardware that supports mixed-precision operands on mainstream architectures.