作业

Homework4

庄镇华 502022370071

A Neural Networks Homework Assignment

2023 年 5 月 26 日

注意：对于 4、5 题，请在 pdf 文件中贴关键代码和实验结果，并配以必要的注释。

题目一

Batch normalization 的输出服从什么样的分布（指出分布中的具体参数）。

解答： Batch normalization 的输出服从均值为 β，方差为 γ^2 的正态分布。

题目二

Batch normalization 为什么归一化后还有放缩（γ）和平移（β)?

解答： 因为在神经网络中经常使用 Sigmoid 函数，归一化后的数据一般集中在 0 附近，而变量值在 0 附近的 Sigmoid 函数可以近似看作一个线性函数，减弱了整个 Sigmoid 函数的非线性性质。因此通过仿射变换，把数据映射到表征能力更大的空间，也即把在 0 附近的值映射到 Sigmoid 函数非线性映射能力更强的地方，可以充分利用 Sigmoid 函数的性质。

题目三

目前有一批病人的身体数据（体重变化，血液指标等）和他们是否患有肺癌的真实标签，其中患肺癌的样本只占非常小的比例。数据直接送入一个神经网络中，求问应该使用什么样的初始化？数据中不同的特征数值差异过大，求问如何改进能够让网络更好地学习数据中的分布?

解答： **针对数据中患肺癌的样本占比非常小的问题**，这是一种“类别不平衡”问题。从数据方面出发，可以通过过采样（SMOTE 算法）或欠采样方法，或者根据类别数量来设置损失的权重，或者选择 F1 分数、ROC 或 AUC 等适合的评价指标来进行处理。

至于神经网络初始化方法，可以通过合理设置初始偏置 b 来进行处理，假设初始 $w \cdot x$ 接近于 0，则输出概率主要受偏差 b 影响，对于样本均衡数据集，初始令 $b = 0$，则样本属于正负类的概率均为 1/2，符合预期；而对于类别不平衡样本，若在初始化时将模型预测类别倾向于样本数多的类别，则模型将在训练阶段偏向于对类别样本数较少的样本预测进行修正，专注于学习样本数小的类别的差异化特征。

对于神经网络参数的初始化，常用的方法如 Xavier 初始化和 He 初始化，它们的适用条件如下：Xavier 初始化根据某一层网络输入、输出节点数，将权重初始化在合理的范围内，适用于激活函数为 Sigmoid 和 Tanh 的情况；而 He 初始化基于 Xavier，更适合 ReLU 及其变体作为激活函数的情况。

针对数据中不同特征的数值差异过大的问题，可以进行特征缩放来处理。特征缩放是在数据预处理阶段使用的一种方法，它可以将所有特征的尺度统一到相同的范围，例如标准化和归一化方法。

题目四

参考以下代码使用 scikit-learn 包获取波士顿住房数据集（如果无法正确获取，请检查 scikit-learn 的版本）

```python
from torch import nn
# 导入数据集
from sklearn.datasets import load_boston
boston = load_boston()
# 定义网络模型
class MLP(nn.Module):
    def __init__(self):
        # First hidden layer
        self.h1 = nn.Linear(in_features=13, out_features=20, bias=True)
        self.a1 = nn.ReLU()
        # Second hidden layer
        self.h2 = nn.Linear(in_features=20, out_features=10)
        self.a2 = nn.ReLU()
        # regression predict layer
        self.regression = nn.Linear(in_features=10, out_features=1)

    def forward(self, x):
        x = self.h1(x)
        x = self.a1(x)
        x = self.h2(x)
        x = self.a2(x)
        output = self.regression(x)
        return output
```

结合上述代码完成：

a) 请把获取的数据集分为训练集和验证集，它们的比例为 7:3，设计一个三层的神经网络（不建议调用 MLPRegressor 函数，可以使用 PyTorch 框架自行搭建）分别在进行归一化和不对数据进行归一化的情况下进行训练，给出训练集 loss 的折线图，以及验证集真实值和预测值的差异图，比较二者之间的区别。（推荐 batch size 设置为 8，迭代次数为 5，优化器使用 SGD 优化器，学习率为 0.001，损失函数为平方误差损失）

b) 分别使用平方误差损失，绝对值误差损失和 Huber 损失作为损失函数进行训练，给出训练集 loss 的折线图，以及验证集真实值和预测值的差异图，比较彼此之间的区别。

c) 改变不同的学习率（其余参数固定），给出训练集 loss 的折线图，以及验证集真实值和预测值的差异图。再比较固定学习率和学习率衰减二者之间的区别，给出训练集 loss 的折线图，以及验证集真实值和预测值的差异图。

解答： 由于 batch_size 较小，随机梯度方差较大，容易造成震荡现象，因此本题中我采用全批量数据进行训练，即 batch_size 为训练集大小。

a) 归一化对结果的影响

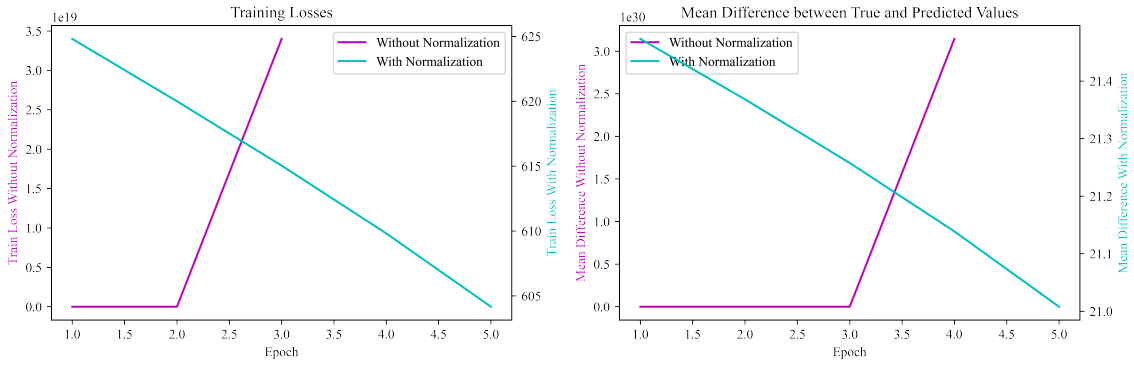

图 1: 归一化对结果的影响

从实验结果可以看到，归一化数据通常能使神经网络的训练过程更加平稳，收敛速度更快。原因是归一化可以把输入特征的数值范围限定在一个相对较小的范围内，使得模型更容易学习到每个特征和预测目标之间的关系，而不会被一些数值特别大或者特别小的特征值影响。此外，归一化后的数据能使优化器更有效地找到误差最小的方向，从而加快训练速度。

```
# 加载数据集
boston = load_boston()
X = boston.data
y = boston.target

# 以7：3分割训练集和数据集
X_train, X_val, y_train, y_val = train_test_split(X, y, test_size=0.3,
    random_state=42)

# 对数据进行标准归一化
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_val_scaled = scaler.transform(X_val)

# 转化为tensor张量
X_train_tensor = torch.tensor(X_train, dtype=torch.float)
y_train_tensor = torch.tensor(y_train, dtype=torch.float)
X_val_tensor = torch.tensor(X_val, dtype=torch.float)
y_val_tensor = torch.tensor(y_val, dtype=torch.float)

X_train_scaled_tensor = torch.tensor(X_train_scaled, dtype=torch.float)
X_val_scaled_tensor = torch.tensor(X_val_scaled, dtype=torch.float)

loss_fn = nn.MSELoss()

# 训练过程
def train(model, X_train, y_train, X_val, y_val, num_epochs=5):
    # SGD优化器
    optimizer = SGD(model.parameters(), lr=0.001)
    train_losses = []
    val_losses = []
    val_pred_list = []

```

```
    for epoch in range(num_epochs):
        model.train() # 模型训练
        y_pred = model(X_train)
        loss = loss_fn(y_pred.squeeze(), y_train)
        train_losses.append(loss.item())
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        model.eval() # 模型评估
        with torch.no_grad():
            y_val_pred = model(X_val)
            val_loss = loss_fn(y_val_pred.squeeze(), y_val)
            val_losses.append(val_loss.item())
            val_pred_list.append(y_val_pred)
        print(f'Epoch {epoch+1}, Train Loss: {loss.item()}, Validation Loss: {
    val_loss.item()}')
    return train_losses, val_losses, val_pred_list

# 不对数据进行归一化
model = MLP()
train_losses, val_losses, val_pred_list = train(model, X_train_tensor,
    y_train_tensor, X_val_tensor, y_val_tensor)

# 对数据进行归一化
model_scaled = MLP()
train_losses_scaled, val_losses_scaled, val_pred_list_scaled = train(model_scaled,
     X_train_scaled_tensor, y_train_tensor, X_val_scaled_tensor, y_val_tensor)
```

b) 损失函数对结果的影响

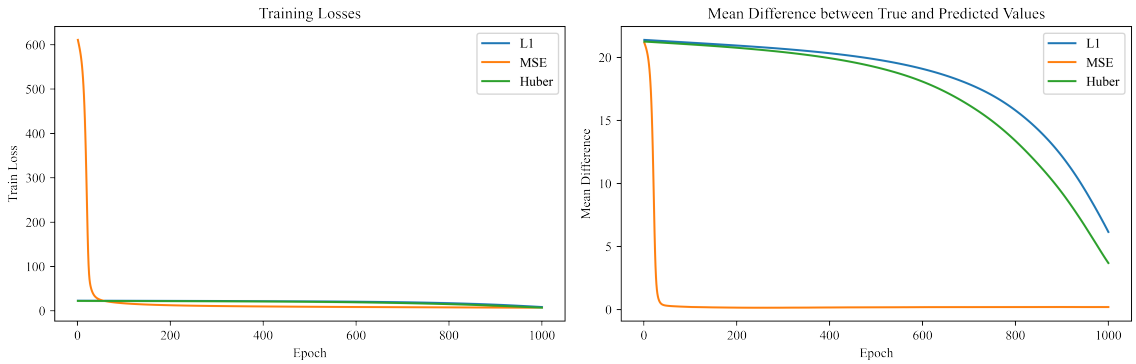

图 2: 损失函数对结果的影响

关于三种损失之间的区别和联系，通过实验图像可以发现，平方误差损失对于异常值非常敏感，因为它将差异平方，如果数据中有很多异常值，那么损失可能会很高；绝对值误差损失对异常值不太敏感，因为它只是考虑差值的绝对值，然而这可能使得训练更难收敛，因为绝对值函数在零点处不可微；而 Huber 损失是 MSE 和 L1 损失的结合。对于较小的误差，它的行为类似于 MSE，而对于较大的误差，它的行为则类似于 L1，因此它结合两者的优点，既不太敏感于异常值，又能保持良好的微分性质，实验也验证了这些观点。

```python
num_epochs = 1000

# 修改训练函数以接受损失函数作为参数
def train(model, X_train, y_train, X_val, y_val, loss_fn, num_epochs=1000):
    optimizer = SGD(model.parameters(), lr=0.001)
    train_losses = []
    val_losses = []
    val_pred_list = []

    for epoch in range(num_epochs):
        model.train()
        y_pred = model(X_train)
        loss = loss_fn(y_pred.squeeze(), y_train)
        train_losses.append(loss.item())
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        model.eval()
        with torch.no_grad():
            y_val_pred = model(X_val)
            val_loss = loss_fn(y_val_pred.squeeze(), y_val)
            val_losses.append(val_loss.item())
            val_pred_list.append(y_val_pred)

        print(f'Epoch {epoch+1}, Train Loss: {loss.item()}, Validation Loss: {val_loss.item()}')

    return train_losses, val_losses, val_pred_list

# 定义不同的损失函数
loss_functions = [MSELoss(), L1Loss(), SmoothL1Loss()]  # 平方误差损失，绝对值误差损失和 Huber 损失
loss_names = ['MSE', 'L1', 'Huber']

# 对于每个损失函数，训练模型
for loss_fn, loss_name in zip(loss_functions, loss_names):
    model = MLP()
    train_losses, val_losses, val_pred_list = train(model, X_train_scaled_tensor, y_train_tensor, X_val_scaled_tensor, y_val_tensor, loss_fn)
```

c) 学习率对结果的影响

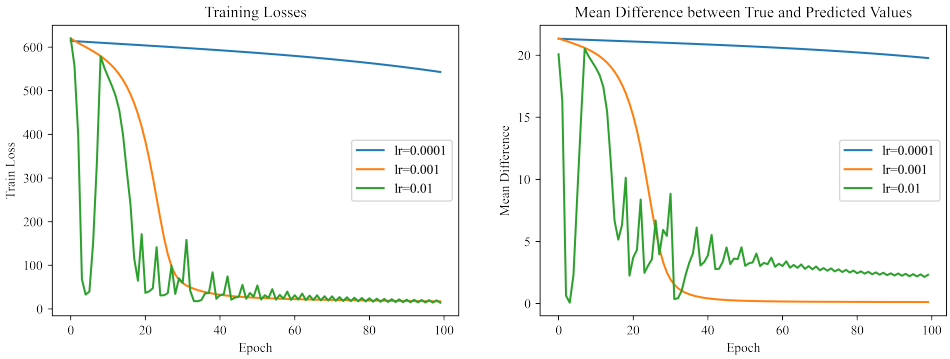

图 3: 学习率对结果的影响

通过比较不同学习率的训练损失和预测与真实值的平均差值，可以观察到较高的学习率可能会导致训练过程中损失的剧烈波动，使模型在最优解附近震荡而无法收敛。较低的学习率则可能使训练过程变得缓慢，但可能会使模型更容易收敛到最优解。

```python
learning_rates = [1e-4, 1e-3, 0.01]  # 三种不同的学习率

loss_fn = MSELoss()  # 使用MSE作为损失函数

def train(model, X_train, y_train, X_val, y_val, loss_fn, optimizer, num_epochs=1000):
    train_losses = []
    val_losses = []
    val_pred_list = []

    for epoch in range(num_epochs):
        model.train()
        y_pred = model(X_train)
        loss = loss_fn(y_pred.squeeze(), y_train)
        train_losses.append(loss.item())
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        model.eval()
        with torch.no_grad():
            y_val_pred = model(X_val)
            val_loss = loss_fn(y_val_pred.squeeze(), y_val)
            val_losses.append(val_loss.item())
            val_pred_list.append(y_val_pred)

        print(f'Epoch {epoch+1}, Train Loss: {loss.item()}, Validation Loss: {val_loss.item()}')

    return train_losses, val_losses, val_pred_list

# 对于每个学习率，训练模型
for lr in learning_rates:
    model = MLP()
    optimizer = SGD(model.parameters(), lr=lr)
```

```
    train_losses, val_losses, val_pred_list = train(model, X_train_scaled_tensor,
    y_train_tensor, X_val_scaled_tensor, y_val_tensor, loss_fn, optimizer,
    num_epochs=100)
```

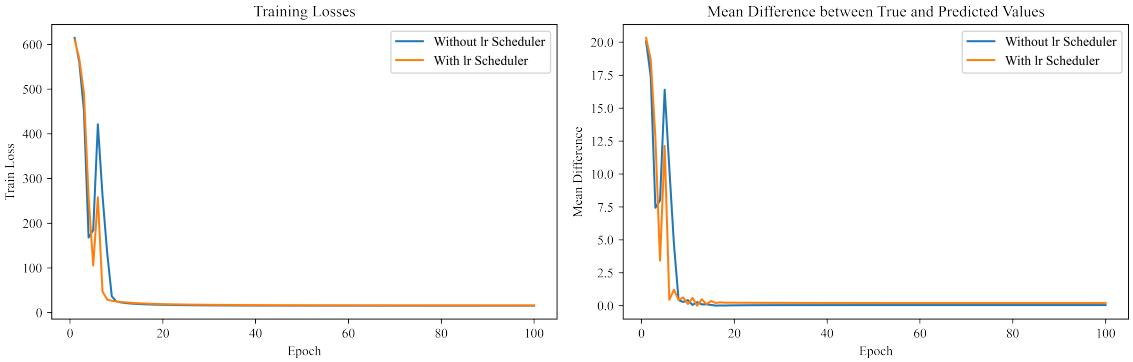

图 4: 衰减学习率对结果的影响

通过实验发现，学习率衰减通常可以帮助模型在初期快速收敛，然后在接近最优解时减小步长，避免震荡情况，以更精确地定位最优解。

```
from torch.optim import SGD, lr_scheduler
lr = 1e-2 # 学习率
num_epochs = 100 # 总运行次数
lr_decay_factor = 0.8  # 学习率衰减因子
decay_steps = 3  # 每隔多少个epoch衰减一次学习率

def train_with_scheduler(model, X_train, y_train, X_val, y_val, loss_fn,
    num_epochs=1000, use_scheduler=False):
    optimizer = SGD(model.parameters(), lr=lr)
    scheduler = lr_scheduler.StepLR(optimizer, step_size=decay_steps, gamma=
    lr_decay_factor)  # 定义学习率衰减策略
    train_losses = []
    val_losses = []
    val_pred_list = []

    for epoch in range(num_epochs):
        model.train()
        y_pred = model(X_train)
        loss = loss_fn(y_pred.squeeze(), y_train)
        train_losses.append(loss.item())
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        scheduler.step() # 更新学习率

        model.eval()
        with torch.no_grad():
            y_val_pred = model(X_val)
            val_loss = loss_fn(y_val_pred.squeeze(), y_val)
            val_losses.append(val_loss.item())
            val_pred_list.append(y_val_pred)
```

```
        print(f'Epoch {epoch+1}, Train Loss: {loss.item()}, Validation Loss: {val_loss.item()}')
    return train_losses, val_losses, val_pred_list

model = MLP()
# 使用学习率衰减策略
train_losses_schedule, val_losses_schedule, val_pred_list_schedule = train_with_scheduler(model, X_train_scaled_tensor, y_train_tensor,
                                                                                           X_val_scaled_tensor, y_val_tensor, loss_fn, num_epochs, True)
model = MLP()
# 不使用学习率衰减策略
train_losses, val_losses, val_pred_list = train_with_scheduler(model, X_train_scaled_tensor, y_train_tensor,
                                                               X_val_scaled_tensor, y_val_tensor, loss_fn, num_epochs, False)
```

题目五

参考下列代码下载 mnist 数据集并使用一个两层的神经网络进行分类（实例代码如下），请分别画出训练集准确率，训练集 loss 及验证集准确率的折线图。（推荐 batchsize 设置为 100，迭代次数为 3，优化器使用 SGD 优化器，学习率为 0.001，损失函数为交叉熵损失）

```
# 载入数据集
train_data = torchvision.datasets.MNIST(
    root= 'MNIST',
    train= True,
    transform = torchvision.transforms.ToTensor(),
    download = True
)
test_data = torchvision.datasets.MNIST(
    root = 'MNIST',
    train = False,
    transform = torchvision.transforms.ToTensor(),
    download  = True
)
# 网络模型
class MLP(nn.Module):
    def __init__ (self,):
        super(MLP, self).__init__()
        # 第一个线性层
        self.layer1 = nn.Linear(784, 2048)
        self.layer2 = nn.Linear(2048, 10)
        self.relu = nn.ReLU()
        # 前向传播
    def forward(self, input) :
        out = self.layer1(input)
        out = self.relu(out)
        out = self.layer2(out)
        return out
```

结合上述代码完成：

a) 针对上述神经网络，请至少使用三种不同的初始化方式，并分别画出训练集准确率，训练集 loss 及验证集准确率的折线图。

b) 针对上述神经网络，请分别进行批归一化，层归一化，实例归一化及组归一化，并画出训练集准确率，训练集 loss 及验证集准确率的折线图。

解答： **a)** 通过实验可以发现，初始化方法的选择可能会对训练效果产生显著影响。例如，Xavier 初始化属于基于方差缩减的方法，其被设计为在前向传播和反向传播时保持激活函数的方差一致，这有助于保持信号在各层间的稳定传递。相比之下，零初始化可能会导致网络训练过程陷入困境，因为所有神经元的梯度将相同，无法进行有效的学习。而高斯初始化属于基于固定方差的方法，其缺点为随着网络层数的增加，越靠后的层，其激活函数的输出值越集中于 0，这样极易出现梯度消失现象。

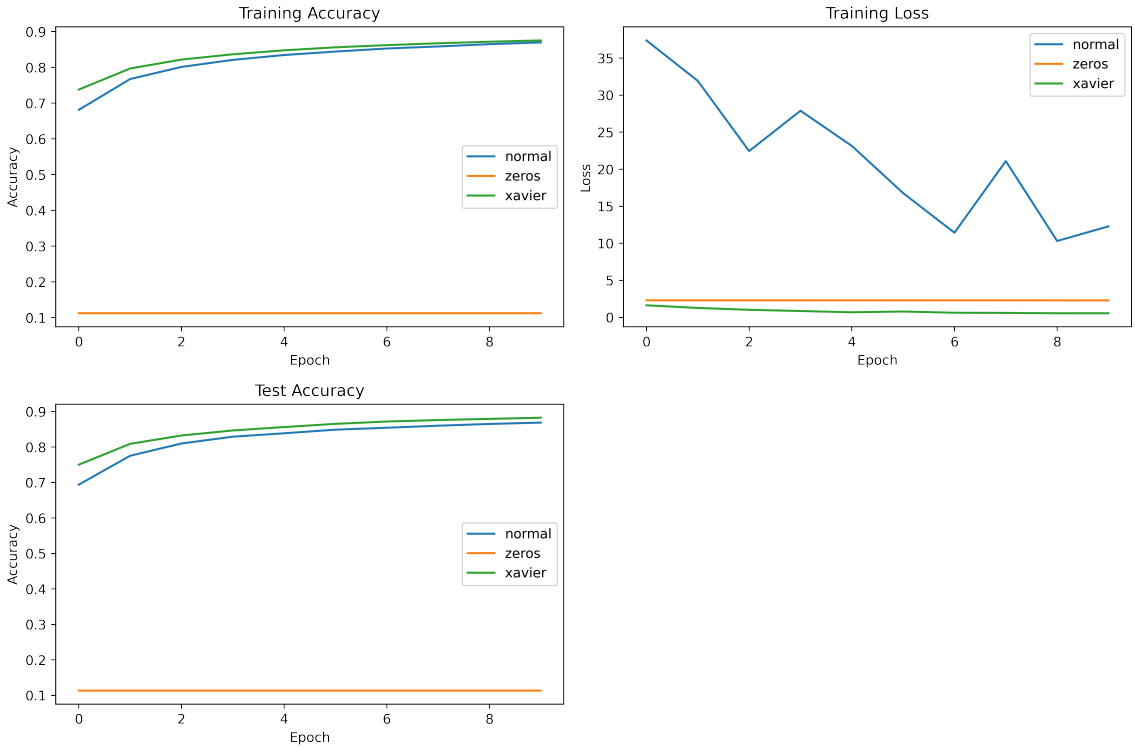

图 5: 初始化方法对结果的影响

```python
# 设备加载
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

# 数据加载器
train_loader = torch.utils.data.DataLoader(dataset=train_data, batch_size=100, shuffle=True)
test_loader = torch.utils.data.DataLoader(dataset=test_data, batch_size=100, shuffle=False)

class MLP(nn.Module):
    ......

    def reset_parameters(self):
        if self.init_method == 'normal':
            nn.init.normal_(self.layer1.weight)
            nn.init.normal_(self.layer2.weight)
        elif self.init_method == 'zeros':
            nn.init.zeros_(self.layer1.weight)
            nn.init.zeros_(self.layer2.weight)
        elif self.init_method == 'xavier':
            nn.init.xavier_uniform_(self.layer1.weight)
            nn.init.xavier_uniform_(self.layer2.weight)
        else:
            raise ValueError('Invalid init method!')

# 定义训练函数
```

```python
def train_model(model, train_loader, test_loader, device, num_epochs, criterion, optimizer):
    n_total_steps = len(train_loader)
    train_acc_list, train_loss_list, test_acc_list = [], [], []

    for epoch in range(num_epochs):
        for i, (images, labels) in enumerate(train_loader):
            images = images.reshape(-1, 28*28).to(device)
            labels = labels.to(device)

            # 前向传播
            outputs = model(images)
            loss = criterion(outputs, labels)

            # 反向传播和优化
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

        with torch.no_grad():
            correct_train = 0
            total_train = 0
            for images, labels in train_loader:
                images = images.reshape(-1, 28*28).to(device)
                labels = labels.to(device)
                outputs = model(images)
                _, predicted = torch.max(outputs.data, 1)
                total_train += labels.size(0)
                correct_train += (predicted == labels).sum().item()
            train_acc_list.append(correct_train/total_train)
            train_loss_list.append(loss.item())

            correct_test = 0
            total_test = 0
            for images, labels in test_loader:
                images = images.reshape(-1, 28*28).to(device)
                labels = labels.to(device)
                outputs = model(images)
                _, predicted = torch.max(outputs.data, 1)
                total_test += labels.size(0)
                correct_test += (predicted == labels).sum().item()
            test_acc_list.append(correct_test/total_test)

    print('Finished Training')

    return train_acc_list, train_loss_list, test_acc_list

# 以不同初始化方法训练并保存结果
init_methods = ['normal', 'zeros', 'xavier']
results = {}

for init_method in init_methods:
    model = MLP(init_method).to(device)
    model.reset_parameters()
```

```
    criterion = nn.CrossEntropyLoss()
    optimizer = torch.optim.SGD(model.parameters(), lr=0.001)
    num_epochs = 10

    train_acc_list, train_loss_list, test_acc_list = train_model(model,
    train_loader, test_loader, device, num_epochs, criterion, optimizer)

    results[init_method] = {
        'train_acc': train_acc_list,
        'train_loss': train_loss_list,
        'test_acc': test_acc_list
    }
```

b) 实验选用的批大小为 100，相比于其他任务而言较小，通过实验发现组归一化的效果确实要好于其他归一化方法，然后是层归一化方法性能和组归一化基本相当，实例归一化方法次之，且由于批大小较小，批归一化方法性能最差，这和理论基本相符。

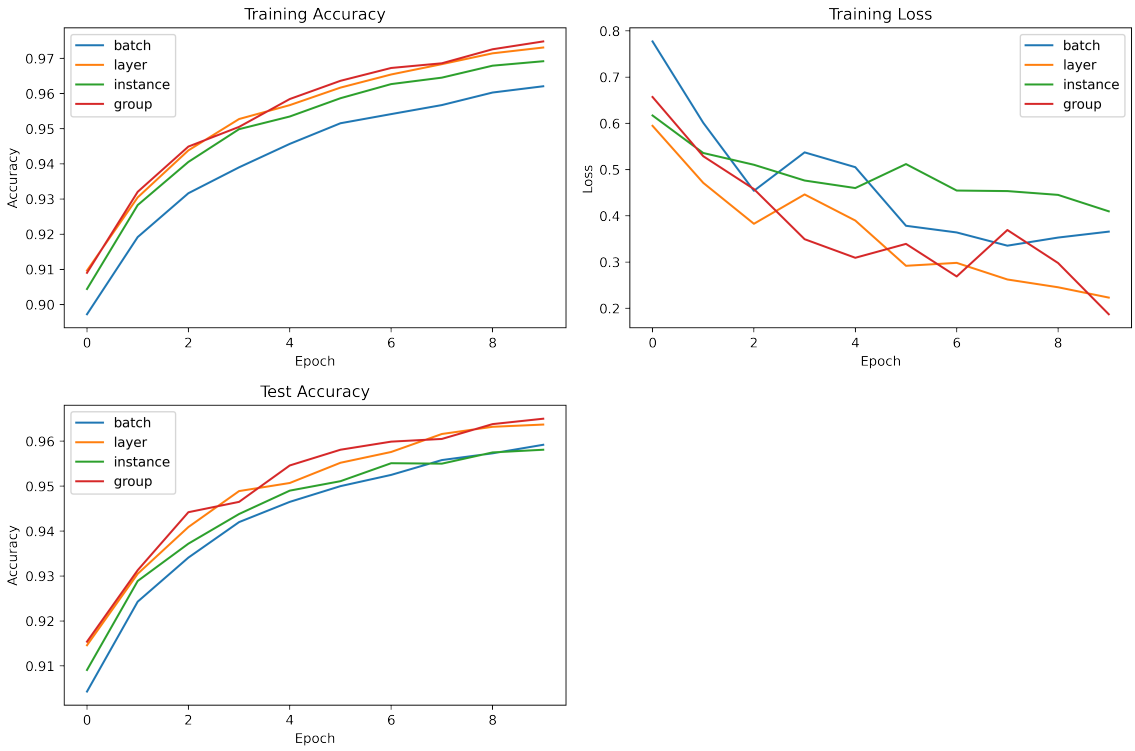

图 6: 归一化方法对结果的影响

下面针对各种归一化方法做一下总结:

批量归一化 (Batch Normalization):

批归一化对每一批数据进行归一化。

公式: $\hat{x}_i = \frac{xi-\mu_B}{\sqrt{\sigma_B^2+\epsilon}}$

优点: 减少内部协变量偏移，使网络收敛更快，使得更深的网络可以被训练，具有一定的正则化效果，可能代替 dropout 和 L2 正则化方法。

缺点: 对批次大小有一定依赖，可能导致批次大小较小时性能下降，对于 RNN 类型的

网络结构不太适用。

层归一化 (Layer Normalization):

层归一化的基本思想是对中间层的所有神经元进行归一化。

公式：$\hat{x}_i = \frac{xi-\mu_L}{\sqrt{\sigma_L^2+\epsilon}}$

优点：不依赖批次大小，适用于 RNN 类型的网络结构。

缺点：在处理卷积类型的网络时，可能效果不如批量归一化。

实例归一化 (Instance Normalization):

公式：$\hat{x}_i = \frac{xi-\mu_I}{\sqrt{\sigma_I^2+\epsilon}}$

优点：在风格迁移任务中表现出色，不依赖批次大小。

缺点：在其他任务中可能效果一般。

组归一化 (Group Normalization):

公式：$\hat{x}_i = \frac{xi-\mu_G}{\sqrt{\sigma_G^2+\epsilon}}$

优点：在批次大小变动和网络深度变化的情况下，性能稳定。

缺点：增加了超参数（组的数量），需要调整以获得最佳性能。

其中，x_i 是输入数据，μ 和 σ^2 分别代表均值和方差，ϵ 是一个防止分母为 0 的小常数。

```python
# 定义模型
class MLP(nn.Module):
    def __init__(self, normalization):
        super(MLP, self).__init__()
        self.layer1 = nn.Linear(784, 2048)
        self.layer2 = nn.Linear(2048, 10)
        self.relu = nn.ReLU()

        if normalization == 'batch':
            self.norm1 = nn.BatchNorm1d(2048)
            self.norm2 = nn.BatchNorm1d(10)
        elif normalization == 'layer':
            self.norm1 = nn.LayerNorm(2048)
            self.norm2 = nn.LayerNorm(10)
        elif normalization == 'instance':
            self.norm1 = nn.InstanceNorm1d(2048)
            self.norm2 = nn.InstanceNorm1d(10)
        elif normalization == 'group':
            self.norm1 = nn.GroupNorm(32, 2048)  # 假设使用32个组
            self.norm2 = nn.GroupNorm(1, 10)
        else:
            self.norm1 = None
            self.norm2 = None

    def forward(self, input):
        out = self.layer1(input)
        if self.norm1:
            out = self.norm1(out)
        out = self.relu(out)

        out = self.layer2(out)
        if self.norm2:
            out = self.norm2(out)
        return out
```

```
35
36 # 使用不同归一化方法初始化模型
37 normalizations = ['batch', 'layer', 'instance', 'group']
38 results = {}
39
40 num_epochs = 10
41 for norm in normalizations:
42     model = MLP(norm).to(device)
43     optimizer = torch.optim.SGD(model.parameters(), lr=0.001)
44     criterion = nn.CrossEntropyLoss()
45     train_acc, train_loss, test_acc = train_model(model, train_loader, test_loader
       , device, num_epochs, criterion, optimizer)
46     results[norm] = {'train_acc': train_acc, 'train_loss': train_loss, 'test_acc':
        test_acc}
```