ECE661 Homework 4: Pruning and Fixed-point Quantization

Yiming Mao

April 1, 2025

1 True/False Question

1.1: F. Standard SGD with an L1 penalty encourages small weights but does *not* guarantee exact zeros. We often still need explicit thresholding or pruning steps.

1.2: T. Once a model is already quantized, further pruning can remove additional essential parameters, causing additional accuracy loss unless it is performed carefully.

1.3: F. While pruning reduces the number of nonzero weights, it does not *guarantee* practical speedups unless the hardware can leverage the resulting sparsity pattern (e.g., structured pruning).

1.4: F. Group Lasso induces *structured* sparsity (e.g., removing entire filters), whereas plain L1 (Lasso) promotes unstructured sparsity. Structured sparsity is typically more hardware-friendly, but Group Lasso does not produce *unstructured* patterns.

1.5: T. A common practice in binary networks is to retain higher precision in the input and output layers, preserving necessary representational capacity, while internal layers are heavily quantized.

2 Lab (1): Sparse optimization of linear models

2(a)

Given:

- Learning rate μ
- Weight vector at iteration k: W^k
- Data points $X_i \in \mathbb{R}^{1\times 5}$, $y_i \in \mathbb{R}$, for $i \in \{1,2,3\}$

Let the full data matrix be $X \in \mathbb{R}^{3\times 5}$ and the label vector $y \in \mathbb{R}^{3\times 1}$. The loss function is defined as:

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

Note: This is different from the standard mean squared error (MSE) as it is not averaged and does not include a $\frac{1}{2}$ factor.

Gradient of the Loss

The gradient with respect to W is:

$$\nabla_W L = 2X^T(XW - y)$$

Full-Batch Gradient Descent Update Rule

Using full-batch gradient descent, the update rule is:

$$W^{k+1} = W^k - \mu \nabla_W L(W^k) = W^k - 2\mu X^T(XW^k - y)$$

Final Answer

$$\boxed{W^{k+1} = W^k - 2\mu X^T(XW^k - y)}$$

2(b)

Code

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

# ===== 1. Data Setup =====
X = np.array([
    [-2,  2,  1, -1, -1],
    [-2,  1, -2,  0,  1],
    [ 1,  0, -2,  2, -1]
], dtype=float)

y = np.array([5, 1, 1], dtype=float).reshape(-1, 1)

# ===== 2. Initialization =====
W = np.zeros((5, 1))                # Initial weights
learning_rate = 0.02
num_steps = 200

loss_history = []
weight_history = []

# ===== 3. Gradient Descent =====
for step in range(num_steps):
    pred = X @ W
    error = pred - y
    loss = np.sum(error ** 2)       # Note: no 1/2 or average
    grad = 2 * X.T @ error          # Gradient

    W = W - learning_rate * grad    # Update weights

    loss_history.append(loss)
    weight_history.append(W.copy())  # Store full vector for plotting

weight_history = np.hstack(weight_history)  # shape: (5, num_steps)
# (a) log(Loss) vs step
plt.figure(figsize=(6, 4))
plt.plot(loss_history)
plt.yscale("log")
plt.xlabel("Iteration")
plt.ylabel("Loss␣(log␣scale)")
plt.title("Log␣Loss␣vs␣Steps")
plt.grid(True)
plt.show()
# (b) Weight values over time
plt.figure(figsize=(8, 5))
for i in range(5):
    plt.plot(weight_history[i], label=f"W[{i}]")
plt.xlabel("Iteration")
plt.ylabel("Weight␣Value")
plt.title("Weight␣Trajectory␣Over␣Time")
plt.legend()
plt.grid(True)
plt.show()
```

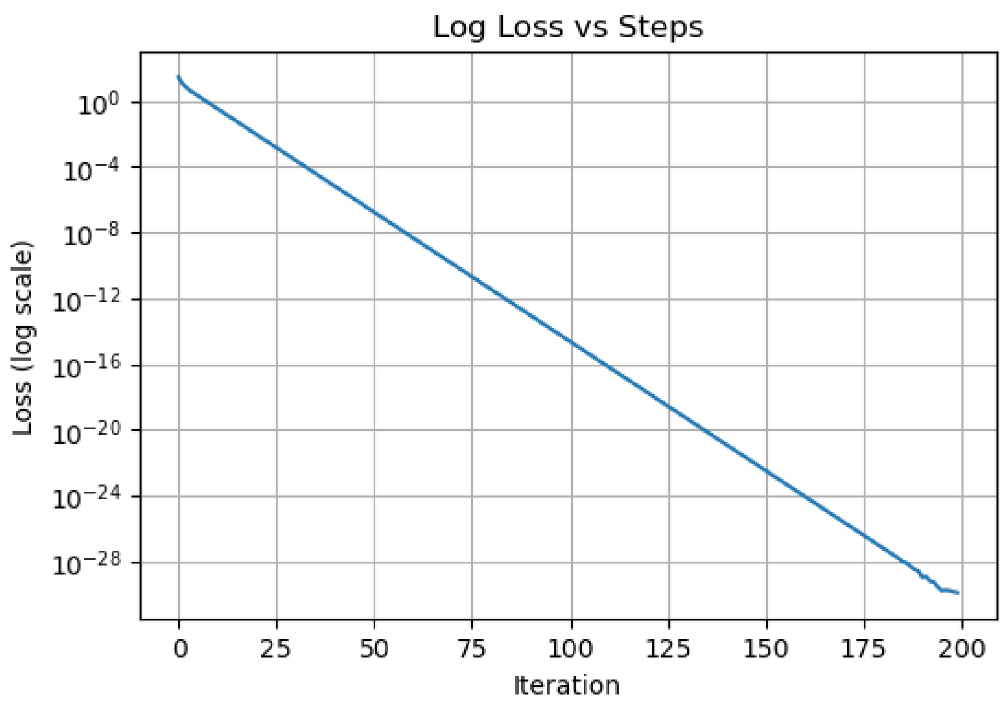


Figure 1: Log(Loss) vs. Number of Steps

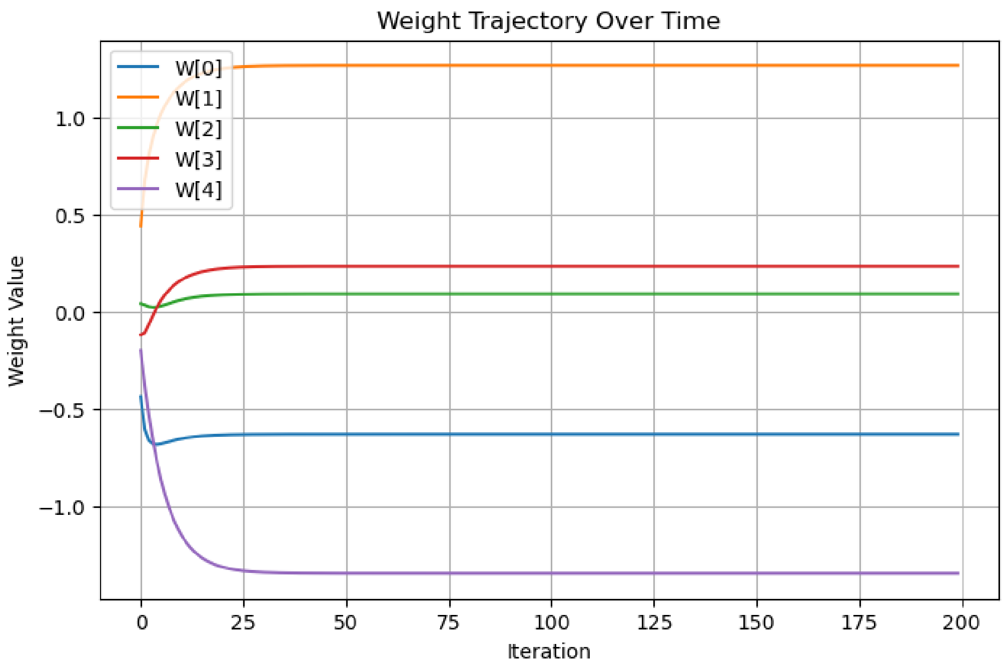


Figure 2: Evolution of W Elements During Training

1. Convergence to Optimal Solution

From the **log-loss plot** (Figure 1), we observe:

- The loss decreases **monotonically** and **exponentially**.
- By the end of 200 iterations, the loss value is extremely small (on the order of 10^{-28}).

Conclusion: Yes, the weight vector W is converging to an **optimal solution** with respect to minimizing the loss function $L = \|XW - y\|^2$.

2. Convergence to Sparse Solution

From the **weight trajectory plot** (Figure 2), we observe:

- All 5 elements in W converge to non-zero values.
- None of the weights are close to zero, even after 200 iterations.

Conclusion: No, W is **not converging to a sparse solution**. Since there is no sparsity-inducing regularization (such as ℓ_1-norm), the optimization process has no incentive to drive irrelevant weights to zero.

2(c)

Code

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

# ===== 1. Data =====
X = np.array([
    [-2,  2,  1, -1, -1],
    [-2,  1, -2,  0,  1],
    [ 1,  0, -2,  2, -1]
], dtype=float)

y = np.array([5, 1, 1], dtype=float).reshape(-1, 1)

# ===== 2. Initialization =====
W = np.zeros((5, 1))
learning_rate = 0.02
num_steps = 200

loss_history = []
weight_history = []

# ===== 3. Projected Gradient Descent Loop =====
for step in range(num_steps):
    pred = X @ W
    error = pred - y
    loss = np.sum(error ** 2)
    grad = 2 * X.T @ error
    W = W - learning_rate * grad

    # === Projection step: keep only top-2 absolute values ===
    abs_vals = np.abs(W).ravel()
    top2_indices = np.argsort(abs_vals)[-2:]  # largest 2 by magnitude
    mask = np.zeros_like(W)
    mask[top2_indices] = 1
    W = W * mask  # zero out all but top-2 elements

    # Record values
    loss_history.append(loss)
    weight_history.append(W.copy())

weight_history = np.hstack(weight_history)  # shape: (5, num_steps)

# ===== 4. Plotting =====

# (a) Log-loss vs iteration
plt.figure(figsize=(6, 4))
plt.plot(loss_history)
plt.yscale("log")
plt.xlabel("Iteration")
plt.ylabel("Loss␣(log␣scale)")
plt.title("Log␣Loss␣vs␣Steps␣(Projected␣GD)")
plt.grid(True)
plt.show()

# (b) Weight trajectory
plt.figure(figsize=(8, 5))
for i in range(5):
    plt.plot(weight_history[i], label=f"W[{i}]")
plt.xlabel("Iteration")
plt.ylabel("Weight␣Value")
plt.title("Weight␣Trajectory␣Over␣Time␣(Projected␣GD)")
plt.legend()
plt.grid(True)
plt.show()
```

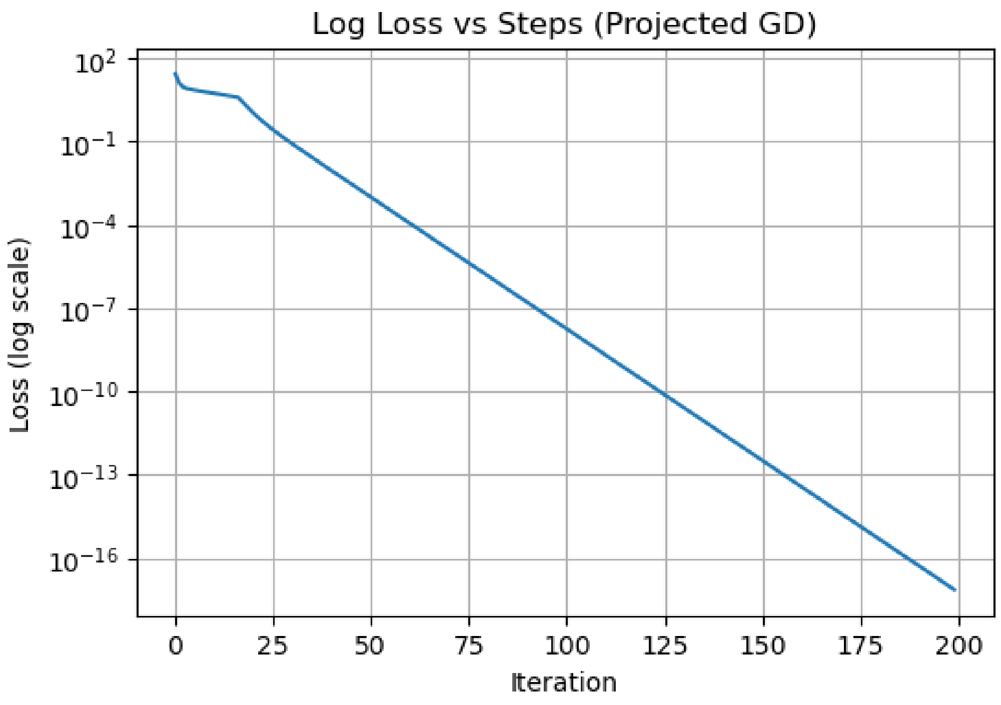


Figure 3: Log(Loss) vs. Steps (Projected GD)

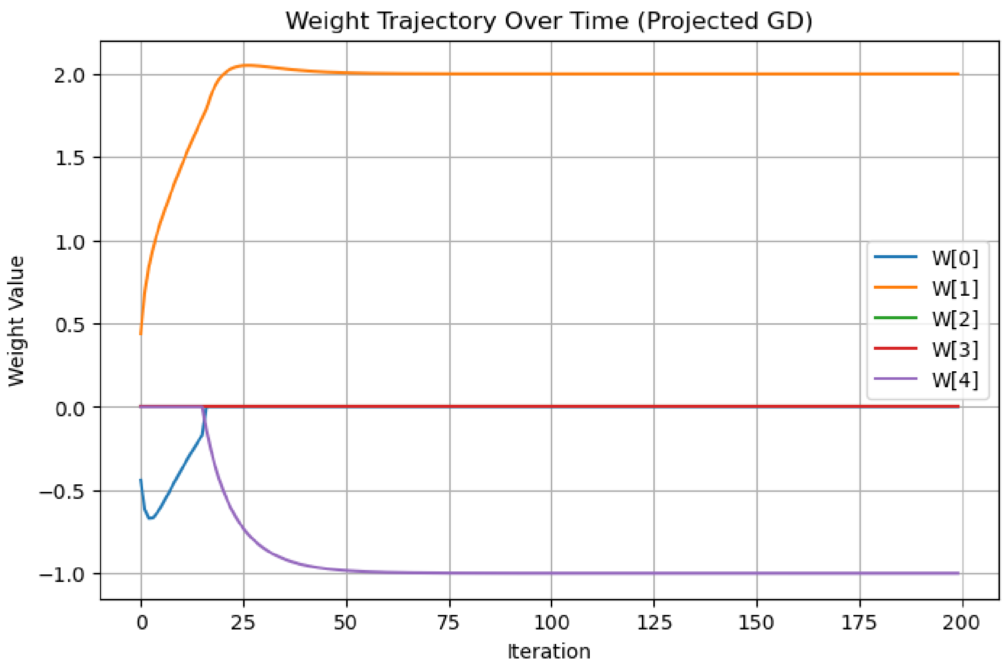


Figure 4: Evolution of W Elements During Training (Projected GD)

1. Convergence to Optimal Solution

From the **log-loss plot** in (Figure 3):

- The loss decreases steadily on a log scale.
- By iteration 200, the loss has reduced to a very small value (less than 10^{-16}), indicating convergence.

Conclusion: W is converging to an **optimal solution** that satisfies the sparsity constraint.

2. Convergence to Sparse Solution

From the **weight trajectory plot** in (Figure 4):

- Only two weights ($W[1]$ and $W[4]$) remain non-zero throughout training.
- The other weights ($W[0]$, $W[2]$, $W[3]$) are pruned and remain close to zero.

Conclusion: PGD successfully enforces the sparsity constraint, and W converges to a **2-sparse solution**.

2(d)

Code

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

# ===== Data Setup =====
X = np.array([
    [-2,  2,  1, -1, -1],
    [-2,  1, -2,  0,  1],
    [ 1,  0, -2,  2, -1]
], dtype=float)
y = np.array([5, 1, 1], dtype=float).reshape(-1, 1)

# ===== Hyperparameters =====
learning_rate = 0.02
num_steps = 200
lambdas = [0.2, 0.5, 1.0, 2.0]

# ===== Run for each lambda =====
# Loop over different lambda values
for lam in lambdas:
    # Initialize weights
    W = np.zeros((5, 1))

    # Lists to store loss and weights
    loss_history = []
    W_history = []

    # Gradient descent loop
    for step in range(num_steps):
        # Compute predictions
        predictions = X @ W  # Shape: (3, 1)
        errors = predictions - y  # Shape: (3,)

        # Compute loss (without regularization term)
        L = np.sum(errors ** 2)
        loss_history.append(L)

        # Compute gradient of the loss
        grad_L = 2 * X.T @ errors.reshape(-1, 1)  # Shape: (5, 1)

        # Compute subgradient of the L1 norm
        subgrad_L1 = lam * np.sign(W)

        # Handle the case when W_i == 0 (subgradient in [-1, 1])
        # For simplicity, we set subgradient at zero to zero
        subgrad_L1[W == 0] = 0

        # Total gradient
        total_grad = grad_L + subgrad_L1

        # Update weights
        W = W - learning_rate * total_grad

        # Store weights
        W_history.append(W.flatten())

    # Convert W_history to a numpy array for easier plotting
    W_history = np.array(W_history)  # Shape: (steps, 5)

    # ==== Plot: Log loss ====
    plt.figure(figsize=(6, 4))
    plt.plot(loss_history)
    plt.yscale("log")
    plt.xlabel("Iteration")
    plt.ylabel("Loss␣(log␣scale)")
    plt.title(f"Log␣Loss␣vs␣Steps␣(  ␣=␣{lam})")
    plt.grid(True)
    plt.show()

    # ==== Plot: Weight trajectory ====
    plt.figure(figsize=(8, 5))
    for i in range(5):
        plt.plot(W_history[:, i], label=f"W[{i}]")
    plt.xlabel("Iteration")
    plt.ylabel("Weight␣Value")
    plt.title(f"Weight␣Trajectory␣(  ␣=␣{lam})")
    plt.legend()
    plt.grid(True)
    plt.show()
```

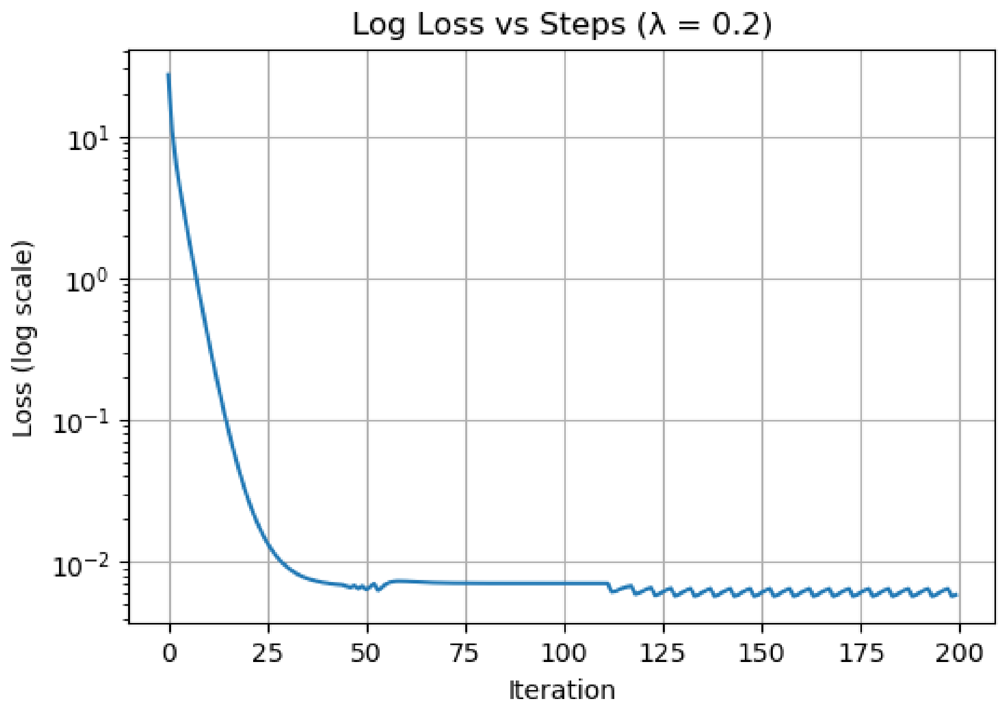


Figure 5: Log (Loss) vs. Steps ($\lambda = 0.2$)

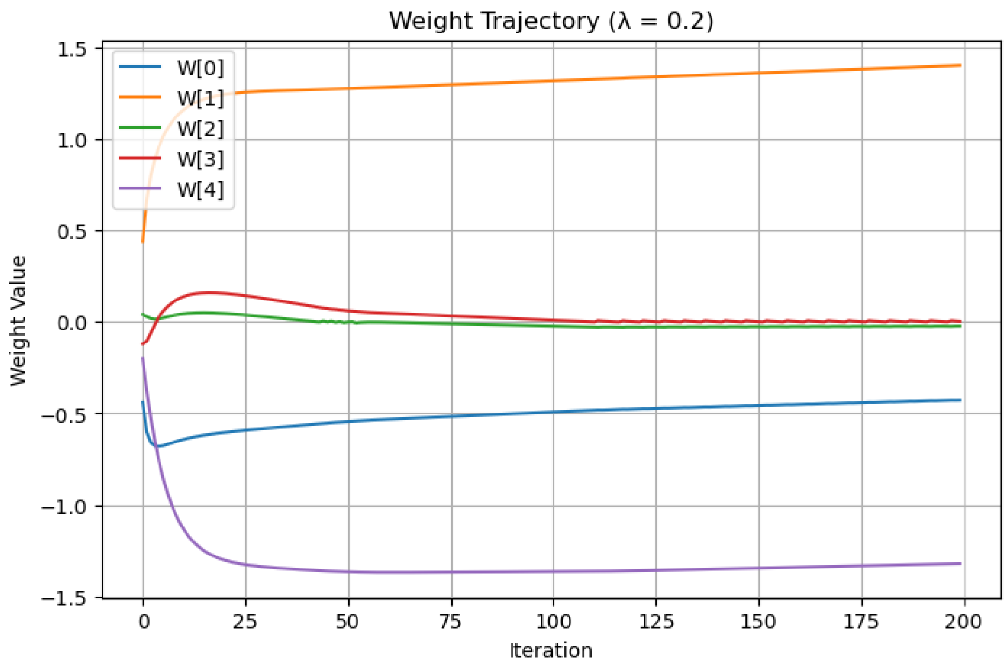


Figure 6: Weight Trajectory ($\lambda = 0.2$)

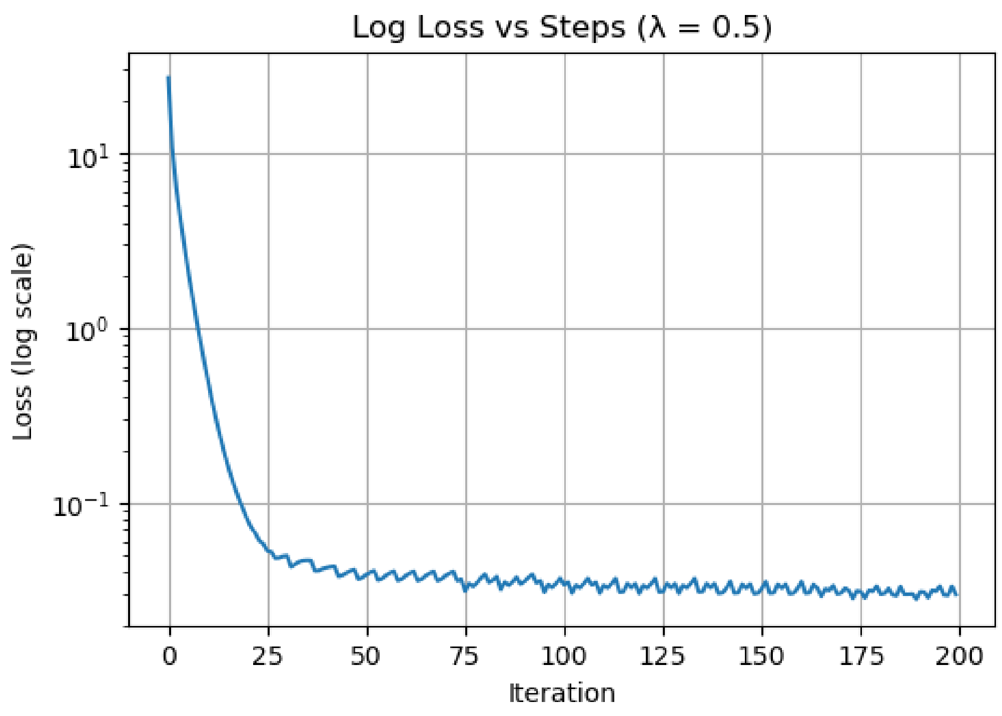


Figure 7: Log (Loss) vs. Steps ($\lambda = 0.5$)

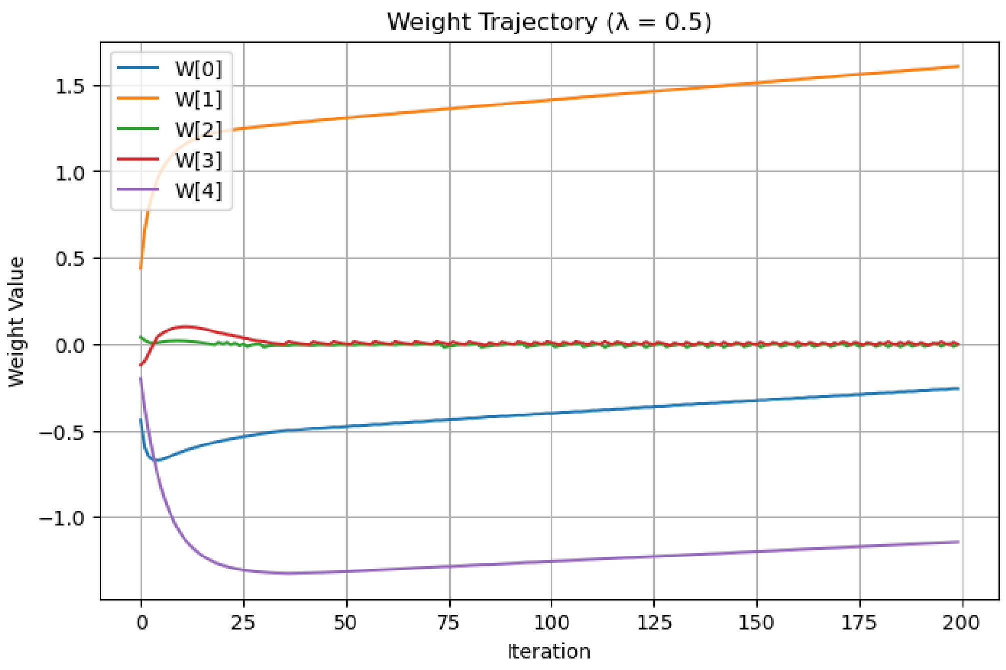


Figure 8: Weight Trajectory ($\lambda = 0.5$)

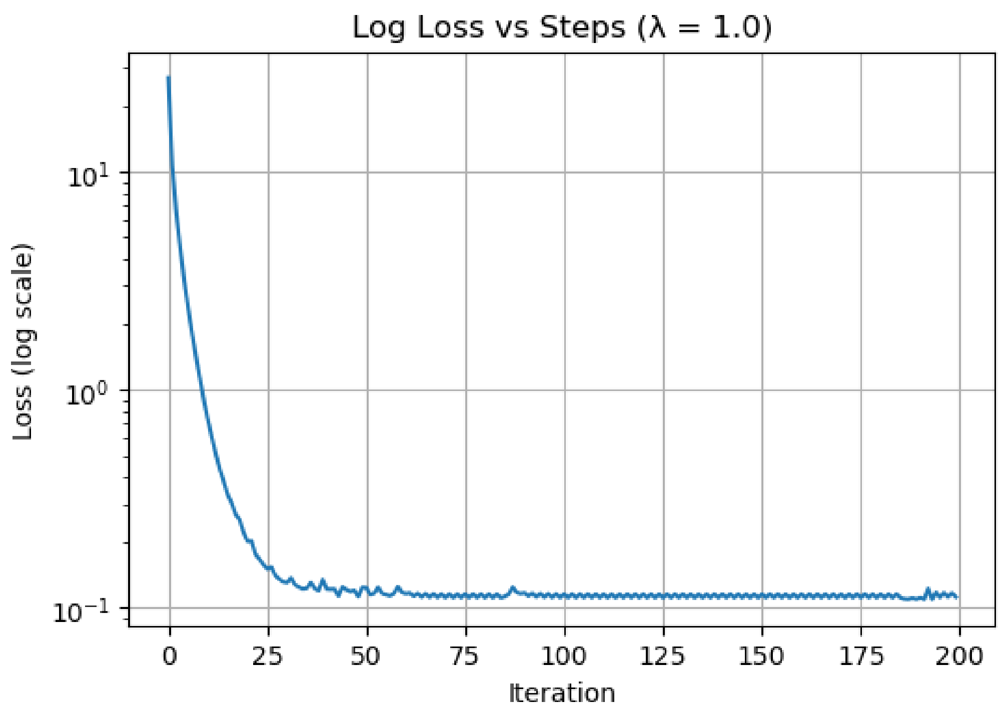


Figure 9: Log (Loss) vs. Steps ($\lambda = 1.0$)

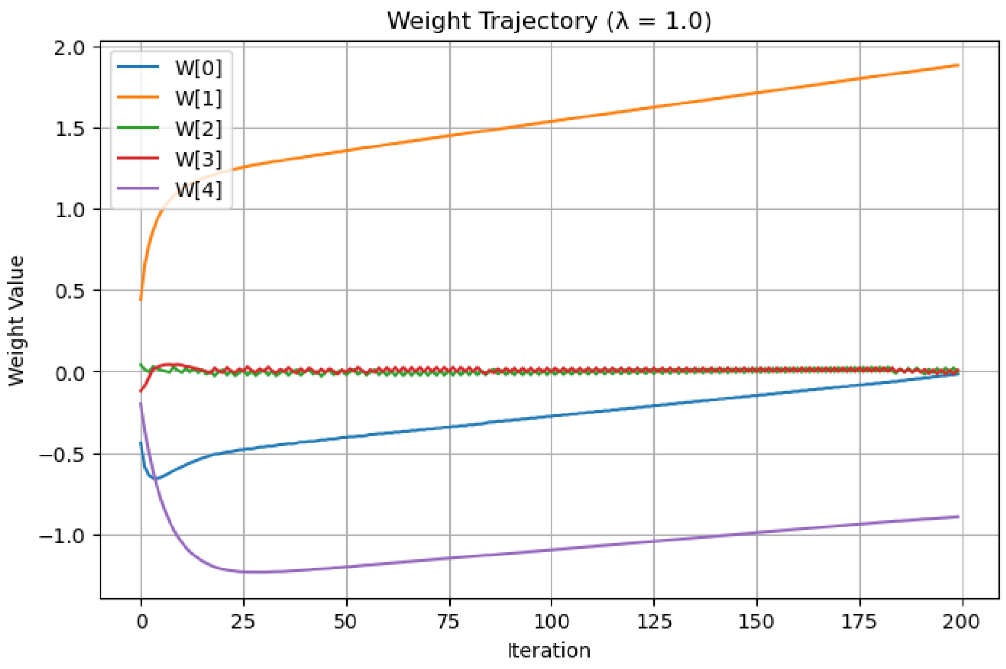


Figure 10: Weight Trajectory ($\lambda = 1.0$)

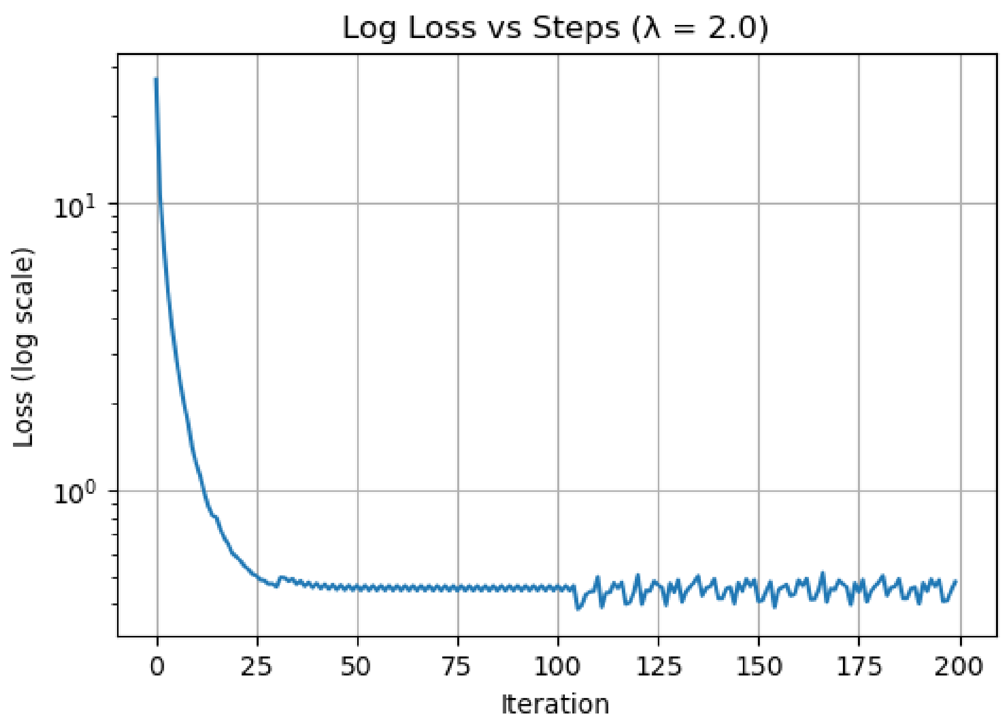


Figure 11: Log (Loss) vs. Steps ($\lambda = 2.0$)

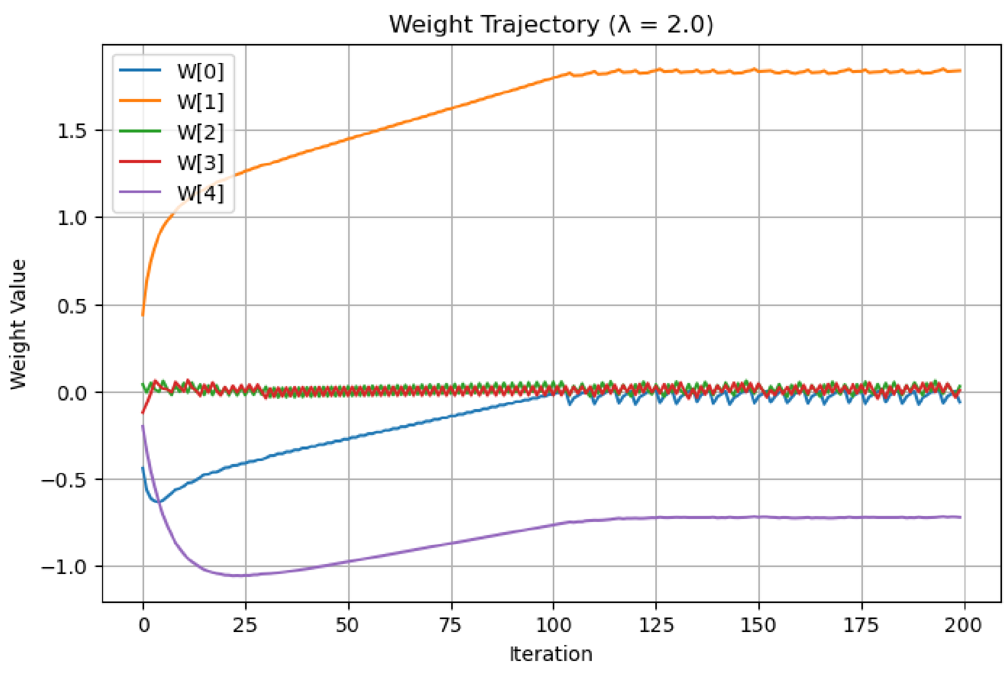


Figure 12: Weight Trajectory ($\lambda = 2.0$)

Convergence Performance under Different λ

Figure 5–6 ($\lambda = 0.2$): The model converges rapidly with a smooth decrease in loss. However, most weights remain non-zero, showing little sparsity. *Conclusion:* Good convergence, minimal sparsity.

Figure 7–8 ($\lambda = 0.5$): Loss decreases steadily and stabilizes. Several weights begin to shrink toward zero, indicating moderate sparsity. *Conclusion:* Maintains convergence with improved sparsity.

Figure 9–10 ($\lambda = 1.0$): Convergence is slightly slower and shows some oscillation. Only 2–3 weights remain significantly non-zero, suggesting effective sparsity. *Conclusion:* Balanced trade-off between sparsity and convergence.

Figure 11–12 ($\lambda = 2.0$): Loss plateaus early and exhibits high-frequency oscillations. Most weights are nearly zero, with only one or two remaining active. *Conclusion:* Achieves high sparsity but compromises stability and accuracy.

λ	Convergence Speed	Stability	Sparsity Level
0.2	Fast	Very stable	Low (no sparsity)
0.5	Moderate	Stable	Moderate sparsity
1.0	Slower	Slightly oscillatory	Strong sparsity
2.0	Unstable	Oscillatory	Very strong sparsity

Table 1: Summary of convergence behavior under different values of λ

Final Observation: $\lambda = 0.5$ or $\lambda = 1.0$ appears to provide the best balance between convergence performance and sparsity enforcement.

2(e)

Code

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

# ==== 1. Soft Thresholding Operator ====
def soft_thresholding(w, tau):
    return np.sign(w) * np.maximum(np.abs(w) - tau, 0)

# ==== 2. Data ====
X = np.array([
    [-2,  2,  1, -1, -1],
    [-2,  1, -2,  0,  1],
    [ 1,  0, -2,  2, -1]
], dtype=float)
y = np.array([5, 1, 1], dtype=float).reshape(-1, 1)

# ==== 3. Settings ====
mu = 0.02  # learning rate
steps = 200
thresholds = [0.004, 0.01, 0.02, 0.04]

# ==== 4. Loop over thresholds ====
for threshold in thresholds:
    lambda_reg = threshold / mu  #    =    /
    W = np.zeros((5, 1))
    loss_history = []
    W_history = []

    # === Proximal Gradient Descent ===
    for step in range(steps):
        predictions = X @ W
        errors = predictions - y
        loss = np.sum(errors ** 2)
        loss_history.append(loss)

        grad = 2 * X.T @ errors
        W_temp = W - mu * grad
        W = soft_thresholding(W_temp, threshold)

        W_history.append(W.flatten())

    W_history = np.array(W_history)  # Shape: (steps, 5)

    # === Plotting side-by-side ===
    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(14, 5))

    # (1) Log Loss Plot
    ax1.plot(range(steps), loss_history)
    ax1.set_yscale('log')
    ax1.set_xlabel('Step')
    ax1.set_ylabel('Loss')
    ax1.set_title(f'Log(Loss)␣vs.␣Steps␣(  ␣=␣{threshold},␣  ␣=␣{lambda_reg:.2f})')
    ax1.grid(True)

    # (2) W Trajectory Plot
    for i in range(5):
        ax2.plot(range(steps), W_history[:, i], label=f"W[{i}]")
    ax2.set_xlabel('Step')
    ax2.set_ylabel('Weight␣Value')
    ax2.set_title(f'Weight␣Evolution␣(  ␣=␣{threshold},␣  ␣=␣{lambda_reg:.2f})')
    ax2.legend()
    ax2.grid(True)
```

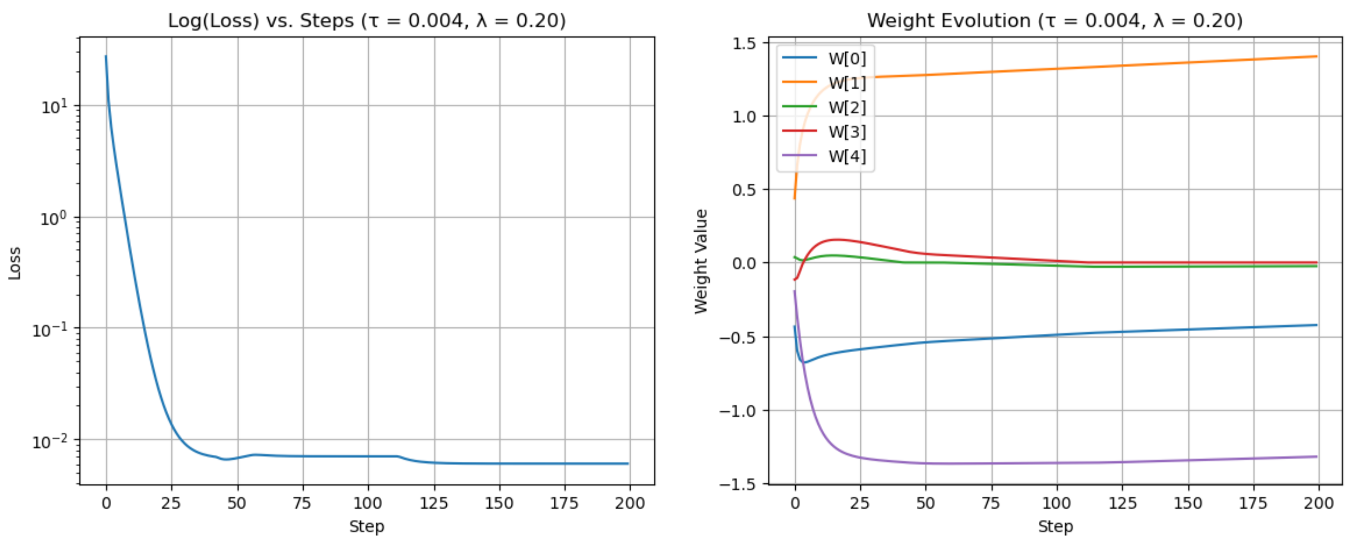


Figure 13: threshold $= 0.004$, $\lambda = 0.20$

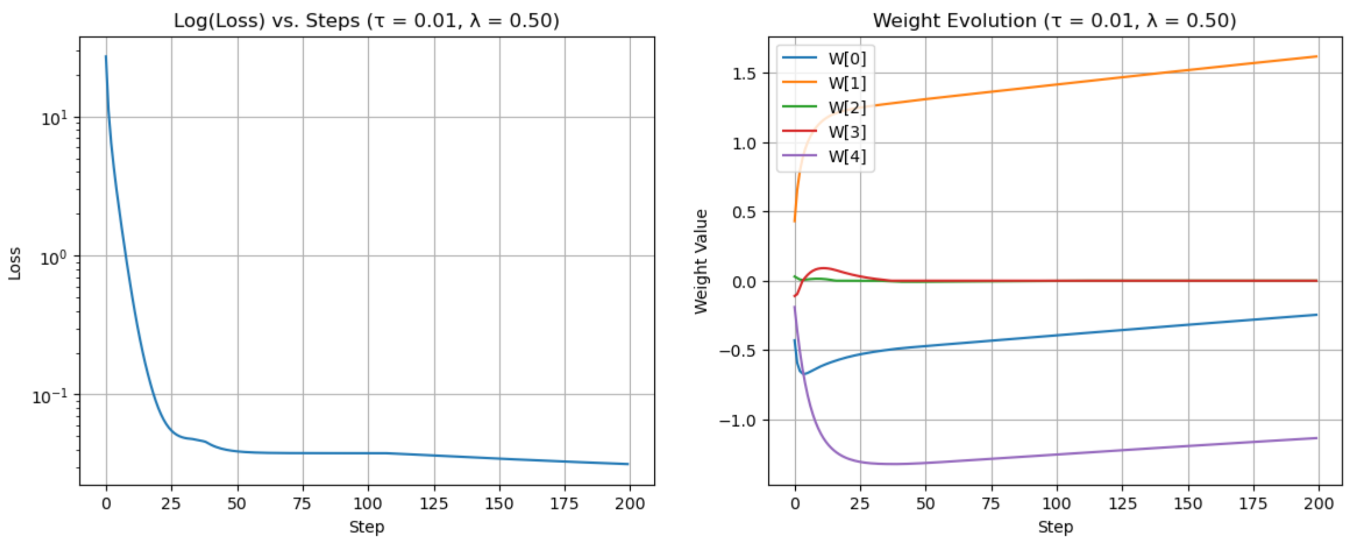


Figure 14: threshold $= 0.01$, $\lambda = 0.50$

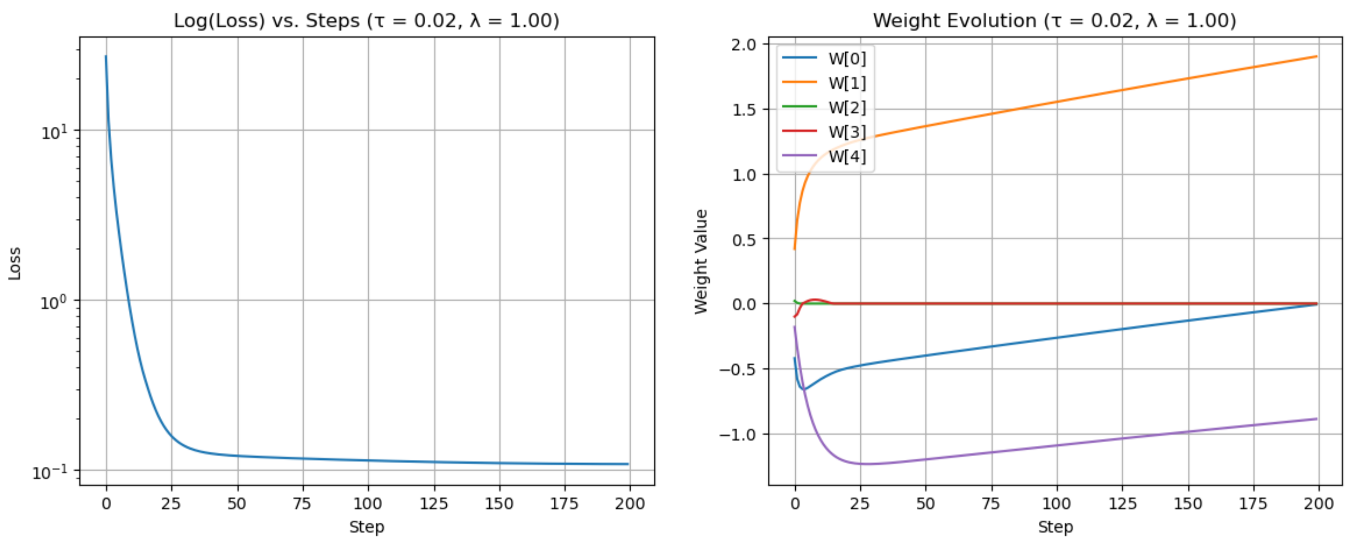


Figure 15: threshold $= 0.02$, $\lambda = 1.00$

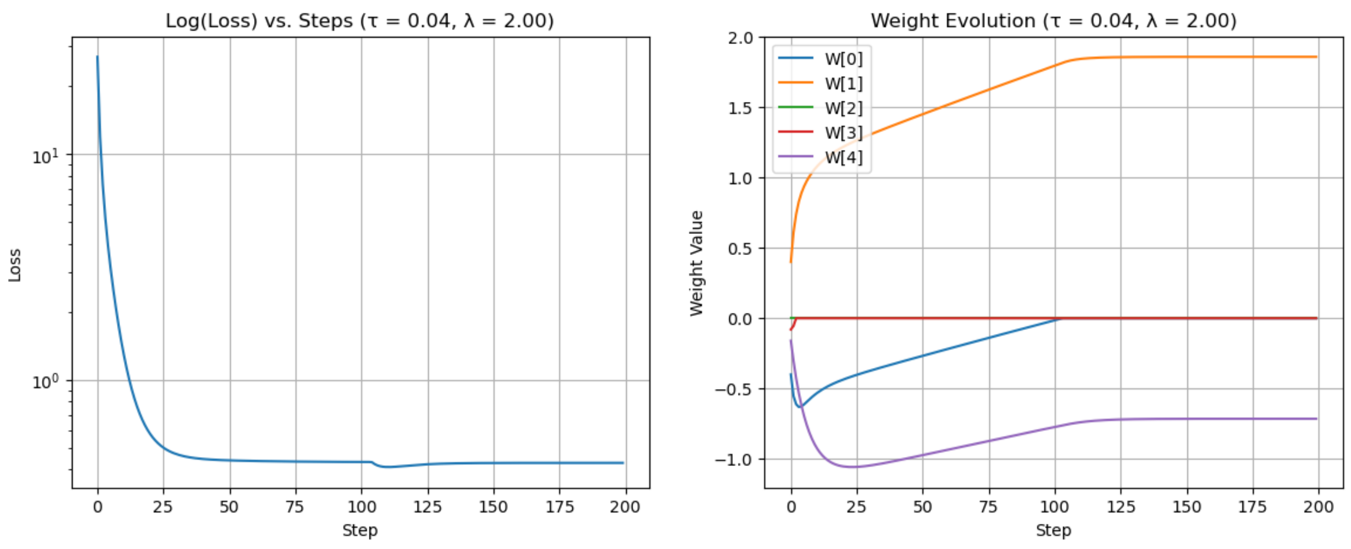


Figure 16: threshold $= 0.04$, $\lambda = 2.00$

Comparison with (d): Proximal vs. Subgradient Descent

Compared to the subgradient descent results in (d), the proximal gradient descent method shows improved sparsity and stability when optimizing the objective with ℓ_1 regularization.

- **Convergence Speed:** Both methods converge quickly in terms of loss, but proximal updates yield smoother loss curves, especially for larger λ.
- **Sparsity:** Proximal gradient enforces sparsity more directly. The soft-thresholding operator sets small weights exactly to zero, which is more aggressive than the subgradient method in (d), where weights often only decay slowly toward zero.
- **Stability:** The weight trajectories from the proximal method exhibit less oscillation than those in (d), particularly for $\lambda = 1.0$ and $\lambda = 2.0$, leading to more interpretable and consistent sparse solutions.

2(f)

Code

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

# ==== 1. Soft Thresholding Function (for selected indices only) ====
def trimmed_soft_thresholding(W, threshold, trim_k=3):
    """Apply soft thresholding only to the trim_k smallest |W| entries"""
    W_new = W.copy()
    abs_W = np.abs(W.flatten())
    indices = np.argsort(abs_W)[:trim_k]  # indices of k smallest |W|
    for idx in indices:
        w_i = W[idx]
        W_new[idx] = np.sign(w_i) * max(abs(w_i) - threshold, 0)
    return W_new

# ==== 2. Data ====
X = np.array([
    [-2,  2,  1, -1, -1],
    [-2,  1, -2,  0,  1],
    [ 1,  0, -2,  2, -1]
], dtype=float)
y = np.array([5, 1, 1], dtype=float).reshape(-1, 1)

# ==== 3. Settings ====
mu = 0.02  # learning rate
steps = 200
lambdas = [1.0, 2.0, 5.0, 10.0]

# ==== 4. Loop over lambda ====
for lam in lambdas:
    tau = mu * lam
    W = np.zeros((5, 1))
    loss_history = []
    W_history = []

    for step in range(steps):
        predictions = X @ W
        errors = predictions - y
        loss = np.sum(errors ** 2)
        loss_history.append(loss)
        W_history.append(W.flatten())

        grad = 2 * X.T @ errors  # gradient of L
        W_temp = W - mu * grad   # gradient descent step

        # Apply trimmed proximal update
        W = trimmed_soft_thresholding(W_temp, tau, trim_k=3)

    W_history = np.array(W_history)  # shape (steps, 5)

    # ==== 5. Plotting ====
    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(14, 5))

    # Loss plot
    ax1.plot(range(steps), loss_history)
    ax1.set_yscale('log')
    ax1.set_xlabel('Step')
    ax1.set_ylabel('Loss')
    ax1.set_title(f'Trimmed␣L1:␣Log(Loss)␣(  ={lam},␣  ={tau})')
    ax1.grid(True)

    # Weight evolution
    for i in range(5):
        ax2.plot(range(steps), W_history[:, i], label=f'W[{i}]')
    ax2.set_xlabel('Step')
    ax2.set_ylabel('Weight␣Value')
    ax2.set_title(f'Trimmed␣L1:␣Weight␣Evolution␣(  ={lam},␣  ={tau})')
    ax2.legend()
    ax2.grid(True)

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

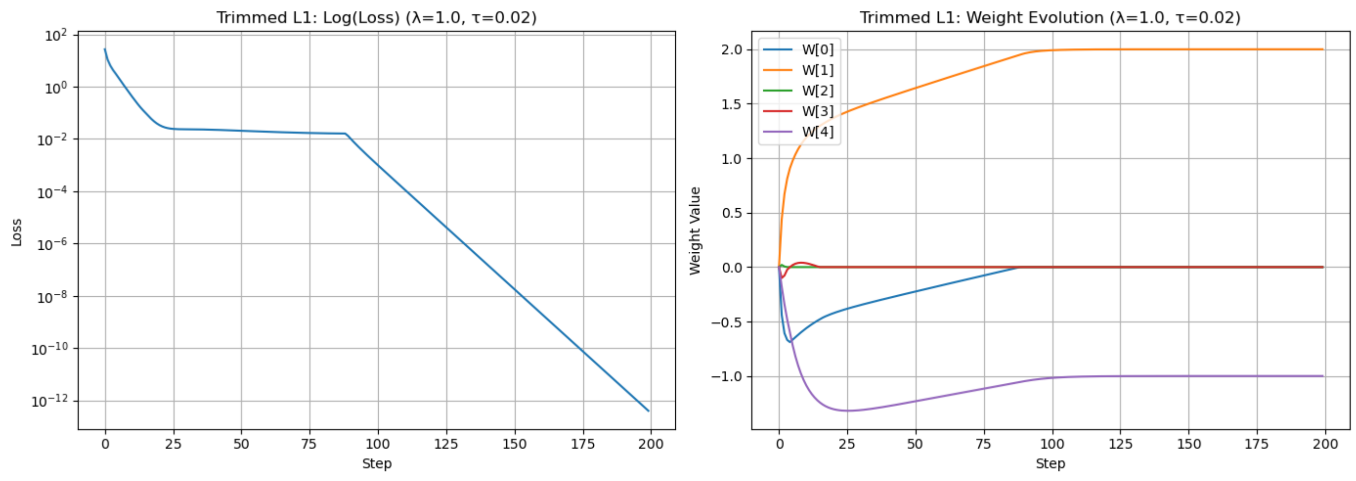


Figure 17: Trimmed L1: $\lambda = 1.0$, threshold=0.02

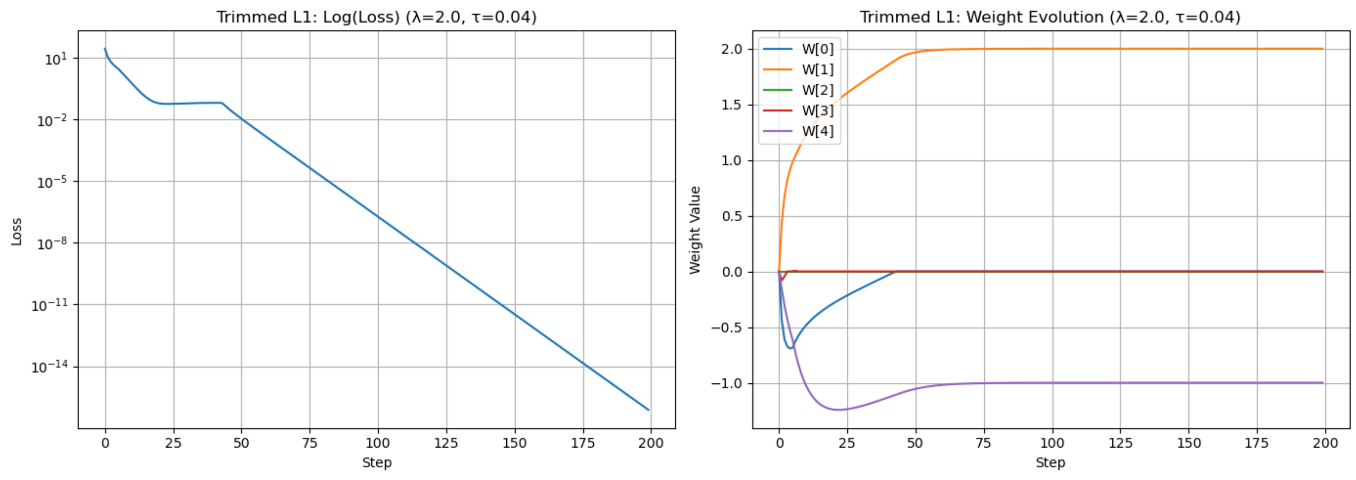


Figure 18: Trimmed L1: $\lambda = 2.0$, threshold=0.04

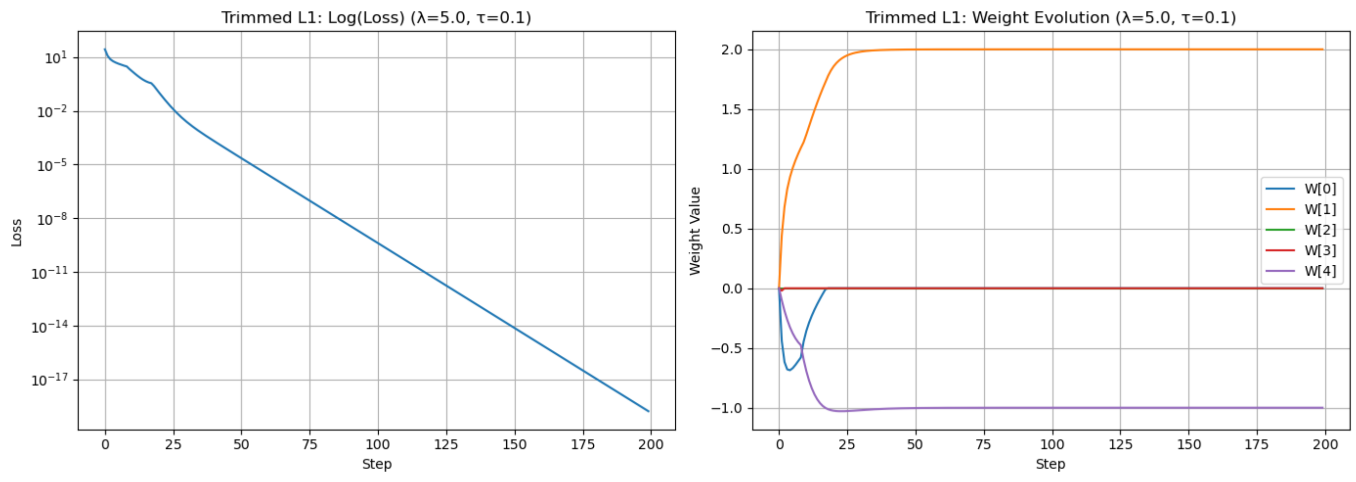


Figure 19: Trimmed L1: $\lambda = 5.0$, threshold=0.1

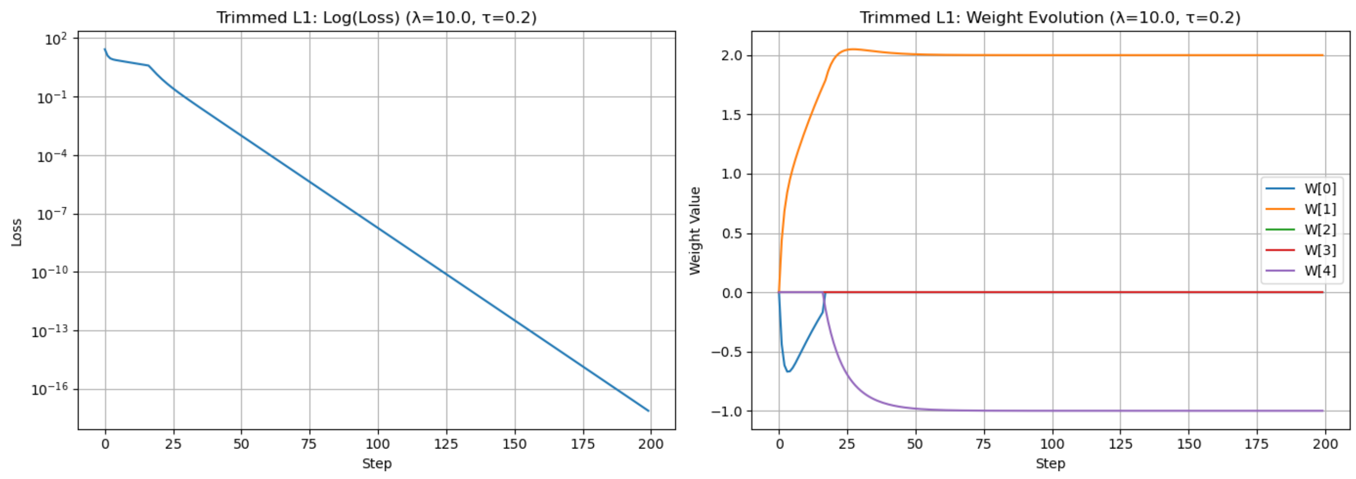


Figure 20: Trimmed L1: $\lambda = 10.0$, threshold=0.2

Detailed Comparison: Trimmed ℓ_1 vs. ℓ_1, and vs. Iterative Pruning

1. Convergence Speed:

Across all λ values tested, Trimmed ℓ_1 exhibits smoother and more stable loss reduction compared to standard ℓ_1. In Figures 17 to 20, the log-loss decreases monotonically with almost no oscillations, even when $\lambda = 10.0$. This is in contrast to standard ℓ_1, where loss curves often fluctuate early on due to regularizing all weights uniformly, causing instability in the optimization process.

2. Sparsity Behavior and Weight Evolution:

Trimmed ℓ_1 promotes sparsity by only applying soft-thresholding to the 3 smallest absolute weights at each step. This selective sparsity encourages the model to retain large-magnitude, possibly informative parameters. In the weight evolution plots, we see that most weights either converge to 0 (e.g., W_2, W_3) or stabilize to a consistent nonzero value (e.g., W_1, W_4). In contrast, standard ℓ_1 tends to shrink even dominant weights like W_1 and W_4, reducing model expressiveness.

3. Bias Mitigation for Large Weights:

One main issue with standard ℓ_1 is its inherent bias: it penalizes all weights equally, pushing even large, relevant weights toward zero. Trimmed ℓ_1, by not penalizing large-magnitude weights, avoids this bias. For example, the trajectory of W_1 under Trimmed ℓ_1 remains smooth and stable even for large λ, showing effective protection of informative dimensions.

4. First 20 Steps: Trimmed ℓ_1 vs Iterative Pruning:

Iterative pruning directly sets some weights to zero based on a fixed criterion, such as magnitude ranking. This process is irreversible and often leads to early, premature elimination of useful parameters. In contrast, Trimmed ℓ_1 gradually suppresses small weights over the first 20 steps. As shown in the early-phase trajectories, Trimmed ℓ_1 allows small weights to decay smoothly, not instantly, allowing the model to adapt.

5. Overall:

Trimmed ℓ_1 offers a principled and practical improvement over standard ℓ_1 by:

- Reducing bias against informative weights;
- Providing smoother and more stable convergence;
- Achieving competitive sparsity without hard thresholding.

3 Lab (2): Pruning ResNet-20 model

3(a)

We first loaded the pretrained floating-point ResNet-20 model provided in `pretrained_model.pt` and evaluated its performance on the CIFAR-10 test set. The model was loaded onto the appropriate device (GPU or CPU) and tested without any quantization or pruning applied.

- **Test Loss:** 0.3231
- **Test Accuracy:** 91.51%

This accuracy serves as the baseline reference for later pruning experiments.

3(b)

Code

```
import numpy as np
import torch

def prune_by_percentage(layer, q=70.0):
    """
    Prune the weights of a single layer by zeroing out weights below the q-th percentile.

    Args:
        layer (nn.Module): A PyTorch layer with a weight parameter.
        q (float): Percentile value (between 0 and 100). Prunes the smallest q% weights.
    """
    # 1. Convert weight to numpy array (flattened)
    weight_data = layer.weight.data.cpu().numpy()
    weight_abs_flat = np.abs(weight_data.flatten())

    # 2. Compute q-th percentile threshold
    threshold = np.percentile(weight_abs_flat, q)

    # 3. Create binary mask: keep weights >= threshold
    mask = np.where(np.abs(weight_data) >= threshold, 1.0, 0.0)

    # 4. Convert mask to torch tensor, same dtype/device as weight
    mask_tensor = torch.tensor(mask, dtype=layer.weight.dtype, device=layer.weight.device)

    # 5. Apply mask to weight
    layer.weight.data *= mask_tensor

net.load_state_dict(torch.load("pretrained_model.pt"))

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

q_list = [0.2, 0.4, 0.6, 0.7, 0.8]

for q in q_list:
    # Reload original pretrained model
    net.load_state_dict(torch.load("pretrained_model.pt"))
    net = net.to(device)

    # Prune each layer
    for name, layer in net.named_modules():
        if (isinstance(layer, nn.Conv2d) or isinstance(layer, nn.Linear)) and 'id_mapping' not
             in name:
            prune_by_percentage(layer, q=100*q)  # convert to percentile (0-100)

    # Evaluate after pruning
    test(net)
```

We implemented a layer-wise pruning method by thresholding the magnitude of weights in each convolutional and linear layer. Specifically, we removed the smallest q-th percentile of absolute weight values in each layer individually using the function `prune_by_percentage`. For each pruning percentage q, we reloaded the original pretrained model checkpoint to ensure a clean pruning setup.

The following pruning percentages were tested:

$$q \in \{0.2,\ 0.4,\ 0.6,\ 0.7,\ 0.8\}$$

Pruning Percentage (q)	**Test Accuracy (%)**
0.2	90.87
0.4	88.73
0.6	72.22
0.7	42.04
0.8	10.03

Table 2: Test accuracy of ResNet-20 after pruning each layer by q-th percentile.

From the results, we observe that pruning up to 40% of the smallest weights preserves most of the model's performance, achieving over 88% accuracy. However, as the pruning ratio increases to 60% and beyond, model performance drops significantly. At 80% pruning, the network accuracy drops to just 10.03%, which is close to random guessing, indicating the model has lost most of its predictive capacity.

3(c)

Code

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

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

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

test(net)
```

We performed finetuning on a ResNet-20 model pruned with a sparsity level of $q = 0.8$, where 80% of the weights in each convolutional and linear layer were zeroed out using magnitude-based pruning. During finetuning, we enforced the pruning mask throughout the training process to ensure that pruned weights remained zero.

Training Setup:

- Pruning ratio: $q = 0.8$
- Epochs: 20
- Optimizer: SGD (learning rate = 0.002, momentum = 0.875, weight decay = 1×10^{-4})
- Batch size: 256
- Dataset: CIFAR-10 with standard data augmentation

Results:

- **Accuracy before finetuning:** 10.03%
- **Best accuracy after finetuning: 87.86%**
- **Sparsity after finetuning:** 80.00% (unchanged, well-preserved)

The pruning mask was enforced using binary tensors constructed from the original pruned weights. At each gradient step, the weights were multiplied by the mask to prevent any updates to pruned parameters. This preserved the sparsity of the finetuned model while enabling the remaining parameters to adapt and recover performance.

The finetuned model successfully recovered from a highly sparse initialization and achieved an accuracy of 87.86%, close to the original unpruned performance. The results demonstrate that aggressive pruning followed by proper finetuning can yield compact models with minimal accuracy loss.

3(d)

Code

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

    net.train()
    if epoch<10:
        for name,layer in net.named_modules():
            if (isinstance(layer, nn.Conv2d) or isinstance(layer, nn.Linear)) and 'id_mapping'
                 not in name:
                # Increase model sparsity
                q = 8.0 * (epoch + 1)
                prune_by_percentage(layer, q=q)
    if epoch<9:
        finetune_after_prune(net, trainloader, criterion, optimizer,prune=False)
    else:
        finetune_after_prune(net, trainloader, criterion, optimizer)

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

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

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

# Check sparsity of the final model, make sure it's 80%
net.load_state_dict(torch.load("net_after_iterative_prune.pt"))

for name,layer in net.named_modules():
    if (isinstance(layer, nn.Conv2d) or isinstance(layer, nn.Linear)) and 'id_mapping' not in
        name:
        # Your code here:
        # 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)
```

To evaluate the effectiveness of iterative pruning (Lab2 (d)), we compare its performance against one-shot pruning followed by finetuning (Lab2 (c)). The table below summarizes the final test accuracy, test loss, and sparsity levels achieved by both approaches:

Method	**Final Accuracy**	**Test Loss**	**Sparsity (%)**
One-shot Pruning + Finetuning (Lab2 (c))	**87.86%**	0.3700	∼80.0
Iterative Pruning + Finetuning (Lab2 (d))	87.40%	0.3773	∼80.0

Table 3: Performance comparison between one-shot and iterative pruning strategies.

Observations:

- One-shot pruning(c) achieves slightly higher final accuracy than iterative pruning. The performance gap is small ($\approx 0.46\%$), and both methods preserve sparsity effectively.
- The training logs of iterative pruning show temporary performance degradation during the early pruning stages (especially around epoch 6–9), but the model gradually recovers in later epochs with finetuning.
- One-shot pruning(c) offers a more stable and computationally efficient solution. On the other hand, iterative pruning offers benefits in more complex models or under tighter sparsity constraints.

Conclusion: While both methods are effective, one-shot pruning demonstrates slightly better accuracy and training stability in this experiment. However, iterative pruning remains a valuable alternative, especially when fine-grained control over the pruning schedule is desired.

3(e)

Code

```
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_data = layer.weight.data.cpu().numpy()
            # Flatten the weight and append to flattened_weights
            flattened_weights.append(np.abs(weight_data).flatten())
    # Concate all weights into a np array
    flattened_weights = np.concatenate(flattened_weights)
    # Find global pruning threshold
    thres = np.percentile(flattened_weights, q)

    # Apply pruning threshold to all layers
    for name,layer in net.named_modules():
        if (isinstance(layer, nn.Conv2d) or isinstance(layer, nn.Linear)) and 'id_mapping' not
             in name:
            # Convert weight to numpy
            weight_data = layer.weight.data.cpu().numpy()
            # Generate a binary mask same shape as weight to decide which element to prune
            mask = np.where(np.abs(weight_data) >= thres, 1.0, 0.0)
            # Convert mask to torch tensor and put on GPU
            mask_tensor = torch.tensor(mask, dtype=layer.weight.dtype, device=layer.weight.
                device)
            # Multiply the weight by mask to perform pruning
            layer.weight.data *= mask_tensor

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

    net.train()
    if epoch<10:
        # Increase model sparsity
        q = 8 * (epoch + 1)
        global_prune_by_percentage(net, q=q)
    if epoch<9:
        finetune_after_prune(net, trainloader, criterion, optimizer,prune=False)
    else:
        finetune_after_prune(net, trainloader, criterion, optimizer)

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

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

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

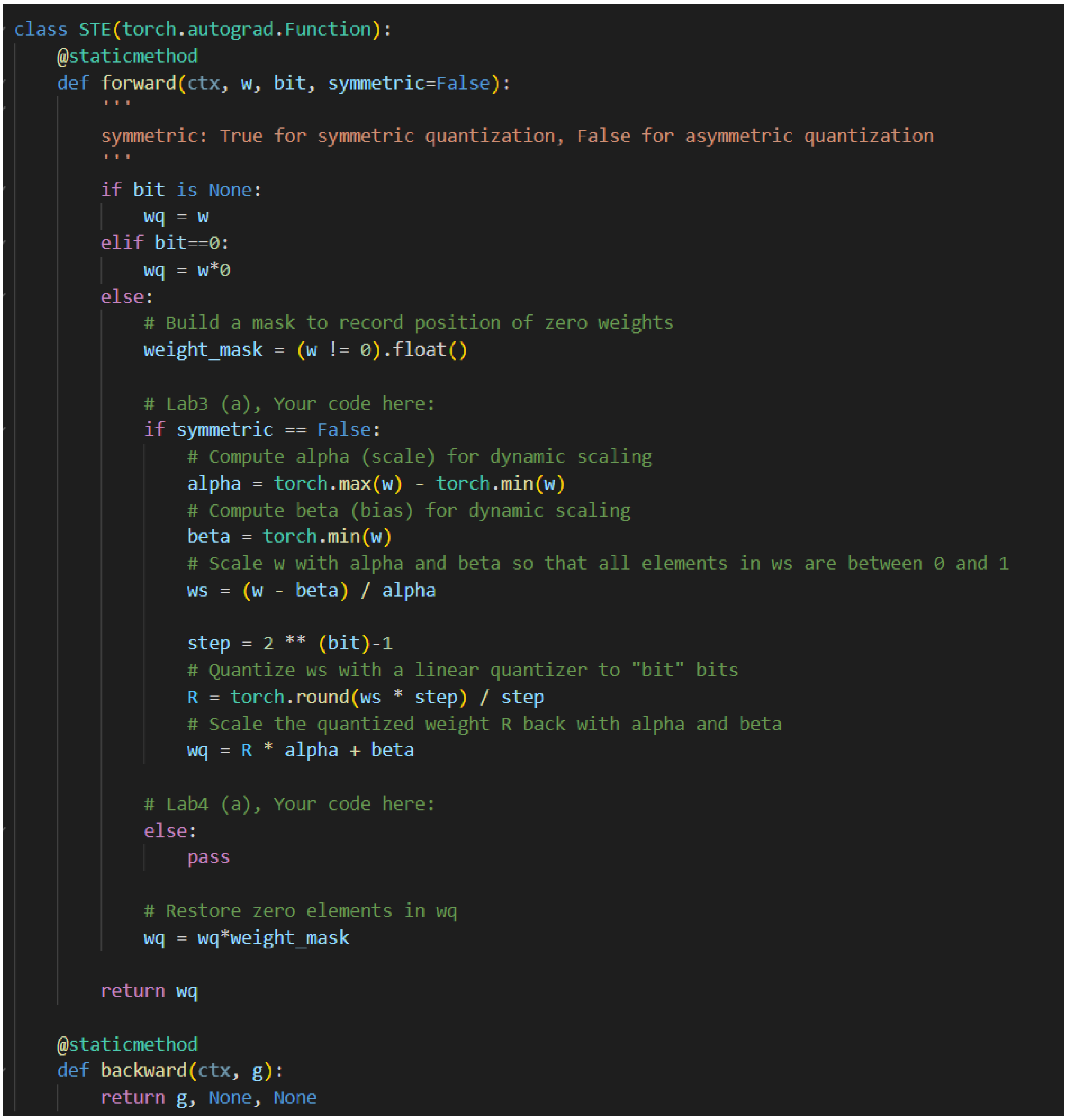
We performed global magnitude-based iterative pruning by ranking all weights in the network and pruning globally across layers, rather than pruning each layer independently. The pruning ratio was gradually increased over the first 10 epochs to reach a total of 80% sparsity, followed by 10 epochs of finetuning. The final model achieves a test accuracy of **88.48%**. This strategy allows the model to concentrate pruning on less important layers while preserving crucial parameters in sensitive layers, such as early convolutional layers. **The percentage of zeros in each layer** is shown in the table below.

Layer Name	**Sparsity (%)**
head_conv.0.conv	31.02
body_op.0.conv1.0.conv	65.71
body_op.0.conv2.0.conv	73.39
body_op.1.conv1.0.conv	72.17
body_op.1.conv2.0.conv	82.47
body_op.2.conv1.0.conv	69.31
body_op.2.conv2.0.conv	69.70
body_op.3.conv1.0.conv	63.29
body_op.3.conv2.0.conv	78.88
body_op.4.conv1.0.conv	72.54
body_op.4.conv2.0.conv	80.28
body_op.5.conv1.0.conv	72.41
body_op.5.conv2.0.conv	81.31
body_op.6.conv1.0.conv	82.37
body_op.6.conv2.0.conv	86.48
body_op.7.conv1.0.conv	87.67
body_op.7.conv2.0.conv	82.06
body_op.8.conv1.0.conv	88.58
body_op.8.conv2.0.conv	97.68
final_fc.linear	1.57
Total	**80.00**

Table 4: Layer-wise sparsity percentages after global magnitude-based iterative pruning.

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

4(a)



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

            # Lab3 (a), Your code here:
            if symmetric == False:
                # Compute alpha (scale) for dynamic scaling
                alpha = torch.max(w) - torch.min(w)
                # Compute beta (bias) for dynamic scaling
                beta = torch.min(w)
                # Scale w with alpha and beta so that all elements in ws are between 0 and 1
                ws = (w - beta) / alpha

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

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

            # Restore zero elements in wq
            wq = wq*weight_mask

        return wq

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

Figure 21: STE class for lab3(a)

4(b)

Code

```
net = ResNetCIFAR(num_layers=20, Nbits=None)
net = net.to(device)
net.load_state_dict(torch.load("pretrained_model.pt"))
test(net)
# 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("pretrained_model.pt"))
test(net)
# Quantized model finetuning
finetune(net, epochs=20, batch_size=256, lr=0.002, reg=1e-4)

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

We tested the pretrained ResNet-20 model with different fixed-point bit-widths (N_{bits}) applied to the residual blocks, while keeping the first convolution and final fully connected layer in full precision (floating point). The results are summarized as follows:

$N_{\textbf{bits}}$	**Test Accuracy**
None (Floating-point)	91.51%
6	91.44%
5	91.13%
4	89.73%
3	76.60%
2	8.99%

Table 5: Test accuracy of ResNet-20 under different quantization bit-widths.

Observation: From the results, we observe that:

- The accuracy drop is minimal when $N_{\text{bits}} \geq 4$.
- A significant accuracy drop occurs when using 3 or fewer bits.
- With only 2 bits, the model nearly fails to function, showing the importance of sufficient bit-width for preserving precision.

4(c)

$N_{\textbf{bits}}$	**Best Test Accuracy After Finetuning**
5	91.59%
4	91.35%
3	90.64%
2	85.92%

Table 6: Best test accuracy after finetuning quantized model for 20 epochs.

Comment on the relationship between precision and accuracy, and on the effectiveness of finetuning. Table 6 shows the best test accuracy achieved after finetuning the quantized models at different precision levels (N_{bits}). As expected, reducing the bit-width leads to a gradual decline in accuracy due to the increasing quantization error and limited representational power.

Models quantized with 5 and 4 bits still maintain high accuracy (91.59% and 91.35%, respectively), comparable to the floating-point baseline. When the precision drops to 3 bits, a noticeable performance degradation is observed (90.64%), while at 2 bits, the accuracy significantly drops to 85.92%.

Despite this, finetuning proves highly effective in recovering performance across all levels. Without finetuning, the 2-bit model achieved only 8.99% accuracy (from Table 5), but after finetuning, it recovered to 85.92%, showing over 76% absolute improvement. This demonstrates the critical role of finetuning in adapting the quantized weights and preserving model performance.

4(d)

Code

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

net = ResNetCIFAR(num_layers=20, Nbits=Nbits)
net = net.to(device)
net.load_state_dict(torch.load("net_after_global_iterative_prune.pt"))
test(net)
# Quantized model finetuning
finetune(net, epochs=20, batch_size=256, lr=0.002, reg=1e-4)

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

Nbits	Accuracy Before Finetuning	Accuracy After Finetuning
5	87.78%	90.30%
4	86.26%	89.78%
3	72.72%	87.28%
2	10.00%	32.64%

Table 7: Test accuracy of pruned + quantized model before and after finetuning

Observations & Comparison with (c):

$N_{\textbf{bits}}$	(c) Accuracy	(d) Accuracy
5	91.59%	90.30%
4	91.35%	89.78%
3	90.64%	87.28%
2	85.92%	32.64%

Table 8: (c) Accuracy vs. (d) Accuracy

We compare the accuracy after quantization with and without pruning. From the results:

- Accuracy drops significantly when both pruning and low-bit quantization (e.g., 2-bit) are applied.
- Finetuning improves the performance considerably, especially for 3-bit and higher.
- Compared with results from Lab 3 (c), accuracy after finetuning is still lower when pruning is applied, showing that pruning introduces additional performance degradation.
- Nonetheless, for moderate bit-width (e.g., 4 or 5), finetuned models still achieve relatively high accuracy (close to 90%), indicating that pruning and quantization can be effectively combined.

5 Lab (4): Bonus

Code

```
# Lab4 (a), Your code here:
if symmetric:
    max_val = torch.max(torch.abs(w))
    ws = w / (2 * max_val) + 0.5  # scale to [0,1]
    step = 2 ** bit - 1
    R = torch.round(ws * step) / step
    wq = (R - 0.5) * (2 * max_val)
```

Nbits	**Asymmetric Accuracy**	**Symmetric Accuracy**
6	91.44%	91.34%
5	91.13%	90.71%
4	89.73%	85.31%
3	76.60%	71.51%
2	8.99%	10.00%

Table 9: Comparison between Asymmetric and Symmetric Quantization Accuracy

From Table 9, we observe that asymmetric quantization generally outperforms symmetric quantization across most bit-widths, especially at higher precision levels. While both methods maintain high accuracy at 6 and 5 bits, the performance gap becomes more evident as the number of bits decreases. For example, at 4 bits, asymmetric quantization achieves 89.73% while symmetric drops to 85.31%. At 3 bits, the gap remains large (76.60% vs. 71.51%). Interestingly, at 2 bits, symmetric performs slightly better, but both methods suffer from severe degradation. This indicates that symmetric quantization is more sensitive to reduced precision, likely due to its constraint of enforcing zero-centered quantization levels, which limits the expressiveness of the quantized values. On the other hand, asymmetric quantization allows a flexible range shift, better capturing the distribution of weights and achieving improved accuracy, especially under aggressive quantization.