[HW4_prob1]_VGG16_Post_Training_Quantization

October 24, 2021

[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
model_name = "VGG16"
model = VGG16()
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),
```

```python
        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))
```

```python
        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
```

```python
            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
```

```python
        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 resnet, the lr starts from 0.1, and is divided by 10 at 80 and 120␣
 ↪epochs"""
    adjust_list = [150, 225]
    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(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
    (1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (2): ReLU(inplace=True)
    (3): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
    (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): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
    (8): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (9): ReLU(inplace=True)
    (10): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
    (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): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
    (15): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (16): ReLU(inplace=True)
    (17): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
    (18): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (19): ReLU(inplace=True)
    (20): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
    (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): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
    (25): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (26): ReLU(inplace=True)
    (27): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
    (28): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (29): ReLU(inplace=True)
    (30): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
    (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): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
    (35): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (36): ReLU(inplace=True)
    (37): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
    (38): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (39): ReLU(inplace=True)
```

```
    (40): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
    (41): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
    (42): ReLU(inplace=True)
    (43): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
    (44): 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
```

[ ]:

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

lr = 4e-2
weight_decay = 1e-4
epochs = 300
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)
```

[ ]:
```
# HW

#  1. Load your saved VGG16 model
#  2. Replace your model's all the Conv's weight with quantized weight
#  3. Apply reasonable alpha
#  4. Then, try to 4,8 bits and measure accuracy
```

[ ]:
```
PATH = "result/VGG16/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)))
```

[2]:
```
def act_quantization(b):

    def uniform_quant(x, b=3):
        xdiv = x.mul(2 ** b - 1)
        xhard = xdiv.round().div(2 ** b - 1)
        return xhard
```

```python
    class uq(torch.autograd.Function):   # here single underscore means this␣
↪class is for internal use

        def forward(ctx, input, alpha):
            input_d = input/alpha
            input_c = input_d.clamp(max=1)  # Mingu edited for Alexnet
            input_q = uniform_quant(input_c, b)
            ctx.save_for_backward(input, input_q)
            input_q_out = input_q.mul(alpha)
            return input_q_out

    return uq().apply




def weight_quantization(b):

    def uniform_quant(x, b):
        xdiv = x.mul((2 ** b - 1))
        xhard = xdiv.round().div(2 ** b - 1)
        return xhard

    class uq(torch.autograd.Function):

        def forward(ctx, input, alpha):
            input_d = input/alpha                          # weights are first␣
↪divided by alpha
            input_c = input_d.clamp(min=-1, max=1)       # then clipped to␣
↪[-1,1]
            sign = input_c.sign()
            input_abs = input_c.abs()
            input_q = uniform_quant(input_abs, b).mul(sign)
            ctx.save_for_backward(input, input_q)
            input_q_out = input_q.mul(alpha)               # rescale to the␣
↪original range
            return input_q_out

    return uq().apply



class weight_quantize_fn(nn.Module):
    def __init__(self, w_bit):
        super(weight_quantize_fn, self).__init__()
        self.w_bit = w_bit-1
        self.weight_q = weight_quantization(b=self.w_bit)
```

```
        self.wgt_alpha = 0.0

    def forward(self, weight):
        weight_q = self.weight_q(weight, self.wgt_alpha)

        return weight_q
```

[3]:
```
w_alpha = 2.0  # cliping value
w_bits = 8

x_alpha = 4.0  # clipping value
x_bits = 8
```

[4]:
```
PATH = "result/VGG16/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)))

for w_alpha in torch.range(0, 2, 0.1):
    PATH = "result/VGG16/model_best.pth.tar"
    checkpoint = torch.load(PATH)
    model.load_state_dict(checkpoint['state_dict'])
    device = torch.device("cuda")

    print("\n")
    print(w_alpha)
    weight_quant = weight_quantize_fn(w_bit= w_bits)  ## define quant function
    weight_quant.wgt_alpha = torch.tensor(w_alpha)
```

```
for layer in model.modules():
    if isinstance(layer, torch.nn.Conv2d):
        layer.weight = torch.nn.Parameter(weight_quant(layer.weight))

criterion = nn.CrossEntropyLoss().cuda()
model.eval()
model.cuda()
prec = validate(testloader, model, criterion)
```

```
/opt/conda/lib/python3.9/site-packages/torch/nn/functional.py:718: UserWarning:
Named tensors and all their associated APIs are an experimental feature and
subject to change. Please do not use them for anything important until they are
released as stable. (Triggered internally at
/pytorch/c10/core/TensorImpl.h:1156.)
  return torch.max_pool2d(input, kernel_size, stride, padding, dilation,
ceil_mode)


Test set: Accuracy: 9074/10000 (91%)




tensor(0.)
```

```
/tmp/ipykernel_12388/1543001428.py:25: UserWarning: torch.range is deprecated
and will be removed in a future release because its behavior is inconsistent
with Python's range builtin. Instead, use torch.arange, which produces values in
[start, end).
  for w_alpha in torch.range(0, 2, 0.1):
/tmp/ipykernel_12388/1543001428.py:34: UserWarning: To copy construct from a
tensor, it is recommended to use sourceTensor.clone().detach() or
sourceTensor.clone().detach().requires_grad_(True), rather than
torch.tensor(sourceTensor).
  weight_quant.wgt_alpha = torch.tensor(w_alpha)
```

```
Test: [0/79]    Time 0.597 (0.597)      Loss 2.5045 (2.5045)    Prec 11.719%
(11.719%)
 * Prec 10.000%



tensor(0.1000)
Test: [0/79]    Time 1.011 (1.011)      Loss 5.1071 (5.1071)    Prec 14.062%
(14.062%)
 * Prec 12.120%



tensor(0.2000)
Test: [0/79]    Time 0.507 (0.507)      Loss 1.9959 (1.9959)    Prec 53.906%
```

```
(53.906%)
 * Prec 52.630%


tensor(0.3000)
Test: [0/79]    Time 0.728 (0.728)      Loss 0.7508 (0.7508)    Prec 76.562%
(76.562%)
 * Prec 78.980%


tensor(0.4000)
Test: [0/79]    Time 1.033 (1.033)      Loss 0.5189 (0.5189)    Prec 85.156%
(85.156%)
 * Prec 84.550%


tensor(0.5000)
Test: [0/79]    Time 0.528 (0.528)      Loss 0.4364 (0.4364)    Prec 88.281%
(88.281%)
 * Prec 87.060%


tensor(0.6000)
Test: [0/79]    Time 0.678 (0.678)      Loss 0.3966 (0.3966)    Prec 89.062%
(89.062%)
 * Prec 88.840%


tensor(0.7000)
Test: [0/79]    Time 0.569 (0.569)      Loss 0.3700 (0.3700)    Prec 89.062%
(89.062%)
 * Prec 89.790%


tensor(0.8000)
Test: [0/79]    Time 0.484 (0.484)      Loss 0.3479 (0.3479)    Prec 89.844%
(89.844%)
 * Prec 90.310%


tensor(0.9000)
Test: [0/79]    Time 0.890 (0.890)      Loss 0.3226 (0.3226)    Prec 90.625%
(90.625%)
 * Prec 90.530%


tensor(1.)
Test: [0/79]    Time 0.572 (0.572)      Loss 0.3237 (0.3237)    Prec 90.625%
```

```
(90.625%)
 * Prec 90.600%


tensor(1.1000)
Test: [0/79]    Time 0.661 (0.661)      Loss 0.2907 (0.2907)    Prec 90.625%
(90.625%)
 * Prec 90.720%


tensor(1.2000)
Test: [0/79]    Time 0.575 (0.575)      Loss 0.2919 (0.2919)    Prec 90.625%
(90.625%)
 * Prec 90.580%


tensor(1.3000)
Test: [0/79]    Time 0.672 (0.672)      Loss 0.2820 (0.2820)    Prec 91.406%
(91.406%)
 * Prec 90.510%


tensor(1.4000)
Test: [0/79]    Time 0.696 (0.696)      Loss 0.2659 (0.2659)    Prec 92.188%
(92.188%)
 * Prec 90.560%


tensor(1.5000)
Test: [0/79]    Time 0.580 (0.580)      Loss 0.2584 (0.2584)    Prec 92.188%
(92.188%)
 * Prec 90.650%


tensor(1.6000)
Test: [0/79]    Time 0.679 (0.679)      Loss 0.2541 (0.2541)    Prec 92.969%
(92.969%)
 * Prec 90.540%


tensor(1.7000)
Test: [0/79]    Time 0.526 (0.526)      Loss 0.2629 (0.2629)    Prec 91.406%
(91.406%)
 * Prec 90.430%


tensor(1.8000)
Test: [0/79]    Time 0.538 (0.538)      Loss 0.2630 (0.2630)    Prec 92.188%
```

```
(92.188%)
 * Prec 90.240%


tensor(1.9000)
Test: [0/79]    Time 0.540 (0.540)      Loss 0.2779 (0.2779)    Prec 91.406%
(91.406%)
 * Prec 89.580%


tensor(2.)
Test: [0/79]    Time 0.534 (0.534)      Loss 0.3210 (0.3210)    Prec 89.062%
(89.062%)
 * Prec 89.100%
```

[ ]:
```
#We get best accuracy at alpha = 1.5 (90.65% accuracy)
```

[ ]:

[ ]:

[ ]:

[ ]: