ECE 284 Final Project - Software

Vanilla Training VGGNet with 4 bit activation and weight quantisation

Initialisation Code

```
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 = "VGG16_quant"
model = VGG16_quant()

print(model)

normalize = transforms.Normalize(mean=[0.491, 0.482, 0.447], std=[0.24

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=ba

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=batc

print_freq = 100 # every 100 batches, accuracy printed. Here, each bat
# 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:
```

```
            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('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=los
                       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.t

def adjust_learning_rate(optimizer, epoch):
    """For resnet, the lr starts from 0.1, and is divided by 10 at 80
    adjust_list = [150, 225]
    if epoch in adjust_list:
        for param_group in optimizer.param_groups:
            param_group['lr'] = param_group['lr'] * 0.1
```

```
=> Building model...
VGG_quant(
  (features): Sequential(
    (0): QuantConv2d(
      3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=Fa
      (weight_quant): weight_quantize_fn()
    )
    (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_ru
    (2): ReLU(inplace=True)
    (3): QuantConv2d(
      64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=F
      (weight_quant): weight_quantize_fn()
```

```
    (weight_quant): weight_quantize_fn()
  )
  (4): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_ru
  (5): ReLU(inplace=True)
  (6): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil
  (7): QuantConv2d(
    64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=
    (weight_quant): weight_quantize_fn()
  )
  (8): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_r
  (9): ReLU(inplace=True)
  (10): QuantConv2d(
    128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
    (weight_quant): weight_quantize_fn()
  )
  (11): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_
  (12): ReLU(inplace=True)
  (13): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, cei
  (14): QuantConv2d(
    128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
    (weight_quant): weight_quantize_fn()
  )
  (15): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_
  (16): ReLU(inplace=True)
  (17): QuantConv2d(
    256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
    (weight_quant): weight_quantize_fn()
  )
  (18): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_
  (19): ReLU(inplace=True)
  (20): QuantConv2d(
    256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
    (weight_quant): weight_quantize_fn()
  )
  (21): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_
  (22): ReLU(inplace=True)
  (23): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, cei
  (24): QuantConv2d(
    256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
    (weight_quant): weight_quantize_fn()
  )
  (25): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_
  (26): ReLU(inplace=True)
  (27): QuantConv2d(
    512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
    (weight_quant): weight_quantize_fn()
  )
```

Chokepoint Block and SkipBlock Optimisation

```
# [SEE custom_vgg.py for details on SkipBlock optimisation]
from models.custom_vgg import SkipBlock

use_skip_block = False
```

```
if use_skip_block:
    print("Using SkipBlock optimisation")
    skip_block = SkipBlock()
    model.features[20] = skip_block
    for i in range(21, 33):
        model.features[i] = nn.Identity()
else:
    print("Not using SkipBlock optimisation")
    model.features[24] = QuantConv2d(256, 8, kernel_size=3, padding=1, b
    model.features[25] = nn.BatchNorm2d(8)
    model.features[26] = nn.ReLU(inplace=True)

    model.features[27] = QuantConv2d(8, 8, kernel_size=3, padding=1, bia
    model.features[28] = nn.Identity()
    model.features[29] = nn.ReLU(inplace=True)

    model.features[30] = QuantConv2d(8, 512, kernel_size=3, padding=1, b
    model.features[31] = nn.BatchNorm2d(512)
    model.features[32] = nn.ReLU(inplace=True)

print(model)
```

```
Not using SkipBlock optimisation
VGG_quant(
  (features): Sequential(
    (0): QuantConv2d(
      3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=Fa
      (weight_quant): weight_quantize_fn()
    )
    (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_ru
    (2): ReLU(inplace=True)
    (3): QuantConv2d(
      64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=F
      (weight_quant): weight_quantize_fn()
    )
    (4): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_ru
    (5): ReLU(inplace=True)
    (6): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil
    (7): QuantConv2d(
      64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=
      (weight_quant): weight_quantize_fn()
    )
    (8): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_r
    (9): ReLU(inplace=True)
    (10): QuantConv2d(
      128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
      (weight_quant): weight_quantize_fn()
    )
    (11): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_
    (12): ReLU(inplace=True)
    (13): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, cei
    (14): QuantConv2d(
      128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
      (weight_quant): weight_quantize_fn()
```

```
    )
    (15): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_
    (16): ReLU(inplace=True)
    (17): QuantConv2d(
      256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
      (weight_quant): weight_quantize_fn()
    )
    (18): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_
    (19): ReLU(inplace=True)
    (20): QuantConv2d(
      256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
      (weight_quant): weight_quantize_fn()
    )
    (21): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_
    (22): ReLU(inplace=True)
    (23): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, cei
    (24): QuantConv2d(
      256, 8, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=F
      (weight_quant): weight_quantize_fn()
    )
    (25): BatchNorm2d(8, eps=1e-05, momentum=0.1, affine=True, track_ru
    (26): ReLU(inplace=True)
    (27): QuantConv2d(
      8, 8, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=Fal
      (weight_quant): weight_quantize_fn()
    )
```

