```
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)))
```