VGG16 with Gradual Channel Squeeze

Motivation: Avoiding Sudden Squeezing

The original VGG16_quant model has a **sudden squeeze** in the bottleneck:

- `512 → 8 → 512` channels

This abrupt reduction can cause:

1. **Information loss** - too much compression in a single step
2. **Gradient instability** - large parameter changes needed
3. **Harder quantization** - fewer channels means each must carry more information

Solution: Gradual Squeeze Layers

The `VGG16_quant_gradual` model adds intermediate layers for a smoother transition:

- `512 → 128 → 32 → 8 → 32 → 128 → 512` channels

Benefits:

1. **Smoother information compression** - gradual reduction allows the network to learn better representations
2. **Better gradient flow** - smaller changes at each layer
3. **Improved quantization** - each layer has a more manageable task

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

from models import *


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

```
batch_size = 128
# Using the gradual squeeze model to avoid sudden channel reduction
# Original: 512 -> 8 -> 512 (sudden squeeze)
# Gradual:  512 -> 128 -> 32 -> 8 -> 32 -> 128 -> 512 (smooth transition)
model_name = "VGG16_quant_gradual"
model = VGG16_quant_gradual()

print(model)

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


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


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, 


print_freq = 100 # every 100 batches, accuracy printed. Here, each batch inclu
# 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 ou
                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
    adjust_list = [80, 120]
    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_gradual(
  (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_s
tats=True)
    (2): ReLU(inplace=True)
    (3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=F
alse)
    (4): QuantConv2d(
      64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
      (weight_quant): weight_quantize_fn()
    )
    (5): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_
stats=True)
    (6): ReLU(inplace=True)
    (7): QuantConv2d(
      128, 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): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=
False)
    (11): QuantConv2d(
      128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
      (weight_quant): weight_quantize_fn()
    )
    (12): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running
_stats=True)
    (13): ReLU(inplace=True)
    (14): QuantConv2d(
      256, 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): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=
False)
    (21): QuantConv2d(
      256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
      (weight_quant): weight_quantize_fn()
    )
    (22): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running
_stats=True)
    (23): ReLU(inplace=True)
    (24): QuantConv2d(
      512, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
```

```
      (weight_quant): weight_quantize_fn()
    )
    (25): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running
_stats=True)
    (26): ReLU(inplace=True)
    (27): QuantConv2d(
      128, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
      (weight_quant): weight_quantize_fn()
    )
    (28): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_
stats=True)
    (29): ReLU(inplace=True)
    (30): QuantConv2d(
      32, 8, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
      (weight_quant): weight_quantize_fn()
    )
    (31): BatchNorm2d(8, eps=1e-05, momentum=0.1, affine=True, track_running_s
tats=True)
    (32): ReLU(inplace=True)
    (33): QuantConv2d(
      8, 8, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
      (weight_quant): weight_quantize_fn()
    )
    (34): ReLU(inplace=True)
    (35): QuantConv2d(
      8, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
      (weight_quant): weight_quantize_fn()
    )
    (36): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_
stats=True)
    (37): ReLU(inplace=True)
    (38): QuantConv2d(
      32, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
      (weight_quant): weight_quantize_fn()
    )
    (39): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running
_stats=True)
    (40): ReLU(inplace=True)
    (41): QuantConv2d(
      128, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
      (weight_quant): weight_quantize_fn()
    )
    (42): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running
_stats=True)
    (43): ReLU(inplace=True)
    (44): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=
False)
    (45): QuantConv2d(
      512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
      (weight_quant): weight_quantize_fn()
    )
    (46): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running
_stats=True)
    (47): ReLU(inplace=True)
    (48): QuantConv2d(
      512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
      (weight_quant): weight_quantize_fn()
    )
    (49): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running
_stats=True)
```

```
    (50): ReLU(inplace=True)
    (51): QuantConv2d(
      512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
      (weight_quant): weight_quantize_fn()
    )
    (52): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running
_stats=True)
    (53): ReLU(inplace=True)
    (54): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=
False)
    (55): 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 [ ]:

```python
PATH = "result/VGG16_quant_gradual/model_best.pth.tar"
if os.path.exists(PATH):
    checkpoint = torch.load(PATH)
    model.load_state_dict(checkpoint['state_dict'])
else:
    print(f"No checkpoint found at {PATH}. Will train from scratch.")

# This cell won't be given, but students will complete the training

lr = 0.01
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_de
#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(30, 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 [2]:

```
PATH = "result/VGG16_quant_gradual/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: 9044/10000 (90%)
```

In [21]:

```
# 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[33]  # Current conv layer
target_layer_next = model.features[35]  # 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, 8, 4, 4])
Captured next layer input shape (features[29]): torch.Size([128, 8, 4, 4])
```

In [22]:
```
# Convert quantized weights to integers
w_bit = 4
target_conv = model.features[33]
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)
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: 2.2114, Weight delta: 0.3159
Weight int shape: torch.Size([8, 8, 3, 3])
Weight int sample (first few values): tensor([-5, -7,  3, -4, -3,  0,  0, -4,
-1, -4], device='cuda:0',
       dtype=torch.int32)
```

In [23]:
```
# Convert quantized activations to integers
x_bit = 4
target_conv = model.features[33]
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: 5.7510, Activation delta: 0.3834
x_int shape: torch.Size([128, 8, 4, 4])
x_int sample (first few values): tensor([0, 0, 0, 0, 1, 0, 0, 0, 0, 0], device
='cuda:0', dtype=torch.int32)
```

In [24]:
```
# Perform integer convolution and recover floating-point output
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(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, 8, 4, 4])
Output recovered sample: tensor([[-2.7858,  3.5126,  3.3914,  1.0901],
        [-2.0591,  2.5436,  3.1492,  2.6647],
        [-4.6027, -4.3604, -3.0281, -0.8479],
        [-1.5746, -2.7858, -1.0901, -0.1211]], device='cuda:0',
       grad_fn=<SliceBackward0>)
```

In [25]:
```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():.10f}")
print(f"Difference max: {difference.max():.10f}")
print(f"Difference should be < 10^-3: {difference.mean() < 1e-3}")
```

```
Difference mean: 0.0000007252
Difference max: 0.0000152588
Difference should be < 10^-3: True
```

In [20]:
```python
difference = abs( output_ref - output_recovered )
print(difference.mean())  ## It should be small, e.g.,2.3 in my trainned model
```

```
tensor(3.3950, device='cuda:0', grad_fn=<MeanBackward0>)
```

In [ ]:
```python

```