Training Loop

```
lr = 1e-2
weight_decay = 1e-5
epochs = 150
best_prec = 0

model.cuda()
criterion = nn.CrossEntropyLoss(label_smoothing=0.1).cuda()
optimizer = torch.optim.SGD(model.parameters(), lr=lr, momentum=0.9, w
scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(optimizer, T_ma

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)
    scheduler.step()

    # evaluate on test set
    print("Validation starts")
```

```
    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)
```

```
Epoch: [0][0/391]       Time 0.939 (0.939)      Data 0.420 (0.420)
Epoch: [0][100/391]     Time 0.049 (0.056)      Data 0.001 (0.005)
Epoch: [0][200/391]     Time 0.047 (0.052)      Data 0.001 (0.003)
Epoch: [0][300/391]     Time 0.048 (0.050)      Data 0.001 (0.003)
Validation starts
Test: [0/79]    Time 0.289 (0.289)      Loss 1.4663 (1.4663)    Prec 56
 * Prec 53.660%
best acc: 53.660000
Epoch: [1][0/391]       Time 0.213 (0.213)      Data 0.181 (0.181)
Epoch: [1][100/391]     Time 0.047 (0.049)      Data 0.001 (0.004)
Epoch: [1][200/391]     Time 0.047 (0.048)      Data 0.001 (0.003)
Epoch: [1][300/391]     Time 0.046 (0.048)      Data 0.001 (0.002)
Validation starts
Test: [0/79]    Time 0.184 (0.184)      Loss 1.2687 (1.2687)    Prec 66
 * Prec 64.730%
best acc: 64.730000
Epoch: [2][0/391]       Time 0.334 (0.334)      Data 0.305 (0.305)
Epoch: [2][100/391]     Time 0.043 (0.050)      Data 0.001 (0.005)
Epoch: [2][200/391]     Time 0.046 (0.049)      Data 0.001 (0.003)
Epoch: [2][300/391]     Time 0.049 (0.048)      Data 0.001 (0.003)
Validation starts
Test: [0/79]    Time 0.195 (0.195)      Loss 1.0513 (1.0513)    Prec 71
 * Prec 74.680%
best acc: 74.680000
Epoch: [3][0/391]       Time 0.377 (0.377)      Data 0.345 (0.345)
Epoch: [3][100/391]     Time 0.048 (0.050)      Data 0.001 (0.005)
Epoch: [3][200/391]     Time 0.047 (0.049)      Data 0.001 (0.003)
Epoch: [3][300/391]     Time 0.050 (0.048)      Data 0.001 (0.002)
Validation starts
Test: [0/79]    Time 0.306 (0.306)      Loss 1.1308 (1.1308)    Prec 75
 * Prec 72.620%
best acc: 74.680000
Epoch: [4][0/391]       Time 0.342 (0.342)      Data 0.313 (0.313)
Epoch: [4][100/391]     Time 0.048 (0.050)      Data 0.001 (0.004)
Epoch: [4][200/391]     Time 0.048 (0.049)      Data 0.001 (0.003)
Epoch: [4][300/391]     Time 0.047 (0.048)      Data 0.001 (0.002)
Validation starts
Test: [0/79]    Time 0.349 (0.349)      Loss 0.9826 (0.9826)    Prec 81
 * Prec 77.030%
best acc: 77.030000
Epoch: [5][0/391]       Time 0.380 (0.380)      Data 0.346 (0.346)
Epoch: [5][100/391]     Time 0.047 (0.050)      Data 0.001 (0.005)
```

```
Epoch: [5][100/391]     Time 0.047 (0.050)      Data 0.001 (0.005)
Epoch: [5][200/391]     Time 0.047 (0.049)      Data 0.001 (0.003)
Epoch: [5][300/391]     Time 0.048 (0.048)      Data 0.001 (0.002)
Validation starts
Test: [0/79]    Time 0.176 (0.176)      Loss 0.9228 (0.9228)    Prec 79
 * Prec 79.610%
best acc: 79.610000
Epoch: [6][0/391]       Time 0.389 (0.389)      Data 0.330 (0.330)
Epoch: [6][100/391]     Time 0.046 (0.050)      Data 0.010 (0.005)
Epoch: [6][200/391]     Time 0.047 (0.049)      Data 0.001 (0.003)
Epoch: [6][300/391]     Time 0.047 (0.048)      Data 0.001 (0.003)
Validation starts
Test: [0/79]    Time 0.186 (0.186)      Loss 0.9262 (0.9262)    Prec 80
 * Prec 78.790%
best acc: 79.610000
Epoch: [7][0/391]       Time 0.380 (0.380)      Data 0.348 (0.348)
Epoch: [7][100/391]     Time 0.046 (0.051)      Data 0.001 (0.005)
```

Testing Loop

```
PATH = "result/VGG16_quant/model_best.pth.tar"
checkpoint = torch.load(PATH)
model.load_state_dict(checkpoint['state_dict'])
device = torch.device("cuda")

model.cuda()
model.eval()

print(model)

test_loss = 0
correct = 0

with torch.no_grad():
    for data, target in testloader:
        data, target = data.to(device), target.to(device) # loading to
        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)))
```

```
VGG_quant(
  (features): Sequential(
    (0): QuantConv2d(
      3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=Fa
      (weight_quant): weight_quantize_fn()
    )
```

```
  (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_ru
  (2): ReLU(inplace=True)
  (3): QuantConv2d(
    64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=F
    (weight_quant): weight_quantize_fn()
  )
  (4): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_ru
  (5): ReLU(inplace=True)
  (6): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil
  (7): QuantConv2d(
    64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=
    (weight_quant): weight_quantize_fn()
  )
  (8): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_r
  (9): ReLU(inplace=True)
  (10): QuantConv2d(
    128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
    (weight_quant): weight_quantize_fn()
  )
  (11): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_
  (12): ReLU(inplace=True)
  (13): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, cei
  (14): QuantConv2d(
    128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
    (weight_quant): weight_quantize_fn()
  )
  (15): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_
  (16): ReLU(inplace=True)
  (17): QuantConv2d(
    256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
    (weight_quant): weight_quantize_fn()
  )
  (18): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_
  (19): ReLU(inplace=True)
  (20): QuantConv2d(
    256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
    (weight_quant): weight_quantize_fn()
  )
  (21): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_
  (22): ReLU(inplace=True)
  (23): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, cei
  (24): QuantConv2d(
    256, 8, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=F
    (weight_quant): weight_quantize_fn()
  )
  (25): BatchNorm2d(8, eps=1e-05, momentum=0.1, affine=True, track_ru
  (26): ReLU(inplace=True)
  (27): QuantConv2d(
    8, 8, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=Fal
    (weight_quant): weight_quantize_fn()
  )
  (28): Identity()
```

