1. Warmup:

First hyperparameter tuning:

```
net = VGG16_half()
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
# Tune the hyperparameters here.
train(net, epochs=10, batch_size=233, lr=0.00, reg=0.0)
```

```
Epoch: 9
[Step=1936]     Loss=2.7532     acc=0.0773      531.3 examples/second
[Step=1952]     Loss=2.6958     acc=0.0997      1637.4 examples/second
[Step=1968]     Loss=2.7016     acc=0.0974      2402.2 examples/second
[Step=1984]     Loss=2.7003     acc=0.0981      2978.3 examples/second
[Step=2000]     Loss=2.6988     acc=0.0990      2848.2 examples/second
[Step=2016]     Loss=2.6987     acc=0.1003      2767.6 examples/second
[Step=2032]     Loss=2.7017     acc=0.0998      2761.1 examples/second
[Step=2048]     Loss=2.7034     acc=0.0985      2911.6 examples/second
[Step=2064]     Loss=2.7038     acc=0.0985      1839.3 examples/second
[Step=2080]     Loss=2.7054     acc=0.0982      1773.0 examples/second
[Step=2096]     Loss=2.7065     acc=0.0980      1800.0 examples/second
[Step=2112]     Loss=2.7068     acc=0.0980      3079.0 examples/second
[Step=2128]     Loss=2.7034     acc=0.0984      3432.5 examples/second
[Step=2144]     Loss=2.7045     acc=0.0980      5976.7 examples/second
Test Loss=2.6494, Test acc=0.0909
```

Accuracy = 9.09

Second Hyperparamter tuning:

```
net = VGG16_half()
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
# Tune the hyperparameters here.
train(net, epochs=70, batch_size=128, lr=0.009, reg=0.001)
```

```
Epoch: 69
[Step=26992]    Loss=0.0444    acc=0.9886    252.8 examples/second
[Step=27008]    Loss=0.0457    acc=0.9860    1291.4 examples/second
[Step=27024]    Loss=0.0481    acc=0.9861    2114.6 examples/second
[Step=27040]    Loss=0.0472    acc=0.9862    2373.6 examples/second
[Step=27056]    Loss=0.0477    acc=0.9856    1745.4 examples/second
[Step=27072]    Loss=0.0477    acc=0.9853    1521.3 examples/second
[Step=27088]    Loss=0.0474    acc=0.9852    1536.0 examples/second
[Step=27104]    Loss=0.0472    acc=0.9852    1542.6 examples/second
[Step=27120]    Loss=0.0469    acc=0.9853    2096.7 examples/second
[Step=27136]    Loss=0.0466    acc=0.9854    2398.0 examples/second
[Step=27152]    Loss=0.0456    acc=0.9858    2361.2 examples/second
[Step=27168]    Loss=0.0455    acc=0.9860    2414.6 examples/second
[Step=27184]    Loss=0.0455    acc=0.9859    2273.3 examples/second
[Step=27200]    Loss=0.0455    acc=0.9859    2362.5 examples/second
[Step=27216]    Loss=0.0453    acc=0.9860    2405.0 examples/second
[Step=27232]    Loss=0.0453    acc=0.9859    2381.6 examples/second
[Step=27248]    Loss=0.0455    acc=0.9858    2240.8 examples/second
[Step=27264]    Loss=0.0456    acc=0.9858    2483.4 examples/second
[Step=27280]    Loss=0.0454    acc=0.9860    2293.9 examples/second
[Step=27296]    Loss=0.0456    acc=0.9861    2028.0 examples/second
[Step=27312]    Loss=0.0460    acc=0.9859    1518.1 examples/second
[Step=27328]    Loss=0.0457    acc=0.9861    1507.3 examples/second
[Step=27344]    Loss=0.0460    acc=0.9860    1856.4 examples/second
[Step=27360]    Loss=0.0459    acc=0.9861    5448.2 examples/second
Test Loss=0.3289, Test acc=0.9082
```

Accuracy = 90.82

Third Hyperparamter tuning:

```
net = VGG16_half()
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
# Tune the hyperparameters here.
train(net, epochs=50, batch_size=128, lr=0.009, reg=0.001)
```

```
Epoch: 49
[Step=19168]    Loss=0.0658    acc=0.9818    276.1 examples/second
[Step=19184]    Loss=0.0817    acc=0.9772    1185.1 examples/second
[Step=19200]    Loss=0.0850    acc=0.9739    1869.6 examples/second
[Step=19216]    Loss=0.0890    acc=0.9723    2171.9 examples/second
[Step=19232]    Loss=0.0883    acc=0.9719    2140.3 examples/second
[Step=19248]    Loss=0.0884    acc=0.9721    1536.3 examples/second
[Step=19264]    Loss=0.0902    acc=0.9715    1508.2 examples/second
[Step=19280]    Loss=0.0894    acc=0.9717    1623.5 examples/second
[Step=19296]    Loss=0.0890    acc=0.9717    1713.6 examples/second
[Step=19312]    Loss=0.0896    acc=0.9715    2278.0 examples/second
[Step=19328]    Loss=0.0885    acc=0.9721    2541.9 examples/second
[Step=19344]    Loss=0.0891    acc=0.9719    2291.1 examples/second
[Step=19360]    Loss=0.0893    acc=0.9721    2357.1 examples/second
[Step=19376]    Loss=0.0899    acc=0.9720    2417.3 examples/second
[Step=19392]    Loss=0.0896    acc=0.9723    2425.8 examples/second
[Step=19408]    Loss=0.0887    acc=0.9727    2284.1 examples/second
[Step=19424]    Loss=0.0883    acc=0.9729    2370.6 examples/second
[Step=19440]    Loss=0.0877    acc=0.9731    2362.9 examples/second
[Step=19456]    Loss=0.0883    acc=0.9730    1799.3 examples/second
[Step=19472]    Loss=0.0885    acc=0.9729    2354.7 examples/second
[Step=19488]    Loss=0.0887    acc=0.9728    1488.0 examples/second
[Step=19504]    Loss=0.0892    acc=0.9726    1619.5 examples/second
[Step=19520]    Loss=0.0896    acc=0.9725    1623.1 examples/second
[Step=19536]    Loss=0.0895    acc=0.9725    5004.1 examples/second
Test Loss=0.3211, Test acc=0.9023
```

