Project Part Alpha Resnet20

November 30, 2024

0.0.1 Part Alpha Resnet20 Quantization Aware Training

[1]:
```python
# Import libraries and load data (CIFAR 10)

# system library
import os
import time
import shutil

# NN library
import torch
import torch.nn as nn

# datasets library
import torchvision
import torchvision.transforms as transforms

# model library
from models import resnet_quant
from models import quant_layer


# data loading
batch_size = 100
num_workers = 2
normalize = transforms.Normalize(mean=[0.491, 0.482, 0.447], std=[0.247, 0.243,␣
↪0.262])


train_data = torchvision.datasets.CIFAR10(
    root='data',
    train=True,
    download=True,
    transform=transforms.Compose([
        transforms.RandomCrop(32, padding=4),
        transforms.RandomHorizontalFlip(),
        transforms.ToTensor(),
        normalize,
```

```
    ]))


test_data = torchvision.datasets.CIFAR10(
    root='data',
    train=False,
    download=True,
    transform=transforms.Compose([
        transforms.ToTensor(),
        normalize,
    ]))


train_loader = torch.utils.data.DataLoader(train_data, batch_size=batch_size,␣
 ↪shuffle=True, num_workers=num_workers)
test_loader = torch.utils.data.DataLoader(test_data, batch_size=batch_size,␣
 ↪shuffle=False, num_workers=num_workers)
```

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

[2]:

```
# Define functions for training, validation etc.
print_freq = 100

def train(train_loader, model, criterion, optimizer, epoch):
    batch_time = AverageMeter()   ## at the begining of each epoch, this should␣
 ↪be reset
    data_time = AverageMeter()
    losses = AverageMeter()
    top1 = AverageMeter()

    # switch to train mode
    model.train()
    end = time.time()

    for i, (x_train, y_train) in enumerate(train_loader):
        # record data loading time
        data_time.update(time.time() - end)

        # compute output and loss
        x_train = x_train.cuda()
        y_train = y_train.cuda()
        output = model(x_train)
        loss = criterion(output, y_train)

        # measure accuracy and record loss
        prec = accuracy(output, y_train)[0]
```

```
        losses.update(loss.item(), x_train.size(0))
        top1.update(prec.item(), x_train.size(0))

        # compute gradient and do SGD step
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        # output epoch time and loss
        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(train_loader), batch_time=batch_time,
                   data_time=data_time, loss=losses, top1=top1))


def validate(test_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, (x_test, y_test) in enumerate(test_loader):
            # compute output
            x_test = x_test.cuda()
            y_test = y_test.cuda()
            output = model(x_test)
            loss = criterion(output, y_test)

            # measure accuracy and record loss
            prec = accuracy(output, y_test)[0]
            losses.update(loss.item(), x_test.size(0))
            top1.update(prec.item(), x_test.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(test_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) # topk(k, dim=None,␣
 ↪largest=True, sorted=True)
                                               # will output (max value, its␣
 ↪index)
    pred = pred.t()                                          # transpose
    correct = pred.eq(target.view(1, -1).expand_as(pred))    # "-1": calculate␣
 ↪automatically

    res = []
    for k in topk:
        correct_k = correct[:k].view(-1).float().sum(0)  # view(-1): make a␣
 ↪flattened 1D tensor
        res.append(correct_k.mul_(100.0 / batch_size))   # correct: size of␣
 ↪[maxk, 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     ## n is impact factor
        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 = [150, 250]
    if epoch in adjust_list:
        for param_group in optimizer.param_groups:
            param_group['lr'] = param_group['lr'] * 0.1
```

[3]:

```
# We replace the batchnorm layer (after the modified conv layer) with an empty␣
 ↪layer
# By implementing this, we don't need to modify resnet20's forward function

class EmptyLayer(nn.Module):
    def __init__(self):
        super(EmptyLayer, self).__init__()

    def forward(self, x):
        return x
```

[4]:

```
# Configure model
model_name = 'project_alpha'
model_alpha = resnet_quant.resnet20_quant()

# Adjust certain layers
model_alpha.layer3[0].conv1 = quant_layer.QuantConv2d(32, 8, kernel_size=3,␣
 ↪padding=1, stride=2)
model_alpha.layer3[0].conv2 = quant_layer.QuantConv2d(8,  8, kernel_size=3,␣
 ↪padding=1)
model_alpha.layer3[0].downsample = nn.Sequential(
                quant_layer.QuantConv2d(32, 8, kernel_size=1, stride=2),
                nn.BatchNorm2d(8)
            )
```

```
model_alpha.layer3[0].bn1 = EmptyLayer()
model_alpha.layer3[0].bn2 = nn.BatchNorm2d(8)

model_alpha.layer3[1].conv1 = quant_layer.QuantConv2d(8, 64, kernel_size=3,␣
 ↪padding=1)
model_alpha.layer3[1].downsample = nn.Sequential(
                quant_layer.QuantConv2d(8, 64, kernel_size=1),
                nn.BatchNorm2d(64)
            )

# parameters for training
lr = 0.001
weight_decay = 1e-4
epochs = 100
best_prec = 0

model_alpha = model_alpha.cuda()
criterion = nn.CrossEntropyLoss().cuda()
optimizer = torch.optim.SGD(model_alpha.parameters(), lr=lr, momentum=0.8,␣
 ↪weight_decay=weight_decay)

# saving path
if not os.path.exists('result'):
    os.makedirs('result')
fdir = 'result/'+str(model_name)
if not os.path.exists(fdir):
    os.makedirs(fdir)
```

[ ]:

```
# Train and validate 4bit resnet20 model
for epoch in range(140, 100+epochs):
    adjust_learning_rate(optimizer, epoch)
    train(train_loader, model_alpha, criterion, optimizer, epoch)

    # evaluate on test set
    print("Validation starts")
    prec = validate(test_loader, model_alpha, 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_alpha.state_dict(),
        'best_prec': best_prec,
        'optimizer': optimizer.state_dict(),
    }, is_best, fdir)
```

[5]:
```
# Validate 4bit res20 model on test dataset
fdir = 'result/'+str(model_name)+'/model_best.pth.tar'
checkpoint = torch.load(fdir)
model_alpha.load_state_dict(checkpoint['state_dict'])

criterion = nn.CrossEntropyLoss().cuda()
model_alpha.eval()
model_alpha.cuda()
prec = validate(test_loader, model_alpha, criterion)
```

```
Test: [0/100]	Time 1.104 (1.104)	Loss 0.2402 (0.2402)	Prec 92.000%
(92.000%)
 * Prec 89.080%
```

[6]:
```
# Prehook
class SaveOutput:
    def __init__(self):
        self.outputs = []
    def __call__(self, module, module_in):
        self.outputs.append(module_in)
    def clear(self):
        self.outputs = []

save_output = SaveOutput()
for layer in model_alpha.layer3.modules():
    if isinstance(layer, quant_layer.QuantConv2d) or isinstance(layer,
 ↪EmptyLayer):
        layer.register_forward_pre_hook(save_output)

dataiter = iter(train_loader)
images, labels = next(dataiter)
images = images.cuda()
out = model_alpha(images)

# print(len(save_output.outputs))
print("model_alpha layer3 block0 conv2 layer's input: ", save_output.
 ↪outputs[1][0].size())
```

```
model_alpha layer3 block0 conv2 layer's input:  torch.Size([100, 8, 8, 8])
```

[7]:
```
# Find x_int and w_int for the 8*8 convolution layer
layer = model_alpha.layer3[0].conv2
x = save_output.outputs[1][0]

w_bits = 4
w_alpha = layer.weight_quant.wgt_alpha
w_delta = w_alpha/(2**(w_bits-1)-1)
```

```
weight_q = layer.weight_q # quantized value is stored during the training
weight_int = weight_q/w_delta

x_bits = 4
x_alpha  = layer.act_alpha
x_delta = x_alpha/(2**x_bits-1)
act_quant_fn = quant_layer.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
```

[8]:
```
# Check the recovered p_sum has similar value to the un-quantized original␣
 ↪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)
output_int = conv_int(x_int)
output_recovered = output_int * w_delta * x_delta
output_ref = model_alpha.layer3[0].conv2(save_output.outputs[1][0])

# calculate the differences
print((output_recovered - output_ref).sum())
```

```
tensor(0.0398, device='cuda:0', grad_fn=<SumBackward0>)
```