```python
from __future__ import print_function
import argparse
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torchvision import datasets, transforms
from torch.optim.lr_scheduler import StepLR


class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, 3, 1)
        self.conv2 = nn.Conv2d(32, 64, 3, 1)
        self.dropout1 = nn.Dropout2d(0.25)
        self.dropout2 = nn.Dropout2d(0.5)
        self.fc1 = nn.Linear(9216, 128)
        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):
        x = self.conv1(x)
        x = F.relu(x)
        x = self.conv2(x)
        x = F.max_pool2d(x, 2)
        x = self.dropout1(x)
        x = torch.flatten(x, 1)
        x = self.fc1(x)
        x = F.relu(x)
        x = self.dropout2(x)
        x = self.fc2(x)
        output = F.log_softmax(x, dim=1)
        return output


def train(args, model, device, train_loader,
optimizer, epoch):
    model.train()
    for batch_idx, (data, target) in
```

```
enumerate(train_loader):
        data, target = data.to(device),
target.to(device)
        optimizer.zero_grad()
        output = model(data)
        loss = F.nll_loss(output, target)
        loss.backward()
        optimizer.step()
        if batch_idx % args.log_interval == 0:
            print('Train Epoch: {} [{}/{} ({:.0f}%)]
\tLoss: {:.6f}'.format(
                epoch, batch_idx * len(data),
len(train_loader.dataset),
                100. * batch_idx / len(train_loader),
loss.item()))


def test(args, model, device, test_loader):
    model.eval()
    test_loss = 0
    correct = 0
    with torch.no_grad():
        for data, target in test_loader:
            data, target = data.to(device),
target.to(device)
            output = model(data)
            test_loss += F.nll_loss(output, target,
reduction='sum').item()  # sum up batch loss
            pred = output.argmax(dim=1, keepdim=True)
# get the index of the max log-probability
            correct +=
pred.eq(target.view_as(pred)).sum().item()

    test_loss /= len(test_loader.dataset)

    print('\nTest set: Average loss: {:.4f}, Accuracy:
{}/{} ({:.0f}%)\n'.format(
        test_loss, correct, len(test_loader.dataset),
        100. * correct / len(test_loader.dataset)))
```

```
def main():
    # Training settings
    parser =
argparse.ArgumentParser(description='PyTorch MNIST
Example')
    parser.add_argument('--batch-size', type=int,
default=64, metavar='N',
                        help='input batch size for
training (default: 64)')
    parser.add_argument('--test-batch-size', type=int,
default=1000, metavar='N',
                        help='input batch size for
testing (default: 1000)')
    parser.add_argument('--epochs', type=int,
default=14, metavar='N',
                        help='number of epochs to
train (default: 14)')
    parser.add_argument('--lr', type=float,
default=1.0, metavar='LR',
                        help='learning rate (default:
1.0)')
    parser.add_argument('--gamma', type=float,
default=0.7, metavar='M',
                        help='Learning rate step gamma
(default: 0.7)')
    parser.add_argument('--no-cuda',
action='store_true', default=False,
                        help='disables CUDA training')
    parser.add_argument('--seed', type=int, default=1,
metavar='S',
                        help='random seed (default:
1)')
    parser.add_argument('--log-interval', type=int,
default=10, metavar='N',
                        help='how many batches to wait
before logging training status')
```

```
    parser.add_argument('--save-model',
action='store_true', default=False,
                        help='For Saving the current
Model')
    args = parser.parse_args()
    use_cuda = not args.no_cuda and
torch.cuda.is_available()

    torch.manual_seed(args.seed)

    device = torch.device("cuda" if use_cuda else
"cpu")

    kwargs = {'num_workers': 1, 'pin_memory': True} if
use_cuda else {}
    train_loader = torch.utils.data.DataLoader(
        datasets.MNIST('../data', train=True,
download=True,
                       transform=transforms.Compose([
                           transforms.ToTensor(),

transforms.Normalize((0.1307,), (0.3081,))
                       ])),
        batch_size=args.batch_size, shuffle=True,
**kwargs)
    test_loader = torch.utils.data.DataLoader(
        datasets.MNIST('../data', train=False,
transform=transforms.Compose([
                           transforms.ToTensor(),

transforms.Normalize((0.1307,), (0.3081,))
                       ])),
        batch_size=args.test_batch_size, shuffle=True,
**kwargs)

    model = Net().to(device)
    optimizer = optim.Adadelta(model.parameters(),
lr=args.lr)
```

```
    scheduler = StepLR(optimizer, step_size=1,
gamma=args.gamma)
    for epoch in range(1, args.epochs + 1):
        train(args, model, device, train_loader,
optimizer, epoch)
        test(args, model, device, test_loader)
        scheduler.step()

    if args.save_model:
        torch.save(model.state_dict(), "mnist_cnn.pt")


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

```
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()
```

Deep Convolution Generative Adversarial Networks

This example implements the paper Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks

The implementation is very close to the Torch implementation dcgan.torch

After every 100 training iterations, the files `real_samples.png` and `fake_samples.png` are written to disk with the samples from the generative model.

After every epoch, models are saved to: `netG_epoch_%d.pth` and `netD_epoch_%d.pth`

Downloading the dataset

You can download the LSUN dataset by cloning this repo and running

```
python download.py -c bedroom
```

Usage

```
usage: main.py [-h] --dataset DATASET --dataroot DATAROOT [--workers WOR
               [--batchSize BATCHSIZE] [--imageSize IMAGESIZE] [--nz NZ]
               [--ngf NGF] [--ndf NDF] [--niter NITER] [--lr LR]
               [--beta1 BETA1] [--cuda] [--ngpu NGPU] [--netG NETG]
               [--netD NETD]

optional arguments:
  -h, --help            show this help message and exit
  --dataset DATASET     cifar10 | lsun | mnist |imagenet | folder | lfw
  --dataroot DATAROOT   path to dataset
  --workers WORKERS     number of data loading workers
  --batchSize BATCHSIZE input batch size
  --imageSize IMAGESIZE the height / width of the input image to network
  --nz NZ               size of the latent z vector
  --ngf NGF
  --ndf NDF
  --niter NITER         number of epochs to train for
  --lr LR               learning rate, default=0.0002
  --beta1 BETA1         beta1 for adam. default=0.5
  --cuda                enables cuda
  --ngpu NGPU           number of GPUs to use
  --netG NETG           path to netG (to continue training)
  --netD NETD           path to netD (to continue training)
  --outf OUTF           folder to output images and model checkpoints
  --manualSeed SEED     manual seed
```

```
  --classes CLASSES     comma separated list of classes for the lsun dat
```

```
from __future__ import print_function
import argparse
import os
import random
import torch
import torch.nn as nn
import torch.nn.parallel
import torch.backends.cudnn as cudnn
import torch.optim as optim
import torch.utils.data
import torchvision.datasets as dset
import torchvision.transforms as transforms
import torchvision.utils as vutils


parser = argparse.ArgumentParser()
parser.add_argument('--dataset', required=True,
help='cifar10 | lsun | mnist |imagenet | folder |
lfw | fake')
parser.add_argument('--dataroot', required=True,
help='path to dataset')
parser.add_argument('--workers', type=int,
help='number of data loading workers', default=2)
parser.add_argument('--batchSize', type=int,
default=64, help='input batch size')
parser.add_argument('--imageSize', type=int,
default=64, help='the height / width of the input
image to network')
parser.add_argument('--nz', type=int, default=100,
help='size of the latent z vector')
parser.add_argument('--ngf', type=int, default=64)
parser.add_argument('--ndf', type=int, default=64)
parser.add_argument('--niter', type=int,
default=25, help='number of epochs to train for')
parser.add_argument('--lr', type=float,
default=0.0002, help='learning rate,
default=0.0002')
```

```
parser.add_argument('--beta1', type=float,
default=0.5, help='beta1 for adam. default=0.5')
parser.add_argument('--cuda', action='store_true',
help='enables cuda')
parser.add_argument('--ngpu', type=int, default=1,
help='number of GPUs to use')
parser.add_argument('--netG', default='',
help="path to netG (to continue training)")
parser.add_argument('--netD', default='',
help="path to netD (to continue training)")
parser.add_argument('--outf', default='.',
help='folder to output images and model
checkpoints')
parser.add_argument('--manualSeed', type=int,
help='manual seed')
parser.add_argument('--classes', default='bedroom',
help='comma separated list of classes for the lsun
data set')

opt = parser.parse_args()
print(opt)

try:
    os.makedirs(opt.outf)
except OSError:
    pass

if opt.manualSeed is None:
    opt.manualSeed = random.randint(1, 10000)
print("Random Seed: ", opt.manualSeed)
random.seed(opt.manualSeed)
torch.manual_seed(opt.manualSeed)

cudnn.benchmark = True

if torch.cuda.is_available() and not opt.cuda:
    print("WARNING: You have a CUDA device, so you
```

```
should probably run with --cuda")

if opt.dataset in ['imagenet', 'folder', 'lfw']:
    # folder dataset
    dataset = dset.ImageFolder(root=opt.dataroot,

transform=transforms.Compose([

transforms.Resize(opt.imageSize),

transforms.CenterCrop(opt.imageSize),

transforms.ToTensor(),

transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5,
0.5)),
                               ]))
    nc=3
elif opt.dataset == 'lsun':
    classes = [ c + '_train' for c in
opt.classes.split(',')]
    dataset = dset.LSUN(root=opt.dataroot,
classes=classes,

transform=transforms.Compose([

transforms.Resize(opt.imageSize),

transforms.CenterCrop(opt.imageSize),
                            transforms.ToTensor(),

transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5,
0.5)),
                        ]))
    nc=3
elif opt.dataset == 'cifar10':
    dataset = dset.CIFAR10(root=opt.dataroot,
```

```
download=True,

transform=transforms.Compose([

transforms.Resize(opt.imageSize),

transforms.ToTensor(),

transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5,
0.5)),
                           ]))
    nc=3

elif opt.dataset == 'mnist':
        dataset = dset.MNIST(root=opt.dataroot,
download=True,

transform=transforms.Compose([

transforms.Resize(opt.imageSize),

transforms.ToTensor(),

transforms.Normalize((0.5,), (0.5,)),
                           ]))
        nc=1

elif opt.dataset == 'fake':
    dataset = dset.FakeData(image_size=(3,
opt.imageSize, opt.imageSize),

transform=transforms.ToTensor())
    nc=3

assert dataset
dataloader = torch.utils.data.DataLoader(dataset,
batch_size=opt.batchSize,
```

```
shuffle=True, num_workers=int(opt.workers))

device = torch.device("cuda:0" if opt.cuda else
"cpu")
ngpu = int(opt.ngpu)
nz = int(opt.nz)
ngf = int(opt.ngf)
ndf = int(opt.ndf)


# custom weights initialization called on netG and
netD
def weights_init(m):
    classname = m.__class__.__name__
    if classname.find('Conv') != -1:
        m.weight.data.normal_(0.0, 0.02)
    elif classname.find('BatchNorm') != -1:
        m.weight.data.normal_(1.0, 0.02)
        m.bias.data.fill_(0)


class Generator(nn.Module):
    def __init__(self, ngpu):
        super(Generator, self).__init__()
        self.ngpu = ngpu
        self.main = nn.Sequential(
            # input is Z, going into a convolution
            nn.ConvTranspose2d(     nz, ngf * 8, 4,
1, 0, bias=False),
            nn.BatchNorm2d(ngf * 8),
            nn.ReLU(True),
            # state size. (ngf*8) x 4 x 4
            nn.ConvTranspose2d(ngf * 8, ngf * 4, 4,
2, 1, bias=False),
            nn.BatchNorm2d(ngf * 4),
            nn.ReLU(True),
```

```
            # state size. (ngf*4) x 8 x 8
            nn.ConvTranspose2d(ngf * 4, ngf * 2, 4,
2, 1, bias=False),
            nn.BatchNorm2d(ngf * 2),
            nn.ReLU(True),
            # state size. (ngf*2) x 16 x 16
            nn.ConvTranspose2d(ngf * 2,     ngf, 4,
2, 1, bias=False),
            nn.BatchNorm2d(ngf),
            nn.ReLU(True),
            # state size. (ngf) x 32 x 32
            nn.ConvTranspose2d(    ngf,      nc, 4,
2, 1, bias=False),
            nn.Tanh()
            # state size. (nc) x 64 x 64
        )

    def forward(self, input):
        if input.is_cuda and self.ngpu > 1:
            output =
nn.parallel.data_parallel(self.main, input,
range(self.ngpu))
        else:
            output = self.main(input)
        return output


netG = Generator(ngpu).to(device)
netG.apply(weights_init)
if opt.netG != '':
    netG.load_state_dict(torch.load(opt.netG))
print(netG)


class Discriminator(nn.Module):
    def __init__(self, ngpu):
        super(Discriminator, self).__init__()
```

```
        self.ngpu = ngpu
        self.main = nn.Sequential(
            # input is (nc) x 64 x 64
            nn.Conv2d(nc, ndf, 4, 2, 1, bias=False),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. (ndf) x 32 x 32
            nn.Conv2d(ndf, ndf * 2, 4, 2, 1,
bias=False),
            nn.BatchNorm2d(ndf * 2),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. (ndf*2) x 16 x 16
            nn.Conv2d(ndf * 2, ndf * 4, 4, 2, 1,
bias=False),
            nn.BatchNorm2d(ndf * 4),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. (ndf*4) x 8 x 8
            nn.Conv2d(ndf * 4, ndf * 8, 4, 2, 1,
bias=False),
            nn.BatchNorm2d(ndf * 8),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. (ndf*8) x 4 x 4
            nn.Conv2d(ndf * 8, 1, 4, 1, 0,
bias=False),
            nn.Sigmoid()
        )

    def forward(self, input):
        if input.is_cuda and self.ngpu > 1:
            output =
nn.parallel.data_parallel(self.main, input,
range(self.ngpu))
        else:
            output = self.main(input)

        return output.view(-1, 1).squeeze(1)
```

```
netD = Discriminator(ngpu).to(device)
netD.apply(weights_init)
if opt.netD != '':
    netD.load_state_dict(torch.load(opt.netD))
print(netD)

criterion = nn.BCELoss()

fixed_noise = torch.randn(opt.batchSize, nz, 1, 1,
device=device)
real_label = 1
fake_label = 0

# setup optimizer
optimizerD = optim.Adam(netD.parameters(),
lr=opt.lr, betas=(opt.beta1, 0.999))
optimizerG = optim.Adam(netG.parameters(),
lr=opt.lr, betas=(opt.beta1, 0.999))

for epoch in range(opt.niter):
    for i, data in enumerate(dataloader, 0):
        ############################
        # (1) Update D network: maximize log(D(x))
+ log(1 - D(G(z)))
        ###########################
        # train with real
        netD.zero_grad()
        real_cpu = data[0].to(device)
        batch_size = real_cpu.size(0)
        label = torch.full((batch_size,),
real_label, device=device)

        output = netD(real_cpu)
        errD_real = criterion(output, label)
        errD_real.backward()
        D_x = output.mean().item()
```

```
        # train with fake
        noise = torch.randn(batch_size, nz, 1, 1,
device=device)
        fake = netG(noise)
        label.fill_(fake_label)
        output = netD(fake.detach())
        errD_fake = criterion(output, label)
        errD_fake.backward()
        D_G_z1 = output.mean().item()
        errD = errD_real + errD_fake
        optimizerD.step()

        ############################
        # (2) Update G network: maximize
log(D(G(z)))
        ###########################
        netG.zero_grad()
        label.fill_(real_label)  # fake labels are
real for generator cost
        output = netD(fake)
        errG = criterion(output, label)
        errG.backward()
        D_G_z2 = output.mean().item()
        optimizerG.step()

        print('[%d/%d][%d/%d] Loss_D: %.4f Loss_G:
%.4f D(x): %.4f D(G(z)): %.4f / %.4f'
              % (epoch, opt.niter, i,
len(dataloader),
                 errD.item(), errG.item(), D_x,
D_G_z1, D_G_z2))
        if i % 100 == 0:
            vutils.save_image(real_cpu,
                    '%s/real_samples.png' %
opt.outf,
                    normalize=True)
            fake = netG(fixed_noise)
```

```
            vutils.save_image(fake.detach(),
                    '%s/
fake_samples_epoch_%03d.png' % (opt.outf, epoch),
                    normalize=True)

    # do checkpointing
    torch.save(netG.state_dict(), '%s/
netG_epoch_%d.pth' % (opt.outf, epoch))
    torch.save(netD.state_dict(), '%s/
netD_epoch_%d.pth' % (opt.outf, epoch))
```

```
# Basic VAE Example

This is an improved implementation of the paper
[Auto-Encoding Variational Bayes](http://arxiv.org/
abs/1312.6114) by Kingma and Welling.
It uses ReLUs and the adam optimizer, instead of
sigmoids and adagrad. These changes make the
network converge much faster.

```bash
pip install -r requirements.txt
python main.py
```
from __future__ import print_function
import argparse
import torch
import torch.utils.data
from torch import nn, optim
from torch.nn import functional as F
from torchvision import datasets, transforms
from torchvision.utils import save_image

parser = argparse.ArgumentParser(description='VAE
MNIST Example')
parser.add_argument('--batch-size', type=int,
default=128, metavar='N',
                    help='input batch size for
training (default: 128)')
parser.add_argument('--epochs', type=int,
default=10, metavar='N',
                    help='number of epochs to train
(default: 10)')
parser.add_argument('--no-cuda',
action='store_true', default=False,
                    help='enables CUDA training')
parser.add_argument('--seed', type=int, default=1,
```

```
metavar='S',
                    help='random seed (default: 1)')
parser.add_argument('--log-interval', type=int,
default=10, metavar='N',
                    help='how many batches to wait
before logging training status')
args = parser.parse_args()
args.cuda = not args.no_cuda and
torch.cuda.is_available()

torch.manual_seed(args.seed)

device = torch.device("cuda" if args.cuda else
"cpu")

kwargs = {'num_workers': 1, 'pin_memory': True} if
args.cuda else {}
train_loader = torch.utils.data.DataLoader(
    datasets.MNIST('../data', train=True,
download=True,
                   transform=transforms.ToTensor()),
    batch_size=args.batch_size, shuffle=True,
**kwargs)
test_loader = torch.utils.data.DataLoader(
    datasets.MNIST('../data', train=False,
transform=transforms.ToTensor()),
    batch_size=args.batch_size, shuffle=True,
**kwargs)


class VAE(nn.Module):
    def __init__(self):
        super(VAE, self).__init__()

        self.fc1 = nn.Linear(784, 400)
        self.fc21 = nn.Linear(400, 20)
        self.fc22 = nn.Linear(400, 20)
```

```
        self.fc3 = nn.Linear(20, 400)
        self.fc4 = nn.Linear(400, 784)

    def encode(self, x):
        h1 = F.relu(self.fc1(x))
        return self.fc21(h1), self.fc22(h1)

    def reparameterize(self, mu, logvar):
        std = torch.exp(0.5*logvar)
        eps = torch.randn_like(std)
        return mu + eps*std

    def decode(self, z):
        h3 = F.relu(self.fc3(z))
        return torch.sigmoid(self.fc4(h3))

    def forward(self, x):
        mu, logvar = self.encode(x.view(-1, 784))
        z = self.reparameterize(mu, logvar)
        return self.decode(z), mu, logvar


model = VAE().to(device)
optimizer = optim.Adam(model.parameters(), lr=1e-3)


# Reconstruction + KL divergence losses summed over
all elements and batch
def loss_function(recon_x, x, mu, logvar):
    BCE = F.binary_cross_entropy(recon_x,
x.view(-1, 784), reduction='sum')

    # see Appendix B from VAE paper:
    # Kingma and Welling. Auto-Encoding Variational
Bayes. ICLR, 2014
    # https://arxiv.org/abs/1312.6114
    # 0.5 * sum(1 + log(sigma^2) - mu^2 - sigma^2)
```

```
    KLD = -0.5 * torch.sum(1 + logvar - mu.pow(2) -
logvar.exp())

    return BCE + KLD


def train(epoch):
    model.train()
    train_loss = 0
    for batch_idx, (data, _) in
enumerate(train_loader):
        data = data.to(device)
        optimizer.zero_grad()
        recon_batch, mu, logvar = model(data)
        loss = loss_function(recon_batch, data, mu,
logvar)
        loss.backward()
        train_loss += loss.item()
        optimizer.step()
        if batch_idx % args.log_interval == 0:
            print('Train Epoch: {} [{}/{} ({:.0f}%)]
\tLoss: {:.6f}'.format(
                epoch, batch_idx * len(data),
len(train_loader.dataset),
                100. * batch_idx /
len(train_loader),
                loss.item() / len(data)))

    print('====> Epoch: {} Average loss: {:.
4f}'.format(
          epoch, train_loss /
len(train_loader.dataset)))


def test(epoch):
    model.eval()
    test_loss = 0
```

```
    with torch.no_grad():
        for i, (data, _) in enumerate(test_loader):
            data = data.to(device)
            recon_batch, mu, logvar = model(data)
            test_loss += loss_function(recon_batch,
data, mu, logvar).item()
            if i == 0:
                n = min(data.size(0), 8)
                comparison = torch.cat([data[:n],

recon_batch.view(args.batch_size, 1, 28, 28)[:n]])
                save_image(comparison.cpu(),
                         'results/reconstruction_'
+ str(epoch) + '.png', nrow=n)

    test_loss /= len(test_loader.dataset)
    print('====> Test set loss: {:.
4f}'.format(test_loss))

if __name__ == "__main__":
    for epoch in range(1, args.epochs + 1):
        train(epoch)
        test(epoch)
        with torch.no_grad():
            sample = torch.randn(64, 20).to(device)
            sample = model.decode(sample).cpu()
            save_image(sample.view(64, 1, 28, 28),
                       'results/sample_' +
str(epoch) + '.png')
```

Superresolution using an efficient sub-pixel convolutional neural network

This example illustrates how to use the efficient sub-pixel convolution layer described in "Real-Time Single Image and Video Super-Resolution Using an Efficient Sub-Pixel Convolutional Neural Network" - Shi et al. for increasing spatial resolution within your network for tasks such as superresolution.

```
usage: main.py [-h] --upscale_factor UPSCALE_FACTOR [--batchSize BATCHSI
               [--testBatchSize TESTBATCHSIZE] [--nEpochs NEPOCHS] [--lr
               [--cuda] [--threads THREADS] [--seed SEED]

PyTorch Super Res Example

optional arguments:
  -h, --help            show this help message and exit
  --upscale_factor      super resolution upscale factor
  --batchSize           training batch size
  --testBatchSize       testing batch size
  --nEpochs             number of epochs to train for
  --lr                  Learning Rate. Default=0.01
  --cuda                use cuda
  --threads             number of threads for data loader to use Default
  --seed                random seed to use. Default=123
```

This example trains a super-resolution network on the BSD300 dataset, using crops from the `200` training images, and evaluating on crops of the `100` test images. A snapshot of the model after every epoch with filename model_epoch_.pth

Example Usage:

Train

```
python main.py --upscale_factor 3 --batchSize 4 --testBatchSize
100 --nEpochs 30 --lr 0.001
```

Super Resolve

```
python super_resolve.py --input_image dataset/BSDS300/images/
test/16077.jpg --model model_epoch_500.pth --output_filename
out.png
```

```
from os.path import exists, join, basename
from os import makedirs, remove
from six.moves import urllib
import tarfile
from torchvision.transforms import Compose,
CenterCrop, ToTensor, Resize

from dataset import DatasetFromFolder


def download_bsd300(dest="dataset"):
    output_image_dir = join(dest, "BSDS300/images")

    if not exists(output_image_dir):
        makedirs(dest)
        url = "http://www2.eecs.berkeley.edu/
Research/Projects/CS/vision/bsds/BSDS300-images.tgz"
        print("downloading url ", url)

        data = urllib.request.urlopen(url)

        file_path = join(dest, basename(url))
        with open(file_path, 'wb') as f:
            f.write(data.read())

        print("Extracting data")
        with tarfile.open(file_path) as tar:
            for item in tar:
                tar.extract(item, dest)

        remove(file_path)

    return output_image_dir


def calculate_valid_crop_size(crop_size,
upscale_factor):
```

```
    return crop_size - (crop_size % upscale_factor)


def input_transform(crop_size, upscale_factor):
    return Compose([
        CenterCrop(crop_size),
        Resize(crop_size // upscale_factor),
        ToTensor(),
    ])


def target_transform(crop_size):
    return Compose([
        CenterCrop(crop_size),
        ToTensor(),
    ])


def get_training_set(upscale_factor):
    root_dir = download_bsd300()
    train_dir = join(root_dir, "train")
    crop_size = calculate_valid_crop_size(256,
upscale_factor)

    return DatasetFromFolder(train_dir,

input_transform=input_transform(crop_size,
upscale_factor),

target_transform=target_transform(crop_size))


def get_test_set(upscale_factor):
    root_dir = download_bsd300()
    test_dir = join(root_dir, "test")
    crop_size = calculate_valid_crop_size(256,
upscale_factor)
```

```
    return DatasetFromFolder(test_dir,

input_transform=input_transform(crop_size,
upscale_factor),

target_transform=target_transform(crop_size))
import torch.utils.data as data

from os import listdir
from os.path import join
from PIL import Image


def is_image_file(filename):
    return any(filename.endswith(extension) for
extension in [".png", ".jpg", ".jpeg"])


def load_img(filepath):
    img = Image.open(filepath).convert('YCbCr')
    y, _, _ = img.split()
    return y


class DatasetFromFolder(data.Dataset):
    def __init__(self, image_dir,
input_transform=None, target_transform=None):
        super(DatasetFromFolder, self).__init__()
        self.image_filenames = [join(image_dir, x)
for x in listdir(image_dir) if is_image_file(x)]

        self.input_transform = input_transform
        self.target_transform = target_transform

    def __getitem__(self, index):
        input =
```

```
load_img(self.image_filenames[index])
        target = input.copy()
        if self.input_transform:
            input = self.input_transform(input)
        if self.target_transform:
            target = self.target_transform(target)

        return input, target

    def __len__(self):
        return len(self.image_filenames)
from __future__ import print_function
import argparse
from math import log10

import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader
from model import Net
from data import get_training_set, get_test_set

# Training settings
parser =
argparse.ArgumentParser(description='PyTorch Super
Res Example')
parser.add_argument('--upscale_factor', type=int,
required=True, help="super resolution upscale
factor")
parser.add_argument('--batchSize', type=int,
default=64, help='training batch size')
parser.add_argument('--testBatchSize', type=int,
default=10, help='testing batch size')
parser.add_argument('--nEpochs', type=int,
default=2, help='number of epochs to train for')
parser.add_argument('--lr', type=float,
default=0.01, help='Learning Rate. Default=0.01')
```

```
parser.add_argument('--cuda', action='store_true',
help='use cuda?')
parser.add_argument('--threads', type=int,
default=4, help='number of threads for data loader
to use')
parser.add_argument('--seed', type=int,
default=123, help='random seed to use. Default=123')
opt = parser.parse_args()

print(opt)

if opt.cuda and not torch.cuda.is_available():
    raise Exception("No GPU found, please run
without --cuda")

torch.manual_seed(opt.seed)

device = torch.device("cuda" if opt.cuda else "cpu")

print('===> Loading datasets')
train_set = get_training_set(opt.upscale_factor)
test_set = get_test_set(opt.upscale_factor)
training_data_loader =
DataLoader(dataset=train_set,
num_workers=opt.threads, batch_size=opt.batchSize,
shuffle=True)
testing_data_loader = DataLoader(dataset=test_set,
num_workers=opt.threads,
batch_size=opt.testBatchSize, shuffle=False)

print('===> Building model')
model =
Net(upscale_factor=opt.upscale_factor).to(device)
criterion = nn.MSELoss()

optimizer = optim.Adam(model.parameters(),
lr=opt.lr)
```

```
def train(epoch):
    epoch_loss = 0
    for iteration, batch in
enumerate(training_data_loader, 1):
        input, target = batch[0].to(device),
batch[1].to(device)

        optimizer.zero_grad()
        loss = criterion(model(input), target)
        epoch_loss += loss.item()
        loss.backward()
        optimizer.step()

        print("===> Epoch[{}]({}/{}): Loss: {:.
4f}".format(epoch, iteration,
len(training_data_loader), loss.item()))

    print("===> Epoch {} Complete: Avg. Loss: {:.
4f}".format(epoch, epoch_loss /
len(training_data_loader)))


def test():
    avg_psnr = 0
    with torch.no_grad():
        for batch in testing_data_loader:
            input, target = batch[0].to(device),
batch[1].to(device)

            prediction = model(input)
            mse = criterion(prediction, target)
            psnr = 10 * log10(1 / mse.item())
            avg_psnr += psnr
    print("===> Avg. PSNR: {:.4f}
dB".format(avg_psnr / len(testing_data_loader)))
```

```
def checkpoint(epoch):
    model_out_path =
"model_epoch_{}.pth".format(epoch)
    torch.save(model, model_out_path)
    print("Checkpoint saved to
{}".format(model_out_path))

for epoch in range(1, opt.nEpochs + 1):
    train(epoch)
    test()
    checkpoint(epoch)
import torch
import torch.nn as nn
import torch.nn.init as init


class Net(nn.Module):
    def __init__(self, upscale_factor):
        super(Net, self).__init__()

        self.relu = nn.ReLU()
        self.conv1 = nn.Conv2d(1, 64, (5, 5), (1,
1), (2, 2))
        self.conv2 = nn.Conv2d(64, 64, (3, 3), (1,
1), (1, 1))
        self.conv3 = nn.Conv2d(64, 32, (3, 3), (1,
1), (1, 1))
        self.conv4 = nn.Conv2d(32, upscale_factor
** 2, (3, 3), (1, 1), (1, 1))
        self.pixel_shuffle =
nn.PixelShuffle(upscale_factor)

        self._initialize_weights()

    def forward(self, x):
```

```
        x = self.relu(self.conv1(x))
        x = self.relu(self.conv2(x))
        x = self.relu(self.conv3(x))
        x = self.pixel_shuffle(self.conv4(x))
        return x

    def _initialize_weights(self):
        init.orthogonal_(self.conv1.weight,
init.calculate_gain('relu'))
        init.orthogonal_(self.conv2.weight,
init.calculate_gain('relu'))
        init.orthogonal_(self.conv3.weight,
init.calculate_gain('relu'))
        init.orthogonal_(self.conv4.weight)
from __future__ import print_function
import argparse
import torch
from PIL import Image
from torchvision.transforms import ToTensor

import numpy as np

# Training settings
parser =
argparse.ArgumentParser(description='PyTorch Super
Res Example')
parser.add_argument('--input_image', type=str,
required=True, help='input image to use')
parser.add_argument('--model', type=str,
required=True, help='model file to use')
parser.add_argument('--output_filename', type=str,
help='where to save the output image')
parser.add_argument('--cuda', action='store_true',
help='use cuda')
opt = parser.parse_args()

print(opt)
```

```
img = Image.open(opt.input_image).convert('YCbCr')
y, cb, cr = img.split()

model = torch.load(opt.model)
img_to_tensor = ToTensor()
input = img_to_tensor(y).view(1, -1, y.size[1],
y.size[0])

if opt.cuda:
    model = model.cuda()
    input = input.cuda()

out = model(input)
out = out.cpu()
out_img_y = out[0].detach().numpy()
out_img_y *= 255.0
out_img_y = out_img_y.clip(0, 255)
out_img_y = Image.fromarray(np.uint8(out_img_y[0]),
mode='L')

out_img_cb = cb.resize(out_img_y.size,
Image.BICUBIC)
out_img_cr = cr.resize(out_img_y.size,
Image.BICUBIC)
out_img = Image.merge('YCbCr', [out_img_y,
out_img_cb, out_img_cr]).convert('RGB')

out_img.save(opt.output_filename)
print('output image saved to ', opt.output_filename)
```

```
from __future__ import print_function
import argparse
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.multiprocessing as mp

from train import train, test

# Training settings
parser =
argparse.ArgumentParser(description='PyTorch MNIST
Example')
parser.add_argument('--batch-size', type=int,
default=64, metavar='N',
                    help='input batch size for
training (default: 64)')
parser.add_argument('--test-batch-size', type=int,
default=1000, metavar='N',
                    help='input batch size for
testing (default: 1000)')
parser.add_argument('--epochs', type=int,
default=10, metavar='N',
                    help='number of epochs to train
(default: 10)')
parser.add_argument('--lr', type=float,
default=0.01, metavar='LR',
                    help='learning rate (default:
0.01)')
parser.add_argument('--momentum', type=float,
default=0.5, metavar='M',
                    help='SGD momentum (default:
0.5)')
parser.add_argument('--seed', type=int, default=1,
metavar='S',
                    help='random seed (default: 1)')
parser.add_argument('--log-interval', type=int,
```

```
default=10, metavar='N',
                    help='how many batches to wait
before logging training status')
parser.add_argument('--num-processes', type=int,
default=2, metavar='N',
                    help='how many training
processes to use (default: 2)')
parser.add_argument('--cuda', action='store_true',
default=False,
                    help='enables CUDA training')

class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 10, kernel_size=5)
        self.conv2 = nn.Conv2d(10, 20,
kernel_size=5)
        self.conv2_drop = nn.Dropout2d()
        self.fc1 = nn.Linear(320, 50)
        self.fc2 = nn.Linear(50, 10)

    def forward(self, x):
        x = F.relu(F.max_pool2d(self.conv1(x), 2))
        x =
F.relu(F.max_pool2d(self.conv2_drop(self.conv2(x)),
2))
        x = x.view(-1, 320)
        x = F.relu(self.fc1(x))
        x = F.dropout(x, training=self.training)
        x = self.fc2(x)
        return F.log_softmax(x, dim=1)

if __name__ == '__main__':
    args = parser.parse_args()

    use_cuda = args.cuda and
torch.cuda.is_available()
```

```
    device = torch.device("cuda" if use_cuda else
"cpu")
    dataloader_kwargs = {'pin_memory': True} if
use_cuda else {}

    torch.manual_seed(args.seed)
    mp.set_start_method('spawn')

    model = Net().to(device)
    model.share_memory() # gradients are allocated
lazily, so they are not shared here

    processes = []
    for rank in range(args.num_processes):
        p = mp.Process(target=train, args=(rank,
args, model, device, dataloader_kwargs))
        # We first train the model across
`num_processes` processes
        p.start()
        processes.append(p)
    for p in processes:
        p.join()

    # Once training is complete, we can test the
model
    test(args, model, device, dataloader_kwargs)
import os
import torch
import torch.optim as optim
import torch.nn.functional as F
from torchvision import datasets, transforms


def train(rank, args, model, device,
dataloader_kwargs):
    torch.manual_seed(args.seed + rank)
```

```
    train_loader = torch.utils.data.DataLoader(
        datasets.MNIST('../data', train=True,
download=True,
                    transform=transforms.Compose([
                        transforms.ToTensor(),

transforms.Normalize((0.1307,), (0.3081,))
                    ])),
        batch_size=args.batch_size, shuffle=True,
num_workers=1,
        **dataloader_kwargs)

    optimizer = optim.SGD(model.parameters(),
lr=args.lr, momentum=args.momentum)
    for epoch in range(1, args.epochs + 1):
        train_epoch(epoch, args, model, device,
train_loader, optimizer)


def test(args, model, device, dataloader_kwargs):
    torch.manual_seed(args.seed)

    test_loader = torch.utils.data.DataLoader(
        datasets.MNIST('../data', train=False,
transform=transforms.Compose([
            transforms.ToTensor(),
            transforms.Normalize((0.1307,),
(0.3081,))
        ])),
        batch_size=args.batch_size, shuffle=True,
num_workers=1,
        **dataloader_kwargs)

    test_epoch(model, device, test_loader)


def train_epoch(epoch, args, model, device,
```

```
data_loader, optimizer):
    model.train()
    pid = os.getpid()
    for batch_idx, (data, target) in
enumerate(data_loader):
        optimizer.zero_grad()
        output = model(data.to(device))
        loss = F.nll_loss(output, target.to(device))
        loss.backward()
        optimizer.step()
        if batch_idx % args.log_interval == 0:
            print('{}\tTrain Epoch: {} [{}/{} ({:.
0f}%)]\tLoss: {:.6f}'.format(
                pid, epoch, batch_idx * len(data),
len(data_loader.dataset),
                100. * batch_idx /
len(data_loader), loss.item()))


def test_epoch(model, device, data_loader):
    model.eval()
    test_loss = 0
    correct = 0
    with torch.no_grad():
        for data, target in data_loader:
            output = model(data.to(device))
            test_loss += F.nll_loss(output,
target.to(device), reduction='sum').item() # sum up
batch loss
            pred = output.max(1)[1] # get the index
of the max log-probability
            correct +=
pred.eq(target.to(device)).sum().item()

    test_loss /= len(data_loader.dataset)
    print('\nTest set: Average loss: {:.4f},
Accuracy: {}/{} ({:.0f}%)\n'.format(
```

```
        test_loss, correct,
len(data_loader.dataset),
        100. * correct / len(data_loader.dataset)))
```

```
# Reinforcement learning training example

```bash
pip install -r requirements.txt
# For REINFORCE:
python reinforce.py
# For actor critic:
python actor_critic.py
```
torch
numpy
gym
import argparse
import gym
import numpy as np
from itertools import count

import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
from torch.distributions import Categorical

parser =
argparse.ArgumentParser(description='PyTorch
REINFORCE example')
parser.add_argument('--gamma', type=float,
default=0.99, metavar='G',
                    help='discount factor (default:
0.99)')
parser.add_argument('--seed', type=int,
default=543, metavar='N',
                    help='random seed (default:
543)')
parser.add_argument('--render', action='store_true',
                    help='render the environment')
```

```
parser.add_argument('--log-interval', type=int,
default=10, metavar='N',
                    help='interval between training
status logs (default: 10)')
args = parser.parse_args()


env = gym.make('CartPole-v1')
env.seed(args.seed)
torch.manual_seed(args.seed)


class Policy(nn.Module):
    def __init__(self):
        super(Policy, self).__init__()
        self.affine1 = nn.Linear(4, 128)
        self.dropout = nn.Dropout(p=0.6)
        self.affine2 = nn.Linear(128, 2)

        self.saved_log_probs = []
        self.rewards = []

    def forward(self, x):
        x = self.affine1(x)
        x = self.dropout(x)
        x = F.relu(x)
        action_scores = self.affine2(x)
        return F.softmax(action_scores, dim=1)


policy = Policy()
optimizer = optim.Adam(policy.parameters(), lr=1e-2)
eps = np.finfo(np.float32).eps.item()


def select_action(state):
    state =
```

```
torch.from_numpy(state).float().unsqueeze(0)
    probs = policy(state)
    m = Categorical(probs)
    action = m.sample()

policy.saved_log_probs.append(m.log_prob(action))
    return action.item()


def finish_episode():
    R = 0
    policy_loss = []
    returns = []
    for r in policy.rewards[::-1]:
        R = r + args.gamma * R
        returns.insert(0, R)
    returns = torch.tensor(returns)
    returns = (returns - returns.mean()) /
(returns.std() + eps)
    for log_prob, R in zip(policy.saved_log_probs,
returns):
        policy_loss.append(-log_prob * R)
    optimizer.zero_grad()
    policy_loss = torch.cat(policy_loss).sum()
    policy_loss.backward()
    optimizer.step()
    del policy.rewards[:]
    del policy.saved_log_probs[:]


def main():
    running_reward = 10
    for i_episode in count(1):
        state, ep_reward = env.reset(), 0
        for t in range(1, 10000):  # Don't infinite
loop while learning
            action = select_action(state)
```

```
            state, reward, done, _ =
env.step(action)
            if args.render:
                env.render()
            policy.rewards.append(reward)
            ep_reward += reward
            if done:
                break

        running_reward = 0.05 * ep_reward + (1 -
0.05) * running_reward
        finish_episode()
        if i_episode % args.log_interval == 0:
            print('Episode {}\tLast reward: {:.2f}
\tAverage reward: {:.2f}'.format(
                  i_episode, ep_reward,
running_reward))
        if running_reward >
env.spec.reward_threshold:
            print("Solved! Running reward is now {}
and "
                  "the last episode runs to {} time
steps!".format(running_reward, t))
            break


if __name__ == '__main__':
    main()
import argparse
import gym
import numpy as np
from itertools import count
from collections import namedtuple

import torch
import torch.nn as nn
import torch.nn.functional as F
```

```
import torch.optim as optim
from torch.distributions import Categorical

# Cart Pole

parser =
argparse.ArgumentParser(description='PyTorch actor-
critic example')
parser.add_argument('--gamma', type=float,
default=0.99, metavar='G',
                    help='discount factor (default:
0.99)')
parser.add_argument('--seed', type=int,
default=543, metavar='N',
                    help='random seed (default:
543)')
parser.add_argument('--render', action='store_true',
                    help='render the environment')
parser.add_argument('--log-interval', type=int,
default=10, metavar='N',
                    help='interval between training
status logs (default: 10)')
args = parser.parse_args()


env = gym.make('CartPole-v0')
env.seed(args.seed)
torch.manual_seed(args.seed)


SavedAction = namedtuple('SavedAction',
['log_prob', 'value'])


class Policy(nn.Module):
    """
    implements both actor and critic in one model
```

```
    """
    def __init__(self):
        super(Policy, self).__init__()
        self.affine1 = nn.Linear(4, 128)

        # actor's layer
        self.action_head = nn.Linear(128, 2)

        # critic's layer
        self.value_head = nn.Linear(128, 1)

        # action & reward buffer
        self.saved_actions = []
        self.rewards = []

    def forward(self, x):
        """
        forward of both actor and critic
        """
        x = F.relu(self.affine1(x))

        # actor: choses action to take from state
s_t
        # by returning probability of each action
        action_prob =
F.softmax(self.action_head(x), dim=-1)

        # critic: evaluates being in the state s_t
        state_values = self.value_head(x)

        # return values for both actor and critic
as a tupel of 2 values:
        # 1. a list with the probability of each
action over the action space
        # 2. the value from state s_t
        return action_prob, state_values
```

```
model = Policy()
optimizer = optim.Adam(model.parameters(), lr=3e-2)
eps = np.finfo(np.float32).eps.item()


def select_action(state):
    state = torch.from_numpy(state).float()
    probs, state_value = model(state)

    # create a categorical distribution over the
list of probabilities of actions
    m = Categorical(probs)

    # and sample an action using the distribution
    action = m.sample()

    # save to action buffer

model.saved_actions.append(SavedAction(m.log_prob(action),
state_value))

    # the action to take (left or right)
    return action.item()


def finish_episode():
    """
    Training code. Calcultes actor and critic loss
and performs backprop.
    """
    R = 0
    saved_actions = model.saved_actions
    policy_losses = [] # list to save actor
(policy) loss
    value_losses = [] # list to save critic (value)
loss
```

```
    returns = [] # list to save the true values

    # calculate the true value using rewards
returned from the environment
    for r in model.rewards[::-1]:
        # calculate the discounted value
        R = r + args.gamma * R
        returns.insert(0, R)

    returns = torch.tensor(returns)
    returns = (returns - returns.mean()) /
(returns.std() + eps)

    for (log_prob, value), R in zip(saved_actions,
returns):
        advantage = R - value.item()

        # calculate actor (policy) loss
        policy_losses.append(-log_prob * advantage)

        # calculate critic (value) loss using L1
smooth loss
        value_losses.append(F.smooth_l1_loss(value,
torch.tensor([R])))

    # reset gradients
    optimizer.zero_grad()

    # sum up all the values of policy_losses and
value_losses
    loss = torch.stack(policy_losses).sum() +
torch.stack(value_losses).sum()

    # perform backprop
    loss.backward()
    optimizer.step()
```

```
    # reset rewards and action buffer
    del model.rewards[:]
    del model.saved_actions[:]


def main():
    running_reward = 10

    # run inifinitely many episodes
    for i_episode in count(1):

        # reset environment and episode reward
        state = env.reset()
        ep_reward = 0

        # for each episode, only run 9999 steps so
that we don't
        # infinite loop while learning
        for t in range(1, 10000):

            # select action from policy
            action = select_action(state)

            # take the action
            state, reward, done, _ =
env.step(action)

            if args.render:
                env.render()

            model.rewards.append(reward)
            ep_reward += reward
            if done:
                break

        # update cumulative reward
        running_reward = 0.05 * ep_reward + (1 -
```

```
0.05) * running_reward

        # perform backprop
        finish_episode()

        # log results
        if i_episode % args.log_interval == 0:
            print('Episode {}\tLast reward: {:.2f}
\tAverage reward: {:.2f}'.format(
                  i_episode, ep_reward,
running_reward))

        # check if we have "solved" the cart pole
problem
        if running_reward >
env.spec.reward_threshold:
            print("Solved! Running reward is now {}
and "
                  "the last episode runs to {} time
steps!".format(running_reward, t))
            break


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

```
# Time Sequence Prediction
This is a toy example for beginners to start with.
It is helpful for learning both pytorch and time
sequence prediction. Two LSTMCell units are used in
this example to learn some sine wave signals
starting at different phases. After learning the
sine waves, the network tries to predict the signal
values in the future. The results is shown in the
picture below.

## Usage

```

python generate_sine_wave.py
python train.py
```


## Result
The initial signal and the predicted results are
shown in the image. We first give some initial
signals (full line). The network will  subsequently
give some predicted results (dash line). It can be
concluded that the network can generate new sine
waves.
![image](https://cloud.githubusercontent.com/assets/
1419566/24184438/
e24f5280-0f08-11e7-8f8b-4d972b527a81.png)
import numpy as np
import torch

np.random.seed(2)

T = 20
L = 1000
N = 100

x = np.empty((N, L), 'int64')
```

```
x[:] = np.array(range(L)) + np.random.randint(-4 *
T, 4 * T, N).reshape(N, 1)
data = np.sin(x / 1.0 / T).astype('float64')
torch.save(data, open('traindata.pt', 'wb'))
from __future__ import print_function
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np
import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt

class Sequence(nn.Module):
    def __init__(self):
        super(Sequence, self).__init__()
        self.lstm1 = nn.LSTMCell(1, 51)
        self.lstm2 = nn.LSTMCell(51, 51)
        self.linear = nn.Linear(51, 1)

    def forward(self, input, future = 0):
        outputs = []
        h_t = torch.zeros(input.size(0), 51,
dtype=torch.double)
        c_t = torch.zeros(input.size(0), 51,
dtype=torch.double)
        h_t2 = torch.zeros(input.size(0), 51,
dtype=torch.double)
        c_t2 = torch.zeros(input.size(0), 51,
dtype=torch.double)

        for i, input_t in
enumerate(input.chunk(input.size(1), dim=1)):
            h_t, c_t = self.lstm1(input_t, (h_t,
c_t))
            h_t2, c_t2 = self.lstm2(h_t, (h_t2,
c_t2))
```

```
            output = self.linear(h_t2)
            outputs += [output]
        for i in range(future):# if we should
predict the future
            h_t, c_t = self.lstm1(output, (h_t,
c_t))
            h_t2, c_t2 = self.lstm2(h_t, (h_t2,
c_t2))
            output = self.linear(h_t2)
            outputs += [output]
        outputs = torch.stack(outputs, 1).squeeze(2)
        return outputs


if __name__ == '__main__':
    # set random seed to 0
    np.random.seed(0)
    torch.manual_seed(0)
    # load data and make training set
    data = torch.load('traindata.pt')
    input = torch.from_numpy(data[3:, :-1])
    target = torch.from_numpy(data[3:, 1:])
    test_input = torch.from_numpy(data[:3, :-1])
    test_target = torch.from_numpy(data[:3, 1:])
    # build the model
    seq = Sequence()
    seq.double()
    criterion = nn.MSELoss()
    # use LBFGS as optimizer since we can load the
whole data to train
    optimizer = optim.LBFGS(seq.parameters(),
lr=0.8)
    #begin to train
    for i in range(15):
        print('STEP: ', i)
        def closure():
            optimizer.zero_grad()
```

```
            out = seq(input)
            loss = criterion(out, target)
            print('loss:', loss.item())
            loss.backward()
            return loss
        optimizer.step(closure)
        # begin to predict, no need to track
gradient here
        with torch.no_grad():
            future = 1000
            pred = seq(test_input, future=future)
            loss = criterion(pred[:, :-future],
test_target)
            print('test loss:', loss.item())
            y = pred.detach().numpy()
        # draw the result
        plt.figure(figsize=(30,10))
        plt.title('Predict future values for time
sequences\n(Dashlines are predicted values)',
fontsize=30)
        plt.xlabel('x', fontsize=20)
        plt.ylabel('y', fontsize=20)
        plt.xticks(fontsize=20)
        plt.yticks(fontsize=20)
        def draw(yi, color):
            plt.plot(np.arange(input.size(1)),
yi[:input.size(1)], color, linewidth = 2.0)
            plt.plot(np.arange(input.size(1),
input.size(1) + future), yi[input.size(1):], color
+ ':', linewidth = 2.0)
        draw(y[0], 'r')
        draw(y[1], 'g')
        draw(y[2], 'b')
        plt.savefig('predict%d.pdf'%i)
        plt.close()
```

fast-neural-style :city_sunrise: :rocket:
This repository contains a pytorch implementation of an algorithm for artistic style transfer. The algorithm can be used to mix the content of an image with the style of another image. For example, here is a photograph of a door arch rendered in the style of a stained glass painting.

The model uses the method described in [Perceptual Losses for Real-Time Style Transfer and Super-Resolution](https://arxiv.org/abs/1603.08155) along with [Instance Normalization](https://arxiv.org/pdf/1607.08022.pdf). The saved-models for examples shown in the README can be downloaded from [here](https://www.dropbox.com/s/lrvwfehqdcxoza8/saved_models.zip?dl=0).

<p align="center">
    <img src="images/style-images/mosaic.jpg" height="200px">
    <img src="images/content-images/amber.jpg" height="200px">
    <img src="images/output-images/amber-mosaic.jpg" height="440px">
</p>

Requirements
The program is written in Python, and uses [pytorch](http://pytorch.org/), [scipy](https://www.scipy.org). A GPU is not necessary, but can provide a significant speed up especially for training a new model. Regular sized images can be styled on a laptop or desktop using saved models.

Usage
Stylize image
```
```

```
python neural_style/neural_style.py eval --content-
image </path/to/content/image> --model </path/to/
saved/model> --output-image </path/to/output/image>
--cuda 0
```
* `--content-image`: path to content image you want to stylize.
* `--model`: saved model to be used for stylizing the image (eg: `mosaic.pth`)
* `--output-image`: path for saving the output image.
* `--content-scale`: factor for scaling down the content image if memory is an issue (eg: value of 2 will halve the height and width of content-image)
* `--cuda`: set it to 1 for running on GPU, 0 for CPU.

Train model
```bash
python neural_style/neural_style.py train --dataset
</path/to/train-dataset> --style-image </path/to/
style/image> --save-model-dir </path/to/save-model/
folder> --epochs 2 --cuda 1
```

There are several command line arguments, the important ones are listed below
* `--dataset`: path to training dataset, the path should point to a folder containing another folder with all the training images. I used COCO 2014 Training images dataset [80K/13GB] [(download)](http://mscoco.org/dataset/#download).
* `--style-image`: path to style-image.
* `--save-model-dir`: path to folder where trained model will be saved.
* `--cuda`: set it to 1 for running on GPU, 0 for CPU.

Refer to ``neural_style/neural_style.py`` for other command line arguments. For training new models you might have to tune the values of `--content-weight` and `--style-weight`. The mosaic style model shown above was trained with `--content-weight 1e5` and `--style-weight 1e10`. The remaining 3 models were also trained with similar order of weight parameters with slight variation in the `--style-weight` (`5e10` or `1e11`).

Models

Models for the examples shown below can be downloaded from [here](https://www.dropbox.com/s/lrvwfehqdcxoza8/saved_models.zip?dl=0) or by running the script ``download_saved_models.py``.

<div align='center'>
  <img src='images/content-images/amber.jpg' height="174px">
</div>

<div align='center'>
  <img src='images/style-images/mosaic.jpg' height="174px">
  <img src='images/output-images/amber-mosaic.jpg' height="174px">
  <img src='images/output-images/amber-candy.jpg' height="174px">
  <img src='images/style-images/candy.jpg' height="174px">

  <img src='images/style-images/rain-princess-cropped.jpg' height="174px">
  <img src='images/output-images/amber-rain-princess.jpg' height="174px">

```
  <img src='images/output-images/amber-udnie.jpg'
height="174px">
  <img src='images/style-images/udnie.jpg'
height="174px">
</div>
import os
import zipfile

# PyTorch 1.1 moves _download_url_to_file
#   from torch.utils.model_zoo to torch.hub
# PyTorch 1.0 exists another _download_url_to_file
#   2 argument
# TODO: If you remove support PyTorch 1.0 or older,
#       You should remove torch.utils.model_zoo
#       Ref. PyTorch #18758
#         https://github.com/pytorch/pytorch/pull/
18758/commits
try:
    from torch.utils.model_zoo import
_download_url_to_file
except ImportError:
    from torch.hub import _download_url_to_file


def unzip(source_filename, dest_dir):
    with zipfile.ZipFile(source_filename) as zf:
        zf.extractall(path=dest_dir)


if __name__ == '__main__':
    _download_url_to_file('https://www.dropbox.com/
s/lrvwfehqdcxoza8/saved_models.zip?dl=1',
'saved_models.zip', None, True)
    unzip('saved_models.zip', '.')
```

```
import argparse
import os
import sys
import time
import re

import numpy as np
import torch
from torch.optim import Adam
from torch.utils.data import DataLoader
from torchvision import datasets
from torchvision import transforms
import torch.onnx

import utils
from transformer_net import TransformerNet
from vgg import Vgg16


def check_paths(args):
    try:
        if not os.path.exists(args.save_model_dir):
            os.makedirs(args.save_model_dir)
        if args.checkpoint_model_dir is not None
and not (os.path.exists(args.checkpoint_model_dir)):
            os.makedirs(args.checkpoint_model_dir)
    except OSError as e:
        print(e)
        sys.exit(1)


def train(args):
    device = torch.device("cuda" if args.cuda else
"cpu")

    np.random.seed(args.seed)
    torch.manual_seed(args.seed)
```

```
    transform = transforms.Compose([
        transforms.Resize(args.image_size),
        transforms.CenterCrop(args.image_size),
        transforms.ToTensor(),
        transforms.Lambda(lambda x: x.mul(255))
    ])
    train_dataset =
datasets.ImageFolder(args.dataset, transform)
    train_loader = DataLoader(train_dataset,
batch_size=args.batch_size)

    transformer = TransformerNet().to(device)
    optimizer = Adam(transformer.parameters(),
args.lr)
    mse_loss = torch.nn.MSELoss()

    vgg = Vgg16(requires_grad=False).to(device)
    style_transform = transforms.Compose([
        transforms.ToTensor(),
        transforms.Lambda(lambda x: x.mul(255))
    ])
    style = utils.load_image(args.style_image,
size=args.style_size)
    style = style_transform(style)
    style = style.repeat(args.batch_size, 1, 1,
1).to(device)

    features_style =
vgg(utils.normalize_batch(style))
    gram_style = [utils.gram_matrix(y) for y in
features_style]

    for e in range(args.epochs):
        transformer.train()
        agg_content_loss = 0.
        agg_style_loss = 0.
```

```
        count = 0
        for batch_id, (x, _) in
enumerate(train_loader):
            n_batch = len(x)
            count += n_batch
            optimizer.zero_grad()

            x = x.to(device)
            y = transformer(x)

            y = utils.normalize_batch(y)
            x = utils.normalize_batch(x)

            features_y = vgg(y)
            features_x = vgg(x)

            content_loss = args.content_weight *
mse_loss(features_y.relu2_2, features_x.relu2_2)

            style_loss = 0.
            for ft_y, gm_s in zip(features_y,
gram_style):
                gm_y = utils.gram_matrix(ft_y)
                style_loss += mse_loss(gm_y,
gm_s[:n_batch, :, :])
            style_loss *= args.style_weight

            total_loss = content_loss + style_loss
            total_loss.backward()
            optimizer.step()

            agg_content_loss += content_loss.item()
            agg_style_loss += style_loss.item()

            if (batch_id + 1) % args.log_interval
== 0:
                mesg = "{}\tEpoch {}:\t[{}/{}]
```

```
\tcontent: {:.6f}\tstyle: {:.6f}\ttotal: {:.
6f}".format(
                    time.ctime(), e + 1, count,
len(train_dataset),

agg_content_loss / (batch_id + 1),
                                  agg_style_loss /
(batch_id + 1),
                                  (agg_content_loss
+ agg_style_loss) / (batch_id + 1)
                )
                print(mesg)

            if args.checkpoint_model_dir is not
None and (batch_id + 1) % args.checkpoint_interval
== 0:
                transformer.eval().cpu()
                ckpt_model_filename = "ckpt_epoch_"
+ str(e) + "_batch_id_" + str(batch_id + 1) + ".pth"
                ckpt_model_path =
os.path.join(args.checkpoint_model_dir,
ckpt_model_filename)

torch.save(transformer.state_dict(),
ckpt_model_path)
                transformer.to(device).train()

    # save model
    transformer.eval().cpu()
    save_model_filename = "epoch_" +
str(args.epochs) + "_" +
str(time.ctime()).replace(' ', '_') + "_" + str(
        args.content_weight) + "_" +
str(args.style_weight) + ".model"
    save_model_path =
os.path.join(args.save_model_dir,
save_model_filename)
```

```
    torch.save(transformer.state_dict(),
save_model_path)

    print("\nDone, trained model saved at",
save_model_path)


def stylize(args):
    device = torch.device("cuda" if args.cuda else
"cpu")

    content_image =
utils.load_image(args.content_image,
scale=args.content_scale)
    content_transform = transforms.Compose([
        transforms.ToTensor(),
        transforms.Lambda(lambda x: x.mul(255))
    ])
    content_image = content_transform(content_image)
    content_image =
content_image.unsqueeze(0).to(device)

    if args.model.endswith(".onnx"):
        output = stylize_onnx_caffe2(content_image,
args)
    else:
        with torch.no_grad():
            style_model = TransformerNet()
            state_dict = torch.load(args.model)
            # remove saved deprecated running_*
keys in InstanceNorm from the checkpoint
            for k in list(state_dict.keys()):
                if re.search(r'in\d+\.running_(mean|
var)$', k):
                    del state_dict[k]
            style_model.load_state_dict(state_dict)
            style_model.to(device)
```

```
            if args.export_onnx:
                assert
args.export_onnx.endswith(".onnx"), "Export model
file should end with .onnx"
                output =
torch.onnx._export(style_model, content_image,
args.export_onnx).cpu()
            else:
                output =
style_model(content_image).cpu()
    utils.save_image(args.output_image, output[0])


def stylize_onnx_caffe2(content_image, args):
    """
    Read ONNX model and run it using Caffe2
    """

    assert not args.export_onnx

    import onnx
    import onnx_caffe2.backend

    model = onnx.load(args.model)

    prepared_backend =
onnx_caffe2.backend.prepare(model, device='CUDA' if
args.cuda else 'CPU')
    inp = {model.graph.input[0].name:
content_image.numpy()}
    c2_out = prepared_backend.run(inp)[0]

    return torch.from_numpy(c2_out)


def main():
    main_arg_parser =
```

```
argparse.ArgumentParser(description="parser for
fast-neural-style")
    subparsers =
main_arg_parser.add_subparsers(title="subcommands",
dest="subcommand")

    train_arg_parser =
subparsers.add_parser("train", help="parser for
training arguments")
    train_arg_parser.add_argument("--epochs",
type=int, default=2,
                                  help="number of
training epochs, default is 2")
    train_arg_parser.add_argument("--batch-size",
type=int, default=4,
                                  help="batch size
for training, default is 4")
    train_arg_parser.add_argument("--dataset",
type=str, required=True,
                                  help="path to
training dataset, the path should point to a folder
"
                                       "containing
another folder with all the training images")
    train_arg_parser.add_argument("--style-image",
type=str, default="images/style-images/mosaic.jpg",
                                  help="path to
style-image")
    train_arg_parser.add_argument("--save-model-
dir", type=str, required=True,
                                  help="path to
folder where trained model will be saved.")
    train_arg_parser.add_argument("--checkpoint-
model-dir", type=str, default=None,
                                  help="path to
folder where checkpoints of trained models will be
saved")
```

```
    train_arg_parser.add_argument("--image-size",
type=int, default=256,
                                  help="size of
training images, default is 256 X 256")
    train_arg_parser.add_argument("--style-size",
type=int, default=None,
                                  help="size of
style-image, default is the original size of style
image")
    train_arg_parser.add_argument("--cuda",
type=int, required=True,
                                  help="set it to 1
for running on GPU, 0 for CPU")
    train_arg_parser.add_argument("--seed",
type=int, default=42,
                                  help="random seed
for training")
    train_arg_parser.add_argument("--content-
weight", type=float, default=1e5,
                                  help="weight for
content-loss, default is 1e5")
    train_arg_parser.add_argument("--style-weight",
type=float, default=1e10,
                                  help="weight for
style-loss, default is 1e10")
    train_arg_parser.add_argument("--lr",
type=float, default=1e-3,
                                  help="learning
rate, default is 1e-3")
    train_arg_parser.add_argument("--log-interval",
type=int, default=500,
                                  help="number of
images after which the training loss is logged,
default is 500")
    train_arg_parser.add_argument("--checkpoint-
interval", type=int, default=2000,
                                  help="number of
```

```
batches after which a checkpoint of the trained
model will be created")

    eval_arg_parser = subparsers.add_parser("eval",
help="parser for evaluation/stylizing arguments")
    eval_arg_parser.add_argument("--content-image",
type=str, required=True,
                                 help="path to
content image you want to stylize")
    eval_arg_parser.add_argument("--content-scale",
type=float, default=None,
                                 help="factor for
scaling down the content image")
    eval_arg_parser.add_argument("--output-image",
type=str, required=True,
                                 help="path for
saving the output image")
    eval_arg_parser.add_argument("--model",
type=str, required=True,
                                 help="saved model
to be used for stylizing the image. If file ends
in .pth - PyTorch path is used, if in .onnx -
Caffe2 path")
    eval_arg_parser.add_argument("--cuda",
type=int, required=True,
                                 help="set it to 1
for running on GPU, 0 for CPU")
    eval_arg_parser.add_argument("--export_onnx",
type=str,
                                 help="export ONNX
model to a given file")

    args = main_arg_parser.parse_args()

    if args.subcommand is None:
        print("ERROR: specify either train or eval")
        sys.exit(1)
```

```
    if args.cuda and not torch.cuda.is_available():
        print("ERROR: cuda is not available, try
running on CPU")
        sys.exit(1)

    if args.subcommand == "train":
        check_paths(args)
        train(args)
    else:
        stylize(args)


if __name__ == "__main__":
    main()
import torch


class TransformerNet(torch.nn.Module):
    def __init__(self):
        super(TransformerNet, self).__init__()
        # Initial convolution layers
        self.conv1 = ConvLayer(3, 32,
kernel_size=9, stride=1)
        self.in1 = torch.nn.InstanceNorm2d(32,
affine=True)
        self.conv2 = ConvLayer(32, 64,
kernel_size=3, stride=2)
        self.in2 = torch.nn.InstanceNorm2d(64,
affine=True)
        self.conv3 = ConvLayer(64, 128,
kernel_size=3, stride=2)
        self.in3 = torch.nn.InstanceNorm2d(128,
affine=True)
        # Residual layers
        self.res1 = ResidualBlock(128)
        self.res2 = ResidualBlock(128)
        self.res3 = ResidualBlock(128)
```

```
        self.res4 = ResidualBlock(128)
        self.res5 = ResidualBlock(128)
        # Upsampling Layers
        self.deconv1 = UpsampleConvLayer(128, 64,
kernel_size=3, stride=1, upsample=2)
        self.in4 = torch.nn.InstanceNorm2d(64,
affine=True)
        self.deconv2 = UpsampleConvLayer(64, 32,
kernel_size=3, stride=1, upsample=2)
        self.in5 = torch.nn.InstanceNorm2d(32,
affine=True)
        self.deconv3 = ConvLayer(32, 3,
kernel_size=9, stride=1)
        # Non-linearities
        self.relu = torch.nn.ReLU()

    def forward(self, X):
        y = self.relu(self.in1(self.conv1(X)))
        y = self.relu(self.in2(self.conv2(y)))
        y = self.relu(self.in3(self.conv3(y)))
        y = self.res1(y)
        y = self.res2(y)
        y = self.res3(y)
        y = self.res4(y)
        y = self.res5(y)
        y = self.relu(self.in4(self.deconv1(y)))
        y = self.relu(self.in5(self.deconv2(y)))
        y = self.deconv3(y)
        return y


class ConvLayer(torch.nn.Module):
    def __init__(self, in_channels, out_channels,
kernel_size, stride):
        super(ConvLayer, self).__init__()
        reflection_padding = kernel_size // 2
        self.reflection_pad =
```

```
torch.nn.ReflectionPad2d(reflection_padding)
        self.conv2d = torch.nn.Conv2d(in_channels,
out_channels, kernel_size, stride)

    def forward(self, x):
        out = self.reflection_pad(x)
        out = self.conv2d(out)
        return out


class ResidualBlock(torch.nn.Module):
    """ResidualBlock
    introduced in: https://arxiv.org/abs/1512.03385
    recommended architecture: http://torch.ch/blog/
2016/02/04/resnets.html
    """

    def __init__(self, channels):
        super(ResidualBlock, self).__init__()
        self.conv1 = ConvLayer(channels, channels,
kernel_size=3, stride=1)
        self.in1 =
torch.nn.InstanceNorm2d(channels, affine=True)
        self.conv2 = ConvLayer(channels, channels,
kernel_size=3, stride=1)
        self.in2 =
torch.nn.InstanceNorm2d(channels, affine=True)
        self.relu = torch.nn.ReLU()

    def forward(self, x):
        residual = x
        out = self.relu(self.in1(self.conv1(x)))
        out = self.in2(self.conv2(out))
        out = out + residual
        return out
```

```
class UpsampleConvLayer(torch.nn.Module):
    """UpsampleConvLayer
    Upsamples the input and then does a
convolution. This method gives better results
    compared to ConvTranspose2d.
    ref: http://distill.pub/2016/deconv-
checkerboard/
    """

    def __init__(self, in_channels, out_channels,
kernel_size, stride, upsample=None):
        super(UpsampleConvLayer, self).__init__()
        self.upsample = upsample
        reflection_padding = kernel_size // 2
        self.reflection_pad =
torch.nn.ReflectionPad2d(reflection_padding)
        self.conv2d = torch.nn.Conv2d(in_channels,
out_channels, kernel_size, stride)

    def forward(self, x):
        x_in = x
        if self.upsample:
            x_in =
torch.nn.functional.interpolate(x_in,
mode='nearest', scale_factor=self.upsample)
        out = self.reflection_pad(x_in)
        out = self.conv2d(out)
        return out
import torch
from PIL import Image


def load_image(filename, size=None, scale=None):
    img = Image.open(filename)
    if size is not None:
        img = img.resize((size, size),
Image.ANTIALIAS)
```

```
    elif scale is not None:
        img = img.resize((int(img.size[0] / scale),
int(img.size[1] / scale)), Image.ANTIALIAS)
    return img


def save_image(filename, data):
    img = data.clone().clamp(0, 255).numpy()
    img = img.transpose(1, 2, 0).astype("uint8")
    img = Image.fromarray(img)
    img.save(filename)


def gram_matrix(y):
    (b, ch, h, w) = y.size()
    features = y.view(b, ch, w * h)
    features_t = features.transpose(1, 2)
    gram = features.bmm(features_t) / (ch * h * w)
    return gram


def normalize_batch(batch):
    # normalize using imagenet mean and std
    mean = batch.new_tensor([0.485, 0.456,
0.406]).view(-1, 1, 1)
    std = batch.new_tensor([0.229, 0.224,
0.225]).view(-1, 1, 1)
    batch = batch.div_(255.0)
    return (batch - mean) / std
from collections import namedtuple

import torch
from torchvision import models


class Vgg16(torch.nn.Module):
    def __init__(self, requires_grad=False):
```

```
        super(Vgg16, self).__init__()
        vgg_pretrained_features = 
models.vgg16(pretrained=True).features
        self.slice1 = torch.nn.Sequential()
        self.slice2 = torch.nn.Sequential()
        self.slice3 = torch.nn.Sequential()
        self.slice4 = torch.nn.Sequential()
        for x in range(4):
            self.slice1.add_module(str(x), 
vgg_pretrained_features[x])
        for x in range(4, 9):
            self.slice2.add_module(str(x), 
vgg_pretrained_features[x])
        for x in range(9, 16):
            self.slice3.add_module(str(x), 
vgg_pretrained_features[x])
        for x in range(16, 23):
            self.slice4.add_module(str(x), 
vgg_pretrained_features[x])
        if not requires_grad:
            for param in self.parameters():
                param.requires_grad = False

    def forward(self, X):
        h = self.slice1(X)
        h_relu1_2 = h
        h = self.slice2(h)
        h_relu2_2 = h
        h = self.slice3(h)
        h_relu3_3 = h
        h = self.slice4(h)
        h_relu4_3 = h
        vgg_outputs = namedtuple("VggOutputs", 
['relu1_2', 'relu2_2', 'relu3_3', 'relu4_3'])
        out = vgg_outputs(h_relu1_2, h_relu2_2, 
h_relu3_3, h_relu4_3)
        return out
```