test_custom_VGGNet16

December 6, 2024

[2]:
```
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.quant_layer import *
from models.VGG16_custom import *
```

[3]:
```
use_gpu = torch.cuda.is_available()
device = torch.device("cuda" if use_gpu else "cpu")
use_gpu, torch.cuda.get_device_name()
```

[3]:
```
(True, 'NVIDIA GeForce GTX 1080 Ti')
```

[4]:
```
batch_size = 256
model_name = "VGG16_custom1"
model = VGG16_custom()
```

[5]:
```
fdir = 'result/'+str(model_name)+'/model_best.pth.tar'
checkpoint = torch.load(fdir)
model.load_state_dict(checkpoint['state_dict'])
```

[5]:
```
<All keys matched successfully>
```

[6]:
```
# means and stds for individual RGB channels
# image = (image - mean) / std
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,
    ]))

test_dataset = torchvision.datasets.CIFAR10(
    root='./data',
    train=False,
    download=True,
    transform=transforms.Compose([
        transforms.ToTensor(),
        normalize,
    ]))

trainloader = torch.utils.data.DataLoader(train_dataset, batch_size=batch_size,␣
 ↪shuffle=True, num_workers=2)
testloader = torch.utils.data.DataLoader(test_dataset, batch_size=batch_size,␣
 ↪shuffle=False, num_workers=2)
```

```
Files already downloaded and verified
Files already downloaded and verified
```

[7]:
```
print_freq = len(testloader) / 4
print(print_freq)
```

```
10.0
```

[8]:
```
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
```

[9]:
```
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) # topk(k, dim=None,␣
↪largest=True, sorted=True)
                                                # will output (max value, its␣
↪index)
    pred = pred.t()           # transpose
    correct = pred.eq(target.view(1, -1).expand_as(pred))   # "-1": calculate␣
↪automatically

    res = []
    for k in topk:
        correct_k = correct[:k].view(-1).float().sum(0)  # view(-1): make a␣
↪flattened 1D tensor
        res.append(correct_k.mul_(100.0 / batch_size))   # correct: size of␣
↪[maxk, batch_size]
    return res
```

[10]:
```
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     ## n is impact factor
        self.count += n
        self.avg = self.sum / self.count
```

[11]:
```
criterion = nn.CrossEntropyLoss().cuda()

model.eval()
model.cuda()

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

```
Test: [0/40]    Time 4.897 (4.897)      Loss 0.3328 (0.3328)    Prec 92.578%
(92.578%)
Test: [10/40]   Time 0.044 (0.490)      Loss 0.2382 (0.3521)    Prec 94.141%
(92.045%)
Test: [20/40]   Time 0.044 (0.282)      Loss 0.2607 (0.3790)    Prec 91.406%
(91.574%)
Test: [30/40]   Time 0.029 (0.210)      Loss 0.4853 (0.3782)    Prec 87.891%
(91.557%)
 * Prec 91.570%
```

[11]:
```
model
```

[11]:
```
VGG_quant(
  (features): Sequential(
    (0): QuantConv2d(
      3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
      (weight_quant): weight_quantize_fn()
    )
    (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (2): ReLU(inplace=True)
    (3): QuantConv2d(
      64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
```

```
      (weight_quant): weight_quantize_fn()
    )
    (4): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (5): ReLU(inplace=True)
    (6): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
    (7): QuantConv2d(
      64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
      (weight_quant): weight_quantize_fn()
    )
    (8): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (9): ReLU(inplace=True)
    (10): QuantConv2d(
      128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
      (weight_quant): weight_quantize_fn()
    )
    (11): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (12): ReLU(inplace=True)
    (13): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
    (14): QuantConv2d(
      128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
      (weight_quant): weight_quantize_fn()
    )
    (15): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (16): ReLU(inplace=True)
    (17): QuantConv2d(
      256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
      (weight_quant): weight_quantize_fn()
    )
    (18): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (19): ReLU(inplace=True)
    (20): QuantConv2d(
      256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
      (weight_quant): weight_quantize_fn()
    )
    (21): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (22): ReLU(inplace=True)
    (23): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
    (24): QuantConv2d(
```

