# Report

a. Screenshots:

i. Testing accuracy of original dense vgg13 model.

A screenshot of a computer

Description automatically generated

ii. Results of testing accuracy and sparsity (test\_sparsity function) of your four pruned model.

Omp-unstructured-epochs 10

A computer screen shot of a black screen

Description automatically generated

Imp-unstructured-epochs 10

A computer screen shot of a black screen

Description automatically generated

Omp - filter -epochs 10

A screenshot of a computer screen

Description automatically generated

Imp - filter -epochs 10

A screenshot of a computer

Description automatically generated

iii. Your .yaml file.

prune\_ratios\_unstructured:

  # features.0.weight: 0.5  # First Conv Layer

  features.3.weight: 0.8   # conv layer

  features.4.weight: 0.8

  features.7.weight: 0.8

  features.8.weight: 0.8

  features.10.weight: 0.9

  features.11.weight: 0.9

  features.14.weight: 0.8

  features.15.weight: 0.8

  features.17.weight: 0.8

  features.18.weight: 0.8

  features.21.weight: 0.8

  features.22.weight: 0.8

  features.24.weight: 0.8

  features.25.weight: 0.8

  features.28.weight: 0.8

  features.29.weight: 0.8

  features.31.weight: 0.8

  features.32.weight: 0.8

prune\_ratios\_filter:

  # features.0.weight: 0.5  # First Conv Layer

  features.3.weight: 0.4

  features.4.weight: 0.4

  features.7.weight: 0.4

  features.8.weight: 0.6

  features.10.weight: 0.6

  features.11.weight: 0.4

  features.14.weight: 0.6

  features.15.weight: 0.4

  features.17.weight: 0.4

  features.18.weight: 0.4

  features.21.weight: 0.4

  features.22.weight: 0.4

  features.24.weight: 0.4

  features.25.weight: 0.4

  features.28.weight: 0.4

  features.29.weight: 0.4

  features.31.weight: 0.4

  features.32.weight: 0.4

b. Five (1+4) figures mentioned in Requirement 4.

Draw one figure to show the sparsity and accuracy comparison of the four pruned models. X-axis: overall sparsity ratio, Y-axis: accuracy.

A screenshot of a graph

Description automatically generated

b. Draw four figures to show the sparsity mask of one layer.

i. Arbitrary layer is fine.

ii. One figure for one model.

iii. Reshape the 4-D weights to 2-D format (each row represents all the weights from the same filter, number of columns = # of channels \* kernal\_hight \* kernal\_width)

iv. Use one color to represent non-zero weights, use other color to represent zero/pruned weights.

Omp-unstructured-epochs 10

A screenshot of a computer

Description automatically generated

Imp-unstructured-epochs 10

A screenshot of a computer

Description automatically generated

Omp - filter -epochs 10

A screenshot of a computer

Description automatically generated

Imp - filter -epochs 10

A screenshot of a computer

Description automatically generated

c. Answers of the questions listed in prune\_channels\_after\_filter\_prune() function.

A screenshot of a computer

Description automatically generated

1. After applying this function (further prune the corresponding channels), what is the change in sparsity?

Sparsity increased, from 41.5% to 78.4%

2. Will accuracy decrease, increase, or not change?

Accuracy decreased, went from 87.9% to 10.0%

3. Based on question 2, explain why?

The decrease in accuracy happens because pruning more channels reduces the capacity of the model to capture complex features. After fine-tuning, the model may have adapt to the reduced number of filters, but further reducing channels in the next layers may impair its ability to generalize.

4. Can we apply this function to ResNet and get the same conclusion? Why?

Yes, we can apply this function to ResNet. However, the results might vary. ResNet has residual connections that help mitigate the impact of removing filters by allowing the network to learn residual functions. The decrease in accuracy might be less pronounced compared to a plain convolutional network because of these skip connections. But, ultimately, the removal of channels still reduces the representational capacity of the network, and accuracy may decrease.

d. Link of your models (Google Drive or OneDrive).

Link for pruned models

<https://drive.google.com/drive/folders/1mX_jeLrcNvOwseXcKPHbFrxXzclX071N?usp=sharing>

e. Your code. (text or screenshot)

from \_\_future\_\_ import print\_function

import os

import sys

import logging

import argparse

import time

from time import strftime

import torch

import torch.optim as optim

import torch.nn as nn

import torch.nn.functional as F

from torchvision import datasets, transforms

import numpy as np

import yaml

import matplotlib.pyplot as plt

from matplotlib.colors import Normalize

from vgg\_cifar import vgg13

# settings

parser = argparse.ArgumentParser(description='PyTorch CIFAR10 admm training')

parser.add\_argument('--epochs', type=int, default=160, metavar='N',

                    help='number of epochs to train (default: 160)')

parser.add\_argument('--batch-size', type=int, default=64, metavar='N',

                    help='training batch size (default: 64)')

parser.add\_argument('--seed', type=int, default=1, metavar='S',

                    help='random seed (default: 1)')

parser.add\_argument('--load-model-path', type=str, default="./model/cifar10\_vgg13\_acc\_94.730.pt",

                    help='Path to pretrained model')

parser.add\_argument('--sparsity-type', type=str, default='unstructured',

                    help="define sparsity\_type: [unstructured, filter, etc.]")

parser.add\_argument('--sparsity-method', type=str, default='omp',

                    help="define sparsity\_method: [omp, imp, etc.]")

parser.add\_argument('--yaml-path', type=str, default="./vgg13.yaml",

                    help='Path to yaml file')

parser.add\_argument('--show-graph', type=bool, default=False,help='to show 2d graph of one conv layer')

parser.add\_argument('--prune-channels-after-filter-prune',type=bool,default=False,help='to prunes the corresponding channels in the next CONV layer')

args = parser.parse\_args()

# --- for dubeg use ---------

# args\_list = [

#     "--epochs", "160",

#     "--seed", "123",

#     # ... add other arguments and their values ...

# ]

# args = parser.parse\_args(args\_list)

def test(model, device, test\_loader):

    model.eval()

    test\_loss = 0

    correct = 0

    with torch.no\_grad():

        for data, target in test\_loader:

            data, target = data.to(device), target.to(device)

            output = model(data)

            test\_loss += F.nll\_loss(output, target, reduction='sum').item()  # sum up batch loss

            pred = output.max(1, keepdim=True)[1]  # get the index of the max log-probability

            correct += pred.eq(target.view\_as(pred)).sum().item()

    test\_loss /= len(test\_loader.dataset)

    accuracy = 100. \* float(correct) / float(len(test\_loader.dataset))

    print('\nTest set: Average loss: {:.4f}, Accuracy: {}/{} ({:.4f}%)\n'.format(

        test\_loss, correct, len(test\_loader.dataset), accuracy))

    return accuracy

def get\_dataloaders(args):

    train\_loader = torch.utils.data.DataLoader(

        datasets.CIFAR10('./data.cifar10', train=True, download=True,

                         transform=transforms.Compose([

                             transforms.Pad(4),

                             transforms.RandomCrop(32),

                             transforms.RandomHorizontalFlip(),

                             transforms.ToTensor(),

                             transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010))

                         ])),

        batch\_size=args.batch\_size, shuffle=True)

    test\_loader = torch.utils.data.DataLoader(

        datasets.CIFAR10('./data.cifar10', train=False, download=True,

                         transform=transforms.Compose([

                            transforms.ToTensor(),

                            transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010))

                        ])),

        batch\_size=256, shuffle=False)

    return train\_loader, test\_loader

# ============= the functions that you need to complete start from here =============

def visualize\_weights\_with\_gradient(weights, title\_prefix="Weights Visualization"):

    """

    Visualize the absolute values of weights with a color gradient for non-zero weights

    and red for zero weights.

    Args:

        weights (torch.Tensor): The 4D weight tensor of a layer (filters, channels, height, width).

        title\_prefix (str): Prefix for the figure title.

    """

    # Detach weights from computation graph and convert to NumPy

    filters, channels, height, width = weights.shape

    reshaped\_weights = weights.detach().view(filters, -1).cpu().numpy()  # Detach before converting to numpy

    # Use absolute values of weights for visualization

    abs\_weights = np.abs(reshaped\_weights)

    # Create a mask where all zeros are pruned weights and non-zero weights are highlighted

    weight\_mask = np.where(abs\_weights == 0, 0, abs\_weights)

    # Normalize weights (now using absolute values)

    norm = Normalize(vmin=np.min(abs\_weights), vmax=np.max(abs\_weights))

    # Plot the weights visualization

    plt.figure(figsize=(10, 6))

    # Zero weights are shown in red

    plt.imshow(weight\_mask, cmap="Reds", aspect="auto", interpolation="nearest")

    # Non-zero weights (absolute values) are visualized with a diverging colormap

    plt.imshow(abs\_weights, cmap="coolwarm", aspect="auto", alpha=0.5, interpolation="nearest", norm=norm)

    # Add a colorbar

    plt.colorbar(label="Weight Magnitude (Zero=pruned, Non-zero=gradient)")

    # Title and labels

    plt.title(f"{title\_prefix}")

    plt.xlabel("Channels × Kernel Height × Kernel Width")

    plt.ylabel("Filters")

    plt.show()

def save\_model(model, pruning\_type, sparsity, accuracy,sparsity\_type, save\_path="./"):

    """

    Saves the model with a specific naming convention.

    Args:

        model: The model to be saved.

        pruning\_type: Type of pruning (e.g., "omp", "imp").

        sparsity: Sparsity level as a float or string (e.g., 0.80 or "0.80").

        accuracy: Accuracy of the model as a float or string (e.g., 0.85 or "0.85").

        save\_path: The directory path where the model will be saved.

    """

    # Ensure sparsity and accuracy are floats

    sparsity = float(sparsity)

    accuracy = float(accuracy)

    # Create the filename based on the convention

    filename = f"{pruning\_type}\_{sparsity\_type}\_{sparsity:.2f}\_acc\_{accuracy:.3f}.pt"

    # Save the model state\_dict

    torch.save(model.state\_dict(), f"{save\_path}/{filename}")

    print(f"Model saved as: {filename}")

def read\_prune\_ratios\_from\_yaml(file\_name, model,sparsity\_type):

        """

            This function will read user-defined layer-wise target pruning ratios from yaml file.

            The ratios are stored in "prune\_ratio\_dict" dictionary,

            where the key is the layer name and value is the corresponding pruning ratio.

            Your task:

                Write a snippet of code to check if the layer names you provided in yaml file match the real layer name in the model.

                This can make sure your yaml file is correctly written.

        """

        if not isinstance(file\_name, str):

            raise Exception("filename must be a str")

        with open(file\_name, "r") as stream:

            try:

                raw\_dict = yaml.safe\_load(stream)

                prune\_ratio\_dict = {}

                if sparsity\_type=='unstructured':

                    prune\_ratio\_dict = raw\_dict['prune\_ratios\_unstructured']

                elif sparsity\_type=='filter':

                    prune\_ratio\_dict = raw\_dict['prune\_ratios\_filter']

                return prune\_ratio\_dict

            except yaml.YAMLError as exc:

                print(exc)

def unstructured\_prune(tensor: torch.Tensor, sparsity : float) -> torch.Tensor:

    """

    Implement magnitude-based unstructured pruning for weight tensor (of a layer)

    :param tensor: torch.(cuda.)Tensor, weight of conv/fc layer

    :param sparsity: float, pruning sparsity

    :return:

        torch.(cuda.)Tensor, pruning mask (1 for nonzeros, 0 for zeros)

    """

    ##################### YOUR CODE STARTS HERE #####################

    # Step 1: Calculate how many weights should be pruned

    # Step 2: Find the threshold of weight magnitude (th) based on sparsity.

    # Step 3: Get the pruning mask tensor based on the th. The mask tensor should have same shape as the weight tensor

    #         |weight| <= th -> mask=0,

    #         |weight| >  th -> mask=1

    # Step 4: Apply mask tensor to the weight tensor

    #         weight\_pruned = weight \* mask

    ##################### YOUR CODE ENDS HERE #######################

    # return the mask to record the pruning location ()

    num\_elements = tensor.numel()

    num\_pruned = int(num\_elements \* sparsity)

    num\_pruned = min(num\_pruned, num\_elements - 1)

    if num\_pruned == 0:

        return torch.ones\_like(tensor)

    flat\_tensor = tensor.view(-1)

    threshold = torch.kthvalue(torch.abs(flat\_tensor), num\_pruned)[0]

    mask = (torch.abs(tensor) > threshold).float()

    return mask

def filter\_prune(tensor: torch.Tensor, sparsity: float) -> torch.Tensor:

    """

    Implement L2-norm-based filter pruning for weight tensor (of a layer).

    Args:

        tensor (torch.Tensor): Weight of a convolutional layer (4D tensor).

        sparsity (float): Fraction of filters to prune (between 0 and 1).

    Returns:

        torch.Tensor: Pruning mask (1 for nonzeros, 0 for zeros).

    """

    # If tensor is not 4D, return a mask with all ones

    if tensor.ndim != 4:

        # print(f"Tensor is not 4D (shape: {tensor.shape}). Returning a mask with all ones.")

        return torch.ones\_like(tensor)

    # Step 1: Calculate how many filters should be pruned

    num\_filters = tensor.shape[0]  # Number of output channels (filters)

    num\_prune = int(num\_filters \* sparsity)  # Number of filters to prune

    # Step 2: Calculate L2 norm of each filter

    # Flatten the filter weights to calculate L2 norm per filter

    l2\_norms = torch.norm(tensor.view(num\_filters, -1), dim=1)

    # Step 3: Determine the pruning threshold based on the sparsity

    if num\_prune > 0:

        threshold = torch.topk(l2\_norms, num\_prune, largest=False).values[-1]

    else:

        threshold = 0.0

    # Step 4: Generate the pruning mask

    # Filters with L2 norm <= threshold will be pruned

    mask = (l2\_norms > threshold).float()  # Shape: [num\_filters]

    # Reshape the mask to match the weight tensor shape

    mask = mask.view(-1, 1, 1, 1)  # Shape: [num\_filters, 1, 1, 1]

    # Convert mask to the same device and data type as the input tensor

    mask = mask.to(dtype=tensor.dtype, device=tensor.device)

    # Return the mask to indicate the pruning locations

    return mask

def apply\_pruning(model, sparity\_type, prune\_ratio\_dict):

    # calculate layer\_wise prune ratio for current round (if IMP)

    # call unstructured\_prune()

    # or

    # call filter\_prune (...)

    print()

    # print("=========================layer names====================================")

    # for name, param in model.named\_parameters():

    #     print(f"{name}")

    #     # print(f"Shape: {param.shape}")

    # print("=========================layer names====================================")

    print()

    prune\_masks\_store = {}

    if sparity\_type =='unstructured':

        for layer\_name, tensor in prune\_ratio\_dict.items():

                # layer = dict(model.named\_parameters())[layer\_name]

                # print(layer\_name)

            pruning\_mask = unstructured\_prune(dict(model.named\_parameters())[layer\_name],tensor)

            prune\_masks\_store[layer\_name] = pruning\_mask

            # print(pruning\_mask)

            with torch.no\_grad():

                layer = dict(model.named\_parameters())[layer\_name]

                layer.data \*= pruning\_mask

        return model,prune\_masks\_store

    elif sparity\_type =='filter':

        for layer\_name, tensor in prune\_ratio\_dict.items():

                # layer = dict(model.named\_parameters())[layer\_name]

                # print(layer\_name)

            pruning\_mask = filter\_prune(dict(model.named\_parameters())[layer\_name],tensor)

            prune\_masks\_store[layer\_name] = pruning\_mask

            # print(pruning\_mask)

            with torch.no\_grad():

                layer = dict(model.named\_parameters())[layer\_name]

                layer.data \*= pruning\_mask

        return model,prune\_masks\_store

    else:

        raise ValueError("Invalid sparsity type. Only 'unstructured' and 'filter' are supported.")

def test\_sparity(model, sparsity\_type):

    """

    Check the sparsity of a model.

    Args:

        model: The PyTorch model to analyze.

        sparsity\_type: "unstructured" or "filter", denoting the type of sparsity.

    """

    print(f"Sparsity type is: {sparsity\_type}")

    total\_zeros = 0

    total\_nonzeros = 0

    total\_filters = 0

    empty\_filters = 0

    for name, param in model.named\_parameters():

        if "weight" in name and param.requires\_grad:

            if sparsity\_type == "unstructured":

                # Count zero and non-zero elements in the weight tensor

                zeros = torch.sum(param == 0).item()

                total = param.numel()

                sparsity = zeros / total \* 100

                print(f"(zero/total) weights of {name} is: ({zeros}/{total}). Sparsity is: {sparsity:.2f}%")

                total\_zeros += zeros

                total\_nonzeros += total - zeros

            elif sparsity\_type == "filter":

                # Check filter sparsity (assume param is 4D: [out\_channels, in\_channels, h, w])

                if param.dim() == 4:  # Only consider convolutional filters

                    filters = param.shape[0]

                    empty = 0

                    for i in range(filters):

                        if torch.sum(param[i]) == 0:

                            empty += 1

                    total\_filters += filters

                    empty\_filters += empty

                    sparsity = empty / filters \* 100

                    print(f"(empty/total) filter of {name} is: ({empty}/{filters}). Filter sparsity is: {sparsity:.2f}%")

    print("---------------------------------------------------------------------------")

    if sparsity\_type == "unstructured":

        total\_params = total\_zeros + total\_nonzeros

        overall\_sparsity = total\_zeros / total\_params \* 100

        print(f"Total number of zeros: {total\_zeros}, non-zeros: {total\_nonzeros}, overall sparsity is: {overall\_sparsity:.4f}%")

        return overall\_sparsity

    elif sparsity\_type == "filter":

        overall\_filter\_sparsity = empty\_filters / total\_filters \* 100

        print(f"Total number of filters: {total\_filters}, empty-filters: {empty\_filters}, overall filter sparsity is: {overall\_filter\_sparsity:.4f}%")

        return overall\_filter\_sparsity

def masked\_retrain(model, prune\_masks, optimizer, loss\_fn, data\_loader, test\_data\_loader, num\_epochs=1, device='cuda'):

    """

    Fine-tune the pruned model, updating only the unpruned weights, and print the accuracy after each epoch on both

    the training and test data.

    :param model: nn.Module, the pruned model to fine-tune

    :param prune\_masks: dict, dictionary containing the pruning mask for each layer

    :param optimizer: optimizer, optimizer for training the model

    :param loss\_fn: loss function, criterion to calculate the loss

    :param data\_loader: data loader, provides batches of input data and targets

    :param test\_data\_loader: test data loader, provides batches of test input data and targets

    :param num\_epochs: int, number of epochs to retrain the model

    :param device: str, device to use for training ('cuda' or 'cpu')

    """

    # Move the model to the specified device (e.g., 'cuda' or 'cpu')

    model.to(device)

    for epoch in range(num\_epochs):

        if epoch == 7:

            for param\_group in optimizer.param\_groups:

                param\_group['lr'] = 0.01

        # Training phase

        model.train()  # Set model to training mode

        total\_correct\_train = 0

        total\_samples\_train = 0

        epoch\_loss\_train = 0

        for batch\_idx, (inputs, targets) in enumerate(data\_loader):

            inputs, targets = inputs.to(device), targets.to(device)

            # Forward pass

            outputs = model(inputs)

            loss = loss\_fn(outputs, targets)

            # Compute accuracy for the current batch

            \_, predicted = torch.max(outputs, 1)

            total\_correct\_train += (predicted == targets).sum().item()

            total\_samples\_train += targets.size(0)

            epoch\_loss\_train += loss.item()

            # Backward pass and optimization

            optimizer.zero\_grad()

            loss.backward()

            optimizer.step()

            # Reapply the pruning mask to keep pruned weights at zero

            with torch.no\_grad():

                for layer\_name, mask in prune\_masks.items():

                    layer = dict(model.named\_parameters())[layer\_name]

                    layer.data \*= mask

        # Calculate training accuracy and average loss

        train\_accuracy = (total\_correct\_train / total\_samples\_train) \* 100

        avg\_train\_loss = epoch\_loss\_train / len(data\_loader)

        # Evaluation phase on test data

        model.eval()  # Set model to evaluation mode

        total\_correct\_test = 0

        total\_samples\_test = 0

        epoch\_loss\_test = 0

        with torch.no\_grad():

            for inputs, targets in test\_data\_loader:

                inputs, targets = inputs.to(device), targets.to(device)

                # Forward pass

                outputs = model(inputs)

                loss = loss\_fn(outputs, targets)

                # Compute accuracy for the test batch

                \_, predicted = torch.max(outputs, 1)

                total\_correct\_test += (predicted == targets).sum().item()

                total\_samples\_test += targets.size(0)

                epoch\_loss\_test += loss.item()

        # Calculate test accuracy and average loss

        test\_accuracy = (total\_correct\_test / total\_samples\_test) \* 100

        avg\_test\_loss = epoch\_loss\_test / len(test\_data\_loader)

        # Print loss and accuracy for each epoch on both training and test data

        print(f"Epoch [{epoch + 1}/{num\_epochs}]")

        print(f"    Training Loss: {avg\_train\_loss:.4f}, Training Accuracy: {train\_accuracy:.2f}%")

        print(f"    Test Loss: {avg\_test\_loss:.4f}, Test Accuracy: {test\_accuracy:.2f}%")

    print()

    print()

    return model

def oneshot\_magnitude\_prune(model, sparsity\_type, prune\_ratio\_dict,train\_loader,test\_loader,optimizer,loss\_fn,epochs,show\_graph):

    model,prune\_masks=apply\_pruning(model, sparsity\_type, prune\_ratio\_dict)

    model=masked\_retrain(model, prune\_masks, optimizer, loss\_fn, train\_loader,test\_loader, epochs)

    for layer\_name, tensor in prune\_ratio\_dict.items():

        if show\_graph and layer\_name=='features.3.weight':

            visualize\_weights\_with\_gradient(dict(model.named\_parameters())[layer\_name], title\_prefix="Weight Visualization with Gradient")

    sparsity=test\_sparity(model, sparsity\_type)

    # masked\_retrain()

    use\_cuda = torch.cuda.is\_available()

    device = torch.device("cuda" if use\_cuda else "cpu")

    accuracy=test(model, device, test\_loader)

    save\_model(model, 'omp', sparsity, accuracy,sparsity\_type)

    # Implement the function that conducting oneshot magnitude pruning

    # Target sparsity ratio dict should contains the sparsity ratio of each layer

    # the per-layer sparsity ratio should be read from a external .yaml file

    # This function should also include the masked\_retrain() function to conduct fine-tuning to restore the accuracy

    return model

def iterative\_magnitude\_prune(model, sparsity\_type, target\_sparsity\_dict, train\_loader, test\_loader, optimizer, loss\_fn, epochs,show\_graph, use\_cuda=False):

    # Iterative pruning: start with a low sparsity and increase progressively

    current\_sparsity\_dict = {k: 0.0 for k in target\_sparsity\_dict}  # Initial sparsity is 0% for all layers

    num\_iterations = 4 # One iteration per layer

    sparsity, accuracy=-1,-1

    for i in range(num\_iterations):

        # Gradually increase the sparsity level for each layer

        for layer, target\_sparsity in target\_sparsity\_dict.items():

            # Increase sparsity by a certain ratio in each iteration (this could be done progressively)

            current\_sparsity\_dict[layer] = min(current\_sparsity\_dict[layer] + (target\_sparsity / num\_iterations), target\_sparsity)

        # Apply pruning based on current sparsity ratio for each layer

        model, prune\_masks = apply\_pruning(model, sparsity\_type, current\_sparsity\_dict)

        # Retrain the model with the updated pruning mask

        model=masked\_retrain(model, prune\_masks, optimizer, loss\_fn, train\_loader, test\_loader, epochs)

        # Evaluate the model after retraining

        device = torch.device("cuda" if use\_cuda else "cpu")

        #test(model, device, test\_loader)

        # Optionally, print the current sparsity for each layer after pruning

        print(f"Iteration {i + 1}: Sparsity updated to {current\_sparsity\_dict}")

    sparsity=test\_sparity(model, sparsity\_type)

    for layer\_name, tensor in target\_sparsity\_dict.items():

        if show\_graph and layer\_name=='features.3.weight':

            visualize\_weights\_with\_gradient(dict(model.named\_parameters())[layer\_name], title\_prefix="Weight Visualization with Gradient")

    # Final test to evaluate the model after all iterations

    print("Final model evaluation:")

    use\_cuda = torch.cuda.is\_available()

    device = torch.device("cuda" if use\_cuda else "cpu")

    accuracy=test(model, device, test\_loader)

    save\_model(model, 'imp', sparsity, accuracy,sparsity\_type)

    return model

def prune\_channels\_after\_filter\_prune(pruned\_model):

    """

    Prunes the weights in the next convolutional layers by setting to zero the weights

    corresponding to the pruned filters in the current convolutional layer.

    This keeps the dimensions intact without changing the number of channels in the layers.

    """

    layers = list(pruned\_model.children())  # Get layers from the model

    # Iterate through layers

    # count=1

    for i, layer in enumerate(layers):

        # Check if the current layer is a Sequential block

        if isinstance(layer, nn.Sequential):

            sequential\_layers = list(layer.children())

            for j, sublayer in enumerate(sequential\_layers):

                if isinstance(sublayer, nn.Conv2d):

                    # Identify pruned filters in the current Conv2d layer

                    weights = sublayer.weight.data  # Shape: (out\_channels, in\_channels, kernel\_h, kernel\_w)

                    pruned\_indices = [

                        idx for idx in range(weights.size(0)) if torch.all(weights[idx] == 0)

                    ]

                    if pruned\_indices:

                        # For each Conv2d layer after the current one, set corresponding weights to zero

                        for k in range(j + 1, len(sequential\_layers)):

                            next\_layer = sequential\_layers[k]

                            if isinstance(next\_layer, nn.Conv2d):

                                # Next layer's weights

                                next\_weights = next\_layer.weight.data  # Shape: (out\_channels, in\_channels, kernel\_h, kernel\_w)

                                # weights2 = next\_layer.weight.data  # Shape: (out\_channels, in\_channels, kernel\_h, kernel\_w)

                                # pruned\_indices2 = [

                                #         idx for idx in range(weights2.size(0)) if torch.all(weights2[idx] == 0)

                                # ]

                                # Set the weights corresponding to the pruned channels to zero

                                mask = torch.ones\_like(next\_weights)

                                for pruned\_idx in pruned\_indices:

                                    mask[pruned\_idx, :, :, :] = 0  # Set pruned channels to zero in the mask

                                next\_weights \*= mask  # Set pruned channels to zero

                                # Update the next Conv2d layer weights

                                next\_layer.weight.data = next\_weights

                                # if count==1:

                                #     print(pruned\_indices)

                                #     weights1 = next\_layer.weight.data  # Shape: (out\_channels, in\_channels, kernel\_h, kernel\_w)

                                #     pruned\_indices1 = [

                                #         idx for idx in range(weights1.size(0)) if torch.all(weights1[idx] == 0)

                                #     ]

                                    # print(weights1)

                                    # print(next\_layer.weight.data)

                                    # print(pruned\_indices2)

                                    # print(pruned\_indices1)

                                    # print(f"Filter at Position 4: {next\_weights[4]}")  # 1st filter

                                    # count+=1

                                break

    return pruned\_model

def main():

    use\_cuda = torch.cuda.is\_available()

    device = torch.device("cuda" if use\_cuda else "cpu")

    print(device)

    # setup random seed

    np.random.seed(args.seed)

    torch.manual\_seed(args.seed)

    if use\_cuda:

        torch.cuda.manual\_seed(args.seed)

    torch.backends.cudnn.deterministic = True

    torch.backends.cudnn.benchmark = False

    # set up model archetecture and load pretrained dense model

    model = vgg13()

    model.load\_state\_dict(torch.load(args.load\_model\_path))

    if use\_cuda:

        model.cuda()

    train\_loader, test\_loader = get\_dataloaders(args)

    # Select loss function. You may change to whatever loss function you want.

    criterion = nn.CrossEntropyLoss()

    # Select optimizer. You may change to whatever optimizer you want.

    optimizer = torch.optim.SGD(model.parameters(), lr=0.1, momentum=0.9, weight\_decay=1e-4)

    # you may use this lr scheduler to fine-tune/mask-retrain your pruned model.

    scheduler = optim.lr\_scheduler.CosineAnnealingLR(optimizer, T\_max=args.epochs \* len(train\_loader), eta\_min=4e-08)

    # ========= your code starts here ========

    # ...

    # pruning\_process()

    # masked\_retrain()

    # ...

    # ---- you can test your model accuracy and sparity using the following fuction ---------

    # test\_sparity()

    # test(model, device, test\_loader)

    # ========================================

    print()

    print("+++++++++++++++++++++++++++++++ Example Commands ++++++++++++++++++++++++++++++++++++++++++++")

    print("python main.py --sparsity-method omp --sparsity-type unstructured --epochs 10")

    print("python main.py --sparsity-method omp --sparsity-type filter --epochs 10")

    print("python main.py --sparsity-method imp --sparsity-type unstructured --epochs 10")

    print("python main.py --sparsity-method imp --sparsity-type filter --epochs 10")

    print()

    print("python main.py --sparsity-method omp --sparsity-type unstructured --epochs 10 --show-graph True")

    print("python main.py --sparsity-method omp --sparsity-type filter --epochs 10 --prune-channels-after-filter-prune True")

    print("+++++++++++++++++++++++++++++++ Example Commands ++++++++++++++++++++++++++++++++++++++++++++")

    prune\_ratio\_dict=read\_prune\_ratios\_from\_yaml(args.yaml\_path,args.load\_model\_path,args.sparsity\_type)

    print()

    print("=========================================loaded yaml dictonary===========================================================")

    print(args.sparsity\_type,prune\_ratio\_dict)

    print("=========================================loaded yaml dictonary===========================================================")

    print()

    # test(model, device, test\_loader)

    if args.sparsity\_method =='omp':

        model=oneshot\_magnitude\_prune(model, args.sparsity\_type, prune\_ratio\_dict,train\_loader,test\_loader,optimizer,criterion,args.epochs,args.show\_graph)

    elif args.sparsity\_method == 'imp':

        model=iterative\_magnitude\_prune(model, args.sparsity\_type, prune\_ratio\_dict, train\_loader, test\_loader, optimizer, criterion, args.epochs,args.show\_graph, device)

    else:

        print("Invalid sparsity method. Choose either 'omp' or 'imp'.")

    if args.prune\_channels\_after\_filter\_prune:

        print()

        print("+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++")

        test(model, device, test\_loader)

        test\_sparity(model, args.sparsity\_type)

        pruned\_model = prune\_channels\_after\_filter\_prune(model)

        print()

        print()

        print()

        print("Final model evaluation: after prune\_channels\_after\_filter\_prune")

        test(pruned\_model, device, test\_loader)

        test\_sparity(pruned\_model, args.sparsity\_type)

if \_\_name\_\_ == '\_\_main\_\_':

    main()