Activation_Sparsity_Analysis

December 14, 2025

[1]:
```
import os, time, math, shutil
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import torchvision, torchvision.transforms as T

from models import *
from models.vgg_quant_part2 import VGG16_quant_part2
from models.quant_layer import QuantConv2d
```

[2]:
```
# ----------------- Config -----------------
DEVICE      = torch.device("cuda" if torch.cuda.is_available() else "cpu")
BATCH_SIZE  = 128
PRINT_FREQ  = 100
EPOCHS      = 200
LR_BASE     = 0.1
WEIGHT_DECAY= 1e-4

# Part 2 spec
NBIT_W      = 4    # weights 4-bit
NBIT_A      = 2    # activations 2-bit
RESULT_DIR  = "result/Part2_VGG_2bitA_4bitW"
os.makedirs(RESULT_DIR, exist_ok=True)

# This is the squeezed 16->16 conv index in model.features
SQUEEZE_IDX = 26
```

[3]:
```
# ----------------- Helpers -----------------
class AverageMeter:
    def __init__(self): self.reset()
    def reset(self):
        self.val = 0; self.avg = 0; self.sum = 0; self.count = 0
    def update(self, v, n=1):
        self.val = v
        self.sum += v * n
        self.count += n
```

```
        self.avg = self.sum / max(self.count, 1)

def accuracy(output, target, topk=(1,)):
    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].reshape(-1).float().sum(0)
        res.append(correct_k.mul_(100.0 / batch_size))
    return res

def save_ckpt(state, is_best, fdir, tag):
    p = os.path.join(fdir, f"ckpt_{tag}.pth")
    torch.save(state, p)
    if is_best:
        shutil.copyfile(p, os.path.join(fdir, f"best_{tag}.pth"))

MILESTONES = (60, 120, 150)
GAMMA_LR   = 0.1
def set_lr_epoch(optim, base_lr, epoch):
    drops = sum(m <= epoch for m in MILESTONES)
    lr = base_lr * (GAMMA_LR ** drops)
    for g in optim.param_groups:
        g["lr"] = lr
    return lr
```

[4]:
```
# ----------------- Data -----------------
normalize = T.Normalize(mean=[0.491,0.482,0.447],
                        std=[0.247,0.243,0.262])
train_tf = T.Compose([
    T.RandomCrop(32, padding=4),
    T.RandomHorizontalFlip(),
    T.ToTensor(),
    normalize,
])
val_tf = T.Compose([
    T.ToTensor(),
    normalize,
])

trainset = torchvision.datasets.CIFAR10(
    root='./data', train=True, download=True, transform=train_tf)
trainloader = torch.utils.data.DataLoader(
```

```
    trainset, batch_size=BATCH_SIZE, shuffle=True, num_workers=2,␣
 ↪pin_memory=True)

testset = torchvision.datasets.CIFAR10(
    root='./data', train=False, download=True, transform=val_tf)
testloader = torch.utils.data.DataLoader(
    testset, batch_size=BATCH_SIZE, shuffle=False, num_workers=2,␣
 ↪pin_memory=True)
```

