Memory_data_generator

December 13, 2025

1 Part1 4bit-activation and 4bit-weight

[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 = 512
model_name = "VGG16_quant_project_part1_92_prec"    #"Resnet20_quant"
model = VGG16_quant_project_part1()

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, 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): ReLU(inplace=True)
    (31): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
    (32): QuantConv2d(
      512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
      (weight_quant): weight_quantize_fn()
    )
    (33): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (34): ReLU(inplace=True)
    (35): QuantConv2d(
      512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
      (weight_quant): weight_quantize_fn()
    )
    (36): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (37): ReLU(inplace=True)
    (38): QuantConv2d(
      512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
      (weight_quant): weight_quantize_fn()
    )
    (39): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (40): ReLU(inplace=True)
    (41): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
    (42): AvgPool2d(kernel_size=1, stride=1, padding=0)
  )
  (classifier): Linear(in_features=512, out_features=10, bias=True)
)
```

[2]:
```
fdir = 'result/'+str(model_name)+'/model_best.pth.tar'

checkpoint = torch.load(fdir)
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: 9162/10000 (92%)
```

[3]:
```
class SaveOutput:
    def __init__(self):
        self.outputs = []
    def __call__(self, module, module_in):
        self.outputs.append(module_in)
    def clear(self):
        self.outputs = []
######### Save inputs from selected layer ##########
save_output = SaveOutput()
i = 0
for layer in model.modules():
    i = i+1
    if isinstance(layer, (QuantConv2d, torch.nn.ReLU)):
        layer.register_forward_pre_hook(save_output)
####################################################

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

[4]:
```
model.features[27].show_params()
```

```
clipping threshold weight alpha: 2.374000, activation alpha: 4.564000
```

[7]:
```
weight_q = model.features[27].weight_q
w_alpha = model.features[27].weight_quant.wgt_alpha
w_bit = 4
```

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

[8]:
```
act = save_output.outputs[15][0]
act_alpha = model.features[27].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))
act_delta = act_alpha / (2 ** act_bit - 1)
```

[9]:
```
conv_int = torch.nn.Conv2d(in_channels = 8, out_channels=8, kernel_size = 3,␣
 ↪padding=1, bias = False)
conv_int.weight = torch.nn.parameter.Parameter(weight_int)
output_int = conv_int(act_int) # output_int can be calculated with conv_int␣␣
 ↪and x_int
output_recovered = output_int * act_delta * w_delta # recover with x_delta and␣
output_recovered = F.relu(output_recovered)
```

[10]:
```
output_ref = save_output.outputs[17][0]
```

[11]:
```
difference = abs(output_ref - output_recovered )
print(difference.mean())
```

```
tensor(3.2444e-07, device='cuda:0', grad_fn=<MeanBackward0>)
```

[12]:
```
out_relu = F.relu(output_int)
```

[13]:
```
out = torch.reshape(out_relu, (out_relu.size(0), out_relu.size(1), -1))
```

[14]:
```
out = out[0]
```

[15]:
```
# act_int.size = torch.Size([128, 64, 32, 32])  <- batch_size, input_ch, ni, nj
a_int = act_int[0,:,:,:]  # pick only one input out of batch
# a_int.size() = [64, 32, 32]

# conv_int.weight.size() = torch.Size([64, 64, 3, 3])  <- output_ch, input_ch,␣
 ↪ki, kj
w_int = torch.reshape(weight_int, (weight_int.size(0), weight_int.size(1), -1))␣
 ↪ # merge ki, kj index to kij
# w_int.weight.size() = torch.Size([64, 64, 9])

padding = 1
stride = 1
array_size = 8 # row and column number

nig = range(a_int.size(1))  ## ni group
njg = range(a_int.size(2))  ## nj group
```

```
icg = range(int(w_int.size(1)))  ## input channel
ocg = range(int(w_int.size(0)))  ## output channel

ic_tileg = range(int(len(icg)/array_size))
oc_tileg = range(int(len(ocg)/array_size))

kijg = range(w_int.size(2))
ki_dim = int(math.sqrt(w_int.size(2)))  ## Kernel's 1 dim size

######## Padding before Convolution #######
a_pad = torch.zeros(len(icg), len(nig)+padding*2, len(nig)+padding*2).cuda()
# a_pad.size() = [64, 32+2pad, 32+2pad]
a_pad[ :, padding:padding+len(nig), padding:padding+len(njg)] = a_int.cuda()
a_pad = torch.reshape(a_pad, (a_pad.size(0), -1))
# a_pad.size() = [64, (32+2pad)*(32+2pad)]


a_tile = torch.zeros(len(ic_tileg), array_size,    a_pad.size(1)).cuda()
w_tile = torch.zeros(len(oc_tileg)*len(ic_tileg), array_size, array_size,␣
 ↪len(kijg)).cuda()

for ic_tile in ic_tileg:
    a_tile[ic_tile,:,:] = a_pad[ic_tile*array_size:(ic_tile+1)*array_size,:]

for ic_tile in ic_tileg:
    for oc_tile in oc_tileg:
        w_tile[oc_tile*len(oc_tileg) + ic_tile,:,:,:] =␣
 ↪w_int[oc_tile*array_size:(oc_tile+1)*array_size, ic_tile*array_size:
 ↪(ic_tile+1)*array_size, :]



##########################################

p_nijg = range(a_pad.size(1)) ## psum nij group

psum = torch.zeros(len(ic_tileg), len(oc_tileg), array_size, len(p_nijg),␣
 ↪len(kijg)).cuda()

for kij in kijg:
    for ic_tile in ic_tileg:       # Tiling into array_sizeXarray_size array
        for oc_tile in oc_tileg:   # Tiling into array_sizeXarray_size array   ␣
 ↪
            for nij in p_nijg:       # time domain, sequentially given input
                    m = nn.Linear(array_size, array_size, bias=False)
```

```
                    #m.weight = torch.nn.Parameter(w_int[oc_tile*array_size:
↪(oc_tile+1)*array_size, ic_tile*array_size:(ic_tile+1)*array_size, kij])
                    m.weight = torch.nn.
↪Parameter(w_tile[len(oc_tileg)*oc_tile+ic_tile,:,:,kij])
                    psum[ic_tile, oc_tile, :, nij, kij] = m(a_tile[ic_tile,:
↪,nij]).cuda()
```

[20]:
```
a_tile.size()
```

[20]:
```
torch.Size([1, 8, 36])
```

1.0.1 Output Stationary

[ ]:
```
import math
import torch

bw = 4
psum_bw = 16
len_kij = 9
USE_IC = 3
row = 8
col = 8
padding = 1
stride = 1
oc_tile = 0
nij_out_base = 0

kij_num = w_int.size(2)                 # 9
ki_dim  = int(math.sqrt(kij_num))       # 3


HW = a_pad.size(1)
Hpad = int(math.sqrt(HW))
Wpad = Hpad

Hout = (Hpad - ki_dim) // stride + 1
Wout = (Wpad - ki_dim) // stride + 1

oc0 = oc_tile * col
oc1 = oc0 + col

# ---- 4-bit/16-bit ----
def enc_twos(v, bits):
    v = int(round(float(v)))
    return v & ((1 << bits) - 1)

def dec_twos(u, bits):
```

```
    u = int(u) & ((1 << bits) - 1)
    if u >= (1 << (bits - 1)):
        u -= (1 << bits)
    return u

# ========= activation =========
# row order t = in_ch  kij inside
# each row lane = nij_out(0..7) 8 nibble MSB..LSB nij_out=7
with open("activation_os_3ic_tile0.txt", "w") as f:
    for in_ch in range(USE_IC):
        for kij in range(len_kij):
            ki = kij // ki_dim
            kj = kij %  ki_dim

            #   nij_out=7..0  row7->row0
            for lane_row in range(row-1, -1, -1):
                nij_out = nij_out_base + lane_row
                oi = nij_out // Wout
                oj = nij_out %  Wout

                y = oi * stride + ki
                x = oj * stride + kj
                idx = y * Wpad + x

                a_u4 = enc_twos(a_pad[in_ch, idx].item(), bw)
                f.write(f"{a_u4:0{bw}b}")
            f.write("\n")

# ========= weight =========
# each row lane = nij_out(0..7) 8 nibble MSB..LSB nij_out=7
with open("weight_os_3ic.txt", "w") as f:
    for in_ch in range(USE_IC):
        for kij in range(len_kij):
            for lane_oc in range(col-1, -1, -1):  # oc7..oc0
                w_u4 = enc_twos(w_int[oc0 + lane_oc, in_ch, kij].item(), bw)
                f.write(f"{w_u4:0{bw}b}")
            f.write("\n")


# ========= out =========
# TB read 8 times each time read 128b token coresponding nij_out=0..7
# each token 8 oc   16bit psum MSB...LSB
with open("out.txt", "w") as f:
    for lane_row in range(row):  # nij_out = 0..7
        nij_out = nij_out_base + lane_row
        oi = nij_out // Wout
        oj = nij_out %  Wout
```

```
        acc = [0] * col  # signed int accumulator per oc

        for in_ch in range(USE_IC):
            for kij in range(len_kij):
                ki = kij // ki_dim
                kj = kij %  ki_dim

                y = oi * stride + ki
                x = oj * stride + kj
                idx = y * Wpad + x

                a_u4 = enc_twos(a_pad[in_ch, idx].item(), bw)
                a_s  = dec_twos(a_u4, bw)

                for oc in range(col):
                    w_u4 = enc_twos(w_int[oc0 + oc, in_ch, kij].item(), bw)
                    w_s  = dec_twos(w_u4, bw)
                    acc[oc] += a_s * w_s

        # RELU
        for oc in range(col):
            if acc[oc] < 0:
                acc[oc] = 0

        # pack to 128b oc7..oc0 each 16b
        line = ""
        for oc in range(col-1, -1, -1):
            u16 = enc_twos(acc[oc], psum_bw)
            line += f"{u16:0{psum_bw}b}"
        f.write(line + "\n")
```

1.0.2 Weight Stationary

[ ]:
```
### show this cell partially. The following cells should be printed by students␣
 ↪###
tile_id = 0
nij = 0 # just a random number
X = a_tile[tile_id,:,nij:nij+36]  # [tile_num, array row num, time_steps]

bit_precision = 4
file = open('activation_tile0.txt', 'w') #write to file
# file.write('#time0row7[msb-lsb],time0row6[msb-lst],....
 ↪,time0row0[msb-lst]#\n')
# file.write('#time1row7[msb-lsb],time1row6[msb-lst],....
 ↪,time1row0[msb-lst]#\n')
# file.write('#................#\n')
```

```
for i in range(X.size(1)):  # time step
    for j in range(X.size(0)): # row #
        X_bin = '{0:04b}'.format(round(X[7-j,i].item()))
        for k in range(bit_precision):
            file.write(X_bin[k])
        #file.write(' ')  # for visibility with blank between words, you can use
    file.write('\n')
file.close() #close file
```

[ ]:
```
### Complete this cell ###
tile_id = 0
kij = 9
  # w_tile[tile_num, array col num, array row num, kij]

bit_precision = 4
for i in range(kij):
    file = open(f'weight_itile0_otile0_kij{i}.txt', 'w') #write to file
    # file.write('#col0row7[msb-lsb],col0row6[msb-lst],....
 ↪,col0row0[msb-lst]#\n')
    # file.write('#col1row7[msb-lsb],col1row6[msb-lst],....
 ↪,col1row0[msb-lst]#\n')
    # file.write('#................#\n')

    W = w_tile[tile_id,:,:,i]
    for col in range(W.size(1)):
        for row in range(W.size(0)):
            W_value = W[col, 7 - row]
            if W_value < 0:
                W_value += 2 ** bit_precision
            W_bin = '{0:04b}'.format(round(W_value.item()))
            for k in range(bit_precision):
                file.write(W_bin[k])
        file.write('\n')
    file.close()
```

[ ]:
```
bit_precision = 16


file = open('out.txt', 'w') #write to file
# file.write('#time0col7[msb-lsb],time0col6[msb-lst],....
 ↪,time0col0[msb-lst]#\n')
# file.write('#time1col7[msb-lsb],time1col6[msb-lst],....
 ↪,time1col0[msb-lst]#\n')
# file.write('#................#\n')

for t in range(out.size(1)):
```

```
    for col in range(out.size(0)):
        p_value = out[7 - col, t]
        if p_value < 0:
            p_value += 2 ** bit_precision
        psum_bin = '{0:016b}'.format(round(p_value.item()))
        for k in range(bit_precision):
            file.write(psum_bin[k])
    file.write('\n')
file.close()
```