VGG16_Quantization_Aware_Training

December 11, 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
print('=> Building model...')



batch_size = 128
model_name = "VGG16_quant_part2_2bit"
model = VGG16_quant_part2_2bit()

print(model)

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


train_dataset = torchvision.datasets.CIFAR10(
    root='./data',
    train=True,
    download=True,
```

```
    transform=transforms.Compose([
        transforms.RandomCrop(32, padding=4),
        transforms.RandomHorizontalFlip(),
        transforms.ToTensor(),
        normalize,
    ]))
trainloader = torch.utils.data.DataLoader(train_dataset, batch_size=batch_size,␣
 ↪shuffle=True, num_workers=2)


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

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


print_freq = 100 # every 100 batches, accuracy printed. Here, each batch␣
 ↪includes "batch_size" data points
# CIFAR10 has 50,000 training data, and 10,000 validation data.

def train(trainloader, model, criterion, optimizer, epoch):
    batch_time = AverageMeter()
    data_time = AverageMeter()
    losses = AverageMeter()
    top1 = AverageMeter()

    model.train()

    end = time.time()
    for i, (input, target) in enumerate(trainloader):
        # measure data loading time
        data_time.update(time.time() - end)

        input, target = input.cuda(), target.cuda()

        # compute output
        output = model(input)
        loss = criterion(output, target)

        # measure accuracy and record loss
```

```
        prec = accuracy(output, target)[0]
        losses.update(loss.item(), input.size(0))
        top1.update(prec.item(), input.size(0))

        # compute gradient and do SGD step
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        # measure elapsed time
        batch_time.update(time.time() - end)
        end = time.time()


        if i % print_freq == 0:
            print('Epoch: [{0}][{1}/{2}]\t'
                  'Time {batch_time.val:.3f} ({batch_time.avg:.3f})\t'
                  'Data {data_time.val:.3f} ({data_time.avg:.3f})\t'
                  'Loss {loss.val:.4f} ({loss.avg:.4f})\t'
                  'Prec {top1.val:.3f}% ({top1.avg:.3f}%)'.format(
                   epoch, i, len(trainloader), batch_time=batch_time,
                   data_time=data_time, loss=losses, top1=top1))



def validate(val_loader, model, criterion ):
    batch_time = AverageMeter()
    losses = AverageMeter()
    top1 = AverageMeter()

    # switch to evaluate mode
    model.eval()

    end = time.time()
    with torch.no_grad():
        for i, (input, target) in enumerate(val_loader):

            input, target = input.cuda(), target.cuda()

            # compute output
            output = model(input)
            loss = criterion(output, target)

            # measure accuracy and record loss
            prec = accuracy(output, target)[0]
            losses.update(loss.item(), input.size(0))
            top1.update(prec.item(), input.size(0))
```

```
            # measure elapsed time
            batch_time.update(time.time() - end)
            end = time.time()

            if i % print_freq == 0:  # This line shows how frequently print out the status. e.g., i%5 => every 5 batch, prints out
                print('Test: [{0}/{1}]\t'
                  'Time {batch_time.val:.3f} ({batch_time.avg:.3f})\t'
                  'Loss {loss.val:.4f} ({loss.avg:.4f})\t'
                  'Prec {top1.val:.3f}% ({top1.avg:.3f}%)'.format(
                   i, len(val_loader), batch_time=batch_time, loss=losses,
                   top1=top1))

    print(' * Prec {top1.avg:.3f}% '.format(top1=top1))
    return top1.avg


def accuracy(output, target, topk=(1,)):
    """Computes the precision@k for the specified values of k"""
    maxk = max(topk)
    batch_size = target.size(0)

    _, pred = output.topk(maxk, 1, True, True)
    pred = pred.t()
    correct = pred.eq(target.view(1, -1).expand_as(pred))

    res = []
    for k in topk:
        correct_k = correct[:k].view(-1).float().sum(0)
        res.append(correct_k.mul_(100.0 / batch_size))
    return res


class AverageMeter(object):
    """Computes and stores the average and current value"""
    def __init__(self):
        self.reset()

    def reset(self):
        self.val = 0
        self.avg = 0
        self.sum = 0
        self.count = 0

    def update(self, val, n=1):
        self.val = val
```

```
        self.sum += val * n
        self.count += n
        self.avg = self.sum / self.count


def save_checkpoint(state, is_best, fdir):
    filepath = os.path.join(fdir, 'checkpoint.pth')
    torch.save(state, filepath)
    if is_best:
        shutil.copyfile(filepath, os.path.join(fdir, 'model_best.pth.tar'))


def adjust_learning_rate(optimizer, epoch):
    """For VGGNet, the lr starts from 0.01, and is divided by 10 at 50 and 100␣
 ↪epochs"""
    adjust_list = [80, 120, 160]
    if epoch in adjust_list:
        for param_group in optimizer.param_groups:
            param_group['lr'] = param_group['lr'] * 0.1
```

