ECE 661: Homework #4
Pruning and Fixed-point Quantization

Hai Li

ECE Department, Duke University — Spring 2025

Objectives

Homework #4 covers the contents of Lectures 14 ~ 17. This assignment starts with conceptual questions on model pruning and quantization techniques, followed by lab questions evaluating the effectiveness of different sparse optimization methods on linear models, iterative pruning of a pretrained CNN model, and training fixed-point quantized CNNs.

Warning: You are asked to complete the assignment independently.

This lab has a total of **100** points. You must submit your report in PDF format and your original codes for the lab questions through **Gradescope** before **11:55:00pm, March 31**. You need to submit **three individual files** including:
(1) *a self-contained report in PDF format* that provides answers to all the conceptual questions and clearly demonstrates all your lab codes, results and observations (figures and explanations);
(2) *a single notebook file* used to produce the results for Lab1;
(3) *the completed notebook file* for Lab 2 and Lab 3.

1 True/False Questions (15 pts)

For each question, please provide a short explanation to support your judgment.

Problem 1.1 (3 pts) Using sparsity-inducing regularizers like L-1 in DNN optimization with SGD guarantees exact zero values in weight elements, making further pruning unnecessary.
False. L-1 regularization with SGD pushes weights toward zero but doesn't guarantee exact zeros, so pruning is still necessary.

Problem 1.2 (3 pts) Applying weight pruning to a quantized model typically results in decreased model accuracy.
True. When you apply both weight pruning and quantization to a model, you typically lose some accuracy. Combining these two compression techniques usually creates more accuracy loss than using just one technique alone.

Problem 1.3 (3 pts) Pruning DNNs leads to reduced storage costs and lower inference latency, regardless of the distribution of remaining weights.
False. Pruning can reduce storage costs and latency, but it depends on how the remaining weights are distributed. If the remaining weights are not efficiently distributed, the benefits may not be as significant.

Problem 1.4 (3 pts) Compared to the standard Lasso regularizer, Group Lasso can lead to unstructured sparsity of DNNs, which is more hardware-friendly.
False. Group Lasso tends to create **structured sparsity** by grouping related features together, not unstructured sparsity. This structured sparsity can be more hardware-friendly.

Problem 1.5 (3 pts) The accuracy of a binarized DNN can be preserved by keeping the first and last layers at full precision and quantizing only the weights to a scaled representation.
True. When binarizing neural networks, keeping the first and last layers at full precision while quantizing the middle layers helps maintain accuracy. This is because the first layer processes the input data directly and the last layer produces the final output, making them more sensitive to precision loss.

2 Lab 1: Sparse optimization of linear models (30 pts)

By now you have seen multiple ways to induce a sparse solution in the optimization process. This problem will provide you some examples under linear regression setting so that you can compare the effectiveness of different methods. For this problem, consider the case where we are trying to find a sparse weight W that can minimize $L = \sum_i (X_i W - y_i)^2$. Specifically, we have $X_i \in \mathbb{R}^{1\times 5}$, $W \in \mathbb{R}^{5\times 1}$ and $||W||_0 \leq 2$.

For Problem (a) - (f), consider the case where we have 3 data points: $(X_1 = [-2, 2, 1, -1, -1], y_1 = 5)$; $(X_2 = [-2, 1, -2, 0, 1], y_2 = 1)$; $(X_3 = [1, 0, -2, 2, -1], y_3 = 1)$. For stability the objective L should be minimized through full-batch gradient descent, with initial weight W^0 set to $[0; 0; 0; 0; 0]$ and use learning rate $\mu = 0.02$ throughout the process. Please run gradient descent for 200 steps for all the following problems. **For $log(L)$ plot, please use matplotlib.pyplot.yscale('log')**

Lab 1 (30 points)

(a) (4 pts) Theoretical analysis: with learning rate μ, suppose the weight you have after step k is W^k, derive the symbolic formulation of weight W^{k+1} after step k+1 of full-batch gradient descent with $X_i, y_i, i \in \{1, 2, 3\}$. (Hint: note the loss L we have is defined differently from standard MSE loss.)

(b) (3 pts) In Python, directly minimize the objective L without any sparsity-inducing regularization/constraint. Plot the value of $log(L)$ vs. #steps throughout the training, and use another figure to plot how the value of each element in W is changing throughout the training. From your result, is W converging to an optimal solution? Is W converging to a sparse solution?

(c) (6 pts) Since we have the knowledge that the ground-truth weight should have $||W||_0 \leq 2$, we can apply **projected gradient descent** to enforce this sparse constraint. Redo the optimization process in (b), this time prune the elements in W after every gradient descent step to ensure $||W^l||_0 \leq 2$. Plot the value of $log(L)$ throughout the training, and use another figure to plot the value of each element in W in each step. From your result, is W converging to an optimal solution? Is W converging to a sparse solution?

(d) (5 pts) In this problem we apply ℓ_1 regularization to induce the sparse solution. The minimization objective therefore changes to $L + \lambda||W||_1$. Please use full-batch gradient descent to minimize this objective, with $\lambda = \{0.2, 0.5, 1.0, 2.0\}$ respectively. For each case, plot the value of $log(L)$ throughout the training, and use another figure to plot the value of each element in W in each step. From your result, comment on the convergence performance under different λ.

(e) (6 pts) Here we optimize the same objective as in (d), this time using **proximal gradient update**. Recall that the proximal operator of the ℓ_1 regularizer is the soft thresholding function. Set the threshold in the soft thresholding function to $\{0.004, 0.01, 0.02, 0.04\}$ respectively. Plot the value of $log(L)$ throughout the training, and use another figure to plot the value of each element in W in each step. Compare the convergence performance with the results in (d). (Hint: Optimizing $L + \lambda||W||_1$ using gradient descent with learning rate μ should correspond to proximal gradient update with threshold $\mu\lambda$)

(f) (6 pts) Trimmed ℓ_1 ($T\ell_1$) regularizer is proposed to solve the "bias" problem of ℓ_1. For simplicity you may implement the $T\ell_1$ regularizer as applying a ℓ_1 regularization with strength λ on the 3 elements of W **with the smallest absolute value**, with no penalty on other elements. Minimize $L + \lambda T\ell_1(W)$ **using proximal gradient update** with $\lambda = \{1.0, 2.0, 5.0, 10.0\}$ (correspond the soft thresholding threshold $\{0.02, 0.04, 0.1, 0.2\}$). Plot the value of $log(L)$ throughout the training, and use another figure to plot the value of each element in W in each step. Comment on the convergence comparison of the Trimmed ℓ_1 and the ℓ_1. Also compare the behavior of the early steps (e.g. first 20) between the Trimmed ℓ_1 and the iterative pruning.

a) Our loss function that we want to minimize is:

$$L = \sum_{i=1}^{3} (X_i W - y_i)^2$$

Also, the gradient descent rule is:

$$W^{k+1} = W^k - \mu \nabla L(W^k)$$

So we want to find $\nabla L(W^k)$

$$\nabla L(W^k) = 2 \sum_{i=1}^{3} X_i^T (X_i W - y_i)$$

So then we have

$$W^{k+1} = W^k - 2\mu \sum_{i=1}^{3} X_i^T (X_i W - y_i)$$

Or

$$W^{k+1} = W^k - 2\mu X_1^T (X_1 W - y_1) - 2\mu X_2^T (X_2 W - y_2) - 2\mu X_3^T (X_3 W - y_3)$$

b)

```
import numpy as np
import matplotlib.pyplot as plt
```

✓ 0.3s

```
X = np.array([[-2, 2, 1, -1, -1],
              [-2, 1, -2, 0, 1],
              [1, 0, -2, 2, -1]])

y = np.array([5, 1, 1])
lr = 0.02
steps = 200
W = np.zeros(5)

loss_history = []
weight_history = []

for step in range(steps):

    loss = np.sum((np.dot(X, W) - y)**2)
    loss_history.append(loss)

    weight_history.append(W.copy())

    W = W - 2 * lr * np.dot(X.T, (np.dot(X, W) - y))

weight_history = np.array(weight_history)
```

✓ 0.0s

```
plt.figure(figsize=(12, 5))

plt.subplot(1, 2, 1)
plt.plot(range(steps), loss_history)
plt.yscale('log')
plt.xlabel('Steps')
plt.ylabel('log(Loss)')
plt.title('log(Loss) vs. Steps')

plt.subplot(1, 2, 2)
for i in range(5):
    plt.plot(range(steps), weight_history[:, i], label=f'W[{i}]')
plt.xlabel('Steps')
plt.ylabel('Weight Value')
plt.title('Evolution of W Elements')
plt.legend()

plt.tight_layout()
plt.show()
```

```
weight_history[-1]
```

✓ 0.0s

```
array([-0.63143255,  1.26564673,  0.09040334,  0.23226704, -1.34770515])
```

- W is converging to an optimal solution. The log(Loss) decreases constantly over 200 iterations, indicating exponential decay in loss.
- W is not converging to a sparse solution. None of the elements of W approach zero, as seen in the weight evolution plot, so the solution is not sparse.

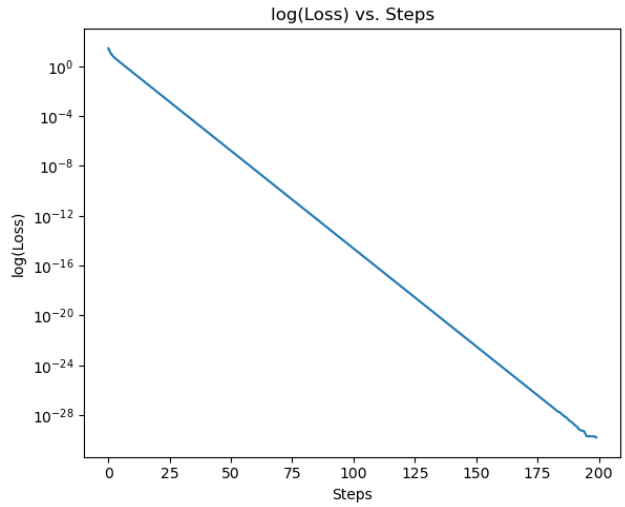
c)

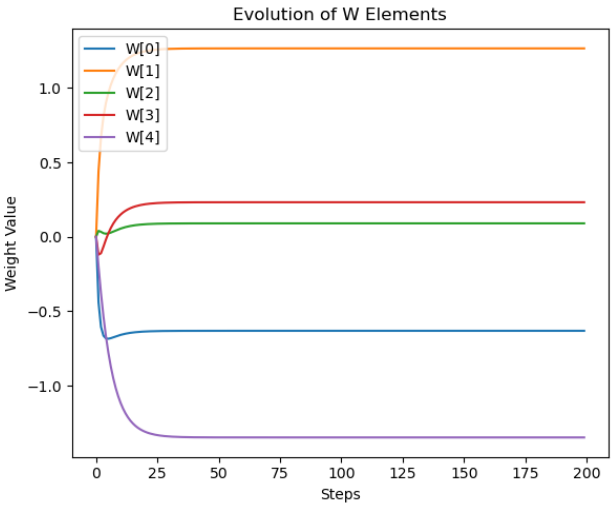
```
W = np.zeros(5)
loss_history = []
weight_history = []

for step in range(steps):
    loss = np.sum((np.dot(X, W) - y)**2)
    loss_history.append(loss)
    weight_history.append(W.copy())

    gradient = 2 * np.dot(X.T, (np.dot(X, W) - y))
    W = W - lr * gradient

    if np.count_nonzero(W) > 2:
        indices = np.argsort(np.abs(W))
        W[indices[:-2]] = 0

weight_history = np.array(weight_history)
```

✓ 0.0s

```
plt.figure(figsize=(12, 5))

plt.subplot(1, 2, 1)

plt.plot(range(steps), loss_history)
plt.yscale('log')
plt.xlabel('Steps')
plt.ylabel('log(Loss)')
plt.title('log(Loss) vs. Steps')

plt.subplot(1, 2, 2)
for i in range(5):
    plt.plot(range(steps), weight_history[:, i], label=f'W[{i}]')
plt.xlabel('Steps')
plt.ylabel('Weight Value')
plt.title('Evolution of W Elements')
plt.legend()

plt.tight_layout()
plt.show()
```

✓ 0.2s






```
weight_history[-1]
```

[7] ✓ 0.0s

```
array([ 0.,  2.,  0.,  0., -1.])
```

After applying projected gradient descent to enforce the sparse constraint $||W||_0 \leq 2$ we can see in the plots that:

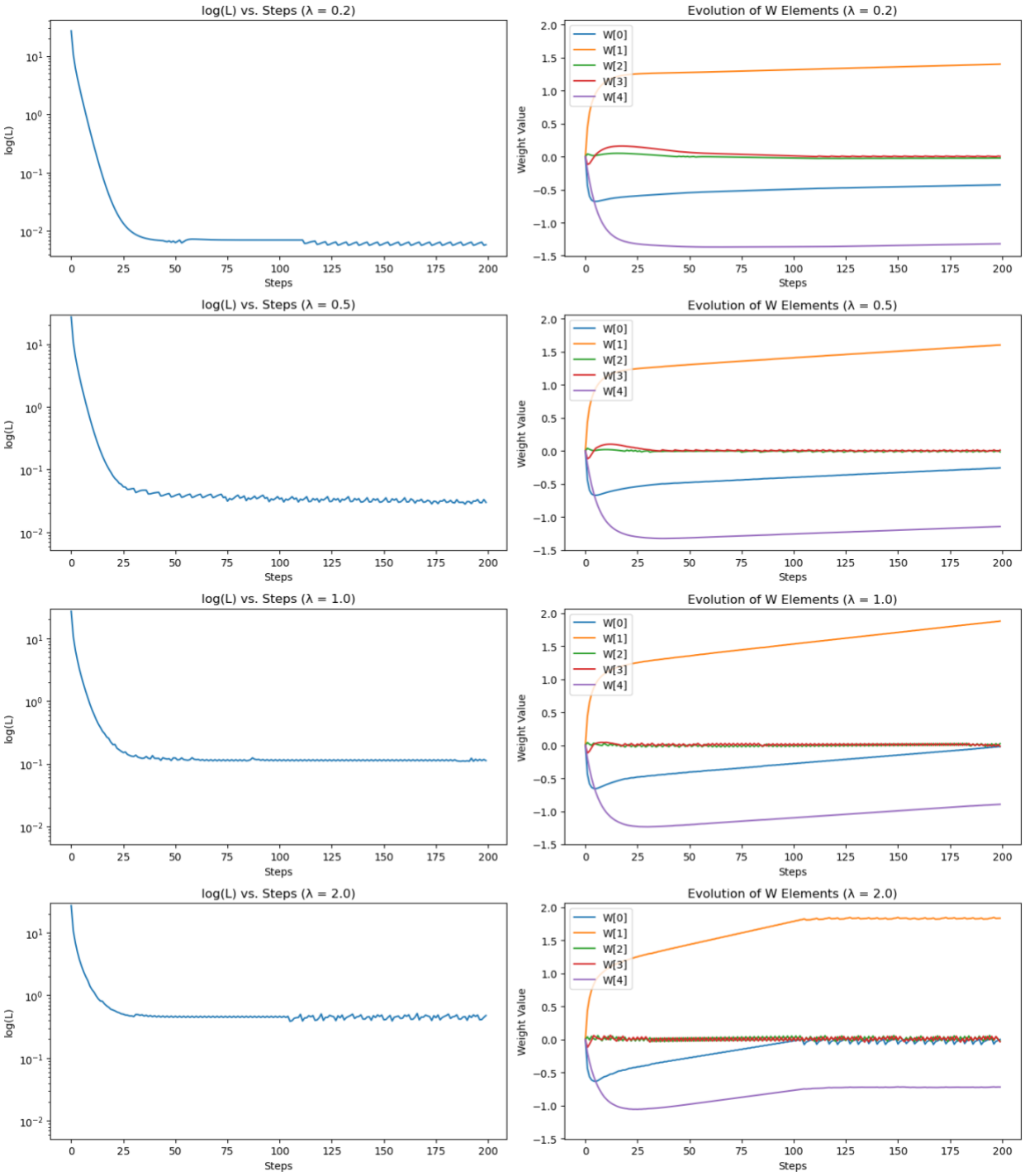
- W is converging to an optimal solution. The plot shows that the loss function steadily decreases.
- The solution is sparse. The weight evolution plot demonstrates that only 2 weights have non-zero values at convergence (w1= 2, w4=-1), while the other three weights remain at zero. (w0, w2, w3)

d)

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

all_loss_history = []
all_weight_history = []
for lambda_val in lambda_values:
    W = np.zeros(5)
    loss_history = []
    weight_history = []

    for step in range(steps):
        predictions = X @ W
        residuals = predictions - y

        mse_loss = np.sum(residuals**2)
        l1_norm = lambda_val * np.sum(np.abs(W))
        loss = mse_loss + l1_norm

        loss_history.append(mse_loss)
        weight_history.append(W.copy())

        grad_mse = 2 * X.T @ residuals
        grad_l1 = lambda_val * np.sign(W)

        W = W - lr * (grad_mse + grad_l1)

    weight_history = np.array(weight_history)
    all_loss_history.append(loss_history)
    all_weight_history.append(weight_history)
```

✓ 0.0s

```python
min_loss = min([min(losses) for losses in all_loss_history])
max_loss = max([max(losses) for losses in all_loss_history])

min_weight = min([np.min(weights) for weights in all_weight_history])
max_weight = max([np.max(weights) for weights in all_weight_history])

fig, axs = plt.subplots(len(lambda_values), 2, figsize=(14, 4*len(lambda_values)))

for idx, lambda_val in enumerate(lambda_values):
    loss_history = all_loss_history[idx]
    weight_history = all_weight_history[idx]

    # log(L)
    axs[idx, 0].plot(range(steps), loss_history)
    axs[idx, 0].set_yscale('log')
    axs[idx, 0].set_xlabel('Steps')
    axs[idx, 0].set_ylabel('log(L)')
    axs[idx, 0].set_title(f'log(L) vs. Steps (λ = {lambda_val})')

    if idx > 0:
        axs[idx, 0].set_ylim([min_loss * 0.9, max_loss * 1.1])

    # Weight
    for i in range(weight_history.shape[1]):
        axs[idx, 1].plot(range(steps), weight_history[:, i], label=f'W[{i}]')
    axs[idx, 1].set_xlabel('Steps')
    axs[idx, 1].set_ylabel('Weight Value')
    axs[idx, 1].set_title(f'Evolution of W Elements (λ = {lambda_val})')
    axs[idx, 1].legend()

    axs[idx, 1].set_ylim([min_weight * 1.1, max_weight * 1.1])

plt.tight_layout()
plt.show()
```



- When λ increases, the regularization effect becomes stronger, resulting in sparser solutions. With λ = 0.2 or 0.5, 2 of the weights approach zero, while with λ = 1.0 and 2.0, 3 weights are pushed toward zero, making the solution sparser.

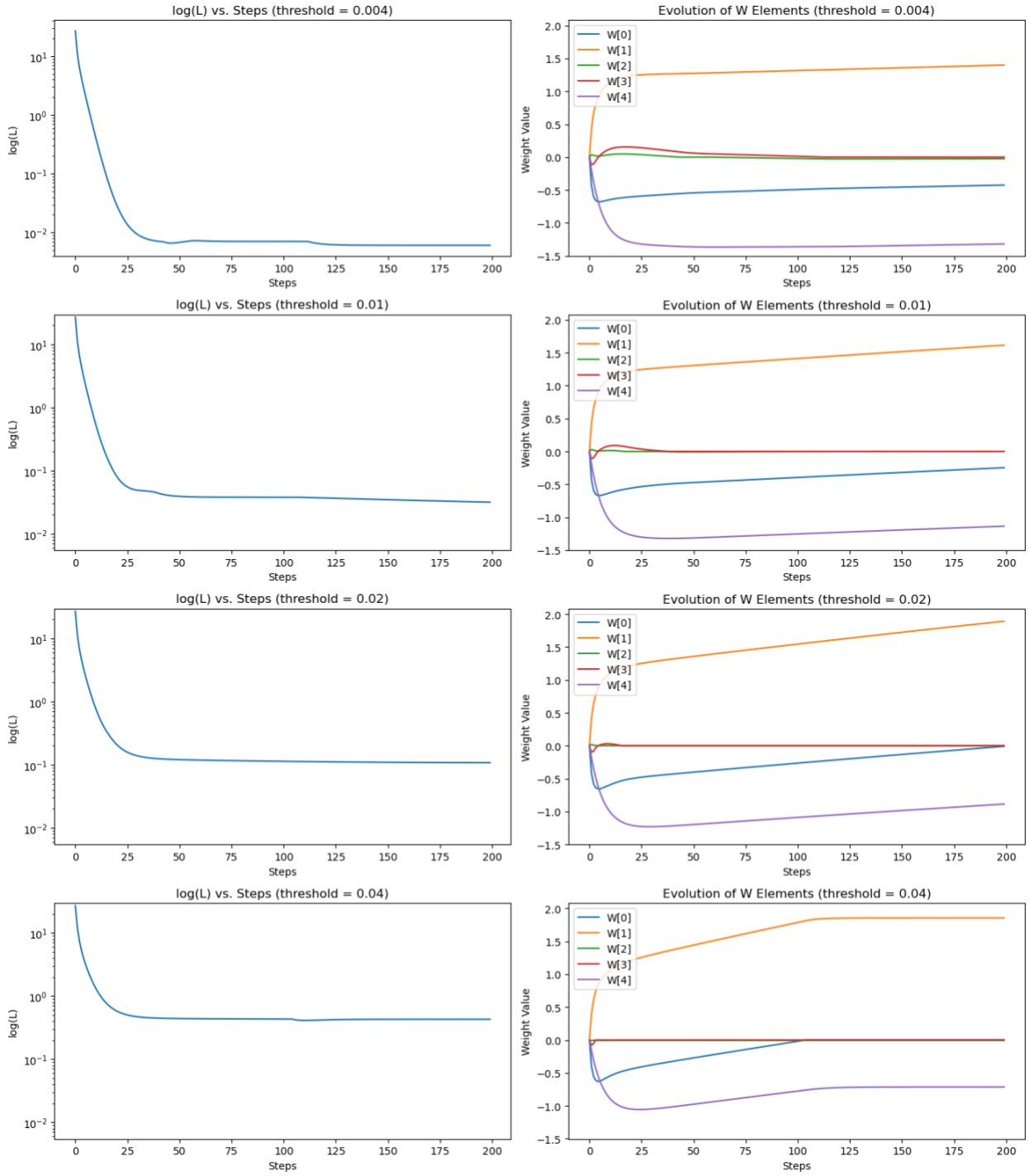
- Although all models converge in a similar number of iterations (around 50), higher λ values (like λ =2) produce more oscillations after initial convergence, indicating less stability in the optimization. Additionally, higher λ values result in a higher final loss, demonstrating the trade-off between sparsity and fitting accuracy.

- In general, using higher λ values produces a simpler model with fewer non-zero weights (in this case 2 instead of 3), but this comes at the cost of potentially less stable optimization and higher error in fitting the data.

e)

```
threshold_values = [0.004, 0.01, 0.02, 0.04]

all_loss_history = []
all_weight_history = []

for threshold in threshold_values:
    W = np.zeros(5)
    loss_history = []
    weight_history = []

    for step in range(steps):
        predictions = X @ W
        residuals = predictions - y

        mse_loss = np.sum(residuals**2)
        loss_history.append(mse_loss)
        weight_history.append(W.copy())

        grad_mse = 2 * X.T @ residuals

        W_temp = W - lr * grad_mse

        W = np.sign(W_temp) * np.maximum(np.abs(W_temp) - threshold, 0)

    weight_history = np.array(weight_history)
    all_loss_history.append(loss_history)
    all_weight_history.append(weight_history)

min_loss = min([min(losses) for losses in all_loss_history])
max_loss = max([max(losses) for losses in all_loss_history])

min_weight = min([np.min(weights) for weights in all_weight_history])
max_weight = max([np.max(weights) for weights in all_weight_history])
```

```
fig, axs = plt.subplots(len(threshold_values), 2, figsize=(14, 4*len(threshold_values)))

for idx, threshold in enumerate(threshold_values):
    loss_history = all_loss_history[idx]
    weight_history = all_weight_history[idx]

    axs[idx, 0].plot(range(steps), loss_history)
    axs[idx, 0].set_yscale('log')
    axs[idx, 0].set_xlabel('Steps')
    axs[idx, 0].set_ylabel('log(L)')
    axs[idx, 0].set_title(f'log(L) vs. Steps (threshold = {threshold})')

    if idx > 0:
        axs[idx, 0].set_ylim([min_loss * 0.9, max_loss * 1.1])

    for i in range(weight_history.shape[1]):
        axs[idx, 1].plot(range(steps), weight_history[:, i], label=f'W[{i}]')
    axs[idx, 1].set_xlabel('Steps')
    axs[idx, 1].set_ylabel('Weight Value')
    axs[idx, 1].set_title(f'Evolution of W Elements (threshold = {threshold})')
    axs[idx, 1].legend()

    axs[idx, 1].set_ylim([min_weight * 1.1, max_weight * 1.1])

plt.tight_layout()
plt.show()
0.9s
```



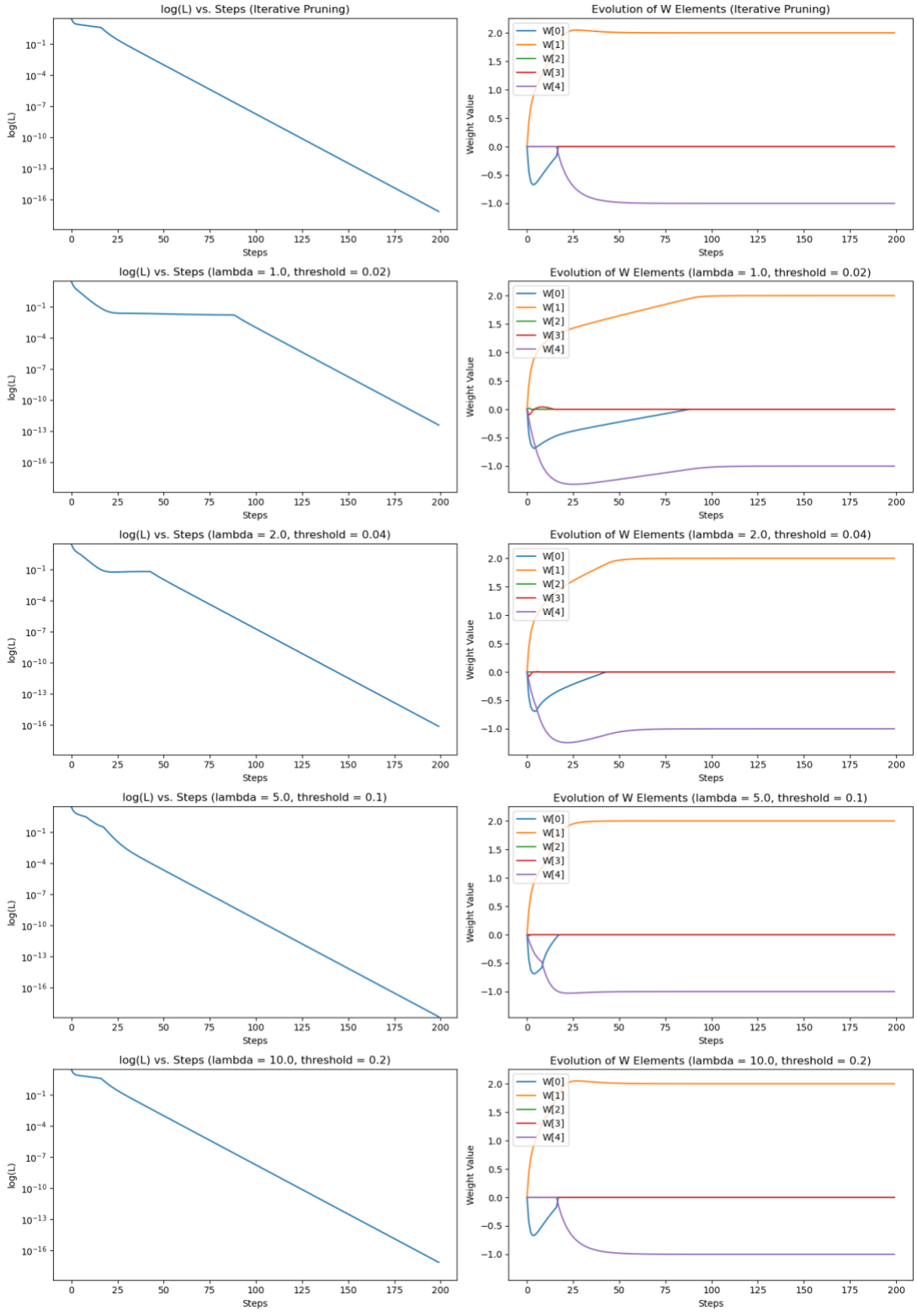
The proximal gradient descent method with soft thresholding converges well for all threshold values. Compared to L1 regularization from part (d):

- With smaller thresholds (0.004, 0.01), the model has 3 non-zero weights, similar to L1 regularization with small λ. These smaller thresholds result in a better fit to the data (lower final loss), but with less sparsity.
- As the threshold increases (0.02, 0.04), the model becomes sparser, with only 2 non-zero weights, similar to L1 regularization with higher λ.
- The proximal gradient method shows more stable convergence than standard gradient descent with L1, with fewer oscillations, especially at higher thresholds.

f)

```
lambda_values = [1.0, 2.0, 5.0, 10.0]
threshold_values = [lr * lam for lam in lambda_values]

all_loss_history = []
all_weight_history = []

W_pruning = np.zeros(5)
pruning_loss_history = []
pruning_weight_history = []

for i in range(steps):
    predictions = X @ W_pruning
    residuals = predictions - y

    mse_loss = np.sum(residuals**2)
    pruning_loss_history.append(mse_loss)
    pruning_weight_history.append(W_pruning.copy())

    grad_mse = 2 * X.T @ residuals
    W_pruning = W_pruning - lr * grad_mse

    indices = np.argsort(np.abs(W_pruning))
    W_pruning[indices[:-2]] = 0

for threshold in threshold_values:
    W = np.zeros(5)
    loss_history = []
    weight_history = []

    for step in range(steps):
        predictions = X @ W
        residuals = predictions - y

        mse_loss = np.sum(residuals**2)
        loss_history.append(mse_loss)
        weight_history.append(W.copy())

        grad_mse = 2 * X.T @ residuals

        W_temp = W - lr * grad_mse

        k = 2
        largest_indices = np.argsort(np.abs(W_temp))[-k:]
        smallest_indices = np.argsort(np.abs(W_temp))[:-k]

        W = W_temp.copy()
        W[smallest_indices] = np.sign(W_temp[smallest_indices]) * np.maximum(
            np.abs(W_temp[smallest_indices]) - threshold, 0)

    weight_history = np.array(weight_history)
    all_loss_history.append(loss_history)
    all_weight_history.append(weight_history)
```

```
all_losses = [pruning_loss_history] + all_loss_history
all_weights = [pruning_weight_history] + all_weight_history

min_loss = min([min(losses) for losses in all_losses])
max_loss = max([max(losses) for losses in all_losses])

weight_arrays = [np.array(pruning_weight_history)]
for weights in all_weight_history:
    weight_arrays.append(np.array(weights))

min_weight = min([np.min(weights) for weights in weight_arrays])
max_weight = max([np.max(weights) for weights in weight_arrays])

min_weight = min_weight - 0.1 * abs(min_weight)
max_weight = max_weight + 0.1 * abs(max_weight)

fig, axs = plt.subplots(len(threshold_values) + 1, 2, figsize=(14, 4*(len(threshold_values) + 1)))

axs[0, 0].plot(range(steps), pruning_loss_history)
axs[0, 0].set_yscale('log')
axs[0, 0].set_xlabel('Steps')
axs[0, 0].set_ylabel('log(L)')
axs[0, 0].set_title('log(L) vs. Steps (Iterative Pruning)')
axs[0, 0].set_ylim([min_loss * 0.9, max_loss * 1.1])

pruning_weight_history = np.array(pruning_weight_history)
for i in range(pruning_weight_history.shape[1]):
    axs[0, 1].plot(range(steps), pruning_weight_history[:, i], label=f'W[{i}]')
axs[0, 1].set_xlabel('Steps')
axs[0, 1].set_ylabel('Weight Value')
axs[0, 1].set_title('Evolution of W Elements (Iterative Pruning)')
axs[0, 1].legend()
axs[0, 1].set_ylim([min_weight, max_weight])

for idx, (lam, threshold) in enumerate(zip(lambda_values, threshold_values)):
    loss_history = all_loss_history[idx]
    weight_history = all_weight_history[idx]

    axs[idx + 1, 0].plot(range(steps), loss_history)
    axs[idx + 1, 0].set_yscale('log')
    axs[idx + 1, 0].set_xlabel('Steps')
    axs[idx + 1, 0].set_ylabel('log(L)')
    axs[idx + 1, 0].set_title(f'log(L) vs. Steps (lambda = {lam}, threshold = {threshold})')
    axs[idx + 1, 0].set_ylim([min_loss * 0.9, max_loss * 1.1])

    for i in range(weight_history.shape[1]):
        axs[idx + 1, 1].plot(range(steps), weight_history[:, i], label=f'W[{i}]')
    axs[idx + 1, 1].set_xlabel('Steps')
    axs[idx + 1, 1].set_ylabel('Weight Value')
    axs[idx + 1, 1].set_title(f'Evolution of W Elements (lambda = {lam}, threshold = {threshold})')
    axs[idx + 1, 1].legend()
    axs[idx + 1, 1].set_ylim([min_weight, max_weight])

plt.tight_layout()
plt.show()
```



Trimmed ℓ1 regularization offers key benefits over standard ℓ1:

- The loss decreases faster and reaches much lower values (10^-16 to 10^-18), suggesting better optimization. All λ values consistently lead to a sparse solution, with 2 non-zero weights. Trimmed ℓ1 only penalizes small weights, allowing larger weights.
- In the first 20 iterations, both methods quickly focus on W[1] and W[4], with iterative pruning showing more aggressive changes and some oscillations. Trimmed ℓ1 converges more smoothly, especially with higher λ values.
- The advantage of Trimmed ℓ1 is its continuous optimization, which keeps important features throughout training, while iterative pruning makes hard decisions, leading to instability. It provides good sparsity and low loss, addressing the bias issue in standard ℓ1.

3 Lab 2: Pruning ResNet-20 model (25 pts)

ResNet-20 is a popular convolutional neural network (CNN) architecture for image classification. Compared to early CNN designs such as VGG-16, ResNet-20 is much more compact. Thus, conducing the model compression on ResNet-20 is more challenging.

This lab explores the element-wise pruning of ResNet-20 model on CIFAR-10 dataset. We will observe the difference between single step pruning and iterative pruning, plus exploring different ways of setting pruning threshold. Everything you need for this lab can be found in `HW4.zip`.

Lab 2 (25 points)

(a) (2 pts) In `hw4.ipynb`, run through the first three code block, report the accuracy of the floating-point pretrained model.

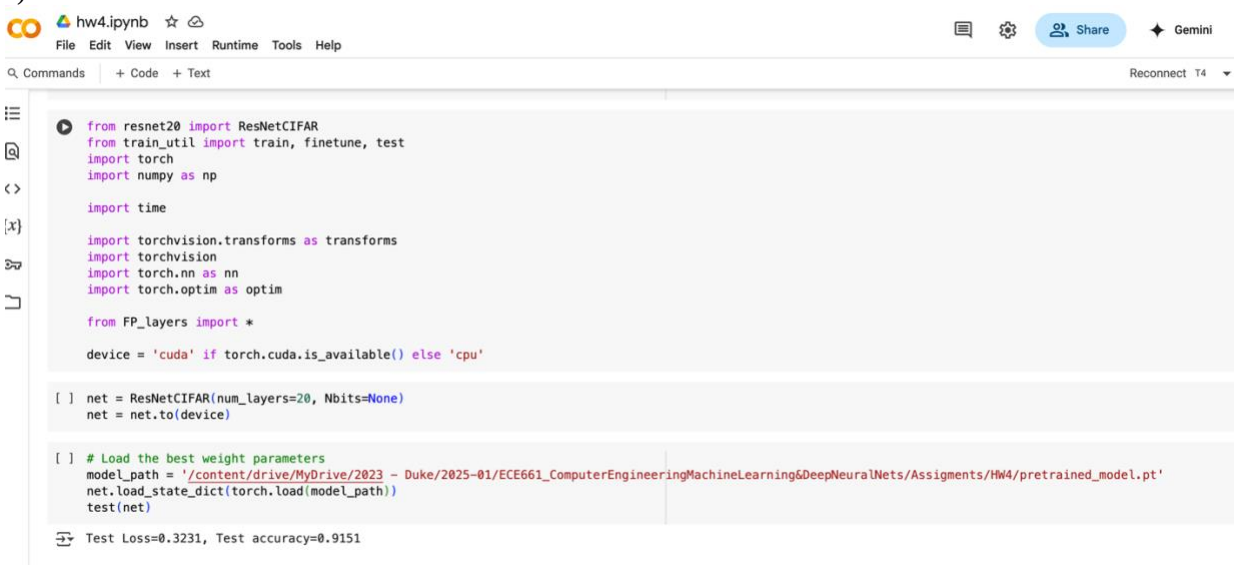
(b) (6 pts) Complete the implementation of *pruning by percentage* function in the notebook. Here we determines the pruning threshold in each DNN layer by the **'q-th percentile'** value in the absolute value of layer's weight element. Use the next block to call your implemented *pruning by percentage*. Try pruning percentage q = 0.2, 0.4, 0.6, 0.7, 0.8. Report the test accuracy q. **(Hint:** You need to reload the full model checkpoint before applying the prune function with a different q).

(c) (6 pts) Fill in the `finetune_after_prune` function for pruned model finetuning. Make sure the pruned away elements in previous step are kept as 0 throughout the finetuning process. Finetune the pruned model with q=0.8 for 20 epochs with the provided training pipeline. Report the best accuracy achieved during finetuning. Finish the code for sparsity evaluation to check if the finetuned model preserves the sparsity.

(d) (5 pts) Implement iterative pruning. Instead of applying single step pruning before finetuning, try iteratively increasing the sparsity of the model before each epoch of finetuning. Linearly increase the pruning percentage for 10 epochs until reaching 80% in the final epoch (prune $(8 \times e)\%$ before epoch e) then continue finetune for 10 epochs. Pruned weights can be recovered during the iterative pruning process before the final pruning step. Compare the performance with (c).

(e) (6 pts) Perform magnitude-based global iterative pruning. Previously we set the pruning threshold of each layer following the weight distribution of the layer and prune all layers to the same sparsity. This will constrain the flexibility in the final sparsity pattern across layers. In this question, Fill in the `global_prune_by_percentage` function to perform a global ranking of the weight magnitude from all the layers, and determine a single pruning threshold by percentage for all the layers. Repeat iterative pruning to 80% sparsity, and report final accuracy and the percentage of zeros in each layer.

a)



Test Loss=0.3231, Test accuracy=0.9151

b)

Lab2 (b) 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
    weight = layer.weight.data.cpu().numpy()

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

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

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

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

    return threshold
```

```
[9] pretrained_model_path = '/content/drive/MyDrive/2023 - Duke/2025-01/ECE661_ComputerEngineeringMachineLearning&DeepNeuralNets/Assigments/HW4/pretrained_model.pt'
```

q = 20%

```
net.load_state_dict(torch.load(pretrained_model_path))
q=20

for name,layer in net.named_modules():
    if (isinstance(layer, nn.Conv2d) or isinstance(layer, nn.Linear)) and 'id_mapping' not in name:
        # change q value
        prune_by_percentage(layer, q)

        # Optional: Check the sparsity you achieve in each layer
        # 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: 0.2013888888888889
Sparsity of body_op.0.conv1.0.conv: 0.20008680555555555
Sparsity of body_op.0.conv2.0.conv: 0.20008680555555555
Sparsity of body_op.1.conv1.0.conv: 0.20008680555555555
Sparsity of body_op.1.conv2.0.conv: 0.20008680555555555
Sparsity of body_op.2.conv1.0.conv: 0.20008680555555555
Sparsity of body_op.2.conv2.0.conv: 0.20008680555555555
Sparsity of body_op.3.conv1.0.conv: 0.20008680555555555
Sparsity of body_op.3.conv2.0.conv: 0.20008680555555555
Sparsity of body_op.4.conv1.0.conv: 0.20008680555555555
Sparsity of body_op.4.conv2.0.conv: 0.20008680555555555
Sparsity of body_op.5.conv1.0.conv: 0.20008680555555555
Sparsity of body_op.5.conv2.0.conv: 0.20008680555555555
Sparsity of body_op.6.conv1.0.conv: 0.20003255208333334
Sparsity of body_op.6.conv2.0.conv: 0.2000054253472222
Sparsity of body_op.7.conv1.0.conv: 0.2000054253472222
Sparsity of body_op.7.conv2.0.conv: 0.2000054253472222
Sparsity of body_op.8.conv1.0.conv: 0.2000054253472222
Sparsity of body_op.8.conv2.0.conv: 0.2000054253472222
Sparsity of final_fc.linear: 0.2
Test Loss=0.3449, Test accuracy=0.9086
```

q = 40%

```
[11] net.load_state_dict(torch.load(pretrained_model_path))
     q=40

     for name,layer in net.named_modules():
         if (isinstance(layer, nn.Conv2d) or isinstance(layer, nn.Linear)) and 'id_mapping' not in name:
             # change q value
             prune_by_percentage(layer, q)

             # Optional: Check the sparsity you achieve in each layer
             # 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: 0.40046296296296297
Sparsity of body_op.0.conv1.0.conv: 0.4001736111111111
Sparsity of body_op.0.conv2.0.conv: 0.4001736111111111
Sparsity of body_op.1.conv1.0.conv: 0.4001736111111111
Sparsity of body_op.1.conv2.0.conv: 0.4001736111111111
Sparsity of body_op.2.conv1.0.conv: 0.4001736111111111
Sparsity of body_op.2.conv2.0.conv: 0.4001736111111111
Sparsity of body_op.3.conv1.0.conv: 0.3999565972222222
Sparsity of body_op.3.conv2.0.conv: 0.4000651041666667
Sparsity of body_op.4.conv1.0.conv: 0.4000651041666667
Sparsity of body_op.4.conv2.0.conv: 0.4000651041666667
Sparsity of body_op.5.conv1.0.conv: 0.4000651041666667
Sparsity of body_op.5.conv2.0.conv: 0.4000651041666667
Sparsity of body_op.6.conv1.0.conv: 0.4000108506944444
Sparsity of body_op.6.conv2.0.conv: 0.4000108506944444
Sparsity of body_op.7.conv1.0.conv: 0.4000108506944444
Sparsity of body_op.7.conv2.0.conv: 0.4000108506944444
Sparsity of body_op.8.conv1.0.conv: 0.4000108506944444
Sparsity of body_op.8.conv2.0.conv: 0.4000108506944444
Sparsity of final_fc.linear: 0.4
Test Loss=0.4311, Test accuracy=0.8874
```

q = 60%

```
net.load_state_dict(torch.load(pretrained_model_path))
q=60

for name,layer in net.named_modules():
    if (isinstance(layer, nn.Conv2d) or isinstance(layer, nn.Linear)) and 'id_mapping' not in name:
        # change q value
        prune_by_percentage(layer, q)

        # Optional: Check the sparsity you achieve in each layer
        # 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)
```

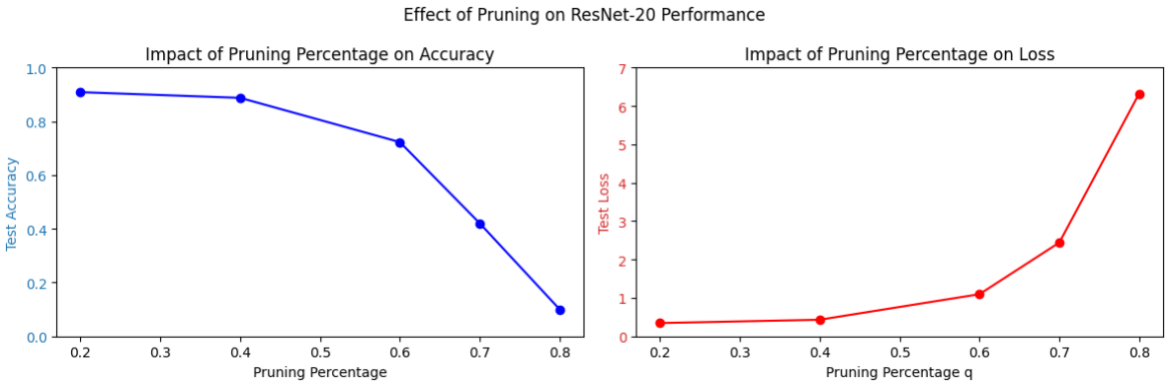
```
Sparsity of head_conv.0.conv: 0.5995370370370371
Sparsity of body_op.0.conv1.0.conv: 0.5998263888888888
Sparsity of body_op.0.conv2.0.conv: 0.5998263888888888
Sparsity of body_op.1.conv1.0.conv: 0.5998263888888888
Sparsity of body_op.1.conv2.0.conv: 0.5998263888888888
Sparsity of body_op.2.conv1.0.conv: 0.5998263888888888
Sparsity of body_op.2.conv2.0.conv: 0.5998263888888888
Sparsity of body_op.3.conv1.0.conv: 0.6000434027777778
Sparsity of body_op.3.conv2.0.conv: 0.6000434027777778
Sparsity of body_op.4.conv1.0.conv: 0.6000434027777778
Sparsity of body_op.4.conv2.0.conv: 0.6000434027777778
Sparsity of body_op.5.conv1.0.conv: 0.6000434027777778
Sparsity of body_op.5.conv2.0.conv: 0.6000434027777778
Sparsity of body_op.6.conv1.0.conv: 0.5999891493055556
Sparsity of body_op.6.conv2.0.conv: 0.5999891493055556
Sparsity of body_op.7.conv1.0.conv: 0.5999891493055556
Sparsity of body_op.7.conv2.0.conv: 0.5999891493055556
Sparsity of body_op.8.conv1.0.conv: 0.5999891493055556
Sparsity of body_op.8.conv2.0.conv: 0.5999891493055556
Sparsity of final_fc.linear: 0.6
Test Loss=1.0955, Test accuracy=0.7226
```

q = 80%

```
[14] net.load_state_dict(torch.load(pretrained_model_path))
     q=80

     for name,layer in net.named_modules():
         if (isinstance(layer, nn.Conv2d) or isinstance(layer, nn.Linear)) and 'id_mapping' not in name:
             # change q value
             prune_by_percentage(layer, q)

             # Optional: Check the sparsity you achieve in each layer
             # 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: 0.7986111111111112
Sparsity of body_op.0.conv1.0.conv: 0.7999131944444444
Sparsity of body_op.0.conv2.0.conv: 0.7999131944444444
Sparsity of body_op.1.conv1.0.conv: 0.7999131944444444
Sparsity of body_op.1.conv2.0.conv: 0.7999131944444444
Sparsity of body_op.2.conv1.0.conv: 0.7999131944444444
Sparsity of body_op.2.conv2.0.conv: 0.7999131944444444
Sparsity of body_op.3.conv1.0.conv: 0.7999131944444444
Sparsity of body_op.3.conv2.0.conv: 0.8000217013888888
Sparsity of body_op.4.conv1.0.conv: 0.8000217013888888
Sparsity of body_op.4.conv2.0.conv: 0.8000217013888888
Sparsity of body_op.5.conv1.0.conv: 0.8000217013888888
Sparsity of body_op.5.conv2.0.conv: 0.8000217013888888
Sparsity of body_op.6.conv1.0.conv: 0.7999674479166666
Sparsity of body_op.6.conv2.0.conv: 0.7999945746527778
Sparsity of body_op.7.conv1.0.conv: 0.7999945746527778
Sparsity of body_op.7.conv2.0.conv: 0.7999945746527778
Sparsity of body_op.8.conv1.0.conv: 0.7999945746527778
Sparsity of body_op.8.conv2.0.conv: 0.7999945746527778
Sparsity of final_fc.linear: 0.8
Test Loss=6.3235, Test accuracy=0.1003
```

```
[7] import matplotlib.pyplot as plt

    sparsity = [0.2, 0.4, 0.6, 0.7, 0.8]
    accuracy = [0.9086, 0.8874, 0.7226, 0.4204, 0.1003]
    loss = [0.3449, 0.4311, 1.0955, 2.4417, 6.3235]

    y_acc_min, y_acc_max = 0.0, 1.0
    y_loss_min, y_loss_max = 0.0, 7.0

    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 4))

    # Accuracy
    ax1.plot(sparsity, accuracy, "bo-", label="Test Accuracy")
    ax1.set_xlabel("Pruning Percentage")
    ax1.set_ylabel("Test Accuracy", color="tab:blue")
    ax1.tick_params(axis="y", labelcolor="tab:blue")
    ax1.set_title("Impact of Pruning Percentage on Accuracy")
    ax1.set_ylim(y_acc_min, y_acc_max)

    # Loss
    ax2.plot(sparsity, loss, "ro-", label="Test Loss")
    ax2.set_xlabel("Pruning Percentage q")
    ax2.set_ylabel("Test Loss", color="tab:red")
    ax2.tick_params(axis="y", labelcolor="tab:red")
    ax2.set_title("Impact of Pruning Percentage on Loss")
    ax2.set_ylim(y_loss_min, y_loss_max)

    fig.suptitle("Effect of Pruning on ResNet-20 Performance")
    plt.tight_layout()
    plt.show()
```



The graph shows the effect of pruning on ResNet-20 performance. As the pruning percentage increases from 20% to 80%, test accuracy steadily decreases, with a significant drop after 60% pruning. Simultaneously, test loss remains relatively stable until 60% pruning, after which it increases dramatically, indicating that aggressive pruning beyond 60% significantly degrades model performance while offering diminishing returns in model compression.

c)

Lab2 (c) Finetune pruned model

```python
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.data != 0).float()
    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 = 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()
```

```python
# Get pruned model
#net.load_state_dict(torch.load("pretrained_model.pt"))

net.load_state_dict(torch.load(pretrained_model_path))
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=80.0)

# 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..
```

```python
# Model finetuning
#for epoch in range(20):
for epoch in range(20):
    print('\nEpoch: %d' % epoch)
    net.train()
    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 val_acc > best_acc:
        best_acc = val_acc
        print("Saving...")
        torch.save(net.state_dict(), "net_after_finetune.pt")
```

```
Epoch: 0
[Step=50]       Loss=0.9794     acc=0.6882      1788.3 examples/second
[Step=100]      Loss=0.8078     acc=0.7371      2210.7 examples/second
[Step=150]      Loss=0.7196     acc=0.7637      2525.2 examples/second
Test Loss=0.5583, Test acc=0.8169
Saving...

Epoch: 1
[Step=50]       Loss=0.4720     acc=0.8406      1456.8 examples/second
[Step=100]      Loss=0.4624     acc=0.8421      2827.1 examples/second
[Step=150]      Loss=0.4507     acc=0.8467      2193.3 examples/second
Test Loss=0.4881, Test acc=0.8401
Saving...

Epoch: 2
[Step=50]       Loss=0.4036     acc=0.8626      1567.8 examples/second
[Step=100]      Loss=0.4021     acc=0.8629      2541.7 examples/second
[Step=150]      Loss=0.3959     acc=0.8653      2794.6 examples/second
Test Loss=0.4562, Test acc=0.8500
Saving...

Epoch: 3
[Step=50]       Loss=0.3621     acc=0.8752      1734.6 examples/second
[Step=100]      Loss=0.3609     acc=0.8757      2134.0 examples/second
[Step=150]      Loss=0.3624     acc=0.8764      2792.4 examples/second
Test Loss=0.4364, Test acc=0.8561
Saving...

Epoch: 4
[Step=50]       Loss=0.3455     acc=0.8805      1566.4 examples/second
[Step=100]      Loss=0.3437     acc=0.8816      2590.4 examples/second
[Step=150]      Loss=0.3419     acc=0.8829      2263.1 examples/second
Test Loss=0.4227, Test acc=0.8599
Saving...

Epoch: 5
[Step=50]       Loss=0.3301     acc=0.8870      1460.6 examples/second
[Step=100]      Loss=0.3268     acc=0.8882      2751.5 examples/second
[Step=150]      Loss=0.3265     acc=0.8879      2288.0 examples/second
Test Loss=0.4125, Test acc=0.8630
Saving...

Epoch: 6
[Step=50]       Loss=0.3189     acc=0.8914      1577.9 examples/second
[Step=100]      Loss=0.3206     acc=0.8910      2586.2 examples/second
[Step=150]      Loss=0.3161     acc=0.8927      2850.1 examples/second
Test Loss=0.4061, Test acc=0.8656
Saving...

Epoch: 7
[Step=50]       Loss=0.2945     acc=0.8993      1766.4 examples/second
[Step=100]      Loss=0.2990     acc=0.8968      2152.3 examples/second
[Step=150]      Loss=0.2989     acc=0.8973      2801.8 examples/second
Test Loss=0.3988, Test acc=0.8672
Saving...
```