Accuracy = 90.23

Fourth Hyperparamter tuning:

Full-precision model training

[4]

```python
net = VGG16_half()
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
# Tune the hyperparameters here.
train(net, epochs=50, batch_size=128, lr=0.01, reg=0.005)
```

```
Epoch: 49
[Step=19168]    Loss=0.1342     acc=0.9653      309.3 examples/second
[Step=19184]    Loss=0.1113     acc=0.9731      1379.1 examples/second
[Step=19200]    Loss=0.1172     acc=0.9701      1720.0 examples/second
[Step=19216]    Loss=0.1198     acc=0.9688      2500.3 examples/second
[Step=19232]    Loss=0.1217     acc=0.9680      2647.3 examples/second
[Step=19248]    Loss=0.1200     acc=0.9685      2426.5 examples/second
[Step=19264]    Loss=0.1182     acc=0.9681      2673.6 examples/second
[Step=19280]    Loss=0.1160     acc=0.9693      2429.5 examples/second
[Step=19296]    Loss=0.1160     acc=0.9689      2586.1 examples/second
[Step=19312]    Loss=0.1162     acc=0.9686      2389.2 examples/second
[Step=19328]    Loss=0.1177     acc=0.9678      2683.6 examples/second
[Step=19344]    Loss=0.1178     acc=0.9677      2734.4 examples/second
[Step=19360]    Loss=0.1179     acc=0.9676      2434.4 examples/second
[Step=19376]    Loss=0.1186     acc=0.9675      2336.4 examples/second
[Step=19392]    Loss=0.1199     acc=0.9671      2559.7 examples/second
[Step=19408]    Loss=0.1197     acc=0.9671      1610.0 examples/second
[Step=19424]    Loss=0.1187     acc=0.9678      1556.3 examples/second
[Step=19440]    Loss=0.1189     acc=0.9675      1690.3 examples/second
[Step=19456]    Loss=0.1187     acc=0.9677      1624.5 examples/second
[Step=19472]    Loss=0.1192     acc=0.9676      1992.4 examples/second
[Step=19488]    Loss=0.1190     acc=0.9677      2390.3 examples/second
[Step=19504]    Loss=0.1189     acc=0.9678      2466.4 examples/second
[Step=19520]    Loss=0.1192     acc=0.9676      2584.5 examples/second
[Step=19536]    Loss=0.1193     acc=0.9677      5989.3 examples/second
Test Loss=0.2908, Test acc=0.9133
```

```python
# Load the best weight paramters
net.load_state_dict(torch.load("net_before_pruning.pt"))
test(net)
```

```
Files already downloaded and verified
Test Loss=0.2860, Test accuracy=0.9145
```

Accuracy = 91.45

Hence, accuracy more than 90 is achieved.

2. Weight Pruning:

(a)

```python
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.
    """
    """
    Prune the weight connections by percentage. Calculate the sparisty after
    pruning and store it into 'self.sparsity'.
    Store the pruning pattern in 'self.mask' for further fine-tuning process
    with pruned connections.
    --------------Your Code---------------------
    """
    weight_values = self.linear.weight.data.abs().view(-1)
    threshold = np.percentile(weight_values.cpu().numpy(), q)

    self.mask = (self.linear.weight.data.abs() >= threshold).type(torch.float)
    self.linear.weight.data *= self.mask  # Apply the mask to weights

    sparsity = 1.0 - (self.mask.sum() / self.mask.numel())
    self.sparsity = sparsity

    pass
```

```python
def prune_by_percentage(self, q=5.0):
    """
    Pruning by a factor of the standard deviation value.
    :param s: (scalar) factor of the standard deviation value.
    Weight magnitude below np.std(weight)*std
    will be pruned.
    """

    """
    Prune the weight connections by percentage. Calculate the sparisty after
    pruning and store it into 'self.sparsity'.
    Store the pruning pattern in 'self.mask' for further fine-tuning process
    with pruned connections.
    --------------Your Code---------------------
    """

    weight_values = self.conv.weight.data.abs().view(-1)
    threshold = np.percentile(weight_values.cpu().numpy(), q)

    self.mask = (self.conv.weight.data.abs() >= threshold).type(torch.float)
    self.conv.weight.data *= self.mask  # Apply the mask to weights

    sparsity = 1.0 - (self.mask.sum() / self.mask.numel())
    self.sparsity = sparsity
```

(b)

```python
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 np.std(weight)*std
    will be pruned.
    """

    """
    Prune the weight connections by standarad deviation.
    Calculate the sparisty after pruning and store it into 'self.sparsity'.
    Store the pruning pattern in 'self.mask' for further fine-tuning process
    with pruned connections.
    --------------Your Code---------------------
    """
   # Calculate the standard deviation of weight values
    weight_std = torch.std(self.linear.weight.data)

    threshold = weight_std * s

    self.mask = (self.linear.weight.data.abs() >= threshold).type(torch.float)
    self.linear.weight.data *= self.mask  # Apply the mask to weights

    sparsity = 1.0 - (self.mask.sum() / self.mask.numel())
    self.sparsity = sparsity

    pass
```

```python
def prune_by_std(self, s=0.25):
    """
    Pruning by a factor of the standard deviation value.
    :param s: (scalar) factor of the standard deviation value.
    Weight magnitude below np.std(weight)*std
    will be pruned.
    """

    """
    Prune the weight connections by standarad deviation.
    Calculate the sparisty after pruning and store it into 'self.sparsity'.
    Store the pruning pattern in 'self.mask' for further fine-tuning process
    with pruned connections.
    --------------Your Code---------------------
    """
    # Calculate the standard deviation of weight values
    weight_std = torch.std(self.conv.weight.data)

    threshold = weight_std * s

    self.mask = (self.conv.weight.data.abs() >= threshold).type(torch.float)
    self.conv.weight.data *= self.mask  # Apply the mask to weights

    sparsity = 1.0 - (self.mask.sum() / self.mask.numel())
    self.sparsity = sparsity

    pass
```

(c)

```python
[ ]  # Test accuracy before fine-tuning
     prune(net, method='std', q=45.0, s=0.75)
     test(net)
```

```
Files already downloaded and verified
Test Loss=0.5929, Test accuracy=0.8387
```

Test accuracy observed is 83.87

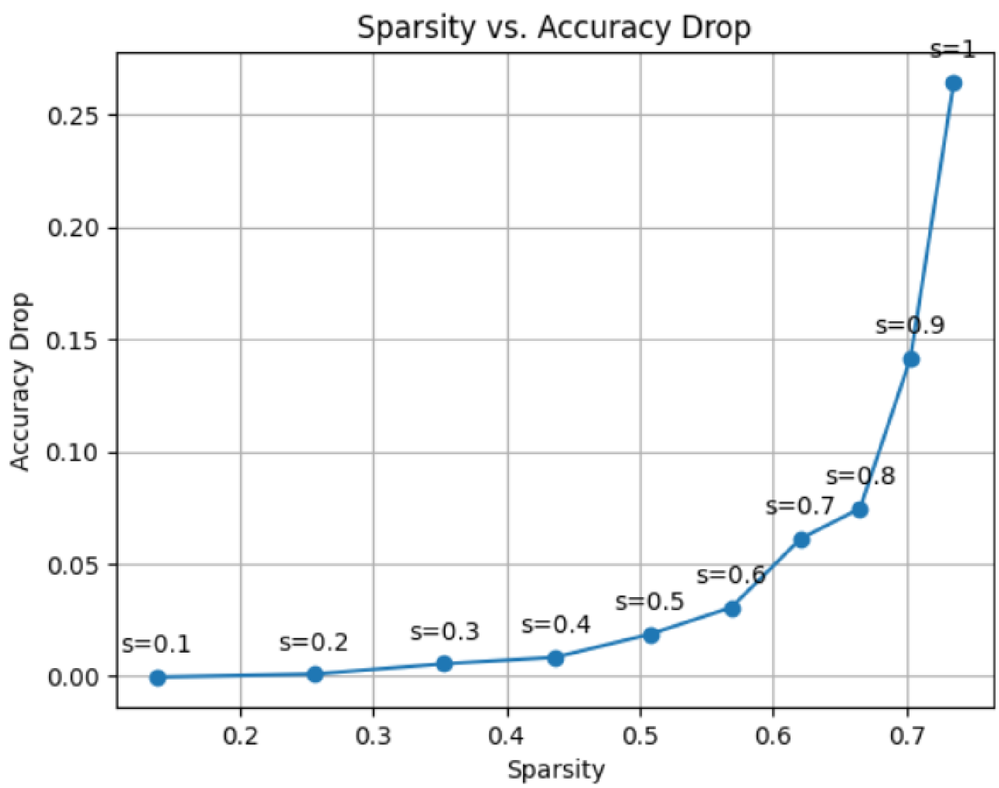
No, the test accuracy does not remain the same on CIFAR 10 dataset.

The test accuracy is decreased from 91.45 to 83.87 because of the pruning technique which we have applied on the model. This can result in the loss of fine-grained features and patterns that the model had learned during training.. This loss results in a decrease in accuracy.

```
46                Linear          2560
Total nonzero parameters: 1315679
Total parameters: 3811680
Total sparsity: 0.654830
-------------------------------
```

Sparsity of pruned network is observed as 0.654

(d)



We have varied the sensitivity values as shown in the above plot from 0.1 to 1. We observed the changes in accuracy drop and sparsity and plotted it as above.

We can clearly conclude from the plot that as we increase the sensitivity of the model the accuracy drop increases which means accuracy is decreased and at the same time sparsity of the network model is increased.

(e)

From the below snapshot we have changed the method to percentage and chose q=65.485 to get the same sparsity as given in (c) which is 0.654

Accuracy after changing to percentage and q value as same as to get same sparsity as std:

```
[29] # Test accuracy before fine-tuning
     prune(net, method='percentage', q=65.485, s=0.75)
     test(net)

     Files already downloaded and verified
     Test Loss=0.6855, Test accuracy=0.7958
     0.7958
```

The test accuracy is observed as 79.58 after changing to a percentage method.
Sparsity after changing to percentage:

```
46                 Linear           2560
Total nonzero parameters: 1315605
Total parameters: 3811680
Total sparsity: 0.654849
-------------------------------
```

No, the test accuracy does not remain the same on CIFAR 10 dataset.
There is a drop in accuracy as compared to the standard deviation method which is applied in part c of the question.

It is observed that there is a significant drop in accuracy in percentage pruning as compared to the standard deviation pruning method and so standard deviation method is better than percentage pruning method.

3. **Fine-tuning Pruned DNNs:**

```python
"""
Zero the gradients of the pruned variables.
-----------------------Your Code------------------------
"""

for param in net.parameters():
    param.grad =  torch.zeros_like(param.data)
```

First hyperparameter tuning:

```python
# Uncomment to load pretrained weights
# net.load_state_dict(torch.load("net_after_pruning.pt"))
# Comment if you have loaded pretrained weights
finetune_after_prune(net, epochs=50, batch_size=128, lr=0.001, reg=5e-5)
```

```
Epoch: 49
[Step=19168]    Loss=0.2389    acc=0.9332    338.5 examples/second
[Step=19184]    Loss=0.2373    acc=0.9363    2014.8 examples/second
[Step=19200]    Loss=0.2469    acc=0.9339    2303.5 examples/second
[Step=19216]    Loss=0.2495    acc=0.9330    2135.0 examples/second
[Step=19232]    Loss=0.2466    acc=0.9338    1411.6 examples/second
[Step=19248]    Loss=0.2471    acc=0.9341    1486.6 examples/second
[Step=19264]    Loss=0.2449    acc=0.9352    1542.2 examples/second
[Step=19280]    Loss=0.2402    acc=0.9376    1700.9 examples/second
[Step=19296]    Loss=0.2416    acc=0.9365    2290.1 examples/second
[Step=19312]    Loss=0.2398    acc=0.9374    2322.5 examples/second
[Step=19328]    Loss=0.2410    acc=0.9371    2357.2 examples/second
[Step=19344]    Loss=0.2408    acc=0.9371    2229.2 examples/second
[Step=19360]    Loss=0.2410    acc=0.9372    2258.1 examples/second
[Step=19376]    Loss=0.2414    acc=0.9371    2357.4 examples/second
[Step=19392]    Loss=0.2402    acc=0.9371    2284.3 examples/second
[Step=19408]    Loss=0.2402    acc=0.9371    2341.8 examples/second
[Step=19424]    Loss=0.2412    acc=0.9367    2316.6 examples/second
[Step=19440]    Loss=0.2418    acc=0.9365    2256.1 examples/second
[Step=19456]    Loss=0.2421    acc=0.9362    2177.1 examples/second
[Step=19472]    Loss=0.2410    acc=0.9367    1548.8 examples/second
[Step=19488]    Loss=0.2413    acc=0.9368    1545.1 examples/second
[Step=19504]    Loss=0.2410    acc=0.9369    1501.9 examples/second
[Step=19520]    Loss=0.2409    acc=0.9370    1733.0 examples/second
[Step=19536]    Loss=0.2401    acc=0.9372    5275.4 examples/second
Test Loss=0.3721, Test acc=0.8915
```

Accuracy = 89.15

Second hyperparameter tuning:

```
# Uncomment to load pretrained weights
# net.load_state_dict(torch.load("net_after_pruning.pt"))
# Comment if you have loaded pretrained weights
finetune_after_prune(net, epochs=30, batch_size=64, lr=0.001, reg=0.001)
```

```
[Step=23136]    Loss=0.5300    acc=0.9291    2149.3 examples/second
[Step=23152]    Loss=0.5297    acc=0.9292    2000.8 examples/second
[Step=23168]    Loss=0.5294    acc=0.9296    2080.3 examples/second
[Step=23184]    Loss=0.5298    acc=0.9294    2153.7 examples/second
[Step=23200]    Loss=0.5293    acc=0.9299    2041.4 examples/second
[Step=23216]    Loss=0.5294    acc=0.9299    2148.2 examples/second
[Step=23232]    Loss=0.5298    acc=0.9298    2180.9 examples/second
[Step=23248]    Loss=0.5295    acc=0.9298    2133.9 examples/second
[Step=23264]    Loss=0.5296    acc=0.9298    2105.6 examples/second
[Step=23280]    Loss=0.5299    acc=0.9299    1492.8 examples/second
[Step=23296]    Loss=0.5301    acc=0.9299    1285.6 examples/second
[Step=23312]    Loss=0.5305    acc=0.9300    1337.8 examples/second
[Step=23328]    Loss=0.5310    acc=0.9298    1300.8 examples/second
[Step=23344]    Loss=0.5316    acc=0.9296    1129.5 examples/second
[Step=23360]    Loss=0.5313    acc=0.9298    1139.1 examples/second
[Step=23376]    Loss=0.5312    acc=0.9299    1138.3 examples/second
[Step=23392]    Loss=0.5311    acc=0.9300    1237.2 examples/second
[Step=23408]    Loss=0.5311    acc=0.9299    1202.9 examples/second
[Step=23424]    Loss=0.5312    acc=0.9300    1230.7 examples/second
[Step=23440]    Loss=0.5314    acc=0.9300    2204.2 examples/second
[Step=23456]    Loss=0.5311    acc=0.9300    4003.3 examples/second
Test Loss=0.5820, Test acc=0.8928
```

Accuracy = 89.28

Third hyperparameter tuning:

```
# Uncomment to load pretrained weights
# net.load_state_dict(torch.load("net_after_pruning.pt"))
# Comment if you have loaded pretrained weights
finetune_after_prune(net, epochs=20, batch_size=64, lr=0.001, reg=0.0001)
```

```
[Step=15376]    Loss=0.2870    acc=0.9271    2128.0 examples/second
[Step=15392]    Loss=0.2862    acc=0.9275    1969.3 examples/second
[Step=15408]    Loss=0.2866    acc=0.9273    2065.1 examples/second
[Step=15424]    Loss=0.2871    acc=0.9272    2105.8 examples/second
[Step=15440]    Loss=0.2862    acc=0.9276    1893.8 examples/second
[Step=15456]    Loss=0.2862    acc=0.9275    2133.2 examples/second
[Step=15472]    Loss=0.2858    acc=0.9276    1975.6 examples/second
[Step=15488]    Loss=0.2863    acc=0.9276    2202.1 examples/second
[Step=15504]    Loss=0.2867    acc=0.9276    1999.9 examples/second
[Step=15520]    Loss=0.2864    acc=0.9278    2099.7 examples/second
[Step=15536]    Loss=0.2860    acc=0.9280    1663.6 examples/second
[Step=15552]    Loss=0.2859    acc=0.9281    1245.9 examples/second
[Step=15568]    Loss=0.2861    acc=0.9278    1280.6 examples/second
[Step=15584]    Loss=0.2859    acc=0.9278    1304.6 examples/second
[Step=15600]    Loss=0.2860    acc=0.9276    1205.8 examples/second
[Step=15616]    Loss=0.2855    acc=0.9279    1840.5 examples/second
[Step=15632]    Loss=0.2848    acc=0.9281    3800.1 examples/second
Test Loss=0.3801, Test acc=0.8918
```

Accuracy = 89.18

Fourth hyperparameter tuning:

```
[7] # Uncomment to load pretrained weights
    # net.load_state_dict(torch.load("net_after_pruning.pt"))
    # Comment if you have loaded pretrained weights
    finetune_after_prune(net, epochs=30, batch_size=64, lr=0.01, reg=0.00001)
```

```
# Load the best weight paramters
net.load_state_dict(torch.load("net_after_pruning.pt"))
test(net)
```

```
Files already downloaded and verified
Test Loss=0.3754, Test accuracy=0.8945
0.8945
```

Accuracy = 89.45

Note that 89.45 was the highest test accuracy observed after fine tuning the pruned DNN on CIFAR-10 dataset.

Fine tuning processes have changed the accuracy from 83.87 to 89.45.
Note that 83.87 was observed in standard deviation weight pruning method.

Also the model's original accuracy is 91.45 which is not fully recovered after the fine tuning. Hence fine tuning did not able to recover the accuracy loss fully but it recovered partially up to 89.45

So in the last part of this project i.e. in question 6 while doing iterative pruning we tried to recover the accuracy loss.

4. Quantization and Weight Sharing:

(a)

```
cluster_centers = []
assert isinstance(net, nn.Module)
layer_ind = 0
for n, m in net.named_modules():
    if isinstance(m, PrunedConv):
        pass
        """
        Apply quantization for the PrunedConv layer.
        --------------Your Code---------------------
        """
        weights = m.conv.weight.data.cpu().numpy().flatten()
        kmeans = KMeans(n_clusters=2**bits,init='k-means++')
        kmeans.fit(weights.reshape(-1,1))
        quantized_weights = kmeans.cluster_centers_[kmeans.predict(weights.reshape(-1,1))].reshape(weights.shape)
        m.conv.weight.data = torch.from_numpy(quantized_weights.reshape(m.conv.weight.data.shape)).to(device)
        cluster_centers.append(kmeans.cluster_centers_)


        layer_ind += 1
        print("Complete %d layers quantization..." %layer_ind)
    elif isinstance(m, PruneLinear):
        """
        Apply quantization for the PrunedLinear layer.
        --------------Your Code---------------------
        """
        weights = m.linear.weight.data.cpu().numpy().flatten()
        kmeans = KMeans(n_clusters=2**bits,init='k-means++')
        kmeans.fit(weights.reshape(-1,1))
        quantized_weights = kmeans.cluster_centers_[kmeans.predict(weights.reshape(-1,1))].reshape(weights.shape)
        m.linear.weight.data = torch.from_numpy(quantized_weights.reshape(m.linear.weight.data.shape)).to(device)
        cluster_centers.append(kmeans.cluster_centers_)
```

```
[ ]  centers = quantize_whole_model(net, bits=5)
     np.save("codebook_vgg16.npy", centers)
```

```
test(net)
```

```
Files already downloaded and verified
Test Loss=0.3752, Test accuracy=0.8938
0.8938
```

After setting the quantization to bits = 5 accuracy observed is 89.38.
Quantization has reduced the accuracy of the model very little which is almost negligible i.e from 89.45 to 89.38.

Quantization affects the performance of DNN models depending on how much lower precision we are aiming to represent each weight element. Quantization reduces the bit width required to represent weights and activations.

In our case we are reducing the bits from 5 to 4 for more compression. This is done by weight sharing which increases efficient inference.

There is not much drop in accuracy while we are compressing the model using quantization.

(b)

```
[8] centers = quantize_whole_model(net, bits=4)
    np.save("codebook_vgg16.npy", centers)
```

```
[8] test(net)
```

```
Files already downloaded and verified
Test Loss=0.3803, Test accuracy=0.8920
0.892
```

When we change bits from 5 to 4 , test accuracy is further dropped from 89.38 to 89.2. This drop is not very significant.

```
[8] centers = quantize_whole_model(net, bits=3)
    np.save("codebook_vgg16.npy", centers)
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/cl
```

```
test(net)
```

```
Files already downloaded and verified
Test Loss=0.4699, Test accuracy=0.8673
0.8673
```

We selected the optimal bit as 4 since the accuracy drop since using 3 bits the accuracy is dropped significantly to 86.73 as shown above

Now the optimal bit is selected as 4 due to small drop of accuracy i.e. actually from 89.45 to 89.2 which can be regained if we do iterative pruning and then applying quantization as done in question 6.

Also as compared to 5 bit , 4 bit has less huffman length which is calculated in the next question which impacts compression rate.

There is a tradeoff between accuracy and memory consumption.

5 bits has slightly more accuracy than 4 bits but 4 bits has less memory consumption due to the high compression rate as calculated in question 6.

Hence if we want more accuracy without caring memory consumption then we can go for 5 bits , but if we want more compression rate and we can manage the accuracy via iterative pruning then we can go for 4 bits as quant bits.

