# ECE 661 Homework Assignment 4

I, THWISHA NAHENDER ([TN130@DUKE.EDU], CHOOSE TO USE 1 LATE DAYS FOR HOMEWORK 4.)
HOMEWORK 4, FALL 2023

## True/False Questions:

1. **True.** Weight pruning is removing unnecessary weights from the model whereas weight quantization is used for reducing the number of bits required to represent each weight. Weight pruning and weight quantization can be performed on the same model to achieve the best compression results, they are often applied independently.
2. **False.** Weight pruning removes unimportant weights thereby reducing the number of parameters in a neural network, this can affect the inference latency. If too many weights are removed or if the weights removed were not removed in a uniform manner, this can lead to decrease in accuracy.
3. **True.** In deep compression pipeline even if you skip quantization step, a pruned model can still be effectively encoded using Huffman coding. The sparsity that is achieved after pruning allows for effectively encoding using Huffman coding.
4. **False.** When we use SGD to optimize sparsity inducing regularizes, it promotes small weights but do not guarantee that all weights will become zero, pruning on the other hand, sets weights to zero.
5. **False**. Using soft thresholding operator can lead to better results compared to using L-1 regularization directly. Soft thresholding operator is basically the proximal mapping of the L-1 norm, it can address the bias problem of L-1.
6. **False.** Group Lasso does help introduce structured sparsity, L1 regularization is applied on L2 regularization.
7. **True.** Proximal gradient descent introduces an additional proximity term to the optimization objective to handle regularization. The proximity term encourages properties in the weight parameters, often sparsity.
8. **True.** Models equipped with early exits allows some inputs to be computed when the entire model is not processed. This helps to overcome the issue of overfitting and overthinking by allowing easier samples to exit early.
9. **False.** When implementing quantization aware training with STE, gradients are not quantized during backpropagation, the gradient information is maintained during back propagation. Quantization aware training with STE is during forward propagation where weights are quantized.
10. **True.** Mixed precision quantization scheme can reach higher accuracy with a similar sized model as compared to quantizing all the layers in a DNN model to the same precision, this is because mixed precision allows different layers of the DNN to be quantized to different precisions, by doing so you can allocate higher bit precision to layers that are more critical for preserving accuracy.

# LAB 1: Sparse optimization of linear models:

## Question (a):

**The loss function L is defined as:**

$$L = \Sigma_i (XiW - yi)^2$$

We know that the $W^{k+1}$ can be derived as:

$$W^{k+1} = W^k - \mu \frac{\delta l}{\delta W}(W^k)$$

where

$$\frac{\delta l}{\delta W}(W^k) = 2\sum_i (X_i W^k - y_i) X_i$$

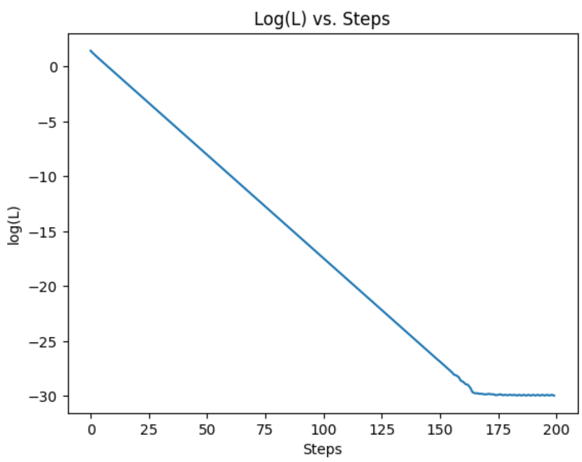
```
X = np.array([[1, -2, -1, -1, 1],
              [2, -1, 2, 0, -2],
              [-1, 0, 2, 2, 1]])
y = np.array([7, 1, 1]).reshape(-1, 1)
learning_rate = 0.02
num_steps = 200
```

## Question (b):

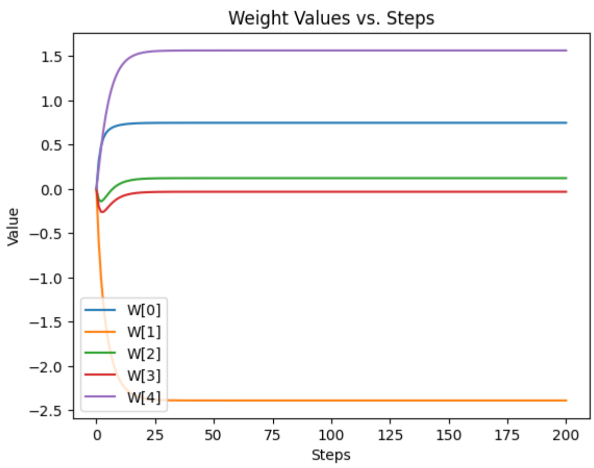
```
W_1 = np.zeros((5,1))
loss_1 = []
weight_1 = []

for step in range(num_steps):

  weight_1.append(W_1.copy())

  gradient= 2 * np.dot(X.T, (np.dot(X, W_1) - y))
  W_1 -= learning_rate * gradient


  loss = np.sum((X.dot(W_1) - y) ** 2)
  loss_1.append(np.log10(loss))

weight_1.append(W_1.copy())
loss_1 = np.array(loss_1)
weight_1 = np.array(weight_1)

print(W_1)
plt.figure(1)
plt.plot(range(num_steps), loss_1)
plt.xlabel('Steps')
plt.ylabel('log(L)')
plt.title('Log(L) vs. Steps')


plt.figure(2)
for i in range(5):
    plt.plot(range(num_steps+1), weight_1[:, i], label=f'W[{i}]')

plt.xlabel('Steps')
plt.ylabel('Value')
plt.title('Weight Values vs. Steps')
plt.legend()

plt.show()
```

```
Weights are
 [[ 0.74759615]
 [-2.38942308]
 [ 0.12259615]
 [-0.03125    ]
 [ 1.56490385]]
```

**Log (Loss) vs Number of steps throughout the training:**




**Value of each element in W changing throughout the training:**




**Is W converging to an optimal or sparse solution?**

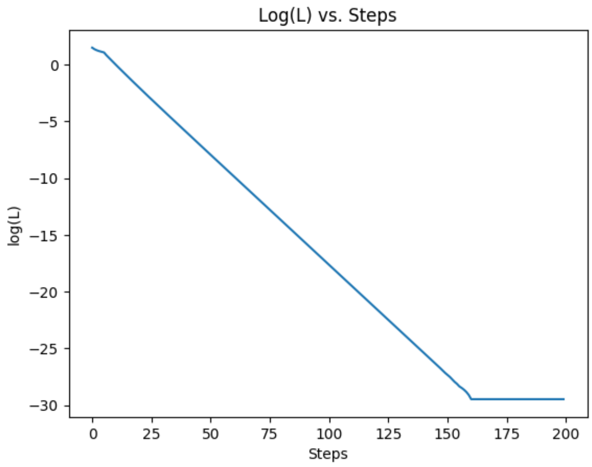
By observing the above log (loss) vs steps graph, we can see that at the beginning of the training process the log loss is relatively high, as the training progresses the loss function steadily decreases indicating that the model is improving its predictions, thus we can say that it is converging to an optimal solution. When we observe the evolution of the individual weights, we can see that none of the weights become zero, hence we can say that W is not converging towards a sparse solution.

## Question (c):

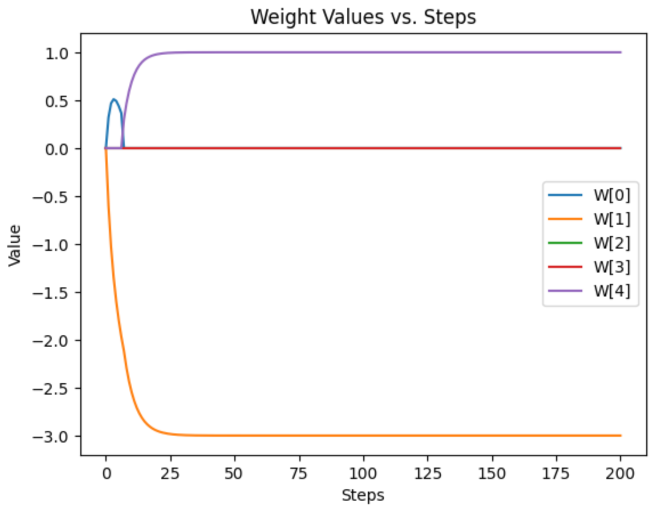
```
W_2 = np.zeros((5,1))
loss_2 = []
weight_2 = []
n=2

for step in range(num_steps):

  weight_2.append(W_2.copy())

  gradient= 2 * np.dot(X.T, (np.dot(X, W_2) - y))
  W_2 -= learning_rate * gradient
  if np.count_nonzero(W_2)>n:
      indices = np.argsort(np.absolute(W_2), axis=0)
      W_2[indices[:3]] = 0


  loss = np.sum((X.dot(W_2) - y) ** 2)
  loss_2.append(np.log10(loss))

weight_2.append(W_2.copy())
loss_2 = np.array(loss_2)
weight_2 = np.array(weight_2)

print("Weights are\n",W_2)

plt.figure(1)
plt.plot(range(num_steps), loss_2)
plt.xlabel('Steps')
plt.ylabel('log(L)')
plt.title('Log(L) vs. Steps')


plt.figure(2)
for i in range(5):
    plt.plot(range(num_steps+1), weight_2[:, i], label=f'W[{i}]')

plt.xlabel('Steps')
plt.ylabel('Value')
plt.title('Weight Values vs. Steps')
plt.legend()

plt.show()
```

```
Weights are
 [[ 0.]
 [-3.]
 [ 0.]
 [ 0.]
 [ 1.]]
```

**Log (Loss) vs Number of steps throughout the training:**



**<u>Value of each element in W changing throughout the training:</u>**




**Is W converging to an optimal or sparse solution?**

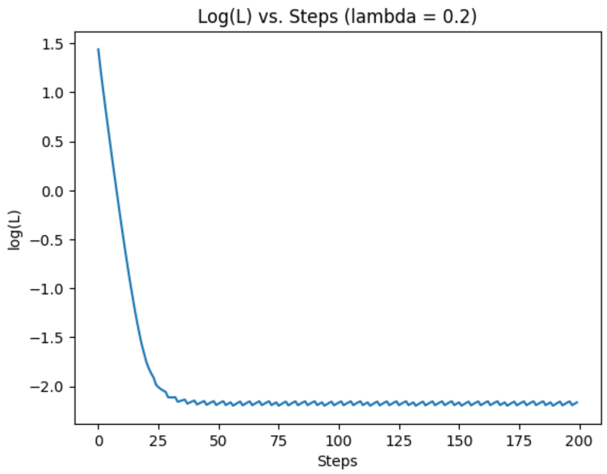
By observing the above log (loss) vs steps graph, we can see that at the beginning of the training process the log loss is relatively high, as the training progresses the loss function steadily decreases indicating that the model is improving its predictions, thus we can say it is converging to an optimal solution. When we observe the evolution of the individual weights we can see that three of the weights become exactly zero, this can be seen in the weights vs steps graph as well as the weights matrix, hence W is converging to a sparse solution.

**Question (d):**

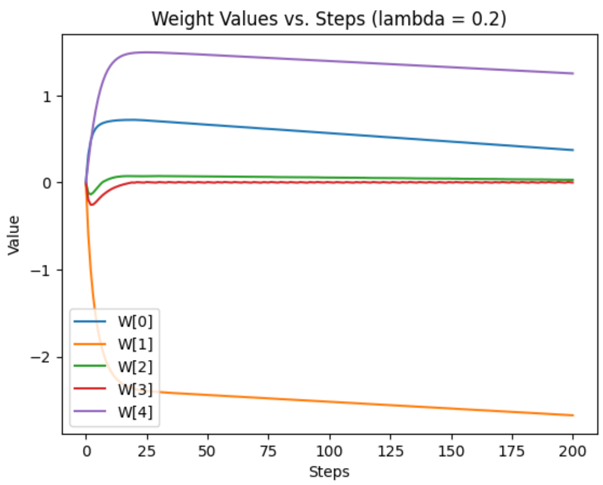
```python
lambdas = [0.2, 0.5, 1.0, 2.0]

for lambda_value in lambdas:
    W_3 = np.zeros((5,1))

    loss_values = []
    weight_values = []
    weight_values.append(W_3.copy())

    for step in range(num_steps):
        gradient = 2 * X.T.dot(X.dot(W_3) - y) + lambda_value * np.sign(W_3)
        W_3 -= learning_rate * gradient
        loss = np.sum((X.dot(W_3) - y) ** 2)
        loss_values.append(np.log10(loss))
        weight_values.append(W_3.copy())

    loss_values = np.array(loss_values)
    weight_values = np.array(weight_values)



    plt.figure()
    plt.plot(range(num_steps), loss_values)
    plt.xlabel('Steps')
    plt.ylabel('log(L)')
    plt.title(f'Log(L) vs. Steps (lambda = {lambda_value})')

    plt.figure()
    for i in range(5):
        plt.plot(range(num_steps + 1), weight_values[:, i], label=f'W[{i}]')

    plt.xlabel('Steps')
    plt.ylabel('Value')
    plt.title(f'Weight Values vs. Steps (lambda = {lambda_value})')

    plt.legend()
```

```
Weights for lambda=0.2 is
 [[ 3.73490632e-01]
 [-2.67447522e+00]
 [ 3.12622639e-02]
 [ 3.51021267e-04]
 [ 1.25352897e+00]]
Weights for lambda=0.5 is
 [[ 9.86962942e-03]
 [-2.94696785e+00]
 [-7.87580251e-04]
 [ 8.60508674e-04]
 [ 9.57922261e-01]]
Weights for lambda=1.0 is
 [[-0.00767726]
 [-2.89320225]
 [ 0.00957581]
 [ 0.00316848]
 [ 0.91767585]]
Weights for lambda=2.0 is
 [[-0.00552779]
 [-2.77999112]
 [-0.04210149]
 [-0.00459144]
 [ 0.84427393]]
```

**Log (Loss) vs Number of steps throughout the training for lambda=0.2:**

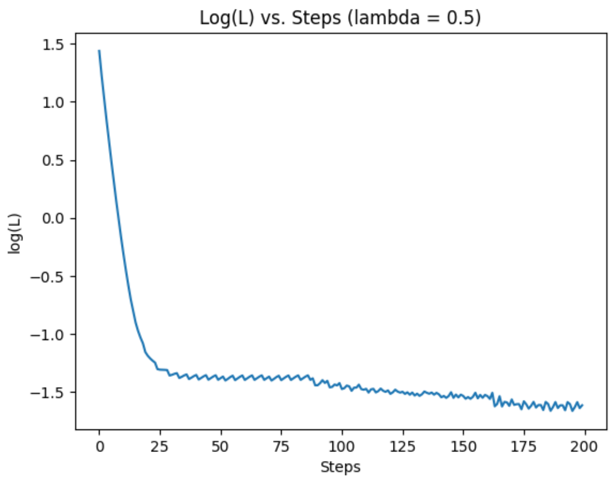



**Value of each element in W changing throughout the training for lambda=0.2:**

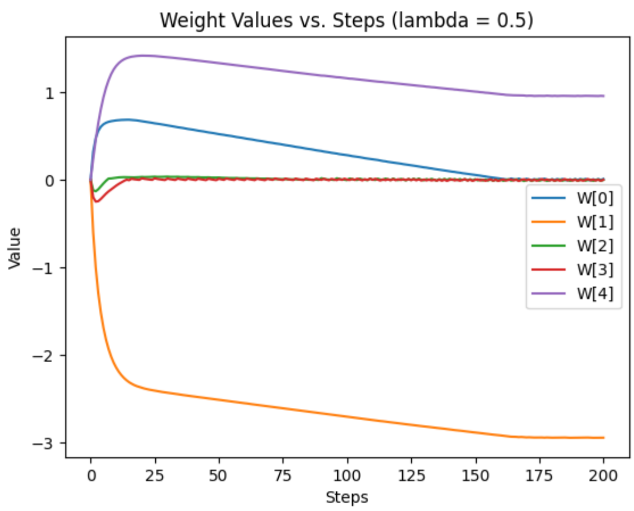


Observing the graph of Log Loss versus the number of steps, it becomes evident that beyond the 30th step, the loss values stabilize at a low level, indicating convergence towards an optimal solution. However, when examining the graph of weights versus steps for λ=0.2, it is notable that none of the weights reach zero. Consequently, it can be concluded that W is not converging towards a sparse solution in this scenario.

**<u>Log (Loss) vs Number of steps throughout the training for lambda=0.5:</u>**

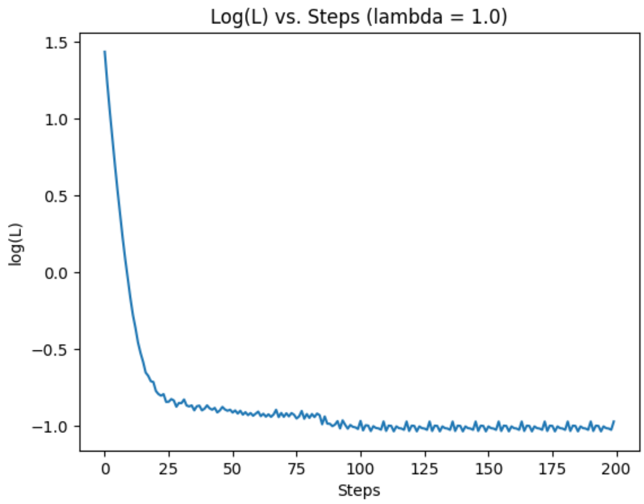



**<u>Value of each element in W changing throughout the training for lambda=0.5:</u>**

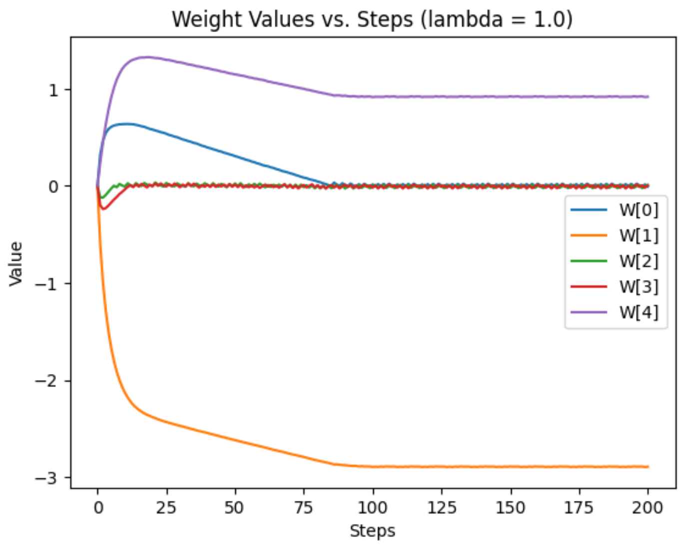



Examining the Log Loss versus the number of steps graph for λ=0.5 reveals a decrease in loss values at the start of training, with further reduction occurring after the 80th step. However, it does not reach convergence to an optimal solution. By observing the weights versus the number of steps graph, it becomes evident that by the 160th step, three weights have precisely become zero. Consequently, it can be asserted that W is converging towards a sparse solution in this case.

**Log (Loss) vs Number of steps throughout the training for lambda=1.0:**

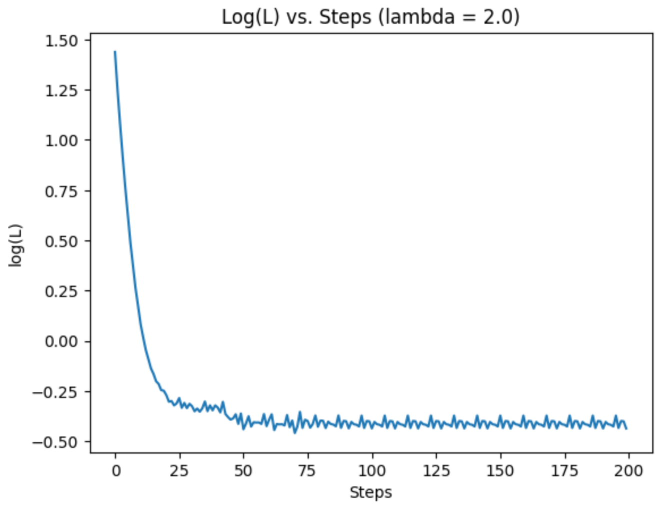



**Value of each element in W changing throughout the training for lambda=1.0:**

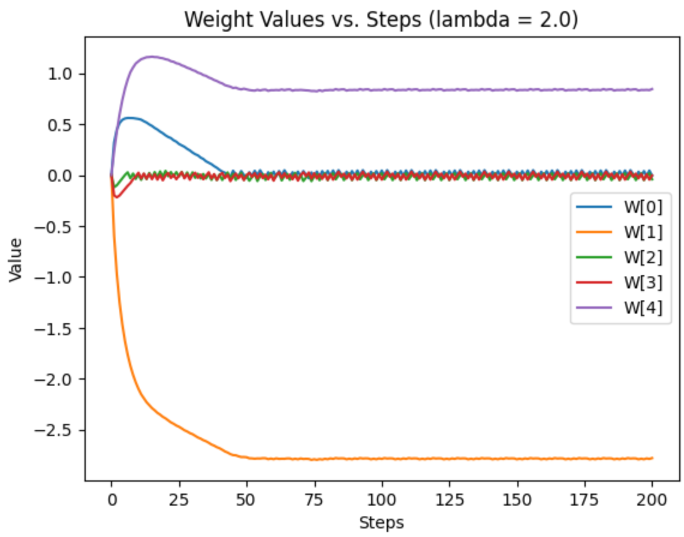



Having established that for λ=0.2, the loss values dipped below -2, by observing the log loss versus the number of steps graph for λ=1.0 reveals that the loss values do not reach below -2, they stay around -1. This suggests a lack of convergence to an optimal solution. Looking at the weight versus steps graphs, it becomes apparent that by the 80th step, three weights reach zero. Consequently, it can be concluded that W is converging towards a sparse solution in this scenario.

**Log (Loss) vs Number of steps throughout the training for lambda=2.0:**




**Value of each element in W changing throughout the training for lambda=2.0:**



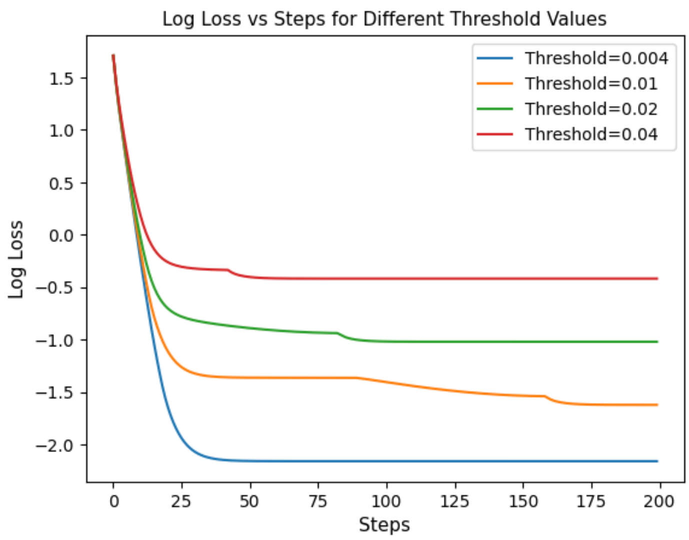

Having established that for λ=0.2, the loss values dipped below -2, by observing the log loss versus the number of steps graph for λ=1.0 reveals that the loss values do not reach below -2, they stay around -0.5. This suggests a lack of convergence to an optimal solution. Looking at the weight versus steps graphs, it becomes apparent that by the 60th step, three weights reach zero. Consequently, it can be concluded that W is converging towards a sparse solution in this scenario.

Increasing λ enhances the strength of the L1 penalty, intensifying the sparsity effect. Nevertheless, elevated λ levels impede convergence, resulting in underfitting due to excessively strong regularization. This is evident in the log loss plot, where higher λ values lead to increased loss penalties, illustrating the underfitting phenomenon. Additionally, the elevated λ values contribute to more weights reaching zero, showcasing the induction of sparsity.

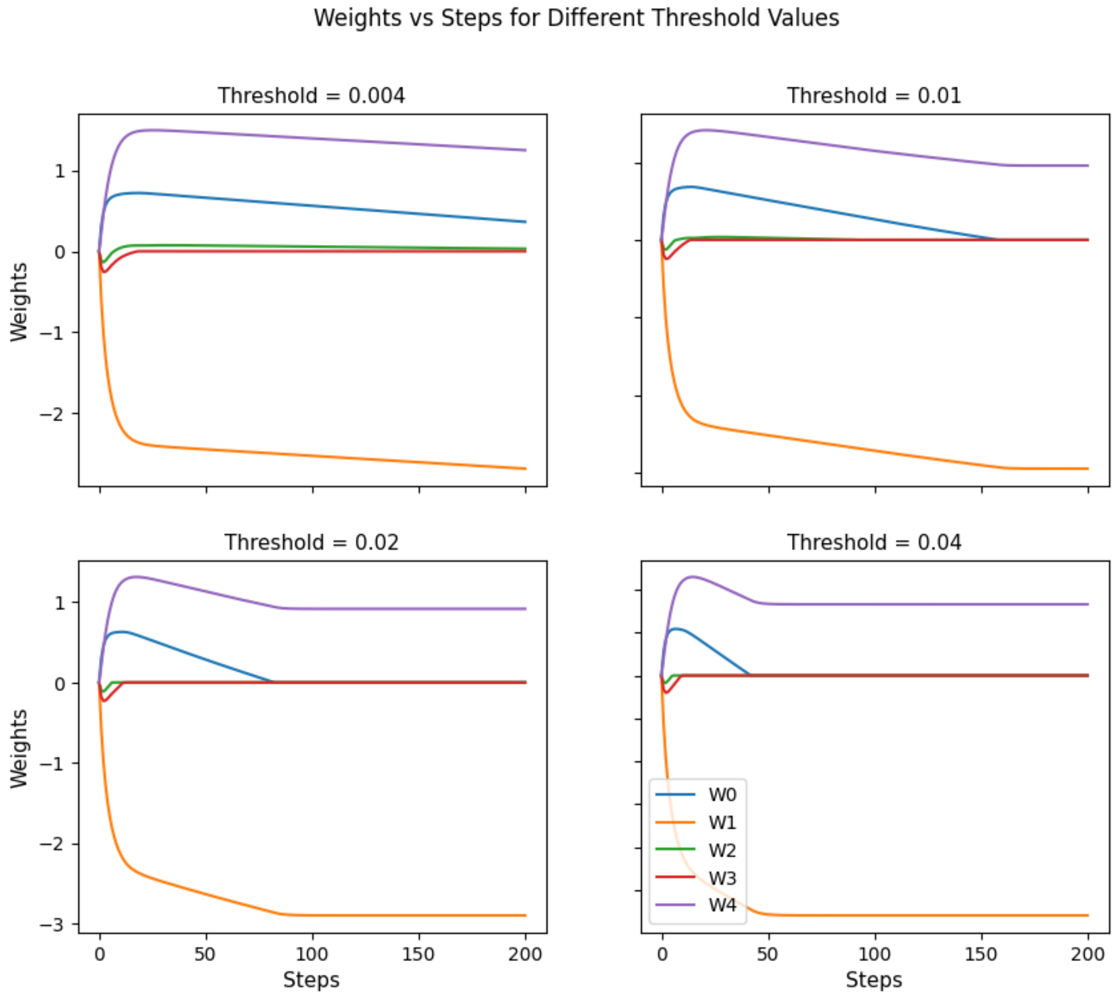
## Question (e):

```python
def proximal_lasso(threshold):
    weights_matrix = np.zeros((5, 1))
    regularization_strength = 2

    log_loss_history, weights_history = [], [weights_matrix.T[0]]

    for step in range(num_steps):
        loss = sum((X.dot(weights_matrix) - y) ** 2)
        gradient = 2 * np.dot(X.T, (X.dot(weights_matrix) - y))
        weights_matrix = weights_matrix - learning_rate * gradient
        weights_matrix = np.sign(weights_matrix) * np.maximum(abs(weights_matrix) - threshold, 0)

        log_loss_history.append(loss)
        weights_history.append(weights_matrix.T[0])

    weights_history = np.array(weights_history)

    return np.log10(log_loss_history), weights_history

threshold_values = [0.004, 0.01, 0.02, 0.04]
fig, axes = plt.subplots(2, 2, sharex=True, sharey=False, figsize=(10, 8))

losses_list = []
for j, threshold in enumerate(threshold_values):
    log_loss, weights_history = proximal_lasso(threshold)
    k, l = (round(max(j - 1, 0) / 2 + 0.1), j % 2)
    losses_list.append(log_loss.reshape(1, -1)[0])
    for i in range(5):
        axes[k, l].plot(np.arange(0, num_steps + 1), weights_history[:, i], label=f'W{i}')
    axes[k, l].set_xlabel('Steps', fontsize=11)
    axes[k, l].set_ylabel('Weights', fontsize=11)
    axes[k, l].set_title(f'Threshold = {threshold}', fontsize=11)
    axes[k, l].label_outer()

plt.suptitle('Weights vs Steps for Different Threshold Values', fontsize=12)
plt.legend()
plt.show()


losses_df = pd.DataFrame(np.array(losses_list).T, columns=[f'Threshold={threshold}' for threshold in threshold_values])
losses_df.plot()
plt.xlabel('Steps', fontsize=11)
plt.ylabel('Log Loss', fontsize=11)
plt.title('Log Loss vs Steps for Different Threshold Values', fontsize=11)
plt.show()
```

## <u>Log Loss vs Steps for Different Threshold values:</u>



**Weights vs Steps for Different Threshold Values:**




**Comparing results between (d) and (e)**

Upon examining the log loss versus the number of steps graphs for inducing sparsity with L1 regularization and proximity gradient update (PGD), the log loss graphs when inducing sparsity with L1 regularisation is seen to have a lot of noise, hence it is possible to get a less optimal solution as compared to when inducing sparsity using PGD.

Furthermore, analysing the evolution of weights reveals that the graphs are noticeably smoother when employing PGD compared to inducing sparsity with L1 regularization. The smoother evolution indicates less noise in the graphs. This characteristic is particularly beneficial when dealing with larger datasets, where the noise introduced by L1 regularization might result in a less optimal solution.

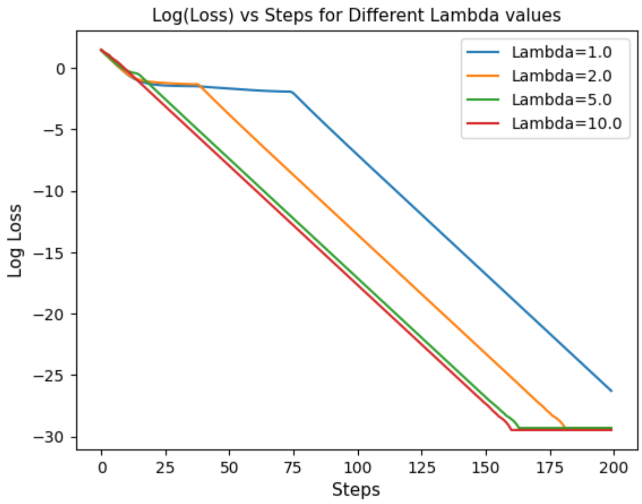
**Question (f):**

```
def update_weights(W, soft, mask):
    W_value = np.sign(W) * np.maximum(abs(W) - soft, 0)
    W[mask[:3]] = W_value[mask[:3]]
    return W

def trimmed_lasso_regularization(regularization_strength, threshold=2):
    weights = np.zeros((5, 1))
    weight_history = np.zeros((num_steps + 1, 5))
    log_loss_history = []

    for step in range(num_steps):
        gradient = 2 * np.dot(X.T, X.dot(weights) - y)
        weights -= learning_rate * gradient

        if np.count_nonzero(weights) > threshold:
            mask = np.argsort(np.abs(weights), axis=0).T[0]

        weights = update_weights(weights, regularization_strength * learning_rate, mask[:3])
        weight_history[step + 1] = weights.reshape(5)


        loss = sum((X.dot(weights) - y) ** 2)
        log_loss_history.append(loss)

    return np.log10(log_loss_history), weight_history

lambda_values = [1.0, 2.0, 5.0, 10.0]
losses_list = []
weights_list = []

for reg_strength in lambda_values:
    log_loss, weight_history = trimmed_lasso_regularization(reg_strength)
    losses_list.append(log_loss.reshape(1, -1))
    weights_list.append(weight_history)
    print(f"Lambda={reg_strength}, Weights = {weight_history[-1]}")
```

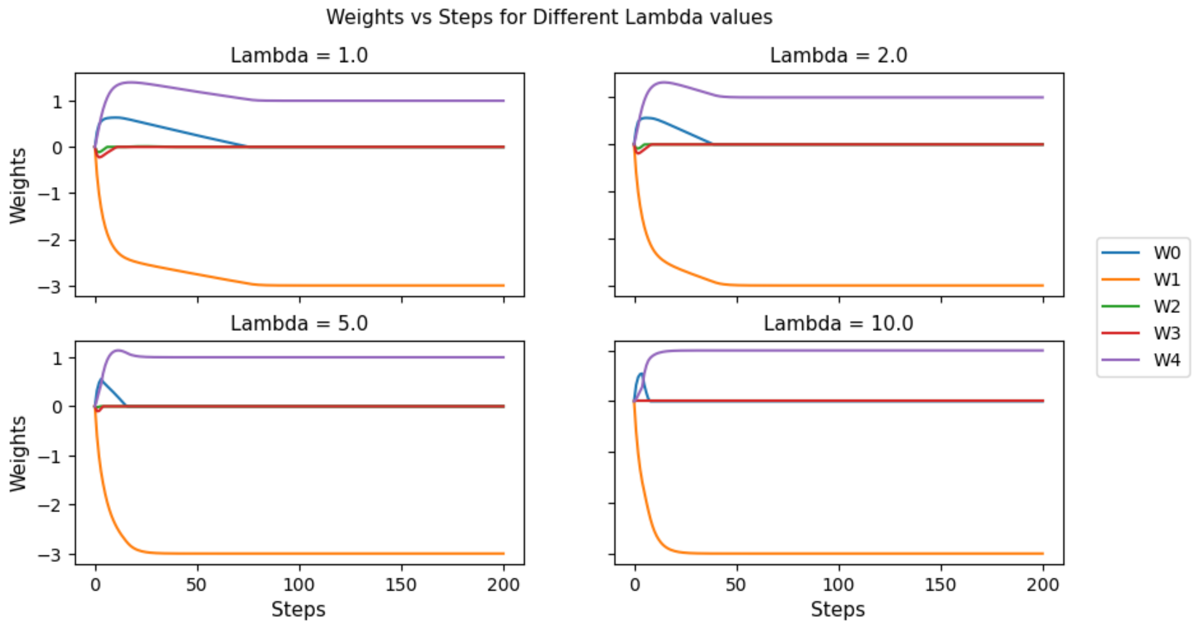
```
losses_df = pd.DataFrame(np.concatenate(losses_list, axis=0), index=[f'Lambda={reg_strength}' for reg_strength in lambda_values])
losses_df.T.plot()
plt.xlabel('Steps', fontsize=11)
plt.ylabel('Log Loss', fontsize=11)
plt.title('Log(Loss) vs Steps for Different Lambda values', fontsize=11)
plt.show()

fig, axes = plt.subplots(2, 2, sharex=True, sharey=False, figsize=(10, 5))
for j, reg_strength in enumerate(lambda_values):
    weight_history = weights_list[j]
    k, l = (round(max(j - 1, 0) / 2 + 0.1), j % 2)
    for i in range(5):
        axes[k, l].plot(np.arange(0, num_steps + 1), weight_history[:, i], label=f'W{i}')
    axes[k, l].set_xlabel('Steps', fontsize=11)
    axes[k, l].set_ylabel('Weights', fontsize=11)
    axes[k, l].set_title(f'Lambda = {reg_strength}', fontsize=11)
    axes[k, l].label_outer()

axes[1, 1].legend(loc='upper right', bbox_to_anchor=(1, 1, 0.3, 0.5))

plt.suptitle('Weights vs Steps for Different Lambda values ', fontsize=11)
plt.show()
```

```
Lambda=1.0, Weights = [ 0. -3.  0. -0.  1.]
Lambda=2.0, Weights = [ 0. -3.  0. -0.  1.]
Lambda=5.0, Weights = [ 0. -3.  0. -0.  1.]
Lambda=10.0, Weights = [ 0. -3. -0. -0.  1.]
```

**Log Loss vs Steps for Different Lambda values:**




**Weights vs Steps for Different Lambda values:**




**Comparing Trimmed l1 with l1 regularization**

When we observe the log loss graphs for Trimmed l1 vs the log loss graphs for l1 regularization, we see that the loss values decrease drastically, the loss seems to converge to an optimal solution. For just l1 regularization the lowest loss values were seen when lambda was 0.2. In the case of trimmed l1, all the models reduce rapidly and each step.

When we observe the evolution of weights graphs for trimmed l1 it is seen that the weights are converging much faster as compared to just l1 regularization, the sparsity achieved is also much faster in trimmed l1 regularization. When lambda is 5 and 10, we can see that 3 out of 5 weights reduce to zero after 20 epochs.

**Comparing Trimmed l1 with iterative pruning**

During the early steps, that is the first few iterations trimmed l1 and iterative pruning perform similarly, but as the training progresses we see that trimmed l1 has a lower bias. Trimmed l1 has resilience to noise and outliers.

# LAB 2: Pruning ResNet-20 model:

**Question(a):**

**Test accuracy=0.9151**

```
net = ResNetCIFAR(num_layers=20, Nbits=None)
net = net.to(device)

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

Files already downloaded and verified
Test Loss=0.3231, Test accuracy=0.9151
```

**Question(b):**

Defining function **prune_by_percentage**

```
def prune_by_percentage(layer, q=70.0):
    """
    Pruning the weight paramters by threshold.
    :param q: pruning percentile. 'q' percent of the least
    significant weight parameters will be pruned.
    """
    # Convert the weight of "layer" to numpy array
    weights=layer.weight.data.cpu().numpy()

    # Compute the q-th percentile of the abs of the converted array
    threshold=np.percentile(np.abs(weights),q)

    # Generate a binary mask same shape as weight to decide which element to prune
    mask=np.abs(weights) >= threshold

    # Convert mask to torch tensor and put on GPU
    mask=torch.from_numpy(mask).float().to(device)

    # Multiply the weight by mask to perform pruning
    layer.weight.data *= mask
    pass
```

```
q_values = [30, 50, 70]

for q in q_values:
    print("Pruning with q =", q)
    net.load_state_dict(torch.load("pretrained_model.pt"))

    for name, layer in net.named_modules():
        if (isinstance(layer, nn.Conv2d) or isinstance(layer, nn.Linear)) and 'id_mapping' not in name:
            # Apply pruning by percentage
            prune_by_percentage(layer, q=q)

            # Count the number of zeros and total parameters for sparsity
            np_weight = layer.weight.data.cpu().numpy()
            zeros = np.sum(np_weight == 0)
            total = np_weight.size

            # Calculate and print sparsity
            sparsity = zeros / total
            print(f'Sparsity of {name}: {sparsity:.2%}')

    # Test the pruned model
    test(net)
```

**Pruning % q=0.3, Test accuracy obtained=0.9028**

```
Pruning with q = 30
Sparsity of head_conv.0.conv: 30.09%
Sparsity of body_op.0.conv1.0.conv: 29.99%
Sparsity of body_op.0.conv2.0.conv: 29.99%
Sparsity of body_op.1.conv1.0.conv: 29.99%
Sparsity of body_op.1.conv2.0.conv: 29.99%
Sparsity of body_op.2.conv1.0.conv: 29.99%
Sparsity of body_op.2.conv2.0.conv: 29.99%
Sparsity of body_op.3.conv1.0.conv: 30.01%
Sparsity of body_op.3.conv2.0.conv: 30.00%
Sparsity of body_op.4.conv1.0.conv: 30.00%
Sparsity of body_op.4.conv2.0.conv: 30.00%
Sparsity of body_op.5.conv1.0.conv: 30.00%
Sparsity of body_op.5.conv2.0.conv: 30.00%
Sparsity of body_op.6.conv1.0.conv: 30.00%
Sparsity of body_op.6.conv2.0.conv: 30.00%
Sparsity of body_op.7.conv1.0.conv: 30.00%
Sparsity of body_op.7.conv2.0.conv: 30.00%
Sparsity of body_op.8.conv1.0.conv: 30.00%
Sparsity of body_op.8.conv2.0.conv: 30.00%
Sparsity of final_fc.linear: 30.00%
Files already downloaded and verified
Test Loss=0.3698, Test accuracy=0.9028
```

**Pruning % q=0.5, Test accuracy obtained=0.8210**

```
Pruning with q = 50
Sparsity of head_conv.0.conv: 50.00%
Sparsity of body_op.0.conv1.0.conv: 50.00%
Sparsity of body_op.0.conv2.0.conv: 50.00%
Sparsity of body_op.1.conv1.0.conv: 50.00%
Sparsity of body_op.1.conv2.0.conv: 50.00%
Sparsity of body_op.2.conv1.0.conv: 50.00%
Sparsity of body_op.2.conv2.0.conv: 50.00%
Sparsity of body_op.3.conv1.0.conv: 50.00%
Sparsity of body_op.3.conv2.0.conv: 50.00%
Sparsity of body_op.4.conv1.0.conv: 50.00%
Sparsity of body_op.4.conv2.0.conv: 50.00%
Sparsity of body_op.5.conv1.0.conv: 50.00%
Sparsity of body_op.5.conv2.0.conv: 50.00%
Sparsity of body_op.6.conv1.0.conv: 50.00%
Sparsity of body_op.6.conv2.0.conv: 50.00%
Sparsity of body_op.7.conv1.0.conv: 50.00%
Sparsity of body_op.7.conv2.0.conv: 50.00%
Sparsity of body_op.8.conv1.0.conv: 50.00%
Sparsity of body_op.8.conv2.0.conv: 50.00%
Sparsity of final_fc.linear: 50.00%
Files already downloaded and verified
Test Loss=0.6774, Test accuracy=0.8210
```

**Pruning % q=0.7, Test accuracy obtained=0.4204**

```
Pruning with q = 70
Sparsity of head_conv.0.conv: 69.91%
Sparsity of body_op.0.conv1.0.conv: 70.01%
Sparsity of body_op.0.conv2.0.conv: 70.01%
Sparsity of body_op.1.conv1.0.conv: 70.01%
Sparsity of body_op.1.conv2.0.conv: 70.01%
Sparsity of body_op.2.conv1.0.conv: 70.01%
Sparsity of body_op.2.conv2.0.conv: 70.01%
Sparsity of body_op.3.conv1.0.conv: 69.99%
Sparsity of body_op.3.conv2.0.conv: 70.00%
Sparsity of body_op.4.conv1.0.conv: 70.00%
Sparsity of body_op.4.conv2.0.conv: 70.00%
Sparsity of body_op.5.conv1.0.conv: 70.00%
Sparsity of body_op.5.conv2.0.conv: 70.00%
Sparsity of body_op.6.conv1.0.conv: 70.00%
Sparsity of body_op.6.conv2.0.conv: 70.00%
Sparsity of body_op.7.conv1.0.conv: 70.00%
Sparsity of body_op.7.conv2.0.conv: 70.00%
Sparsity of body_op.8.conv1.0.conv: 70.00%
Sparsity of body_op.8.conv2.0.conv: 70.00%
Sparsity of final_fc.linear: 70.00%
Files already downloaded and verified
Test Loss=2.4417, Test accuracy=0.4204
```

**Question (c):**

Defining function **finetune_after_prune**

```
def finetune_after_prune(net, trainloader, criterion, optimizer, prune=True):
    """
    Finetune the pruned model for a single epoch
    Make sure pruned weights are kept as zero
    """
    # Build a dictionary for the nonzero weights
    weight_mask = {}
    for name,layer in net.named_modules():
        if (isinstance(layer, nn.Conv2d) or isinstance(layer, nn.Linear)) and 'id_mapping' not in name:
            # Your code here: generate a mask in GPU torch tensor to have 1 for nonzero element and 0 for zero element
            weight_mask[name] = (layer.weight != 0).float().to(device)

    global_steps = 0
    train_loss = 0
    correct = 0
    total = 0
    start = time.time()
    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()
        optimizer.step()

        if prune:
            for name,layer in net.named_modules():
                if (isinstance(layer, nn.Conv2d) or isinstance(layer, nn.Linear)) and 'id_mapping' not in name:
                    # Your code here: Use weight_mask to make sure zero elements remains zero
                    layer.weight.data *= weight_mask[name]

        train_loss += loss.item()
        _, predicted = outputs.max(1)
        total += targets.size(0)
        correct += predicted.eq(targets).sum().item()
        global_steps += 1

        if global_steps % 50 == 0:
            end = time.time()
            batch_size = 256
            num_examples_per_second = 50 * batch_size / (end - start)
            print("[Step=%d]\tLoss=%.4f\tacc=%.4f\t%.1f examples/second"
                 % (global_steps, train_loss / (batch_idx + 1), (correct / total), num_examples_per_second))
            start = time.time()
```

**Finetuning the pruned model with q=0.7**

```
# Get pruned model
net.load_state_dict(torch.load("pretrained_model.pt"))
for name,layer in net.named_modules():
    if (isinstance(layer, nn.Conv2d) or isinstance(layer, nn.Linear)) and 'id_mapping' not in name:
        prune_by_percentage(layer, q=70.0)

# Training setup, do not change
batch_size=256
lr=0.002
reg=1e-4

print('==> Preparing data..')
transform_train = transforms.Compose([
    transforms.RandomCrop(32, padding=4),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010)),
])

transform_test = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010)),
])
best_acc = 0  # best test accuracy
start_epoch = 0  # start from epoch 0 or last checkpoint epoch
trainset = torchvision.datasets.CIFAR10(root='./data', train=True, download=True, transform=transform_train)
trainloader = torch.utils.data.DataLoader(trainset, batch_size=batch_size, shuffle=True, num_workers=16)

testset = torchvision.datasets.CIFAR10(root='./data', train=False, download=True, transform=transform_test)
testloader = torch.utils.data.DataLoader(testset, batch_size=100, shuffle=False, num_workers=2)

criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=lr, momentum=0.875, weight_decay=reg, nesterov=False)
```

```
==> Preparing data..
Files already downloaded and verified
Files already downloaded and verified
```

```python
# Model finetuning
for epoch in range(20):
    print('\nEpoch: %d' % epoch)
    net.train()
    finetune_after_prune(net, trainloader, criterion, optimizer,prune=True)
    #Start the testing code.
    net.eval()
    test_loss = 0
    correct = 0
    total = 0
    with torch.no_grad():
        for batch_idx, (inputs, targets) in enumerate(testloader):
            inputs, targets = inputs.to(device), targets.to(device)
            outputs = net(inputs)
            loss = criterion(outputs, targets)

            test_loss += loss.item()
            _, predicted = outputs.max(1)
            total += targets.size(0)
            correct += predicted.eq(targets).sum().item()
    num_val_steps = len(testloader)
    val_acc = correct / total
    print("Test Loss=%.4f, Test acc=%.4f" % (test_loss / (num_val_steps), val_acc))

    if val_acc > best_acc:
        best_acc = val_acc
        print("Saving...")
        torch.save(net.state_dict(), "net_after_finetune.pt")
```

```
Epoch: 0
[Step=50]       Loss=0.4122     acc=0.8602      1151.1 examples/second
[Step=100]      Loss=0.3620     acc=0.8764      2643.2 examples/second
[Step=150]      Loss=0.3306     acc=0.8866      1678.5 examples/second
Test Loss=0.4251, Test acc=0.8682
Saving...

Epoch: 1
[Step=50]       Loss=0.2529     acc=0.9120      1179.3 examples/second
[Step=100]      Loss=0.2469     acc=0.9143      1514.4 examples/second
[Step=150]      Loss=0.2402     acc=0.9163      1803.8 examples/second
Test Loss=0.3958, Test acc=0.8759
Saving...

Epoch: 2
[Step=50]       Loss=0.2197     acc=0.9210      1172.0 examples/second
[Step=100]      Loss=0.2149     acc=0.9246      2384.1 examples/second
[Step=150]      Loss=0.2120     acc=0.9254      1749.0 examples/second
Test Loss=0.3800, Test acc=0.8800
Saving...

Epoch: 3
[Step=50]       Loss=0.1958     acc=0.9306      1148.7 examples/second
[Step=100]      Loss=0.1987     acc=0.9297      2347.9 examples/second
[Step=150]      Loss=0.1992     acc=0.9297      1909.6 examples/second
Test Loss=0.3705, Test acc=0.8840
Saving...

Epoch: 4
[Step=50]       Loss=0.1901     acc=0.9350      1190.4 examples/second
[Step=100]      Loss=0.1883     acc=0.9359      2543.4 examples/second
[Step=150]      Loss=0.1901     acc=0.9348      2218.8 examples/second
Test Loss=0.3649, Test acc=0.8845
Saving...

Epoch: 5
[Step=50]       Loss=0.1750     acc=0.9375      1317.4 examples/second
[Step=100]      Loss=0.1834     acc=0.9352      2138.7 examples/second
[Step=150]      Loss=0.1804     acc=0.9371      2369.8 examples/second
Test Loss=0.3586, Test acc=0.8858
Saving...

Epoch: 6
[Step=50]       Loss=0.1736     acc=0.9408      1434.6 examples/second
[Step=100]      Loss=0.1767     acc=0.9391      1801.5 examples/second
[Step=150]      Loss=0.1778     acc=0.9376      2376.9 examples/second
Test Loss=0.3543, Test acc=0.8871
Saving...

Epoch: 7
[Step=50]       Loss=0.1761     acc=0.9382      1524.8 examples/second
[Step=100]      Loss=0.1738     acc=0.9389      1772.0 examples/second
[Step=150]      Loss=0.1713     acc=0.9399      2460.7 examples/second
Test Loss=0.3498, Test acc=0.8880
```

```
Epoch: 8
[Step=50]       Loss=0.1635     acc=0.9420      1554.3 examples/second
[Step=100]      Loss=0.1649     acc=0.9429      1728.9 examples/second
[Step=150]      Loss=0.1663     acc=0.9423      2535.7 examples/second
Test Loss=0.3494, Test acc=0.8883
Saving...

Epoch: 9
[Step=50]       Loss=0.1634     acc=0.9445      1274.9 examples/second
[Step=100]      Loss=0.1610     acc=0.9443      1989.1 examples/second
[Step=150]      Loss=0.1630     acc=0.9434      1956.7 examples/second
Test Loss=0.3442, Test acc=0.8898
Saving...

Epoch: 10
[Step=50]       Loss=0.1562     acc=0.9467      1361.6 examples/second
[Step=100]      Loss=0.1619     acc=0.9448      2041.8 examples/second
[Step=150]      Loss=0.1585     acc=0.9463      2053.9 examples/second
Test Loss=0.3439, Test acc=0.8896

Epoch: 11
[Step=50]       Loss=0.1575     acc=0.9454      1297.8 examples/second
[Step=100]      Loss=0.1557     acc=0.9450      2230.1 examples/second
[Step=150]      Loss=0.1548     acc=0.9457      1907.1 examples/second
Test Loss=0.3417, Test acc=0.8917
Saving...

Epoch: 12
[Step=50]       Loss=0.1601     acc=0.9430      1182.8 examples/second
[Step=100]      Loss=0.1592     acc=0.9459      2565.0 examples/second
[Step=150]      Loss=0.1564     acc=0.9464      1752.0 examples/second
Test Loss=0.3413, Test acc=0.8915

Epoch: 13
[Step=50]       Loss=0.1531     acc=0.9450      1193.8 examples/second
[Step=100]      Loss=0.1518     acc=0.9471      2518.0 examples/second
[Step=150]      Loss=0.1538     acc=0.9467      1987.6 examples/second
Test Loss=0.3386, Test acc=0.8909

Epoch: 14
[Step=50]       Loss=0.1530     acc=0.9475      1246.1 examples/second
[Step=100]      Loss=0.1493     acc=0.9483      2510.5 examples/second
[Step=150]      Loss=0.1488     acc=0.9485      2593.8 examples/second
Test Loss=0.3381, Test acc=0.8933
Saving...

Epoch: 15
[Step=50]       Loss=0.1477     acc=0.9493      1574.0 examples/second
[Step=100]      Loss=0.1513     acc=0.9490      1851.2 examples/second
[Step=150]      Loss=0.1501     acc=0.9493      2658.2 examples/second
Test Loss=0.3366, Test acc=0.8927
```

```
Epoch: 16
[Step=50]       Loss=0.1494     acc=0.9487      1522.9 examples/second
[Step=100]      Loss=0.1476     acc=0.9496      1611.0 examples/second
[Step=150]      Loss=0.1467     acc=0.9494      1850.2 examples/second
Test Loss=0.3366, Test acc=0.8939
Saving...

Epoch: 17
[Step=50]       Loss=0.1439     acc=0.9501      1403.1 examples/second
[Step=100]      Loss=0.1432     acc=0.9503      1991.1 examples/second
[Step=150]      Loss=0.1440     acc=0.9497      2218.3 examples/second
Test Loss=0.3358, Test acc=0.8938

Epoch: 18
[Step=50]       Loss=0.1427     acc=0.9495      1311.5 examples/second
[Step=100]      Loss=0.1397     acc=0.9496      2288.2 examples/second
[Step=150]      Loss=0.1422     acc=0.9493      1948.3 examples/second
Test Loss=0.3335, Test acc=0.8956
Saving...

Epoch: 19
[Step=50]       Loss=0.1428     acc=0.9493      1176.1 examples/second
[Step=100]      Loss=0.1417     acc=0.9500      2592.8 examples/second
[Step=150]      Loss=0.1421     acc=0.9497      1763.1 examples/second
Test Loss=0.3352, Test acc=0.8928
```

**The best test accuracy obtained=0.8928**

```
# Check sparsity of the finetuned model, make sure it's not changed
net.load_state_dict(torch.load("net_after_finetune.pt"))

for name,layer in net.named_modules():
    if (isinstance(layer, nn.Conv2d) or isinstance(layer, nn.Linear)) and 'id_mapping' not in name:
        # Your code here:
        # Convert the weight of "layer" to numpy array
        np_weight = layer.weight.data.cpu().numpy()
        # Count number of zeros
        zeros = np.sum(np_weight == 0)
        # Count number of parameters
        total = np_weight.size
        # Print sparsity
        print('Sparsity of '+name+': '+str(zeros/total))

test(net)
```

```
Sparsity of head_conv.0.conv: 0.6990740740740741
Sparsity of body_op.0.conv1.0.conv: 0.7000868055555556
Sparsity of body_op.0.conv2.0.conv: 0.7000868055555556
Sparsity of body_op.1.conv1.0.conv: 0.7000868055555556
Sparsity of body_op.1.conv2.0.conv: 0.7000868055555556
Sparsity of body_op.2.conv1.0.conv: 0.7000868055555556
Sparsity of body_op.2.conv2.0.conv: 0.7000868055555556
Sparsity of body_op.3.conv1.0.conv: 0.6998697916666666
Sparsity of body_op.3.conv2.0.conv: 0.6999782986111112
Sparsity of body_op.4.conv1.0.conv: 0.6999782986111112
Sparsity of body_op.4.conv2.0.conv: 0.6999782986111112
Sparsity of body_op.5.conv1.0.conv: 0.6999782986111112
Sparsity of body_op.5.conv2.0.conv: 0.6999782986111112
Sparsity of body_op.6.conv1.0.conv: 0.6999782986111112
Sparsity of body_op.6.conv2.0.conv: 0.7000054253472222
Sparsity of body_op.7.conv1.0.conv: 0.7000054253472222
Sparsity of body_op.7.conv2.0.conv: 0.7000054253472222
Sparsity of body_op.8.conv1.0.conv: 0.7000054253472222
Sparsity of body_op.8.conv2.0.conv: 0.7000054253472222
Sparsity of final_fc.linear: 0.7
Files already downloaded and verified
Test Loss=0.3335, Test accuracy=0.8956
```

**Sparsity of the model =70%**

Looking at the sparsity values obtained, we can see that the sparsity has been preserved and is 70% sparsity.

## Question (d):

```python
net.load_state_dict(torch.load("pretrained_model.pt"))
best_acc = 0.
for epoch in range(20):
    print('\nEpoch: %d' % epoch)

    net.train()
    if epoch<10:
        for name,layer in net.named_modules():
            if (isinstance(layer, nn.Conv2d) or isinstance(layer, nn.Linear)) and 'id_mapping' not in name:
                # Increase model sparsity
                q = (epoch+1)*7
                prune_by_percentage(layer, q=q)
    if epoch<9:
        finetune_after_prune(net, trainloader, criterion, optimizer,prune=False)
    else:
        finetune_after_prune(net, trainloader, criterion, optimizer)

    #Start the testing code.
    net.eval()
    test_loss = 0
    correct = 0
    total = 0
    with torch.no_grad():
        for batch_idx, (inputs, targets) in enumerate(testloader):
            inputs, targets = inputs.to(device), targets.to(device)
            outputs = net(inputs)
            loss = criterion(outputs, targets)

            test_loss += loss.item()
            _, predicted = outputs.max(1)
            total += targets.size(0)
            correct += predicted.eq(targets).sum().item()
    num_val_steps = len(testloader)
    val_acc = correct / total
    print("Test Loss=%.4f, Test acc=%.4f" % (test_loss / (num_val_steps), val_acc))

    if epoch>=10:
        if val_acc > best_acc:
            best_acc = val_acc
            print("Saving...")
            torch.save(net.state_dict(), "net_after_iterative_prune.pt")
```

```
Epoch: 0
[Step=50]       Loss=0.0472     acc=0.9854      1467.9 examples/second
[Step=100]      Loss=0.0471     acc=0.9852      2290.7 examples/second
[Step=150]      Loss=0.0481     acc=0.9845      2072.9 examples/second
Test Loss=0.3260, Test acc=0.9151

Epoch: 1
[Step=50]       Loss=0.0473     acc=0.9858      1272.5 examples/second
[Step=100]      Loss=0.0496     acc=0.9839      2673.8 examples/second
[Step=150]      Loss=0.0493     acc=0.9842      1936.9 examples/second
Test Loss=0.3261, Test acc=0.9154

Epoch: 2
[Step=50]       Loss=0.0502     acc=0.9841      1277.4 examples/second
[Step=100]      Loss=0.0490     acc=0.9843      2634.7 examples/second
[Step=150]      Loss=0.0509     acc=0.9839      2513.8 examples/second
Test Loss=0.3252, Test acc=0.9150

Epoch: 3
[Step=50]       Loss=0.0559     acc=0.9806      1441.3 examples/second
[Step=100]      Loss=0.0533     acc=0.9822      2072.6 examples/second
[Step=150]      Loss=0.0527     acc=0.9830      2613.5 examples/second
Test Loss=0.3291, Test acc=0.9135

Epoch: 4
[Step=50]       Loss=0.0611     acc=0.9807      1543.6 examples/second
[Step=100]      Loss=0.0622     acc=0.9799      1845.0 examples/second
[Step=150]      Loss=0.0620     acc=0.9798      2619.9 examples/second
Test Loss=0.3370, Test acc=0.9107

Epoch: 5
[Step=50]       Loss=0.0706     acc=0.9744      1520.2 examples/second
[Step=100]      Loss=0.0697     acc=0.9751      2079.4 examples/second
[Step=150]      Loss=0.0696     acc=0.9748      2323.8 examples/second
Test Loss=0.3360, Test acc=0.9087

Epoch: 6
[Step=50]       Loss=0.0907     acc=0.9694      1375.7 examples/second
[Step=100]      Loss=0.0862     acc=0.9710      2344.5 examples/second
[Step=150]      Loss=0.0861     acc=0.9706      1961.4 examples/second
Test Loss=0.3339, Test acc=0.9066

Epoch: 7
[Step=50]       Loss=0.1301     acc=0.9544      1276.0 examples/second
[Step=100]      Loss=0.1234     acc=0.9577      2524.2 examples/second
[Step=150]      Loss=0.1193     acc=0.9591      2009.2 examples/second
Test Loss=0.3362, Test acc=0.9021

Epoch: 8
[Step=50]       Loss=0.1647     acc=0.9439      1309.3 examples/second
[Step=100]      Loss=0.1591     acc=0.9444      2606.6 examples/second
[Step=150]      Loss=0.1545     acc=0.9457      2474.0 examples/second
Test Loss=0.3458, Test acc=0.8953
```

```
Epoch: 9
[Step=50]       Loss=0.2704     acc=0.9062      1513.7 examples/second
[Step=100]      Loss=0.2536     acc=0.9128      1918.6 examples/second
[Step=150]      Loss=0.2423     acc=0.9158      2519.2 examples/second
Test Loss=0.3892, Test acc=0.8759

Epoch: 10
[Step=50]       Loss=0.2026     acc=0.9319      1626.8 examples/second
[Step=100]      Loss=0.2064     acc=0.9294      1859.6 examples/second
[Step=150]      Loss=0.2058     acc=0.9295      2634.7 examples/second
Test Loss=0.3720, Test acc=0.8811
Saving...

Epoch: 11
[Step=50]       Loss=0.2008     acc=0.9313      1587.3 examples/second
[Step=100]      Loss=0.1924     acc=0.9332      2126.8 examples/second
[Step=150]      Loss=0.1912     acc=0.9334      2218.9 examples/second
Test Loss=0.3657, Test acc=0.8824
Saving...

Epoch: 12
[Step=50]       Loss=0.1852     acc=0.9351      1321.1 examples/second
[Step=100]      Loss=0.1823     acc=0.9366      2687.6 examples/second
[Step=150]      Loss=0.1813     acc=0.9367      1898.2 examples/second
Test Loss=0.3598, Test acc=0.8845
Saving...

Epoch: 13
[Step=50]       Loss=0.1753     acc=0.9383      1240.2 examples/second
[Step=100]      Loss=0.1725     acc=0.9393      2671.8 examples/second
[Step=150]      Loss=0.1737     acc=0.9403      2141.0 examples/second
Test Loss=0.3566, Test acc=0.8847
Saving...

Epoch: 14
[Step=50]       Loss=0.1767     acc=0.9421      1287.0 examples/second
[Step=100]      Loss=0.1773     acc=0.9394      2521.1 examples/second
[Step=150]      Loss=0.1726     acc=0.9411      2638.6 examples/second
Test Loss=0.3518, Test acc=0.8873
Saving...

Epoch: 15
[Step=50]       Loss=0.1668     acc=0.9418      1558.7 examples/second
[Step=100]      Loss=0.1643     acc=0.9434      1905.7 examples/second
[Step=150]      Loss=0.1634     acc=0.9436      2692.0 examples/second
Test Loss=0.3493, Test acc=0.8883
Saving...

Epoch: 16
[Step=50]       Loss=0.1656     acc=0.9424      1591.5 examples/second
[Step=100]      Loss=0.1590     acc=0.9441      1944.3 examples/second
[Step=150]      Loss=0.1611     acc=0.9438      2384.8 examples/second
Test Loss=0.3443, Test acc=0.8897
```

```
Epoch: 16
[Step=50]       Loss=0.1656     acc=0.9424      1591.5 examples/second
[Step=100]      Loss=0.1590     acc=0.9441      1944.3 examples/second
[Step=150]      Loss=0.1611     acc=0.9438      2384.8 examples/second
Test Loss=0.3443, Test acc=0.8897
Saving...

Epoch: 17
[Step=50]       Loss=0.1556     acc=0.9451      1408.1 examples/second
[Step=100]      Loss=0.1604     acc=0.9441      2388.4 examples/second
[Step=150]      Loss=0.1598     acc=0.9438      1876.5 examples/second
Test Loss=0.3452, Test acc=0.8902
Saving...

Epoch: 18
[Step=50]       Loss=0.1562     acc=0.9473      1231.3 examples/second
[Step=100]      Loss=0.1541     acc=0.9466      2667.6 examples/second
[Step=150]      Loss=0.1563     acc=0.9457      2031.1 examples/second
Test Loss=0.3420, Test acc=0.8899

Epoch: 19
[Step=50]       Loss=0.1521     acc=0.9483      1353.0 examples/second
[Step=100]      Loss=0.1519     acc=0.9476      2424.8 examples/second
[Step=150]      Loss=0.1505     acc=0.9481      2641.7 examples/second
Test Loss=0.3421, Test acc=0.8909
Saving...
```

**The best test accuracy obtained=0.8909**

**Checking sparsity:**

```
# Check sparsity of the final model, make sure it's 70%
net.load_state_dict(torch.load("net_after_iterative_prune.pt"))

for name,layer in net.named_modules():
    if (isinstance(layer, nn.Conv2d) or isinstance(layer, nn.Linear)) and 'id_mapping' not in name:
        # Your code here: can copy from previous question
         # Convert the weight of "layer" to numpy array
        np_weight = layer.weight.data.cpu().numpy()
        # Count number of zeros
        zeros = np.sum(np_weight == 0)
        # Count number of parameters
        total = np_weight.size
        # Print sparsity
        print('Sparsity of '+name+': '+str(zeros/total))


test(net)
```

```
Sparsity of head_conv.0.conv: 0.6990740740740741
Sparsity of body_op.0.conv1.0.conv: 0.7000868055555556
Sparsity of body_op.0.conv2.0.conv: 0.7000868055555556
Sparsity of body_op.1.conv1.0.conv: 0.7000868055555556
Sparsity of body_op.1.conv2.0.conv: 0.7000868055555556
Sparsity of body_op.2.conv1.0.conv: 0.7000868055555556
Sparsity of body_op.2.conv2.0.conv: 0.7000868055555556
Sparsity of body_op.3.conv1.0.conv: 0.6998697916666666
Sparsity of body_op.3.conv2.0.conv: 0.6999782986111112
Sparsity of body_op.4.conv1.0.conv: 0.6999782986111112
Sparsity of body_op.4.conv2.0.conv: 0.6999782986111112
Sparsity of body_op.5.conv1.0.conv: 0.6999782986111112
Sparsity of body_op.5.conv2.0.conv: 0.6999782986111112
Sparsity of body_op.6.conv1.0.conv: 0.6999782986111112
Sparsity of body_op.6.conv2.0.conv: 0.7000054253472222
Sparsity of body_op.7.conv1.0.conv: 0.7000054253472222
Sparsity of body_op.7.conv2.0.conv: 0.7000054253472222
Sparsity of body_op.8.conv1.0.conv: 0.7000054253472222
Sparsity of body_op.8.conv2.0.conv: 0.7000054253472222
Sparsity of final_fc.linear: 0.7
Files already downloaded and verified
Test Loss=0.3421, Test accuracy=0.8909
```

**Sparsity of the model=70%**

**Comparing iterative pruning with finetune pruned model:**

**Test accuracy for finetune pruned model =0.8928**

**Test accuracy for iterative pruning=0.8909**

The test accuracy for iterative pruning is lower compared to the test accuracy for the finetuned pruned model. This is because in iterative pruning, pruning is performed for only 10 epochs, whereas in the finetuned pruned model, pruning is applied throughout all 20 epochs.

**Question (e):**

Defining function **global_prune_by_percentage**

```python
def global_prune_by_percentage(net, q=70.0):
    """
    Pruning the weight paramters by threshold.
    :param q: pruning percentile. 'q' percent of the least
    significant weight parameters will be pruned.
    """
    # A list to gather all the weights
    flattened_weights = []
    # Find global pruning threshold
    for name,layer in net.named_modules():
        if (isinstance(layer, nn.Conv2d) or isinstance(layer, nn.Linear)) and 'id_mapping' not in name:
            # Convert weight to numpy
            np_weight = layer.weight.data.cpu().numpy()

            # Flatten the weight and append to flattened_weights
            flattended_weight = np_weight.flatten()
            flattened_weights.append(flattended_weight)


    # Concate all weights into a np array
    flattened_weights = np.concatenate(flattened_weights)
    # Find global pruning threshold
    thres = np.percentile(np.abs(flattened_weights), q)

    # Apply pruning threshold to all layers
    for name,layer in net.named_modules():
        if (isinstance(layer, nn.Conv2d) or isinstance(layer, nn.Linear)) and 'id_mapping' not in name:
            # Convert weight to numpy
            np_weight = layer.weight.data.cpu().numpy()

            # Generate a binary mask same shape as weight to decide which element to prune
            mask = (np.abs(np_weight) > thres).astype(np.float32)

            # Convert mask to torch tensor and put on GPU
            mask = torch.from_numpy(mask).to(device)

            # Multiply the weight by mask to perform pruning
            layer.weight.data *= mask
```

```python
net.load_state_dict(torch.load("pretrained_model.pt"))
best_acc = 0.
for epoch in range(20):
    print('\nEpoch: %d' % epoch)
    q=(epoch+1)*7

    net.train()
    # Increase model sparsity
    if epoch<10:
        global_prune_by_percentage(net, q=q)
    if epoch<9:
        finetune_after_prune(net, trainloader, criterion, optimizer,prune=False)
    else:
        finetune_after_prune(net, trainloader, criterion, optimizer)

    #Start the testing code.
    net.eval()
    test_loss = 0
    correct = 0
    total = 0
    with torch.no_grad():
        for batch_idx, (inputs, targets) in enumerate(testloader):
            inputs, targets = inputs.to(device), targets.to(device)
            outputs = net(inputs)
            loss = criterion(outputs, targets)
            test_loss += loss.item()
            _, predicted = outputs.max(1)
            total += targets.size(0)
            correct += predicted.eq(targets).sum().item()
    num_val_steps = len(testloader)
    val_acc = correct / total
    print("Test Loss=%.4f, Test acc=%.4f" % (test_loss / (num_val_steps), val_acc))

    if epoch>=10:
        if val_acc > best_acc:
            best_acc = val_acc
            print("Saving...")
            torch.save(net.state_dict(), "net_after_global_iterative_prune.pt")
```

```
Epoch: 0
[Step=50]       Loss=0.0481     acc=0.9852      1402.2 examples/second
[Step=100]      Loss=0.0477     acc=0.9855      2195.9 examples/second
[Step=150]      Loss=0.0469     acc=0.9861      2658.5 examples/second
Test Loss=0.3242, Test acc=0.9151

Epoch: 1
[Step=50]       Loss=0.0503     acc=0.9842      1537.9 examples/second
[Step=100]      Loss=0.0475     acc=0.9848      1834.4 examples/second
[Step=150]      Loss=0.0474     acc=0.9851      2621.5 examples/second
Test Loss=0.3267, Test acc=0.9155

Epoch: 2
[Step=50]       Loss=0.0480     acc=0.9845      1595.6 examples/second
[Step=100]      Loss=0.0476     acc=0.9854      1783.7 examples/second
[Step=150]      Loss=0.0486     acc=0.9847      2526.8 examples/second
Test Loss=0.3281, Test acc=0.9146

Epoch: 3
[Step=50]       Loss=0.0512     acc=0.9847      1464.0 examples/second
[Step=100]      Loss=0.0516     acc=0.9837      2130.5 examples/second
[Step=150]      Loss=0.0523     acc=0.9834      2091.4 examples/second
Test Loss=0.3310, Test acc=0.9128

Epoch: 4
[Step=50]       Loss=0.0522     acc=0.9823      1247.4 examples/second
[Step=100]      Loss=0.0524     acc=0.9827      2511.4 examples/second
[Step=150]      Loss=0.0534     acc=0.9823      1591.2 examples/second
Test Loss=0.3283, Test acc=0.9139

Epoch: 5
[Step=50]       Loss=0.0608     acc=0.9806      1313.3 examples/second
[Step=100]      Loss=0.0611     acc=0.9799      2350.1 examples/second
[Step=150]      Loss=0.0609     acc=0.9800      1885.6 examples/second
Test Loss=0.3251, Test acc=0.9122

Epoch: 6
[Step=50]       Loss=0.0702     acc=0.9762      1242.0 examples/second
[Step=100]      Loss=0.0699     acc=0.9759      2589.2 examples/second
[Step=150]      Loss=0.0710     acc=0.9752      1847.9 examples/second
Test Loss=0.3264, Test acc=0.9089

Epoch: 7
[Step=50]       Loss=0.0954     acc=0.9666      1241.8 examples/second
[Step=100]      Loss=0.0925     acc=0.9677      2639.1 examples/second
[Step=150]      Loss=0.0898     acc=0.9688      2147.6 examples/second
Test Loss=0.3298, Test acc=0.9047

Epoch: 8
[Step=50]       Loss=0.1177     acc=0.9598      1344.6 examples/second
[Step=100]      Loss=0.1189     acc=0.9589      2248.1 examples/second
[Step=150]      Loss=0.1165     acc=0.9596      2534.0 examples/second
Test Loss=0.3264, Test acc=0.9020
```

```
Epoch: 9
[Step=50]       Loss=0.1816     acc=0.9361      1592.3 examples/second
[Step=100]      Loss=0.1747     acc=0.9395      1815.7 examples/second
[Step=150]      Loss=0.1712     acc=0.9411      2659.5 examples/second
Test Loss=0.3452, Test acc=0.8886

Epoch: 10
[Step=50]       Loss=0.1625     acc=0.9445      1595.8 examples/second
[Step=100]      Loss=0.1590     acc=0.9455      1829.2 examples/second
[Step=150]      Loss=0.1567     acc=0.9464      2551.7 examples/second
Test Loss=0.3354, Test acc=0.8909
Saving...

Epoch: 11
[Step=50]       Loss=0.1543     acc=0.9473      1472.9 examples/second
[Step=100]      Loss=0.1496     acc=0.9484      2077.7 examples/second
[Step=150]      Loss=0.1483     acc=0.9493      2182.2 examples/second
Test Loss=0.3308, Test acc=0.8936
Saving...

Epoch: 12
[Step=50]       Loss=0.1474     acc=0.9481      1328.4 examples/second
[Step=100]      Loss=0.1436     acc=0.9500      2329.1 examples/second
[Step=150]      Loss=0.1462     acc=0.9484      1923.9 examples/second
Test Loss=0.3266, Test acc=0.8943
Saving...

Epoch: 13
[Step=50]       Loss=0.1415     acc=0.9517      1198.8 examples/second
[Step=100]      Loss=0.1398     acc=0.9528      2594.8 examples/second
[Step=150]      Loss=0.1404     acc=0.9524      1900.1 examples/second
Test Loss=0.3254, Test acc=0.8944
Saving...

Epoch: 14
[Step=50]       Loss=0.1385     acc=0.9542      1180.9 examples/second
[Step=100]      Loss=0.1390     acc=0.9532      2584.9 examples/second
[Step=150]      Loss=0.1397     acc=0.9529      2249.8 examples/second
Test Loss=0.3248, Test acc=0.8951
Saving...

Epoch: 15
[Step=50]       Loss=0.1384     acc=0.9544      1311.6 examples/second
[Step=100]      Loss=0.1357     acc=0.9550      2278.5 examples/second
[Step=150]      Loss=0.1364     acc=0.9543      2530.2 examples/second
Test Loss=0.3220, Test acc=0.8956
Saving...

Epoch: 16
[Step=50]       Loss=0.1311     acc=0.9553      1531.6 examples/second
[Step=100]      Loss=0.1338     acc=0.9544      1809.8 examples/second
[Step=150]      Loss=0.1324     acc=0.9548      2573.9 examples/second
Test Loss=0.3214, Test acc=0.8969
```

```
Epoch: 17
[Step=50]       Loss=0.1246     acc=0.9569      1562.0 examples/second
[Step=100]      Loss=0.1298     acc=0.9554      1791.7 examples/second
[Step=150]      Loss=0.1304     acc=0.9552      2474.4 examples/second
Test Loss=0.3194, Test acc=0.8968

Epoch: 18
[Step=50]       Loss=0.1305     acc=0.9563      1448.2 examples/second
[Step=100]      Loss=0.1305     acc=0.9563      2137.6 examples/second
[Step=150]      Loss=0.1289     acc=0.9570      2016.6 examples/second
Test Loss=0.3176, Test acc=0.8972
Saving...

Epoch: 19
[Step=50]       Loss=0.1285     acc=0.9571      1221.8 examples/second
[Step=100]      Loss=0.1242     acc=0.9580      2620.1 examples/second
[Step=150]      Loss=0.1236     acc=0.9586      1769.3 examples/second
Test Loss=0.3179, Test acc=0.8972
```

**The best test accuracy obtained=0.8972**

## Checking sparsity:

```
net.load_state_dict(torch.load("net_after_global_iterative_prune.pt"))

zeros_sum = 0
total_sum = 0
for name,layer in net.named_modules():
    if (isinstance(layer, nn.Conv2d) or isinstance(layer, nn.Linear)) and 'id_mapping' not in name:
        # Your code here:
        # Convert the weight of "layer" to numpy array
        np_weight = layer.weight.data.cpu().numpy()
        # Count number of zeros
        zeros = np.sum(np_weight == 0)
        # Count number of parameters
        total = np_weight.size
        zeros_sum+=zeros
        total_sum+=total
        print('Sparsity of '+name+': '+str(zeros/total))
print('Total sparsity of: '+str(zeros_sum/total_sum))
test(net)
```

```
Sparsity of head_conv.0.conv: 0.24305555555555555
Sparsity of body_op.0.conv1.0.conv: 0.5503472222222222
Sparsity of body_op.0.conv2.0.conv: 0.5295138888888888
Sparsity of body_op.1.conv1.0.conv: 0.5186631944444444
Sparsity of body_op.1.conv2.0.conv: 0.5525173611111112
Sparsity of body_op.2.conv1.0.conv: 0.5186631944444444
Sparsity of body_op.2.conv2.0.conv: 0.5655381944444444
Sparsity of body_op.3.conv1.0.conv: 0.5251736111111112
Sparsity of body_op.3.conv2.0.conv: 0.5830078125
Sparsity of body_op.4.conv1.0.conv: 0.6159939236111112
Sparsity of body_op.4.conv2.0.conv: 0.6763237847222222
Sparsity of body_op.5.conv1.0.conv: 0.6111111111111112
Sparsity of body_op.5.conv2.0.conv: 0.7034505208333334
Sparsity of body_op.6.conv1.0.conv: 0.6154513888888888
Sparsity of body_op.6.conv2.0.conv: 0.6510959201388888
Sparsity of body_op.7.conv1.0.conv: 0.6623263888888888
Sparsity of body_op.7.conv2.0.conv: 0.718994140625
Sparsity of body_op.8.conv1.0.conv: 0.7478841145833334
Sparsity of body_op.8.conv2.0.conv: 0.9371202256944444
Sparsity of final_fc.linear: 0.1171875
Total sparsity of: 0.6999992546657922
Files already downloaded and verified
Test Loss=0.3176, Test accuracy=0.8972
```

**Sparsity of the model=69.99%**

## Percentage of zeros in each layer:

The percentage of zeros is equivalent to the sparsity in each layer.


```
Sparsity of head_conv.0.conv: 0.24305555555555555
Sparsity of body_op.0.conv1.0.conv: 0.5503472222222222
Sparsity of body_op.0.conv2.0.conv: 0.5295138888888888
Sparsity of body_op.1.conv1.0.conv: 0.5186631944444444
Sparsity of body_op.1.conv2.0.conv: 0.5525173611111112
Sparsity of body_op.2.conv1.0.conv: 0.5186631944444444
Sparsity of body_op.2.conv2.0.conv: 0.5655381944444444
Sparsity of body_op.3.conv1.0.conv: 0.5251736111111112
Sparsity of body_op.3.conv2.0.conv: 0.5830078125
Sparsity of body_op.4.conv1.0.conv: 0.6159939236111112
Sparsity of body_op.4.conv2.0.conv: 0.6763237847222222
Sparsity of body_op.5.conv1.0.conv: 0.6111111111111112
Sparsity of body_op.5.conv2.0.conv: 0.7034505208333334
Sparsity of body_op.6.conv1.0.conv: 0.6154513888888888
Sparsity of body_op.6.conv2.0.conv: 0.6510959201388888
Sparsity of body_op.7.conv1.0.conv: 0.6623263888888888
Sparsity of body_op.7.conv2.0.conv: 0.718994140625
Sparsity of body_op.8.conv1.0.conv: 0.7478841145833334
Sparsity of body_op.8.conv2.0.conv: 0.9371202256944444
Sparsity of final_fc.linear: 0.1171875
Total sparsity of: 0.6999992546657922
```

**Comparing the performance of different pruning methods:**

| Method | Test accuracy |
|---|---|
| Finetune after prune | 0.8928 |
| Iterative pruning | 0.8909 |
| Global pruning | 0.8972 |

From the above table we can conclude that the global pruning method performs better than the other two methods of pruning and gives a test accuracy of 0.8972.

## Lab 3: Fixed-point quantization and finetuning

**Question (a):**

```
class STE(torch.autograd.Function):
    @staticmethod
    def forward(ctx, w, bit, symmetric=False):
        '''
        symmetric: True for symmetric quantization, False for asymmetric quantization
        '''
        if bit is None:
            wq = w
        elif bit==0:
            wq = w*0
        else:
            # Build a mask to record position of zero weights
            weight_mask = torch.where(w == 0, torch.tensor(0.0), torch.tensor(1.0))

            # Lab3 (a), Your code here:
            if symmetric == False:
                # Compute alpha (scale) for dynamic scaling
                alpha = torch.max(w)-torch.min(w)

                # Compute beta (bias) for dynamic scaling
                beta = torch.min(w)
                # Scale w with alpha and beta so that all elements in ws are between 0 and 1
                ws = (w - beta) / alpha

                step = 2 ** (bit)-1
                # Quantize ws with a linear quantizer to "bit" bits
                R = torch.round(ws * step) / step
                # Scale the quantized weight R back with alpha and beta
                wq = R * alpha + beta
```

```python
# Lab3 (e), Your code here:
else:
    alpha=torch.max(torch.abs(w))
    beta=0
    ws = (w - beta) / alpha
    step = 2 ** (bit-1)-1
    R = torch.round(ws * step) / step
    wq = R * alpha + beta

    pass
```

**Question (b):**

```python
Nbits = 6 #Change this value to finish (b) and (c)

net_6 = ResNetCIFAR(num_layers=20, Nbits=Nbits)
net_6 = net_6.to(device)
net_6.load_state_dict(torch.load("pretrained_model.pt"))
test(net_6)
```

```
Files already downloaded and verified
Test Loss=0.3364, Test accuracy=0.9145
```

```python
Nbits = 5 #Change this value to finish (b) and (c)

net_5 = ResNetCIFAR(num_layers=20, Nbits=Nbits)
net_5 = net_5.to(device)
net_5.load_state_dict(torch.load("pretrained_model.pt"))
test(net_5)
```

```
Files already downloaded and verified
Test Loss=0.3390, Test accuracy=0.9112
```

```python
Nbits = 4 #Change this value to finish (b) and (c)

net_4 = ResNetCIFAR(num_layers=20, Nbits=Nbits)
net_4 = net_4.to(device)
net_4.load_state_dict(torch.load("pretrained_model.pt"))
test(net_4)
```

```
Files already downloaded and verified
Test Loss=0.3861, Test accuracy=0.8972
```

```python
Nbits = 3 #Change this value to finish (b) and (c)

net_3 = ResNetCIFAR(num_layers=20, Nbits=Nbits)
net_3 = net_3.to(device)
net_3.load_state_dict(torch.load("pretrained_model.pt"))
test(net_3)
```

```
Files already downloaded and verified
Test Loss=0.9874, Test accuracy=0.7662
```

```python
Nbits = 2 #Change this value to finish (b) and (c)

net_2 = ResNetCIFAR(num_layers=20, Nbits=Nbits)
net_2 = net_2.to(device)
net_2.load_state_dict(torch.load("pretrained_model.pt"))
test(net_2)
```

```
Files already downloaded and verified
Test Loss=9.5441, Test accuracy=0.0899
```

| Nbits | Test accuracy |
|---|---|
| 6 | 0.9145 |
| 5 | 0.9112 |
| 4 | 0.8972 |
| 3 | 0.7662 |
| 2 | 0.0899 |

**Question (c):**

**<u>Nbits=4</u>**

```python
# Quantized model finetuning
finetune(net_4, epochs=20, batch_size=256, lr=0.002, reg=1e-4)

# Load the model with best accuracy
net_4.load_state_dict(torch.load("quantized_net_after_finetune.pt"))
test(net_4)
```

```
Epoch: 16
[Step=3150]     Loss=0.0563     acc=0.9799      1058.2 examples/second
[Step=3200]     Loss=0.0528     acc=0.9825      1719.0 examples/second
[Step=3250]     Loss=0.0526     acc=0.9827      2072.9 examples/second
[Step=3300]     Loss=0.0527     acc=0.9829      2158.2 examples/second
Test Loss=0.3339, Test acc=0.9102

Epoch: 17
[Step=3350]     Loss=0.0546     acc=0.9813      1069.2 examples/second
[Step=3400]     Loss=0.0517     acc=0.9825      1645.5 examples/second
[Step=3450]     Loss=0.0549     acc=0.9812      2190.7 examples/second
[Step=3500]     Loss=0.0545     acc=0.9816      2129.8 examples/second
Test Loss=0.3320, Test acc=0.9129

Epoch: 18
[Step=3550]     Loss=0.0523     acc=0.9849      1053.4 examples/second
[Step=3600]     Loss=0.0507     acc=0.9838      1583.2 examples/second
[Step=3650]     Loss=0.0501     acc=0.9836      2268.1 examples/second
[Step=3700]     Loss=0.0518     acc=0.9833      2363.7 examples/second
Test Loss=0.3345, Test acc=0.9120

Epoch: 19
[Step=3750]     Loss=0.0511     acc=0.9842      1036.9 examples/second
[Step=3800]     Loss=0.0500     acc=0.9841      1590.7 examples/second
[Step=3850]     Loss=0.0526     acc=0.9827      2260.7 examples/second
[Step=3900]     Loss=0.0527     acc=0.9828      2363.1 examples/second
Test Loss=0.3360, Test acc=0.9130
Files already downloaded and verified
Test Loss=0.3289, Test accuracy=0.9136
```

**Test accuracy obtained=0.9136**

**<u>Nbits=3</u>**

```python
# Quantized model finetuning
finetune(net_3, epochs=20, batch_size=256, lr=0.002, reg=1e-4)

# Load the model with best accuracy
net_3.load_state_dict(torch.load("quantized_net_after_finetune.pt"))
test(net_3)
```

```
Epoch: 16
[Step=3150]     Loss=0.0947     acc=0.9685      1056.5 examples/second
[Step=3200]     Loss=0.0886     acc=0.9689      1746.1 examples/second
[Step=3250]     Loss=0.0869     acc=0.9697      2098.8 examples/second
[Step=3300]     Loss=0.0877     acc=0.9691      2344.4 examples/second
Test Loss=0.3603, Test acc=0.9023

Epoch: 17
[Step=3350]     Loss=0.0931     acc=0.9657      1065.6 examples/second
[Step=3400]     Loss=0.0854     acc=0.9696      1758.9 examples/second
[Step=3450]     Loss=0.0865     acc=0.9688      2138.1 examples/second
[Step=3500]     Loss=0.0876     acc=0.9684      2426.3 examples/second
Test Loss=0.3652, Test acc=0.9046

Epoch: 18
[Step=3550]     Loss=0.0947     acc=0.9693      1001.4 examples/second
[Step=3600]     Loss=0.0899     acc=0.9692      1712.3 examples/second
[Step=3650]     Loss=0.0877     acc=0.9697      2163.3 examples/second
[Step=3700]     Loss=0.0893     acc=0.9689      2519.0 examples/second
Test Loss=0.3592, Test acc=0.9042

Epoch: 19
[Step=3750]     Loss=0.0786     acc=0.9740      1013.0 examples/second
[Step=3800]     Loss=0.0824     acc=0.9720      1658.5 examples/second
[Step=3850]     Loss=0.0832     acc=0.9714      2293.0 examples/second
[Step=3900]     Loss=0.0837     acc=0.9710      2570.5 examples/second
Test Loss=0.3517, Test acc=0.9034
Files already downloaded and verified
Test Loss=0.3589, Test accuracy=0.9049
```

**Test accuracy obtained=0.9049**

**Nbits=2**

```
# Quantized model finetuning
finetune(net_2, epochs=20, batch_size=256, lr=0.002, reg=1e-4)

# Load the model with best accuracy
net_2.load_state_dict(torch.load("quantized_net_after_finetune.pt"))
test(net_2)
```

```
Epoch: 16
[Step=3150]     Loss=0.2816     acc=0.9023      1058.0 examples/second
[Step=3200]     Loss=0.2823     acc=0.9002      1789.9 examples/second
[Step=3250]     Loss=0.2810     acc=0.9002      2165.4 examples/second
[Step=3300]     Loss=0.2827     acc=0.8993      2099.9 examples/second
Test Loss=0.4719, Test acc=0.8587
Saving...

Epoch: 17
[Step=3350]     Loss=0.2961     acc=0.8989      1044.5 examples/second
[Step=3400]     Loss=0.2856     acc=0.8964      1680.4 examples/second
[Step=3450]     Loss=0.2845     acc=0.8979      2304.9 examples/second
[Step=3500]     Loss=0.2836     acc=0.8989      2088.7 examples/second
Test Loss=0.5002, Test acc=0.8472

Epoch: 18
[Step=3550]     Loss=0.2846     acc=0.8990      1034.9 examples/second
[Step=3600]     Loss=0.2798     acc=0.9001      1675.3 examples/second
[Step=3650]     Loss=0.2799     acc=0.9001      2361.3 examples/second
[Step=3700]     Loss=0.2776     acc=0.9016      2140.1 examples/second
Test Loss=0.4878, Test acc=0.8537

Epoch: 19
[Step=3750]     Loss=0.2840     acc=0.9016      1049.3 examples/second
[Step=3800]     Loss=0.2828     acc=0.9023      1695.2 examples/second
[Step=3850]     Loss=0.2789     acc=0.9028      2377.8 examples/second
[Step=3900]     Loss=0.2775     acc=0.9027      2338.1 examples/second
Test Loss=0.5776, Test acc=0.8330
Files already downloaded and verified
Test Loss=0.4719, Test accuracy=0.8587
```

**Test accuracy obtained=0.8587**

| Nbits | Finetuned Test Accuracy |
|---|---|
| 4 | 0.9136 |
| 3 | 0.9049 |
| 2 | 0.8587 |

When we set Nbits to 4, we observe the highest accuracy, reaching 0.9136, compared to the other precision settings. Conversely, when Nbits is reduced to 2, we observe the lowest accuracy of 0.8587, making it the least accurate configuration. Lower precision quantization results in smaller and more efficient models, which require fewer computational resources. However, these models suffer from reduced accuracy due to the loss of information about the weights. On the other hand, higher precision settings, such as Nbits=4, lead to more accurate models. While they maintain better accuracy, they also demand higher computational resources. The reason for this is that with higher precision, the models retain more information about the weights, allowing them to better capture the underlying patterns in the data. In cases where models lose a significant amount of information during , fine-tuning becomes less effective in recovering accuracy. This is evident in the results, where even after fine-tuning, models with lower precision struggle to achieve the same level of accuracy as models with higher precision.

**Question (d):**

**Nbits=4**

**Before finetuning test accuracy=0.8722**

```
# Define quantized model and load weight
Nbits = 4 #Change this value to finish (d)

net_4B = ResNetCIFAR(num_layers=20, Nbits=Nbits)
net_4B= net_4B.to(device)
net_4B.load_state_dict(torch.load("net_after_global_iterative_prune.pt"))
test(net_4B)
```

```
Files already downloaded and verified
Test Loss=0.4115, Test accuracy=0.8722
```

**After finetuning test accuracy=0.9031**

```
# Quantized model finetuning
finetune(net_4B, epochs=20, batch_size=256, lr=0.002, reg=1e-4)

# Load the model with best accuracy
net_4B.load_state_dict(torch.load("quantized_net_after_finetune.pt"))
test(net_4B)
```

```
Epoch: 17
[Step=3350]	Loss=0.0941	acc=0.9670	1064.4 examples/second
[Step=3400]	Loss=0.1021	acc=0.9630	1716.8 examples/second
[Step=3450]	Loss=0.1061	acc=0.9622	2345.9 examples/second
[Step=3500]	Loss=0.1062	acc=0.9617	2250.7 examples/second
Test Loss=0.3308, Test acc=0.9032

Epoch: 18
[Step=3550]	Loss=0.0994	acc=0.9640	1050.5 examples/second
[Step=3600]	Loss=0.1019	acc=0.9633	1692.1 examples/second
[Step=3650]	Loss=0.0991	acc=0.9641	2351.3 examples/second
[Step=3700]	Loss=0.0989	acc=0.9645	2408.9 examples/second
Test Loss=0.3476, Test acc=0.9021

Epoch: 19
[Step=3750]	Loss=0.1011	acc=0.9639	1040.0 examples/second
[Step=3800]	Loss=0.0974	acc=0.9650	1701.8 examples/second
[Step=3850]	Loss=0.1000	acc=0.9644	2268.2 examples/second
[Step=3900]	Loss=0.0984	acc=0.9646	2399.9 examples/second
Test Loss=0.3469, Test acc=0.9031
Files already downloaded and verified
Test Loss=0.3369, Test accuracy=0.9051
```

## Nbits=3

**Before finetuning test accuracy=0.6331**

```
# Define quantized model and load weight
Nbits = 3 #Change this value to finish (d)

net_3B = ResNetCIFAR(num_layers=20, Nbits=Nbits)
net_3B= net_3B.to(device)
net_3B.load_state_dict(torch.load("net_after_global_iterative_prune.pt"))
test(net_3B)
```

```
Files already downloaded and verified
Test Loss=1.1141, Test accuracy=0.6331
```

**After finetuning test accuracy=0.8846**

```
# Quantized model finetuning
finetune(net_3B, epochs=20, batch_size=256, lr=0.002, reg=1e-4)

# Load the model with best accuracy
net_3B.load_state_dict(torch.load("quantized_net_after_finetune.pt"))
test(net_3B)
```

```
Epoch: 17
[Step=3350]	Loss=0.2061	acc=0.9269	1064.0 examples/second
[Step=3400]	Loss=0.2018	acc=0.9266	1653.6 examples/second
[Step=3450]	Loss=0.2001	acc=0.9282	2374.6 examples/second
[Step=3500]	Loss=0.1980	acc=0.9290	2299.7 examples/second
Test Loss=0.3636, Test acc=0.8846
Saving...

Epoch: 18
[Step=3550]	Loss=0.1898	acc=0.9318	1074.3 examples/second
[Step=3600]	Loss=0.1940	acc=0.9306	1654.4 examples/second
[Step=3650]	Loss=0.1938	acc=0.9315	2385.2 examples/second
[Step=3700]	Loss=0.1948	acc=0.9313	2416.1 examples/second
Test Loss=0.3791, Test acc=0.8791

Epoch: 19
[Step=3750]	Loss=0.1855	acc=0.9331	1062.8 examples/second
[Step=3800]	Loss=0.1882	acc=0.9337	1638.1 examples/second
[Step=3850]	Loss=0.1902	acc=0.9330	2423.2 examples/second
[Step=3900]	Loss=0.1908	acc=0.9330	2560.5 examples/second
Test Loss=0.3688, Test acc=0.8835
Files already downloaded and verified
Test Loss=0.3636, Test accuracy=0.8846
```

**<u>Nbits=2</u>**

**Before finetuning test accuracy=0.1000**

```
# Define quantized model and load weight
Nbits = 2 #Change this value to finish (d)

net_2B = ResNetCIFAR(num_layers=20, Nbits=Nbits)
net_2B= net_2B.to(device)
net_2B.load_state_dict(torch.load("net_after_global_iterative_prune.pt"))
test(net_2B)
```

```
Files already downloaded and verified
Test Loss=7358.1566, Test accuracy=0.1000
```

**After finetuning test accuracy=0.3778**

```
# Quantized model finetuning
finetune(net_2B, epochs=20, batch_size=256, lr=0.002, reg=1e-4)

# Load the model with best accuracy
net_2B.load_state_dict(torch.load("quantized_net_after_finetune.pt"))
test(net_2B)
```

```
Epoch: 17
[Step=3350]     Loss=1.7247     acc=0.3479      1048.7 examples/second
[Step=3400]     Loss=1.7047     acc=0.3668      1697.7 examples/second
[Step=3450]     Loss=1.7018     acc=0.3670      2147.4 examples/second
[Step=3500]     Loss=1.6980     acc=0.3672      2118.0 examples/second
Test Loss=1.6954, Test acc=0.3702
Saving...

Epoch: 18
[Step=3550]     Loss=1.6943     acc=0.3681      1059.4 examples/second
[Step=3600]     Loss=1.6873     acc=0.3721      1682.2 examples/second
[Step=3650]     Loss=1.6767     acc=0.3749      2294.0 examples/second
[Step=3700]     Loss=1.6717     acc=0.3776      2280.1 examples/second
Test Loss=1.7121, Test acc=0.3528

Epoch: 19
[Step=3750]     Loss=1.6440     acc=0.3839      1038.3 examples/second
[Step=3800]     Loss=1.6481     acc=0.3830      1688.8 examples/second
[Step=3850]     Loss=1.6574     acc=0.3801      2215.2 examples/second
[Step=3900]     Loss=1.6502     acc=0.3848      2313.3 examples/second
Test Loss=1.6763, Test acc=0.3778
Saving...
Files already downloaded and verified
Test Loss=1.6763, Test accuracy=0.3778
```

| Nbits | Test Accuracy | Finetuned Accuracy |
|---|---|---|
| 4 | 0.8722 | 0.9031 |
| 3 | 0.6331 | 0.8846 |
| 2 | 0.1000 | 0.3778 |

When we observe the accuracies, finetuning has improved the model performance, we can see that the test accuracies have increased. The performance of finetuned test accuracies without pruning is still higher, this can be because of the sparsity introduced in the model.

**Question (e):**

```
Nbits = 6 #Change this value to finish (b) and (c)


net = ResNetCIFAR(num_layers=20, Nbits=Nbits,symmetric=True)
net = net.to(device)
net.load_state_dict(torch.load("pretrained_model.pt"))
test(net)
```

```
Files already downloaded and verified
Test Loss=0.3276, Test accuracy=0.9124
```

```
Nbits = 5 #Change this value to finish (b) and (c)


net = ResNetCIFAR(num_layers=20, Nbits=Nbits,symmetric=True)
net = net.to(device)
net.load_state_dict(torch.load("pretrained_model.pt"))
test(net)
```

```
Files already downloaded and verified
Test Loss=0.3520, Test accuracy=0.9083
```

```
Nbits = 4 #Change this value to finish (b) and (c)


net = ResNetCIFAR(num_layers=20, Nbits=Nbits,symmetric=True)
net = net.to(device)
net.load_state_dict(torch.load("pretrained_model.pt"))
test(net)
```

```
Files already downloaded and verified
Test Loss=0.4227, Test accuracy=0.8875
```

```
Nbits = 3#Change this value to finish (b) and (c)


net = ResNetCIFAR(num_layers=20, Nbits=Nbits,symmetric=True)
net = net.to(device)
net.load_state_dict(torch.load("pretrained_model.pt"))
test(net)
```

```
Files already downloaded and verified
Test Loss=2.3739, Test accuracy=0.5185
```

```
Nbits = 2 #Change this value to finish (b) and (c)


net = ResNetCIFAR(num_layers=20, Nbits=Nbits,symmetric=True)
net = net.to(device)
net.load_state_dict(torch.load("pretrained_model.pt"))
test(net)
```

```
Files already downloaded and verified
Test Loss=42.7781, Test accuracy=0.1000
```

**Comparing the performance between Symmetric and Asymmetric Quantization:**

**Symmetric quantization:**

| Nbits | Test Accuracy |
|---|---|
| 6 | 0.9124 |
| 5 | 0.9083 |
| 4 | 0.8875 |
| 3 | 0.5185 |
| 2 | 0.1000 |

**Asymmetric quantization:**

| Nbits | Test accuracy |
|---|---|
| 6 | 0.9145 |
| 5 | 0.9112 |
| 4 | 0.8972 |
| 3 | 0.7662 |
| 2 | 0.0899 |

From the above two tables we can conclude that Asymmetric quantization performs better than symmetric quantization. Symmetric quantization tends to have lower accuracy because it discards information about the sign of the values, on the other hand asymmetric quantization preserves the sign information leading to higher accuracy.