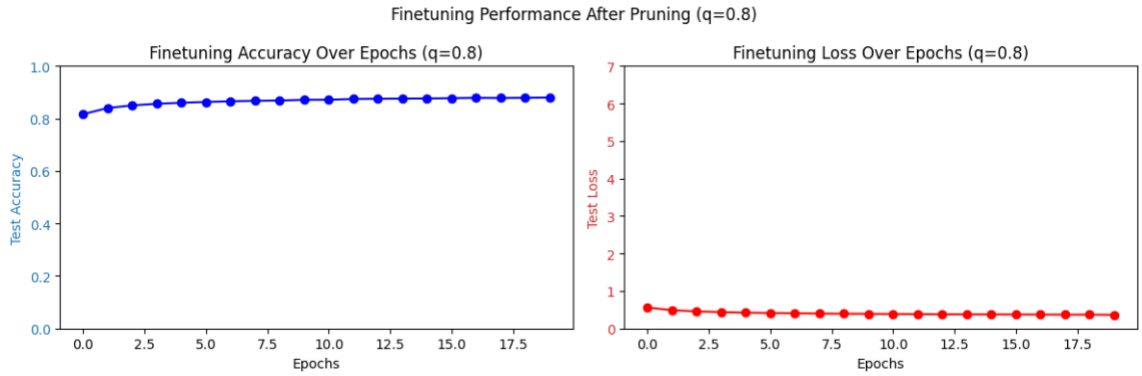
```
Epoch: 8
[Step=50]       Loss=0.3008     acc=0.8988      1554.3 examples/second
[Step=100]      Loss=0.2967     acc=0.8991      2840.2 examples/second
[Step=150]      Loss=0.2968     acc=0.8985      2131.6 examples/second
Test Loss=0.3934, Test acc=0.8685
Saving...

Epoch: 9
[Step=50]       Loss=0.2880     acc=0.9007      1430.7 examples/second
[Step=100]      Loss=0.2855     acc=0.9010      2869.7 examples/second
[Step=150]      Loss=0.2889     acc=0.9005      2556.9 examples/second
Test Loss=0.3894, Test acc=0.8711
Saving...

Epoch: 10
[Step=50]       Loss=0.2837     acc=0.9024      1662.3 examples/second
[Step=100]      Loss=0.2830     acc=0.9027      2445.4 examples/second
[Step=150]      Loss=0.2817     acc=0.9028      2686.1 examples/second
Test Loss=0.3850, Test acc=0.8714
Saving...

Epoch: 11
[Step=50]       Loss=0.2748     acc=0.9048      1800.1 examples/second
[Step=100]      Loss=0.2749     acc=0.9043      2183.0 examples/second
[Step=150]      Loss=0.2705     acc=0.9058      2832.1 examples/second
Test Loss=0.3818, Test acc=0.8743
Saving...

Epoch: 12
[Step=50]       Loss=0.2658     acc=0.9080      1565.2 examples/second
[Step=100]      Loss=0.2663     acc=0.9079      2849.4 examples/second
[Step=150]      Loss=0.2679     acc=0.9087      2165.3 examples/second
Test Loss=0.3789, Test acc=0.8752
Saving...

Epoch: 13
[Step=50]       Loss=0.2575     acc=0.9084      1418.1 examples/second
[Step=100]      Loss=0.2651     acc=0.9081      2843.9 examples/second
[Step=150]      Loss=0.2671     acc=0.9071      2589.5 examples/second
Test Loss=0.3768, Test acc=0.8757
Saving...

Epoch: 14
[Step=50]       Loss=0.2636     acc=0.9109      1712.9 examples/second
[Step=100]      Loss=0.2616     acc=0.9103      2091.8 examples/second
[Step=150]      Loss=0.2621     acc=0.9100      2869.6 examples/second
Test Loss=0.3748, Test acc=0.8762
Saving...

Epoch: 15
[Step=50]       Loss=0.2503     acc=0.9127      1832.6 examples/second
[Step=100]      Loss=0.2532     acc=0.9122      2425.9 examples/second
[Step=150]      Loss=0.2563     acc=0.9117      2289.5 examples/second
Test Loss=0.3736, Test acc=0.8772
Saving...


Epoch: 16
[Step=50]       Loss=0.2584     acc=0.9102      1444.2 examples/second
[Step=100]      Loss=0.2543     acc=0.9118      2870.2 examples/second
[Step=150]      Loss=0.2538     acc=0.9121      2156.8 examples/second
Test Loss=0.3704, Test acc=0.8788
Saving...

Epoch: 17
[Step=50]       Loss=0.2468     acc=0.9137      1477.8 examples/second
[Step=100]      Loss=0.2500     acc=0.9139      2856.0 examples/second
[Step=150]      Loss=0.2489     acc=0.9143      2791.6 examples/second
Test Loss=0.3690, Test acc=0.8780

Epoch: 18
[Step=50]       Loss=0.2410     acc=0.9168      1649.5 examples/second
[Step=100]      Loss=0.2416     acc=0.9170      2150.0 examples/second
[Step=150]      Loss=0.2411     acc=0.9163      2828.3 examples/second
Test Loss=0.3685, Test acc=0.8789
Saving...

Epoch: 19
[Step=50]       Loss=0.2393     acc=0.9174      1792.5 examples/second
[Step=100]      Loss=0.2411     acc=0.9160      2281.9 examples/second
[Step=150]      Loss=0.2432     acc=0.9154      2491.2 examples/second
Test Loss=0.3641, Test acc=0.8798
Saving...
```

```python
# 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: 0.7986111111111112
Sparsity of body_op.0.conv1.0.conv: 0.7999131944444444
Sparsity of body_op.0.conv2.0.conv: 0.7999131944444444
Sparsity of body_op.1.conv1.0.conv: 0.7999131944444444
Sparsity of body_op.1.conv2.0.conv: 0.7999131944444444
Sparsity of body_op.2.conv1.0.conv: 0.7999131944444444
Sparsity of body_op.2.conv2.0.conv: 0.7999131944444444
Sparsity of body_op.3.conv1.0.conv: 0.7999131944444444
Sparsity of body_op.3.conv2.0.conv: 0.8000217013888888
Sparsity of body_op.4.conv1.0.conv: 0.8000217013888888
Sparsity of body_op.4.conv2.0.conv: 0.8000217013888888
Sparsity of body_op.5.conv1.0.conv: 0.8000217013888888
Sparsity of body_op.5.conv2.0.conv: 0.8000217013888888
Sparsity of body_op.6.conv1.0.conv: 0.7999674479166666
Sparsity of body_op.6.conv2.0.conv: 0.7999945746527778
Sparsity of body_op.7.conv1.0.conv: 0.7999945746527778
Sparsity of body_op.7.conv2.0.conv: 0.7999945746527778
Sparsity of body_op.8.conv1.0.conv: 0.7999945746527778
Sparsity of body_op.8.conv2.0.conv: 0.7999945746527778
Sparsity of final_fc.linear: 0.8
Test Loss=0.3641, Test accuracy=0.8798
```

The finetuned model preserves the sparsity.

```python
epochs = list(range(20))
test_accuracy = [0.8169, 0.8401, 0.8500, 0.8561, 0.8599, 0.8630, 0.8656,
                 0.8672, 0.8685, 0.8711, 0.8714, 0.8743, 0.8752, 0.8757,
                 0.8762, 0.8772, 0.8788, 0.8780, 0.8789, 0.8798]
test_loss = [0.5583, 0.4881, 0.4562, 0.4364, 0.4227, 0.4125, 0.4061,
             0.3988, 0.3934, 0.3894, 0.3850, 0.3818, 0.3789, 0.3768,
             0.3748, 0.3736, 0.3704, 0.3690, 0.3685, 0.3641]

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 4))

# Accuracy over epochs
ax1.plot(epochs, test_accuracy, 'bo-', label='Test Accuracy')
ax1.set_xlabel("Epochs")
ax1.set_ylabel("Test Accuracy", color='tab:blue')
ax1.tick_params(axis='y', labelcolor='tab:blue')
ax1.set_title("Finetuning Accuracy Over Epochs (q=0.8)")
ax1.set_ylim(y_acc_min, y_acc_max)

# Loss over epochs
ax2.plot(epochs, test_loss, 'ro-', label='Test Loss')
ax2.set_xlabel("Epochs")
ax2.set_ylabel("Test Loss", color='tab:red')
ax2.tick_params(axis='y', labelcolor='tab:red')
ax2.set_title("Finetuning Loss Over Epochs (q=0.8)")
ax2.set_ylim(y_loss_min, y_loss_max)

fig.suptitle("Finetuning Performance After Pruning (q=0.8)")
plt.tight_layout()
plt.show()
```



best accuracy = 0.8789

After applying pruning with q=0.8 and fine-tuning for 20 epochs, the model achieved a maximum accuracy of 0.8798 (epoch 19) with Loss = 0.3641. The sparsity evaluation after fine-tuning confirmed that the model maintained approximately 80% sparsity across all layers, verifying that the implementation of the finetune_after_prune function correctly preserves the pruned weights at zero during training.

Even with 80% of weights pruned to zero, the network can adapt its remaining parameters through fine-tuning to achieve good classification.

d)

Lab2 (d) Iterative pruning

```python
#net.load_state_dict(torch.load("pretrained_model.pt"))
net.load_state_dict(torch.load(pretrained_model_path))
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 = 8 * (epoch + 1)
                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")
```

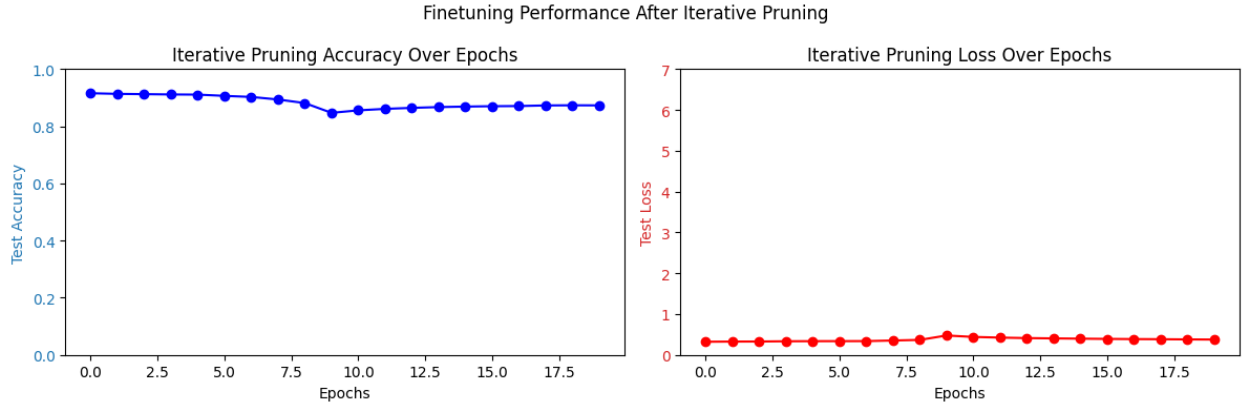
```
Epoch: 0
[Step=50]       Loss=0.0477     acc=0.9845      1406.9 examples/second
[Step=100]      Loss=0.0493     acc=0.9843      2828.6 examples/second
[Step=150]      Loss=0.0484     acc=0.9846      2878.6 examples/second
Test Loss=0.3222, Test acc=0.9156

Epoch: 1
[Step=50]       Loss=0.0500     acc=0.9838      1879.5 examples/second
[Step=100]      Loss=0.0483     acc=0.9839      2192.9 examples/second
[Step=150]      Loss=0.0498     acc=0.9836      2789.1 examples/second
Test Loss=0.3264, Test acc=0.9134

Epoch: 2
[Step=50]       Loss=0.0514     acc=0.9826      1655.0 examples/second
[Step=100]      Loss=0.0515     acc=0.9825      2970.5 examples/second
[Step=150]      Loss=0.0524     acc=0.9820      2232.2 examples/second
Test Loss=0.3278, Test acc=0.9124

Epoch: 3
[Step=50]       Loss=0.0587     acc=0.9802      1558.2 examples/second
[Step=100]      Loss=0.0581     acc=0.9808      2918.4 examples/second
[Step=150]      Loss=0.0582     acc=0.9804      2958.4 examples/second
Test Loss=0.3338, Test acc=0.9113

Epoch: 4
[Step=50]       Loss=0.0724     acc=0.9773      1817.1 examples/second
[Step=100]      Loss=0.0704     acc=0.9767      2216.3 examples/second
[Step=150]      Loss=0.0682     acc=0.9773      3042.3 examples/second
Test Loss=0.3362, Test acc=0.9108

Epoch: 5
[Step=50]       Loss=0.0929     acc=0.9676      1656.1 examples/second
[Step=100]      Loss=0.0898     acc=0.9692      2922.4 examples/second
[Step=150]      Loss=0.0864     acc=0.9706      2106.5 examples/second
Test Loss=0.3370, Test acc=0.9064

Epoch: 6
[Step=50]       Loss=0.1331     acc=0.9531      1487.3 examples/second
[Step=100]      Loss=0.1253     acc=0.9558      2929.7 examples/second
[Step=150]      Loss=0.1202     acc=0.9581      2971.9 examples/second
Test Loss=0.3379, Test acc=0.9026

Epoch: 7
[Step=50]       Loss=0.1905     acc=0.9361      1861.0 examples/second
[Step=100]      Loss=0.1770     acc=0.9386      2218.7 examples/second
[Step=150]      Loss=0.1683     acc=0.9406      2869.3 examples/second
Test Loss=0.3526, Test acc=0.8937

Epoch: 8
[Step=50]       Loss=0.2855     acc=0.8981      1512.8 examples/second
[Step=100]      Loss=0.2607     acc=0.9077      2940.0 examples/second
[Step=150]      Loss=0.2479     acc=0.9136      2166.7 examples/second
Test Loss=0.3675, Test acc=0.8812
```

```
Epoch: 9
[Step=50]       Loss=0.5730     acc=0.8031      1541.9 examples/second
[Step=100]      Loss=0.5107     acc=0.8248      2721.9 examples/second
[Step=150]      Loss=0.4802     acc=0.8364      2928.8 examples/second
Test Loss=0.4753, Test acc=0.8470

Epoch: 10
[Step=50]       Loss=0.3818     acc=0.8694      1860.4 examples/second
[Step=100]      Loss=0.3743     acc=0.8714      1933.8 examples/second
[Step=150]      Loss=0.3722     acc=0.8717      2701.3 examples/second
Test Loss=0.4401, Test acc=0.8557
Saving...

Epoch: 11
[Step=50]       Loss=0.3423     acc=0.8840      1549.0 examples/second
[Step=100]      Loss=0.3388     acc=0.8848      2190.2 examples/second
[Step=150]      Loss=0.3382     acc=0.8844      2884.2 examples/second
Test Loss=0.4237, Test acc=0.8609
Saving...

Epoch: 12
[Step=50]       Loss=0.3183     acc=0.8905      1530.5 examples/second
[Step=100]      Loss=0.3143     acc=0.8909      2805.6 examples/second
[Step=150]      Loss=0.3128     acc=0.8923      2121.9 examples/second
Test Loss=0.4118, Test acc=0.8644
Saving...

Epoch: 13
[Step=50]       Loss=0.3141     acc=0.8899      1378.0 examples/second
[Step=100]      Loss=0.3069     acc=0.8948      2777.5 examples/second
[Step=150]      Loss=0.3035     acc=0.8957      2494.3 examples/second
Test Loss=0.4043, Test acc=0.8671
Saving...

Epoch: 14
[Step=50]       Loss=0.2838     acc=0.9026      1616.7 examples/second
[Step=100]      Loss=0.2947     acc=0.8979      2247.1 examples/second
[Step=150]      Loss=0.2915     acc=0.8993      2836.5 examples/second
Test Loss=0.3975, Test acc=0.8689
Saving...

Epoch: 15
[Step=50]       Loss=0.2854     acc=0.9009      1828.3 examples/second
[Step=100]      Loss=0.2868     acc=0.9009      2032.8 examples/second
[Step=150]      Loss=0.2853     acc=0.9017      2615.6 examples/second
Test Loss=0.3918, Test acc=0.8702
Saving...

Epoch: 16
[Step=50]       Loss=0.2661     acc=0.9070      1557.6 examples/second
[Step=100]      Loss=0.2744     acc=0.9046      2941.9 examples/second
[Step=150]      Loss=0.2743     acc=0.9052      2114.2 examples/second
Test Loss=0.3873, Test acc=0.8708
Saving...

Epoch: 17
[Step=50]       Loss=0.2695     acc=0.9077      1487.2 examples/second
[Step=100]      Loss=0.2720     acc=0.9066      2903.2 examples/second
[Step=150]      Loss=0.2733     acc=0.9063      2590.8 examples/second
Test Loss=0.3843, Test acc=0.8729
Saving...

Epoch: 18
[Step=50]       Loss=0.2532     acc=0.9139      1543.6 examples/second
[Step=100]      Loss=0.2632     acc=0.9095      2513.5 examples/second
[Step=150]      Loss=0.2654     acc=0.9090      2866.2 examples/second
Test Loss=0.3793, Test acc=0.8734
Saving...

Epoch: 19
[Step=50]       Loss=0.2592     acc=0.9116      1811.6 examples/second
[Step=100]      Loss=0.2587     acc=0.9117      2091.6 examples/second
[Step=150]      Loss=0.2563     acc=0.9123      2860.9 examples/second
Test Loss=0.3761, Test acc=0.8731
```

```python
# Check sparsity of the final model, make sure it's 80%
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: 0.7986111111111112
Sparsity of body_op.0.conv1.0.conv: 0.7999131944444444
Sparsity of body_op.0.conv2.0.conv: 0.7999131944444444
Sparsity of body_op.1.conv1.0.conv: 0.7999131944444444
Sparsity of body_op.1.conv2.0.conv: 0.7999131944444444
Sparsity of body_op.2.conv1.0.conv: 0.7999131944444444
Sparsity of body_op.2.conv2.0.conv: 0.7999131944444444
Sparsity of body_op.3.conv1.0.conv: 0.7999131944444444
Sparsity of body_op.3.conv2.0.conv: 0.8000217013888888
Sparsity of body_op.4.conv1.0.conv: 0.8000217013888888
Sparsity of body_op.4.conv2.0.conv: 0.8000217013888888
Sparsity of body_op.5.conv1.0.conv: 0.8000217013888888
Sparsity of body_op.5.conv2.0.conv: 0.8000217013888888
Sparsity of body_op.6.conv1.0.conv: 0.7999674479166666
Sparsity of body_op.6.conv2.0.conv: 0.7999945746527778
Sparsity of body_op.7.conv1.0.conv: 0.7999945746527778
Sparsity of body_op.7.conv2.0.conv: 0.7999945746527778
Sparsity of body_op.8.conv1.0.conv: 0.7999945746527778
Sparsity of body_op.8.conv2.0.conv: 0.7999945746527778
Sparsity of final_fc.linear: 0.8
Test Loss=0.3793, Test accuracy=0.8734
```

```python
epochs = list(range(20))
test_accuracy = [0.9156, 0.9134, 0.9124, 0.9113, 0.9108, 0.9064, 0.9026, 0.8937, 0.8812, 0.8470, 0.8557, 0.8609, 0.8644, 0.8671, 0.8689, 0.8702, 0.8708, 0.8729, 0.8734, 0.8731]
test_loss = [0.3222, 0.3264, 0.3278, 0.3338, 0.3362, 0.3370, 0.3379, 0.3526, 0.3675, 0.4753, 0.4401, 0.4237, 0.4118, 0.4043, 0.3975, 0.3918, 0.3873, 0.3843, 0.3793, 0.3761]

y_acc_min, y_acc_max = 0.0, 1.0
y_loss_min, y_loss_max = 0.0, 7.0

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 4))

# Accuracy
ax1.plot(epochs, test_accuracy, 'bo-', label='Test Accuracy')
ax1.set_xlabel("Epochs")
ax1.set_ylabel("Test Accuracy", color='tab:blue')
ax1.tick_params(axis='y', labelcolor='tab:blue')
ax1.set_title("Iterative Pruning Accuracy Over Epochs")
ax1.set_ylim(y_acc_min, y_acc_max)

# Loss
ax2.plot(epochs, test_loss, 'ro-', label='Test Loss')
ax2.set_xlabel("Epochs")
ax2.set_ylabel("Test Loss", color='tab:red')
ax2.tick_params(axis='y', labelcolor='tab:red')
ax2.set_title("Iterative Pruning Loss Over Epochs")
ax2.set_ylim(y_loss_min, y_loss_max)

fig.suptitle("Finetuning Performance After Iterative Pruning")
plt.tight_layout()
plt.show()
```



best accuracy = 0.9156

After implementing iterative pruning, the model shows significantly better initial performance compared to part (c)'s one-shot pruning. The model starts with ~90% accuracy versus ~80% in c and maintains lower loss throughout training. As pruning gradually increases, accuracy declines to 84.7% at epoch 10, then recovers during fine-tuning to reach 87.3%. Both methods achieve similar final accuracy, but the iterative approach provides a smoother transition and allows weights to be recovered during the pruning phase, resulting in better stability.

e)

Lab2 (e) Global iterative pruning

```
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
            flattened_weights.append(np.abs(np_weight.flatten()))

    # Concate all weights into a np array
    flattened_weights = np.concatenate(flattened_weights)
    # Find global pruning threshold
    thres = np.percentile(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
            weight = layer.weight.data.cpu().numpy()

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

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

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

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

    net.train()
    if epoch<10:
        # Increase model sparsity
        q = 8 * (epoch + 1)
        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")
```

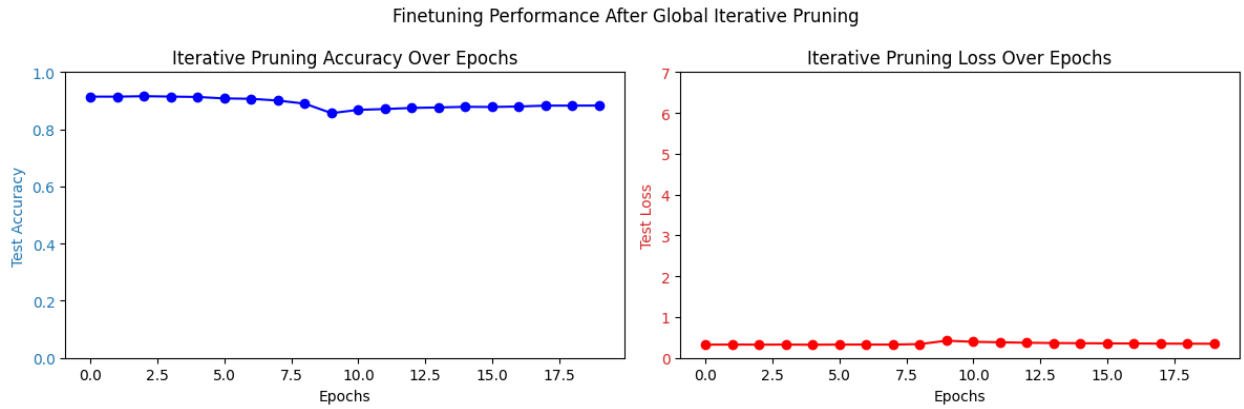
```
Epoch: 0
/usr/local/lib/python3.11/dist-packages/torch/utils/data/dataloader.py:624: UserW
  warnings.warn(
[Step=50]       Loss=0.0484     acc=0.9841      1858.6 examples/second
[Step=100]      Loss=0.0485     acc=0.9844      2404.4 examples/second
[Step=150]      Loss=0.0483     acc=0.9845      2442.8 examples/second
Test Loss=0.3243, Test acc=0.9142

Epoch: 1
[Step=50]       Loss=0.0483     acc=0.9841      1495.2 examples/second
[Step=100]      Loss=0.0483     acc=0.9846      2905.1 examples/second
[Step=150]      Loss=0.0490     acc=0.9844      2313.4 examples/second
Test Loss=0.3268, Test acc=0.9142

Epoch: 2
[Step=50]       Loss=0.0496     acc=0.9850      1601.6 examples/second
[Step=100]      Loss=0.0497     acc=0.9840      2700.9 examples/second
[Step=150]      Loss=0.0491     acc=0.9838      2872.0 examples/second
Test Loss=0.3241, Test acc=0.9157

Epoch: 3
[Step=50]       Loss=0.0535     acc=0.9820      1758.4 examples/second
[Step=100]      Loss=0.0539     acc=0.9825      2231.0 examples/second
[Step=150]      Loss=0.0546     acc=0.9822      2897.0 examples/second
Test Loss=0.3262, Test acc=0.9144

Epoch: 4
[Step=50]       Loss=0.0614     acc=0.9793      1643.5 examples/second
[Step=100]      Loss=0.0589     acc=0.9802      2664.8 examples/second
[Step=150]      Loss=0.0582     acc=0.9809      2269.2 examples/second
Test Loss=0.3222, Test acc=0.9130

Epoch: 5
[Step=50]       Loss=0.0726     acc=0.9760      1530.1 examples/second
[Step=100]      Loss=0.0700     acc=0.9770      2958.7 examples/second
[Step=150]      Loss=0.0696     acc=0.9770      2628.3 examples/second
Test Loss=0.3262, Test acc=0.9079

Epoch: 6
[Step=50]       Loss=0.0925     acc=0.9676      1819.9 examples/second
[Step=100]      Loss=0.0911     acc=0.9687      2257.6 examples/second
[Step=150]      Loss=0.0890     acc=0.9698      2935.2 examples/second
Test Loss=0.3266, Test acc=0.9066

Epoch: 7
[Step=50]       Loss=0.1353     acc=0.9529      1736.2 examples/second
[Step=100]      Loss=0.1290     acc=0.9555      2571.1 examples/second
[Step=150]      Loss=0.1258     acc=0.9570      2457.9 examples/second
Test Loss=0.3261, Test acc=0.9005

Epoch: 8
[Step=50]       Loss=0.2074     acc=0.9272      1492.2 examples/second
[Step=100]      Loss=0.1936     acc=0.9326      2790.9 examples/second
[Step=150]      Loss=0.1897     acc=0.9342      2306.2 examples/second
Test Loss=0.3385, Test acc=0.8896

Epoch: 9
[Step=50]       Loss=0.4206     acc=0.8545      1548.3 examples/second
[Step=100]      Loss=0.3880     acc=0.8662      2770.6 examples/second
[Step=150]      Loss=0.3746     acc=0.8715      2716.7 examples/second
Test Loss=0.4207, Test acc=0.8564

Epoch: 10
[Step=50]       Loss=0.3217     acc=0.8902      1806.0 examples/second
[Step=100]      Loss=0.3169     acc=0.8923      2219.9 examples/second
[Step=150]      Loss=0.3123     acc=0.8936      2885.6 examples/second
Test Loss=0.3950, Test acc=0.8680
Saving...

Epoch: 11
[Step=50]       Loss=0.3031     acc=0.8977      1620.8 examples/second
[Step=100]      Loss=0.2930     acc=0.9010      2882.8 examples/second
[Step=150]      Loss=0.2892     acc=0.9027      2216.3 examples/second
Test Loss=0.3819, Test acc=0.8708
Saving...

Epoch: 12
[Step=50]       Loss=0.2833     acc=0.9052      1388.3 examples/second
[Step=100]      Loss=0.2769     acc=0.9079      2727.1 examples/second
[Step=150]      Loss=0.2755     acc=0.9075      2203.8 examples/second
Test Loss=0.3712, Test acc=0.8747
Saving...

Epoch: 13
[Step=50]       Loss=0.2647     acc=0.9107      1271.8 examples/second
[Step=100]      Loss=0.2666     acc=0.9104      2477.1 examples/second
[Step=150]      Loss=0.2629     acc=0.9123      2848.4 examples/second
Test Loss=0.3642, Test acc=0.8764
Saving...

Epoch: 14
[Step=50]       Loss=0.2519     acc=0.9148      1793.0 examples/second
[Step=100]      Loss=0.2512     acc=0.9155      2249.4 examples/second
[Step=150]      Loss=0.2515     acc=0.9150      2671.7 examples/second
Test Loss=0.3595, Test acc=0.8786
Saving...

Epoch: 15
[Step=50]       Loss=0.2426     acc=0.9190      1868.5 examples/second
[Step=100]      Loss=0.2436     acc=0.9186      2475.3 examples/second
[Step=150]      Loss=0.2432     acc=0.9186      2525.1 examples/second
Test Loss=0.3557, Test acc=0.8780
```

```
Epoch: 16
[Step=50]       Loss=0.2472     acc=0.9159      1497.2 examples/second
[Step=100]      Loss=0.2437     acc=0.9170      2934.0 examples/second
[Step=150]      Loss=0.2422     acc=0.9164      2281.8 examples/second
Test Loss=0.3530, Test acc=0.8797
Saving...

Epoch: 17
[Step=50]       Loss=0.2366     acc=0.9182      1468.0 examples/second
[Step=100]      Loss=0.2372     acc=0.9181      2795.3 examples/second
[Step=150]      Loss=0.2340     acc=0.9202      2783.4 examples/second
Test Loss=0.3499, Test acc=0.8828
Saving...

Epoch: 18
[Step=50]       Loss=0.2280     acc=0.9252      1854.3 examples/second
[Step=100]      Loss=0.2302     acc=0.9220      2080.7 examples/second
[Step=150]      Loss=0.2294     acc=0.9223      2864.4 examples/second
Test Loss=0.3477, Test acc=0.8827

Epoch: 19
[Step=50]       Loss=0.2256     acc=0.9240      1547.3 examples/second
[Step=100]      Loss=0.2212     acc=0.9256      2887.3 examples/second
[Step=150]      Loss=0.2213     acc=0.9254      1985.9 examples/second
Test Loss=0.3465, Test acc=0.8830
Saving...
```

```python
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: 0.3101851851851852
Sparsity of body_op.0.conv1.0.conv: 0.6575520833333334
Sparsity of body_op.0.conv2.0.conv: 0.6397569444444444
Sparsity of body_op.1.conv1.0.conv: 0.6263020833333334
Sparsity of body_op.1.conv2.0.conv: 0.6488715277777778
Sparsity of body_op.2.conv1.0.conv: 0.6315104166666666
Sparsity of body_op.2.conv2.0.conv: 0.6701388888888888
Sparsity of body_op.3.conv1.0.conv: 0.6234809027777778
Sparsity of body_op.3.conv2.0.conv: 0.6879340277777778
Sparsity of body_op.4.conv1.0.conv: 0.7261284722222222
Sparsity of body_op.4.conv2.0.conv: 0.7822265625
Sparsity of body_op.5.conv1.0.conv: 0.7247178819444444
Sparsity of body_op.5.conv2.0.conv: 0.8133680555555556
Sparsity of body_op.6.conv1.0.conv: 0.7324761284722222
Sparsity of body_op.6.conv2.0.conv: 0.7647569444444444
Sparsity of body_op.7.conv1.0.conv: 0.7769368489583334
Sparsity of body_op.7.conv2.0.conv: 0.8260633680555556
Sparsity of body_op.8.conv1.0.conv: 0.8525119357638888
Sparsity of body_op.8.conv2.0.conv: 0.9767252604166666
Sparsity of final_fc.linear: 0.1578125
Total sparsity of: 0.8000007453342078
Test Loss=0.3465, Test accuracy=0.8830
```

```python
epochs = list(range(20))
test_accuracy = [0.9142, 0.9142, 0.9157, 0.9144, 0.9130, 0.9079, 0.9066, 0.9005, 0.8896, 0.8564,
                 0.8680, 0.8708, 0.8747, 0.8764, 0.8786, 0.8780, 0.8797, 0.8828, 0.8827, 0.8830]
test_loss = [0.3243, 0.3268, 0.3241, 0.3262, 0.3222, 0.3262, 0.3266, 0.3261, 0.3385, 0.4207,
             0.3950, 0.3819, 0.3712, 0.3642, 0.3595, 0.3557, 0.3530, 0.3499, 0.3477, 0.3465]


y_acc_min, y_acc_max = 0.0, 1.0
y_loss_min, y_loss_max = 0.0, 7.0

fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(12, 4))

# Accuracy
ax1.plot(epochs, test_accuracy, 'bo-', label='Test Accuracy')
ax1.set_xlabel("Epochs")
ax1.set_ylabel("Test Accuracy", color='tab:blue')
ax1.tick_params(axis='y', labelcolor='tab:blue')
ax1.set_title("Iterative Pruning Accuracy Over Epochs")
ax1.set_ylim(y_acc_min, y_acc_max)

# Loss
ax2.plot(epochs, test_loss, 'ro-', label='Test Loss')
ax2.set_xlabel("Epochs")
ax2.set_ylabel("Test Loss", color='tab:red')
ax2.tick_params(axis='y', labelcolor='tab:red')
ax2.set_title("Iterative Pruning Loss Over Epochs")
ax2.set_ylim(y_loss_min, y_loss_max)

fig.suptitle("Finetuning Performance After Global Iterative Pruning")
plt.tight_layout()
plt.show()
```



best accuracy = 0.9157 → best option in this lab (I will use in lab 3)

After applying magnitude-based global iterative pruning in part (e), the model maintains high accuracy (~92%) in the early epochs, with only a gradual drop as pruning increases. A significant dip to ~85% occurs around epochs 8-10 at maximum pruning, then recovers to 88-90% during fine-tuning. Loss values stay stable, showing efficient adaptation. The main benefit of this approach is the flexibility to create non-uniform sparsity across layers, preserving important parameters while pruning more in less critical layers. This likely explains the strong performance even with 80% overall sparsity.

4 Lab 3: Fixed-point quantization and finetuning (30 pts)

Besides pruning, fixed-point quantization is another important technique applied for deep neural network compression. In this Lab, you will convert the ResNet-20 model we used in previous lab into a quantized model, evaluate is performance and apply finetuning on the model.

Lab 3 (30 points)

(a) (15 pts) As is mentioned in lecture 17, to train a quantized model we need to use floating-point weight as trainable variable while use a straight-through estimator (STE) in forward and backward pass to convert the weight into quantized value. Intuitively, the forward pass of STE converts a float weight into fixed-point, while the backward pass passes the gradient straightly through the quantizer to the float weight.

To start with, implement the STE forward function in `FP_layers.py`, so that it serves as a linear quantizer with dynamic scaling, as introduced on page 9 of lecture 17. Please follow the comments in the code to figure out the expected functionality of each line. **Take a screen shot** of the finished STE class and paste it into the report. Submission of the `FP_layers.py` file is not required. (**Hint:** Please consider zeros in the weight as being pruned away, and build a mask to ensure that STE is only applied on non-zero weight elements for quantization.)

(b) (5 pts) In `hw4.ipynb`, load pretrained ResNet-20 model, report the accuracy of the floating-point pretrained model. Then set `Nbits` in the first line of block 4 to 6, 5, 4, 3, and 2 respectively, run it and report the test accuracy you got. (Hint: In this block the line defining the ResNet model (second line) will set the residual blocks in all three stages to `Nbits` fixed-point, while keeping the first conv and final FC layer still as floating point.)

(c) (5 pts) With `Nbits` set to 5, 4, 3, and 2 respectively, run the provided code to finetune the quantized model for 20 epochs. You do not need to change other parameter in the `finetune` function. For each precision, report the highest testing accuracy you get during finetuning. Comment on the relationship between precision and accuracy, and on the effectiveness of finetuning.

(d) (5 pts) In practice, we want to apply both pruning and quantization on the DNN model. Here we explore how pruning will affect quantization performance. Please load the checkpoint of the 80% sparsity model with the best accuracy from Lab 2, repeat the process in (c), report the accuracy before and after finetuning, and discuss your observations comparing to (c)'s results.

Lab 4 (Bonus 5 points)

Please first finish all the required coding for Lab 3, then proceed to the final code block of the notebook file.

(a) (5 pts) Symmetric quantization is a commonly used and hardware-friendly quantization approach. In symmetric quantization, the quantization levels are symmetric to zero. Implement symmetric quantization in the STE class and repeat the process in (b). Compare and analyze the performance of symmetric quantization and asymmetric quantization.

a)

```python
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 = (w != 0).float()

            # 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

            # Lab4 (a), Your code here:
            else:
                pass

            # Restore zero elements in wq
            wq = wq*weight_mask

        return wq

    @staticmethod
    def backward(ctx, g):
        return g, None, None
```

b) and c)

Nbits = 6

```python
# Define quantized model and load weight
Nbits = 6 #Change this value to finish (b) and (c)

net = ResNetCIFAR(num_layers=20, Nbits=Nbits)
net = net.to(device)

net.load_state_dict(torch.load(pretrained_model_path))
test(net)
```

```
Test Loss=0.3364, Test accuracy=0.9145
```

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

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

```
==> Preparing data..
```

```
==> Preparing data..

Epoch: 0
[Step=50]       Loss=0.0499     acc=0.9838      1301.4 examples/second
[Step=100]      Loss=0.0492     acc=0.9839      2305.3 examples/second
[Step=150]      Loss=0.0485     acc=0.9841      1956.1 examples/second
Test Loss=0.3245, Test acc=0.9154
Saving...

Epoch: 1
[Step=200]      Loss=0.0640     acc=0.9775      1253.4 examples/second
[Step=250]      Loss=0.0490     acc=0.9837      2135.3 examples/second
[Step=300]      Loss=0.0498     acc=0.9833      2702.3 examples/second
[Step=350]      Loss=0.0497     acc=0.9836      1995.1 examples/second
Test Loss=0.3243, Test acc=0.9151

Epoch: 2
[Step=400]      Loss=0.0396     acc=0.9863      1116.7 examples/second
[Step=450]      Loss=0.0461     acc=0.9853      1928.1 examples/second
[Step=500]      Loss=0.0488     acc=0.9842      2499.7 examples/second
[Step=550]      Loss=0.0485     acc=0.9846      2074.9 examples/second
Test Loss=0.3249, Test acc=0.9157
Saving...

Epoch: 3
[Step=600]      Loss=0.0481     acc=0.9857      1117.5 examples/second
[Step=650]      Loss=0.0488     acc=0.9855      2273.5 examples/second
[Step=700]      Loss=0.0470     acc=0.9861      2007.7 examples/second
[Step=750]      Loss=0.0477     acc=0.9856      1557.2 examples/second
Test Loss=0.3280, Test acc=0.9146

Epoch: 4
[Step=800]      Loss=0.0491     acc=0.9832      991.6 examples/second
[Step=850]      Loss=0.0478     acc=0.9834      2534.6 examples/second
[Step=900]      Loss=0.0481     acc=0.9840      1939.7 examples/second
[Step=950]      Loss=0.0480     acc=0.9839      2607.0 examples/second
Test Loss=0.3256, Test acc=0.9147

Epoch: 5
[Step=1000]     Loss=0.0493     acc=0.9822      1093.3 examples/second
[Step=1050]     Loss=0.0483     acc=0.9850      2655.1 examples/second
[Step=1100]     Loss=0.0477     acc=0.9852      1982.4 examples/second
[Step=1150]     Loss=0.0472     acc=0.9855      2634.8 examples/second
Test Loss=0.3264, Test acc=0.9162
Saving...

Epoch: 6
[Step=1200]     Loss=0.0443     acc=0.9875      1047.0 examples/second
[Step=1250]     Loss=0.0472     acc=0.9855      2498.7 examples/second
[Step=1300]     Loss=0.0462     acc=0.9860      1970.6 examples/second
[Step=1350]     Loss=0.0466     acc=0.9856      2752.2 examples/second
Test Loss=0.3284, Test acc=0.9152
```

Etc.....

Nbits = 5

```
# Define quantized model and load weight
Nbits = 5 #Change this value to finish (b) and (c)

net = ResNetCIFAR(num_layers=20, Nbits=Nbits)
net = net.to(device)

net.load_state_dict(torch.load(pretrained_model_path))
test(net)
```

```
Test Loss=0.3390, Test accuracy=0.9112
```

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

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

```
==> Preparing data..

Epoch: 0
[Step=50]       Loss=0.0535     acc=0.9826      1421.5 examples/second
[Step=100]      Loss=0.0517     acc=0.9832      2682.8 examples/second
[Step=150]      Loss=0.0525     acc=0.9828      2057.2 examples/second
Test Loss=0.3308, Test acc=0.9138
Saving...

Epoch: 1
[Step=200]      Loss=0.0482     acc=0.9854      1368.3 examples/second
[Step=250]      Loss=0.0517     acc=0.9821      1900.5 examples/second
[Step=300]      Loss=0.0515     acc=0.9826      2672.0 examples/second
[Step=350]      Loss=0.0511     acc=0.9832      2278.4 examples/second
Test Loss=0.3303, Test acc=0.9134

Epoch: 2
[Step=400]      Loss=0.0496     acc=0.9839      1292.1 examples/second
[Step=450]      Loss=0.0509     acc=0.9838      2022.0 examples/second
```

Nbits = 4

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

net = ResNetCIFAR(num_layers=20, Nbits=Nbits)
net = net.to(device)

net.load_state_dict(torch.load(pretrained_model_path))
test(net)
```

```
Test Loss=0.3861, Test accuracy=0.8972
```

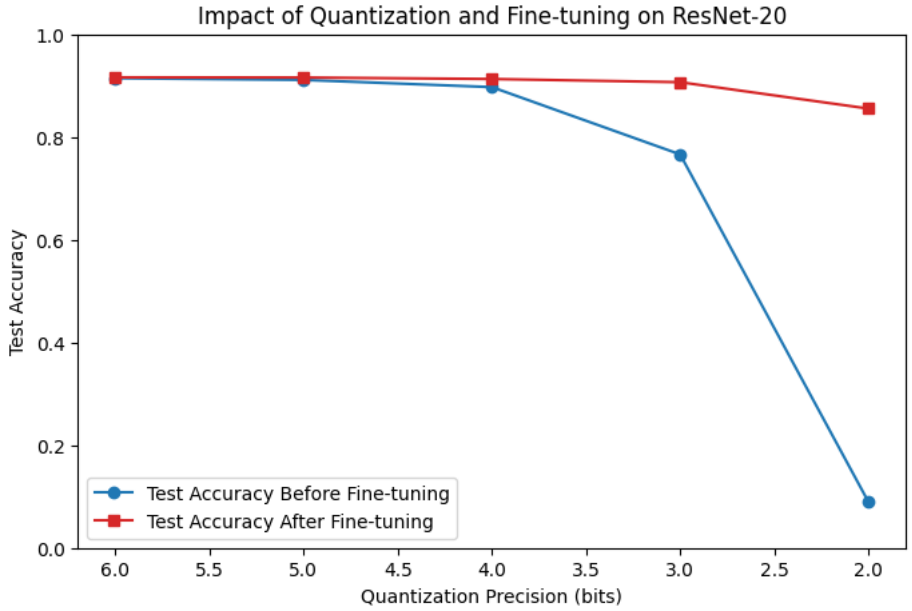
```
59] # Quantized model finetuning
finetune(net, epochs=20, batch_size=256, lr=0.002, reg=1e-4)

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

....

I ran the code several times changing Nbits = 6, 5, 4, 3, 2
The summary results is the following:

Precision	Test accuracy before Fine-tuning	Test accuracy after Fine-tuning
6 bits	0.9145	0.9162
5 bits	0.9112	0.9159
4 bits	0.8972	0.9130
3 bits	0.7662	0.9068
2 bits	0.0899	0.8554

```python
bits = [6, 5, 4, 3, 2]
accuracy_before = [0.9145, 0.9112, 0.8972, 0.7662, 0.0899]
accuracy_after = [0.9162, 0.9159, 0.9130, 0.9068, 0.8554]

y_acc_min, y_acc_max = 0.0, 1.0

plt.figure(figsize=(8, 5))
plt.plot(bits, accuracy_before, marker='o', linestyle='-', label='Test Accuracy Before Fine-tuning', color='tab:blue')
plt.plot(bits, accuracy_after, marker='s', linestyle='-', label='Test Accuracy After Fine-tuning', color='tab:red')

plt.xlabel('Quantization Precision (bits)')
plt.ylabel('Test Accuracy')
plt.title('Impact of Quantization and Fine-tuning on ResNet-20')
plt.ylim(y_acc_min, y_acc_max)
plt.gca().invert_xaxis()
plt.legend()
plt.show()
```



- We can observe a significant drop in accuracy as bit precision decreases. While 6, 5, and 4-bit models maintain relatively high accuracy (above 80%), there's a dramatic decline with 3-bit (approximately 77%) and especially with 2-bit quantization (only around 10%), showing the limitations of extreme quantization.
- Fine -tuning proves effective results across all precision levels. Even the severely degraded 2-bit model recovers to around 85% accuracy after fine-tuning. Fine-tuning becomes crucial for maintaining model performance at lower precision levels.

d)

My best model was Global Iterative Pruning in lab 2) e

For the question, we follow a process similar to that of (b) and (c) but with my best model in lab 2. I performed this process repeatedly, changing Nbits to 6, 5, 4, 3, and 2.

Nbits = 6

```
# Define quantized model and load weight
Nbits = 6

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

```
Test Loss=0.3480, Test accuracy=0.8820
```

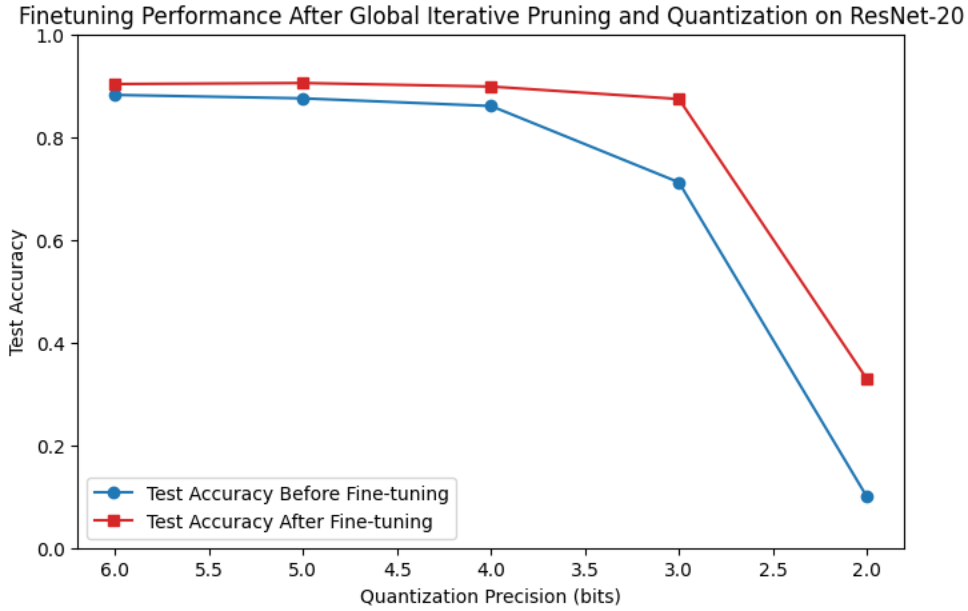
```
[74] # Quantized model finetuning
finetune(net, epochs=20, batch_size=256, lr=0.002, reg=1e-4)

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

```
==> Preparing data..

Epoch: 0
[Step=50]       Loss=0.2440     acc=0.9152      1256.6 examples/second
[Step=100]      Loss=0.2297     acc=0.9198      1651.0 examples/second
[Step=150]      Loss=0.2223     acc=0.9222      2003.8 examples/second
Test Loss=0.3744, Test acc=0.8847
Saving...

Epoch: 1
[Step=200]      Loss=0.2097     acc=0.9316      1076.0 examples/second
[Step=250]      Loss=0.1960     acc=0.9321      1496.4 examples/second
[Step=300]      Loss=0.1916     acc=0.9331      2187.2 examples/second
[Step=350]      Loss=0.1951     acc=0.9319      2418.3 examples/second
Test Loss=0.3473, Test acc=0.8893
Saving...

Epoch: 2
[Step=400]      Loss=0.1894     acc=0.9263      1127.4 examples/second
[Step=450]      Loss=0.1876     acc=0.9334      1642.5 examples/second
[Step=500]      Loss=0.1850     acc=0.9342      2492.9 examples/second
```

The summary results is the following:

Precision	Test accuracy before Fine-tuning	Test accuracy after Fine-tuning
6 bits	0.8820	0.9032
5 bits	0.8753	0.9052
4 bits	0.8604	0.8983
3 bits	0.7118	0.8740
2 bits	0.1000	0.3292

```
bits = [6, 5, 4, 3, 2]
accuracy_before = [0.8820, 0.8753, 0.8604, 0.7118, 0.1000]
accuracy_after = [0.9032, 0.9052, 0.8983, 0.8740, 0.3292]

y_acc_min, y_acc_max = 0.0, 1.0

plt.figure(figsize=(8, 5))
plt.plot(bits, accuracy_before, marker='o', linestyle='-', label='Test Accuracy Before Fine-tuning', color='tab:blue')
plt.plot(bits, accuracy_after, marker='s', linestyle='-', label='Test Accuracy After Fine-tuning', color='tab:red')

plt.xlabel('Quantization Precision (bits)')
plt.ylabel('Test Accuracy')
plt.title('Finetuning Performance After Global Iterative Pruning and Quantization on ResNet-20')
plt.ylim(y_acc_min, y_acc_max)
plt.gca().invert_xaxis()
plt.legend()
plt.show()
```



The graph shows how fine-tuning helps maintain accuracy when both pruning and quantization are applied to ResNet-20. With higher bit precision (6-5 bits), accuracy stays stable before and after fine-tuning. As precision decreases, the gap before fine-tuning and after fine-tuning grows larger, showing fine-tuning becomes more important. At very low precision (2 bits), both lines drop significantly, but fine-tuning still manages to keep accuracy around 30% instead of 10% (like past experiment). This demonstrates that fine-tuning is essential when combining multiple compression techniques.

Lab 4 (Bonus 5 points)

Please first finish all the required coding for Lab 3, then proceed to the final code block of the notebook file.

(a) (5 pts) Symmetric quantization is a commonly used and hardware-friendly quantization approach. In symmetric quantization, the quantization levels are symmetric to zero. Implement symmetric quantization in the STE class and repeat the process in (b). Compare and analyze the performance of symmetric quantization and asymmetric quantization.

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

            # 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

            # Lab4 (a), Your code here:
            else:
                alpha = torch.max(torch.abs(w))
                ws = w / alpha
                step = 2 ** (bit - 1) - 1
                R = torch.round(ws * step) / step
                wq = R * alpha

            # Restore zero elements in wq
            wq = wq*weight_mask

        return wq
```

Lab4 (a) Symmetric quantization

Implement symmetric quantization in FP_layers.py, and repeat the process in (b)

```
[ ] # check the performance of symmetric quantization with 6, 5, 4, 3, 2 bits
```

```
[87] fp_path = '/content/drive/MyDrive/2023 - Duke/2025-01/ECE661_ComputerEngineeringMachineLearning&DeepNeuralNets/Assigments/HW4/V2FP_layers.py'
     spec = importlib.util.spec_from_file_location("FP_layers", fp_path)
     FP_layers = importlib.util.module_from_spec(spec)
     sys.modules["FP_layers"] = FP_layers

     from FP_layers import *
```

Nbits = 6

```
Nbits = 6
symmetric = True

net = ResNetCIFAR(num_layers=20, Nbits=Nbits, symmetric=symmetric)
net = net.to(device)
net.load_state_dict(torch.load(pretrained_model_path))
test(net)
```

```
Test Loss=0.3231, Test accuracy=0.9151
```

Nbits = 5

```
[89] Nbits = 5
     symmetric = True

     net = ResNetCIFAR(num_layers=20, Nbits=Nbits, symmetric=symmetric)
     net = net.to(device)
     net.load_state_dict(torch.load(pretrained_model_path))
     test(net)
```

```
Test Loss=0.3231, Test accuracy=0.9151
```

Nbits = 4

```
[90] Nbits = 4
     symmetric = True

     net = ResNetCIFAR(num_layers=20, Nbits=Nbits, symmetric=symmetric)
     net = net.to(device)
     net.load_state_dict(torch.load(pretrained_model_path))
     test(net)
```

```
Test Loss=0.3231, Test accuracy=0.9151
```

Nbits = 3

```
[91] Nbits = 3
     symmetric = True

     net = ResNetCIFAR(num_layers=20, Nbits=Nbits, symmetric=symmetric)
     net = net.to(device)
     net.load_state_dict(torch.load(pretrained_model_path))
     test(net)
```

```
Test Loss=0.3231, Test accuracy=0.9151
```

Nbits = 2

```
[92] Nbits = 2
     symmetric = True

     net = ResNetCIFAR(num_layers=20, Nbits=Nbits, symmetric=symmetric)
     net = net.to(device)
     net.load_state_dict(torch.load(pretrained_model_path))
     test(net)
```

```
Test Loss=0.3231, Test accuracy=0.9151
```