Anthony DiSpirito
ECE 590 – Comp. Eng. M.L. and D.N.Ns
Dr. Yiran Chen
11/8/19

Lab 3:

Pruning, Quantization, and Huffman Coding to Compress DNNs

NOTE: Code and Documentation can also be found at github.com under my Github account name "axd465" and my repo "ECE590-CompEngML-DL-Lab3"

Link: **https://github.com/axd465/ECE590-CompEngML-DL-Lab3**

Assignment 1:

- For this problem, my goal was to achieve the best model possible, to give myself the best starting chance for the rest of the project. The only parameters we were able to modify were: learning rate, epochs, and the L2 regularization (which promotes many non-zero smaller magnitude weights). Note here that batch size was always kept consistent at 256. From previous experience, I knew that the most important factor for tuning would be the learning rate. I took a look at the train_util.py document to see the optimization algorithm was SGD with momentum (not Nesterov). From the slides and previous knowledge, I knew a good starting point would be the default value of initial learning rate = 0.01 and reg = 5e-4. From previous experience I knew that I should start with epochs = 20, to establish general trends. What followed were a serious of experiments and results recorded in the first table below, where the regularization was kept constant and the learning rate was modified. I chose what seemed like the best learning rate (between 0.06 and 0.05 at 0.055) and kept that constant when modifying the regularization.

TABLE 1: REG CONSTANT			
LR	REG	Final Test Acc	Final Test Loss
0.01	5e-4	0.8484	0.4450
0.05	5e-4	0.8656	0.3147
0.005	5e-4	0.8237	0.51
0.1	5e-4	0.8596	0.4204
0.075	5e-4	0.8640	0.4135
0.06	5e-4	0.8650	0.4032

TABLE 2: LR CONSTANT			
LR	REG	Final Test Acc	Final Test Loss
0.055	3e-4	0.8626	0.4171
0.055	10e-4	0.8717	0.3911
0.055	8e-4	0.8718	0.3908

- From these tests, I decided to try and reach above 90% with LR = 0.055 and REG = 8e-4 by training for 75 epochs as this combination seemed to converge smoothly without bottoming out too early. This resulted in a final testing accuracy of 0.9152 (as depicted below, with model summary). As this was well above 90%, I decided my tuning was done and moved on.

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

      Files already downloaded and verified
      Test Loss=0.3108, Test accuracy=0.9152

[13]: print("-----Summary before pruning-----")
      summary(net)
      print("-------------------------------")

      -----Summary before pruning-----
      Layer id        Type            Parameter       Non-zero parameter      Sparsity(\%)
      1               Convolutional   864             864                     0.000000
      2               BatchNorm       N/A             N/A                     N/A
      3               ReLU            N/A             N/A                     N/A
      4               Convolutional   9216            9216                    0.000000
      5               BatchNorm       N/A             N/A                     N/A
      6               ReLU            N/A             N/A                     N/A
      7               Convolutional   18432           18432                   0.000000
      8               BatchNorm       N/A             N/A                     N/A
      9               ReLU            N/A             N/A                     N/A
      10              Convolutional   36864           36864                   0.000000
      11              BatchNorm       N/A             N/A                     N/A
      12              ReLU            N/A             N/A                     N/A
      13              Convolutional   73728           73728                   0.000000
      14              BatchNorm       N/A             N/A                     N/A
      15              ReLU            N/A             N/A                     N/A
      16              Convolutional   147456          147456                  0.000000
      17              BatchNorm       N/A             N/A                     N/A
      18              ReLU            N/A             N/A                     N/A
      19              Convolutional   147456          147456                  0.000000
      20              BatchNorm       N/A             N/A                     N/A
      21              ReLU            N/A             N/A                     N/A
      22              Convolutional   294912          294912                  0.000000
      23              BatchNorm       N/A             N/A                     N/A
      24              ReLU            N/A             N/A                     N/A
      25              Convolutional   589824          589824                  0.000000
      26              BatchNorm       N/A             N/A                     N/A
      27              ReLU            N/A             N/A                     N/A
      28              Convolutional   589824          589824                  0.000000
      29              BatchNorm       N/A             N/A                     N/A
      30              ReLU            N/A             N/A                     N/A
      31              Convolutional   589824          589824                  0.000000
      32              BatchNorm       N/A             N/A                     N/A
      33              ReLU            N/A             N/A                     N/A
      34              Convolutional   589824          589824                  0.000000
      35              BatchNorm       N/A             N/A                     N/A
      36              ReLU            N/A             N/A                     N/A
      37              Convolutional   589824          589824                  0.000000
      38              BatchNorm       N/A             N/A                     N/A
      39              ReLU            N/A             N/A                     N/A
      40              Linear          65536           65536                   0.000000
      41              BatchNorm       N/A             N/A                     N/A
      42              ReLU            N/A             N/A                     N/A
      43              Linear          65536           65536                   0.000000
      44              BatchNorm       N/A             N/A                     N/A
      45              ReLU            N/A             N/A                     N/A
      46              Linear          2560            2560                    0.000000
      Total nonzero parameters: 3811680
      Total parameters: 3811680
      Total sparsity: 0.000000
      -------------------------------
```

Assignment 2:

a) See below code for the completed prune_by_percentage sections of the pruned_layers.py document (in both PruneLinear and PruneConv respectively).

```
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 = self.linear.weight.data.cpu().numpy()
    self.mask = abs(weight) >= np.percentile(abs(weight), q)
    self.linear.weight.data = torch.from_numpy(weight*self.mask).float().to(device)
    self.sparsity = 1 - np.sum(self.mask)/self.mask.size
```

```
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 = self.conv.weight.data.cpu().numpy()
    self.mask = abs(weight) >= np.percentile(abs(weight), q)
    self.conv.weight.data = torch.from_numpy(weight*self.mask).float().to(device)
    self.sparsity = 1 - np.sum(self.mask)/self.mask.size
```

b) See below code for completed prune_by_std sections of the pruned_layers.py document (in both PruneLinear and PruneConv respectively).