Weight and Activation recovery verification

Getting the right target layer and attaching hooks for fwd pass

```
if use_skip_block:
    target_layer = model.features[20].conv27
    next_layer = model.features[20].bn28
else:
    target_layer = model.features[27]
    next_layer = model.features[28]

# some sanity checking to ensure we're extracting the right layer
assert isinstance(target_layer, QuantConv2d)

input_records = {}
def get_input_hook(name):
    def hook(model, input, output):
        input_records[name] = input[0].detach().clone()
    return hook

target_layer.register_forward_hook(get_input_hook('conv8x8_input'))
next_layer.register_forward_hook(get_input_hook('next_layer_input'))

# Running a sample forward pass to extract inputs
data, _ = next(iter(testloader))
data = data.cuda()
model = model.cuda()
model.eval()
with torch.no_grad():
    output = model(data)
conv8x8_input = input_records['conv8x8_input'].cpu().numpy()
print(f"Captured Transient Input for 8x8 layer: {conv8x8_input}")

target_layer = target_layer.cpu()
```

```
Captured Transient Input for 8x8 layer: [[[[0.7528543  1.7992511  1.746
   [0.         0.26089427 0.         0.        ]
   [0.         0.05128749 0.6869564  0.42997885]
   [0.         0.         0.         0.        ]]

  [[0.12536542 1.0985982  0.97800106 0.8837508 ]
   [0.12086887 1.5830147  0.19548273 0.        ]
   [0.93421614 1.587489   0.32703155 0.        ]
   [0.         0.30065754 0.         0.        ]]

  [[0.         0.         0.         0.04210338]
   [0.         0.         0.         0.2601679 ]
   [0.         0.30907613 1.5854645  1.0959355 ]
   [0.         0.18303858 0.48659316 1.2263494 ]]

  ...

  [[0.         0.         0.         0.5830591 ]
```

```
  [[0.         0.         0.         0.5830591 ]
   [0.         0.         0.56545293 1.357479  ]
   [0.         0.         0.         0.47578564]
   [0.         0.         0.         0.        ]]

  [[0.541478   0.19551845 0.         0.7155491 ]
   [0.4184445  0.         0.         0.        ]
   [0.         0.         0.         0.        ]
   [0.45869854 0.         0.         0.11493356]]

  [[0.36285886 1.3648515  1.1481204  0.49288452]
   [0.         0.         0.         0.        ]
   [0.         0.         0.         0.        ]
   [0.         0.         0.         0.        ]]]


 [[[0.         0.03617152 0.32693967 0.5940028 ]
   [0.         0.         0.01565718 0.        ]
   [0.         0.         0.         0.        ]
   [0.         0.         0.         0.        ]]

  [[0.38401628 0.12330111 0.         0.        ]
   [0.7368569  0.45192885 0.         0.        ]
   [0.56377393 0.2985332  0.         0.        ]
   [0.         0.         0.         0.        ]]

  [[0.         0.8330142  1.5854013  1.0274963 ]
   [0.03767974 1.3031905  2.9192286  2.3465035 ]
   [0.         0.         0.92690575 1.8312359 ]
   [0.         0.         0.         1.2882979 ]]

  ...

  [[0.         0.         0.         0.        ]
   [0.         0.         0.         0.        ]
   [0.         0.         0.         0.        ]
   [0.         0.         0.         0.1870321 ]]

  [[0.9044368  0.03007763 0.363258   0.87050474]
   [0.25070333 0.         0.         0.        ]
   [0.         0.         0.         0.        ]
```

Extracting weights for 8x8 layer

```
w_bit = 4
weight_float_q = target_layer.weight_q
w_alpha = target_layer.weight_quant.wgt_alpha
w_delta = w_alpha / (2**(w_bit - 1)-1)

weight_int = (weight_float_q / w_delta)
print(f"Sample Weights (check if they're all integers for sanity): {we

weight_int = weight_int.round()
print(f"Sample Weights after rounding: {weight_int[0][0][:][:]}")
```

```
print("Sample weights after rounding:", weight_int[0][0][:,:])
```

```
Sample Weights (check if they're all integers for sanity): tensor([[-1.
        [-3.0000, -1.0000, -1.0000],
        [ 1.0000, -3.0000, -1.0000]], grad_fn=<SliceBackward0>)
Sample Weights after rounding: tensor([[-1., -1., -2.],
        [-3., -1., -1.],
        [ 1., -3., -1.]], grad_fn=<SliceBackward0>)
```

```
WGT_BITS = 4

weight_cpu = weight_int.detach().cpu().clone()  # [8, 8, 3, 3]
OC, IC, kH, kW = weight_cpu.shape
print("weight shape:", weight_cpu.shape)

assert OC == 8 and IC == 8 and kH == 3 and kW == 3

# col in your Verilog should be 8 (same as IC)
col = IC

# Enumerate kernel positions in some order; you can change naming if n
kernel_positions = [(i, j) for i in range(3) for j in range(3)]

for idx, (kh, kw) in enumerate(kernel_positions):
    # Get the 8x8 weight matrix for this kernel tap
    # W_k[oc, ic] = weight for (out_channel, in_channel) at this (kh,k
    W_k = weight_cpu[:, :, kh, kw]   # [8, 8]

    filename = f"weight_itile0_otile0_kij{idx}.txt"   # k0..k8; rename
    with open(filename, "w") as f:
        # No header lines here, because your Verilog uses "%32b" direc
        # If you add comments, you must skip them in Verilog.

        # ic = input channel index, corresponds to t in Verilog loop
        for oc in range(OC):          # t = 0..7
            # Build one 32-bit word: row7..row0, each 4 bits
            line_bits = []
            for ic in range(IC):       # rows
                val = int(W_k[7 - oc, ic].item())  # row7 first, down

                # Mask into 4 bits (unsigned view)
                val &= (1 << WGT_BITS) - 1
                bits = f"{val:0{WGT_BITS}b}"       # "0000".."1111"
                line_bits.append(bits)

            # Concatenate into 32-bit string
            f.write("".join(line_bits) + "\n")

    print(f"Saved {filename} for kernel tap (kh={kh}, kw={kw})")
```

```
weight shape: torch.Size([8, 8, 3, 3])
```

```
Saved weight_itile0_otile0_kij0.txt for kernel tap (kh=0, kw=0)
Saved weight_itile0_otile0_kij1.txt for kernel tap (kh=0, kw=1)
Saved weight_itile0_otile0_kij2.txt for kernel tap (kh=0, kw=2)
Saved weight_itile0_otile0_kij3.txt for kernel tap (kh=1, kw=0)
Saved weight_itile0_otile0_kij4.txt for kernel tap (kh=1, kw=1)
Saved weight_itile0_otile0_kij5.txt for kernel tap (kh=1, kw=2)
Saved weight_itile0_otile0_kij6.txt for kernel tap (kh=2, kw=0)
Saved weight_itile0_otile0_kij7.txt for kernel tap (kh=2, kw=1)
Saved weight_itile0_otile0_kij8.txt for kernel tap (kh=2, kw=2)
```

Extracting Inputs for 8x8 layer

```
x_bit = 4
x = conv8x8_input
x_alpha = target_layer.act_alpha.detach()
x_delta = x_alpha / (2**x_bit - 1)

x_div = x / x_alpha
x_clamped = torch.clamp(x_div, 0, 1)
x_int = (x_clamped * (2**x_bit - 1)).round()

print(f"Sample Inputs after quantization: {x_int[0][0][:][:]}")
```

```
Sample Inputs after quantization: tensor([[1., 3., 3., 3.],
        [0., 1., 0., 0.],
        [0., 0., 1., 1.],
        [0., 0., 0., 0.]])
```

```
ACT_BITS = 4 # 4-bit activations

# Choose which sample in the batch you want to export
sample_id = 0   # 0..127
x_int_cpu = x_int.detach().cpu().clone()      # [128, 8, 4, 4]

# Take one sample: shape [8, 4, 4]
x_int_sample = x_int_cpu[sample_id]           # [C=8, H=4, W=4]

C, H, W = x_int_sample.shape
print("single sample shape:", x_int_sample.shape)
assert C == 8, f"Expected 8 channels, got {C}"
assert H * W == 16, f"Expected 4x4 spatial, got {H}x{W}"

# Flatten spatial into time: [rows=8, timesteps=16]
# Here: each channel → one systolic row, 16 time steps
X = x_int_sample.reshape(C, H * W)            # [8, 16]

with open('activation.txt', 'w') as f:
    f.write('#time0row7[msb-lsb],time0row6[msb-lsb],....,time0row0[msb
    f.write('#time1row7[msb-lsb],time1row6[msb-lsb],....,time1row0[msb
    f.write('#...............#\n')
```

```
    # i: time step, j: row index
    for i in range(X.size(1)):       # time steps (0..15)
        for j in range(X.size(0)):   # row index (0..7)
            X_bin = '{0:04b}'.format(round(X[7-j,i].item()))
            #val = int(round(X[7 - j, i].item()))   # write row7..row0

            # If you're treating activations as unsigned 4-bit:
            #val = val & ((1 << ACT_BITS) - 1)

            #X_bin = f'{val:0{ACT_BITS}b}'         # "0000".."1111"
            for k in range(ACT_BITS):
                f.write(X_bin[k])
        f.write('\n')

print("Saved activation.txt")
```

```
single sample shape: torch.Size([8, 4, 4])
Saved activation.txt
```

Verifying Psum Integrity with reference psum

```
conv_int = nn.Conv2d(8, 8, kernel_size=3, padding=1, bias=False)
conv_int.weight = nn.Parameter(weight_int.float())
psum_int = conv_int(torch.tensor(x_int))
output_recovered = psum_int * x_delta * w_delta

act_fn = act_quantization(b=4)
x_model_q = act_fn(torch.tensor(x), torch.tensor(x_alpha))
output_ref = F.conv2d(torch.tensor(x_model_q), torch.tensor(target_lay

difference = torch.abs(output_recovered - output_ref)
print(f"Mean Absolute Error: {difference.mean().item()}")
if difference.mean().item() < 1e-3:
    print("Success... Psum integrity verified!")
else:
    print("Error too high... Psum integrity failed.")
```

```
/tmp/ipykernel_346/2530370086.py:3: UserWarning: To copy construct from
  psum_int = conv_int(torch.tensor(x_int))
/tmp/ipykernel_346/2530370086.py:7: UserWarning: To copy construct from
  x_model_q = act_fn(torch.tensor(x), torch.tensor(x_alpha))
Mean Absolute Error: 9.626764949643984e-07
Success... Psum integrity verified!
/tmp/ipykernel_346/2530370086.py:8: UserWarning: To copy construct from
  output_ref = F.conv2d(torch.tensor(x_model_q), torch.tensor(target_la
```

```
print("x_int shape:", x_int.shape)
print("weight_int shape:", weight_int.shape)
```

```
print("psum_int shape:", psum_int.shape)
```

```
PSUM_BITS = 16

sample_id = 0
p_sample = psum_int[sample_id]      # [8, 4, 4]
C, H, W = p_sample.shape
assert C == 8 and H == 4 and W == 4

# Flatten spatial dimension
P = p_sample.reshape(C, H*W)        # [8, 16]

with open("psum.txt", "w") as f:
    f.write("#time0row7[msb-lsb],time0row6[msb-lsb],...,time0row0[msb-
    f.write("#.............................#\n")

    for t in range(P.size(1)):      # 16 time steps
        for r in range(P.size(0)):  # 8 rows
            val = int(P[7 - r, t].item())
            val &= (1 << PSUM_BITS) - 1

            bits = f"{val:0{PSUM_BITS}b}"
            f.write(bits)
        f.write("\n")

print("Saved psum.txt")
```

```
Saved psum.txt
```

```
PSUM_BITS = 16   # or match your RTL width

sample_id = 0
p_sample = psum_int[sample_id]      # shape [8,4,4]

C, H, W = p_sample.shape
assert C == 8 and H == 4 and W == 4

# Flatten spatial → time
P = p_sample.reshape(C, H*W)    # [8, 16]

with open("psum_relu.txt", "w") as f:
    f.write("#ReLUed psum output\n")
    f.write("#time0row7[msb-lsb],time0row6[msb-lsb],...,row0#\n")
    f.write("#.................................#\n")

    for t in range(P.size(1)):      # 16 timesteps
        for r in range(P.size(0)):  # 8 rows
            val = int(P[7-r,t].item())
```

```
                # Apply ReLU
                val = max(0, val)

                # Fit into chosen bit width
                val &= (1 << PSUM_BITS) - 1

                bits = f"{val:0{PSUM_BITS}b}"
                f.write(bits)

            f.write("\n")

    print("Saved psum_relu.txt")
```

```
Saved psum_relu.txt
```

```
PSUM_BITS = 16
sample_id = 0

x_sample = x_int[sample_id].detach().cpu().clone()      # [8,4,4]
w_cpu    = weight_int.detach().cpu().clone()            # [8,8,3,3]

OC, IC, kH, kW = w_cpu.shape
C, H, W = x_sample.shape
assert H==4 and W==4 and C==8

kernel_positions = [(i,j) for i in range(3) for j in range(3)]

# pad input (padding=1)
x_pad = torch.zeros((C, H+2, W+2), dtype=torch.int32)
x_pad[:,1:H+1,1:W+1] = x_sample

for idx, (kh, kw) in enumerate(kernel_positions):
    # Extract weight matrix for this tap
    W_k = w_cpu[:,:,kh,kw]   # [8,8]

    # Output psum for this kernel (8,4,4)
    psum_k = torch.zeros((OC,H,W), dtype=torch.int32)

    # Compute per-tap convolution manually
    for oc in range(OC):
        for ic in range(IC):
            w_val = int(W_k[oc, ic].item())
            for h in range(H):
                for w in range(W):
                    # shifted input index = (h + kh, w + kw)
                    xi = x_pad[ic, h + kh, w + kw].item()
                    psum_k[oc, h, w] += xi * w_val

    # Now dump as file with 16-bit mask
    filename = f"psum_{idx}.txt"
```

```
        filename = f"psum_{idx}.txt"
        with open(filename, "w") as f:
            for t in range(H*W):         # 16 timesteps
                h = t // W
                w = t %  W

                line_bits = []
                for oc in range(OC):    # row7..row0
                    val = int(psum_k[7 - oc, h, w].item())
                    val &= (1 << PSUM_BITS) - 1
                    bits = f"{val:0{PSUM_BITS}b}"
                    line_bits.append(bits)

                f.write("".join(line_bits) + "\n")

        print("Saved", filename)
```

```
Saved psum_0.txt
Saved psum_1.txt
Saved psum_2.txt
Saved psum_3.txt
Saved psum_4.txt
Saved psum_5.txt
Saved psum_6.txt
Saved psum_7.txt
Saved psum_8.txt
```

Verifying Psum Integrity with next layer input

```
difference2 = torch.abs(input_records['next_layer_input'].cpu() - outp
print(f"Mean Absolute Error with next layer input: {difference2.mean()
if difference2.mean().item() < 1e-3:
    print("Success... Psum integrity verified!")
else:
    print("Error too high... Psum integrity failed.")
```

```
Mean Absolute Error with next layer input: 1.4162951629259624e-06
Success... Psum integrity verified!
```

ECE 284 Final Project - Software

Vanilla Training VGGNet with 2 bit activation and 4 bit weight quantisation

Initialisation Code

```
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 = "VGG16_quant"
model = VGG16_quant(wbit=4, abit=2)

print(model)

normalize = transforms.Normalize(mean=[0.491, 0.482, 0.447], std=[0.24

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=ba

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=batc
```

```
print_freq = 100 # every 100 batches, accuracy printed. Here, each bat
# 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('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=los
                   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.t

def adjust_learning_rate(optimizer, epoch):
    """For resnet, the lr starts from 0.1, and is divided by 10 at 80
    adjust_list = [150, 225]
    if epoch in adjust_list:
        for param_group in optimizer.param_groups:
            param_group['lr'] = param_group['lr'] * 0.1
```

```
=> Building model...
VGG_quant(
  (features): Sequential(
    (0): QuantConv2d(
      3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=Fa
      (weight_quant): weight_quantize_fn()
    )
    (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_ru
    (2): ReLU(inplace=True)
    (3): QuantConv2d(
      64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=F
      (weight_quant): weight_quantize_fn()
    )
    (4): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_ru
    (5): ReLU(inplace=True)
    (6): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil
    (7): QuantConv2d(
      64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=
      (weight_quant): weight_quantize_fn()
    )
    (8): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_r
    (9): ReLU(inplace=True)
    (10): QuantConv2d(
      128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
      (weight_quant): weight_quantize_fn()
    )
```

```
,
(11): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_
(12): ReLU(inplace=True)
(13): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, cei
(14): QuantConv2d(
  128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
  (weight_quant): weight_quantize_fn()
)
(15): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_
(16): ReLU(inplace=True)
(17): QuantConv2d(
  256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
  (weight_quant): weight_quantize_fn()
)
(18): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_
(19): ReLU(inplace=True)
(20): QuantConv2d(
  256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
  (weight_quant): weight_quantize_fn()
)
(21): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_
(22): ReLU(inplace=True)
(23): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, cei
(24): QuantConv2d(
  256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
  (weight_quant): weight_quantize_fn()
)
(25): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_
(26): ReLU(inplace=True)
(27): QuantConv2d(
  512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
  (weight_quant): weight_quantize_fn()
)
```

