联邦学习-实验报告

胡若凡 3200102312

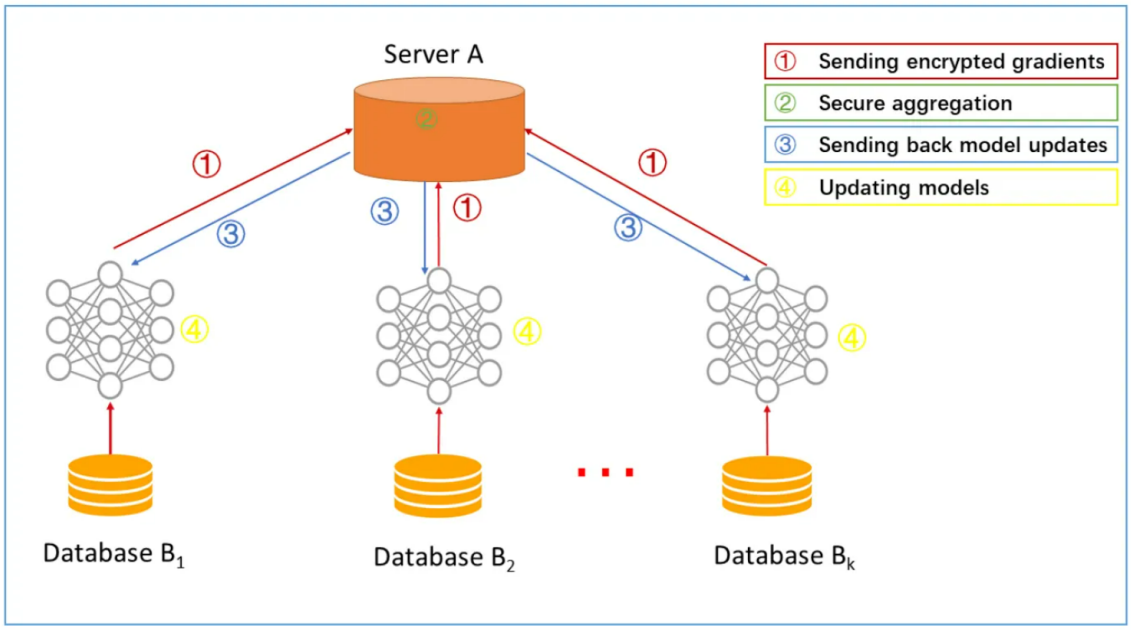
1. 实验介绍

1.1 联邦学习介绍

联邦学习是一种机器学习范式，可以在一个中心服务器的协调下让多个客户端互相合作，即便在数据分散在客户端的情况下也可以得到一个完整的机器学习模型。

一般而言，联邦学习有以下四个步骤（以基于梯度训练深度神经网络为例）：

1. 所有客户端分别独立地在本地数据上进行训练；
2. 客户端对模型参数或者模型参数的梯度进行加密，上传到服务器；
3. 服务器对搜集到的客户端的模型参数或者梯度进行聚合，并且是安全聚合（Secure Aggregation）；
4. 服务器将模型下发到各个客户端；



1.2 实验目的

- 在本地模拟联邦学习的过程，体会多个客户端各自使用一部分数据，并借助当前的全局模型进行训练的过程
- 学习联邦学习中的基础算法FedAvg，并学习联邦学习中各类算法的概念
- 通过联邦学习的实验，体会安全性实验的重要性
- 更好的理解课堂上对于隐私性等问题的实际操作

2. 数据集描述

2.1 MNSIT

MNIST数据集是一个经典的手写数字图片数据集，包括0-9十个数字的手写数字，其包含了60,000个训练样本和10,000个测试样本，每张图片的尺寸为28x28像素。它来自美国国家标准与技术研究所(NIST)的Special Database 3，Yann LeCun等人收集整理。近年来，它被广泛应用在计算机视觉领域的基础实验中，其简单和易用性使它成为机器学习的标准数据集之一。

2.2 CIFAR-10

CIFAR-10（Canadian Institute For Advanced Research）数据集是一个经典的图像分类数据集，其中包含了10类共计60000张32*32大小的彩色图像，每个类别有6000张图片。这10个类别分别为飞机、汽车、鸟类、猫、鹿、狗、青蛙、马、船和卡车。该数据集由加拿大高级研究所(Canadian Institute for Advanced Research)创建，被广泛地应用于计算机视觉领域，如模式识别、图像处理和深度学习等方面。相对于其它图像数据集，CIFAR-10数据集具有较小的规模，因此是一个很好的深度学习入门数据集。其图像类别比较容易区分，但是图像本身的细节比较复杂，因此对于深度学习算法的训练和评价具有一定难度。

3. 模型思想

3.1 方法介绍

FedAvg是一种常用的联邦学习算法，它通过加权平均来聚合模型参数。FedAvg的基本思想是将本地模型的参数上传到服务器，服务器计算所有模型参数的平均值，然后将这个平均值广播回所有本地设备。这个过程可以迭代多次，直到收敛。

为了保证模型聚合的准确性，FedAvg算法采用加权平均的方式进行模型聚合。具体来说，每个设备上传的模型参数将赋予一个权重，然后进行加权平均。设备上传的模型参数的权重是根据设备上的本地数据量大小进行赋值的，数据量越多的设备权重越大。

算法的伪代码如下：

Algorithm 1 Federated Averaging

Server executes:initialize w_0 **for** each round $t = 1, 2, \dots$ **do**

$S_t = (\text{random set of } \max(\cdot K, 1) \text{ clients})$

for each client $k \in S_t$ **in parallel do**

$w_{t+1}^k \leftarrow \text{ClientUpdate}(k, w_t)$

end for

$w_{t+1} \leftarrow \sum_{t=1}^{\frac{n_k}{n}} w_{t+1}^k$

end for

ClientUpdate(k, w): *// Executed on client k*

for each local epoch i from 1 to **do**

batches $\leftarrow$ (data $_k$ split into batches of size)

for batch b in batches **do**

$w \leftarrow w - \eta(w; b)$

end for

end for

return w to server =0

3.2 代码阅读分析

下面结合代码进行阅读分析，首先要注意的是，**尽管FedAvg中对每个模型赋予不同的权重，在本次实验中为了方便，我们假设所有模型的地位同等**，当然，这也和我们每一个client使用的是同样的算法模型有关，因为假定数据随机打乱后，独立同分布，如果使用的模型一样，理应是同等地位的。

3.2.1 Dataset.py

该函数的主要目的是引入数据集，并对数据集中的数据做Tensor转换和相应的增强，使得训练出的模型效果会更好。

由于本实验使用的model里，对于MNIST数据集黑白像素不匹配，因此在这个模块，我修改了MNIST的转换，以适配实验调用的模型，具体是调整到3 channels，并且小幅度的做了数据增强。

```
import torch
from torchvision import datasets, transforms

def get_dataset(dir, name):

  if name == 'mnist':
    input_size = 224
    transforms_train = transforms.Compose([
      transforms.Resize((input_size, input_size)),
      transforms.Grayscale(num_output_channels=3),
      transforms.RandomHorizontalFlip(),
      transforms.RandomRotation(degrees=15),
      transforms.ToTensor(),
```

```
      transforms.Normalize((0.1307,), (0.3081,))
    ])
    transforms_eval = transforms.Compose([
      transforms.Resize((input_size, input_size)),
      transforms.Grayscale(num_output_channels=3),
      transforms.ToTensor(),
      transforms.Normalize((0.1307,), (0.3081,))
    ])

    train_dataset = datasets.MNIST(dir, train=True, download=True,
transform=transforms_train)
    eval_dataset = datasets.MNIST(dir, train=False, transform=transforms_eval)

  elif name=='cifar':
    transform_train = transforms.Compose([
      transforms.RandomCrop(32, padding=4),
      transforms.RandomHorizontalFlip(),
      transforms.ToTensor(),
      transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010)),
    ])

    transform_test = transforms.Compose([
      transforms.ToTensor(),
      transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010)),
    ])

    train_dataset = datasets.CIFAR10(dir, train=True, download=True,
                    transform=transform_train)
    eval_dataset = datasets.CIFAR10(dir, train=False, transform=transform_test)


  return train_dataset, eval_dataset
```

3.2.2 Client.py

Client模块，进行的就是联邦学习中每一个客户端使用自己的数据进行单独训练的过程。在本次实验中，又分为

- 导入数据：**由于实验的简化目的，**而且也是模拟联邦学习，因此对于数据集进行了平均划分，根据Client的id号，将对应的一部分数据给传回
- 本地训练：客户端会把当前的global_model的参数传过来，并且Client利用该参数初始化，然后用自己的数据集进行训练。
- 差异返回：训练后相应的epoch后，计算每个参数的差异，并且用字典进行返回。

```
import models, torch, copy
class Client(object):
  """
  用传入的模型参数model进行初始化，然后使用本地的SGD优化器进行local_model模型训练
  最后返回一个包含model与local_model参数差异的字典
  """
    def __init__(self, conf, model, train_dataset, id=-1):

        self.conf = conf            # 联邦学习的超参数配置
        self.local_model = models.get_model(self.conf["model_name"])        # 实
例化本地模型
        self.client_id = id     # 客户端id
        self.train_dataset = train_dataset  # 客户端的训练数据集
```

```
        all_range = list(range(len(self.train_dataset)))     # 训练数据集的索引范围
        data_len = int(len(self.train_dataset) / self.conf['no_models'])     # 每
个模型所需的数据集长度
        train_indices = all_range[id * data_len: (id + 1) * data_len]       # 获
得当前客户端的训练数据集的索引范围
        #因为这里是要进行模拟,所以每个模型分到自己的一部分数据,就类似于联邦学习里大家各自拥有
一部分数据

        # 实例化dataloader，用于客户端的模型训练
        self.train_loader = torch.utils.data.DataLoader(
            self.train_dataset,
            batch_size=conf["batch_size"],
            sampler=torch.utils.data.sampler.SubsetRandomSampler(train_indices)
#打乱以便训练
        )

    def local_train(self, model):

        # 将模型参数复制到本地模型中
        for name, param in model.state_dict().items():
            self.local_model.state_dict()[name].copy_(param.clone())

        # 实例化优化器（随机梯度下降SGD）
        optimizer = torch.optim.SGD(
            self.local_model.parameters(),
            lr=self.conf['lr'],
            momentum=self.conf['momentum']
        )

        # 开始训练本地模型
        self.local_model.train()
        for e in range(self.conf["local_epochs"]):
            for batch_id, batch in enumerate(self.train_loader):
                data, target = batch

                if torch.cuda.is_available():
                    data = data.cuda()
                    target = target.cuda()

                optimizer.zero_grad()
                output = self.local_model(data)     # 前向传播
                loss = torch.nn.functional.cross_entropy(output, target)    # 计
算损失
                loss.backward()                     # 反向传播
                optimizer.step()                        # 更新参数
            print("Epoch %d done." % e) # 每个epoch结束后，在控制台打印一条信息

        # 计算本地模型与传入模型之间的参数差异，并返回
        diff = dict()
        for name, data in self.local_model.state_dict().items():
            diff[name] = (data - model.state_dict()[name])
        return diff
```

3.2.3 Server.py

Server模块主要进行的是全局模型的评估与调整。正如上述所言，在客户端各自训练之后，将会返回字典，主函数将会利用这些差异，去对global_model做更新调整，幅度大小受超参数控制

```
import models, torch

class Server(object):
  """
  进行模型的评估
  还有对模型进行参数更新，更新的幅度受到conf中的lambda影响
  """
    def __init__(self, conf, eval_dataset):
        """
        初始化Server对象
        """
        self.conf = conf    # 设置的超参数配置
        self.global_model = models.get_model(self.conf["model_name"])    # 实例化
全局模型
        self.eval_loader = torch.utils.data.DataLoader(
            eval_dataset,
            batch_size=self.conf["batch_size"],
            shuffle=True
        )   # 生成用于评估的DataLoader

    def model_aggregate(self, weight_accumulator):
        """
        计算全局模型的参数更新
        """
        for name, data in self.global_model.state_dict().items():
      #weight_accumulator是一个字典，包含每个客户端的参数差异
        #update_per_layer则是每一层模型参数的平均更新量
            update_per_layer = weight_accumulator[name] * self.conf["lambda"]

      #保证更新量和模型参数的类型一致
            if data.type() != update_per_layer.type():
                data.add_(update_per_layer.to(torch.int64))
            else:
                data.add_(update_per_layer)

    def model_eval(self):
        """
        评估全局模型的效果
        """
        self.global_model.eval()

        total_loss = 0.0         #总损失
        correct = 0          #正确的预测数量
        dataset_size = 0         #数据集总大小
        for batch_id, batch in enumerate(self.eval_loader):
            data, target = batch
            dataset_size += data.size()[0]  #统计数据集大小

            if torch.cuda.is_available():
                data = data.cuda()
                target = target.cuda()

            output = self.global_model(data)     # 前向传播
            total_loss += torch.nn.functional.cross_entropy(output, target,
                                                reduction='sum').item() #计算总损失
（CrossEntropy）
```

```
            pred = output.data.max(1)[1]  # get the index of the max log-
probability
            correct += pred.eq(target.data.view_as(pred)).cpu().sum().item()
 # 统计正确的预测数量

        # 计算准确率和平均损失
        acc = 100.0 * (float(correct) / float(dataset_size))
        total_l = total_loss / dataset_size
        return acc, total_l
```

3.2.4 Main.py

本次实验的主程序，对上述三个文件进行一个整合调用，正如算法所言，做global_epochs次，每一次随机选取conf["k"]个客户端参与训练，每个客户端不止训练一次。在客户端训练完成后，根据差异和字典去调整global_model，再把新的global_model作为初始参数传给下一轮的conf["k"]个客户端。

```
import argparse, json
import datetime
import os
import logging
import torch, random

from server import *
from client import *
import models, datasets


# 本次上机实验为本地模拟联邦学习过程,需一至两小时的训练时间
# run with configuration
# python main.py -c ./utils/conf.json

# 各个文件的简要介绍
# ./utils/conf.json 提供参数配置,如数据集选择、模型选择、客户端数量、抽样数量等
# datasets.py 提供mnist和cifar数据集(已经下载在data文件夹中)
# models.py 提供若干模型
# server.py 为服务端,即整合各客户端模型
# client.py 为客户端,即本地训练模型

"""
这是一个联邦学习的主程序,负责调用client.py和server.py实现联邦学习过程。
主要包含以下文件:
- conf.json:提供参数配置,如数据集选择、模型选择、客户端数量、抽样数量等
- datasets.py:提供mnist和cifar数据集(已经下载在data文件夹中)
- models.py:提供若干模型
- server.py:为服务端,即整合各客户端模型
- client.py:为客户端,即本地训练模型
"""
if __name__ == '__main__':
  # 使用 argparse 获取命令行输入的配置文件,获取conf路径,例如:python main.py -c
./utils/conf.json
  parser = argparse.ArgumentParser(description='Federated Learning')
  parser.add_argument('-c', '--conf', dest='conf')
  args = parser.parse_args()

  # 读取conf.json文件
  with open(args.conf, 'r') as f:
    conf = json.load(f)
```

```
# 读取训练集和测试集数据
train_datasets, eval_datasets = datasets.get_dataset("./data/", conf["type"])

# 初始化服务器对象
server = Server(conf, eval_datasets)
clients = []

# 根据客户端数量创建客户端对象
for c in range(conf["no_models"]):
  clients.append(Client(conf, server.global_model, train_datasets, c))

# 开始联邦学习过程
print("\n\n")
for e in range(conf["global_epochs"]):

  # 随机抽取k个客户端参与本轮训练
  candidates = random.sample(clients, conf["k"])

  # 初始化权重累加器
  weight_accumulator = {}

  for name, params in server.global_model.state_dict().items():
    weight_accumulator[name] = torch.zeros_like(params)

  # 所有候选客户端参与联邦学习过程
  for c in candidates:
    # 执行本地训练并返回更新后的权重值
    diff = c.local_train(server.global_model)

    # 将每个客户端更新的权重进行求和并存入累加器
    for name, params in server.global_model.state_dict().items():
      weight_accumulator[name].add_(diff[name])

  # 使用所有客户端更新后的权重计算全局模型的平均权重
  server.model_aggregate(weight_accumulator)

  # 使用模型进行验证并打印结果
  acc, loss = server.model_eval()
  print("Epoch %d, acc: %f, loss: %f\n" % (e, acc, loss))
```

3.3 实验结果

3.3.1 MNIST

使用所有参数预设值，只是把**数据进行了相应的增强**，最后只进行了13个epoch，精确率就已经达到了98.27%

```
Epoch 0 done.
Epoch 1 done.
Epoch 2 done.
Epoch 0 done.
Epoch 1 done.
Epoch 2 done.
Epoch 0 done.
Epoch 1 done.
Epoch 2 done.
Epoch 0 done.
Epoch 1 done.
Epoch 2 done.
Epoch 11, acc: 98.260000, loss: 0.059453

Epoch 0 done.
Epoch 1 done.
Epoch 2 done.
Epoch 0 done.
Epoch 1 done.
Epoch 2 done.
Epoch 0 done.
Epoch 1 done.
Epoch 2 done.
Epoch 0 done.
Epoch 1 done.
Epoch 2 done.
Epoch 0 done.
Epoch 1 done.
Epoch 2 done.
Epoch 12, acc: 98.270000, loss: 0.056441

Epoch 0 done.
```

3.3.2 CIFAR-10

使用所有的预参数设置，不做调整，进行了20次epoch，精确率最后为68.89%

```
Epoch 0 done.
Epoch 1 done.
Epoch 2 done.
Epoch 0 done.
Epoch 1 done.
Epoch 2 done.
Epoch 0 done.
Epoch 1 done.
Epoch 2 done.
Epoch 0 done.
Epoch 1 done.
Epoch 2 done.
Epoch 18, acc: 68.760000, loss: 0.903930

Epoch 0 done.
Epoch 1 done.
Epoch 2 done.
Epoch 0 done.
Epoch 1 done.
Epoch 2 done.
Epoch 0 done.
Epoch 1 done.
Epoch 2 done.
Epoch 0 done.
Epoch 1 done.
Epoch 2 done.
Epoch 0 done.
Epoch 1 done.
Epoch 2 done.
Epoch 19, acc: 68.890000, loss: 0.897391

(AI_security) huruofan@ai-server-2:~/AI_security/Federated_Learning$
```

4. 优化与改进

针对本次实验，同样也有一些可以进行优化的方向， 这里提出一些我思考的可能方案

4.1 模型存取

在本次实验中，有一个明显的问题在于，模型没有对前后两次的Accuracy进行比较，只是不断的使用当前epoch得到的模型使用。最后输出的也就直接是底第20个epoch的模型。所以很明显的，可以设置一个全局变量ACCURACY，专门记录当前最好的global_model，并且不断比较，**将最好的global_model保存下来。**

4.2 动态权重

在本次实验中，**每一个Client**的变化我们都是以相同的权重进行了累加，这其实不一定合理。原因在于，不同的数据集，不同的实验方法，可能会有各种各样的问题。如果一个用户的模型效果很差，比如说它的模型出现了过拟合的特点，这其实并不一定有利于我们的global_model对其做吸收学习。所以，这样不好的模型，我认为应该附上**更小的权重**。我们应该评估每一个Client模型，去研究其是否应该被纳入这一次的考量之中，或许可以动态赋予权重。

我会有这样的思考，是因为我在实验中观察epoch，曾出现过一次明显的下降：

4.3 客户端选择

在本次实验中，我们的每个Client的dataset基本是一样性能的，所以不会有特别明显的好坏区别，但是在实际操作中，可能有的客户端的数据就是污染的，有害处的，甚至进一步说，**我们之前课程也提到过针对模式与数据的攻击，可能有的Client模型，就是在其不知觉中，有潜在攻击性质的，因为有蓄意者可能注入了有毒数据，污染了模型，**所以对模型是应该有所选择的，这也是为了提高global_client模型所必要的一步。

4.4 传输问题

在本次实验中，所有客户端全部部署在本地，所以是没有太大的传输问题的，但是这也是联邦学习中很重要的一点，我在学习过程中，也了解到要调整传输策略，也就是所谓的动态通信。

动态通信的目的是基于网络的实时状态来确定最佳的通信时间和方式。例如，如果网络带宽较小或者网络拥塞，则可以选择减少通信频率或者使用更轻量级的通信方式，例如压缩数据、使用更小的模型等。反之，如果网络状态较好，则可以选择增加通信频率或者使用更复杂的通信方式，例如交叉验证等。

除了根据网络状况调整通信周期和方式外，还可以根据客户端的特点来调整通信策略。例如，一些客户端可能计算能力较弱，需要更大的通信周期来充分利用每一个机会提交模型更新。

5. 实验心得

在这次的实验中，我碰到了一个很神奇的地方，就是准确率突增的状况，如下图所示：

```
Epoch 2 done.
Epoch 0 done.
Epoch 1 done.
Epoch 2 done.
Epoch 0 done.
Epoch 1 done.
Epoch 2 done.
Epoch 0 done.
Epoch 1 done.
Epoch 2 done.
Epoch 0 done.
Epoch 1 done.
Epoch 2 done.
Epoch 1, acc: 25.930000, loss: 3.979818

Epoch 0 done.
Epoch 1 done.
Epoch 2 done.
Epoch 0 done.
Epoch 1 done.
Epoch 2 done.
Epoch 0 done.
Epoch 1 done.
Epoch 2 done.
Epoch 0 done.
Epoch 1 done.
Epoch 2 done.
Epoch 0 done.
Epoch 1 done.
Epoch 2 done.
Epoch 2, acc: 73.630000, loss: 1.508134
```

而在这之后，准确率又逐渐平缓下来：

```
Epoch 2 done.
Epoch 0 done.
Epoch 1 done.
Epoch 2 done.
Epoch 0 done.
Epoch 1 done.
Epoch 2 done.
Epoch 0 done.
Epoch 1 done.
Epoch 2 done.
Epoch 0 done.
Epoch 1 done.
Epoch 2 done.
Epoch 5, acc: 97.260000, loss: 0.108716

Epoch 0 done.
Epoch 1 done.
Epoch 2 done.
Epoch 0 done.
Epoch 1 done.
Epoch 2 done.
Epoch 0 done.
Epoch 1 done.
Epoch 2 done.
Epoch 0 done.
Epoch 1 done.
Epoch 2 done.
Epoch 0 done.
Epoch 1 done.
Epoch 2 done.
Epoch 6, acc: 97.730000, loss: 0.089427
```

在我查询了资料之后，了解到了这通常被称为**“奇异点”（或“跳跃点”）**现象。其原因可能是：在联邦学习中，数据分布通常会随着时间变化。如果在训练过程中，数据分布发生了变化，模型可能无法充分适应新的数据分布。而当数据分布在某个时间点稳定下来或者发生较小的变化时，模型可能会快速适应并产生显著的性能提升。这是我实验中觉得十分有趣的一点。

另外，通过这次实验，我了解到了联邦学习的相关知识，在实验中，我感觉最重要的选择合适的超参数，例如学习率、优化器和正则化系数等，此外，还要一定注意权重的分配，这些都是很重要的参数内容。

此外，我们这次的实验中是没有通信开销的问题的。在实验中，我们需要仔细设计通信策略，以便在保证模型准确度的同时降低通信开销。通信方式可以是同步或异步的，通信时可以进行数据压缩、差分隐私等处理。

可惜这个实验的运行十分缓慢，所以我只是为两个数据集都使用预置参数来进行了一下实验，并从理论角度提出了一些相关的注意事项和方法，如果有机会，希望可以用更快的GPU来运行，可以做更好的消融实验。