hw6

November 12, 2025

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

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=[0.247, 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 = [150, 225]
    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, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
      (weight_quant): weight_quantize_fn()
    )
    (25): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (26): ReLU(inplace=True)
    (27): QuantConv2d(
      512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
      (weight_quant): weight_quantize_fn()
    )
    (28): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
```

```
    (29): ReLU(inplace=True)
    (30): QuantConv2d(
      512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
      (weight_quant): weight_quantize_fn()
    )
    (31): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (32): ReLU(inplace=True)
    (33): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
    (34): QuantConv2d(
      512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
      (weight_quant): weight_quantize_fn()
    )
    (35): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (36): ReLU(inplace=True)
    (37): QuantConv2d(
      512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
      (weight_quant): weight_quantize_fn()
    )
    (38): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (39): ReLU(inplace=True)
    (40): QuantConv2d(
      512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
      (weight_quant): weight_quantize_fn()
    )
    (41): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (42): ReLU(inplace=True)
    (43): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
    (44): 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
```

[2]:
```python
# train the VGGG quant model
print('GPU: ', use_gpu)
print('=> Building model...')
model_name = "VGG16_quant"

lr = 1e-3
weight_decay = 1e-5
```

```
epochs = 80
best_prec = 0

model.cuda()
criterion = nn.CrossEntropyLoss().cuda()
optimizer = torch.optim.SGD(model.parameters(), lr=lr, momentum=0.85,␣
 ↪weight_decay=weight_decay)

fdir = 'result/'+str(model_name)
if not os.path.exists('result'):
    os.makedirs('result')
if not os.path.exists(fdir):
    os.makedirs(fdir)
else:
    checkpoint = torch.load(fdir+ '/model_best.pth.tar')
    model.load_state_dict(checkpoint['state_dict'])

# 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)
```

```
GPU:  True
=> Building model…
```

[3]:
```
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()

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: 8745/10000 (87%)
```

[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()
i = 0

for layer in model.modules():
    i = i+1
    if isinstance(layer, QuantConv2d):
        print(i,"-th layer prehooked")
        layer.register_forward_pre_hook(save_output)
####################################################

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

```
3 -th layer prehooked
7 -th layer prehooked
```

```
12 -th layer prehooked
16 -th layer prehooked
21 -th layer prehooked
25 -th layer prehooked
29 -th layer prehooked
34 -th layer prehooked
38 -th layer prehooked
42 -th layer prehooked
47 -th layer prehooked
51 -th layer prehooked
55 -th layer prehooked
```

[5]:
```
weight_q = model.features[3].weight_q
w_alpha = model.features[3].weight_quant.wgt_alpha
w_bit = 4

weight_int = weight_q / (w_alpha / (2**(w_bit-1)-1))
# print(weight_int)
```

[6]:
```
act = save_output.outputs[1][0]
act_alpha  = model.features[3].act_alpha
act_bit = 4
act_quant_fn = act_quantization(act_bit)

act_q = act_quant_fn(act, act_alpha)

act_int = act_q / (act_alpha / (2**act_bit-1))
# print(act_int)
```

[7]:
```
conv_int = torch.nn.Conv2d(in_channels = 64, out_channels=64, kernel_size = 3,␣
  ↪padding=1)
conv_int.weight = torch.nn.parameter.Parameter(weight_int)
conv_int.bias = model.features[3].bias
output_int = conv_int(act_int)
output_recovered = output_int * (act_alpha / (2**act_bit-1)) * (w_alpha /␣
  ↪(2**(w_bit-1)-1))
# print(output_recovered)
```

[8]:
```
conv_ref = torch.nn.Conv2d(in_channels = 64, out_channels=64, kernel_size = 3,␣
  ↪padding=1)
conv_ref.weight = model.features[3].weight_q
conv_ref.bias = model.features[3].bias
output_ref = conv_ref(act)
# print(output_ref)
```

[9]:
```
a_int = act_int[0,:,:,:]  # [64, 32, 32] - [ic, ni, nj]
```

```
w_int = torch.reshape(weight_int, (weight_int.size(0), weight_int.size(1), -1))␣
 ↪# [64, 64, 9] - [oc, ic, kij]

padding = 1
stride = 1
arr = 16

ki = weight_int.size(2) # ki=3
kj = weight_int.size(3) # kj=3
ni = a_int.size(1)  # ni=32
nj = a_int.size(2)  # nj=32
ic = w_int.size(1)  # ic=64
oc = w_int.size(0)  # oc=64

ic_tile = ic // arr # 4
oc_tile = oc // arr # 4

# pad activation
a_pad = torch.zeros(ic, ni+2*padding, nj+2*padding).cuda()
a_pad[ :, padding:padding+ni, padding:padding+nj] = a_int.cuda()
a_pad = torch.reshape(a_pad, (a_pad.size(0), -1)) # [64, 1156] => [ic, padded␣
 ↪nij]

pad_nij = a_pad.size(1) # 1156

# partial sum
psum = torch.zeros(ic_tile, oc_tile, arr, a_pad.size(1), ki*kj).cuda()
# [4, 4, 16, 1156, 9] => [ic_tile, oc_tile, arr, pad_nij, kij]

for kij in range(ki*kj):
    for nij in range(pad_nij):
        for ic_t in range(ic_tile):
            for oc_t in range(oc_tile):
                w = w_int[oc_t*arr:(oc_t+1)*arr, ic_t*arr:(ic_t+1)*arr, kij].
 ↪cuda()
                x = a_pad[ic_t*arr:(ic_t+1)*arr, nij].unsqueeze(0).cuda()  #␣
 ↪[1, 16]

                m = nn.Linear(arr, arr, bias=False).cuda()
                m.weight = torch.nn.Parameter(w)

                psum[ic_t, oc_t, :, nij, kij] = m(x).squeeze(0)
```

[10]:
```
# output
o_ni = (ni+2*padding- (ki-1) - 1) // stride + 1 # 32
o_nj = (nj+2*padding- (kj-1) - 1) // stride + 1 # 32
out = torch.zeros(oc, o_ni * o_nj).cuda() # [64, 1024] => [oc, output nij]
```

```
ni_scale = ni+2*padding # 34

# SFP accumulation
for oi in range(o_ni):
    for oj in range(o_nj):
        for kk_i in range(ki):
            for kk_j in range(kj):
                for ic_t in range(ic_tile):
                    for oc_t in range(oc_tile):
                        o_nij = oi * o_ni + oj
                        nij = (oi + kk_i) * ni_scale + (oj + kk_j)
                        kij = kk_i * kj + kk_j
                        partial_sum = psum[ic_t, oc_t, :, nij, kij]
                        out[oc_t * arr : (oc_t + 1) * arr, o_nij] += partial_sum
```

[ ]:

[11]:
```
out_ref = output_int[0,:,:,:]
out_ref = torch.reshape(out_ref, (out_ref.size(0), -1))
difference = (out - out_ref)
print(difference.abs().sum())
```

```
tensor(1.0993, device='cuda:0', grad_fn=<SumBackward0>)
```

[12]:
```
print(out)
```

```
tensor([[ 1.7000e+01,  4.1000e+01,  7.3000e+01,  …,  1.2000e+02,
          5.8000e+01,  3.5000e+01],
        [ 2.4700e+02,  3.1600e+02,  2.9600e+02,  …,  2.9000e+02,
          3.0300e+02,  2.1600e+02],
        [ 1.1000e+01,  1.2500e+02,  1.7600e+02,  …, -8.0000e+01,
         -2.4000e+01,  7.7000e+01],
        …,
        [ 7.0000e+01,  4.7000e+01,  9.0000e+00,  …,  1.3000e+01,
          3.9000e+01,  1.7000e+01],
        [-3.1000e+01, -7.9000e+01, -1.0100e+02,  …, -1.4000e+01,
         -5.3000e+01,  3.2000e+01],
        [-2.2000e+01, -1.9800e+02, -2.1100e+02,  …,  1.1000e+01,
          2.2000e+01,  4.7684e-07]], device='cuda:0', grad_fn=<CopySlices>)
```