High Performance Machine Learning
Lab 5

Ankit Rajvanshi
ar7996

Q1. Batch size vs Training Time on 1 GPU

Batch Size: 32
Training Time: 23.013953601941466 secs
Batch Size: 128
Training Time: 17.198095166124403 secs
Batch Size: 512
Training Time: 18.245301387272775 secs
Batch Size: 2048
Training Time: 18.884235872421414 secs
Batch Size: 8192
Training Time: 18.082406002562493 secs

With larger batch size, training takes less time because of less data loading overhead. Initially, with increasing batch size, training time reduce significantly due to increase in utilization of available bandwidth to load data (CPU to GPU). Afterwards there is not much significant change in training time (due to change from bandwidth limited to throughput limited).

Q2.

		1-GPU	2-GPU	4-GPU
Batch-size 32 per GPU	**Training Time(sec)**	23.0139536	32.71642571	18.88489087
	Running Time (sec)	24.22828047	33.5876941	19.55294256
	Speedup	1	0.721343966	1.23911173
Batch-size 128 per GPU	**Training Time(sec)**	17.19809517	12.25876194	6.788041316
	Running Time (sec)	17.7556792	13.01399675	13.07991362
	Speedup	1	1.364352516	1.357476794
Batch-size 512 per GPU	**Training Time(sec)**	18.24530139	9.796424931	5.060331174
	Running Time (sec)	18.74045668	13.20317178	13.30563801
	Speedup	1	1.419390507	1.408459832
Batch-size 2048 per GPU	**Training Time(sec)**	18.88423587	9.676747838	4.985887006
	Running Time (sec)	20.1137135	13.94979644	14.82708593
	Speedup	1	1.441864302	1.356552029
Batch-size 8192 per GPU	**Training Time(sec)**	18.082406	9.653193856	4.896082186
	Running Time (sec)	22.55010017	20.73705432	22.74536428
	Speedup	1	1.087430251	0.991415213

Table 1: Training Time, Running Time and SpeedUp for various batch sizes on 1,2,4 GPUs

In above table, while measuring running time per epoch we can see that with increasing number of GPUs, time required to copy the load the data from memory remains same in all the cases and it is a huge factor and hence, it is a weak scaling.

Whereas, if we see training time per epoch with increasing number of GPUs, we can observe significant decrease in training time and hence better speedup and thus strong scaling. This is because total data which each GPU must process decreases with increase in number of GPUs.

Q3.1 How much time spent in computation and communication:

Using the result of 1 GPU, we can find the computation time per image and per batch.
Since, all GPUs are of same configuration, computation time of each GPU per image/batch will be same.

Hence,
$T_{\text{Compute, N}} = T_{\text{Compute, Batch, 1}} * (\#\text{Batch} - 1) + T_{\text{Compute, Image, 1}} * (\#\text{images in last batch})$

OR we can simply write,

$T_{\text{Compute, N}} = T_{\text{Compute, Image, 1}} * (\#\text{images per GPU})$

$T_{\text{Communication, N}} = (T_{\text{Overall, N}} - T_{\text{Compute, N}})$

		2-GPU	4-GPU
Batch-size 32 per GPU	**Compute (sec)**	11.5069768	5.7534884
	Comm (sec)	21.20944891	13.13140246
Batch-size 128 per GPU	**Compute (sec)**	8.599047583	4.299523792
	Comm (sec)	3.659714357	2.488517525
Batch-size 512 per GPU	**Compute (sec)**	9.122650694	4.561325347
	Comm (sec)	0.673774237	0.499005827
Batch-size 2048 per GPU	**Compute (sec)**	9.442117936	4.721058968
	Comm (sec)	0.234629901	0.264828038
Batch-size 8192 per GPU	**Compute (sec)**	9.041203001	4.520601501
	Comm (sec)	0.611990854	0.375480685

Table 2: Compute and Communication Time for 2,4 GPUs for various batch sizes

Q3.2 Communication bandwidth utilization

Assuming PyTorch DP implements all-reduce algorithm, communication time overhead calculated above will itself become the time required for all-reduce algorithm as other than computation (including CPU to GPU), only all-reduce will be executed by PyTorch DP in that case.

To calculate Bandwidth utilization, we need to calculate how much data was transferred during communication phase in one epoch.

Each device will send N(P-1)/P amount of data in scatter-reduce phase and N(P-1)/P amount data in AllGather phase and hence, total data sent by each device will be equal to 2N(P-1)/P in one iteration.

Total data transferred among P devices will be equal to 2N(P-1) in one iteration.

Hence,
Total Data = *2N(P-1) * #Iterations * 4* bytes
= (*2N(P-1) * #Iterations * 4) / 1000000000* GB

#Iterations => Number of batches per GPU

Bandwidth Utilization = Total Data transferred during All-Reduce / All-Reduce Time

		2-GPU	**4-GPU**
Batch-size-per-GPU 32	**Bandwidth Utilization (GB/s)**	3.295903942	7.985168354
Batch-size-per-GPU 128	**Bandwidth Utilization (GB/s)**	4.787469924	10.56096988
Batch-size-per-GPU 512	**Bandwidth Utilization (GB/s)**	6.50098039	13.43546876
Batch-size-per-GPU 2048	**Bandwidth Utilization (GB/s)**	4.952872764	7.088470043
Batch-size-per-GPU 8192	**Bandwidth Utilization (GB/s)**	0.584268182	1.428436128

Table 3: Communication Bandwidth utilization

Q4.1 Accuracy when using large batch:

```
Optimizer: sgd, num_workers: 2, Device: cuda
Number of Devices: 1, Batch Size per GPU: 128
Number of Batches: 391

Epoch: 5/5 Training loss: 0.8889151154576665  Training acc: 0.6860174233346339
Training time: 17.187113458756357 secs, Running time: 17.76101703196764 secs


Optimizer: sgd, num_workers: 2, Device: cuda
Number of Devices: 4, Batch Size per GPU: 8192
Number of Batches: 2

Epoch: 5/5, Training loss: 4.432891249656677, Training acc: 0.10917582735419273
Training time: 4.738258120138198 secs, Running time: 22.63015639036894 secs
```

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

Remedy 1: If batch size is large, we can use hyper parameter free linear scaling rule for learning rate i.e. When the minibatch size is multiplied by k, multiply the learning rate by k. But this strategy fails during initial stages of training when networks change rapidly.

Remedy 2: To address the above issue, we can use warmup strategy where low constant learning rate is used during initial stages (up to 5 epochs) and afterwards, it can use above rule.

Q5 Distributed Data Parallel

In case of distributed mode, set_epoch() must be called i.e epoch_ID must be set to data loader before creating iterator to ensure that shuffling works properly across all the epochs else same order will be used in all epochs while in non-distributed mode, shuffling occurs automatically at the start of new epoch.
Automatic shuffling is disabled in distributed mode to maintain shuffling consistency across all devices.

Q6 What are passed on network?

No, other than gradients, BN statistics are also communicated among learners optionally. BN statistics are synchronized to maintain consistent normalization across the distributed model.

Q7 What if we only communicate gradients?

Yes, it will be sufficient if we only communicate gradients across 4 GPUs. As per "Accurate, large minibatch SGD: training ImageNet in 1 hour,", BN statistics should be computed locally not across the workers so that the underlying loss function remain optimized and moreover it also reduces extra overhead.

Part – B

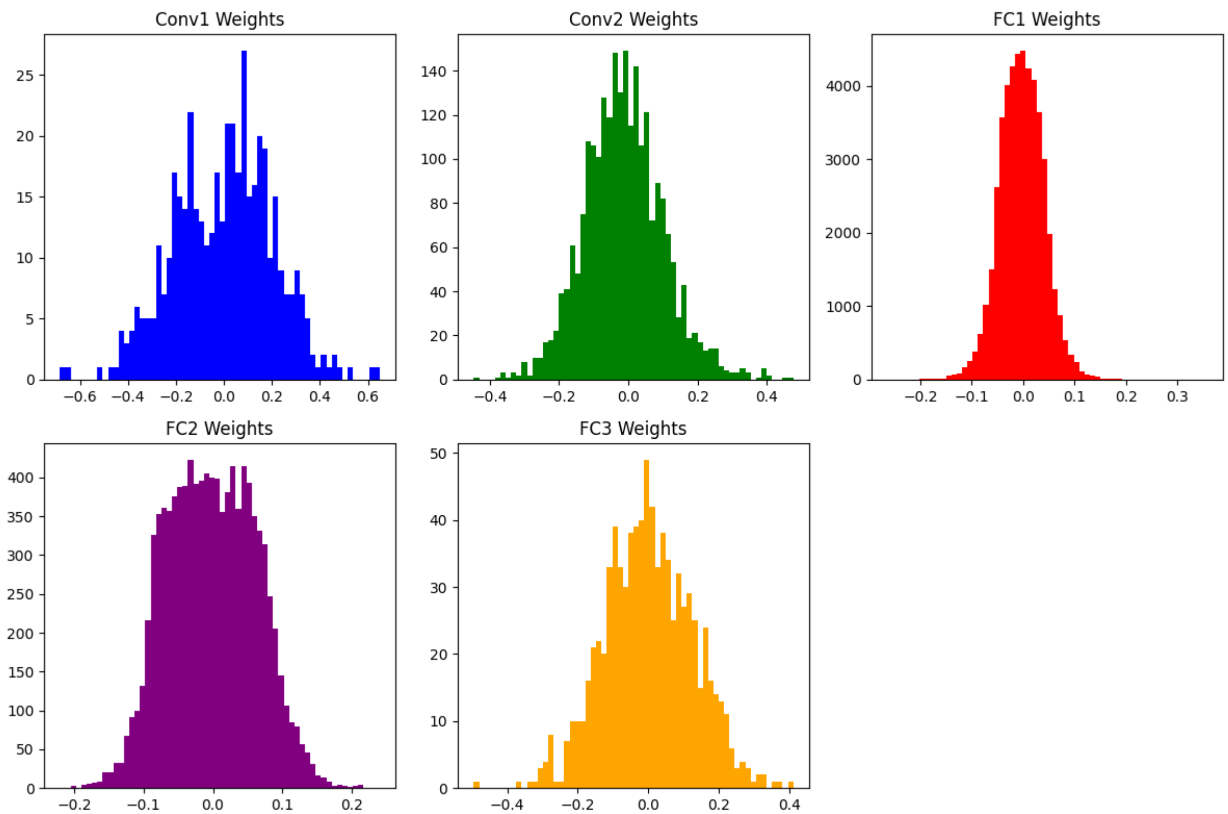
Q1 Visualize Weights

Code:

```
conv1_weights = net.conv1.weight.data.cpu().view(-1)
conv2_weights = net.conv2.weight.data.cpu().view(-1)
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)

plt.figure(figsize=(12, 8))

plt.subplot(2, 3, 1)
plt.hist(conv1_weights, bins=60, color='blue')
plt.title('Conv1 Weights')

plt.subplot(2, 3, 2)
plt.hist(conv2_weights, bins=60, color='green')
plt.title('Conv2 Weights')

plt.subplot(2, 3, 3)
plt.hist(fc1_weights, bins=60, color='red')
plt.title('FC1 Weights')

plt.subplot(2, 3, 4)
```

```
plt.hist(fc2_weights, bins=60, color='purple')
plt.title('FC2 Weights')

plt.subplot(2, 3, 5)
plt.hist(fc3_weights, bins=60, color='orange')
plt.title('FC3 Weights')

plt.tight_layout()
plt.savefig("q1.png")
plt.show()
```

Plots:



Q2 Quantize Weights

Code:

```
max_x = torch.max(weights)
min_x = torch.min(weights)
step_size = (max_x - min_x) / (pow(2, 8-1) - 1)

scale = 1 / step_size
result = (weights * scale).round()
return torch.clamp(result, min=-128, max=127), scale
```

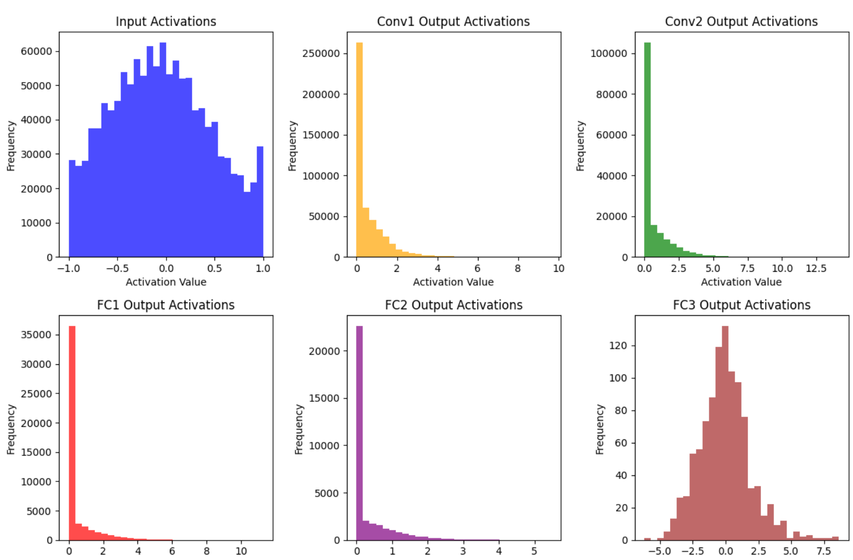
Q3 Visualize Activations

Code:

```
# Plot histograms
plt.figure(figsize=(12, 8))

# Input activations
plt.subplot(2, 3, 1)
plt.hist(input_activations, bins=30, color='blue', alpha=0.7)
plt.title('Input Activations')
plt.xlabel('Activation Value')
plt.ylabel('Frequency')

# Conv1 output activations
plt.subplot(2, 3, 2)
plt.hist(conv1_output_activations, bins=30, color='orange', alpha=0.7)
plt.title('Conv1 Output Activations')
plt.xlabel('Activation Value')
plt.ylabel('Frequency')

# Conv2 output activations
plt.subplot(2, 3, 3)
plt.hist(conv2_output_activations, bins=30, color='green', alpha=0.7)
plt.title('Conv2 Output Activations')
plt.xlabel('Activation Value')
plt.ylabel('Frequency')

# FC1 output activations
plt.subplot(2, 3, 4)
plt.hist(fc1_output_activations, bins=30, color='red', alpha=0.7)
plt.title('FC1 Output Activations')
plt.xlabel('Activation Value')
plt.ylabel('Frequency')

# FC2 output activations
plt.subplot(2, 3, 5)
plt.hist(fc2_output_activations, bins=30, color='purple', alpha=0.7)
plt.title('FC2 Output Activations')
plt.xlabel('Activation Value')
plt.ylabel('Frequency')

# FC3 output activations
plt.subplot(2, 3, 6)
plt.hist(fc3_output_activations, bins=30, color='brown', alpha=0.7)
plt.title('FC3 Output Activations')
plt.xlabel('Activation Value')
plt.ylabel('Frequency')
```

```
plt.tight_layout()
plt.savefig("q3.png")
plt.show()

# Calculate ranges and 3-sigma ranges
variables = {
    '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
}

for name, data in variables.items():
    min_val = np.min(data)
    max_val = np.max(data)
    range_val = max_val - min_val

    mean = np.mean(data)
    std_dev = np.std(data)
    three_sigma = 3 * std_dev
    lower_limit = mean - three_sigma
    upper_limit = mean + three_sigma

    print("Layer: {}".format(name))
    print("Total Range:: Min: {:.4f}, Max: {:.4f}".format(min_val,
max_val))
    print("3 Sigma Range:: Lower Limit: {:.4f}, Upper Limit:
{:.4f}\n".format(lower_limit, upper_limit))
```

Plots:



Output:

```
Layer: Input Activations
Total Range:: Min: -1.0000, Max: 1.0000
3 Sigma Range:: Lower Limit: -1.5618, Upper Limit: 1.4560

Layer: Conv1 Output Activations
Total Range:: Min: 0.0000, Max: 9.6383
3 Sigma Range:: Lower Limit: -1.9290, Upper Limit: 3.0603

Layer: Conv2 Output Activations
Total Range:: Min: 0.0000, Max: 14.1339
3 Sigma Range:: Lower Limit: -2.7353, Upper Limit: 4.0610

Layer: FC1 Output Activations
Total Range:: Min: 0.0000, Max: 11.2787
3 Sigma Range:: Lower Limit: -2.3141, Upper Limit: 3.1560

Layer: FC2 Output Activations
Total Range:: Min: 0.0000, Max: 5.4686
3 Sigma Range:: Lower Limit: -1.5531, Upper Limit: 2.2252

Layer: FC3 Output Activations
Total Range:: Min: -6.1661, Max: 8.6315
3 Sigma Range:: Lower Limit: -5.8402, Upper Limit: 5.7733
```

Q4 Quantize Activations

Code:

```
@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_x = np.max(pixels)
        min_x = np.min(pixels)
        step_size = (max_x - min_x) / (pow(2, 8-1) - 1)
```

```
        scale = 1 / step_size
        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.
        '''
        l = len(ns)

        #val = 1/S1 * 1/S2
        if l == 0:
          val = n_w.item() * n_initial_input

        else:
          val = n_w.item() * ns[-1][1]

        max_x = np.max(activations)
        min_x = np.min(activations)

        #M = S1*S2 / S3
        #scale = 1/(val*S3)
        step_size = (max_x - min_x) / (pow(2, 8-1) - 1) #S3
        scale = 1/(val*step_size)
        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

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

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

        x = self.pool(F.relu(self.conv2(x)))
        x = (x * self.conv2.output_scale).round()
        x = torch.clamp(x, min=-128, max=127)

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

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

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

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

        return x
```

Q5 Quantize Bias

```
@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.
        '''

        # S_bias = S1 * S2
        scale = n_w.item() * ns[-1][1]

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