```
=> Building model…
VGG16_quant_part2_2(
  (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, 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_stats=True)
    (26): ReLU(inplace=True)
    (27): QuantConv2d(
      16, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
      (weight_quant): weight_quantize_fn()
    )
    (28): ReLU(inplace=True)
    (29): QuantConv2d(
      16, 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
```

[ ]:
```
# # Training Cell

# # PATH = "result/VGG16_quant_part2_2bit/model_best.pth.tar"
# # checkpoint = torch.load(PATH)
# # model.load_state_dict(checkpoint['state_dict'])

# lr = 5e-2
# weight_decay = 1e-4
# epochs = 200
# 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=weight_decay)
# #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)


# 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)
```

[3]:
```
# Load trained model
PATH = "result/VGG16_quant_part2_2bit/model_best.pth.tar"
checkpoint = torch.load(PATH)
model.load_state_dict(checkpoint['state_dict'])
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

model.cuda()
model.eval()

# Evaluate model accuracy
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()

print('\nTest set: Accuracy: {}/{} ({:.0f}%)\n'.format(
        correct, len(testloader.dataset),
        100. * correct / len(testloader.dataset)))
```

```
Test set: Accuracy: 8901/10000 (89%)
```

[4]:
```
# Capture layer inputs using forward pre-hooks
class SaveOutput:
    def __init__(self):
        self.outputs = []
    def __call__(self, module, module_in):
        self.outputs.append(module_in[0])
    def clear(self):
        self.outputs = []

# Register hooks for both layers we need to capture
save_output = SaveOutput()
save_output_next = SaveOutput()
target_layer = model.features[27]  # Current conv layer
target_layer_next = model.features[29]  # Next conv layer (after ReLU)
target_layer.register_forward_pre_hook(save_output)
target_layer_next.register_forward_pre_hook(save_output_next)

# Get a sample batch
dataiter = iter(testloader)
images, labels = next(dataiter)
images = images.cuda()

# Single forward pass to capture both inputs
model.eval()
with torch.no_grad():
    _ = model(images)

x = save_output.outputs[0]  # Input to features[27]
x_next_ref = save_output_next.outputs[0]  # Input to features[29] (after ReLU)
print(f"Captured input shape (features[27]): {x.shape}")
print(f"Captured next layer input shape (features[29]): {x_next_ref.shape}")
```

```
Captured input shape (features[27]): torch.Size([128, 16, 4, 4])
Captured next layer input shape (features[29]): torch.Size([128, 16, 4, 4])
```

[5]:
```
# Convert quantized weights to integers
w_bit = 4
target_conv = model.features[27]
```

```
weight_q = target_conv.weight_q.data
w_alpha = target_conv.weight_quant.wgt_alpha.data.item()
w_delta = w_alpha / (2 ** (w_bit-1) - 1)
# divide by alpha first to get normalized values and multiply by (2^(w_bit - 1)
 ↪- 1) to get integers
weight_int = torch.round(weight_q / w_delta).int()
print(f"Weight alpha: {w_alpha:.4f}, Weight delta: {w_delta:.4f}")
print(f"Weight int shape: {weight_int.shape}")
print(f"Weight int sample (first few values): {weight_int.flatten()[:10]}")
```

```
Weight alpha: 1.2017, Weight delta: 0.1717
Weight int shape: torch.Size([16, 16, 3, 3])
Weight int sample (first few values): tensor([ 1,  2,  7,  5, -3,  7,  7,  7,
7,  0], device='cuda:0',
       dtype=torch.int32)
```

[6]:
```
# Convert quantized activations to integers
x_bit = 2
target_conv = model.features[27]
x_alpha = target_conv.act_alpha.data.item()
x_delta = x_alpha / (2 ** x_bit - 1)

act_quant_fn = act_quantization(x_bit)
x_q = act_quant_fn(x, x_alpha)
x_int = torch.round(x_q / x_delta).int()
print(f"Activation alpha: {x_alpha:.4f}, Activation delta: {x_delta:.4f}")
print(f"x_int shape: {x_int.shape}")
print(f"x_int sample (first few values): {x_int.flatten()[:10]}")
```

```
Activation alpha: 4.1917, Activation delta: 1.3972
x_int shape: torch.Size([128, 16, 4, 4])
x_int sample (first few values): tensor([0, 0, 0, 0, 0, 0, 0, 0, 1, 1],
device='cuda:0', dtype=torch.int32)
```

[7]:
```
# Perform integer convolution and recover floating-point output
conv_int = torch.nn.Conv2d(
    in_channels=16,
    out_channels=16,
    kernel_size=3,
    padding=1,
    bias=False
)
conv_int.weight = torch.nn.parameter.Parameter(weight_int.float())

output_int = conv_int(x_int.float())
output_recovered = output_int * x_delta * w_delta
print(f"Output recovered shape: {output_recovered.shape}")
print(f"Output recovered sample: {output_recovered[0, 0, :5, :5]}")
```

```
Output recovered shape: torch.Size([128, 16, 4, 4])
Output recovered sample: tensor([[-16.3110, -26.8652, -19.4293, -11.9934],
        [-31.4227, -49.8925, -37.6593, -20.3888],
        [  7.1960,   5.9967,   5.2771,  -3.3582],
        [ 24.2267,  27.8247,  29.2639,  12.9529]], device='cuda:0',
       grad_fn=<SliceBackward0>)
```

[8]:
```python
# Apply ReLU to the recovered output and compare with reference
output_recovered_relu = F.relu(output_recovered)
difference = abs(x_next_ref - output_recovered_relu)
print(f"Difference mean: {difference.mean():.6f}")
print(f"Difference max: {difference.max():.6f}")
print(f"Difference should be < 10^-3: {difference.mean() < 1e-3}")
```

```
Difference mean: 0.000001
Difference max: 0.000046
Difference should be < 10^-3: True
```

[9]:
```python
# Prepare data for systolic array computation
# Extract single sample from batch
a_int = x_int[0,:,:,:]  # Shape: [16, 4, 4] - input channels, height, width

# Reshape weights: [out_ch, in_ch, ki, kj] -> [out_ch, in_ch, kij]
w_int = torch.reshape(weight_int, (weight_int.size(0), weight_int.size(1), -1))
 ↪ # [16, 16, 9]

padding = 1
stride = 1
# For 2-bit activations: array handles 16(Input Channel) x 8(Output Channel)
# For 4-bit activations: array handles 8(Input Channel) x 8(Output Channel)
ic_array_size = 16  # Input channels per tile for 2-bit mode (16 IC fit in one
 ↪tile)
oc_array_size = 8   # Output channels per tile (8 OC per tile, need 2 tiles for
 ↪16 OC)

# Define ranges
nig = range(a_int.size(1))  # ni (height)
njg = range(a_int.size(2))   # nj (width)
icg = range(int(w_int.size(1)))  # input channels (16)
ocg = range(int(w_int.size(0)))  # output channels (16)
ic_tileg = range(int(len(icg)/ic_array_size))  # input channel tiles: 16/16 = 1
 ↪tile
oc_tileg = range(int(len(ocg)/oc_array_size))  # output channel tiles: 16/8 = 2
 ↪tiles
kijg = range(w_int.size(2))  # kernel elements (3x3 = 9)
ki_dim = int(math.sqrt(w_int.size(2)))  # kernel dimension (3)

# Pad activations
```

```
a_pad = torch.zeros(len(icg), len(nig)+padding*2, len(njg)+padding*2).cuda()
a_pad[:, padding:padding+len(nig), padding:padding+len(njg)] = a_int.cuda()
a_pad = torch.reshape(a_pad, (a_pad.size(0), -1))  # [16, 36] - flattened␣
 ↪spatial dimensions

# Create tiles for systolic array
# For 2-bit: a_tile shape [ic_tiles=1, 16_IC, spatial_positions]
#           w_tile shape [oc_tiles*ic_tiles=2, 8_OC, 16_IC, kij]
a_tile = torch.zeros(len(ic_tileg), ic_array_size, a_pad.size(1)).cuda()
w_tile = torch.zeros(len(oc_tileg)*len(ic_tileg), oc_array_size, ic_array_size,␣
 ↪len(kijg)).cuda()