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

[5]:
```
# ----------------- Model utils -----------------
def set_bitwidth(model, nbit_w:int, nbit_a:int):
    """
    Set bitwidth on all QuantConv2d layers.
    Works with templates that have attributes:
      - nbit_w, nbit_a, or
      - weight_quant.bitwidth / act_quant.bitwidth
    """
    for m in model.modules():
        if isinstance(m, QuantConv2d):
            if hasattr(m, 'nbit_w'): m.nbit_w = nbit_w
            if hasattr(m, 'nbit_a'): m.nbit_a = nbit_a
            if hasattr(m, 'weight_quant') and hasattr(m.weight_quant,␣
 ↪'bitwidth'):
                m.weight_quant.bitwidth = nbit_w
            if hasattr(m, 'act_quant') and hasattr(m.act_quant, 'bitwidth'):
                m.act_quant.bitwidth = nbit_a
    return model

def build_vgg_part2():
    # We already modified vgg_quant so one conv layer has 16 in/out channels
    # and the BN after that layer is removed.
    model = VGG16_quant_part2()
    model = set_bitwidth(model, NBIT_W, NBIT_A)
    return model


# ----------------- Train / Validate -----------------
def train_one_epoch(loader, model, criterion, optimizer, epoch):
    model.train()
    losses = AverageMeter()
    top1  = AverageMeter()
    st = time.time()
    for i, (x, y) in enumerate(loader):
        x = x.to(DEVICE); y = y.to(DEVICE)
```

```
        out = model(x)
        loss = criterion(out, y)

        prec1 = accuracy(out, y)[0]
        losses.update(loss.item(), x.size(0))
        top1.update(prec1.item(), x.size(0))

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        if i % PRINT_FREQ == 0:
            print(f"Epoch[{epoch}] Iter[{i}/{len(loader)}] "
                  f"Loss {losses.val:.4f}({losses.avg:.4f})  "
                  f"Acc {top1.val:.2f}%({top1.avg:.2f}%)")

def validate(loader, model, criterion):
    model.eval()
    losses = AverageMeter()
    top1  = AverageMeter()
    with torch.no_grad():
        for i, (x, y) in enumerate(loader):
            x = x.to(DEVICE); y = y.to(DEVICE)
            out = model(x)
            loss = criterion(out, y)

            prec1 = accuracy(out, y)[0]
            losses.update(loss.item(), x.size(0))
            top1.update(prec1.item(), x.size(0))

            if i % PRINT_FREQ == 0:
                print(f"Val Iter[{i}/{len(loader)}] "
                      f"Loss {losses.val:.4f}({losses.avg:.4f})  "
                      f"Acc {top1.val:.2f}%({top1.avg:.2f}%)")
    print(f"* Acc {top1.avg:.2f}%")
    return top1.avg
```

0.0.1 Report Results

[7]:
```
# ============================================================
# Software Alpha: Activation Sparsity → MAC Skipping Potential
# Assumes:
#  - DEVICE, testloader defined
#  - validate() defined
#  - VGG16_quant_part2 available
# ============================================================
```

```python
import torch
import torch.nn as nn

# -------- Load trained 2A/4W model --------
CKPT = "result/Part2_VGG_2bitA_4bitW/best_vgg_2A4W.pth"

model = VGG16_quant_part2().to(DEVICE)
ckpt = torch.load(CKPT, map_location=DEVICE)
model.load_state_dict(ckpt["state_dict"], strict=True)
model.eval()

# -------- Hook ReLU outputs --------
relu_stats = []

def relu_hook(m, inp, out):
    z = (out == 0).float().mean().item()
    relu_stats.append(z)

hooks = []
for m in model.modules():
    if isinstance(m, nn.ReLU):
        hooks.append(m.register_forward_hook(relu_hook))

# -------- Run a few validation batches --------
NUM_BATCHES = 10
with torch.no_grad():
    for i, (x, _) in enumerate(testloader):
        if i >= NUM_BATCHES:
            break
        x = x.to(DEVICE)
        _ = model(x)

for h in hooks:
    h.remove()

# -------- Report --------
avg_zero = sum(relu_stats) / len(relu_stats)
max_zero = max(relu_stats)
min_zero = min(relu_stats)

print("\n===== SOFTWARE ALPHA: ACTIVATION SPARSITY =====")
print(f"Evaluated {len(relu_stats)} ReLU layers over {NUM_BATCHES} batches")
print(f"Average zero activation ratio : {avg_zero*100:.2f}%")
print(f"Min zero activation ratio     : {min_zero*100:.2f}%")
print(f"Max zero activation ratio     : {max_zero*100:.2f}%")

# -------- MAC skipping estimate --------
```

```
print("\nEstimated MAC savings:")
print(f"~{avg_zero*100:.1f}% MACs can be skipped via zero-activation gating")
print("================================================\n")
```

```
===== SOFTWARE ALPHA: ACTIVATION SPARSITY =====
Evaluated 130 ReLU layers over 10 batches
Average zero activation ratio : 68.32%
Min zero activation ratio     : 48.48%
Max zero activation ratio     : 98.79%

Estimated MAC savings:
~68.3% MACs can be skipped via zero-activation gating
================================================
```

[ ]: