

ECE 661: Homework 4

Pruning and Fixed-point Quantization

John Coogan

True/False Questions (15 pts)

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 incentivizes zero weight values over L-2 regularization but does not guarantee that weights will be zero.

Problem 1.2 (3 pts) While weight pruning and weight quantization both compress DNN models, they interfere with each other's processes.

False, pruning and quantization do not interfere with each other. Pruning is the process of ignoring or dropping weights at certain points during training while quantization is the more efficient representation of these weights. These processes can work together.

Problem 1.3 (3 pts) In weight pruning techniques, the distribution of the remaining weights affects the inference latency.

True, certain pruning methods can lead to sparse distributions of weights which can be challenging for hardware that may not support or be optimized for sparse matrix operations. That said, if you achieve structured sparsity, standard CPU/GPU hardware may be able to capitalize on this optimization for better inference latency.

Problem 1.4 (3 pts) Group Lasso can lead to structured sparsity on DNNs, which is more hardwarefriendly. The idea of Group Lasso comes from applying L-2 regularization to the L-1 norm of all of the groups.

False, this statement is almost entirely correct but the idea of Group Lasso comes from applying L-1 regularization to the L-2 norm of all the groups. This results in structured sparsity by inducing all-zero groups.

Problem 1.5 (3 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.

True, soft thresholding partially solves this problem. The bias problem is that the absolute value of all large weights is reduced by Lambda. Soft thresholding only applies this proximal smoothing to small weights and leaves large weights untouched, this allows for the preservation of variance.

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: $(X_1 = [-1, 2, 1, 1, -1], y_1 = 5)$; $(X_2 = [-2, 1, -2, 0, 2], y_2 = 1)$; $(X_3 = [1, 0, -2, -2, -1], y_3 = 1)$. 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 $\mu = 0.02$ 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')`

(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 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.)

The standard update equation holds here:

$$W^{(k+1)} = W^{(k)} - \mu \nabla L(W) : L(W) = \sum_i (X_i W - y_i)^2$$

$$\nabla L(W) = \frac{d}{dW} [\sum_i (X_i W - y_i)^2] = 2 \sum_i X_i (X_i W - y_i)$$

Therefore, for full batch from 1 to 3:

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

(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?

Code for Gradient Descent and Training:

```

 1 def mod_gradient_descent(
 2     X: np.array,
 3     W: np.array,
 4     y: np.array,
 5     lr: float,
 6     regularize=False,
 7     reg_strength=0,
 8     projected_gradient=False,
 9     hard_threshold=0,
10     proximal_gradient=False,
11     trimmed_L1=False,
12 ):
13     """Vectorized gradient descent
14
15     Args:
16         X (np.array): Design matrix
17         W (np.array): Weights
18         y (np.array): Target values
19         lr (float): Learning rate
20         regularize (bool, optional): Regularize the loss. Defaults to False.
21         reg_strength (int, optional): Regularization strength. Defaults to 0.
22         projected_gradient (bool, optional): Projected gradient descent. Defaults to False.
23         projected_gradient_strength (int, optional): Projection strength. Defaults to 0.
24         proximal_gradient (bool, optional): Proximal gradient descent. Defaults to False.
25         trimmed_L1 (bool, optional): Trimmed L1 regularization. Defaults to False.
26
27     Returns:
28         np.array: Updated weights
29         float: Loss
30     """
31     grad = 2 * X.T.dot(X.dot(W) - y)
32
33     if regularize:
34         grad += reg_strength * np.sign(W)
35         W -= lr * grad
36
37     elif trimmed_L1:
38         # find the lowest 3 values of the gradient
39         lowest_3_idx = np.argsort(np.abs(grad))[:3]
40         for idx in lowest_3_idx:
41             grad[idx] += reg_strength * np.sign(W[idx])
42
43         W -= lr * grad
44     else:
45         W -= lr * grad
46
47
48     if projected_gradient:
49         W[np.argsort(np.abs(W), axis=0)[-hard_threshold]] = 0
50
51     if proximal_gradient:
52         threshold = reg_strength * lr
53
54         for i in range(len(W)):
55             if W[i] > threshold:
56                 W[i] -= threshold
57             elif abs(W[i]) <= threshold:
58                 W[i] = 0
59             else:
60                 W[i] += threshold
61
62     loss = (X.dot(W) - y) ** 2
63     return W, np.sum(loss)
64
65     ✓ 0.0s

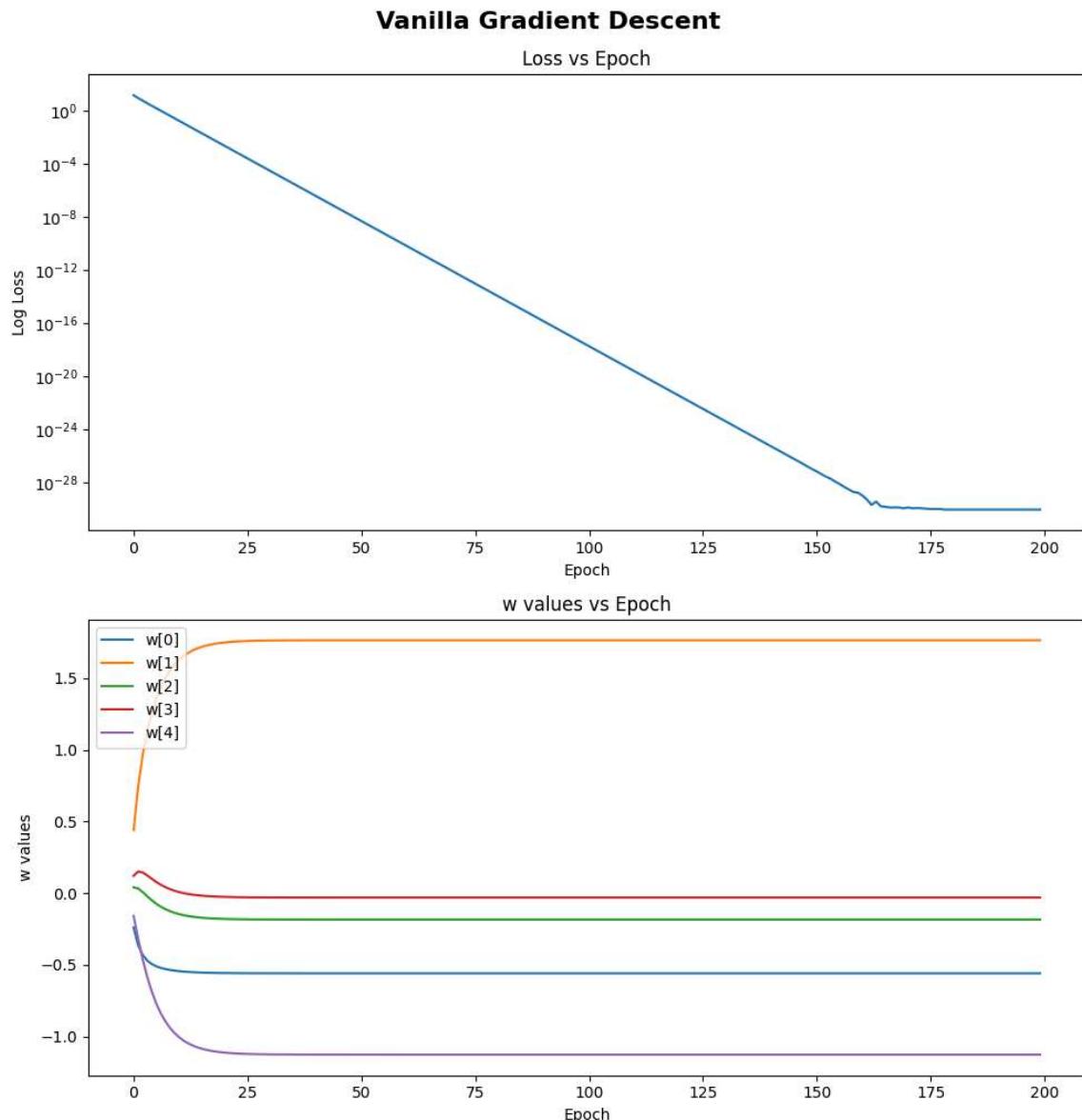
```

Python

```

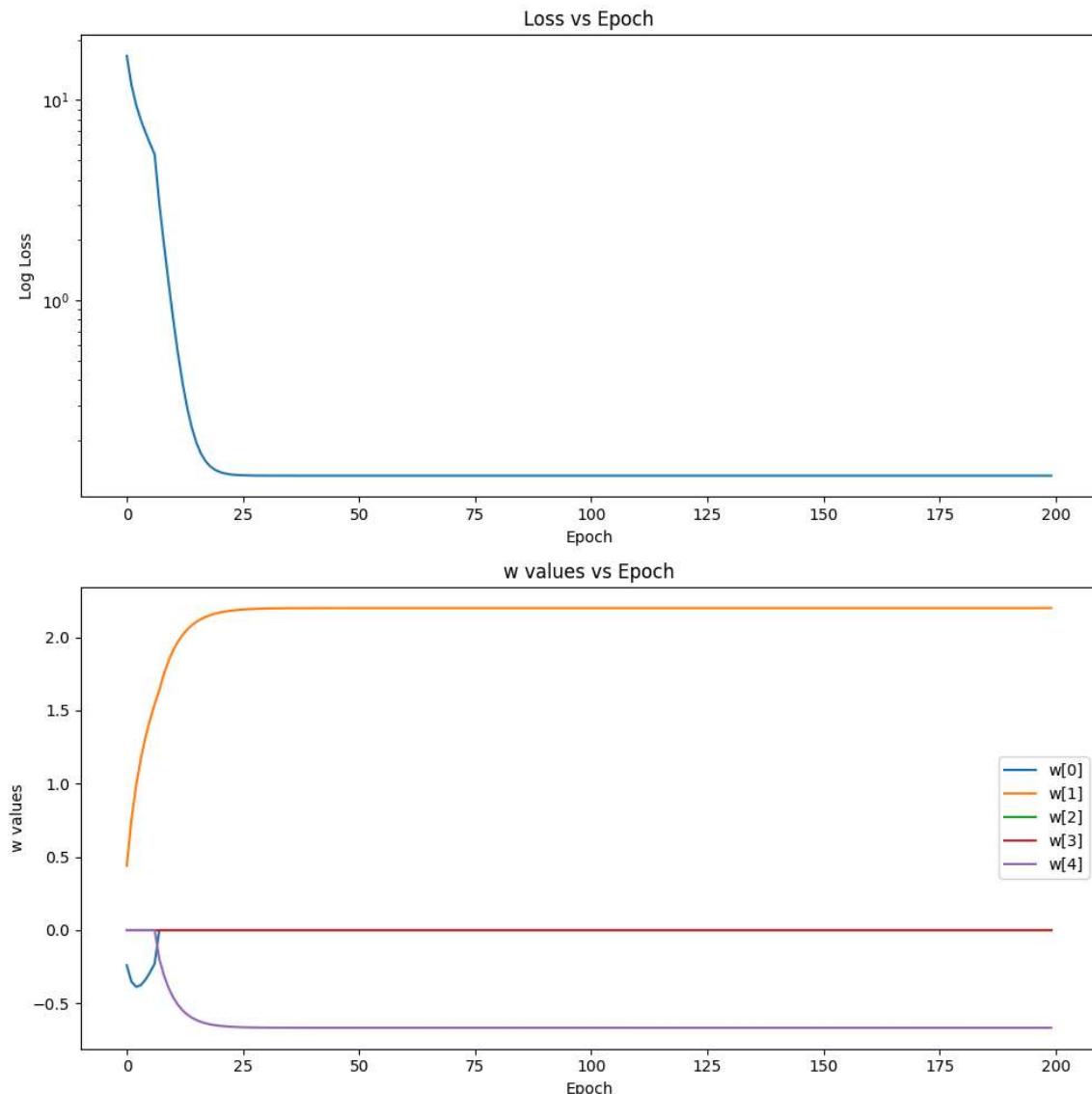
1  def train(
2      regularize=False,
3      reg_strength=0,
4      projected_gradient=False,
5      hard_thresholded=False,
6      proximal_gradient=False,
7      plot=True,
8      trained_it=False,
9  ):
10     """
11         This function will execute 200 epochs of training
12
13     Args:
14         regularize (bool, optional): Regularize the loss. Defaults to False.
15         reg_strength (int, optional): Regularization strength. Defaults to 0.
16         projected_gradient (bool, optional): Projected gradient descent. Defaults to False.
17         hard_threshold (bool, optional): Threshold for projected gradient descent. Defaults to 0.
18         proximal_gradient (bool, optional): Proximal gradient descent. Defaults to False.
19         plot (bool, optional): Plot the results. Defaults to True.
20
21     Returns:
22         list: Epochs
23         list: losses
24         list: weights
25         figure: Figure object
26
27
28     X = np.array(
29         [
30             [-1.0, 2.0, 1.0, 1.0, -1.0],
31             [-2.0, 1.0, -1.0, 0.0, 2.0],
32             [1.0, 0.0, -1.0, -2.0, -1.0],
33         ]
34     )
35
36     y = np.array([5.0, 1.0, 1.0])
37     w = np.array([0.0, 0.0, 0.0, 0.0, 0.0])
38     losses = []
39     epoch = []
40     lr = 0.02
41     w_values = []
42     for i in range(200):
43         w, loss = grad_descent(
44             X,
45             w,
46             y,
47             lr,
48             regularize,
49             reg_strength,
50             projected_gradient,
51             hard_threshold,
52             proximal_gradient,
53             trained_it,
54         )
55         w = w
56
57         losses.append(loss)
58         epoch.append(i)
59         w_values.append(w)
60
61         # print(f"The loss at epoch {i} is {loss}")
62         w_values = np.array(w_values)
63         fig = None
64         if plot:
65             # Create subplots
66             fig, axs = plt.subplots(2, 1, figsize=(10, 10))
67
68             # Plot loss vs epoch
69             axs[0].plot(epoch, losses)
70             axs[0].set_ylabel("Log")
71             axs[0].set_xlabel("Epoch")
72             axs[0].set_ylabel("Log loss")
73             axs[0].set_title("Loss vs Epoch")
74             axs[0].set_xlim(0, 200)
75
76             # Plot w values vs epoch
77             for j in range(w_values.shape[1]):
78                 axs[1].plot(epoch, w_values[:, j], label=f"w({j})")
79                 axs[1].set_xlabel("Epoch")
80                 axs[1].set_ylabel("w values")
81                 axs[1].set_title("w values vs Epoch")
82                 axs[1].legend()
83
84             plt.tight_layout()
85
86     return epoch, losses, w_values, fig
87

```



The W values are converging at around 25 epochs, this solution appears to be an optimal solution because the loss is low but it is not a sparse solution because none of the W values are zero.

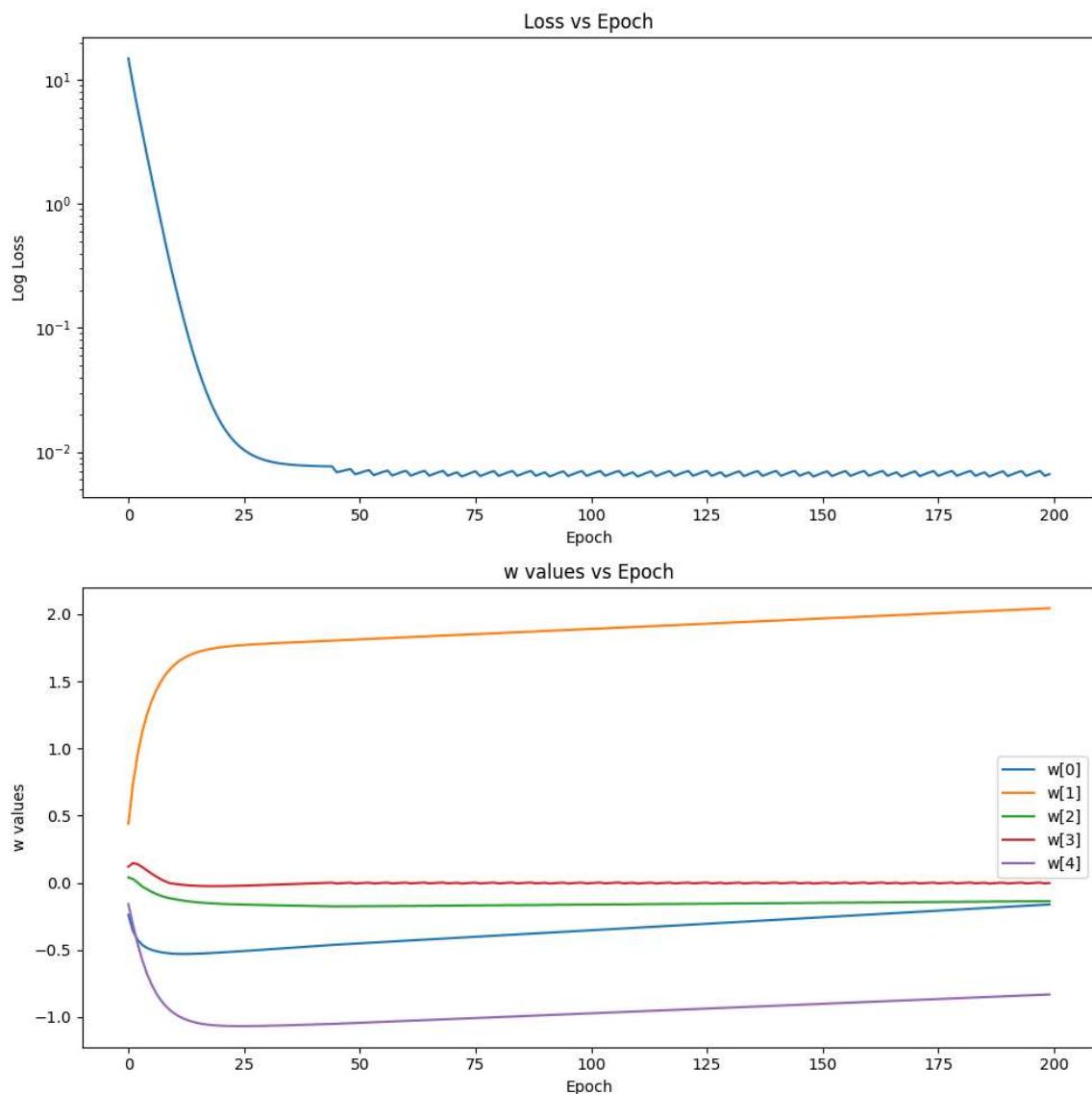
(c) (6 pts) Since we have the knowledge that the ground-truth weight should have $\|W\|_0 \leq 2$, we can apply projected gradient descent to enforce this sparse constraint. Redo the optimization process in (b), this time prune the elements in W after every gradient descent step to ensure $\|W\|_0 \leq 2$. 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?

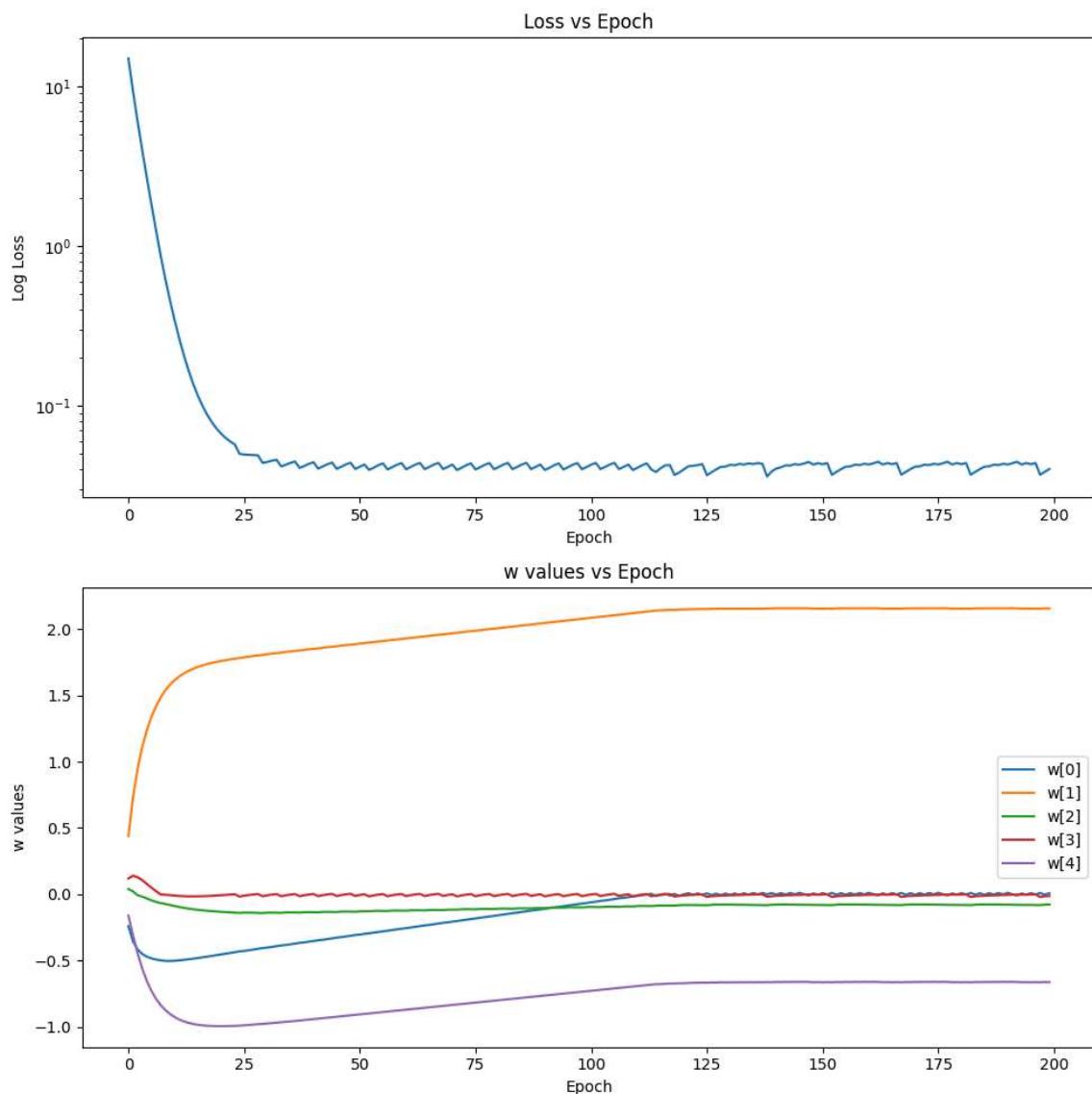
Projected Gradient Descent (Threshold=2)

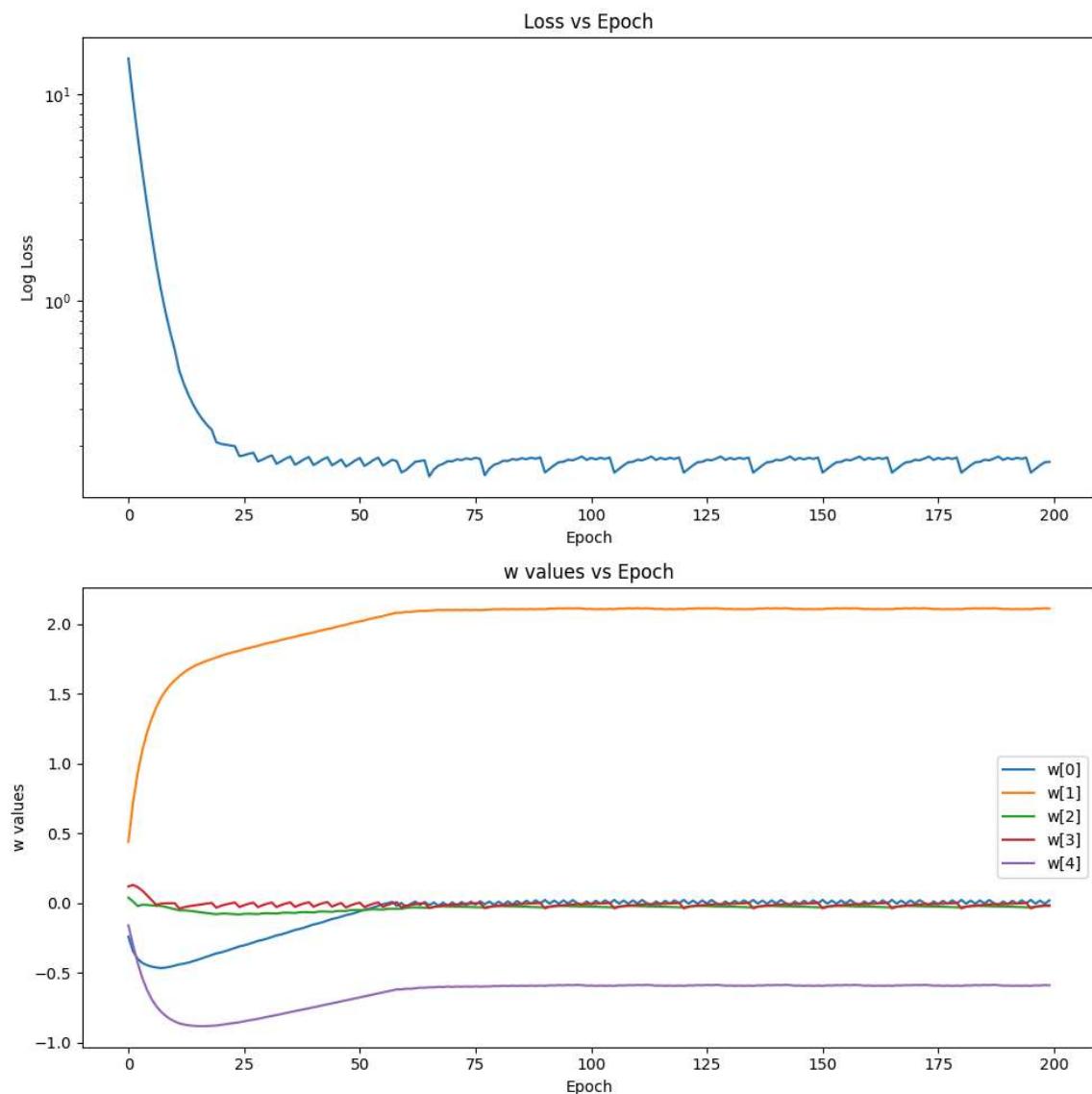
Here we can see that W is converging on a sparse solution (only two of the W values are non-zero). This sparsification is at the expense of significant loss increase compared to the base vanilla gradient descent.

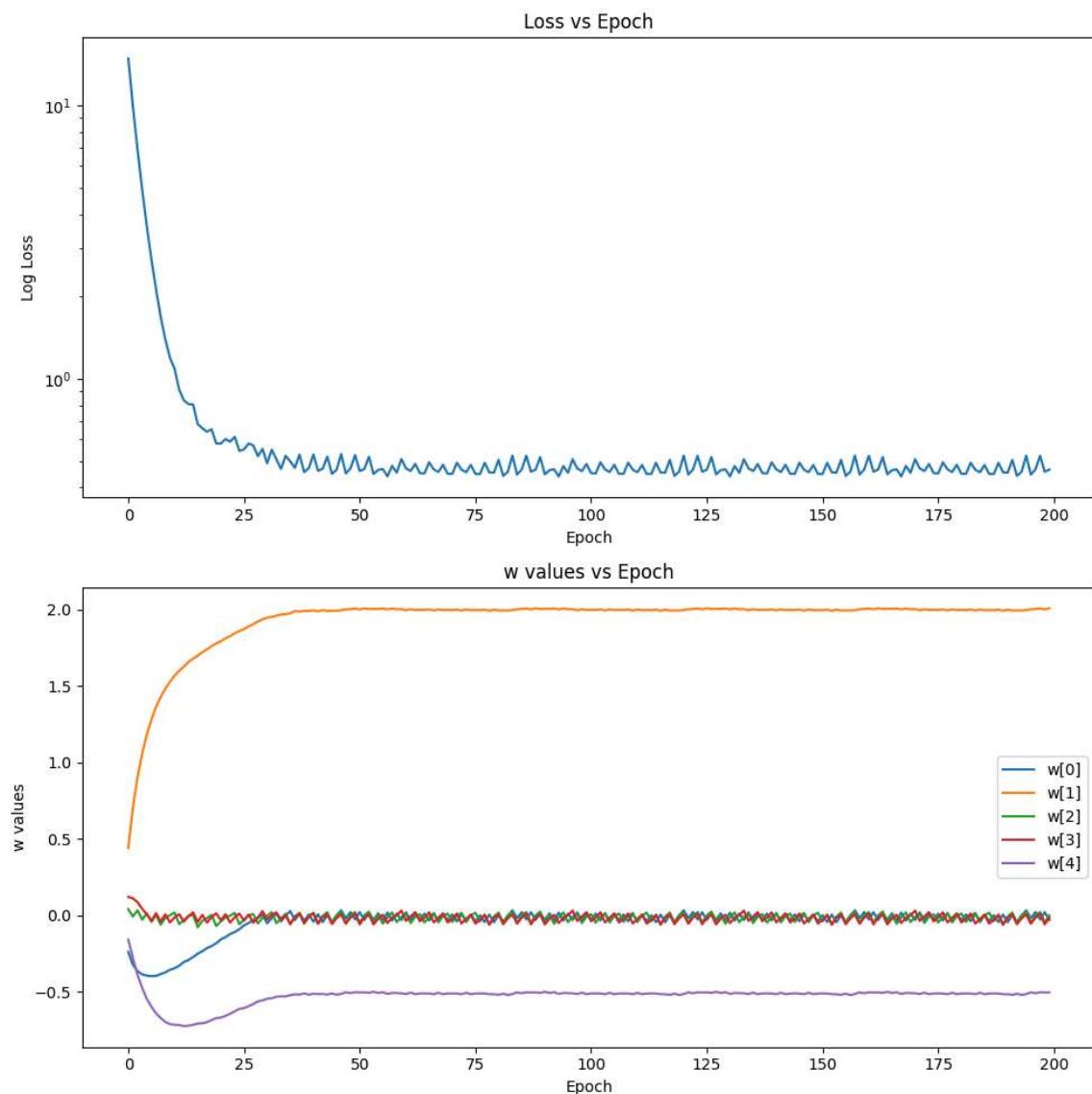
(d) (5 pts) In this problem we apply ℓ_1 regularization to induce the sparse solution. The minimization objective therefore changes to $L + \lambda||W||_1$. Please use full-batch gradient descent to minimize this objective, with $\lambda = \{0.2, 0.5, 1.0, 2.0\}$ respectively. For each case, plot the 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 .

Below are the individual plots for each lambda value for L1 regularization:

L1 Reg Gradient Descent (Lambda=0.2)

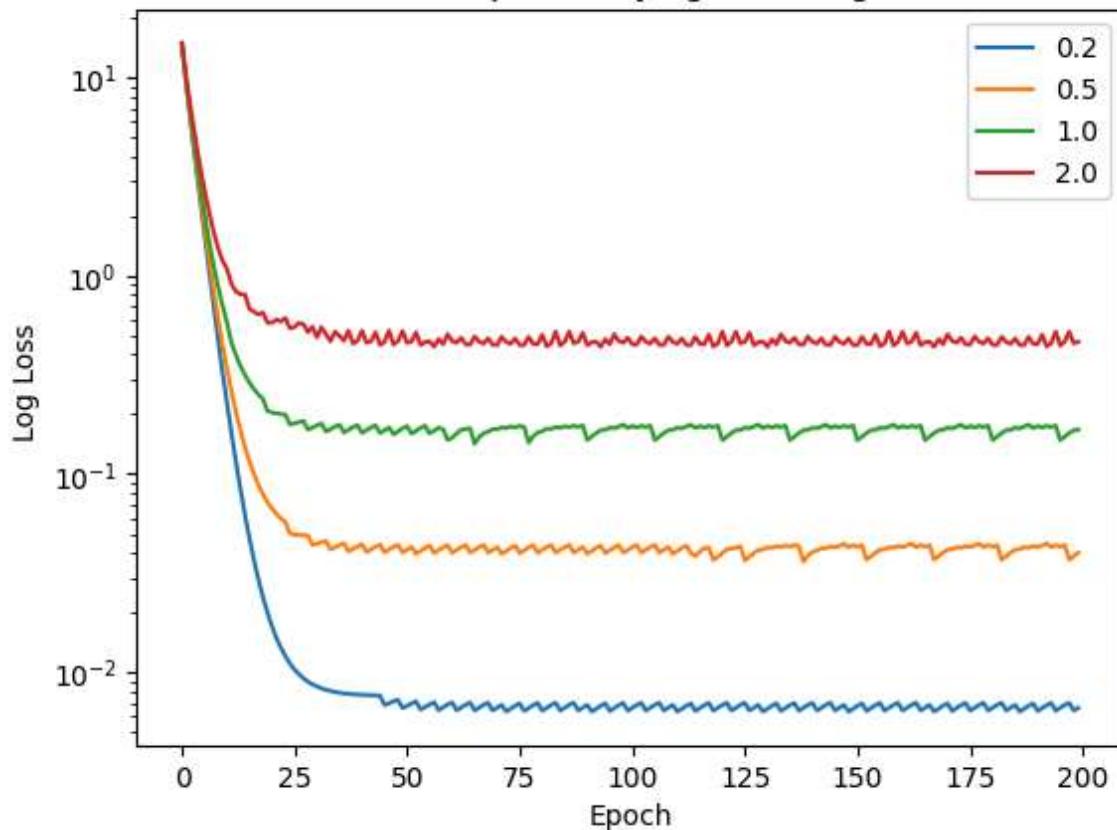
L1 Reg Gradient Descent (Lambda=0.5)

L1 Reg Gradient Descent (Lambda=1.0)

L1 Reg Gradient Descent (Lambda=2.0)

We also have all the L1 regularization plots together to see the differences:

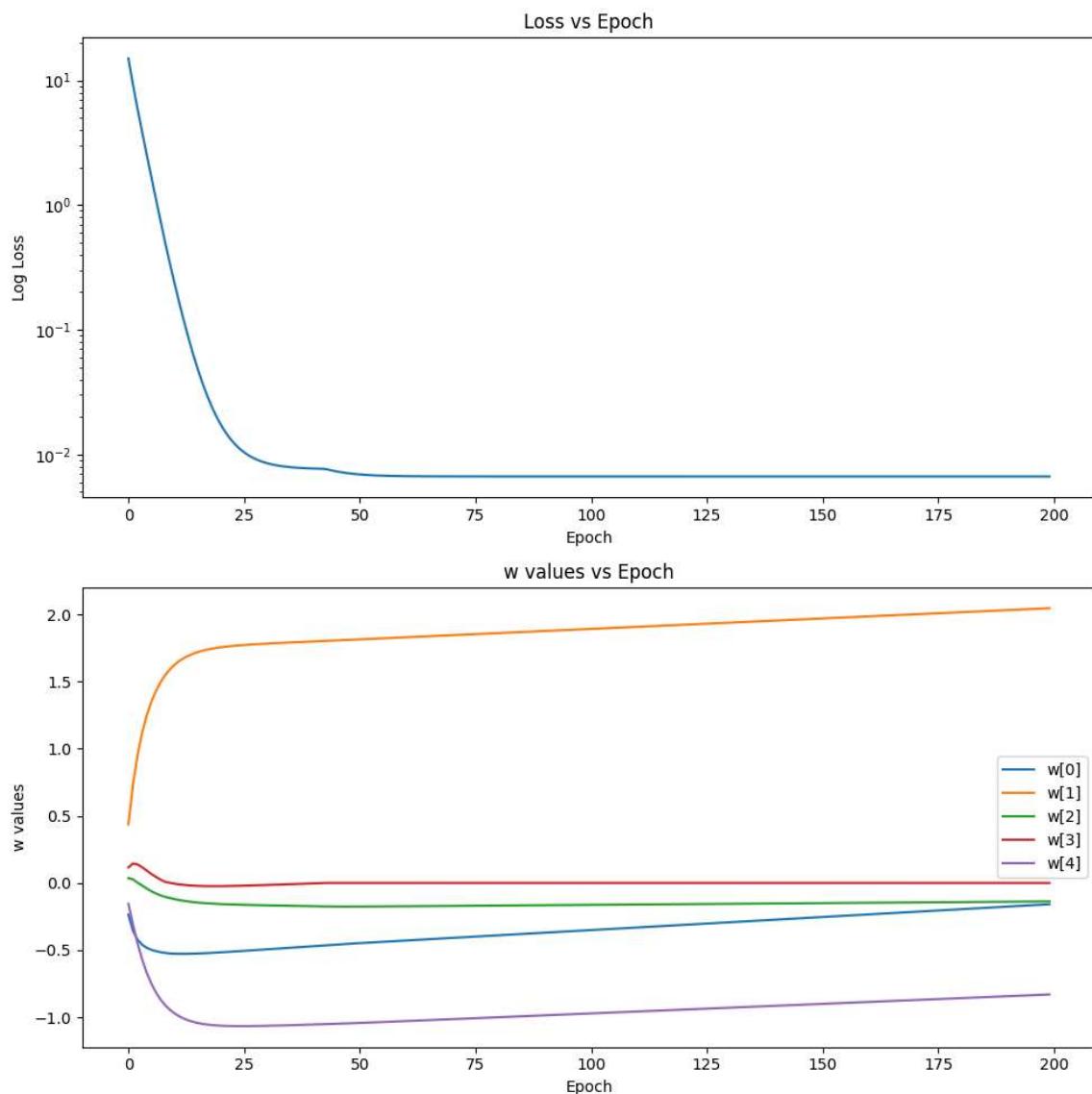
Loss vs Epoch Varying L1 strengths

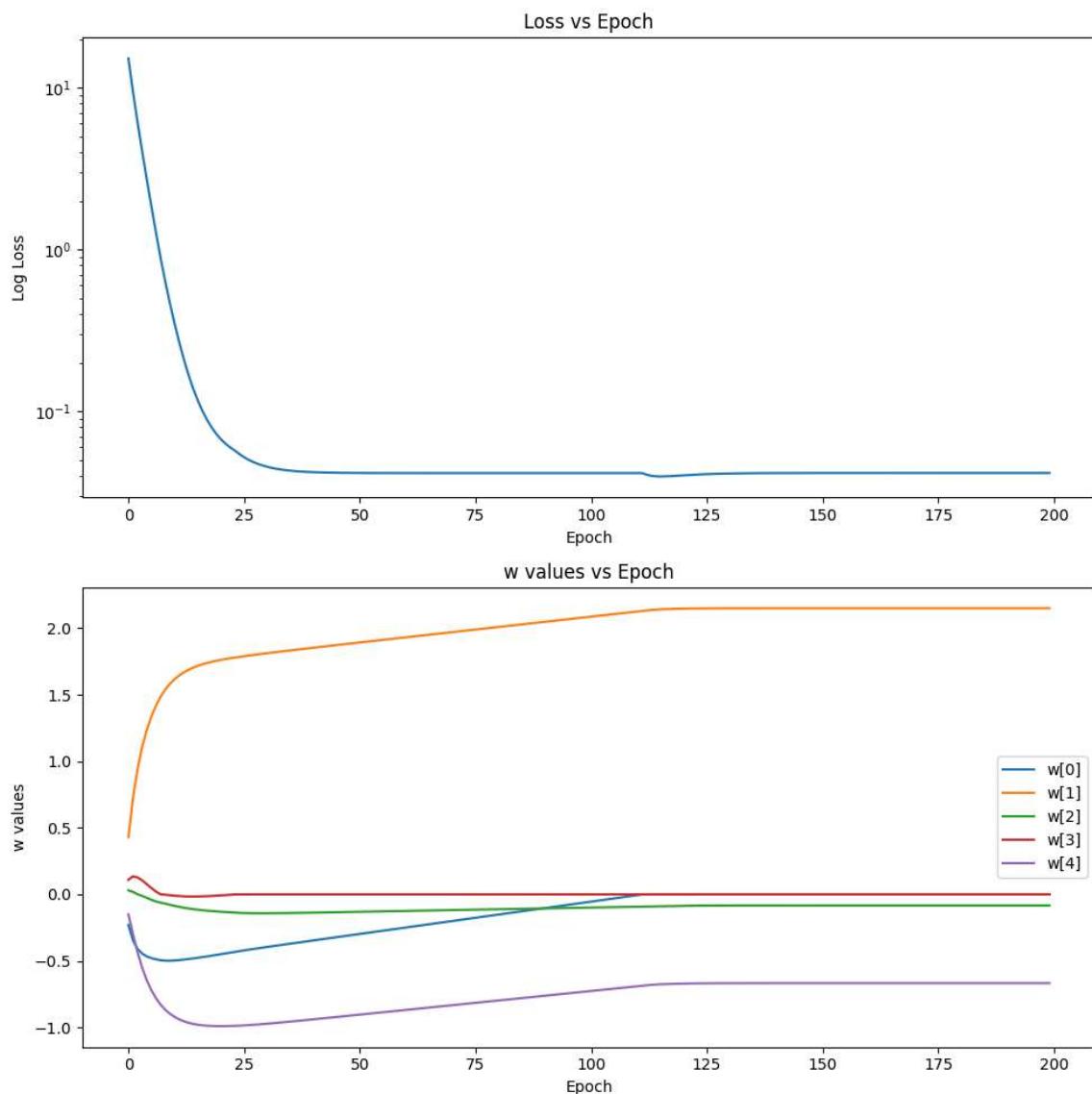


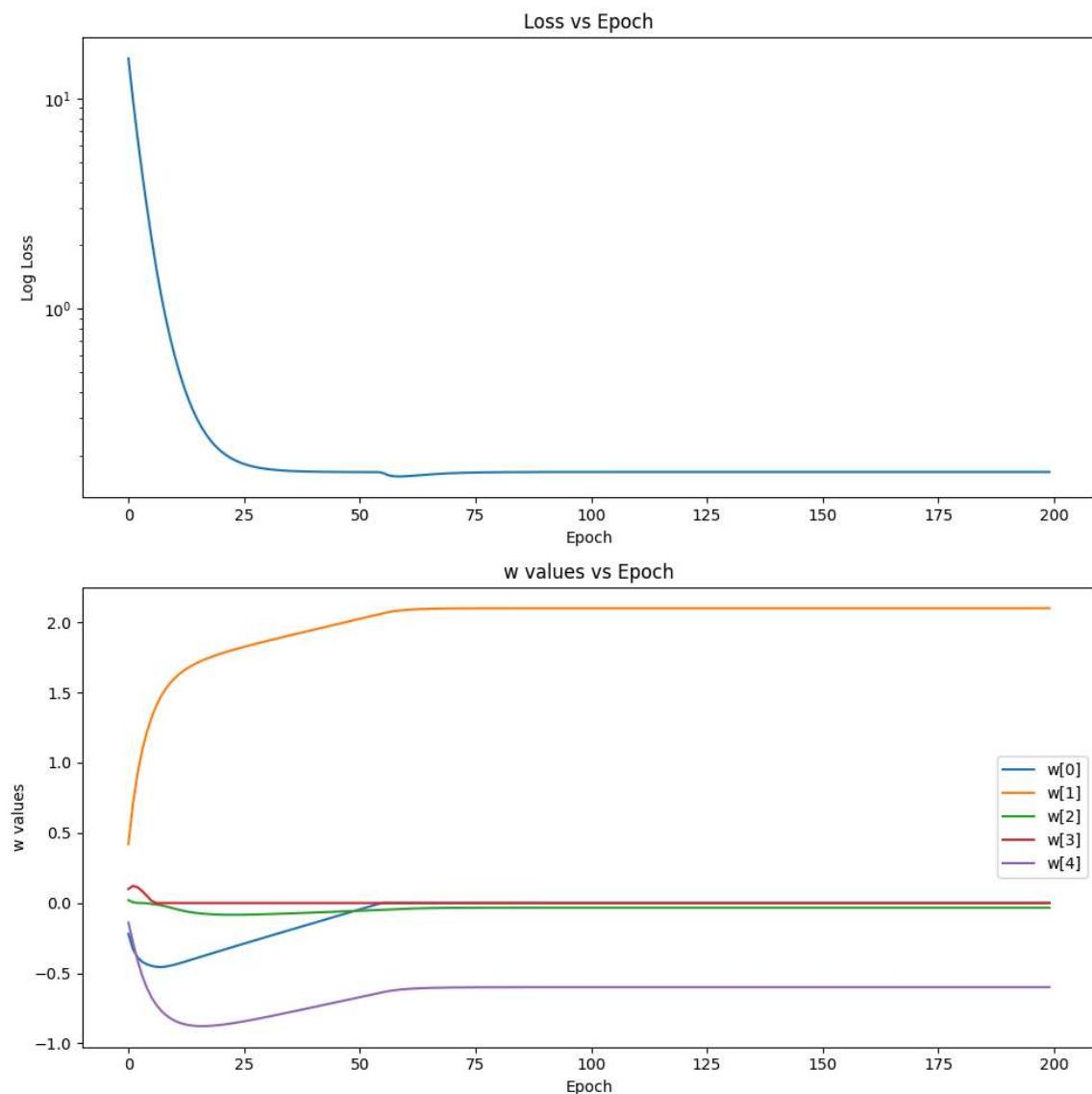
What we see with increasing L1 strength is that the model weights are increasingly pushed towards zero leading to a sparser solution but this comes at the cost of higher loss values. Our model becomes more efficient (sparse) but at the expense of accuracy.

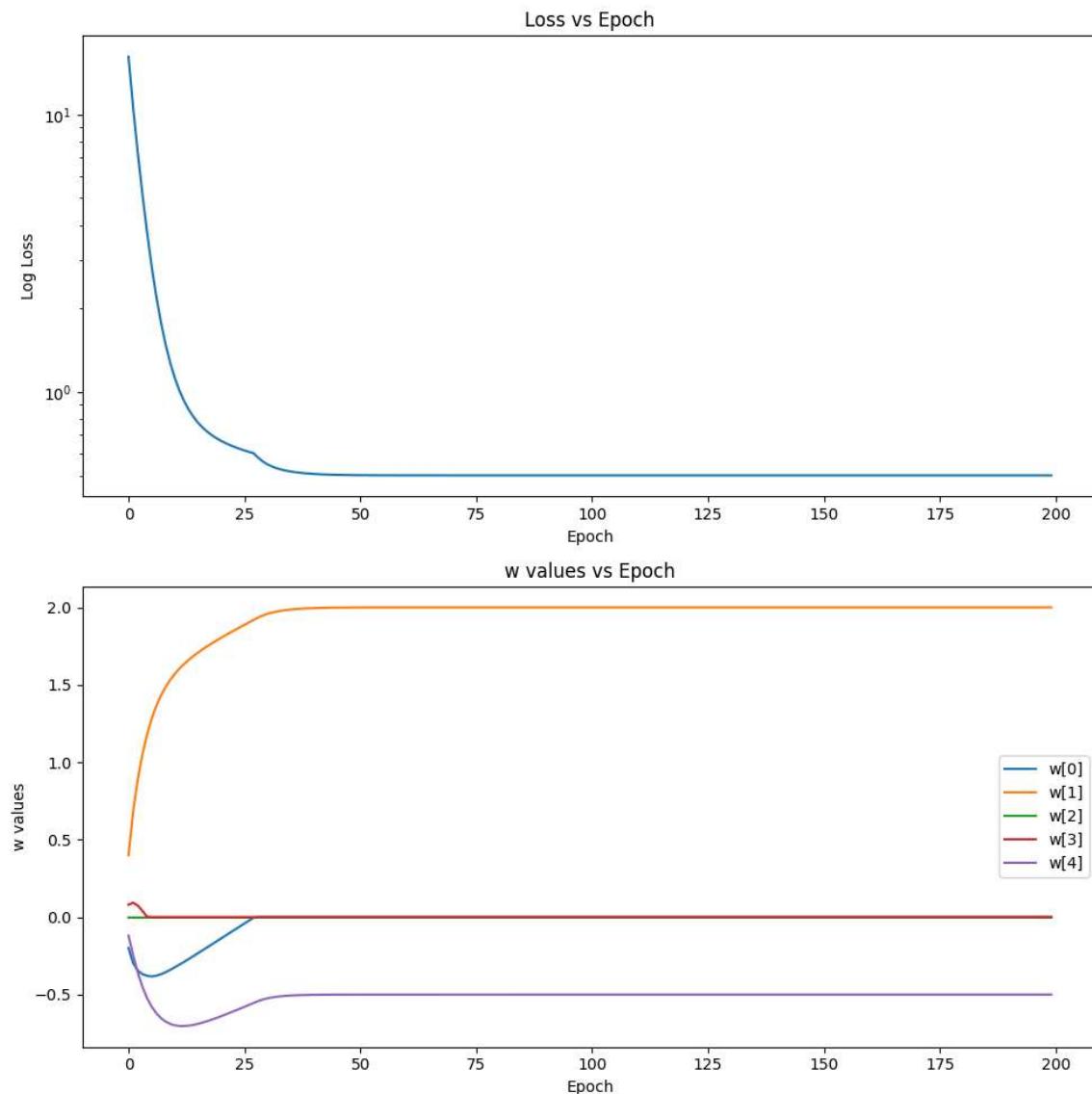
(e) (6 pts) Here we optimize the same objective as in (d), this time using proximal gradient update. Recall that the proximal operator of the ℓ_1 regularizer is the soft thresholding function. Set the threshold in the soft thresholding function to $\{0.004, 0.01, 0.02, 0.04\}$ respectively. Plot the 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 $\mu\lambda$).

We can take a look at several individual plot for proximal gradient update with varying thresholding functions.

Proximal Gradient Descent (Lambda=0.2)

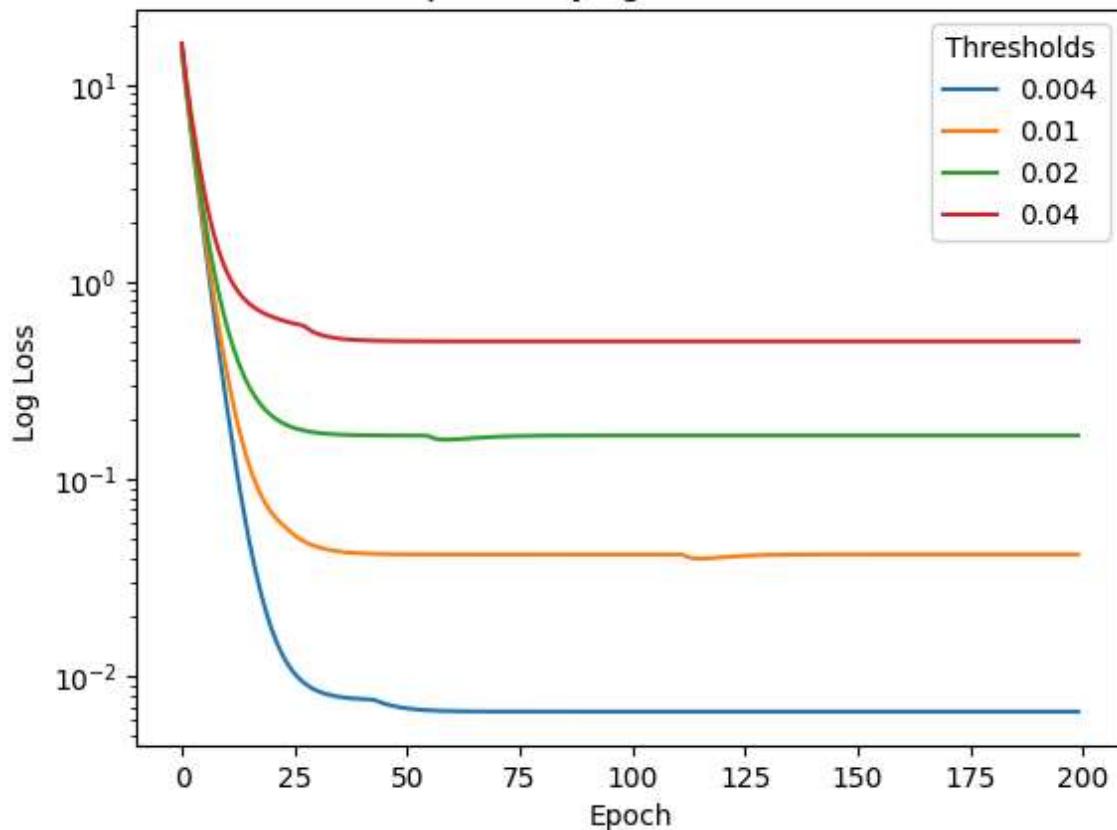
Proximal Gradient Descent (Lambda=0.5)

Proximal Gradient Descent (Lambda=1.0)

Proximal Gradient Descent (Lambda=2.0)

we also have each loss for proximal gradient update plotted together:

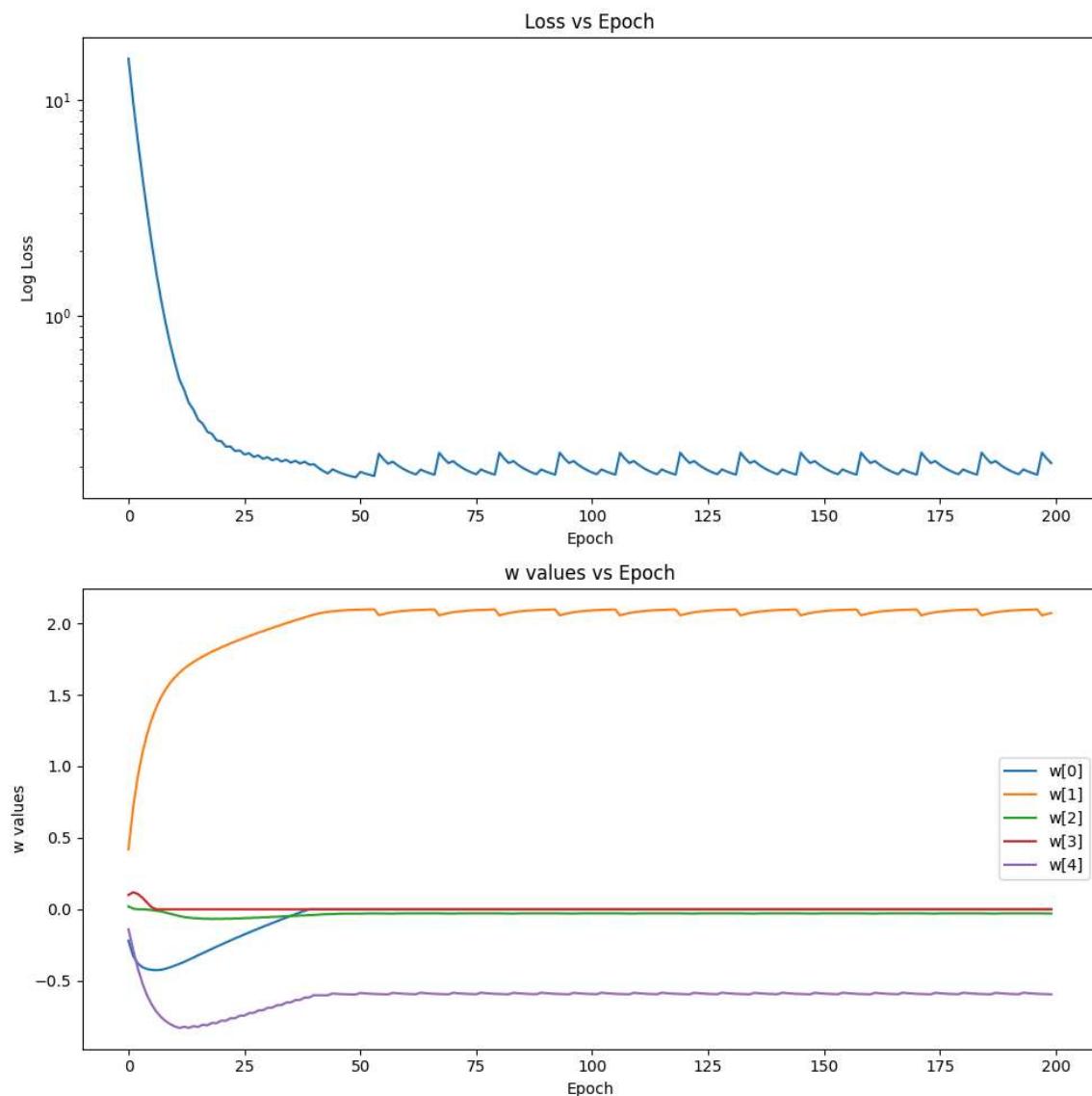
Loss vs Epoch Varying Proximal Thresholds

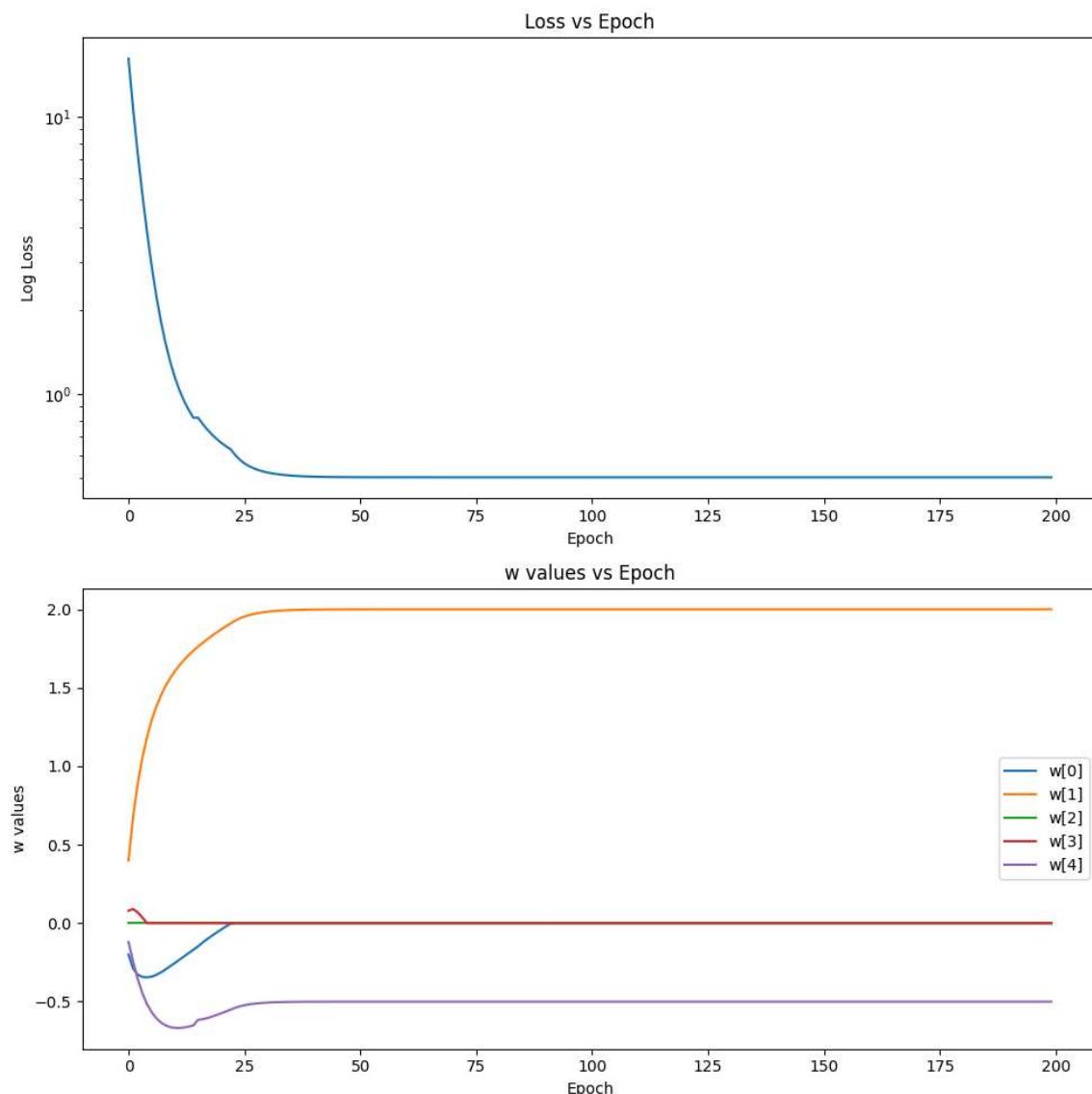


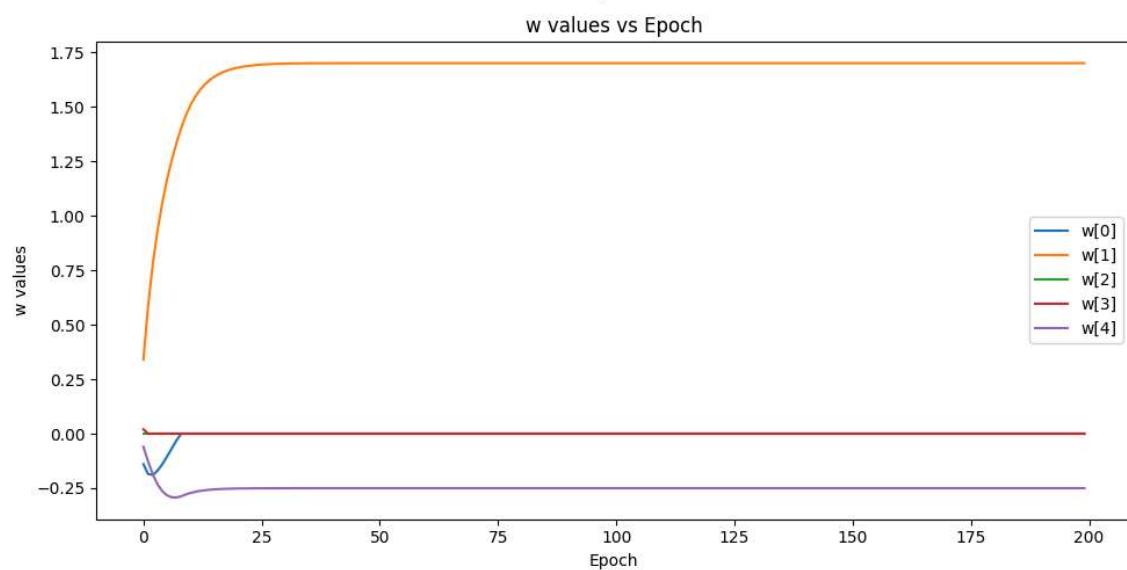
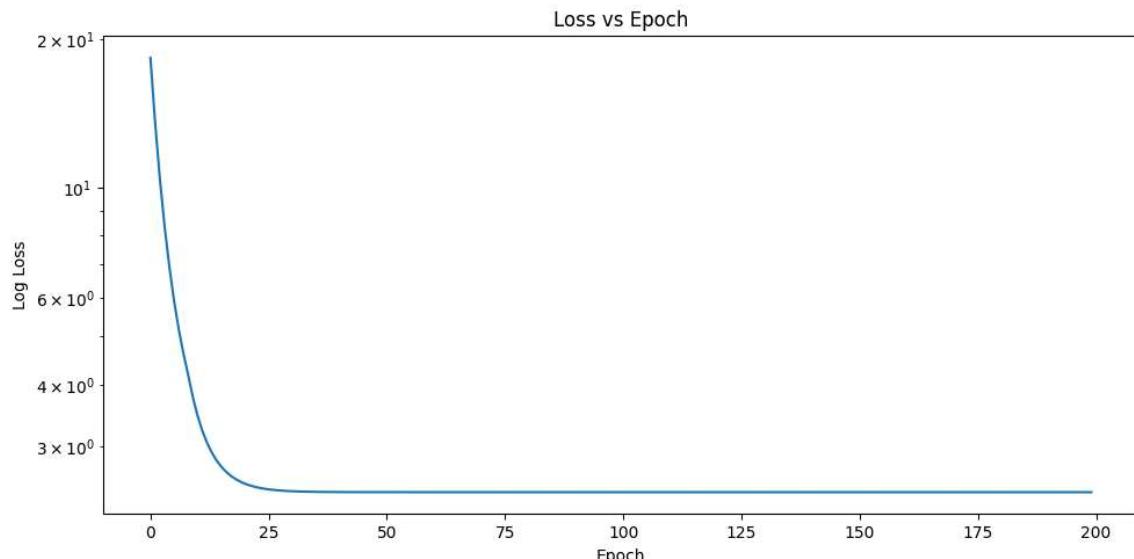
We can see that the proximal term succeeds in allowing for smoother convergence toward the overall objective. We see the same tradeoff for L1 regularization in that we can incentivize sparser solutions at the expense of loss.

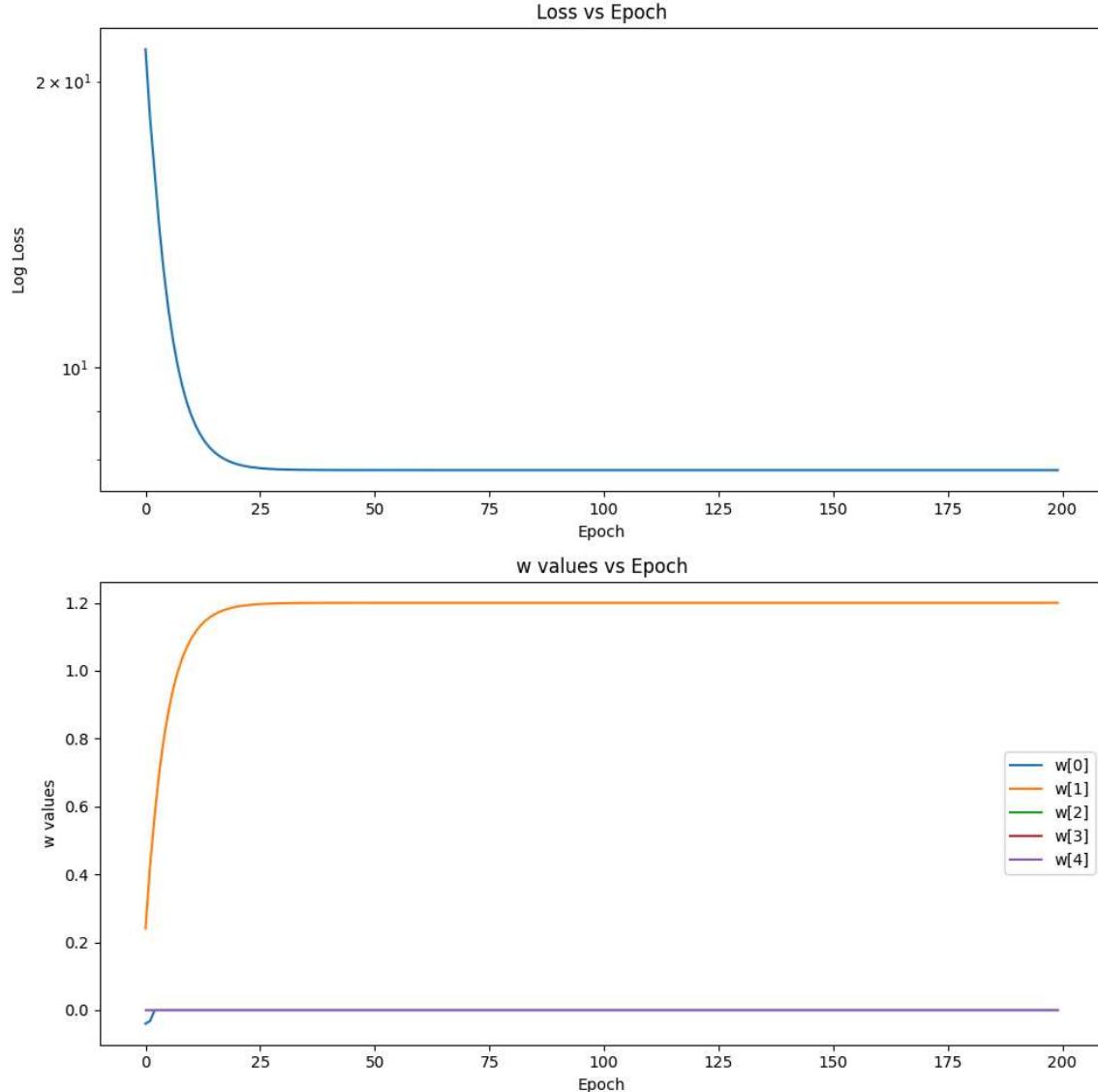
(f) (6 pts) Trimmed ℓ_1 ($T\ell_1$) regularizer is proposed to solve the “bias” problem of ℓ_1 . For simplicity you may implement the $T\ell_1$ regularizer as applying a ℓ_1 regularization with strength λ on the 3 elements of W with the smallest absolute value, with no penalty on other elements. Minimize $L + \lambda T\ell_1(W)$ using proximal gradient update with $\lambda = \{1.0, 2.0, 5.0, 10.0\}$ (correspond to 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.

Here we look at loss and weight curves for multiple lambda values with trimming:

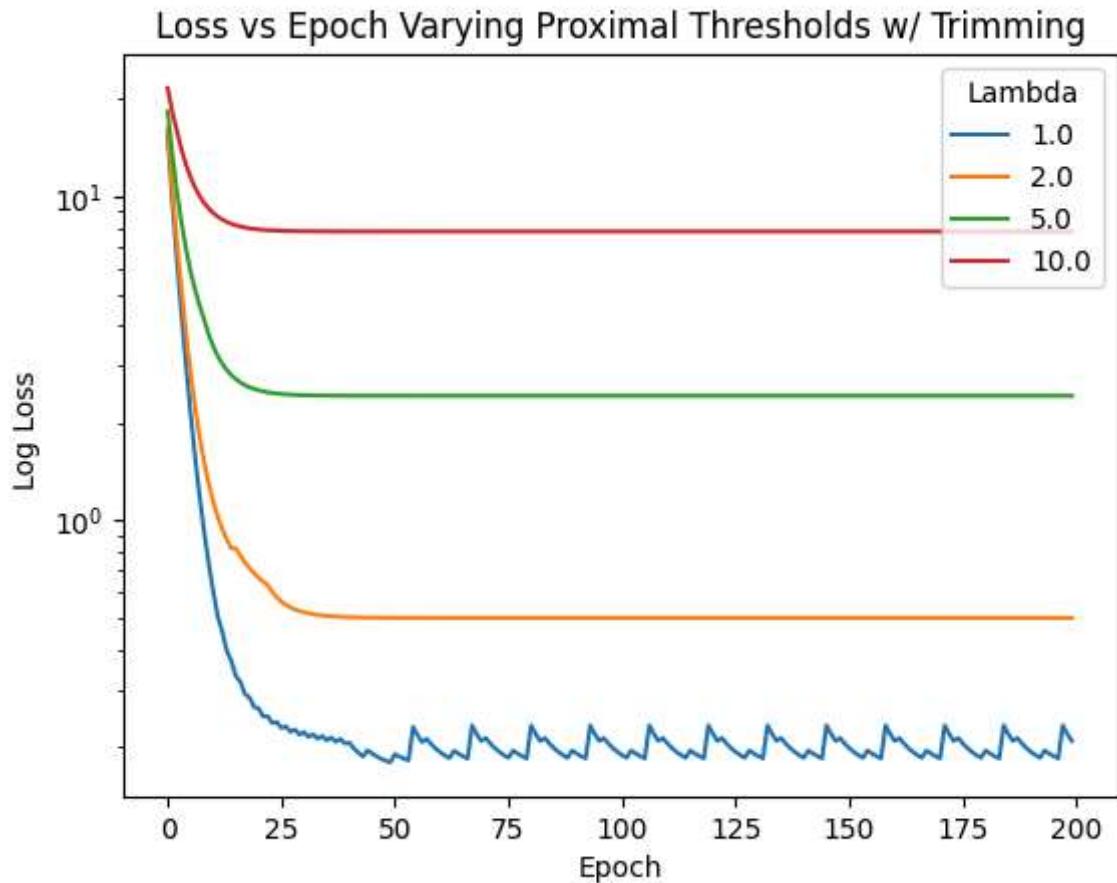
Trimmed L1 Reg Gradient Descent (Lambda=1.0)

Trimmed L1 Reg Gradient Descent (Lambda=2.0)

Trimmed L1 Reg Gradient Descent (Lambda=5.0)

Trimmed L1 Reg Gradient Descent (Lambda=10.0)

And we can look at all the loss curves on top of each other



We can see that the iterative pruning yields very similar weight behavior as our L1 trimming. This is most obvious in our trimming with strength 5, which selects the same weights albeit with lower magnitude.

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, conducting 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) (2pts) In hw4.ipynb, run through the first three code block, report the accuracy of the floatingpoint pretrained model.

```

❷ # Load the best weight parameters
net.load_state_dict(torch.load("/content/drive/My Drive/DLHW4/pretrained_model.pt"))
test(net)

❸ <ipython-input-2d-2aefb02ae3>:2: FutureWarning: You are using 'torch.load' with 'weights_only=False' (the current default value), which uses the default pickle module implicitly. It is possible to construct
    net.load_state_dict(torch.load("/content/drive/My Drive/DLHW4/pretrained_model.pt"))
  Downloading https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz to ./data/cifar-10-python.tar.gz
  100% |██████████| 170M/170M [00:04<00:00, 35.4MB/s]
  Extracting ./data/cifar-10-python.tar.gz to ./data
  Test Loss=0.3231, Test accuracy=0.9151

```

(b) (6pts) 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).

Pruning implementation:

```

❶ def prune_by_percentage(layer, q=70.0):
    """
    Pruning the weight parameters by threshold.
    :param q: pruning percentile. 'q' percent of the least
    significant weight parameters will be pruned.
    """
    # Convert the weight of "layer" to numpy array
    weights = layer.weight.data.cpu().numpy()
    # Compute the q-th percentile of the abs of the converted array
    threshold = np.percentile(np.abs(weights), q)
    # Generate a binary mask same shape as weight to decide which element to prune
    mask = np.abs(weights) >= threshold
    # Convert mask to torch tensor and put on GPU
    mask_tensor = torch.tensor(mask, dtype=torch.float32, device=layer.weight.device)
    # Multiply the weight by mask to perform pruning
    layer.weight.data.mul_(mask_tensor)

pass

```

0.7 Pruning results:

```

❷ net.load_state_dict(torch.load("/content/drive/My Drive/DLHW4/pretrained_model.pt"))

for name, layer in net.named_modules():
    if (
        isinstance(layer, nn.Conv2d) or isinstance(layer, nn.Linear)
    ) and "id_mapping" not in name:
        # change q value
        prune_by_percentage(layer, q=70.0)

    # 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)

<ipython-input-2d-2aefb02ae3>:1: FutureWarning: You are using 'torch.load' with 'weights_only=False' (the current default value), which uses the default pickle module implicitly. It is p
net.load_state_dict(torch.load("/content/drive/My Drive/DLHW4/pretrained_model.pt"))
Sparsity of head.conv0.conv: 0.699974074074074
Sparsity of body_op_0.conv1_0.conv: 0.7000680555555556
Sparsity of body_op_0.conv2_0.conv: 0.7000680555555556
Sparsity of body_op_1.conv1_0.conv: 0.7000680555555556
Sparsity of body_op_1.conv2_0.conv: 0.7000680555555556
Sparsity of body_op_2.conv1_0.conv: 0.7000680555555556
Sparsity of body_op_2.conv2_0.conv: 0.7000680555555556
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: 0.700054253472222
Sparsity of body_op_7.conv1_0.conv: 0.700054253472222
Sparsity of body_op_7.conv2_0.conv: 0.700054253472222
Sparsity of body_op_8.conv1_0.conv: 0.700054253472222
Sparsity of body_op_8.conv2_0.conv: 0.700054253472222
Sparsity of final_fc.linear: 0.7
Files already downloaded and verified
Test Loss=2.4417, Test accuracy=0.4204

```

0.5 Pruning results:

```

net.load_state_dict(torch.load("/content/drive/My Drive/DLH4/pretrained_model.pt"))

for name, layer in net.named_modules():
    if (
        isinstance(layer, nn.Conv2d) or isinstance(layer, nn.Linear)
    ) and "id_mapping" not in name:
        # change q value
        prune_by_percentage(layer, q=50.0)

        # 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)

<ipython-input-24-257f1763c857>:1: FutureWarning: You are using `torch.load` with `weights_only=False` (the current default value), which uses the default pickle module implicitly. It is recommended to use `weights_only=True` to avoid potential security risks.
net.load_state_dict(torch.load("/content/drive/My Drive/DLH4/pretrained_model.pt"))
sparsity of head_conv_0.conv: 0.5
Sparsity of body_op_0.conv1_0.conv: 0.5
Sparsity of body_op_0.conv2_0.conv: 0.5
Sparsity of body_op_1.conv1_0.conv: 0.5
Sparsity of body_op_1.conv2_0.conv: 0.5
Sparsity of body_op_2.conv1_0.conv: 0.5
Sparsity of body_op_2.conv2_0.conv: 0.5
Sparsity of body_op_3.conv1_0.conv: 0.5
Sparsity of body_op_3.conv2_0.conv: 0.5
Sparsity of body_op_4.conv1_0.conv: 0.5
Sparsity of body_op_4.conv2_0.conv: 0.5
Sparsity of body_op_5.conv1_0.conv: 0.5
Sparsity of body_op_5.conv2_0.conv: 0.5
Sparsity of body_op_6.conv1_0.conv: 0.5
Sparsity of body_op_6.conv2_0.conv: 0.5
Sparsity of body_op_7.conv1_0.conv: 0.5
Sparsity of body_op_7.conv2_0.conv: 0.5
Sparsity of body_op_8.conv1_0.conv: 0.5
Sparsity of body_op_8.conv2_0.conv: 0.5
Sparsity of final_fc.linear: 0.5
Files already downloaded and verified
Test Loss=0.6774, Test accuracy=0.8210

```

0.3 Pruning results:

```

net.load_state_dict(torch.load("/content/drive/My Drive/DLH4/pretrained_model.pt"))

for name, layer in net.named_modules():
    if (
        isinstance(layer, nn.Conv2d) or isinstance(layer, nn.Linear)
    ) and "id_mapping" not in name:
        # change q value
        prune_by_percentage(layer, q=30.0)

        # 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)

<ipython-input-25-3157261b996c>:1: FutureWarning: You are using `torch.load` with `weights_only=False` (the current default value), which uses the default pickle module implicitly. It is recommended to use `weights_only=True` to avoid potential security risks.
net.load_state_dict(torch.load("/content/drive/My Drive/DLH4/pretrained_model.pt"))
Sparsity of head_conv_0.conv: 0.2999131944444444
Sparsity of body_op_0.conv1_0.conv: 0.2999131944444444
Sparsity of body_op_0.conv2_0.conv: 0.2999131944444444
Sparsity of body_op_1.conv1_0.conv: 0.2999131944444444
Sparsity of body_op_1.conv2_0.conv: 0.2999131944444444
Sparsity of body_op_2.conv1_0.conv: 0.2999131944444444
Sparsity of body_op_2.conv2_0.conv: 0.2999131944444444
Sparsity of body_op_3.conv1_0.conv: 0.3001302883333333
Sparsity of body_op_3.conv2_0.conv: 0.3000217013888889
Sparsity of body_op_4.conv1_0.conv: 0.3000217013888889
Sparsity of body_op_4.conv2_0.conv: 0.3000217013888889
Sparsity of body_op_5.conv1_0.conv: 0.3000217013888889
Sparsity of body_op_5.conv2_0.conv: 0.3000217013888889
Sparsity of body_op_6.conv1_0.conv: 0.2999945746527778
Sparsity of body_op_6.conv2_0.conv: 0.2999945746527778
Sparsity of body_op_7.conv1_0.conv: 0.2999945746527778
Sparsity of body_op_7.conv2_0.conv: 0.2999945746527778
Sparsity of body_op_8.conv1_0.conv: 0.2999945746527778
Sparsity of body_op_8.conv2_0.conv: 0.2999945746527778
Sparsity of final_fc.linear: 0.3
Files already downloaded and verified
Test Loss=0.3698, Test accuracy=0.9092

```

(c) (6pts) 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.

Here is the finetune code:

```

❶ 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.detach().cpu().numpy() != 0).astype("float")
            weight_mask[name] = torch.tensor(
                weight_mask[name], dtype=torch.float32, device=layer.weight.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=%f\tacc=%f\t%1f examples/second"
            % (
                global_steps,
                train_loss / (batch_idx + 1),
                (correct / total),
                num_examples_per_second,
            )
        )
    start = time.time()

```

and the results of the fine tuned model:

```

❷ # Check sparsity of the finetuned model, make sure it's not changed
net.load_state_dict(torch.load("/content/drive/My Drive/DLHW4/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)

❸ <ipython-input-22-2b35f64faef01>:2: FutureWarning: You are using 'torch.load' with 'weights_only=False' (the current default value), which uses the default pickle module implicitly. It is possible to construct
net.load_state_dict(torch.load("/content/drive/My Drive/DLHW4/net_after_finetune.pt"))
Sparsity of head_conv_0.conv: 0.6998688689555556
Sparsity of body_op_0.conv1.0.conv: 0.7800088689555556
Sparsity of body_op_0.conv1.1.conv: 0.6999782986111112
Sparsity of body_op_1.conv1.0.conv: 0.7800088689555556
Sparsity of body_op_1.conv1.1.conv: 0.6999782986111112
Sparsity of body_op_2.conv1.0.conv: 0.7800088689555556
Sparsity of body_op_2.conv1.1.conv: 0.6999782986111112
Sparsity of body_op_3.conv1.0.conv: 0.6999869791666666
Sparsity of body_op_3.conv1.1.conv: 0.6999782986111112
Sparsity of body_op_4.conv1.0.conv: 0.6999782986111112
Sparsity of body_op_4.conv1.1.conv: 0.6999782986111112
Sparsity of body_op_5.conv1.0.conv: 0.6999782986111112
Sparsity of body_op_5.conv1.1.conv: 0.6999782986111112
Sparsity of body_op_6.conv1.0.conv: 0.6999782986111112
Sparsity of body_op_6.conv1.1.conv: 0.7800054253472222
Sparsity of body_op_7.conv1.0.conv: 0.7800054253472222
Sparsity of body_op_7.conv1.1.conv: 0.7800054253472222
Sparsity of body_op_8.conv1.0.conv: 0.7800054253472222
Sparsity of body_op_8.conv1.1.conv: 0.7800054253472222
Sparsity of final_fc.linear: 0.7
Files already downloaded and verified
Test Loss=0.337, Test accuracy=0.8943

```

we can see that the sparsification of the models has not changed but the accuracy has increased from 0.4204 to 0.8943 which is a significant improvement given that the model is now 70% sparse.

(d) (5pts) Implement iterative pruning. Instead of applying single step pruning before finetuning, 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 $(7 \times e)\%$ 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)

```

● net.load_state_dict(torch.load("./content/drive/My Drive/DLHW4/pretrained_model.pt"))
best_acc = 0.0
for epoch in range(20):
    print("\nEpoch: %d" % epoch)
    net.train()
    if epoch < 10:
        for name, layer in net.named_modules():
            if (
                isinstance(layer, nn.Conv2d) or isinstance(layer, nn.Linear)
            ) and "id_mapping" not in name:
                # Increase model sparsity
                q = (epoch + 1) *
                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=%f, Test acc=%f" % (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(), "/content/drive/My Drive/DLHW4/net_after_iterative_prune.pt")

```

Here are the results:

```

● # Check sparsity of the final model, make sure it's 70%
net.load_state_dict(torch.load("./content/drive/My Drive/DLHW4/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:
        # 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))
#updates
test(net)

#>>> cipython-3.8-py38chde978>:2: FutureWarning: You are using `torch.load` with `weights_only=False` (the current default value), which uses the default pickle module implicitly. It is possible to construct
#>>> net.load_state_dict(torch.load("./content/drive/My Drive/DLHW4/net_after_iterative_prune.pt"))
Sparsity of head.conv0.conv: 0.6998688055555556
Sparsity of body.op_0.conv1.0.conv: 0.7000868055555556
Sparsity of body.op_0.conv1.0.conv: 0.7000868055555556
Sparsity of body.op_1.conv1.0.conv: 0.7000868055555556
Sparsity of body.op_1.conv1.0.conv: 0.7000868055555556
Sparsity of body.op_2.conv1.0.conv: 0.6999697916666666
Sparsity of body.op_2.conv1.0.conv: 0.6999697916666666
Sparsity of body.op_3.conv1.0.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_9.conv2.0.conv: 0.7000054253472222
Sparsity of fc.linear.0.conv: 0.7000054253472222
Files already downloaded and verified
Test Loss=0.3387, Test accuracy=0.8919

```

It appears that full pruning does slightly better

(e) (6pts) 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.

Global iterative prunign implementation:

```

def global_prune_by_percentage(net, q=70.0):
    """
    Pruning the weight parameters 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
            weight = layer.weight.data.cpu().numpy()
            # Flatten weight and append to flattened_weights
            flattened_weights.append(weight.flatten())
    # Concat all weights into a np array
    flattened_weights = np.concatenate(flattened_weights)
    # Find global pruning threshold
    thres = np.percentile(np.abs(flattened_weights), q)
    # Prune the model

    # 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 prune
            mask = np.abs(weight) >= thres
            # Convert mask to torch tensor and put on GPU
            mask_tensor = torch.tensor(mask, dtype=torch.float32, device=layer.weight.device)
            # Multiply the weight by mask to perform pruning
            layer.weight.data.mul_(mask_tensor)

```

Results:

```

net.load_state_dict(torch.load("/content/drive/My Drive/DLHW4/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)

<ipython-input-32-e49f61d38fb>:1: FutureWarning: You are using 'torch.load' with 'weights_only=False' (the current default value), which uses the default pickle module implicitly. It is possible to construct
net.load_state_dict(torch.load("/content/drive/My Drive/DLHW4/net_after_global_iterative_prune.pt"))
Sparsity of head_conv.0.conv: 0.2453703783787388
Sparsity of body_op.0.conv: 0.2222222222222222
Sparsity of body_op.0.conv2.0.conv: 0.5277777777777778
Sparsity of body_op.1.conv1.0.conv: 0.5196527777777778
Sparsity of body_op.1.conv2.0.conv: 0.5533854166666666
Sparsity of body_op.2.conv1.0.conv: 0.5169398555555556
Sparsity of body_op.2.conv2.0.conv: 0.5659722222222222
Sparsity of body_op.3.conv1.0.conv: 0.5264916666666666
Sparsity of body_op.3.conv2.0.conv: 0.5161161161116111
Sparsity of body_op.4.conv1.0.conv: 0.61594677777778
Sparsity of body_op.4.conv2.0.conv: 0.6777083472222222
Sparsity of body_op.5.conv1.0.conv: 0.6115451388888888
Sparsity of body_op.5.conv2.0.conv: 0.703598277777778
Sparsity of body_op.6.conv1.0.conv: 0.6151881215277778
Sparsity of body_op.6.conv2.0.conv: 0.650687934927778
Sparsity of body_op.7.conv1.0.conv: 0.6260562605626056
Sparsity of body_op.7.conv2.0.conv: 0.719670138888888
Sparsity of body_op.8.conv1.0.conv: 0.4791124131944444
Sparsity of body_op.8.conv2.0.conv: 0.5371747916666666
Sparsity of final_fc.linear: 0.1283125
Total sparsity of: 0.6999992546565792
Files already downloaded and verified
Test Loss:0.318, test accuracy:0.897

```

Here we see our best performance yet with global iterative pruning outperforming the other methods.

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

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 its performance and apply finetuning on the model.

(a) (15 pts) As is mentioned in lecture 15, to train a quantized model we need to use floatingpoint 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.)

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

                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 * beta + alpha

            # Lab3 (e), Your code here:
            else:
                pass

            # Restore zero elements in wq
            wq = wq * weight_mask

        return wq
```

(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.)

Nbits = 6

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

net = ResNetCIFAR(num_layers=20, NbBits=NbBits)
net = net.to(device)
net.load_state_dict(torch.load("./content/drive/My Drive/DLHW4/pretrained_model.pt"))
test(net)

cipython-input-7-83de8eb67baf>:6: FutureWarning: You are using `torch.load` with `weights_only=False` (the current default value), which uses the default | net.load_state_dict(torch.load("./content/drive/My Drive/DLHW4/pretrained_model.pt"))
Files already downloaded and verified
Test Loss=0.3364, Test accuracy=0.9145
```

NbBits = 5

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

net = ResNetCIFAR(num_layers=20, NbBits=NbBits)
net = net.to(device)
net.load_state_dict(torch.load("./content/drive/My Drive/DLHW4/pretrained_model.pt"))
test(net)

cipython-input-8-946295d7d0a8>:6: FutureWarning: You are using `torch.load` with `weights_only=False` (the current default value), which uses the default | net.load_state_dict(torch.load("./content/drive/My Drive/DLHW4/pretrained_model.pt"))
Files already downloaded and verified
Test Loss=0.3390, Test accuracy=0.9112
```

NbBits = 4

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

net = ResNetCIFAR(num_layers=20, NbBits=NbBits)
net = net.to(device)
net.load_state_dict(torch.load("./content/drive/My Drive/DLHW4/pretrained_model.pt"))
test(net)

cipython-input-5-48327ccf2db4>:6: FutureWarning: You are using `torch.load` with `weights_only=False` (the current default value), which uses the default | net.load_state_dict(torch.load("./content/drive/My Drive/DLHW4/pretrained_model.pt"))
Downloading https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz to ./data/cifar-10-python.tar.gz
100%[=====] 170W/170W [00:04<00:00, 42.5MB/s]
Extracting ./data/cifar-10-python.tar.gz to ./data
Test Loss=0.3861, Test accuracy=0.8972
```

NbBits = 3

```
# Define quantized model and load weight
NbBits = 3 # Change this value to finish (b) and (c)

net = ResNetCIFAR(num_layers=20, NbBits=NbBits)
net = net.to(device)
net.load_state_dict(torch.load("./content/drive/My Drive/DLHW4/pretrained_model.pt"))
test(net)

cipython-input-9-707521cecad>:6: FutureWarning: You are using `torch.load` with `weights_only=False` (the current default value), which uses the default | net.load_state_dict(torch.load("./content/drive/My Drive/DLHW4/pretrained_model.pt"))
Files already downloaded and verified
Test Loss=0.9874, Test accuracy=0.7662
```

NbBits = 2

```
# Define quantized model and load weight
NbBits = 2 # Change this value to finish (b) and (c)

net = ResNetCIFAR(num_layers=20, NbBits=NbBits)
net = net.to(device)
net.load_state_dict(torch.load("./content/drive/My Drive/DLHW4/pretrained_model.pt"))
test(net)

cipython-input-10-cf41695ac89d>:6: FutureWarning: You are using `torch.load` with `weights_only=False` (the current default value), which uses the default | net.load_state_dict(torch.load("./content/drive/My Drive/DLHW4/pretrained_model.pt"))
Files already downloaded and verified
Test Loss=9.5441, Test accuracy=0.0899
```

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

Nbits = 4 (test acc = 0.9134)

```
[45] # Define quantized model and load weight
# Multi-bit: change this value to float (N) and (1)
# net = MultiBitNet()
# net.load_state_dict(torch.load("./content/drive/My Drive/UMN/pretrained_model.pt"))
# net.load_state_dict(torch.load("./content/drive/My Drive/UMN/quantized_net_after_finetune.pt"))

22 [45]: import os
# FutureWarning: You are using `torch.load` with 'weights_only=True' (the current default value), which uses the default pickle module implicitly. It is possible to construct malicious pickle data which will execute arbitrary code during unpickling (See https://github.com/pytorch/pytorch/pull/50417)
net.load_state_dict(torch.load("./content/drive/My Drive/UMN/quantized_net_after_finetune.pt"))
# net.load_state_dict(torch.load("./content/drive/My Drive/UMN/pretrained_model.pt"))

23 [45]: print("Test Loss={:.4f}, Test acc={:.4f}%".format(test_loss, test_accuracy))
# Test Loss=0.3631, Test accuracy=0.9134
# 4

[46] # Quantized model finetuning
# fine tune net, epoch=20, batch_size=256, lr=0.001, reg=0.4
# net = MultiBitNet()
# net.load_state_dict(torch.load("./content/drive/My Drive/UMN/quantized_net_after_finetune.pt"))

22 [46]: from time import time
# FutureWarning: You are using `torch.load` with 'weights_only=True' (the current default value), which uses the default pickle module implicitly. It is possible to construct malicious pickle data which will execute arbitrary code during unpickling (See https://github.com/pytorch/pytorch/pull/50417)
net.load_state_dict(torch.load("./content/drive/My Drive/UMN/pretrained_model.pt"))

23 [46]: print("Epoch: 0")
print("Step: 0000 Loss={:.4f} acc={:.4f}%".format(0.3631, 91.34))
print("Step: 0001 Loss={:.4f} acc={:.4f}%".format(0.3613, 91.51))
print("Step: 0002 Loss={:.4f} acc={:.4f}%".format(0.3595, 91.68))
print("Step: 0003 Loss={:.4f} acc={:.4f}%".format(0.3578, 91.85))
print("Step: 0004 Loss={:.4f} acc={:.4f}%".format(0.3561, 92.02))
print("Step: 0005 Loss={:.4f} acc={:.4f}%".format(0.3544, 92.19))
print("Step: 0006 Loss={:.4f} acc={:.4f}%".format(0.3527, 92.36))
print("Step: 0007 Loss={:.4f} acc={:.4f}%".format(0.351, 92.53))
print("Step: 0008 Loss={:.4f} acc={:.4f}%".format(0.3493, 92.7))
print("Step: 0009 Loss={:.4f} acc={:.4f}%".format(0.3476, 92.87))
print("Step: 0010 Loss={:.4f} acc={:.4f}%".format(0.3459, 93.04))
print("Step: 0011 Loss={:.4f} acc={:.4f}%".format(0.3442, 93.21))
print("Step: 0012 Loss={:.4f} acc={:.4f}%".format(0.3425, 93.38))
print("Step: 0013 Loss={:.4f} acc={:.4f}%".format(0.3408, 93.55))
print("Step: 0014 Loss={:.4f} acc={:.4f}%".format(0.3391, 93.72))
print("Step: 0015 Loss={:.4f} acc={:.4f}%".format(0.3374, 93.89))
print("Step: 0016 Loss={:.4f} acc={:.4f}%".format(0.3357, 94.06))
print("Step: 0017 Loss={:.4f} acc={:.4f}%".format(0.334, 94.23))
print("Step: 0018 Loss={:.4f} acc={:.4f}%".format(0.3323, 94.4))
print("Step: 0019 Loss={:.4f} acc={:.4f}%".format(0.3306, 94.57))
print("Step: 0020 Loss={:.4f} acc={:.4f}%".format(0.3289, 94.74))
print("Step: 0021 Loss={:.4f} acc={:.4f}%".format(0.3272, 94.91))
print("Step: 0022 Loss={:.4f} acc={:.4f}%".format(0.3255, 95.08))
print("Step: 0023 Loss={:.4f} acc={:.4f}%".format(0.3238, 95.25))
print("Step: 0024 Loss={:.4f} acc={:.4f}%".format(0.3221, 95.42))
print("Step: 0025 Loss={:.4f} acc={:.4f}%".format(0.3204, 95.59))
print("Step: 0026 Loss={:.4f} acc={:.4f}%".format(0.3187, 95.76))
print("Step: 0027 Loss={:.4f} acc={:.4f}%".format(0.317, 95.93))
print("Step: 0028 Loss={:.4f} acc={:.4f}%".format(0.3153, 96.1))
print("Step: 0029 Loss={:.4f} acc={:.4f}%".format(0.3136, 96.27))
print("Step: 0030 Loss={:.4f} acc={:.4f}%".format(0.3119, 96.44))
print("Step: 0031 Loss={:.4f} acc={:.4f}%".format(0.3102, 96.61))
print("Step: 0032 Loss={:.4f} acc={:.4f}%".format(0.3085, 96.78))
print("Step: 0033 Loss={:.4f} acc={:.4f}%".format(0.3068, 96.95))
print("Step: 0034 Loss={:.4f} acc={:.4f}%".format(0.3051, 97.12))
print("Step: 0035 Loss={:.4f} acc={:.4f}%".format(0.3034, 97.29))
print("Step: 0036 Loss={:.4f} acc={:.4f}%".format(0.3017, 97.46))
print("Step: 0037 Loss={:.4f} acc={:.4f}%".format(0.3, 97.63))
print("Step: 0038 Loss={:.4f} acc={:.4f}%".format(0.299, 97.8))
print("Step: 0039 Loss={:.4f} acc={:.4f}%".format(0.2973, 97.97))
print("Step: 0040 Loss={:.4f} acc={:.4f}%".format(0.2956, 98.14))
print("Step: 0041 Loss={:.4f} acc={:.4f}%".format(0.2939, 98.31))
print("Step: 0042 Loss={:.4f} acc={:.4f}%".format(0.2922, 98.48))
print("Step: 0043 Loss={:.4f} acc={:.4f}%".format(0.2905, 98.65))
print("Step: 0044 Loss={:.4f} acc={:.4f}%".format(0.2888, 98.82))
print("Step: 0045 Loss={:.4f} acc={:.4f}%".format(0.2871, 98.99))
print("Step: 0046 Loss={:.4f} acc={:.4f}%".format(0.2854, 99.16))
print("Step: 0047 Loss={:.4f} acc={:.4f}%".format(0.2837, 99.33))
print("Step: 0048 Loss={:.4f} acc={:.4f}%".format(0.282, 99.5))
print("Step: 0049 Loss={:.4f} acc={:.4f}%".format(0.2803, 99.67))
print("Step: 0050 Loss={:.4f} acc={:.4f}%".format(0.2786, 99.84))
print("Step: 0051 Loss={:.4f} acc={:.4f}%".format(0.2769, 99.91))
print("Step: 0052 Loss={:.4f} acc={:.4f}%".format(0.2752, 99.98))
print("Step: 0053 Loss={:.4f} acc={:.4f}%".format(0.2735, 100.05))
print("Step: 0054 Loss={:.4f} acc={:.4f}%".format(0.2718, 100.12))
print("Step: 0055 Loss={:.4f} acc={:.4f}%".format(0.2701, 100.19))
print("Step: 0056 Loss={:.4f} acc={:.4f}%".format(0.2684, 100.26))
print("Step: 0057 Loss={:.4f} acc={:.4f}%".format(0.2667, 100.33))
print("Step: 0058 Loss={:.4f} acc={:.4f}%".format(0.265, 100.4))
print("Step: 0059 Loss={:.4f} acc={:.4f}%".format(0.2633, 100.47))
print("Step: 0060 Loss={:.4f} acc={:.4f}%".format(0.2616, 100.54))
print("Step: 0061 Loss={:.4f} acc={:.4f}%".format(0.2599, 100.61))
print("Step: 0062 Loss={:.4f} acc={:.4f}%".format(0.2582, 100.68))
print("Step: 0063 Loss={:.4f} acc={:.4f}%".format(0.2565, 100.75))
print("Step: 0064 Loss={:.4f} acc={:.4f}%".format(0.2548, 100.82))
print("Step: 0065 Loss={:.4f} acc={:.4f}%".format(0.2531, 100.89))
print("Step: 0066 Loss={:.4f} acc={:.4f}%".format(0.2514, 100.96))
print("Step: 0067 Loss={:.4f} acc={:.4f}%".format(0.2497, 101.03))
print("Step: 0068 Loss={:.4f} acc={:.4f}%".format(0.248, 101.1))
print("Step: 0069 Loss={:.4f} acc={:.4f}%".format(0.2463, 101.17))
print("Step: 0070 Loss={:.4f} acc={:.4f}%".format(0.2446, 101.24))
print("Step: 0071 Loss={:.4f} acc={:.4f}%".format(0.2429, 101.31))
print("Step: 0072 Loss={:.4f} acc={:.4f}%".format(0.2412, 101.38))
print("Step: 0073 Loss={:.4f} acc={:.4f}%".format(0.2395, 101.45))
print("Step: 0074 Loss={:.4f} acc={:.4f}%".format(0.2378, 101.52))
print("Step: 0075 Loss={:.4f} acc={:.4f}%".format(0.2361, 101.59))
print("Step: 0076 Loss={:.4f} acc={:.4f}%".format(0.2344, 101.66))
print("Step: 0077 Loss={:.4f} acc={:.4f}%".format(0.2327, 101.73))
print("Step: 0078 Loss={:.4f} acc={:.4f}%".format(0.231, 101.8))
print("Step: 0079 Loss={:.4f} acc={:.4f}%".format(0.2293, 101.87))
print("Step: 0080 Loss={:.4f} acc={:.4f}%".format(0.2276, 101.94))
print("Step: 0081 Loss={:.4f} acc={:.4f}%".format(0.2259, 102.01))
print("Step: 0082 Loss={:.4f} acc={:.4f}%".format(0.2242, 102.08))
print("Step: 0083 Loss={:.4f} acc={:.4f}%".format(0.2225, 102.15))
print("Step: 0084 Loss={:.4f} acc={:.4f}%".format(0.2208, 102.22))
print("Step: 0085 Loss={:.4f} acc={:.4f}%".format(0.2191, 102.29))
print("Step: 0086 Loss={:.4f} acc={:.4f}%".format(0.2174, 102.36))
print("Step: 0087 Loss={:.4f} acc={:.4f}%".format(0.2157, 102.43))
print("Step: 0088 Loss={:.4f} acc={:.4f}%".format(0.214, 102.5))
print("Step: 0089 Loss={:.4f} acc={:.4f}%".format(0.2123, 102.57))
print("Step: 0090 Loss={:.4f} acc={:.4f}%".format(0.2106, 102.64))
print("Step: 0091 Loss={:.4f} acc={:.4f}%".format(0.2089, 102.71))
print("Step: 0092 Loss={:.4f} acc={:.4f}%".format(0.2072, 102.78))
print("Step: 0093 Loss={:.4f} acc={:.4f}%".format(0.2055, 102.85))
print("Step: 0094 Loss={:.4f} acc={:.4f}%".format(0.2038, 102.92))
print("Step: 0095 Loss={:.4f} acc={:.4f}%".format(0.2021, 102.99))
print("Step: 0096 Loss={:.4f} acc={:.4f}%".format(0.2004, 103.06))
print("Step: 0097 Loss={:.4f} acc={:.4f}%".format(0.1987, 103.13))
print("Step: 0098 Loss={:.4f} acc={:.4f}%".format(0.197, 103.2))
print("Step: 0099 Loss={:.4f} acc={:.4f}%".format(0.1953, 103.27))
print("Step: 0100 Loss={:.4f} acc={:.4f}%".format(0.1936, 103.34))
print("Step: 0101 Loss={:.4f} acc={:.4f}%".format(0.1919, 103.41))
print("Step: 0102 Loss={:.4f} acc={:.4f}%".format(0.1902, 103.48))
print("Step: 0103 Loss={:.4f} acc={:.4f}%".format(0.1885, 103.55))
print("Step: 0104 Loss={:.4f} acc={:.4f}%".format(0.1868, 103.62))
print("Step: 0105 Loss={:.4f} acc={:.4f}%".format(0.1851, 103.69))
print("Step: 0106 Loss={:.4f} acc={:.4f}%".format(0.1834, 103.76))
print("Step: 0107 Loss={:.4f} acc={:.4f}%".format(0.1817, 103.83))
print("Step: 0108 Loss={:.4f} acc={:.4f}%".format(0.180, 103.9))
print("Step: 0109 Loss={:.4f} acc={:.4f}%".format(0.1783, 103.97))
print("Step: 0110 Loss={:.4f} acc={:.4f}%".format(0.1766, 104.04))
print("Step: 0111 Loss={:.4f} acc={:.4f}%".format(0.1749, 104.11))
print("Step: 0112 Loss={:.4f} acc={:.4f}%".format(0.1732, 104.18))
print("Step: 0113 Loss={:.4f} acc={:.4f}%".format(0.1715, 104.25))
print("Step: 0114 Loss={:.4f} acc={:.4f}%".format(0.1698, 104.32))
print("Step: 0115 Loss={:.4f} acc={:.4f}%".format(0.1681, 104.39))
print("Step: 0116 Loss={:.4f} acc={:.4f}%".format(0.1664, 104.46))
print("Step: 0117 Loss={:.4f} acc={:.4f}%".format(0.1647, 104.53))
print("Step: 0118 Loss={:.4f} acc={:.4f}%".format(0.163, 104.6))
print("Step: 0119 Loss={:.4f} acc={:.4f}%".format(0.1613, 104.67))
print("Step: 0120 Loss={:.4f} acc={:.4f}%".format(0.1596, 104.74))
print("Step: 0121 Loss={:.4f} acc={:.4f}%".format(0.1579, 104.81))
print("Step: 0122 Loss={:.4f} acc={:.4f}%".format(0.1562, 104.88))
print("Step: 0123 Loss={:.4f} acc={:.4f}%".format(0.1545, 104.95))
print("Step: 0124 Loss={:.4f} acc={:.4f}%".format(0.1528, 105.02))
print("Step: 0125 Loss={:.4f} acc={:.4f}%".format(0.1511, 105.09))
print("Step: 0126 Loss={:.4f} acc={:.4f}%".format(0.1494, 105.16))
print("Step: 0127 Loss={:.4f} acc={:.4f}%".format(0.1477, 105.23))
print("Step: 0128 Loss={:.4f} acc={:.4f}%".format(0.146, 105.3))
print("Step: 0129 Loss={:.4f} acc={:.4f}%".format(0.1443, 105.37))
print("Step: 0130 Loss={:.4f} acc={:.4f}%".format(0.1426, 105.44))
print("Step: 0131 Loss={:.4f} acc={:.4f}%".format(0.1409, 105.51))
print("Step: 0132 Loss={:.4f} acc={:.4f}%".format(0.1392, 105.58))
print("Step: 0133 Loss={:.4f} acc={:.4f}%".format(0.1375, 105.65))
print("Step: 0134 Loss={:.4f} acc={:.4f}%".format(0.1358, 105.72))
print("Step: 0135 Loss={:.4f} acc={:.4f}%".format(0.1341, 105.79))
print("Step: 0136 Loss={:.4f} acc={:.4f}%".format(0.1324, 105.86))
print("Step: 0137 Loss={:.4f} acc={:.4f}%".format(0.1307, 105.93))
print("Step: 0138 Loss={:.4f} acc={:.4f}%".format(0.129, 106.0))
print("Step: 0139 Loss={:.4f} acc={:.4f}%".format(0.1273, 106.07))
print("Step: 0140 Loss={:.4f} acc={:.4f}%".format(0.1256, 106.14))
print("Step: 0141 Loss={:.4f} acc={:.4f}%".format(0.1239, 106.21))
print("Step: 0142 Loss={:.4f} acc={:.4f}%".format(0.1222, 106.28))
print("Step: 0143 Loss={:.4f} acc={:.4f}%".format(0.1205, 106.35))
print("Step: 0144 Loss={:.4f} acc={:.4f}%".format(0.1188, 106.42))
print("Step: 0145 Loss={:.4f} acc={:.4f}%".format(0.1171, 106.49))
print("Step: 0146 Loss={:.4f} acc={:.4f}%".format(0.1154, 106.56))
print("Step: 0147 Loss={:.4f} acc={:.4f}%".format(0.1137, 106.63))
print("Step: 0148 Loss={:.4f} acc={:.4f}%".format(0.112, 106.7))
print("Step: 0149 Loss={:.4f} acc={:.4f}%".format(0.1103, 106.77))
print("Step: 0150 Loss={:.4f} acc={:.4f}%".format(0.1086, 106.84))
print("Step: 0151 Loss={:.4f} acc={:.4f}%".format(0.1069, 106.91))
print("Step: 0152 Loss={:.4f} acc={:.4f}%".format(0.1052, 106.98))
print("Step: 0153 Loss={:.4f} acc={:.4f}%".format(0.1035, 107.05))
print("Step: 0154 Loss={:.4f} acc={:.4f}%".format(0.1018, 107.12))
print("Step: 0155 Loss={:.4f} acc={:.4f}%".format(0.1, 107.19))
print("Step: 0156 Loss={:.4f} acc={:.4f}%".format(0.0991, 107.26))
print("Step: 0157 Loss={:.4f} acc={:.4f}%".format(0.0974, 107.33))
print("Step: 0158 Loss={:.4f} acc={:.4f}%".format(0.0957, 107.4))
print("Step: 0159 Loss={:.4f} acc={:.4f}%".format(0.094, 107.47))
print("Step: 0160 Loss={:.4f} acc={:.4f}%".format(0.0923, 107.54))
print("Step: 0161 Loss={:.4f} acc={:.4f}%".format(0.0906, 107.61))
print("Step: 0162 Loss={:.4f} acc={:.4f}%".format(0.0889, 107.68))
print("Step: 0163 Loss={:.4f} acc={:.4f}%".format(0.0872, 107.75))
print("Step: 0164 Loss={:.4f} acc={:.4f}%".format(0.0855, 107.82))
print("Step: 0165 Loss={:.4f} acc={:.4f}%".format(0.0838, 107.89))
print("Step: 0166 Loss={:.4f} acc={:.4f}%".format(0.0821, 107.96))
print("Step: 0167 Loss={:.4f} acc={:.4f}%".format(0.0804, 108.03))
print("Step: 0168 Loss={:.4f} acc={:.4f}%".format(0.0787, 108.1))
print("Step: 0169 Loss={:.4f} acc={:.4f}%".format(0.077, 108.17))
print("Step: 0170 Loss={:.4f} acc={:.4f}%".format(0.0753, 108.24))
print("Step: 0171 Loss={:.4f} acc={:.4f}%".format(0.0736, 108.31))
print("Step: 0172 Loss={:.4f} acc={:.4f}%".format(0.0719, 108.38))
print("Step: 0173 Loss={:.4f} acc={:.4f}%".format(0.0702, 108.45))
print("Step: 0174 Loss={:.4f} acc={:.4f}%".format(0.0685, 108.52))
print("Step: 0175 Loss={:.4f} acc={:.4f}%".format(0.0668, 108.59))
print("Step: 0176 Loss={:.4f} acc={:.4f}%".format(0.0651, 108.66))
print("Step: 0177 Loss={:.4f} acc={:.4f}%".format(0.0634, 108.73))
print("Step: 0178 Loss={:.4f} acc={:.4f}%".format(0.0617, 108.8))
print("Step: 0179 Loss={:.4f} acc={:.4f}%".format(0.060, 108.87))
print("Step: 0180 Loss={:.4f} acc={:.4f}%".format(0.0583, 108.94))
print("Step: 0181 Loss={:.4f} acc={:.4f}%".format(0.0566, 109.01))
print("Step: 0182 Loss={:.4f} acc={:.4f}%".format(0.0549, 109.08))
print("Step: 0183 Loss={:.4f} acc={:.4f}%".format(0.0532, 109.15))
print("Step: 0184 Loss={:.4f} acc={:.4f}%".format(0.0515, 109.22))
print("Step: 0185 Loss={:.4f} acc={:.4f}%".format(0.0498, 109.29))
print("Step: 0186 Loss={:.4f} acc={:.4f}%".format(0.0481, 109.36))
print("Step: 0187 Loss={:.4f} acc={:.4f}%".format(0.0464, 109.43))
print("Step: 0188 Loss={:.4f} acc={:.4f}%".format(0.0447, 109.5))
print("Step: 0189 Loss={:.4f} acc={:.4f}%".format(0.043, 109.57))
print("Step: 0190 Loss={:.4f} acc={:.4f}%".format(0.0413, 109.64))
print("Step: 0191 Loss={:.4f} acc={:.4f}%".format(0.0396, 109.71))
print("Step: 0192 Loss={:.4f} acc={:.4f}%".format(0.0379, 109.78))
print("Step: 0193 Loss={:.4f} acc={:.4f}%".format(0.0362, 109.85))
print("Step: 0194 Loss={:.4f} acc={:.4f}%".format(0.0345, 109.92))
print("Step: 0195 Loss={:.4f} acc={:.4f}%".format(0.0328, 109.99))
print("Step: 0196 Loss={:.4f} acc={:.4f}%".format(0.0311, 110.06))
print("Step: 0197 Loss={:.4f} acc={:.4f}%".format(0.0294, 110.13))
print("Step: 0198 Loss={:.4f} acc={:.4f}%".format(0.0277, 110.2)))
print("Step: 0199 Loss={:.4f} acc={:.4f}%".format(0.026, 110.27))
print("Step: 0200 Loss={:.4f} acc={:.4f}%".format(0.0243, 110.34))
print("Step: 0201 Loss={:.4f} acc={:.4f}%".format(0.0226, 110.41))
print("Step: 0202 Loss={:.4f} acc={:.4f}%".format(0.0209, 110.48))
print("Step: 0203 Loss={:.4f} acc={:.4f}%".format(0.0192, 110.55))
print("Step: 0204 Loss={:.4f} acc={:.4f}%".format(0.0175, 110.62))
print("Step: 0205 Loss={:.4f} acc={:.4f}%".format(0.0158, 110.69))
print("Step: 0206 Loss={:.4f} acc={:.4f}%".format(0.0141, 110.76))
print("Step: 0207 Loss={:.4f} acc={:.4f}%".format(0.0124, 110.83))
print("Step: 0208 Loss={:.4f} acc={:.4f}%".format(0.0107, 110.9)))
print("Step: 0209 Loss={:.4f} acc={:.4f}%".format(0.009, 110.97))
print("Step: 0210 Loss={:.4f} acc={:.4f}%".format(0.0073, 111.04))
print("Step: 0211 Loss={:.4f} acc={:.4f}%".format(0.0056, 111.11))
print("Step: 0212 Loss={:.4f} acc={:.4f}%".format(0.004, 111.18))
print("Step: 0213 Loss={:.4f} acc={:.4f}%".format(0.0023, 111.25))
print("Step: 0214 Loss={:.4f} acc={:.4f}%".format(0.0006, 111.32))
print("Step: 0215 Loss={:.4f} acc={:.4f}%".format(-0.0011, 111.39))
print("Step: 0216 Loss={:.4f} acc={:.4f}%".format(-0.0028, 111.46))
print("Step: 0217 Loss={:.4f} acc={:.4f}%".format(-0.0045, 111.53))
print("Step: 0218 Loss={:.4f} acc={:.4f}%".format(-0.0062, 111.6)))
print("Step: 0219 Loss={:.4f} acc={:.4f}%".format(-0.0079, 111.67))
print("Step: 0220 Loss={:.4f} acc={:.4f}%".format(-0.0096, 111.74))
print("Step: 0221 Loss={:.4f} acc={:.4f}%".format(-0.0113, 111.81))
print("Step: 0222 Loss={:.4f} acc={:.4f}%".format(-0.013, 111.88))
print("Step: 0223 Loss={:.4f} acc={:.4f}%".format(-0.0147, 111.95))
print("Step: 0224 Loss={:.4f} acc={:.4f}%".format(-0.0164, 112.02))
print("Step: 0225 Loss={:.4f} acc={:.4f}%".format(-0.0181, 112.09))
print("Step: 0226 Loss={:.4f} acc={:.4f}%".format(-0.0198, 112.16))
print("Step: 0227 Loss={:.4f} acc={:.4f}%".format(-0.0215, 112.23))
print("Step: 0228 Loss={:.4f} acc={:.4f}%".format(-0.0232, 112.3)))
print("Step: 0229 Loss={:.4f} acc={:.4f}%".format(-0.0249, 112.37))
print("Step: 0230 Loss={:.4f} acc={:.4f}%".format(-0.0266, 112.44))
print("Step: 0231 Loss={:.4f} acc={:.4f}%".format(-0.0283, 112.51))
print("Step: 0232 Loss={:.4f} acc={:.4f}%".format(-0.03, 112.58))
print("Step: 0233 Loss={:.4f} acc={:.4f}%".format(-0.0317, 112.65))
print("Step: 0234 Loss={:.4f} acc={:.4f}%".format(-0.0334, 112.72))
print("Step: 0235 Loss={:.4f} acc={:.4f}%".format(-0.0351, 112.79))
print("Step: 0236 Loss={:.4f} acc={:.4f}%".format(-0.0368, 112.86))
print("Step: 0237 Loss={:.4f} acc={:.4f}%".format(-0.0385, 112.93))
print("Step: 0238 Loss={:.4f} acc={:.4f}%".format(-0.0402, 113.0)))
print("Step: 0239 Loss={:.4f} acc={:.4f}%".format(-0.0419, 113.07))
print("Step: 0240 Loss={:.4f} acc={:.4f}%".format(-0.0436, 113.14))
print("Step: 0241 Loss={:.4f} acc={:.4f}%".format(-0.0453, 113.21))
print("Step: 0242 Loss={:.4f} acc={:.4f}%".format(-0.047, 113.28))
print("Step: 0243 Loss={:.4f} acc={:.4f}%".format(-0.0487, 113.35))
print("Step: 0244 Loss={:.4f} acc={:.4f}%".format(-0.0504, 113.42))
print("Step: 0245 Loss={:.4f} acc={:.4f}%".format(-0.0521, 113.49))
print("Step: 0246 Loss={:.4f} acc={:.4f}%".format(-0.0538, 113.56))
print("Step: 0247 Loss={:.4f} acc={:.4f}%".format(-0.0555, 113.63))
print("Step: 0248 Loss={:.4f} acc={:.4f}%".format(-0.0572, 113.7)))
print("Step: 0249 Loss={:.4f} acc={:.4f}%".format(-0.0589, 113.77))
print("Step: 0250 Loss={:.4f} acc={:.4f}%".format(-0.0606, 113.84))
print("Step: 0251 Loss={:.4f} acc={:.4f}%".format(-0.0623, 113.91))
print("Step: 0252 Loss={:.4f} acc={:.4f}%".format(-0.064, 113.98))
print("Step: 0253 Loss={:.4f} acc={:.4f}%".format(-0.0657, 114.05))
print("Step: 0254 Loss={:.4f} acc={:.4f}%".format(-0.0674, 114.12))
print("Step: 0255 Loss={:.4f} acc={:.4f}%".format(-0.0691, 114.19))
print("Step: 0256 Loss={:.4f} acc={:.4f}%".format(-0.0708, 114.26))
print("Step: 0257 Loss={:.4f} acc={:.4f}%".format(-0.0725, 114.33))
print("Step: 0258 Loss={:.4f} acc={:.4f}%".format(-0.0742, 114.4)))
print("Step: 0259 Loss={:.4f} acc={:.4f}%".format(-0.0759, 114.47))
print("Step: 0260 Loss={:.4f} acc={:.4f}%".format(-0.0776, 114.54))
print("Step: 0261 Loss={:.4f} acc={:.4f}%".format(-0.0793, 114.61))
print("Step: 0262 Loss={:.4f} acc={:.4f}%".format(-0.081, 114.68))
print("Step: 0263 Loss={:.4f} acc={:.4f}%".format(-0.0827, 114.75))
print("Step: 0264 Loss={:.4f} acc={:.4f}%".format(-0.0844, 114.82))
print("Step: 0265 Loss={:.4f} acc={:.4f}%".format(-0.0861, 114.89))
print("Step: 0266 Loss={:.4f} acc={:.4f}%".format(-0.0878, 114.96))
print("Step: 0267 Loss={:.4f} acc={:.4f}%".format(-0.0895, 115.03))
print("Step: 0268 Loss={:.4f} acc={:.4f}%".format(-0.0912, 115.1)))
print("Step: 0269 Loss={:.4f} acc={:.4f}%".format(-0.0929, 115.17))
print("Step: 0270 Loss={:.4f} acc={:.4f}%".format(-0.0946, 115.24))
print("Step: 0271 Loss={:.4f} acc={:.4f}%".format(-0.0963, 115.31))
print("Step: 0272 Loss={:.4f} acc={:.4f}%".format(-0.098, 115.38))
print("Step: 0273 Loss={:.4f} acc={:.4f}%".format(-0.0997, 115.45))
print("Step: 0274 Loss={:.4f} acc={:.4f}%".format(-0.0101, 115.52))
print("Step: 0275 Loss={:.4f} acc={:.4f}%".format(-0.0108, 115.59))
print("Step: 027
```

We can see that for lower precision (lower Nbit values) we get lower accuracy. This is expected as we are reducing the number of bits used to represent the weights. Finetuning is effective in that it allows us to recover some of the lost accuracy from quantization. In all cases the finetuning performance is much better than the quantized pretrained model.

(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 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.

Nbits = 4 (test acc = 0.9047)

Nbits = 3 (test acc = 0.8817)

Nbits = 2 (test acc = 0.8919)

This result is interesting because our nbit 2 precision actually outperformed our nbit 3 precision in terms of test accuracy. But our nbit 4 precision model performed comparably to the nbit 3 precision model with no pruning. This means that we get a quantized and pruned model that has fairly good performance and that these techniques can be used together to achieve a good balance between model size and performance.