VGG16_Quantization_Aware_Training

December 14, 2025

[8]:
```
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
```

[ ]:
```
# ----------------- Train Part 2 model ------------------
model = build_vgg_part2().to(DEVICE)
criterion = nn.CrossEntropyLoss().to(DEVICE)
optimizer = optim.SGD(model.parameters(), lr=LR_BASE,
                      momentum=0.9, weight_decay=WEIGHT_DECAY)

resume_path = os.path.join(RESULT_DIR, "best_vgg_2A4W.pth")
start_epoch = 0
best_acc = 0.0
```

```
if os.path.isfile(resume_path):
    ckpt = torch.load(resume_path, map_location=DEVICE)
    model.load_state_dict(ckpt["state_dict"], strict=True)
    if "optimizer" in ckpt:
        optimizer.load_state_dict(ckpt["optimizer"])
    best_acc = float(ckpt.get("best", 0.0))
    start_epoch = int(ckpt.get("epoch", 0)) + 1
    print(f"Resumed best@{best_acc:.2f}% from epoch {start_epoch-1}")

for epoch in range(start_epoch, EPOCHS):
    lr = set_lr_epoch(optimizer, LR_BASE, epoch)
    print(f"\n=== Epoch {epoch}  LR={lr:.5f} ===")
    train_one_epoch(trainloader, model, criterion, optimizer, epoch)
    acc = validate(testloader, model, criterion)
    is_best = acc > best_acc
    best_acc = max(best_acc, float(acc))
    save_ckpt({
        "epoch": epoch,
        "state_dict": model.state_dict(),
        "best": best_acc,
        "optimizer": optimizer.state_dict()
    }, is_best, RESULT_DIR, tag="vgg_2A4W")

print(f"Best Part2 VGG Acc (2A/4W): {best_acc:.2f}%")
```

```
[ ]: # ----------------- psum_recovered for squeezed 16x16 layer -----------------
     # We now:
     # 1) Grab input to squeezed conv (pre-hook)
     # 2) Grab input to NEXT layer (pre-hook) => output after conv+ReLU
     # 3) Reconstruct psum_recovered from integer conv and compare.

     def try_get_attr(obj, names):
         for n in names:
             if hasattr(obj, n):
                 return getattr(obj, n)
         return None

     def qparams(alpha: torch.Tensor, nbit: int, signed: bool):
         if signed:
             qmax = (2**(nbit-1)) - 1
             qmin = -(2**(nbit-1))
         else:
             qmax = (2**nbit) - 1
             qmin = 0
         delta = alpha / qmax
         return qmin, qmax, delta
```

```
def to_broadcast(alpha: torch.Tensor, w: torch.Tensor):
    # alpha scalar or [outC]
    if alpha.dim() == 0:
        return alpha
    if alpha.dim() == 1 and alpha.numel() == w.size(0):
        return alpha.view(-1,1,1,1)
    return alpha.max()  # fallback scalar


# Load best ckpt for analysis
if os.path.isfile(resume_path):
    ckpt = torch.load(resume_path, map_location=DEVICE)
    model.load_state_dict(ckpt["state_dict"], strict=True)
model.eval()

features = getattr(model, "features", None)
assert isinstance(features, nn.Sequential), "Expected model.features to be nn.
 ↪Sequential"

# --- auto-find the 16x16 squeezed QuantConv2d ---
squeeze_idx = None
for i, m in enumerate(features):
    if isinstance(m, QuantConv2d) and getattr(m, "in_channels", None) == 16 and␣
 ↪getattr(m, "out_channels", None) == 16:
        squeeze_idx = i
        break

assert squeeze_idx is not None, "Could not find 16x16 QuantConv2d"
squeezed_layer = features[squeeze_idx]

# "next layer": first module AFTER this block (after conv+ReLU)
# skip BN / ReLU / Identity and hook the next real layer (conv or pool)
next_idx = squeeze_idx + 1
while isinstance(features[next_idx], (nn.BatchNorm2d, nn.ReLU, nn.Identity)):
    next_idx += 1
next_layer = features[next_idx]

_cached = {}

def pre_squeezed_hook(m, inp):
    _cached["x_in"] = inp[0].detach().to(DEVICE)

def pre_next_hook(m, inp):
    _cached["x_next_in"] = inp[0].detach().to(DEVICE)

h1 = squeezed_layer.register_forward_pre_hook(pre_squeezed_hook)
h2 = next_layer.register_forward_pre_hook(pre_next_hook)
```

```
with torch.no_grad():
    xb, _ = next(iter(testloader))
    xb = xb.to(DEVICE)
    _ = model(xb)

h1.remove(); h2.remove()
x_in     = _cached["x_in"]        # input to 16x16 conv
x_next   = _cached["x_next_in"]   # input to next layer (after conv+ReLU)

# Get quant parameters from squeezed_layer (QuantConv2d)
assert isinstance(squeezed_layer, QuantConv2d), "Squeezed layer is not␣
 ↪QuantConv2d"

with torch.no_grad():
    # Weights (quantized or float)
    w_q = try_get_attr(squeezed_layer, ["weight_q"])
    if w_q is None:
        w_float = squeezed_layer.weight.detach()
        w_alpha_fb = w_float.abs().max()
        qmin_w, qmax_w, delta_w = qparams(w_alpha_fb, NBIT_W, signed=True)
        w_int_tmp = torch.clamp(torch.round(w_float / delta_w), qmin_w, qmax_w)
        w_q = (w_int_tmp * delta_w).to(w_float.dtype)
    else:
        w_q = w_q.detach()

    # Weight alpha
    w_alpha = None
    wq_mod = try_get_attr(squeezed_layer, ["weight_quant"])
    if wq_mod is not None:
        w_alpha = try_get_attr(wq_mod, ["alpha", "scale", "s", "delta", "a"])
        if isinstance(w_alpha, (float, int)):
            w_alpha = torch.tensor(w_alpha, device=DEVICE, dtype=w_q.dtype)
        if w_alpha is not None:
            w_alpha = w_alpha.detach()
    if w_alpha is None:
        w_alpha = w_q.abs().max()
    w_alpha_b = to_broadcast(w_alpha, w_q)

    # Activation alpha
    aq_mod = try_get_attr(squeezed_layer, ["act_quant"])
    x_signed = bool(try_get_attr(aq_mod, ["signed"])) if aq_mod is not None␣
 ↪else False
    x_alpha = try_get_attr(aq_mod, ["alpha", "scale", "s", "delta", "a"])
    if isinstance(x_alpha, (float, int)):
        x_alpha = torch.tensor(x_alpha, device=DEVICE, dtype=x_in.dtype)
    if x_alpha is None:
        # post-ReLU activations are >=0 → unsigned
        x_alpha = x_in.detach().max()
    x_alpha = x_alpha.to(DEVICE)

    # qparams for weights / activations
    qmin_w, qmax_w, delta_w = qparams(w_alpha_b, NBIT_W, signed=True)
    qmin_x, qmax_x, delta_x = qparams(x_alpha,   NBIT_A, signed=x_signed)

    # Quantize x to integers
    x_int = torch.clamp(torch.round(x_in / delta_x), qmin_x, qmax_x).to(torch.
↪int32)
    x_q   = x_int.float() * delta_x

    # Quantize weights to integers (based on w_q and delta_w)
    w_int = torch.round(w_q / delta_w).to(torch.int32)

    # Integer conv (no bias), then scale back, add bias, ReLU
    stride   = squeezed_layer.stride
    padding  = squeezed_layer.padding
    groups   = squeezed_layer.groups
    bias     = squeezed_layer.bias

    psum_int = F.conv2d(x_int.float(), w_int.float(), bias=None,
                        stride=stride, padding=padding, groups=groups)
    psum_fp  = psum_int * (delta_x * delta_w)  # broadcast if per-channel

    if bias is not None:
        psum_fp = psum_fp + bias.view(1, -1, 1, 1)

    psum_relu = torch.clamp(psum_fp, min=0.0)  # ReLU

# Reference: conv(x_q, w_q) with same stride/padding, then ReLU
with torch.no_grad():
    conv_ref = nn.Conv2d(
        in_channels=w_q.size(1),
        out_channels=w_q.size(0),
        kernel_size=w_q.shape[2:],
        stride=stride,
        padding=padding,
        bias=(bias is not None)
    ).to(DEVICE)

    conv_ref.weight = nn.Parameter(w_q.clone())
    if bias is not None:
        conv_ref.bias = nn.Parameter(bias.clone())

    y_ref = conv_ref(x_q)              # float conv on quantized activations
    y_ref_relu = torch.clamp(y_ref, min=0.0)

    mse = (psum_relu - y_ref_relu).pow(2).mean().item()
    max_abs = (psum_relu - y_ref_relu).abs().max().item()

print(f"[Part2 psum_recovered] MSE vs conv_ref: {mse:.6e}")
print(f"[Part2 psum_recovered] Max abs error  : {max_abs:.6e}")
```

0.0.1 Report Results

[7]:
```
import os
import torch
import torch.nn as nn
import torch.nn.functional as F

def _try_get_attr(obj, names):
    for n in names:
        if hasattr(obj, n):
            return getattr(obj, n)
    return None

def _qparams(alpha: torch.Tensor, nbit: int, signed: bool):
    if signed:
        qmax = (2**(nbit-1)) - 1
        qmin = -(2**(nbit-1))
    else:
        qmax = (2**nbit) - 1
        qmin = 0
    delta = alpha / qmax
    return qmin, qmax, delta

def _to_broadcast(alpha: torch.Tensor, w: torch.Tensor):
    if alpha.dim() == 0:
        return alpha
    if alpha.dim() == 1 and alpha.numel() == w.size(0):
        return alpha.view(-1,1,1,1)
    return alpha.max()

def _set_bitwidth(model, nbit_w:int, nbit_a:int):
    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 compute_quant_error(model, testloader, nbit_a, nbit_w):
    model.eval()
    features = model.features
    assert isinstance(features, nn.Sequential)

    # find squeezed 16x16 QuantConv2d
    squeeze_idx = None
    for i, m in enumerate(features):
        if isinstance(m, QuantConv2d) and getattr(m, "in_channels", None) == 16␣
 ↪and getattr(m, "out_channels", None) == 16:
            squeeze_idx = i
            break
    assert squeeze_idx is not None, "Could not find 16x16 QuantConv2d"
    squeezed_layer = features[squeeze_idx]

    # hook input to that layer
    _cached = {}
    def pre_hook(m, inp):
        _cached["x_in"] = inp[0].detach().to(DEVICE)

    h = squeezed_layer.register_forward_pre_hook(pre_hook)
    with torch.no_grad():
        xb, _ = next(iter(testloader))
        xb = xb.to(DEVICE)
        _ = model(xb)
    h.remove()
    x_in = _cached["x_in"]

    with torch.no_grad():
        # weights (quantized or derive)
        w_q = _try_get_attr(squeezed_layer, ["weight_q"])
        if w_q is None:
            w_float = squeezed_layer.weight.detach()
            w_alpha_fb = w_float.abs().max()
            qmin_w, qmax_w, delta_w = _qparams(w_alpha_fb, nbit_w, signed=True)
            w_int_tmp = torch.clamp(torch.round(w_float / delta_w), qmin_w,␣
 ↪qmax_w)
            w_q = (w_int_tmp * delta_w).to(w_float.dtype)
        else:
            w_q = w_q.detach()

        # weight alpha
        w_alpha = None
        wq_mod = _try_get_attr(squeezed_layer, ["weight_quant"])
        if wq_mod is not None:
            w_alpha = _try_get_attr(wq_mod, ["alpha", "scale", "s", "delta",␣
↪"a"])
            if isinstance(w_alpha, (float, int)):
                w_alpha = torch.tensor(w_alpha, device=DEVICE, dtype=w_q.dtype)
            if w_alpha is not None:
                w_alpha = w_alpha.detach()
        if w_alpha is None:
            w_alpha = w_q.abs().max()
        w_alpha_b = _to_broadcast(w_alpha, w_q)

        # activation alpha
        aq_mod = _try_get_attr(squeezed_layer, ["act_quant"])
        x_signed = bool(_try_get_attr(aq_mod, ["signed"])) if aq_mod is not␣
↪None else False
        x_alpha = _try_get_attr(aq_mod, ["alpha", "scale", "s", "delta", "a"])
        if isinstance(x_alpha, (float, int)):
            x_alpha = torch.tensor(x_alpha, device=DEVICE, dtype=x_in.dtype)
        if x_alpha is None:
            x_alpha = x_in.detach().max()
        x_alpha = x_alpha.to(DEVICE)

        # qparams
        qmin_w, qmax_w, delta_w = _qparams(w_alpha_b, nbit_w, signed=True)
        qmin_x, qmax_x, delta_x = _qparams(x_alpha,   nbit_a, signed=x_signed)

        # quantize
        x_int = torch.clamp(torch.round(x_in / delta_x), qmin_x, qmax_x).
↪to(torch.int32)
        x_q   = x_int.float() * delta_x
        w_int = torch.round(w_q / delta_w).to(torch.int32)

        stride   = squeezed_layer.stride
        padding  = squeezed_layer.padding
        groups   = squeezed_layer.groups
        bias     = squeezed_layer.bias

        # recovered path
        psum_int = F.conv2d(x_int.float(), w_int.float(), bias=None,
                            stride=stride, padding=padding, groups=groups)
        psum_fp  = psum_int * (delta_x * delta_w)
        if bias is not None:
            psum_fp = psum_fp + bias.view(1, -1, 1, 1)
        psum_relu = torch.clamp(psum_fp, min=0.0)

        # reference path: float conv on quantized tensors
        conv_ref = nn.Conv2d(
```

```python
            in_channels=w_q.size(1),
            out_channels=w_q.size(0),
            kernel_size=w_q.shape[2:],
            stride=stride,
            padding=padding,
            bias=(bias is not None)
        ).to(DEVICE)
        conv_ref.weight = nn.Parameter(w_q.clone())
        if bias is not None:
            conv_ref.bias = nn.Parameter(bias.clone())

        y_ref = conv_ref(x_q)
        y_ref_relu = torch.clamp(y_ref, min=0.0)

        mse = (psum_relu - y_ref_relu).pow(2).mean().item()
        max_abs = (psum_relu - y_ref_relu).abs().max().item()

    return mse, max_abs

def eval_ckpt(ckpt_path, nbit_a, nbit_w):
    model = VGG16_quant_part2().to(DEVICE)
    model = _set_bitwidth(model, nbit_w, nbit_a)
    ckpt = torch.load(ckpt_path, map_location=DEVICE)
    model.load_state_dict(ckpt["state_dict"], strict=True)
    crit = nn.CrossEntropyLoss().to(DEVICE)
    acc = validate(testloader, model, crit)
    mse, max_abs = compute_quant_error(model, testloader, nbit_a, nbit_w)
    return acc, mse, max_abs

CKPT_2A4W = "result/Part2_VGG_2bitA_4bitW/best_vgg_2A4W.pth"

print("\n===== FINAL REPORT =====")
if os.path.isfile(CKPT_2A4W):
    acc, mse, mx = eval_ckpt(CKPT_2A4W, nbit_a=2, nbit_w=4)
    print(f"VGG16 2A/4W  Acc: {acc:.2f}%   QuantErr(MSE): {mse:.3e}   MaxAbs:␣
  ↪{mx:.3e}")
else:
    print("VGG16 2A/4W  (checkpoint not found, skipped)")
print("===== DONE =====\n")
```

```
===== FINAL REPORT =====
Val Iter[0/79] Loss 0.2193(0.2193)  Acc 92.97%(92.97%)
* Acc 90.27%
VGG16 2A/4W  Acc: 90.27%   QuantErr(MSE): 7.330e-13   MaxAbs: 3.052e-05
===== DONE =====
```

[ ]:
```

```