



南开大学
Nankai University

南开大学

计算机学院和密码与网络空间安全学院

《深度学习及应用》课程作业

Lab01：前馈神经网络

姓名：陆皓喆

学号：2211044

专业：信息安全

指导教师：李重仪

2025 年 5 月 5 日

目录

1 实验目的	2
2 实验要求	2
3 实验原理	2
4 实验环境	2
5 复现——运行 MLP Base 模型	3
6 优化——调节 MLP 部分参数	4
7 进阶——实现 MLP-Mixer	5
8 实验结果与分析	8
8.1 MLP Base 架构	8
8.2 优化后的 MLP 架构	9
8.3 MLP Mixer 架构	9
8.4 总结	10
9 探究——optim 算法与参数对实验结果的影响	11
9.1 改变 optim 类型进行测试	11
9.2 改变 lr 参数进行测试	11
10 总结与体会	12
11 文件目录结构	13

1 实验目的

1. 掌握前馈神经网络（FFN）的基本原理；
2. 学会使用 PyTorch 搭建简单的 FFN 实现 MNIST 数据集分类；
3. 掌握如何改进网络结构、调试参数以提升网络识别性能。

2 实验要求

1. 运行原始版本 MLP，查看网络结构、损失和准确度曲线；
2. 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；
3. 分析与总结格式不限；
4. 挑选 **MLP-Mixer**[1]，**ResMLP**[2]，**Vision Permutator**[3] 中的一种进行实现（加分项）。

3 实验原理

前馈神经网络（Feed - Forward Neural Network, FFN）是一种最基本的神经网络结构，是一种单向的神经网络，信息从输入层开始，经过隐藏层的处理，最终传递到输出层，在这个过程中，信息始终是向前传递的，不会出现反馈连接。

前馈神经网络主要由以下部分构成：

- **输入层：**负责接收外部输入数据，将数据传递给下一层。输入层的神经元数量取决于输入数据的特征数量。
- **隐藏层：**位于输入层和输出层之间，可以有一层或多层。隐藏层中的神经元对输入数据进行非线性变换和特征提取，是神经网络进行学习和表示复杂函数的关键部分。
- **输出层：**根据隐藏层的输出产生最终的预测结果或决策。输出层的神经元数量通常与任务的目标相关，例如在分类任务中，输出层的神经元数量可能等于类别数。

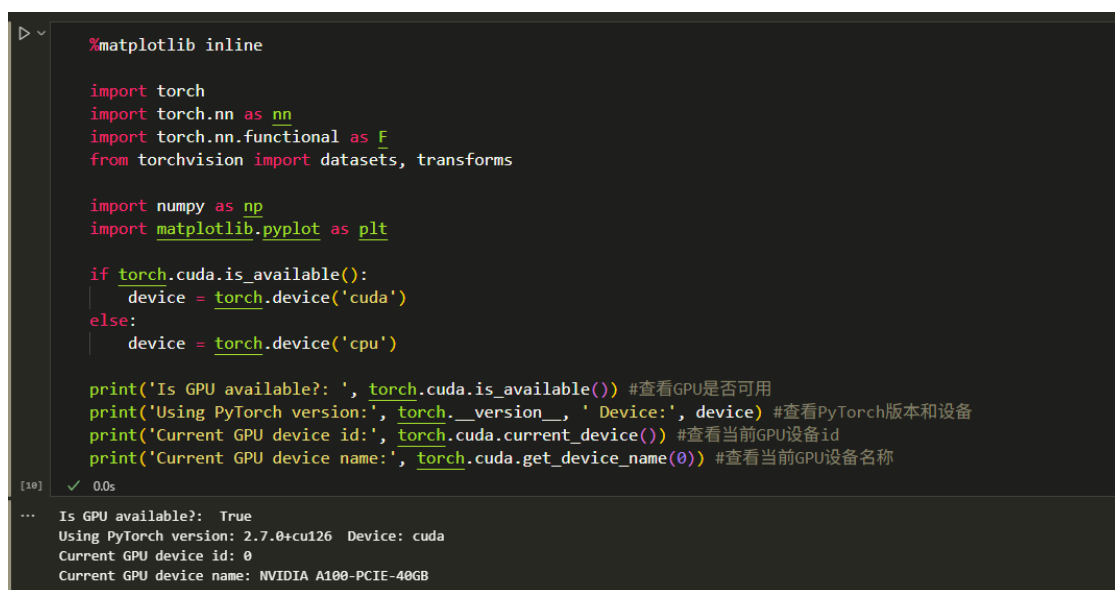
其中，比较关键的一步就是激活函数的选取。激活函数的作用是神经网络引入非线性特性，使得神经网络能够表示复杂的非线性函数。常见的激活函数有 **Sigmoid 函数**、**ReLU 函数**等。此处课堂上均进行介绍，此处就不再进行展开阐述。

4 实验环境

首先，本人对实验文件中的库进行安装，为了方便复现，本人将实验所需的库汇总成了一个需求文件 requirements.txt，直接运行下面的命令行就可以完成实验依赖库的安装。

```
1 pip install -r requirements.txt
```

安装完毕后，本人编写代码，对实验环境进行查看。



```
%matplotlib inline

import torch
import torch.nn as nn
import torch.nn.functional as F
from torchvision import datasets, transforms

import numpy as np
import matplotlib.pyplot as plt

if torch.cuda.is_available():
    device = torch.device('cuda')
else:
    device = torch.device('cpu')

print('Is GPU available?: ', torch.cuda.is_available()) #查看GPU是否可用
print('Using PyTorch version:', torch.__version__, ' Device:', device) #查看PyTorch版本和设备
print('Current GPU device id:', torch.cuda.current_device()) #查看当前GPU设备id
print('Current GPU device name:', torch.cuda.get_device_name(0)) #查看当前GPU设备名称

[10] ✓ 0.0s

... Is GPU available?: True
Using PyTorch version: 2.7.0+cu126 Device: cuda
Current GPU device id: 0
Current GPU device name: NVIDIA A100-PCIE-40GB
```

图 4.1: 实验环境查看

如图4.1所示，本次实验本人的环境情况为：

- GPU 是否可用：是
- PyTorch 版本：2.7.0+cu126
- 使用的 GPU id：0（默认）
- GPU 显卡类型：NVIDIA A100-PCIE-40GB

本次实验我选用了清华大学某实验室的 4 卡 A100-40GB 的服务器（linux 环境）进行实验。下面，就可以开始本次的实验了！

5 复现——运行 MLP Base 模型

本人按照本次实验给出的代码进行基础 MLP 模型的复现。从实验代码可以看出，MLP Base 实现了一个很简单的 MLP 结构，由三个全连接层构成，具体设计代码如下所示：

```
1 class BaseNet(nn.Module):
2     def __init__(self):
3         super(BaseNet, self).__init__()
4         self.fc1 = nn.Linear(28*28, 100)
5         self.fc1_drop = nn.Dropout(0.2)
6         self.fc2 = nn.Linear(100, 80)
7         self.fc2_drop = nn.Dropout(0.2)
8         self.fc3 = nn.Linear(80, 10)
9
10    def forward(self, x):
11        x = x.view(-1, 28*28)
12        x = F.relu(self.fc1(x))
```

```
13         x = self.fc1_drop(x)
14         x = F.relu(self.fc2(x))
15         x = self.fc2_drop(x)
16         return F.log_softmax(self.fc3(x), dim=1)
17
18 model = BaseNet().to(device)
19 optimizer = torch.optim.SGD(model.parameters(), lr=0.01, momentum=0.5)
20 criterion = nn.CrossEntropyLoss()
```

运行该文件，输出对应的模型参数，如下所示：

```
1 BaseNet(
2   (fc1): Linear(in_features=784, out_features=100, bias=True)
3   (fc1_drop): Dropout(p=0.2, inplace=False)
4   (fc2): Linear(in_features=100, out_features=80, bias=True)
5   (fc2_drop): Dropout(p=0.2, inplace=False)
6   (fc3): Linear(in_features=80, out_features=10, bias=True)
7 )
```

可以看到，该模型使用了 Dropout 来进行正则化，防止训练的过拟合。本人首先对其进行复现测试，发现其训练准确率基本可以达到 0.9847 左右（结果会在实验结果与分析一节中具体阐述），说明实际上该模型的准确率已经非常高了。

另外，我发现该示例代码采用 optim 中的 SGD 模块进行优化，对应的参数为 lr=0.01, mometum=0.5，此处我想到：这两个参数对于模型的准确率和损失是否有影响？是否可以通过调整这两个参数的值来对模型进行优化？是否可以修改优化算法来优化我的模型？这些问题，留到本报告的最后进行测试与分析。

6 优化——调节 MLP 部分参数

经过前面的复现测试，我也已经对 MLP 的结构有了大致的了解了。这个小节我将对原先的 MLP 结构进行修改，从而达到优化的效果。注意，此处的优化是指对 MLP 自身参数和结构的调整，而不是对 optim 参数进行调整。在这个部分本人暂且固定对应的优化模型和优化参数。

经过大量的调参实验，我发现，如果在原始的三层全连接层的基础上对参数进行调整，是达不到很好的效果的。因此，我设计了一个四层全连接层的 MLP 网络，具体设计代码如下所示：

```
1 class FineTuneNet(nn.Module):
2     def __init__(self):
3         super(FineTuneNet, self).__init__()
4         self.fc1 = nn.Linear(28*28, 512)
5         self.fc1_drop = nn.Dropout(0.2)
6         self.fc2 = nn.Linear(512, 512)
7         self.fc2_drop = nn.Dropout(0.5)
8         self.fc3 = nn.Linear(512, 128)
```

```
9         self.fc3_drop = nn.Dropout(0.2)
10        self.fc4 = nn.Linear(128, 10)
11
12        def forward(self, x):
13            x = x.view(-1, 28*28)
14            x = F.relu(self.fc1(x))
15            x = self.fc1_drop(x)
16            x = F.relu(self.fc2(x))
17            x = self.fc2_drop(x)
18            x = F.relu(self.fc3(x))
19            x = self.fc3_drop(x)
20            return self.fc4(x)
```

运行该文件，输出以下的模型参数，如下所示：

```
1 FineTuneNet(
2   (fc1): Linear(in_features=784, out_features=512, bias=True)
3   (fc1_drop): Dropout(p=0.2, inplace=False)
4   (fc2): Linear(in_features=512, out_features=512, bias=True)
5   (fc2_drop): Dropout(p=0.5, inplace=False)
6   (fc3): Linear(in_features=512, out_features=128, bias=True)
7   (fc3_drop): Dropout(p=0.2, inplace=False)
8   (fc4): Linear(in_features=128, out_features=10, bias=True)
9 )
```

本人在原先的基础上，添加了一个全连接层和一个防止过拟合层，顺便修改了部分特征参数。在中间层，本人添加了一个“拟合修改层”，也就是输入和输出特征数均为 512，然后使用参数 p 为 0.5 的 Dropout 来防止过拟合。这样，基本上可以保证在训练轮次为第二轮的时候就基本完成训练操作，后续的调整会更加优化我的模型。

通过测试，可以发现，基本上训练准确率达到了 **0.9920** 左右，相比于前面的基础模型有了很大的提升！（后续展示全部结果）

7 进阶——实现 MLP-Mixer

这一部分，本人阅读文献 [1]，准备实现 MLP-Mixer。在实现之前，需要简单了解一下 Mixer 的整体架构。本人截取论文中的模型示例图进行详细阐释，如图7.2所示：

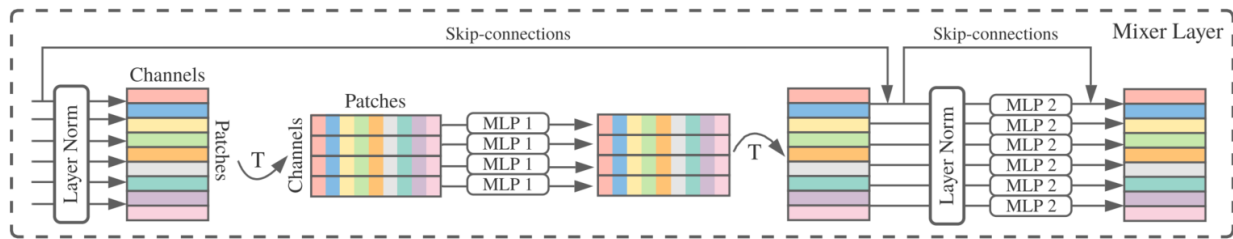


图 7.2: MLP-Mixer 总体架构 [1]

经过文章的阅读，我了解到，Mixer 的整体思路为：先将输入图片拆分成多个 patches，通过 Per-patch Fully-connected 层的操作将每个 patch 转换成 feature embedding，然后送入 N 个 Mixer Layer 中。最后，Mixer 将标准分类头与全局平均池化层配合使用，随后使用 Fully-connected 进行分类。

根据图7.3可以发现，实际上该 MLP 的结构为：

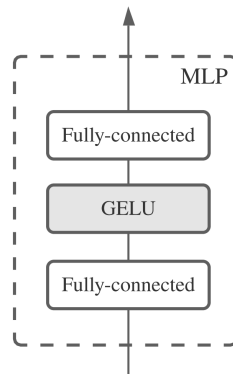


图 7.3: Mixer 的 MLP 结构 [1]

可以发现，该结构是由一个全连接层、一个 GELU 层和一个全连接层构成。对不同的 patches，都会使用不同的 MLP 去进行处理，每个 MLP 的结构是相同的。

熟悉了整体的设计思路后，就可以开始复现该结构了。最后，本人设计的 MLP-Mixer 架构如下所示：

```

1 class MixerNet(nn.Module):
2     def __init__(self, image_size=28, channels=1, patch_size=7, hidden_dim=512,
3         ↪ num_classes=10):
4         super(MixerNet, self).__init__()
5
6         self.patch_size = patch_size
7         self.num_patches = (image_size // patch_size) ** 2
8         self.hidden_dim = hidden_dim
9
10        self.fc1 = nn.Linear(channels * patch_size * patch_size, hidden_dim)
11        self.fc2 = nn.Linear(self.num_patches, self.num_patches)
12        self.fc3 = nn.Linear(self.num_patches, self.num_patches)
13        self.fc4 = nn.Linear(hidden_dim, hidden_dim)
14        self.fc5 = nn.Linear(hidden_dim, hidden_dim)

```

```
14         self.fc6 = nn.Linear(hidden_dim, num_classes)
15
16     def forward(self, x):
17
18         N, C, H, W = x.size()
19
20         patch_size = self.patch_size
21         unfold = nn.Unfold(kernel_size=patch_size, stride=patch_size)
22         x = unfold(x)
23         x = x.transpose(1, 2)
24
25         x = self.fc1(x)
26
27         y = x.transpose(1, 2)
28         y = F.gelu(self.fc2(y))
29         y = F.gelu(self.fc3(y))
30         y = y.transpose(1, 2)
31         x = x + y
32
33         y = F.gelu(self.fc4(x))
34         y = F.gelu(self.fc5(y))
35         x = x + y
36
37         x = x.mean(dim=1)
38
39         x = self.fc6(x)
40         return x
```

因为输入的图片大小为 28×28 ，所以需要将 patch 大小设计为 7×7 ，一张图像会被划分成 16 个 patch。

运行上述文件，得到该模型的具体架构如下所示：

```
1 MixerNet(
2   (fc1): Linear(in_features=49, out_features=512, bias=True)
3   (fc2): Linear(in_features=16, out_features=16, bias=True)
4   (fc3): Linear(in_features=16, out_features=16, bias=True)
5   (fc4): Linear(in_features=512, out_features=512, bias=True)
6   (fc5): Linear(in_features=512, out_features=512, bias=True)
7   (fc6): Linear(in_features=512, out_features=10, bias=True)
8 )
```

本人发现，通过 MLP-Mixer 的优化，我的训练准确率也来到了 **0.9951!**

而且，该模型的训练相比于前面两个模型，具有一定的特点。在第一轮的训练中，其验证集的准

准确率就可以达到 0.94 左右，这是其他两个模型所做不到的。我认为这是由于该模型的设计思路较为巧妙，使得首轮验证效果就非常好。

8 实验结果与分析

下面，就上面三个模型，本人来进行统一的结果分析。首先，说明一下本人的测试方法。我编写了统一的 train.py 文件和 main.py 文件，在主函数中编写 python 命令行的参数，通过不同的语句来完成不同模型的调用和训练。训练轮数均选用 20 轮，batch 大小均选用 64，lr 参数固定使用 0.001，优化器统一使用 Adam。

8.1 MLP Base 架构

为了充分展示我的训练过程，本人在此展示一下我的训练日志，如图8.4所示。

```
Epoch 1/20, Train Loss: 0.4281, Train Accuracy: 0.8757, Valid Loss: 0.1678, Valid Accuracy: 0.9495, Time: 11.87s
Epoch 2/20, Train Loss: 0.1873, Train Accuracy: 0.9445, Valid Loss: 0.1188, Valid Accuracy: 0.9663, Time: 11.25s
Epoch 3/20, Train Loss: 0.1398, Train Accuracy: 0.9583, Valid Loss: 0.0974, Valid Accuracy: 0.9784, Time: 11.52s
Epoch 4/20, Train Loss: 0.1208, Train Accuracy: 0.9641, Valid Loss: 0.0872, Valid Accuracy: 0.9738, Time: 11.12s
Epoch 5/20, Train Loss: 0.1053, Train Accuracy: 0.9677, Valid Loss: 0.0845, Valid Accuracy: 0.9737, Time: 11.30s
Epoch 6/20, Train Loss: 0.0958, Train Accuracy: 0.9703, Valid Loss: 0.0815, Valid Accuracy: 0.9756, Time: 11.49s
Epoch 7/20, Train Loss: 0.0850, Train Accuracy: 0.9730, Valid Loss: 0.0776, Valid Accuracy: 0.9770, Time: 11.17s
Epoch 8/20, Train Loss: 0.0791, Train Accuracy: 0.9748, Valid Loss: 0.0764, Valid Accuracy: 0.9770, Time: 10.85s
Epoch 9/20, Train Loss: 0.0745, Train Accuracy: 0.9763, Valid Loss: 0.0789, Valid Accuracy: 0.9778, Time: 11.06s
Epoch 10/20, Train Loss: 0.0716, Train Accuracy: 0.9776, Valid Loss: 0.0766, Valid Accuracy: 0.9777, Time: 11.16s
Epoch 11/20, Train Loss: 0.0674, Train Accuracy: 0.9780, Valid Loss: 0.0799, Valid Accuracy: 0.9776, Time: 11.04s
Epoch 12/20, Train Loss: 0.0640, Train Accuracy: 0.9794, Valid Loss: 0.0784, Valid Accuracy: 0.9779, Time: 10.97s
Epoch 13/20, Train Loss: 0.0599, Train Accuracy: 0.9802, Valid Loss: 0.0792, Valid Accuracy: 0.9790, Time: 10.86s
Epoch 14/20, Train Loss: 0.0580, Train Accuracy: 0.9806, Valid Loss: 0.0821, Valid Accuracy: 0.9776, Time: 11.00s
Epoch 15/20, Train Loss: 0.0575, Train Accuracy: 0.9810, Valid Loss: 0.0746, Valid Accuracy: 0.9790, Time: 11.09s
Epoch 16/20, Train Loss: 0.0539, Train Accuracy: 0.9825, Valid Loss: 0.0814, Valid Accuracy: 0.9772, Time: 11.25s
Epoch 17/20, Train Loss: 0.0517, Train Accuracy: 0.9830, Valid Loss: 0.0754, Valid Accuracy: 0.9776, Time: 11.23s
Epoch 18/20, Train Loss: 0.0499, Train Accuracy: 0.9833, Valid Loss: 0.0815, Valid Accuracy: 0.9794, Time: 10.88s
Epoch 19/20, Train Loss: 0.0497, Train Accuracy: 0.9845, Valid Loss: 0.0770, Valid Accuracy: 0.9789, Time: 11.04s
Epoch 20/20, Train Loss: 0.0482, Train Accuracy: 0.9840, Valid Loss: 0.0770, Valid Accuracy: 0.9798, Time: 11.35s
Training Finished
Model Saved!
```

图 8.4: MLP Base 的训练日志

对应展示一下可视化的结果，如图8.5所示。

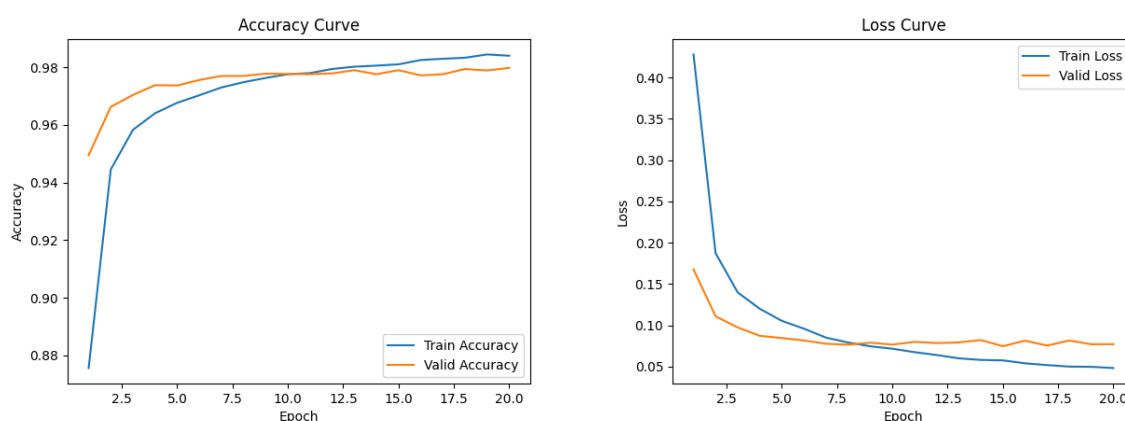


图 8.5: MLP Base 的 acc 展示

我们发现，基本上基础模型的训练准确率已经达到了 98.40%，验证准确率达到到了 97.98%，且没有发生过拟合的现象。

8.2 优化后的 MLP 架构

同样，展示一下本人的训练日志，如图8.6所示。

```
Epoch 1/20, Train Loss: 0.3812, Train Accuracy: 0.9077, Valid Loss: 0.1138, Valid Accuracy: 0.9661, Time: 11.03s
Epoch 2/20, Train Loss: 0.1237, Train Accuracy: 0.9634, Valid Loss: 0.0953, Valid Accuracy: 0.9717, Time: 11.36s
Epoch 3/20, Train Loss: 0.0957, Train Accuracy: 0.9721, Valid Loss: 0.0774, Valid Accuracy: 0.9779, Time: 12.19s
Epoch 4/20, Train Loss: 0.0780, Train Accuracy: 0.9763, Valid Loss: 0.0813, Valid Accuracy: 0.9761, Time: 11.77s
Epoch 5/20, Train Loss: 0.0686, Train Accuracy: 0.9795, Valid Loss: 0.0780, Valid Accuracy: 0.9790, Time: 11.56s
Epoch 6/20, Train Loss: 0.0567, Train Accuracy: 0.9826, Valid Loss: 0.0719, Valid Accuracy: 0.9793, Time: 11.43s
Epoch 7/20, Train Loss: 0.0525, Train Accuracy: 0.9839, Valid Loss: 0.0677, Valid Accuracy: 0.9808, Time: 11.37s
Epoch 8/20, Train Loss: 0.0497, Train Accuracy: 0.9850, Valid Loss: 0.0742, Valid Accuracy: 0.9806, Time: 12.07s
Epoch 9/20, Train Loss: 0.0460, Train Accuracy: 0.9857, Valid Loss: 0.0716, Valid Accuracy: 0.9814, Time: 13.37s
Epoch 10/20, Train Loss: 0.0409, Train Accuracy: 0.9873, Valid Loss: 0.0793, Valid Accuracy: 0.9810, Time: 11.92s
Epoch 11/20, Train Loss: 0.0410, Train Accuracy: 0.9876, Valid Loss: 0.0669, Valid Accuracy: 0.9827, Time: 11.31s
Epoch 12/20, Train Loss: 0.0367, Train Accuracy: 0.9888, Valid Loss: 0.0809, Valid Accuracy: 0.9822, Time: 11.34s
Epoch 13/20, Train Loss: 0.0355, Train Accuracy: 0.9891, Valid Loss: 0.0782, Valid Accuracy: 0.9804, Time: 11.52s
Epoch 14/20, Train Loss: 0.0350, Train Accuracy: 0.9894, Valid Loss: 0.0697, Valid Accuracy: 0.9824, Time: 12.58s
Epoch 15/20, Train Loss: 0.0303, Train Accuracy: 0.9909, Valid Loss: 0.0761, Valid Accuracy: 0.9822, Time: 11.71s
Epoch 16/20, Train Loss: 0.0303, Train Accuracy: 0.9905, Valid Loss: 0.0753, Valid Accuracy: 0.9819, Time: 11.54s
Epoch 17/20, Train Loss: 0.0288, Train Accuracy: 0.9913, Valid Loss: 0.0915, Valid Accuracy: 0.9815, Time: 11.18s
Epoch 18/20, Train Loss: 0.0296, Train Accuracy: 0.9915, Valid Loss: 0.0843, Valid Accuracy: 0.9837, Time: 11.42s
Epoch 19/20, Train Loss: 0.0267, Train Accuracy: 0.9921, Valid Loss: 0.0800, Valid Accuracy: 0.9836, Time: 11.33s
Epoch 20/20, Train Loss: 0.0263, Train Accuracy: 0.9920, Valid Loss: 0.0869, Valid Accuracy: 0.9835, Time: 11.31s
Training Finished
Model Saved!
```

图 8.6: MLP MySelf 的训练日志

对应展示一下可视化的结果，如图8.7所示。

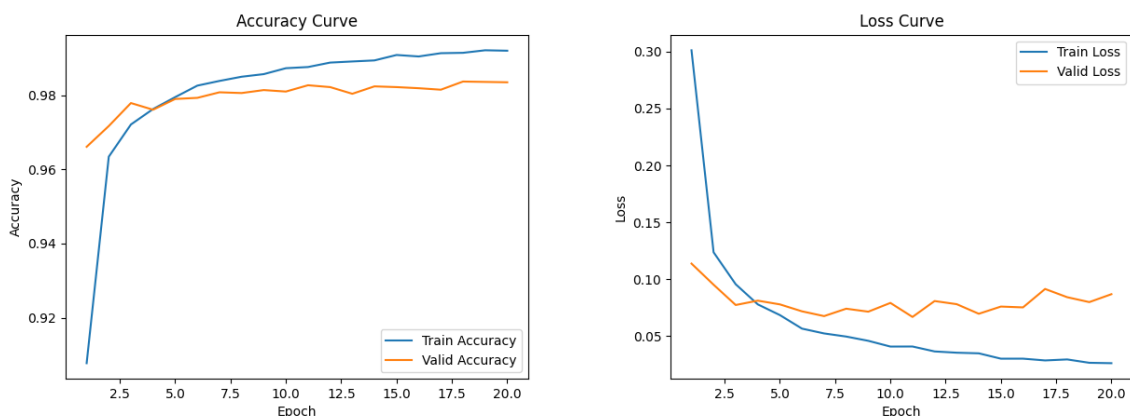


图 8.7: MLP myself 的 acc 展示

我发现，在本人的优化下，训练准确率成功从原先的 **98.40%** 提升到了 **99.20%**，而且验证准确率也得到了一些提升。说明我们的 MLP 结构的改进还是有一定的作用的。

8.3 MLP Mixer 架构

展示一下 Mixer 的训练日志，如图8.8所示。

```
Epoch 1/20, Train Loss: 0.4925, Train Accuracy: 0.8416, Valid Loss: 0.2073, Valid Accuracy: 0.9396, Time: 13.26s
Epoch 2/20, Train Loss: 0.1692, Train Accuracy: 0.9492, Valid Loss: 0.1323, Valid Accuracy: 0.9607, Time: 13.27s
Epoch 3/20, Train Loss: 0.1206, Train Accuracy: 0.9634, Valid Loss: 0.1225, Valid Accuracy: 0.9610, Time: 14.24s
Epoch 4/20, Train Loss: 0.0987, Train Accuracy: 0.9701, Valid Loss: 0.0974, Valid Accuracy: 0.9696, Time: 14.34s
Epoch 5/20, Train Loss: 0.0832, Train Accuracy: 0.9739, Valid Loss: 0.0994, Valid Accuracy: 0.9718, Time: 12.91s
Epoch 6/20, Train Loss: 0.0693, Train Accuracy: 0.9781, Valid Loss: 0.0751, Valid Accuracy: 0.9774, Time: 14.38s
Epoch 7/20, Train Loss: 0.0606, Train Accuracy: 0.9809, Valid Loss: 0.0744, Valid Accuracy: 0.9766, Time: 15.17s
Epoch 8/20, Train Loss: 0.0524, Train Accuracy: 0.9829, Valid Loss: 0.0911, Valid Accuracy: 0.9748, Time: 12.89s
Epoch 9/20, Train Loss: 0.0464, Train Accuracy: 0.9850, Valid Loss: 0.0792, Valid Accuracy: 0.9774, Time: 12.32s
Epoch 10/20, Train Loss: 0.0412, Train Accuracy: 0.9868, Valid Loss: 0.0693, Valid Accuracy: 0.9798, Time: 14.52s
Epoch 11/20, Train Loss: 0.0381, Train Accuracy: 0.9872, Valid Loss: 0.0748, Valid Accuracy: 0.9778, Time: 13.99s
Epoch 12/20, Train Loss: 0.0324, Train Accuracy: 0.9889, Valid Loss: 0.0752, Valid Accuracy: 0.9792, Time: 13.00s
Epoch 13/20, Train Loss: 0.0272, Train Accuracy: 0.9904, Valid Loss: 0.0823, Valid Accuracy: 0.9770, Time: 13.07s
Epoch 14/20, Train Loss: 0.0251, Train Accuracy: 0.9913, Valid Loss: 0.0826, Valid Accuracy: 0.9785, Time: 13.46s
Epoch 15/20, Train Loss: 0.0248, Train Accuracy: 0.9917, Valid Loss: 0.0764, Valid Accuracy: 0.9794, Time: 13.32s
Epoch 16/20, Train Loss: 0.0215, Train Accuracy: 0.9921, Valid Loss: 0.0703, Valid Accuracy: 0.9818, Time: 14.91s
Epoch 17/20, Train Loss: 0.0184, Train Accuracy: 0.9935, Valid Loss: 0.0866, Valid Accuracy: 0.9793, Time: 13.29s
Epoch 18/20, Train Loss: 0.0204, Train Accuracy: 0.9932, Valid Loss: 0.0941, Valid Accuracy: 0.9766, Time: 12.70s
Epoch 19/20, Train Loss: 0.0180, Train Accuracy: 0.9937, Valid Loss: 0.0871, Valid Accuracy: 0.9798, Time: 13.57s
Epoch 20/20, Train Loss: 0.0138, Train Accuracy: 0.9951, Valid Loss: 0.1074, Valid Accuracy: 0.9754, Time: 13.07s
Training Finished
Model Saved!
```

图 8.8: MLP Mixer 的训练日志

对应的，展示一下我的可视化结果，如图8.9所示。

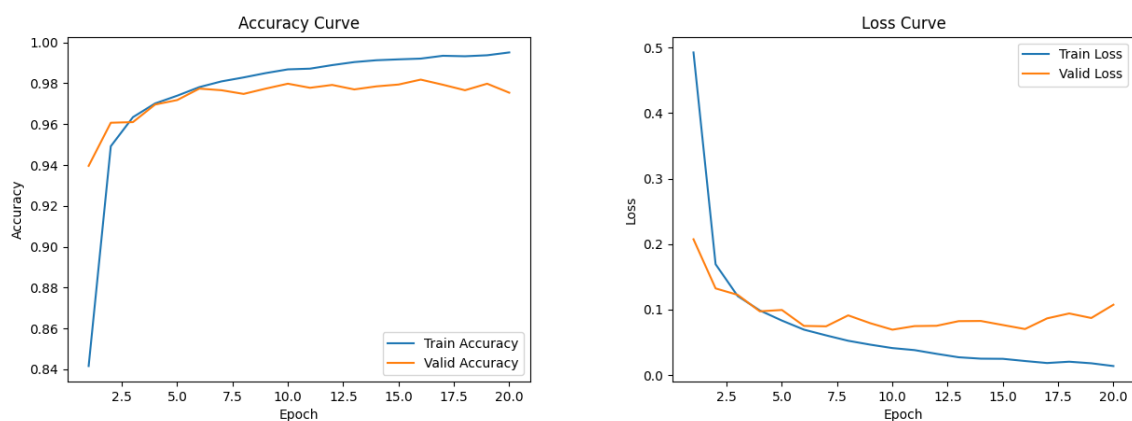


图 8.9: MLP mixer 的 acc 展示

我惊奇地发现，使用 Mixer 结构后，我们的训练正确率提升到了 **99.51%**，几乎来到了最高值。但是我们的验证准确率似乎没有比改进后的 MLP 要来得好。我猜测，可能是由于我们的**训练和测试的数据集过于简单**，导致我们复杂的 Mixer 结构没有发挥最好的效果。

8.4 总结

下面，我们列出一个表格1，来总结我们的三个模型的训练效果。

模型名称	训练准确率	验证准确率	训练损失	验证损失
MLP Base	98.40%	97.98%	4.82%	7.7%
MLP MySelf	99.20%	98.35%	2.63%	8.69%
MLP Mixer	99.51%	97.54%	1.38%	10.74%

表 1: 对比总结三个模型的各项数据

我们发现，**Mixer 模型**在训练准确率和训练损失上表现最好；而在验证的两项数据上不如其他两个模型。我们**优化后的 MLP 模型**在验证准确率上高于基础模型，但是**基础模型**的验证损失要比优化后的来的小。

到这里，我们的基本任务 + 进阶任务就都完成了。但是，上文中还有一个疑问没有解决。那就是针对优化机制的调整，是否能影响我们的训练结果？

9 探究——optim 算法与参数对实验结果的影响

为了解决上述的问题，我们在此处进一步展开研究与测试，探究 optim 类型和具体参数对我们训练效果的影响。

为了方便研究，此处我们选定**验证集准确率**为我们的优化目标，也就是说，只要验证集准确率高，就说明我们的结果好。我们固定选取 MLP 模型为**优化后的 MLP 架构**，调整我们的 **optim 类型**、**lr 参数**来探究我们的优化效果。

9.1 改变 optim 类型进行测试

我们稍微修改一下我们的测试代码，加入一个有关 optim 的列表，遍历列表进行测试，并分别输出其测试结果，方便我们进行对比。由于测试时间较长，此处仅选取某几个优化模型进行测试。

```
1 supported_optimizers = {  
2     'Adagrad': optim.Adagrad,  
3     'Adam': optim.Adam,  
4     'Rprop': optim.Rprop,  
5     'SGD': optim.SGD,  
6 }
```

运行，得到对应的结果如表2所示。

optim	模型名称	训练准确率	验证准确率	训练损失	验证损失
	Adagrad	0.9479	0.9572	0.1771	0.1420
	Adam	0.9914	0.9804	0.0273	0.0942
	Rprop	0.8203	0.8813	15.6644	9.0684
	SGD	0.7630	0.8290	0.7506	0.5917

表 2: optim 修改测试

我们发现，实际上 Adam 在我们的训练测试中表现的最好，其他的一些模型的准确率都不是特别高。这也是我们为什么选用 Adam 来进行实验的原因之一。

9.2 改变 lr 参数进行测试

我们接下来测试改变学习率，对我们训练效果的影响。我们分别选取五个不同的学习率，为 0.001, 0.01, 0.1, 0.2, 0.5，来进行测试。

训练结果如表3所示。

lr 学习率值	训练准确率	验证准确率	训练损失	验证损失
lr=0.001	0.9922	0.9824	0.0253	0.0869
lr=0.01	0.8758	0.9192	0.5667	0.5748
lr=0.1	0.1038	0.0980	2.3114	2.3115
lr=0.2	0.1025	0.1135	2.3186	2.3097
lr=0.5	0.1012	0.1028	2.3376	2.3155

表 3: lr 修改测试

我们发现，实际上默认的 lr 是最好的，学习率越大，会导致我们的训练结果大不如原先的结果。所以，我发现，学习率的改变会很大程度上对我们的训练结果产生影响！

10 总结与体会

本次实验，我深入了解了前馈神经网络的基本原理，还通过大量的对参数和网络结构进行调整的实验，了解到各参数对于实验结果的影响。下面，我就来简单地总结一下本次实验的收获。

Q1：本次实验我做了些什么？

1. 首先，我完成了基础 MLP 模型的设计，并进行了测试，发现在数据集上的表现非常好；
2. 进一步，通过修改层数和参数，我优化了原始的 MLP 模型，并进行了测试，发现结果要优于原先的模型，但是优化的程度不大；
3. 通过阅读文献，复现 MLP-Mixer 模型，并进行测试，获得与其余两种模型相似的结果；
4. 通过对优化器的测试和对各优化参数的测试，比较了各参数之间的实验结果，并发现实际上 Adam 优化器的效果要远远高于其他的模型；
5. 我还发现，随着学习率的上升，会极大程度上的影响识别率。一般来说，默认的值 0.001 是最好的；
6. 网络架构并不是越复杂越好。在一些简单的数据集上，越复杂的网络反而会造成过拟合的情况，会很大程度上降低我们的实验表现。对于本实验的数据集来说，一般最简单的三层网络或者四层网络已经是足够了；
7. 在层与层中间添加一层 Dropout 层可以很大程度防止出现过拟合现象。

Q2：还有哪些疑问？

1. 还是不太了解具体的 Adam 等优化器的内部原理，为什么 Adam 的效果就是要比其他的模型好？
2. 还是不太清楚 MLP-Mixer 为什么在本数据集上表现为什么不如 base 模型。尽管我猜测是因为 Mixer 过于复杂，但是并没有确切的事实依据可以证明这一点，在后续的学习中还有待商榷。

3. 在确定层数后，各参数的调整有没有一个具体的规律可以探寻？还是只能通过随机尝试来获得一个最佳的模型？

以上就是本次实验我的全部心得与体会了。在本次实验中，我基本上掌握了简单的 MLP 模型，并在不断的调参和尝试中学会了一些简单的优化策略。希望在后续的实验中，可以学习更多的深度学习的模型！

11 文件目录结构

本次实验的代码文件目录结构如下所示：

```
codes ..... 项目总目录
├── .gitignore ..... gitignore 文件
├── load_data.py ..... 用于加载数据
├── main.py ..... 主函数，用于训练
├── param_lr.py ..... 测试 lr 产生的影响
├── param_optim.py ..... 测试 optim 产生的影响
├── plot.py ..... 用于画图
├── README.md ..... 项目 README 文档
├── requirements.txt ..... 项目需求库
├── train.py ..... 编写了训练函数
├── __init__.py ..... 项目初始化，用于查看 GPU
├── results ..... 训练结果存储地址
│   ├── optimizer_comparison ..... optim 优化结果存放地址
│   │   ├── lr_optimizer_results.txt ..... 调整 lr 得到的结果
│   │   └── optim_optimizer_results.txt ..... 调整 optim 得到的结果
│   ├── mlp_myself ..... 优化后的 MLP
│   │   ├── accuracy_curve.png ..... 优化后的 MLP 准确率图像
│   │   ├── loss_curve.png ..... 优化后的 MLP 损失值图像
│   │   └── model.pth ..... 优化后的 MLP 模型文件
│   ├── mlp_mixer ..... MLP-Mixer
│   │   ├── accuracy_curve.png ..... MLP-Mixer 准确率图像
│   │   ├── loss_curve.png ..... MLP-Mixer 损失值图像
│   │   └── model.pth ..... MLP-Mixer 模型文件
│   └── mlp_base ..... 基础 MLP
│       ├── accuracy_curve.png ..... 基础 MLP 准确率图像
│       ├── loss_curve.png ..... 基础 MLP 损失值图像
│       └── model.pth ..... 基础 MLP 模型文件
├── model ..... 存放我们的 MLP 模型实现
│   ├── MLP_Base.py ..... MLP 基础代码
│   ├── MLP_Mixer.py ..... Mixer 复现代码
│   └── MLP_MySelf.py ..... 优化后的 MLP 代码
└── data ..... 主要存放实验使用的数据
```

本次实验的有关代码和文件，都已经上传至我的个人 github 中。您可以通过访问[此链接](#)来查阅我的代码文件。

参考文献

- [1] Ilya Tolstikhin, Neil Houlsby, Alexander Kolesnikov, Lucas Beyer, Xiaohua Zhai, Thomas Unterthiner, Jessica Yung, Andreas Steiner, Daniel Keysers, Jakob Uszkoreit, Mario Lucic, and Alexey Dosovitskiy. Mlp-mixer: An all-mlp architecture for vision, 2021.
- [2] Hugo Touvron, Piotr Bojanowski, Mathilde Caron, Matthieu Cord, Alaaeldin El-Nouby, Edouard Grave, Gautier Izacard, Armand Joulin, Gabriel Synnaeve, Jakob Verbeek, and Hervé Jégou. Resmlp: Feedforward networks for image classification with data-efficient training, 2021.
- [3] Qibin Hou, Zihang Jiang, Li Yuan, Ming-Ming Cheng, Shuicheng Yan, and Jiashi Feng. Vision permutator: A permutable mlp-like architecture for visual recognition, 2021.