resnet_alpha1

December 6, 2024

[1]:
```
import argparse
import os
import time
import shutil

import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
import torch.backends.cudnn as cudnn


import torchvision
import torchvision.transforms as transforms

import torch
import torch.nn as nn
import math
from models.quant_layer import *


def conv3x3(in_planes, out_planes, stride=1):
    " 3x3 convolution with padding "
    return nn.Conv2d(in_planes, out_planes, kernel_size=3, stride=stride,␣
 ↪padding=1, bias=False)


def Quantconv3x3(in_planes, out_planes, stride=1):
    " 3x3 quantized convolution with padding "
    return QuantConv2d(in_planes, out_planes, kernel_size=3, stride=stride,␣
 ↪padding=1, bias=False)


class BasicBlock(nn.Module):
    expansion=1
```

```
    def __init__(self, inplanes, planes, stride=1, downsample=None,␣
 ↪float=False):
        super(BasicBlock, self).__init__()
        if float:
            self.conv1 = conv3x3(inplanes, planes, stride)
            self.conv2 = conv3x3(planes, planes)
        else:
            self.conv1 = Quantconv3x3(inplanes, planes, stride)
            self.conv2 = Quantconv3x3(planes, planes)
        self.bn1 = nn.BatchNorm2d(planes)
        self.relu = nn.ReLU(inplace=True)
        self.bn2 = nn.BatchNorm2d(planes)
        self.downsample = downsample
        self.stride = stride

    def forward(self, x):
        residual = x
        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)

        out = self.conv2(out)
        out = self.bn2(out)

        if self.downsample is not None:
            residual = self.downsample(x)

        out += residual
        out = self.relu(out)

        return out


class Bottleneck(nn.Module):
    expansion=4

    def __init__(self, inplanes, planes, stride=1, downsample=None):
        super(Bottleneck, self).__init__()
        self.conv1 = nn.Conv2d(inplanes, planes, kernel_size=1, bias=False)
        self.bn1 = nn.BatchNorm2d(planes)
        self.conv2 = nn.Conv2d(planes, planes, kernel_size=3, stride=stride,␣
 ↪padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(planes)
        self.conv3 = nn.Conv2d(planes, planes*4, kernel_size=1, bias=False)
        self.bn3 = nn.BatchNorm2d(planes*4)
        self.relu = nn.ReLU(inplace=True)
        self.downsample = downsample
```

```
        self.stride = stride

    def forward(self, x):
        residual = x

        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)

        out = self.conv2(out)
        out = self.bn2(out)
        out = self.relu(out)

        out = self.conv3(out)
        out = self.bn3(out)

        if self.downsample is not None:
            residual = self.downsample(x)

        out += residual
        out = self.relu(out)

        return out


class ResNet_Cifar(nn.Module):

    def __init__(self, block, layers, num_classes=10, float=False):
        super(ResNet_Cifar, self).__init__()
        self.inplanes = 16
        self.conv1 = nn.Conv2d(3, 16, kernel_size=3, stride=1, padding=1,␣
↪bias=False)
        self.bn1 = nn.BatchNorm2d(16)
        self.relu = nn.ReLU(inplace=True)
        self.layer1 = self._make_layer(block, 16, layers[0], float=float)
        self.layer2 = self._make_layer(block, 32, layers[1], stride=2,␣
↪float=float)
        self.layer3 = self._make_layer(block, 64, layers[2], stride=2,␣
↪float=float)
        self.avgpool = nn.AvgPool2d(8, stride=1)
        self.fc = nn.Linear(64 * block.expansion, num_classes)

        for m in self.modules():
            if isinstance(m, nn.Conv2d):
                n = m.kernel_size[0] * m.kernel_size[1] * m.out_channels
                m.weight.data.normal_(0, math.sqrt(2. / n))
            elif isinstance(m, nn.BatchNorm2d):
```

```
                m.weight.data.fill_(1)
                m.bias.data.zero_()

    def _make_layer(self, block, planes, blocks, stride=1, float=False):
        downsample = None
        if stride != 1 or self.inplanes != planes * block.expansion:
            downsample = nn.Sequential(
                QuantConv2d(self.inplanes, planes * block.expansion,␣
↪kernel_size=1, stride=stride, bias=False)
                if float is False else nn.Conv2d(self.inplanes, planes * block.
↪expansion, kernel_size=1,
                                                 stride=stride, bias=False),
                nn.BatchNorm2d(planes * block.expansion)
            )

        layers = []
        layers.append(block(self.inplanes, planes, stride, downsample,␣
↪float=float))
        self.inplanes = planes * block.expansion
        for _ in range(1, blocks):
            layers.append(block(self.inplanes, planes, float=float))

        return nn.Sequential(*layers)

    def forward(self, x):
        x = self.conv1(x)
        x = self.bn1(x)
        x = self.relu(x)

        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)

        x = self.avgpool(x)
        x = x.view(x.size(0), -1)
        x = self.fc(x)

        return x

    def show_params(self):
        for m in self.modules():
            if isinstance(m, QuantConv2d):
                m.show_params()


def resnet20_quant(**kwargs):
    model = ResNet_Cifar(BasicBlock, [3, 3, 3], **kwargs)
```

```
    return model


def resnet32_quant(**kwargs):
    model = ResNet_Cifar(BasicBlock, [5, 5, 5], **kwargs)
    return model


def resnet44_quant(**kwargs):
    model = ResNet_Cifar(BasicBlock, [7, 7, 7], **kwargs)
    return model


def resnet56_quant(**kwargs):
    model = ResNet_Cifar(BasicBlock, [9, 9, 9], **kwargs)
    return model


def resnet110_quant(**kwargs):
    model = ResNet_Cifar(BasicBlock, [18, 18, 18], **kwargs)
    return model


def resnet1202_quant(**kwargs):
    model = ResNet_Cifar(BasicBlock, [200, 200, 200], **kwargs)
    return model


def resnet164_quant(**kwargs):
    model = ResNet_Cifar(Bottleneck, [18, 18, 18], **kwargs)
    return model


def resnet1001_quant(**kwargs):
    model = ResNet_Cifar(Bottleneck, [111, 111, 111], **kwargs)
    return model


if __name__ == '__main__':
    pass
    # net = resnet20_cifar(float=True)
    # y = net(torch.randn(1, 3, 64, 64))
    # print(net)
    # print(y.size())


global best_prec
```

```
use_gpu = torch.cuda.is_available()
print('=> Building model...')




batch_size = 128
model_name = "resnet20_quant4bit"
model = resnet20_quant()
# Modify initial conv and batch norm layers
model.conv1 = nn.Conv2d(3, 8, kernel_size=3, stride=1, padding=1, bias=False)
model.bn1 = nn.BatchNorm2d(8, eps=1e-05, momentum=0.1, affine=True,␣
 ↪track_running_stats=True)

# Modify layer1[0] conv layers
model.layer1[0].conv1 = QuantConv2d(8, 8, kernel_size=3, stride=1, padding=1,␣
 ↪bias=False)
model.layer1[0].conv2 = QuantConv2d(8, 16, kernel_size=3, stride=1, padding=1,␣
 ↪bias=False)

# Replace batch norm layers for quantized convolutions
model.layer1[0].bn1 = nn.Sequential()  # Replace if not used
model.layer1[0].bn2 = nn.Sequential()  # Replace if not used

# Add downsample for residual connection
model.layer1[0].downsample = nn.Sequential(
    nn.Conv2d(8, 16, kernel_size=1, stride=1, bias=False),
    nn.BatchNorm2d(16)
)

print(model)

normalize = transforms.Normalize(mean=[0.491, 0.482, 0.447], std=[0.247, 0.243,␣
 ↪0.262])



train_dataset = torchvision.datasets.CIFAR10(
    root='./data',
    train=True,
    download=True,
    transform=transforms.Compose([
        transforms.RandomCrop(32, padding=4),
        transforms.RandomHorizontalFlip(),
        transforms.ToTensor(),
        normalize,
    ]))
```

```
trainloader = torch.utils.data.DataLoader(train_dataset, batch_size=batch_size,␣
 ↪shuffle=True, num_workers=2)


test_dataset = torchvision.datasets.CIFAR10(
    root='./data',
    train=False,
    download=True,
    transform=transforms.Compose([
        transforms.ToTensor(),
        normalize,
    ]))

testloader = torch.utils.data.DataLoader(test_dataset, batch_size=batch_size,␣
 ↪shuffle=False, num_workers=2)


print_freq = 100 # every 100 batches, accuracy printed. Here, each batch␣
 ↪includes "batch_size" data points
# CIFAR10 has 50,000 training data, and 10,000 validation data.

def train(trainloader, model, criterion, optimizer, epoch):
    batch_time = AverageMeter()
    data_time = AverageMeter()
    losses = AverageMeter()
    top1 = AverageMeter()

    model.train()

    end = time.time()
    for i, (input, target) in enumerate(trainloader):
        # measure data loading time
        data_time.update(time.time() - end)

        input, target = input.cuda(), target.cuda()

        # compute output
        output = model(input)
        loss = criterion(output, target)

        # measure accuracy and record loss
        prec = accuracy(output, target)[0]
        losses.update(loss.item(), input.size(0))
        top1.update(prec.item(), input.size(0))

        # compute gradient and do SGD step
        optimizer.zero_grad()
```

```
        loss.backward()
        optimizer.step()

        # measure elapsed time
        batch_time.update(time.time() - end)
        end = time.time()


        if i % print_freq == 0:
            print('Epoch: [{0}][{1}/{2}]\t'
                  'Time {batch_time.val:.3f} ({batch_time.avg:.3f})\t'
                  'Data {data_time.val:.3f} ({data_time.avg:.3f})\t'
                  'Loss {loss.val:.4f} ({loss.avg:.4f})\t'
                  'Prec {top1.val:.3f}% ({top1.avg:.3f}%)'.format(
                   epoch, i, len(trainloader), batch_time=batch_time,
                   data_time=data_time, loss=losses, top1=top1))



def validate(val_loader, model, criterion ):
    batch_time = AverageMeter()
    losses = AverageMeter()
    top1 = AverageMeter()

    # switch to evaluate mode