```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---------------------
    """
    weight = self.linear.weight.data.cpu().numpy()
    self.mask = abs(weight) >= np.std(weight)*s
    self.linear.weight.data = torch.from_numpy(weight*self.mask).float().to(device)
    self.sparsity = 1 - np.sum(self.mask)/self.mask.size
```

```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---------------------
    """
    weight = self.conv.weight.data.cpu().numpy()
    self.mask = abs(weight) >= np.std(weight)*s
    self.conv.weight.data = torch.from_numpy(weight*self.mask).float().to(device)
    self.sparsity = 1 - np.sum(self.mask)/self.mask.size
```

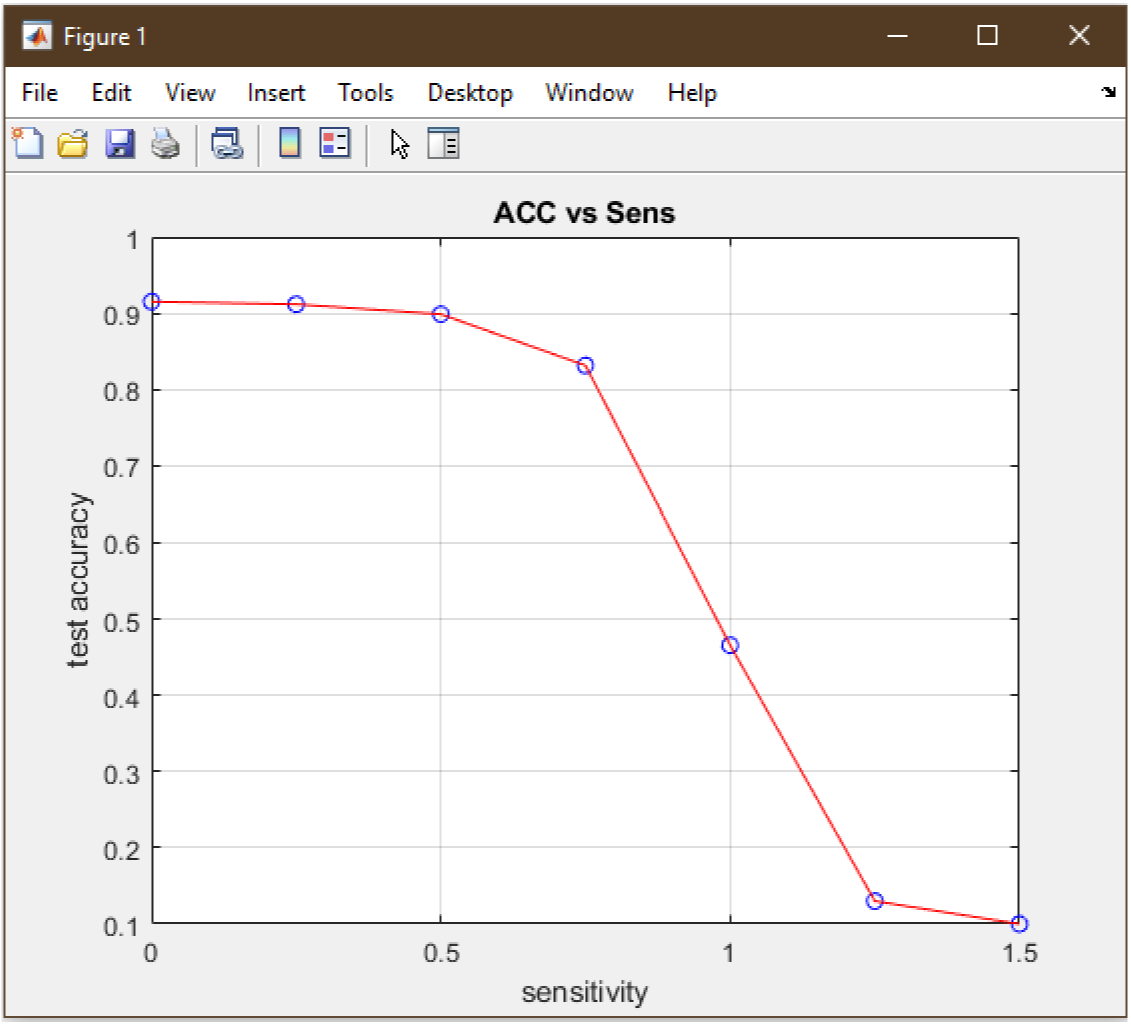
c) After pruning with the std method and s = 0.75, my test accuracy was 0.8314 (as shown below), which is down from my previous result of 0.9152. Below is also depicted the model summary, with a total sparsity of 0.668753. This drop in accuracy is likely due to the fact that some weights (67%) that previously held significance within the model have now been zeroed without retraining. The only reason the accuracy did not drop lower is because on average the larger magnitude weights (unpruned) are more important.

```
[14]: # Test accuracy before fine-tuning
      prune(net, method='std', q=45.0, s = 0.75)
      #prune(net, method='std', q=66.8753, s = 1.5) # 1.25
      test(net)

      Files already downloaded and verified
      Test Loss=0.5618, Test accuracy=0.8314
```

```
-----Summary After pruning-----
Layer id        Type            Parameter       Non-zero parameter      Sparsity(\%)
1               Convolutional   864             244                     0.717593
2               BatchNorm       N/A             N/A                     N/A
3               ReLU            N/A             N/A                     N/A
4               Convolutional   9216            2449                    0.734266
5               BatchNorm       N/A             N/A                     N/A
6               ReLU            N/A             N/A                     N/A
7               Convolutional   18432           7572                    0.589193
8               BatchNorm       N/A             N/A                     N/A
9               ReLU            N/A             N/A                     N/A
10              Convolutional   36864           16383                   0.555583
11              BatchNorm       N/A             N/A                     N/A
12              ReLU            N/A             N/A                     N/A
13              Convolutional   73728           32602                   0.557807
14              BatchNorm       N/A             N/A                     N/A
15              ReLU            N/A             N/A                     N/A
16              Convolutional   147456          66148                   0.551405
17              BatchNorm       N/A             N/A                     N/A
18              ReLU            N/A             N/A                     N/A
19              Convolutional   147456          64546                   0.562269
20              BatchNorm       N/A             N/A                     N/A
21              ReLU            N/A             N/A                     N/A
22              Convolutional   294912          117636                  0.601115
23              BatchNorm       N/A             N/A                     N/A
24              ReLU            N/A             N/A                     N/A
25              Convolutional   589824          219740                  0.627448
26              BatchNorm       N/A             N/A                     N/A
27              ReLU            N/A             N/A                     N/A
28              Convolutional   589824          215260                  0.635044
29              BatchNorm       N/A             N/A                     N/A
30              ReLU            N/A             N/A                     N/A
31              Convolutional   589824          189024                  0.679525
32              BatchNorm       N/A             N/A                     N/A
33              ReLU            N/A             N/A                     N/A
34              Convolutional   589824          165859                  0.718799
35              BatchNorm       N/A             N/A                     N/A
36              ReLU            N/A             N/A                     N/A
37              Convolutional   589824          112748                  0.808845
38              BatchNorm       N/A             N/A                     N/A
39              ReLU            N/A             N/A                     N/A
40              Linear          65536           28576                   0.563965
41              BatchNorm       N/A             N/A                     N/A
42              ReLU            N/A             N/A                     N/A
43              Linear          65536           23331                   0.643997
44              BatchNorm       N/A             N/A                     N/A
45              ReLU            N/A             N/A                     N/A
46              Linear          2560            490                     0.808594
Total nonzero parameters: 1262608
Total parameters: 3811680
Total sparsity: 0.668753
-------------------------------
```

d) Upon testing multiple values for the sensitivity I found the below relationship to test accuracy. At first, the change in sensitivity has little effect, then there is a critical point of dramatic accuracy drop, followed by the bottoming out of accuracy at the near random point. It seems there is a tolerable range of sparsity before the model needs to be retrained/fine-tuned (about 0.75).



e) As seen below, after changing my method to percentage, I was able to achieve a comparable sparsity to the std method at q = 66.8753 (just inputted the final sparsity desired so they were equal). However, as seen below, the test accuracy was just 0.6816 as compared to the std method which had a test accuracy of 0.8314 with the same total sparsity. This makes sense, as the std method is able to vary by layer and is tied to the variance of the normal weight distribution, as opposed to having a hard threshold imposed on all the weight distributions (see below where every layer had nearly the same sparsity). This lack of variability layer by layer, distribution by distribution caused more of the important weights to be pruned away.

-----Summary After pruning-----

Layer id	Type	Parameter	Non-zero parameter	Sparsity(\
1	Convolutional	864	286	0.668981
2	BatchNorm	N/A	N/A	N/A
3	ReLU	N/A	N/A	N/A
4	Convolutional	9216	3053	0.668728
5	BatchNorm	N/A	N/A	N/A
6	ReLU	N/A	N/A	N/A
7	Convolutional	18432	6106	0.668728
8	BatchNorm	N/A	N/A	N/A
9	ReLU	N/A	N/A	N/A
10	Convolutional	36864	12211	0.668755
11	BatchNorm	N/A	N/A	N/A
12	ReLU	N/A	N/A	N/A
13	Convolutional	73728	24422	0.668755
14	BatchNorm	N/A	N/A	N/A
15	ReLU	N/A	N/A	N/A
16	Convolutional	147456	48845	0.668749
17	BatchNorm	N/A	N/A	N/A
18	ReLU	N/A	N/A	N/A
19	Convolutional	147456	48845	0.668749
20	BatchNorm	N/A	N/A	N/A
21	ReLU	N/A	N/A	N/A
22	Convolutional	294912	97689	0.668752
23	BatchNorm	N/A	N/A	N/A
24	ReLU	N/A	N/A	N/A
25	Convolutional	589824	195378	0.668752
26	BatchNorm	N/A	N/A	N/A
27	ReLU	N/A	N/A	N/A
28	Convolutional	589824	195378	0.668752
29	BatchNorm	N/A	N/A	N/A
30	ReLU	N/A	N/A	N/A
31	Convolutional	589824	195378	0.668752
32	BatchNorm	N/A	N/A	N/A
33	ReLU	N/A	N/A	N/A
34	Convolutional	589824	195378	0.668752
35	BatchNorm	N/A	N/A	N/A
36	ReLU	N/A	N/A	N/A
37	Convolutional	589824	195378	0.668752
38	BatchNorm	N/A	N/A	N/A
39	ReLU	N/A	N/A	N/A
40	Linear	65536	21709	0.668747
41	BatchNorm	N/A	N/A	N/A
42	ReLU	N/A	N/A	N/A
43	Linear	65536	21709	0.668747
44	BatchNorm	N/A	N/A	N/A
45	ReLU	N/A	N/A	N/A
46	Linear	2560	848	0.668750

Total nonzero parameters: 1262613
Total parameters: 3811680
Total sparsity: 0.668752

<u>Assignment 3:</u>

a) See below as well as the attached code how I completed the train_util.py document to zero out the pruned gradients (so they remain at zero). After fine-tuning/retraining with the stock hyperparameters (which seemed reasonable – smaller initial learning rate and smaller L2 regularization) and s = 1.25 for the std method (total sparsity of 0.8424) for 10 epochs, I got a test accuracy of 0.8984. This was well above the 0.13 I was getting without fine-tuning. I would say this method could probably recover most of the accuracy lost in pruning.

```
for batch_idx, (inputs, targets) in enumerate(trainloader):
    inputs, targets = inputs.to(device), targets.to(device)
    optimizer.zero_grad()
    outputs = net(inputs)
    loss = criterion(outputs, targets)
    loss.backward()

    # before optimizer.step(), manipulate the gradient
    """
    Zero the gradients of the pruned variables.
    -----------------------Your Code-------------------------
    """
    for group in net.parameters():
        group.grad.data[group.data == 0] = 0
    '''
    -----------------------------------------------------------
    '''
    optimizer.step()
    train_loss += loss.item()
    _, predicted = outputs.max(1)
    total += targets.size(0)
    correct += predicted.eq(targets).sum().item()
    global_steps += 1
```

```
Epoch: 9
[Step=1776]     Loss=0.1574     acc=0.9505      129.7 examples/second
[Step=1792]     Loss=0.1540     acc=0.9516      5025.8 examples/second
[Step=1808]     Loss=0.1563     acc=0.9503      5031.9 examples/second
[Step=1824]     Loss=0.1576     acc=0.9500      5038.1 examples/second
[Step=1840]     Loss=0.1577     acc=0.9492      5031.9 examples/second
[Step=1856]     Loss=0.1564     acc=0.9494      5031.9 examples/second
[Step=1872]     Loss=0.1565     acc=0.9494      5022.6 examples/second
[Step=1888]     Loss=0.1554     acc=0.9492      5025.8 examples/second
[Step=1904]     Loss=0.1564     acc=0.9489      4887.8 examples/second
[Step=1920]     Loss=0.1557     acc=0.9489      5025.8 examples/second
[Step=1936]     Loss=0.1557     acc=0.9490      5069.3 examples/second
[Step=1952]     Loss=0.1565     acc=0.9488      5113.5 examples/second
Test Loss=0.3400, Test acc=0.8984
Saving...
```

Assignment 4:

a) Below is how I performed the quantization procedure in the code (PrunedConv and linear are nearly identical). After quantizing to 5 bits via weight sharing on my aforementioned 0.84 sparsity model, my test accuracy was 0.8941 (as seen below). The quantization of the sparse weights seems to have very little impact on the test accuracy of the model when quantizing to 5 bits. This is likely because the pruned weight distribution already has some decreased variability (kind of like the bottleneck effect in biological populations). Therefore, it should be relatively straightforward to group these weights to similar values when using $2^5 = 32$ clusters. This is almost like a sampling/discretization procedure on the weight distribution, with the bottleneck effect probably being the best analogy.

```
"""
Apply quantization for the PrunedConv layer.
--------------Your Code---------------------
"""
weights = m.conv.weight.data.cpu().numpy()
original_shape = weights.shape
non_zero_weights = weights[np.nonzero(weights)]
weights = weights.reshape(-1,1)
non_zero_weights = non_zero_weights.reshape(-1,1)
kmean = KMeans(n_clusters = 2**bits,
               init='k-means++',
               n_init=25,
               max_iter=50)
kmean.fit(non_zero_weights)
cluster_centers.append(kmean.cluster_centers_)
#print(cluster_centers[layer_ind])
#print(weights.shape)
quant_weights = np.zeros(weights.shape, dtype=np.float32)
for i in range(len(weights)):
    weight = weights[i]
    if weight != 0:
        label = kmean.predict(weight.reshape(1, -1))
        quant_weights[i] = cluster_centers[layer_ind][label]
#if layer_ind == 0:
#    print(quant_weights)
m.conv.weight.data = torch.from_numpy(quant_weights.reshape(original_shape)).float().to(device)

layer_ind += 1
print("Complete %d layers quantization..." %layer_ind)
```

```
[46]: test(net)

Files already downloaded and verified
Test Loss=0.3439, Test accuracy=0.8941
```

b) After playing around with the number of quantization bits, I found the following results for different quantization amounts [bits = 2, acc = 0.7308], [bits = 3, acc = 0.8675], and [bits = 4, acc = 0.8912]. The optimal bit here seems to be bits = 3, which is about 3% less accuracy than the 5 bit quantization with a quarter of the bits. At 2 bits there is a steep drop-off, so 3 bits seems optimal (especially because this is without retraining/ quantization).

Assignment 5:

a) The Huffman coding reduces the memory footprints of DNNs because it can compress the number of bits necessary to store the values in the weight matrix. It does this by encoding the weights based on the frequency of occurrence. Those weights with higher frequency get shorter encodings. For example, let's say you have a matrix filled entirely with a two or three large numbers. Every time those numbers occur as weights, you would normally need to store that large number again. This can get very expensive. Using Huffman encoding, you could create a Huffman table (relatively low cost for this example) and simply store these weights as a much smaller footprint encoding. Thus reducing the total memory consumption dramatically.
b) Below is how I implemented Huffman coding via a Huffman node object and the creation of a tree. I also thought of a clever idea that appended to the leaf encodings every time overarching branches merged. My Huffman object stores each node in the Huffman tree. The leaves have an attribute (leaf=True), that is used when appending an addition to the encoding. Each node has a left and right branch, where these values are set to None for the leaves. I then use the Huffman coding algorithm to remove leaves from a queue (those with the least two frequencies), and form a new Huffman node out of that. The differentiation between leaves and other such nodes is made explicit in the instantiation. Non-leaves have pointers to other Huffman objects assigned to left and right and their frequencies are determined by adding the left and right frequencies. Whenever a node combination occurs, I recursively go down the right and left node pointers to add the appropriate encoding addition to the leaves. After node combination, the top node object is placed back into the queue (which is then resorted by frequency). After the queue only has the root left, I access the encodings via a list of pointers to the original leaf objects.

```python
class HuffmanNode():
    def __init__(self, key = '<!$>_ANTHONY_<$!>', freq = 0, right = None, left = None, leaf = False):
        if leaf:
            self.key = key
            self.freq = freq
            self.right = None
            self.left = None
            self.leaf = True
            self.encode = ''
        else:
            self.key = key
            self.freq = right.freq + left.freq
            self.right = right
            self.left = left
            self.leaf = False
            right.add_encode('1')
            left.add_encode('0')
        return
    def add_encode(self, addition):
        if self.leaf == False:
            if self.left == None:
                self.right.add_encode(addition)
            elif self.right == None:
                self.left.add_encode(addition)
            else:
                self.right.add_encode(addition)
                self.left.add_encode(addition)
            if self.right == None and self.left == None:
                print('Error: Recursively iterating on a leaf')
        else:
            self.encode = addition + self.encode
        return
```

```python
def convert_freq_dict_to_encodings(freq):
    original_freq = freq.copy() # Just in Case I want to check something later

    leaf_list = []
    for centroid, frequency in freq.items():
        leaf_list.append(HuffmanNode(key = centroid,
                                     freq = frequency,
                                     leaf = True))
    tree = []
    tree.extend(leaf_list)

    MaxIter = 500
    iter = 0
    not_root = True

    # Forming Huffman Tree and Setting Encoding
    while not_root and iter < MaxIter:
        least_freq_item = tree.pop(-1)
        second_least_freq_item = tree.pop(-1)
        tree.append(HuffmanNode(key = 'Branch ' + str(iter),
                                right = second_least_freq_item,
                                left = least_freq_item))
        iter+=1
        not_root = len(tree) > 1
        if not_root:
            if tree[-1].freq > tree[-2].freq:
                tree = sorted(tree, key=lambda node: node.freq, reverse = True)
    encodings = {}
    for leaf in leaf_list:
        encodings[leaf.key] = leaf.encode
    return encodings
```

```
def _huffman_coding_per_layer(weight, centers):
    """
    Huffman coding for each layer
    :param weight: weight parameter of the current layer.
    :param centers: KMeans centroids in the quantization codebook of the current weight layer.
    :return:
            'encodings': Encoding map mapping each weight parameter to its Huffman coding.
            'frequency': Frequency map mapping each weight parameter to the total number of its appearance.
            'encodings' should be in this format:
            {"0.24315": '0', "-0.2145": "100", "1.1234e-5": "101", ...
            }
            'frequency' should be in this format:
            {"0.25235": 100, "-0.2145": 42, "1.1234e-5": 36, ...
            }
            'encodings' and 'frequency' does not need to be ordered in any way.
    """
    """
    Generate Huffman Coding and Frequency Map according to incoming weights and centers (KMeans centroids).
    --------------Your Code---------------------
    """

    # TECHNICALLY WE DO NOT NEED THE CENTERS ARRAY AS WE GET IT FROM THE QUANTIZED WEIGHTS ARRAY VIA MY METHOD

    non_zero_weights = list(map(str, weight[np.nonzero(weight)]))# creates string array of non-zero weight values
    ordered = Counter(non_zero_weights)
    # creates dictionary of centroids in decending order of frequency
    frequency = {}
    for item in ordered.most_common(len(ordered)):
        key = item[0]
        value = item[1]
        frequency[key] = value
    encodings = convert_freq_dict_to_encodings(frequency) # converts freq dict to centroid encodings
    return encodings, frequency
```

c) Below is the quantitative analysis of the additional memory reduction with the usage of Huffman coding and the calculated average encoding length (per layer as a weighted average). The calculation below uses the 84% sparsity, 4 bit quantized model.

- <u>Layer 1:</u> (3.8308 bits per param * 130 param) = 498.004 bits
- <u>Layer 2:</u> (3.6427 * 1324) = 4822.9348 bits
- <u>Layer 3:</u> (3.6256 * 3411) = 12366.9 bits
- <u>Layer 4:</u> (3.6545 * 7559) = 27624.4 bits
- <u>Layer 5:</u> (3.6771 * 14994) = 55134.4 bits
- <u>Layer 6:</u> (3.6570 * 30509) = 111571.4 bits
- <u>Layer 7:</u> (3.6576 * 29566) = 108140.6 bits
- <u>Layer 8:</u> (3.5781 * 54161) = 193793.5 bits
- <u>Layer 9:</u> (3.4396 * 101689) = 349769 bits
- <u>Layer 10:</u> (3.4553 * 97833) = 338042 bits
- <u>Layer 11:</u> (3.4398 * 90486) = 311253 bits

- Layer 12: (3.5039 * 84444) = 295883.3 bits
- Layer 13: (3.5311 * 61583) = 217455.7 bits
- Layer 14: (3.5598 * 12959) = 46131.4 bits
- Layer 15: (3.6947 * 9750) = 36032.3 bits
- Layer 16: (3.8077 * 286) = 1089 bits
- Average Encoding Length = sum(Layer Lengths)/total_num_param = 2103607.84/600684 = 3.502 bits
- Memory Reduction (%): = 1 – average_Huffman_length/original_length x 100% = 1 – 3.502/4 x 100% = 12.45% reduction in memory

-----Summary After pruning-----

Layer id	Type	Parameter	Non-zero parameter	Sparsity(\%)
1	Convolutional	864	130	0.849537
2	BatchNorm	N/A	N/A	N/A
3	ReLU	N/A	N/A	N/A
4	Convolutional	9216	1324	0.856337
5	BatchNorm	N/A	N/A	N/A
6	ReLU	N/A	N/A	N/A
7	Convolutional	18432	3411	0.814941
8	BatchNorm	N/A	N/A	N/A
9	ReLU	N/A	N/A	N/A
10	Convolutional	36864	7559	0.794949
11	BatchNorm	N/A	N/A	N/A
12	ReLU	N/A	N/A	N/A
13	Convolutional	73728	14994	0.796631
14	BatchNorm	N/A	N/A	N/A
15	ReLU	N/A	N/A	N/A
16	Convolutional	147456	30509	0.793098
17	BatchNorm	N/A	N/A	N/A
18	ReLU	N/A	N/A	N/A
19	Convolutional	147456	29566	0.799493
20	BatchNorm	N/A	N/A	N/A
21	ReLU	N/A	N/A	N/A
22	Convolutional	294912	54161	0.816349
23	BatchNorm	N/A	N/A	N/A
24	ReLU	N/A	N/A	N/A
25	Convolutional	589824	101689	0.827594
26	BatchNorm	N/A	N/A	N/A
27	ReLU	N/A	N/A	N/A
28	Convolutional	589824	97833	0.834132
29	BatchNorm	N/A	N/A	N/A
30	ReLU	N/A	N/A	N/A
31	Convolutional	589824	90486	0.846588
32	BatchNorm	N/A	N/A	N/A
33	ReLU	N/A	N/A	N/A
34	Convolutional	589824	84444	0.856832
35	BatchNorm	N/A	N/A	N/A
36	ReLU	N/A	N/A	N/A
37	Convolutional	589824	61583	0.895591
38	BatchNorm	N/A	N/A	N/A
39	ReLU	N/A	N/A	N/A
40	Linear	65536	12959	0.802261
41	BatchNorm	N/A	N/A	N/A
42	ReLU	N/A	N/A	N/A
43	Linear	65536	9750	0.851227
44	BatchNorm	N/A	N/A	N/A
45	ReLU	N/A	N/A	N/A
46	Linear	2560	286	0.888281

Total nonzero parameters: 600684
Total parameters: 3811680
Total sparsity: 0.842410

```
Original storage for each parameter: 4.0000 bits
Average storage for each parameter after Huffman Coding: 3.8308 bits
Complete 1 layers for Huffman Coding...
Original storage for each parameter: 4.0000 bits
Average storage for each parameter after Huffman Coding: 3.6427 bits
Complete 2 layers for Huffman Coding...
Original storage for each parameter: 4.0000 bits
Average storage for each parameter after Huffman Coding: 3.6256 bits
Complete 3 layers for Huffman Coding...
Original storage for each parameter: 4.0000 bits
Average storage for each parameter after Huffman Coding: 3.6545 bits
Complete 4 layers for Huffman Coding...
Original storage for each parameter: 4.0000 bits
Average storage for each parameter after Huffman Coding: 3.6771 bits
Complete 5 layers for Huffman Coding...
Original storage for each parameter: 4.0000 bits
Average storage for each parameter after Huffman Coding: 3.6570 bits
Complete 6 layers for Huffman Coding...
Original storage for each parameter: 4.0000 bits
Average storage for each parameter after Huffman Coding: 3.6576 bits
Complete 7 layers for Huffman Coding...
Original storage for each parameter: 4.0000 bits
Average storage for each parameter after Huffman Coding: 3.5781 bits
Complete 8 layers for Huffman Coding...
Original storage for each parameter: 4.0000 bits
Average storage for each parameter after Huffman Coding: 3.4396 bits
Complete 9 layers for Huffman Coding...
Original storage for each parameter: 4.0000 bits
Average storage for each parameter after Huffman Coding: 3.4553 bits
Complete 10 layers for Huffman Coding...
Original storage for each parameter: 4.0000 bits
Average storage for each parameter after Huffman Coding: 3.4398 bits
Complete 11 layers for Huffman Coding...
Original storage for each parameter: 4.0000 bits
Average storage for each parameter after Huffman Coding: 3.5039 bits
Complete 12 layers for Huffman Coding...
Original storage for each parameter: 4.0000 bits
Average storage for each parameter after Huffman Coding: 3.5311 bits
Complete 13 layers for Huffman Coding...
Original storage for each parameter: 4.0000 bits
Average storage for each parameter after Huffman Coding: 3.5598 bits
Complete 14 layers for Huffman Coding...
Original storage for each parameter: 4.0000 bits
Average storage for each parameter after Huffman Coding: 3.6947 bits
Complete 15 layers for Huffman Coding...
Original storage for each parameter: 4.0000 bits
Average storage for each parameter after Huffman Coding: 3.8077 bits
Complete 16 layers for Huffman Coding...
```

Assignment 6:

- See Below for depiction of accuracy at various stages, as well as parameters used and final accuracy after compression (final accuracy reducing compression step is the quantization).
- In order to achieve this final accuracy and compression ratio, I implemented a couple tricks in addition to those already discussed in the lab. I performed iterative pruning, where I trained the full network to 0.9152 accuracy, pruned with s = 0.75 (my optimal starting point found before), then I retrained using the fine-tuning procedure (7 epochs), pruned the network again at s = 1.0, retrained using fine-tuning (7 epochs), and pruned a final time at s = 1.25 to achieve the sparsity I was aiming for (above 85%) and did the fine-tuning again (15 epochs). My thought here is that although I could have started at 1.25 and fine-tuned, that this gradual iterative technique would give the probability distribution more time to shift and thus overall keep more of the important weights. My theory held true (as seen below), because my final accuracy after pruning was 0.8995. Then I performed some testing on the quantization. In the quantization, I quantized at 5 bits, then used my function finetune_after_quantization() which is nearly the same as the pruning fine-tune function except it saved the net to a file "net_after_quantization.pt." I performed one iteration of iterative quantization where I quantized to 5 bits, fine-tuned, and re-quantized (as my fine-tuning function did not zero the gradients between the centroids). By doing this I was able to achieve an accuracy of 0.8972. As this is rather close to the cut-off, I decided to not quantize further and see if I can achieve the specified compression ratio with this level of quantization. I then performed Huffman coding, performed the compression calculation you see below, and found I had met/exceeded the specified criteria so I decided to stop.
- Note, to check that the quantization occurred correctly, I created some code to access and depict the weight information of one layer. I then used a histogram with many more bins than quantized centroids. The discretization of the weight distribution told me that weights were being correctly shared (as seen below).
- Note, the final outputs are all saved within the DeepCompression.ipynb notebook as well

- Below is the calculation for the Huffman length:
  - Layer 1: (4.7788 * 113) = 540 bits
  - Layer 2: (4.6694 * 1107) = 5169 bits
  - Layer 3: (4.6401 * 2856) = 13252.1 bits
  - Layer 4: (4.6302 * 6190) = 28660.9 bits
  - Layer 5: (4.6285 * 12272) = 56801 bits
  - Layer 6: (4.6752 * 24806) = 115973 bits
  - Layer 7: (4.6237 * 24098) = 111421.9 bits
  - Layer 8: (4.5612 * 44970) = 205117.2 bits
  - Layer 9: (4.4531 * 84209) = 374991.1 bits
  - Layer 10: (4.4547 * 81276) = 362060.2 bits
  - Layer 11: (4.4317 * 75491) = 334553.5 bits
  - Layer 12: (4.4503 * 72493) = 322615.6 bits
  - Layer 13: (4.4675 * 54521) = 243572.6 bits
  - Layer 14: (4.5048 * 10625) = 47863.5 bits
  - Layer 15: (4.6289 * 8450) = 39114.2 bits
  - Layer 16: (4.9011 * 283) = 1387 bits
  - Average Encoding Length = sum(Layer Lengths)/total_num_param = 2263092.8/503760 = 4.49 bits
  - Memory Reduction (%): = 1 – average_Huffman_length/original_length x 100% = 1 – 4.49/5 x 100% = 10.2% reduction in memory needed
- **Therefore, my final accuracy is 89.72>89.5 and my final compression rate is equal to 503760/3811680 x 5/32 x 4.49/5 = 0.01854. This constitutes a compression ratio of 1/0.01854 = 53.93x > 40x = 1/0.025. I have therefore, by my calculations met all the requirements and exceeded them by a fair margin.**

Full-precision model training

[2]:
```
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.
INITIAL_LR = 0.055
REG = 8e-4
EPOCHS = 75 #20
BATCH_SIZE = 256
train(net, epochs=EPOCHS, batch_size=BATCH_SIZE, lr=INITIAL_LR, reg=REG)
```

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

```
Files already downloaded and verified
Test Loss=0.3108, Test accuracy=0.9152
```

[ ]:
```
# Test accuracy before fine-tuning
prune(net, method='std', q=66.8753, s = 1.25) # 1.25
test(net)
```

[ ]:
```
# 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)
finetune_after_prune(net, epochs=10, batch_size=256, lr=0.001, reg=5e-5)
```

[5]:
```
# Load the best weight parameters
net.load_state_dict(torch.load("net_after_pruning.pt"))
test(net)
```

```
Files already downloaded and verified
Test Loss=0.3480, Test accuracy=0.8995
```

```
-----Summary After pruning-----
Layer id        Type            Parameter       Non-zero parameter      Sparsity(\%)
1               Convolutional   864             113                     0.869213
2               BatchNorm       N/A             N/A                     N/A
3               ReLU            N/A             N/A                     N/A
4               Convolutional   9216            1107                    0.879883
5               BatchNorm       N/A             N/A                     N/A
6               ReLU            N/A             N/A                     N/A
7               Convolutional   18432           2856                    0.845052
8               BatchNorm       N/A             N/A                     N/A
9               ReLU            N/A             N/A                     N/A
10              Convolutional   36864           6190                    0.832086
11              BatchNorm       N/A             N/A                     N/A
12              ReLU            N/A             N/A                     N/A
13              Convolutional   73728           12272                   0.833550
14              BatchNorm       N/A             N/A                     N/A
15              ReLU            N/A             N/A                     N/A
16              Convolutional   147456          24806                   0.831774
17              BatchNorm       N/A             N/A                     N/A
18              ReLU            N/A             N/A                     N/A
19              Convolutional   147456          24098                   0.836575
20              BatchNorm       N/A             N/A                     N/A
21              ReLU            N/A             N/A                     N/A
22              Convolutional   294912          44970                   0.847514
23              BatchNorm       N/A             N/A                     N/A
24              ReLU            N/A             N/A                     N/A
25              Convolutional   589824          84209                   0.857230
26              BatchNorm       N/A             N/A                     N/A
27              ReLU            N/A             N/A                     N/A
28              Convolutional   589824          81276                   0.862203
29              BatchNorm       N/A             N/A                     N/A
30              ReLU            N/A             N/A                     N/A
31              Convolutional   589824          75491                   0.872011
32              BatchNorm       N/A             N/A                     N/A
33              ReLU            N/A             N/A                     N/A
34              Convolutional   589824          72493                   0.877094
35              BatchNorm       N/A             N/A                     N/A
36              ReLU            N/A             N/A                     N/A
37              Convolutional   589824          54521                   0.907564
38              BatchNorm       N/A             N/A                     N/A
39              ReLU            N/A             N/A                     N/A
40              Linear          65536           10625                   0.837875
41              BatchNorm       N/A             N/A                     N/A
42              ReLU            N/A             N/A                     N/A
43              Linear          65536           8450                    0.871063
44              BatchNorm       N/A             N/A                     N/A
45              ReLU            N/A             N/A                     N/A
46              Linear          2560            283                     0.889453
Total nonzero parameters: 503760
Total parameters: 3811680
Total sparsity: 0.867838
-------------------------------
```

```
[9]: net.load_state_dict(torch.load("net_after_pruning.pt"))
     centers = quantize_whole_model(net, bits=5)

     # np.save("codebook_vgg16.npy", centers)

     # print("Saving...")
     # torch.save(net.state_dict(), "net_after_quantization.pt")

     test(net)
```

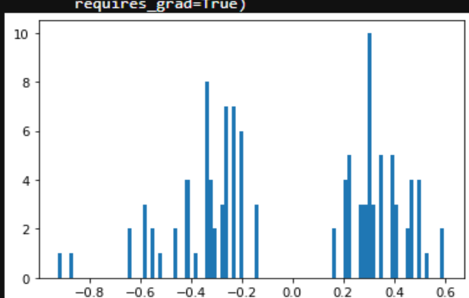
```
Complete 1 layers quantization...
Complete 2 layers quantization...
Complete 3 layers quantization...
Complete 4 layers quantization...
Complete 5 layers quantization...
Complete 6 layers quantization...
Complete 7 layers quantization...
Complete 8 layers quantization...
Complete 9 layers quantization...
Complete 10 layers quantization...
Complete 11 layers quantization...
Complete 12 layers quantization...
Complete 13 layers quantization...
Complete 14 layers quantization...
Complete 15 layers quantization...
Complete 16 layers quantization...
Files already downloaded and verified
Test Loss=0.3525, Test accuracy=0.8972
```

```
[7]: # Uncomment to load pretrained weights
     net.load_state_dict(torch.load("net_after_quantization.pt"))
     # Comment if you have loaded pretrained weights
     #finetune_after_quantization(net, epochs=5, batch_size=256, lr=0.001, reg=5e-5)
     test(net)
```

```
Files already downloaded and verified
Test Loss=0.3525, Test accuracy=0.8972
```

```
# Visualize Weight Distributions
params = list(net.parameters())
print(params[:][1])
layer1weights = np.array(params[0].data.cpu().numpy()).flatten()[np.nonzero(np.array(params[0].data.cpu().numpy()).flatten())]
plt.hist(layer1weights, bins = 100)
plt.show()
```

```
Parameter containing:
tensor([0.3798, 0.3615, 0.1040, 0.2815, 0.4390, 0.1532, 0.5035, 0.1199, 0.0064,
        0.2706, 0.1908, 0.2142, 0.2110, 0.0194, 0.3515, 0.3837, 0.0053, 0.3496,
        0.1642, 0.2042, 0.2816, 0.1916, 0.2265, 0.5013, 0.0033, 0.3221, 0.0905,
        0.4349, 0.3615, 0.3752, 0.3417, 0.3994], device='cuda:0',
       requires_grad=True)
```



```
Original storage for each parameter: 5.0000 bits
Average storage for each parameter after Huffman Coding: 4.7788 bits
Complete 1 layers for Huffman Coding...
Original storage for each parameter: 5.0000 bits
Average storage for each parameter after Huffman Coding: 4.6694 bits
Complete 2 layers for Huffman Coding...
Original storage for each parameter: 5.0000 bits
Average storage for each parameter after Huffman Coding: 4.6401 bits
Complete 3 layers for Huffman Coding...
Original storage for each parameter: 5.0000 bits
Average storage for each parameter after Huffman Coding: 4.6302 bits
Complete 4 layers for Huffman Coding...
Original storage for each parameter: 5.0000 bits
Average storage for each parameter after Huffman Coding: 4.6285 bits
Complete 5 layers for Huffman Coding...
Original storage for each parameter: 5.0000 bits
Average storage for each parameter after Huffman Coding: 4.6752 bits
Complete 6 layers for Huffman Coding...
Original storage for each parameter: 5.0000 bits
Average storage for each parameter after Huffman Coding: 4.6237 bits
Complete 7 layers for Huffman Coding...
Original storage for each parameter: 5.0000 bits
Average storage for each parameter after Huffman Coding: 4.5612 bits
Complete 8 layers for Huffman Coding...
Original storage for each parameter: 5.0000 bits
Average storage for each parameter after Huffman Coding: 4.4531 bits
Complete 9 layers for Huffman Coding...
Original storage for each parameter: 5.0000 bits
Average storage for each parameter after Huffman Coding: 4.4547 bits
Complete 10 layers for Huffman Coding...
Original storage for each parameter: 5.0000 bits
Average storage for each parameter after Huffman Coding: 4.4317 bits
Complete 11 layers for Huffman Coding...
Original storage for each parameter: 5.0000 bits
Average storage for each parameter after Huffman Coding: 4.4503 bits
Complete 12 layers for Huffman Coding...
Original storage for each parameter: 5.0000 bits
Average storage for each parameter after Huffman Coding: 4.4675 bits
Complete 13 layers for Huffman Coding...
Original storage for each parameter: 5.0000 bits
Average storage for each parameter after Huffman Coding: 4.5048 bits
Complete 14 layers for Huffman Coding...
Original storage for each parameter: 5.0000 bits
Average storage for each parameter after Huffman Coding: 4.6289 bits
Complete 15 layers for Huffman Coding...
Original storage for each parameter: 5.0000 bits
Average storage for each parameter after Huffman Coding: 4.9011 bits
Complete 16 layers for Huffman Coding...
```