Loading Pre trained w4a4 model for faster convergence

```
def load_checkpoint_special(model, checkpoint_path):
    print(f"=> Loading checkpoint '{checkpoint_path}'")
    checkpoint = torch.load(checkpoint_path)
    pretrained_dict = checkpoint['state_dict']
    model_dict = model.state_dict()
    pretrained_dict_filtered = {}
    for k, v in pretrained_dict.items():
        if k in model_dict:
            if model_dict[k].shape == v.shape:
                pretrained_dict_filtered[k] = v
            else:
                print(f"\tSkipping layer '{k}': Shape mismatch. "
                    f"New: {model_dict[k].shape} vs Old: {v.shape}")
        else:
            print(f"\tSkipping layer '{k}': Not found in new model.")
    model_dict.update(pretrained_dict_filtered)
    model.load_state_dict(model_dict)
```

```
        print("=> Partial load complete. Mismatched layers initialized ran

    load_checkpoint_special(model, './result/VGG16_4a4w_quant_noskipblock/
```

```
    => Loading checkpoint './result/VGG16_4a4w_quant_noskipblock/checkpoint
    /tmp/ipykernel_1037538/1623649740.py:3: FutureWarning: You are using `t
      checkpoint = torch.load(checkpoint_path)
            Skipping layer 'features.24.weight': Shape mismatch. New: torch
            Skipping layer 'features.24.weight_q': Shape mismatch. New: tor
            Skipping layer 'features.25.weight': Shape mismatch. New: torch
            Skipping layer 'features.25.bias': Shape mismatch. New: torch.S
            Skipping layer 'features.25.running_mean': Shape mismatch. New:
            Skipping layer 'features.25.running_var': Shape mismatch. New:
            Skipping layer 'features.27.weight': Shape mismatch. New: torch
            Skipping layer 'features.27.weight_q': Shape mismatch. New: tor
            Skipping layer 'features.30.weight': Shape mismatch. New: torch
            Skipping layer 'features.30.weight_q': Shape mismatch. New: tor
    => Partial load complete. Mismatched layers initialized randomly.
```

Chokepoint Block and SkipBlock Optimisation

```
    # [SEE custom_vgg.py for details on SkipBlock optimisation]
    from models.custom_vgg import SkipBlock

    use_skip_block = False

    model_name = "VGG16_2a4w_quant_skipblock" if use_skip_block else "VGG1

    if use_skip_block:
        print("Using SkipBlock optimisation")
        skip_block = SkipBlock()
        model.features[20] = skip_block
        for i in range(21, 33):
            model.features[i] = nn.Identity()
    else:
        print("Not using SkipBlock optimisation")
        model.features[24] = QuantConv2d(256, 16, kernel_size=3, padding=1
        model.features[25] = nn.BatchNorm2d(16)
        model.features[26] = nn.ReLU(inplace=True)

        model.features[27] = QuantConv2d(16, 16, kernel_size=3, padding=1,
        model.features[28] = nn.Identity()
        model.features[29] = nn.ReLU(inplace=True)

        model.features[30] = QuantConv2d(16, 512, kernel_size=3, padding=1
        model.features[31] = nn.BatchNorm2d(512)
        model.features[32] = nn.ReLU(inplace=True)

    print(model)
```

```
    Not using SkipBlock optimisation
```

```
Not using SkipBlock optimisation
VGG_quant(
  (features): Sequential(
    (0): QuantConv2d(
      3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=Fa
      (weight_quant): weight_quantize_fn()
    )
    (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_ru
    (2): ReLU(inplace=True)
    (3): QuantConv2d(
      64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=F
      (weight_quant): weight_quantize_fn()
    )
    (4): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_ru
    (5): ReLU(inplace=True)
    (6): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil
    (7): QuantConv2d(
      64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=
      (weight_quant): weight_quantize_fn()
    )
    (8): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_r
    (9): ReLU(inplace=True)
    (10): QuantConv2d(
      128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
      (weight_quant): weight_quantize_fn()
    )
    (11): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_
    (12): ReLU(inplace=True)
    (13): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, cei
    (14): QuantConv2d(
      128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
      (weight_quant): weight_quantize_fn()
    )
    (15): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_
    (16): ReLU(inplace=True)
    (17): QuantConv2d(
      256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
      (weight_quant): weight_quantize_fn()
    )
    (18): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_
    (19): ReLU(inplace=True)
    (20): QuantConv2d(
      256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias
      (weight_quant): weight_quantize_fn()
    )
    (21): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_
    (22): ReLU(inplace=True)
    (23): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, cei
    (24): QuantConv2d(
      256, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=
      (weight_quant): weight_quantize_fn()
    )
    (25): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_r
    (26): ReLU(inplace=True)
    (27): QuantConv2d(
      16, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=F
```

```
        (weight_quant): weight_quantize_fn()
    )
```

