

[HW3_prob1]_CNN_Training_with_resnet20

October 25, 2022

```
[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 *    # bring everything in the folder models

global best_prec
use_gpu = torch.cuda.is_available()
print('=> Building model...')
device = torch.device("cuda" if use_gpu else "cpu")

batch_size = 128

model_name = "resnet20_cifar"
model = resnet20_cifar()

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() ## at the begining of each epoch, this should
    ↪be reset
    data_time = AverageMeter()
    losses = AverageMeter()
    top1 = AverageMeter()

    model.train()

    end = time.time() # measure current time

    for i, (input, target) in enumerate(trainloader):
        # measure data loading time
        data_time.update(time.time() - end) # data loading time

        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) # time spent to process one batch
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    ## 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, 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...

Files already downloaded and verified

Files already downloaded and verified

```

[ ]: import matplotlib.pyplot as plt
import numpy as np

# functions to show an image

def imshow(img):
    img = img / 2 + 0.5    # unnormalize
    npimg = img.numpy()
    plt.imshow(np.transpose(npimg, (1, 2, 0)))
    plt.show()

# get some random training images
dataiter = iter(testloader)

```

```
images, labels = dataiter.next() ## If you run this line, the next data batch  
→ is called subsequently.
```

```
# show images  
imshow(torchvision.utils.make_grid(images))
```

```
[ ]: # This cell is from the website
```

```
lr = 4.3e-2  
weight_decay = 1e-4  
epochs = 200  
best_prec = 0  
  
model = model.cuda()  
criterion = nn.CrossEntropyLoss().cuda()  
optimizer = torch.optim.SGD(model.parameters(), lr=lr, momentum=0.9,  
→ weight_decay=weight_decay)  
# weight decay: for regularization to prevent overfitting  
  
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)
```

```
[ ]: fdir = 'result/'+str(model_name)+'/' + 'model_best.pth.tar'
```

```
checkpoint = torch.load(fdir)
model.load_state_dict(checkpoint['state_dict'])
```

```
criterion = nn.CrossEntropyLoss().cuda()
```

```
model.eval()
model.cuda()
```

```
prec = validate(testloader, model, criterion)
```

```
[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
```

```
    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

```

```

[7]: w_alpha = 1.5 # clipping value
     w_bits = 4

     fdir = 'result/'+str(model_name)+'/' + 'model_best.pth.tar'
     checkpoint = torch.load(fdir)
     model.load_state_dict(checkpoint['state_dict'])

     for layer in model.modules():
         if isinstance(layer, torch.nn.Conv2d):
             print(layer.weight.max())
             weight_quant = weight_quantize_fn(w_bit= w_bits)  ## define quant
             →function
             weight_quant.wgt_alpha = torch.tensor(w_alpha)
             w_quant = weight_quant(layer.weight)

```



```

        #print("W_int",w_int)
        layer.weight = torch.nn.Parameter(w_quant)
        #print("Layer.weight",layer.weight)

criterion = nn.CrossEntropyLoss().cuda()
model.eval()
model.cuda()

prec = validate(testloader, model, criterion)

```

```

tensor(2.0048, device='cuda:0', grad_fn=<MaxBackward1>)
tensor(0.8558, device='cuda:0', grad_fn=<MaxBackward1>)
tensor(0.7491, device='cuda:0', grad_fn=<MaxBackward1>)
tensor(1.0326, device='cuda:0', grad_fn=<MaxBackward1>)
tensor(0.6198, device='cuda:0', grad_fn=<MaxBackward1>)
tensor(0.7209, device='cuda:0', grad_fn=<MaxBackward1>)
tensor(0.8455, device='cuda:0', grad_fn=<MaxBackward1>)
tensor(0.8549, device='cuda:0', grad_fn=<MaxBackward1>)
tensor(1.1671, device='cuda:0', grad_fn=<MaxBackward1>)
tensor(1.9485, device='cuda:0', grad_fn=<MaxBackward1>)
tensor(0.6710, device='cuda:0', grad_fn=<MaxBackward1>)
tensor(0.5790, device='cuda:0', grad_fn=<MaxBackward1>)
tensor(0.7601, device='cuda:0', grad_fn=<MaxBackward1>)
tensor(0.6992, device='cuda:0', grad_fn=<MaxBackward1>)
tensor(0.5459, device='cuda:0', grad_fn=<MaxBackward1>)
tensor(0.5653, device='cuda:0', grad_fn=<MaxBackward1>)
tensor(1.1730, device='cuda:0', grad_fn=<MaxBackward1>)
tensor(0.5766, device='cuda:0', grad_fn=<MaxBackward1>)
tensor(0.5345, device='cuda:0', grad_fn=<MaxBackward1>)
tensor(0.5773, device='cuda:0', grad_fn=<MaxBackward1>)
tensor(0.2915, device='cuda:0', grad_fn=<MaxBackward1>)
Test: [0/79]    Time 0.381 (0.381)      Loss 4.0180 (4.0180)      Prec 15.625%
(15.625%)
* Prec 15.430%

```

```
[ ]:
```

```

[ ]: 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.modules():
    if isinstance(layer, torch.nn.Conv2d):
        print("prehooked")
        layer.register_forward_pre_hook(save_output)

dataiter = iter(trainloader)
images, labels = dataiter.next()
images = images.to(device)
out = model(images)

```

```

[ ]: for layer in model.modules():
      print(layer)

```

```

[ ]: save_output.outputs[1][0]

```

```

[ ]: my_input = save_output.outputs[1][0]
      images.size()

```

```

[ ]: #Prehooked BasicBlock0 to BasicBlock1
      co = model.layer1
      conv_1 = model.layer1[0].conv1
      conv_2 = model.layer1[0].conv2
      bn_1 = model.layer1[0].bn1
      Rel = model.layer1[0].relu
      bn_2 = model.layer1[0].bn2
      co

```

```

[ ]: res = my_input
      my_output = Rel((bn_2(conv_2(Rel(bn_1(conv_1(my_input)))))))+res)

```

```

[ ]: (my_output - save_output.outputs[3][0]).sum()

```

```

[ ]:

```