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

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

# Compute partial sums using matrix multiplication (simulating systolic array)
# For 2-bit: psum shape [ic_tiles=1, oc_tiles=2, 8_OC, spatial_positions, kij]
p_nijg = range(a_pad.size(1))
psum = torch.zeros(len(ic_tileg), len(oc_tileg), oc_array_size, len(p_nijg),␣
 ↪len(kijg)).cuda()

for kij in kijg:
    for ic_tile in ic_tileg:
        for oc_tile in oc_tileg:
            for nij in p_nijg:
                # Matrix multiply: [8 OC] = [8 OC x 16 IC] @ [16 IC]
                m = nn.Linear(ic_array_size, oc_array_size, bias=False)
                m.weight = torch.nn.
 ↪Parameter(w_tile[len(ic_tileg)*oc_tile+ic_tile,:,:,kij])
                psum[ic_tile, oc_tile, :, nij, kij] = m(a_tile[ic_tile,:,nij]).
 ↪cuda()
```

[10]:
```
# Accumulate partial sums across tiles and kernel elements
a_pad_ni_dim = int(math.sqrt(a_pad.size(1)))  # 6 (4+2*padding)
o_ni_dim = int((a_pad_ni_dim - (ki_dim- 1) - 1)/stride + 1)  # 4
o_nijg = range(o_ni_dim**2)  # 16 output spatial positions

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

```
# Accumulate across output tiles and input tiles
for oc_tile in oc_tileg:  # Loop over output tiles (0, 1 for 16 output channels
 ↪/ 8 OC per tile)
    partial_out = torch.zeros(oc_array_size, len(o_nijg)).cuda()

    # Accumulate across input tiles
    for ic_tile in ic_tileg:  # For 2-bit: only 1 IC tile (IC 0-15 fit in one
 ↪tile)
        for kij in kijg:
            ky = kij // ki_dim     # kernel row
            kx = kij %  ki_dim     # kernel col

            for o_nij in o_nijg:
                oy = o_nij // o_ni_dim
                ox = o_nij %  o_ni_dim
                nij_src = (oy + ky) * a_pad_ni_dim + (ox + kx)

                partial_out[:, o_nij] += psum[
                    ic_tile,    # input tile index
                    oc_tile,    # output tile index
                    :,          # 8 output channels (PE columns)
                    nij_src,    # spatial index
                    kij         # kernel element
                ]

    # Store this 8-channel block
    out[oc_tile * oc_array_size:(oc_tile + 1) * oc_array_size, :] = partial_out

# Reshape output to match original shape [16, 4, 4]
out_reshaped = out.view(len(ocg), o_ni_dim, o_ni_dim)

# Recover the output and apply ReLU
out_recovered = out_reshaped * x_delta * w_delta
out_recovered_relu = F.relu(out_recovered)

print(f"Output recovered shape: {out_recovered_relu.shape}")
print(f"Output recovered sample: {out_recovered_relu[0, :5, :5]}")

# Compare with reference
difference_tiled = abs(x_next_ref[0] - out_recovered_relu)
print(f"\nTiled computation difference mean: {difference_tiled.mean():.6f}")
print(f"Tiled computation difference max: {difference_tiled.max():.6f}")
print(f"Tiled computation difference should be < 10^-3: {difference_tiled.
 ↪mean() < 1e-3}")
```

```
Output recovered shape: torch.Size([16, 4, 4])
```

```
Output recovered sample: tensor([[ 0.0000,  0.0000,  0.0000,  0.0000],
        [ 0.0000,  0.0000,  0.0000,  0.0000],
        [ 7.1960,  5.9967,  5.2771,  0.0000],
        [24.2267, 27.8247, 29.2639, 12.9529]], device='cuda:0',
       grad_fn=<SliceBackward0>)

Tiled computation difference mean: 0.000001
Tiled computation difference max: 0.000011
Tiled computation difference should be < 10^-3: True
```

[11]:
```python
# Generate activation data file for hardware testing (2-bit activations)
# For 2-bit: array has 8 rows, each row processes 2 input channels (16 IC total)
# So we need to pack 2 channels per row: row 0 = IC[0,1], row 1 = IC[2,3], ..., row 7 = IC[14,15]
ic_tile_id = 0
nij = 0
nij_end = min(nij + 64, a_tile.size(1))
X_2b = a_tile[ic_tile_id, :, nij:nij_end]  # Shape: [16 IC, time]

bit_precision = 2
file = open('activation_tile0.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 2-bit mode: 8 rows, each row has 2 channels (packed as 2 bits each)
# Format: row 0 = IC[1,0], row 1 = IC[3,2], ..., row 7 = IC[15,14]
for i in range(X_2b.size(1)):        # time
    for row_idx in range(8):          # 8 PE rows
        # Each row processes 2 channels: row 0 handles IC[0,1], row 1 handles IC[2,3], etc.
        ic0_idx = row_idx * 2         # First channel for this row
        ic1_idx = row_idx * 2 + 1     # Second channel for this row

        # Get values for both channels
        val0 = int(round(X_2b[ic0_idx, i].item()))
        val1 = int(round(X_2b[ic1_idx, i].item()))

        # Convert to unsigned if negative
        if val0 < 0:
            val0 = val0 + (2**bit_precision)
        if val1 < 0:
            val1 = val1 + (2**bit_precision)

        # Write both 2-bit values: IC1 then IC0 (as per hardware expectation)
        X_bin1 = '{0:02b}'.format(val1)  # Second channel (IC1)
        X_bin0 = '{0:02b}'.format(val0)  # First channel (IC0)
```

```
        file.write(X_bin1 + X_bin0)      # Write IC1 then IC0 (4 bits total per
 ↪row)
    file.write('\n')
file.close()
print(f"Written activation data: shape {X_2b.shape}, nij range [{nij}:
 ↪{nij_end}]")
print(f"Format: 8 rows × 4 bits (2 channels × 2 bits each) per time step")
```

```
Written activation data: shape torch.Size([16, 16]), nij range [0:16]
Format: 8 rows × 4 bits (2 channels × 2 bits each) per time step
```

[12]:
```
# Generate weight data files for hardware testing
# Generate files for all kij values (0-8) as expected by testbench
ic_tile_id = 0
oc_tile_id = 0
tile_id = oc_tile_id * len(ic_tileg) + ic_tile_id

bit_precision = 4
len_kij = 9  # Number of kernel iterations (0-8)

for kij in range(len_kij):
    W = w_tile[tile_id,:,:,kij]

    # Generate filename matching testbench expectation:
 ↪weight_itile{ic}_otile{oc}_kij{kij}.txt
    filename = f'weight_itile{ic_tile_id}_otile{oc_tile_id}_kij{kij}.txt'
    file = open(filename, 'w')

    # Write 3 comment lines (testbench skips first 3 lines)
    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')

    # Write 8 lines (one per column), each with 64 bits (16 rows × 4 bits) for
 ↪2-bit mode
    # For 2-bit: W shape is [8 OC, 16 IC], so we need 16 rows per column
    for i in range(W.size(0)):  # columns (8 output channels)
        for j in range(W.size(1)):  # rows (16 input channels)
            weight_val = round(W[i, 15-j].item())  # Reverse row order: row 15
 ↪to row 0
            if weight_val < 0:
                weight_val = weight_val + (2**bit_precision)  # Convert to
 ↪unsigned
            W_bin = '{0:04b}'.format(weight_val)
            for k in range(bit_precision):
                file.write(W_bin[k])
        file.write('\n')
```

```
    file.close()
    print(f"Written weight data: shape {W.shape}, tile_id={tile_id}, kij={kij},␣
 ↪file={filename}")

print(f"\nGenerated {len_kij} weight files for hardware testbench")
```

```
Written weight data: shape torch.Size([8, 16]), tile_id=0, kij=0,
file=weight_itile0_otile0_kij0.txt
Written weight data: shape torch.Size([8, 16]), tile_id=0, kij=1,
file=weight_itile0_otile0_kij1.txt
Written weight data: shape torch.Size([8, 16]), tile_id=0, kij=2,
file=weight_itile0_otile0_kij2.txt
Written weight data: shape torch.Size([8, 16]), tile_id=0, kij=3,
file=weight_itile0_otile0_kij3.txt
Written weight data: shape torch.Size([8, 16]), tile_id=0, kij=4,
file=weight_itile0_otile0_kij4.txt
Written weight data: shape torch.Size([8, 16]), tile_id=0, kij=5,
file=weight_itile0_otile0_kij5.txt
Written weight data: shape torch.Size([8, 16]), tile_id=0, kij=6,
file=weight_itile0_otile0_kij6.txt
Written weight data: shape torch.Size([8, 16]), tile_id=0, kij=7,
file=weight_itile0_otile0_kij7.txt
Written weight data: shape torch.Size([8, 16]), tile_id=0, kij=8,
file=weight_itile0_otile0_kij8.txt

Generated 9 weight files for hardware testbench
```

[13]:
```
# Generate partial sum data file for hardware testing
ic_tile_id = 0
oc_tile_id = 0
kij = 0
nij = 0
nij_end = min(nij + 64, psum.size(3))
psum_tile = psum[ic_tile_id, oc_tile_id, :, nij:nij_end, kij]

bit_precision = 16
file = open('psum.txt', 'w')
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
    for j in range(psum_tile.size(0)):  # column (PE)
        psum_val = round(psum_tile[7-j, i].item())
        if psum_val < 0:
            psum_val = psum_val + (2**bit_precision)  # Convert to unsigned
        psum_bin = '{0:16b}'.format(psum_val)
```

```
        for k in range(bit_precision):
            file.write(psum_bin[k])
    file.write('\n')
file.close()
print(f"Written psum data: shape {psum_tile.shape}, nij range [{nij}:
 ↪{nij_end}], kij={kij}")
```

```
Written psum data: shape torch.Size([8, 36]), nij range [0:36], kij=0
```

[14]:
```
# Generate expected output file for hardware verification (out.txt)
# This file contains the final output after accumulation and ReLU
# Uses out_recovered_relu from Cell 11 computation

ic_tile_id = 0
oc_tile_id = 0

# Extract output for first output tile (channels 0-7, i.e., first 8 channels)
# out_recovered_relu shape from Cell 11: [16, 4, 4] (16 channels, 4x4 spatial)
# For oc_tile_id=0, we need channels 0-7
out_final = out_recovered_relu[oc_tile_id * 8:(oc_tile_id + 1) * 8, :, :].cpu()␣
 ↪ # Shape: [8, 4, 4]

# Reshape to [8 channels, 16 spatial positions]
o_ni_dim = out_final.size(1)  # 4
out_final_flat = out_final.view(8, -1)  # [8, 16]

bit_precision = 16
file = open('out.txt', 'w')
file.write('#out0col7[msb-lsb],out0col6[msb-lst],....,out0col0[msb-lst]#\n')
file.write('#out1col7[msb-lsb],out1col6[msb-lst],....,out1col0[msb-lst]#\n')
file.write('#................#\n')

# Write 16 lines (one per output spatial position)
# Each line: 128 bits = 8 columns × 16 bits
for i in range(out_final_flat.size(1)):  # 16 output positions
    for j in range(out_final_flat.size(0)):  # 8 columns (output channels)
        out_val = round(out_final_flat[7-j, i].item())
        if out_val < 0:
            out_val = 0  # ReLU already applied, but ensure no negatives
        out_bin = '{0:016b}'.format(out_val & ((2**bit_precision)-1))  # Ensure␣
 ↪16 bits
        file.write(out_bin)
    file.write('\n')
file.close()
print(f"Written output data: shape {out_final_flat.shape}, file=out.txt")
```

```
Written output data: shape torch.Size([8, 16]), file=out.txt
```

[15]:
```
# Generate accumulation address file (acc.txt)
# This file contains memory addresses where partial sums are stored
# Format: 11-bit binary addresses, total of len_onij × len_kij = 16 × 9 = 144␣
 ↪addresses

len_onij = 16  # Number of output spatial positions
len_kij = 9    # Number of kernel iterations

file = open('acc.txt', 'w')

# Generate addresses for each output position and each kij iteration
# Address calculation: kij * len_onij + output_position
for output_pos in range(len_onij):
    for kij in range(len_kij):
        address = kij * len_onij + output_pos
        # Convert to 11-bit binary
        addr_bin = '{0:011b}'.format(address & ((2**11)-1))  # Ensure 11 bits
        file.write(addr_bin + '\n')

file.close()
print(f"Written accumulation address file: {len_onij * len_kij} addresses,␣
 ↪file=acc.txt")
```

```
Written accumulation address file: 144 addresses, file=acc.txt
```

[ ]: