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
import sys

sys.path.insert(1, '../../software')
from models import *


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



batch_size = 128
model_name = "final_VGG16_quant"
model = final_VGG16_quant()

print(model)

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 = [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=T
rue)
    (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=T
rue)
    (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=T
rue)
    (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): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats
=True)
    (31): ReLU(inplace=True)
    (32): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (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_stats
=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_stats
=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_stats
=True)
    (41): ReLU(inplace=True)
    (42): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (43): 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
```

In [2]:
```
# This cell won't be given, but students will complete the training

lr = 1e-2
weight_decay = 8e-4
epochs = 150
best_prec = 0

#model = nn.DataParallel(model).cuda()
```

```
model.cuda()
criterion = nn.CrossEntropyLoss().cuda()
optimizer = torch.optim.SGD(model.parameters(), lr=lr, momentum=0.9, weight_decay=w
#cudnn.benchmark = True

if not os.path.exists('result'):
    os.makedirs('result')
fdir = 'result/'+str(model_name)
if not os.path.exists(fdir):
    os.makedirs(fdir)


untrained = False

if (untrained):
    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)
```

In [3]:

```
# HW

#  1. Train with 4 bits for both weight and activation to achieve >90% accuracy
#  2. Find x_int and w_int for the 2nd convolution layer
#  3. Check the recovered psum has similar value to the un-quantized original psum
#     (such as example 1 in W3S2)
```

In [4]:

```
PATH = "result/final_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: 9019/10000 (90%)
```

In [5]:
```
#send an input and grap the value by using prehook like HW3
```

In [6]:
```
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()

for layer in model.modules():
    if isinstance(layer, torch.nn.Conv2d):
        print("prehooked")
        layer.register_forward_pre_hook(save_output)       ## Input for the module
#####################################################

use_gpu = torch.cuda.is_available()
device = torch.device("cuda" if use_gpu else "cpu")

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

print("1st convolution's input size:", save_output.outputs[0][0].size())
print("2nd convolution's input size:", save_output.outputs[1][0].size())
```

```
prehooked
prehooked
prehooked
prehooked
prehooked
prehooked
prehooked
prehooked
prehooked
prehooked
prehooked
prehooked
prehooked
1st convolution's input size: torch.Size([128, 3, 32, 32])
2nd convolution's input size: torch.Size([128, 64, 32, 32])
```

In [7]:
```
for index in range(len(save_output.outputs)):
    print(str(index) + ": " + str(save_output.outputs[index][0].size()))
```

```
0: torch.Size([128, 3, 32, 32])
1: torch.Size([128, 64, 32, 32])
2: torch.Size([128, 64, 16, 16])
3: torch.Size([128, 128, 16, 16])
4: torch.Size([128, 128, 8, 8])
5: torch.Size([128, 256, 8, 8])
6: torch.Size([128, 256, 8, 8])
7: torch.Size([128, 256, 4, 4])
8: torch.Size([128, 8, 4, 4])
9: torch.Size([128, 8, 4, 4])
10: torch.Size([128, 512, 2, 2])
11: torch.Size([128, 512, 2, 2])
12: torch.Size([128, 512, 2, 2])
```

In [8]:
```
w_bit = 4
weight_q = model.features[27].weight_q # quantized value is stored during the train
w_alpha = model.features[27].weight_quant.wgt_alpha  # alpha is defined in your mod
w_delta = w_alpha / (2**(w_bit-1)-1)    # delta can be calculated by using alpha an
weight_int = weight_q / w_delta # w_int can be calculated by weight_q and w_delta
#print(weight_int) # you should see clean integer numbers
```

In [9]:
```
x_bit = 4
x = save_output.outputs[8][0]  # input of the 2nd conv layer
x_alpha  = model.features[27].act_alpha
x_delta = x_alpha / (2**(x_bit)-1)

act_quant_fn = act_quantization(x_bit) # define the quantization function
x_q = act_quant_fn(x, x_alpha)         # create the quantized value for x

x_int = x_q / x_delta
#print(x_int) # you should see clean integer numbers
```

In [10]:
```
conv_int = torch.nn.Conv2d(in_channels = 8, out_channels=8, kernel_size = 3, paddin
conv_int.weight = torch.nn.parameter.Parameter(weight_int)

output_int =  conv_int.forward(x_int)    # output_int can be calculated with conv_i
output_recovered = output_int * x_delta * w_delta  # recover with x_delta and w_del
```

```
relu = torch.nn.ReLU()
relu_output_recovered = relu(output_recovered)
#print(output_recovered)
```

In [11]:
```
next_input = save_output.outputs[9][0]

print(next_input.size())
print(relu_output_recovered.size())
```

```
torch.Size([128, 8, 4, 4])
torch.Size([128, 8, 4, 4])
```

In [12]:
```
difference = abs( next_input - relu_output_recovered )
print(difference.mean())  ## It should be small, e.g.,2.3 in my trainned model
```

```
tensor(2.9908e-08, device='cuda:0', grad_fn=<MeanBackward0>)
```

In [13]:
```
print(x_int[0,:,:,:].size())
```

```
torch.Size([8, 4, 4])
```

In [14]:
```
print(x_int.size())
```

```
torch.Size([128, 8, 4, 4])
```

In [15]:
```
# act_int.size = torch.Size([128, 64, 32, 32])  <- batch_size, input_ch, ni, nj
a_int = x_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:(o



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

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 [16]:
```
print(a_tile.size())
print(psum.size())
```

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

In [17]:
```
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)

print(len(o_nijg))

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
```

```
16
```

In [18]:
```
print(o_ni_dim)
print(a_pad_ni_dim)
print(ki_dim)
```

```
4
6
3
```

In [19]:
```
temp_acc = 0
for row in range(8):
    for col in range(1):
        print(a_tile[0,row,7])
        print(w_tile[0,col,row,0])
        temp_acc = temp_acc + a_tile[0,row,7] * w_tile[0, col, row, 0]
print(temp_acc)
```

```
tensor(1., device='cuda:0', grad_fn=<SelectBackward0>)
tensor(-7.0000, device='cuda:0', grad_fn=<SelectBackward0>)
tensor(0., device='cuda:0', grad_fn=<SelectBackward0>)
tensor(-7.0000, device='cuda:0', grad_fn=<SelectBackward0>)
tensor(0., device='cuda:0', grad_fn=<SelectBackward0>)
tensor(-7.0000, device='cuda:0', grad_fn=<SelectBackward0>)
tensor(4., device='cuda:0', grad_fn=<SelectBackward0>)
tensor(-7.0000, device='cuda:0', grad_fn=<SelectBackward0>)
tensor(4., device='cuda:0', grad_fn=<SelectBackward0>)
tensor(-7.0000, device='cuda:0', grad_fn=<SelectBackward0>)
tensor(2., device='cuda:0', grad_fn=<SelectBackward0>)
tensor(7.0000, device='cuda:0', grad_fn=<SelectBackward0>)
tensor(1., device='cuda:0', grad_fn=<SelectBackward0>)
tensor(7.0000, device='cuda:0', grad_fn=<SelectBackward0>)
tensor(0., device='cuda:0', grad_fn=<SelectBackward0>)
tensor(-7.0000, device='cuda:0', grad_fn=<SelectBackward0>)
tensor(-42.0000, device='cuda:0', grad_fn=<AddBackward0>)
```

In [20]:
```
### show this cell partially. The following cells should be printed by students ###
tile_id = 0
nij = 200 # just a random number
X = a_tile[tile_id,:,:]  # [tile_num, array row num, time_steps]

bit_precision = 4
file = open('activation.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[X.size(0)-1-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
```

In [21]:
```
### Complete this cell ###
tile_id = 0
kij = 0


bit_precision = 4
for kij in range(9):
    W = w_tile[tile_id,:,:,kij]  # w_tile[tile_num, array col num, array row num, k
    file = open('weight_' + str(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)):  # col
        for j in range(W.size(1)): # row #
            W_bin = '{0:04b}'.format(round(W[i,7-j].item()) + (16 if (round(W[i,7-j
            for k in range(bit_precision):
                file.write(W_bin[k])
            #file.write(' ')  # for visibility with blank between words, you can us
        file.write('\n')
    file.close() #close file
```

In [22]:
```
### Complete this cell ###
ic_tile_id = 0
oc_tile_id = 0


kij = 0
nij = 200
# psum[len(ic_tileg), len(oc_tileg), array_size, len(p_nijg), len(kijg)]


bit_precision = 16

for kij in range(9):
    psum_tile = psum[ic_tile_id,oc_tile_id,:,:,kij]
    file = open('psum_' + str(kij) + '.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 i in range(psum_tile.size(1)):  # time step
        for j in range(psum_tile.size(0)): # col #
            #psum_bin = '{0:016b}'.format(round(psum_tile[psum_tile.size(0)-1-j,i].
            curr_psum = round(psum_tile[psum_tile.size(0)-1-j,i].item())
            if (i == 7 and kij == 0):
                print(curr_psum)
            if (curr_psum < 0):
                if (i == 7 and kij == 0):
                    print('{0:016b}'.format(curr_psum))
                curr_psum = curr_psum + 2**bit_precision
                if (i == 7 and kij == 0):
                    print('{0:016b}'.format(curr_psum))
            psum_bin = '{0:016b}'.format(curr_psum)
```

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

```
28
-56
-000000000111000
1111111111001000
-28
-000000000011100
1111111111100100
14
56
-14
-000000000001110
1111111111110010
-28
-000000000011100
1111111111100100
-42
-000000000101010
1111111111010110
```

In [23]:
```
### Complete this cell ###


bit_precision = 16
# out is array of size columns x len(o_nijg)

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 i in range(out.size(1)):  # time step
    for j in range(out.size(0)): # row #
        out_bin = '{0:016b}'.format(0 if (round(out[out.size(0)-1-j,i].item()) < 0)
        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
```

In [24]:
```
print(out[0,:])
```

```
tensor([-1.4000e+01,  1.4000e+01, -1.3300e+02, -7.0000e+00, -1.1444e-05,
         1.3300e+02, -1.2600e+02, -3.8147e-06, -7.0000e+00, -2.8000e+01,
        -1.7500e+02, -1.1900e+02,  1.4000e+01,  4.9000e+01, -4.9000e+01,
         5.6000e+01], device='cuda:0', grad_fn=<SliceBackward0>)
```

In [25]:
```
print(out[0,:])
```

```
tensor([-1.4000e+01,  1.4000e+01, -1.3300e+02, -7.0000e+00, -1.1444e-05,
         1.3300e+02, -1.2600e+02, -3.8147e-06, -7.0000e+00, -2.8000e+01,
        -1.7500e+02, -1.1900e+02,  1.4000e+01,  4.9000e+01, -4.9000e+01,
         5.6000e+01], device='cuda:0', grad_fn=<SliceBackward0>)
```

In [26]:
```
print(X.size())
```

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

In [27]:
```
print(o_ni_dim)
print(a_pad_ni_dim)
```

```
4
6
```

In [28]:
```
print(weight_int.size())
```

```
torch.Size([8, 8, 3, 3])
```

In [ ]:

In [ ]:

In [ ]: