Report

a. Screenshots:

i. Testing accuracy of original dense vgg13 model.

```
G VISHAL@DESKTOP-20BA5T1 MINGW64 ~/OneDrive/Desktop/temp/pruningi/HW2_pruning/git/VGG13_Pruning (main)
$ python main.py --sparsity-method omp --sparsity-type filter --epochs 10 --show-graph True
cuda
C:\Users\G VISHAL\OneDrive\Desktop\temp\pruningi\HW2_pruning\git\VGG13_Pruning\main.py:573: FutureWarning: You
itly. It is possible to construct malicious pickle data which will execute arbitrary code during unpickling (Se
efault value for `weights_only` will be flipped to `True`. This limits the functions that could be executed du
sted by the user via `torch.serialization.add_safe_globals`. We recommend you start setting `weights_only=True`
ated to this experimental feature.
  model.load_state_dict(torch.load(args.load_model_path))
Files already downloaded and verified
Files already downloaded and verified

=========================================loaded yaml dictonary============================================
filter {'features.3.weight': 0.4, 'features.4.weight': 0.4, 'features.7.weight': 0.4, 'features.8.weight': 0.6
.weight': 0.4, 'features.18.weight': 0.4, 'features.21.weight': 0.4, 'features.22.weight': 0.4, 'features.24.we
features.32.weight': 0.4}
=========================================loaded yaml dictonary============================================


Test set: Average loss: -3.2496, Accuracy: 9473/10000 (94.7300%)
```

ii. Results of testing accuracy and sparsity (test_sparsity function) of your four pruned model.

Omp-unstructured-epochs 10

```
Sparsity type is: unstructured
(zero/total) weights of features.0.weight is: (0/1728). Sparsity is: 0.00%
(zero/total) weights of features.1.weight is: (0/64). Sparsity is: 0.00%
(zero/total) weights of features.3.weight is: (29491/36864). Sparsity is: 80.00%
(zero/total) weights of features.4.weight is: (51/64). Sparsity is: 79.69%
(zero/total) weights of features.7.weight is: (58982/73728). Sparsity is: 80.00%
(zero/total) weights of features.8.weight is: (102/128). Sparsity is: 79.69%
(zero/total) weights of features.10.weight is: (132710/147456). Sparsity is: 90.00%
(zero/total) weights of features.11.weight is: (115/128). Sparsity is: 89.84%
(zero/total) weights of features.14.weight is: (235929/294912). Sparsity is: 80.00%
(zero/total) weights of features.15.weight is: (204/256). Sparsity is: 79.69%
(zero/total) weights of features.17.weight is: (471859/589824). Sparsity is: 80.00%
(zero/total) weights of features.18.weight is: (204/256). Sparsity is: 79.69%
(zero/total) weights of features.21.weight is: (943718/1179648). Sparsity is: 80.00%
(zero/total) weights of features.22.weight is: (409/512). Sparsity is: 79.88%
(zero/total) weights of features.24.weight is: (1887436/2359296). Sparsity is: 80.00%
(zero/total) weights of features.25.weight is: (409/512). Sparsity is: 79.88%
(zero/total) weights of features.28.weight is: (1887436/2359296). Sparsity is: 80.00%
(zero/total) weights of features.29.weight is: (409/512). Sparsity is: 79.88%
(zero/total) weights of features.31.weight is: (1887436/2359296). Sparsity is: 80.00%
(zero/total) weights of features.32.weight is: (409/512). Sparsity is: 79.88%
(zero/total) weights of classifier.weight is: (0/5120). Sparsity is: 0.00%
---------------------------------------------------------------------------
Total number of zeros: 7537309, non-zeros: 1872803, overall sparsity is: 80.0980%

Test set: Average loss: -5.6488, Accuracy: 7700/10000 (77.0000%)

Model saved as: omp_unstructured_80.10_acc_77.000.pt
```

Imp-unstructured-epochs 10

```
Iteration 4: Sparsity updated to {'features.3.weight': 0.8, 'features.4.weight': 0.8, 'features.7.weight'
weight': 0.8, 'features.17.weight': 0.8, 'features.18.weight': 0.8, 'features.21.weight': 0.8, 'features.
eatures.31.weight': 0.8, 'features.32.weight': 0.8}
Sparsity type is: unstructured
(zero/total) weights of features.0.weight is: (0/1728). Sparsity is: 0.00%
(zero/total) weights of features.1.weight is: (0/64). Sparsity is: 0.00%
(zero/total) weights of features.3.weight is: (29491/36864). Sparsity is: 80.00%
(zero/total) weights of features.4.weight is: (51/64). Sparsity is: 79.69%
(zero/total) weights of features.7.weight is: (58982/73728). Sparsity is: 80.00%
(zero/total) weights of features.8.weight is: (102/128). Sparsity is: 79.69%
(zero/total) weights of features.10.weight is: (132710/147456). Sparsity is: 90.00%
(zero/total) weights of features.11.weight is: (115/128). Sparsity is: 89.84%
(zero/total) weights of features.14.weight is: (235929/294912). Sparsity is: 80.00%
(zero/total) weights of features.15.weight is: (204/256). Sparsity is: 79.69%
(zero/total) weights of features.17.weight is: (471859/589824). Sparsity is: 80.00%
(zero/total) weights of features.18.weight is: (204/256). Sparsity is: 79.69%
(zero/total) weights of features.21.weight is: (943718/1179648). Sparsity is: 80.00%
(zero/total) weights of features.22.weight is: (409/512). Sparsity is: 79.88%
(zero/total) weights of features.24.weight is: (1887436/2359296). Sparsity is: 80.00%
(zero/total) weights of features.25.weight is: (409/512). Sparsity is: 79.88%
(zero/total) weights of features.28.weight is: (1887436/2359296). Sparsity is: 80.00%
(zero/total) weights of features.29.weight is: (409/512). Sparsity is: 79.88%
(zero/total) weights of features.31.weight is: (1887436/2359296). Sparsity is: 80.00%
(zero/total) weights of features.32.weight is: (409/512). Sparsity is: 79.88%
(zero/total) weights of classifier.weight is: (0/5120). Sparsity is: 0.00%
---------------------------------------------------------------------------
Total number of zeros: 7537309, non-zeros: 1872803, overall sparsity is: 80.0980%
Final model evaluation:

Test set: Average loss: -6.2644, Accuracy: 8164/10000 (81.6400%)

Model saved as: imp_unstructured_80.10_acc_81.640.pt

G VISHAL@DESKTOP-20BA5T1 MINGW64 ~/OneDrive/Desktop/temp/pruningi/HW2_pruning/git/VGG13_Pruning (main)
```

Omp - filter -epochs 10

```
Sparsity type is: filter
(empty/total) filter of features.0.weight is: (0/64). Filter sparsity is: 0.00%
(empty/total) filter of features.3.weight is: (25/64). Filter sparsity is: 39.06%
(empty/total) filter of features.7.weight is: (51/128). Filter sparsity is: 39.84%
(empty/total) filter of features.10.weight is: (76/128). Filter sparsity is: 59.38%
(empty/total) filter of features.14.weight is: (153/256). Filter sparsity is: 59.77%
(empty/total) filter of features.17.weight is: (102/256). Filter sparsity is: 39.84%
(empty/total) filter of features.21.weight is: (204/512). Filter sparsity is: 39.84%
(empty/total) filter of features.24.weight is: (204/512). Filter sparsity is: 39.84%
(empty/total) filter of features.28.weight is: (204/512). Filter sparsity is: 39.84%
(empty/total) filter of features.31.weight is: (204/512). Filter sparsity is: 39.84%
---------------------------------------------------------------------------
Total number of filters: 2944, empty-filters: 1223, overall filter sparsity is: 41.5421%

Test set: Average loss: -7.8062, Accuracy: 8799/10000 (87.9900%)

Model saved as: omp_filter_41.54_acc_87.990.pt

G VISHAL@DESKTOP-20BA5T1 MINGW64 ~/OneDrive/Desktop/temp/pruningi/HW2_pruning/git/VGG13_Pruning (main)
```

Imp - filter -epochs 10

```
Iteration 4: Sparsity updated to {'features.3.weight': 0.4, 'features.4.weight': 0.4, 'features.7.weight'
weight': 0.4, 'features.17.weight': 0.4, 'features.18.weight': 0.4, 'features.21.weight': 0.4, 'features.
eatures.31.weight': 0.4, 'features.32.weight': 0.4}
Sparsity type is: filter
(empty/total) filter of features.0.weight is: (0/64). Filter sparsity is: 0.00%
(empty/total) filter of features.3.weight is: (25/64). Filter sparsity is: 39.06%
(empty/total) filter of features.7.weight is: (51/128). Filter sparsity is: 39.84%
(empty/total) filter of features.10.weight is: (76/128). Filter sparsity is: 59.38%
(empty/total) filter of features.14.weight is: (153/256). Filter sparsity is: 59.77%
(empty/total) filter of features.17.weight is: (102/256). Filter sparsity is: 39.84%
(empty/total) filter of features.21.weight is: (204/512). Filter sparsity is: 39.84%
(empty/total) filter of features.24.weight is: (204/512). Filter sparsity is: 39.84%
(empty/total) filter of features.28.weight is: (204/512). Filter sparsity is: 39.84%
(empty/total) filter of features.31.weight is: (204/512). Filter sparsity is: 39.84%
--------------------------------------------------------------------------
Total number of filters: 2944, empty-filters: 1223, overall filter sparsity is: 41.5421%
Final model evaluation:

Test set: Average loss: -9.3342, Accuracy: 9095/10000 (90.9500%)

Model saved as: imp_filter_41.54_acc_90.950.pt

G VISHAL@DESKTOP-20BA5T1 MINGW64 ~/OneDrive/Desktop/temp/pruningi/HW2_pruning/git/VGG13_Pruning (main)
```

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

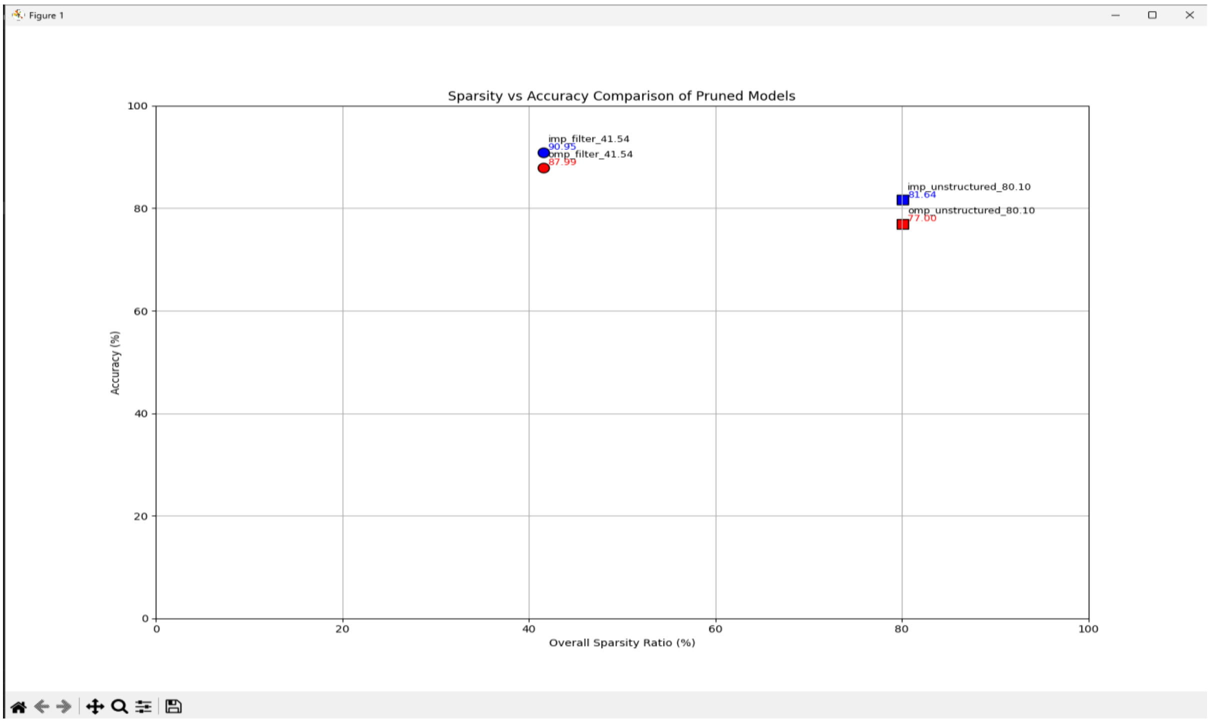
features.29.weight: 0.4

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.



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

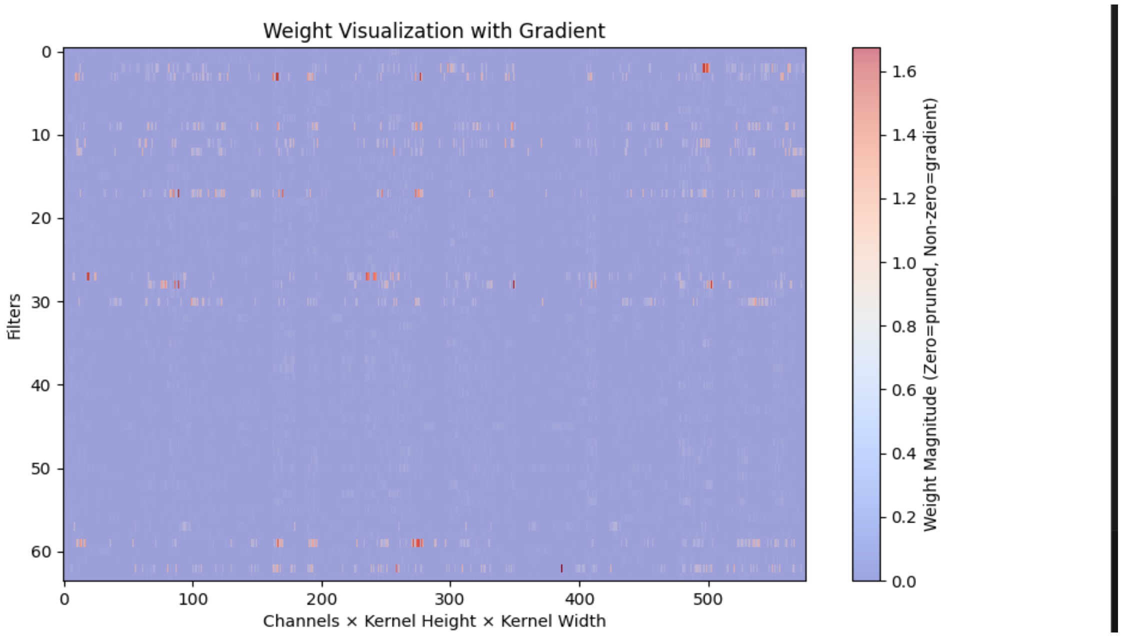
i. Arbitrary layer is fine.

ii. One figure for one model.

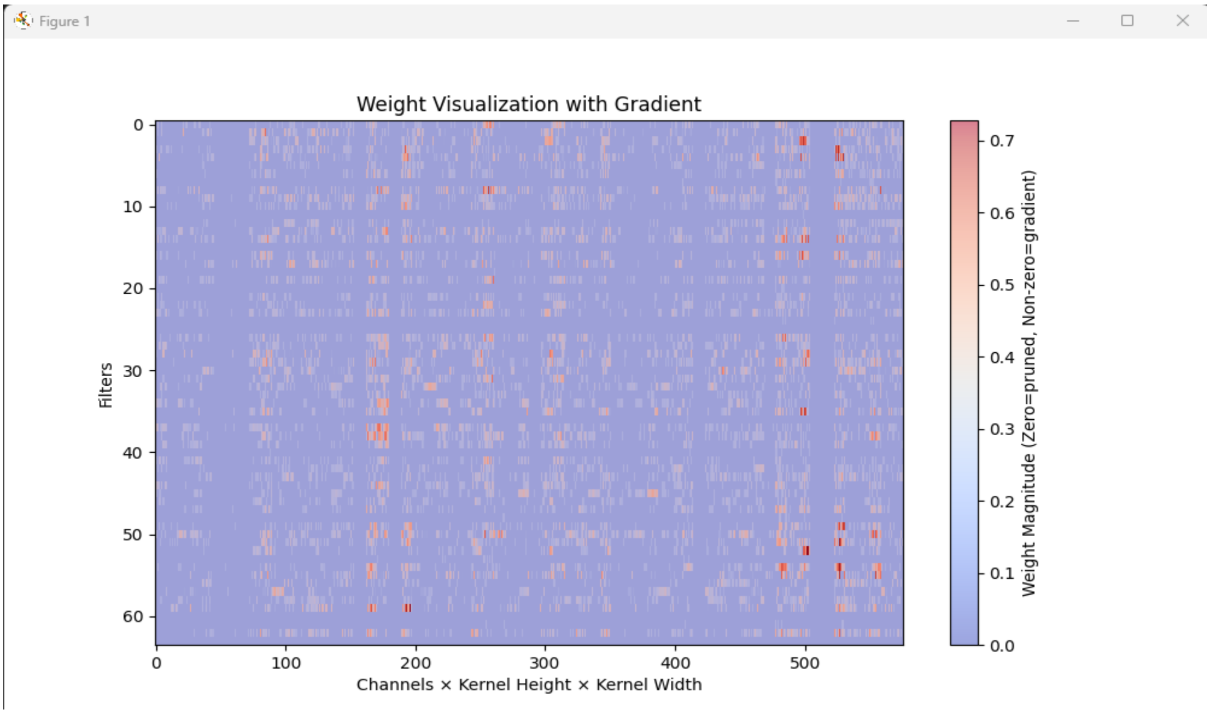
iii. Reshape the 4-D weights to 2-D format (each row represents all the weights from the same filter, number of columns = # of channels * kernal_hight * kernal_width)

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

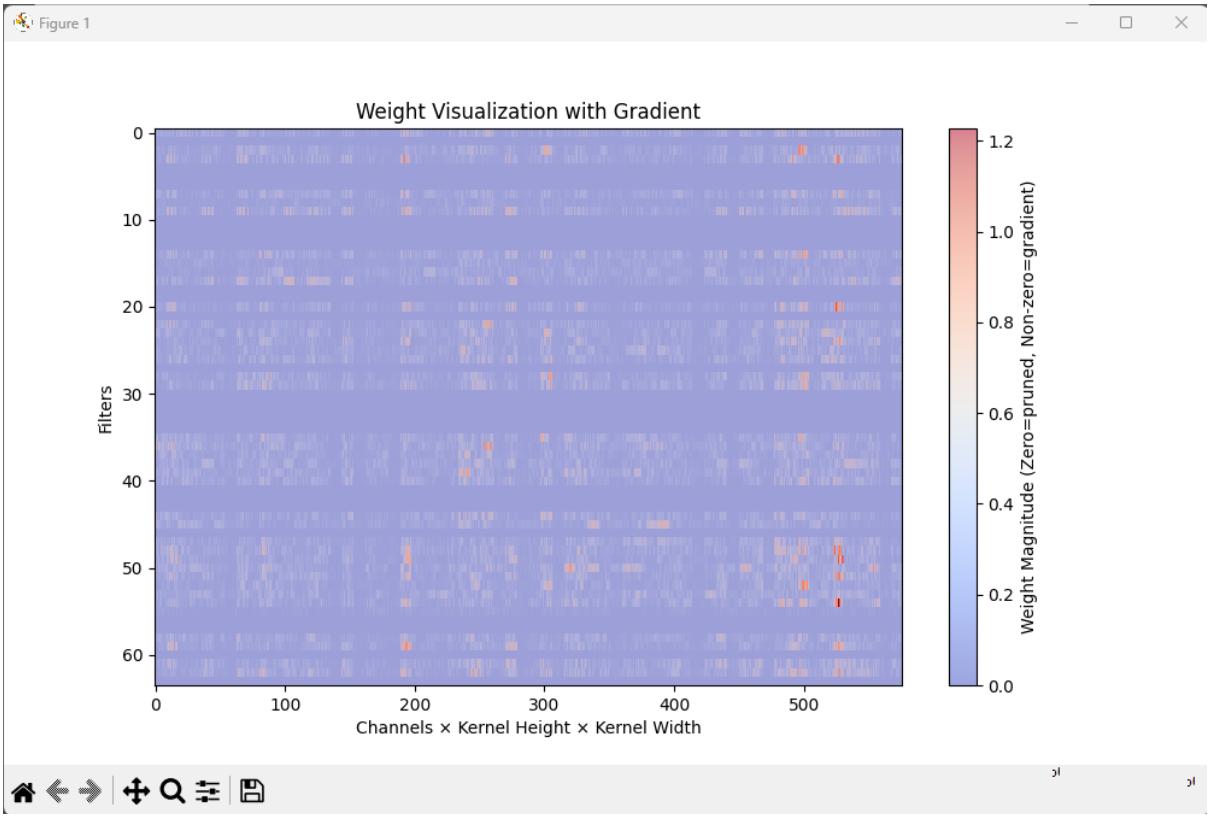
Omp-unstructured-epochs 10



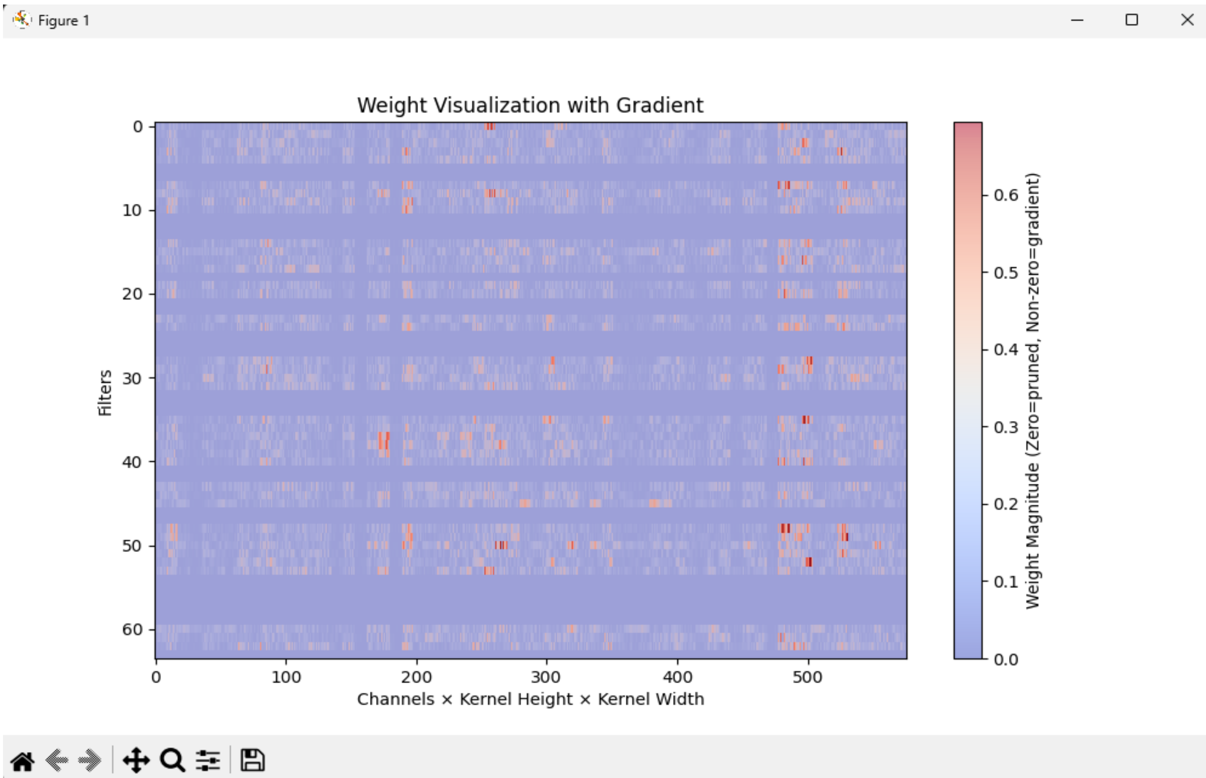
Imp-unstructured-epochs 10



Omp - filter -epochs 10



Imp - filter -epochs 10



c. Answers of the questions listed in prune_channels_after_filter_prune() function.

```
Model saved as: omp_filter_41.54_acc_87.990.pt

++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

Test set: Average loss: -7.8062, Accuracy: 8799/10000 (87.9900%)

Sparsity type is: filter
(empty/total) filter of features.0.weight is: (0/64). Filter sparsity is: 0.00%
(empty/total) filter of features.3.weight is: (25/64). Filter sparsity is: 39.06%
(empty/total) filter of features.7.weight is: (51/128). Filter sparsity is: 39.84%
(empty/total) filter of features.10.weight is: (76/128). Filter sparsity is: 59.38%
(empty/total) filter of features.14.weight is: (153/256). Filter sparsity is: 59.77%
(empty/total) filter of features.17.weight is: (102/256). Filter sparsity is: 39.84%
(empty/total) filter of features.21.weight is: (204/512). Filter sparsity is: 39.84%
(empty/total) filter of features.24.weight is: (204/512). Filter sparsity is: 39.84%
(empty/total) filter of features.28.weight is: (204/512). Filter sparsity is: 39.84%
(empty/total) filter of features.31.weight is: (204/512). Filter sparsity is: 39.84%
---------------------------------------------------------------------------
Total number of filters: 2944, empty-filters: 1223, overall filter sparsity is: 41.5421%



Final model evaluation: after prune_channels_after_filter_prune

Test set: Average loss: -0.0003, Accuracy: 1000/10000 (10.0000%)

Sparsity type is: filter
(empty/total) filter of features.0.weight is: (0/64). Filter sparsity is: 0.00%
(empty/total) filter of features.3.weight is: (25/64). Filter sparsity is: 39.06%
(empty/total) filter of features.7.weight is: (68/128). Filter sparsity is: 53.12%
(empty/total) filter of features.10.weight is: (106/128). Filter sparsity is: 82.81%
(empty/total) filter of features.14.weight is: (206/256). Filter sparsity is: 80.47%
(empty/total) filter of features.17.weight is: (227/256). Filter sparsity is: 88.67%
(empty/total) filter of features.21.weight is: (342/512). Filter sparsity is: 66.80%
(empty/total) filter of features.24.weight is: (411/512). Filter sparsity is: 80.27%
(empty/total) filter of features.28.weight is: (455/512). Filter sparsity is: 88.87%
(empty/total) filter of features.31.weight is: (471/512). Filter sparsity is: 91.99%
---------------------------------------------------------------------------
Total number of filters: 2944, empty-filters: 2311, overall filter sparsity is: 78.4986%

G VISHAL@DESKTOP-20BA5T1 MINGW64 ~/OneDrive/Desktop/temp/pruningi/HW2_pruning/git/VGG13_Pruning (main)
```

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
```

```python
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()
```