Trained_projectVGG16_quant

December 4, 2024

[ ]:
```
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_quant"
model = VGG16_quant()

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 resnet, the lr starts from 0.1, and is divided by 10 at 80 and 120␣
 ↪epochs"""
    adjust_list = [50, 90]
    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)
```

[6]:
```
print(model.features[24])
print(model.features[25])
print(model.features[26])
print(model.features[27])
print(model.features[28])
```

```
QuantConv2d(
  256, 8, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
  (weight_quant): weight_quantize_fn()
)
BatchNorm2d(8, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
ReLU(inplace=True)
QuantConv2d(
  8, 8, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False
  (weight_quant): weight_quantize_fn()
)
ReLU(inplace=True)
```

[ ]:
```
lr = 6e-4
weight_decay = 6e-3
```

```
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

fdir = 'result/'+'VGG16_quant'



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. Train with 4 bits for both weight and activation to achieve >90% accuracy
#  2. Find x_int and w_int for the 2nd convolution layer
#  3. Check the recovered psum has similar value to the un-quantized original␣
 ↪psum
```

[27]:
```
PATH = "result/VGG16_quant/model_best.pth.tar"
checkpoint = torch.load(PATH)
model.load_state_dict(checkpoint['state_dict'])
device = torch.device("cuda" if use_gpu else "cpu")

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: 9072/10000 (91%)
```

[28]:

```
#send an input and grap the value by using prehook like HW3
class SaveOutput:
    def __init__(self):
        self.outputs = []
    def __call__(self, module, module_in):
        self.outputs.append(module_in)
    def clear(self):
        self.outputs = []
######### Save inputs from selected layer ##########
save_output = SaveOutput()
model.features[27].register_forward_pre_hook(save_output)
dataiter = iter(trainloader)
images, labels = next(dataiter)
images = images.to(device)
labels = labels.to(device)
out = model(images)
```

[29]:

```
w_bits = 4
weight_q = model.features[27].weight_q # quantized value is stored during the␣
 ↪training
w_alpha =  model.features[27].weight_quant.wgt_alpha  # alpha is defined in␣
 ↪your model already. bring it out here
w_delta =  w_alpha/(2**(w_bits-1)-1)  # delta can be calculated by using alpha␣
 ↪and w_bit
weight_int = weight_q/w_delta # w_int can be calculated by weight_q and w_delta
```

[30]:

```
x_bits = 4
x = save_output.outputs[0][0]  # input of the 2nd conv layer
x_alpha  = model.features[27].act_alpha.item()
x_delta = x_alpha/(2**(x_bits)-1)

act_quant_fn = act_quantization(x_bits) # define the quantization function
x_q = act_quant_fn(x, x_alpha)         # create the quantized value for x
x_int = x_q/x_delta
```

[31]:

```
conv_int = torch.nn.Conv2d(in_channels = 8, out_channels=8, kernel_size = 3,␣
 ↪stride = 1, padding = 1, bias = False)
conv_int.weight = torch.nn.parameter.Parameter(weight_int)

output_int =  conv_int(x_int)    # output_int can be calculated with conv_int␣
 ↪and x_int
output_recovered = output_int * x_delta * w_delta # recover with x_delta and␣
 ↪w_delta
my_out = model.features[28](output_recovered)
```

[34]:

```
conv_ref = torch.nn.Conv2d(in_channels = 8, out_channels=8, kernel_size = 3,␣
 ↪padding=1, stride = 1)
conv_ref.weight = model.features[27].weight_q
conv_ref.bias = model.features[27].bias
output_ref = conv_ref(x)
my_out2 = model.features[28](output_ref)
diff = ((my_out) - my_out2).abs().mean()
print(diff)
```

```
tensor(0.0096, device='cuda:0', grad_fn=<MeanBackward0>)
```

[39]:

```
#send an input and grap the value by using prehook
######### Save inputs from selected layer ##########
save_out = SaveOutput()
model.features[29].register_forward_pre_hook(save_out)

dataiter = iter(trainloader)
images, labels = next(dataiter)
images = images.to(device)
labels = labels.to(device)
out = model(images)
diff = ((my_out) - save_out.outputs[0][0]).abs().mean()
print(diff)
```

```
tensor(0.1161, device='cuda:0', grad_fn=<MeanBackward0>)
```

[ ]:

```
```