FedAvg算法复现

1. 准备工作

FedAvg算法过程如下：

Algorithm 1 FederatedAveraging. The K clients are indexed by k; B is the local minibatch size, E is the number of local epochs, and η is the learning rate.

Server executes:
  initialize w_0
  for each round $t = 1, 2, \dots$ **do**
    $m \leftarrow \max(C \cdot K, 1)$
    $S_t \leftarrow$ (random set of m clients)
    for each client $k \in S_t$ **in parallel do**
      $w_{t+1}^k \leftarrow \text{ClientUpdate}(k, w_t)$
    $w_{t+1} \leftarrow \sum_{k=1}^{K} \frac{n_k}{n} w_{t+1}^k$

ClientUpdate(k, w)**:**   *// Run on client k*
  $\mathcal{B} \leftarrow$ (split $\mathcal{P}_k$ into batches of size B)
  for each local epoch i from 1 to E **do**
    for batch $b \in \mathcal{B}$ **do**
      $w \leftarrow w - \eta \nabla \ell(w; b)$
  return w to server

数据集介绍：

CIFAR-10是一个更接近普适物体的彩色图像数据集。CIFAR-10 是由Hinton 的学生Alex Krizhevsky 和Ilya Sutskever 整理的一个用于识别普适物体的小型数据集。一共包含10 个类别的RGB 彩色图片：飞机（airplane）、汽车（automobile）、鸟类（bird）、猫（cat）、鹿（deer）、狗（dog）、蛙类（frog）、马（horse）、船（ship）和卡车（truck）。每个图片的尺寸为32 × 32，每个类别有6000个图像，数据集中一共有50000 张训练图片和10000 张测试图片。

2. 分割数据集

```
def get_datasets(data_name, dataroot, normalize=True, val_size=10000):
    """
    get_datasets returns train/val/test data splits of CIFAR10/100 datasets
    :param data_name: name of dataset, choose from [cifar10, cifar100]
    :param dataroot: root to data dir
```

```
    :param normalize: True/False to normalize the data
    :param val_size: validation split size (in #samples)
    :return: train_set, val_set, test_set (tuple of pytorch dataset/subset)
    """

    if data_name =='cifar10':
        normalization = transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010))
        data_obj = CIFAR10
    elif data_name == 'cifar100':
        normalization = transforms.Normalize((0.5071, 0.4865, 0.4409), (0.2673, 0.2564, 0.2762))
        data_obj = CIFAR100
    else:
        raise ValueError("choose data_name from ['mnist', 'cifar10', 'cifar100']")

    trans = [transforms.ToTensor()]

    if normalize:
        trans.append(normalization)

    transform = transforms.Compose(trans)

    dataset = data_obj(
        dataroot,
        train=True,
        download=True,
        transform=transform
    )

    test_set = data_obj(
        dataroot,
        train=False,
        download=True,
        transform=transform
    )

    train_size = len(dataset) - val_size
    train_set, val_set = torch.utils.data.random_split(dataset, [train_size, val_size])    # 切割数据集伟训练集与验证集

    return train_set, val_set, test_set


def get_num_classes_samples(dataset):
    """
    extracts info about certain dataset
    :param dataset: pytorch dataset object
    :return: dataset info number of classes, number of samples, list of labels
    """
    # ---------------#
    # Extract labels #
    # ---------------#
    if isinstance(dataset, torch.utils.data.Subset):
        if isinstance(dataset.dataset.targets, list):
```

```python
            data_labels_list = np.array(dataset.dataset.targets)[dataset.indices]
        else:
            data_labels_list = dataset.dataset.targets[dataset.indices]
    else:
        if isinstance(dataset.targets, list):
            data_labels_list = np.array(dataset.targets)
        else:
            data_labels_list = dataset.targets
    classes, num_samples = np.unique(data_labels_list, return_counts=True)
    num_classes = len(classes)
    return num_classes, num_samples, data_labels_list


def gen_classes_per_node(dataset, num_users, classes_per_user=2, high_prob=0.6, low_prob=0.4):
    """
    creates the data distribution of each client
    :param dataset: pytorch dataset object
    :param num_users: number of clients
    :param classes_per_user: number of classes assigned to each client
    :param high_prob: highest prob sampled
    :param low_prob: lowest prob sampled
    :return: dictionary mapping between classes and proportions, each entry refers to other client
    """
    num_classes, num_samples, _ = get_num_classes_samples(dataset)

    # -------------------------------------------#
    # Divide classes + num samples for each user #
    # -------------------------------------------#
    assert (classes_per_user * num_users) % num_classes == 0, "equal classes appearance is needed"
    count_per_class = (classes_per_user * num_users) // num_classes
    class_dict = {}
    for i in range(num_classes):
        # sampling alpha_i_c
        probs = np.random.uniform(low_prob, high_prob, size=count_per_class)
        # normalizing
        probs_norm = (probs / probs.sum()).tolist()
        class_dict[i] = {'count': count_per_class, 'prob': probs_norm}

    # -------------------------------------#
    # Assign each client with data indexes #
    # -------------------------------------#
    class_partitions = defaultdict(list)
    for i in range(num_users):
        c = []
        for _ in range(classes_per_user):
            class_counts = [class_dict[i]['count'] for i in range(num_classes)]
            max_class_counts = np.where(np.array(class_counts) == max(class_counts))[0]
            c.append(np.random.choice(max_class_counts))
            class_dict[c[-1]]['count'] -= 1
        class_partitions['class'].append(c)
```

```
        class_partitions['prob'].append([class_dict[i]['prob'].pop() for i in c])
    return class_partitions


def gen_data_split(dataset, num_users, class_partitions):
    """
    divide data indexes for each client based on class_partition
    :param dataset: pytorch dataset object (train/val/test)
    :param num_users: number of clients
    :param class_partitions: proportion of classes per client
    :return: dictionary mapping client to its indexes
    """
    num_classes, num_samples, data_labels_list = get_num_classes_samples(dataset)

    # -------------------------- #
    # Create class index mapping #
    # -------------------------- #
    data_class_idx = {i: np.where(data_labels_list == i)[0] for i in range(num_classes)}

    # --------- #
    # Shuffling #
    # --------- #
    for data_idx in data_class_idx.values():
        random.shuffle(data_idx)

    # ------------------------------ #
    # Assigning samples to each user #
    # ------------------------------ #
    user_data_idx = [[] for i in range(num_users)]
    for usr_i in range(num_users):
        for c, p in zip(class_partitions['class'][usr_i], class_partitions['prob'][usr_i]):
            end_idx = int(num_samples[c] * p)
            user_data_idx[usr_i].extend(data_class_idx[c][:end_idx])
            data_class_idx[c] = data_class_idx[c][end_idx:]

    return user_data_idx


def gen_random_loaders(data_name, data_path, num_users, bz, classes_per_user):
    """
    generates train/val/test loaders of each client
    :param data_name: name of dataset, choose from [cifar10, cifar100]
    :param data_path: root path for data dir
    :param num_users: number of clients
    :param bz: batch size
    :param classes_per_user: number of classes assigned to each client
    :return: train/val/test loaders of each client, list of pytorch dataloaders
    """
    loader_params = {"batch_size": bz, "shuffle": False, "pin_memory": True, "num_workers": 0}
    dataloaders = []
    datasets = get_datasets(data_name, data_path, normalize=True)
```

```
    for i, d in enumerate(datasets):
        # ensure same partition for train/test/val
        if i == 0:
            cls_partitions = gen_classes_per_node(d, num_users, classes_per_user)
            loader_params['shuffle'] = True
        usr_subset_idx = gen_data_split(d, num_users, cls_partitions)
        # create subsets for each client
        subsets = list(map(lambda x: torch.utils.data.Subset(d, x), usr_subset_idx))
        # create dataloaders from subsets
        dataloaders.append(list(map(lambda x: torch.utils.data.DataLoader(x, **loader_params), subsets)))

    return dataloaders

```

3. 数据节点类

```
from experiments.dataset import gen_random_loaders


class BaseNodes:
    def __init__(
            self,
            data_name,
            data_path,
            n_nodes,
            batch_size=128,
            classes_per_node=2
    ):

        self.data_name = data_name
        self.data_path = data_path
        self.n_nodes = n_nodes
        self.classes_per_node = classes_per_node

        self.batch_size = batch_size

        self.train_loaders, self.val_loaders, self.test_loaders = None, None, None
        self._init_dataloaders()

    def _init_dataloaders(self):
        self.train_loaders, self.val_loaders, self.test_loaders = gen_random_loaders(
            self.data_name,
            self.data_path,
            self.n_nodes,
            self.batch_size,
            self.classes_per_node
        )

    def __len__(self):
        return self.n_nodes
```

4. CNN模型类

```
import torch.nn.functional as F
from torch import nn
import numpy as np
import torch
from torch.utils.data import TensorDataset
from torch.utils.data import DataLoader


class CNN(nn.Module):
    def __init__(self, in_channels=3, n_kernels=16, out_dim=10):
        super(CNN, self).__init__()

        self.conv1 = nn.Conv2d(in_channels=in_channels, out_channels=n_kernels, kernel_size=5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(in_channels=n_kernels, out_channels=2 * n_kernels, kernel_size=5)
        self.fc1 = nn.Linear(in_features=2 * n_kernels * 5 * 5, out_features=120)
        self.fc2 = nn.Linear(in_features=120, out_features=84)
        self.fc3 = nn.Linear(in_features=84, out_features=out_dim)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(x.shape[0], -1)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x


class Client(object):
    def __int__(self, trainDataSet, dev):
        self.train_ds = trainDataSet
        self.dev = dev
        self.train_dl = None
        self.local_parameter = None
```

5. 利用FedAvg算法训练

```
def train(data_name: str, data_path: str, classes_per_node: int, num_nodes: int,
          steps: int, node_iter: int, optim: str, lr: float, inner_lr: float,
          embed_lr: float, wd: float, inner_wd: float, embed_dim: int, hyper_hid: int,
          n_hidden: int, n_kernels: int, bs: int, device, eval_every: int, save_path: Path,
          seed: int) -> None:
    ###############################
```

```
    # init nodes, hnet, local net #
    ###############################
    steps = 5
    node_iter = 5
    nodes = BaseNodes(data_name, data_path, num_nodes, classes_per_node=classes_per_node,
                      batch_size=bs)
    net = CNN(n_kernels=n_kernels)
    # hnet = hnet.to(device)
    net = net.to(device)

    ##################
    # init optimizer #
    ##################
    # embed_lr = embed_lr if embed_lr is not None else lr
    optimizer = torch.optim.SGD(
        net.parameters(), lr=inner_lr, momentum=.9, weight_decay=inner_wd
    )
    criteria = torch.nn.CrossEntropyLoss()

    ################
    # init metrics #
    ################
    # step_iter = trange(steps)
    step_iter = range(steps)
    # train process
    # record  the global parameters
    global_parameters = {}
    for key, parameter in net.state_dict().items():
        global_parameters[key] = parameter.clone()
    for step in step_iter:

        local_parameters_list = {}
        # 需要训练的node数目
        for i in range(node_iter):
            # 随机选择一个客户端
            node_id = random.choice(range(num_nodes))
            # 用全局模型参数训练当前客户端
            local_parameters = local_upload(nodes.train_loaders[node_id], 5, net, criteria, optimizer,
                                            global_parameters, dev='cpu')
            print("\nEpoch: {}, Node Count: {}, Node ID: {}".format(step + 1, i + 1, node_id), end="")
            evaluate(net, local_parameters, nodes.val_loaders[node_id], 'cpu')
            local_parameters_list[i] = local_parameters

        # 更新当前轮次模型的参数
        sum_parameters = None
        for node_id, parameters in local_parameters_list.items():
            if sum_parameters is None:
                sum_parameters = parameters
            else:
                for key in parameters.keys():
                    sum_parameters[key] += parameters[key]
        for var in global_parameters:
            global_parameters[var] = (sum_parameters[var] / node_iter)
    # test
```

```
    net.load_state_dict(global_parameters, strict=True)
    net.eval()
    for data_set in nodes.test_loaders:
        running_correct = 0
        running_samples = 0
        for data, label in data_set:
            pred = net(data)
            running_correct += pred.argmax(1).eq(label).sum().item()
            running_samples += len(label)
        print("\t" + 'accuracy: %.2f' % (running_correct / running_samples),
end="")
```

6. client训练函数

```
def local_upload(train_data_set, local_epoch, net, loss_fun, opt,
global_parameters, dev):
    # 加载当前通信中最新全局参数
    net.load_state_dict(global_parameters, strict=True)
    # 设置迭代次数
    net.train()
    for epoch in range(local_epoch):
        for data, label in train_data_set:
            data, label = data.to(dev), label.to(dev)
            # 模型上传入数据
            predict = net(data)
            loss = loss_fun(predict, label)
            # 反向传播
            loss.backward()
            # 计算梯度，并更新梯度
            opt.step()
            # 将梯度归零，初始化梯度
            opt.zero_grad()
    # 返回当前Client基于自己的数据训练得到的新的模型参数
    return net.state_dict()
```

7. 模型评估函数

```
def evaluate(net, global_parameters, testDataLoader, dev):
    net.load_state_dict(global_parameters, strict=True)
    running_correct = 0
    running_samples = 0
    net.eval()
    # 载入测试集
    for data, label in testDataLoader:
        data, label = data.to(dev), label.to(dev)
        pred = net(data)
        running_correct += pred.argmax(1).eq(label).sum().item()
        running_samples += len(label)
    print("\t" + 'accuracy: %.2f' % (running_correct / running_samples),
end="")
```

8. 模型训练结果

因为设备原因，暂时无法训练出论文中的模型

```
Epoch: 2, Node Count: 35, Node ID: 39   accuracy: 0.7173376083374023
Epoch: 2, Node Count: 36, Node ID: 43   accuracy: 0.5102083086967468
Epoch: 2, Node Count: 37, Node ID: 41   accuracy: 0.6944901347160339
Epoch: 2, Node Count: 38, Node ID: 4    accuracy: 0.4871794879436493
Epoch: 2, Node Count: 39, Node ID: 47   accuracy: 0.46875
Epoch: 2, Node Count: 40, Node ID: 25   accuracy: 0.741122126579284‍7
Epoch: 2, Node Count: 41, Node ID: 39   accuracy: 0.6196151375770569
Epoch: 2, Node Count: 42, Node ID: 40   accuracy: 0.784040212631225‍6
Epoch: 2, Node Count: 43, Node ID: 16   accuracy: 0.7732007503509521
Epoch: 2, Node Count: 44, Node ID: 8    accuracy: 0.7512019276618958
Epoch: 2, Node Count: 45, Node ID: 27   accuracy: 0.6020998954772949
Epoch: 2, Node Count: 46, Node ID: 5    accuracy: 0.4984374940395355
Epoch: 2, Node Count: 47, Node ID: 40   accuracy: 0.7109375
Epoch: 2, Node Count: 48, Node ID: 49   accuracy: 0.7680289149284363
```

附录：关键函数记录

torch.nn.Module.load_state_dict

load_state_dict(state_dict, strict=True)

使用 state_dict 反序列化模型参数字典。用来加载模型参数。将 state_dict 中的 parameters 和 buffers 复制到此 module 及其子节点中。
概况：给模型对象加载训练好的模型参数，即加载模型参数
state_dict (字典类型) – 一个包含参数和持续性缓冲的字典，往往是pytorch模型pth文件

strict (布尔类型, 可选) – 该参数用来指明是否需要强制严格匹配, 即:state_dict中的关键字是否需要和该模块的state_dict()方法返回的关键字强制严格匹配.默认值是True

nn.utils.clip_grad_norm_

nn.utils.clip_grad_norm_(parameters, max_norm, norm_type=2)

这个函数是根据参数的范数来衡量的

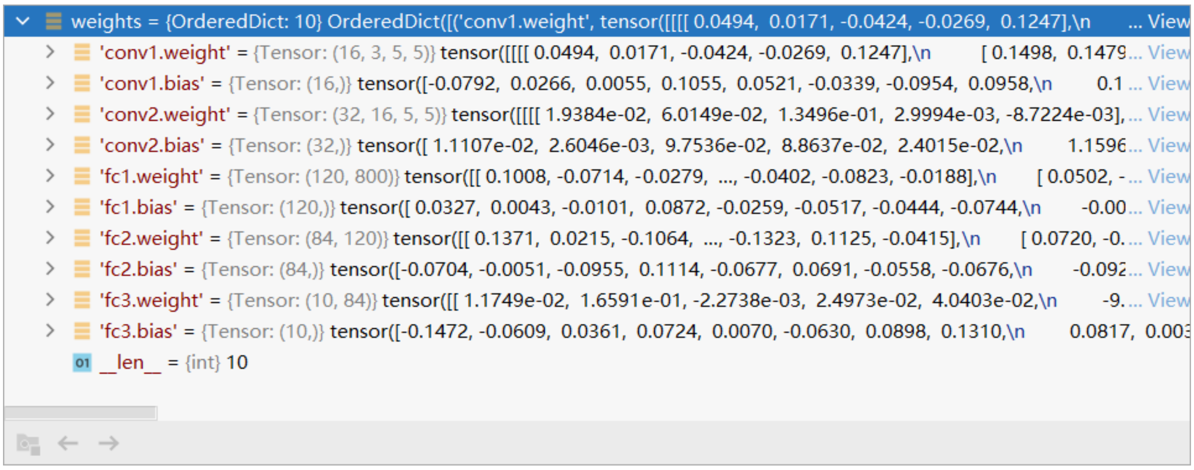
Parameters:

parameters (Iterable[Variable]) – 一个基于变量的迭代器，会进行归一化（原文：an iterable of Variables that will have gradients normalized）
max_norm (float or int) – 梯度的最大范数（原文：max norm of the gradients）
norm_type(float or int) – 规定范数的类型，默认为L2（原文：type of the used p-norm. Can be'inf'for infinity norm）
Returns:参数的总体范数（作为单个向量来看）（原文：Total norm of the parameters (viewed as a single vector).）

torch.nn.Embedding

```python
torch.nn.Embedding(num_embeddings, embedding_dim, padding_idx=None, max_norm=None, norm_type=2.0, scale_grad_by_freq=False, sparse=False, _weight=None, device=None, dtype=None)
```

一个简单的查找表，用于存储固定字典和大小的嵌入。该模块通常用于存储词嵌入并使用索引检索它们。模块的输入是索引列表，输出是相应的词嵌入。




```
CNNTarget(
  (conv1): Conv2d(3, 16, kernel_size=(5, 5), stride=(1, 1))
  (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
   ceil_mode=False)
  (conv2): Conv2d(16, 32, kernel_size=(5, 5), stride=(1, 1))
  (fc1): Linear(in_features=800, out_features=120, bias=True)
  (fc2): Linear(in_features=120, out_features=84, bias=True)
  (fc3): Linear(in_features=84, out_features=10, bias=True)
)
```