    model.eval()

    end = time.time()
    with torch.no_grad():
        for i, (input, target) in enumerate(val_loader):

            input, target = input.cuda(), target.cuda()

            # compute output
            output = model(input)
            loss = criterion(output, target)

            # measure accuracy and record loss
            prec = accuracy(output, target)[0]
            losses.update(loss.item(), input.size(0))
            top1.update(prec.item(), input.size(0))

            # measure elapsed time
            batch_time.update(time.time() - end)
            end = time.time()
```

```
            if i % print_freq == 0:  # This line shows how frequently print out␣
 ↪the status. e.g., i%5 => every 5 batch, prints out
                print('Test: [{0}/{1}]\t'
                  'Time {batch_time.val:.3f} ({batch_time.avg:.3f})\t'
                  'Loss {loss.val:.4f} ({loss.avg:.4f})\t'
                  'Prec {top1.val:.3f}% ({top1.avg:.3f}%)'.format(
                   i, len(val_loader), batch_time=batch_time, loss=losses,
                   top1=top1))

    print(' * Prec {top1.avg:.3f}% '.format(top1=top1))
    return top1.avg


def accuracy(output, target, topk=(1,)):
    """Computes the precision@k for the specified values of k"""
    maxk = max(topk)
    batch_size = target.size(0)

    _, pred = output.topk(maxk, 1, True, True)
    pred = pred.t()
    correct = pred.eq(target.view(1, -1).expand_as(pred))

    res = []
    for k in topk:
        correct_k = correct[:k].view(-1).float().sum(0)
        res.append(correct_k.mul_(100.0 / batch_size))
    return res


class AverageMeter(object):
    """Computes and stores the average and current value"""
    def __init__(self):
        self.reset()

    def reset(self):
        self.val = 0
        self.avg = 0
        self.sum = 0
        self.count = 0

    def update(self, val, n=1):
        self.val = val
        self.sum += val * n
        self.count += n
        self.avg = self.sum / self.count
```

```
def save_checkpoint(state, is_best, fdir):
    filepath = os.path.join(fdir, 'checkpoint.pth')
    torch.save(state, filepath)
    if is_best:
        shutil.copyfile(filepath, os.path.join(fdir, 'model_best.pth.tar'))


def adjust_learning_rate(optimizer, epoch):
    """For resnet, the lr starts from 0.1, and is divided by 10 at 80 and 120␣
 ↪epochs"""
    adjust_list = [ 70, 90]
    if epoch in adjust_list:
        for param_group in optimizer.param_groups:
            param_group['lr'] = param_group['lr'] * 0.1

#model = nn.DataParallel(model).cuda()
#all_params = checkpoint['state_dict']
#model.load_state_dict(all_params, strict=False)
#criterion = nn.CrossEntropyLoss().cuda()
#validate(testloader, model, criterion)
```

```
=> Building model…
ResNet_Cifar(
  (conv1): Conv2d(3, 8, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
  (bn1): BatchNorm2d(8, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
  (relu): ReLU(inplace=True)
  (layer1): Sequential(
    (0): BasicBlock(
      (conv1): QuantConv2d(
        8, 8, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
        (weight_quant): weight_quantize_fn()
      )
      (conv2): QuantConv2d(
        8, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
        (weight_quant): weight_quantize_fn()
      )
      (bn1): Sequential()
      (relu): ReLU(inplace=True)
      (bn2): Sequential()
      (downsample): Sequential(
        (0): Conv2d(8, 16, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (1): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
```

```
    (1): BasicBlock(
      (conv1): QuantConv2d(
        16, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
        (weight_quant): weight_quantize_fn()
      )
      (conv2): QuantConv2d(
        16, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
        (weight_quant): weight_quantize_fn()
      )
      (bn1): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (bn2): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
    (2): BasicBlock(
      (conv1): QuantConv2d(
        16, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
        (weight_quant): weight_quantize_fn()
      )
      (conv2): QuantConv2d(
        16, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
        (weight_quant): weight_quantize_fn()
      )
      (bn1): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (bn2): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
  )
  (layer2): Sequential(
    (0): BasicBlock(
      (conv1): QuantConv2d(
        16, 32, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False
        (weight_quant): weight_quantize_fn()
      )
      (conv2): QuantConv2d(
        32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
        (weight_quant): weight_quantize_fn()
      )
      (bn1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (bn2): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (downsample): Sequential(
        (0): QuantConv2d(
```

```
          16, 32, kernel_size=(1, 1), stride=(2, 2), bias=False
          (weight_quant): weight_quantize_fn()
        )
        (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): QuantConv2d(
        32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
        (weight_quant): weight_quantize_fn()
      )
      (conv2): QuantConv2d(
        32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
        (weight_quant): weight_quantize_fn()
