_vgg16_float_ws_prune

December 3, 2024

[ ]:
```
import os
import time
import shutil

import torch
import torch.nn as nn

import torchvision
import torchvision.transforms as transforms

from models import *
from models.prune_util import *

import os
os.environ["CUDA_DEVICE_ORDER"]="PCI_BUS_ID"
os.environ["CUDA_VISIBLE_DEVICES"]="0"

import gc
gc.collect()
torch.cuda.empty_cache()

global best_prec

FULL_MODEL_PATH = f"result/VGG16_Full/model_best.pth.tar"

batch_size = 64
model_name = f"VGG16_ws_iter_prune_0.78"
fdir = 'result/' + model_name
model = VGG16()
checkpoint = torch.load(FULL_MODEL_PATH)
model.load_state_dict(checkpoint['state_dict'])
model.cuda()
device = torch.device("cuda")
lr = 3e-3
epochs = 100
prune_schedule = {0: 4/9,
                  5: 1/9,
                  15: 1/9,
                  25: 1/9}

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)


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)


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):
```

```python
        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, new_lr):
    for param_group in optimizer.param_groups:
        param_group['lr'] = new_lr

def train_model(model, fdir, criterion, optimizer, epochs, prune_schedule:
 ↪dict=None):
    os.makedirs(fdir, exist_ok=True)

    best_prec = 0

    #model = nn.DataParallel(model).cuda()
    model.cuda()
    criterion = criterion.cuda()
    #cudnn.benchmark = True

    for epoch in range(0, epochs):

        if prune_schedule is not None and epoch in prune_schedule:
            ws_prune_vgg16(model, prune_schedule[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
        if prune_schedule is None or (prune_schedule is not None and epoch >=␣
↪list(prune_schedule.keys())[-1]):
            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)

def val_model(model):

    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)))
```

```
/tmp/ipykernel_60378/3379942828.py:30: FutureWarning: You are using `torch.load`
with `weights_only=False` (the current default value), which uses the default
pickle module implicitly. It is possible to construct malicious pickle data
which will execute arbitrary code during unpickling (See
https://github.com/pytorch/pytorch/blob/main/SECURITY.md#untrusted-models for
more details). In a future release, the default value for `weights_only` will be
flipped to `True`. This limits the functions that could be executed during
unpickling. Arbitrary objects will no longer be allowed to be loaded via this
mode unless they are explicitly allowlisted by the user via
`torch.serialization.add_safe_globals`. We recommend you start setting
`weights_only=True` for any use case where you don't have full control of the
```

```
loaded file. Please open an issue on GitHub for any issues related to this
experimental feature.
  checkpoint = torch.load(FULL_MODEL_PATH)
```

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

[ ]:
```
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=lr)

train_model(model, fdir, criterion, optimizer, epochs, prune_schedule)
```

[4]:
```
model = VGG16()
ws_prune_vgg16(model, 0.78)

PATH = f"{fdir}/model_best.pth.tar"
checkpoint = torch.load(PATH)
model.load_state_dict(checkpoint['state_dict'])
model.cuda()

val_model(model)

print_sparsity(model)
```

```
Pruning 7 kij-sticks out of 9 kij-sticks per output channel (77.8% pruned)
Pruning 7 kij-sticks out of 9 kij-sticks per output channel (77.8% pruned)
Pruning 7 kij-sticks out of 9 kij-sticks per output channel (77.8% pruned)
Pruning 7 kij-sticks out of 9 kij-sticks per output channel (77.8% pruned)
Pruning 7 kij-sticks out of 9 kij-sticks per output channel (77.8% pruned)
Pruning 7 kij-sticks out of 9 kij-sticks per output channel (77.8% pruned)
Pruning 7 kij-sticks out of 9 kij-sticks per output channel (77.8% pruned)
Pruning 7 kij-sticks out of 9 kij-sticks per output channel (77.8% pruned)
Pruning 7 kij-sticks out of 9 kij-sticks per output channel (77.8% pruned)
Pruning 7 kij-sticks out of 9 kij-sticks per output channel (77.8% pruned)
Pruning 7 kij-sticks out of 9 kij-sticks per output channel (77.8% pruned)
Pruning 7 kij-sticks out of 9 kij-sticks per output channel (77.8% pruned)
```

```
/tmp/ipykernel_60378/3294384175.py:5: FutureWarning: You are using `torch.load`
with `weights_only=False` (the current default value), which uses the default
pickle module implicitly. It is possible to construct malicious pickle data
which will execute arbitrary code during unpickling (See
https://github.com/pytorch/pytorch/blob/main/SECURITY.md#untrusted-models for
more details). In a future release, the default value for `weights_only` will be
flipped to `True`. This limits the functions that could be executed during
unpickling. Arbitrary objects will no longer be allowed to be loaded via this
mode unless they are explicitly allowlisted by the user via
`torch.serialization.add_safe_globals`. We recommend you start setting
`weights_only=True` for any use case where you don't have full control of the
loaded file. Please open an issue on GitHub for any issues related to this
```

```
experimental feature.
  checkpoint = torch.load(PATH)

Test set: Accuracy: 9077/10000 (91%)
```