## 前馈神经网络实验报告

姓名：张铭徐 学号：2113615

实验要求：

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

报告内容：

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

前馈神经网络（FFN）是一种人工神经网络，其中节点之间的连接不形成循环。这是最简单的神经网络架构之一，也是许多更复杂模型的基础。FFN 由输入层、一个或多个隐藏层以及输出层组成。下面是对其组件及其工作方式的简要概述：

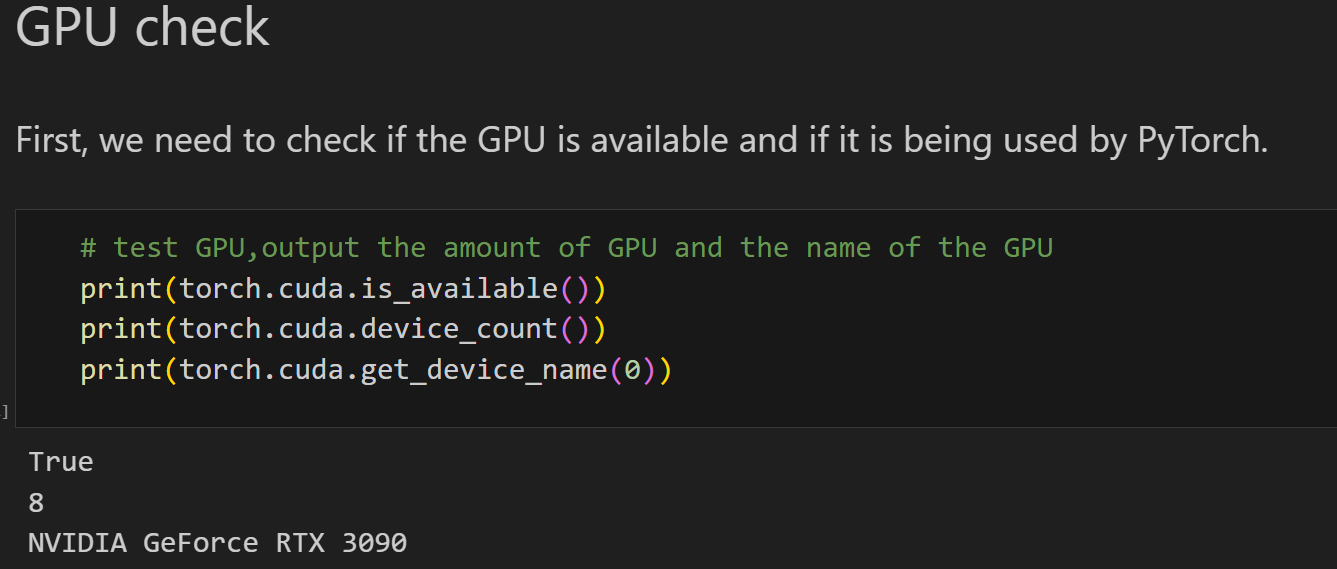
输入层：此层由接收输入特征并将它们传递到下一层的神经元（节点）组成，而不对它们进行任何计算。输入层中的每个神经元代表输入数据中的一个特征。

隐藏层：由一个或多个隐藏层使用权重连接和传递函数对来自前一层的输入进行计算。这些层是大部分处理通过应用激活函数到它们的输入的神经元进行的地方。

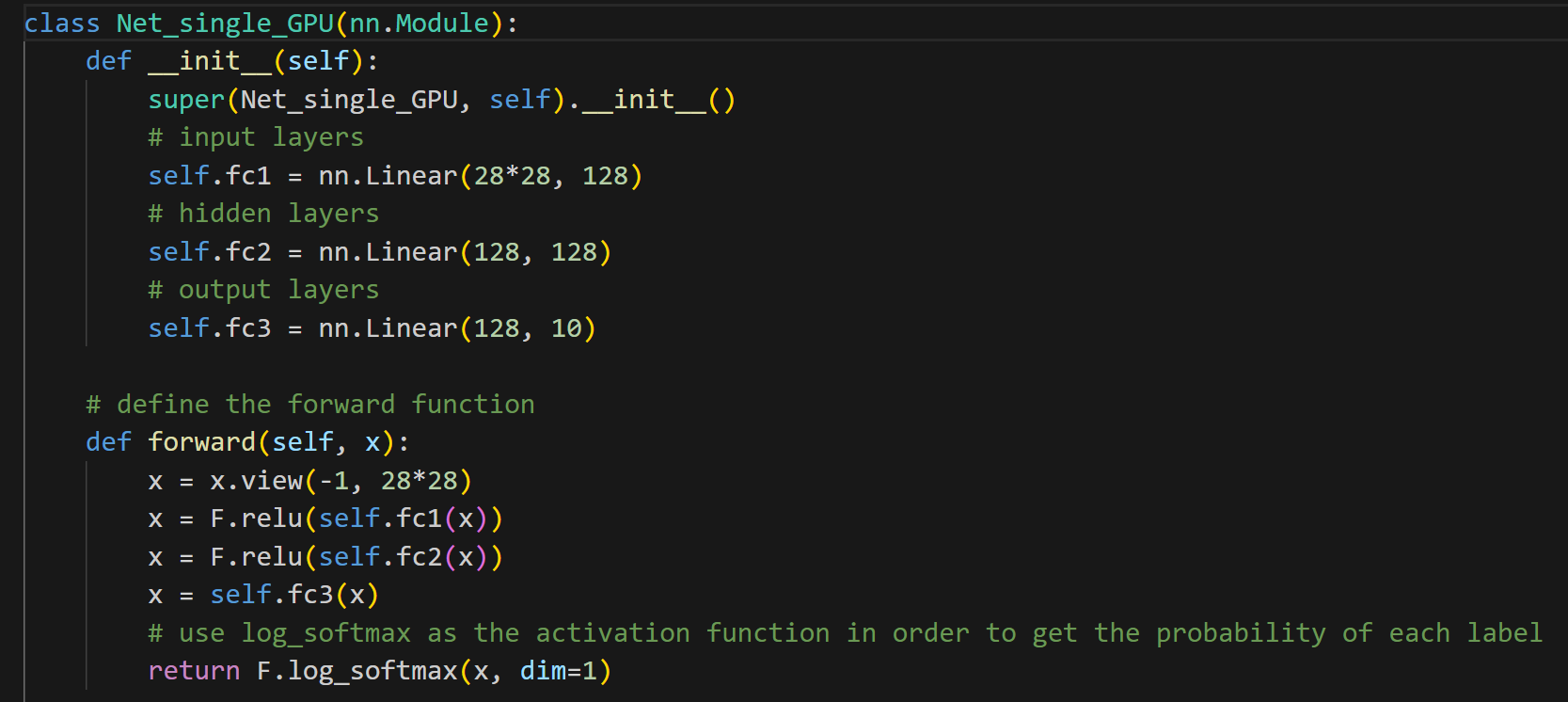
激活函数：在这些层中使用的函数如 ReLU（修正线性单元）、Sigmoid 或 Tanh，用于引入非线性，使模型能够学习复杂的模式。

输出层：前馈神经网络的最后一层。它将数据从网络的复杂内部表示简化为我们想要的输出形式（例如，分类任务的概率）。

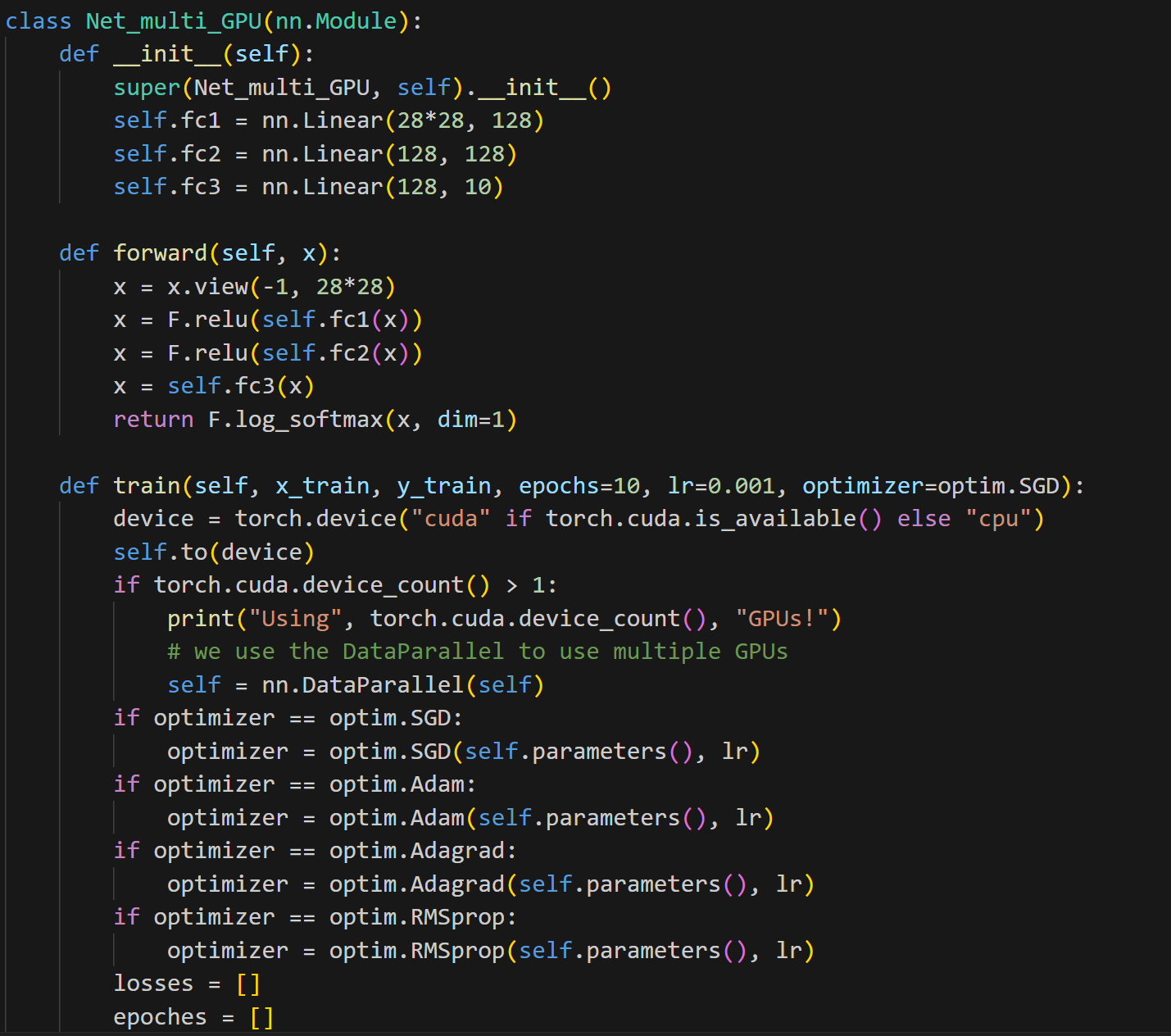
在本次实验中，我们使用8卡3090进行实验，并将实现的MLP运行在GPU上。



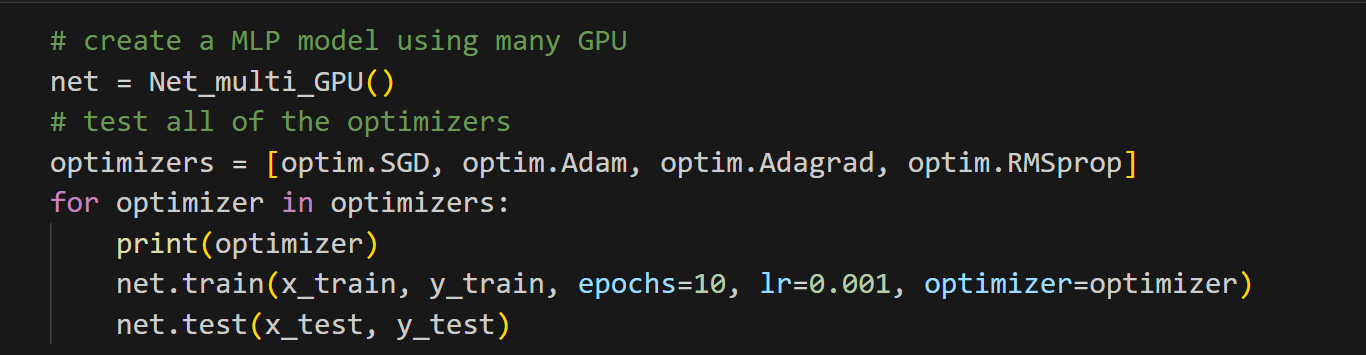
接下来我们自行实现MLP：



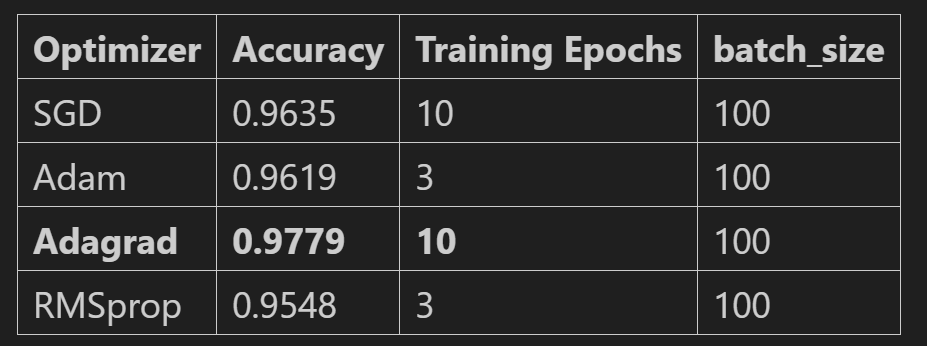
上图为单GPU实现MLP，我们同样实现了多GPU并行训练代码：



我们将使用多种优化器进行训练，探究不同优化器对性能的影响：



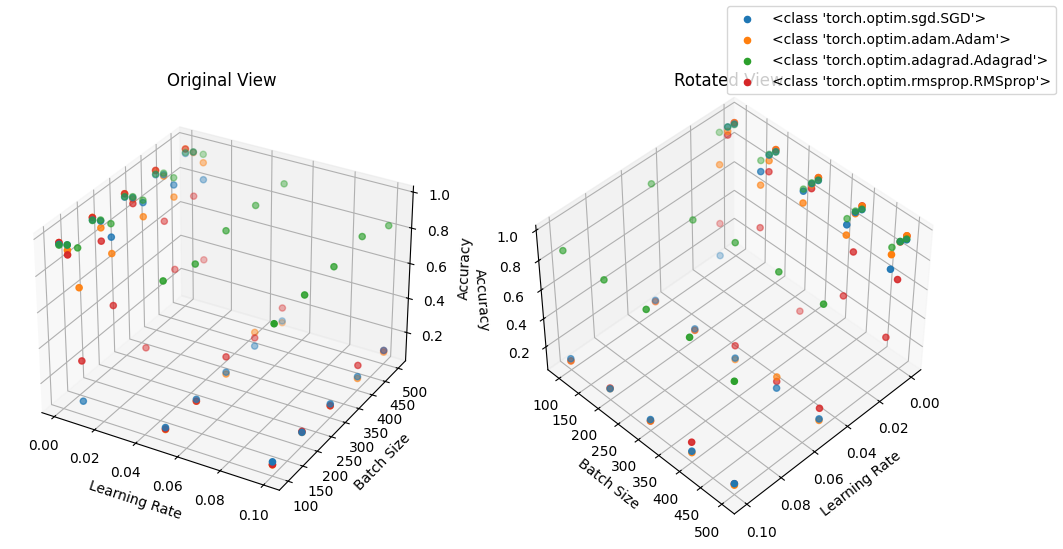
我们给出不同优化器的实验结果：



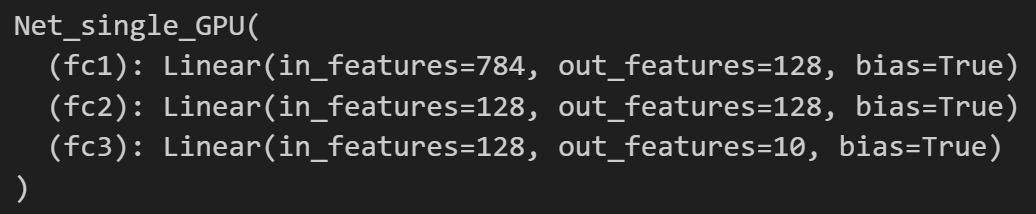
接下来我们尝试探究不同超参数对模型性能的影响：



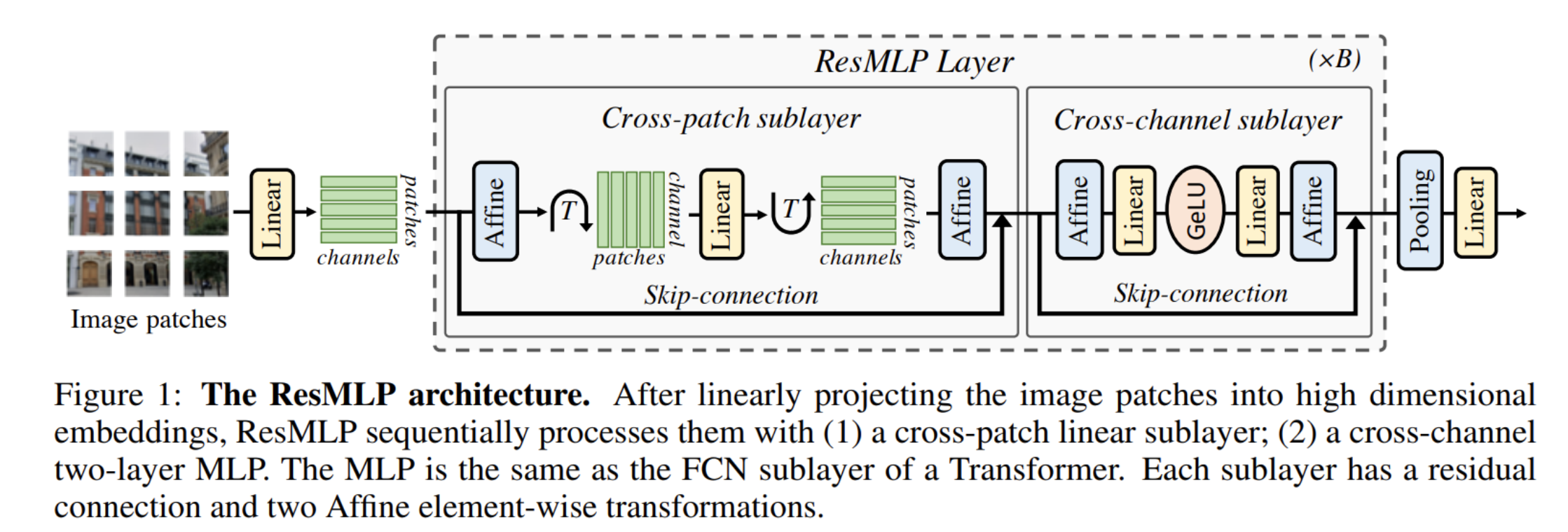
我们直接给出最后的结果图：



我们探究出最佳性能组合：Adam+0.01（lr）+200(batchsize)，在此实验设置之下能够得到0.9688的准确率。整体网络结果很简单，如下所示：



接下来我们实现ResMLP：



上图为ReSMLP结构图，其结合了ResNet和MLP，同样采用了跳连结构，只不过不使用卷积神经网络。我们给出代码：# implement affine transformation in order to change the input data dimension

class Affine(nn.Module):

    '''

    Arg:

    dim: the dimension of the input data

    self.alpha: linear parameter

    self.beta: offset parameter

    return alpha \* x + beta

    '''

    def \_\_init\_\_(self, dim):

        super().\_\_init\_\_()

        self.alpha = nn.Parameter(torch.ones(dim))

        self.beta = nn.Parameter(torch.zeros(dim))

    def forward(self, x):

        return self.alpha \* x + self.beta

# implement the cross-patch sublayer(in the middle of the figure)

class CrossPatchSublayer(nn.Module):

    def \_\_init\_\_(self, num\_patches, hidden\_dim):

        super().\_\_init\_\_()

        self.linear = nn.Linear(num\_patches, num\_patches)

        self.affine = Affine(hidden\_dim)

    def forward(self, x):

        # we change the dimension of the input data and then use the linear layer to change the data

        x = self.linear(x.transpose(1, 2)).transpose(1, 2)

        return self.affine(x)

# implement the cross-channel sublayer(in the last of the figure)

class CrossChannelSublayer(nn.Module):

    def \_\_init\_\_(self, hidden\_dim):

        super().\_\_init\_\_()

        self.mlp = nn.Sequential(

            nn.Linear(hidden\_dim, hidden\_dim \* 4),

            # GELU activation function

            nn.GELU(),

            nn.Linear(hidden\_dim \* 4, hidden\_dim)

        )

        self.affine = Affine(hidden\_dim)

    def forward(self, x):

        return self.affine(self.mlp(x))

# implement the ResMLP layer, which is the combination of the cross-patch sublayer and the cross-channel sublayer

class ResMLPLayer(nn.Module):

    def \_\_init\_\_(self, num\_patches, hidden\_dim):

        super().\_\_init\_\_()

        self.cross\_patch = CrossPatchSublayer(num\_patches, hidden\_dim)

        self.cross\_channel = CrossChannelSublayer(hidden\_dim)

    def forward(self, x):

        # Cross-patch sublayer with skip connection

        x = x + self.cross\_patch(x)

        # Cross-channel sublayer with skip connection

        x = x + self.cross\_channel(x)

        return x

# implement the ResMLP model using the ResMLP layer and the pooling layer

class ResMLP(nn.Module):

    def \_\_init\_\_(self, in\_channels, image\_size, patch\_size, hidden\_dim, num\_layers, num\_classes):

        super(ResMLP, self).\_\_init\_\_()

        # step 1: calculate the number of patches

        num\_patches = (image\_size // patch\_size) \*\* 2

        # step 2: image patching

        self.patch\_embed = nn.Conv2d(in\_channels, hidden\_dim, kernel\_size=patch\_size, stride=patch\_size)

        # step 3: position embedding

        self.pos\_embed = nn.Parameter(torch.zeros(1, hidden\_dim, num\_patches))

        # step 4: Repeat ResMLP layers

        self.layers = nn.Sequential(\*[ResMLPLayer((image\_size // patch\_size) \*\* 2, hidden\_dim) for \_ in range(num\_layers)])

        # step 5: pooling layer

        self.pooling = nn.AdaptiveAvgPool1d(1)

        # step 6: final linear layer

        self.final\_linear = nn.Linear(hidden\_dim, num\_classes)

    def forward(self, x):

        x = self.patch\_embed(x)

        # print("original shape",x.shape)

        x = x.flatten(2)

        # print("after\_flatten", x.shape)

        # x = x.transpose(1, 2)

        # print("afer transpose",x.shape)

        # print("pos\_embed", self.pos\_embed.shape)

        x = x + self.pos\_embed

        x = x.transpose(1, 2)

        x = self.layers(x)

        x = self.pooling(x.transpose(1, 2)).flatten(1)

        x = self.final\_linear(x)

        return x

    # save the model

    def save(self, path):

        torch.save(self.state\_dict(), path)

    # load the model

    def load(self, path):

        self.load\_state\_dict(torch.load(path))

    # train the model

    def train\_model(self, device, train\_loader, optimizer, epoch):

        self.train()

        criterion = nn.CrossEntropyLoss()

        for batch\_idx, (data, target) in enumerate(train\_loader):

            data, target = data.to(device), target.to(device)

            optimizer.zero\_grad()

            output = self(data)

            loss = criterion(output, target)

            loss.backward()

            optimizer.step()

            if batch\_idx % 100 == 0:

                print(f'Train Epoch: {epoch} [{batch\_idx \* len(data)}/{len(train\_loader.dataset)} ({100. \* batch\_idx / len(train\_loader):.0f}%)]\tLoss: {loss.item():.6f}')

    # test the model

    def test\_model(self, device, test\_loader):

        self.eval()

        criterion = nn.CrossEntropyLoss(reduction='sum')

        test\_loss = 0

        correct = 0

        with torch.no\_grad():

            for data, target in test\_loader:

                data, target = data.to(device), target.to(device)

                output = self(data)

                test\_loss += criterion(output, target).item()

                pred = output.argmax(dim=1, keepdim=True)

                correct += pred.eq(target.view\_as(pred)).sum().item()

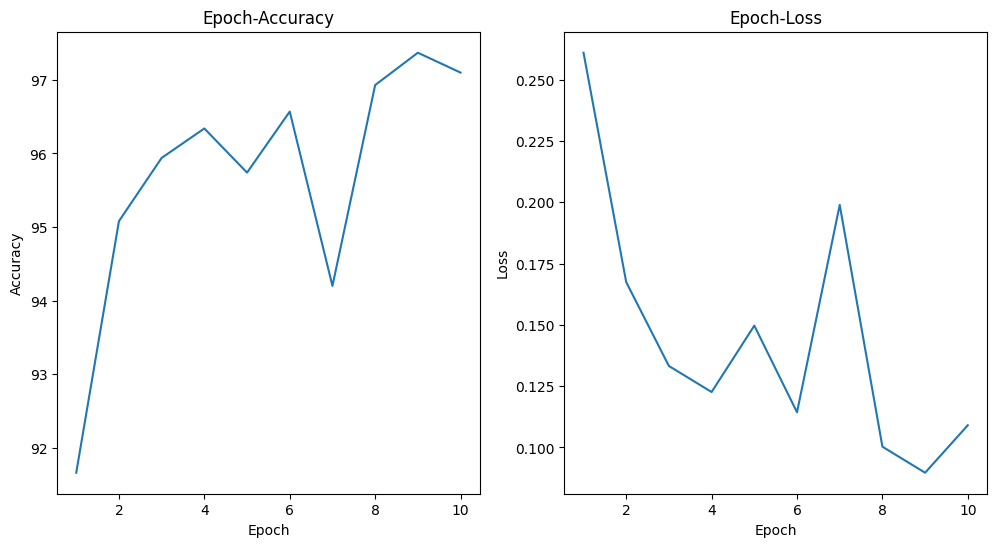
        test\_loss /= len(test\_loader.dataset)

        accuracy = 100. \* correct / len(test\_loader.dataset)

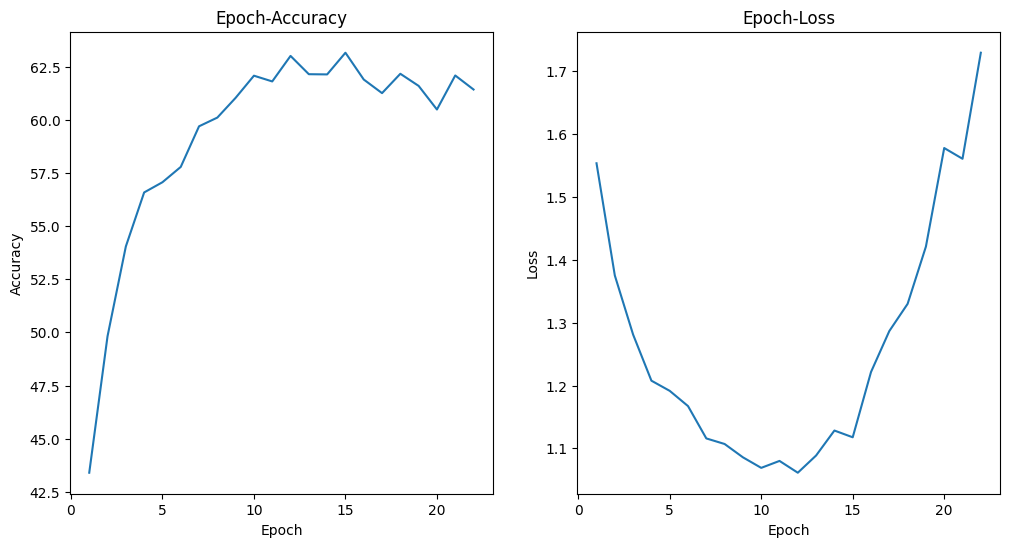
        print(f'\nTest set: Average loss: {test\_loss:.4f}, Accuracy: {correct}/{len(test\_loader.dataset)} ({accuracy:.0f}%)\n')

        return accuracy, test\_loss

在上面的代码中，我们进行训练，得到如下曲线：



我们可以看到，我们最终可以达到97.37%的准确率，比之前最佳超参数设置下的性能要好。同样的，我们在这里使用CIFAR-10数据集进行测试：



从该曲线中，我们可以看到，在CIFAR-10数据集上出现了较为严重的过拟合现象，我们可以采用L2正则化等手段进行惩罚，避免过大参数的存在，减少过拟合，增强模型泛化能力。