proj_part1_vgg

November 21, 2022

[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

from models import *


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



batch_size = 128
model_name = "VGG_quant_proj"
model = VGG16_quant_proj()

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 = [15, 30, 45]
    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…
VGG_quant(
  (features): Sequential(
    (0): QuantConv2d(
      3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
      (weight_quant): weight_quantize_fn()
    )
    (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (2): ReLU(inplace=True)
    (3): QuantConv2d(
      64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
      (weight_quant): weight_quantize_fn()
    )
    (4): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (5): ReLU(inplace=True)
    (6): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
    (7): QuantConv2d(
```

```
      64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
      (weight_quant): weight_quantize_fn()
    )
    (8): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (9): ReLU(inplace=True)
    (10): QuantConv2d(
      128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
      (weight_quant): weight_quantize_fn()
    )
    (11): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (12): ReLU(inplace=True)
    (13): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
    (14): QuantConv2d(
      128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
      (weight_quant): weight_quantize_fn()
    )
    (15): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (16): ReLU(inplace=True)
    (17): QuantConv2d(
      256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
      (weight_quant): weight_quantize_fn()
    )
    (18): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (19): ReLU(inplace=True)
    (20): QuantConv2d(
      256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
      (weight_quant): weight_quantize_fn()
    )
    (21): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (22): ReLU(inplace=True)
    (23): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
    (24): QuantConv2d(
      256, 8, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
      (weight_quant): weight_quantize_fn()
    )
    (25): BatchNorm2d(8, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (26): ReLU(inplace=True)
    (27): QuantConv2d(
      8, 8, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
      (weight_quant): weight_quantize_fn()
```

```
    )
    (28): ReLU(inplace=True)
    (29): QuantConv2d(
      8, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
      (weight_quant): weight_quantize_fn()
    )
    (30): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (31): ReLU(inplace=True)
    (32): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
    (33): QuantConv2d(
      512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
      (weight_quant): weight_quantize_fn()
    )
    (34): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (35): ReLU(inplace=True)
    (36): QuantConv2d(
      512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
      (weight_quant): weight_quantize_fn()
    )
    (37): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (38): ReLU(inplace=True)
    (39): QuantConv2d(
      512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
      (weight_quant): weight_quantize_fn()
    )
    (40): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (41): ReLU(inplace=True)
    (42): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
    (43): AvgPool2d(kernel_size=1, stride=1, padding=0)
  )
  (classifier): Linear(in_features=512, 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 # orig 4e-2
weight_decay = 1e-3 # orig 1e-4
epochs = 60
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)
```

[ ]:
```
# HW

#  1. Train with 4 bits for both weight and activation to achieve >90% accuracy
#  2. Find x_int and w_int for the 2nd convolution layer
#  3. Check the recovered psum has similar value to the un-quantized original␣
↪psum
#     (such as example 1 in W3S2)
```

0.1 Quantization is set to 4 bits. Below is accuracy after training

[2]:
```
PATH = "result/VGG_quant_proj/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: 9160/10000 (92%)
```

[3]:
```
#send an input and grap the value by using prehook like HW3
```

0.2 Prehook

[4]:
```
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()
for layer in model.modules():
    if isinstance(layer, QuantConv2d):
        print("prehooked")
```

```
        #print(layer)

        layer.register_forward_pre_hook(save_output)       ## Input for the␣
 ↪module will be grapped
#####################################################
```

```
prehooked
prehooked
prehooked
prehooked
prehooked
prehooked
prehooked
prehooked
prehooked
prehooked
prehooked
prehooked
prehooked
```

[5]:
```
use_gpu = torch.cuda.is_available()
device = torch.device("cuda" if use_gpu else "cpu")


dataiter = iter(trainloader)
images, labels = dataiter.next()
images = images.to(device)
out = model(images)
print("1st convolution's input size:", save_output.outputs[7][0].size())
print("2nd convolution's input size:", save_output.outputs[8][0].size())
print("3rd convolution's input size:", save_output.outputs[9][0].size())
print("4th convolution's input size:", save_output.outputs[10][0].size())
```

```
1st convolution's input size: torch.Size([128, 256, 4, 4])
2nd convolution's input size: torch.Size([128, 8, 4, 4])
3rd convolution's input size: torch.Size([128, 8, 4, 4])
4th convolution's input size: torch.Size([128, 512, 2, 2])
```

[6]:
```
w_bit = 4

weight_q = model.features[27].weight_q # quantized value is stored during the␣
 ↪training
w_alpha = model.features[27].weight_quant.wgt_alpha.data.item()
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]],

