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

Hai Li
ECE Department, Duke University — October 18, 2023

Objectives

Homework #4 covers the contents of Lectures 12 ~ 15. 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 **Sakai** before **11:59:00pm, Wednesday, November 8**. 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 results and observations, (2) *a single code/notebook file* used to produce the results for Lab1: Sparse optimization of linear models, and (3) *the completed notebook file* for Lab 2 and Lab 3.

Note that 20 percent of the grade will be deducted for the submissions uploaded in a zip file.

1 True/False Questions (20 pts)

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

Problem 1.1 (2 pts) Generally speaking, the weight pruning does not intervene the weight quantization, as they are orthogonal techniques in compressing the size of DNN models.

True. Weight pruning and weight quantization are two different techniques for compressing DNN models, and they address different aspects of the model.

Problem 1.2 (2 pts) In weight pruning techniques, the distribution of the remaining weights does not affect the inference latency.

False. The distribution and structure of the remaining weights can influence the efficiency of the hardware implementations and thus affect the inference latency. For instance, structured and unstructured pruning will lead to different sparsity, which affect the inference efficiency.

Problem 1.3 (2 pts) In deep compression pipeline, even if we skip the quantization step, the pruned model can still be effectively encoded by the following Huffman coding process as pruning greatly reduces the number of weight variables.

False. The input of Huffman coding needs to be discrete, so the weight variables need to be quantized first and then perform Huffman coding.

Problem 1.4 (2 pts) Directly using SGD to optimize a sparsity-inducing regularizer (i.e. L-1, DeepHoyer etc.) with the training loss will lead to exact zero values in the weight elements, there's no need to apply additional pruning step after the optimization process.

False. Using sparsity-inducing regularizer, for example L-1, still need one final pruning step with a small constant threshold (e.g. 1e-4) to reach a sparse model (Lecture 13 Slide19).

Problem 1.5 (2 pts) Using soft thresholding operator will lead to better results comparing to using L-1 regularization directly as it solves the "bias" problem of L-1.

False. Soft thresholding still has the biased problem. On the other hand, Trimmed l_1 regularizer is proposed to solve the "bias" problem of l_1.

Problem 1.6 (2 pts) Group Lasso can lead to structured sparsity on DNNs, which is more hardware- friendly. The idea of Group Lasso comes from applying L-2 regularization to the L-1 norm of all of the groups.

False. Group lasso can lead to structured sparsity, but the mechanism is to apply L-1 regularization to the L-2 norms of all the groups (Lecture 13, Slide 20-21).

Problem 1.7 (2 pts) Proximal gradient descent introduces an additional proximity term to minimize regularization loss in the proximity of weight parameters. The proximity term allows smoother convergence of the overall objective.

True. From Lecture 14 Slide 6: $\theta^{(l)} \coloneqq prox_{\lambda R}(\hat{\theta}^{(l)}) = argmin_z \left(\lambda R(z) + \frac{1}{2} \| z - \hat{\theta}^{(l)} \|_2^2 \right)$

The added proximity term will allow smoother convergence of the overall objective.

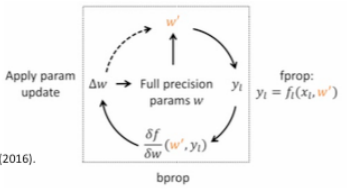
Problem 1.8 (2 pts) Models equipped with early exits allows some inputs to be computed with only part of the model, thus overcoming the issue of overfitting and overthinking.

True. From Lecture 14 Slide 27, we know that easy-to-classify samples can be processed using only a fraction of the model's layers, while more complex samples can be processed deeper into the network. The effectiveness of early exiting include overcoming overfitting and overthinking, as well as identifying easy data at early stage.

Problem 1.9 (2 pts) When implementing quantization-aware training with STE, gradients are quantized during backpropagation, ensuring that updates are consistent with the quantized weights.

False. In Lecture 15 Slide 10, gradient is computed with quantized weights, but gradients are not quantized during backpropagation.

- DNN training with STE: train with full precision weight w, loss computation with quantized w'
  - Forward: $w' = Q(w), L = f(w')$
  - Backward: $\frac{\partial L}{\partial w} = \frac{\partial L}{\partial w'}$



Gysel, Philipp. "Ristretto: Hardware-oriented approximation of convolutional neural networks." *arXiv preprint arXiv:1605.06402* (2016).

1

Problem 1.10 (2 pts) Comparing to quantizing all the layers in a DNN to the same precision, mixed-precision quantization scheme can reach higher accuracy with a similar-sized model.

True. As Lecture 15 Slide 28 suggests, performing mixed-precision quantization (assign different precisions to different layers) can provide better size/latency-accuracy tradeoff than fixed quantization.

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 $X_i \in R^{1\times5}$, $W \in R^{5\times1}$ and $||W||_0 \le 2$.

For Problem (a) - (f), consider the case where we have 3 data points: (X_1 = [1, −2, −1, −1, 1], y_1 = 7); (X_2 = [2, −1, 2, 0, −2], 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 μ = 0.02 throughout the process. Please run gradient descent for 200 steps for all the following problems.

You will need to use NumPy to finish this set of questions, please put all your code for this set of questions into one python/notebook file and submit it on Sakai. Please include all your results, figures and observations into your PDF report.

(a) (4pts) 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.)

ANSWER:

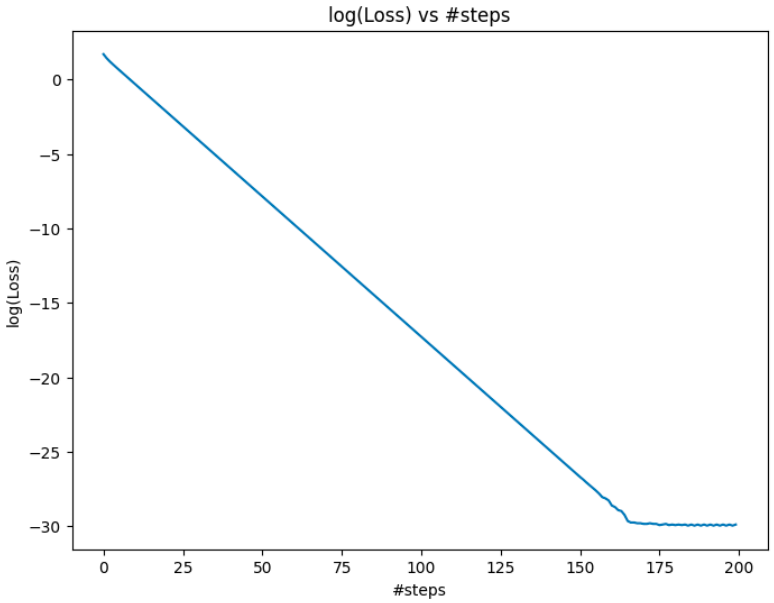
$$\frac{\partial L}{\partial W} = \frac{\partial}{\partial W}\left(\sum_i (X_i W - y_i)^2\right) = \sum_i 2(X_i W - y_i) X_i^T$$

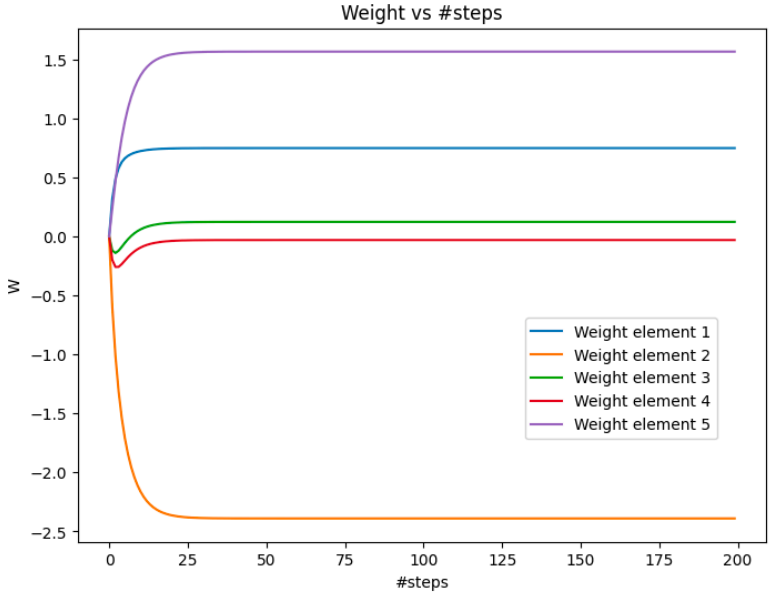
$$W^{k+1} = W^k - \mu \frac{\partial L}{\partial W} = W^k - \mu \sum_i 2(X_i W_k - y_i) X_i^T$$

(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?

ANSWER:

```
W = W_0
l_weight = []
l_loss = []
for _ in range(STPES):
    l_weight.append(W)
    loss = np.sum((np.dot(X, W) - y)**2)
    l_loss.append(loss)
    W = W - 2 * LR * np.dot(X.T, (np.dot(X, W) - y))
```



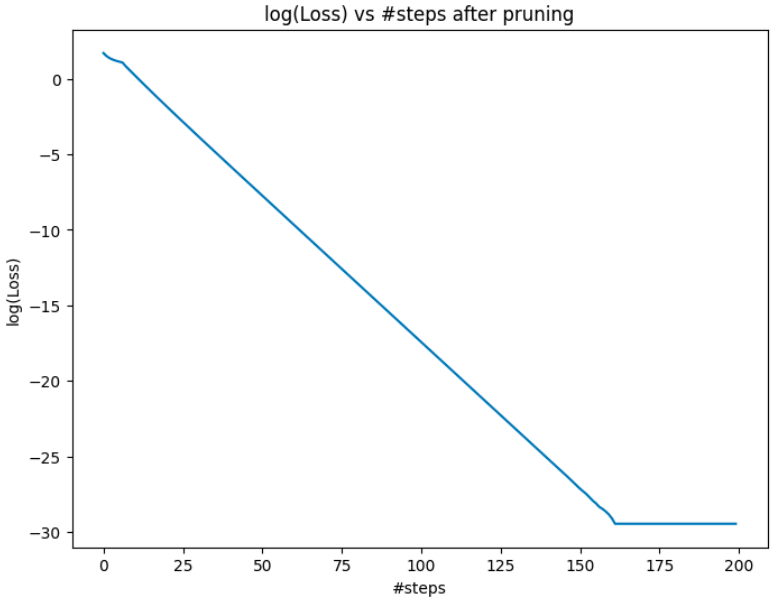


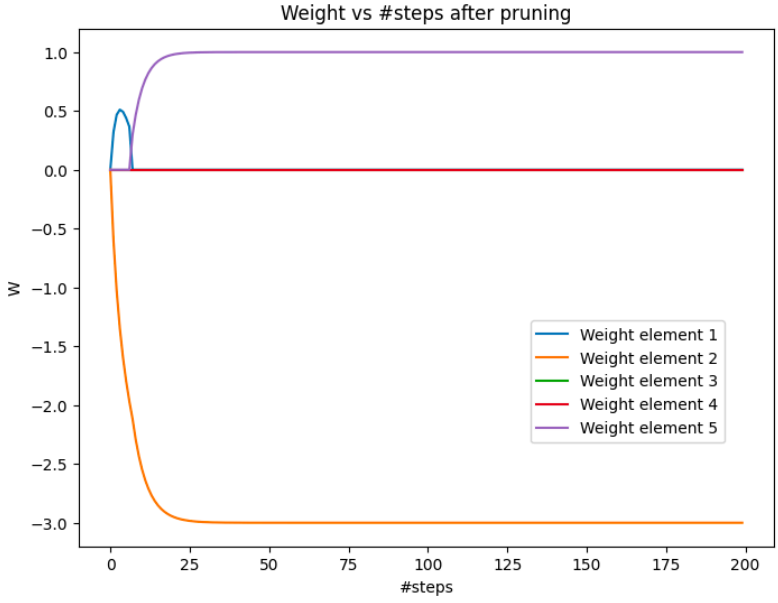
In this question, I minimized loss by directly applying the equation derived in (a) to update the weights. From the plot, weight W is **converging to optimal** as it reached plateaus and loss kept decreasing until it plateaus throughout the training process. However, it **didn't converge to a sparse solution** as a minimum of 3 elements are not converging to 0.

(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 $||W^j||_0 \leq 2$. 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?

ANSWER:

```
W = W_0
l_weight = []
l_loss = []
for _ in range(STPES):
    l_weight.append(W)
    loss = np.sum((np.dot(X, W) - y)**2)
    l_loss.append(loss)
    W = W - 2 * LR * np.dot(X.T, (np.dot(X, W) - y))
    # if non zero element in W is more than 2
    if np.count_nonzero(W) > 2:
        sorted_indices = np.argsort(np.abs(W.ravel()), axis=0)
        # set the W where sorted_indices is 2,3,4 to 0
        W[sorted_indices[:3]] = 0
```





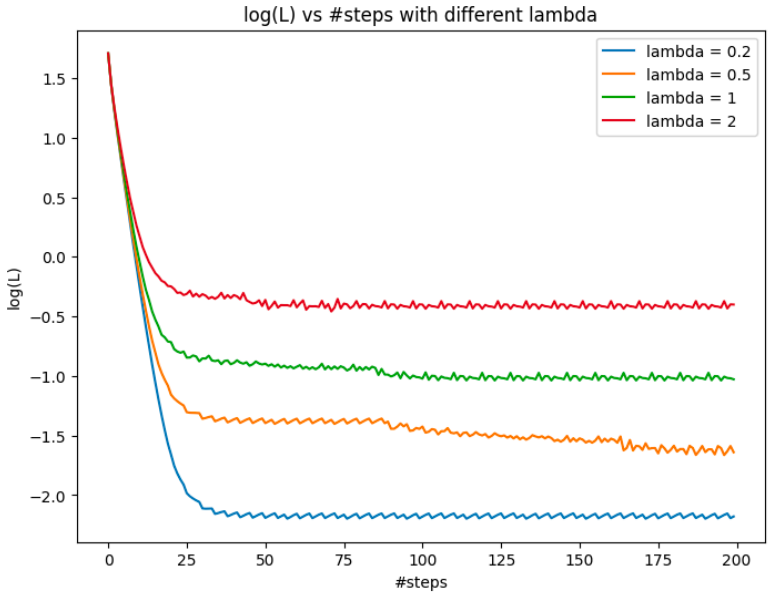
To apply projected gradient descent, I set the 3 smallest weight elements to 0 if non-zero element in W is more than 2 after gradient descent. After applying projected gradient descent, the weight **W is converging to**

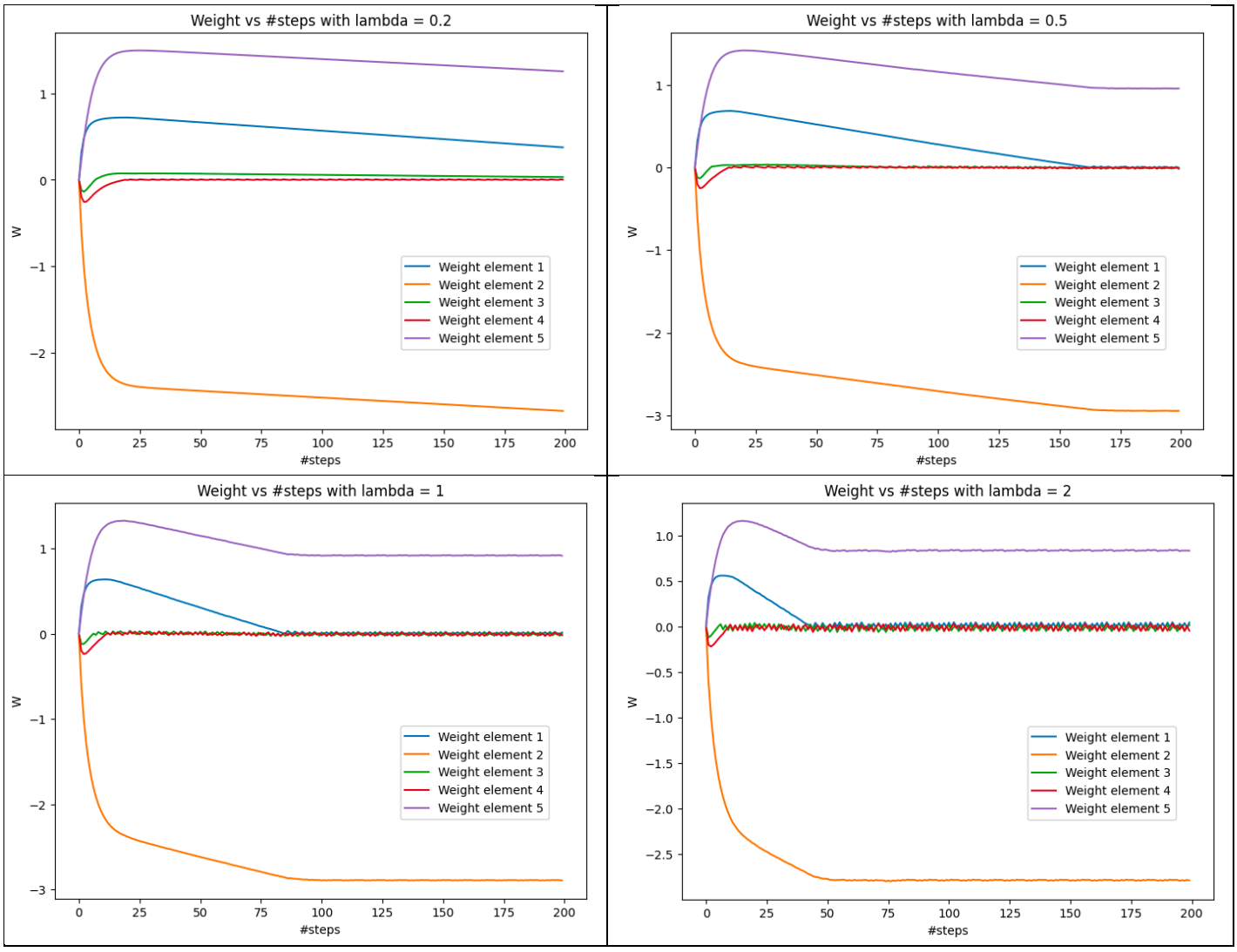
an optimal and sparse solution. This is because the loss and weight elements plateaus during the training, and 3 elements of the weight reached 0 at the end of the training process.

(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.

ANSWER:

```
lambdas = [0.2, 0.5, 1, 2]
loss_all_lambda = []
weight_all_lambda = []
for l in lambdas:
    W = W_0
    l_loss = []
    l_weight = []
    for _ in range(STPES):
        l_weight.append(W)
        loss = np.sum((np.dot(X, W) - y)**2)
        l_loss.append(loss)
        gr = 2 * np.dot(X.T, (np.dot(X, W) - y)) + l * np.sign(W)
        W = W - LR * gr
    loss_all_lambda.append(l_loss)
    weight_all_lambda.append(l_weight)
```





After introducing L1 regularization, the weight update of gradient descent changed to:

$$\frac{\partial L}{\partial W} = \sum_i 2(X_i W - y_i) X_i^T + \lambda \cdot sign(W)$$

$$W^{k+1} = W^k - \mu \frac{\partial L}{\partial W}$$

For loss, a higher λ value led to higher converged value of log(L), indicating a compromise on the loss for the sake of sparsity. In other words, when we enforce stronger sparsity constraints by increasing λ, the model may not fit the training data very well. For weight elements, more weight elements are driven towards 0 as λ increases, indicating the increasing sparsity. For both loss and weight, the larger the λ, the faster they converge. Therefore, the choice of λ needs to strike a balance between model performance and sparsity.

(e) (6pts) 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 $\mu\lambda$)

ANSWER:

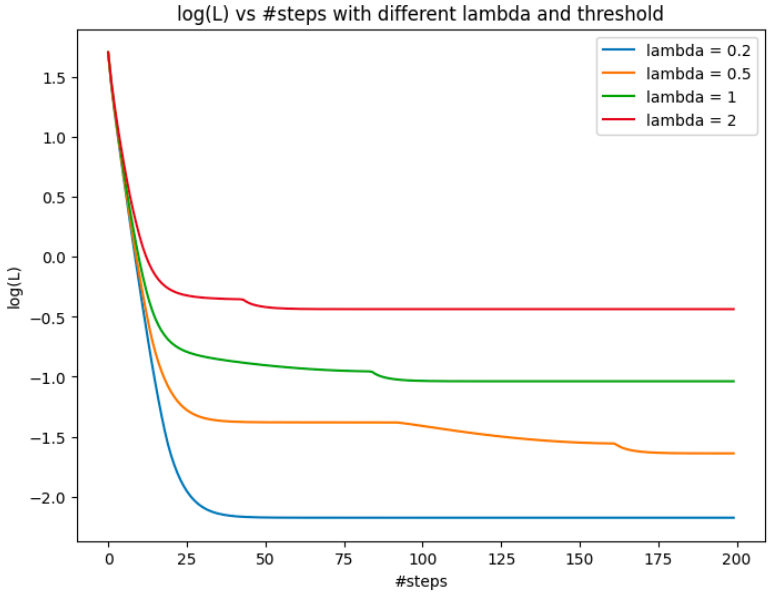
```python
thresholds = [0.004, 0.01, 0.02, 0.04]
lambdas = [0.2, 0.5, 1, 2]
```

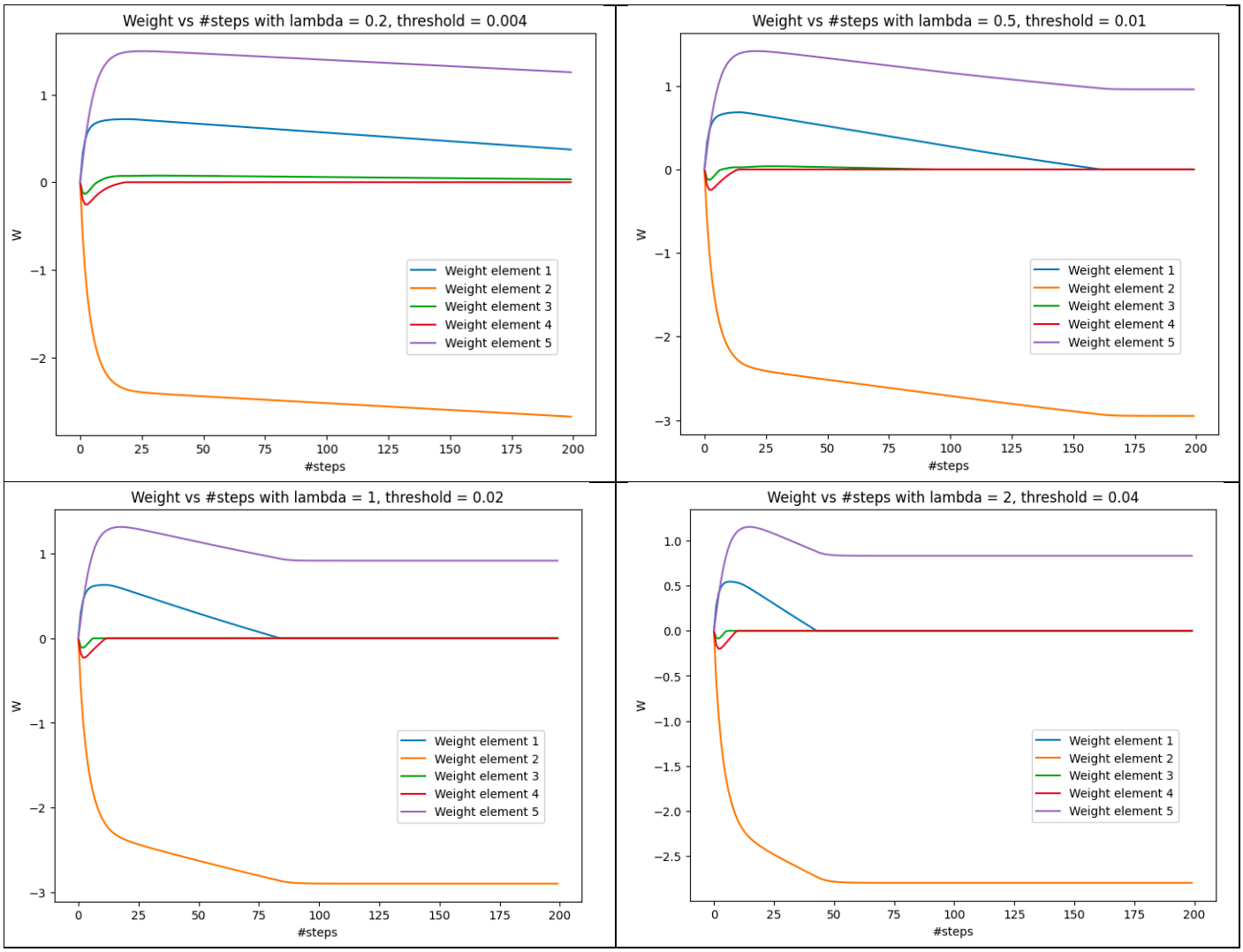
Python

```python
def soft_thresholding(theta, lamda):
    if theta > lamda:
        return theta - lamda
    elif theta < -lamda:
        return theta + lamda
    else:
        return 0
```

Python

```python
loss_all_lambda = []
weight_all_lambda = []
for i in range(len(lambdas)):
    W = W_0
    l_loss = []
    l_weight = []
    for _ in range(STPES):
        l_weight.append(W)
        loss = np.sum((np.dot(X, W) - y)**2)
        l_loss.append(loss)
        # gr = 2 * np.dot(X.T, (np.dot(X, W) - y)) + lambdas[i]
        gr = 2 * np.dot(X.T, (np.dot(X, W) - y))
        W = W - LR * gr
        for j in range(len(W)):
            W[j] = soft_thresholding(W[j], thresholds[i])
    loss_all_lambda.append(l_loss)
    weight_all_lambda.append(l_weight)
```





To apply soft thresholding, I passed every weight element after gradient descent to the following equation:

$$prox_{\lambda l_1} = \begin{cases} \theta_i - \lambda, & \theta_i > \lambda \\ 0, & |\theta_i| \leq \lambda \\ \theta_i + \lambda, & \theta_i < -\lambda \end{cases}$$

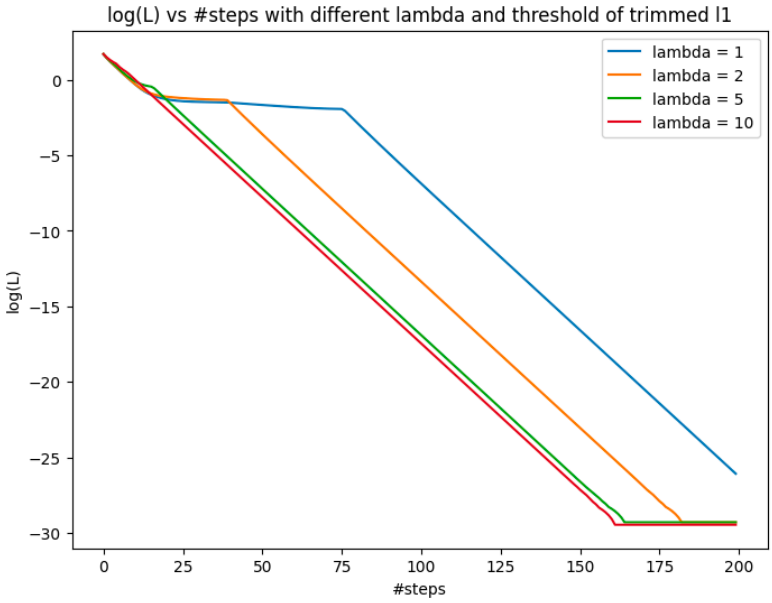
With soft thresholding, the general trend of loss and weight elements with increasing λ is similar to (d), but the training process is more stable compared with model in (d). The plot from (d) shows fluctuations in both loss and weight elements towards the latter part of the training, particularly when λ is large. However, the plot in (e) demonstrates a smooth convergence for all λ values.

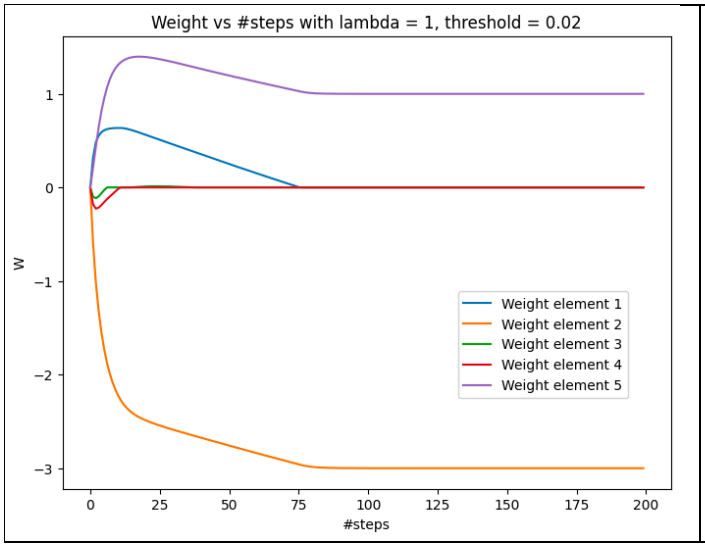
(f) (6 pts) Trimmed l_1 *(Tl₁)* regularizer is proposed to solve the "bias" problem of l_1. For simplicity you may implement the *T* l_1 regularizer as applying a l_1 regularization with strength λ on the 3 elements of W **with the smallest absolute value**, with no penalty on other elements. Minimize $L + \lambda T\ l_1(W)$ **using proximal gradient update** with λ = {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.

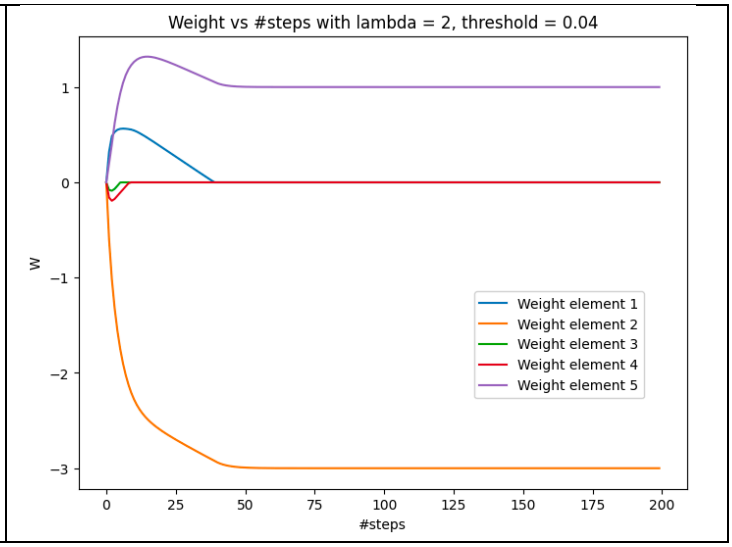
ANSWER:

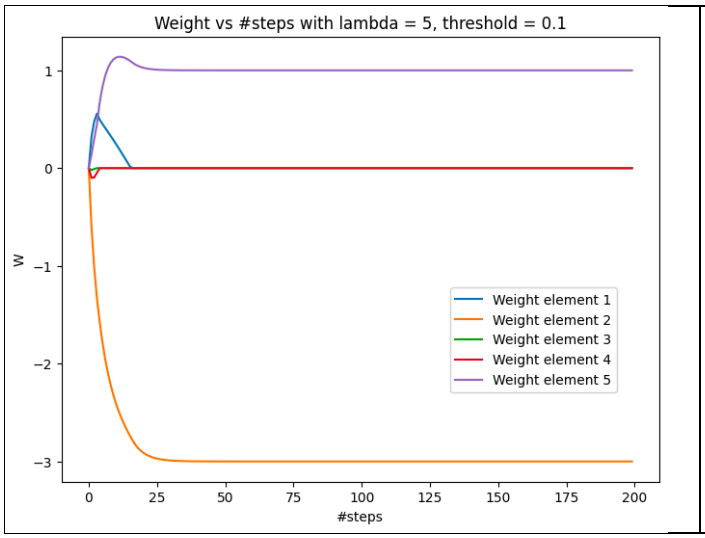
```python
loss_all_lambda = []
weight_all_lambda = []
for l in range(len(lambdas)):
    W = W_0
    l_loss = []
    l_weight = []
    for _ in range(STPES):
        l_weight.append(W)
        loss = np.sum((np.dot(X, W) - y)**2)
        l_loss.append(loss)
        gr = 2 * np.dot(X.T, (np.dot(X, W) - y))
        W = W - LR * gr
        # trim the 3 elements with smallest absolute value to soft thresholding
        smallest_indices = np.argsort(np.abs(W.ravel()), axis=0)[:3]
        for i in smallest_indices:
            W[i] = soft_thresholding(W[i], thresholds[l])
    loss_all_lambda.append(l_loss)
    weight_all_lambda.append(l_weight)
```

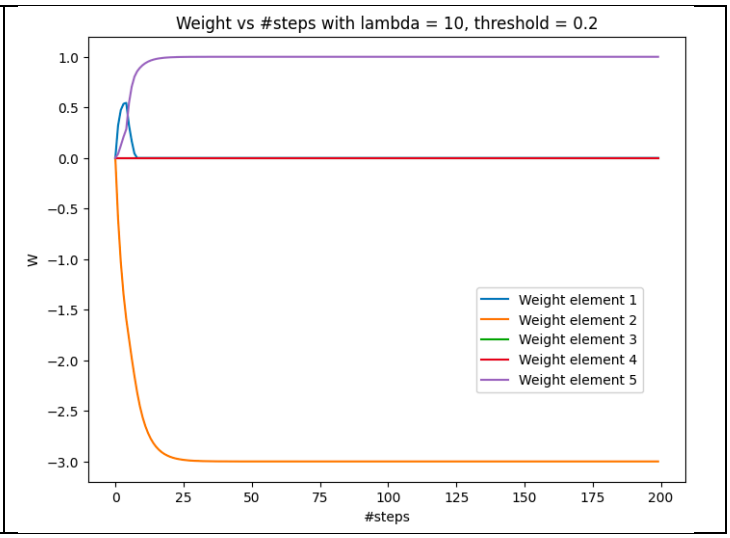
Python











To apply trimmed l_1, I trim the 3 smallest weight elements using the equation in (e).
Comparing trimmed l_1 and l_1, trimmed l_1 converges faster and converges to a lower loss value, indicating a better model performance. It is apparent that for λ=1 and 2, loss continued to decrease after it plateau around log(L)=-2. Faster convergence is also observed in weight elements: weight elements in trimmed l_1 reached to 0 much faster than those in l_1. In addition, the convergence in trimmed l_1 is more stable than l_1 as there is no fluctuation observed in the plots.
Comparing trimmed l_1 and iterative pruning, trimmed l_1 converges faster and reaches sparsity faster than iterative pruning. Most weight elements reach 0 after the first 20 steps in trimmed l_1 while in iterative pruning, it takes more steps to reach optimal sparsity.

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.

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

ANSWER:

After running through the first 3 code blocks, the test accuracy is **91.51%.**

(b) (6 pts) Complete the implementation of *pruning by percentage* function in the notebook. Here we determine 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 = 0.3, 0.5, 0.7. Report the test accuracy q. (**Hint:** You need to reload the full model checkpoint before applying the prune function with a different q).

ANSWER:

```
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 = np.abs(weight) > threshold      # return True or False

    # 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.mul_(mask)

    pass
```

q	Test accuracy
0.3	0.9028
0.5	0.8210
0.7	0.4204

The test accuracy is 90.28%, 82.10%, and 42.04% for 30th, 50th, and 70th percentile. Test accuracy kept decreasing while increasing percentage of pruning.

(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=0.7 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.

ANSWER:

The best accuracy achieved during fine-tuning is **89.41%**. After doing the sparsity evaluation, the finetuned model preserves the 70% sparsity.

```
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().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.mul_(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
# 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)
```

✓ 2.5s

```
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=0.3341, Test accuracy=0.8941
```

(d) (5pts) Implement iterative pruning. Instead of applying single step pruning before fine tuning, try iteratively increase the sparsity of the model before each epoch of finetuning. Linearly increase the pruning percentage for 10 epochs until reaching 70% in the final epoch (prune (7 × e)% before epoch e) then continue finetune for 10 epochs. Pruned weight can be recovered during the iterative pruning process before the final pruning step. Compare performance with (c)

ANSWER:

Model	Loss	Test accuracy
After fine tune (c)	0.3341	0.8941
After iterative pruning (d)	0.3368	0.8921

After iterative pruning, the loss increases 0.0027, and the test accuracy decreases 0.2%.

```
net.load_state_dict(torch.load("pretrained_model.pt"))
best_acc = 0.
l_val_acc = []
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 = 7 * (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))
    l_val_acc.append(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")
```

```python
# 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)
```

✓ 2.5s

```
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=0.3368, Test accuracy=0.8921
```

(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 70% sparsity, and report final accuracy and the percentage of zeros in each layer.

ANSWER:

The final test accuracy is **89.84%,** which is higher than that in question (d). The percentage of zeros in each layer is shown in the screenshot, and the total sparsity is 70%.

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

    # 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

            # 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.mul_(mask)
```

```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)
```

✓ 2.5s

```
Sparsity of head_conv.0.conv: 0.24305555555555555
Sparsity of body_op.0.conv1.0.conv: 0.5494791666666666
Sparsity of body_op.0.conv2.0.conv: 0.5282118055555556
Sparsity of body_op.1.conv1.0.conv: 0.5190972222222222
Sparsity of body_op.1.conv2.0.conv: 0.5520833333333334
Sparsity of body_op.2.conv1.0.conv: 0.5182291666666666
Sparsity of body_op.2.conv2.0.conv: 0.5638020833333334
Sparsity of body_op.3.conv1.0.conv: 0.5260416666666666
Sparsity of body_op.3.conv2.0.conv: 0.5826822916666666
Sparsity of body_op.4.conv1.0.conv: 0.6153428819444444
Sparsity of body_op.4.conv2.0.conv: 0.6766493055555556
Sparsity of body_op.5.conv1.0.conv: 0.611328125
Sparsity of body_op.5.conv2.0.conv: 0.7038845486111112
Sparsity of body_op.6.conv1.0.conv: 0.6153428819444444
Sparsity of body_op.6.conv2.0.conv: 0.6513129340277778
Sparsity of body_op.7.conv1.0.conv: 0.6624891493055556
Sparsity of body_op.7.conv2.0.conv: 0.718994140625
Sparsity of body_op.8.conv1.0.conv: 0.7477756076388888
Sparsity of body_op.8.conv2.0.conv: 0.93701171875
Sparsity of final_fc.linear: 0.1203125
Total sparsity of: 0.6999992546657922
Files already downloaded and verified
Test Loss=0.3161, Test accuracy=0.8984
```

4 Lab 3: Fixed-point quantization and finetuning (25 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.

(a) (10 pts) As is mentioned in lecture 15, 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 15. 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.)

ANSWER:

```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()   # weight=0 -> mask=0, weight!=0 -> mask=1

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

            # Lab3 (e), Your code here:
            else:
                # Compute the absolute maximum value for symmetric scaling
                abs_max = w.abs().max()
                # Normalize the weights such that they fall between -1 and 1
                ws = w / abs_max

                step = 2 ** (bit-1) - 1  # Adjusting the range for symmetric quantization
                # Quantize ws with a linear quantizer to "bit" bits
                R = torch.round(ws * step) / step
                # De-normalize the quantized weights
                wq = R * abs_max

            # Restore zero elements in wq
            wq = wq*weight_mask

        return wq
```

(b) (2 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.)

ANSWER:

```
for i in [2, 3, 4, 5, 6]:
    print(f"Current Nbits: {i}")
    net = ResNetCIFAR(num_layers=20, Nbits=i)
    net = net.to(device)
    net.load_state_dict(torch.load("pretrained_model.pt"))
    test(net)

    # 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)
    print('========================')
```

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

The test accuracy is 8.99%, 76.62%, 89.72%, 91.12% and 91.45% for Nbits is 2, 3, 4, 5 and 6, respectively.

(c) (5pts) With Nbits set to 4, 3, and 2 respectively, run code block 4 and 5 to fine tune 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.

ANSWER:

Nbits	Test accuracy
2	0.8550
3	0.9063
4	0.9148

Code for fine-tuning is also shown in the screenshot in question (b). Increasing Nbits from 2, 3, to 4, test accuracy after finetuning also increases from 85.5%, 90.63% to 91.48%. This indicates that greater precision leads to higher accuracy. Comparing model performance with/without finetuning at the same Nbits, it is apparent that test accuracy increased a lot after finetuning. For instance, at Nbits = 2, test accuracy is 85.5% with finetuning whereas it's only 8.99% without it.

(d) (4 pts) In practice, we want to aply both pruning and quantization on the DNN model. Here we explore how pruning will affect quantization performance. Please load the checkpoint of the 70% 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.

ANSWER:

```
for i in [2,3,4]:
    print(f"Current Nbits: {i}")
    net = ResNetCIFAR(num_layers=20, Nbits=i)
    net = net.to(device)
    net.load_state_dict(torch.load("net_after_global_iterative_prune.pt"))
    test(net)

    # 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)
    print('========================')
```

Nbits	Test accuracy before fine-tuning	Test accuracy after fine-tuning
2	0.1000	0.3590
3	0.5773	0.8839
4	0.8783	0.9033

With both pruning and quantization, the test accuracy before and after finetuning is shown in the table. It is apparent that finetuning effectively improved model performance. Compared with results in (c), there's a decline in test accuracy when pruning is combined with quantization at all Nbits. As Nbits decreases, the decrease in test accuracy between pruning + quantization and just quantization becomes larger.

(e) (4 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.

ANSWER:

Nbits	Test accuracy (symmetric)	Test accuracy (asymmetric)
2	0.1000	0.0899
3	0.5185	0.7662
4	0.8875	0.8972
5	0.9083	0.9112
6	0.9124	0.9145

Code for symmetric quantization is shown in the screenshot in question (a). In symmetric quantization, the scale factor is $q_x = \frac{2^{n-1}-1}{max(abs(x_f))}$, and the quantized tensor is $x_q = round(q_x x_f)$ (reference: https://intellabs.github.io/distiller/algo_quantization.html). The test accuracy of symmetric quantization is shown in the table. After comparing the performance of symmetric quantization and asymmetric quantization, we can see that asymmetric quantization generally performs better than symmetric

quantization except Nbits=2. As Nbits gets larger, the difference in model performance between symmetric and asymmetric quantization gets smaller.