中山大学计算机学院

人工智能

本科生实验报告

（2023 学年春季学期）

课程名称：Artificial Intelligence

教学班级	**刘咏梅老师班级**	专业（方向）	**信息与计算科学**
学号	**22336313**	姓名	**郑鸿鑫**

一、 实验题目

利用 pytorch 框架搭建 CNN 神经网络实现中药图片分类

二、 实验内容

1. 算法原理

本实验的算法原理基于卷积神经网络（Convolutional Neural Network, CNN），这是一种深度学习架构，常用于图像分类、图像识别、视频分析和自然语言处理等任务。以下是 CNN 的一些关键原理：

- 卷积层（Convolutional Layer）：卷积层是 CNN 的核心，它使用一组可学习的过滤器（或称为卷积核）来提取图像中的局部特征。每个卷积核在输入图像上滑动（通过覆盖图像的局部区域），计算卷积核和覆盖区域的点积，生成特征图（feature map），这些特征图捕捉了图像中的局部特征。
- 激活函数（Activation Function）：通常在卷积层之后使用非线性激活函数，如 ReLU（Rectified Linear Unit），增加模型的非线性表达能力，帮助网络学习更复杂的特征。
- 池化层（Pooling Layer）：池化层用于降低特征图的空间尺寸，从而

减少参数数量和计算量，同时使特征检测更加鲁棒。最常见的池化操作是最大池化（Max Pooling），它提取区域内的最大值。

- 全连接层（Fully Connected Layer）：在多个卷积和池化层之后，CNN 通常包含一个或多个全连接层，其中节点与前一层的所有激活值相连。全连接层负责整合从卷积层提取的特征，进行最终的分类或回归任务。
- 损失函数（Loss Function）：损失函数衡量模型的预测输出与真实标签之间的差异。对于分类问题，交叉熵损失（Cross-Entropy Loss）是常用的损失函数。优化器（Optimizer）：优化器如 SGD（随机梯度下降）或 Adam 用于根据损失函数的结果更新网络的权重，目的是最小化损失函数。
- 反向传播（Backpropagation）：训练过程中，反向传播算法用于计算损失相对于每个参数的梯度，这些梯度指示了如何调整参数以减少损失。

2. 伪代码

```
//数据预处理
Procedure preprocess_data()
    Begin
        transform = Compose([
            Resize((224, 224)),
            ToTensor(),
            Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
        ])
    EndProcedure
//加载数据集
Procedure load_data(train_data_dir, test_data_dir)
    Begin
        train_dataset = ImageFolder(root=train_data_dir, transform=transform)
        test_dataset = ImageFolder(root=test_data_dir, transform=transform)
    EndProcedure
Procedure create_data_loaders(train_dataset, test_dataset)
    Begin
```

```
        train_loader = DataLoader(dataset=train_dataset, batch_size=32, shuffle=True)
        test_loader = DataLoader(dataset=test_dataset, batch_size=32, shuffle=False)
    EndProcedure
//建立卷积神经网络模型
Class SimpleCNN Inherits nn.Module
    Procedure __init__(num_classes)
        Begin
            Initialize conv1, conv2 as Sequential modules with Conv2d, ReLU, MaxPool2d
            Initialize fc1, dropout, fc2 layers
        EndProcedure
    Procedure forward(x)
        Begin
            x = conv1(x)
            x = conv2(x)
            x = Flatten(x)
            x = fc1(x)
            x = dropout(x)
            x = fc2(x)
            Return x
        EndProcedure
EndClass
//计算准确率的函数
Procedure calculate_accuracy(model, test_loader)
    Begin
        model.eval()
        correct = 0
        total = 0
        For Each (images, labels) In test_loader
            outputs = model(images)
            predicted = outputs.max(1)[1]
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
        EndFor
        Return (correct / total) * 100
    EndProcedure
//训练模型
Procedure train_model(num_epochs, model, train_loader, test_loader)
    Begin
        Initialize criterion As CrossEntropyLoss
        Initialize optimizer As Adam(model.parameters(), lr=0.001)
        Initialize scheduler As StepLR(optimizer, step_size=7, gamma=0.1)
        For epoch From 1 To num_epochs
            model.train()
            running_loss = 0.0
            For Each (images, labels) In train_loader
                optimizer.zero_grad()
                outputs = model(images)
                loss = criterion(outputs, labels)
                loss.backward()
                optimizer.step()
                running_loss += loss.item()
            EndFor
            scheduler.step()
            epoch_loss = running_loss / len(train_loader)
            losses.append(epoch_loss)
            accuracy = calculate_accuracy(model, test_loader)
```

```
                accuracies.append(accuracy)
                Print Epoch, Loss, and Accuracy Information
            EndFor
        EndProcedure
    //绘制 loss 和准确率的曲线
    Procedure plot_training_curves(losses, accuracies)
        Begin
            Use matplotlib to plot losses and accuracies over epochs
        EndProcedure
    // 主程序
    Begin
        preprocess_data()
        load_data(train_data_dir, test_data_dir)
        create_data_loaders(train_dataset, test_dataset)
        num_classes = 5
        model = SimpleCNN(num_classes)
        device = Determine Device (GPU if available, else CPU)
        model.to(device)
        num_epochs = 10
        train_model(num_epochs, model, train_loader, test_loader)
        plot_training_curves(losses, accuracies)
    End
```

3. 关键代码展示

- **搭建简单的 CNN 模型：**

```python
class SimpleCNN(nn.Module):
    def __init__(self, num_classes=5):
        super(SimpleCNN, self).__init__()
        # 定义第一个卷积层，修改 in_channels 为 3 以处理 RGB 图片
        self.conv1 = nn.Sequential(
            nn.Conv2d(3, 16, kernel_size=5, stride=1, padding=2),
            nn.ReLU(),#激活函数，在大于 0 时 y = x,小于 0 时 y = 0
            nn.MaxPool2d(kernel_size=2),#最大池化，大小为 2 的池化窗口
        )
        # 第二个卷积层
        self.conv2 = nn.Sequential(
            nn.Conv2d(16, 32, kernel_size=5, stride=1, padding=2),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2),
        )
        # 假设经过 conv1 和 pool1 后，特征图变为 (224-5+2*2)/2+1 = 112
        # 经过 conv2 和 pool2 后，特征图变为 (112-5+2*2)/2+1 = 56
        # 因此，全连接层的输入特征数为 32*56*56
        self.fc1 = nn.Linear(32*56*56, 5)#32 个通道，56*56 的特征图
    def forward(self, x):
        x = self.conv1(x)
        x = self.conv2(x)
        # 展平操作，将多维张量变成一个二维张量
        x = x.view(x.size(0), -1)
        x = self.fc1(x)
        return x
```

- **计算在测试集上的准确率：**

```
def calculate_accuracy(model, test_loader):
    model.eval()#将模型设定为评估模式
    correct = 0
    total = 0
    with torch.no_grad():#无梯度计算
        for images, labels in test_loader:
            #将数据移动到 GPU 上
            images = images.to(device)
            labels = labels.to(device)
            #将图像经过模型前向传播，预测类别
            outputs = model(images)
            #在结果的每一行中选取最大值作为预测标签
            _, predicted = torch.max(outputs.data, 1)
            #这个张量的第一维度，指的是这批样本的大小
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
            #print("真实值：",labels,"预测值：",predicted)
    accuracy = 100 * correct / total
    return accuracy
```

- **数据预处理：**

```
 # 实例化模型
num_classes = 5  # 中药图片有 5 个类别
model = SimpleCNN(num_classes)
#确定使用的设备，并将模型移到设备上
device = torch.device("cuda:0" if torch.cuda.is_available() else
"cpu")
model.to(device)
# 定义数据预处理
transform = transforms.Compose([
    transforms.Resize((224, 224)),  # 调整图片大小为 224x224
    transforms.ToTensor(),  # 转换为 Tensor
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229,
0.224, 0.225])  # 归一化
])
# 假设训练数据和测试数据的文件夹路径分别为：
train_data_dir = 'C:\\Users\\26618\\Desktop\\人工智能实验\\实验
7\\cnn_data\\train'
test_data_dir = 'C:\\Users\\26618\\Desktop\\人工智能实验\\实验
7\\cnn_data\\test'

# 加载训练数据集，用 ImageFolder 打开文件夹中的图片
train_dataset = datasets.ImageFolder(root=train_data_dir,
transform=transform)
# 创建训练数据加载器
train_loader = DataLoader(dataset=train_dataset, batch_size=32,
shuffle=True)
# 加载测试数据集
test_dataset = datasets.ImageFolder(root=test_data_dir,
transform=transform)
```

```
# 创建测试数据加载器
test_loader = DataLoader(dataset=test_dataset, batch_size=10,
shuffle=False)
#shuffle 参数用于选择是否打乱图片的顺序，训练集打乱，测试集不打乱
```

- **训练模型：**

```
 # 定义损失函数和优化器（交叉熵损失）
criterion = nn.CrossEntropyLoss()
# 训练模型
num_epochs = 10#遍历 10 次训练集
losses, accuracies = [], []
#优化器，parameters 返回模型中所有的参数，用于梯度更新；lr 学习率
optimizer = optim.Adam(model.parameters(), lr=0.001)
#学习率调度器，每隔 7 代学习率减小到 10%
scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=7,
gamma=0.1)
for epoch in range(num_epochs):
    model.train()#设置为训练模式
    running_loss = 0.0
    for images, labels in train_loader:
        images = images.to(device)
        labels = labels.to(device)
        optimizer.zero_grad()#每次迭代前梯度清零
        outputs = model(images)#得出预测
        loss = criterion(outputs, labels)#计算损失
        loss.backward()#反向传播
        optimizer.step()#根据计算的梯度更新模型参数
        running_loss += loss.item()
    scheduler.step()#更新学习率
    epoch_loss = running_loss / len(train_loader)
    losses.append(epoch_loss)
    accuracy = calculate_accuracy(model, test_loader)
    accuracies.append(accuracy)
    print(f'Epoch [{epoch + 1}/{num_epochs}], Loss:
{epoch_loss:.4f}, Accuracy: {accuracy:.2f}%')
```

- **绘制 loss 和准确率曲线：**

```
 # 绘制训练损失和测试准确率曲线
plt.figure(figsize=(12, 4))
plt.subplot(1, 2, 1)
plt.plot(losses, label='Training Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.title('Training Loss Over Epochs')
plt.legend()
plt.subplot(1, 2, 2)
plt.plot(accuracies, label='Test Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy (%)')
plt.ylim(0,110)
plt.title('Test Accuracy Over Epochs')
plt.legend()
```

```
plt.tight_layout()
plt.show()
```

4. 创新点&优化

优化的具体形式有下列几种，结果分析详见评测指标展示及分析

1. 将卷积层的层数从 2 层增加到 3 层

2. 在进入全连接层前加上 droppout 层

3. 将优化器从 SGD 换为 Adam，可实现动态调整学习率

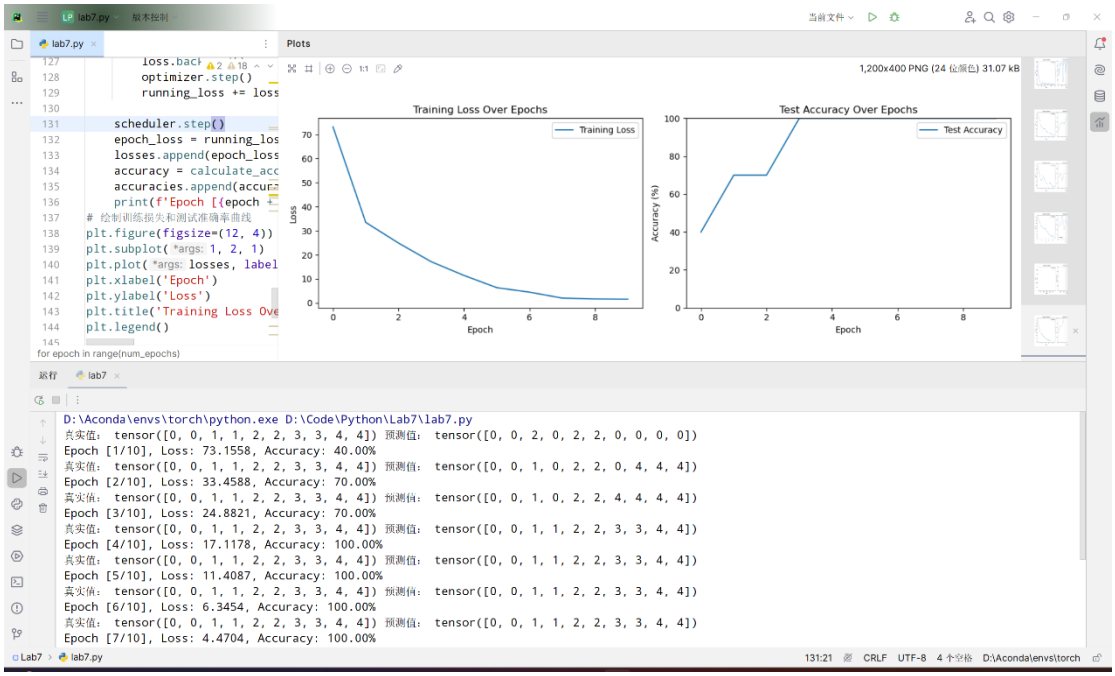
4. 对训练的数据做数据增强

5. 将全连接层从 1 层替换为 2 层

三、 实验结果及分析

1. 实验结果展示示例

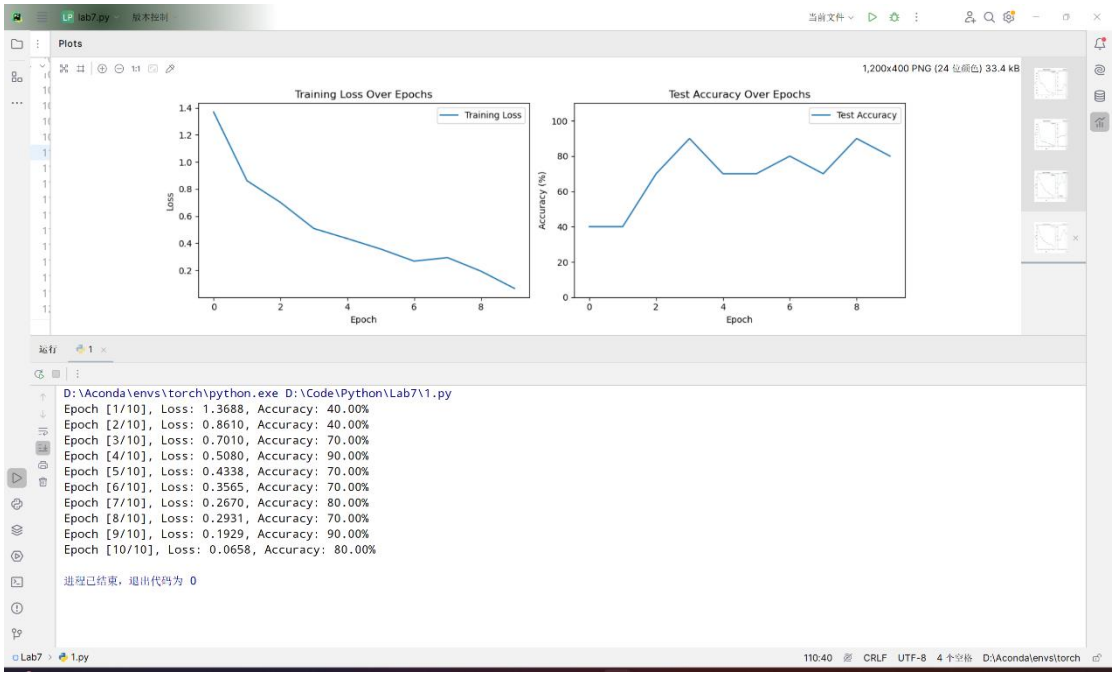
此处选取优化到最后的一次实验结果作为示例：



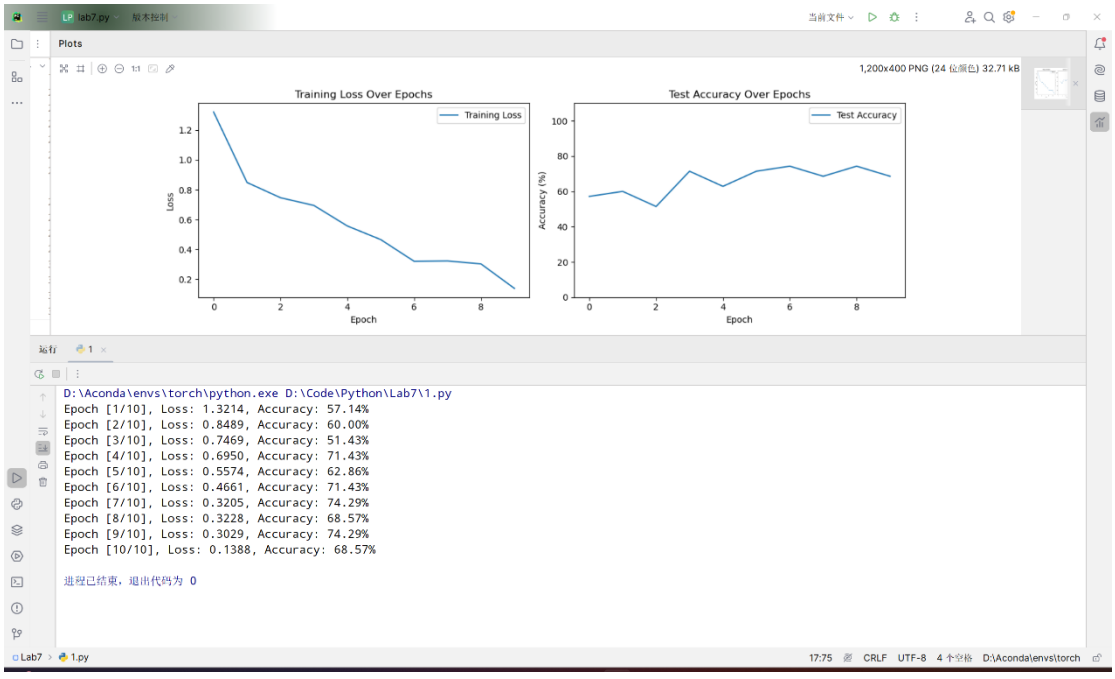
可以看到损失函数持续下降，对实验给定的测试集预测的准确率最终也可以保持在 100%。

2. 评测指标展示及分析

a. 两层卷积层和 SGD 优化器：



可以看见损失函数逐渐减少，预测的准确率虽然一开始是上升，但是后续会出现波动，怀疑是数据集过少，偶然性较大，故我们每个标签下增加 5 张图片作为测试。

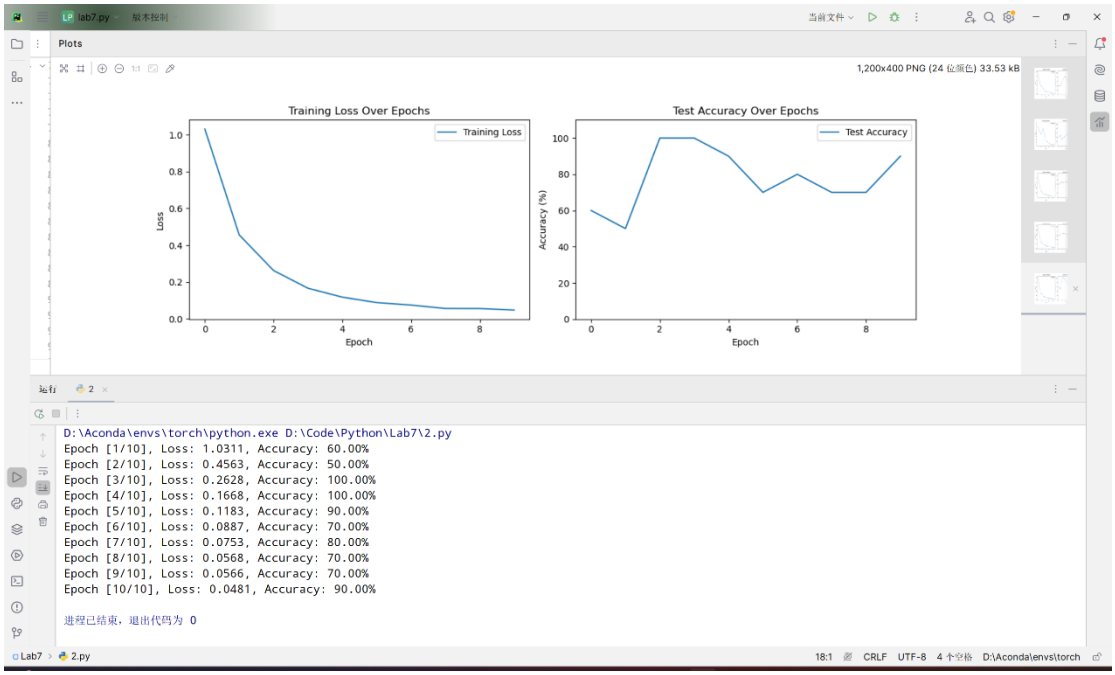


增大测试集后，结果依然不是很好，由于卷积层只有简单的两层，全连接层也只有一层，所以我们通过增加卷积层到 3 层来训练模型

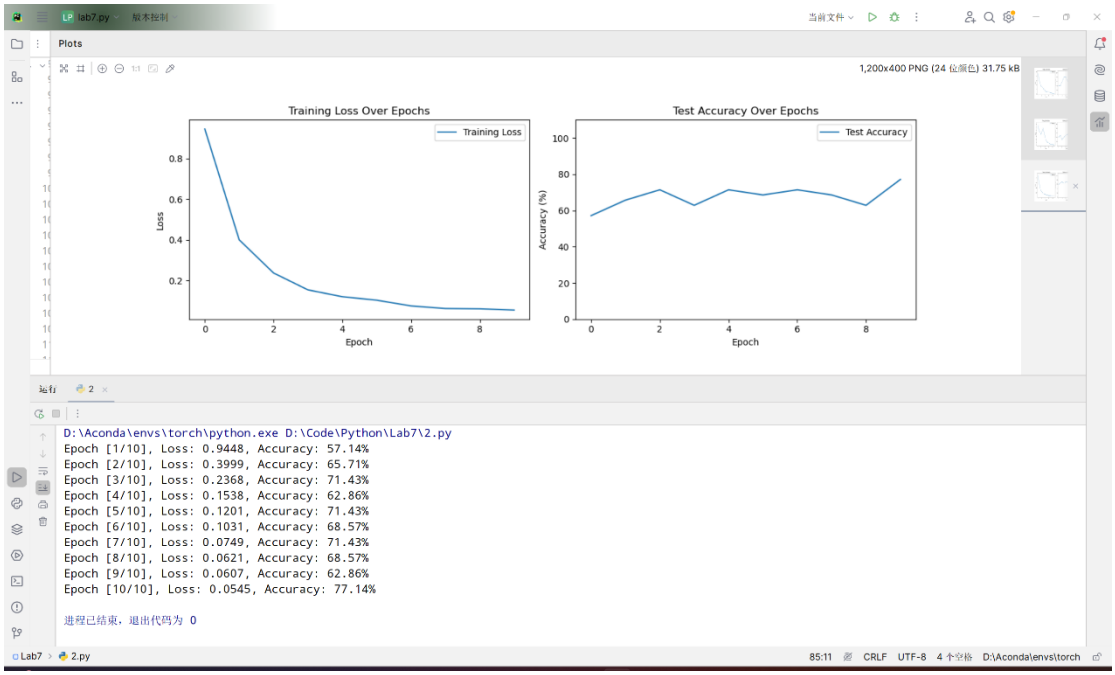
b. 三层卷积层与 SGD 优化器

将卷积层的层数增加为 3 层，代码如下：

```
class SimpleCNN(nn.Module):
    def __init__(self, num_classes=5):
        super(SimpleCNN, self).__init__()
        self.conv1 = nn.Sequential(
            nn.Conv2d(3, 16, kernel_size=5, stride=1, padding=2),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2),
            nn.BatchNorm2d(16),
        )
        self.conv2 = nn.Sequential(
            nn.Conv2d(16, 32, kernel_size=5, stride=1, padding=2),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2),
            nn.BatchNorm2d(32),
        )
        self.conv3 = nn.Sequential(
            nn.Conv2d(32, 64, kernel_size=5, stride=1, padding=2),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2),
            nn.BatchNorm2d(64),
        )
        self.fc = nn.Linear(64*28*28, num_classes)
    def forward(self, x):
        x = self.conv1(x)
        x = self.conv2(x)
        x = self.conv3(x)
        x = x.view(x.size(0), -1)
        x = self.fc(x)
        return x
```



遗憾的是增加层数后，准确率依然在损失函数下降的情况下准确率会出现较大波动，甚至损失函数也会出现波动。



在训练集增加到 35 张后得到的准确率与两层接近，且训练过程中仍然存在波动，经过分析我们认为在此实验中，增加卷积层的数目以增加模型的复杂度，可能会导致模型学习到训练数据中的噪声，而不是

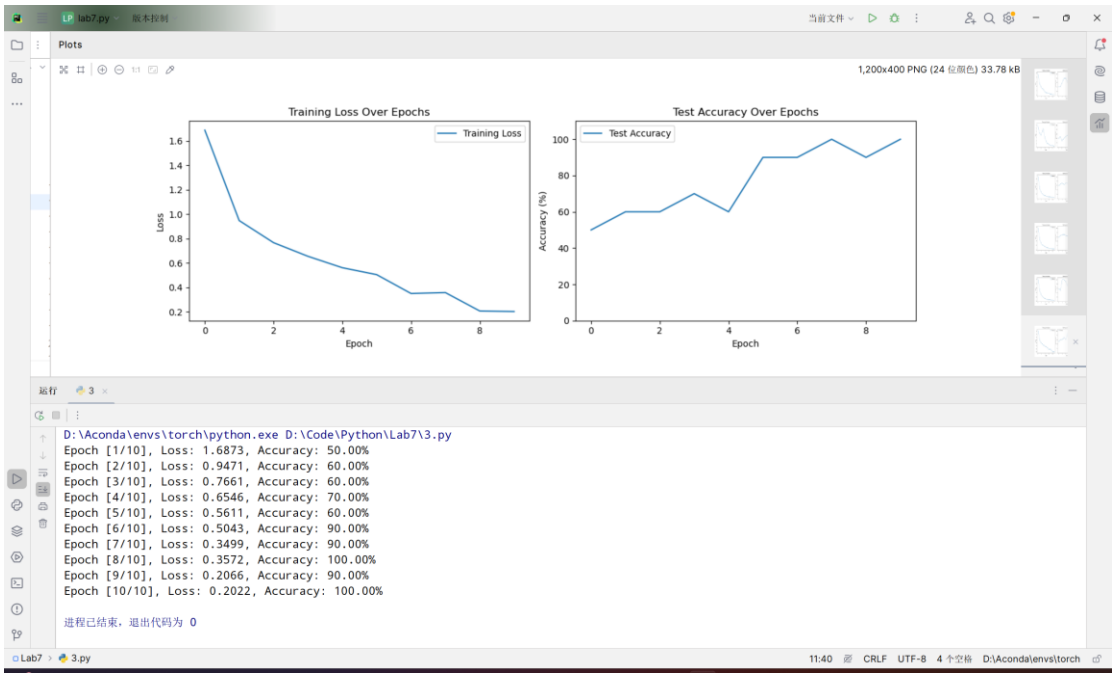
泛化的特征，所以后续的优化选取在两层卷积层的基础上进行优化。

c. **两层卷积层添加 droppout 层**

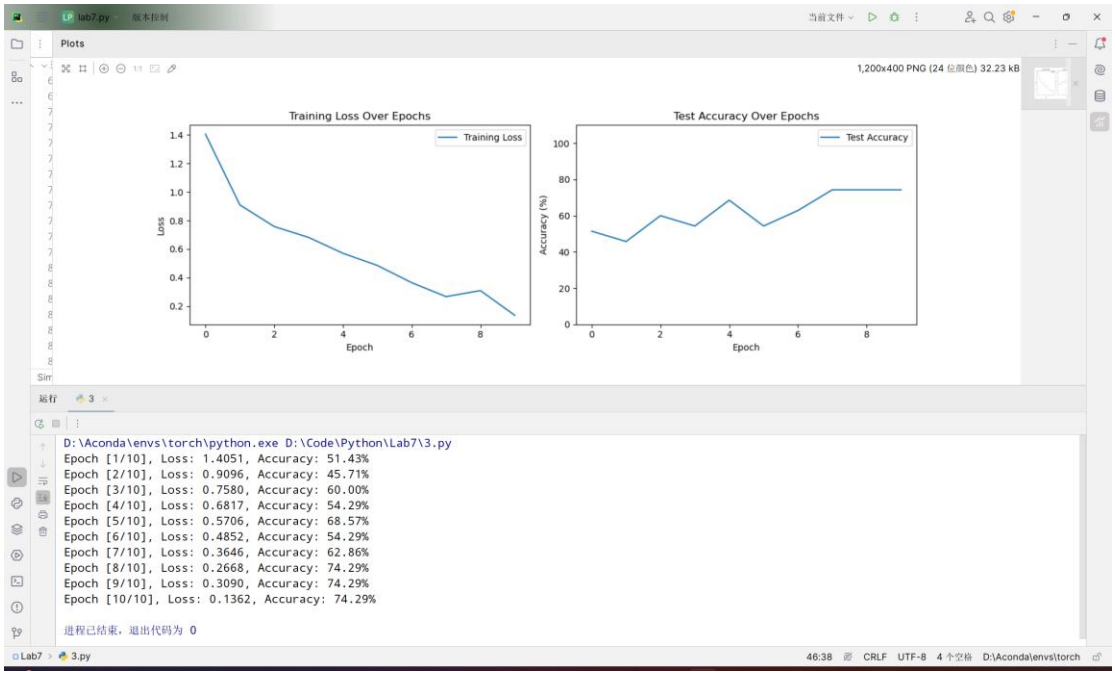
添加 droppout 层并在进入全连接层之前调用来防止过拟合：

```
self.dropout = nn.Dropout(0.5)
def forward(self, x):
    x = self.conv1(x)
    x = self.conv2(x)
    x = x.view(x.size(0), -1)
    x = self.dropout(x)
    x = self.fc(x)
    return x
```

在 10 张图片的测试集上测试：



最后的准确率已经较为理想，但还是出现轻微波动，应该是与训练集太少有关，再在 35 张图片的训练集上加以测试：



准确率收敛到了 75%左右，于是我们尝试从另一个角度改进，采用能自适应调整学习率的优化器 Adam。

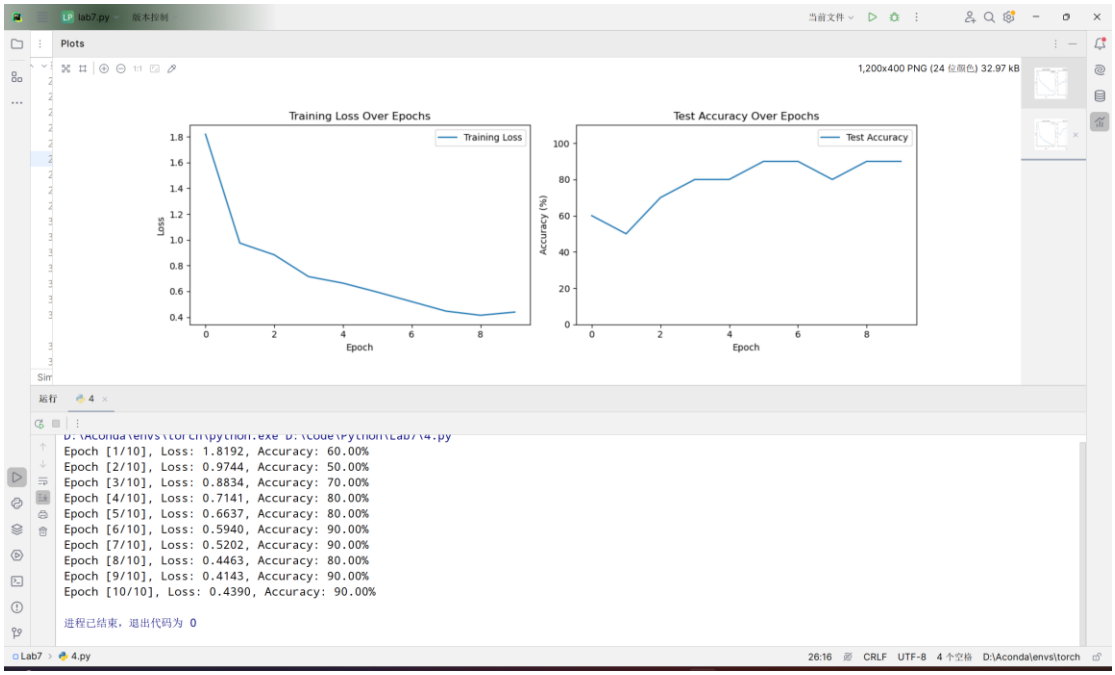
d. 将 SGD 优化器更换为 Adam 优化器：

```
lr = 0.01
optimizer = optim.SGD(model.parameters(),lr=lr)
```

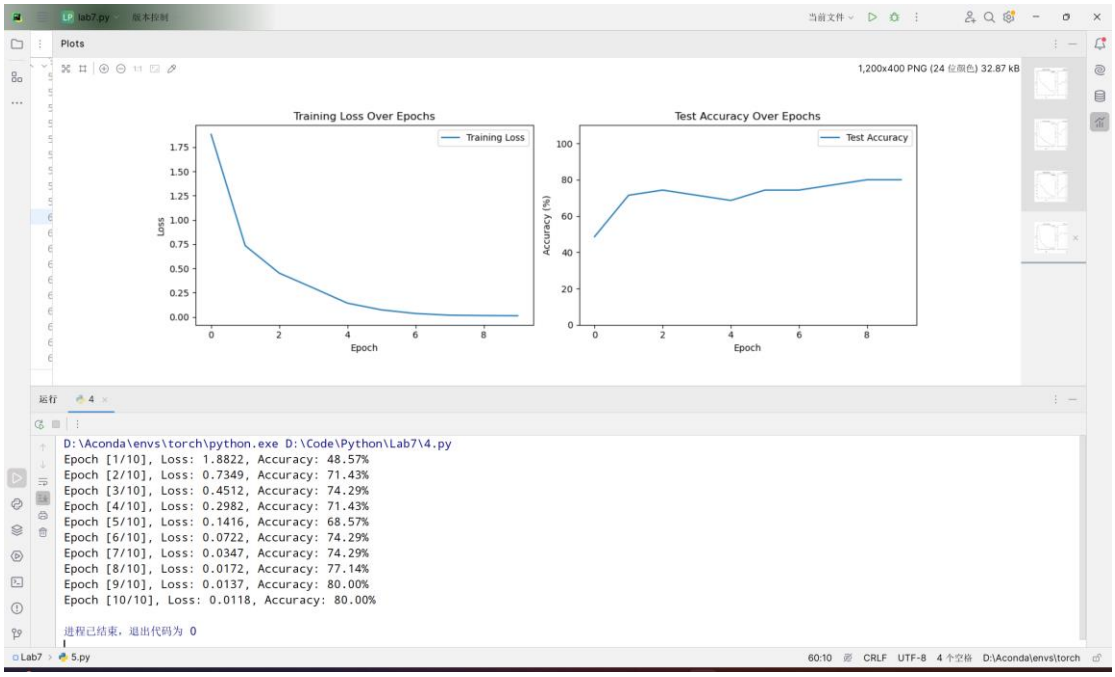
换为：

```
optimizer = optim.Adam(model.parameters(), lr=0.001)
scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=7,
gamma=0.1)
```

在 10 张图片的训练集上测试：



再在 35 张图片的测试集上测试：



可以看到这次准确率可以收敛到 80%左右，但是过程中仍然存在波动，通过参考网上对于小样本的数据使用数据增强来提升数据的随机性，即对每张训练集的图片进行随机的旋转，裁剪，平移，翻转等操作。

e. 在原基础上对训练集进行数据增强：

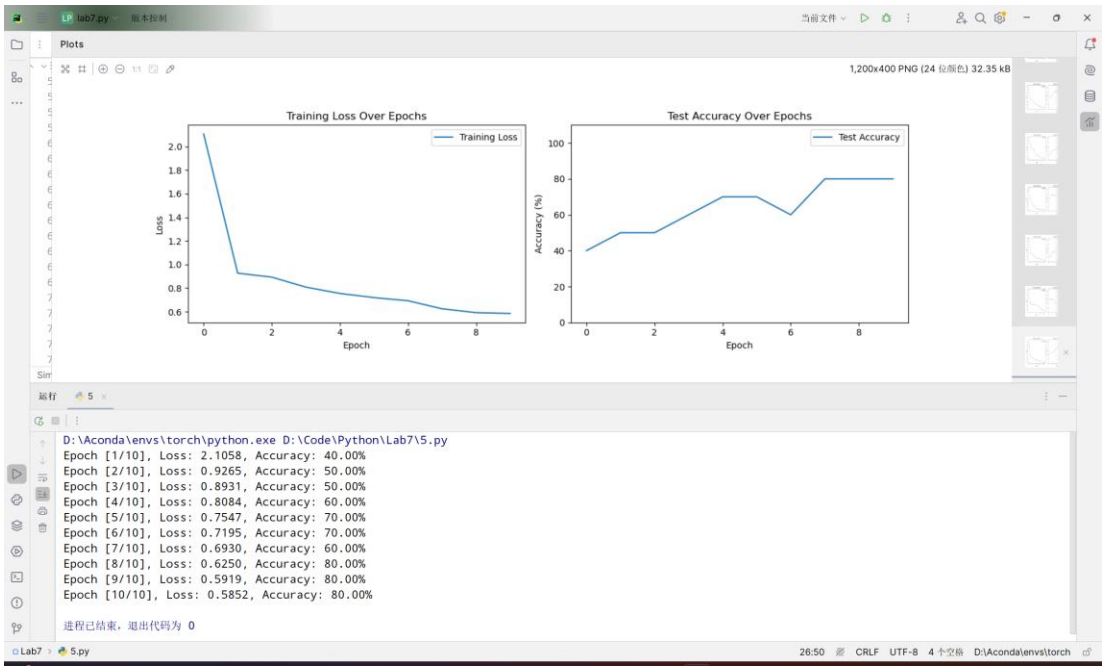
transforms 由原来训练集和测试集统一的：

```
transform = transforms.Compose([
    transforms.Resize((224, 224)),  # 调整图片大小为 224x224
    transforms.ToTensor(),  # 转换为 Tensor
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229,
0.224, 0.225])  # 归一化
])
```

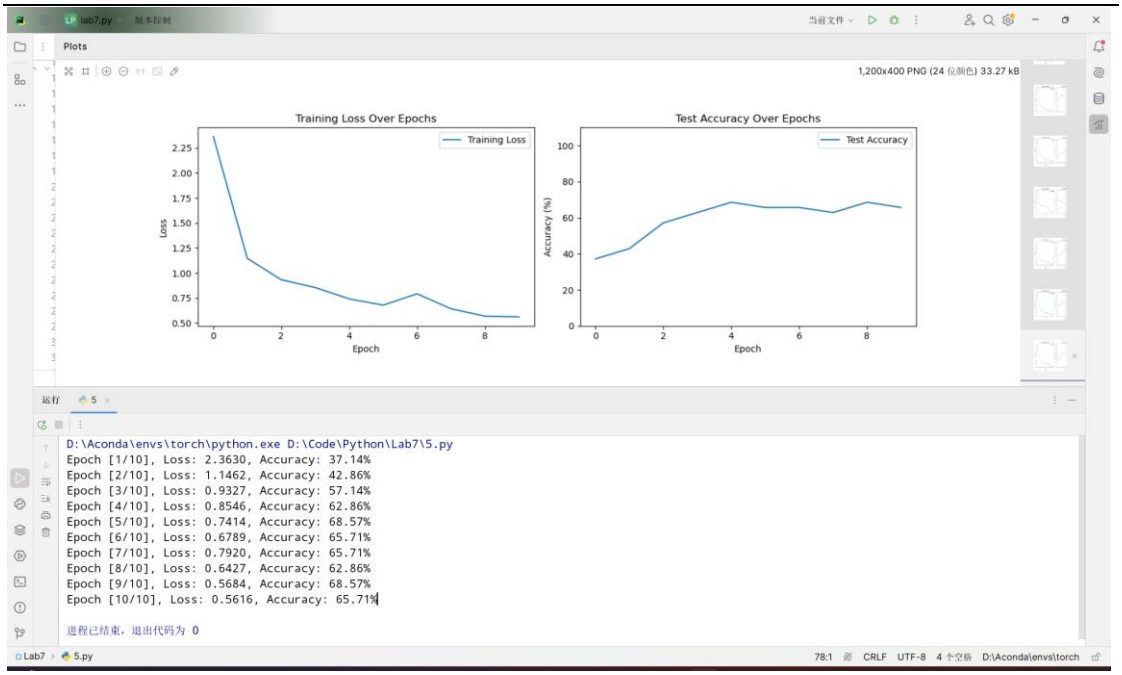
更换为对训练集做数据增强，对测试集则保持不变：

```
transform1 = transforms.Compose([
    transforms.Resize((224, 224)),  # 调整图片大小为 224x224
    transforms.RandomRotation(45), #正负 45 度的随机旋转
    transforms.RandomHorizontalFlip(p = 0.5),#随机的水平翻转
    transforms.RandomVerticalFlip(p = 0.5),#随机的垂直翻转
    transforms.ToTensor(),  # 转换为 Tensor
    transforms.Normalize(mean=[0.485, 0.456, 0.406],
std=[0.229, 0.224, 0.225])  # 归一化
])
transform2 = transforms.Compose([
    transforms.Resize((224, 224)),  # 调整图片大小为 224x224
    transforms.ToTensor(),  # 转换为 Tensor
    transforms.Normalize(mean=[0.485, 0.456, 0.406],
std=[0.229, 0.224, 0.225])  # 归一化
])
```

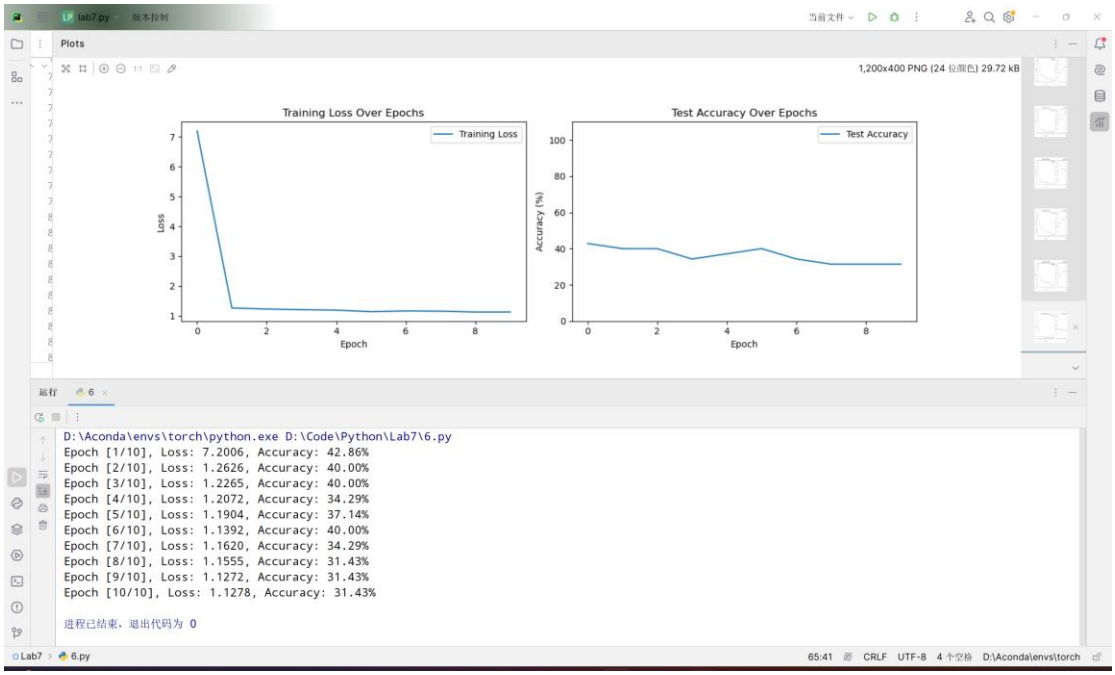
在 10 张图片的训练集上进行测试：



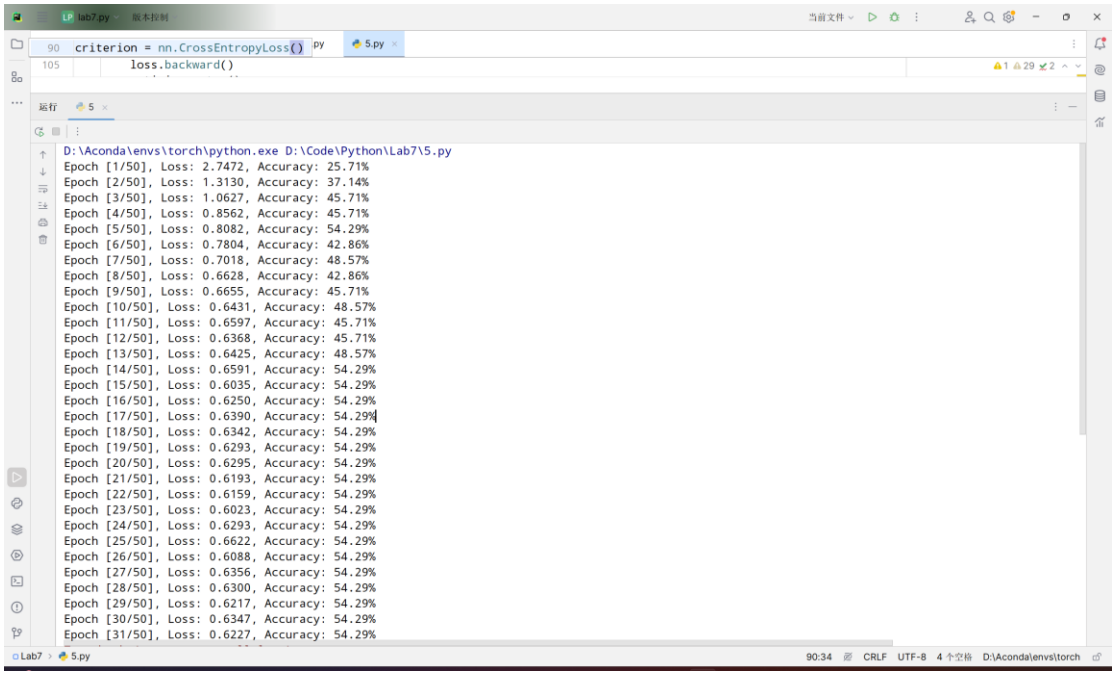
在 35 张图片的训练集上进行测试：



我们看到准确率并不理想，于是查找资料寻求合理的解释：[不是所有数据增强都可以提升精度_数据增强后精度下降了-CSDN 博客](), 在这篇文章中提到了数据增强的强度过高会与原数据有较大差异，不可避免的带来噪声问题。如果数据增强的程度太强，增强后的数据可能与原始数据分布差异过大，模型可能无法从这些极端的变换中学习到有用的信息，所以我们降低数据增强的强度（将随机旋转由正负 45 度改为正负 10 度），结果甚至更加糟糕：



还有一个可能的成立的解释，数据增强可能会增加模型训练的难度，因为增强的数据可能需要更多的时间来学习。如果训练周期没有相应增加，模型可能没有足够的时间来适应增强的数据。所以我们决定增加训练的周期来观察后期是否可以提高准确率。



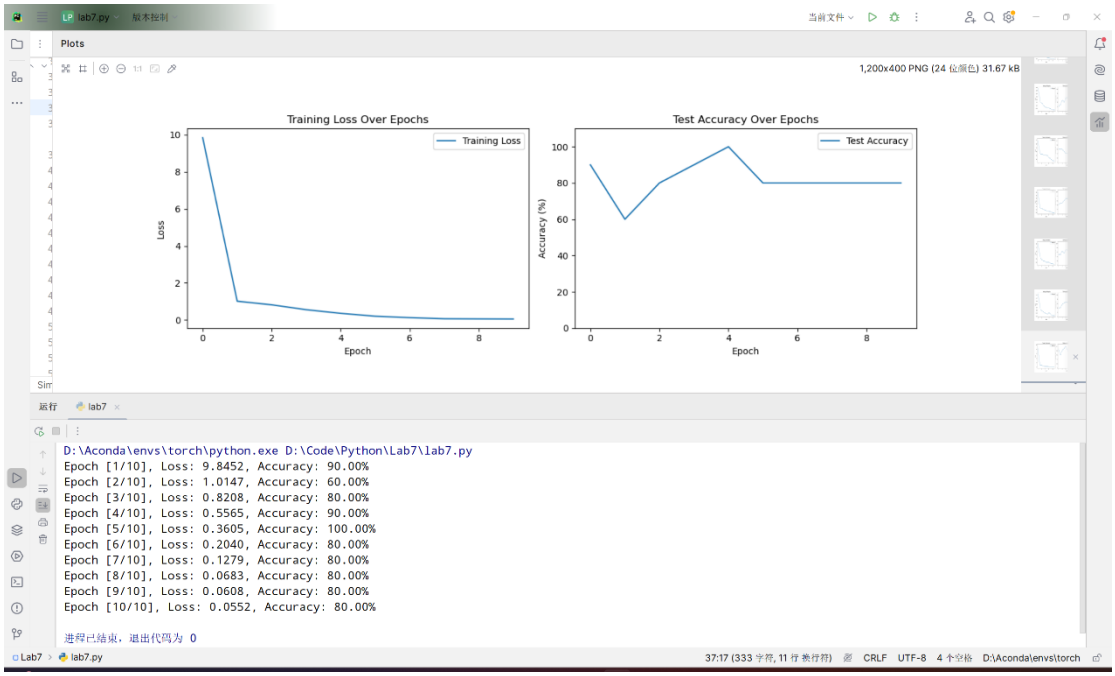
遗憾的是效果并不好，可以看到数据增强后准确率会收敛到仅 54%

左右。所以我们只能舍弃数据增强这个想法，最后我们尝试从增加全连接层的层数来提高模型预测的准确率

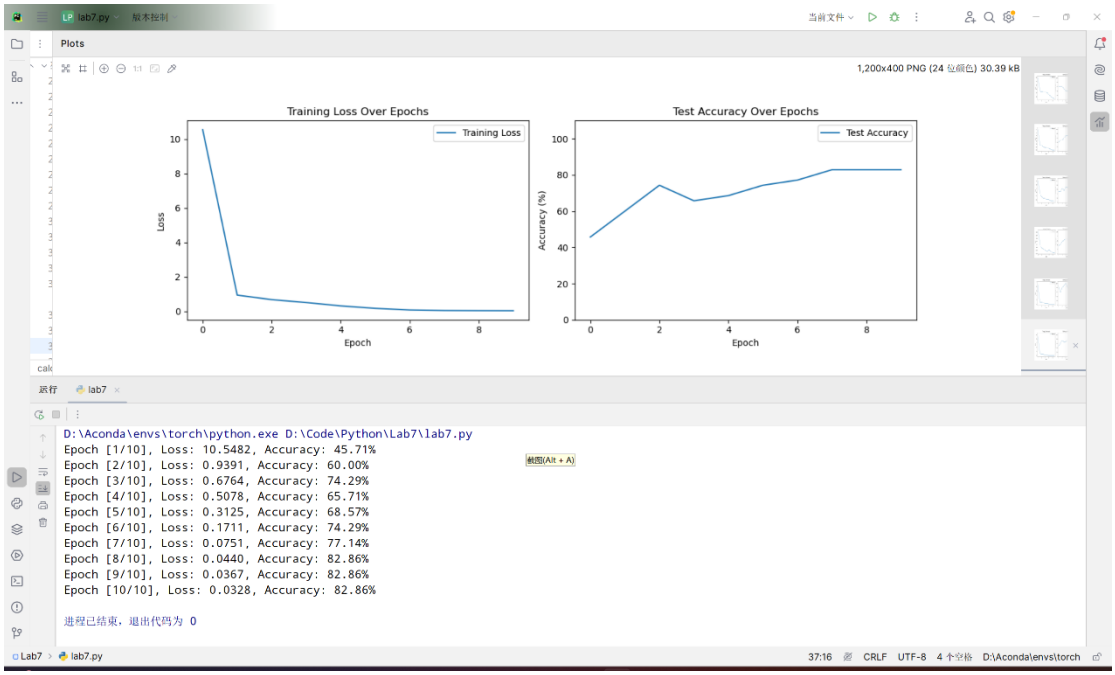
f. 多加一个全连接层：

```
  self.fc1 = nn.Linear(32*56*56, 512)
    self.droppout = nn.Dropout(0.1)
    self.fc2 = nn.Linear(512,5)
def forward(self, x):
    x = self.conv1(x)
    x = self.conv2(x)
    # 展平操作
    x = x.view(x.size(0), -1)
    x = self.fc1(x)
    x = self.droppout(x)
    x = self.fc2(x)
    return x
```

在 10 张图片的测试集上进行测试：



在 35 张图片的测试集上进行测试：



在增加一个全连接层后，模型在准确率上已经达到较为理想，而且在最后几个训练周期中已经不再波动，会收敛到 80%以上。

四、 参考资料

- 深入浅出：图像处理中的卷积神经网络（CNN）-CSDN 博客
- ImageFolder — Torchvision 0.18 documentation (pytorch.org)
- 4.1 CNN 卷积神经网络 哔哩哔哩 bilibili
- 不是所有数据增强都可以提升精度_数据增强后精度下降了-CSDN 博客
- 3-数据集与模型选择 哔哩哔哩 bilibili