南開大學
Nankai University

南　　开　　大　　学

计　算　机　学　院

深度学习及应用实验报告

实验一　MLP 的实现与改进

姓名：苏航

学号：2111039

专业：信息安全

2024 年 5 月 29 日

摘要

本实验在理解 MLP 原理的基础上, 学会使用 pytorch 框架运行 MLP 算法实现 MNIST 数据集的分类, 并且学会了改进网络参数或优化器参数等提高准确率, 并且改进了 MLP 网络结构, 实现了 ResMLP

关键字：FFN；MLP；参数优化;ResMLP

目录

一、 实验要求

本次实验要求掌握前馈神经网络（FFN）的基本原理并学会使用 PyTorch 搭建简单的 FFN 实现 MNIST 数据集分类, 同时掌握如何改进网络结构、调试参数以提升网络识别性能.

二、 实验内容

本实验要求运行原始版本 MLP，查看网络结构、损失和准确度曲线, 并尝试调节 MLP 的全连接层参数（深度、宽度等）、优化器参数等，以提高准确度, 并挑选 MLP-Mixer，ResMLP，Vision Permutator 中的一种进行实现

三、 实验过程

(一) 前馈神经网络的基本原理

前馈神经网络由一个输入层、一个或多个隐藏层以及一个输出层组成。各层之间通过权重连接，每个节点（或神经元）接收来自上一层的信息，并将输出传递到下一层。在训练过程中，前馈神经网络使用反向传播算法来更新权重，以最小化预测输出与实际输出之间的误差。具体的模型结构如下所示:

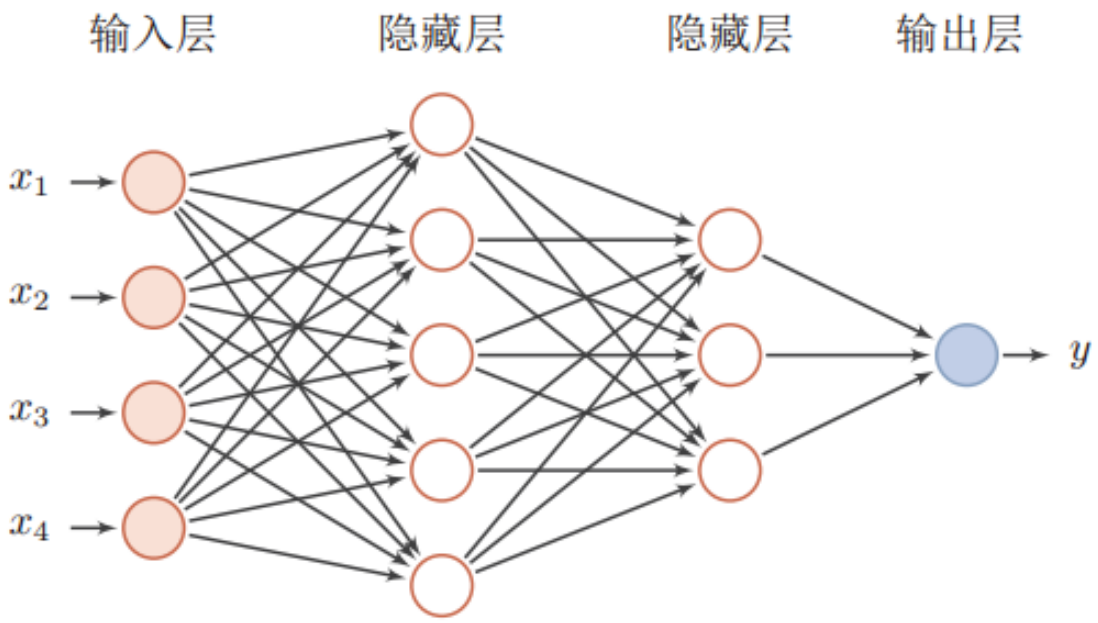

图 1: 前馈神经网络模型图

(二) 原始 MLP 的实现

实验要求我们实现最简单的 MLP 网络结构, 我们就可以只定义一个隐藏层, 原始的数据集有 784 个特征, 要求分类有 10 种, 我们就可以知道输入层的参数是 784, 输出层的参数是 10, 中间的隐藏层参数未知, 我们可以随便设定, 具体的实现代码如下所示:

```
1 class MLP(nn.Module):
2 def __init__(self):
```

```
        super(MLP, self).__init__()
        self.fc1 = nn.Linear(784, 256) #前一个是输入 后一个是输出
        self.fc2 = nn.Linear(256, 64)
        self.fc3 = nn.Linear(64, 10)
        self.relu = nn.ReLU() # 这里选取的是ReLU函数作为激活函数

    def forward(self, x):
        x = x.view(x.size(0), -1) # 将输入展平
        x = self.relu(self.fc1(x))
        x = self.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

搭建成功模型后我们可以把它输出出来:

```
MLP(
  (fc1): Linear(in_features=784, out_features=256, bias=True)
  (fc2): Linear(in_features=256, out_features=64, bias=True)
  (fc3): Linear(in_features=64, out_features=10, bias=True)
  (relu): ReLU()
)
```

图 2: 原始 MLP 网络结构图

由此可见该网络的结构还是非常简单的

(三) 原始 MLP 的训练与评估

搭建完基本的 MLP 后我们就可以开始对其展开训练与评估, 实验要求我们采用 MNIST 数据集, 在进行训练之前我们需要对其进行预处理, 代码如下:

```
    from torch.utils.data import DataLoader
transform = transforms.Compose([
    transforms.ToTensor(), #将图像转化为tensor, 并将像素值缩到[0,1]之间
    transforms.Normalize((0.1307,), (0.3081,)) #对数据进行标准化 这两个数字
        式统计计算后得到的 MNIST 数据集的均值和标准差 有助于加快收敛速度
])
# 加载MNIST训练集和测试集
train_dataset = torchvision.datasets.MNIST(root='./data', train=True,
    transform=transform, download=True)
test_dataset = torchvision.datasets.MNIST(root='./data', train=False,
    transform=transform, download=True)
# 创建数据加载器
train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=64, shuffle=False)
```

训练函数也很简单, 具体如下:

```
    def train(model, train_loader, optimizer, criterion, device):
    # 将模型设置为训练模式
    model.train()
    train_loss = 0
    correct = 0
```

```
    total = 0
    for inputs, labels in train_loader:
        inputs, labels = inputs.to(device), labels.to(device)
        optimizer.zero_grad()
        outputs = model(inputs)
        # 计算损失
        loss = criterion(outputs, labels)
        # 反向传播更新参数
        loss.backward()
        optimizer.step()
        # 统计总损失、预测正确的样本数和总样本数
        train_loss += loss.item()
        _, predicted = outputs.max(1)
        total += labels.size(0)
        correct += predicted.eq(labels).sum().item()
    acc = correct / total
    return train_loss, acc
```

实现该网络的训练和评估后我们得到下面的结果:

```
Epoch 1/10, Train Loss: 223.7061, Train Acc: 0.9293, Test Loss: 16.7788, Test Acc: 0.9656
Epoch 2/10, Train Loss: 90.3047, Train Acc: 0.9701, Test Loss: 13.8140, Test Acc: 0.9725
Epoch 3/10, Train Loss: 62.6340, Train Acc: 0.9791, Test Loss: 13.4113, Test Acc: 0.9741
Epoch 4/10, Train Loss: 48.5457, Train Acc: 0.9833, Test Loss: 14.9220, Test Acc: 0.9701
Epoch 5/10, Train Loss: 36.9639, Train Acc: 0.9867, Test Loss: 12.9287, Test Acc: 0.9756
Epoch 6/10, Train Loss: 31.6200, Train Acc: 0.9891, Test Loss: 11.6431, Test Acc: 0.9794
Epoch 7/10, Train Loss: 25.7453, Train Acc: 0.9906, Test Loss: 14.8336, Test Acc: 0.9765
Epoch 8/10, Train Loss: 24.2344, Train Acc: 0.9915, Test Loss: 15.6067, Test Acc: 0.9740
Epoch 9/10, Train Loss: 21.4461, Train Acc: 0.9922, Test Loss: 14.4017, Test Acc: 0.9764
Epoch 10/10, Train Loss: 17.4317, Train Acc: 0.9937, Test Loss: 14.9420, Test Acc: 0.9777
```

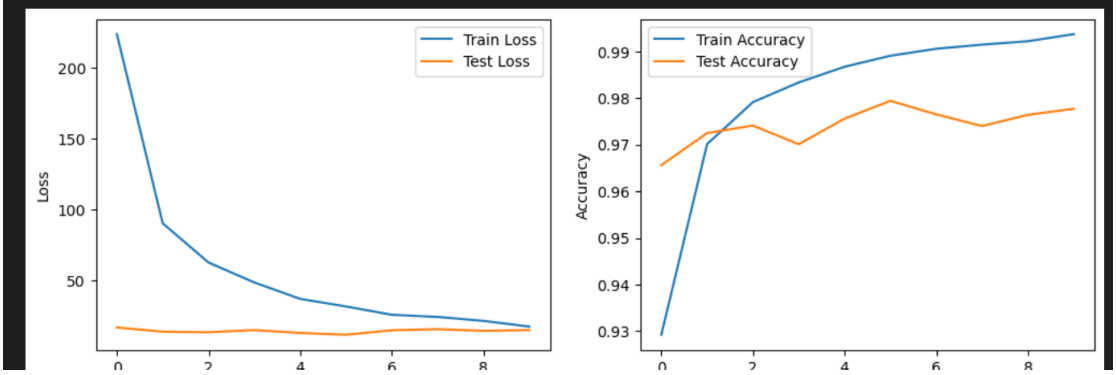

图 3: 模型评估结果

我们可以看到原始网络结构的准确率已经相当高了, 最后的准确率在 0.97 左右, 下面我们将尝试优化网络结构或者是参数选择, 看看准确率能否更高.

(四) MLP 参数优化

1. 更深层的 MLP

我们调整 MLP 的层数, 并在每一层后面都加上批归一化层, 并在激活函数后面加上 dropout 层, 我们定义下面的网络结构:

```
    class DeepMLP(nn.Module):
```

```
    def __init__(self):
        super(DeepMLP, self).__init__()
        self.fc1 = nn.Linear(784, 512)
        self.bn1 = nn.BatchNorm1d(512)
        self.fc2 = nn.Linear(512, 256)
        self.bn2 = nn.BatchNorm1d(256)
        self.fc3 = nn.Linear(256, 128)
        self.bn3 = nn.BatchNorm1d(128)
        self.fc4 = nn.Linear(128, 64)
        self.bn4 = nn.BatchNorm1d(64)
        self.fc5 = nn.Linear(64, 10)
        self.relu = nn.ReLU()
        self.dropout = nn.Dropout(0.5)

    def forward(self, x):
        x = x.view(x.size(0), -1)
        x = self.relu(self.bn1(self.fc1(x)))
        x = self.dropout(x)
        x = self.relu(self.bn2(self.fc2(x)))
        x = self.dropout(x)
        x = self.relu(self.bn3(self.fc3(x)))
        x = self.dropout(x)
        x = self.relu(self.bn4(self.fc4(x)))
        x = self.dropout(x)
        x = self.fc5(x)
        return x
```

之后我们就可以对其进行训练和评估, 观察到有如下结果:

```
Epoch 1/10, Train Loss: 591.8549, Train Acc: 0.8268, Test Loss: 24.1409, Test Acc: 0.9548
Epoch 2/10, Train Loss: 302.8744, Train Acc: 0.9168, Test Loss: 18.0425, Test Acc: 0.9671
Epoch 3/10, Train Loss: 246.1358, Train Acc: 0.9330, Test Loss: 15.9895, Test Acc: 0.9712
Epoch 4/10, Train Loss: 214.6568, Train Acc: 0.9418, Test Loss: 14.5827, Test Acc: 0.9726
Epoch 5/10, Train Loss: 199.0383, Train Acc: 0.9464, Test Loss: 13.1452, Test Acc: 0.9759
Epoch 6/10, Train Loss: 180.1114, Train Acc: 0.9504, Test Loss: 12.1757, Test Acc: 0.9780
Epoch 7/10, Train Loss: 168.1900, Train Acc: 0.9550, Test Loss: 11.9869, Test Acc: 0.9778
Epoch 8/10, Train Loss: 159.0253, Train Acc: 0.9569, Test Loss: 11.2830, Test Acc: 0.9799
Epoch 9/10, Train Loss: 152.2637, Train Acc: 0.9591, Test Loss: 11.0407, Test Acc: 0.9805
Epoch 10/10, Train Loss: 145.0187, Train Acc: 0.9601, Test Loss: 11.1012, Test Acc: 0.9792
```

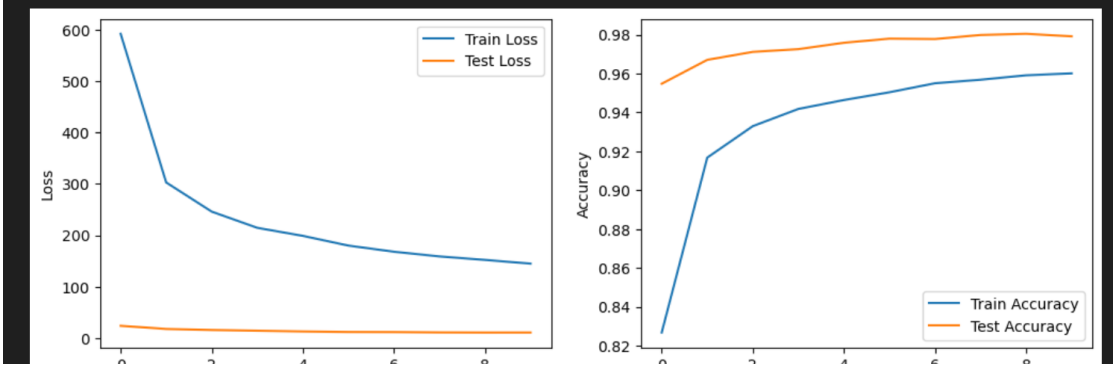

图 4: 深层 MLP 的评估结果

我们观察到我们设计的更深层的 MLP 具有比原始 MLP 更高的准确度, 达到了 0.98.

这是符合预期的, 因为更深层的网络具有更强的表达能力和特征提取能力, 但是相应的计算开销和过拟合的风险也会增大, 上图就展示了过拟合的情况

2. 调节全连接层的宽度

接下来我们尝试调节 MLP 全连接层的宽度, 将原始 MLP 的隐藏层的参数变为 512, 我们观察到如下结果

```
Epoch 1/10, Train Loss: 208.1651, Train Acc: 0.9322, Test Loss: 17.5645, Test Acc: 0.9645
Epoch 2/10, Train Loss: 81.2691, Train Acc: 0.9731, Test Loss: 14.8479, Test Acc: 0.9698
Epoch 3/10, Train Loss: 55.1731, Train Acc: 0.9812, Test Loss: 10.3777, Test Acc: 0.9801
Epoch 4/10, Train Loss: 41.6617, Train Acc: 0.9857, Test Loss: 12.3175, Test Acc: 0.9768
Epoch 5/10, Train Loss: 36.2846, Train Acc: 0.9876, Test Loss: 10.2069, Test Acc: 0.9802
Epoch 6/10, Train Loss: 26.6799, Train Acc: 0.9903, Test Loss: 12.6861, Test Acc: 0.9786
Epoch 7/10, Train Loss: 23.5218, Train Acc: 0.9919, Test Loss: 12.9256, Test Acc: 0.9772
Epoch 8/10, Train Loss: 19.4462, Train Acc: 0.9929, Test Loss: 13.5735, Test Acc: 0.9778
Epoch 9/10, Train Loss: 20.4269, Train Acc: 0.9926, Test Loss: 11.0868, Test Acc: 0.9808
Epoch 10/10, Train Loss: 17.0010, Train Acc: 0.9943, Test Loss: 14.3588, Test Acc: 0.9799
```

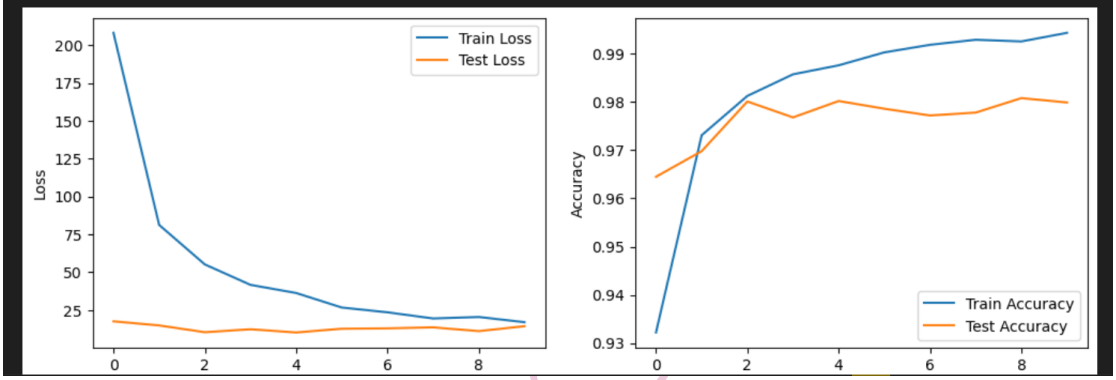

图 5: 隐藏层为 512 的评估结果

我们观察到相较于原始 MLP, 改进之后的收敛速度更快, 还是因为增加了隐藏层神经元的个数, 增强模型表达能力, 但是选择怎么样的宽度会使准确率变得更大有待进一步优化验证.

3. 优化器参数优化

在上面我们选择的优化器均为 Adam 优化器, 参数均为 0.001, 下面我们将尝试优化优化器参数. 我们选取了学习率范围测试的方法寻找最优学习率, 具体代码如下”

```
def find_lr(model, train_loader, criterion, init_lr=1e-5, max_lr=1,
    num_iter=100):
optimizer = optim.Adam(model.parameters(), lr=init_lr)
lr_values = []
losses = []

for i, (inputs, labels) in enumerate(train_loader):
    if i >= num_iter:
        break
    lr = init_lr * (max_lr / init_lr) ** (i / num_iter)
    optimizer.param_groups[0]['lr'] = lr

    optimizer.zero_grad()
    outputs = model(inputs)
    loss = criterion(outputs, labels)
```

```
        loss.backward()
        optimizer.step()

        lr_values.append(lr)
        losses.append(loss.item())

    plt.plot(lr_values, losses)
    plt.xscale('log')
    plt.xlabel('Learning Rate')
    plt.ylabel('Loss')
    plt.title('Learning Rate vs Loss')
    plt.show()
```

然后通过上面的代码我们得到最优的学习率在 0.001 0.01 之间, 我们调整参数为 0.0015, 得到如下的结果:

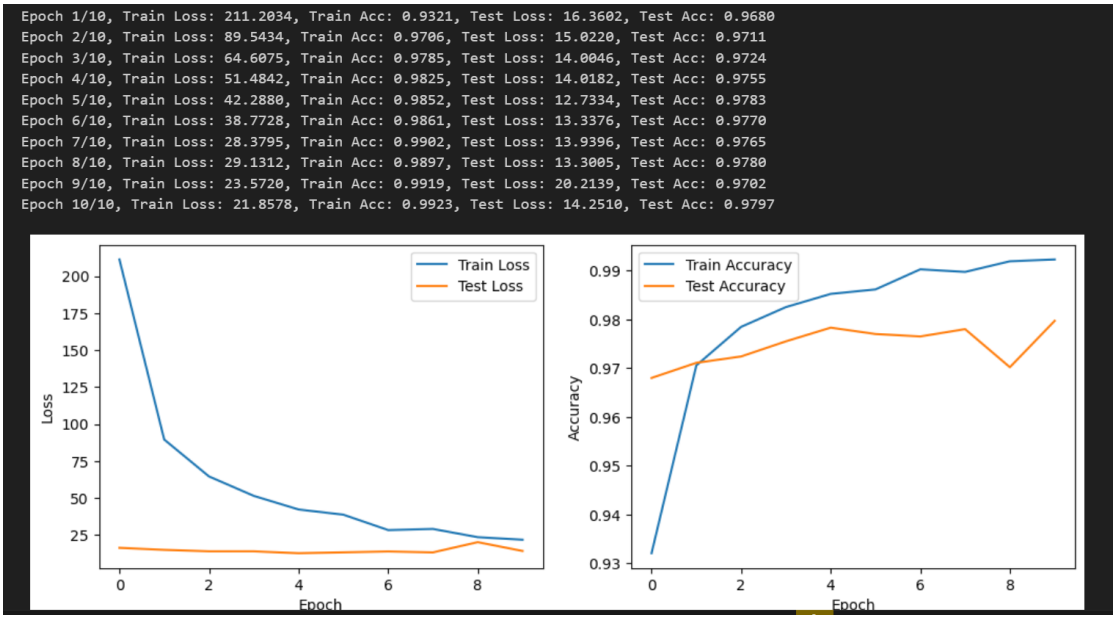

图 6: 优化器参数为 0.0015 的结果

我们观察到改变优化器参数并没有显著提高最后的结果, 反而使得准确率变低, 如果优化器参数调整到 0.002, 一开始的结果就会变得非常低. 由此可见, 优化器参数的改变不能显著增加准确率, 在具体的模型训练中, 选择一个固定的学习率并不是一个很好的方法, 最好的方法是学习率随训练过程的变化, 限于时间, 本人 bing'wei'shi'xian

(五) ResMLP 的实现

ResMLP 相对于传统 MLP 增加了残差连接的结构, 具体实现代码如下所示:

```
class ResMLP(nn.Module):
def __init__(self):
    super(ResMLP, self).__init__()
    self.fc1 = nn.Linear(784, 256)
    self.fc2 = nn.Linear(256, 256)  # 确保输入输出维度相同以便残差连接
```

```
        self.fc3 = nn.Linear(256, 64)
        self.fc4 = nn.Linear(64, 64)  # 确保输入输出维度相同以便残差连接
        self.fc5 = nn.Linear(64, 10)
        self.relu = nn.ReLU()

    def forward(self, x):
        x = x.view(x.size(0), -1)  # 展平输入

        # 第一层
        out = self.relu(self.fc1(x))

        # 第二层 + 残差连接
        residual = out
        out = self.relu(self.fc2(out))
        out = out + residual

        # 第三层
        out = self.relu(self.fc3(out))

        # 第四层 + 残差连接
        residual = out
        out = self.relu(self.fc4(out))
        out = out + residual# 添加残差

        # 输出层
        out = self.fc5(out)

        return out
```

接下来我们观察其结果:

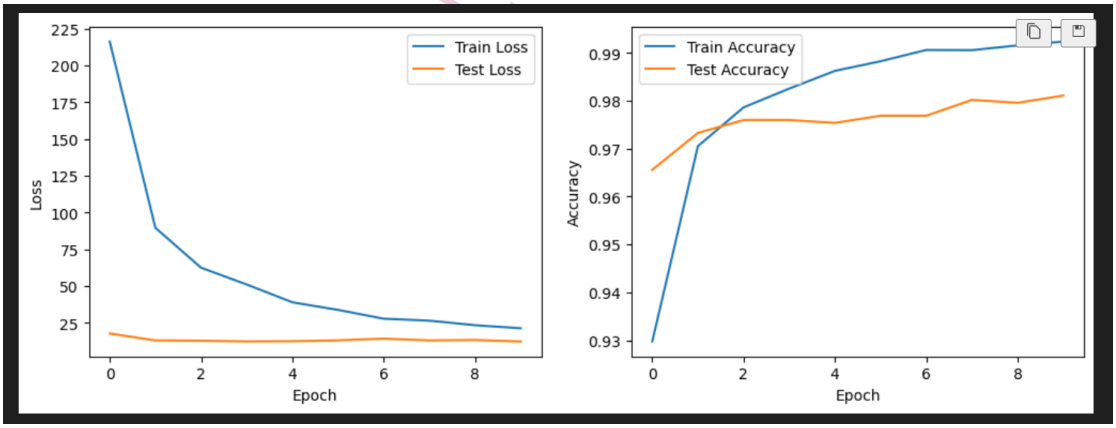

图 7: ResMLP 的评估结果

我们观察到增加残差连接的 MLP 的准确率显然要比没有的更好, 这是因为其能够保留前面学习到的特征, 帮助模型更好地捕捉数据的复杂模式, 并提高模型的性能和稳定性。

四、 总结

本次实验成功实现 MLP 模型, 并通过调整网络结构成功让准确率提升, 还实现了基本的 RES 结构, 通过这次实验我们前馈神经网络的结构更加熟悉了.