Training Loop

```
lr = 1e-2
weight_decay = 1e-5
epochs = 300
best_prec = 0

model.cuda()

all_params = model.named_parameters()
params_main = []
params_quant = []

for name, param in all_params:
    if 'alpha' in name:
        params_quant.append(param)
    else:
        params_main.append(param)

criterion = nn.CrossEntropyLoss(label_smoothing=0.1).cuda()
optimizer = torch.optim.SGD([
    {'params': params_main, 'weight_decay': weight_decay, 'lr': lr},
    {'params': params_quant, 'weight_decay': 0.0, 'lr': lr * 0.1}
], momentum=0.9)
scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(optimizer, T_ma

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)
    scheduler.step()

    # 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)
        # if best_prec > 87.0:
        #     print("Achieved accuracy greater than 87%, early stopping.")
        #     break
```

```
Epoch: [0][0/391]       Time 0.174 (0.174)      Data 0.130 (0.130)
Epoch: [0][100/391]     Time 0.035 (0.034)      Data 0.002 (0.003)
Epoch: [0][200/391]     Time 0.034 (0.034)      Data 0.002 (0.003)
Epoch: [0][300/391]     Time 0.034 (0.033)      Data 0.002 (0.002)
Validation starts
Test: [0/79]    Time 0.114 (0.114)      Loss 1.4094 (1.4094)    Prec 60
 * Prec 61.730%
best acc: 61.730000
Epoch: [1][0/391]       Time 0.189 (0.189)      Data 0.149 (0.149)
Epoch: [1][100/391]     Time 0.030 (0.035)      Data 0.002 (0.004)
Epoch: [1][200/391]     Time 0.036 (0.034)      Data 0.002 (0.003)
Epoch: [1][300/391]     Time 0.027 (0.033)      Data 0.002 (0.003)
Validation starts
Test: [0/79]    Time 0.109 (0.109)      Loss 1.1909 (1.1909)    Prec 70
 * Prec 67.380%
best acc: 67.380000
Epoch: [2][0/391]       Time 0.166 (0.166)      Data 0.127 (0.127)
Epoch: [2][100/391]     Time 0.035 (0.034)      Data 0.001 (0.003)
Epoch: [2][200/391]     Time 0.038 (0.034)      Data 0.002 (0.003)
Epoch: [2][300/391]     Time 0.030 (0.033)      Data 0.002 (0.002)
Validation starts
Test: [0/79]    Time 0.116 (0.116)      Loss 1.1769 (1.1769)    Prec 67
 * Prec 68.710%
best acc: 68.710000
Epoch: [3][0/391]       Time 0.183 (0.183)      Data 0.137 (0.137)
Epoch: [3][100/391]     Time 0.033 (0.034)      Data 0.002 (0.003)
Epoch: [3][200/391]     Time 0.032 (0.034)      Data 0.002 (0.003)
Epoch: [3][300/391]     Time 0.035 (0.034)      Data 0.001 (0.003)
Validation starts
Test: [0/79]    Time 0.113 (0.113)      Loss 1.1819 (1.1819)    Prec 71
 * Prec 69.920%
best acc: 69.920000
Epoch: [4][0/391]       Time 0.172 (0.172)      Data 0.129 (0.129)
Epoch: [4][100/391]     Time 0.033 (0.035)      Data 0.002 (0.003)
Epoch: [4][200/391]     Time 0.036 (0.034)      Data 0.002 (0.003)
Epoch: [4][300/391]     Time 0.033 (0.034)      Data 0.003 (0.002)
Validation starts
Test: [0/79]    Time 0.106 (0.106)      Loss 1.0431 (1.0431)    Prec 78
 * Prec 72.340%
best acc: 72.340000
Epoch: [5][0/391]       Time 0.167 (0.167)      Data 0.128 (0.128)
Epoch: [5][100/391]     Time 0.030 (0.035)      Data 0.003 (0.003)
Epoch: [5][200/391]     Time 0.032 (0.034)      Data 0.002 (0.003)
Epoch: [5][300/391]     Time 0.035 (0.033)      Data 0.003 (0.003)
Validation starts
Test: [0/79]    Time 0.093 (0.093)      Loss 1.1015 (1.1015)    Prec 75
 * Prec 72.480%
```

```
best acc: 72.480000
Epoch: [6][0/391]       Time 0.166 (0.166)      Data 0.129 (0.129)
Epoch: [6][100/391]     Time 0.030 (0.035)      Data 0.002 (0.003)
Epoch: [6][200/391]     Time 0.038 (0.034)      Data 0.003 (0.003)
Epoch: [6][300/391]     Time 0.033 (0.033)      Data 0.001 (0.003)
Validation starts
Test: [0/79]    Time 0.105 (0.105)      Loss 1.0625 (1.0625)    Prec 77
 * Prec 74.000%
best acc: 74.000000
Epoch: [7][0/391]       Time 0.175 (0.175)      Data 0.136 (0.136)
Epoch: [7][100/391]     Time 0.037 (0.033)      Data 0.002 (0.003)
```

Testing Loop

```
PATH = "result/VGG16_2a4w_quant_noskipblock/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
        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)))
```

```
/tmp/ipykernel_1037538/3160766666.py:2: FutureWarning: You are using `t
  checkpoint = torch.load(PATH)

Test set: Accuracy: 8802/10000 (88%)
```

Weight and Activation recovery verification

Getting the right target layer and attaching hooks for fwd pass

```
if use_skip_block:
    target_layer = model.features[20].conv27
    next_layer = model.features[20].bn28
else:
    target_layer = model.features[27]
    next_layer = model.features[28]

# some sanity checking to ensure we're extracting the right layer
assert isinstance(target_layer, QuantConv2d)

input_records = {}
def get_input_hook(name):
    def hook(model, input, output):
        input_records[name] = input[0].detach().clone()
    return hook

target_layer.register_forward_hook(get_input_hook('conv16x16_input'))
next_layer.register_forward_hook(get_input_hook('next_layer_input'))

# Running a sample forward pass to extract inputs
data, _ = next(iter(testloader))
data = data.cuda()
model = model.cuda()
model.eval()
with torch.no_grad():
    output = model(data)
conv16x16_input = input_records['conv16x16_input'].cpu().numpy()
print(f"Captured Transient Input for 16x16 layer: {conv16x16_input}")

target_layer = target_layer.cpu()
```

```
Captured Transient Input for 16x16 layer: [[[[1.23831189e+00 0.00000000
   [0.00000000e+00 0.00000000e+00 6.71324193e-01 0.00000000e+00]
   [0.00000000e+00 1.32691607e-01 1.76276970e+00 7.28025019e-01]
   [5.15404284e-01 1.03986478e+00 1.73442006e+00 1.47927773e+00]]

  [[1.89826801e-01 0.00000000e+00 0.00000000e+00 0.00000000e+00]
   [0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00]
   [0.00000000e+00 1.77257344e-01 4.03556108e-01 1.32132292e+00]
   [0.00000000e+00 0.00000000e+00 4.41273540e-01 0.00000000e+00]]

  [[1.32021332e+00 1.73326778e+00 2.71155310e+00 3.33113289e+00]
   [0.00000000e+00 1.26586592e+00 2.74416304e+00 3.22243619e+00]
   [0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00]
   [6.13675117e-01 0.00000000e+00 1.01585865e+00 1.68978882e+00]]

  ...

  [[0.00000000e+00 7.08765149e-01 1.70264459e+00 9.02694762e-01]
   [2.22382259e+00 4.81758881e+00 6.25991678e+00 4.12672472e+00]
   [3.03589177e+00 3.92067838e+00 6.18719578e+00 4.76910830e+00]
   [4.17880744e-01 1.49659550e+00 2.19957924e+00 9.39056873e-01]]
```

```
  [[9.05456662e-01 1.02590010e-01 3.28392297e-01 8.05088580e-01]
   [7.92542458e-01 0.00000000e+00 0.00000000e+00 0.00000000e+00]
   [0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00]
   [0.00000000e+00 0.00000000e+00 6.49496689e-02 0.00000000e+00]]

  [[0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00]
   [0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00]
   [0.00000000e+00 2.19504178e-01 0.00000000e+00 7.39737689e-01]
   [0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00]]]


 [[[0.00000000e+00 0.00000000e+00 6.14625454e-01 0.00000000e+00]
   [1.45093071e+00 2.42897654e+00 2.41480136e+00 3.02786738e-01]
   [2.06043863e+00 4.14410114e+00 3.39284849e+00 1.18517034e-01]
   [2.10296106e+00 3.67634010e+00 2.85421348e+00 1.53597653e+00]]

  [[0.00000000e+00 1.64682180e-01 1.32131982e+00 9.56728280e-01]
   [1.13274086e+00 7.17859447e-01 1.24589002e+00 4.03555989e-01]
   [4.03556913e-01 1.12016833e+00 6.92715287e-01 2.63913665e-02]
   [0.00000000e+00 0.00000000e+00 1.64685622e-01 0.00000000e+00]]

  [[7.01814219e-02 0.00000000e+00 0.00000000e+00 0.00000000e+00]
   [0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00]
   [0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00]
   [9.19203609e-02 0.00000000e+00 0.00000000e+00 4.72366720e-01]]

  ...

  [[6.48173094e-01 0.00000000e+00 0.00000000e+00 0.00000000e+00]
   [0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00]
   [0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00]
   [0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00]]

  [[0.00000000e+00 0.00000000e+00 5.54193497e-01 1.84629440e+00]
   [3.66022855e-01 1.90395907e-01 0.00000000e+00 9.93255615e-01]
   [0.00000000e+00 0.00000000e+00 0.00000000e+00 0.00000000e+00]
```

Extracting weights for 16x16 layer

```
w_bit = 4
weight_float_q = target_layer.weight_q
w_alpha = target_layer.weight_quant.wgt_alpha
w_delta = w_alpha / (2**(w_bit - 1)-1)

weight_int = (weight_float_q / w_delta)
print(f"Sample Weights (check if they're all integers for sanity): {we

weight_int = weight_int.round()
print(f"Sample Weights after rounding: {weight_int[0][0][:][:]}")
```

```
Sample Weights (check if they're all integers for sanity): tensor([[-0.
        [1., 2., 2.],
        [3., 3., 2.]], grad_fn=<SliceBackward0>)
```

```
Sample Weights after rounding: tensor([[-0., 3., 2.],
        [1., 2., 2.],
        [3., 3., 2.]], grad_fn=<SliceBackward0>)
```

Extracting Inputs for 16x16 layer

```
x_bit = 2
x = conv16x16_input
x_alpha = target_layer.act_alpha.detach()
x_delta = x_alpha / (2**x_bit - 1)

x_div = x / x_alpha
x_clamped = torch.clamp(x_div, 0, 1)
x_int = (x_clamped * (2**x_bit - 1)).round()

print(f"Sample Inputs after quantization: {x_int[0][0][:][:]}")
```

```
Sample Inputs after quantization: tensor([[1., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 1., 0.],
        [0., 1., 1., 1.]])
```

Verifying Psum Integrity with reference psum

```
conv_int = nn.Conv2d(16, 16, kernel_size=3, padding=1, bias=False)
conv_int.weight = nn.Parameter(weight_int.float())
psum_int = conv_int(torch.tensor(x_int))
output_recovered = psum_int * x_delta * w_delta

act_fn = act_quantization(b=2)
x_model_q = act_fn(torch.tensor(x), torch.tensor(x_alpha))
output_ref = F.conv2d(torch.tensor(x_model_q), torch.tensor(target_lay

difference = torch.abs(output_recovered - output_ref)
print(f"Mean Absolute Error: {difference.mean().item()}")
if difference.mean().item() < 1e-3:
    print("Success... Psum integrity verified!")
else:
    print("Error too high... Psum integrity failed.")
```

```
Mean Absolute Error: 1.5899438494670903e-06
Success... Psum integrity verified!
/tmp/ipykernel_1037538/3644988574.py:3: UserWarning: To copy construct
  psum_int = conv_int(torch.tensor(x_int))
/tmp/ipykernel_1037538/3644988574.py:7: UserWarning: To copy construct
  x_model_q = act_fn(torch.tensor(x), torch.tensor(x_alpha))
/tmp/ipykernel_1037538/3644988574.py:8: UserWarning: To copy construct
  output_ref = F.conv2d(torch.tensor(x_model_q), torch.tensor(target_la
```

Verifying Psum Integrity with next layer input

```
difference2 = torch.abs(input_records['next_layer_input'].cpu() - outp
print(f"Mean Absolute Error with next layer input: {difference2.mean()
if difference2.mean().item() < 1e-3:
    print("Success... Psum integrity verified!")
else:
    print("Error too high... Psum integrity failed.")
```

```
Mean Absolute Error with next layer input: 1.6895540966288536e-06
Success... Psum integrity verified!
```