Preetham Rakshith Prakash
pp2959 | pp2959@nyu.edu

HPML Assignment 5 Report

The Output files are as follows :

gpus1.out : Output for using 1 GPUs for batch sizes of 32, 128, 512, 2048 and 8192 / gpu.
gpus2.out : Output for using 2 GPUs for batch sizes of 32, 128, 512, 2048 and 8192 / gpu.
gpus4.out : Output for using 4 GPUs for batch sizes of 32, 128, 512, 2048 and 8192 / gpu.
q4.out : Output for using 4 GPUs with batch size of 512 / gpu (best batch size) for q4.

Q1:

On Model V100 GPU, it takes **23.76 seconds** to train one epoch (w/o data loading) using **mini-batch size 32**, **17.21 seconds** using **batch-size of 128**, **17.89 seconds** using **batch size of 512**, **18.53 seconds** using **batch size of 2048** and **18.24 seconds** using **batch size of 8192**. One common reason why larger batch sizes might take longer to train is due to memory constraints. When we increase the batch size, we are effectively processing more data in parallel during each training step.

This can lead to increased memory usage, which can sometimes exceed the available memory on the device. When this happens, the system needs to resort to slower methods, such as swapping memory with host and device which can significantly slow down the training process.

Q2:

	Batch size 32		Batch size 128		Batch size 512		Batch size 2048		Batch size 8192	
sec	Time	Speedup	Time	Speedup	Time	Speedup	Time	Speedup	Time	Speedup
1-GPU	24.94	1	17.57	1	18.07	1	18.65	1	18.34	1
2-GPU	33.48	0.74	12.55	1.40	10.20	1.77	9.66	1.93	9.68	1.89
4-GPU	28.45	0.87	9.34	1.88	6.66	2.71	6.60	2.82	5.02	3.65

From the above measurements we are looking at strong scaling behaviour. The problem size which is our total size of the dataset remains unchanged as the number of processes increases which is the number of gpus. With increase in the number of gpus with respect to the batch size the total train times per epoch reduces, thus this is a strong scaling behaviour.

In case of weak scaling, we could double the dataset size for 2-GPUs and keep it the same for 1-GPU. In this case, the total train time will be doubled in theory, that is for 2-GPU with batch size of 512/gpu the train time would be 10.20 x 2 = 20.4 seconds. And for 1-GPU with batch size of 512/gpu the train time will remain the same, 18.07 seconds.

Therefore, the speedup will be **0.88** which is **less than the speedup of 1.77** with strong scaling. Similarly with 4-GPUs and batch size of 8192/gpu, the train times can be calculated as 5.02 * 4 = 20.08 seconds. With a speedup of 18.34/20.08 = **0.91**. Thus, we can observe that **speedups with weak scaling would be less than 1 in most cases**

Preetham Rakshith Prakash
pp2959 | pp2959@nyu.edu

based on our data. And thus **strong scaling is better than weak scaling in this instance**.

Q 3.1:

	Batch size 32		Batch size 128		Batch size 512		Batch size 2048		Batch size 8192	
sec	Comp	Comm	Comp	Comm	Comp	Comm	Comp	Comm	Comp	Comm
2-GPU	32.78 s	20.90 s	12.29 s	3.68 s	10.05 s	1.10 s	9.55 s	0.28 s	9.58 s	0.46 s
4-GPU	27.95 s	22.01 s	9.11 s	4.80 s	5.48 s	1.00 s	4.95 s	0.32 s	4.96 s	0.40 s

We can measure training time on one GPU and using this time as a reference we can measure the communication overhead. That is, if t1 is training time on one GPU and t2 using k GPUs,

communication overhead = t2 – (t1 / k)

Q 3.2:

Assuming PyTorch DP implements the all-reduce algorithm. The total data transfer per iteration can be calculated using the formula :

$$\text{Data Transferred} = 2(N-1)\frac{K}{N}$$

Where N is the number of GPUs per iteration and K is the total amount of data being summed across the GPUs during an all-reduce operation for a given iteration.

Here in our ResNet18 implementation the total number of parameters are **11,173,962** floats that are 4 bytes in size, making **K = 44,695,848 bytes**.

Data Transfer per iter for **N = 2** is : **0.0447 GB**
Data Transfer per iter for **N = 4** is : **0.067 GB**

And the total number of GPUs per iterations N = 2 and 4. Therefore, we can calculate the bandwidth as:

GPU-GPU Bandwidth = Data Transfer per iter / compute time per iter

	Batch size 32	Batch size 128	Batch size 512	Batch size 2048	Batch size 8192
GB/sec	Bandwidth	Bandwidth	Bandwidth	Bandwidth	Bandwidth
2-GPU	1.066 GB/s	0.712 GB/s	0.218 GB/s	0.060 GB/s	0.018 GB/s
4-GPU	0.938 GB/s	0.721 GB/s	0.306 GB/s	0.095 GB/s	0.027 GB/s

Preetham Rakshith Prakash
pp2959 | pp2959@nyu.edu

Q 4.1:

The average training loss and training accuracy for the 5th epoch for batch size per gpu **512** on 4 GPUs is **1.344** and **50.944 %**.

And the average training loss and training accuracy for the 5th epoch for batch size **128** from lab2 is **0.686** and **76.308 %**. The convergence is better from lab2 experiments compared to convergence from using 4-GPUs with batch size per gpu of 512.

Q 4.2:

In the paper "Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour" by Goyal et al., two key remedies were proposed to address the challenges associated with training accuracy when using large batch sizes:

Linear Learning Rate Scaling: This remedy involves adjusting the learning rate proportionally to the batch size. When using larger batch sizes, the gradients become noisier due to the decreased variance caused by fewer samples. Consequently, increasing the learning rate can help offset this effect. However, simply increasing the learning rate without scaling it appropriately can lead to instability during training. The paper suggests a linear scaling rule, where the learning rate is multiplied by a factor proportional to the square root of the batch size. This adjustment helps to maintain stability while training with larger batch sizes and contributes to improved accuracy.

Warmup Training: Warmup training involves gradually increasing the learning rate from a very small value to the desired learning rate over a certain number of iterations at the beginning of training. This approach allows the model to stabilize and adapt to the large batch size before reaching higher learning rates, which helps mitigate the risk of divergence or poor convergence caused by large initial learning rates. By gradually increasing the learning rate during the warmup phase, the model can effectively navigate the optimization landscape and achieve better training accuracy, especially when using large batch sizes.

Q5:

The epoch ID is used to determine which portion of the dataset each process or node will handle during a particular epoch. Each process may be responsible for a different subset of the data, and the epoch ID helps in ensuring that all processes iterate through the entire dataset without redundancy.

In DDP, because each process operates independently and may have its own data loader, it's essential to synchronize the starting point of each epoch across all processes. This synchronization ensures that all processes are working on the same epoch and processing unique samples without overlap.

On the other hand, in Data Parallel (DP) training, which typically involves synchronous training across multiple GPUs within a single node, there's no need for such explicit epoch ID setup because all GPUs share the same memory space and can access the data loader

directly. Hence, the management of epochs is handled more transparently without the need for explicit setup.

Q6:

No, the local statistics that are calculated in batch norm layers are communicated across gpus to compute global statistics for normalization. Thus, while gradients are still the primary message communicated between gpus during all-reduce, batch normalization layers can introduce additional communication for synchronizing statistics across gpus to ensure consistent normalization and stable training.

Q7:

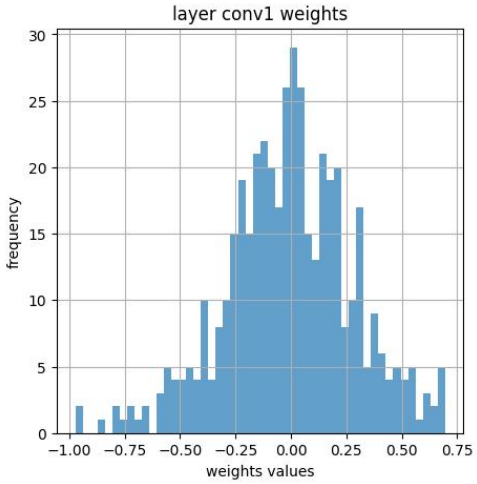
No, because of the batch norm layers present in the models across all 4 GPUs and considering the large batch size of 512 per GPU, It is required to communicate local statistics along with gradients among the GPUs during all-reduce in order to synchronize these statistics for consistent normalization.

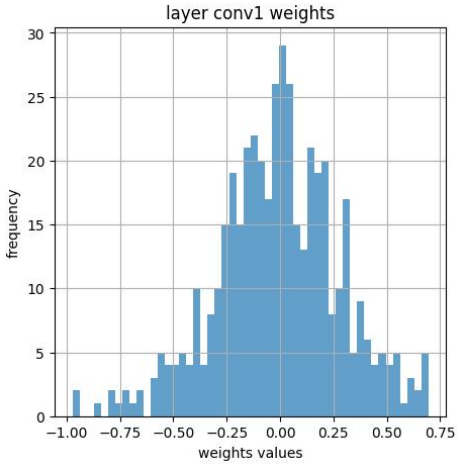
Preetham Rakshith Prakash
pp2959 | pp2959@nyu.edu

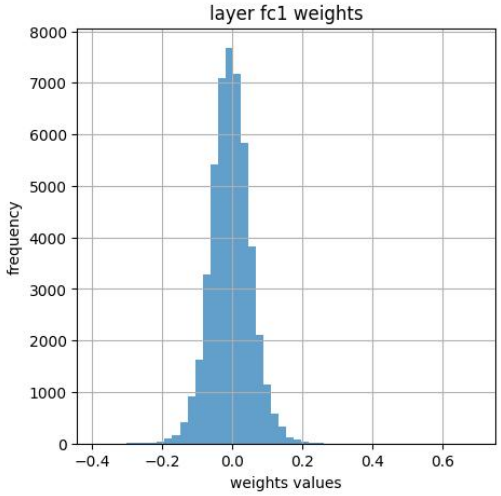
PART B

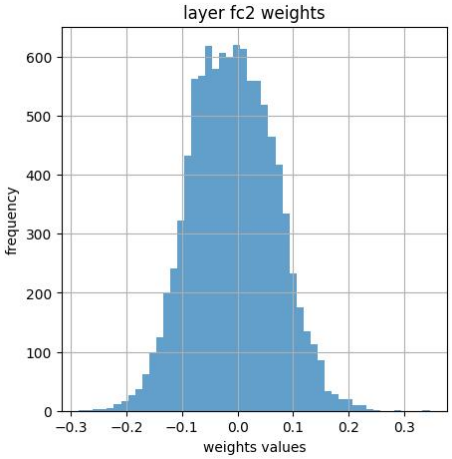
Notebook Link : https://colab.research.google.com/drive/1dmdPLToyq1NIQTB7G-xcM5VGMDTcljzl?usp=sharing

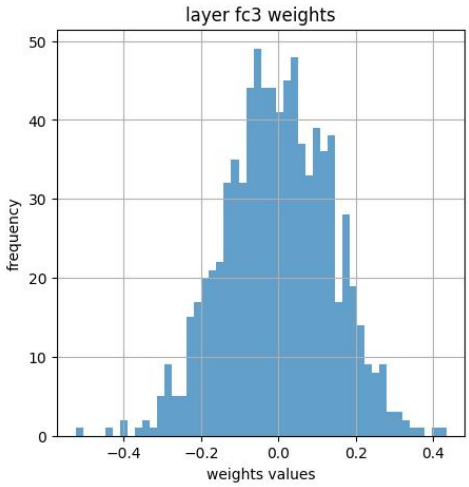
Q1:











Preetham Rakshith Prakash
pp2959 | pp2959@nyu.edu

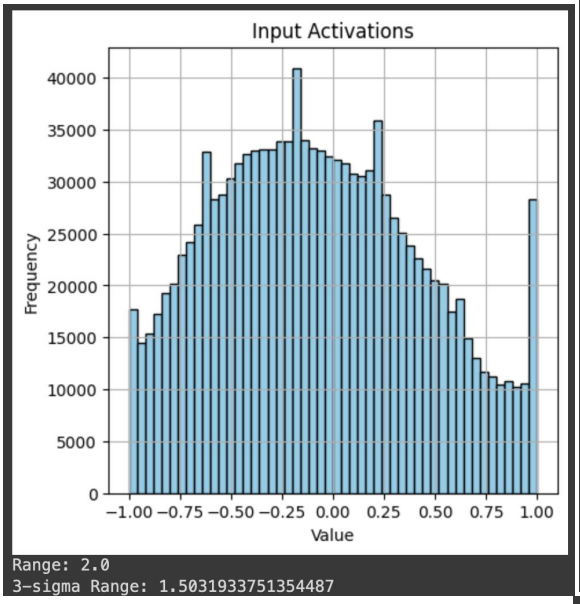
Q2:

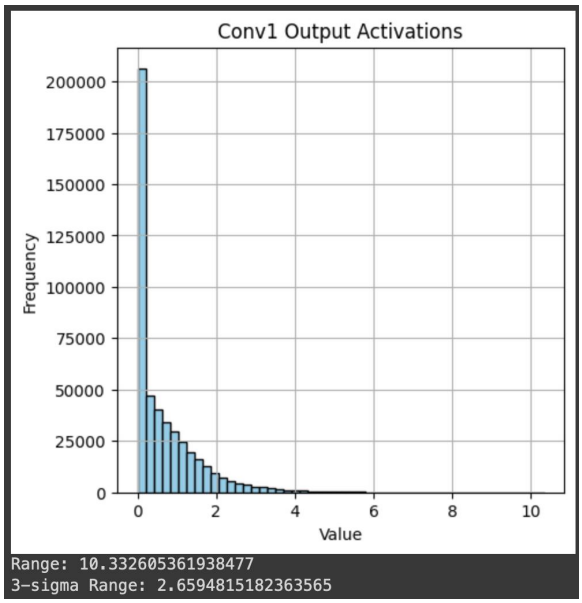
```
max_val = torch.max(torch.abs(weights))
scale_factor = 127 / max_val.item()  # Scale factor to map max_val to 127
quantized_weights = (weights * scale_factor).round().clamp(-128, 127)

return quantized_weights, scale_factor
```

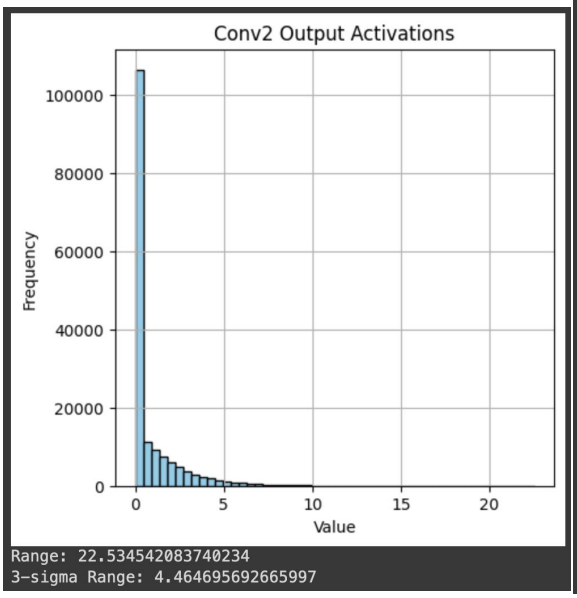
```
score = test(net_q2, testloader)
print('Accuracy of the network after quantizing all weights: {}%'.format(score))
```

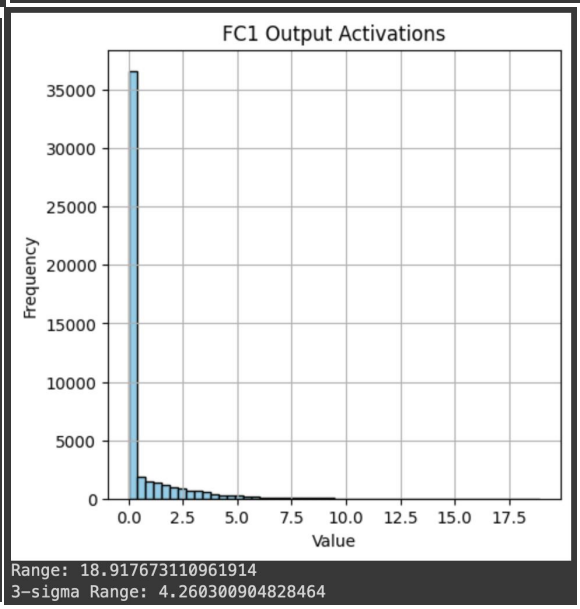
```
Accuracy of the network after quantizing all weights: 58.62%
```

Q3:

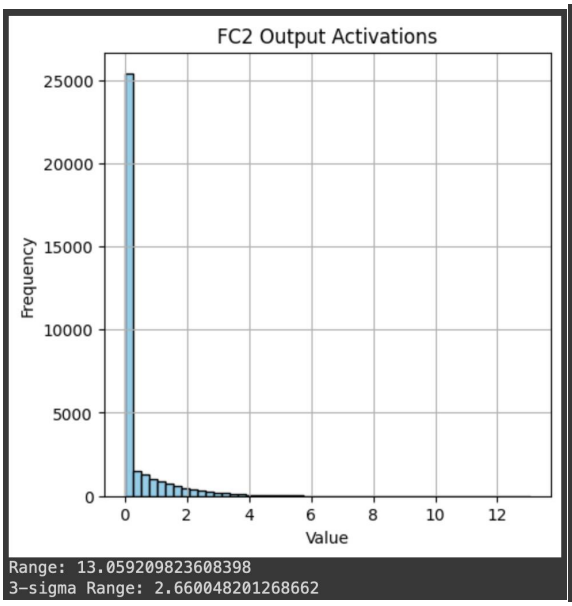




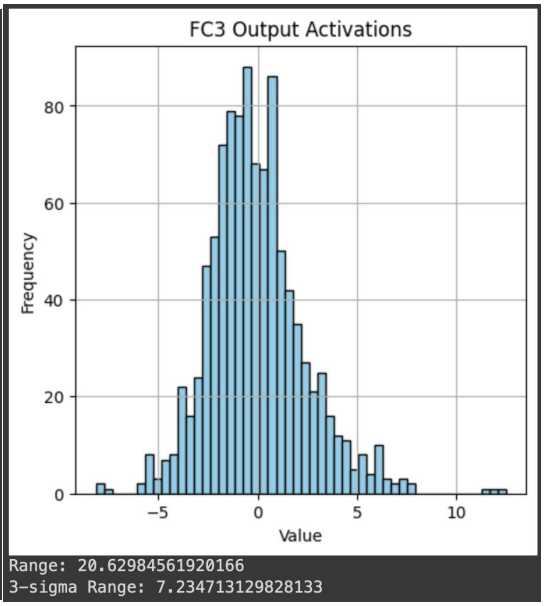




Preetham Rakshith Prakash
pp2959 | pp2959@nyu.edu



Range: 13.059209823608398
3-sigma Range: 2.660048201268662



Range: 20.62984561920166
3-sigma Range: 7.234713129828133

Q4:

Quantize initial inputs :

```
@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.
    '''
    max_abs = np.max(np.abs(pixels))
    scale = 127 / max_abs

    return scale
```

Quantize the activation layers :

```
# Initialize scaling factor
scaling_factor = 1
epsilon = 1e-10

# Apply scaling factors of preceding layers
for weight_scale, output_scale in ns:
    scaling_factor *= weight_scale * output_scale

# Adjust scaling factor with current layer's scaling
scaling_factor *= n_w * n_initial_input

# Normalize activations and calculate maximum value
max_activation = np.max(np.abs(activations))
# max_activation = max(epsilon, max_activation)

# Adjust scaling factor based on the maximum activation
scaling_factor /= max_activation
scaling_factor = 127/scaling_factor

return scaling_factor
```

Preetham Rakshith Prakash
pp2959 | pp2959@nyu.edu

Forward pass logic using the calculated activation scales :

```
def forward(self, x: torch.Tensor) -> torch.Tensor:

    x = (x * self.input_scale).round().clamp(-128, 127)

    # Pass through conv1
    x = F.relu(self.conv1(x))
    x = (x * self.conv1.output_scale).round().clamp(0, 127)

    x = self.pool(x)

    # Pass through conv2
    x = F.relu(self.conv2(x))
    x = (x * self.conv2.output_scale).round().clamp(0, 127)

    x = self.pool(x)

    x = x.view(-1, 16 * 5 * 5)

    # Pass through fc1
    x = F.relu(self.fc1(x))
    x = (x * self.fc1.output_scale).round().clamp(0, 127)

    # Pass through fc2
    x = F.relu(self.fc2(x))
    x = (x * self.fc2.output_scale).round().clamp(0, 127)

    # Pass through fc3
    x = self.fc3(x)
    x = (x * self.fc3.output_scale).round().clamp(-128, 127)

    return x
```

Accuracy after quantizing activations as well :

```
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: 35.61%
```

Preetham Rakshith Prakash
pp2959 | pp2959@nyu.edu

Q 5:

Logic for quantizing bias :

```
preceding_scale_product = 1.0
for w_scale, o_scale in ns:
    preceding_scale_product *= w_scale * o_scale

# Calculate the scaling factor
scaling_factor = n_w * n_initial_input * preceding_scale_product

scaling_factor = 2147483647 / scaling_factor

# Quantize the bias tensor
quantized_bias = (bias * scaling_factor).round().clamp(-2147483648, 2147483647)

return quantized_bias
```

Accuracy after quantizing weights and bias :

```
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: {}%
```

```
Accuracy of the network on the test images after all the weights and the bias are quantized: 33.53%
```