```
      256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
      (weight_quant): weight_quantize_fn()
    )
    (25): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (26): ReLU(inplace=True)
    (27): QuantConv2d(
      512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
      (weight_quant): weight_quantize_fn()
    )
    (28): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (29): ReLU(inplace=True)
    (30): QuantConv2d(
      512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
      (weight_quant): weight_quantize_fn()
    )
    (31): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (32): ReLU(inplace=True)
    (33): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
    (34): QuantConv2d(
      512, 8, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
      (weight_quant): weight_quantize_fn()
    )
    (35): BatchNorm2d(8, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (36): ReLU(inplace=True)
    (37): QuantConv2d(
      8, 8, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
      (weight_quant): weight_quantize_fn()
    )
    (38): ReLU(inplace=True)
    (39): QuantConv2d(
      8, 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)
)
```

[12]:
```python
class SaveOutput:
    def __init__(self):
        self.outputs = []
    def __call__(self, module, module_in):
        self.outputs.append(module_in)  # Save the input tensor
    def clear(self):
        self.outputs = []

save_output = SaveOutput()
device = torch.device("cuda" if use_gpu else "cpu")
i = 0
count=0

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

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

print(count)
```

```
3 -th layer prehooked
7 -th layer prehooked
12 -th layer prehooked
16 -th layer prehooked
21 -th layer prehooked
25 -th layer prehooked
29 -th layer prehooked
34 -th layer prehooked
38 -th layer prehooked
42 -th layer prehooked
47 -th layer prehooked
51 -th layer prehooked
54 -th layer prehooked
13
```

[13]:
```python
layer_input = save_output.outputs[11][0]
layer_output = save_output.outputs[12][0]
layer_input.size(), layer_output.size()
```

[13]:
```
(torch.Size([256, 8, 2, 2]), torch.Size([256, 8, 2, 2]))
```

[14]:
```python
layer_input = layer_input[0]
layer_output = layer_output[0]
layer_input.size(), layer_output.size()
```

[14]:
```
(torch.Size([8, 2, 2]), torch.Size([8, 2, 2]))
```

[15]:
```python
# grab data from the 37th layer!!!

layer = model.features[37]
print(layer)

print(layer._parameters.keys())

print(layer.weight_quant._parameters)
```

```
QuantConv2d(
  8, 8, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
  (weight_quant): weight_quantize_fn()
)
odict_keys(['weight', 'bias', 'act_alpha', 'weight_q'])
OrderedDict([('wgt_alpha', Parameter containing:
tensor(2.3064, device='cuda:0', requires_grad=True))])
```

[16]:
```python
bw = 4
weight_q = layer.weight_q
w_alpha = layer.weight_quant.wgt_alpha
w_delta = w_alpha / (2**(bw-1)-1)
w_int = weight_q / w_delta

print(w_int.shape)
print(w_int)
```

```
torch.Size([8, 8, 3, 3])
tensor([[[[-0.0000,  0.0000,  0.0000],
          [ 0.0000,  1.0000, -3.0000],
          [-1.0000,  2.0000, -1.0000]],

         [[-1.0000,  4.0000, -2.0000],
          [-3.0000,  7.0000,  2.0000],
          [-0.0000,  5.0000, -4.0000]],

         [[ 1.0000,  7.0000, -5.0000],
          [ 1.0000, -0.0000,  5.0000],
          [-2.0000, -4.0000, -2.0000]],

         [[-1.0000,  2.0000, -1.0000],
          [ 3.0000,  3.0000, -2.0000],
          [-1.0000,  6.0000, -4.0000]],
```

```
 [[-1.0000, -5.0000, -5.0000],
  [-2.0000,  1.0000,  0.0000],
  [-2.0000,  0.0000, -1.0000]],

 [[-2.0000,  1.0000, -1.0000],
  [ 0.0000, -2.0000, -3.0000],
  [-4.0000, -4.0000,  1.0000]],

 [[ 1.0000, -2.0000,  2.0000],
  [ 1.0000,  5.0000,  4.0000],
  [ 0.0000, -1.0000,  2.0000]],

 [[-4.0000, -3.0000, -2.0000],
  [-2.0000, -1.0000,  1.0000],
  [-5.0000, -1.0000, -3.0000]]],


[[[ 3.0000,  3.0000,  4.0000],
  [-3.0000, -2.0000, -1.0000],
  [-3.0000, -2.0000, -2.0000]],

 [[-4.0000,  2.0000,  5.0000],
  [-6.0000,  2.0000, -1.0000],
  [-5.0000,  1.0000,  6.0000]],

 [[ 2.0000,  6.0000,  0.0000],
  [-3.0000,  1.0000, -4.0000],
  [-1.0000, -7.0000, -6.0000]],

 [[ 1.0000, -4.0000,  1.0000],
  [ 0.0000,  6.0000,  0.0000],
  [-6.0000, -3.0000, -4.0000]],

 [[-1.0000,  3.0000,  2.0000],
  [ 6.0000,  7.0000,  0.0000],
  [ 4.0000,  2.0000, -2.0000]],

 [[ 3.0000, -6.0000, -1.0000],
  [-2.0000, -4.0000, -7.0000],
  [-3.0000,  2.0000, -4.0000]],

 [[-0.0000, -2.0000, -3.0000],
  [ 2.0000,  1.0000,  0.0000],
  [ 1.0000, -0.0000, -2.0000]],

 [[-4.0000,  1.0000, -3.0000],
  [-1.0000,  3.0000, -3.0000],
```

```
  [-3.0000,  3.0000,  3.0000]]],


[[[-1.0000, -2.0000, -2.0000],
  [-1.0000,  1.0000, -2.0000],
  [-2.0000, -0.0000,  3.0000]],

 [[ 1.0000, -3.0000, -2.0000],
  [ 2.0000, -1.0000,  2.0000],
  [ 6.0000, -4.0000, -2.0000]],

 [[-3.0000,  0.0000,  4.0000],
  [-2.0000, -2.0000, -1.0000],
  [-1.0000,  7.0000,  1.0000]],

 [[-1.0000, -3.0000, -0.0000],
  [-5.0000,  1.0000,  5.0000],
  [ 0.0000,  3.0000,  4.0000]],

 [[ 6.0000,  5.0000, -6.0000],
  [ 4.0000,  7.0000,  4.0000],
  [ 0.0000, -1.0000, -1.0000]],

 [[ 5.0000, -2.0000, -1.0000],
  [ 1.0000,  3.0000,  2.0000],
  [-1.0000, -1.0000,  2.0000]],

 [[ 1.0000,  6.0000,  2.0000],
  [-4.0000,  4.0000, -1.0000],
  [-3.0000, -0.0000, -5.0000]],

 [[ 3.0000, -5.0000, -1.0000],
  [ 2.0000, -2.0000, -5.0000],
  [-2.0000, -2.0000, -3.0000]]],


[[[ 0.0000,  4.0000,  0.0000],
  [ 4.0000,  2.0000, -1.0000],
  [-2.0000,  4.0000,  1.0000]],

 [[ 2.0000,  3.0000,  3.0000],
  [ 2.0000, -1.0000, -3.0000],
  [-5.0000, -3.0000, -3.0000]],

 [[-0.0000,  6.0000,  3.0000],
  [ 1.0000,  4.0000, -2.0000],
  [-4.0000,  4.0000,  3.0000]],
```

```
 [[-2.0000, -4.0000, -1.0000],
  [-2.0000, -4.0000,  5.0000],
  [ 5.0000,  4.0000,  4.0000]],

 [[ 3.0000,  4.0000, -2.0000],
  [-5.0000, -1.0000, -4.0000],
  [-7.0000, -0.0000,  0.0000]],

 [[ 3.0000,  2.0000,  1.0000],
  [-0.0000,  6.0000,  5.0000],
  [ 2.0000,  1.0000,  0.0000]],

 [[-1.0000, -2.0000, -5.0000],
  [-0.0000,  0.0000,  3.0000],
  [-2.0000,  3.0000,  2.0000]],

 [[ 4.0000,  0.0000, -4.0000],
  [-2.0000, -5.0000, -2.0000],
  [-0.0000, -4.0000, -3.0000]]],


[[[-0.0000, -0.0000, -4.0000],
  [-3.0000, -3.0000, -2.0000],
  [-5.0000, -0.0000, -4.0000]],

 [[-0.0000, -2.0000, -3.0000],
  [ 1.0000,  3.0000,  0.0000],
  [-1.0000,  2.0000,  1.0000]],

 [[ 0.0000, -6.0000,  5.0000],
  [-2.0000,  5.0000,  5.0000],
  [-2.0000,  0.0000,  1.0000]],

 [[ 1.0000,  4.0000, -0.0000],
  [-1.0000,  4.0000, -1.0000],
  [ 3.0000,  5.0000, -5.0000]],

 [[-3.0000,  0.0000,  1.0000],
  [ 2.0000, -3.0000,  4.0000],
  [ 1.0000, -2.0000, -3.0000]],

 [[ 5.0000,  4.0000,  2.0000],
  [-4.0000, -2.0000,  3.0000],
  [ 3.0000,  3.0000, -5.0000]],

 [[ 1.0000,  3.0000, -2.0000],
  [ 1.0000, -1.0000,  5.0000],
  [-6.0000, -5.0000,  0.0000]],
```

```
 [[ 3.0000,  5.0000, -2.0000],
  [ 3.0000,  5.0000,  1.0000],
  [ 2.0000, -1.0000, -4.0000]]],


[[[-3.0000, -3.0000, -2.0000],
  [-3.0000,  5.0000,  4.0000],
  [-4.0000,  1.0000,  1.0000]],

 [[ 2.0000,  2.0000,  1.0000],
  [ 5.0000,  5.0000,  2.0000],
  [-1.0000,  5.0000,  1.0000]],

 [[-4.0000,  3.0000, -6.0000],
  [-1.0000,  0.0000, -0.0000],
  [-3.0000, -3.0000, -0.0000]],

 [[-2.0000, -1.0000,  4.0000],
  [-1.0000,  3.0000, -3.0000],
  [-5.0000, -4.0000,  4.0000]],

 [[ 5.0000, -4.0000, -3.0000],
  [ 3.0000, -1.0000, -5.0000],
  [-1.0000,  5.0000, -0.0000]],

 [[ 3.0000, -3.0000,  0.0000],
  [-1.0000,  4.0000,  1.0000],
  [ 1.0000,  5.0000, -1.0000]],

 [[ 1.0000,  0.0000,  1.0000],
  [ 1.0000, -7.0000, -1.0000],
  [-4.0000, -4.0000,  2.0000]],

 [[-2.0000,  1.0000,  0.0000],
  [-4.0000,  2.0000,  0.0000],
  [-3.0000, -0.0000,  2.0000]]],


[[[-2.0000, -2.0000,  2.0000],
  [ 2.0000,  1.0000,  2.0000],
  [-1.0000, -1.0000, -3.0000]],

 [[ 1.0000, -0.0000,  4.0000],
  [-2.0000, -3.0000, -5.0000],
  [-1.0000, -1.0000,  2.0000]],

 [[ 0.0000,  6.0000,  1.0000],
```

```
 [-1.0000, -0.0000, -1.0000],
 [-2.0000, -4.0000, -3.0000]],

[[ 0.0000, -4.0000,  2.0000],
 [-2.0000, -2.0000, -5.0000],
 [-2.0000, -0.0000, -1.0000]],

[[-3.0000,  0.0000,  2.0000],
 [ 3.0000, -7.0000, -3.0000],
 [ 5.0000,  1.0000, -3.0000]],

[[ 0.0000, -1.0000, -4.0000],
 [ 5.0000,  4.0000,  3.0000],
 [ 0.0000,  3.0000,  1.0000]],

[[-4.0000,  2.0000,  2.0000],
 [ 5.0000,  7.0000, -1.0000],
 [ 3.0000,  6.0000, -3.0000]],

[[ 2.0000,  6.0000, -2.0000],
 [ 1.0000,  3.0000,  1.0000],
 [-2.0000,  1.0000, -0.0000]]],


[[[-1.0000, -2.0000, -1.0000],
  [ 4.0000,  4.0000,  7.0000],
  [ 1.0000,  2.0000,  1.0000]],

 [[ 1.0000, -2.0000,  2.0000],
  [ 4.0000, -0.0000,  7.0000],
  [ 3.0000, -3.0000,  1.0000]],

 [[-2.0000,  2.0000, -1.0000],
  [-2.0000, -2.0000, -2.0000],
  [-3.0000, -1.0000, -0.0000]],

 [[-0.0000, -1.0000,  2.0000],
  [ 5.0000,  6.0000, -1.0000],
  [ 0.0000, -3.0000,  2.0000]],

 [[ 1.0000, -2.0000, -1.0000],
  [ 2.0000, -4.0000, -2.0000],
  [-4.0000,  2.0000, -4.0000]],

 [[-4.0000,  2.0000,  0.0000],
  [ 3.0000,  2.0000,  3.0000],
  [-3.0000, -0.0000, -0.0000]],
```

```
        [[-4.0000, -2.0000, -3.0000],
         [ 5.0000,  3.0000, -4.0000],
         [ 3.0000, -2.0000, -2.0000]],

        [[-3.0000,  4.0000, -2.0000],
         [-0.0000,  5.0000,  1.0000],
         [-4.0000, -1.0000,  0.0000]]]], device='cuda:0',
      grad_fn=<DivBackward0>)
```

[17]:

```
x = layer_input
x_alpha = model.features[37].act_alpha
x_delta = x_alpha / (2**(bw)-1)

act_quant_fn = act_quantization(bw)
x_q = act_quant_fn(x, x_alpha)

x_int = x_q / x_delta

print(x_int.shape)
print(x_int)
```

```
torch.Size([8, 2, 2])
tensor([[[ 0.0000,  0.0000],
         [ 0.0000,  7.0000]],

        [[ 6.0000,  4.0000],
         [ 0.0000,  0.0000]],

        [[ 0.0000,  9.0000],
         [ 0.0000,  0.0000]],

        [[ 0.0000,  0.0000],
         [ 0.0000,  0.0000]],

        [[11.0000,  0.0000],
         [ 1.0000,  0.0000]],

        [[ 6.0000,  2.0000],
         [ 8.0000, 14.0000]],

        [[ 0.0000,  0.0000],
         [ 0.0000,  0.0000]],

        [[ 5.0000,  0.0000],
         [ 0.0000,  0.0000]]], device='cuda:0', grad_fn=<DivBackward0>)
```

[18]:
```
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(w_int)
output_int = F.relu(conv_int(x_int))
output_recovered = output_int * w_delta * x_delta  # recover with x_delta and␣
    ↪w_delta

print(output_recovered.shape)
print(layer_output.shape)
print(output_int)
```

```
torch.Size([8, 2, 2])
torch.Size([8, 2, 2])
tensor([[[ 58.0000,   0.0000],
         [  0.0000,   9.0000]],

        [[  0.0000,  16.0000],
         [  0.0000,   0.0000]],

        [[122.0000,  34.0000],
         [ 71.0000, 162.0000]],

        [[  0.0000,  42.0000],
         [225.0000, 246.0000]],

        [[  0.0000, 139.0000],
         [ 83.0000,   0.0000]],

        [[101.0000, 149.0000],
         [  0.0000, 190.0000]],

        [[  0.0000,  92.0000],
         [122.0000, 141.0000]],

        [[ 18.0000,  36.0000],
         [100.0000,  74.0000]]], device='cuda:0', grad_fn=<ReluBackward0>)
```

[19]:
```
# calculate the difference between outputs, d should be less than 1e-03
diff = abs(layer_output - output_recovered)
print(diff.mean())
```

```
tensor(3.9116e-07, device='cuda:0', grad_fn=<MeanBackward0>)
```

[20]:
```
print(x_int.size())
```

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

[21]:
```
x_pad = torch.zeros(8, 4, 4).cuda()

x_pad[:, 1:3, 1:3] = x_int.cuda()

X = torch.reshape(x_pad, (x_pad.size(0), -1))

print(X.size())
```

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

[22]:
```
from pathlib import Path
# Define the folder path
folder_path = Path('./vgg_output/')

# Create the folder if it doesn't exist
folder_path.mkdir(parents=True, exist_ok=True)
```

[23]:
```
### store weights ###

bit_precision = 4
file = open('./vgg_output/activation.txt', 'w')
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(' ')  # use this line for visibility with blank between␣
 ↪words
    file.write('\n')
file.close() #close file
```

[24]:
```
print(w_int.size())
W = torch.reshape(w_int, (w_int.size(0), w_int.size(1), -1))
W.size()
```

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

[24]:
```
torch.Size([8, 8, 9])
```

[25]:
```
bit_precision = 4

for kij in range(9):
    file = open('./vgg_output/w{}.txt'.format(str(kij)), 'w')
```

```
    file.write('#col0row7[msb-lsb],col0row6[msb-lsb],....,col0row0[msb-lsb]#\n')
    file.write('#col1row7[msb-lsb],col1row6[msb-lsb],....,col1row0[msb-lsb]#\n')
    file.write('#................#\n')
    for i in range(W.size(0)):
        for j in range(W.size(1)):
            if (W[i, 7-j, kij].item()<0):
                W_bin = '{0:04b}'.format(int(W[i,7-j,kij].
 ↪item()+2**bit_precision+0.001))
            else:
                W_bin = '{0:04b}'.format(int(W[i,7-j,kij].item()+0.001))
            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
```

[25]:
```
### storing weight data ###

bit_precision = 4

file = open('./vgg_output/weight.txt', 'w')
file.write('#col0row7[msb-lsb],col0row6[msb-lsb],....,col0row0[msb-lsb]#\n')
file.write('#col1row7[msb-lsb],col1row6[msb-lsb],....,col1row0[msb-lsb]#\n')
file.write('#................#\n')

for kij in range(9):
    for i in range(W.size(0)):   #col
        for j in range(W.size(1)):    # row
            if (W[i, 7-j, kij].item()<0):
                W_bin = '{0:04b}'.format(round(W[i,7-j, kij].item() +␣
 ↪2**bit_precision))        #check again if it works for neg numbers
            else:
                W_bin = '{0:04b}'.format(round(W[i,7-j, kij].item()))
            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
```

[26]:
```
print(output_int.size())
O = torch.reshape(output_int, (output_int.size(0), -1))
print(O.size())
```

```
torch.Size([8, 2, 2])
torch.Size([8, 4])
```

[27]:
```python
### Store output data ###

bit_precision = 16
file = open('./vgg_output/output.txt', 'w') #write to file
file.write('#time0col7[msb-lsb],time0col6[msb-lsb],....,time0col0[msb-lsb]#\n')
file.write('#time1col7[msb-lsb],time1col6[msb-lsb],....,time1col0[msb-lsb]#\n')
file.write('#................#\n')

for i in range(O.size(1)):
    for j in range(O.size(0)):
        if (O[7-j,i].item()<0):
            O_bin = '{0:016b}'.format(round(O[7-j,i].item() + 2**bit_precision))
        else:
            O_bin = '{0:016b}'.format(round(O[7-j,i].item()))
        for k in range(bit_precision):
            file.write(O_bin[k])
        #file.write(' ')  # for visibility with blank between words, you can use
    file.write('\n')
file.close() #close file
```

[28]:
```python
print(X.size())
```

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

[29]:
```python
psum = torch.zeros(8, 16, 9).cuda()  #initialize an empty psum first with array␣
 ↪size, p_nij and kij
print(psum.size())

# calculate psum value
for kij in range(9):
    for p_nij in range(16):
        m = nn.Linear(8, 8, bias=False)  # array size matched
        m.weight = torch.nn.Parameter(W[:,:,kij])
        psum[:, p_nij, kij] = m(X[:,p_nij]).cuda()
```

```
torch.Size([8, 16, 9])
```

[30]:
```python
### Store psum data, not needed for output stationary model ###

bit_precision = 16

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

for kij in range(9):
```

```
    for i in range(psum.size(1)):  # time step
        for j in range(psum.size(0)): # array size
            if (psum[7-j,i, kij].item()<0):
                psum_bin = '{0:016b}'.format(round(psum[7-j,i, kij].item() +
 ↪2**bit_precision))
            else:
                psum_bin = '{0:016b}'.format(round(psum[7-j,i, kij].item()))
            for k in range(bit_precision):
                file.write(psum_bin[k])
            #file.write(' ')  # for visibility with blank between words, you
 ↪can use
        file.write('\n')
file.close() #close file
```

[ ]:

```
```

[ ]:

```
```