```
import argparse
import os
import random
import shutil
import time
import warnings

import torch
import torch.nn as nn
import torch.nn.parallel
import torch.backends.cudnn as cudnn
import torch.distributed as dist
import torch.optim
import torch.multiprocessing as mp
import torch.utils.data
import torch.utils.data.distributed
import torchvision.transforms as transforms
import torchvision.datasets as datasets
import torchvision.models as models

model_names = sorted(name for name in
models.__dict__
    if name.islower() and not name.startswith("__")
    and callable(models.__dict__[name]))

parser =
argparse.ArgumentParser(description='PyTorch
ImageNet Training')
parser.add_argument('data', metavar='DIR',
                    help='path to dataset')
parser.add_argument('-a', '--arch', metavar='ARCH',
default='resnet18',
                    choices=model_names,
                    help='model architecture: ' +
                        ' | '.join(model_names) +
                        ' (default: resnet18)')
parser.add_argument('-j', '--workers', default=4,
```

```
type=int, metavar='N',
                    help='number of data loading
workers (default: 4)')
parser.add_argument('--epochs', default=90,
type=int, metavar='N',
                    help='number of total epochs to
run')
parser.add_argument('--start-epoch', default=0,
type=int, metavar='N',
                    help='manual epoch number
(useful on restarts)')
parser.add_argument('-b', '--batch-size',
default=256, type=int,
                    metavar='N',
                    help='mini-batch size (default:
256), this is the total '
                         'batch size of all GPUs on
the current node when '
                         'using Data Parallel or
Distributed Data Parallel')
parser.add_argument('--lr', '--learning-rate',
default=0.1, type=float,
                    metavar='LR', help='initial
learning rate', dest='lr')
parser.add_argument('--momentum', default=0.9,
type=float, metavar='M',
                    help='momentum')
parser.add_argument('--wd', '--weight-decay',
default=1e-4, type=float,
                    metavar='W', help='weight decay
(default: 1e-4)',
                    dest='weight_decay')
parser.add_argument('-p', '--print-freq',
default=10, type=int,
                    metavar='N', help='print
frequency (default: 10)')
parser.add_argument('--resume', default='',
```

```
type=str, metavar='PATH',
                    help='path to latest checkpoint
(default: none)')
parser.add_argument('-e', '--evaluate',
dest='evaluate', action='store_true',
                    help='evaluate model on
validation set')
parser.add_argument('--pretrained',
dest='pretrained', action='store_true',
                    help='use pre-trained model')
parser.add_argument('--world-size', default=-1,
type=int,
                    help='number of nodes for
distributed training')
parser.add_argument('--rank', default=-1, type=int,
                    help='node rank for distributed
training')
parser.add_argument('--dist-url', default='tcp://
224.66.41.62:23456', type=str,
                    help='url used to set up
distributed training')
parser.add_argument('--dist-backend',
default='nccl', type=str,
                    help='distributed backend')
parser.add_argument('--seed', default=None,
type=int,
                    help='seed for initializing
training. ')
parser.add_argument('--gpu', default=None, type=int,
                    help='GPU id to use.')
parser.add_argument('--multiprocessing-
distributed', action='store_true',
                    help='Use multi-processing
distributed training to launch '
                         'N processes per node,
which has N GPUs. This is the '
                         'fastest way to use
```

```
PyTorch for either single node or '
                            'multi node data parallel
training')

best_acc1 = 0


def main():
    args = parser.parse_args()

    if args.seed is not None:
        random.seed(args.seed)
        torch.manual_seed(args.seed)
        cudnn.deterministic = True
        warnings.warn('You have chosen to seed
training. '
                      'This will turn on the CUDNN
deterministic setting, '
                      'which can slow down your
training considerably! '
                      'You may see unexpected
behavior when restarting '
                      'from checkpoints.')

    if args.gpu is not None:
        warnings.warn('You have chosen a specific
GPU. This will completely '
                      'disable data parallelism.')

    if args.dist_url == "env://" and
args.world_size == -1:
        args.world_size =
int(os.environ["WORLD_SIZE"])

    args.distributed = args.world_size > 1 or
args.multiprocessing_distributed
```

```
    ngpus_per_node = torch.cuda.device_count()
    if args.multiprocessing_distributed:
        # Since we have ngpus_per_node processes
per node, the total world_size
        # needs to be adjusted accordingly
        args.world_size = ngpus_per_node *
args.world_size
        # Use torch.multiprocessing.spawn to launch
distributed processes: the
        # main_worker process function
        mp.spawn(main_worker,
nprocs=ngpus_per_node, args=(ngpus_per_node, args))
    else:
        # Simply call main_worker function
        main_worker(args.gpu, ngpus_per_node, args)


def main_worker(gpu, ngpus_per_node, args):
    global best_acc1
    args.gpu = gpu

    if args.gpu is not None:
        print("Use GPU: {} for
training".format(args.gpu))

    if args.distributed:
        if args.dist_url == "env://" and args.rank
== -1:
            args.rank = int(os.environ["RANK"])
        if args.multiprocessing_distributed:
            # For multiprocessing distributed
training, rank needs to be the
            # global rank among all the processes
            args.rank = args.rank * ngpus_per_node
+ gpu

dist.init_process_group(backend=args.dist_backend,
```

```
init_method=args.dist_url,

world_size=args.world_size, rank=args.rank)
    # create model
    if args.pretrained:
        print("=> using pre-trained model
'{}'".format(args.arch))
        model = models.__dict__[args.arch]
(pretrained=True)
    else:
        print("=> creating model
'{}'".format(args.arch))
        model = models.__dict__[args.arch]()

    if args.distributed:
        # For multiprocessing distributed,
DistributedDataParallel constructor
        # should always set the single device
scope, otherwise,
        # DistributedDataParallel will use all
available devices.
        if args.gpu is not None:
            torch.cuda.set_device(args.gpu)
            model.cuda(args.gpu)
            # When using a single GPU per process
and per
            # DistributedDataParallel, we need to
divide the batch size
            # ourselves based on the total number
of GPUs we have
            args.batch_size = int(args.batch_size /
ngpus_per_node)
            args.workers = int((args.workers +
ngpus_per_node - 1) / ngpus_per_node)
            model =
torch.nn.parallel.DistributedDataParallel(model,
device_ids=[args.gpu])
```

```
        else:
            model.cuda()
            # DistributedDataParallel will divide
and allocate batch_size to all
            # available GPUs if device_ids are not
set
            model =
torch.nn.parallel.DistributedDataParallel(model)
    elif args.gpu is not None:
        torch.cuda.set_device(args.gpu)
        model = model.cuda(args.gpu)
    else:
        # DataParallel will divide and allocate
batch_size to all available GPUs
        if args.arch.startswith('alexnet') or
args.arch.startswith('vgg'):
            model.features =
torch.nn.DataParallel(model.features)
            model.cuda()
        else:
            model =
torch.nn.DataParallel(model).cuda()

    # define loss function (criterion) and optimizer
    criterion = nn.CrossEntropyLoss().cuda(args.gpu)

    optimizer = torch.optim.SGD(model.parameters(),
args.lr,

momentum=args.momentum,

weight_decay=args.weight_decay)

    # optionally resume from a checkpoint
    if args.resume:
        if os.path.isfile(args.resume):
            print("=> loading checkpoint
```

```
'{}'".format(args.resume))
            if args.gpu is None:
                checkpoint = torch.load(args.resume)
            else:
                # Map model to be loaded to
specified single gpu.
                loc = 'cuda:{}'.format(args.gpu)
                checkpoint =
torch.load(args.resume, map_location=loc)
            args.start_epoch = checkpoint['epoch']
            best_acc1 = checkpoint['best_acc1']
            if args.gpu is not None:
                # best_acc1 may be from a
checkpoint from a different GPU
                best_acc1 = best_acc1.to(args.gpu)

model.load_state_dict(checkpoint['state_dict'])

optimizer.load_state_dict(checkpoint['optimizer'])
            print("=> loaded checkpoint '{}' (epoch
{})"
                  .format(args.resume,
checkpoint['epoch']))
        else:
            print("=> no checkpoint found at
'{}'".format(args.resume))

    cudnn.benchmark = True

    # Data loading code
    traindir = os.path.join(args.data, 'train')
    valdir = os.path.join(args.data, 'val')
    normalize = transforms.Normalize(mean=[0.485,
0.456, 0.406],
                                     std=[0.229,
0.224, 0.225])
```

```
    train_dataset = datasets.ImageFolder(
        traindir,
        transforms.Compose([
            transforms.RandomResizedCrop(224),
            transforms.RandomHorizontalFlip(),
            transforms.ToTensor(),
            normalize,
        ]))

    if args.distributed:
        train_sampler =
torch.utils.data.distributed.DistributedSampler(train_datase
    else:
        train_sampler = None

    train_loader = torch.utils.data.DataLoader(
        train_dataset, batch_size=args.batch_size,
shuffle=(train_sampler is None),
        num_workers=args.workers, pin_memory=True,
sampler=train_sampler)

    val_loader = torch.utils.data.DataLoader(
        datasets.ImageFolder(valdir,
transforms.Compose([
            transforms.Resize(256),
            transforms.CenterCrop(224),
            transforms.ToTensor(),
            normalize,
        ])),
        batch_size=args.batch_size, shuffle=False,
        num_workers=args.workers, pin_memory=True)

    if args.evaluate:
        validate(val_loader, model, criterion, args)
        return

    for epoch in range(args.start_epoch,
```

```
args.epochs):
        if args.distributed:
            train_sampler.set_epoch(epoch)
        adjust_learning_rate(optimizer, epoch, args)

        # train for one epoch
        train(train_loader, model, criterion,
optimizer, epoch, args)

        # evaluate on validation set
        acc1 = validate(val_loader, model,
criterion, args)

        # remember best acc@1 and save checkpoint
        is_best = acc1 > best_acc1
        best_acc1 = max(acc1, best_acc1)

        if not args.multiprocessing_distributed or
(args.multiprocessing_distributed
                and args.rank % ngpus_per_node ==
0):
            save_checkpoint({
                'epoch': epoch + 1,
                'arch': args.arch,
                'state_dict': model.state_dict(),
                'best_acc1': best_acc1,
                'optimizer' :
optimizer.state_dict(),
            }, is_best)


def train(train_loader, model, criterion,
optimizer, epoch, args):
    batch_time = AverageMeter('Time', ':6.3f')
    data_time = AverageMeter('Data', ':6.3f')
    losses = AverageMeter('Loss', ':.4e')
    top1 = AverageMeter('Acc@1', ':6.2f')
```

```
    top5 = AverageMeter('Acc@5', ':6.2f')
    progress = ProgressMeter(
        len(train_loader),
        [batch_time, data_time, losses, top1, top5],
        prefix="Epoch: [{}]".format(epoch))

    # switch to train mode
    model.train()

    end = time.time()
    for i, (images, target) in
enumerate(train_loader):
        # measure data loading time
        data_time.update(time.time() - end)

        if args.gpu is not None:
            images = images.cuda(args.gpu,
non_blocking=True)
        target = target.cuda(args.gpu,
non_blocking=True)

        # compute output
        output = model(images)
        loss = criterion(output, target)

        # measure accuracy and record loss
        acc1, acc5 = accuracy(output, target,
topk=(1, 5))
        losses.update(loss.item(), images.size(0))
        top1.update(acc1[0], images.size(0))
        top5.update(acc5[0], images.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 % args.print_freq == 0:
            progress.display(i)


def validate(val_loader, model, criterion, args):
    batch_time = AverageMeter('Time', ':6.3f')
    losses = AverageMeter('Loss', ':.4e')
    top1 = AverageMeter('Acc@1', ':6.2f')
    top5 = AverageMeter('Acc@5', ':6.2f')
    progress = ProgressMeter(
        len(val_loader),
        [batch_time, losses, top1, top5],
        prefix='Test: ')

    # switch to evaluate mode
    model.eval()

    with torch.no_grad():
        end = time.time()
        for i, (images, target) in
enumerate(val_loader):
            if args.gpu is not None:
                images = images.cuda(args.gpu,
non_blocking=True)
            target = target.cuda(args.gpu,
non_blocking=True)

            # compute output
            output = model(images)
            loss = criterion(output, target)

            # measure accuracy and record loss
            acc1, acc5 = accuracy(output, target,
```

```
topk=(1, 5))
            losses.update(loss.item(),
images.size(0))
            top1.update(acc1[0], images.size(0))
            top5.update(acc5[0], images.size(0))

            # measure elapsed time
            batch_time.update(time.time() - end)
            end = time.time()

            if i % args.print_freq == 0:
                progress.display(i)

        # TODO: this should also be done with the
ProgressMeter
        print(' * Acc@1 {top1.avg:.3f} Acc@5
{top5.avg:.3f}'
              .format(top1=top1, top5=top5))

    return top1.avg


def save_checkpoint(state, is_best,
filename='checkpoint.pth.tar'):
    torch.save(state, filename)
    if is_best:
        shutil.copyfile(filename,
'model_best.pth.tar')


class AverageMeter(object):
    """Computes and stores the average and current
value"""
    def __init__(self, name, fmt=':f'):
        self.name = name
        self.fmt = fmt
        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 __str__(self):
        fmtstr = '{name} {val' + self.fmt + '}
({avg' + self.fmt + '})'
        return fmtstr.format(**self.__dict__)


class ProgressMeter(object):
    def __init__(self, num_batches, meters,
prefix=""):
        self.batch_fmtstr =
self._get_batch_fmtstr(num_batches)
        self.meters = meters
        self.prefix = prefix

    def display(self, batch):
        entries = [self.prefix +
self.batch_fmtstr.format(batch)]
        entries += [str(meter) for meter in
self.meters]
        print('\t'.join(entries))

    def _get_batch_fmtstr(self, num_batches):
        num_digits = len(str(num_batches // 1))
        fmt = '{:' + str(num_digits) + 'd}'
```

```
        return '[' + fmt + '/' +
fmt.format(num_batches) + ']'


def adjust_learning_rate(optimizer, epoch, args):
    """Sets the learning rate to the initial LR
decayed by 10 every 30 epochs"""
    lr = args.lr * (0.1 ** (epoch // 30))
    for param_group in optimizer.param_groups:
        param_group['lr'] = lr


def accuracy(output, target, topk=(1,)):
    """Computes the accuracy over the k top
predictions for the specified values of k"""
    with torch.no_grad():
        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, keepdim=True)
            res.append(correct_k.mul_(100.0 /
batch_size))
        return res


if __name__ == '__main__':
    main()
```