ResNet20quant

October 28, 2025

[1]:
```
# !pip uninstall -y torch torchvision torchaudio
# !pip install torch torchvision --index-url https://download.pytorch.org/whl/
  ↪cu121
```

[2]:
```
from google.colab import drive
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call
drive.mount("/content/drive", force_remount=True).
```

[3]:
```
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
import sys
sys.path.append('/content/drive/MyDrive/colab')
from models.resnet import *

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

batch_size = 128

model_name = "ResNet20"
model = resnet20_cifar()
```

```
normalize = transforms.Normalize(mean=[0.491, 0.482, 0.447], std=[0.247, 0.243,␣
 ↪0.262])

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 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)  # reshape(-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
```

```
=> Building model…
```

[4]:
```
# fdir = 'result/'+str(model_name)+'/model_best.pth.tar'
fdir = '/content/drive/MyDrive/colab/result/ResNet20/model_best.pth.tar'


checkpoint = torch.load(fdir)
model.load_state_dict(checkpoint['state_dict'])
model.cuda()
criterion = nn.CrossEntropyLoss().cuda()
```

[5]:
```
import copy
def quant(bit, model):

  model_quant = copy.deepcopy(model)

  qmax = 2 ** (bit - 1) - 1
  qmin = -2 ** (bit - 1)

  for layer in model_quant.modules():
      if isinstance(layer, torch.nn.Conv2d):

          w = layer.weight.data
          alpha = w.abs().max()
          delta = alpha / qmax

          # quantize to integer
          w_int = (w / delta).round().clamp(qmin, qmax)

          # dequantize back to floating point
          wq = w_int * delta
          layer.weight.data = wq

  return model_quant
```

[6]:
```
print("Before quantization")
for layer in model.modules():
    if isinstance(layer, torch.nn.Conv2d):
        print(layer.weight[0])
        break
print("After 4 bit quantization:")
model_4bit = quant(4, model)
for layer in model_4bit.modules():
```

```
      if isinstance(layer, torch.nn.Conv2d):
        print(layer.weight[0])
        break
print("After 8 bit quantization")
model_8bit = quant(8, model)
for layer in model_8bit.modules():
      if isinstance(layer, torch.nn.Conv2d):
        print(layer.weight[0])
        break
```

```
Before quantization
tensor([[[ 8.8885e-01,  5.7721e-01,  1.9116e-01],
         [ 1.0408e+00, -4.5876e-04, -5.7708e-01],
         [ 1.7385e-01, -1.0765e+00, -1.2311e+00]],

        [[-6.0305e-01, -6.6866e-02,  5.6060e-01],
         [-1.4268e-01,  9.9659e-02,  9.5796e-01],
         [-5.2159e-01, -2.3445e-01,  1.9937e-01]],

        [[-1.0861e+00, -4.7575e-01,  3.2182e-01],
         [-6.1251e-01, -3.6302e-02,  1.2325e+00],
         [-7.0419e-01,  9.5223e-02,  9.9945e-01]]], device='cuda:0',
       grad_fn=<SelectBackward0>)
After 4 bit quantization:
tensor([[[ 0.6084,  0.6084,  0.0000],
         [ 1.2168, -0.0000, -0.6084],
         [ 0.0000, -1.2168, -1.2168]],

        [[-0.6084, -0.0000,  0.6084],
         [-0.0000,  0.0000,  1.2168],
         [-0.6084, -0.0000,  0.0000]],

        [[-1.2168, -0.6084,  0.6084],
         [-0.6084, -0.0000,  1.2168],
         [-0.6084,  0.0000,  1.2168]]], device='cuda:0',
       grad_fn=<SelectBackward0>)
After 8 bit quantization
tensor([[[ 0.9054,  0.5701,  0.2012],
         [ 1.0395, -0.0000, -0.5701],
         [ 0.1677, -1.0731, -1.2407]],

        [[-0.6036, -0.0671,  0.5701],
         [-0.1341,  0.1006,  0.9725],
         [-0.5365, -0.2347,  0.2012]],

        [[-1.0731, -0.4695,  0.3353],
         [-0.6036, -0.0335,  1.2407],
```

```
          [-0.7042,  0.1006,  1.0060]]], device='cuda:0',
       grad_fn=<SelectBackward0>)
```

[7]:
```
model.eval()
model.cuda()

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

```
Test: [0/79]    Time 0.475 (0.475)      Loss 0.5547 (0.5547)    Prec 90.625%
(90.625%)
 * Prec 90.280%
```

[8]:
```
model_4bit.eval()
model_4bit.cuda()

prec = validate(testloader, model_4bit, criterion)
```

```
Test: [0/79]    Time 0.186 (0.186)      Loss 1.1068 (1.1068)    Prec 79.688%
(79.688%)
 * Prec 81.900%
```

[9]:
```
model_8bit.eval()
model_8bit.cuda()

prec = validate(testloader, model_8bit, criterion)
```

```
Test: [0/79]    Time 0.216 (0.216)      Loss 0.5520 (0.5520)    Prec 90.625%
(90.625%)
 * Prec 90.190%
```

My initial model has accuracy of 90.28%. The model with 4 bit quantization has accuracy of 81.9%, which is much lower than the original model. The model with 8 bit quantization has accuracy of 90.19%, which has similar accuracy to the original model but slightly lower. Note that the accuracy for 4 bit quantization and 8 bit quantization model fluctuate when re-run the notebook, where 4 bit quantization always has lower accuracy while 8 bit quantization is always similar to the original accuracy but either higher or lower than it.

[ ]: