Part1_golden_gen

December 14, 2025

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

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

import torchvision
import torchvision.transforms as transforms

from models import *

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


batch_size = 128




##################################### Modify here␣
  ↪########################################
model_name = "VGG16_vanilla"
model = VGG16_vanilla()
##########################################################################################


model.to("cuda")

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

```python
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 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
```

```
=> Building model…
Files already downloaded and verified
```

[2]:
```python
PATH = f"result/{model_name}/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: 8986/10000 (90%)
```

[3]:
```python
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()

model.features[27].register_forward_pre_hook(save_output)
model.features[28].register_forward_pre_hook(save_output)
model.features[29].register_forward_pre_hook(save_output)
print(model.features[27])
print(model.features[28])

dataiter = iter(testloader)
images, labels = next(dataiter)
```

```
images = images.to(device)
out = model(images)

act_grabbed_batch  = save_output.outputs[0][0]
psum_grabbed_batch = save_output.outputs[1][0]
relu_grabbed_batch = save_output.outputs[2][0]
conv_grabbed = model.features[27]



print(act_grabbed_batch.size())
print(psum_grabbed_batch.size())
```

```
QuantConv2d(
  8, 8, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
  (weight_quant): weight_quantize_fn()
)
ReLU(inplace=True)
torch.Size([128, 8, 4, 4])
torch.Size([128, 8, 4, 4])
```

[4]:
```
weight_q = conv_grabbed.weight_q
w_alpha = conv_grabbed.weight_quant.wgt_alpha
w_bit = 4

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


print(sorted(set(weight_int.detach().cpu().numpy().flatten().tolist())))
```

```
[-6.999999523162842, -6.0, -4.999999523162842, -4.0, -3.0, -2.0, -1.0, -0.0,
1.0, 2.0, 3.0, 4.0, 4.999999523162842, 6.0, 6.999999523162842]
```

[5]:
```
act = act_grabbed_batch
act_alpha  = conv_grabbed.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(sorted(set(act_int.detach().cpu().numpy().flatten().tolist())))
```

```
[0.0, 1.0, 2.0, 3.000000238418579, 4.0, 5.0, 6.000000476837158, 7.0, 8.0, 9.0,
10.0, 10.999999046325684, 12.000000953674316, 13.000000953674316, 14.0,
14.999999046325684]
```

[6]:
```
conv_int = torch.nn.Conv2d(in_channels = 8, out_channels=8, kernel_size = 3,␣
 ↪padding=1)
conv_int.weight = torch.nn.parameter.Parameter(weight_int)
conv_int.bias = conv_grabbed.bias
output_int = conv_int(act_int)
output_recovered = output_int * (act_alpha / (2**act_bit-1)) * (w_alpha /␣
 ↪(2**(w_bit-1)-1))
```

[7]:
```
conv_ref = torch.nn.Conv2d(in_channels = 8, out_channels=8, kernel_size = 3,␣
 ↪padding=1)
conv_ref.weight = conv_grabbed.weight_q
conv_ref.bias = conv_grabbed.bias
output_ref = conv_ref(act)
```

[8]:
```
# 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()
```

[9]:
```
import math

a_pad_ni_dim = int(math.sqrt(a_pad.size(1))) # 32

o_ni_dim = int((a_pad_ni_dim - (ki_dim- 1) - 1)/stride + 1)
o_nijg = range(o_ni_dim**2)

out = torch.zeros(len(ocg), len(o_nijg)).cuda()
oc_tile_idx_l, oc_tile_idx_r = 0, 0
```

```
### SFP accumulation ###
for o_nij in o_nijg:
    for kij in kijg:
        for ic_tile in ic_tileg:
            for oc_tile in oc_tileg:
                # select the indices range to accumulate, 0:16, 16:32, 32:48,␣
 ↪48:64
                oc_tile_idx_l, oc_tile_idx_r = oc_tile*array_size,␣
 ↪(oc_tile+1)*array_size
                # different input channels are summed up here
                out[oc_tile_idx_l:oc_tile_idx_r, o_nij] = out[oc_tile_idx_l:
 ↪oc_tile_idx_r, o_nij] + \
                psum[ic_tile, oc_tile, :, int(o_nij/o_ni_dim)*a_pad_ni_dim +␣
 ↪o_nij%o_ni_dim + int(kij/ki_dim)*a_pad_ni_dim + kij%ki_dim, kij]
```

[25]:
```
out_2D = torch.reshape(out, (out.size(0), o_ni_dim, -1))
difference = (out_2D - output_int[0,:,:,:])
print(difference.sum())
```

```
tensor(0.0002, device='cuda:0', grad_fn=<SumBackward0>)
```

[11]:
```
### show this cell partially. The following cells should be printed by students␣
 ↪###
tile_id = 0
X = a_tile[tile_id]  # [tile_num, array row num, time_steps]
print("a_tile.size(): ", a_tile.size())
print("a_tile[0].size(): ", X.size())

bit_precision = 4
file = open(f'golden/Part1/activation_tile{tile_id}.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
```

```
file = open(f'golden/Part1/viz/viz_activation_tile{tile_id}.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()))

        file.write(f'{round(X[7-j,i].item()): 3d}')

        file.write(' ')  # for visibility with blank between words, you can use
    file.write('\n')
file.close() #close file
```

```
a_tile.size():  torch.Size([1, 8, 36])
a_tile[0].size():  torch.Size([8, 36])
```

[12]:
```
### Complete this cell ###

for kij in kijg:
    tile_id = 0
    W = w_tile[tile_id,:,:,kij]  # w_tile[tile_num, array col num, array row␣
 ↪num, kij]

    bit_precision = 4
    file = open(f'golden/Part1/weight_itile0_otile0_kij{kij}.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')


    for i in range(W.size(0)):  # array col num
        for j in range(W.size(1)): # array row num
            # 2's complement
            W_signed = round(W[i, 7-j].item())
            if W_signed < 0:
                W_signed += 16

            W_bin = '{0:04b}'.format(W_signed)
            for k in range(bit_precision):
                file.write(W_bin[k])
            # file.write(' ')  # for visibility with blank between words, you␣
 ↪can use
        file.write('\n')
    file.close() #close file




    file = open(f'golden/Part1/viz/viz_weight_itile0_otile0_kij{kij}.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')

    for i in range(W.size(0)):  # array col num
        for j in range(W.size(1)): # array row num
            # 2's complement
            W_signed = round(W[i, 7-j].item())
            W_signed = f'{W_signed: 3d}'
            file.write(W_signed)
            file.write(' ')  # for visibility with blank between words, you can␣
 ↪use
        file.write('\n')
    file.close() #close file
```

[26]:
```
## save out
# Comment these 2 lines out if you're dealing with conv's output directly.
#
print(out.size())
bit_precision = 16

file = open('golden/Part1/out_raw.txt', 'w') #write to file
file.write('#time0co7[msb-lsb],time0co6[msb-lst],....,time0col0[msb-lst]#\n')
file.write('#time1co7[msb-lsb],time1co6[msb-lst],....,time1col0[msb-lst]#\n')
file.write('#...............#\n')

for i in range(out.size(1)):  # time step
    for j in range(out.size(0)): # array col
        # 2's complement
        out_signed = round(out[7-j, i].item())
        if out_signed < 0:
            out_signed += 65536

        out_bin = '{0:016b}'.format(out_signed)
        for k in range(bit_precision):
            file.write(out_bin[k])
        # file.write(' ')  # for visibility with blank between words, you can␣
 ↪use
    file.write('\n')
file.close() #close file
```

```
file = open('golden/Part1/viz/viz_out_raw.txt', 'w') #write to file
file.write('#time0co7[msb-lsb],time0co6[msb-lst],....,time0col0[msb-lst]#\n')
file.write('#time1co7[msb-lsb],time1co6[msb-lst],....,time1col0[msb-lst]#\n')
file.write('#................#\n')

for i in range(out.size(1)):  # time step
    for j in range(out.size(0)): # array col
        out_signed = round(out[7-j, i].item())
        out_signed = f'{out_signed: 6d}'
        file.write(out_signed)
        file.write(' ')  # for visibility with blank between words, you can use
    file.write('\n')
file.close() #close file
```

```
torch.Size([8, 16])
```

[28]:
```
## save out
# Comment these 2 lines out if you're dealing with conv's output directly.
#
print(out.size())
bit_precision = 16

file = open('golden/Part1/out.txt', 'w') #write to file
file.write('#time0co7[msb-lsb],time0co6[msb-lst],....,time0col0[msb-lst]#\n')
file.write('#time1co7[msb-lsb],time1co6[msb-lst],....,time1col0[msb-lst]#\n')
file.write('#................#\n')

for i in range(out.size(1)):  # time step
    for j in range(out.size(0)): # array col
        # 2's complement
        out_signed = round(out[7-j, i].item())
        if out_signed < 0:
            out_signed = 0

        out_bin = '{0:016b}'.format(out_signed)
        for k in range(bit_precision):
            file.write(out_bin[k])
        # file.write(' ')  # for visibility with blank between words, you can␣
 ↪use
    file.write('\n')
file.close() #close file


file = open('golden/Part1/viz/viz_out.txt', 'w') #write to file
file.write('#time0co7[msb-lsb],time0co6[msb-lst],....,time0col0[msb-lst]#\n')
file.write('#time1co7[msb-lsb],time1co6[msb-lst],....,time1col0[msb-lst]#\n')
file.write('#...............#\n')

for i in range(out.size(1)):  # time step
    for j in range(out.size(0)): # array col
        out_signed = round(out[7-j, i].item())
        if out_signed < 0:
            out_signed = 0
        out_signed = f'{out_signed: 6d}'
        file.write(out_signed)
        file.write(' ')  # for visibility with blank between words, you can use
    file.write('\n')
file.close() #close file
```

```
torch.Size([8, 16])
```

1 Stop here

[20]:
```
### Complete this cell ###
ic_tile_id = 0
oc_tile_id = 0


kij = 0
psum_tile = psum[ic_tile_id,oc_tile_id,:,:,kij]
# psum_tile.shape(): array col num , p_nijg(time step)

relu_grabbed



bit_precision = 16
file = open('psum.txt', 'w') #write to file
file.write('#time0co7[msb-lsb],time0co6[msb-lst],....,time0col0[msb-lst]#\n')
file.write('#time1co7[msb-lsb],time1co6[msb-lst],....,time1col0[msb-lst]#\n')
file.write('#...............#\n')



for i in range(psum_tile.size(1)):  # time step
    for j in range(psum_tile.size(0)): # col #

        # 2's complement
        psum_tile_signed = round(psum_tile[7-j,i].item())
        if psum_tile_signed < 0:
            psum_tile_signed += 65536
        psum_tile_bin = '{0:016b}'.format(psum_tile_signed)

        for k in range(bit_precision):
            file.write(psum_tile_bin[k])
        # file.write(' ')  # for visibility with blank between words, you can␣
 ↪use
    file.write('\n')
file.close() #close file
```

[42]:
```
out_tile_cal_2D = torch.reshape(out, (out.size(0), o_ni_dim, -1)) # nij -> ni &␣
 ↪nj

difference = (out_2D - out_tile_cal_2D)

if difference.abs().sum() == 0.0:
    print(f"Difference: {difference.abs().sum()}")
    print("Tiled output calculation matches the original one!")
```

```
Difference: 0.0
Tiled output calculation matches the original one!
```

[ ]:
```
```