Q1

1.1

True. They are methods that use fundamentally different principles. Weight pruning is to remove unimportant weights while weight quantization is to reduce the number of bits that represent a number.

1.2

False. It is weight sharing (quantization) that takes advantage of look-up table (LUT) to improve the efficency and pruning cannot directly benefit the following Huffman coding process.

1.3

False. It depends. For instance, iterative connection pruning will lead to non-structured sparse weight matrices. It that may not bring much speedup on traditional platforms like GPU, even with specifically designed sparse matrix multiplication algorithm.

1.4

True. Let's see the number of bits we need to represnet all N values is n and we can have $2^n = N$. Then we can get $n = log(N)$. So the expected average length should be $O(log(N))$

1.5

True. We use k bits to represent each weight. And there are N weights so we need Nk bits if we ignore the storage for the centroids.

Q2 Lab 1

(a)

Full-precision model training

[2]:
```
net = ResNetCIFAR(num_layers=20)
net = net.to(device)

# Uncomment to load pretrained weights
# net.load_state_dict(torch.load("net_before_pruning.pt"))
# Comment if you have loaded pretrained weights
train(net, epochs=100, batch_size=128, lr= 0.01, reg=1e-3)
```

[3]:
```
# Load the best weight paramters
net.load_state_dict(torch.load("net_before_pruning.pt"))
test(net)
```

```
Files already downloaded and verified
Test Loss=0.3028, Test accuracy=0.9101
```

(b)

I think Std works better than fixed percentage.Becasethe weight are distributed like Gaussian, a fixed ratio to std also infers a steady percentage. And pruning a fixed percentage of weights in each layer may not be effcient as it need a “ranking” process of all the weight parameters

(c)

I show a part of std implementation. Please refer to code file directly for more details.

```
    def prune_by_std(self, s=0.25):
        """
        Pruning by a factor of the standard deviation value.
        :param std: (scalar) factor of the standard deviation value.
        Weight magnitude below std(weight)*s will be pruned.
        """
        np_weight = self.linear.weight.data.cpu().numpy()
        flattened_weights = np_weight.flatten()
        # Generate the pruning threshold according to 'prune by std'. (Your code: 1 Line)
        thresh = np.std(flattened_weights)*s
        # Generate a mask to determine which weights should be pruned (Your code: <=3 Lines)
        self.mask = np.abs(np_weight) > thresh
        self.mask = np.float32(self.mask)
        # Multiply weight by mask (Your code: 1 Line)
        np_weight = np_weight = np.multiply(np_weight, self.mask)
        # Copy back to linear.weight and assign to device (Your code: 1 Line)
        self.linear.weight.data = torch.from_numpy(np_weight).float().to(device)
        # Compute sparsity (Your code: 1 Line)
        self.sparsity = float((np_weight ==0).sum())/len(flattened_weights)
        # Copy mask to device for faster computation (Your code: 1 Line)
        self.mask = torch.from_numpy(self.mask).float().to(device)
        pass
```

(d)

What do you observe from the sparsity pattern for each layer? E.g., is the sparsity structured or unstructured? Which layers are more likely to be sparse? etc. Give explanations to support your findings.

The sparsity is unstructured as we can see all layers are pruned with the smimilar sparsity as the screenshot shows when s = 0.5.

s	0.0	0.25	0.5	1.0
accuracy	0.9169	0.9134	0.8735	0.4453
sparsity	0.000000	0.237423	0.436723	0.724295

[26]:
```
# Test accuracy before fine-tuning 'percentage'
net.load_state_dict(torch.load("net_before_pruning.pt"))
prune(net, method='std', q=0, s=0.5)
test(net)
```

```
Files already downloaded and verified
Test Loss=0.4016, Test accuracy=0.8735
```

[27]:
```
summary(net)
```

```
Layer id        Type            Parameter       Non-zero parameter      Sparsity(\%)
1               Convolutional   864             374                     0.567130
2               BatchNorm       N/A             N/A                     N/A
3               ReLU            N/A             N/A                     N/A
4               Convolutional   4608            1829                    0.603082
5               BatchNorm       N/A             N/A                     N/A
6               ReLU            N/A             N/A                     N/A
7               Convolutional   2304            1181                    0.487413
8               BatchNorm       N/A             N/A                     N/A
9               Convolutional   512             240                     0.531250
10              BatchNorm       N/A             N/A                     N/A
11              ReLU            N/A             N/A                     N/A
12              Convolutional   2304            1199                    0.479601
13              BatchNorm       N/A             N/A                     N/A
14              ReLU            N/A             N/A                     N/A
15              Convolutional   2304            1138                    0.506076
16              BatchNorm       N/A             N/A                     N/A
17              ReLU            N/A             N/A                     N/A
18              Convolutional   2304            1176                    0.489583
19              BatchNorm       N/A             N/A                     N/A
20              ReLU            N/A             N/A                     N/A
21              Convolutional   2304            1179                    0.488281
```

(e)

I show a part of percentage implementation. Please refer to code file directly for more details.

```
def prune_by_percentage(self, q=5.0):
    """
    Pruning the weight paramters by threshold.
    :param q: pruning percentile. 'q' percent of the least
    significant weight parameters will be pruned.
    """
    np_weight = self.linear.weight.data.cpu().numpy()
    flattened_weights = np.abs(np_weight.flatten())
    # Generate the pruning threshold according to 'prune by percentage'. (Your code: 1 Line)
    thresh =   np.percentile(flattened_weights, q)
    # Generate a mask to determine which weights should be pruned (Your code: <=3 Lines)
    self.mask = np.abs(np_weight) > thresh
    self.mask = np.float32(self.mask)
    # Multiply weight by mask (Your code: 1 Line)
    np_weight = np.multiply(np_weight, self.mask)
    # Copy back to linear.weight and assign to device (Your code: 1 Line)
    self.linear.weight.data = torch.from_numpy(np_weight).float().to(device)
    # Compute sparsity (Your code: 1 Line)
    self.sparsity = float((np_weight ==0).sum())/len(flattened_weights)
    # Copy mask to device for faster computation [Your code: 1 Line]
    self.mask = torch.from_numpy(self.mask).float().to(device)
```

(f)

What do you observe from the sparsity pattern for each layer? E.g., is the sparsity structured or unstructured? Which layers are more likely to be sparse? etc. Give explanations to support your findings.

The sparsity is unstructured as we can see all layers are pruned with the smimilar sparsity as the screenshot shows when q = 50 %.

q	0	25	50	75
accuracy	0.9169	0.9134	0.8371	0.2538
sparsity	0.000084	0.250000	0.50	0.7500

[30]:
```
# Test accuracy before fine-tuning 'percentage'
net.load_state_dict(torch.load("net_before_pruning.pt"))
prune(net, method='percentage', q=50, s=0.0)
test(net)
```

```
Files already downloaded and verified
Test Loss=0.5168, Test accuracy=0.8371
```

[31]:
```
summary(net)
```

Layer id	Type	Parameter	Non-zero parameter	Sparsity(\%)
1	Convolutional	864	432	0.500000
2	BatchNorm	N/A	N/A	N/A
3	ReLU	N/A	N/A	N/A
4	Convolutional	4608	2304	0.500000
5	BatchNorm	N/A	N/A	N/A
6	ReLU	N/A	N/A	N/A
7	Convolutional	2304	1152	0.500000
8	BatchNorm	N/A	N/A	N/A
9	Convolutional	512	256	0.500000
10	BatchNorm	N/A	N/A	N/A
11	ReLU	N/A	N/A	N/A
12	Convolutional	2304	1152	0.500000
13	BatchNorm	N/A	N/A	N/A
14	ReLU	N/A	N/A	N/A
15	Convolutional	2304	1152	0.500000
16	BatchNorm	N/A	N/A	N/A
17	ReLU	N/A	N/A	N/A
18	Convolutional	2304	1152	0.500000
19	BatchNorm	N/A	N/A	N/A
20	ReLU	N/A	N/A	N/A

Q3 Lab2

(a)

We need to use the mask to make sure the weights pruned are always zeros. Compared to the regular training pipelines, I added the for loop to do multiplication between gradient of eay layers with their mask and delete the scheduler.

```
            loss.backward()

            for name, model in net.named_modules():
                if isinstance(model, PrunedConv):
                    model.conv.weight.grad.data.mul_(model.mask)
                elif isinstance(model, PruneLinear):
                    model.linear.weight.grad.data.mul_(model.mask)
                else:
                    pass


            optimizer.step()
```

(b)

```
            loss.backward()

            for name, model in net.named_modules():
                if isinstance(model, PrunedConv):
                    model.conv.weight.grad.data.mul_(model.mask)
                elif isinstance(model, PruneLinear):
                    model.linear.weight.grad.data.mul_(model.mask)
                else:
                    pass


            optimizer.step()
```

(c)

For std pruning, We can see pruning and retraining work very well from the table below. Even for s = 0.5, we recover the test accuracy over 90%. I set learning rate very small because we got close to optimum already and larger learning rate will get us away from the optimum. Besides, we can notice that some pruning can further improve the performance becuase some redudant weights are removed and the model become simpler and powerful. But we cannot remove too many weights, for s = 1.0, the performance of it is below the accuracy without pruning. we need to do tests to find the most suitable s.

s	0.0	0.25	0.5	1.0
retraining settings	lr = 0.0001, reg = 1e-2	lr=0.0001, reg=1e-2	lr=0.0001, reg=1e-2	lr=0.0001, reg=1e-2
accuracy before retraining	0.9169	0.9134	0.8735	0.4453
accuracy after retraining	0.9165	0.9172	0.9175	0.9091
sparsity before retraining	0.000000	0.237423	0.436723	0.724295
sparsity after retraining	0.000000	0.237423	0.436723	0.724295

(d)

For percentage pruning, We can see pruning and retraining work very well from the table below. Even for s = 50, we recover the test accuracy over 91%. I set learning rate very small because we got close to optimum already and larger learning rate will get us away from the optimum. Besides, we can notice that some pruning can further improve the performance becuase some redudant weights are removed and the model become simpler and powerful. But we cannot remove too many weights, for s = 75, the performance of it is below than the accuracy without pruning. we need to do tests to find the most suitable s.

q	0.0	25	50	75
retraining settings	lr = 0.0001, reg = 1e-3	lr = 0.0001, reg = 1e-3	lr=0.0001, reg=1e-2	lr=0.0001, reg=1e-2
accuracy before retraining	0.9169	0.9134	0.8735	0.2538
accuracy after retraining	0.9182	0.9183	0.9167	0.9055
sparsity before retraining	0.000084	0.250000	0.500000	0.750000
sparsity after retraining	0.000084	0.250000	0.500000	0.750000

Q4 Lab3

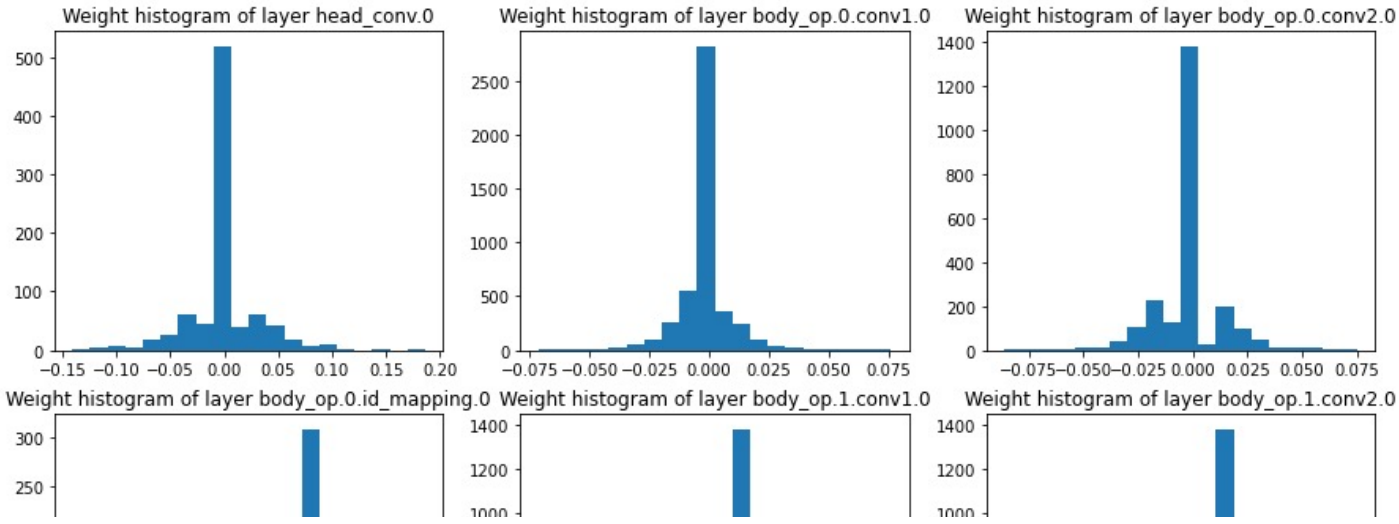
(a)

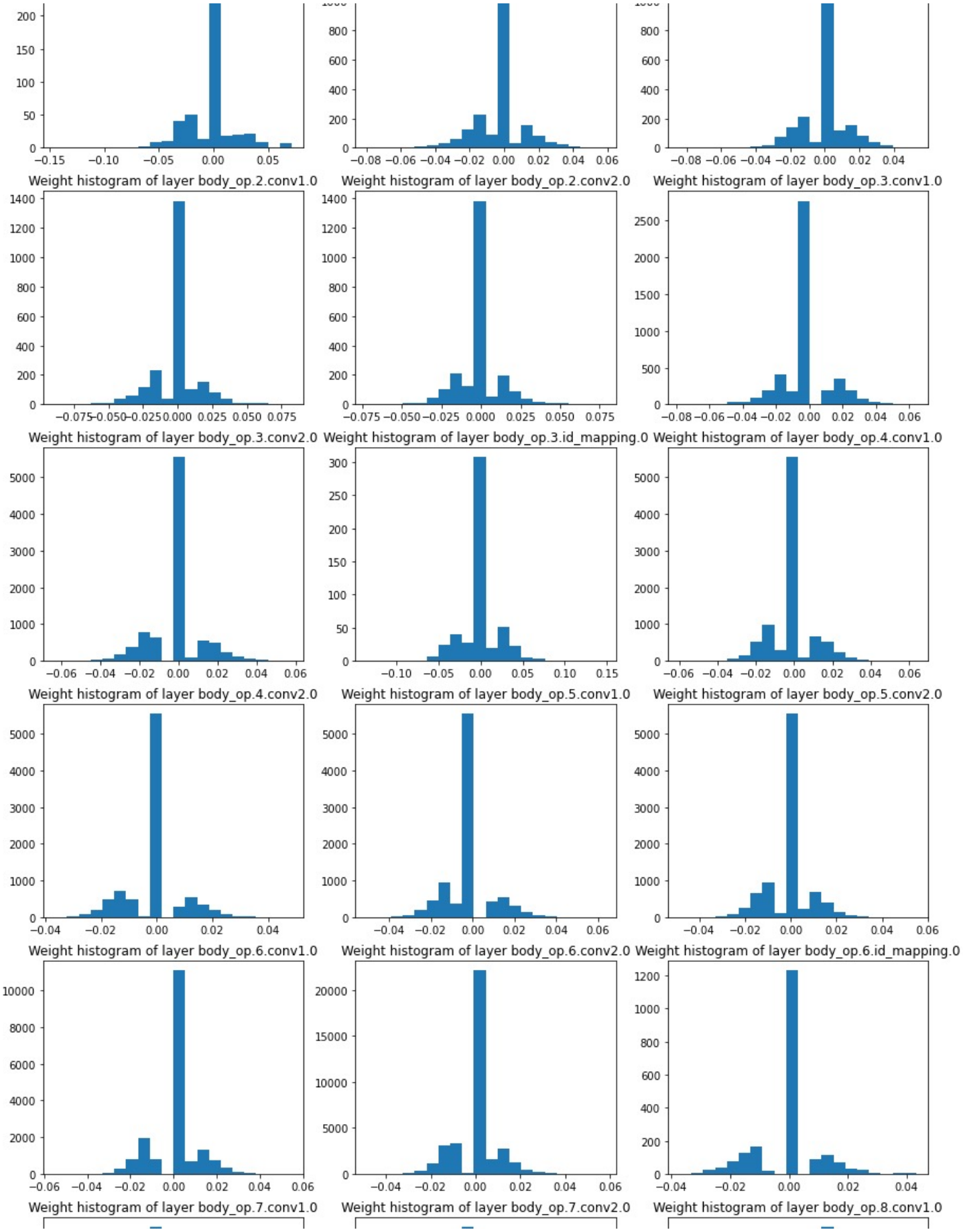
```
def _quantize_layer(weight, bits=8):
    """
    :param weight: A numpy array of any shape.
    :param bits: quantization bits for weight sharing.
    :return quantized weights and centriods.
    """
    # Your code: Implement the quantization (weight sharing) here. Store
    # the quantized weights into 'new_weight' and store kmeans centers into 'centers_'
    flatten_weights = weight.reshape(-1,1)
    index_nonzero = np.nonzero(flatten_weights!=0)[0]
    weights_nonzero = flatten_weights[index_nonzero] # Avoid the cluster with label 0
    kmeans = KMeans(n_clusters=2**bits, random_state=0).fit(weights_nonzero)
    new_weight = flatten_weights.copy()
    centers_ = kmeans.cluster_centers_

    if 0 in centers_: # Avoid the cluster with label 0
        centers_[centers_==0] = [1e-8]*((centers_== 0).sum())

    new_weight[index_nonzero] = np.array([centers_[i][0] for i in
kmeans.labels_]).reshape(len(index_nonzero),-1)
    new_weight = new_weight.reshape(weight.shape)

    return new_weight, centers_
```

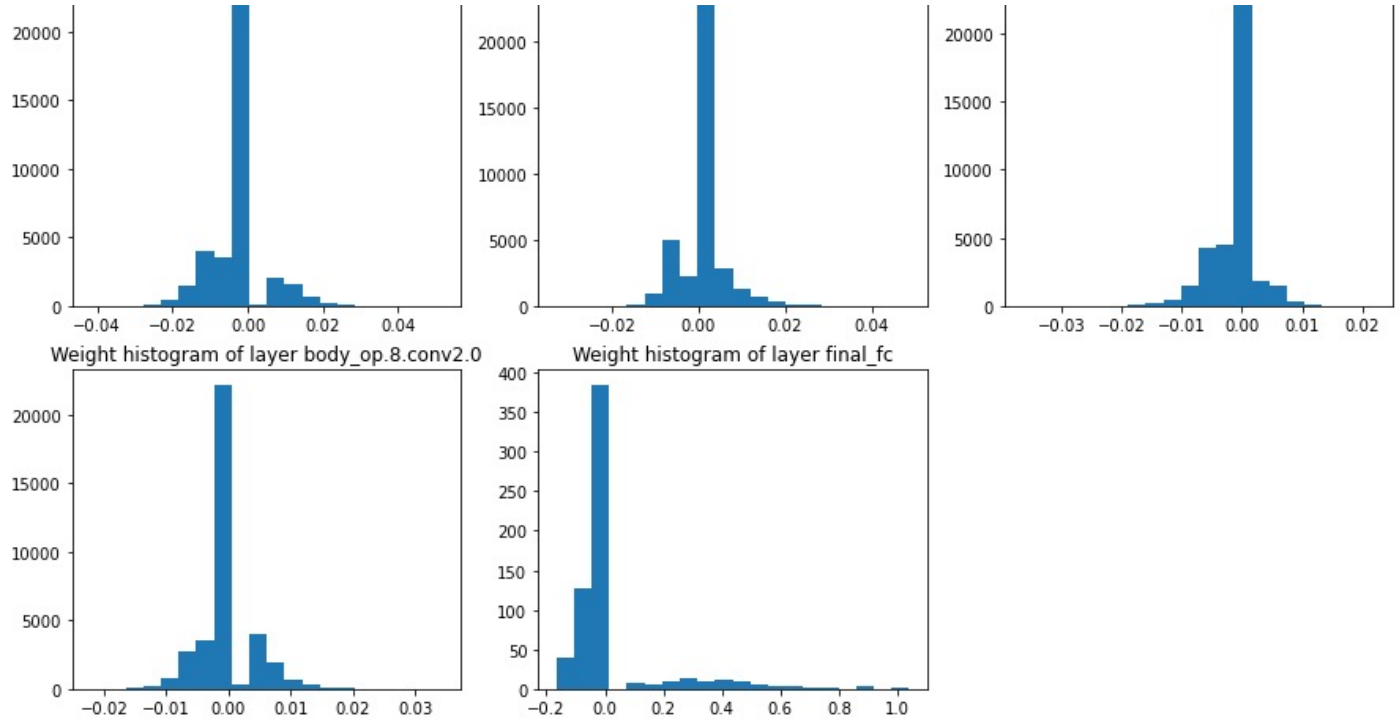
(b)

We can observe that histogram become discrete after doing quantization. For instance, in the figure after quantization with bits =2, we can see there are only 4 bars in weight distribution.

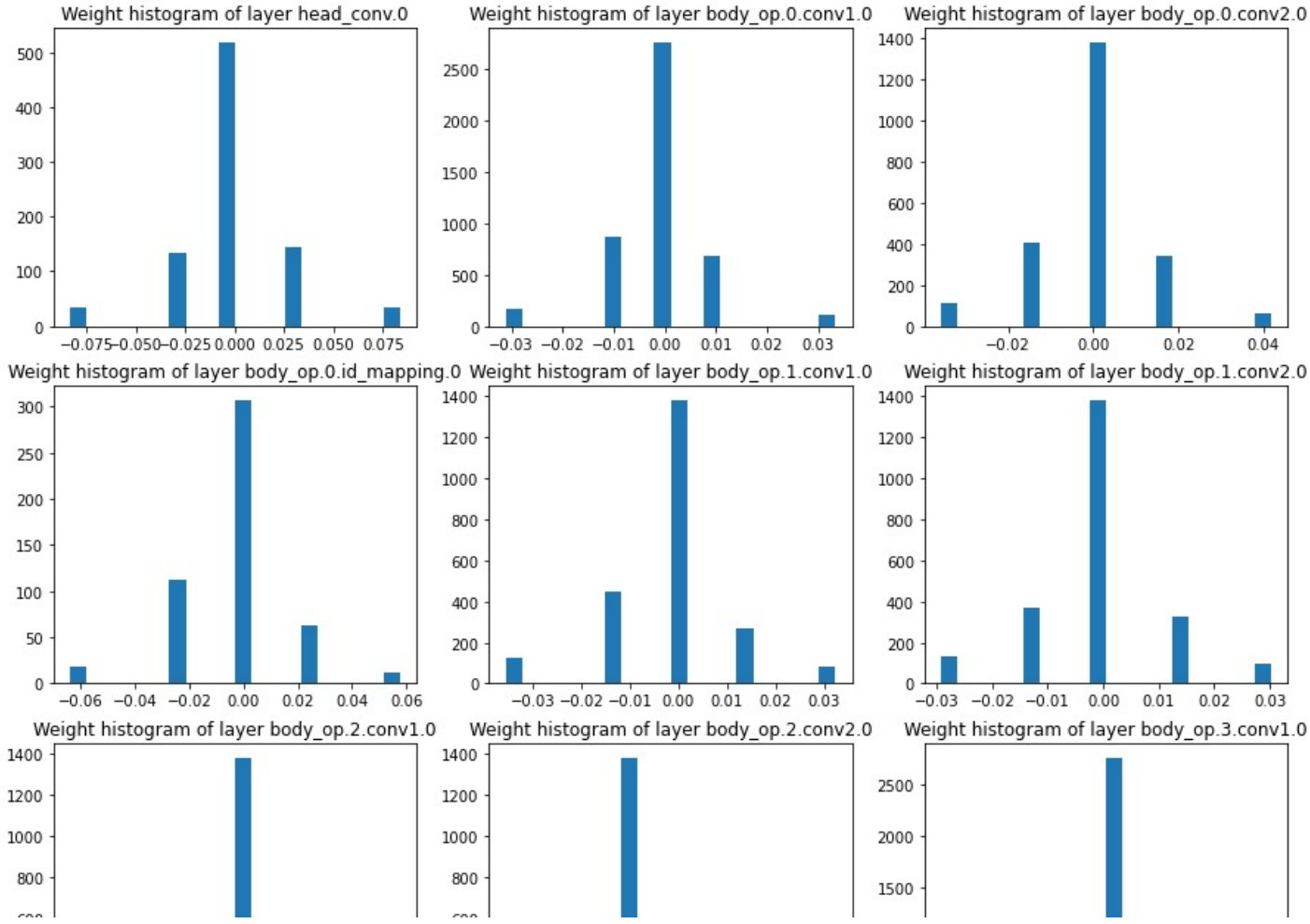
Weights distribution before quantization

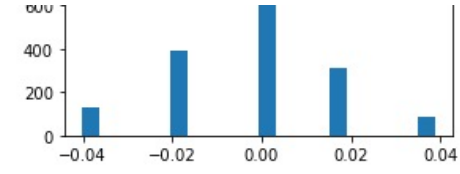


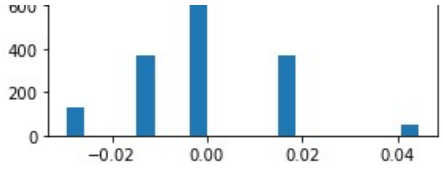

Weight histogram of layer body_op.2.conv1.0
Weight histogram of layer body_op.2.conv2.0
Weight histogram of layer body_op.3.conv1.0
Weight histogram of layer body_op.3.conv2.0
Weight histogram of layer body_op.3.id_mapping.0
Weight histogram of layer body_op.4.conv1.0
Weight histogram of layer body_op.4.conv2.0
Weight histogram of layer body_op.5.conv1.0
Weight histogram of layer body_op.5.conv2.0
Weight histogram of layer body_op.6.conv1.0
Weight histogram of layer body_op.6.conv2.0
Weight histogram of layer body_op.6.id_mapping.0
Weight histogram of layer body_op.7.conv1.0
Weight histogram of layer body_op.7.conv2.0
Weight histogram of layer body_op.8.conv1.0

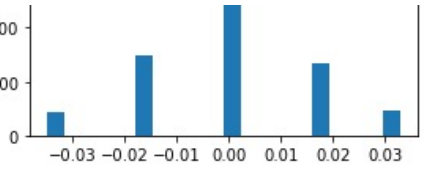


Weights distribution after quantization with bits = 2

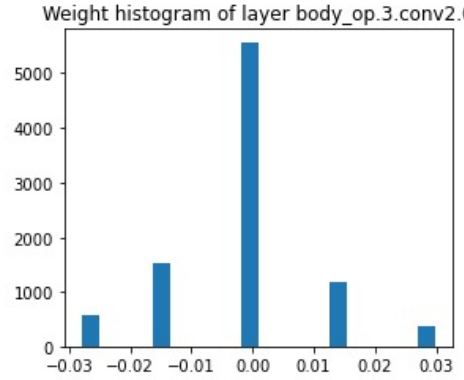


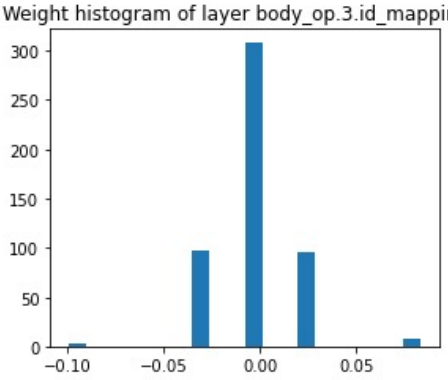


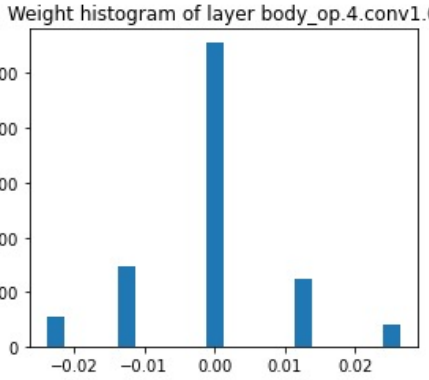




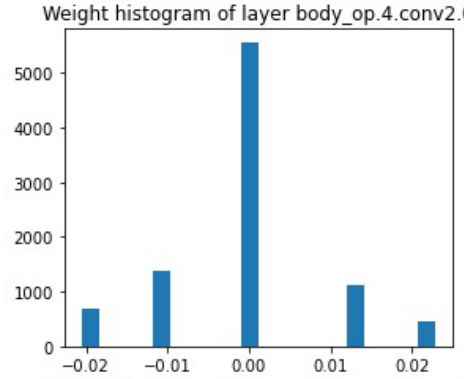
Weight histogram of layer body_op.3.conv2.0 Weight histogram of layer body_op.3.id_mapping.0 Weight histogram of layer body_op.4.conv1.0

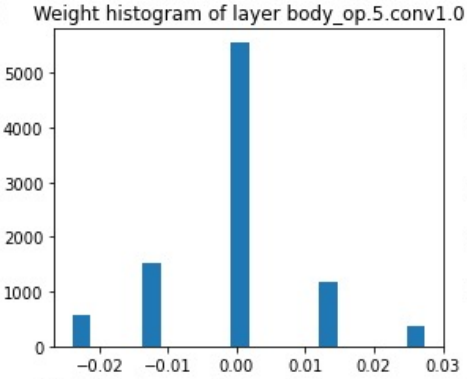


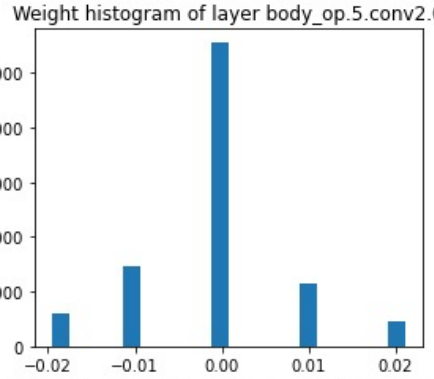




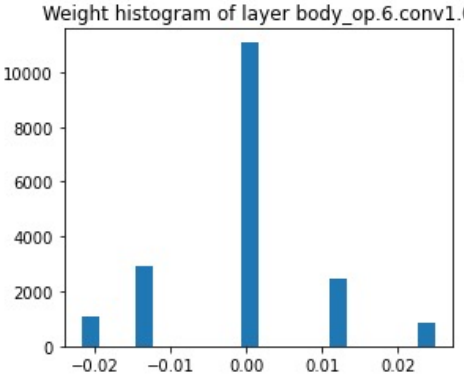
Weight histogram of layer body_op.4.conv2.0 Weight histogram of layer body_op.5.conv1.0 Weight histogram of layer body_op.5.conv2.0

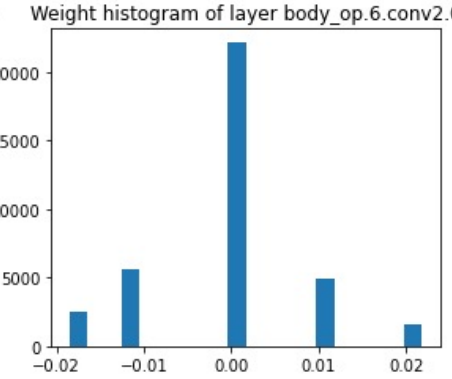


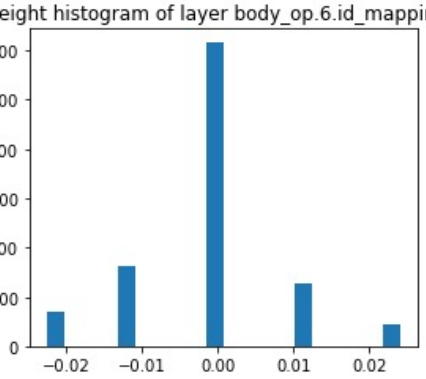




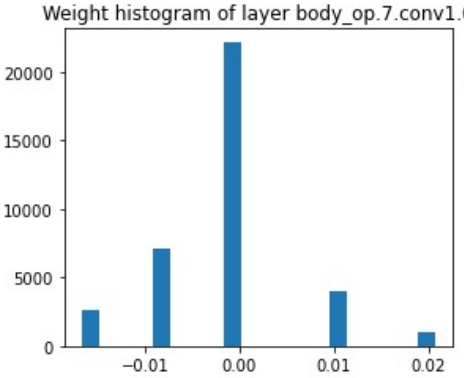
Weight histogram of layer body_op.6.conv1.0 Weight histogram of layer body_op.6.conv2.0 Weight histogram of layer body_op.6.id_mapping.0

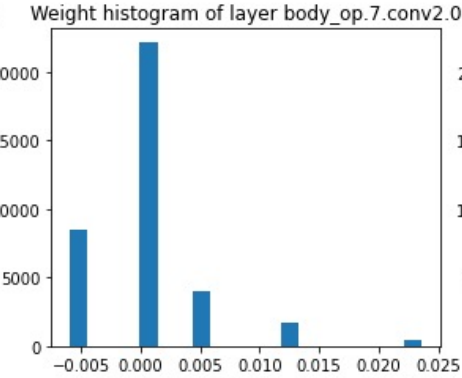


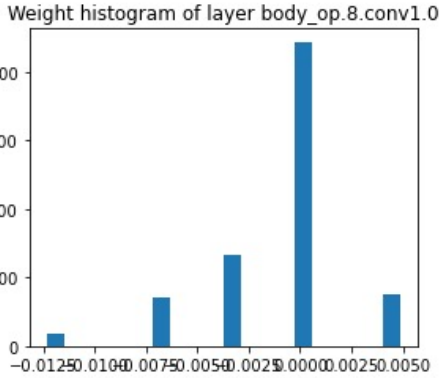




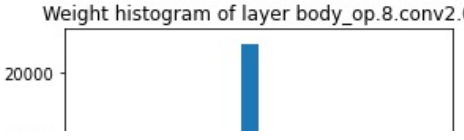
Weight histogram of layer body_op.7.conv1.0 Weight histogram of layer body_op.7.conv2.0 Weight histogram of layer body_op.8.conv1.0

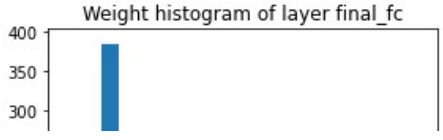


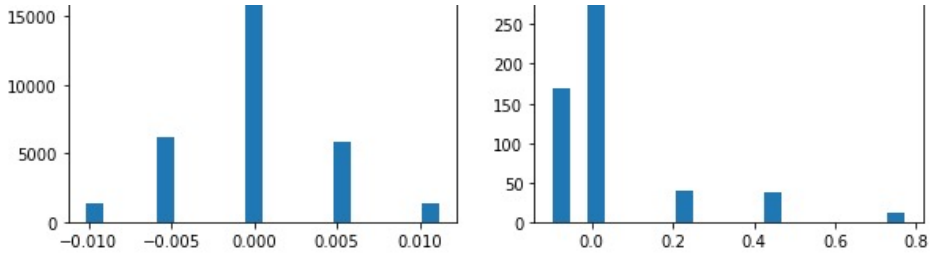




Weight histogram of layer body_op.8.conv2.0 Weight histogram of layer final_fc







(c)

From the table below, we can see that as we decrease the bits to represent each number, the test accuracy will decrease. But we can notice that when bits = 6,5,4, there are no large gap between test accuracy but when we set bits = 3, the test accuracy is below 90%. This is because the less bits we used to represent numebr, the more loss we will have. For instance, 4 bits can have 16 clusters but 3 bits only have 8 clusters.

q	6	5	4	3
accuracy before quantization	0.9133	0.9133	0.9133	0.9133
accuracy after quantization	0.9131	0.9121	0.9080	0.8785

(d)

From the table below, I set bits = 4 and we can see Linear initialization have the best performance with test accuracy over 90% while that of Random and Density-based is below 90%.

Initialization	Random	Density-based	Linear
accuracy before quantization	0.9133	0.9133	0.9133
accuracy after quantization	0.8971	0.8995	0.9023

(e)

The average bit length for the weight parameters: 4.1248654869269785

Q5 Lab5

I set q = 50 for pruning and bits = 4 for quantization and I got the final accuracy = 0.9023.

$$ratio = \frac{109660}{274144} \times \frac{1}{32} \times 3.3600863894650645 = 0.04200199911688052$$

In [ ]: