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 strucutred 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 smothing 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 = P i(XiW − yi)2. Specifically, we have Xi ∈ R1×5, W ∈ R5×1 and ||W||0 ≤ 2. For Problem (a) - (f), consider the case where we have 3 data points: (X1 = [−1, 2, 1, 1,−1], y1 = 5); (X2 = [−2, 1,−2, 0, 2], y2 = 1); (X3 = [1, 0,−2,−2,−1], y3 = 1). For stability the objective L should be minimized through full-batch gradient descent, with initial weight W0 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. 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 isWk, derive the symbolic formulation ofweightWk+1 after step k+1 of full-batch gradient descent with Xi, yi, i ∈ {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:

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

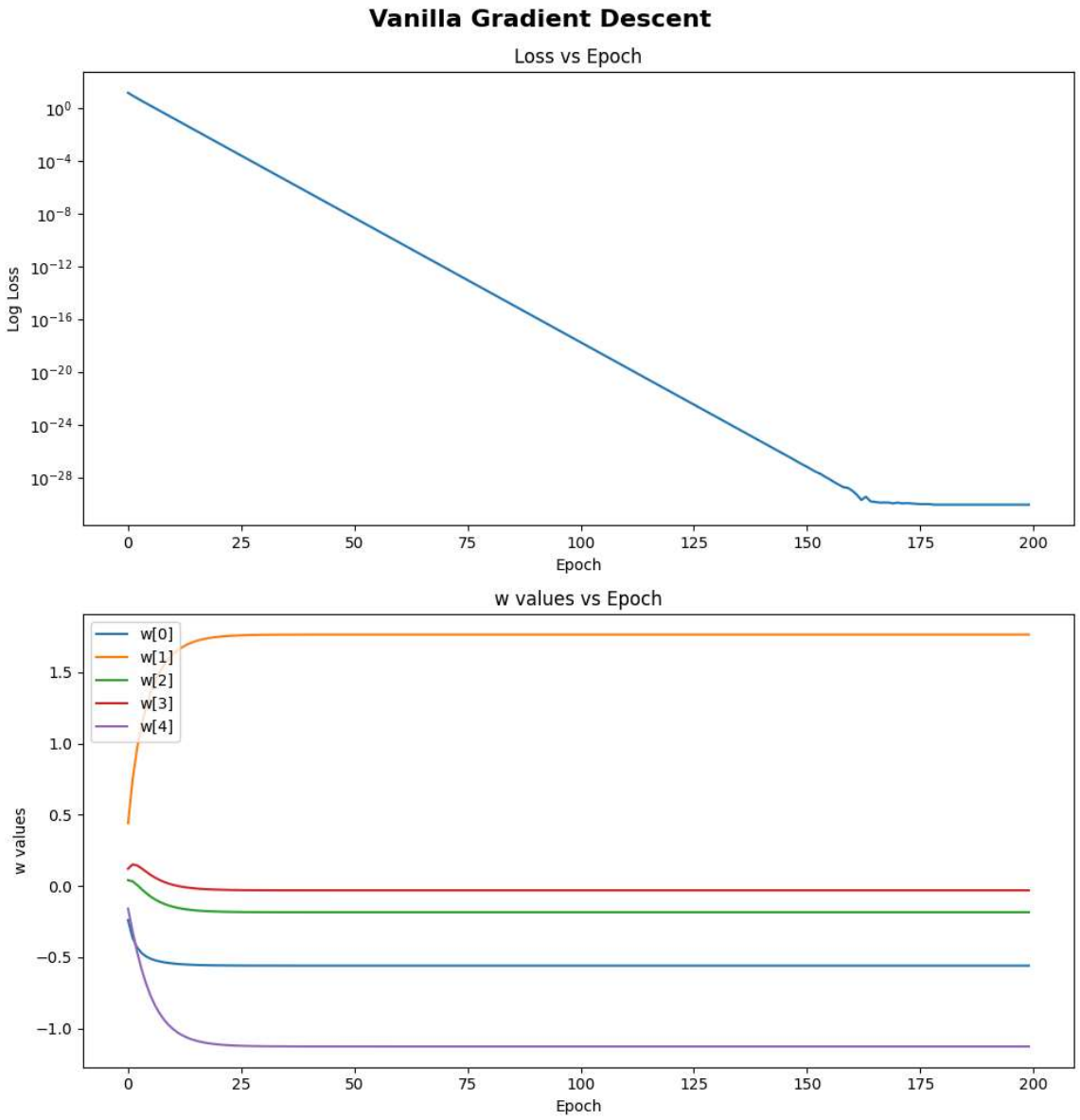
(b) (3 pts) In Python, directly minimize the objective L without any sparsity-inducing regularization/ constraint. Plot the value of log(L) vs. #steps throughout the training, and use another figure to plot how the value of each element in W is changing throughout the training. From your result, is W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

Code for Gradient Descent and Training:

```python
def mod_gradient_descent(
    X: np.array,
    W: np.array,
    y: np.array,
    lr: float,
    regularize=False,
    reg_strength=0,
    projected_gradient=False,
    hard_threshold=0,
    proximal_gradient=False,
    trimmed_L1=False,
):
    """Vectorized gradient descent

    Args:
        X (np.array): Design matrix
        W (np.array): Weights
        Y (np.array): Target values
        lr (float): Learning rate
        regularize (bool, optional): Regularize the loss. Defaults to False.
        reg_strength (int, optional): Regularization strength. Defaults to 0.
        projected_gradient (bool, optional): Projected gradient descent. Defaults to False.
        projected_gradient_strength (int, optional): Projection strength. Defaults to 0.
        proximal_gradient (bool, optional): Proximal gradient descent. Defaults to False.
        trimmed_L1 (bool, optional): Trimmed L1 regularization. Defaults to False.

    Returns:
        np.array: Updated weights
        float: Loss
    """
    grad = 2 * X.T.dot(X.dot(W) - y)

    if regularize:
        grad+=reg_strength*np.sign(W)
        W=W-lr*grad

    elif trimmed_L1:
        # find the lowest 3 values of the gradient
        lowest_3_idx = np.argsort(np.abs(grad))[:3]
        for idx in lowest_3_idx:
            grad[idx] += reg_strength * np.sign(W[idx])

        W = W - lr * grad
    else:
        W = W - lr * grad


    if projected_gradient:
        W[np.argsort(np.abs(W), axis=0)[:-hard_threshold]] = 0

    if proximal_gradient:
        threshold = reg_strength * lr

        for i in range(len(W)):
            if W[i] > threshold:
                W[i] -= threshold
            elif abs(W[i]) <= threshold:
                W[i] = 0
            else:
                W[i] += threshold

    loss = (X.dot(W) - y) ** 2
    return W, np.sum(loss)
```

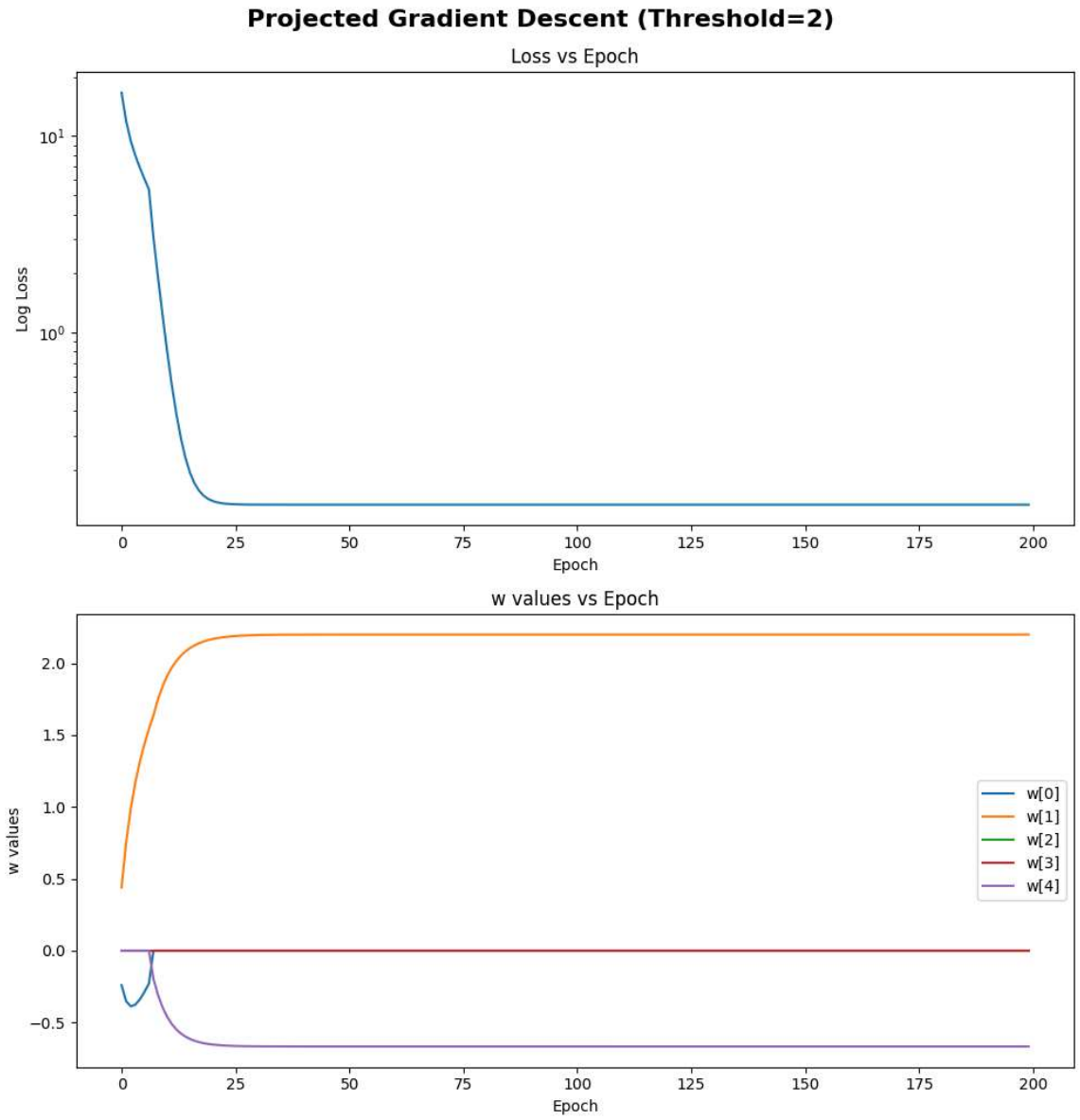
✓ 0.0s Python

```python
def train(
    regularize=False,
    reg_strength=0,
    projected_gradient=False,
    hard_threshold=0,
    proximal_gradient=False,
    plot=True,
    trimmed_l1=False,
):
    """
    This function will execute 200 epochs of training

    Args:
        regularize (bool, optional): Regularize the loss. Defaults to False.
        reg_strength (int, optional): Regularization strength. Defaults to 0.
        projected_gradient (bool, optional): Projected gradient descent. Defaults to False.
        hard_threshold (int, optional): Threshold for projected gradient descent. Defaults to 0.
        proximal_gradient (bool, optional): Proximal gradient descent. Defaults to False.
        plot (bool, optional): Plot the results. Defaults to True.

    Returns:
        list: Epochs
        list: Losses
        list: Weights
        figure: Figure object

    """
    X = np.array(
        [
            [-1.0, 2.0, 1.0, 1.0, -1.0],
            [-2.0, 1.0, -2.0, 0.0, 2.0],
            [1.0, 0.0, -2.0, -2.0, -1.0],
        ]
    )

    y = np.array([5.0, 1.0, 1.0])
    w = np.array([0.0, 0.0, 0.0, 0.0, 0.0])
    losses = []
    epoch = []
    lr = 0.02
    w_values = []
    for i in range(200):
        w, loss = mod_gradient_descent(
            X,
            w,
            y,
            lr,
            regularize,
            reg_strength,
            projected_gradient,
            hard_threshold,
            proximal_gradient,
            trimmed_l1,
        )
        w = w


        losses.append(loss)

        epoch.append(i)
        w_values.append(w)

        # print(f"The loss at epoch {i} is {loss}")
    w_values = np.array(w_values)
    fig = None
    if plot:
        # Create subplots
        fig, axs = plt.subplots(2, 1, figsize=(10, 10))

        # Plot loss vs epoch
        axs[0].plot(epoch, losses)
        axs[0].set_yscale("log")
        axs[0].set_xlabel("Epoch")
        axs[0].set_ylabel("Log Loss")
        axs[0].set_title("Loss vs Epoch")

        # Plot w values vs epoch
        for j in range(w_values.shape[1]):
            axs[1].plot(epoch, w_values[:, j], label=f"w[{j}]")
        axs[1].set_xlabel("Epoch")
        axs[1].set_ylabel("w values")
        axs[1].set_title("w values vs Epoch")
        axs[1].legend()

        plt.tight_layout()

    return epoch, losses, w_values, fig
```



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 spase 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 ≤ 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 ||WI||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. From your result, is W converging to an optimal solution? Is W converging to a sparse solution?

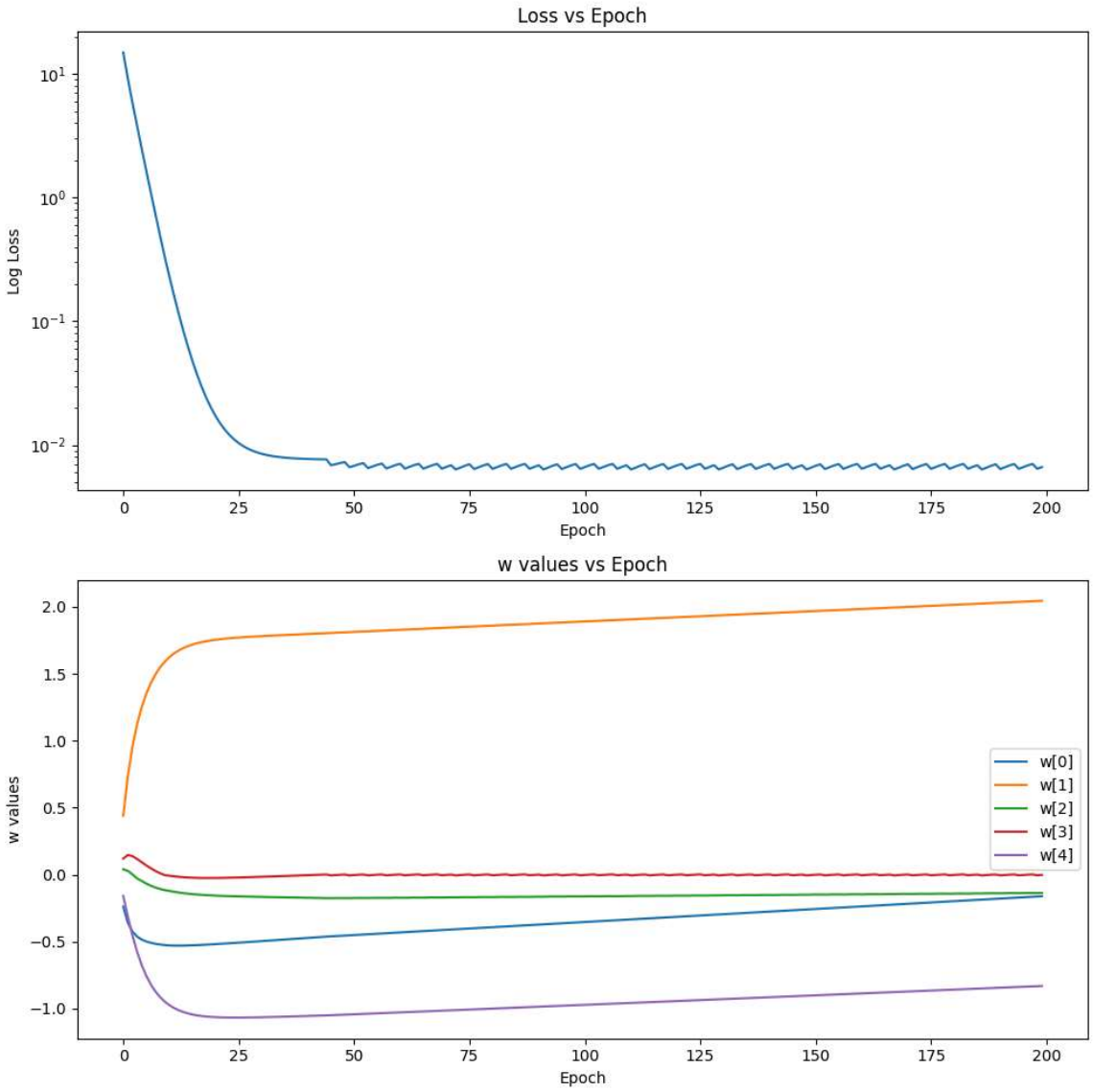


> Here we can see that W is converging on a sparse solution (only two of the W values are non-zero). This sparsification is at the expense of significant loss increase compared to the base vanilla gradient descent.

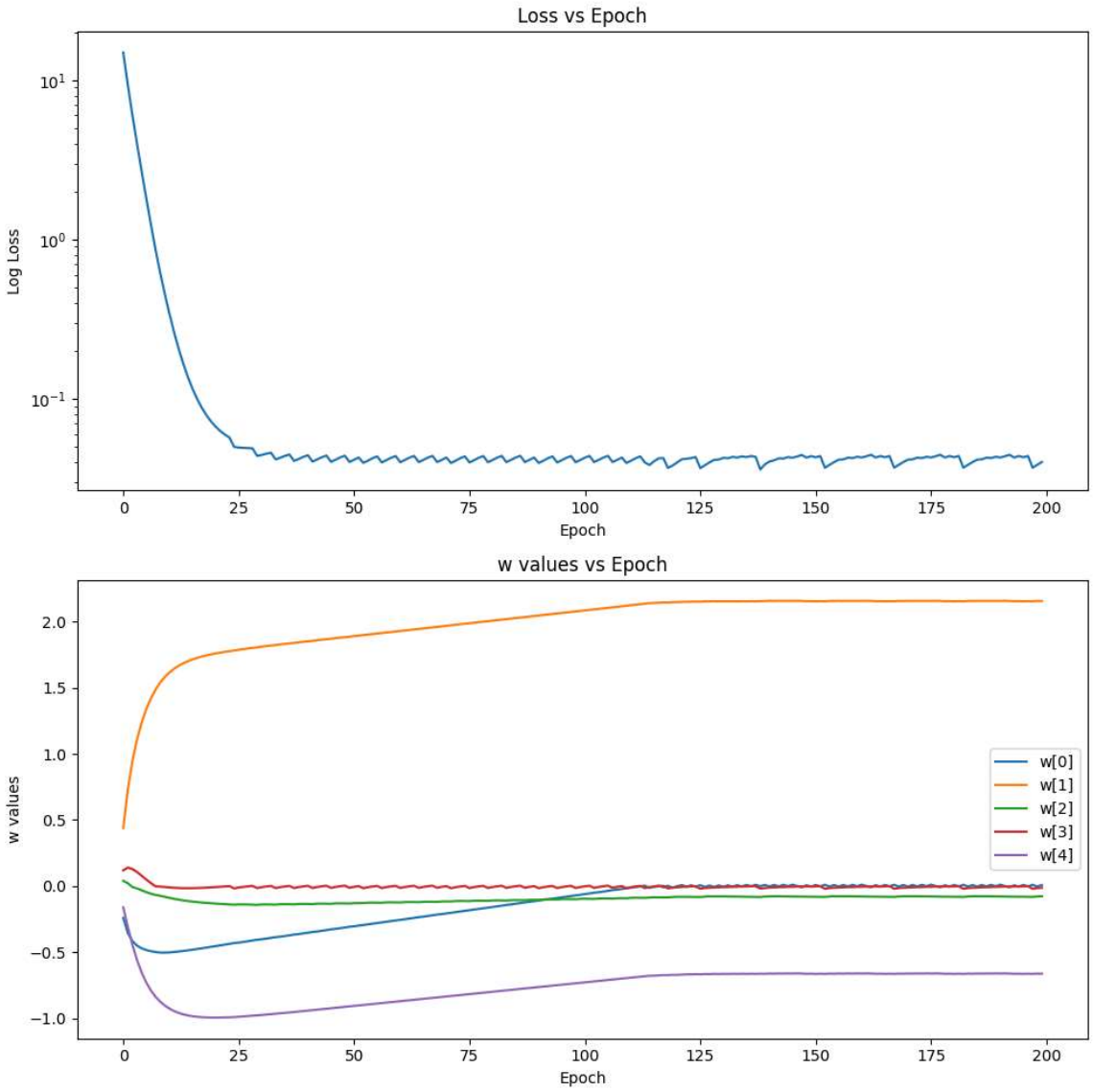
(d) (5 pts) In this problem we apply ℓ1 regularization to induce the sparse solution. The minimization objective therefore changes to L + λ||W||1. Please use full-batch gradient descent to minimize this objective, with λ = {0.2, 0.5, 1.0, 2.0} respectively. For each case, plot the value of log(L) throughout 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.

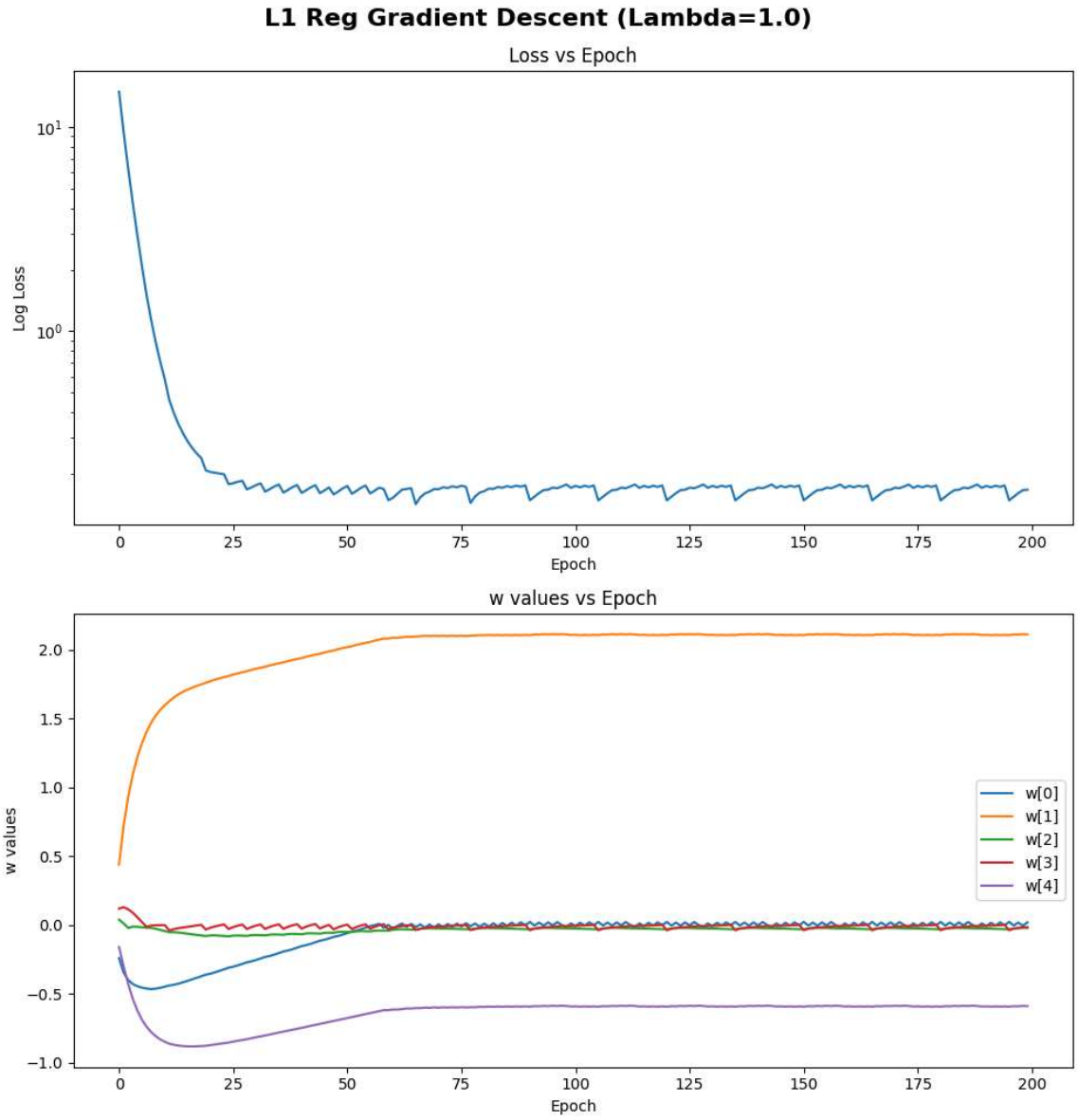
> Below are the individual plots for each lambda value for L1 regularization:

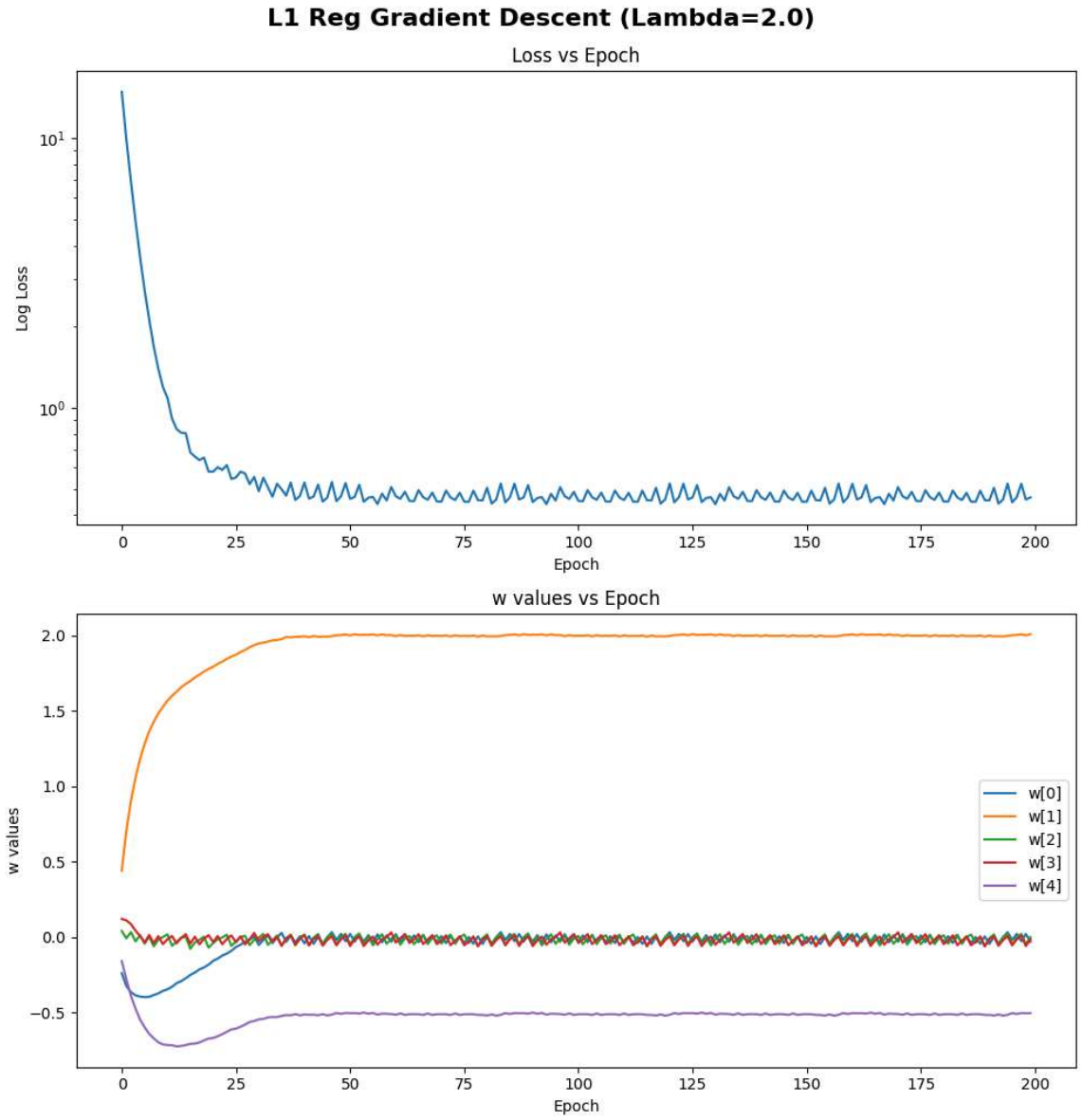
L1 Reg Gradient Descent (Lambda=0.2)



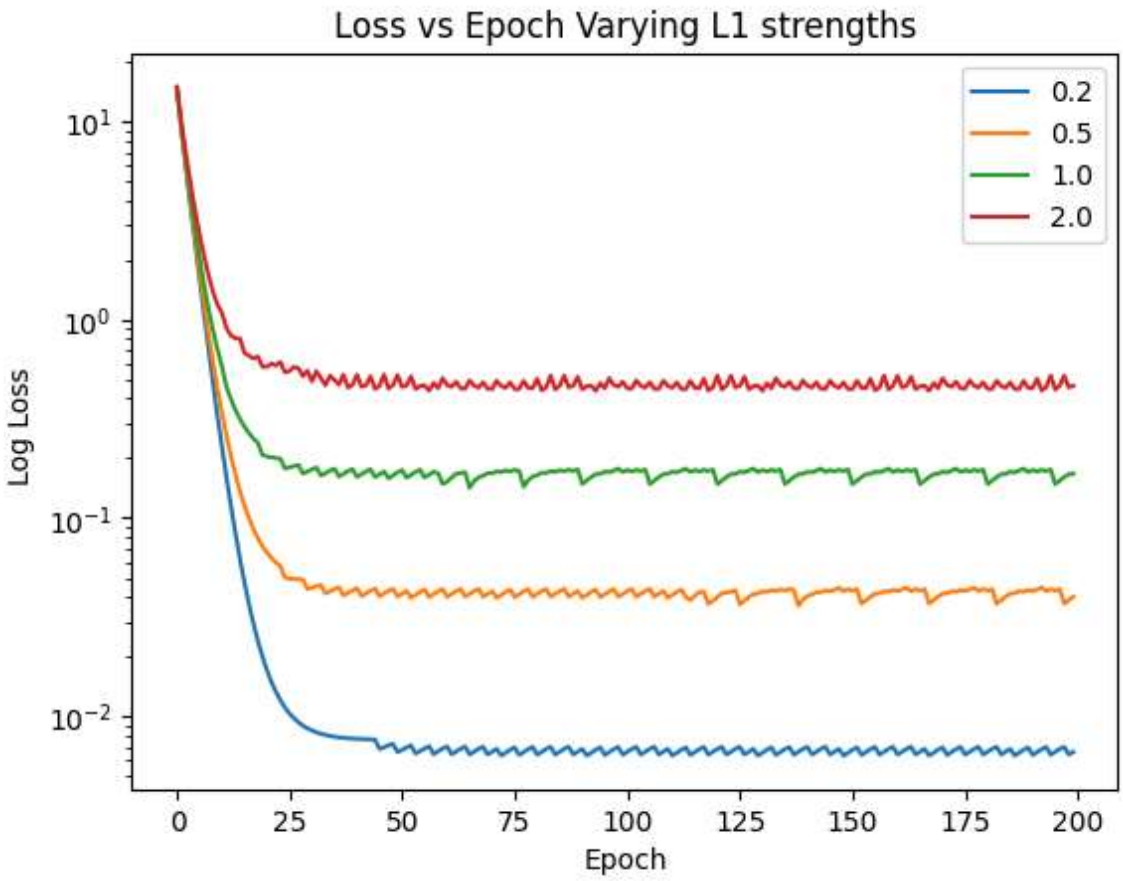
L1 Reg Gradient Descent (Lambda=0.5)




L1 Reg Gradient Descent (Lambda=1.0)
Loss vs Epoch
Log Loss
Epoch
w values vs Epoch
w values
Epoch
w[0]
w[1]
w[2]
w[3]
w[4]



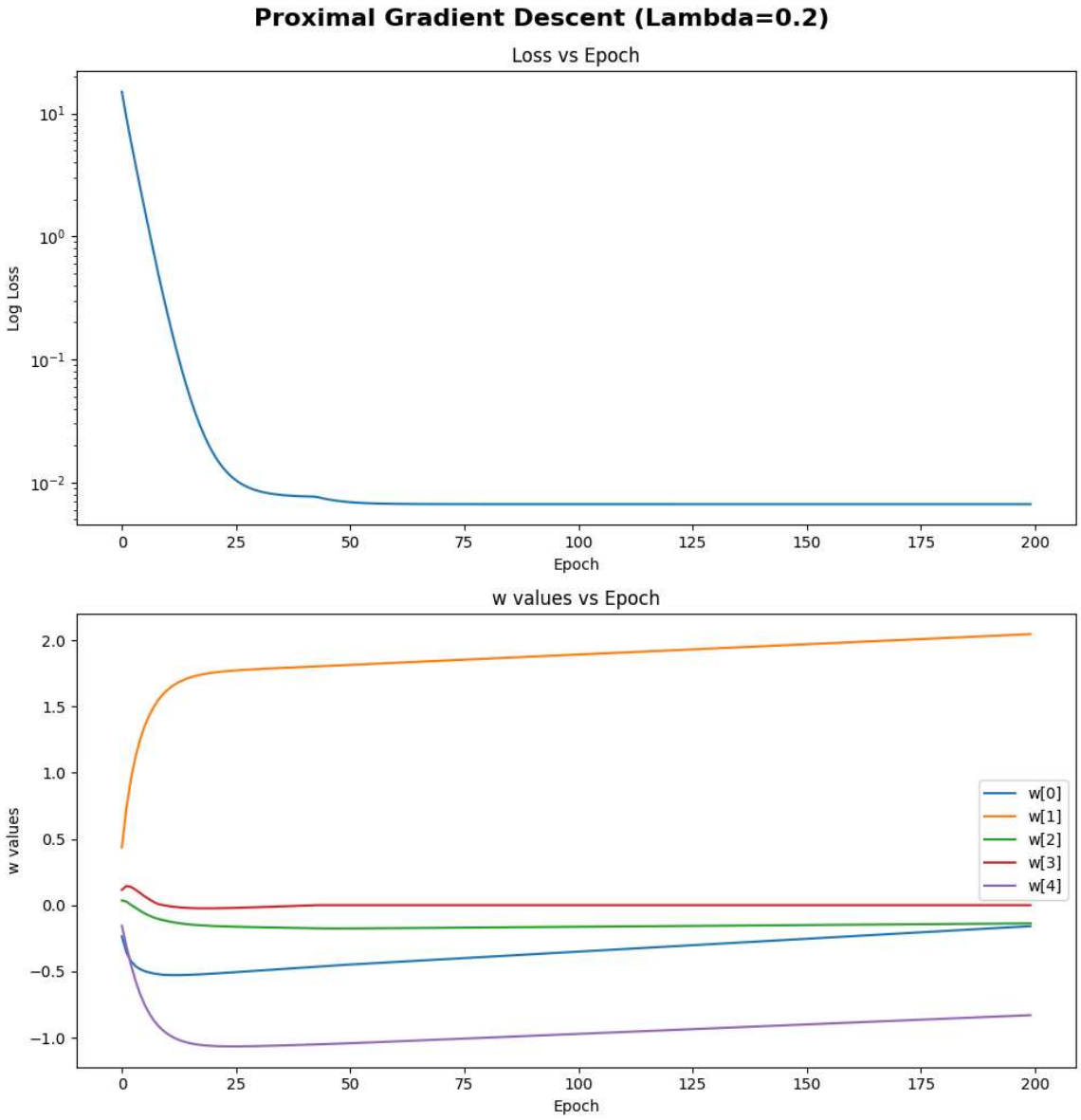
> We also have all the L1 regularization plots together to see the differences:

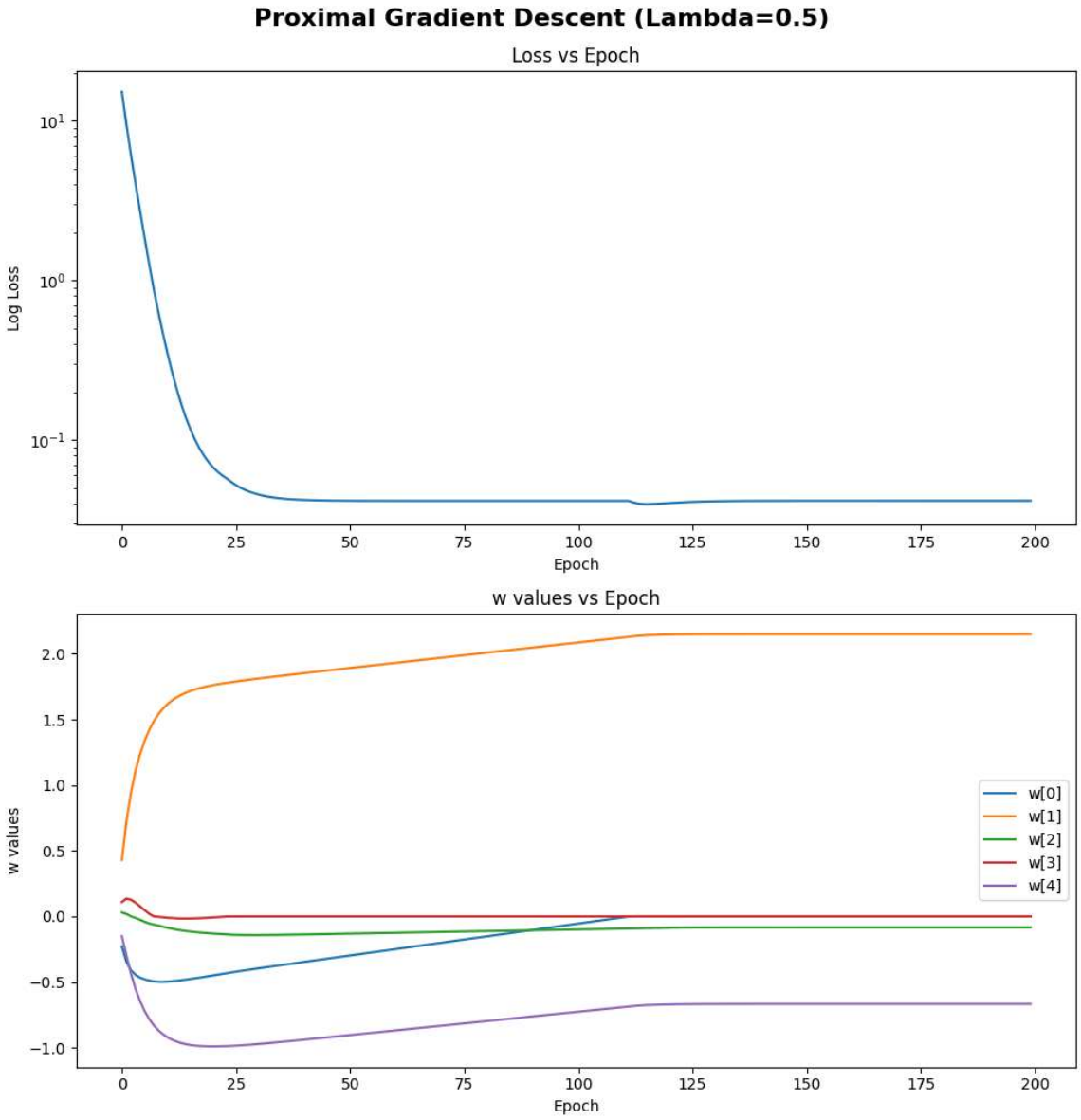


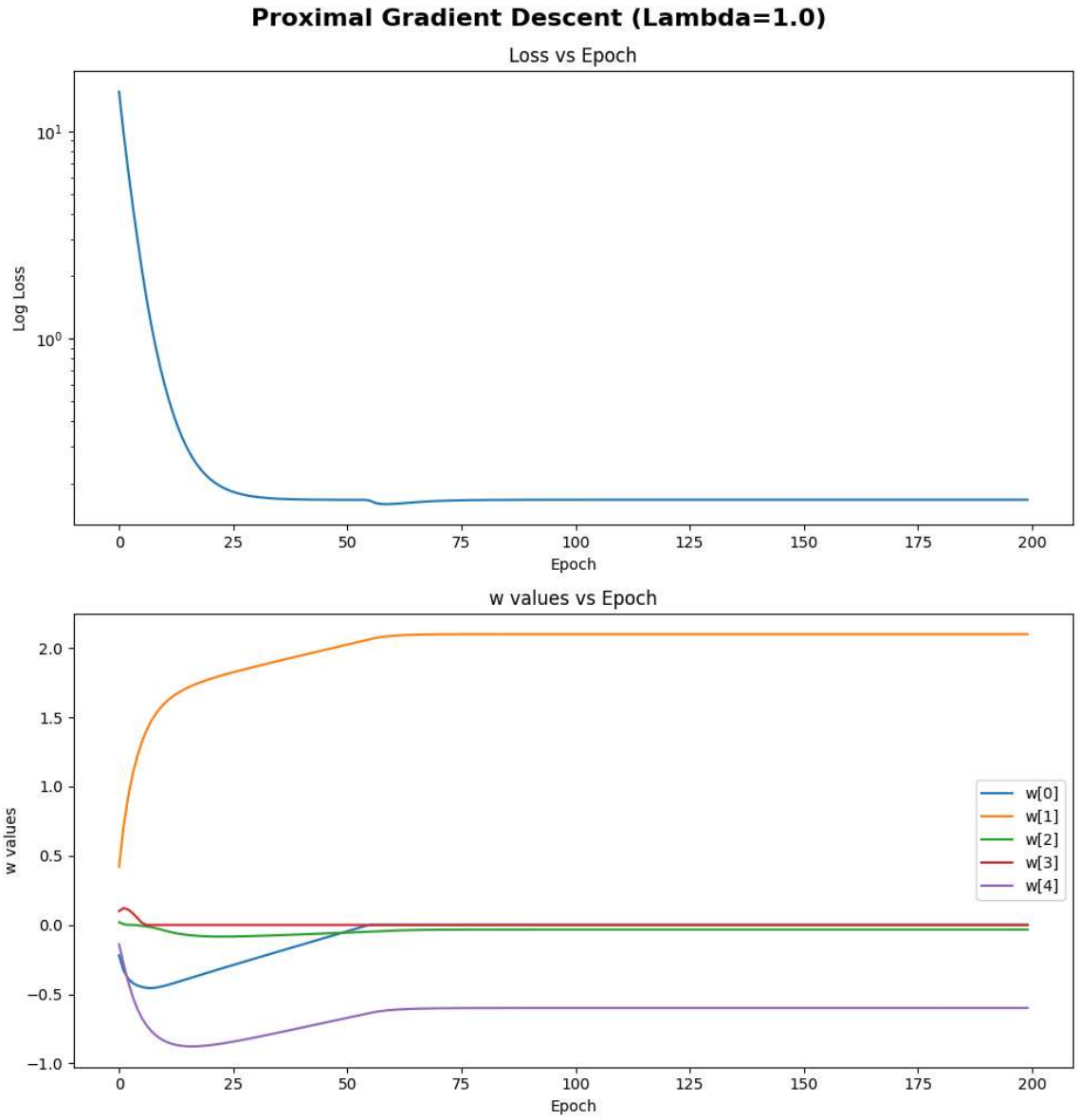
> What we see with increasing L1 strength is that the model weights are increasingly pushed towards zero leading to a sparser solution but this comes at the cost of higher loss values. Our model becomes more efficient (sparse) but at the expense of accuracy.

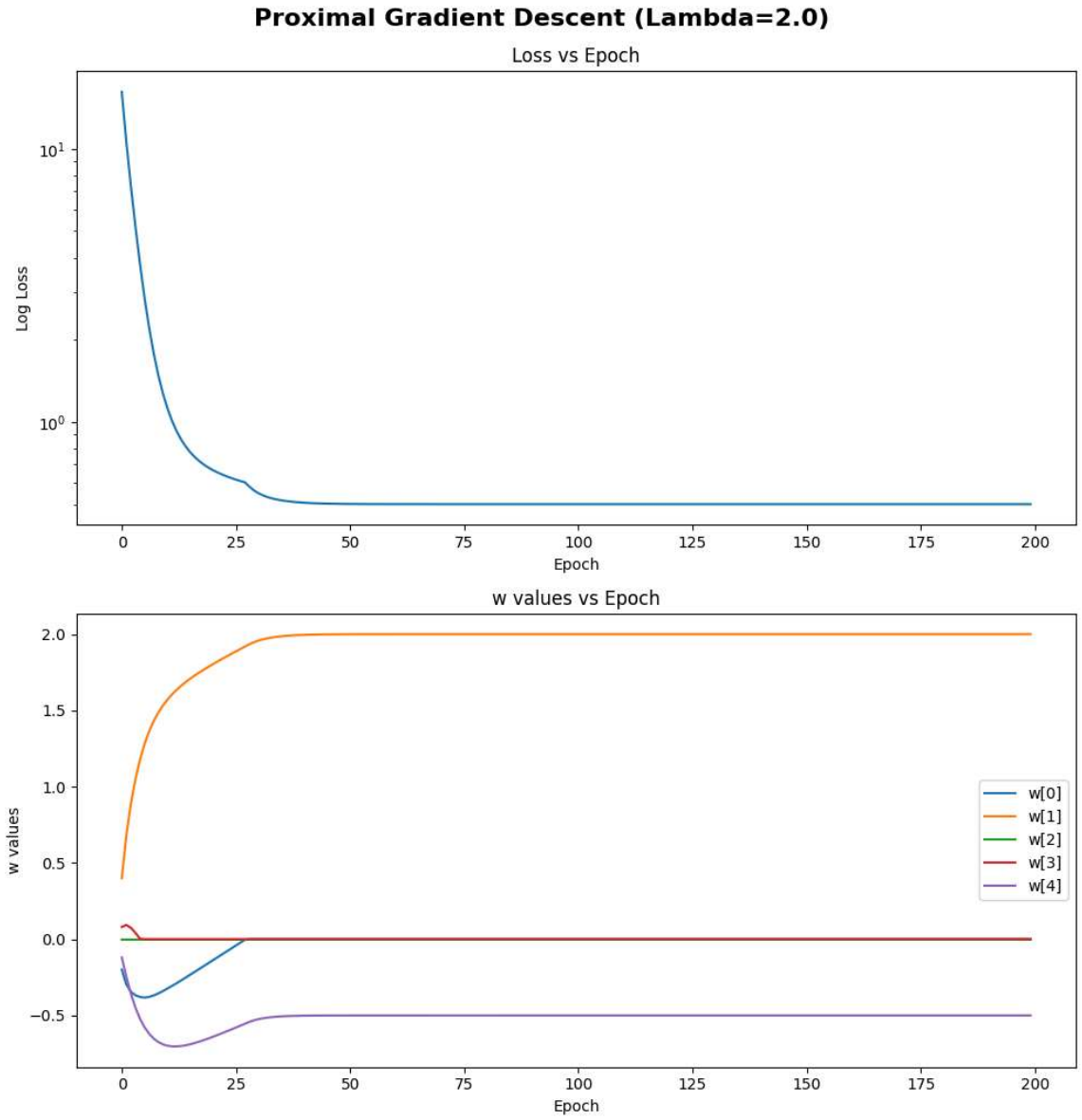
(e) (6 pts) Here we optimize the same objective as in (d), this time using proximal gradient update. Recall that the proximal operator of the ℓ1 regularizer is the soft thresholding function. Set the threshold in the soft thresholding function to {0.004, 0.01, 0.02, 0.04} respectively. Plot the value 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 + λ||W||1 using gradient descent with learning rate μ should correspond to proximal gradient update with threshold μλ).

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

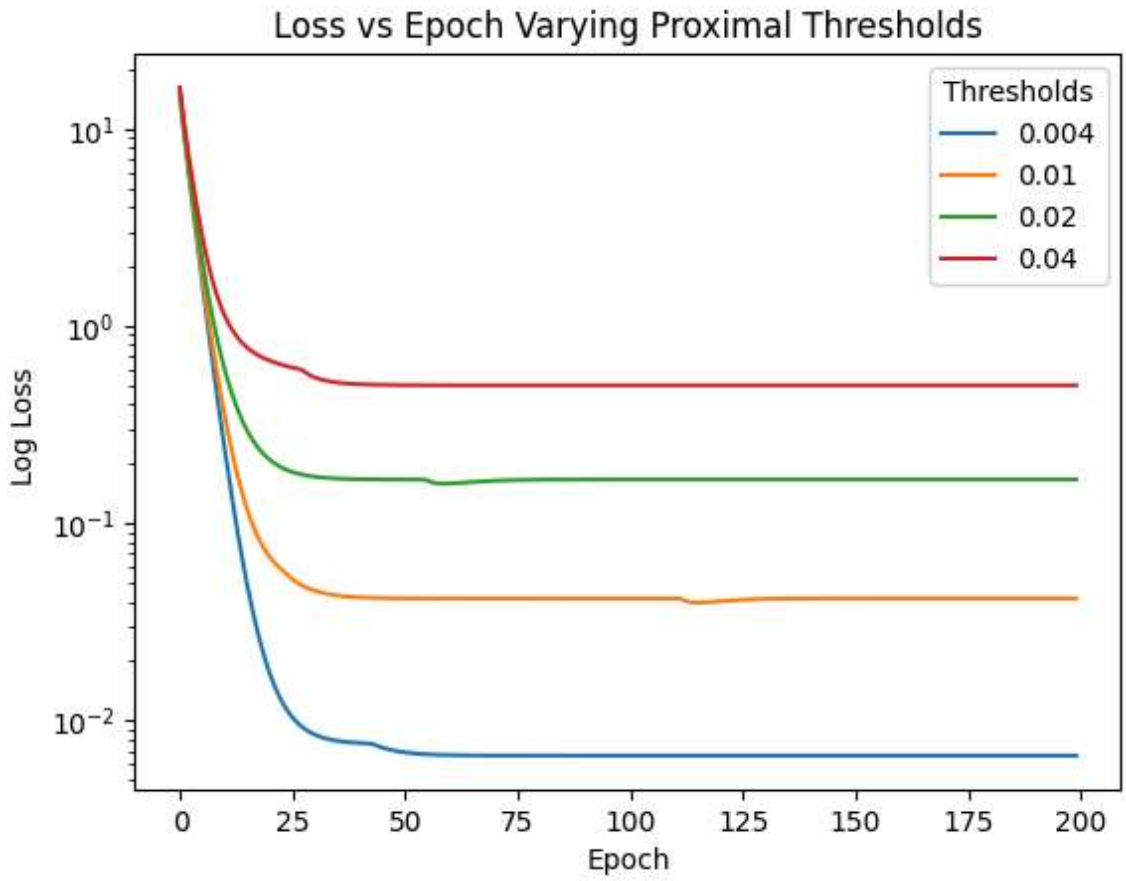

Proximal Gradient Descent (Lambda=0.2)
Loss vs Epoch
Log Loss
Epoch
w values vs Epoch
w values
Epoch
w[0]
w[1]
w[2]
w[3]
w[4]


Proximal Gradient Descent (Lambda=0.5)
Loss vs Epoch
Log Loss
Epoch
w values vs Epoch
w values
Epoch
w[0]
w[1]
w[2]
w[3]
w[4]


Proximal Gradient Descent (Lambda=1.0)
Loss vs Epoch
Log Loss
Epoch
w values vs Epoch
w values
Epoch
w[0]
w[1]
w[2]
w[3]
w[4]



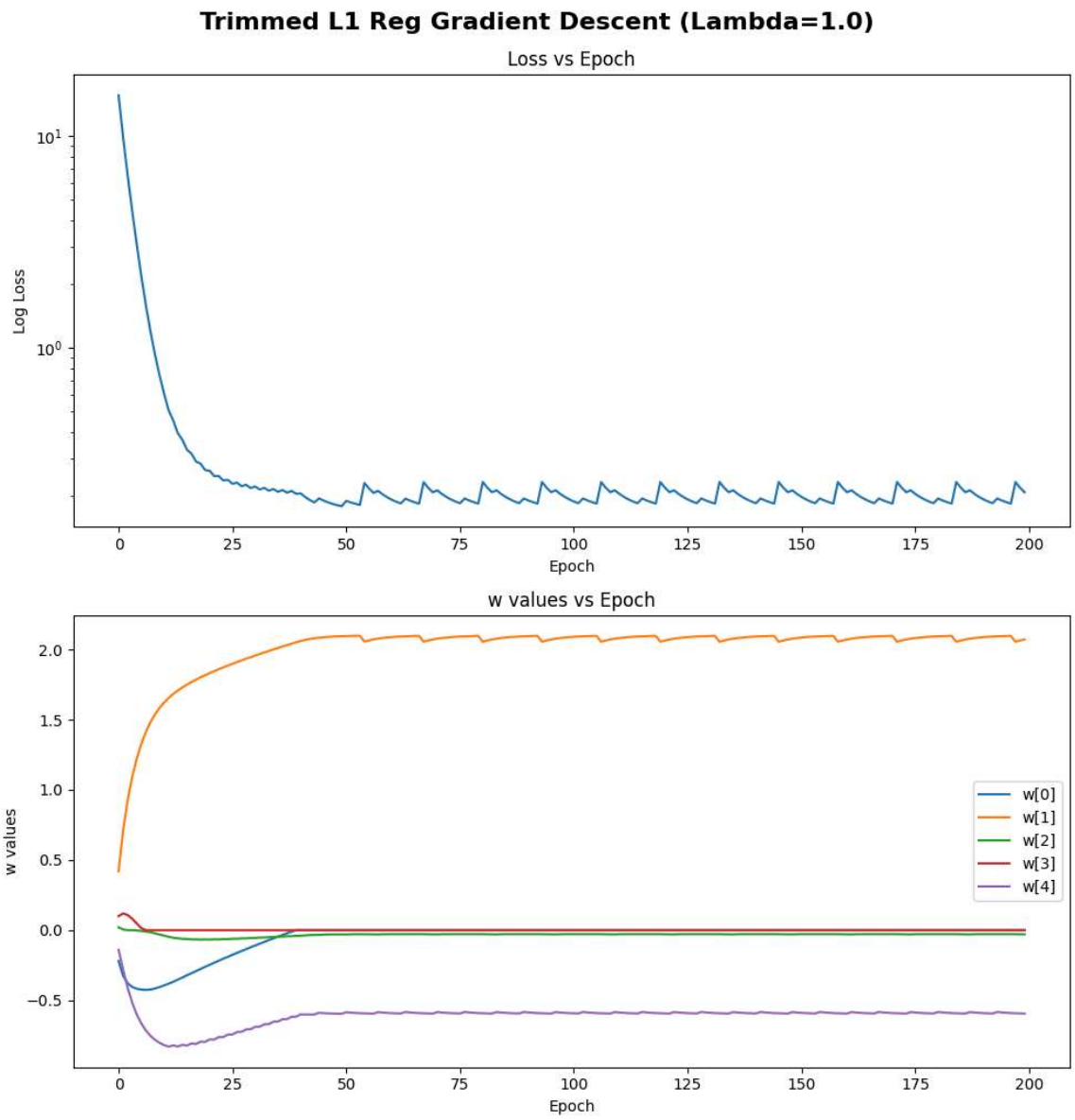
> we also have each loss for proximal gradient update plotted together:

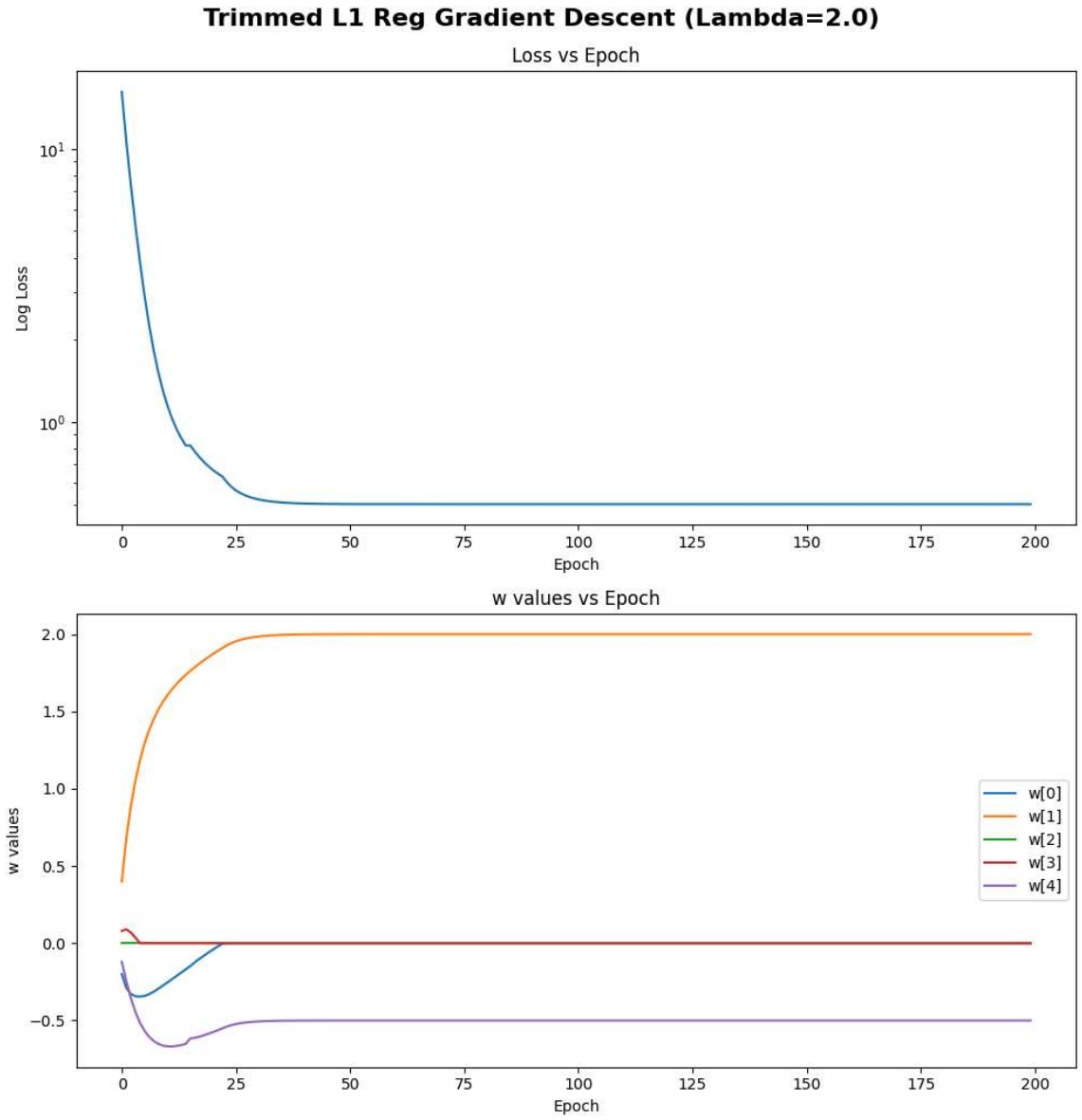


> We can see that the proximal term succeeds in allowing for smoother convergence toward the overal objective. We see the same tradeoff for L1 regularization in that we can incentivize sparser solutions at the expense of loss.

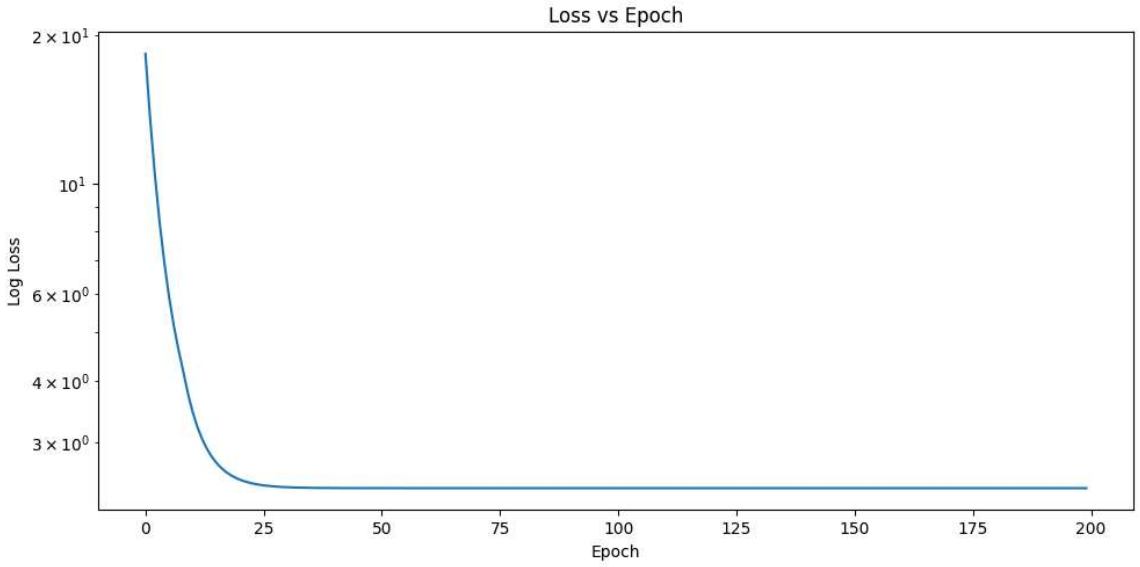
(f) (6 pts) Trimmed ℓ1 (Tℓ1) regularizer is proposed to solve the "bias" problem of ℓ1. For simplicity you may implement the Tℓ1 regularizer as applying a ℓ1 regularization with strength λ on the 3 elements of W with the smallest absolute value, with no penalty on other elements. Minimize L+λTℓ1(W) using 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 ℓ1 and the ℓ1. Also compare the behavior of the early steps (e.g. first 20) between the Trimmed ℓ1 and the iterative pruning.

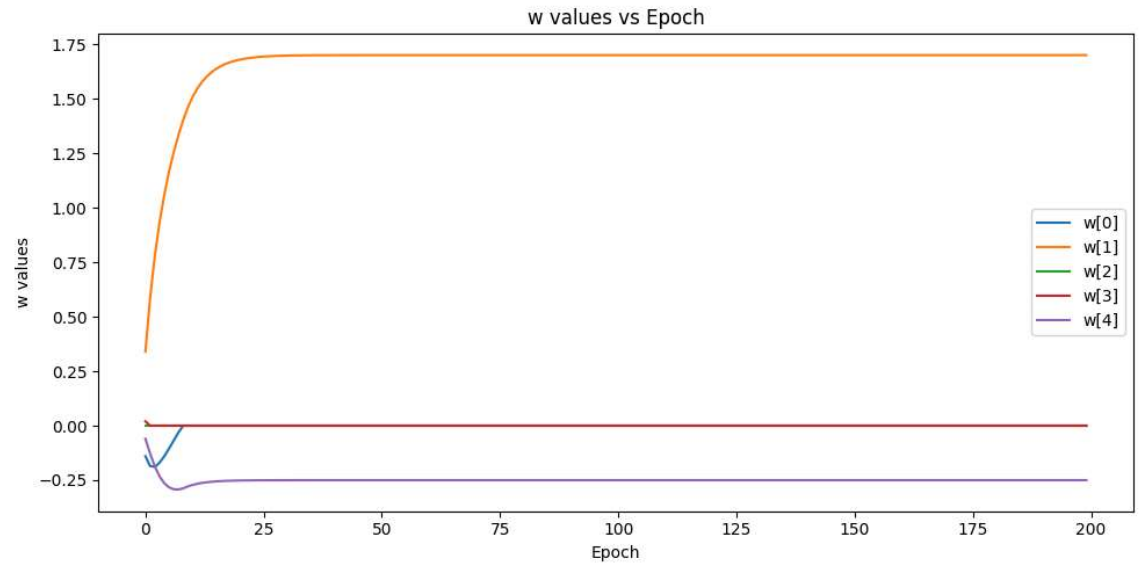
> Here we look at loss and weight curves for multiple lambda values with trimming:

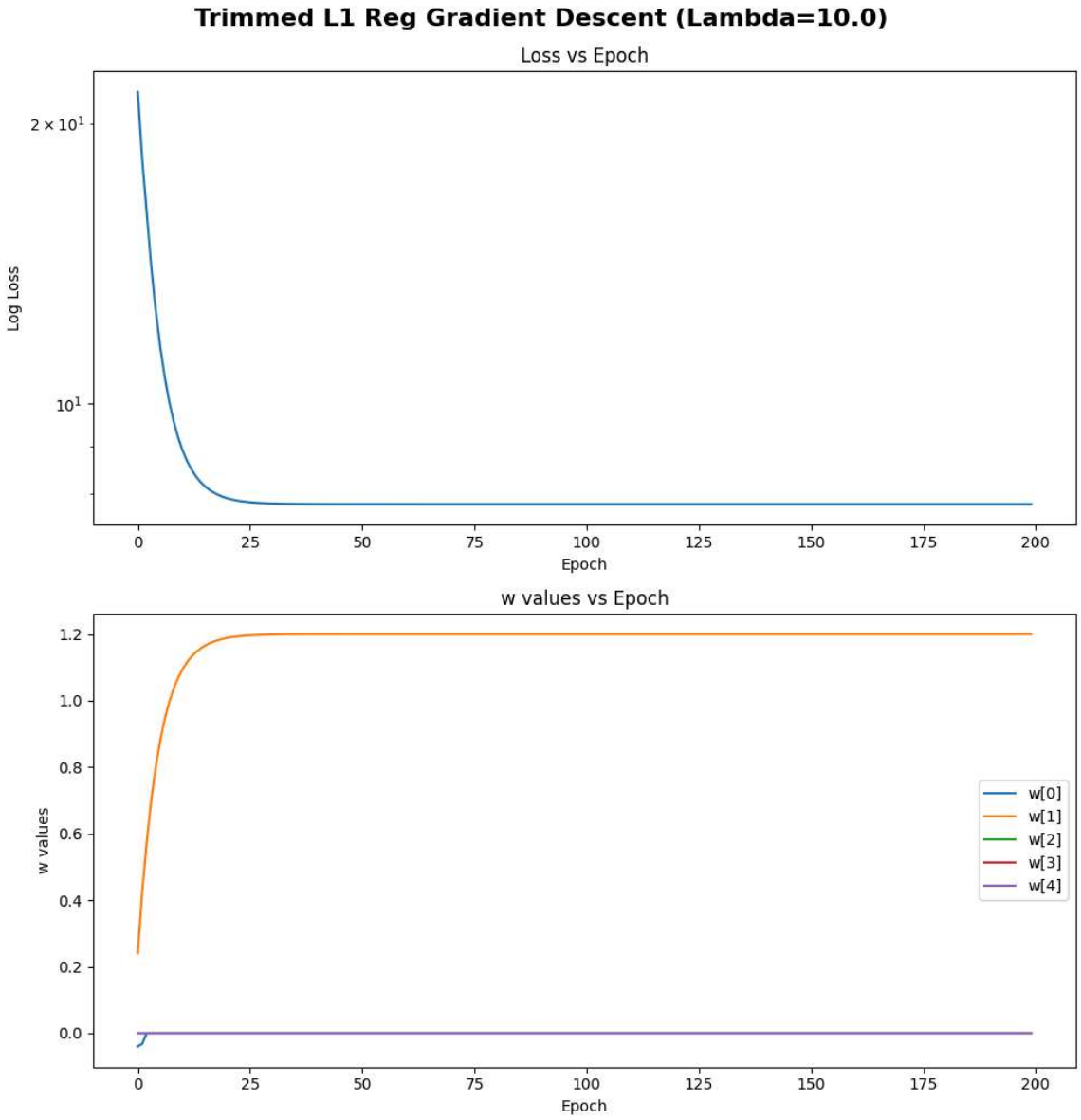

Trimmed L1 Reg Gradient Descent (Lambda=1.0)
Loss vs Epoch
Log Loss
Epoch
w values vs Epoch
w values
Epoch
w[0]
w[1]
w[2]
w[3]
w[4]


Trimmed L1 Reg Gradient Descent (Lambda=2.0)
Loss vs Epoch
Log Loss
Epoch
w values vs Epoch
w values
Epoch
w[0]
w[1]
w[2]
w[3]
w[4]

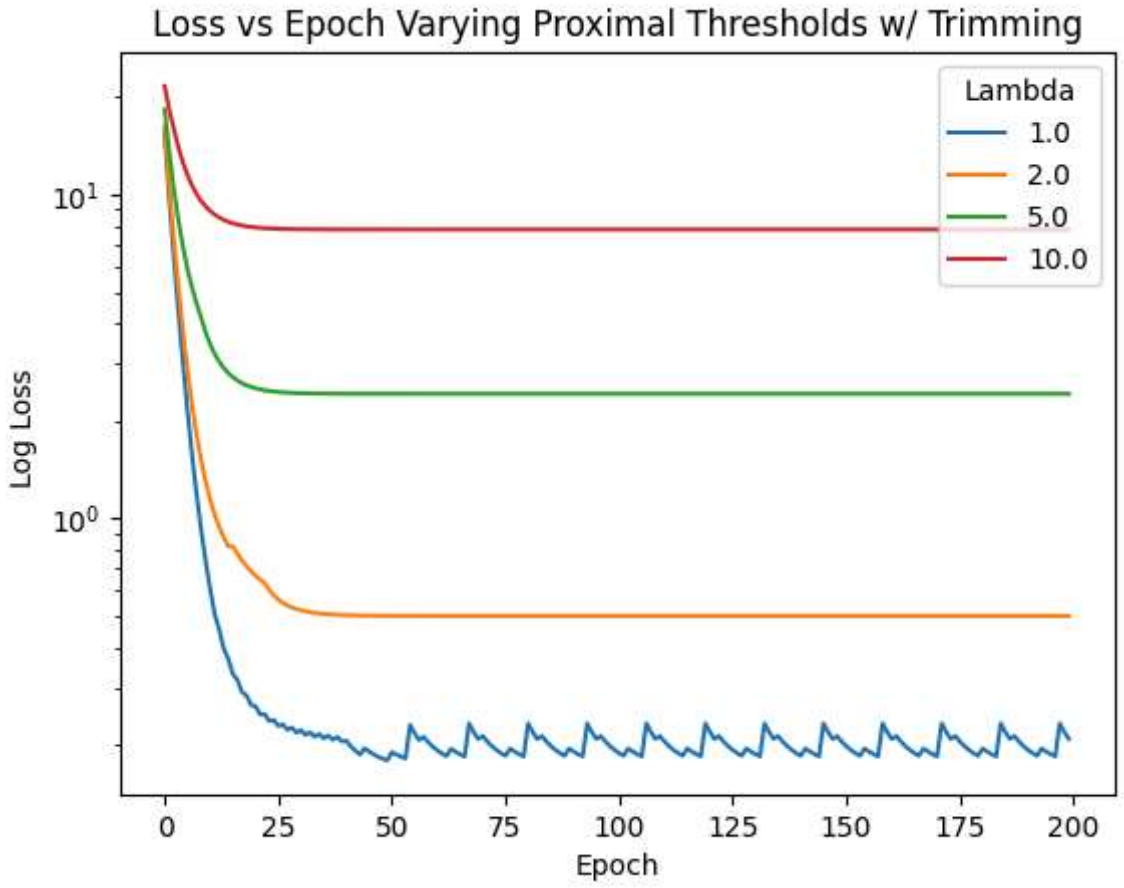
Trimmed L1 Reg Gradient Descent (Lambda=5.0)







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

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

```
<ipython-input-10-7a50f8b27ae8>: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 determines the pruning threshold in each DNN layer by the 'q-th percentile' value in the absolute value of layer's weight element. Use the next block to call your implemented pruning by percentage. Try pruning percentage q = 0.3, 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:

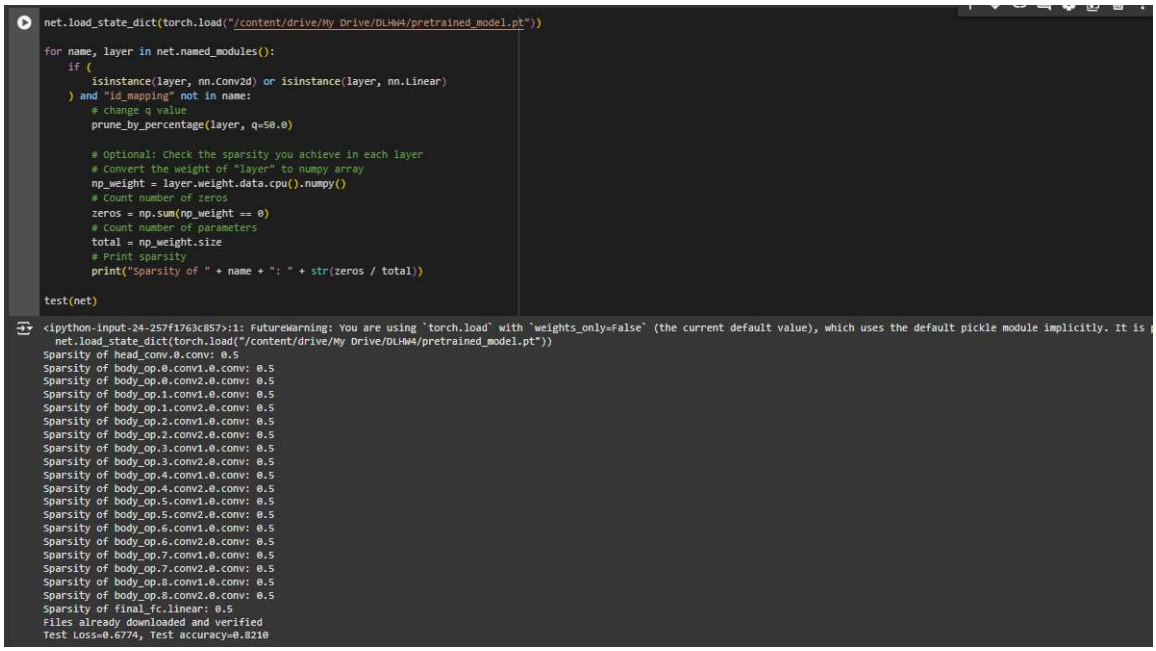
```
def prune_by_percentage(layer, q=70.0):
    """
    Pruning the weight paramters by threshold.
    :param q: pruning percentile. 'q' percent of the least
    significant weight parameters will be pruned.
    """
    # Convert the weight of "layer" to numpy array
    weights = layer.weight.data.cpu().numpy()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

```
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-23-2fe05e8c2c61>: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_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=2.4417, Test accuracy=0.4204
```

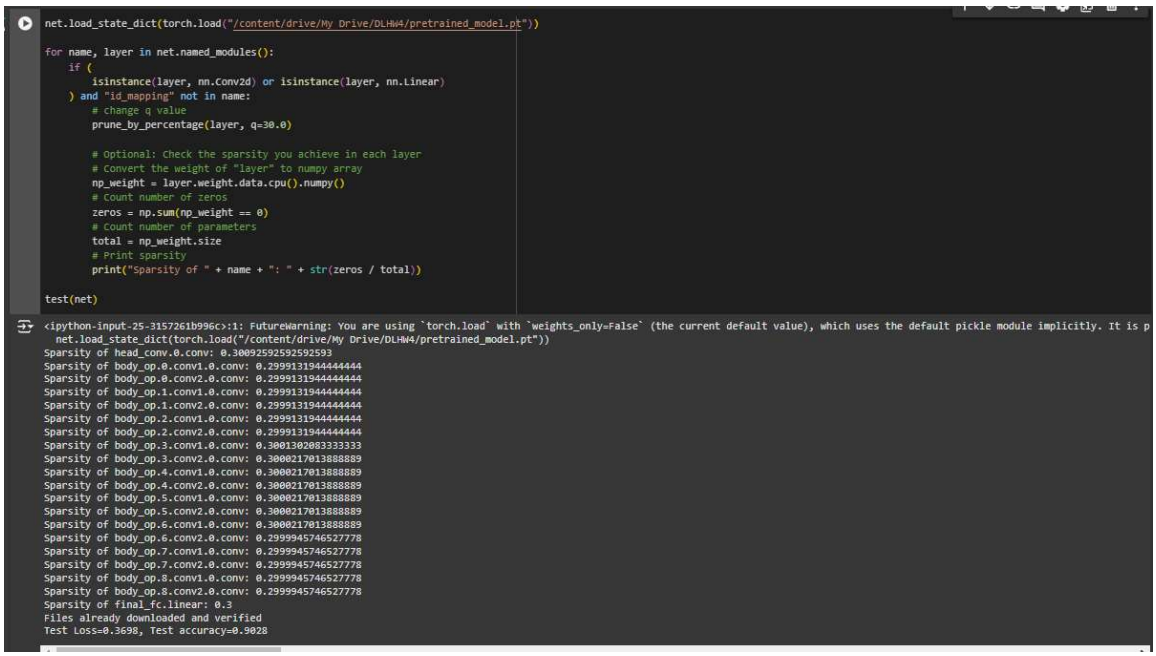
> 0.5 Pruning results:



```
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=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 p
  net.load_state_dict(torch.load("/content/drive/My Drive/DLHW4/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/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=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 p
  net.load_state_dict(torch.load("/content/drive/My Drive/DLHW4/pretrained_model.pt"))
Sparsity of head_conv.0.conv: 0.300925925925925 93
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: 0.3001302083333333
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: 0.3000217013888889
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=0.9028
```

(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:

```python
def finetune_after_prune(net, trainloader, criterion, optimizer, prune=True):
    """
    Finetune the pruned model for a single epoch
    Make sure pruned weights are kept as zero
    """
    # Build a dictionary for the nonzero weights
    weight_mask = {}
    for name, layer in net.named_modules():
        if (
            isinstance(layer, nn.Conv2d) or isinstance(layer, nn.Linear)
        ) and "id_mapping" not in name:
            # Your code here: generate a mask in GPU torch tensor to have 1 for nonzero element and 0 for zero element
            weight_mask[name] = (layer.weight.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=%.4f\tacc=%.4f\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:

```python
# 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)
```

```
<ipython-input-22-2b35f04f0af1>: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: 0.6990740740740741
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=0.3337, 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 × 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) * 7
                prune_by_percentage(layer, q=q)
    if epoch < 9:
        finetune_after_prune(net, trainloader, criterion, optimizer, prune=False)
    else:
        finetune_after_prune(net, trainloader, criterion, optimizer)

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

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

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

```
<ipython-input-28-928cbabe87b8>: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_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=0.3387, 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:

```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
            weight = layer.weight.data.cpu().numpy()
            # Flatten the weight and append to flattened_weights
            flattened_weights.append(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)
    # 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:

```python
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-e491f61d38f0>: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: 0.24537037037037038
Sparsity of body_op.0.conv1.0.conv: 0.5503472222222222
Sparsity of body_op.0.conv2.0.conv: 0.5277777777777778
Sparsity of body_op.1.conv1.0.conv: 0.5199652777777778
Sparsity of body_op.1.conv2.0.conv: 0.5533854166666666
Sparsity of body_op.2.conv1.0.conv: 0.5164930555555556
Sparsity of body_op.2.conv2.0.conv: 0.5659722222222222
Sparsity of body_op.3.conv1.0.conv: 0.5260416666666666
Sparsity of body_op.3.conv2.0.conv: 0.5831163194444444
Sparsity of body_op.4.conv1.0.conv: 0.6156684027777778
Sparsity of body_op.4.conv2.0.conv: 0.6773003472222222
Sparsity of body_op.5.conv1.0.conv: 0.6115451388888888
Sparsity of body_op.5.conv2.0.conv: 0.7035590277777778
Sparsity of body_op.6.conv1.0.conv: 0.6151801215277778
Sparsity of body_op.6.conv2.0.conv: 0.6510687934027778
Sparsity of body_op.7.conv1.0.conv: 0.6620008680555556
Sparsity of body_op.7.conv2.0.conv: 0.7189670138888888
Sparsity of body_op.8.conv1.0.conv: 0.7479112413194444
Sparsity of body_op.8.conv2.0.conv: 0.9371744791666666
Sparsity of final_fc.linear: 0.1203125
Total sparsity of: 0.6999992546657922
Files already downloaded and verified
Test Loss=0.3158, Test accuracy=0.8987
```

> 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 is 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.)

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

            # Lab3 (a), Your code here:
            if symmetric == False:
                # Compute alpha (scale) for dynamic scaling
                alpha = 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
Nbits = 6  # Change this value to finish (b) and (c)

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

```
<ipython-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
```

> Nbits = 5

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

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

```
<ipython-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
```

> Nbits = 4

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

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

```
<ipython-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%|          | 170M/170M [00:04<00:00, 42.5MB/s]
Extracting ./data/cifar-10-python.tar.gz to ./data
Test Loss=0.3861, Test accuracy=0.8972
```

> Nbits = 3

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

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

```
<ipython-input-9-707521ce1cad>: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
```

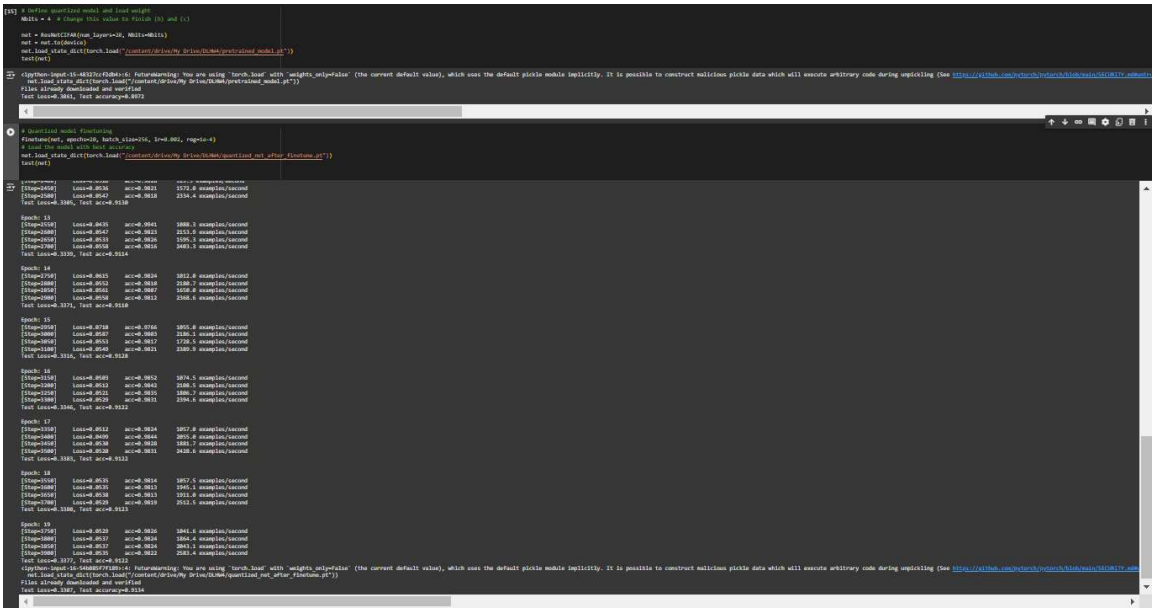
> Nbits = 2

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

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

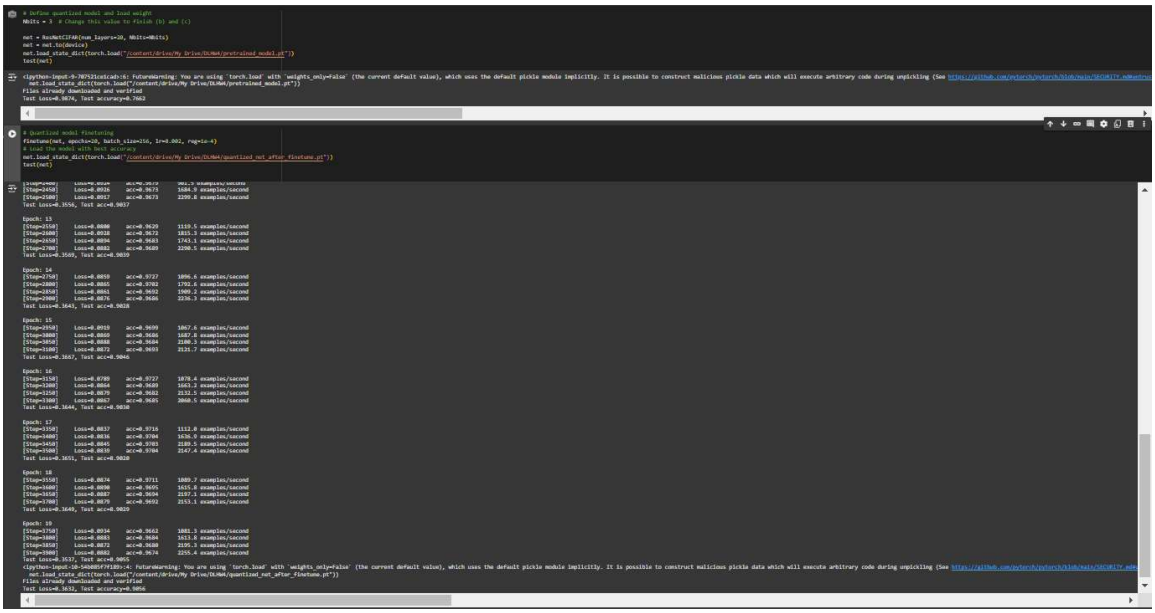
```
<ipython-input-10-cf41695ac89d>:6: FutureWarning: You are using `torch.load` with `weights_only=False` (the current default value), which uses the default
  net.load_state_dict(torch.load("/content/drive/My Drive/DLHW4/pretrained_model.pt"))
Files already downloaded and verified
Test Loss=9.5441,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

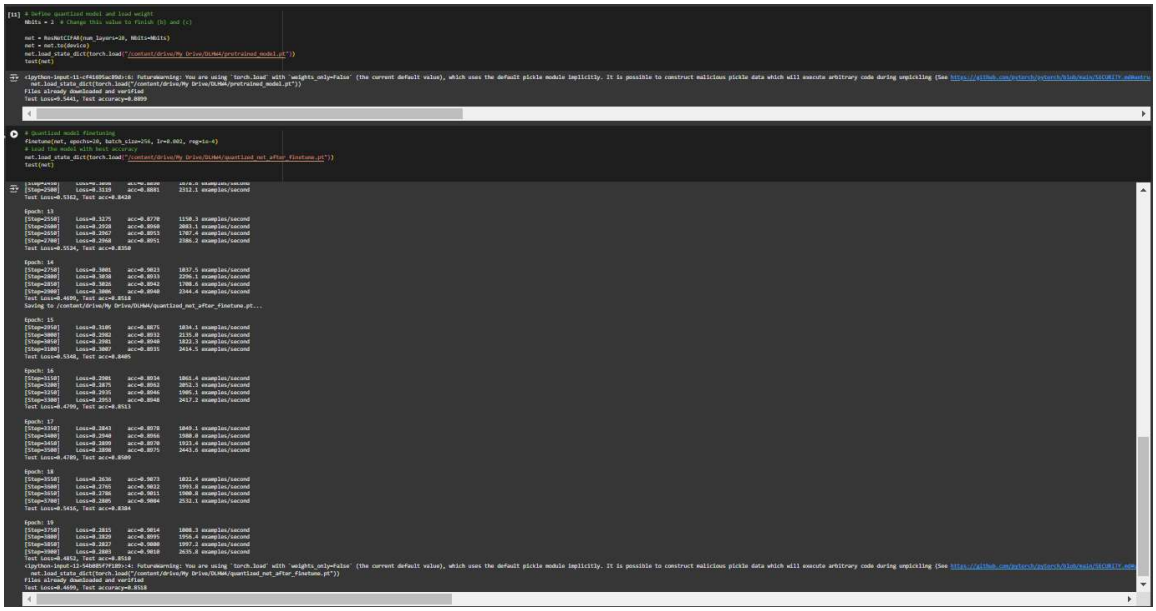
> Nbits = 4 (test acc = 0.9134)



> Nbits = 3 (test acc = 0.9056)



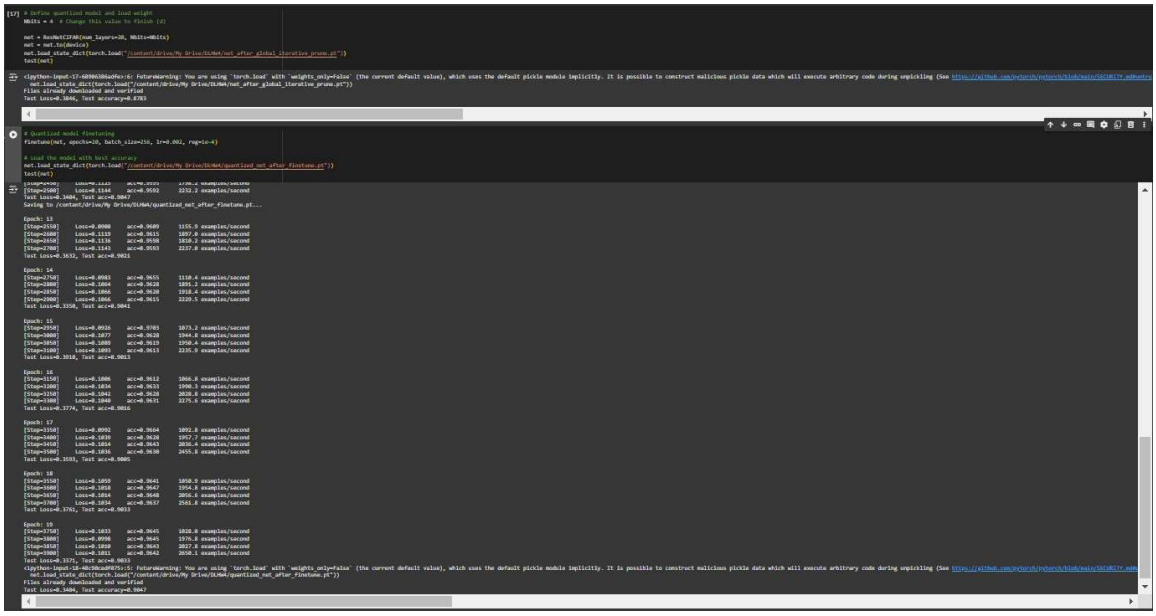
> Nbits = 2 (test acc = 0.8518)



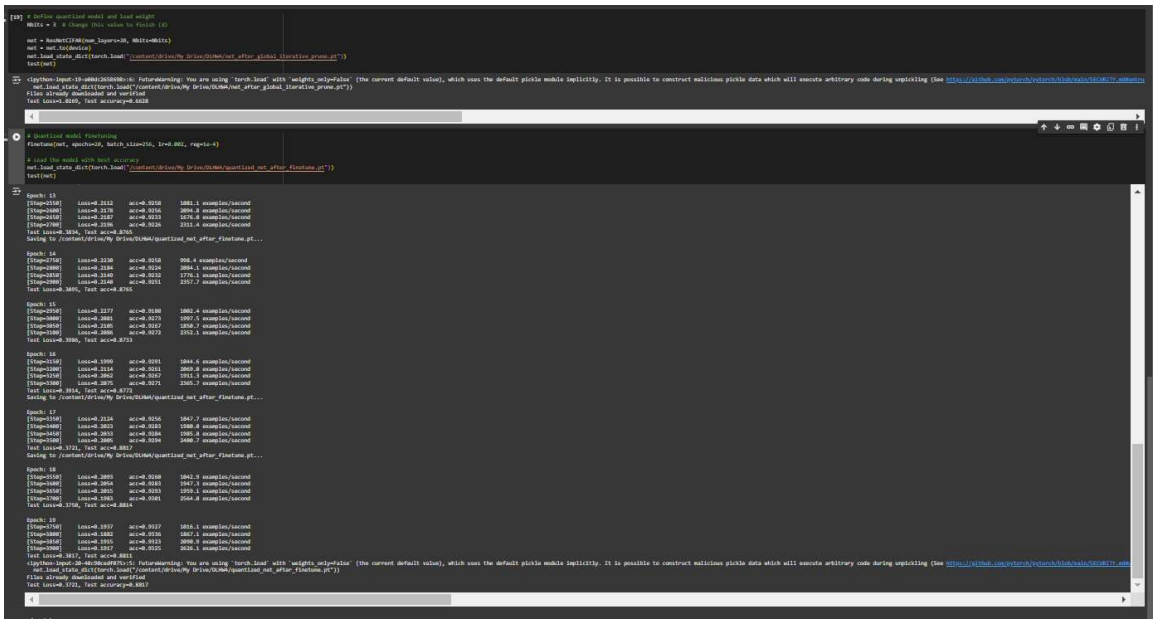
> We can see that for lower precision (lower Nbit values) we get lower accuracy. This is expected as we are reducing the number of bits used to represent the weights. Finetuning is effective in that it allows us to recover some of the lost accuracy from quantization. In all cases the finetuning performance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

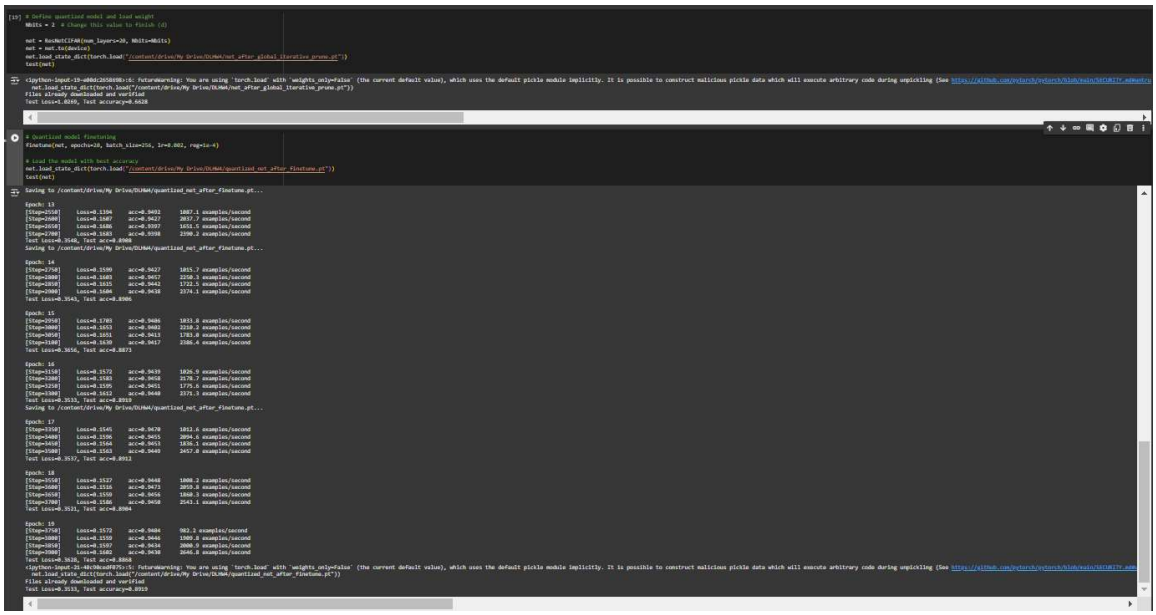
> Nbits = 4 (test acc = 0.9047)



> 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.