5. Huffman Coding

(a)

(i) Huffman coding is applied to the quantized weights. It assigns shorter codes to frequently occurring weight values and longer codes to less common values. In other words, it encodes the weights in a variable-length binary representation, where frequently occurring values are represented by shorter bit sequences, and rare values are represented by longer bit sequences.

(ii) The Huffman-encoded weights are stored along with the Huffman code dictionary. The weights are stored in a compressed format. The compressed weights are loaded into memory during inference.

(iii) The reduced memory footprint translates into faster model loading times and lower memory bandwidth requirements during inference, which can improve the overall efficiency of DNN applications.

Hence Huffman coding reduces the memory footprint of DNNs.

(b)

```
Generate Huffman Coding and Frequency Map according to incoming weights and centers (KMeans centriods).
--------------Your Code---------------------
"""
freq_map = Counter(weight.reshape(-1))  # Count the frequency of each weight parameter

# Build the Huffman tree
heap = [[weight, [val, ""]] for val, weight in freq_map.items()]
heapq.heapify(heap)
while len(heap) > 1:
    lo = heapq.heappop(heap)
    hi = heapq.heappop(heap)
    for pair in lo[1:]:
        pair[1] = '0' + pair[1]
    for pair in hi[1:]:
        pair[1] = '1' + pair[1]
    heapq.heappush(heap, [lo[0] + hi[0]] + lo[1:] + hi[1:])
huffman_tree = sorted(heap[0][1:], key=lambda p: (len(p[-1]), p))

encodings = {val: code for val, code in huffman_tree}
frequency = {val: freq_map[val] for val, _ in huffman_tree}
return encodings, frequency
```

```
Average storage for each parameter after Huffman Coding: 2.8614 bits
Complete 8 layers for Huffman Coding...
Original storage for each parameter: 5.0000 bits
Average storage for each parameter after Huffman Coding: 2.7614 bits
Complete 9 layers for Huffman Coding...
Original storage for each parameter: 5.0000 bits
Average storage for each parameter after Huffman Coding: 2.6600 bits
Complete 10 layers for Huffman Coding...
Original storage for each parameter: 5.0000 bits
Average storage for each parameter after Huffman Coding: 2.4659 bits
Complete 11 layers for Huffman Coding...
Original storage for each parameter: 5.0000 bits
Average storage for each parameter after Huffman Coding: 2.2830 bits
Complete 12 layers for Huffman Coding...
Original storage for each parameter: 5.0000 bits
Average storage for each parameter after Huffman Coding: 1.9238 bits
Complete 13 layers for Huffman Coding...
Original storage for each parameter: 5.0000 bits
Average storage for each parameter after Huffman Coding: 2.8867 bits
Complete 14 layers for Huffman Coding...
Original storage for each parameter: 5.0000 bits
Average storage for each parameter after Huffman Coding: 2.4944 bits
Complete 15 layers for Huffman Coding...
Original storage for each parameter: 5.0000 bits
Average storage for each parameter after Huffman Coding: 1.7047 bits
Complete 16 layers for Huffman Coding...
```

Implementing the huffman code for bits = 5.

```
Original storage for each parameter: 4.0000 bits
Average storage for each parameter after Huffman Coding: 2.5536 bits
Complete 5 layers for Huffman Coding...
Original storage for each parameter: 4.0000 bits
Average storage for each parameter after Huffman Coding: 2.5788 bits
Complete 6 layers for Huffman Coding...
Original storage for each parameter: 4.0000 bits
Average storage for each parameter after Huffman Coding: 2.5360 bits
Complete 7 layers for Huffman Coding...
Original storage for each parameter: 4.0000 bits
Average storage for each parameter after Huffman Coding: 2.4641 bits
Complete 8 layers for Huffman Coding...
Original storage for each parameter: 4.0000 bits
Average storage for each parameter after Huffman Coding: 2.3512 bits
Complete 9 layers for Huffman Coding...
Original storage for each parameter: 4.0000 bits
Average storage for each parameter after Huffman Coding: 2.2780 bits
Complete 10 layers for Huffman Coding...
Original storage for each parameter: 4.0000 bits
Average storage for each parameter after Huffman Coding: 2.1475 bits
Complete 11 layers for Huffman Coding...
Original storage for each parameter: 4.0000 bits
Average storage for each parameter after Huffman Coding: 1.9909 bits
Complete 12 layers for Huffman Coding...
Original storage for each parameter: 4.0000 bits
Average storage for each parameter after Huffman Coding: 1.7117 bits
Complete 13 layers for Huffman Coding...
Original storage for each parameter: 4.0000 bits
Average storage for each parameter after Huffman Coding: 2.4342 bits
Complete 14 layers for Huffman Coding...
Original storage for each parameter: 4.0000 bits
Average storage for each parameter after Huffman Coding: 2.1574 bits
Complete 15 layers for Huffman Coding...
Original storage for each parameter: 4.0000 bits
```

Implementing the huffman code for bits = 4.

(c)

```
for i, (frequencies, encodings) in enumerate(zip(frequency_map, encoding_map), st
    layer_name = str(i)  # You may use the layer index as the name

    # Calculate the weighted sum for the current layer
    weighted_storage_for_layer = sum(frequencies[key] * len(encodings[key]) for ke

    # Add the weighted storage to the total
    total_weighted_storage += weighted_storage_for_layer

    # Add the total parameters for the current layer to the overall total
    total_parameters += sum(frequencies.values())

# Calculate the weighted average storage for the entire network
weighted_average = total_weighted_storage / total_parameters

print("Weighted Average Storage: %.2f" %weighted_average)
```

```
Weighted Average Storage: 2.53
```

Calculating the average encoding length of the clustered weight variables for 5 bits = 2.53.

```
print("Weighted Average Storage: %.2f" %weighted_
```

```
Weighted Average Storage: 2.18
```

Calculating the average encoding length of the clustered weight variables for 4 bits = 2.18.

Huffman length / Original length gives us the additional memory reduction with the usage of huffman coding.

In case of 5 bits it is 2.53/5 = 0.506

In case of 4 bits it is 2.18/4 = 0.545

This gives the quantitative idea of memory reduction which is used in the compression rate ratio.

6. Putting All Together:

After performing steps 1 to 5 following observations are made:

Full precision model accuracy is 91.45 after fine tuning.

After pruning:

Total nonzero parameters: 1315679
Total parameters: 3811680
Total sparsity: 0.654830

With s=0.75

After fine tuning the pruned model:
Accuracy : 89.45

After Quantization by using 5 bits:

Accuracy = 89.38

After Quantization by using 4 bits:

Accuracy = 89.2

Huffman length using 5 bits:
2.53

Huffman length using 4 bits:
2.18

Selecting best parameters to calculate the ratio as :

(Nonzero parameters/total parameters) * (quant bits/32) * (huffman length/original length)

1315679/3811680 * 4/32 * 2.18/4
(Here we are taking quant bits = 4 because its huffman length is smaller than bit = 5 so as to help us in getting compression ratio greater than 40 or less than 1/40)

0.345 * 0.125 * 0.545 = 0.0235.

Which gives us compression rate as 42.55 which is greater than 40
We are able to achieve compression rate but accuracy is still less than 89.5

Hence now we will proceed for iterative tuning:

First round of iterative pruning:

Iteration 1:

Started with s=0.5

```
# Test accuracy before fine-tuning
prune(net, method='std', q=45.0, s=0.5)
test(net)
```

```
Files already downloaded and verified
Test Loss=0.3427, Test accuracy=0.8970
0.897
```

Now hyperparameter tuning:

```
# Uncomment to load pretrained weights
# net.load_state_dict(torch.load("net_after_pruning.pt"))
# Comment if you have loaded pretrained weights
finetune_after_prune(net, epochs=30, batch_size=64, lr=0.001, reg=0.00001)
```

```
# Load the best weight paramters
net.load_state_dict(torch.load("net_after_pruning.pt"))
test(net)
```

```
Files already downloaded and verified
Test Loss=0.3043, Test accuracy=0.9100
0.91
```

```
Total nonzero parameters: 1864351
Total parameters: 3811680
Total sparsity: 0.510885
-------------------------------
```

Iteration 2:

Then we changed s to 0.75

```
[10] # Test accuracy before fine-tuning
     prune(net, method='std', q=45.0, s=0.75)
     test(net)
```

```
Files already downloaded and verified
Test Loss=0.4424, Test accuracy=0.8753
0.8753
```

Now hyperparameter tuning:

```
# Uncomment to load pretrained weights
# net.load_state_dict(torch.load("net_after_pruning.pt"))
# Comment if you have loaded pretrained weights
finetune_after_prune(net, epochs=30, batch_size=64, lr=0.001, reg=0.00001)
```

```
[12] # Load the best weight paramters
     net.load_state_dict(torch.load("net_after_pruning.pt"))
     test(net)
```

```
Files already downloaded and verified
Test Loss=0.3626, Test accuracy=0.8964
0.8964
```

```
Total nonzero parameters: 1338275
Total parameters: 3811680
Total sparsity: 0.648902
-------------------------------
```

Now doing quantization with 4 bits:

```
centers = quantize_whole_model(net, bits=4)
np.save("codebook_vgg16.npy", centers)
```

```
test(net)
```

```
Files already downloaded and verified
Test Loss=0.3715, Test accuracy=0.8944
0.8944
```

Accuracy reduced to 89.44

Hence we did the second round of iterative pruning.

Second round of iterative pruning:

Iteration 1:
Started with s=0.5

```
[6] # Test accuracy before fine-tuning
    prune(net, method='std', q=45.0, s=0.5)
    test(net)
```

```
Files already downloaded and verified
Test Loss=0.3427, Test accuracy=0.8970
0.897
```

Now hyperparameter tuning:

```
# Uncomment to load pretrained weights
# net.load_state_dict(torch.load("net_after_pruning.pt"))
# Comment if you have loaded pretrained weights
finetune_after_prune(net, epochs=30, batch_size=64, lr=0.01, reg=0.00001)
```

The only change from my previous iteration is the learning rate is increased to 0.01 from 0.001

```
[8] # Load the best weight paramters
    net.load_state_dict(torch.load("net_after_pruning.pt"))
    test(net)
```

```
Files already downloaded and verified
Test Loss=0.3075, Test accuracy=0.9106
0.9106
```

```
Total nonzero parameters: 1864351
Total parameters: 3811680
Total sparsity: 0.510885
-------------------------------
```

Iteration 2:
Changed s to 0.75

```
[10] # Test accuracy before fine-tuning
     prune(net, method='std', q=45.0, s=0.75)
     test(net)

     Files already downloaded and verified
     Test Loss=0.4706, Test accuracy=0.8702
     0.8702
```

Now hyperparameter tuning:

```
# Uncomment to load pretrained weights
# net.load_state_dict(torch.load("net_after_pruning.pt"))
# Comment if you have loaded pretrained weights
finetune_after_prune(net, epochs=30, batch_size=64, lr=0.01, reg=0.00001)
```

```
[12] # Load the best weight paramters
     net.load_state_dict(torch.load("net_after_pruning.pt"))
     test(net)

     Files already downloaded and verified
     Test Loss=0.3694, Test accuracy=0.8963
     0.8963
```

```
46              Linear          2560
Total nonzero parameters: 1338572
Total parameters: 3811680
Total sparsity: 0.648824
-------------------------------
```

Now doing quantization with 4 bits

```
[14] centers = quantize_whole_model(net, bits=4)
     np.save("codebook_vgg16.npy", centers)
```

```
[15] test(net)
```

```
Files already downloaded and verified
Test Loss=0.3812, Test accuracy=0.8950
0.895
```

Accuracy of 89.5 is achieved.

Now after huffman coding:

```
weighted_average = total_weighted_storage / total_para

print("Weighted Average Storage: %.2f" %weighted_avera
```

```
Weighted Average Storage: 2.20
```

Huffman length is 2.2
Original length is 4

Calculating ratio by above data:

(Nonzero parameters/total parameters) * (quant bits/32) * (huffman length/original length)

1338572/3811680 * 4/32 * 2.2/4 = 0.0241

Compression rate is 1/0.0241 = 41.49

Hence accuracy of 89.5 and compression rate greater than 40 is achieved

Now to achieve accuracy more than 89.5 we did the third round of iterative pruning.

Third round of iterative pruning:

Iteration 1:

Started with s=0.5

```
[9] # Test accuracy before fine-tuning
    prune(net, method='std', q=45.0, s=0.5)
    test(net)

    Files already downloaded and verified
    Test Loss=0.3427, Test accuracy=0.8970
    0.897
```

Then hyperparameter tuning:

```
[10] # Uncomment to load pretrained weights
     # net.load_state_dict(torch.load("net_after_pruning.pt"))
     # Comment if you have loaded pretrained weights
     finetune_after_prune(net, epochs=30, batch_size=128, lr=0.001, reg=0.00001)
```

The change here is batch size is increased from 64 to 128 and learning rate changed to 0.001

```
[13] # Load the best weight paramters
     net.load_state_dict(torch.load("net_after_pruning.pt"))
     test(net)

     Files already downloaded and verified
     Test Loss=0.3022, Test accuracy=0.9107
     0.9107
```

```
46                  Linear              2560
Total nonzero parameters: 1864351
Total parameters: 3811680
Total sparsity: 0.510885
-------------------------------
```

Iteration 2:
Changed s to 0.75

```python
[ ]  # Test accuracy before fine-tuning
     prune(net, method='std', q=45.0, s=0.75)
     test(net)
```

```
Files already downloaded and verified
Test Loss=0.4706, Test accuracy=0.8702
0.8702
```

Now hyperparameter tuning:

```python
15m [16]  # Uncomment to load pretrained weights
          # net.load_state_dict(torch.load("net_after_pruning.pt"))
          # Comment if you have loaded pretrained weights
          finetune_after_prune(net, epochs=30, batch_size=128, lr=0.001, reg=0.00001)
```

```python
3s [17]  # Load the best weight paramters
         net.load_state_dict(torch.load("net_after_pruning.pt"))
         test(net)
```

```
Files already downloaded and verified
Test Loss=0.3601, Test accuracy=0.8965
0.8965
```

```
46                  Linear              2560
Total nonzero parameters: 1338261
Total parameters: 3811680
Total sparsity: 0.648905
-------------------------------
```

Now doing quantization with 4 bits

```
[19] centers = quantize_whole_model(net, bits=4)
     np.save("codebook_vgg16.npy", centers)
```

```
[20] test(net)
```

```
Files already downloaded and verified
Test Loss=0.3661, Test accuracy=0.8953
0.8953
```

We finally achieved accuracy of **89.53 i.e greater than 89.5**

After doing huffman coding:

```
Complete 6 layers for Huffman Coding...
Original storage for each parameter: 4.0000 bits
Average storage for each parameter after Huffman Coding: 2.5592 bits
Complete 7 layers for Huffman Coding...
Original storage for each parameter: 4.0000 bits
Average storage for each parameter after Huffman Coding: 2.4721 bits
Complete 8 layers for Huffman Coding...
Original storage for each parameter: 4.0000 bits
Average storage for each parameter after Huffman Coding: 2.3856 bits
Complete 9 layers for Huffman Coding...
Original storage for each parameter: 4.0000 bits
Average storage for each parameter after Huffman Coding: 2.3104 bits
Complete 10 layers for Huffman Coding...
Original storage for each parameter: 4.0000 bits
Average storage for each parameter after Huffman Coding: 2.1566 bits
Complete 11 layers for Huffman Coding...
Original storage for each parameter: 4.0000 bits
Average storage for each parameter after Huffman Coding: 2.0181 bits
Complete 12 layers for Huffman Coding...
Original storage for each parameter: 4.0000 bits
Average storage for each parameter after Huffman Coding: 1.7308 bits
Complete 13 layers for Huffman Coding...
Original storage for each parameter: 4.0000 bits
Average storage for each parameter after Huffman Coding: 2.4585 bits
Complete 14 layers for Huffman Coding...
Original storage for each parameter: 4.0000 bits
Average storage for each parameter after Huffman Coding: 2.2005 bits
Complete 15 layers for Huffman Coding...
Original storage for each parameter: 4.0000 bits
Average storage for each parameter after Huffman Coding: 1.5793 bits
Complete 16 layers for Huffman Coding...
```

```
Weighted Average Storage: 2.20
```

```
test(net)
```

```
Files already downloaded and verified
Test Loss=0.3661, Test accuracy=0.8953
0.8953
```

We got huffman length = 2.2
And it is observed that accuracy remained the same as **89.53.**

This is our final model.

Calculating compression ratio with above data:

Nonzero_params = 1338261

Total_params = 3811680

quant-Bits and original_length = 4

Huffman_length = 2.2

So the ratio is = (1338261/3811680) * (4/32) * (2.2/4) = 0.0241

So the compression rate is 1/0.0241 = **41.493**

Hence accuracy and compression rate is achieved.