our-cifar

December 21,2021

[1]:
```
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.utils.data import DataLoader, Dataset
import syft as sy
import copy
import numpy as np
import time
import random
import warnings

import importlib
importlib.import_module('FLDataset')
from statsmodels.tsa.holtwinters import ExponentialSmoothing,␣
↪SimpleExpSmoothing, Holt
from sklearn import preprocessing
from FLDataset import load_dataset, getActualImgs, load_dataset_cifar
from utils import averageModels
```

[2]:
```
# !pip install statsmodels
```

[3]:
```
class Arguments():
    def __init__(self):
        self.images = 60000
        self.clients = 40
        self.rounds = 30
        self.epochs = 10
        self.local_batches = 64
        self.lr = 0.01
        self.C = 0.9
        self.drop_rate = 0.1
        self.torch_seed = 0
        self.log_interval = 10
        self.iid = 'iid'
        self.split_size = int(self.images / self.clients)
```

```
        self.samples = self.split_size / self.images
        self.use_cuda = False
        self.save_model = False
        self.step = 2
        self.count = 0

args = Arguments()

use_cuda = args.use_cuda and torch.cuda.is_available()
device = torch.device("cuda" if use_cuda else "cpu")
kwargs = {'num_workers': 1, 'pin_memory': True} if use_cuda else {}
```

[4]:
```
hook = sy.TorchHook(torch)
clients = []

for i in range(args.clients):
    clients.append({'hook': sy.VirtualWorker(hook, id="client{}".format(i+1))})
```

[5]:
```
global_train, global_test, train_group, test_group = load_dataset_cifar(args.
 ↪clients, args.iid)
```

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

[6]:
```
for inx, client in enumerate(clients):
    trainset_ind_list = list(train_group[inx])
    client['trainset'] = getActualImgs(global_train, trainset_ind_list, args.
 ↪local_batches)
    client['testset'] = getActualImgs(global_test, list(test_group[inx]), args.
 ↪local_batches)
    client['samples'] = len(trainset_ind_list) / args.images
```

[7]:
```
trans_cifar_test = transforms.Compose([
            transforms.ToTensor(),
            transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.
 ↪2010)),
        ])
global_test_dataset = datasets.CIFAR10('./', train=False, download=True,␣
 ↪transform=trans_cifar_test)
global_test_loader = DataLoader(global_test_dataset, batch_size=args.
 ↪local_batches, shuffle=True)
```

```
Files already downloaded and verified
```

[8]:
```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
```

```
#         self.conv1 = nn.Conv2d(1, 20, 5, 1)
#         self.conv2 = nn.Conv2d(20, 50, 5, 1)
#         self.fc1 = nn.Linear(4*4*50, 500)
#         self.fc2 = nn.Linear(500, 10)
        self.conv1 = nn.Conv2d(3, 64, 5)
        self.pool = nn.MaxPool2d(3, 2)
        self.conv2 = nn.Conv2d(64, 64, 5)
        self.fc1 = nn.Linear(64 * 4 * 4, 384)
        self.fc2 = nn.Linear(384, 192)
        self.fc3 = nn.Linear(192, 10)

    def forward(self, x):
#         x = F.relu(self.conv1(x))
#         x = F.max_pool2d(x, 2, 2)
#         x = F.relu(self.conv2(x))
#         x = F.max_pool2d(x, 2, 2)
#         x = x.view(-1, 4*4*50)
#         x = F.relu(self.fc1(x))
#         x = self.fc2(x)
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        #x = x.view(-1, 16 * 5 * 5)
        x = x.view(-1, 64 * 4 * 4)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return F.log_softmax(x, dim=1)
```

[9]:
```
def ClientUpdate(args, device, client):
    client['model'].train()
    client['model'].send(client['hook'])
    epochs = args.epochs + 1+ lc[y].item()
    epochs = int(epochs)
    for epoch in range(1, epochs):
        for batch_idx, (data, target) in enumerate(client['trainset']):
            data = data.send(client['hook'])
            target = target.send(client['hook'])

            data, target = data.to(device), target.to(device)
            client['optim'].zero_grad()
            output = client['model'](data)
            loss = F.nll_loss(output, target)
            loss.backward()
            client['optim'].step()

#             if batch_idx % args.log_interval == 0:
#                 loss = loss.get()
```

```
#                 print('Model {} Train Epoch: {} [{}/{} ({:.0f}%)]\tLoss: {:.
↪6f}'.format(
#                     client['hook'].id,
#                     epoch, batch_idx * args.local_batches,␣
↪len(client['trainset']) * args.local_batches,
#                     100. * batch_idx / len(client['trainset']), loss))

    client['model'].get()
```

[10]:
```
def test(args, model, device, test_loader, name, rounds):
    model.eval()
    test_loss = 0
    correct = 0
    accuracy = 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(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)
    accuracy = 100. *correct / len(test_loader.dataset)

#     print('\nTest set{}: Average loss for {} model: {:.4f}, Accuracy: {}/{}␣
↪({:.0f}%)\n'.format(rounds,
#         name, test_loss, correct, len(test_loader.dataset),100. * correct /␣
↪len(test_loader.dataset)))
    return accuracy,test_loss
```

[11]:
```
def averageModels1(global_model, clients):
    client_models = [clients[i]['model'] for i in range(len(clients))]
    samples = [clients[i]['samples'] for i in range(len(clients))]
    global_dict = global_model.state_dict()
    for k in global_dict.keys():
        global_dict[k] = torch.stack([client_models[i].state_dict()[k].float()␣
↪* count2 for i in range(len(client_models))], 0).sum(0)

    global_model.load_state_dict(global_dict)
    return global_model
```

[12]:
```
def normalization(vec):
    maxVec = max(vec)
```

```
    minVec = min(vec)
    vecM=()
    vecS=()
    for i in vec:
        vecN = (i-minVec)/(maxVec-minVec)
        vecM = vecM+(vecN,)
    for k in vecM:
        d = (k - 0.5)/2
        vecS = vecS+(d,)
        vecS = vecS[:8]
    return vecS
```

[13]:
```
def detect_outliers(data1,data2):
    threshold=3
    mean_d = np.mean(data1)
    std_d = np.std(data1)
    outliers = ()

    for y in data2:
        z_score= (y - mean_d)/std_d
        if np.abs(z_score) > threshold:
            outliers = outliers + (y,)
    return outliers
```

[14]:
```
def time_since(since):
#     torch.cuda.synchronize()
    s = time.time() - since
    m = math.floor(s / 60)
    s -= m*60
    print('%dm %ds' % (m,s))
    return '%dm %ds' % (m,s)
```

[15]:
```
warnings.filterwarnings("ignore")
torch.manual_seed(args.torch_seed)
global_model = Net()
global_model1 = Net()
x = torch.full([args.clients], 0)
lc = torch.full([args.clients], 0)
# step = args.step
acc_list = torch.full([args.rounds,args.clients], 0)
gl_list = torch.full([args.rounds], 0)
l_list = torch.full([args.rounds], 0)
lqd_list = torch.full([args.rounds], 0)
lb_list = torch.full([args.rounds], 0)
loss_list = torch.full([args.rounds], 0)

for client in clients:
```

```
    torch.manual_seed(args.torch_seed)
    client['model'] = Net().to(device)
    client['optim'] = optim.SGD(client['model'].parameters(), lr=args.lr)

for fed_round in range(args.rounds):

    # number of selected clients
    m = int(max(args.C * args.clients, 1))

    # Selected devices
    np.random.seed(fed_round)
    selected_clients_inds = np.random.choice(range(len(clients)), m,␣
↪replace=False)
    selected_clients = [clients[i] for i in selected_clients_inds]

    # Active devices
    np.random.seed(fed_round)
    active_clients_inds = np.random.choice(selected_clients_inds, int((1-args.
↪drop_rate) * m), replace=False)
    active_clients = [clients[i] for i in active_clients_inds]
    fail_inds = ()
    for s in range(0,args.clients):
        n = 0
        for d in range(len(active_clients_inds)):
            if(active_clients_inds[d]==s):
                n += 1
        if(n<1):
            fail_inds = fail_inds+(s,)

    # Training
    up = 0
    for client in active_clients:
        y = active_clients_inds[up].item()
        ClientUpdate(args, device, client)
        x[y] = x[y]+args.epochs+lc[y]
        up += 1

    # Testing
    Acc = ()
    u = 0
    for client in active_clients:
        accuracy,test_loss = test(args, client['model'], device,␣
↪client['testset'], client['hook'].id, fed_round)
        t = active_clients_inds[u].item()
        acc_list[fed_round][t] = accuracy
        Acc = Acc +(accuracy,)
        u += 1
```

```
#     #
#     if(fed_round>=3):
#         acc = ()
#         for xz in range(len(Acc)):
#             shuju = np.zeros(fed_round+1)
#             t = active_clients_inds[xz].item()
#             for xc in range(fed_round+1):
#                 shuju[xc] = acc_list[xc][t]
#             fit3 = SimpleExpSmoothing(shuju).fit(smoothing_level=0.
 ↪6,optimized=False)
#             fit4 = fit3.forecast(1).item()
#             acc = acc +(fit4,)
# #         print(acc)

#     #
#     if(fed_round>=1):
#         z = normalization(acc)
#     else:
#         z = normalization(Acc)
# #     print(z)

#     #
#     for a in range(len(Acc)):
#         l = active_clients_inds[a]
#         if(args.epochs+lc[l]>=5):
#             lc[l] = lc[l]-int(z[a]*step)
#         else:
#             lc[l] = lc[l]
#     data_middle = sorted(z)
#     count = 0
#     #
# #     for i in range((len(z)/4-1),(len(z)/4*3+1)):
#     for i in range(6,18):
#         ziji[count] = data_middle[i]
#         count += 1

    # Averaging
    global_model = averageModels(global_model, active_clients)
    # Testing the average model
    gl_acc,gl_loss =  test(args, global_model, device, global_test_loader,␣
 ↪'Global', fed_round)
    #
    a1 = sorted(Acc)
    count1 = 0
    for a2 in range(len(a1)):
        if(a1[a2]>=gl_acc):
```

```
            count1 +=1
    if(count1 <10):
        count1 = 10

    if(count1>=16):
        count1 = 16
#         if(fed_round+1 ==args.rounds):
#             print(x)

#         print('\nRound{} set: Average loss for Global model:  Accuracy: ({:.
 ↪1f}%)\n'.format(fed_round+1,gl_acc))
#         for client in clients:
#             client['model'].load_state_dict(global_model.state_dict())
    else:
        count2 = (1/(count1*2))
        acc1 =()
        for a3 in range((len(a1)-count1*2),len(a1)):
            acc1 = acc1+(a1[a3],)
        acc2 = list(acc1)
        Acc1 = list(Acc)
    #     print(count1)
    #     print(len(acc2))
    #     print(acc2)
        for a4 in range(len(acc2)):
            for a5 in range(len(Acc)):
                if (acc2[a4]==Acc1[a5]):
                    acc2[a4] = a5
                    Acc1[a5] = 0
                    break
    #     print(acc2)
        acc3 = ()
        for a6 in range(len(acc1)):
            l = active_clients_inds[acc2[a6]]
            acc3 = acc3+(l,)
        ac_b=()
        for a2 in range(len(Acc)):
            if(Acc[a2]>=gl_acc):
                ac_b = ac_b+(Acc[a2],)

        sub_z=()
        z_z=()
        ac_in=()
        for ac in ac_b:
            for i in range(len(Acc)):
                if(ac==Acc[i]):
                    sub_z = sub_z+(Acc[i],)
                    ac_in = ac_in+(i,)
```

```
            else:
                z_z = z_z+(Acc[i],)
    sz = ()
    z = ()
    aci = ()
    for u in sub_z:
        if u not in sz:
            sz = sz+(u,)
    for d in z_z:
        if d not in z:
            z = z+(d,)
    for ii in ac_in:
        if ii not in aci:
            aci = aci+(ii,)
    #
    lqd = detect_outliers(sz,z)
    if lqd == torch.Size([]):
        lqd_list[fed_round] = 0
    else:
        lqd_list[fed_round] = len(lqd)
    #
    i_d = ()
    inds = ()
    for a in lqd:
        for i in range(len(Acc)):
            if(a==Acc[i]):
                i_d = i_d+(i,)
    for d in i_d:
        if d not in inds:
            inds = inds+(d,)
    for s in inds:
        l = active_clients_inds[s]
        lc[l] = -2
    for i1 in aci:
        l = active_clients_inds[i1]
        lc[l] = 2
    #
    if(fed_round>=3):
        lb = random.randint(2,6)
        Z = np.mat([i for i in range(500)])
        X = np.mat([[0,], [0,]])
        P = np.mat([[1, 0], [0, 1]])
        F = np.mat([[1, 1], [0, 1]])
        Q = np.mat([[0.0001, 0], [0, 0.0001]])
        H = np.mat([1, 0])
        R = np.mat([1])
        for i in range(100):
```

```
                x_predict = F * X
                p_predict = F * P * F.T + Q
                K = p_predict * H.T / (H * p_predict * H.T + R)
                X = x_predict + K *(Z[0, i] - H * x_predict)
                P = (np.eye(2) - K * H) * p_predict
            lb_list[fed_round] = lb



        new_clients = [clients[i] for i in acc3]
        global_model1 = averageModels1(global_model1, new_clients)
        gl_acc1,gl_loss1 =  test(args, global_model1, device,␣
↪global_test_loader, 'Global', fed_round)

        gl_list[fed_round] = gl_acc
        loss_list[fed_round] = gl_loss
        print('\nRound{} set: Average loss for Global model: Accuracy: ({:.
↪1f}%) Loss: ({:.4f}) Outliers: {} Filters: {}\n'.
↪format(fed_round+1,gl_acc,loss_list[fed_round],lqd_list[fed_round],lb_list[fed_round]))
        if(fed_round+1 ==args.rounds):
            print(x)
            print(gl_list)
        for  a in range(len(fail_inds)):
            f = fail_inds[a]
            acc_list[fed_round][f] = gl_acc

        # Share the global model with the clients
        for client in clients:
            client['model'].load_state_dict(global_model1.state_dict())

if (args.save_model):

    torch.save(global_model1.state_dict(), "FedAvg.pt")
```

```
Round1 set: Average loss for Global model: Accuracy: (21.7%) Loss: (2.2724)
Outliers: 8.0 Filters: 3.0


Round2 set: Average loss for Global model: Accuracy: (29.3%) Loss: (2.2161)
Outliers: 5.0 Filters: 4.0


Round3 set: Average loss for Global model: Accuracy: (33.6%) Loss: (2.1783)
Outliers: 11.0 Filters: 2.0
```

```
Round4 set: Average loss for Global model: Accuracy: (36.7%) Loss: (2.1476)
Outliers: 10.0 Filters: 3.0

Round5 set: Average loss for Global model: Accuracy: (40.2%) Loss: (2.1239)
Outliers: 10.0 Filters: 2.0

Round6 set: Average loss for Global model: Accuracy: (42.6%) Loss: (2.1037)
Outliers: 15.0 Filters: 5.0

Round7 set: Average loss for Global model: Accuracy: (43.5%) Loss: (2.0891)
Outliers: 10.0 Filters: 3.0

Round8 set: Average loss for Global model: Accuracy: (45.8%) Loss: (2.0763)
Outliers: 11.0 Filters: 5.0

Round9 set: Average loss for Global model: Accuracy: (46.7%) Loss: (2.0648)
Outliers: 12.0 Filters: 6.0

Round10 set: Average loss for Global model: Accuracy: (48.4%) Loss: (2.0506)
Outliers: 15.0 Filters: 3.0

Round11 set: Average loss for Global model: Accuracy: (48.8%) Loss: (2.0392)
Outliers: 14.0 Filters: 4.0

Round12 set: Average loss for Global model: Accuracy: (50.1%) Loss: (2.0284)
Outliers: 13.0 Filters: 6.0

Round13 set: Average loss for Global model: Accuracy: (50.0%) Loss: (2.0175)
Outliers: 11.0 Filters: 6.0

Round14 set: Average loss for Global model: Accuracy: (51.7%) Loss: (2.0061)
Outliers: 13.0 Filters: 5.0

Round15 set: Average loss for Global model: Accuracy: (53.5%) Loss: (1.9942)
Outliers: 17.0 Filters: 5.0
```

```
Round16 set: Average loss for Global model: Accuracy: (52.4%) Loss: (1.9882)
Outliers: 6.0 Filters: 3.0


Round17 set: Average loss for Global model: Accuracy: (53.4%) Loss: (1.9756)
Outliers: 11.0 Filters: 2.0


Round18 set: Average loss for Global model: Accuracy: (54.8%) Loss: (1.9643)
Outliers: 11.0 Filters: 2.0


Round19 set: Average loss for Global model: Accuracy: (54.9%) Loss: (1.9543)
Outliers: 8.0 Filters: 6.0


Round21 set: Average loss for Global model: Accuracy: (56.5%) Loss: (1.9301)
Outliers: 3.0 Filters: 3.0


Round23 set: Average loss for Global model: Accuracy: (58.1%) Loss: (1.9090)
Outliers: 12.0 Filters: 4.0


Round24 set: Average loss for Global model: Accuracy: (58.3%) Loss: (1.9128)
Outliers: 5.0 Filters: 6.0


Round25 set: Average loss for Global model: Accuracy: (59.4%) Loss: (1.8990)
Outliers: 8.0 Filters: 5.0


Round26 set: Average loss for Global model: Accuracy: (60.5%) Loss: (1.8898)
Outliers: 9.0 Filters: 4.0


Round28 set: Average loss for Global model: Accuracy: (61.5%) Loss: (1.8642)
Outliers: 12.0 Filters: 2.0


Round29 set: Average loss for Global model: Accuracy: (60.9%) Loss: (1.8704)
Outliers: 7.0 Filters: 2.0


Round30 set: Average loss for Global model: Accuracy: (61.8%) Loss: (1.8628)
Outliers: 4.0 Filters: 6.0

tensor([186., 230., 190., 196., 184., 210., 278., 234., 202., 178., 218., 230.,
```

```
        258., 246., 266., 198., 222., 208., 222., 134., 278., 266., 206., 194.,
        270., 274., 206., 222., 230., 270., 270., 210., 222., 246., 238., 226.,
        270., 230., 228., 246.])
tensor([21.7000, 29.3100, 33.6300, 36.6700, 40.2200, 42.5800, 43.5300, 45.8200,
        46.6800, 48.3500, 48.7500, 50.0600, 50.0300, 51.7100, 53.5200, 52.3900,
        53.4100, 54.7800, 54.8700,  0.0000, 56.5500,  0.0000, 58.1400, 58.2700,
        59.4400, 60.4500,  0.0000, 61.5100, 60.8800, 61.7500])
```

[17]:
```
print(loss_list)
```

```
tensor([2.2724, 2.2161, 2.1783, 2.1476, 2.1239, 2.1037, 2.0891, 2.0763, 2.0648,
        2.0506, 2.0392, 2.0284, 2.0175, 2.0061, 1.9942, 1.9882, 1.9756, 1.9643,
        1.9543, 0.0000, 1.9301, 0.0000, 1.9090, 1.9128, 1.8990, 1.8898, 0.0000,
        1.8642, 1.8704, 1.8628])
```

[18]:
```
print(lqd_list)
```

```
tensor([ 8.,  5., 11., 10., 10., 15., 10., 11., 12., 15., 14., 13., 11., 13.,
        17.,  6., 11., 11.,  8.,  0.,  3.,  0., 12.,  5.,  8.,  9.,  0., 12.,
         7.,  4.])
```

[19]:
```
print(lb_list)
```

```
tensor([3., 4., 2., 3., 2., 5., 3., 5., 6., 3., 4., 6., 6., 5., 5., 3., 2., 2.,
        6., 0., 3., 0., 4., 6., 5., 4., 0., 2., 2., 6.])
```

[20]:
```
print(acc_list[29])
```

```
tensor([59.2000, 63.2000, 62.0000, 59.6000, 61.7500, 58.4000, 63.2000, 57.2000,
        63.6000, 58.8000, 60.8000, 60.0000, 61.6000, 60.4000, 61.7500, 58.4000,
        59.2000, 53.6000, 59.6000, 61.7500, 69.2000, 61.7500, 60.0000, 56.8000,
        61.7500, 60.8000, 61.2000, 52.4000, 60.0000, 64.4000, 61.7500, 65.6000,
        61.7500, 58.0000, 61.7500, 68.0000, 65.2000, 62.0000, 59.6000, 62.4000])
```