 [[ 6.0000,  7.0000,  7.0000],
  [ 3.0000,  7.0000,  7.0000],
  [ 1.0000,  1.0000,  7.0000]],

 [[ 7.0000,  7.0000,  7.0000],
  [ 7.0000,  7.0000,  7.0000],
  [ 5.0000,  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],
  [ 6.0000,  6.0000,  6.0000]],

 [[-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]],

 [[-1.0000, -6.0000, -7.0000],
  [-7.0000, -7.0000, -7.0000],
  [-7.0000, -7.0000,  2.0000]],

 [[-3.0000, -6.0000,  2.0000],
  [-7.0000, -5.0000,  6.0000],
  [-7.0000,  3.0000,  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]],

 [[-5.0000,  2.0000,  7.0000],
  [-5.0000,  3.0000,  7.0000],
  [-1.0000,  5.0000,  7.0000]],

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


[[[ 1.0000, -0.0000, -6.0000],
  [-0.0000, -4.0000, -7.0000],
  [ 1.0000, -1.0000, -5.0000]],

 [[-7.0000, -7.0000, -0.0000],
  [-7.0000, -7.0000, -2.0000],
  [-7.0000, -7.0000, -1.0000]],

 [[ 7.0000, -1.0000, -4.0000],
  [ 1.0000, -3.0000, -7.0000],
  [-0.0000, -7.0000, -7.0000]],

 [[-2.0000, -3.0000, -5.0000],
  [-5.0000, -7.0000, -7.0000],
  [-7.0000, -7.0000, -7.0000]],

 [[-7.0000, -7.0000, -7.0000],
  [-7.0000, -7.0000, -7.0000],
  [-1.0000,  2.0000,  3.0000]],

 [[-7.0000, -7.0000, -7.0000],
  [-7.0000, -7.0000, -7.0000],
  [-7.0000, -7.0000, -3.0000]],

 [[ 1.0000, -7.0000, -7.0000],
  [ 2.0000,  1.0000, -7.0000],
  [ 7.0000,  4.0000,  2.0000]],

 [[ 1.0000, -1.0000,  2.0000],
  [-2.0000, -4.0000, -2.0000],
  [-7.0000, -7.0000, -5.0000]]],
```

```
[[[-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]],

 [[ 3.0000,  2.0000, -7.0000],
  [ 1.0000,  2.0000,  4.0000],
  [ 7.0000,  7.0000,  7.0000]],

 [[ 2.0000,  6.0000,  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]],

 [[ 1.0000, -7.0000, -5.0000],
  [-5.0000, -7.0000, -2.0000],
  [ 3.0000,  1.0000, -6.0000]],

 [[-1.0000,  6.0000,  7.0000],
```

```
 [ 1.0000,  7.0000,  7.0000],
 [ 3.0000,  2.0000,  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, -6.0000, -7.0000],
  [-5.0000, -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, -3.0000]],

 [[ 7.0000,  7.0000, -7.0000],
  [ 7.0000,  7.0000, -7.0000],
  [ 7.0000, -2.0000, -7.0000]],

 [[ 7.0000, -3.0000, -7.0000],
  [-0.0000, -7.0000, -7.0000],
  [-7.0000, -7.0000, -7.0000]],

 [[-7.0000, -7.0000, -7.0000],
  [-5.0000, -7.0000, -7.0000],
  [-4.0000, -7.0000,  0.0000]],

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

 [[ 5.0000, -6.0000, -7.0000],
  [-4.0000, -7.0000, -7.0000],
  [-5.0000, -7.0000, -4.0000]],
```

```
 [[ 3.0000, -2.0000, -1.0000],
  [-5.0000, -7.0000, -7.0000],
  [ 1.0000, -5.0000, -1.0000]]],


[[[-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]],

 [[ 3.0000,  6.0000,  7.0000],
  [ 2.0000,  4.0000,  3.0000],
  [ 7.0000,  7.0000,  7.0000]],

 [[ 6.0000,  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]]],


[[[ 5.0000, -3.0000,  3.0000],
  [ 6.0000,  3.0000,  3.0000],
  [ 1.0000, -4.0000, -1.0000]],

 [[-7.0000, -6.0000, -2.0000],
  [-7.0000, -7.0000, -3.0000],
  [-7.0000, -7.0000, -6.0000]],

 [[-2.0000, -4.0000, -7.0000],
  [ 3.0000, -3.0000, -7.0000],
```

```
          [ 5.0000,  7.0000, -7.0000]],

         [[-6.0000, -7.0000, -7.0000],
          [-4.0000, -6.0000, -4.0000],
          [ 5.0000,  1.0000,  4.0000]],

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

         [[-7.0000, -7.0000, -4.0000],
          [-7.0000, -7.0000, -7.0000],
          [-7.0000, -7.0000, -7.0000]],

         [[ 7.0000,  6.0000,  1.0000],
          [ 7.0000,  7.0000,  2.0000],
          [ 7.0000,  7.0000,  3.0000]],

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

[7]:
```
x_bit = 4
x = save_output.outputs[8][0]  # input of the 2nd conv layer
x_alpha  = model.features[27].act_alpha.data.item()
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,  1.0000]],

         [[ 2.0000,  1.0000,  1.0000,  0.0000],
          [ 0.0000,  0.0000,  1.0000,  0.0000],
          [ 0.0000,  0.0000,  6.0000,  5.0000],
          [ 0.0000,  0.0000,  3.0000,  4.0000]],

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

```
…,

 [[ 9.0000,  1.0000,  3.0000,  4.0000],
  [10.0000,  0.0000,  0.0000,  0.0000],
  [ 1.0000,  0.0000,  0.0000,  0.0000],
  [ 0.0000,  0.0000,  1.0000,  4.0000]],

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

 [[ 0.0000,  0.0000,  3.0000,  1.0000],
  [ 0.0000,  0.0000,  3.0000,  3.0000],
  [ 2.0000,  0.0000,  5.0000,  4.0000],
  [ 4.0000,  4.0000,  4.0000,  2.0000]]],


[[[ 1.0000,  0.0000,  1.0000,  1.0000],
  [ 3.0000,  2.0000,  0.0000,  0.0000],
  [ 9.0000,  8.0000,  0.0000,  0.0000],
  [13.0000, 10.0000,  0.0000,  0.0000]],

 [[ 4.0000,  5.0000,  5.0000,  5.0000],
  [ 7.0000,  9.0000,  8.0000,  9.0000],
  [ 7.0000,  9.0000,  6.0000,  7.0000],
  [10.0000,  9.0000,  6.0000,  5.0000]],

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

 …,

 [[ 7.0000,  3.0000,  4.0000,  1.0000],
  [15.0000, 10.0000,  9.0000,  2.0000],
  [ 8.0000,  6.0000,  5.0000,  2.0000],
  [15.0000, 15.0000, 13.0000,  8.0000]],

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

 [[ 0.0000,  0.0000,  0.0000,  2.0000],
  [ 2.0000, 10.0000, 12.0000,  8.0000],
```

```
  [ 0.0000,  0.0000,  6.0000,  3.0000],
  [ 0.0000,  0.0000,  3.0000,  2.0000]]],


[[[ 3.0000,  5.0000,  1.0000,  0.0000],
  [ 1.0000,  4.0000,  1.0000,  0.0000],
  [ 2.0000,  4.0000,  1.0000,  0.0000],
  [ 3.0000,  1.0000,  1.0000,  1.0000]],

 [[ 4.0000,  3.0000,  5.0000,  6.0000],
  [ 5.0000,  4.0000,  3.0000,  5.0000],
  [ 4.0000,  5.0000,  4.0000,  5.0000],
  [ 3.0000,  1.0000,  2.0000,  3.0000]],

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

 …,

 [[ 5.0000,  0.0000,  2.0000,  5.0000],
  [10.0000,  1.0000,  3.0000,  8.0000],
  [ 7.0000,  6.0000,  8.0000, 10.0000],
  [ 5.0000,  3.0000,  9.0000,  8.0000]],

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

 [[ 2.0000,  1.0000,  7.0000,  6.0000],
  [ 1.0000,  0.0000,  2.0000,  0.0000],
  [ 0.0000,  0.0000,  1.0000,  0.0000],
  [ 1.0000,  3.0000,  4.0000,  2.0000]]],


…,


[[[ 2.0000,  3.0000,  0.0000,  2.0000],
  [ 0.0000,  0.0000,  0.0000,  0.0000],
  [ 0.0000,  0.0000,  0.0000,  0.0000],
  [ 0.0000,  0.0000,  0.0000,  0.0000]],

 [[ 0.0000,  4.0000,  6.0000,  2.0000],
  [ 1.0000,  4.0000, 10.0000,  4.0000],
  [ 2.0000,  0.0000,  6.0000,  3.0000],
```

```
 [ 0.0000,  0.0000,  3.0000,  2.0000]],

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

…,

[[ 1.0000,  5.0000,  9.0000,  3.0000],
 [ 0.0000,  2.0000, 15.0000,  4.0000],
 [ 5.0000,  8.0000, 15.0000,  8.0000],
 [14.0000, 10.0000, 15.0000,  9.0000]],

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

[[ 0.0000,  0.0000,  0.0000,  0.0000],
 [ 0.0000,  0.0000,  0.0000,  0.0000],
 [ 0.0000,  0.0000,  0.0000,  0.0000],
 [ 3.0000,  5.0000,  5.0000,  0.0000]]],


[[[ 2.0000,  0.0000,  1.0000,  1.0000],
  [ 0.0000,  0.0000,  0.0000,  0.0000],
  [ 0.0000,  0.0000,  0.0000,  0.0000],
  [ 0.0000,  0.0000,  0.0000,  0.0000]],

 [[ 0.0000,  0.0000,  1.0000,  3.0000],
  [ 0.0000,  1.0000,  2.0000,  3.0000],
  [ 0.0000,  0.0000,  0.0000,  0.0000],
  [ 0.0000,  0.0000,  0.0000,  4.0000]],

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

 …,

 [[ 0.0000,  0.0000,  0.0000,  3.0000],
  [ 1.0000,  1.0000,  3.0000,  7.0000],
  [ 4.0000,  1.0000,  0.0000,  0.0000],
  [ 3.0000,  0.0000,  0.0000,  0.0000]],

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

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

        [[ 0.0000,  0.0000,  0.0000,  0.0000],
         [ 0.0000,  0.0000,  1.0000,  2.0000],
         [ 0.0000,  0.0000,  0.0000,  0.0000],
         [ 0.0000,  0.0000,  0.0000,  1.0000]]],


       [[[ 2.0000,  0.0000,  0.0000,  0.0000],
         [ 2.0000,  1.0000,  2.0000,  0.0000],
         [ 3.0000,  3.0000,  1.0000,  0.0000],
         [ 2.0000,  2.0000,  1.0000,  1.0000]],

        [[ 2.0000,  3.0000,  0.0000,  1.0000],
         [ 0.0000,  3.0000,  0.0000,  2.0000],
         [ 1.0000,  6.0000,  1.0000,  4.0000],
         [ 1.0000,  2.0000,  1.0000,  2.0000]],

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

        …,

        [[ 5.0000,  7.0000,  6.0000, 11.0000],
         [ 5.0000,  8.0000,  2.0000, 10.0000],
         [ 6.0000, 14.0000,  9.0000, 12.0000],
         [ 2.0000,  5.0000,  5.0000,  7.0000]],

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

        [[ 0.0000,  4.0000,  5.0000,  5.0000],
         [ 0.0000,  8.0000,  8.0000,  6.0000],
         [ 0.0000,  1.0000,  5.0000,  7.0000],
         [ 0.0000,  0.0000,  0.0000,  2.0000]]]], device='cuda:0',
      grad_fn=<DivBackward0>)
```

[ ]:

[16]:
```
conv_int = torch.nn.Conv2d(in_channels = 8, out_channels=8, kernel_size = 3,␣
↪padding = 1 , bias = False)
```

```
#conv_int = model.features[27]
conv_int.weight = torch.nn.parameter.Parameter(weight_int)

output_int =  conv_int(x_int)
r = torch.nn.ReLU(inplace = True )
output_recovered = r(output_int * w_delta * x_delta)



#(output_recovered)
```

[19]:
```
## COMPARE TO PREHOOK OF NEXT BLOCK
op_ref = save_output.outputs[9][0]  # input of the 2nd conv layer
difference = ( op_ref - output_recovered ).abs().sum()
print(difference ) ## It should be small, e.g.,2.3 in my trainned model
```

```
tensor(0.0006, device='cuda:0', grad_fn=<SumBackward0>)
```

[ ]:

[ ]:

[ ]:

[ ]:

[ ]:

[ ]:

[ ]:

[ ]:

[ ]:

[ ]:

[ ]: