[HW3]CNN_Training_for_CIFAR10_ResNET

October 27, 2025

[HW4]CNN_Training_for_CIFAR10_ResNET_Post_Quant.ipynb Library importing part and the testing part were copied from HW3. Training and pre-hook parts deleted.

[32]:
```python
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

# Importing ResNET model. Use Google Colab so upload single file.
# from resnet import *
from models import *

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


batch_size = 128

model_name = "cifar10_resnet"
# Choosing ResNET20 model
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.
# Print 4 times per epoch.

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=(1,5)):
    """Computes the precision@k for the specified values of k"""
    maxk = max(topk) # 5
    batch_size = target.size(0) # 128

    _, 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: # 1, 5
        correct_k = correct[:k].reshape(-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]
```

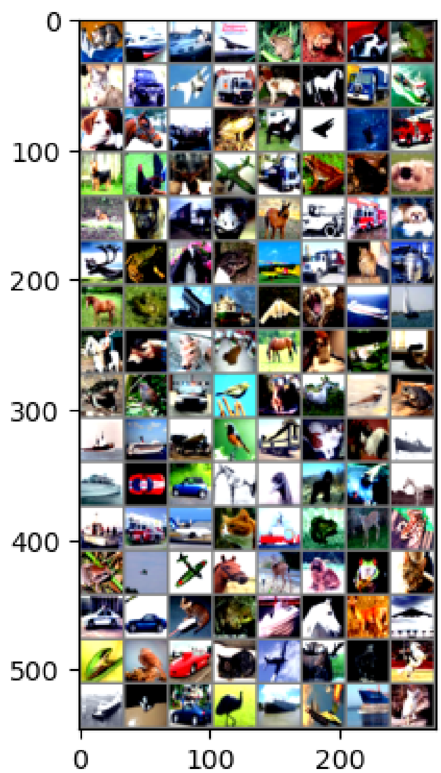
```python
    return res


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"""
    # lr devided by 10 at 50, 100, and 150 (out of 200)
    adjust_list = [50, 100, 200]
    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
```

[33]:

```python
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 = next(dataiter) ## If you run this line, the next data batch is␣
 ↪called subsequently.

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

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for
floats or [0..255] for integers).
```



[40]:
```
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)
```

```
Test: [0/79]	Time 0.092 (0.092)	Loss 0.2254 (0.2254)	Prec 92.188%
(92.188%)
 * Prec 92.110%
```

Implementing post-training Quantization Copied from [Code8]_Weight_and_Activation_Quantization.ipynb

[41]:
```
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 *



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
```

[ ]:
```
Getting weights and implement quantization.
4-bit quantization
```

[37]:
```
w_alpha = 4.0
w_bits  = 4
w_delta = w_alpha / (2**(w_bits - 1) - 1)

for layer in model.modules():
    if isinstance(layer, nn.Conv2d):
        weights = layer.weight.data
        weight_quant = weight_quantize_fn(w_bit=w_bits)
        weight_quant.wgt_alpha = torch.tensor(w_alpha, device=weights.device,␣
 ↪dtype=weights.dtype)
        with torch.no_grad():
            w_quant = weight_quant(layer.weight)
            w_int   = torch.round(w_quant / w_delta)
            w_deq   = w_int * w_delta
            layer.weight.copy_(w_deq)
prec = validate(testloader, model, criterion)
# print("W:", weights)
# print("W with integer:", w_int)
```

```
Test: [0/79]    Time 0.107 (0.107)      Loss 4.2055 (4.2055)    Prec 11.719%
(11.719%)
 * Prec 10.000%
```

[ ]:
```
Getting weights and implement quantization.
8-bit quantization
```

[42]:
```
w_alpha = 4.0
w_bits  = 8
w_delta = w_alpha / (2**(w_bits - 1) - 1)

for layer in model.modules():
    if isinstance(layer, nn.Conv2d):
        weights = layer.weight.data
        weight_quant = weight_quantize_fn(w_bit=w_bits)
        weight_quant.wgt_alpha = torch.tensor(w_alpha, device=weights.device,␣
 ↪dtype=weights.dtype)
        with torch.no_grad():
```

```
            w_quant = weight_quant(layer.weight)
            w_int   = torch.round(w_quant / w_delta)
            w_deq   = w_int * w_delta
            layer.weight.copy_(w_deq)
prec = validate(testloader, model, criterion)
# print("W:", weights)
# print("W with integer:", w_int)
```

```
Test: [0/79]    Time 0.154 (0.154)      Loss 0.3063 (0.3063)    Prec 92.188%
(92.188%)
 * Prec 88.500%
```

[ ]:
```
From the above results, we can see that using 4-bits have a much lower accuracy␣
↪than using 8-bit quantization. After quantization, accuracy decreases, which␣
↪fulfills expectation.
```