机器学习与数据挖掘 作业2实验报告

姓名：林筱涵
学号：22336141

问题:
探索神经网络在图像分类任务上的应用。在给定数据集 CIFAR-10 的训练集上训练模型，并在测试集上验证其性能。
要求:

1. 在给定的训练数据集上，分别训练一个线性分类器（Softmax 分类器），多层感知机
   （MLP）和卷积神经网络（CNN）。
2. 在 MLP 实验中，研究使用不同网络层数和不同神经元数量对模型性能的影响。
3. 在 CNN 实验中，以 LeNet 模型为基础，探索不同模型结构因素（如：卷积层数、滤
   波器数量、Pooling 的使用等）对模型性能的影响。
4. 分别使用 SGD 算法、SGD Momentum 算法和 Adam 算法训练模型，观察并讨论他们
   对模型训练速度和性能的影响。
5. 比较并讨论线性分类器、MLP 和 CNN 模型在 CIFAR-10 图像分类任务上的性能区别
6. 学习一种主流的深度学习框架（如：Tensorfolw，PyTorch，MindSpore），并用其中一种框架完成上述神经网络模型的实验。

1. 算法原理

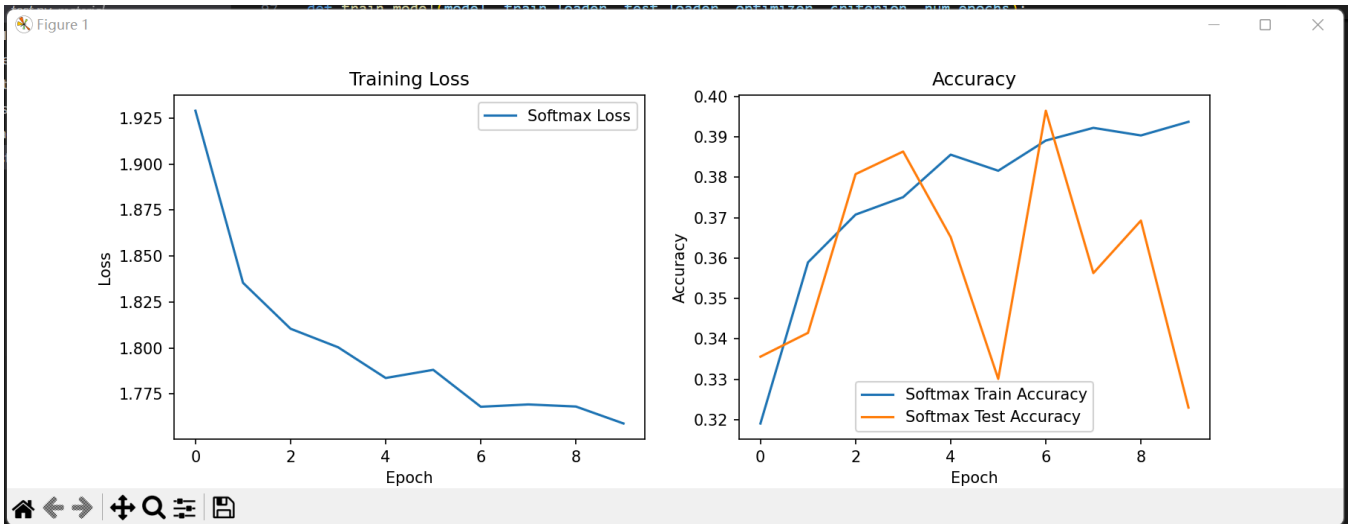
1.1 线性分类器

其输入大小:为3*3232，使用单层全连接，直接从输入到输出；
输出大小: 10（CIFAR-10 类别数）。
nn.linear是全连接层，定义了一个线性变换：
$\mathbf{Z} = \mathbf{X} \cdot \mathbf{W}^{\mathrm{T}} + \mathbf{b}$
其中**X**是输入数据的特征矩阵；**W**是权重矩阵；**b**是偏置向量；**Z**是输出得分矩阵。而Forward 方法将输入数据从多维数组转换为二维数组，然后计算每个样本对应于每个类别的未归一化得分。
具体代码如下：

```
class SoftmaxClassifier(nn.Module):
    def __init__(self, input_size, num_classes):
        super(SoftmaxClassifier, self).__init__()
        self.fc = nn.Linear(input_size, num_classes)

    def forward(self, x):
        x = x.view(x.size(0), -1)
        return self.fc(x)
```

运行结果如下：



1.2 多层感知机MIp

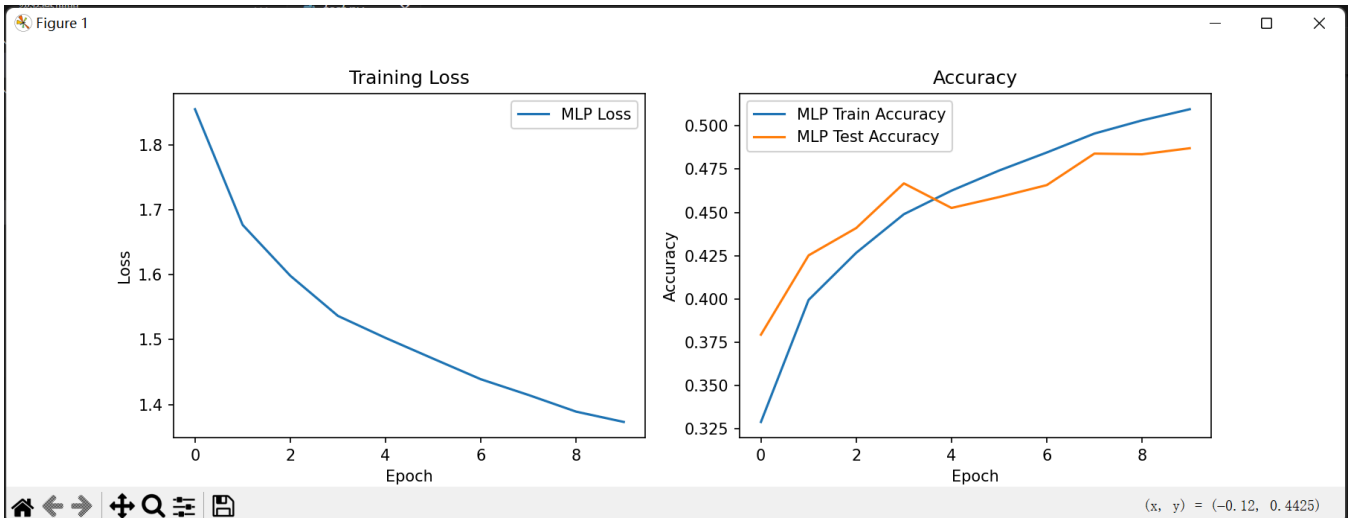
多层感知机是一种前馈神经网络，其由输入层、一个或多个隐藏层和输出层组成。它通过全连接层将输入数据逐层映射为高阶特征，并利用激活函数引入非线性，以实现对复杂映射关系的学习。

具体代码如下：

```
class MLP(nn.Module):
    def __init__(self, input_size, hidden_sizes, num_classes):
        super(MLP, self).__init__()
        layers = []
        for i, hidden_size in enumerate(hidden_sizes):
            layers.append(nn.Linear(input_size if i == 0 else hidden_sizes[i-1], hidden_size))
            layers.append(nn.ReLU())
        layers.append(nn.Linear(hidden_sizes[-1], num_classes))
        self.net = nn.Sequential(*layers)

    def forward(self, x):
        x = x.view(x.size(0), -1)
        return self.net(x)
```

输入层：在 forward 函数中将输入数据 x 的形状调整为二维，通过 .view(x.size(0), -1) 将每个样本展平成一维向量。
隐藏层：通过 nn.Linear 实现了输入层到隐藏层，以及隐藏层之间的连接,第一层的输入维度为input_size，后续隐藏层的输入维度与前一层的输出维度一致。
在每个隐藏层后使用nn.ReLU，将线性输出非线性化。
输出层：最后一层映射到类别数。
用nn.Sequential 构建网络结构，其用self.net将所有的层通过nn.Sequential封装起来，使得网络结构更简洁且易于扩展。 层的构造逻辑： 1.逐层添加隐藏层，层数由hidden_sizes列表长度决定。 2.最后一层连接隐藏层的输出到类别数。
定义MLP为MLP(3 * 32 * 32, [256, 128], 10)，即隐藏层为2层，神经元数量分别是256个和128个，输出层输出类别为10。运行结果如下图：



1.3 卷积神经网络CNN

卷积神经网络是一种专为处理具有网格结构数据而设计的深度学习模型。其核心是通过卷积操作

提取局部特征，逐层构建高阶特征表示，从而实现对复杂模式的捕捉。

具体代码如下：

```
class CNN(nn.Module):
    def __init__(self, num_classes):
        super(CNN, self).__init__()
        self.conv_layers = nn.Sequential(
            nn.Conv2d(3, 16, kernel_size=3, stride=1, padding=1), # Conv1
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),                # Pool1
            nn.Conv2d(16, 32, kernel_size=3, stride=1, padding=1),# Conv2
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),                # Pool2

        )
        self.fc_input_dim = None
        self.fc_layers = None
        self.num_classes = num_classes

    def _initialize_fc_layers(self, x_shape):
        self.fc_input_dim = x_shape[1] * x_shape[2] * x_shape[3]
        self.fc_layers = nn.Sequential(
            nn.Linear(self.fc_input_dim, 128),
            nn.ReLU(),
            nn.Linear(128, self.num_classes)
        )

    def forward(self, x):
        x = self.conv_layers(x)
        if self.fc_layers is None:
            self._initialize_fc_layers(x.shape)
        x = x.view(x.size(0), -1)
        return self.fc_layers(x)
```

构造函数 init：
self.conv_layers：定义了卷积神经网络的特征提取部分，由卷积层、ReLU激活函数、池化层以及新增的批归一化层组成。一共有2块卷积块。第一卷积块nn.Conv2d(3, 16, kernel_size=3, stride=1, padding=1)卷积核数量为16，第二卷积块卷积核数量增加至32。

self.fc_input_dim： 计算全连接层的输入维度

self.fc_layers： 定义分类器部分，包含两层全连接层和一个ReLU 激活函数：

第一层：将特征映射到 128 维。

第二层：将 128 维映射到最终的类别数。

前向传播forward：

1.先进行卷积层特征提取：

将输入图像x依次经过卷积层、激活函数和池化层。

2.全连接层初始化：

如果全连接层未定义，调用 _initialize_fc_layers 基于当前输入形状动态初始化。

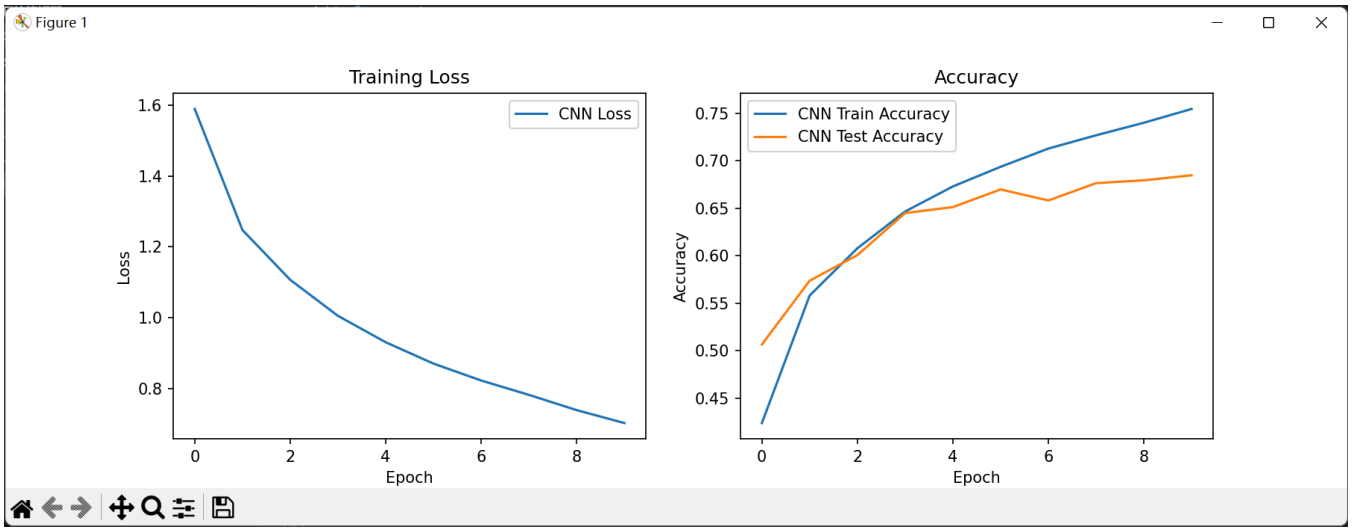
3.展平特征图：

使用x.view(x.size(0), -1) 将卷积层的输出展平为二维张量，作为全连接层的输入。

4.分类器输出：

将展平后的特征输入全连接层，生成最终的分类结果。

运行结果如下：



2. 训练过程和结果

2.1 数据预处理

代码如下：

```python
# 数据加载函数
def load_data(dir):
    import pickle
    import numpy as np
    X_train = []
    Y_train = []
    for i in range(1, 6):
        with open(dir + r'/data_batch_' + str(i), 'rb') as fo:
            dict = pickle.load(fo, encoding='bytes')
        X_train.append(dict[b'data'])
        Y_train += dict[b'labels']
    X_train = np.concatenate(X_train, axis=0)
    with open(dir + r'/test_batch', 'rb') as fo:
        dict = pickle.load(fo, encoding='bytes')
    X_test = dict[b'data']
    Y_test = dict[b'labels']

    return X_train, Y_train, X_test, Y_test

# 加载数据
data_dir = r"E:/robotstudy/Assignment2/material\data"
X_train, Y_train, X_test, Y_test = load_data(data_dir)

# 数据归一化和重塑
X_train = X_train.reshape(-1, 3, 32, 32) / 255.0  # 归一化到 [0, 1]
X_test = X_test.reshape(-1, 3, 32, 32) / 255.0

# 转换为 Tensor
X_train = torch.tensor(X_train, dtype=torch.float32)
Y_train = torch.tensor(Y_train, dtype=torch.long)
X_test = torch.tensor(X_test, dtype=torch.float32)
Y_test = torch.tensor(Y_test, dtype=torch.long)
```

1.数据加载:

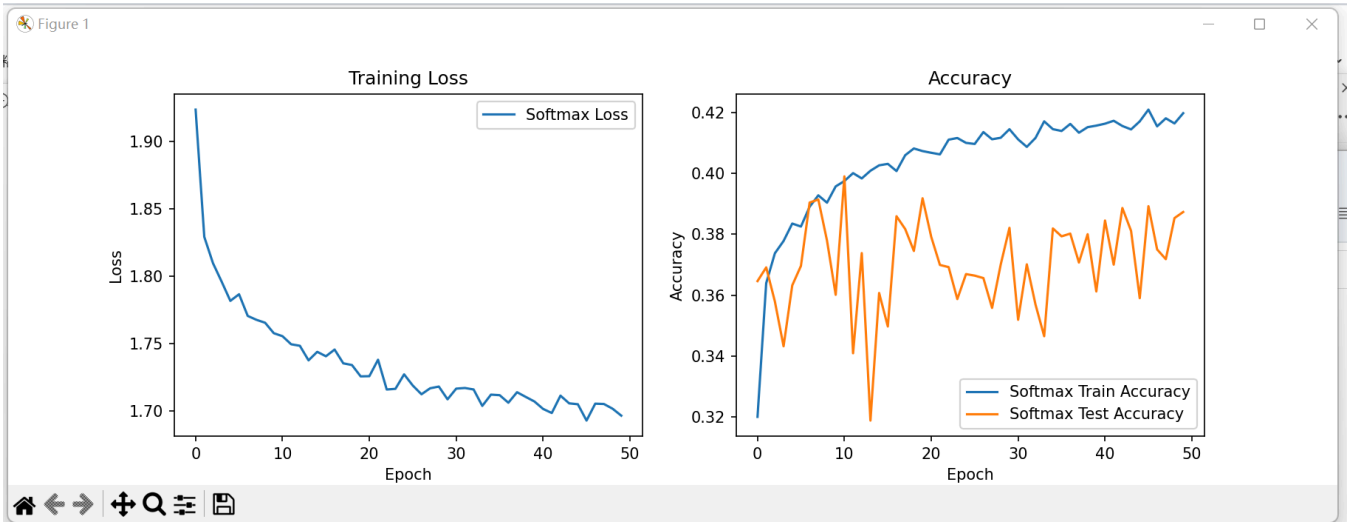
使用 load_data 函数从 CIFAR-10 数据集的本地文件中加载训练和测试数据；训练数据包括5个批次，分别合并为一个整体；测试数据为单独的文件。

2.数据归一化:

将数据从整数像素值 (0-255) 归一化到 [0,1]，通过X_train/255.0和X_test/255.0 实现。

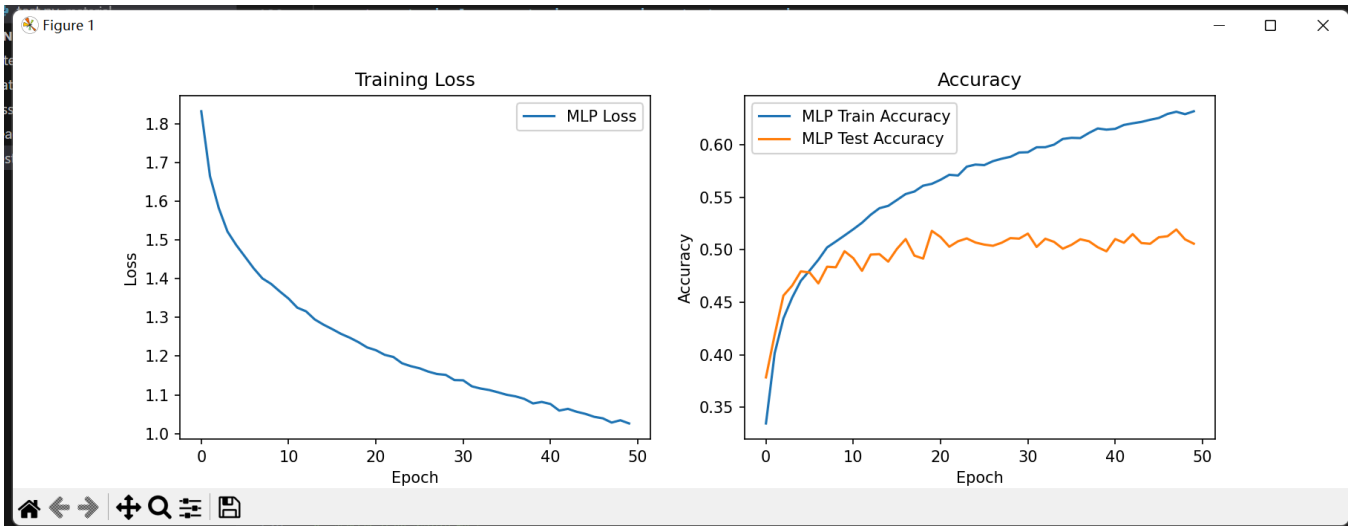
2.2 线性分类器

1.初始化：参数由nn.Linear初始化，默认使用PyTorch内部的Kaiming初始化。
2.超参数选择：输入维度3×32×32，输出类别数为10。
3.优化函数用Adam，迭代次数为50。
4.学习率：0.01。
5.批大小为64，即每次训练64个样本。
运行结果如下：



2.3 MLP

1.初始化：参数由nn.Linear初始化，默认使用PyTorch内部的Kaiming初始化。
2.超参数选择：
输入维度：3×32×32=3072。
隐藏层大小：[256, 128]。
输出类别数：10。
3.优化函数用Adam，迭代次数为50。
4.学习率：0.01。
5.批大小为64，即每次训练64个样本。
运行结果如下：



使用不同网络层数对MLP模型性能影响

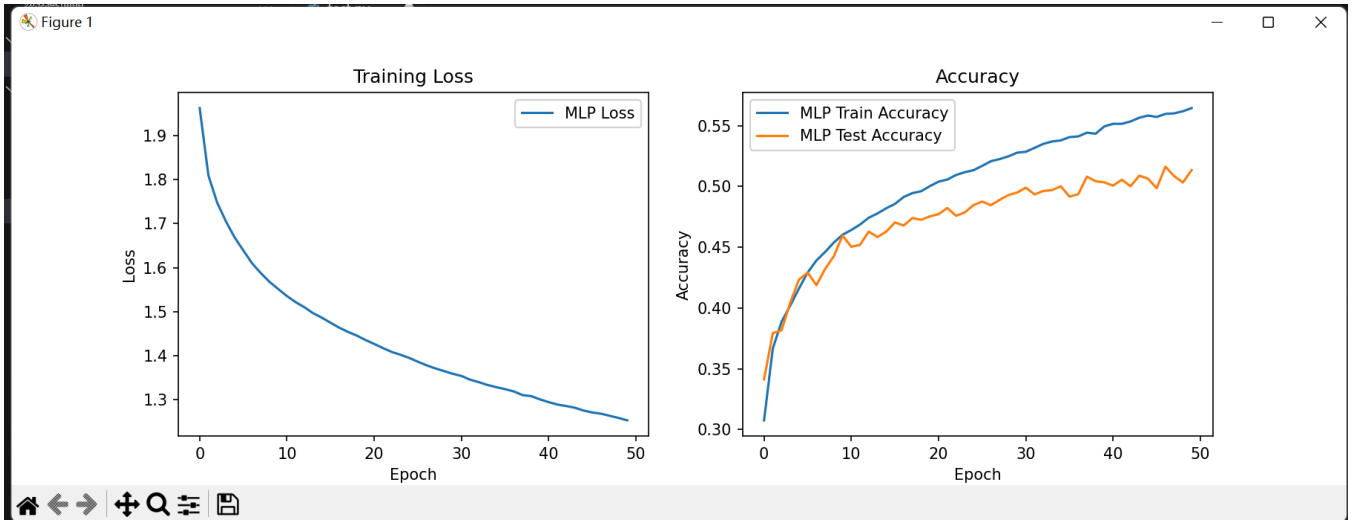
对于MLP，通过修改不同网络层数，该实验采用了3种，1是一层网络层[128],2是2层网络层[256, 128]，3是3层网络层[512, 256, 128]

修改该处代码可修改网络层数：

```
"MLP": MLP(3 * 32 * 32, [256, 128], 10)
```

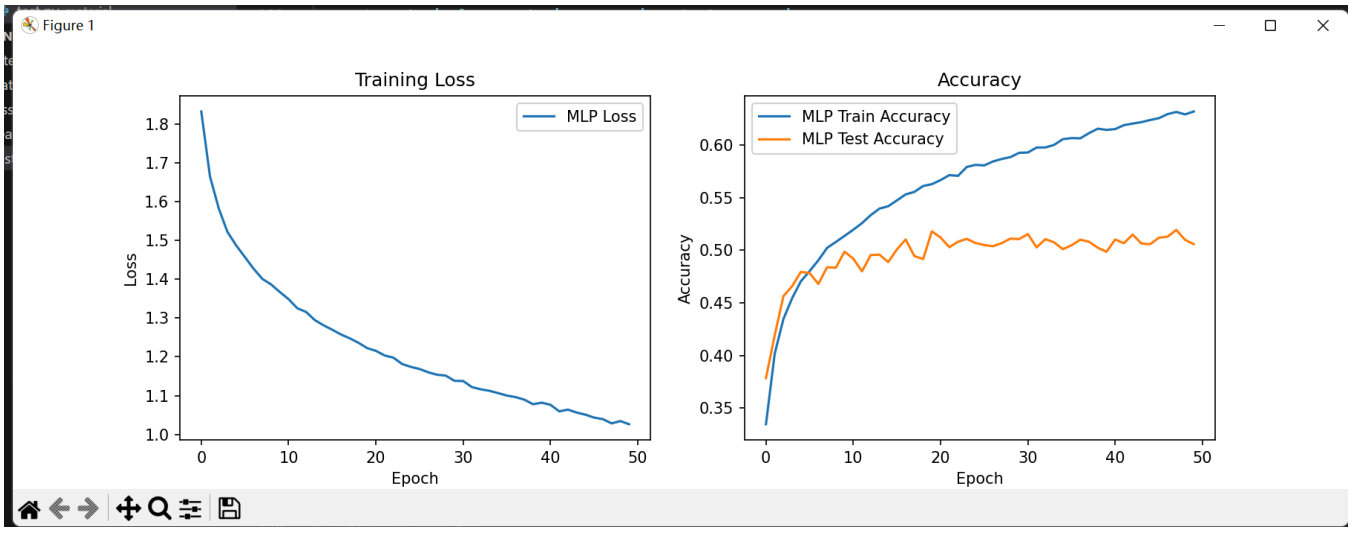
运行结果如下：

[128]:



```
问题  输出  调试控制台  终端  端口
Epoch 39/50, Loss: 1.3081, Train Accuracy: 0.5434, Test Accuracy: 0.5043
Epoch 40/50, Loss: 1.3007, Train Accuracy: 0.5495, Test Accuracy: 0.5034
Epoch 41/50, Loss: 1.2945, Train Accuracy: 0.5516, Test Accuracy: 0.5007
Epoch 42/50, Loss: 1.2890, Train Accuracy: 0.5516, Test Accuracy: 0.5056
Epoch 43/50, Loss: 1.2857, Train Accuracy: 0.5535, Test Accuracy: 0.5002
Epoch 44/50, Loss: 1.2819, Train Accuracy: 0.5566, Test Accuracy: 0.5089
Epoch 45/50, Loss: 1.2754, Train Accuracy: 0.5583, Test Accuracy: 0.5064
Epoch 46/50, Loss: 1.2710, Train Accuracy: 0.5572, Test Accuracy: 0.4986
Epoch 47/50, Loss: 1.2683, Train Accuracy: 0.5597, Test Accuracy: 0.5163
Epoch 48/50, Loss: 1.2634, Train Accuracy: 0.5602, Test Accuracy: 0.5085
Epoch 49/50, Loss: 1.2587, Train Accuracy: 0.5619, Test Accuracy: 0.5033
Epoch 50/50, Loss: 1.2531, Train Accuracy: 0.5645, Test Accuracy: 0.5135
Training Time: 494.54 seconds
```

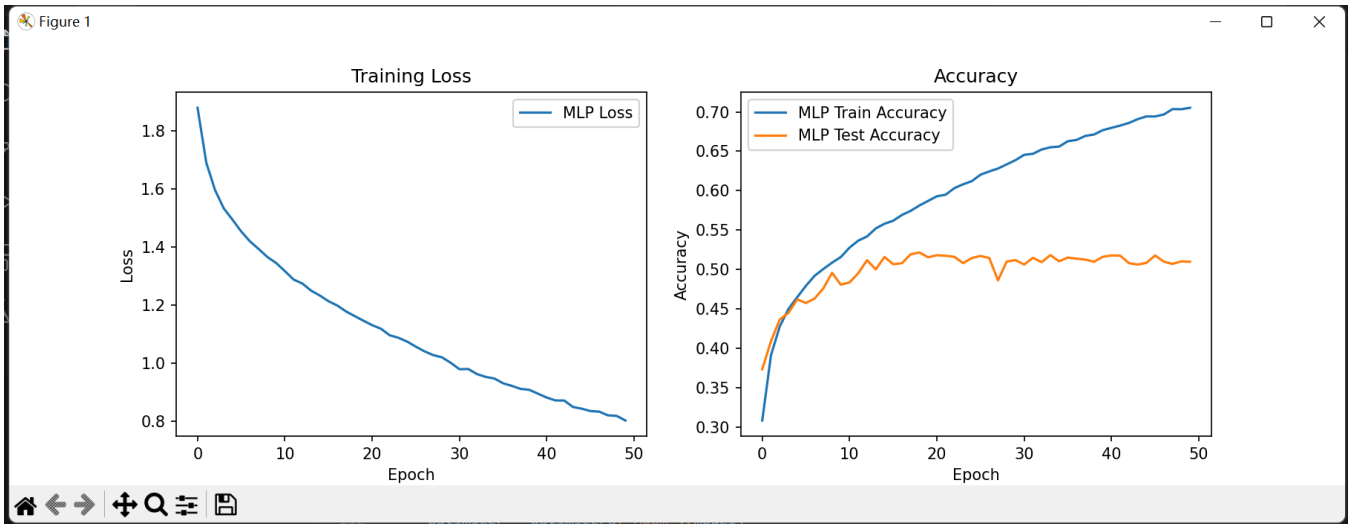
[256，128]:



问题 输出 调试控制台 终端 端口

```
Epoch 41/50, Loss: 1.0763, Train Accuracy: 0.6154, Test Accuracy: 0.5103
Epoch 42/50, Loss: 1.0593, Train Accuracy: 0.6191, Test Accuracy: 0.5068
Epoch 43/50, Loss: 1.0638, Train Accuracy: 0.6206, Test Accuracy: 0.5150
Epoch 44/50, Loss: 1.0563, Train Accuracy: 0.6220, Test Accuracy: 0.5065
Epoch 45/50, Loss: 1.0508, Train Accuracy: 0.6240, Test Accuracy: 0.5058
Epoch 46/50, Loss: 1.0432, Train Accuracy: 0.6258, Test Accuracy: 0.5120
Epoch 47/50, Loss: 1.0393, Train Accuracy: 0.6296, Test Accuracy: 0.5130
Epoch 48/50, Loss: 1.0284, Train Accuracy: 0.6317, Test Accuracy: 0.5194
Epoch 49/50, Loss: 1.0343, Train Accuracy: 0.6294, Test Accuracy: 0.5100
Epoch 50/50, Loss: 1.0261, Train Accuracy: 0.6321, Test Accuracy: 0.5059
Training Time: 1319.85 seconds
```

[512，256，128]:



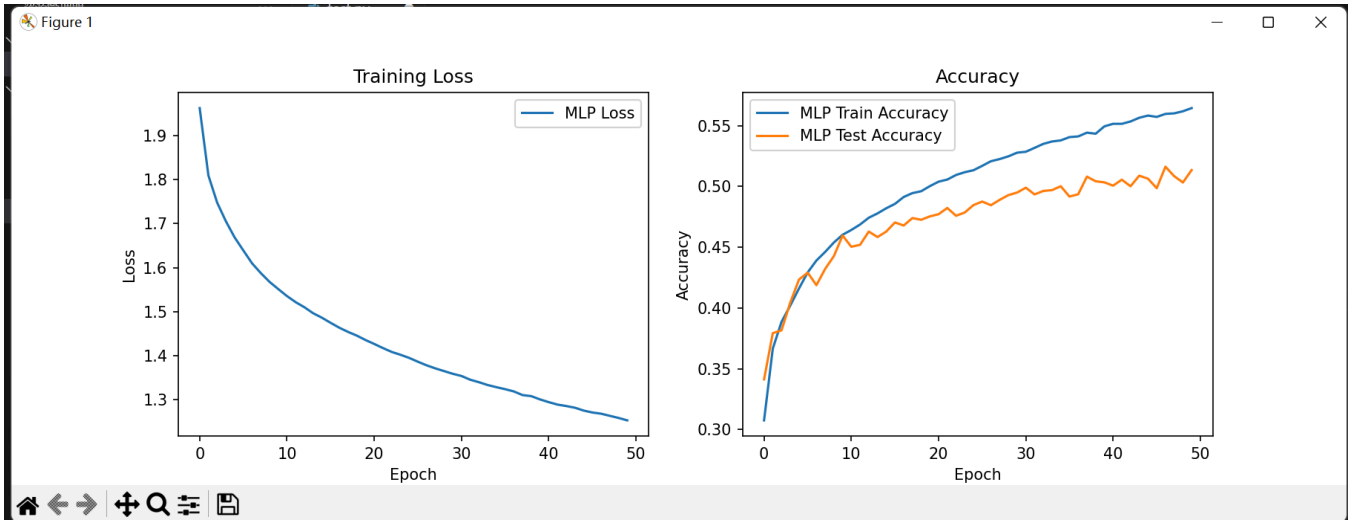
```
问题  输出  调试控制台  终端  端口
Epoch 38/50, Loss: 0.9102, Train Accuracy: 0.6695, Test Accuracy: 0.5127
Epoch 39/50, Loss: 0.9069, Train Accuracy: 0.6713, Test Accuracy: 0.5099
Epoch 40/50, Loss: 0.8935, Train Accuracy: 0.6768, Test Accuracy: 0.5162
Epoch 41/50, Loss: 0.8802, Train Accuracy: 0.6798, Test Accuracy: 0.5179
Epoch 42/50, Loss: 0.8704, Train Accuracy: 0.6826, Test Accuracy: 0.5177
Epoch 43/50, Loss: 0.8700, Train Accuracy: 0.6860, Test Accuracy: 0.5081
Epoch 44/50, Loss: 0.8478, Train Accuracy: 0.6907, Test Accuracy: 0.5064
Epoch 45/50, Loss: 0.8420, Train Accuracy: 0.6942, Test Accuracy: 0.5085
Epoch 46/50, Loss: 0.8338, Train Accuracy: 0.6942, Test Accuracy: 0.5179
Epoch 47/50, Loss: 0.8319, Train Accuracy: 0.6967, Test Accuracy: 0.5100
Epoch 48/50, Loss: 0.8191, Train Accuracy: 0.7036, Test Accuracy: 0.5072
Epoch 49/50, Loss: 0.8173, Train Accuracy: 0.7033, Test Accuracy: 0.5104
Epoch 50/50, Loss: 0.8013, Train Accuracy: 0.7052, Test Accuracy: 0.5099
Training Time: 3224.09 seconds
```

通过几次运行结果对比可以看出网络层数越多，训练时间越长，其准确率更快稳定，但其与训练准确率差别也更大，可能是因为网络过深但数据不足，故导致过拟合。

使用不同神经元数量对MLP模型性能影响

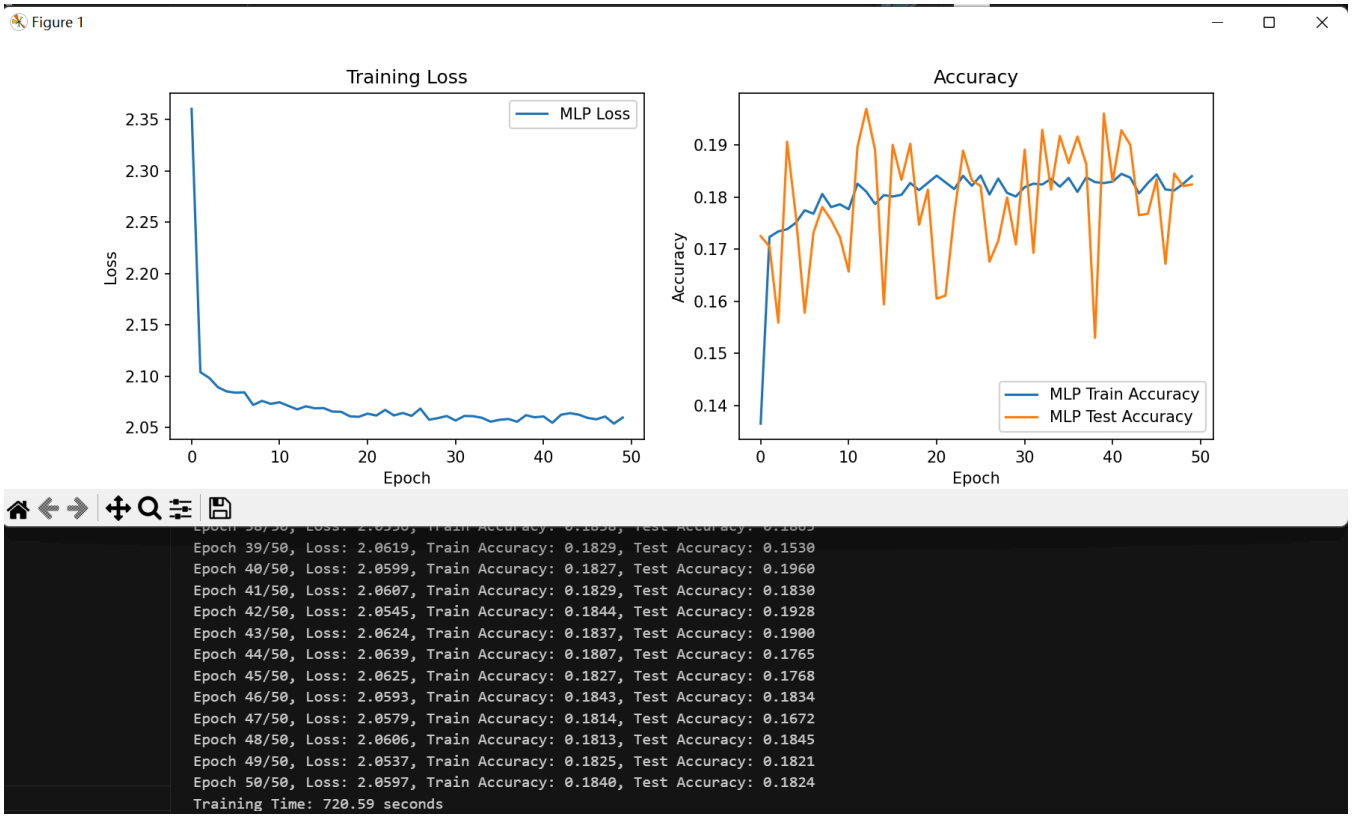
统一用一层网络层，分别用128和256各神经元数量

[128]:



```
问题  输出  调试控制台  终端  端口
Epoch 39/50, Loss: 1.3081, Train Accuracy: 0.5434, Test Accuracy: 0.5043
Epoch 40/50, Loss: 1.3007, Train Accuracy: 0.5495, Test Accuracy: 0.5034
Epoch 41/50, Loss: 1.2945, Train Accuracy: 0.5516, Test Accuracy: 0.5007
Epoch 42/50, Loss: 1.2890, Train Accuracy: 0.5516, Test Accuracy: 0.5056
Epoch 43/50, Loss: 1.2857, Train Accuracy: 0.5535, Test Accuracy: 0.5002
Epoch 44/50, Loss: 1.2819, Train Accuracy: 0.5566, Test Accuracy: 0.5089
Epoch 45/50, Loss: 1.2754, Train Accuracy: 0.5583, Test Accuracy: 0.5064
Epoch 46/50, Loss: 1.2710, Train Accuracy: 0.5572, Test Accuracy: 0.4986
Epoch 47/50, Loss: 1.2683, Train Accuracy: 0.5597, Test Accuracy: 0.5163
Epoch 48/50, Loss: 1.2634, Train Accuracy: 0.5602, Test Accuracy: 0.5085
Epoch 49/50, Loss: 1.2587, Train Accuracy: 0.5619, Test Accuracy: 0.5033
Epoch 50/50, Loss: 1.2531, Train Accuracy: 0.5645, Test Accuracy: 0.5135
Training Time: 494.54 seconds
```

[256]:

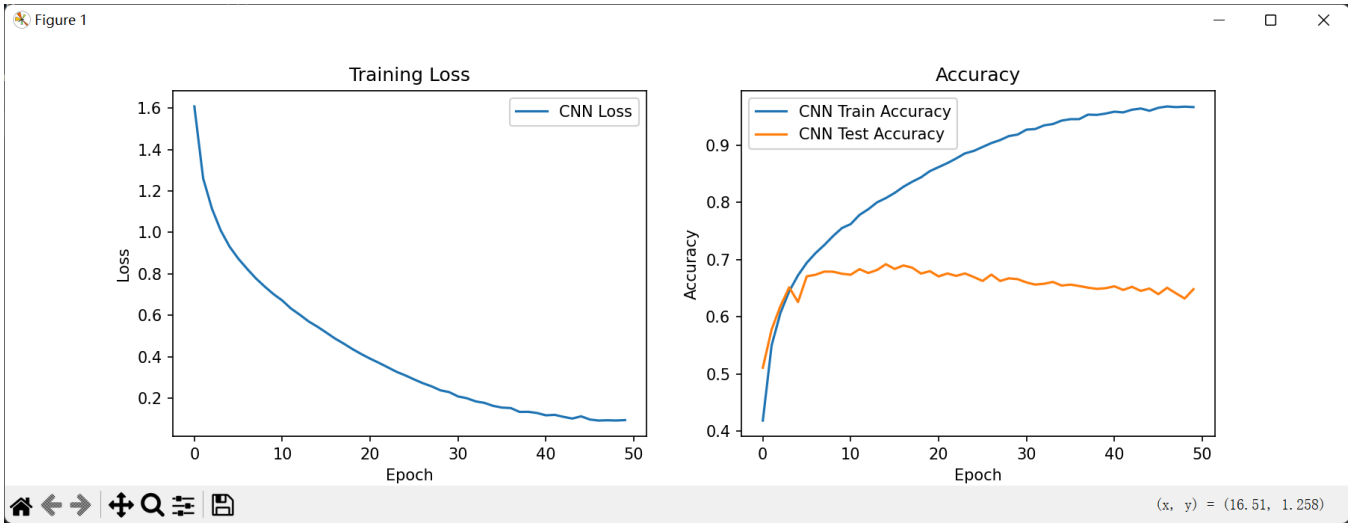



```
Epoch 39/50, Loss: 2.0619, Train Accuracy: 0.1829, Test Accuracy: 0.1530
Epoch 40/50, Loss: 2.0599, Train Accuracy: 0.1827, Test Accuracy: 0.1960
Epoch 41/50, Loss: 2.0607, Train Accuracy: 0.1829, Test Accuracy: 0.1830
Epoch 42/50, Loss: 2.0545, Train Accuracy: 0.1844, Test Accuracy: 0.1928
Epoch 43/50, Loss: 2.0624, Train Accuracy: 0.1837, Test Accuracy: 0.1900
Epoch 44/50, Loss: 2.0639, Train Accuracy: 0.1807, Test Accuracy: 0.1765
Epoch 45/50, Loss: 2.0625, Train Accuracy: 0.1827, Test Accuracy: 0.1768
Epoch 46/50, Loss: 2.0593, Train Accuracy: 0.1843, Test Accuracy: 0.1834
Epoch 47/50, Loss: 2.0579, Train Accuracy: 0.1814, Test Accuracy: 0.1672
Epoch 48/50, Loss: 2.0606, Train Accuracy: 0.1813, Test Accuracy: 0.1845
Epoch 49/50, Loss: 2.0537, Train Accuracy: 0.1825, Test Accuracy: 0.1821
Epoch 50/50, Loss: 2.0597, Train Accuracy: 0.1840, Test Accuracy: 0.1824
Training Time: 720.59 seconds
```

可以看出，神经元数量越多，训练时间越长，但是更大的隐藏层使网络的深度或宽度增加，梯度的传播更容易受到影响，可能出现梯度消失或梯度爆炸的问题，导致训练收敛困难。且较大的模型在小数据集或简单任务上更容易过拟合，而过拟合会导致训练准确率浮动，且泛化能力较差，测试准确率下降。

2.4 CNN

1.模型参数初始化：
卷积层和全连接层的参数由PyTorch默认方法初始化。而批归一化层(即nn.BatchNorm2d),其线性缩放参数初始化为1；偏置参数初始化为0。而其全连接层在第一次前向传播时，根据卷积层输出的形状动态计算输入维度并初始化。
2.超参数选择：
第一层卷积核数量：16，第二层：32。
激活函数：ReLU。
池化层：2×2，步长为 2。
3.优化函数用Adam，迭代次数为50。
4.学习率：0.01。
5.批大小为64，即每次训练64个样本。
运行结果为：



```
问题    输出    调试控制台    终端    端口
Epoch 39/50, Loss: 0.1348, Train Accuracy: 0.9528, Test Accuracy: 0.6488
Epoch 40/50, Loss: 0.1292, Train Accuracy: 0.9550, Test Accuracy: 0.6499
Epoch 41/50, Loss: 0.1180, Train Accuracy: 0.9583, Test Accuracy: 0.6533
Epoch 42/50, Loss: 0.1202, Train Accuracy: 0.9571, Test Accuracy: 0.6469
Epoch 43/50, Loss: 0.1106, Train Accuracy: 0.9617, Test Accuracy: 0.6524
Epoch 44/50, Loss: 0.1024, Train Accuracy: 0.9639, Test Accuracy: 0.6453
Epoch 45/50, Loss: 0.1133, Train Accuracy: 0.9599, Test Accuracy: 0.6494
Epoch 46/50, Loss: 0.0978, Train Accuracy: 0.9651, Test Accuracy: 0.6395
Epoch 47/50, Loss: 0.0928, Train Accuracy: 0.9675, Test Accuracy: 0.6507
Epoch 48/50, Loss: 0.0942, Train Accuracy: 0.9663, Test Accuracy: 0.6411
Epoch 49/50, Loss: 0.0931, Train Accuracy: 0.9671, Test Accuracy: 0.6319
Epoch 50/50, Loss: 0.0950, Train Accuracy: 0.9663, Test Accuracy: 0.6483
Training Time: 557.51 seconds
```

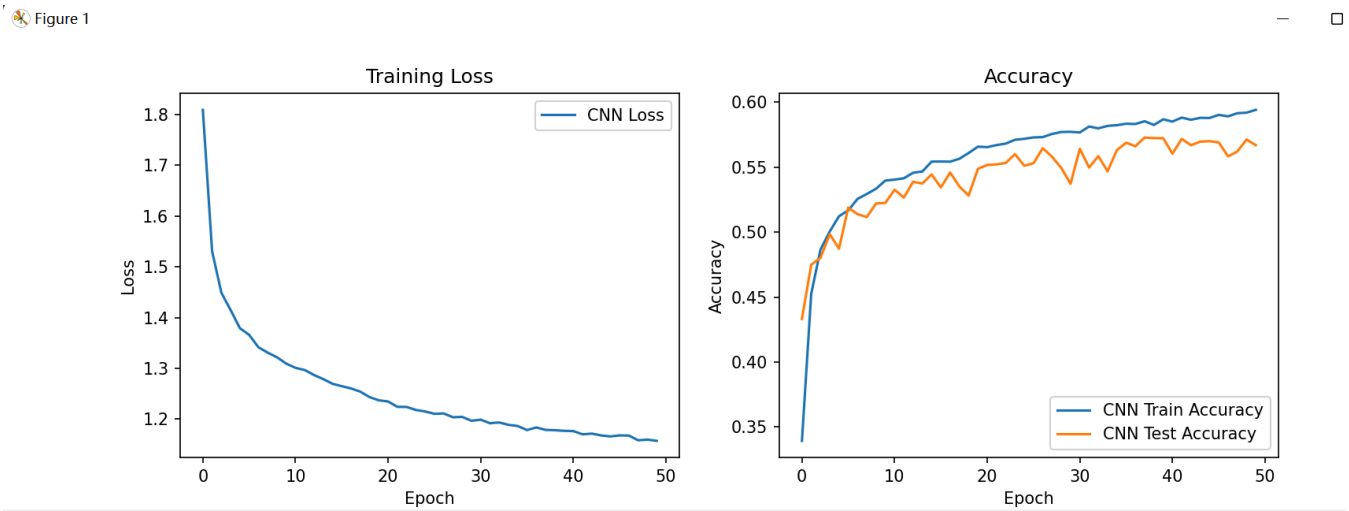
2.4.1 卷积层数对模型性能的影响

增加到3层卷积层

代码如下：

```
self.conv_layers = nn.Sequential(
            nn.Conv2d(3, 16, kernel_size=3, stride=1, padding=1), # Conv1
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),                # Pool1
            nn.Conv2d(16, 32, kernel_size=3, stride=1, padding=1),# Conv2
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),                # Pool2
            #新增一层
            # #------------------
            nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1), # Conv3
            nn.ReLU()
            #------------------------
        )
```

运行结果如下：




```
Epoch 39/50, Loss: 1.1783, Train Accuracy: 0.5825, Test Accuracy: 0.5723
Epoch 40/50, Loss: 1.1770, Train Accuracy: 0.5868, Test Accuracy: 0.5723
Epoch 41/50, Loss: 1.1765, Train Accuracy: 0.5851, Test Accuracy: 0.5604
Epoch 42/50, Loss: 1.1702, Train Accuracy: 0.5881, Test Accuracy: 0.5717
Epoch 43/50, Loss: 1.1716, Train Accuracy: 0.5865, Test Accuracy: 0.5669
Epoch 44/50, Loss: 1.1680, Train Accuracy: 0.5879, Test Accuracy: 0.5696
Epoch 45/50, Loss: 1.1661, Train Accuracy: 0.5878, Test Accuracy: 0.5700
Epoch 46/50, Loss: 1.1680, Train Accuracy: 0.5902, Test Accuracy: 0.5690
Epoch 47/50, Loss: 1.1677, Train Accuracy: 0.5891, Test Accuracy: 0.5583
Epoch 48/50, Loss: 1.1584, Train Accuracy: 0.5915, Test Accuracy: 0.5620
Epoch 49/50, Loss: 1.1598, Train Accuracy: 0.5919, Test Accuracy: 0.5712
Epoch 50/50, Loss: 1.1573, Train Accuracy: 0.5941, Test Accuracy: 0.5669
Training Time: 930.30 seconds
```

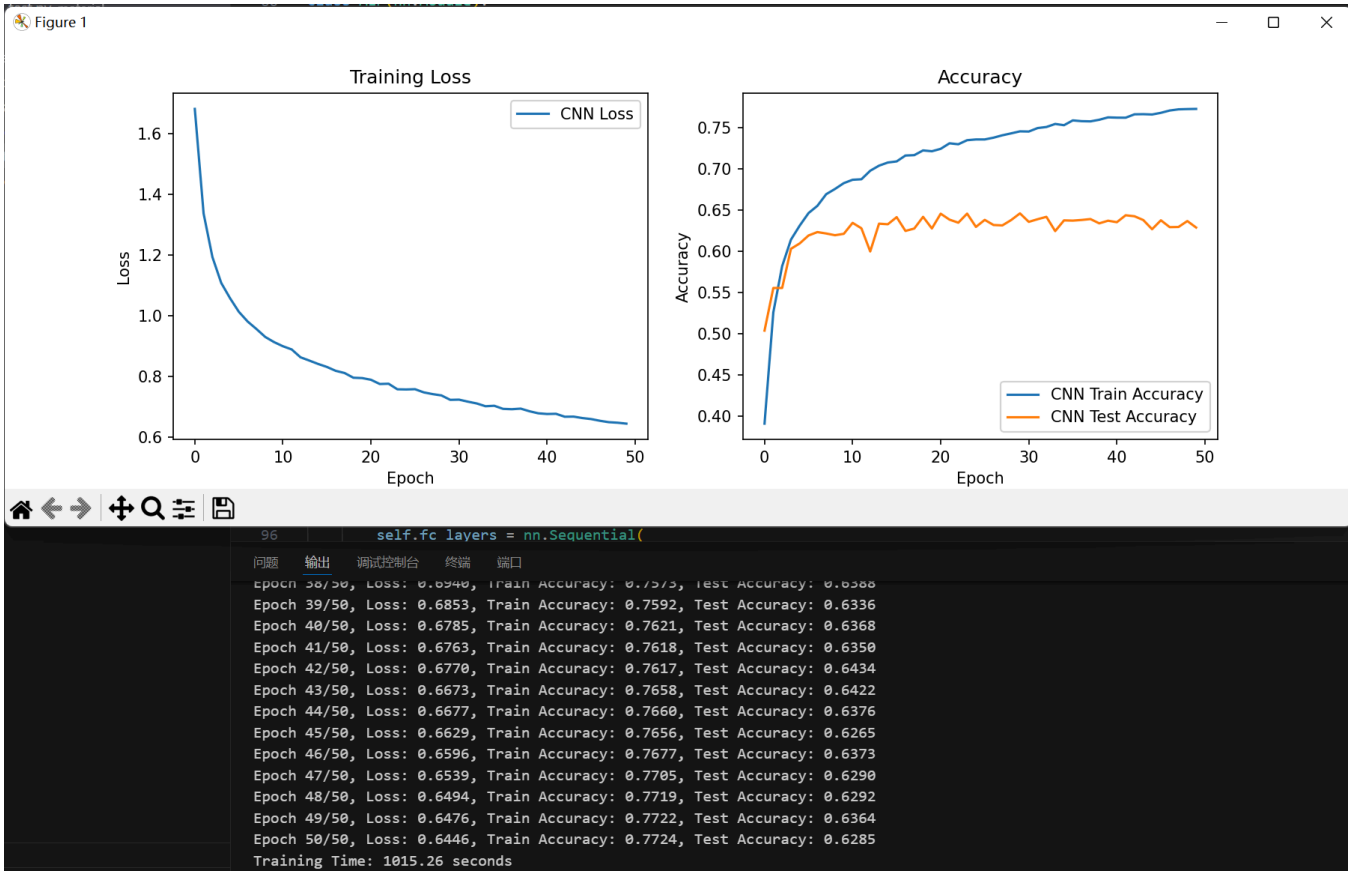
可以发现增加加了一层卷积层后训练时间变长，但拟合得更好了，其说明增加层数可以让模型有更高的容量去拟合训练数据，只是其准确率相比两层卷积层的有所下降，可能是因为三层模型需要更多训练轮次来充分学习，而与两层模型使用相同的迭代次数可能导致其尚未收敛，表现低于两层模型。

2.4.2 滤波器数量对模型性能的影响

在3层卷积层的基础上增加滤波器数量：

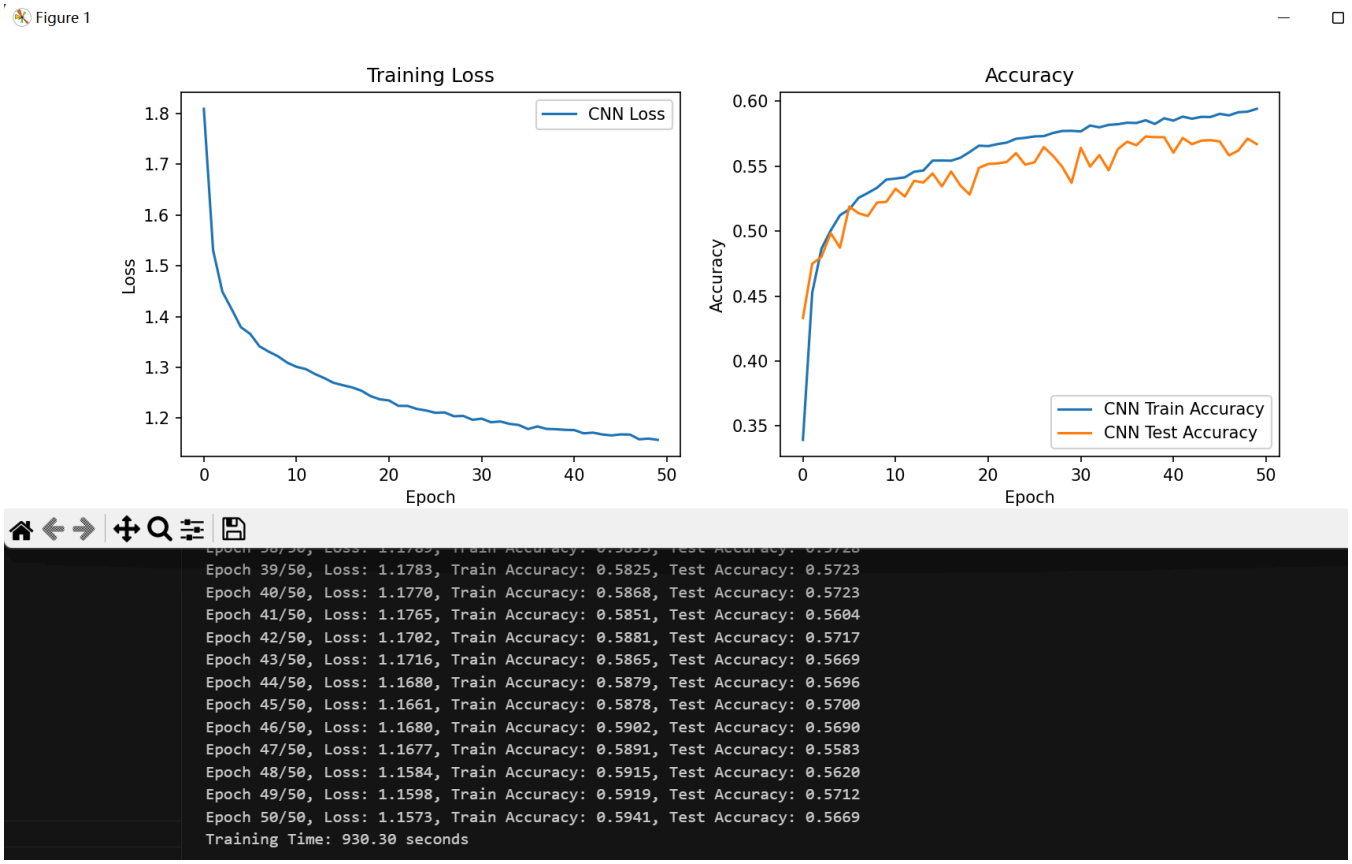
```python
class CNN(nn.Module):
    def __init__(self, num_classes):
        super(CNN, self).__init__()
        self.conv_layers = nn.Sequential(
            nn.Conv2d(3, 32, kernel_size=3, stride=1, padding=1), # Conv1
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),                # Pool1
            nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1),# Conv2
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),                # Pool2
            #新增一层
            #------------------
            nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=1), # Conv3
            nn.ReLU()
            #------------------------
        )
        self.fc_input_dim = None
        self.fc_layers = None
        self.num_classes = num_classes
```

运行结果如下：




```
96          self.fc_layers = nn.Sequential(
问题  输出  调试控制台  终端  端口
Epoch 38/50, Loss: 0.6940, Train Accuracy: 0.7573, Test Accuracy: 0.6388
Epoch 39/50, Loss: 0.6853, Train Accuracy: 0.7592, Test Accuracy: 0.6336
Epoch 40/50, Loss: 0.6785, Train Accuracy: 0.7621, Test Accuracy: 0.6368
Epoch 41/50, Loss: 0.6763, Train Accuracy: 0.7618, Test Accuracy: 0.6350
Epoch 42/50, Loss: 0.6770, Train Accuracy: 0.7617, Test Accuracy: 0.6434
Epoch 43/50, Loss: 0.6673, Train Accuracy: 0.7658, Test Accuracy: 0.6422
Epoch 44/50, Loss: 0.6677, Train Accuracy: 0.7660, Test Accuracy: 0.6376
Epoch 45/50, Loss: 0.6629, Train Accuracy: 0.7656, Test Accuracy: 0.6265
Epoch 46/50, Loss: 0.6596, Train Accuracy: 0.7677, Test Accuracy: 0.6373
Epoch 47/50, Loss: 0.6539, Train Accuracy: 0.7705, Test Accuracy: 0.6290
Epoch 48/50, Loss: 0.6494, Train Accuracy: 0.7719, Test Accuracy: 0.6292
Epoch 49/50, Loss: 0.6476, Train Accuracy: 0.7722, Test Accuracy: 0.6364
Epoch 50/50, Loss: 0.6446, Train Accuracy: 0.7724, Test Accuracy: 0.6285
Training Time: 1015.26 seconds
```

未增加滤波器时：



相比之下可以看出增加滤波器后训练时间变长，因为更多滤波器意味着更多的矩阵运算，增加了每次前向传播和反向传播的计算量，从而延长了训练时间，但可能是因为训练数据量不足，故滤波器数量增加导致了模型过拟合。

2.5 分别使用 SGD 算法、SGD Momentum 算法和 Adam 算法训练模型

因为同时跑3个模型耗时过长，故统一用CNN模型来算

CNN模型代码如下：

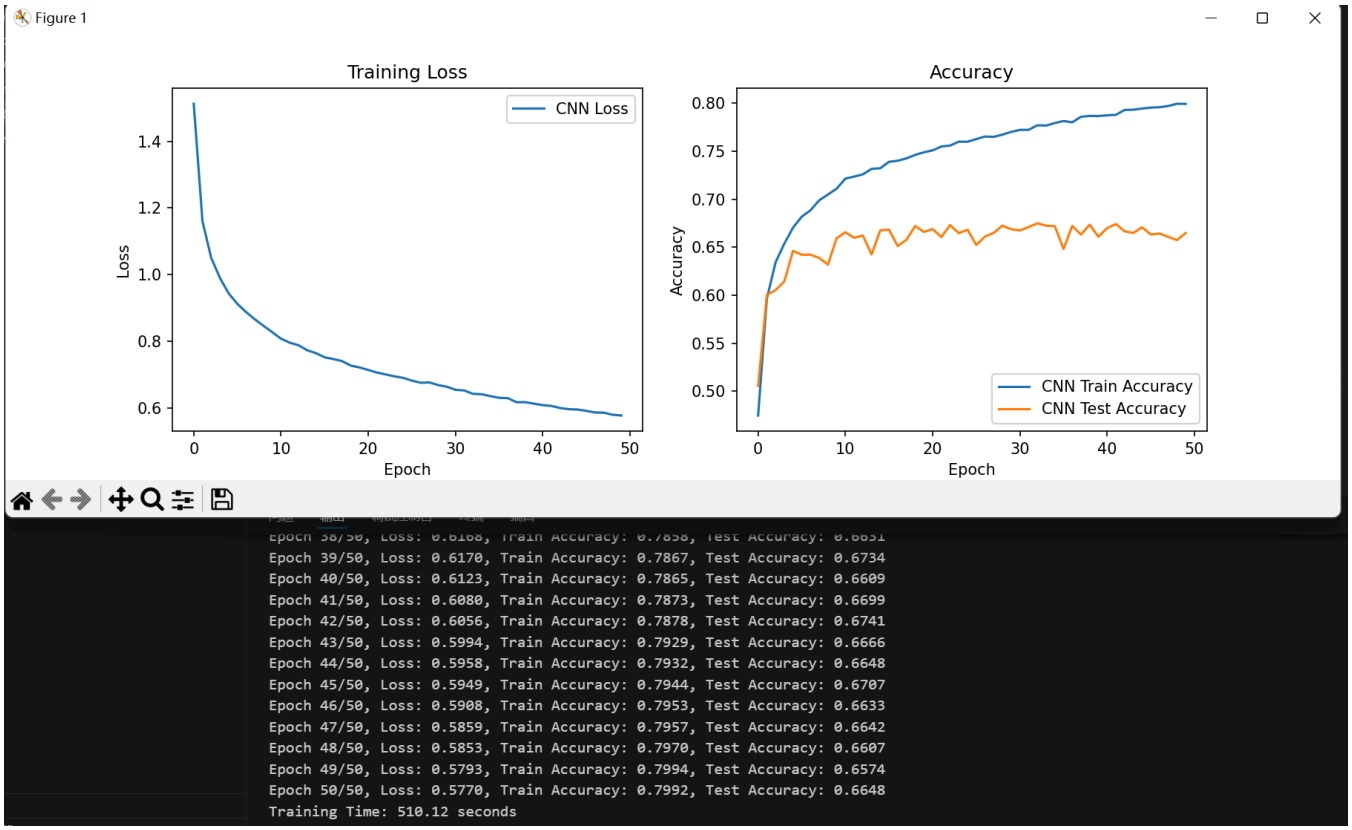
```python
class CNN(nn.Module):
    def __init__(self, num_classes):
        super(CNN, self).__init__()
        self.conv_layers = nn.Sequential(
            nn.Conv2d(3, 16, kernel_size=3, stride=1, padding=1), # Conv1
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),                # Pool1
            nn.Conv2d(16, 32, kernel_size=3, stride=1, padding=1),# Conv2
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),                # Pool2
            #新增一层
            # #------------------
            nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1), # Conv3
            #在每个卷积层后添加批归一化
            nn.BatchNorm2d(64),
            #-----
            nn.ReLU()
            #------------------------
        )
        self.fc_input_dim = None
        self.fc_layers = None
```

不同算法代码：

```
optimizers = {
    "SGD": lambda model: optim.SGD(model.parameters(), lr=0.01),
    "SGD_Momentum": lambda model: optim.SGD(model.parameters(), lr=0.01, momentum=0.9),
    "Adam": lambda model: optim.Adam(model.parameters(), lr=0.01)
}
```

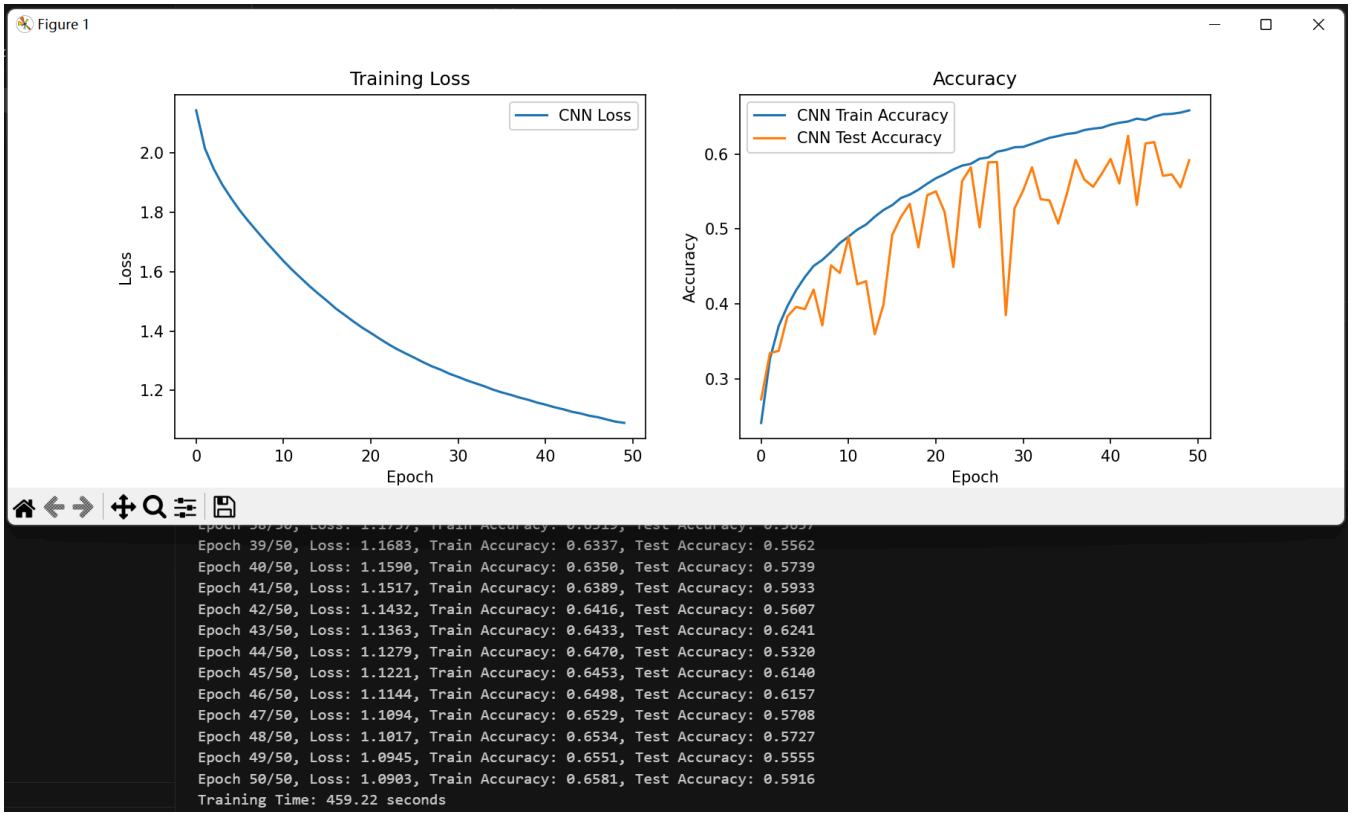
Adam算法：

Adam使用了梯度的一阶矩（均值）和二阶矩（方差）的估计，以自适应地调整每个参数的学习率。Adam 会根据每个参数的梯度历史和梯度平方的历史来调整每个参数的更新步长，使得优化过程更加稳定和高效。



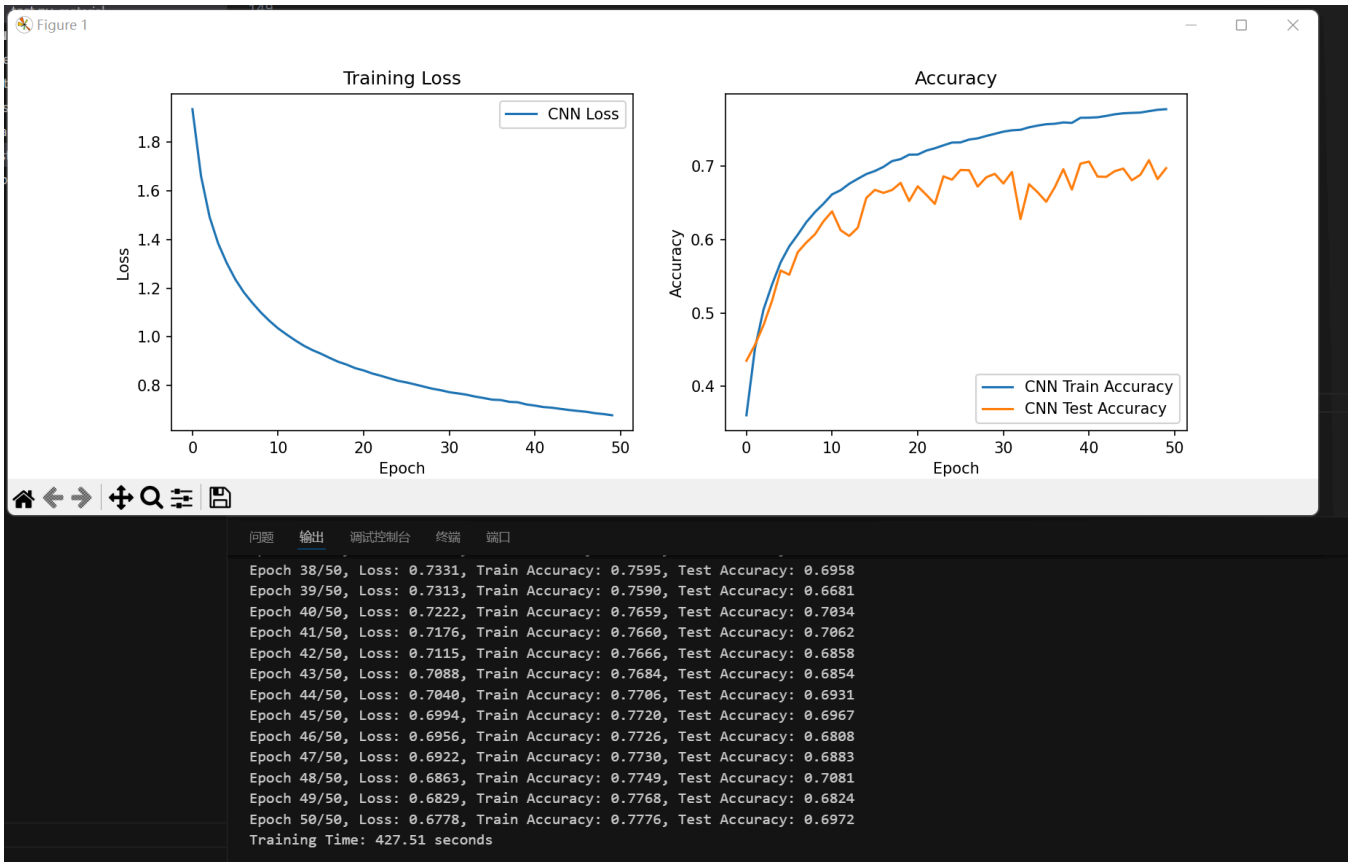
SGD算法：

SGD 是最基本的梯度下降方法，它通过对损失函数的梯度进行估计，来更新模型参数。每次更新时，参数的更新方向与梯度方向一致，并且步长由学习率控制。



SGD Momentum算法：

SGD Momentum 在传统的 SGD 基础上加入了“动量”概念，类似于物理中的惯性。动量是指在梯度更新中考虑前一次更新的方向和幅度，以此来加速参数的更新，减少震荡，尤其是在复杂的损失函数空间中。



对比之下可以看出，Adam算法训练时间最长，SGD Momentum算法训练时间最短，SGD Momentum算法运行出的准确率曲线拟合得比较好，对于训练准确率，SGD Momentum和Adam差不多，SGD训练准确率最低且不稳定。

总结：对于简单任务，可以使用SGD Momentum或SGD，以减少计算开销；对于更复杂的任务或参数空间较大的模型，推荐使用Adam，尽管它训练时间较长，但能更有效地优化复杂的网络结构。

2.6 训练和测试函数

代码如下：

```
def train_model(model, train_loader, test_loader, optimizer, criterion, num_epochs):
    train_losses, train_accuracies = [], []
    test_accuracies = []
    start_time = time.time()

    for epoch in range(num_epochs):
        model.train()
        total_loss, total_correct = 0, 0
        for X_batch, Y_batch in train_loader:
            optimizer.zero_grad()               # 梯度清零
            outputs = model(X_batch)            # 前向传播
            loss = criterion(outputs, Y_batch)  # 计算损失
            loss.backward()                     # 反向传播
            optimizer.step()                    # 更新参数

            total_loss += loss.item()           # 累计损失
            total_correct += (outputs.argmax(dim=1) == Y_batch).sum().item()     # 计算正确样本数

        train_losses.append(total_loss / len(train_loader))                      # 记录平均训练损失
        train_accuracies.append(total_correct / len(train_loader.dataset))       # 记录训练集准确率

        # 测试阶段
        model.eval()
        total_correct = 0
        with torch.no_grad():
            for X_batch, Y_batch in test_loader:
                outputs = model(X_batch)
                total_correct += (outputs.argmax(dim=1) == Y_batch).sum().item()
        test_accuracies.append(total_correct / len(test_loader.dataset))

        print(f"Epoch {epoch+1}/{num_epochs}, Loss: {train_losses[-1]:.4f}, "
              f"Train Accuracy: {train_accuracies[-1]:.4f}, Test Accuracy: {test_accuracies[-1]:.4f}")

    end_time = time.time()
    print(f"Training Time: {end_time - start_time:.2f} seconds")
    return train_losses, train_accuracies, test_accuracies
```

1.train_losses、train_accuracies 和 test_accuracies 用于存储每个epoch的训练损失、训练准确率和测试准确率。 start_time 用于记录训练开始的时间，方便后续计算训练的总时长。

2.对于训练集中的每个epoch，先用optimizer.zero_grad()清空之前的梯度。因为PyTorch的梯度计算是累加的，因此在每次更新之前，需要手动清零梯度。

3.前向传播：通过 model(X_batch) 获取模型的输出。

4.反向传播：通过 loss.backward()计算梯度。

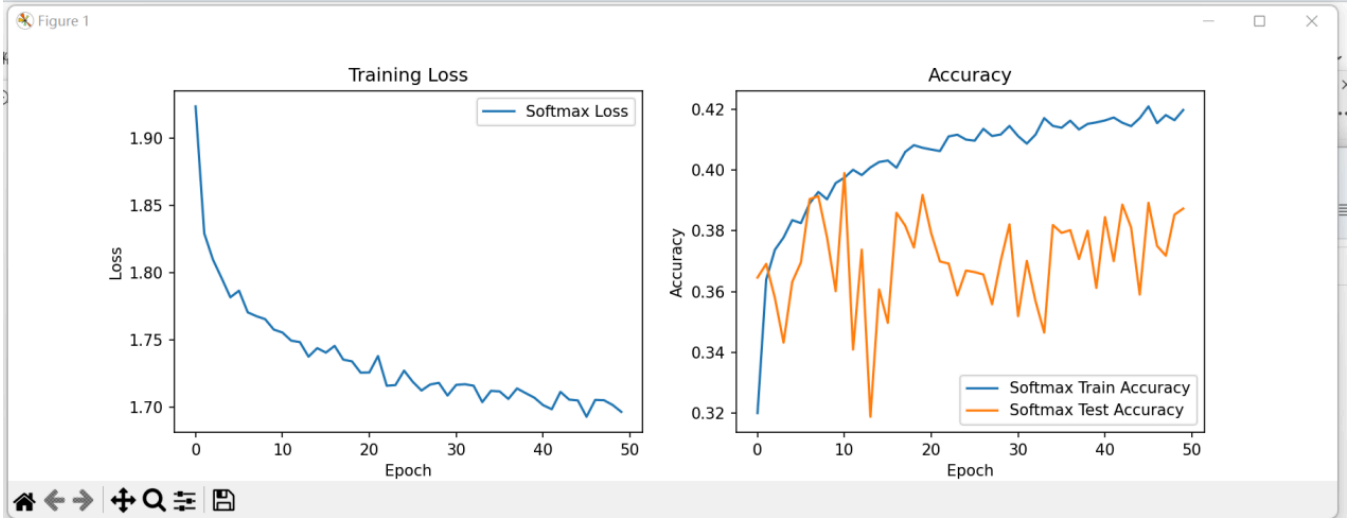
5.用损失函数计算模型输出和真实标签（Y_batch）之间的损失。在每完成一个batch后，计算并累加损失和准确预测的样本数。

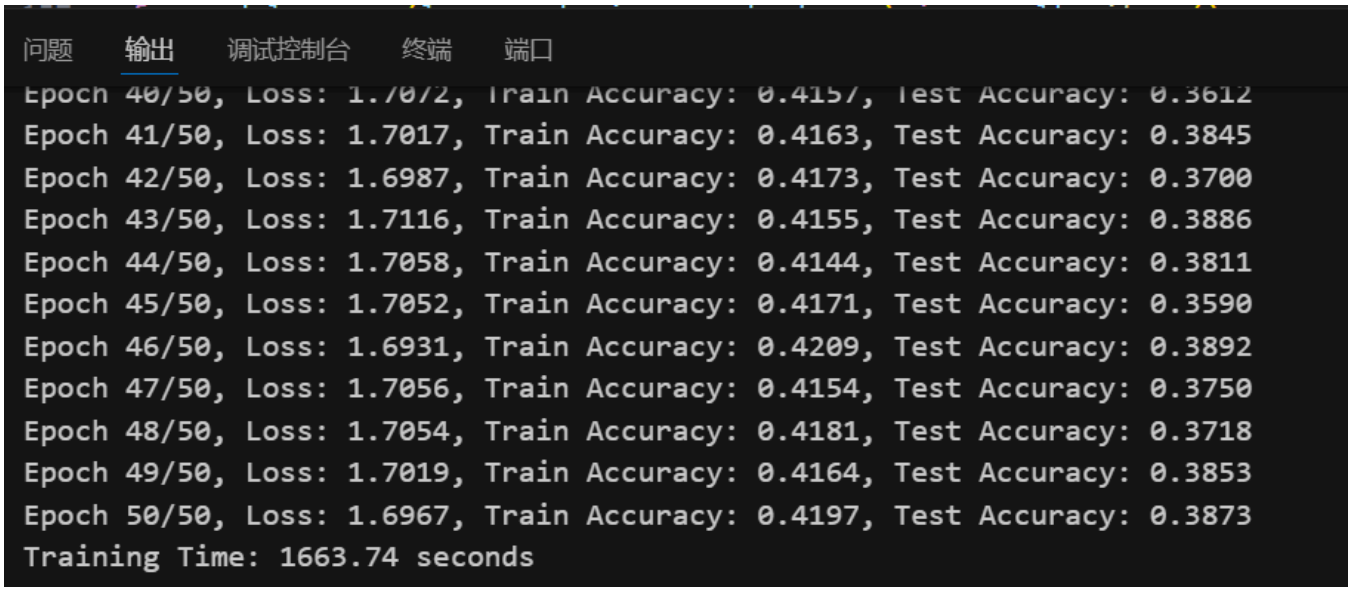
2.7 比较并讨论线性分类器、MLP 和 CNN 模型在 CIFAR-10 图像分类任务上的性能区别

统一学习率为0.01，迭代次数为50。
线性分类器：

```
class SoftmaxClassifier(nn.Module):
    def __init__(self, input_size, num_classes):
        super(SoftmaxClassifier, self).__init__()
        self.fc = nn.Linear(input_size, num_classes)

    def forward(self, x):
        x = x.view(x.size(0), -1)
        return self.fc(x)
...
Softmax": SoftmaxClassifier(3 * 32 * 32, 10)
```

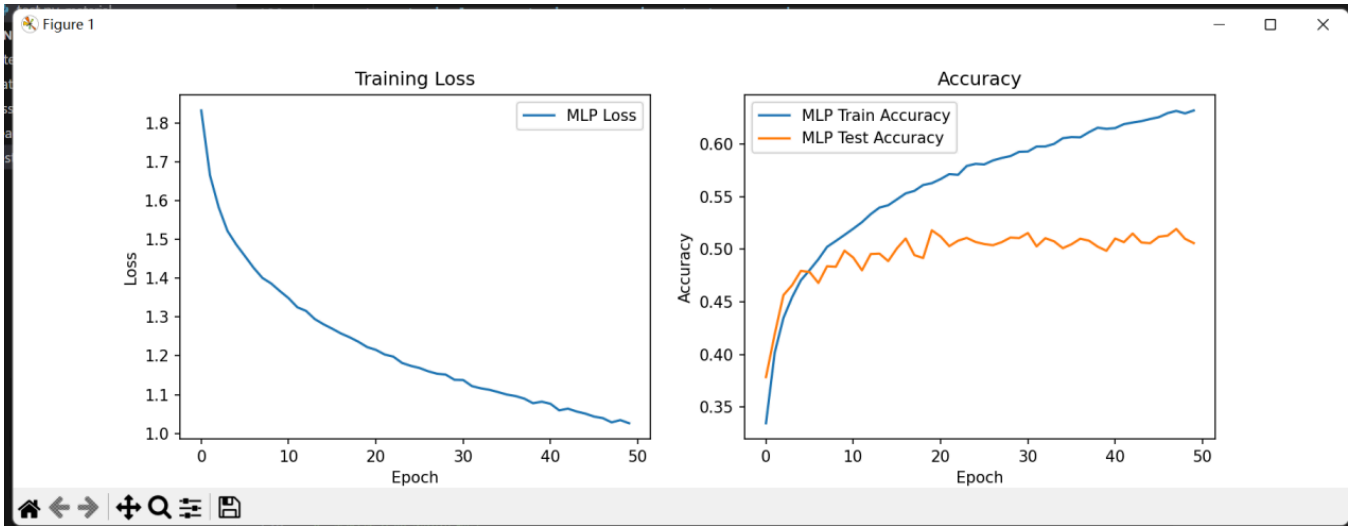




MLP：

```
class MLP(nn.Module):
    def __init__(self, input_size, hidden_sizes, num_classes):
        super(MLP, self).__init__()
        layers = []
        for i, hidden_size in enumerate(hidden_sizes):
            layers.append(nn.Linear(input_size if i == 0 else hidden_sizes[i-1], hidden_size))
            layers.append(nn.ReLU())
        layers.append(nn.Linear(hidden_sizes[-1], num_classes))
        self.net = nn.Sequential(*layers)

    def forward(self, x):
        x = x.view(x.size(0), -1)
        return self.net(x)
...
"MLP": MLP(3 * 32 * 32, [256,128], 10)
```



问题 输出 调试控制台 终端 端口

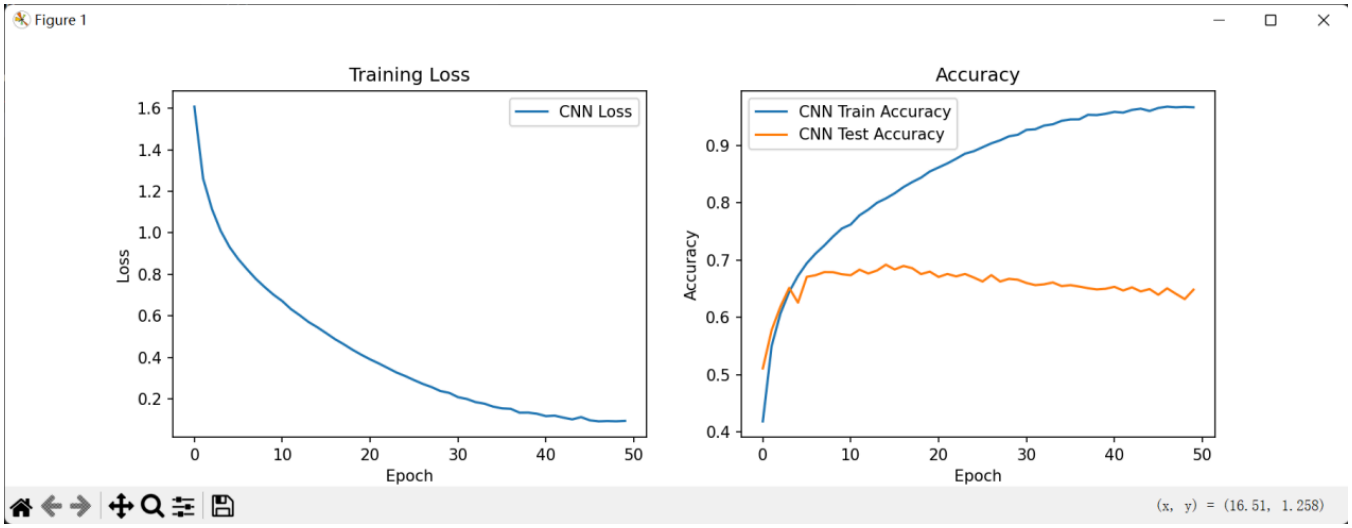
```
Epoch 41/50, Loss: 1.0763, Train Accuracy: 0.6154, Test Accuracy: 0.5103
Epoch 42/50, Loss: 1.0593, Train Accuracy: 0.6191, Test Accuracy: 0.5068
Epoch 43/50, Loss: 1.0638, Train Accuracy: 0.6206, Test Accuracy: 0.5150
Epoch 44/50, Loss: 1.0563, Train Accuracy: 0.6220, Test Accuracy: 0.5065
Epoch 45/50, Loss: 1.0508, Train Accuracy: 0.6240, Test Accuracy: 0.5058
Epoch 46/50, Loss: 1.0432, Train Accuracy: 0.6258, Test Accuracy: 0.5120
Epoch 47/50, Loss: 1.0393, Train Accuracy: 0.6296, Test Accuracy: 0.5130
Epoch 48/50, Loss: 1.0284, Train Accuracy: 0.6317, Test Accuracy: 0.5194
Epoch 49/50, Loss: 1.0343, Train Accuracy: 0.6294, Test Accuracy: 0.5100
Epoch 50/50, Loss: 1.0261, Train Accuracy: 0.6321, Test Accuracy: 0.5059
Training Time: 1319.85 seconds
```

CNN：

```
class CNN(nn.Module):
    def __init__(self, num_classes):
        super(CNN, self).__init__()
        self.conv_layers = nn.Sequential(
            nn.Conv2d(3, 16, kernel_size=3, stride=1, padding=1), # Conv1
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),                # Pool1
            nn.Conv2d(16, 32, kernel_size=3, stride=1, padding=1),# Conv2
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),                # Pool2
            #新增一层
            # #-----------------
            nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1), # Conv3
            #-----
            nn.ReLU()
            #------------------------
        )
        self.fc_input_dim = None
        self.fc_layers = None
        self.num_classes = num_classes

    def _initialize_fc_layers(self, x_shape):
        self.fc_input_dim = x_shape[1] * x_shape[2] * x_shape[3]
        self.fc_layers = nn.Sequential(
            nn.Linear(self.fc_input_dim, 128),
            nn.ReLU(),
            nn.Linear(128, self.num_classes)
        )

    def forward(self, x):
        x = self.conv_layers(x)
        if self.fc_layers is None:
            self._initialize_fc_layers(x.shape)
        x = x.view(x.size(0), -1)
        return self.fc_layers(x)
...
"CNN": CNN(10)
```




```
问题  输出  调试控制台  终端  端口
Epoch 39/50, Loss: 0.1348, Train Accuracy: 0.9528, Test Accuracy: 0.6488
Epoch 40/50, Loss: 0.1292, Train Accuracy: 0.9550, Test Accuracy: 0.6499
Epoch 41/50, Loss: 0.1180, Train Accuracy: 0.9583, Test Accuracy: 0.6533
Epoch 42/50, Loss: 0.1202, Train Accuracy: 0.9571, Test Accuracy: 0.6469
Epoch 43/50, Loss: 0.1106, Train Accuracy: 0.9617, Test Accuracy: 0.6524
Epoch 44/50, Loss: 0.1024, Train Accuracy: 0.9639, Test Accuracy: 0.6453
Epoch 45/50, Loss: 0.1133, Train Accuracy: 0.9599, Test Accuracy: 0.6494
Epoch 46/50, Loss: 0.0978, Train Accuracy: 0.9651, Test Accuracy: 0.6395
Epoch 47/50, Loss: 0.0928, Train Accuracy: 0.9675, Test Accuracy: 0.6507
Epoch 48/50, Loss: 0.0942, Train Accuracy: 0.9663, Test Accuracy: 0.6411
Epoch 49/50, Loss: 0.0931, Train Accuracy: 0.9671, Test Accuracy: 0.6319
Epoch 50/50, Loss: 0.0950, Train Accuracy: 0.9663, Test Accuracy: 0.6483
Training Time: 557.51 seconds
```

对比之后可以看出在CIFAR-10图像分类任务上,线性分类器的训练时间最长，CNN模型训练时间最短，CNN模型准确率最高，线性分类器准确率最低。

线性分类器: 线性分类器的训练时间通常较长。尽管模型本身较为简单，但对于CIFAR-10数据集这种高维数据，线性分类器仍需要对大量的特征进行处理。而由于其简单性，线性分类器的准确率通常较低。

MLP： MLP通常比线性分类器要快一些；然而，MLP训练的时间依然可能受到数据量和网络规模的影响，特别是在图像数据上，展平操作导致输入维度很高，计算复杂度增加。相较于线性分类器，MLP虽然能够取得更好的分类性能，但仍然存在一些局限性，尤其是在处理图像数据时，由于其仅依赖于全连接层，它无法像卷积神经网络那样有效地捕捉图像中的局部空间结构。因此，它的准确率通常会比CNN低。

CNN： CNN的训练时间通常较长，但由于CNN能够有效地提取和共享特征，其训练过程中参数的更新通常比MLP更高效。在CIFAR-10这样的图像分类任务中，CNN的表现是最好的。因为CNN能够有效地捕捉到图像中的空间特征，从而在图像分类任务上取得较高的准确率。