大学生创新创业训练计划

学　　院	微电子学院	
指导教师	刘强	
负责人	赵梓琢	3022232153
成员	权泓儒	3022205077
	王子实	3022232130
	李俊兵	3022232132
	韦泽榕	3022232133

一、项目概述

项目名称：联邦学习框架下差分隐私技术对数据隐私性及模型可用性的影响探究

场景描述：

联邦学习旨在保证“数据不出域，可用不可见”的前提下，通过协调多客户端行为，训练本地模型，交换梯度更新，最终实现全局模型收敛。相比于中心化训练模式，联邦学习训练所得全局模型应保持相应的预测精度(可用性)。然而，联邦学习框架依然面临推理攻击威胁，即攻击者通过截取分享的梯度更新值，利用对抗生成网络(GAN)等手段，尝试获取或恢复智能设备本地的隐私数据。

差分隐私(Differential Privacy，DP)是常用的隐私保护技术，抵御推理攻击时额外引入的计算、通信开销极小，在跨设备联邦学习场景中具有突出优势。同时，由于噪声的引入，其对全局模型的收敛速度及预测精度也会产生影响。目前，利用差分隐私增强联邦学习隐私性，核心挑战在于如何平衡数据隐私性与模型可用性，即如何在尽量不降低模型精度的前提下实现最佳隐私保护。考虑到联邦学习场景中采集数据类型的复杂性，针对不同类型数据(如数值、图像等)评估不同噪声机制、不同噪声添加阶段对全局模型精度的影响是实现上述平衡的重要环节。

本项目拟搭建融合差分隐私技术的联邦学习平台。在此基础上，任务核心围绕对比实验展开，即以未融合差分隐私技术的全局模型精度为基准，对比不同差分隐私方案对全局精度的影响。如：针对数值类型，在本地训练开始前添加拉普拉斯机制噪声(即直接对训练集进行 DP 处理)，相比于基准，所得全局模型精度如何变化。项目重心在对比实验设计及数据处理层面。

(隐私预算、动态分配、隐私性评估比较抽象，量化困难，只从可用性层面设计实验)

任务描述：

1. 搭建联邦学习框架；
2. 实现差分隐私模块；
3. 设计对比实验，整理分析数据。

二、项目实践内容

1. 本项目所使用的平台以及数据库版本

本项目基于python版本为3.9.21 ；pytorch版本为2.5.1；numpy版本为1.26.4；cuda-version 为 12.9；matplotlib 版本为 3.9.2；opacus 版本为 1.5.3。

以上资源包均配置在 anaconda 环境下。

采用的数据集为mnist数据集。

2. 差分隐私算法探究

差分隐私的不同机制：

①拉普拉斯机制：

原理：

在原始数据或查询结果中添加服从 Laplace 分布的噪声。Laplace 分布的概率密度函数为：

$$f(x|\mu,b)=\frac{1}{2b}\exp\left(-\frac{|x-\mu|}{b}\right)$$

其中，μ 是位置参数（通常为 0），b=ϵΔf 是尺度参数，Δf 是查询的敏感度，ϵ 是隐私预算。

特点：

简单高效：适用于实值查询（如求和、平均值）。

参数控制：通过调整 ϵ 控制隐私 - 准确性权衡（ϵ 越小，隐私保护越强，但噪声越大）。

应用广泛：是最常用的差分隐私机制之一。

②高斯机制：

原理：

在原始数据或查询结果中添加服从高斯分布（正态分布）的噪声。高斯机制使用 Rényi 差分隐私（RDP）或 (ε,δ)- 差分隐私，其中 δ 是一个很小的正数，表示隐私泄露的概率上限。

特点：

适合高维数据：在处理高维数据时比 Laplace 机制更高效。

需要双参数控制：通过 ε 和 δ 共同控制隐私保护程度。

数学性质优良：便于与深度学习结合（如梯度添加噪声）。

③瑞丽机制：

原理：

在原始数据或梯度中添加服从瑞利分布的噪声。瑞利分布的概率密度函数为：

$$f(x|\sigma)=\frac{x}{\sigma^2}\exp\left(-\frac{x^2}{2\sigma^2}\right),\quad x\geq 0$$

其中 σ 是尺度参数，与隐私预算 ϵ 相关。

特点：

重尾分布：噪声值较大的概率比高斯分布更高，提供更强的隐私保护。

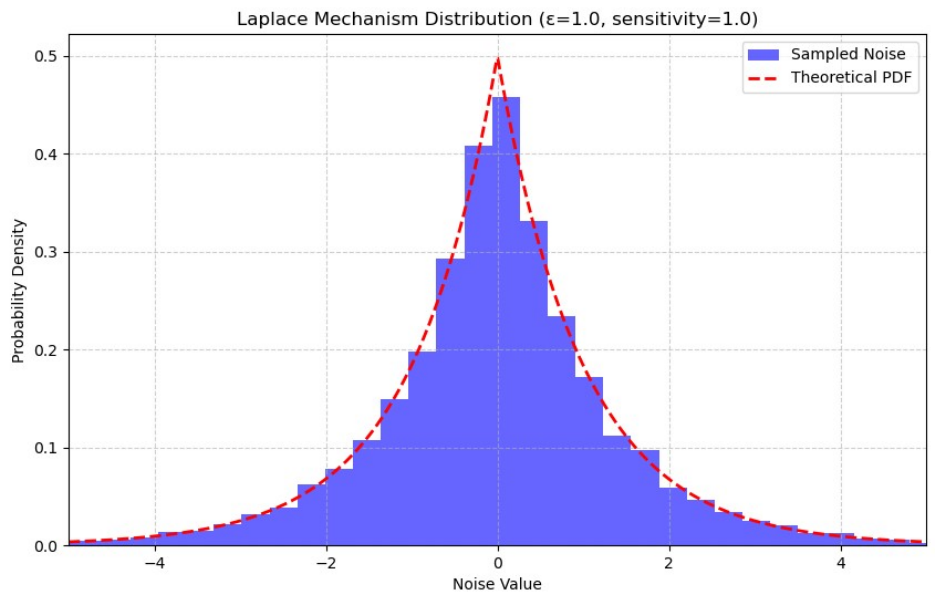
适用于梯度隐私：在联邦学习中常用于梯度扰动。

本项目主要根据瑞丽差分隐私算法进行实践，将瑞丽差分隐私算法与联邦学习算法相结合。

以下为三种机制下产生噪声的代码

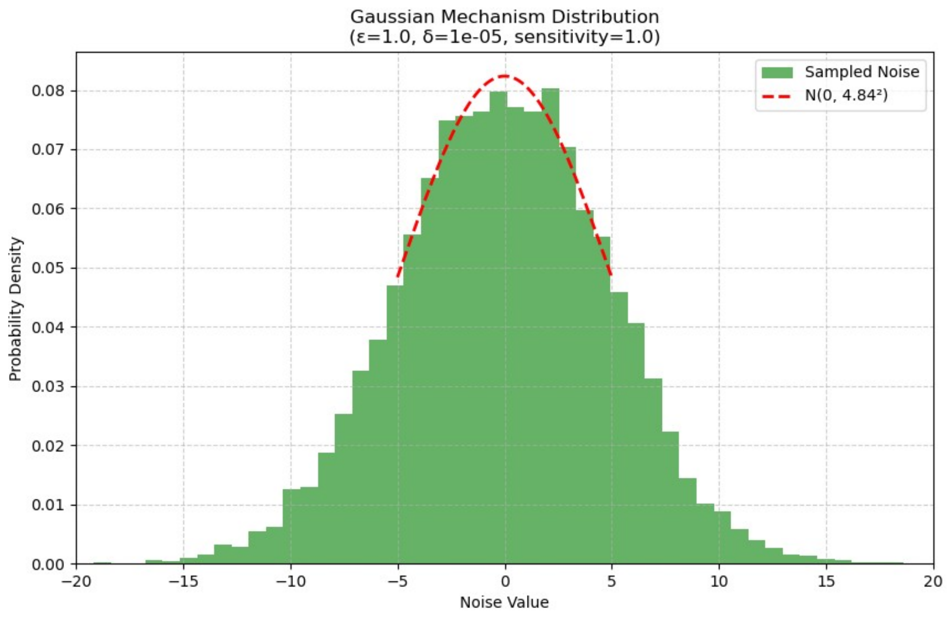
拉普拉斯机制示例代码：

```
    import numpy as np
def laplace_mechanism(data, epsilon, sensitivity=1.0):#输出拉普拉斯机制下的噪声
    noise = np.random.laplace(0, sensitivity / epsilon)
    return data + noise
```



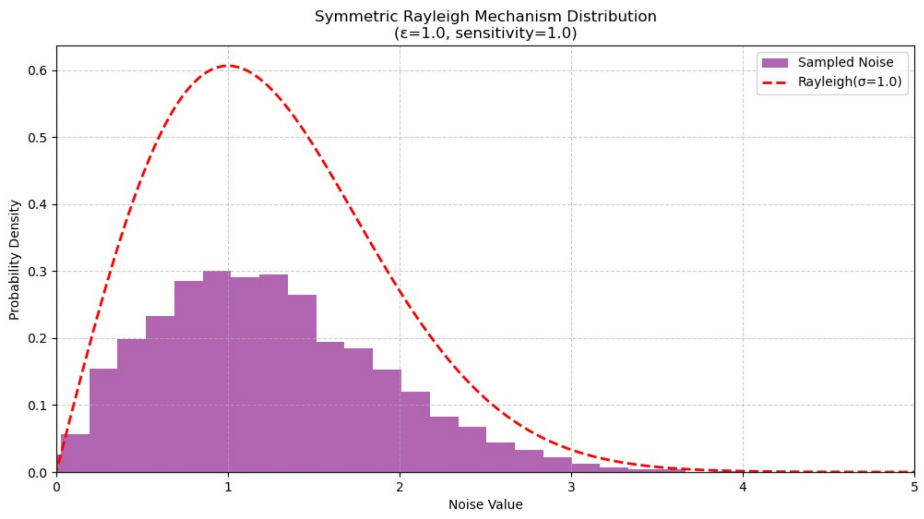
高斯机制示例代码：

```
        import numpy as np
def laplace_mechanism(data, epsilon, sensitivity=1.0):#输出高斯机制下的噪声
        sigma = np.sqrt(2 * np.log(1.25 / delta)) * sensitivity / epsilon
        noise = np.random.normal(0, sigma)
        return data + noise
```



瑞丽机制的示例代码：

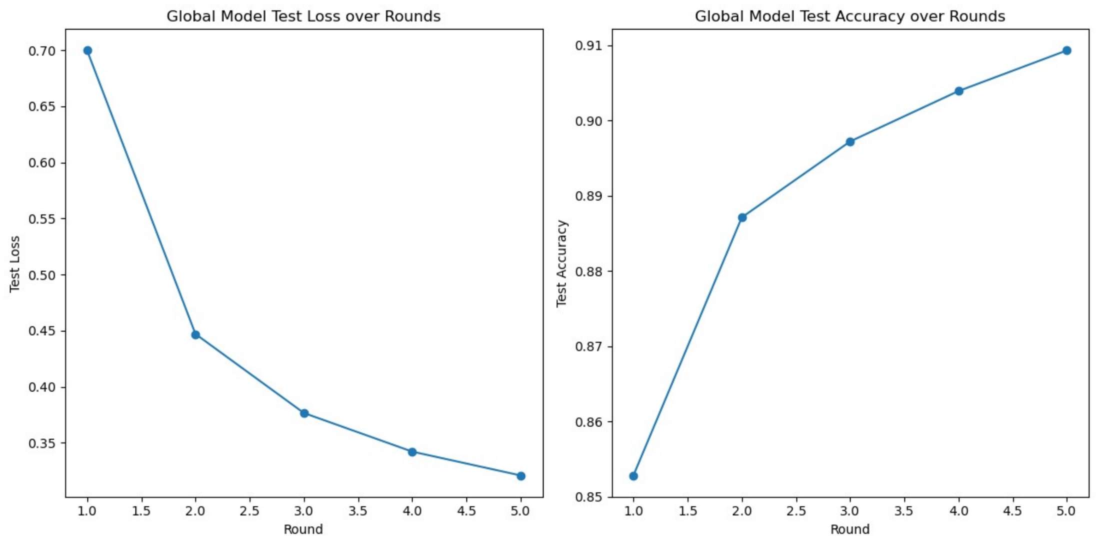
```
        import numpy as np
def laplace_mechanism(data, epsilon, sensitivity=1.0):#输出瑞丽机制下的噪声
        sigma = sensitivity / epsilon
        noise = torch.randn_like(gradient)
        noise = torch.norm(noise) * sigma
        return data + noise
```



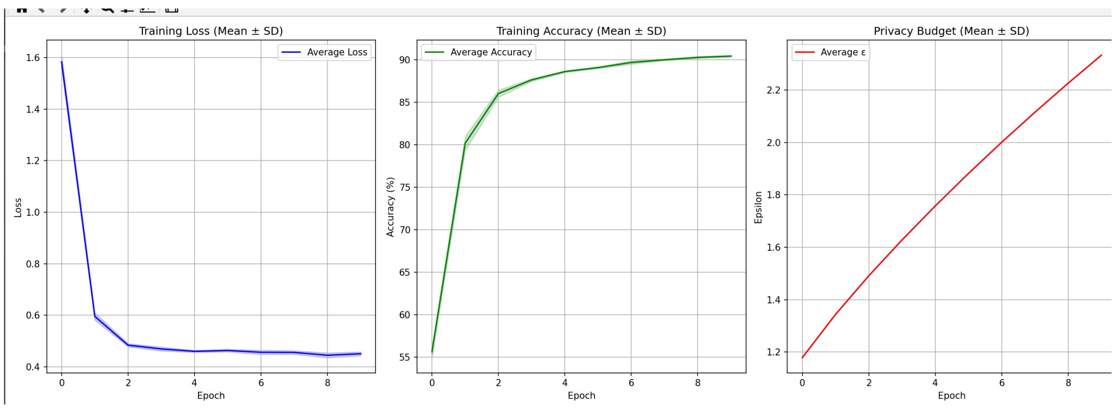
本项目采用瑞丽差分隐私机制进行计算，也可采用 opacus 数据库进行更加简易的使用瑞丽差分隐私机制进行包装，提高隐私的保护能力。

```
import torch
from torch import nn, optim
from torchvision import datasets, transforms
from torch.utils.data import DataLoader
from opacus import PrivacyEngine

# 设置随机种子
torch.manual_seed(42)


# 定义模型
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(784, 256)
        self.fc2 = nn.Linear(256, 128)
        self.fc3 = nn.Linear(128, 10)

    def forward(self, x):
        x = x.view(-1, 784)
        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

```
# 数据加载
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,))
])

train_dataset = datasets.MNIST('./data', train=True, download=True, transform=transform)
train_loader = DataLoader(train_dataset, batch_size=1024, shuffle=True)

# 初始化模型和优化器
model = Net()
optimizer = optim.SGD(model.parameters(), lr=0.05, momentum=0.9)
criterion = nn.CrossEntropyLoss()

# 初始化隐私引擎（Opacus v1.5.3）
privacy_engine = PrivacyEngine(
    accountant="rdp",  # 使用 RDP 会计师
    secure_mode=False
)

# 使训练过程私有
model, optimizer, train_loader = privacy_engine.make_private(
    module=model,
    optimizer=optimizer,
    data_loader=train_loader,
    noise_multiplier=1.1,
    max_grad_norm=1.0,
)


# 训练循环（Opacus v1.5.3）
def train(model, train_loader, optimizer, criterion, epochs, privacy_engine):
    model.train()
    for epoch in range(epochs):
        total_loss = 0
        correct = 0
        total = 0

        for data, target in train_loader:
            optimizer.zero_grad()
            output = model(data)
            loss = criterion(output, target)
            loss.backward()
            optimizer.step()

            total_loss += loss.item()
            pred = output.argmax(dim=1, keepdim=True)
            correct += pred.eq(target.view_as(pred)).sum().item()
            total += data.size(0)

        # 获取隐私预算（新版本）
        epsilon = privacy_engine.get_epsilon(delta=1e-5)

        print(f'Epoch: {epoch + 1}, Loss: {total_loss / len(train_loader):.4f}, '
              f'Accuracy: {100. * correct / total:.2f}%, '
              f'(ε = {epsilon:.2f}, δ = 1e-5)')


# 执行训练
train(model,train_loader,optimizer,criterion,epochs=5,privacy_engine=privacy_engin
e)
```

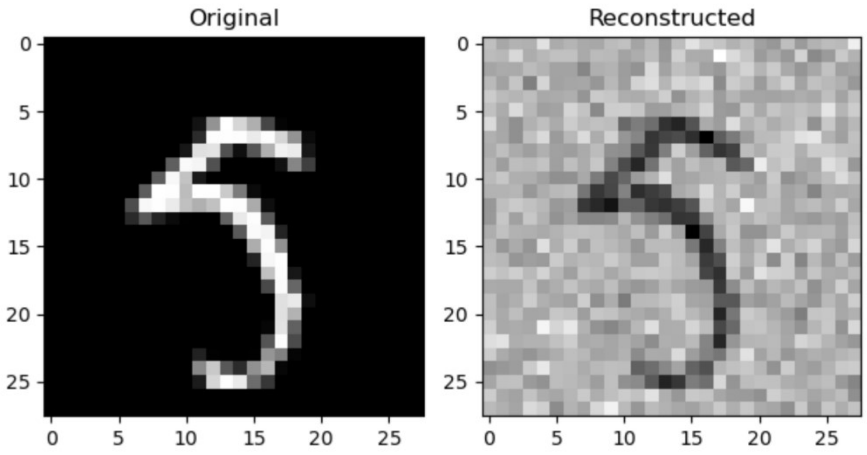


在本地运行的瑞丽机制可做到如上的效果。

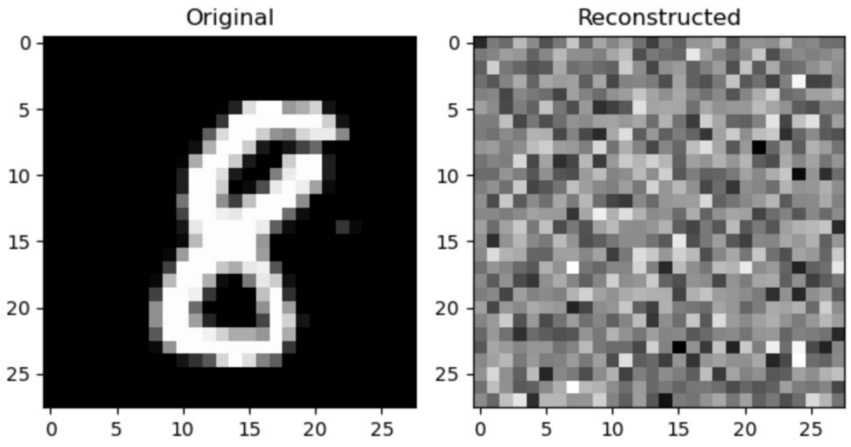


通过多轮运行，我们也能看到由于差分隐私随着每轮运行（共进行十轮）的进行，由于其具有的累加性质，每一次运算都会线性的增加隐私预算的大小。

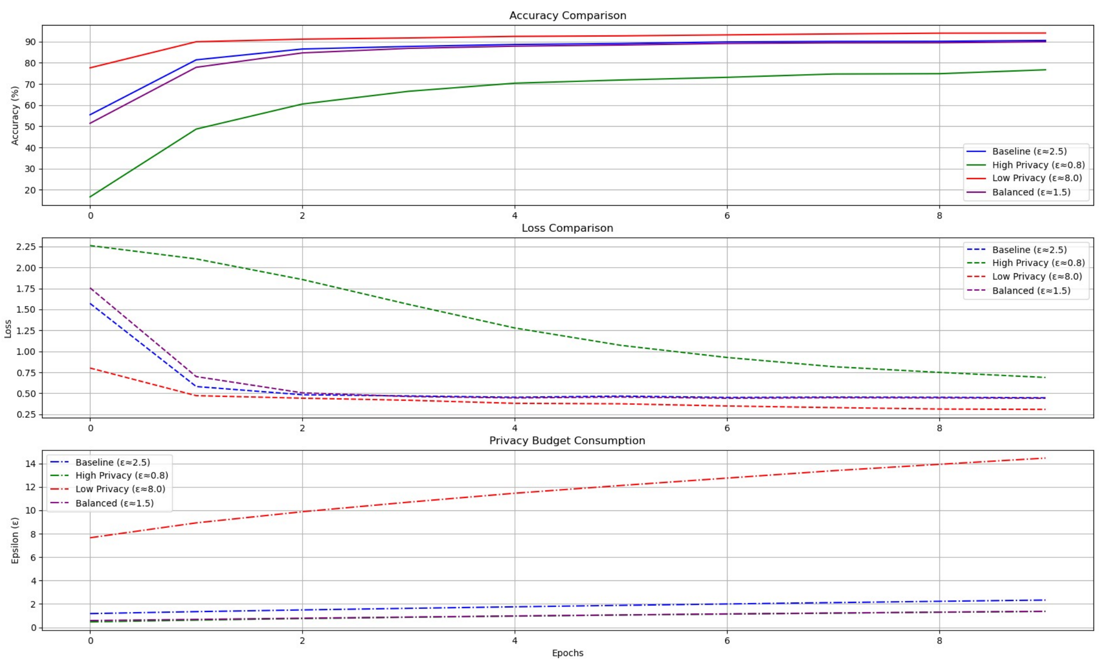
观察本地经过处理后的图像，可得到：



降低隐私预算会提高差分隐私的保护力度但会使得重建图像所包含的信息量下降。



隐私预算量的调整也体现在神经网络学习的各个部分，如下图：



可以看到每次运算时的隐私预算的大小足以影响函数的准确性，学习速率，而保护能力也能通过整个模型的隐私预算得到体现。

3. 联邦学习平台搭建的探究

联邦学习分为三个部分：主服务器学习，客户端学习，整体的人工智能网络框架。

以下为本项目所搭建的联邦学习框架：

```
import torch
from torch import nn, optim
from torchvision import datasets, transforms
from torch.utils.data import DataLoader, Subset
import numpy as np
import random
from typing import List, Dict, Tuple
import matplotlib.pyplot as plt
import copy

# 设置随机种子
torch.manual_seed(42)
np.random.seed(42)
random.seed(42)


# 梯度裁剪，防止梯度爆炸
def clip_gradients(model: nn.Module, max_norm: float) -> None:
    """梯度裁剪，防止梯度爆炸"""
    nn.utils.clip_grad_norm_(model.parameters(), max_norm)
```

```
# 定义模型
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(784, 256)
        self.fc2 = nn.Linear(256, 128)
        self.fc3 = nn.Linear(128, 10)

    def forward(self, x):
        x = x.view(-1, 784)
        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
        x = self.fc3(x)
        return x


# 数据集加载函数，每个客户端有不同的数据集
def load_data(client_id, num_clients=10):
    transform = transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize((0.1307,), (0.3081,))
    ])

    train_dataset = datasets.MNIST('./data', train=True, download=True,
transform=transform)
    test_dataset = datasets.MNIST('./data', train=False, transform=transform)

    # 为每个客户端分配不同的数据子集
    indices = list(range(len(train_dataset)))
    random.seed(client_id)
    random.shuffle(indices)

    # 平均分割数据集给每个客户端
    client_size = len(indices) // num_clients
    start_idx = client_id * client_size
    end_idx = start_idx + client_size if client_id < num_clients - 1 else len(indices)

    client_indices = indices[start_idx:end_idx]
    train_sampler = torch.utils.data.SubsetRandomSampler(client_indices)

    train_loader = DataLoader(train_dataset, batch_size=64, sampler=train_sampler)
    test_loader = DataLoader(test_dataset, batch_size=1000)

    return train_loader, test_loader
```

```
# 本地训练函数
def local_train(model: nn.Module, train_loader: DataLoader, epochs: int,
                lr: float, max_grad_norm: float) -> Dict[str, torch.Tensor]:
    model.train()
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.SGD(model.parameters(), lr=lr)

    for epoch in range(epochs):
        for data, target in train_loader:
            optimizer.zero_grad()
            output = model(data)
            loss = criterion(output, target)
            loss.backward()

            # 梯度裁剪
            clip_gradients(model, max_grad_norm)

            optimizer.step()

    # 返回更新后的模型参数
    return model.state_dict()


# 计算模型参数差异（梯度）
def compute_model_delta(global_model: nn.Module, local_model: Dict[str, torch.Tensor]) -> Dict[str, torch.Tensor]:
    delta = {}
    global_state = global_model.state_dict()
    for name, param in local_model.items():
        delta[name] = param - global_state[name]
    return delta


# 模型聚合函数
def aggregate_models(global_model: nn.Module, client_deltas: List[Dict[str, torch.Tensor]]) -> None:
    ”””
    聚合来自多个客户端的模型更新
    ”””
    global_state = global_model.state_dict()

    # 计算平均梯度
    avg_delta = {}
    for name in global_state.keys():
        # 对所有客户端的梯度求平均
        avg_delta[name] = torch.stack([delta[name] for delta in client_deltas], dim=0).mean(dim=0)

    # 更新全局模型
    with torch.no_grad():
        for name, param in global_state.items():
            param.add_(avg_delta[name])

    global_model.load_state_dict(global_state)


# 测试模型
def test_model(model: nn.Module, test_loader: DataLoader) -> Tuple[float, float]:
    model.eval()
    test_loss = 0
    correct = 0
    criterion = nn.CrossEntropyLoss()

    with torch.no_grad():
        for data, target in test_loader:
            output = model(data)
            test_loss += criterion(output, target).item()
            pred = output.argmax(dim=1, keepdim=True)
            correct += pred.eq(target.view_as(pred)).sum().item()

    test_loss /= len(test_loader.dataset)
    accuracy = 100. * correct / len(test_loader.dataset)

    return test_loss, accuracy


# 联邦学习主函数
def federated_learning(
        num_clients: int = 10,
        num_rounds: int = 10,
        local_epochs: int = 1,
        client_fraction: float = 0.5,
        learning_rate: float = 0.01,
        max_grad_norm: float = 1.0,   # 梯度裁剪阈值
        device: str = 'cpu'
):
```

```
# 初始化全局模型
global_model = Net().to(device)

# 加载数据
train_loaders = []
test_loaders = []
for client_id in range(num_clients):
    train_loader, test_loader = load_data(client_id, num_clients)
    train_loaders.append(train_loader)
    test_loaders.append(test_loader)

# 用于记录结果
global_test_losses = []
global_accuracies = []

# 联邦学习训练过程
for round in range(num_rounds):
    print(f"Round {round + 1}/{num_rounds}")

    # 选择参与本轮的客户端
    num_selected = max(1, int(client_fraction * num_clients))
    selected_clients = random.sample(range(num_clients), num_selected)

    # 客户端本地训练
    client_deltas = []
    for client_id in selected_clients:
        # 创建本地模型
        local_model = Net().to(device)
        local_model.load_state_dict(global_model.state_dict())

        # 本地训练
        local_state = local_train(
            model=local_model,
            train_loader=train_loaders[client_id],
            epochs=local_epochs,
            lr=learning_rate,
            max_grad_norm=max_grad_norm
        )

        # 计算模型参数差异（梯度）
        delta = compute_model_delta(global_model, local_state)
        client_deltas.append(delta)

    # 服务器聚合模型更新
```

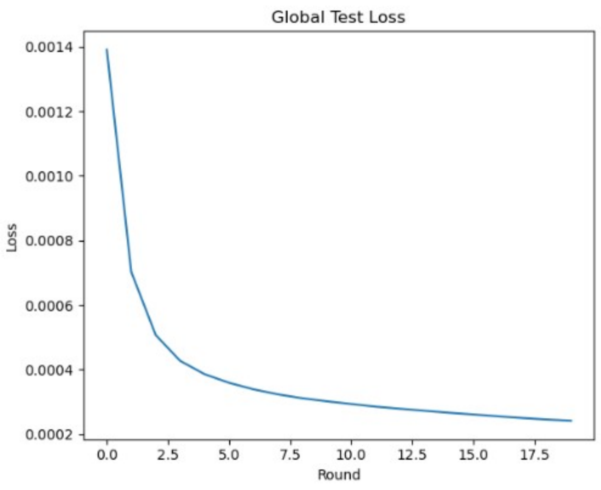
```
        aggregate_models(
            global_model=global_model,
            client_deltas=client_deltas
        )

        # 在全局测试集上评估模型
        test_loss, accuracy = test_model(global_model, test_loaders[0])
        global_test_losses.append(test_loss)
        global_accuracies.append(accuracy)

        print(f"Test Loss: {test_loss:.4f}, Accuracy: {accuracy:.2f}%")

    # 绘制结果
    plot_results(global_test_losses, global_accuracies)

    # 返回最终结果
    return {
        'model': global_model,
        'final_loss': global_test_losses[-1],
        'final_accuracy': global_accuracies[-1],
        'history': {
            'losses': global_test_losses,
            'accuracies': global_accuracies
        },
        'params': {
            'num_clients': num_clients,
            'num_rounds': num_rounds,
            'local_epochs': local_epochs,
            'client_fraction': client_fraction,
            'learning_rate': learning_rate,
            'max_grad_norm': max_grad_norm
        }
    }


# 绘制训练结果
def plot_results(losses, accuracies):
    plt.figure(figsize=(12, 5))

    plt.subplot(1, 2, 1)
    plt.plot(losses)
    plt.title('Global Test Loss')
    plt.xlabel('Round')
    plt.ylabel('Loss')
    plt.subplot(1, 2, 2)
    plt.plot(accuracies)
    plt.title('Global Test Accuracy')
    plt.xlabel('Round')
    plt.ylabel('Accuracy (%)')

    plt.tight_layout()
    plt.show()


# 主函数
if __name__ == "__main__":
    # 训练参数
    params = {
        'num_clients': 10,
        'num_rounds': 20,
        'local_epochs': 2,
        'client_fraction': 0.5,
        'learning_rate': 0.01,
        'max_grad_norm': 1.0  # 梯度裁剪阈值
    }

    # 运行联邦学习
    result = federated_learning(**params)

    # 保存最终模型
    torch.save(result['model'].state_dict(), "federated_model.pth")

    # 显示关键参数和最终结果
    print("\n===== 训练参数 =====")
    for param, value in params.items():
        print(f"{param}: {value}")

    print("\n===== 最终结果 =====")
    print(f"测试损失: {result['final_loss']:.4f}")
    print(f"测试准确率: {result['final_accuracy']:.2f}%")

    # 评估模型性能
    if result['final_accuracy'] > 95:
        print("模型性能: 优秀 (准确率 > 95%)")
    elif result['final_accuracy'] > 90:
        print("模型性能: 良好 (准确率 90-95%)")
    else:
        print("模型性能: 需要改进 (准确率 < 90%)")
```

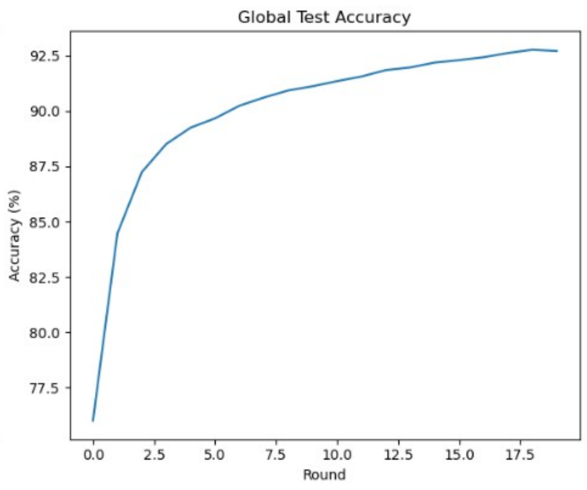
可以看到，本项目使用了一个三层线性层所搭建的简单视觉识别网络，将mnist 数据集随机打乱拆分给互不关联的客户端进行学习训练，客户端将每次学习所获得的梯度信息上传给主服务器，在主服务器中进行参数聚合，随后更新参数。

这就是一个简单的联邦学习平台，他所能提供的保护在于不直接将训练用的数据集上传至主服务器中训练，而是传输每次训练所得到的梯度，在主程序中通过参数聚合来形成正确的模型梯度。

以上实验程序运行结果如下：

```
在实验中的训练参数为：
    num_clients: 10
    num_rounds: 20
    local_epochs: 2
    client_fraction: 0.5
    learning_rate: 0.01
    max_grad_norm: 1.0
结果：
   测试损失: 0.0002
   测试准确率: 92.71%
   模型性能: 良好 (准确率 90-95%)
```



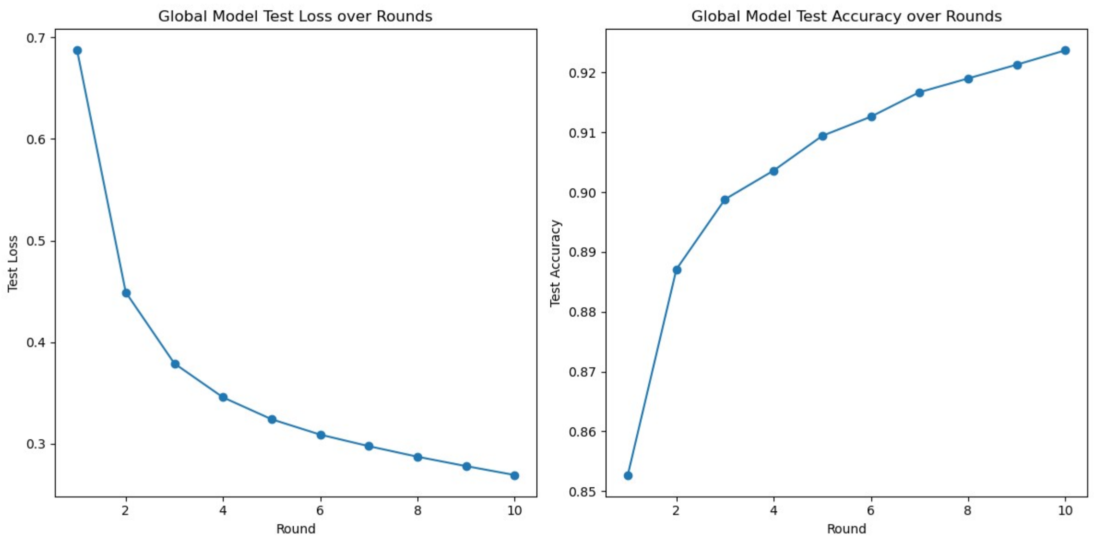


可以看到在未添加差分隐私保护的情况下，本联邦学习框架对于此数据集的学习能力较为良好，但这种简单的联邦分布式学习对于攻击而言较为薄弱（如通过投毒算法攻击数据集的可靠性或是通过客户端传输的梯度反向预测出数据集的信息，造成用户的隐私泄露）

针对投毒算法的联邦学习保护能力。

```
import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import datasets, transforms
import matplotlib.pyplot as plt
import numpy as np
import copy
import random


# 定义神经网络
class SimpleNN(nn.Module):
    def __init__(self):
        super(SimpleNN, self).__init__()
        self.fc1 = nn.Linear(28 * 28, 128)
        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):
        x = x.view(-1, 28 * 28)
        x = torch.relu(self.fc1(x))
        x = self.fc2(x)
        return x


# 客户端类
class Client:
    def __init__(self, client_id, malicious=False):
        self.client_id = client_id
        self.malicious = malicious
        self.train_loader, self.test_loader = self.load_data()
        self.local_model = SimpleNN()

    def load_data(self):
        transform = transforms.Compose([transforms.ToTensor()])
        train_dataset = datasets.MNIST('./data', train=True, download=True,
transform=transform)
        test_dataset = datasets.MNIST('./data', train=False, download=True,
transform=transform)

        random.seed(self.client_id)
        indices = list(range(len(train_dataset)))
        random.shuffle(indices)
        split = int(0.8 * len(indices))
        train_sampler = torch.utils.data.SubsetRandomSampler(indices[:split])
        test_loader = torch.utils.data.DataLoader(test_dataset, batch_size=64,
shuffle=False)
        return (
            torch.utils.data.DataLoader(train_dataset, batch_size=64,
sampler=train_sampler),
                test_loader
            )

    def local_train(self, global_model, lr=0.01):
        criterion = nn.CrossEntropyLoss()
        optimizer = optim.SGD(global_model.parameters(), lr=lr)
        global_model.train()

        for data, target in self.train_loader:
            optimizer.zero_grad()
            output = global_model(data)
            loss = criterion(output, target)
            loss.backward()

            if self.malicious:
                return {n: torch.rand_like(p.grad) for n, p in
global_model.named_parameters()}
            else:
                return {n: p.grad.clone() for n, p in
global_model.named_parameters()}


# 联邦学习框架
class FLExperiment:
    def __init__(self, params):
        self.params = params
        self.clients = []
        self.init_clients()
        self.global_model = SimpleNN()
        self.history = {'loss': [], 'acc': []}

    def init_clients(self):
        malicious_ids = random.sample(
            range(self.params['num_clients']),
            self.params['malicious_clients'])
        for cid in range(self.params['num_clients']):
            self.clients.append(Client(cid, cid in malicious_ids))

    def run(self):
        for _ in range(self.params['num_rounds']):
            selected = random.sample(self.clients,
self.params['selected_clients_num'])

            # 梯度聚合
```

```
                agg_grads = None
                for client in selected:
                    grads = client.local_train(copy.deepcopy(self.global_model),
self.params['lr'])
                    if not agg_grads:
                        agg_grads = grads
                    else:
                        for k in agg_grads:
                            agg_grads[k] += grads[k]

                # 参数更新
                for k in agg_grads:
                    agg_grads[k] /= len(selected)
                    self.global_model.state_dict()[k] -= self.params['lr'] * agg_grads[k]

                # 评估
                loss, acc = self.evaluate()
                self.history['loss'].append(loss)
                self.history['acc'].append(acc)

    def evaluate(self):
        self.global_model.eval()
        criterion = nn.CrossEntropyLoss()
        total_loss, correct = 0, 0

        with torch.no_grad():
            for data, target in self.clients[0].test_loader:
                output = self.global_model(data)
                total_loss += criterion(output, target).item() * data.size(0)
                correct += (output.argmax(1) == target).sum().item()

        return (
            total_loss / len(self.clients[0].test_loader.dataset),
            correct / len(self.clients[0].test_loader.dataset)
        )


# 实验参数配置
param_configs = [
    {
        'name': 'Baseline',
        'num_clients': 10,
        'malicious_clients': 1,
        'selected_clients_num': 5,
        'num_rounds': 20,
        'lr': 0.01
    },
    {
        'name': 'More Malicious (20%)',
        'num_clients': 10,
        'malicious_clients': 2,
        'selected_clients_num': 5,
        'num_rounds': 20,
        'lr': 0.01
    },
    {
        'name': 'Higher LR',
        'num_clients': 10,
        'malicious_clients': 1,
        'selected_clients_num': 5,
        'num_rounds': 20,
        'lr': 0.05
    },
    {
        'name': 'Client Selection (8/10)',
        'num_clients': 10,
        'malicious_clients': 1,
        'selected_clients_num': 8,
        'num_rounds': 20,
        'lr': 0.01
    },
    {
        'name': 'More Clients',
        'num_clients': 20,
        'malicious_clients': 2,
        'selected_clients_num': 10,
        'num_rounds': 20,
        'lr': 0.01
    }
]

# 运行实验
num_experiments = 5
results = {cfg['name']: {'loss': [], 'acc': []} for cfg in param_configs}

for config in param_configs:
    print(f"\nRunning configuration: {config['name']}")
    all_loss = []
    all_acc = []

    for exp in range(num_experiments):
        print(f"Experiment {exp + 1}/{num_experiments}")
        fl = FLExperiment(config)
        fl.run()
        all_loss.append(fl.history['loss'])
        all_acc.append(fl.history['acc'])

    # 存储平均结果
    results[config['name']]['loss'] = np.mean(all_loss, axis=0)
    results[config['name']]['acc'] = np.mean(all_acc, axis=0)

# 可视化对比
plt.figure(figsize=(14, 6))
colors = ['b', 'g', 'r', 'c', 'm']
linestyles = ['-', '--', '-.', ':', '-']

# 准确率曲线
plt.subplot(1, 2, 1)
for idx, (name, data) in enumerate(results.items()):
    plt.plot(data['acc'],
             color=colors[idx],
             linestyle=linestyles[idx],
             linewidth=2,
             label=f"{name}")
plt.xlabel('Communication Round')
plt.ylabel('Accuracy')
plt.title('Accuracy Comparison')
plt.legend()
plt.grid(True)

# 损失曲线
plt.subplot(1, 2, 2)
for idx, (name, data) in enumerate(results.items()):
    plt.plot(data['loss'],
             color=colors[idx],
             linestyle=linestyles[idx],
             linewidth=2,
             label=f"{name}")
plt.xlabel('Communication Round')
plt.ylabel('Loss')
plt.title('Loss Comparison')
plt.legend()
plt.grid(True)

plt.tight_layout()
plt.show()
```



正如上文，在针对扰乱数据集的投毒算法攻击下，单纯的联邦学习架构，也可以通过逐次的拟合，达到一个较高的准确性，但是训练消耗要更大，在经过调整后可使二者的准确性相接近。

我们注意到，以上这种算法对于添加的干扰是对客户端传输梯度上加上随机的噪声，这相对于直接返回随机梯度给主服务器更加有效，也更加符合实际。

因此，结合差分隐私后，就可以实现如下的联邦学习，有效地提升联邦学习架构对恶意攻击的隐私性。

在差分隐私与本地训练的融合中，借助 Opacus 库的 PrivacyEngine 为本地训练过程构建隐私保护机制，其核心包含梯度裁剪与噪声注入两大环节：通过梯度裁剪阈值 max_grad_norm 参数限制单个样本的梯度范数，约束其对模型更新的影响范围，避免异常数据主导训练方向，同时为噪声注入提供规整的梯度基础；利用噪声乘数 noise_multiplier 控制高斯噪声强度并注入聚合梯度，通过混淆真实梯度信息满足差分隐私的严格数学定义，尽管噪声可能引入训练扰动或轻微降低模型精度，但这是保护数据隐私的必要权衡。

隐私预算消耗则通过前面提到的瑞丽差分隐私（RDP）会计动态跟踪，计算每轮训练的隐私参数 ε（设定 δ=1e-5），其中 ε 值越小表明隐私保护强度越高，系统通过这一机制在隐私安全性与模型训练效果之间实现量化平衡，确保每个客户端在独立训练时既能利用本地数据优化模型，又不泄露个体数据的敏感信息。

```
import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import datasets, transforms
import matplotlib.pyplot as plt
import copy
import random
from opacus import PrivacyEngine
```

```
import warnings


# 设置随机种子
torch.manual_seed(42)
random.seed(42)


# 定义模型
class SimpleNN(nn.Module):
    def __init__(self):
        super(SimpleNN, self).__init__()
        self.fc1 = nn.Linear(28 * 28, 128)
        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):
        x = x.view(-1, 28 * 28)
        x = torch.relu(self.fc1(x))
        x = self.fc2(x)
        return x


# 数据集加载函数
def load_data(client_id):
    transform = transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize((0.1307,), (0.3081,))
    ])
    train_dataset = datasets.MNIST(root='./data', train=True, download=True,
transform=transform)
    test_dataset = datasets.MNIST(root='./data', train=False, download=True,
transform=transform)

    random.seed(client_id)
    indices = list(range(len(train_dataset)))
    random.shuffle(indices)
    split = int(0.8 * len(indices))
    train_indices, val_indices = indices[:split], indices[split:]

    train_sampler = torch.utils.data.SubsetRandomSampler(train_indices)
    val_sampler = torch.utils.data.SubsetRandomSampler(val_indices)

    train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=64,
sampler=train_sampler)
```

```
    test_loader = torch.utils.data.DataLoader(test_dataset, batch_size=64, 
shuffle=False)
    return train_loader, test_loader


# 带差分隐私的本地训练函数，返回每轮隐私消耗
def local_train_with_dp(model, train_loader, epochs=1, lr=0.01, noise_multiplier=1.0, 
max_grad_norm=1.0):
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.SGD(model.parameters(), lr=lr)

    # 初始化隐私引擎
    privacy_engine = PrivacyEngine(
        accountant="rdp",
        secure_mode=False
    )

    model.train()   # 确保模型处于训练模式
    model, optimizer, train_loader = privacy_engine.make_private(
        module=model,
        optimizer=optimizer,
        data_loader=train_loader,
        noise_multiplier=noise_multiplier,
        max_grad_norm=max_grad_norm,
    )

    epsilon_before = privacy_engine.get_epsilon(delta=1e-5)

    model.train()
    for epoch in range(epochs):
        for data, target in train_loader:
            optimizer.zero_grad()
            output = model(data)
            loss = criterion(output, target)
            loss.backward()
            optimizer.step()

    epsilon_after = privacy_engine.get_epsilon(delta=1e-5)
    epsilon_consumed = epsilon_after - epsilon_before

    return epsilon_consumed


# 联邦学习过程
```

```
def federated_learning_with_dp(num_clients=10, num_rounds=10,
noise_multiplier=1.0, max_grad_norm=1.0):
    global_model = SimpleNN()

    train_loaders = []
    test_loader = None
    for client_id in range(num_clients):
        train_loader, test_loader_client = load_data(client_id)
        train_loaders.append(train_loader)
        if client_id == 0:
            test_loader = test_loader_client

    test_losses_history = []
    test_accs_history = []
    privacy_history = []

    for round in range(num_rounds):
        local_models = []
        epsilons = []

        for i in range(num_clients):
            local_model = copy.deepcopy(global_model)
            epsilon_consumed = local_train_with_dp(
                local_model,
                train_loaders[i],
                noise_multiplier=noise_multiplier,
                max_grad_norm=max_grad_norm
            )
            local_models.append(local_model.state_dict())
            epsilons.append(epsilon_consumed)

        # 参数聚合
        global_state_dict = global_model.state_dict()
        for key in global_state_dict.keys():
            global_state_dict[key] = torch.stack([local_models[i][key] for i in
range(num_clients)], 0).mean(0)

        global_model.load_state_dict(global_state_dict)

        # 测试全局模型
        global_model.eval()
        running_loss = 0.0
        correct = 0
        total = 0
        with torch.no_grad():
            for data, target in test_loader:
                output = global_model(data)
                loss = nn.CrossEntropyLoss()(output, target)

                running_loss += loss.item() * data.size(0)
                _, predicted = torch.max(output.data, 1)
                total += target.size(0)
                correct += (predicted == target).sum().item()

        test_loss = running_loss / len(test_loader.dataset)
        test_acc = correct / total
        max_epsilon = max(epsilons)
        privacy_history.append(max_epsilon)

        print(
            f'Round {round + 1}: Test Loss: {test_loss:.4f}, Test Accuracy:
{test_acc:.4f}, ε per round = {max_epsilon:.4f}')

    return global_model, test_losses_history, test_accs_history, privacy_history


# 运行联邦学习
global_model, test_losses, test_accs, privacy_history = federated_learning_with_dp(
    num_clients=10,
    num_rounds=10,
    noise_multiplier=1.0,
    max_grad_norm=1.0
)

# 绘制结果
plt.figure(figsize=(18, 6))

plt.subplot(1, 3, 1)
plt.plot(range(1, len(test_losses) + 1), test_losses, marker='o')
plt.xlabel('Round')
plt.ylabel('Test Loss')
plt.title('Global Model Test Loss')

plt.subplot(1, 3, 2)
plt.plot(range(1, len(test_accs) + 1), test_accs, marker='o')
plt.xlabel('Round')
plt.ylabel('Test Accuracy')
plt.title('Global Model Test Accuracy')
plt.subplot(1, 3, 3)
plt.plot(range(1, len(privacy_history) + 1), privacy_history, marker='o')
plt.xlabel('Round')
plt.ylabel('ε (Privacy Budget per round)')
plt.title('Privacy Budget Consumption per Round (δ = 1e-5)')

plt.tight_layout()
plt.show()

# 保存最终模型
torch.save(global_model.state_dict(), "federated_model_with_dp.pth")
```

代码通过三个关键环节将差分隐私无缝融入联邦学习框架：

1.客户端级隐私增强：在每个客户端的本地训练中，通过 Opacus 的 PrivacyEngine 对梯度进行裁剪和噪声注入，确保单个用户数据的隐私不被泄露。这种设计与联邦学习的“数据不动模型动”理念高度契合——原始数据始终保留在客户端，仅经过隐私保护处理的梯度或模型参数参与全局聚合。

2.聚合过程的隐私保留：服务器聚合客户端模型时，直接对已添加噪声的参数取平均，无需额外处理。这种方式既保持了联邦学习的分布式训练优势，又通过差分隐私的数学保证，防止攻击者通过逆向工程从聚合参数中恢复个体数据。

3.隐私预算的联邦化管理：每轮训练的隐私消耗（ε 值）独立计算，确保整个联邦学习过程的隐私保护强度可量化。

差分隐私参数的选择直接影响联邦学习的效果：

1.噪声乘数（noise_multiplier）：增大噪声可增强隐私保护，但会引入梯度偏差，降低模型收敛速度。联邦学习环境中，更多客户端的参与（num_clients）可部分抵消这种偏差。 通过聚合多个客户端的噪声梯度，系统获得更稳定的模型更新。

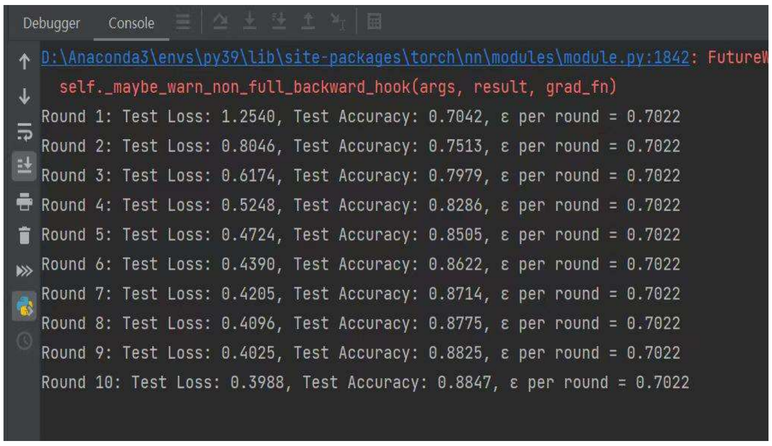
2.梯度裁剪阈值（max_grad_norm）：严格的梯度范数限制（较小的 max_grad_norm）不仅保护隐私，还能缓解联邦学习中常见的“客户端数据异质性”问题。通过约束异常样本的影响，使全局模型更关注共性特征。

3.轮次与客户端数量的权衡：增加训练轮数（num_rounds）会累积隐私消耗，但更多客户端的参与可降低单轮所需的噪声强度。这种配置在保证隐私预算可控的同时，利用分布式数据提升了模型泛化能力。

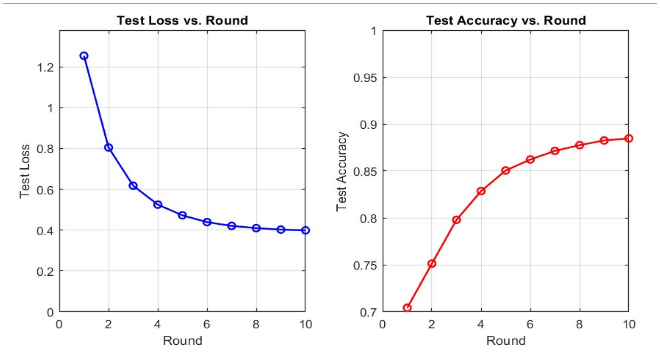
实验结果显示，在每轮均添加差分隐私噪声的情况下，模型仍实现了显著的准确率提升（从 70% 到 88%），证明实验通过合理的参数设置（如梯

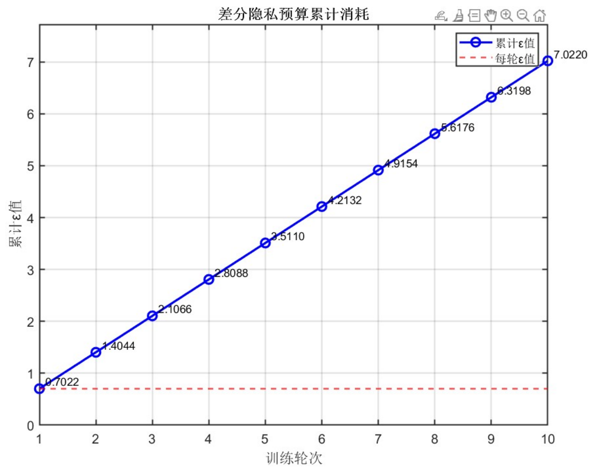
度裁剪与噪声强度）在隐私保护与模型性能间取得了较好平衡。此外，10 轮消耗的总隐私预算在 7.02 左右，在合规范围内。

不足之处在于，我们在使用瑞丽机制（Rényi Differential Privacy, RDP）时观察到每轮隐私预算（ ε 值）变化微小且几乎恒定（累计隐私预算变化随轮数线性增加），可能的原因包括：若代码中噪声乘数 noise_multiplier 过低或梯度裁剪阈值 max_grad_norm 过高，且每轮训练的数据分布、批大小保持一致，会导致每轮隐私消耗理论上相同；某些优化器（如 SGD）的确定性更新规则配合固定噪声参数，可能导致每轮梯度扰动相似，进而 ε 值变化不明显。



```
D:\Anaconda3\envs\py39\lib\site-packages\torch\nn\modules\module.py:1842: FutureW
  self._maybe_warn_non_full_backward_hook(args, result, grad_fn)
Round 1: Test Loss: 1.2540, Test Accuracy: 0.7042, ε per round = 0.7022
Round 2: Test Loss: 0.8046, Test Accuracy: 0.7513, ε per round = 0.7022
Round 3: Test Loss: 0.6174, Test Accuracy: 0.7979, ε per round = 0.7022
Round 4: Test Loss: 0.5248, Test Accuracy: 0.8286, ε per round = 0.7022
Round 5: Test Loss: 0.4724, Test Accuracy: 0.8505, ε per round = 0.7022
Round 6: Test Loss: 0.4390, Test Accuracy: 0.8622, ε per round = 0.7022
Round 7: Test Loss: 0.4205, Test Accuracy: 0.8714, ε per round = 0.7022
Round 8: Test Loss: 0.4096, Test Accuracy: 0.8775, ε per round = 0.7022
Round 9: Test Loss: 0.4025, Test Accuracy: 0.8825, ε per round = 0.7022
Round 10: Test Loss: 0.3988, Test Accuracy: 0.8847, ε per round = 0.7022
```





4. 对于差分隐私的攻击方法

针对差分隐私攻击主要有三种攻击：第一个是风险分析，第二个是攻击代码，第三个是模型训练与攻击。

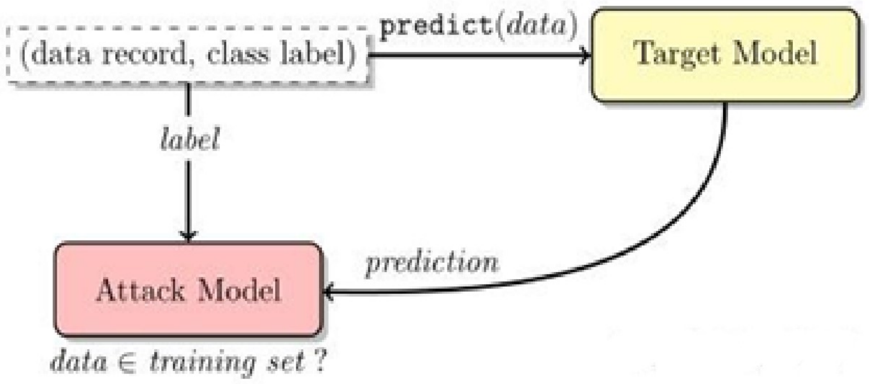
本次实验中，我们使用成员推理攻击（Membership Inference Attack, MIA） 来对训练模型进行攻击。

成员推理攻击（Membership Inference Attack, MIA）是一种针对机器学习模型的隐私攻击手段，其核心目标是推断某个特定样本是否被用于训练目标模型。攻击者通过分析模型的输出或中间状态，结合统计或机器学习方法，判断某样本是否属于模型的训练集。

① 成员推理攻击的攻击机制

攻击者向目标模型提交查询样本，收集模型的预测结果（如置信度分数、类别概率分布）。从模型的预测结果中提取可用于区分“训练集成员”与“非成员”的特征，如置信度（Confidence）、预测熵（Entropy）。然后训练一个二分类模型，用于根据提取的特征判断样本是否为训练集成员。

然而，在攻击模型的训练中，既需要样本真实 label，又需要目标模型预测置信度向量，这样过于苛刻的要求在现实中是很难实现的。因此，Shokri 等人十分巧妙的提出了一个核心思想——影子模型（shadow model）。攻击者基于对目标模型的已知信息，训练多个与目标模型相似的“影子模型”，并记录它们的输入输出对。利用影子模型的输出构建训练集，标记哪些样本是影子模型的成员和非成员，然后基于所生成的训练数据，训练攻击模型学习区分成员与非成员的模式。最后攻击者向目标模型提交待检测的样本，提取特征后输入攻击模型，得到样本是否为成员的攻击结果。



② 成员推理攻击的实现

使用 MIA 攻击瑞丽差分隐私保护的模型，并进行隐私风险评估。

推理攻击部分：

```
import numpy as np
import math

class black_box_benchmarks(object):

    def __init__(self, shadow_train_performance, shadow_test_performance,
                 target_train_performance, target_test_performance,
num_classes):
		'''
		each input contains both model predictions (shape: num_data*num_classes)
and ground-truth labels.
		'''
		self.num_classes = num_classes

		self.s_tr_outputs, self.s_tr_labels = shadow_train_performance
		self.s_te_outputs, self.s_te_labels = shadow_test_performance
		self.t_tr_outputs, self.t_tr_labels = target_train_performance
		self.t_te_outputs, self.t_te_labels = target_test_performance

		self.s_tr_corr = (np.argmax(self.s_tr_outputs,
axis=1)==self.s_tr_labels).astype(int)
		self.s_te_corr = (np.argmax(self.s_te_outputs,
axis=1)==self.s_te_labels).astype(int)
		self.t_tr_corr = (np.argmax(self.t_tr_outputs,
axis=1)==self.t_tr_labels).astype(int)
		self.t_te_corr = (np.argmax(self.t_te_outputs,
axis=1)==self.t_te_labels).astype(int)

		self.s_tr_conf = np.array([self.s_tr_outputs[i, self.s_tr_labels[i]] for i in
range(len(self.s_tr_labels))])
		self.s_te_conf = np.array([self.s_te_outputs[i, self.s_te_labels[i]] for i in
range(len(self.s_te_labels))])
		self.t_tr_conf = np.array([self.t_tr_outputs[i, self.t_tr_labels[i]] for i in
range(len(self.t_tr_labels))])
		self.t_te_conf = np.array([self.t_te_outputs[i, self.t_te_labels[i]] for i in
range(len(self.t_te_labels))])

		self.s_tr_entr = self._entr_comp(self.s_tr_outputs)
		self.s_te_entr = self._entr_comp(self.s_te_outputs)
		self.t_tr_entr = self._entr_comp(self.t_tr_outputs)
		self.t_te_entr = self._entr_comp(self.t_te_outputs)

		self.s_tr_m_entr = self._m_entr_comp(self.s_tr_outputs, self.s_tr_labels)
		self.s_te_m_entr = self._m_entr_comp(self.s_te_outputs, self.s_te_labels)
		self.t_tr_m_entr = self._m_entr_comp(self.t_tr_outputs, self.t_tr_labels)
		self.t_te_m_entr = self._m_entr_comp(self.t_te_outputs, self.t_te_labels)


	def _log_value(self, probs, small_value=1e-30):
		return -np.log(np.maximum(probs, small_value))
```

```
    def _entr_comp(self, probs):
        return np.sum(np.multiply(probs, self._log_value(probs)),axis=1)

    def _m_entr_comp(self, probs, true_labels):
        log_probs = self._log_value(probs)
        reverse_probs = 1-probs
        log_reverse_probs = self._log_value(reverse_probs)
        modified_probs = np.copy(probs)
        modified_probs[range(true_labels.size), true_labels] =
reverse_probs[range(true_labels.size), true_labels]
        modified_log_probs = np.copy(log_reverse_probs)
        modified_log_probs[range(true_labels.size), true_labels] =
log_probs[range(true_labels.size), true_labels]
        return np.sum(np.multiply(modified_probs, modified_log_probs),axis=1)

    def _thre_setting(self, tr_values, te_values):
        value_list = np.concatenate((tr_values, te_values))
        thre, max_acc = 0, 0
        for value in value_list:
            tr_ratio = np.sum(tr_values>=value)/(len(tr_values)+0.0)
            te_ratio = np.sum(te_values<value)/(len(te_values)+0.0)
            acc = 0.5*(tr_ratio + te_ratio)
            if acc > max_acc:
                thre, max_acc = value, acc
        return thre

    def _mem_inf_via_corr(self):
        # perform membership inference attack based on whether the input is
correctly classified or not
        t_tr_acc = np.sum(self.t_tr_corr)/(len(self.t_tr_corr)+0.0)
        t_te_acc = np.sum(self.t_te_corr)/(len(self.t_te_corr)+0.0)
        mem_inf_acc = 0.5*(t_tr_acc + 1 - t_te_acc)
        print('For membership inference attack via correctness, the attack acc is
{acc1:.3f}, with train acc {acc2:.3f} and test acc
{acc3:.3f}'.format(acc1=mem_inf_acc, acc2=t_tr_acc, acc3=t_te_acc) )
        return

    def _mem_inf_thre(self, v_name, s_tr_values, s_te_values, t_tr_values,
t_te_values):
        # perform membership inference attack by thresholding feature values: the
feature can be prediction confidence,
        # (negative) prediction entropy, and (negative) modified entropy
        t_tr_mem, t_te_non_mem = 0, 0
        for num in range(self.num_classes):
            thre = self._thre_setting(s_tr_values[self.s_tr_labels==num],
s_te_values[self.s_te_labels==num])
            t_tr_mem += np.sum(t_tr_values[self.t_tr_labels==num]>=thre)
            t_te_non_mem += np.sum(t_te_values[self.t_te_labels==num]<thre)
        mem_inf_acc = 0.5*(t_tr_mem/(len(self.t_tr_labels)+0.0) +
t_te_non_mem/(len(self.t_te_labels)+0.0))
        print('For membership inference attack via {n}, the attack acc is
{acc:.3f}'.format(n=v_name,acc=mem_inf_acc))
        return



    def _mem_inf_benchmarks(self, all_methods=True, benchmark_methods=[]):
        if (all_methods) or ('correctness' in benchmark_methods):
            self._mem_inf_via_corr()
        if (all_methods) or ('confidence' in benchmark_methods):
            self._mem_inf_thre('confidence', self.s_tr_conf, self.s_te_conf,
self.t_tr_conf, self.t_te_conf)
        if (all_methods) or ('entropy' in benchmark_methods):
            self._mem_inf_thre('entropy', -self.s_tr_entr, -self.s_te_entr,
-self.t_tr_entr, -self.t_te_entr)
        if (all_methods) or ('modified entropy' in benchmark_methods):
            self._mem_inf_thre('modified entropy', -self.s_tr_m_entr,
-self.s_te_m_entr, -self.t_tr_m_entr, -self.t_te_m_entr)

        return
```

风险评估部分：

```
import numpy as np
import matplotlib.pyplot as plt

def distrs_compute(tr_values, te_values, tr_labels, te_labels, num_bins=5,
log_bins=True, plot_name=None):

        ### function to compute and plot the normalized histogram for both training and
test values class by class.
        ### we recommand using the log scale to plot the distribution to get
better-behaved distributions.

        num_classes = len(set(tr_labels))
        sqr_num = int(np.ceil(np.sqrt(num_classes)))
        tr_distrs, te_distrs, all_bins = [], [], []

        plt.figure(figsize = (15,15))
        plt.rc('font', family='serif', size=10)
        plt.rc('axes', linewidth=2)

        for i in range(num_classes):
                tr_list, te_list = tr_values[tr_labels==i], te_values[te_labels==i]
                if log_bins:
                        # when using log scale, avoid very small number close to 0
                        small_delta = 1e-10
                        tr_list[tr_list<=small_delta] = small_delta
                        te_list[te_list<=small_delta] = small_delta
                n1, n2 = np.sum(tr_labels==i), np.sum(te_labels==i)
                all_list = np.concatenate((tr_list, te_list))
                max_v, min_v = np.amax(all_list), np.amin(all_list)

                plt.subplot(sqr_num, sqr_num, i+1)
                if log_bins:
                        bins = np.logspace(np.log10(min_v), np.log10(max_v),num_bins+1)
                        weights = np.ones_like(tr_list)/float(len(tr_list))
                        h1, _,_ = plt.hist(tr_list,bins=bins,facecolor='b',weights=weights,alpha
= 0.5)
                        plt.gca().set_xscale("log")
                        weights = np.ones_like(te_list)/float(len(te_list))
                        h2, _, _ = plt.hist(te_list,bins=bins,facecolor='r',weights=weights,alpha
= 0.5)
                        plt.gca().set_xscale("log")
                else:
                        bins = np.linspace(min_v, max_v,num_bins+1)
                        weights = np.ones_like(tr_list)/float(len(tr_list))
                        h1, _,_ = plt.hist(tr_list,bins=bins,facecolor='b',weights=weights,alpha
= 0.5)
                        weights = np.ones_like(te_list)/float(len(te_list))
                        h2, _, _ = plt.hist(te_list,bins=bins,facecolor='r',weights=weights,alpha
= 0.5)
                tr_distrs.append(h1)
                te_distrs.append(h2)
                all_bins.append(bins)
        if plot_name == None:
                plot_name='./tmp'
        plt.savefig(plot_name+'.png', bbox_inches='tight')
        tr_distrs, te_distrs, all_bins = np.array(tr_distrs), np.array(te_distrs),
np.array(all_bins)
        return tr_distrs, te_distrs, all_bins


def risk_score_compute(tr_distrs, te_distrs, all_bins, data_values, data_labels):
    ### Given training and test distributions (obtained from the shadow classifier),
    ### compute the corresponding privacy risk score for training points (of the
target classifier).

    def find_index(bins, value):
        # for given n bins (n+1 list) and one value, return which bin includes the
value
        if value>=bins[-1]:
            return len(bins)-2 # when value is larger than any bins, we assign the
last bin
        if value<=bins[0]:
            return 0    # when value is smaller than any bins, we assign the first bin
        return np.argwhere(bins<=value)[-1][0]

    def score_calculate(tr_distr, te_distr, ind):
        if tr_distr[ind]+te_distr[ind] != 0:
            return tr_distr[ind]/(tr_distr[ind]+te_distr[ind])
        else: # when both distributions have 0 probabilities, we find the nearest bin
with non-zero probability
            for t_n in range(1, len(tr_distr)):
                t_ind = ind-t_n
                if t_ind>=0:
                    if tr_distr[t_ind]+te_distr[t_ind] != 0:
                        return tr_distr[t_ind]/(tr_distr[t_ind]+te_distr[t_ind])
                t_ind = ind+t_n
                if t_ind<len(tr_distr):
                    if tr_distr[t_ind]+te_distr[t_ind] != 0:
                        return tr_distr[t_ind]/(tr_distr[t_ind]+te_distr[t_ind])

    risk_score = []
    for i in range(len(data_values)):
        c_value, c_label = data_values[i], data_labels[i]
        c_tr_distr, c_te_distr, c_bins = tr_distrs[c_label], te_distrs[c_label],
all_bins[c_label]
        c_index = find_index(c_bins, c_value)
        c_score = score_calculate(c_tr_distr, c_te_distr, c_index)
        risk_score.append(c_score)
    return np.array(risk_score)

def calculate_risk_score(tr_values, te_values, tr_labels, te_labels, data_values,
data_labels,
                         num_bins=5, log_bins=True):
```

```
    ########### tr_values, te_values, tr_labels, te_labels are from shadow
classifier's training and test data
    ########### data_values, data_labels are from target classifier's training data
    ########### potential choice for the value -- entropy, or modified entropy, or
prediction loss (i.e., -np.log(confidence))

    tr_distrs, te_distrs, all_bins = distrs_compute(tr_values, te_values, tr_labels,
te_labels,
                                                    num_bins=num_bins,
log_bins=log_bins)
    risk_score = risk_score_compute(tr_distrs, te_distrs, all_bins, data_values,
data_labels)
    return risk_score
```

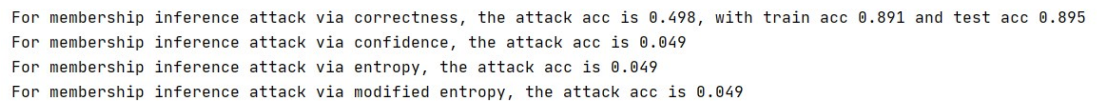
影子模型的训练与结果输出：

```
from membership_inference_attacks import black_box_benchmarks
from privacy_risk_score_utils import calculate_risk_score
class ShadowNet(nn.Module):
    def __init__(self):
        super(ShadowNet, self).__init__()
        self.fc1 = nn.Linear(784, 256)
        self.fc2 = nn.Linear(256, 128)
        self.fc3 = nn.Linear(128, 10)

    def forward(self, x):
        x = x.view(-1, 784)
        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
        x = self.fc3(x)
        return x
# 获取目标模型的训练集和测试集预测结果
target_train_outputs, target_train_labels = get_predictions(model, train_loader)
target_test_outputs, target_test_labels = get_predictions(model, test_loader)

# 训练影子模型
shadow_model = Net()
shadow_optimizer = optim.SGD(shadow_model.parameters(), lr=0.05,
momentum=0.9)

# 划分影子模型的训练集和测试集
shadow_train_size = 30000    # MNIST 训练集共 60000 样本
shadow_indices_train = list(range(0, shadow_train_size))
shadow_indices_test = list(range(shadow_train_size, 60000))

shadow_train_dataset = torch.utils.data.Subset(train_dataset, shadow_indices_train)
```

```
shadow_test_dataset = torch.utils.data.Subset(train_dataset, shadow_indices_test)

shadow_train_loader = DataLoader(shadow_train_dataset, batch_size=1024,
shuffle=True)
shadow_test_loader = DataLoader(shadow_test_dataset, batch_size=1024,
shuffle=False)
# 初始化影子模型和优化器
shadow_model = ShadowNet()
shadow_optimizer = optim.SGD(shadow_model.parameters(), lr=0.05,
momentum=0.9)
shadow_criterion = nn.CrossEntropyLoss()


def train_shadow_model(model, train_loader, optimizer, criterion, epochs=5):
    model.train()
    for epoch in range(epochs):
        total_loss = 0
        correct = 0
        total = 0
        for data, target in train_loader:
            optimizer.zero_grad()
            output = model(data)
            loss = criterion(output, target)
            loss.backward()
            optimizer.step()

            total_loss += loss.item()
            pred = output.argmax(dim=1, keepdim=True)
            correct += pred.eq(target.view_as(pred)).sum().item()
            total += data.size(0)

        print(f'Shadow Epoch {epoch + 1}, Loss: {total_loss / len(train_loader):.4f},
'
               f'Accuracy: {100. * correct / total:.2f}%')


# 执行训练
train_shadow_model(shadow_model, shadow_train_loader, shadow_optimizer,
shadow_criterion, epochs=5)
def get_predictions(model, dataloader):
    model.eval()
    all_outputs = []
    all_labels = []
    with torch.no_grad():
        for data, target in dataloader:
            output = model(data)
            all_outputs.append(output.numpy())
            all_labels.append(target.numpy())
    outputs = np.concatenate(all_outputs, axis=0)
    labels = np.concatenate(all_labels, axis=0)
    return outputs, labels

# 获取影子模型的预测结果
shadow_train_outputs, shadow_train_labels = get_predictions(shadow_model,
shadow_train_loader)
shadow_test_outputs, shadow_test_labels = get_predictions(shadow_model,
shadow_test_loader)
# 计算训练集准确率
shadow_train_preds = np.argmax(shadow_train_outputs, axis=1)
shadow_train_acc = np.mean(shadow_train_preds == shadow_train_labels)
print(f"Shadow Model Train Accuracy: {shadow_train_acc*100:.2f}%")

# 计算测试集准确率
shadow_test_preds = np.argmax(shadow_test_outputs, axis=1)
shadow_test_acc = np.mean(shadow_test_preds == shadow_test_labels)
print(f"Shadow Model Test Accuracy: {shadow_test_acc*100:.2f}%")

# 初始化攻击类
benchmarks = black_box_benchmarks(
    shadow_train_performance=(shadow_train_outputs, shadow_train_labels),
    shadow_test_performance=(shadow_test_outputs, shadow_test_labels),
    target_train_performance=(target_train_outputs, target_train_labels),
    target_test_performance=(target_test_outputs, target_test_labels),
    num_classes=1
)

# 执行攻击
benchmarks._mem_inf_benchmarks()

shadow_train_conf = np.array([shadow_train_outputs[i, shadow_train_labels[i]] for i
in range(len(shadow_train_labels))])
shadow_test_conf = np.array([shadow_test_outputs[i, shadow_test_labels[i]] for i in
range(len(shadow_test_labels))])
tr_values = shadow_train_conf   # 影子模型训练集置信度
te_values = shadow_test_conf      # 影子模型测试集置信度
tr_labels = shadow_train_labels   # 影子模型训练集标签
te_labels = shadow_test_labels     # 影子模型测试集标签
```

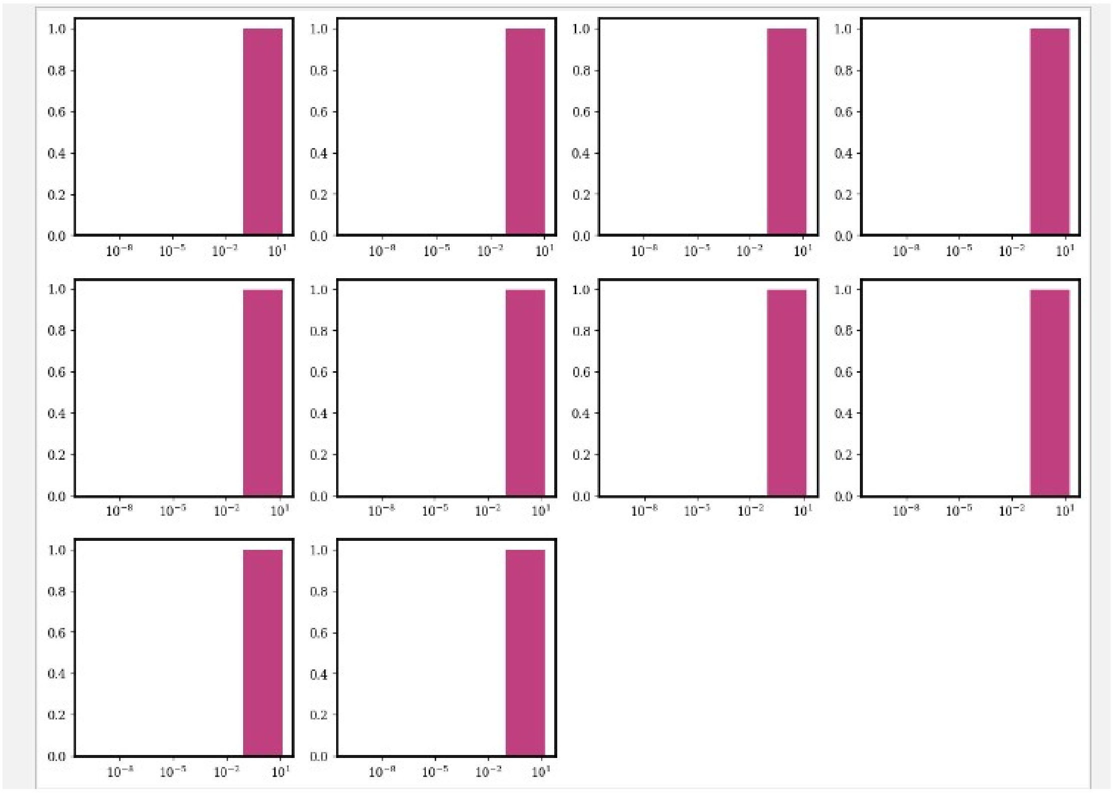
```
# 目标模型训练集的数据
target_train_conf = np.array([target_train_outputs[i, target_train_labels[i]] for i in
range(len(target_train_labels))])
data_values = target_train_conf   # 目标模型训练集置信度
data_labels = target_train_labels   # 目标模型训练集标签

# 计算隐私风险评分
risk_scores = calculate_risk_score(
        tr_values=tr_values,
        te_values=te_values,
        tr_labels=tr_labels,
        te_labels=te_labels,
        data_values=data_values,
        data_labels=data_labels,
        num_bins=5,         # 分箱数量
        log_bins=True       # 是否使用对数分箱
)
# 输出统计信息
print(f"risk_scores: {np.mean(risk_scores):.3f}")
```

③ MIA 攻击结果及分析

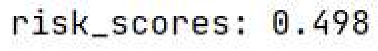


```
For membership inference attack via correctness, the attack acc is 0.498, with train acc 0.891 and test acc 0.895
For membership inference attack via confidence, the attack acc is 0.049
For membership inference attack via entropy, the attack acc is 0.049
For membership inference attack via modified entropy, the attack acc is 0.049
```

如图所示，MIA 以不同特征为攻击点对瑞丽差分隐私保护的模型进行攻击，其准确率皆在 0.5 左右。



如图所示，MNIST 数据集 10 个 lable 均未发生隐私泄露。



```
risk_scores: 0.498
```

隐私风险评分的平均值也处于 0.5 左右。

由此可知，此次攻击未发生隐私泄露，瑞丽差分隐私对模型有较好的保护性。

MIA 从侧面证明了：我们所搭建的瑞丽差分隐私算法，对于 MNIST 数据集而言有着很好的保护力度，这也为我们将这个差分隐私算法移植到联邦学习平台提供了有力的依据。

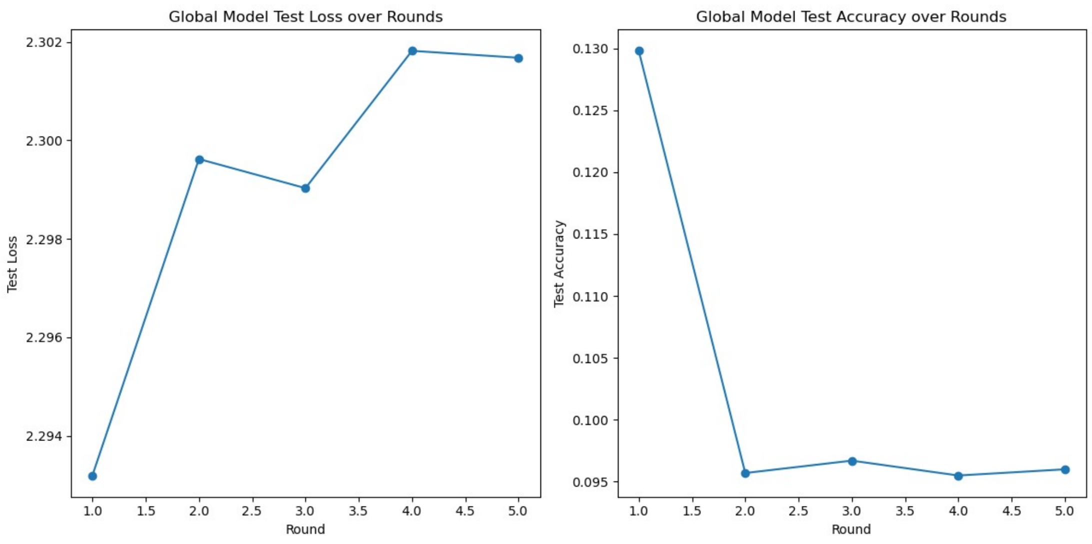
以下是我们对于带有差分隐私算法的联邦学习平台进行学习，与推理攻击的代码，每五轮攻击一次，对于保护力度进行评估。

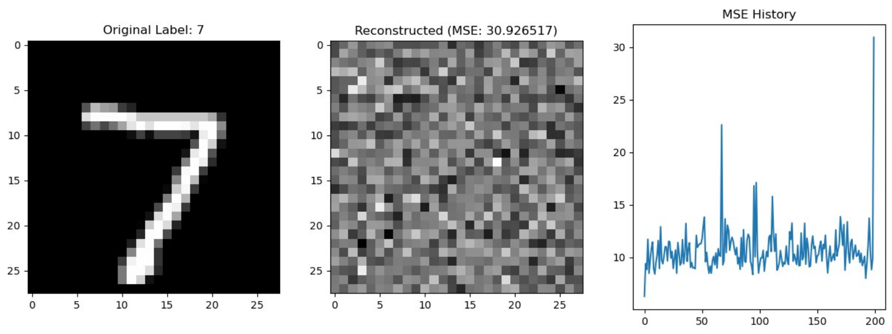
```
import torch
import matplotlib.pyplot as plt
import torch.nn as nn
import torch.optim as optim
from torchvision import datasets, transforms
from torch.utils.data import DataLoader, Subset
import numpy as np
import random
from typing import List, Dict, Tuple
import copy
from opacus import PrivacyEngine

# 设置随机种子（包含设备相关种子）
torch.manual_seed(42)
np.random.seed(42)
random.seed(42)
if torch.cuda.is_available():
    torch.cuda.manual_seed_all(42)


# 定义模型
class Net(nn.Module):
    def __init__(self, device: str = "cpu"):
        super(Net, self).__init__()
        self.device = device
        self.fc1 = nn.Linear(784, 256)
        self.fc2 = nn.Linear(256, 128)
        self.fc3 = nn.Linear(128, 10)
        self.to(device)   # 模型初始化时转移到设备

    def forward(self, x):
        x = x.view(-1, 784).to(self.device)   # 确保输入数据到模型设备
        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
        x = self.fc3(x)
        return x


# 数据集加载函数（带设备转移）
def load_data(client_id, num_clients=10, device="cpu"):
    transform = transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize((0.1307,), (0.3081,))
    ])
    train_full = datasets.MNIST('./data', train=True, download=True,
transform=transform)
    test_dataset = datasets.MNIST('./data', train=False, transform=transform)

    total_samples = len(train_full)
    client_size = total_samples // num_clients
    start_idx = client_id * client_size
    end_idx = start_idx + client_size if client_id < num_clients - 1 else total_samples
    train_subset = Subset(train_full, indices=list(range(start_idx, end_idx)))
    del train_full  # 释放内存

    # 数据加载器不立即转移设备，训练时动态转移
    train_loader = DataLoader(train_subset, batch_size=32, shuffle=True,
pin_memory=torch.cuda.is_available())
    test_loader = DataLoader(test_dataset, batch_size=1000,
pin_memory=torch.cuda.is_available())
    return train_loader, test_loader


# 带 Opacus 隐私保护的本地训练函数（含设备管理）
# 带 Opacus 隐私保护的本地训练函数（处理参数名前缀）
def local_train_with_opacus(
        model: nn.Module,
        train_loader: DataLoader,
        epochs: int,
        lr: float,
        max_grad_norm: float,
        noise_multiplier: float,
        delta: float = 1e-5,
        device: str = "cpu"
) -> Tuple[Dict[str, torch.Tensor], float]:
    model.train()
    criterion = nn.CrossEntropyLoss().to(device)
    optimizer = optim.SGD(model.parameters(), lr=lr)
    privacy_engine = PrivacyEngine(accountant="rdp")
    # 包装后的模型会添加_module 前缀
    model, optimizer, train_loader = privacy_engine.make_private(
        module=model,
        optimizer=optimizer,
        data_loader=train_loader,
        noise_multiplier=noise_multiplier,
        max_grad_norm=max_grad_norm,
        loss_reduction="mean",
        poisson_sampling=True,
    )

    for epoch in range(epochs):
        for data, target in train_loader:
            data = data.to(device, non_blocking=True)
            target = target.to(device, non_blocking=True)

            optimizer.zero_grad()
            output = model(data)
            loss = criterion(output, target)
            loss.backward()
            optimizer.step()

    # 计算隐私预算
    epsilon = privacy_engine.get_epsilon(delta=delta)

    #  原始 state_dict 含_module 前缀，需去除
    wrapped_state = model.state_dict()
    model_state = {name.replace("_module.", ""): param for name, param in
wrapped_state.items()}

    return model_state, epsilon


# 计算模型参数差异（无需修改，键名已统一）
def compute_model_delta(global_model: nn.Module, local_model_state: Dict[str,
torch.Tensor]) -> Dict[str, torch.Tensor]:
    global_state = global_model.state_dict()
    delta = {name: param - global_state[name] for name, param in
local_model_state.items()}
    return delta
```

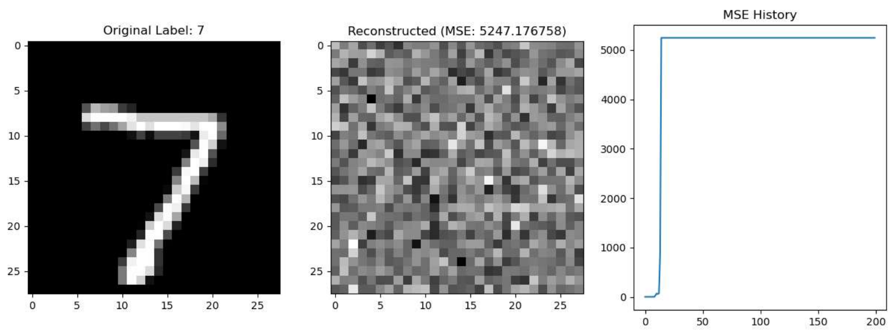
```
# 模型聚合函数
def aggregate_models(global_model: nn.Module, client_deltas: List[Dict[str, torch.Tensor]]) -> None:
    global_state = global_model.state_dict()
    avg_delta = {
        name: torch.stack([delta[name] for delta in client_deltas]).mean(dim=0)
        for name in global_state.keys()
    }
    with torch.no_grad():
        for name, param in global_state.items():
            param.add_(avg_delta[name])
    global_model.load_state_dict(global_state)


# 测试模型（带设备转移）
def test_model(model: nn.Module, test_loader: DataLoader, device: str = "cpu") -> Tuple[float, float]:
    model.eval()
    test_loss = 0.0
    correct = 0
    criterion = nn.CrossEntropyLoss().to(device)  # 损失函数转移到设备
    with torch.no_grad():
        for data, target in test_loader:
            data = data.to(device, non_blocking=True)
            target = target.to(device, non_blocking=True)
            output = model(data)
            test_loss += criterion(output, target).item() * data.size(0)
            correct += output.argmax(dim=1).eq(target).sum().item()
    test_loss /= len(test_loader.dataset)
    accuracy = 100.0 * correct / len(test_loader.dataset)
    return test_loss, accuracy


# DLG 攻击函数（严格设备一致性）
def dlg_attack_on_opacus(
        model: nn.Module,
        true_data: torch.Tensor,
        true_label: torch.Tensor,
        noise_multiplier: float,
        max_grad_norm: float,
        num_iter: int = 200
):
    device = next(model.parameters()).device
    model.eval().to(device)
    true_data = true_data.to(device, non_blocking=True).requires_grad_(True)
    true_label = true_label.to(device, non_blocking=True)

    criterion = nn.CrossEntropyLoss().to(device)
    output = model(true_data)
    loss = criterion(output, true_label)

    # 计算带噪声的梯度（模拟 Opacus 处理）
    model.zero_grad()
    loss.backward()
    nn.utils.clip_grad_norm_(model.parameters(), max_grad_norm)
    clipped_grads = [p.grad.detach().clone() for p in model.parameters()]
    real_grads = [g + torch.randn_like(g) * (noise_multiplier * max_grad_norm) for g
in clipped_grads]

    # 在目标设备上创建虚拟数据
    dummy_data = torch.randn(true_data.size(), requires_grad=True, device=device)
    dummy_label = torch.tensor([true_label.item()], device=device)
    optimizer = torch.optim.LBFGS([dummy_data], lr=1)

    mse_history = []
    for i in range(num_iter):
        def closure():
            optimizer.zero_grad()
            dummy_output = model(dummy_data)
            dummy_loss = criterion(dummy_output, dummy_label)
            dummy_grads = torch.autograd.grad(dummy_loss, model.parameters(),
create_graph=True)
            grad_diff = sum(((a - b) ** 2).sum() for a, b in zip(dummy_grads,
real_grads))
            grad_diff.backward()
            return grad_diff

        optimizer.step(closure)
        current_mse = torch.mean((true_data - dummy_data.detach()) ** 2).item()
        mse_history.append(current_mse)
        if (i + 1) % 50 == 0:
            print(f"Iter {i + 1}/{num_iter}, MSE: {current_mse:.6f}")

    print(f"\nFinal MSE: {current_mse:.6f}")
    plt.figure(figsize=(15, 5))
    plt.subplot(1, 3, 1)
    plt.imshow(true_data.squeeze().detach().cpu().numpy(), cmap='gray')  # 修正此
```

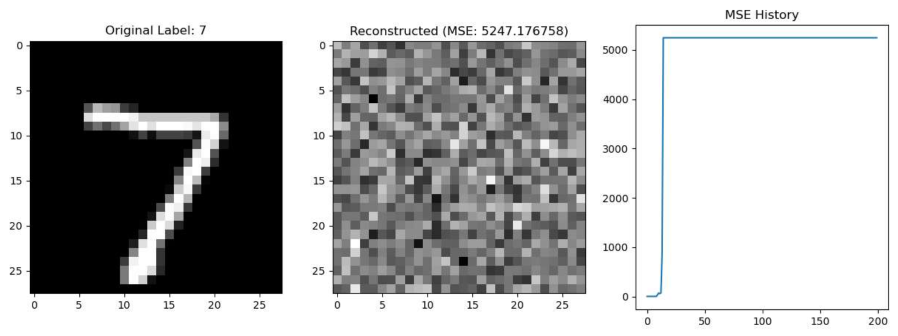
```
处
    plt.title(f'Original Label: {true_label.item()}')

    plt.subplot(1, 3, 2)
    plt.imshow(dummy_data.detach().squeeze().cpu().numpy(),  cmap='gray')    #  修
正此处
    plt.title(f'Reconstructed (MSE: {current_mse:.6f})')

    plt.subplot(1, 3, 3)
    plt.plot(mse_history)
    plt.title('MSE History')
    plt.show()

# 联邦学习主函数（全流程设备管理）
def federated_learning_with_opacus(
        num_clients: int = 10,
        num_rounds: int = 10,
        local_epochs: int = 1,
        client_fraction: float = 0.5,
        learning_rate: float = 0.01,
        max_grad_norm: float = 1.0,
        noise_multiplier: float = 0.5,
        device: str = 'cuda' if torch.cuda.is_available() else 'cpu',
        run_dlg_attack: bool = True,
        dlg_client_id: int = 0
):
    # 初始化全局模型并转移到设备
    global_model = Net(device=device)
    train_loaders = [load_data(i, device=device)[0] for i in range(num_clients)]
    test_loaders = [load_data(i, device=device)[1] for i in range(num_clients)]

    global_test_losses, global_accuracies, privacy_budgets = [], [], []

    for round_idx in range(num_rounds):
        print(f"\nRound {round_idx + 1}/{num_rounds}")
        selected_clients = random.sample(range(num_clients),
                                          max(1,      int(client_fraction       *
num_clients)))

        client_deltas, round_eps = [], []
        for client_id in selected_clients:
            # 本地模型初始化并加载全局参数
            local_model = Net(device=device)
            local_model.load_state_dict(global_model.state_dict())
            # 本地训练（传递设备参数）
            local_state, epsilon = local_train_with_opacus(
                model=local_model,
                train_loader=train_loaders[client_id],
                epochs=local_epochs,
                lr=learning_rate,
                max_grad_norm=max_grad_norm,
                noise_multiplier=noise_multiplier,
                device=device,
            )
            client_deltas.append(compute_model_delta(global_model, local_state))
            round_eps.append(epsilon)

        # 模型聚合
        aggregate_models(global_model, client_deltas)
        avg_epsilon = sum(round_eps) / len(round_eps)
        privacy_budgets.append(avg_epsilon)

        # 测试
        test_loss,    accuracy    =    test_model(global_model,    test_loaders[0],
device=device)
        global_test_losses.append(test_loss)
        global_accuracies.append(accuracy)
        print(f"Test   Loss:   {test_loss:.4f},   Accuracy:   {accuracy:.2f}%,    ε   :
{avg_epsilon:.4f}")

        # 运行 DLG 攻击
        if run_dlg_attack and (round_idx == 0 or (round_idx + 1) % 5 == 0):
            test_data, test_label = next(iter(test_loaders[dlg_client_id]))
            test_sample = test_data[:1]
            test_target = test_label[:1]
            print("\nRunning DLG Attack...")
            dlg_attack_on_opacus(
                model=copy.deepcopy(global_model),
                true_data=test_sample,
                true_label=test_target,
                noise_multiplier=noise_multiplier,
                max_grad_norm=max_grad_norm,
            )

    # 绘制结果
    plt.figure(figsize=(18, 5))
    plt.subplot(1, 3, 1)
    plt.plot(global_test_losses);
    plt.title('Test Loss');
    plt.xlabel('Round')
    plt.subplot(1, 3, 2)
    plt.plot(global_accuracies);
    plt.title('Test Accuracy');
    plt.xlabel('Round')
    plt.subplot(1, 3, 3)
    plt.plot(privacy_budgets);
    plt.title('Privacy Budget (ε)');
    plt.xlabel('Round')
    plt.tight_layout();
    plt.show()

    return {
        'model': global_model,
        'final_loss': global_test_losses[-1],
        'final_accuracy': global_accuracies[-1],
        'privacy_budgets': privacy_budgets
    }


# 主函数
if __name__ == "__main__":
    params = {
        'num_clients': 10,
        'num_rounds': 10,
        'local_epochs': 1,
        'client_fraction': 0.5,
        'learning_rate': 0.01,
        'max_grad_norm': 1.0,
        'noise_multiplier': 0.5,
        'run_dlg_attack': True,
        'dlg_client_id': 0,
        # 自动检测设备
        'device': 'cuda' if torch.cuda.is_available() else 'cpu'
    }

    result = federated_learning_with_opacus(**params)

    # 保存模型（含设备信息）
    torch.save({
        'state_dict': result['model'].state_dict(),
        'device': params['device']
    }, "federated_model_with_opacus.pth")

    print("\n===== 训练参数 =====")
    for param, value in params.items():
        print(f"{param}: {value}")

    print("\n===== 最终结果 =====")
    print(f"测试损失: {result['final_loss']:.4f}")
    print(f"测试准确率: {result['final_accuracy']:.2f}%")
    print(f"最终隐私预算 (ε): {result['privacy_budgets'][-1]:.6f}")
```

生成的图像如下图：









数据输出为：

```
Test Loss: 2.1951, Accuracy: 34.66%,  ε : 6.7468

Running DLG Attack...
Iter 50/200, MSE: 11.261660
Iter 100/200, MSE: 8.486299
Iter 150/200, MSE: 9.461580
Iter 200/200, MSE: 30.926517

Final MSE: 30.926517

Round 2/10
Test Loss: 2.0607, Accuracy: 53.47%,  ε : 6.7468

Round 3/10
Test Loss: 1.8831, Accuracy: 61.28%,  ε : 6.7468

Round 4/10
Test Loss: 1.6823, Accuracy: 66.50%,  ε : 6.7468

Round 5/10
Test Loss: 1.4784, Accuracy: 69.86%,  ε : 6.7468

Running DLG Attack...
Iter 50/200, MSE: 34522.964844
Iter 100/200, MSE: 34522.964844
Iter 150/200, MSE: 34522.964844
Iter 200/200, MSE: 34522.964844

Final MSE: 34522.964844

Round 6/10
Test Loss: 1.2975, Accuracy: 71.56%,  ε : 6.7468

Round 7/10
```

```
Test Loss: 1.1500, Accuracy: 72.63%,  ε : 6.7468

Round 8/10
Test Loss: 1.0290, Accuracy: 73.62%,  ε : 6.7468

Round 9/10
Test Loss: 0.9259, Accuracy: 75.18%,  ε : 6.7468

Round 10/10
Test Loss: 0.8436, Accuracy: 76.83%,  ε : 6.7468

Running DLG Attack...
Iter 50/200, MSE: 5247.176758
Iter 100/200, MSE: 5247.176758
Iter 150/200, MSE: 5247.176758
Iter 200/200, MSE: 5247.176758

Final MSE: 5247.176758

===== 训练参数 =====
num_clients: 10
num_rounds: 10
local_epochs: 1
client_fraction: 0.5
learning_rate: 0.01
max_grad_norm: 1.0
noise_multiplier: 0.5
run_dlg_attack: True
dlg_client_id: 0
device: cuda

===== 最终结果 =====
测试损失: 0.8436
测试准确率: 76.83%
最终隐私预算 (ε): 6.746780
```

从以上输出结果可以发现：

收敛趋势：

测试损失：从第 1 轮的 2.1951 下降到第 10 轮的 0.8436，降幅达 61.5%，表明模型在联邦学习过程中有效学习了数据特征。

准确率：从第 1 轮的 34.66% 提升至第 10 轮的 76.83%，接近非隐私保护下 MNIST 的简单模型性能（通常 80%-90%），说明差分隐私的引入未导致模型完全失效，隐私性能平衡较好。

影响因素：

本地训练轮次：local_epochs=1 较低，可能导致每轮更新的参数差异较小，需更多轮次收敛。

客户端比例：client_fraction=0.5 意味着每轮随机选择 5 个客户端，可能引入一定的随机性，但整体趋势稳定。

隐私预算：

每轮的 ε 固定为 6.7468，最终累积至 6.74678。这是因为：noise_multiplier=0.5 较小，噪声强度低，隐私保护较弱，但是在本实验中可以看到，即使噪声乘数较小依然提供了合理的隐私保护能力。
alphas 参数设置覆盖了广泛的 RDP 阶数，确保预算计算准确。

DLG 攻击结果：

第 1 轮：MSE=30.92，重建图像与原始图像有一定相似性（如轮廓），但细节模糊，说明初始梯度中仍含部分数据信息。

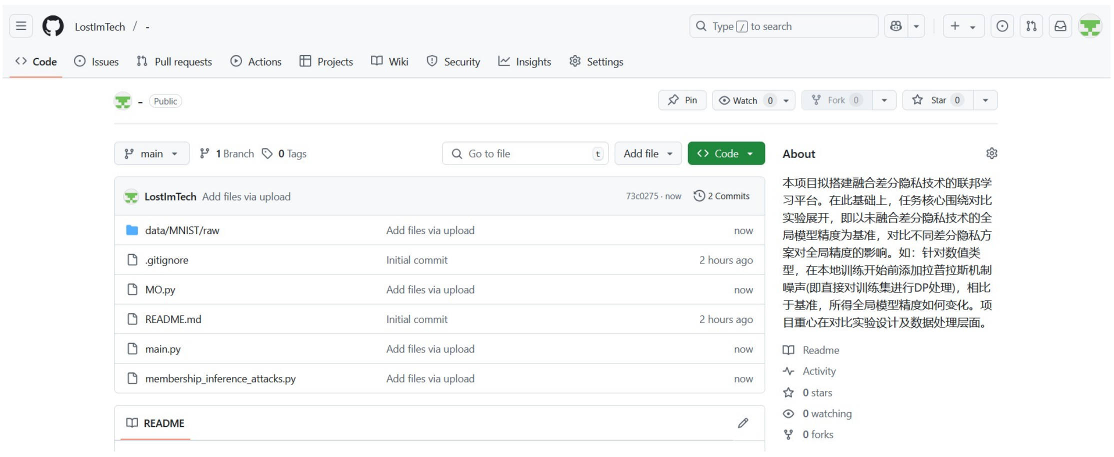
第 5 轮：MSE=34522，重建失败，图像完全随机。

第 10 轮：MSE=5247，重建质量未提升。

结论：差分隐私的噪声添加（noise_multiplier=0.5 + max_grad_norm=1.0）在几轮后显著模糊了梯度中的数据特征，导致攻击失败。隐私保护效果随训练轮次增强，符合差分隐私的累积特性。

由此可见，本项目所要完成的预定目标已基本实现。

5. 将项目成果上传至平台



LostImTech Add files via upload 73c0275 · now 2 Commits

data/MNIST/raw	Add files via upload	now
.gitignore	Initial commit	2 hours ago
MO.py	Add files via upload	now
README.md	Initial commit	2 hours ago
main.py	Add files via upload	now
membership_inference_attacks.py	Add files via upload	now

README

About

本项目拟搭建融合差分隐私技术的联邦学习平台。在此基础上，任务核心围绕对比实验展开，即以未融合差分隐私技术的全局模型精度为基准，对比不同差分隐私方案对全局精度的影响。如：针对数值类型，在本地训练开始前添加拉普拉斯机制噪声(即直接对训练集进行DP处理)，相比于基准，所得全局模型精度如何变化。项目重心在对比实验设计及数据处理层面。

Readme
Activity
0 stars
0 watching
0 forks

三、项目实施过程

初期（2024 年 4~8 月）进行初期软件学习，搭建联邦学习框架
中期（2024 年 9~12 月）实现差分隐私模块并于联邦学习架构相结合

最后（2025 年 1~4 月）设计对比实验，整理分析数据（可视化绘图），撰写实验报告

四、项目经费使用情况

本项目实际未使用任何经费。

五、结论

　　本项目通过搭建联邦学习平台，对于差分隐私算法的使用与机器学习攻击方法的学习有了更加深刻的认识，提高了大家对于该方面知识的自主学习能力和认知水平，对于日后进一步的学习研究打好了基础。

　　本项目所完成的代码与经验，也希望能为后来的学习和创新能提供新的思路。

项目负责人：赵梓琢 2025 年 4 月 14 日