      )
      (bn1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (bn2): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
    (2): BasicBlock(
      (conv1): QuantConv2d(
        32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
        (weight_quant): weight_quantize_fn()
      )
      (conv2): QuantConv2d(
        32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
        (weight_quant): weight_quantize_fn()
      )
      (bn1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (bn2): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
  )
  (layer3): Sequential(
    (0): BasicBlock(
      (conv1): QuantConv2d(
        32, 64, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False
        (weight_quant): weight_quantize_fn()
      )
      (conv2): QuantConv2d(
        64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
        (weight_quant): weight_quantize_fn()
      )
```

```
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (downsample): Sequential(
        (0): QuantConv2d(
          32, 64, kernel_size=(1, 1), stride=(2, 2), bias=False
          (weight_quant): weight_quantize_fn()
        )
        (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      )
    )
    (1): BasicBlock(
      (conv1): QuantConv2d(
        64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
        (weight_quant): weight_quantize_fn()
      )
      (conv2): QuantConv2d(
        64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
        (weight_quant): weight_quantize_fn()
      )
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
    (2): BasicBlock(
      (conv1): QuantConv2d(
        64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
        (weight_quant): weight_quantize_fn()
      )
      (conv2): QuantConv2d(
        64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
        (weight_quant): weight_quantize_fn()
      )
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
      (relu): ReLU(inplace=True)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    )
  )
  (avgpool): AvgPool2d(kernel_size=8, stride=1, padding=0)
  (fc): Linear(in_features=64, out_features=10, bias=True)
)
```

```
Files already downloaded and verified
Files already downloaded and verified
```

[ ]:
```
# This cell won't be given, but students will complete the training

lr = 4e-2
weight_decay = 1e-4
epochs = 100
best_prec = 0

#model = nn.DataParallel(model).cuda()
model.cuda()
criterion = nn.CrossEntropyLoss().cuda()
optimizer = torch.optim.SGD(model.parameters(), lr=lr, momentum=0.9,␣
 ↪weight_decay=weight_decay)
#cudnn.benchmark = True

if not os.path.exists('result'):
    os.makedirs('result')
fdir = 'result/'+str(model_name)
if not os.path.exists(fdir):
    os.makedirs(fdir)


for epoch in range(0, epochs):
    adjust_learning_rate(optimizer, epoch)

    train(trainloader, model, criterion, optimizer, epoch)

    # evaluate on test set
    print("Validation starts")
    prec = validate(testloader, model, criterion)

    # remember best precision and save checkpoint
    is_best = prec > best_prec
    best_prec = max(prec,best_prec)
    print('best acc: {:1f}'.format(best_prec))
    save_checkpoint({
        'epoch': epoch + 1,
        'state_dict': model.state_dict(),
        'best_prec': best_prec,
        'optimizer': optimizer.state_dict(),
    }, is_best, fdir)
```

[2]:
```
PATH = "result/resnet20_quant4bit/model_best.pth.tar"
checkpoint = torch.load(PATH)
model.load_state_dict(checkpoint['state_dict'])
```

```
device = torch.device("cuda")

model.cuda()
model.eval()

test_loss = 0
correct = 0

with torch.no_grad():
    for data, target in testloader:
        data, target = data.to(device), target.to(device) # loading to GPU
        output = model(data)
        pred = output.argmax(dim=1, keepdim=True)
        correct += pred.eq(target.view_as(pred)).sum().item()

test_loss /= len(testloader.dataset)

print('\nTest set: Accuracy: {}/{} ({:.0f}%)\n'.format(
        correct, len(testloader.dataset),
        100. * correct / len(testloader.dataset)))
```

```
Test set: Accuracy: 7990/10000 (80%)
```

[3]:
```
class SaveOutput:
    def __init__(self):
        self.outputs = []
    def __call__(self, module, module_in):
        self.outputs.append(module_in)
    def clear(self):
        self.outputs = []

######### Save inputs from selected layer ##########
save_output = SaveOutput()
device = torch.device("cuda" if use_gpu else "cpu")
counter =0
for layer in model.modules():
    if isinstance(layer, torch.nn.Conv2d):
        print("prehooked")
        counter += 1
        print(layer, counter)
        layer.register_forward_pre_hook(save_output)       ## Input for the␣
 ↪module will be grapped
####################################################

dataiter = iter(trainloader)
```

```
images, labels = next(dataiter)
images = images.to(device)
out = model(images)
```

```
prehooked
Conv2d(3, 8, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False) 1
prehooked
QuantConv2d(
  8, 8, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
  (weight_quant): weight_quantize_fn()
) 2
prehooked
QuantConv2d(
  8, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
  (weight_quant): weight_quantize_fn()
) 3
prehooked
Conv2d(8, 16, kernel_size=(1, 1), stride=(1, 1), bias=False) 4
prehooked
QuantConv2d(
  16, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
  (weight_quant): weight_quantize_fn()
) 5
prehooked
QuantConv2d(
  16, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
  (weight_quant): weight_quantize_fn()
) 6
prehooked
QuantConv2d(
  16, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
  (weight_quant): weight_quantize_fn()
) 7
prehooked
QuantConv2d(
  16, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
  (weight_quant): weight_quantize_fn()
) 8
prehooked
QuantConv2d(
  16, 32, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False
  (weight_quant): weight_quantize_fn()
) 9
prehooked
QuantConv2d(
  32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
  (weight_quant): weight_quantize_fn()
```

```
) 10
prehooked
QuantConv2d(
  16, 32, kernel_size=(1, 1), stride=(2, 2), bias=False
  (weight_quant): weight_quantize_fn()
) 11
prehooked
QuantConv2d(
  32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
  (weight_quant): weight_quantize_fn()
) 12
prehooked
QuantConv2d(
  32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
  (weight_quant): weight_quantize_fn()
) 13
prehooked
QuantConv2d(
  32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
  (weight_quant): weight_quantize_fn()
) 14
prehooked
QuantConv2d(
  32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
  (weight_quant): weight_quantize_fn()
) 15
prehooked
QuantConv2d(
  32, 64, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False
  (weight_quant): weight_quantize_fn()
) 16
prehooked
QuantConv2d(
  64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
  (weight_quant): weight_quantize_fn()
) 17
prehooked
QuantConv2d(
  32, 64, kernel_size=(1, 1), stride=(2, 2), bias=False
  (weight_quant): weight_quantize_fn()
) 18
prehooked
QuantConv2d(
  64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
  (weight_quant): weight_quantize_fn()
) 19
prehooked
QuantConv2d(
```

```
  64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
  (weight_quant): weight_quantize_fn()
) 20
prehooked
QuantConv2d(
  64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
  (weight_quant): weight_quantize_fn()
) 21
prehooked
QuantConv2d(
  64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
  (weight_quant): weight_quantize_fn()
) 22
```

[4]:
```
w_bit = 4
weight_q = model.layer1[0].conv2.weight_q # quantized value is stored during␣
 ↪the training
w_alpha = model.layer1[0].conv2.weight_quant.wgt_alpha
w_delta = w_alpha/(2**(w_bit-1)-1)
weight_int = weight_q/w_delta
print(weight_int) # you should see clean integer numbers
```

```
tensor([[[[-7.0000, -7.0000, -7.0000],
          [-7.0000, -7.0000, -7.0000],
          [-7.0000, -7.0000, -7.0000]],

         [[ 7.0000,  7.0000,  7.0000],
          [ 7.0000,  7.0000,  7.0000],
          [ 7.0000,  7.0000,  7.0000]],

         [[-7.0000, -7.0000, -7.0000],
          [-7.0000, -7.0000, -7.0000],
          [-7.0000, -7.0000, -0.0000]],

         …,

         [[-7.0000, -7.0000, -7.0000],
          [-7.0000, -7.0000, -7.0000],
          [-7.0000, -7.0000, -7.0000]],

         [[-7.0000, -7.0000, -7.0000],
          [-7.0000, -7.0000, -7.0000],
          [-7.0000, -7.0000, -7.0000]],

         [[ 7.0000,  7.0000,  7.0000],
          [ 7.0000,  7.0000,  7.0000],
          [ 7.0000,  7.0000,  7.0000]]],
```

```
[[[-7.0000, -7.0000, -7.0000],
  [-7.0000, -7.0000, -7.0000],
  [-7.0000, -7.0000, -7.0000]],

 [[ 7.0000,  7.0000,  7.0000],
  [ 7.0000,  7.0000,  7.0000],
  [-7.0000,  7.0000, -7.0000]],

 [[-7.0000, -7.0000, -7.0000],
  [-7.0000, -7.0000, -7.0000],
  [-7.0000, -7.0000, -7.0000]],

 …,

 [[-7.0000, -7.0000,  7.0000],
  [-0.0000, -0.0000,  7.0000],
  [-7.0000, -7.0000, -7.0000]],

 [[-0.0000, -7.0000, -7.0000],
  [-7.0000, -0.0000,  0.0000],
  [ 0.0000,  0.0000,  7.0000]],

 [[ 7.0000,  7.0000,  7.0000],
  [ 7.0000,  7.0000,  7.0000],
  [ 7.0000,  7.0000,  7.0000]]],


[[[-7.0000, -7.0000, -7.0000],
  [-7.0000, -7.0000, -7.0000],
  [-7.0000, -7.0000, -7.0000]],

 [[ 7.0000,  7.0000,  7.0000],
  [ 7.0000,  7.0000,  7.0000],
  [ 7.0000,  7.0000,  7.0000]],

 [[ 7.0000,  7.0000,  7.0000],
  [ 7.0000,  7.0000,  7.0000],
  [ 7.0000,  7.0000,  7.0000]],

 …,

 [[-7.0000, -0.0000, -7.0000],
  [-7.0000, -7.0000, -7.0000],
  [-0.0000,  7.0000,  7.0000]],

 [[-7.0000, -7.0000, -7.0000],
  [ 0.0000, -7.0000,  7.0000],
```

```
  [-7.0000, -0.0000,  0.0000]],

 [[ 7.0000,  7.0000,  7.0000],
  [ 7.0000,  7.0000,  7.0000],
  [ 7.0000,  7.0000,  7.0000]]],


…,


[[[-7.0000, -7.0000, -7.0000],
  [-7.0000, -7.0000, -7.0000],
  [-7.0000, -7.0000, -7.0000]],

 [[ 7.0000,  7.0000,  7.0000],
  [ 7.0000,  7.0000,  7.0000],
  [ 7.0000,  7.0000,  7.0000]],

 [[-7.0000, -7.0000, -7.0000],
  [-7.0000, -7.0000, -7.0000],
  [-7.0000, -7.0000, -7.0000]],

 …,

 [[-7.0000, -7.0000, -7.0000],
  [-7.0000, -7.0000, -7.0000],
  [-7.0000, -7.0000, -7.0000]],

 [[-7.0000, -7.0000, -7.0000],
  [-7.0000, -7.0000, -7.0000],
  [-7.0000, -7.0000, -7.0000]],

 [[ 7.0000,  7.0000,  7.0000],
  [ 7.0000,  7.0000,  7.0000],
  [ 7.0000,  7.0000,  7.0000]]],


[[[-7.0000, -7.0000, -7.0000],
  [-7.0000, -7.0000, -7.0000],
  [-7.0000, -7.0000, -7.0000]],

 [[-7.0000,  7.0000,  7.0000],
  [ 7.0000,  7.0000,  7.0000],
  [ 7.0000,  7.0000,  7.0000]],

 [[ 7.0000,  0.0000, -7.0000],
  [ 7.0000,  7.0000,  7.0000],
  [-7.0000, -7.0000, -7.0000]],
```

```
        …,

        [[ 0.0000,  0.0000, -7.0000],
         [ 0.0000,  0.0000, -7.0000],
         [-0.0000, -0.0000, -7.0000]],

        [[-0.0000, -0.0000, -7.0000],
         [ 0.0000,  0.0000, -0.0000],
         [-7.0000, -7.0000, -7.0000]],

        [[ 7.0000,  7.0000,  7.0000],
         [ 7.0000,  7.0000,  7.0000],
         [ 7.0000,  7.0000,  7.0000]]],


       [[[-7.0000, -7.0000, -7.0000],
         [-7.0000, -7.0000, -7.0000],
         [-7.0000, -7.0000, -7.0000]],

        [[-7.0000,  7.0000, -7.0000],
         [-7.0000, -7.0000, -7.0000],
         [ 7.0000,  7.0000, -7.0000]],

        [[-7.0000, -7.0000, -7.0000],
         [ 7.0000,  7.0000,  7.0000],
         [ 7.0000,  7.0000,  7.0000]],

        …,

        [[-7.0000,  7.0000, -7.0000],
         [ 7.0000,  7.0000,  7.0000],
         [-7.0000,  7.0000,  7.0000]],

        [[-7.0000, -7.0000, -7.0000],
         [-7.0000, -7.0000, -7.0000],
         [ 0.0000,  7.0000,  7.0000]],

        [[ 7.0000,  7.0000,  7.0000],
         [ 7.0000,  7.0000,  7.0000],
         [ 7.0000,  7.0000,  7.0000]]]], device='cuda:0',
      grad_fn=<DivBackward0>)
```

[5]:
```
x_bit = 4
x = save_output.outputs[2][0]  # input of the 2nd conv layer
x_alpha  = model.layer1[0].conv2.act_alpha
x_delta = x_alpha/(2**x_bit-1)
```

```
act_quant_fn = act_quantization(x_bit) # define the quantization function
x_q = act_quant_fn(x, x_alpha)         # create the quantized value for x

x_int = x_q/x_delta
print(x_int) # you should see clean integer numbers
```

```
tensor([[[[ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000],
          [ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000],
          [ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000],
          …,
          [ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000],
          [ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000],
          [ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000]],

         [[15.0000, 15.0000, 15.0000,  …, 15.0000, 15.0000, 15.0000],
          [15.0000, 15.0000, 15.0000,  …, 15.0000, 15.0000, 15.0000],
          [15.0000, 15.0000, 15.0000,  …, 15.0000, 15.0000, 15.0000],
          …,
          [15.0000, 15.0000, 15.0000,  …, 15.0000, 15.0000, 15.0000],
          [15.0000, 15.0000, 15.0000,  …, 15.0000, 15.0000, 15.0000],
          [15.0000, 15.0000, 15.0000,  …, 15.0000, 15.0000, 15.0000]],

         [[ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000],
          [ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000],
          [ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000],
          …,
          [ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000],
          [ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000],
          [ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000]],

         …,

         [[12.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000],
          [15.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000],
          [15.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000],
          …,
          [ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000],
          [ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000],
          [ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000]],

         [[12.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000],
          [ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000],
          [ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000],
          …,
          [ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000],
          [ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000],
```

```
  [ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000]],

 [[15.0000, 15.0000, 15.0000,  …, 15.0000, 15.0000, 15.0000],
  [15.0000, 15.0000, 15.0000,  …, 15.0000, 15.0000, 15.0000],
  [15.0000, 15.0000, 15.0000,  …, 15.0000, 15.0000, 15.0000],
  …,
  [15.0000, 15.0000, 15.0000,  …, 15.0000, 15.0000, 15.0000],
  [15.0000, 15.0000, 15.0000,  …, 15.0000, 15.0000, 15.0000],
  [15.0000, 15.0000, 15.0000,  …, 15.0000, 15.0000, 15.0000]]],


[[[ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000],
  [ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000],
  [ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000],
  …,
  [ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000],
  [ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000],
  [ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000]],

 [[15.0000, 15.0000, 15.0000,  …, 15.0000, 15.0000, 15.0000],
  [15.0000, 15.0000, 15.0000,  …, 15.0000, 15.0000, 15.0000],
  [15.0000, 15.0000, 15.0000,  …, 15.0000, 15.0000, 15.0000],
  …,
  [15.0000, 15.0000, 15.0000,  …, 15.0000, 15.0000, 15.0000],
  [15.0000, 15.0000, 15.0000,  …, 15.0000, 15.0000, 15.0000],
  [15.0000, 15.0000, 15.0000,  …, 15.0000, 15.0000, 15.0000]],

 [[ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000],
  [ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000],
  [ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000],
  …,
  [ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000],
  [ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000],
  [ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000]],

 …,

 [[ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000],
  [ 0.0000,  0.0000,  0.0000,  …,  0.0000, 15.0000,  0.0000],
  [ 0.0000,  0.0000,  0.0000,  …,  0.0000, 15.0000,  0.0000],
  …,
  [ 0.0000,  0.0000,  0.0000,  …,  0.0000, 15.0000,  0.0000],
  [ 0.0000,  0.0000,  0.0000,  …,  0.0000, 15.0000,  0.0000],
  [ 0.0000,  0.0000,  0.0000,  …,  0.0000, 12.0000,  0.0000]],

 [[ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000],
  [ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000],
  [ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000],
```

```
  …,
   [ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000],
   [ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000],
   [ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000]],

  [[15.0000, 15.0000, 15.0000,  …, 15.0000, 15.0000, 15.0000],
   [15.0000, 15.0000, 15.0000,  …, 15.0000, 15.0000, 15.0000],
   [15.0000, 15.0000, 15.0000,  …, 15.0000, 15.0000, 15.0000],
  …,
   [15.0000, 15.0000, 15.0000,  …, 15.0000, 15.0000, 15.0000],
   [15.0000, 15.0000, 15.0000,  …, 15.0000, 15.0000, 15.0000],
   [15.0000, 15.0000, 15.0000,  …, 15.0000, 15.0000, 15.0000]]],


[[[ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000],
   [ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000],
   [ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000],
  …,
   [ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000],
   [ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000],
   [ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000]],

  [[15.0000, 15.0000, 15.0000,  …, 15.0000, 15.0000, 15.0000],
   [15.0000, 15.0000, 15.0000,  …, 15.0000, 15.0000, 15.0000],
   [15.0000, 15.0000, 15.0000,  …, 15.0000, 15.0000, 15.0000],
  …,
   [15.0000, 15.0000, 15.0000,  …, 15.0000, 15.0000, 15.0000],
   [15.0000, 15.0000, 15.0000,  …, 15.0000, 15.0000, 15.0000],
   [15.0000, 15.0000, 15.0000,  …, 15.0000, 15.0000, 15.0000]],

  [[ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000],
   [ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000],
   [ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000],
  …,
   [ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000],
   [ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000],
   [ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000]],

  …,

  [[ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000],
   [ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000],
   [ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000],
  …,
   [ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000],
   [ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000],
   [ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000]],
```

```
 [[ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000],
  [ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000],
  [ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000],
  …,
  [ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000],
  [ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000],
  [ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000]],

 [[15.0000, 15.0000, 15.0000,  …, 15.0000, 15.0000, 15.0000],
  [15.0000, 15.0000, 15.0000,  …, 15.0000, 15.0000, 15.0000],
  [15.0000, 15.0000, 15.0000,  …, 15.0000, 15.0000, 15.0000],
  …,
  [15.0000, 15.0000, 15.0000,  …, 15.0000, 15.0000, 15.0000],
  [15.0000, 15.0000, 15.0000,  …, 15.0000, 15.0000, 15.0000],
  [15.0000, 15.0000, 15.0000,  …, 15.0000, 15.0000, 15.0000]]],


…,


[[[ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000],
  [ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000],
  [ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000],
  …,
  [ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000],
  [ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000],
  [ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000]],

 [[15.0000, 15.0000, 15.0000,  …, 15.0000, 15.0000, 15.0000],
  [15.0000, 15.0000, 15.0000,  …, 15.0000, 15.0000, 15.0000],
  [15.0000, 15.0000, 15.0000,  …, 15.0000, 15.0000, 15.0000],
  …,
  [15.0000, 15.0000, 15.0000,  …, 15.0000, 15.0000, 15.0000],
  [15.0000, 15.0000, 15.0000,  …, 15.0000, 15.0000, 15.0000],
  [15.0000, 15.0000, 15.0000,  …, 15.0000, 15.0000, 15.0000]],

 [[ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000],
  [ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000],
  [ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000],
  …,
  [ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000],
  [ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000],
  [ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000]],

 …,

 [[ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000],
  [ 0.0000,  0.0000, 15.0000,  …,  0.0000,  0.0000,  0.0000],
```

```
  [ 0.0000,  0.0000, 15.0000,  …,  0.0000,  0.0000,  0.0000],
 …,
  [ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000],
  [ 0.0000,  0.0000,  0.0000,  …,  0.0000, 15.0000,  0.0000],
  [ 0.0000,  0.0000,  0.0000,  …,  0.0000, 12.0000,  0.0000]],

 [[ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000],
  [ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000],
  [ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000],
 …,
  [ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000],
  [ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000],
  [ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000]],

 [[15.0000, 15.0000, 15.0000,  …, 15.0000, 15.0000, 15.0000],
  [15.0000, 15.0000, 15.0000,  …, 15.0000, 15.0000, 15.0000],
  [15.0000, 15.0000, 15.0000,  …, 15.0000, 15.0000, 15.0000],
 …,
  [15.0000, 15.0000, 15.0000,  …, 15.0000, 15.0000, 15.0000],
  [15.0000, 15.0000, 15.0000,  …, 15.0000, 15.0000, 15.0000],
  [15.0000, 15.0000, 15.0000,  …, 15.0000, 15.0000, 15.0000]]],


[[[ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000],
  [ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000],
  [ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000],
 …,
  [ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000],
  [ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000],
  [ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000]],

 [[15.0000, 15.0000, 15.0000,  …, 15.0000, 15.0000, 15.0000],
  [15.0000, 15.0000, 15.0000,  …, 15.0000, 15.0000, 15.0000],
  [15.0000, 15.0000, 15.0000,  …, 15.0000, 15.0000, 15.0000],
 …,
  [15.0000, 15.0000, 15.0000,  …, 15.0000, 15.0000, 15.0000],
  [15.0000, 15.0000, 15.0000,  …, 15.0000, 15.0000, 15.0000],
  [15.0000, 15.0000, 15.0000,  …, 15.0000, 15.0000, 15.0000]],

 [[ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000],
  [ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000],
  [ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000],
 …,
  [ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000],
  [ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000],
  [ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000]],

 …,
```

```
[[ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000],
 [ 0.0000,  0.0000,  0.0000,  …,  0.0000, 15.0000,  0.0000],
 [ 0.0000,  0.0000,  0.0000,  …,  0.0000, 15.0000,  0.0000],
 …,
 [ 0.0000,  0.0000,  0.0000,  …,  0.0000, 15.0000,  0.0000],
 [ 0.0000,  0.0000,  0.0000,  …,  0.0000, 15.0000,  0.0000],
 [ 0.0000,  0.0000,  0.0000,  …,  0.0000, 15.0000,  0.0000]],

[[ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000],
 [ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000],
 [ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000],
 …,
 [ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000],
 [ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000],
 [ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000]],

[[15.0000, 15.0000, 15.0000,  …, 15.0000, 15.0000, 15.0000],
 [15.0000, 15.0000, 15.0000,  …, 15.0000, 15.0000, 15.0000],
 [15.0000, 15.0000, 15.0000,  …, 15.0000, 15.0000, 15.0000],
 …,
 [15.0000, 15.0000, 15.0000,  …, 15.0000, 15.0000, 15.0000],
 [15.0000, 15.0000, 15.0000,  …, 15.0000, 15.0000, 15.0000],
 [15.0000, 15.0000, 15.0000,  …, 15.0000, 15.0000, 15.0000]]],


[[[ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000],
  [ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000],
  [ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000],
  …,
  [ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000],
  [ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000],
  [ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000]],

 [[15.0000, 15.0000, 15.0000,  …, 15.0000, 15.0000, 15.0000],
  [15.0000, 15.0000, 15.0000,  …, 15.0000, 15.0000, 15.0000],
  [15.0000, 15.0000, 15.0000,  …, 15.0000, 15.0000, 15.0000],
  …,
  [15.0000, 15.0000, 15.0000,  …, 15.0000, 15.0000, 15.0000],
  [15.0000, 15.0000, 15.0000,  …, 15.0000, 15.0000, 15.0000],
  [15.0000, 15.0000, 15.0000,  …, 15.0000, 15.0000, 15.0000]],

 [[ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000],
  [ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000],
  [ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000],
  …,
  [ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000],
  [ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000],
```

```
          [ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000]],

         …,

         [[ 0.0000,  0.0000,  0.0000,  …,  0.0000, 12.0000,  0.0000],
          [ 0.0000,  0.0000,  0.0000,  …,  0.0000, 15.0000,  0.0000],
          [ 0.0000,  0.0000,  0.0000,  …,  0.0000, 15.0000,  0.0000],
          …,
          [ 0.0000,  0.0000,  0.0000,  …,  0.0000, 15.0000,  0.0000],
          [ 0.0000,  0.0000,  0.0000,  …,  0.0000, 15.0000,  0.0000],
          [ 0.0000,  0.0000,  0.0000,  …,  0.0000, 12.0000,  0.0000]],

         [[ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000],
          [ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000],
          [ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000],
          …,
          [ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000],
          [ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000],
          [ 0.0000,  0.0000,  0.0000,  …,  0.0000,  0.0000,  0.0000]],

         [[15.0000, 15.0000, 15.0000,  …, 15.0000, 15.0000, 15.0000],
          [15.0000, 15.0000, 15.0000,  …, 15.0000, 15.0000, 15.0000],
          [15.0000, 15.0000, 15.0000,  …, 15.0000, 15.0000, 15.0000],
          …,
          [15.0000, 15.0000, 15.0000,  …, 15.0000, 15.0000, 15.0000],
          [15.0000, 15.0000, 15.0000,  …, 15.0000, 15.0000, 15.0000],
          [15.0000, 15.0000, 15.0000,  …, 15.0000, 15.0000, 15.0000]]]],
       device='cuda:0', grad_fn=<DivBackward0>)
```

[6]:
```
conv_int = torch.nn.Conv2d(in_channels = 8, out_channels=8, kernel_size = 3,␣
 ↪padding=1, bias = False)
conv_int.weight = torch.nn.parameter.Parameter(weight_int)
relu = nn.ReLU()
bn = nn.BatchNorm2d(8, eps=1e-05, momentum=0.1, affine=True,␣
 ↪track_running_stats=True).to(device)
output_int = conv_int(x_int)
output_int = output_int*w_delta*x_delta
output_recovered = relu(output_int)
```

[7]:
```
#print(save_output.outputs[3][0].size())
output_recovered = output_recovered[:, :8, :, :]
#print(output_recovered.size())
difference = abs(save_output.outputs[3][0] - output_recovered )
print(difference.mean())  ## It should be small, e.g.,2.3 in my trainned model
```

```
tensor(5.0156, device='cuda:0', grad_fn=<MeanBackward0>)
```

[ ]:

[ ]:

[ ]:

[ ]: