In [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 = 64
model_name = "VGG16_quant"
model = VGG16_quant()

device = torch.device("cuda" if use_gpu else "cpu")
normalize = transforms.Normalize(mean=[0.491, 0.482, 0.447], std=[0.247, 0.243, 0.2


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, shu


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, shuff


print_freq = 100 # every 100 batches, accuracy printed. Here, each batch includes '
# 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
                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 epoch
    adjust_list = [35, 50]
    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)
print (model)
```

```
=> Building model...
0
3
6
7
10
13
14
17
20
23
24
27
29
32
33
36
39
42
Downloading https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz to ./data/cifa
r-10-python.tar.gz
```

```
100%|██████████| 170498071/170498071 [00:01<00:00, 92383323.89it/s]
```

```
Extracting ./data/cifar-10-python.tar.gz to ./data
Files already downloaded and verified
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=Fals
e)
    (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_stat
s=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_sta
ts=True)
    (12): ReLU(inplace=True)
    (13): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=Fals
e)
    (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_sta
ts=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_sta
ts=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_sta
ts=True)
    (22): ReLU(inplace=True)
    (23): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=Fals
e)
    (24): QuantConv2d(
      256, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
      (weight_quant): weight_quantize_fn()
    )
    (25): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stat
```

```
s=True)
    (26): ReLU(inplace=True)
    (27): QuantConv2d(
      16, 8, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
      (weight_quant): weight_quantize_fn()
    )
    (28): ReLU(inplace=True)
    (29): QuantConv2d(
      8, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
      (weight_quant): weight_quantize_fn()
    )
    (30): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_sta
ts=True)
    (31): ReLU(inplace=True)
    (32): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=Fals
e)
    (33): QuantConv2d(
      512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
      (weight_quant): weight_quantize_fn()
    )
    (34): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_sta
ts=True)
    (35): ReLU(inplace=True)
    (36): QuantConv2d(
      512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
      (weight_quant): weight_quantize_fn()
    )
    (37): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_sta
ts=True)
    (38): ReLU(inplace=True)
    (39): QuantConv2d(
      512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
      (weight_quant): weight_quantize_fn()
    )
    (40): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_sta
ts=True)
    (41): ReLU(inplace=True)
    (42): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=Fals
e)
    (43): AvgPool2d(kernel_size=1, stride=1, padding=0)
  )
  (classifier): Linear(in_features=512, out_features=10, bias=True)
)
```

In [ ]:

```
# This cell is from the website

lr = 4e-2
weight_decay = 1e-4
epochs = 40
best_prec = 0

model = model.cuda()
criterion = nn.CrossEntropyLoss().cuda()
optimizer = torch.optim.SGD(model.parameters(), lr=lr, momentum=0.9, weight_decay=w
# weight decay: for regularization to prevent overfitting


if not os.path.exists('result_multi'):
    os.makedirs('result_multi')

fdir = 'result_multi/'+str(model_name)

if not os.path.exists(fdir):
    os.makedirs(fdir)
```

```
for epoch in range(0, epochs):
    adjust_learning_rate(optimizer, epoch)

    train(trainloader, model, criterion, optimizer, epoch)

    # evaluate on test set
    print("Validation starts")
    prec = validate(testloader, model, criterion)

    # remember best precision and save checkpoint
    is_best = prec > best_prec
    best_prec = max(prec,best_prec)
    print('best acc: {:1f}'.format(best_prec))
    save_checkpoint({
        'epoch': epoch + 1,
        'state_dict': model.state_dict(),
        'best_prec': best_prec,
        'optimizer': optimizer.state_dict(),
    }, is_best, fdir)
```

```
best acc: 90.490000
Epoch: [39][0/782]      Time 0.171 (0.171)      Data 0.118 (0.118)      Loss 0.033
5 (0.0335)      Prec 100.000% (100.000%)
Epoch: [39][100/782]    Time 0.040 (0.044)      Data 0.002 (0.004)      Loss 0.061
9 (0.0913)      Prec 98.438% (96.767%)
Epoch: [39][200/782]    Time 0.039 (0.048)      Data 0.001 (0.004)      Loss 0.060
6 (0.0963)      Prec 98.438% (96.688%)
Epoch: [39][300/782]    Time 0.041 (0.046)      Data 0.002 (0.004)      Loss 0.159
7 (0.0993)      Prec 95.312% (96.584%)
Epoch: [39][400/782]    Time 0.041 (0.045)      Data 0.002 (0.004)      Loss 0.018
5 (0.1000)      Prec 100.000% (96.579%)
Epoch: [39][500/782]    Time 0.043 (0.047)      Data 0.001 (0.004)      Loss 0.154
2 (0.0991)      Prec 95.312% (96.616%)
Epoch: [39][600/782]    Time 0.040 (0.046)      Data 0.003 (0.004)      Loss 0.067
3 (0.1002)      Prec 96.875% (96.589%)
Epoch: [39][700/782]    Time 0.040 (0.046)      Data 0.002 (0.004)      Loss 0.045
4 (0.1004)      Prec 96.875% (96.563%)
Validation starts
Test: [0/157]   Time 0.128 (0.128)      Loss 0.2969 (0.2969)    Prec 92.188% (92.1
88%)
Test: [100/157] Time 0.023 (0.024)      Loss 0.3865 (0.3302)    Prec 89.062% (90.6
56%)
 * Prec 90.650%
best acc: 90.650000
```

In [2]:

```python
fdir = 'results_multi/'+str(model_name)+'/model_best.pth.tar'

checkpoint = torch.load(fdir)
model.load_state_dict(checkpoint['state_dict'])


criterion = nn.CrossEntropyLoss().cuda()

model.eval()
model.cuda()


prec = validate(testloader, model, criterion)
```

```
Test: [0/157]   Time 2.333 (2.333)      Loss 0.2969 (0.2969)    Prec 92.188% (92.1
88%)
Test: [100/157] Time 0.016 (0.045)      Loss 0.3865 (0.3302)    Prec 89.062% (90.6
56%)
 * Prec 90.650%
```

In [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()
i = 0

for name, layer in model.named_modules():

    if isinstance(layer, torch.nn.Conv2d):
        print(i,"-th layer prehooked")
        layer.register_forward_pre_hook(save_output)
    i = i+1
```

```
####################################################

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

print("7st convolution's input size:", save_output.outputs[8][0].size())
print("7st convolution's input size:", save_output.outputs[9][0].size())
```

```
2 -th layer prehooked
6 -th layer prehooked
11 -th layer prehooked
15 -th layer prehooked
20 -th layer prehooked
24 -th layer prehooked
28 -th layer prehooked
33 -th layer prehooked
37 -th layer prehooked
40 -th layer prehooked
45 -th layer prehooked
49 -th layer prehooked
53 -th layer prehooked
7st convolution's input size: torch.Size([64, 16, 4, 4])
7st convolution's input size: torch.Size([64, 8, 4, 4])
```

In [4]:
```
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))
#print(weight_int)
```

In [5]:
```
act = save_output.outputs[8][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))
#print(act_int)
```

In [6]:
```
conv_int = torch.nn.Conv2d(in_channels = 16, out_channels=8, kernel_size = 3, paddi
conv_int.weight = torch.nn.parameter.Parameter(weight_int)
psum_int = conv_int(act_int)

psum_recovered = psum_int * (act_alpha / (2**act_bit-1)) * (w_alpha / (2**(w_bit-1)
relu = model.features[28]
psum_recovered = relu(psum_recovered)
#print(psum_recovered)
```

In [7]:
```
difference = abs( save_output.outputs[9][0] - psum_recovered )
print(difference.mean())
```

```
tensor(1.7829e-07, device='cuda:0', grad_fn=<MeanBackward0>)
```

In [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,
w_int = torch.reshape(weight_int, (weight_int.size(0), weight_int.size(1), -1))  #
```

```
# 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)

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:(



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

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)

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_til
                m.weight = torch.nn.Parameter(w_tile[len(oc_tileg)*oc_tile+ic_t
                psum[ic_tile, oc_tile, :, nij, kij] = m(a_tile[ic_tile,:,nij]).
```

In [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()


### SFP accumulation ###
```

```
for o_nij in o_nijg:
    for kij in kijg:
        for ic_tile in ic_tileg:
            for oc_tile in oc_tileg:
                out[oc_tile*array_size:(oc_tile+1)*array_size, o_nij] = out[oc_tile
                psum[ic_tile, oc_tile, :, int(o_nij/o_ni_dim)*a_pad_ni_dim + o_nij%
                ## 4th index = (int(o_nij/30)*32 + o_nij%30) + (int(kij/3)*32 + kij
```

In [10]:
```
# out_2D = torch.reshape(out, (out.size(0), o_ni_dim, -1)) # nij -> ni & nj
# difference = (out_2D - output_int[0,:,:,:])
# print(difference.abs().sum())
```

In [11]:
```
### show this cell partially. The following cells should be printed by students ###
tile_id = 0
nij = 0 # just a random number
X = a_tile[:,:,nij:nij+36]
print(X)
print(X.size())
Y = torch.reshape(X, (-1, X.size(2)))
print(Y.size())
print(X[:,:,7])
#print (X[:,:,0].size())# [tile_num, array row num, time_steps]
print(Y[:,7])
bit_precision = 4
file = open('activation_mc_project.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(Y.size(1)):
    for j in range(Y.size(0)): # row #
        X_bin = '{0:04b}'.format(round(Y[15-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
```

```
          0.0000, 8.0000, 9.0000, 7.0000, 6.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000],
         [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 4.0000,
          6.0000, 8.0000, 6.0000, 0.0000, 0.0000, 7.0000, 7.0000, 8.0000,
          5.0000, 0.0000, 0.0000, 6.0000, 7.0000, 6.0000, 4.0000, 0.0000,
          0.0000, 3.0000, 3.0000, 3.0000, 2.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000],
         [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 3.0000,
          1.0000, 2.0000, 2.0000, 0.0000, 0.0000, 2.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 4.0000, 4.0000, 7.0000, 6.0000, 0.0000,
          0.0000, 8.0000, 5.0000, 4.0000, 3.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000],
         [0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000, 0.0000,
          0.0000, 0.0000, 0.0000, 0.0000]]], device='cuda:0',
       grad_fn=<SliceBackward0>)
torch.Size([2, 8, 36])
torch.Size([16, 36])
tensor([[0., 0., 0., 0., 1., 4., 0., 0.],
        [0., 0., 2., 0., 2., 4., 3., 0.]], device='cuda:0',
       grad_fn=<SelectBackward0>)
tensor([0., 0., 0., 0., 1., 4., 0., 0., 0., 0., 2., 0., 2., 4., 3., 0.],
       device='cuda:0', grad_fn=<SelectBackward0>)
```

In [12]:

```
### show this cell partially. The following cells should be printed by students ###
tile_id = 0
nij = 0 # just a random number
X = a_tile[:,:,nij:nij+36]
Y = torch.reshape(X, (-1, X.size(2)))
print(Y.size())
print(X[:,:,8])
#print (X[:,:,0].size())# [tile_num, array row num, time_steps]
print(Y[:,8])
bit_precision = 4
file = open('activation_mc_project_int.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(Y.size(1)):
    for j in range(Y.size(0)): # row #
        file.write(f'{round(Y[15-j,i].item())},')
        #file.write(' ')  # for visibility with blank between words, you can use
    file.write('\n')
file.close() #close file
```

```
torch.Size([16, 36])
tensor([[0., 0., 0., 0., 0., 6., 0., 0.],
        [0., 0., 0., 0., 4., 6., 1., 0.]], device='cuda:0',
       grad_fn=<SelectBackward0>)
tensor([0., 0., 0., 0., 0., 6., 0., 0., 0., 0., 0., 0., 4., 6., 1., 0.],
       device='cuda:0', grad_fn=<SelectBackward0>)
```

In [13]:

```
print(w_tile.size())
```

```
torch.Size([2, 8, 8, 9])
```

In [30]:

```
print(w_tile.size())
W = w_tile[:,:,:,0]  # w_tile[tile_num, array col num, array row num, kij
print(W)
W = w_tile[:,0,:,0]

W_lis = []
```

```
for out_ch in range(w_tile.size(1)):
  W = w_tile[:,out_ch,:,0]
  W = torch.reshape(W, (-1, W.size(0)*W.size(1)))
  W_lis.append(W.tolist()[0])

W = torch.tensor(W_lis)
print(W)
```

```
torch.Size([2, 8, 8, 9])
```

In [31]:

```
# ### Complete this cell ###
# tile_id = 0
# kij = 0
# W = w_tile[:,:,:,kij]  # w_tile[tile_num, array col num, array row num, kij]
# print(w_tile.size())
# bit_precision = 4

# for kij_num in range(9):
#     W = w_tile[:,:,:,kij_num]
#     Z = torch.reshape(W, (-1, W.size(0)*W.size(2)))
#     print(Z)
#     file = open('weight_mc_kij{}.txt'.format(kij_num), '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(Z.size(0)):
#         lis = []
#         for j in range(Z.size(1)): # row #
#             if(int(Z[i,j].item())>=0):
#                 W_bin = '{0:04b}'.format(round(Z[i,j].item()))
#             else:
#                 W_bin = '{0:04b}'.format(round(Z[i,j].item())+16)
#             lis.append(W_bin)
#             #for k in range(bit_precision):
#             #   file.write(W_bin[k])
#             #file.write(' ')  # for visibility with blank between words, you can
#         print(lis)
#         string_line = ''.join(reversed(lis))
#         file.write(string_line)
#         file.write('\n')
#     file.close() #close file
```

In [42]:

```
### Complete this cell ###
tile_id = 0
kij = 0
W = w_tile[:,:,:,1]  # w_tile[tile_num, array col num, array row num, kij]
print(w_tile.size())
bit_precision = 4

for kij_num in range(9):
    W_lis = []
    for out_ch in range(w_tile.size(1)):
      W = w_tile[:,out_ch,:,kij_num]
      W = torch.reshape(W, (-1, W.size(0)*W.size(1)))
      W_lis.append(W.tolist()[0])
    Z = torch.tensor(W_lis)
    file = open('weight_mc_kij{}.txt'.format(kij_num), '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(Z.size(0)):
        lis = []
        for j in range(Z.size(1)): # row #
            if(int(Z[i,j].item())>=0):
```

```
                W_bin = '{0:04b}'.format(round(Z[i,j].item()))
            else:
                W_bin = '{0:04b}'.format(round(Z[i,j].item())+16)
            lis.append(W_bin)
            #for k in range(bit_precision):
            #   file.write(W_bin[k])
            #file.write(' ')  # for visibility with blank between words, you can us
        string_line = ''.join(reversed(lis))
        file.write(string_line)
        file.write('\n')
    file.close() #close file
```

```
torch.Size([2, 8, 8, 9])
```

In [40]:

```
### Complete this cell ###
tile_id = 0
kij = 0
W = w_tile[:,:,:,kij]  # w_tile[tile_num, array col num, array row num, kij]
bit_precision = 4

for kij_num in range(9):
    W_lis = []
    for out_ch in range(w_tile.size(1)):
      W = w_tile[:,out_ch,:,kij_num]
      W = torch.reshape(W, (-1, W.size(0)*W.size(1)))
      W_lis.append(W.tolist()[0])
    Z = torch.tensor(W_lis)
    #print(W[:,0,:])
    #print(Z[0,:])# w_tile[tile_num, array col num, array row num, kij]
    file = open('weight_mc_kij{}_int.txt'.format(kij_num), '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(Z.size(0)):
        lis = []
        for j in range(Z.size(1)): # row #
            lis.append(f'{round(Z[i,j].item())},')
        #print(lis)
        string_line = ''.join(reversed(lis))
        file.write(string_line)
        file.write('\n')
    file.close() #close file
```

In [34]:

```
### Complete this cell ###
ic_tile_id = 0
oc_tile_id = 0



kij = 0
nij = 0

bit_precision = 16

for kij_num in range(9):
    nij = 0
    psum_tile = psum[ic_tile_id,oc_tile_id,:,nij:nij+36,kij_num]
    psum_tile = torch.transpose(psum_tile,0,1)
    print(psum_tile.size())

    file = open('psum_mc_kij{}.txt'.format(kij_num), '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 i in range(psum_tile.size(0)):  # time step
```

```
        lis = []
        for j in range(psum_tile.size(1)): # row #
            if(psum_tile[i,j].item() >= 0):
                psum_tile_bin = '{0:016b}'.format(round(psum_tile[i,j].item()))
            else:
                psum_tile_bin = '{0:016b}'.format(round(psum_tile[i,j].item())+6553
            lis.append(psum_tile_bin)
            #for k in range(bit_precision):
            #   file.write(psum_tile_bin[k])
            #file.write(' ')  # for visibility with blank between words, you can us
        string_line = ''.join(reversed(lis))
        file.write(string_line)
        file.write('\n')
    file.close() #close file
```

```
torch.Size([36, 8])
torch.Size([36, 8])
torch.Size([36, 8])
torch.Size([36, 8])
torch.Size([36, 8])
torch.Size([36, 8])
torch.Size([36, 8])
torch.Size([36, 8])
torch.Size([36, 8])
```

In [ ]:

```
psum[:,:,:,:,:].size()
```

Out[ ]:

```
torch.Size([2, 1, 8, 36, 9])
```

In [35]:

```
### Complete this cell ###
ic_tile_id = 0
oc_tile_id = 0


kij = 0
nij = 0

bit_precision = 16

for kij_num in range(9):
    nij = 0
    psum_tile = psum[ic_tile_id,oc_tile_id,:,nij:nij+36,kij_num]
    psum_tile = torch.transpose(psum_tile,0,1)

    file = open('psum_mc_kij{}_int.txt'.format(kij_num), '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 i in range(psum_tile.size(0)):  # time step
        lis = []
        for j in range(psum_tile.size(1)): # row #
              lis.append(f'{round(psum_tile[i,j].item())},')
        string_line = ''.join(reversed(lis))
        file.write(string_line)
        #file.write(' ')  # for visibility with blank between words, you can use
        file.write('\n')
file.close() #close file
```

In [36]:

```
### Complete this cell ###
ic_tile_id = 0
oc_tile_id = 0
```

```
kij = 0
nij = 0

bit_precision = 16
file = open('out_mc.txt', 'w') #write to file
file.write('#out7feature0[msb-lsb],out6feature0[msb-lst],....,out0feature0[msb-lst]
file.write('#out7feature1[msb-lsb],out6feature1[msb-lst],....,out0feature1[msb-lst]
file.write('#...............#\n')

for i in range(out.size(1)):  # time step
    for j in range(out.size(0)): # row #
        if(out[7-j,i].item() >= 0):
            out_bin = '{0:016b}'.format(round(out[7-j,i].item()))
        else:
            out_bin = '{0:016b}'.format(round(out[7-j,i].item())+65536)
        for k in range(bit_precision):
            file.write(out_bin[k])
    file.write('\n')
file.close() #close file
```

In [37]:
```
### Complete this cell ###
ic_tile_id = 0
oc_tile_id = 0


kij = 0
nij = 0
relu = model.features[28]


bit_precision = 16
file = open('out_mc_relu.txt', 'w') #write to file
file.write('#out7feature0[msb-lsb],out6feature0[msb-lst],....,out0feature0[msb-lst]
file.write('#out7feature1[msb-lsb],out6feature1[msb-lst],....,out0feature1[msb-lst]
file.write('#...............#\n')

for i in range(out.size(1)):  # time step
    for j in range(out.size(0)): # row #
        out_bin = relu(torch.tensor(out[7-j,i].item()))
        out_bin_1 = '{0:016b}'.format(round(out_bin.item()))
        for k in range(bit_precision):
            file.write(out_bin_1[k])
        #file.write(' ')  # for visibility with blank between words, you can use
    file.write('\n')
file.close() #close file
```

In [38]:
```
### Complete this cell ###
ic_tile_id = 0
oc_tile_id = 0


kij = 0
nij = 0
psum_tile = psum[ic_tile_id,oc_tile_id,:,nij:nij+64,kij]
# psum[len(ic_tileg), len(oc_tileg), array_size, len(p_nijg), len(kijg)]

bit_precision = 16
file = open('out_int_relu.txt', 'w') #write to file
file.write('#out7feature0[msb-lsb],out6feature0[msb-lst],....,out0feature0[msb-lst]
file.write('#out7feature1[msb-lsb],out6feature1[msb-lst],....,out0feature1[msb-lst]
file.write('#...............#\n')
```

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

In [ ]: