# 中山大学计算机学院

# 人工智能

# 本科生实验报告

**（2022 学年春季学期）**

课程名称：Artificial Intelligence

| 教学班级 | 计科 2 班 | 专业（方向） | 计算机科学与技术 |
|---|---|---|---|
| 学号 | 22336126 | 姓名 | 李漾 |

## 一、 实验题目

**1、运用 pytorch 框架完成中药图片分类，具体见给出的数据集和测试集。**
**2、需要画出 loss、准确率曲线图。**

## 二、 实验内容

### 1. 算法原理

（1）Tensor 基本数据类型

•tensor 是 PyTorch 的基本数据类型，在使用 torch 框架进行操作时，对象一般都要求是 tensor 类型。

• 要初始化一个 tensor，通常有以下三种方式：
直接初始化、通过原始数据转化、通过 numpy 数据转化

（2）torch 常用数据操作

• 维度变换

• torch.view()或者 torch.reshape()维度重置（但总数要一致），若
根据已有维度可推算出剩下的维度

• torch.reshape()也可以重置维度

• torch.squeeze(dim)若不指定维度，则会将 tensor 中为 1 的 dim 压缩，
若指定则只会压缩对应的维度（必须为 1）

•torch.unsqueeze(dim)维度扩展

（3）torch.nn 搭建神经网络

（4）卷积神经网络

• 卷积神经网络（CNN）是一种深度学习模型，主要应用于图像识别、语音识别
等领域。与传统的神经网络相比，CNN 引入了卷积层和池化层，可以有效地减少
模型参数，提高模型性能。

• CNN 的核心是卷积层，它通过卷积运算来提取输入特征的空间信息。卷积层包括多个卷积核，每个卷积核可以检测输入数据中的某个特定特征，并生成相应的输出特征图。卷积层的参数共享机制可以大大减少模型参数数量，降低过拟合的风险。

•池化层用于进一步降低特征图的维度，同时也可以增强模型的鲁棒性。池化操作通常采用最大池化或平均池化，即对输入特征图中的每个子区域取最大值或平均值，生成新的特征图。

• 卷积操作：卷积操作是卷积神经网络的核心操作之一，其目的是从输入数据中提取特征。

卷积操作的本质是一种线性变换，它通过一个卷积核在输入数据上进行滑动，并计算每个位置上卷积核与输入数据的内积，得到一个新的特征图。卷积操作可以有效地减少需要学习的参数数量，并且具有平移不变性，即如果输入图像发生平移，提取出的特征不会发生改变。

• 池化操作：卷积操作得到的特征图通常比输入数据的尺寸大，为了减少特征图的尺寸，降低计算复杂度，我们通常会使用池化操作对特征图进行下采样。常见的池化操作包括最大池化、平均池化等，它们分别选取特定区域内的最大值或平均值作为该区域的输出，从而将特征图的尺寸降低。

•激活函数：卷积神经网络通常在卷积和池化操作之后添加一个非线性激活函数，例如 ReLU 函数，以增强神经网络的表达能力。

• 全连接层：全连接层是卷积神经网络中的一种常用结构，它将卷积和池化得到的特征图映射到输出类别上。在全连接层中，每个节点都与前一层中的所有节点相连，因此需要学习的参数非常多，计算复杂度也较高。

• Dropout 层：Dropout 操作是常用的正则化技术，可用于防止过拟合。它在训练时随机删除一部分神经元，使得每个神经元的输出不能依赖于其他神经元的存在，从而增加模型的泛化能力。

• 除了卷积层和池化层，CNN 还包括全连接层和激活函数等组件，可以构建非常复杂的模型。CNN 在图像处理任务中表现出色，它是计算机视觉领域的主流模型之一。

nn.Conv2d(in_channels,out_channels,kernel_size,stride=1,padding=0,bias=True)

NOTE: PyTorch 卷积网络输入默认格式为(N,C,H,W)其中 N 为 batch 大小（输入默认 batch 处理），C 为图像通道数（黑白 1 维，彩色 RGB 三维），H 和 W 分别为图像的高度和宽度。Conv2d 的前两个参数分别为输入和输出的通道数，kernel_size 为卷积核大小，stride 为步长默认为 1，padding 为填充默认 0。一般情况下，计算公式为

- Input: $(N, C_{in}, H_{in}, W_{in})$ or $(C_{in}, H_{in}, W_{in})$
- Output: $(N, C_{out}, H_{out}, W_{out})$ or $(C_{out}, H_{out}, W_{out})$, where

$$H_{out} = \frac{H_{in} + 2 \times padding[0] - kernel\_size[0]}{stride[0]} + 1$$

$$W_{out} = \frac{W_{in} + 2 \times padding[1] - kernel\_size[1]}{stride[1]} + 1$$

• 卷积：不再是对图像中每一个像素做处理，而是对图片上每一小块像素区域做处理，加强了图片中像素的连续性，从而处理的一个图形而不是单个像素点

• 神经网络：神经网络是一种计算模型，由大量的神经元以及层与层之间的激活函数组成

• 步骤：

1. 读入训练集和测试集中的数字图片信息以及对图片预处理
2. 用 pytorch 搭建神经网络（包括卷积和全连接神经网络）
3. 将一个 batch 的训练集中的图片输入至神经网络，得到所有数字的预测分类概率（总共 10 个数字,0123456789）
4. 根据真实标签和预测标签，利用交叉熵损失函数计算 loss 值，并进行梯度下降
5. 根据测试集计算准确率，如果准确率没收敛，跳转回步骤 3
6. 画出 loss、测试集准确率的曲线图

## 2. 伪代码

**1.数据预处理和加载：**

作用：对图像数据进行预处理和转换，创建数据集和数据加载器。

步骤：

指定数据集路径。

定义图像转换操作（调整大小、转换为张量、标准化）。

创建 ImageFolder 数据集对象。

创建 DataLoader，用于批量加载数据。

```
# 数据路径
train_path = 'path/to/train_data'
test_path = 'path/to/test_data'

# 数据预处理和转换
transform = [
    Resize((32, 32)),   # 调整图像大小
    ToTensor(),         # 转换为张量
    Normalize(mean, std)   # 标准化
]

# 创建数据集
train_dataset = ImageFolder(train_path, transform)
test_dataset = ImageFolder(test_path, transform)

# 创建数据加载器
batch_size = 16
train_loader = DataLoader(train_dataset, batch_size, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size, shuffle=True)
```

**CNN 类：**

作用：定义一个卷积神经网络，包括特征提取和分类部分。

方法：

__init__(self, num_classes)：初始化网络结构，定义卷积层、池化层和全连接层。

forward(self, x)：定义前向传播过程，应用卷积层、池化层和全连接层进行特征提取和分类。

```
class CNN:
    method __init__(num_classes):
        # 定义特征提取部分
        features = [
            Conv2d(3, 6, kernel_size=5),   # 第一个卷积层
            ReLU(),                        # 激活函数
            MaxPool2d(kernel_size=2),      # 池化层
            Conv2d(6, 16, kernel_size=5), # 第二个卷积层
            ReLU(),                        # 激活函数
```

```
            MaxPool2d(kernel_size=2)          # 池化层
        ]

        # 定义分类部分
        classifier = [
            Linear(16 * 5 * 5, 120),   # 全连接层
            ReLU(),                          # 激活函数
            Linear(120, 84),                # 全连接层
            ReLU(),                          # 激活函数
            Linear(84, num_classes)     # 全连接层
        ]

    method forward(x):
        x = apply(features, x)     # 应用特征提取层
        x = flatten(x)                  # 将特征展平为一维向量
        x = apply(classifier, x) # 应用分类层
        return x
```

**train 函数：**

作用：在训练集上训练模型，计算损失和准确率。

参数：

model：要训练的模型。

dataloader：提供训练数据的 DataLoader。

criterion：损失函数，用于计算模型输出与真实标签之间的误差。

optimizer：优化器，用于更新模型参数。

```
function train(model, dataloader, criterion, optimizer):
    set model to training mode
    initialize running_loss to 0
    initialize correct to 0
    initialize total to 0

    for each batch in dataloader:
        images, labels = batch
        move images and labels to device

        optimizer.zero_grad()   # 清除前一步的梯度

        outputs = model(images)   # 前向传播
        loss = criterion(outputs, labels)   # 计算损失
        loss.backward()   # 反向传播
        optimizer.step()   # 更新参数

        running_loss += loss.item()   # 累加损失
```

```
            predicted = get max(outputs, axis=1)    # 获取预测结果
            total += labels.size(0)
            correct += count correct predictions(predicted, labels)

        epoch_loss = running_loss / len(dataloader)    # 计算平均损失
        accuracy = correct / total    # 计算准确率
        return epoch_loss, accuracy
```

test 函数：

作用：在测试集上评估模型性能，计算损失和准确率。
参数：
model：要评估的模型。
dataloader：提供测试数据的 DataLoader。
criterion：损失函数，用于计算模型输出与真实标签之间的误差。
返回值：测试集上的平均损失和准确率。

```
function test(model, dataloader, criterion):
    set model to evaluation mode
    initialize running_loss to 0
    initialize correct to 0
    initialize total to 0

    with no gradient calculation:
        for each batch in dataloader:
            images, labels = batch
            move images and labels to device

            outputs = model(images)    # 前向传播
            loss = criterion(outputs, labels)    # 计算损失

            running_loss += loss.item()    # 累加损失
            predicted = get max(outputs, axis=1)    # 获取预测结果
            total += labels.size(0)
            correct += count correct predictions(predicted, labels)

    epoch_loss = running_loss / len(dataloader)    # 计算平均损失
    accuracy = correct / total    # 计算准确率
    return epoch_loss, accuracy
```

主程序：

作用：整合模型定义、数据加载、训练和测试过程，进行超参数调整和结果可视化。
步骤：
创建 CNN 模型并移动到设备上。
定义损失函数和优化器。

迭代不同的学习率进行训练和测试，记录每个 epoch 的损失和准确率。
打印训练和测试结果。
绘制训练和测试过程中的损失和准确率曲线图。

```
# 创建 CNN 模型
num_classes = 5
model = CNN(num_classes)

# 设置设备
device = 'cpu'
move model to device

# 定义损失函数
criterion = CrossEntropyLoss()

# 设置训练参数
num_epochs = 60
learning_rates = [0.0001, 0.001, 0.01]
initialize train_losses, train_accuracies, test_losses, test_accuracies for each learning rate

# 迭代不同的学习率
for each lr in learning_rates:
    model = CNN(num_classes)
    move model to device
    optimizer = Adam(model.parameters(), lr)

    for epoch in range(num_epochs):
        train_loss, train_accuracy = train(model, train_loader, criterion, optimizer)
        test_loss, test_accuracy = test(model, test_loader, criterion)

        record train_loss, train_accuracy, test_loss, test_accuracy

        print train and test results

# 绘制损失和准确率曲线图
plot train_losses
plot train_accuracies and test_accuracies
show plots
```

## 3. 关键代码展示（带注释）

（1）CNN 模型

```
# 定义一个简单的卷积神经网络（CNN）模型
class CNN(nn.Module):
    def __init__(self, num_classes):
        super(CNN, self).__init__()
        # 定义特征提取部分的网络层
        self.features = nn.Sequential(
            nn.Conv2d(3, 6, kernel_size=5),  # 输入通道数为3（RGB图像），输出通道数为6，卷积核大小为5x5
            nn.ReLU(inplace=True),  # 使用ReLU激活函数
            nn.MaxPool2d(kernel_size=2, stride=2),  # 最大池化层，池化窗口大小为2x2，步幅为2
            nn.Conv2d(6, 16, kernel_size=5),  # 第二个卷积层，输入通道数为6，输出通道数为16，卷积核大小为5x5
            nn.ReLU(inplace=True),  # 使用ReLU激活函数
            nn.MaxPool2d(kernel_size=2, stride=2)  # 最大池化层，池化窗口大小为2x2，步幅为2
        )
        # 定义分类部分的网络层
        self.classifier = nn.Sequential(
            nn.Linear(16 * 5 * 5, 120),  # 全连接层，输入大小为16x5x5（展平后），输出大小为120
            nn.ReLU(inplace=True),  # 使用ReLU激活函数
            nn.Linear(120, 84),  # 全连接层，输入大小为120，输出大小为84
            nn.ReLU(inplace=True),  # 使用ReLU激活函数
            nn.Linear(84, num_classes)  # 全连接层，输入大小为84，输出大小为num_classes（分类数）
        )

    def forward(self, x):
        x = self.features(x)  # 特征提取部分的前向传播
        x = torch.flatten(x, 1)  # 将特征展平为一维向量
        x = self.classifier(x)  # 分类部分的前向传播
        return x
```

（2）数据预处理

```
# 数据路径
train_path = r'D:\人工智能\22336126_李漾_lab_6\data\train'
test_path = r'D:\人工智能\22336126_李漾_lab_6\data\test'

# 数据预处理和转换
transform = transforms.Compose([
    transforms.Resize((32, 32)),  # 调整图像大小到32x32
    transforms.ToTensor(),  # 转换为张量
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])  # 标准化
])

# 创建数据集
train_dataset = ImageFolder(train_path, transform=transform)
test_dataset = ImageFolder(test_path, transform=transform)

# 创建数据加载器
batch_size = 16
train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
test_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
```

（3）训练函数

```
# 定义训练函数
def train(model, dataloader, criterion, optimizer):
    model.train()  # 设置模型为训练模式
    running_loss = 0.0  # 初始化累计损失
    correct = 0  # 初始化正确预测数
    total = 0  # 初始化总样本数

    for images, labels in dataloader:
        images = images.to(device)  # 将图像数据移到设备上（CPU或GPU）
        labels = labels.to(device)  # 将标签数据移到设备上

        optimizer.zero_grad()  # 清除前一步的梯度

        outputs = model(images)  # 前向传播，计算输出
        loss = criterion(outputs, labels)  # 计算损失
        loss.backward()  # 反向传播，计算梯度
        optimizer.step()  # 更新模型参数

        running_loss += loss.item()  # 累加损失
        _, predicted = outputs.max(1)  # 获取预测结果中概率最高的类别
        total += labels.size(0)  # 累加总样本数
        correct += predicted.eq(labels).sum().item()  # 累加正确预测数

    epoch_loss = running_loss / len(dataloader)  # 计算平均损失
    accuracy = correct / total  # 计算准确率

    return epoch_loss, accuracy
```

（4）测试函数

```
# 定义测试函数
def test(model, dataloader, criterion):
    model.eval()  # 设置模型为评估模式
    running_loss = 0.0  # 初始化累计损失
    correct = 0  # 初始化正确预测数
    total = 0  # 初始化总样本数

    with torch.no_grad():  # 不计算梯度
        for images, labels in dataloader:
            images = images.to(device)  # 将图像数据移到设备上
            labels = labels.to(device)  # 将标签数据移到设备上

            outputs = model(images)  # 前向传播，计算输出
            loss = criterion(outputs, labels)  # 计算损失

            running_loss += loss.item()  # 累加损失
            _, predicted = outputs.max(1)  # 获取预测结果中概率最高的类别
            total += labels.size(0)  # 累加总样本数
            correct += predicted.eq(labels).sum().item()  # 累加正确预测数

    epoch_loss = running_loss / len(dataloader)  # 计算平均损失
    accuracy = correct / total  # 计算准确率

    return epoch_loss, accuracy
```

## 4. 创新点&优化（如果有）

（1）一次显示不同学习率的影响：

```
# 设置训练参数
num_epochs = 60  # 训练迭代次数
learning_rates = [0.0001, 0.001, 0.01]  # 不同的学习率
train_losses = {lr: [] for lr in learning_rates}  # 用于存储训练损失
train_accuracies = {lr: [] for lr in learning_rates}  # 用于存储训练准确率
test_losses = {lr: [] for lr in learning_rates}  # 用于存储测试损失
test_accuracies = {lr: [] for lr in learning_rates}  # 用于存储测试准确率

# 迭代不同的学习率
for lr in learning_rates:
    model = CNN(num_classes)  # 重置模型
    model = model.to(device)  # 将模型移到设备上
    optimizer = optim.Adam(model.parameters(), lr=lr)  # 使用Adam优化器

    for epoch in range(num_epochs):
        train_loss, train_accuracy = train(model, train_loader, criterion, optimizer)  # 训练模型
        test_loss, test_accuracy = test(model, test_loader, criterion)  # 测试模型

        train_losses[lr].append(train_loss)  # 记录训练损失
        train_accuracies[lr].append(train_accuracy)  # 记录训练准确率
        test_losses[lr].append(test_loss)  # 记录测试损失
        test_accuracies[lr].append(test_accuracy)  # 记录测试准确率

        # 打印训练和测试结果
        print(f'Learning Rate: {lr}, Epoch {epoch+1}/{num_epochs}: '
              f'Train Loss: {train_loss:.4f}, Train Accuracy: {train_accuracy:.4f}, '
              f'Test Loss: {test_loss:.4f}, Test Accuracy: {test_accuracy:.1f}')
```

（2）数据增强

在训练过程中进行数据增强，可以有效防止过拟合，提高模型的泛化能力。

```
# 数据预处理和转换
transform = transforms.Compose([
    transforms.Resize((32, 32)),
    transforms.RandomHorizontalFlip(),  # 随机水平翻转
    transforms.RandomRotation(10),      # 随机旋转
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
])
```

（3）批归一化：在每个卷积层后添加批归一化层，提高训练稳定性。

Dropout：在全连接层中添加 Dropout 层，防止过拟合。

```
class CNN(nn.Module):
    def __init__(self, num_classes):
        super(CNN, self).__init__()
        self.features = nn.Sequential(
            nn.Conv2d(3, 6, kernel_size=5),
            nn.BatchNorm2d(6),  # 添加批归一化层
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.Conv2d(6, 16, kernel_size=5),
            nn.BatchNorm2d(16),  # 添加批归一化层
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2)
        )
        self.classifier = nn.Sequential(
            nn.Linear(16 * 5 * 5, 120),
            nn.ReLU(inplace=True),
            nn.Dropout(p=0.5),  # 添加Dropout层
            nn.Linear(120, 84),
            nn.ReLU(inplace=True),
            nn.Dropout(p=0.5),  # 添加Dropout层
            nn.Linear(84, num_classes)
        )

    def forward(self, x):
        x = self.features(x)
        x = torch.flatten(x, 1)
        x = self.classifier(x)
        return x
```

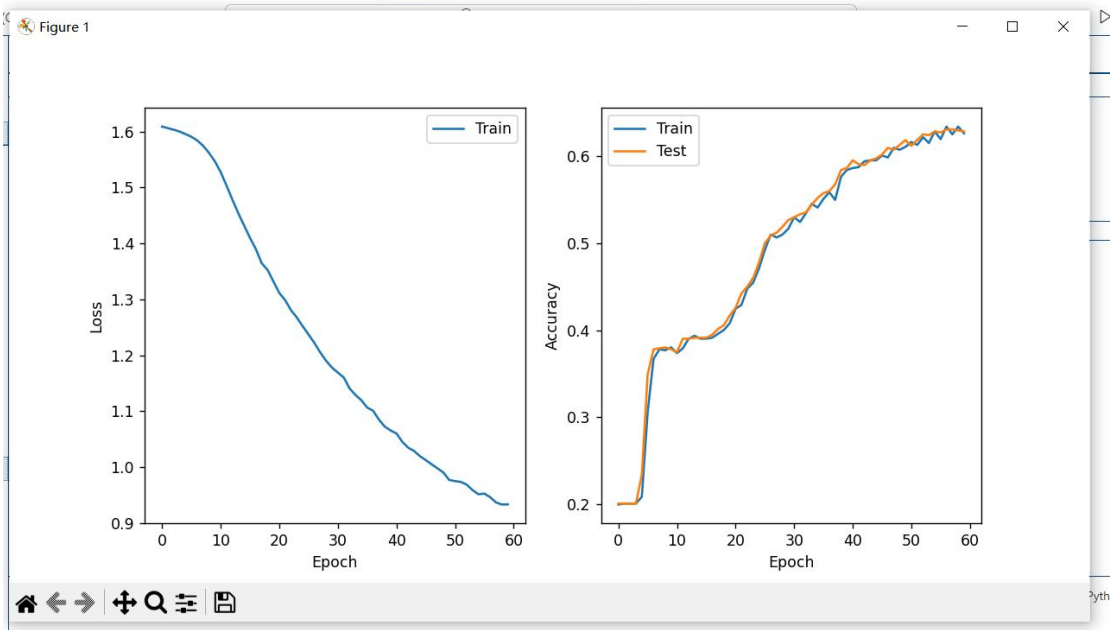
动态学习率调整：使用 StepLR 学习率调度器，根据训练进展动态调整学习率。

```
for lr in learning_rates:
    model = CNN(num_classes=5)
    model = model.to(device)
    optimizer = optim.Adam(model.parameters(), lr=lr)
    scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=20, gamma=0.1)  # 每20个epoch学习率降低为原来的0.1倍

    for epoch in range(num_epochs):
        train_loss, train_accuracy = train(model, train_loader, criterion, optimizer)
        test_loss, test_accuracy = test(model, test_loader, criterion)

        scheduler.step()  # 更新学习率

        train_losses[lr].append(train_loss)
        train_accuracies[lr].append(train_accuracy)
        test_losses[lr].append(test_loss)
        test_accuracies[lr].append(test_accuracy)

        print(f'Learning Rate: {lr}, Epoch {epoch+1}/{num_epochs}: '
              f'Train Loss: {train_loss:.4f}, Train Accuracy: {train_accuracy:.4f}, '
              f'Test Loss: {test_loss:.4f}, Test Accuracy: {test_accuracy:.4f}')
```

# 三、 实验结果及分析

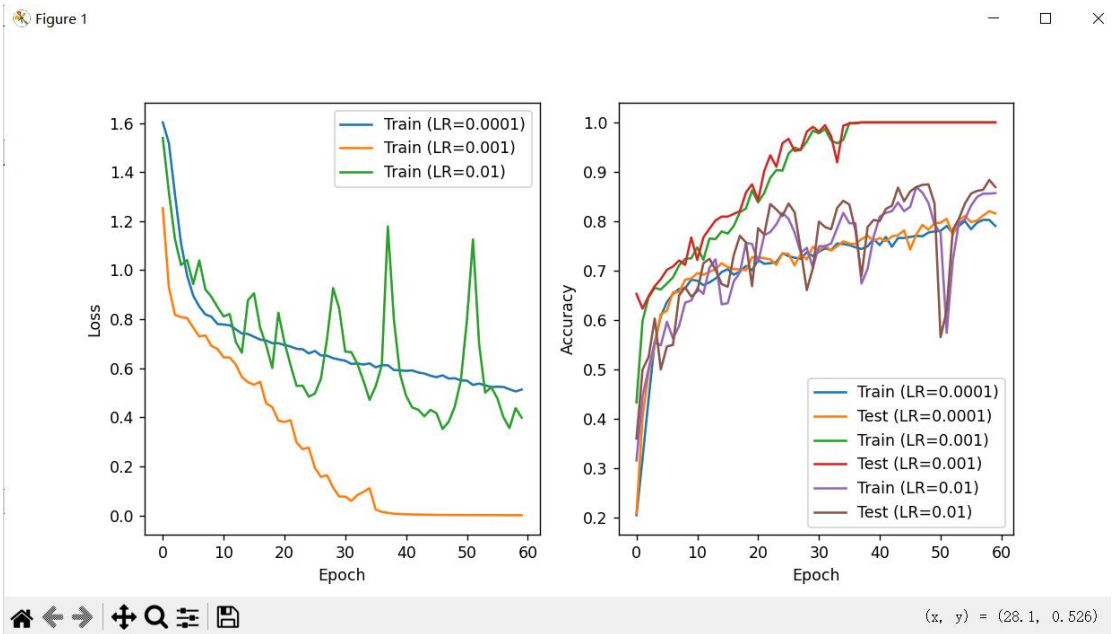
## 1. 实验结果展示示例

单独显示：学习率 lr=0.00001



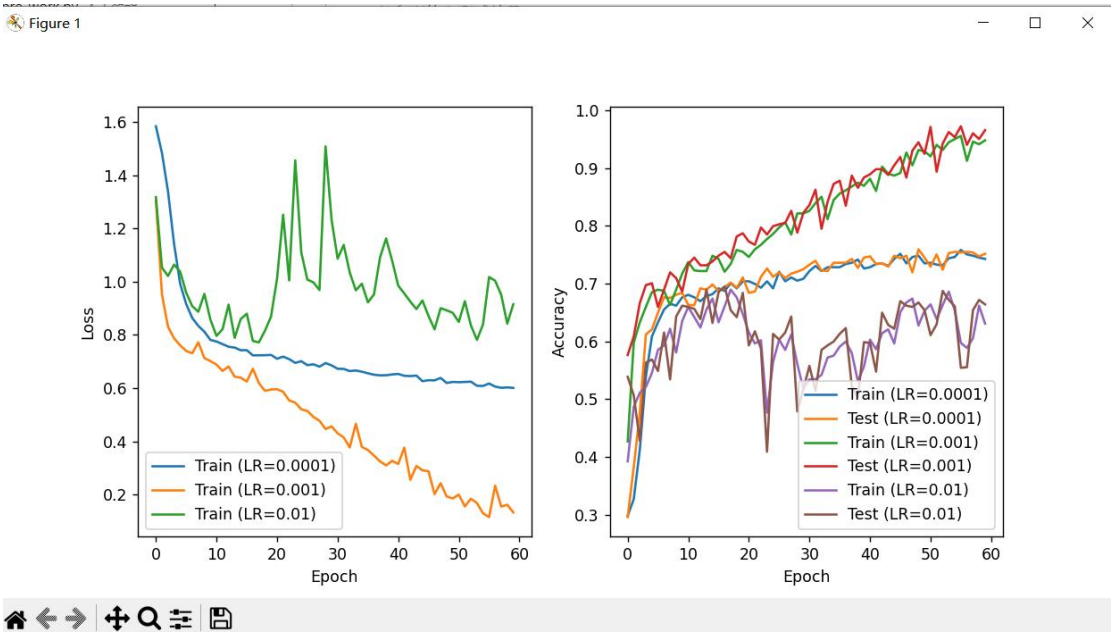

```
Epoch 50/60: Train Loss: 0.9770, Train Accuracy: 0.6109, Test Loss: 0.9747, Test Accuracy: 0.6
Epoch 51/60: Train Loss: 0.9749, Train Accuracy: 0.6164, Test Loss: 0.9705, Test Accuracy: 0.6
Epoch 52/60: Train Loss: 0.9735, Train Accuracy: 0.6131, Test Loss: 0.9677, Test Accuracy: 0.6
Epoch 53/60: Train Loss: 0.9686, Train Accuracy: 0.6220, Test Loss: 0.9634, Test Accuracy: 0.6
Epoch 54/60: Train Loss: 0.9588, Train Accuracy: 0.6153, Test Loss: 0.9542, Test Accuracy: 0.6
Epoch 55/60: Train Loss: 0.9515, Train Accuracy: 0.6286, Test Loss: 0.9513, Test Accuracy: 0.6
Epoch 56/60: Train Loss: 0.9526, Train Accuracy: 0.6197, Test Loss: 0.9403, Test Accuracy: 0.6
Epoch 57/60: Train Loss: 0.9461, Train Accuracy: 0.6341, Test Loss: 0.9320, Test Accuracy: 0.6
Epoch 58/60: Train Loss: 0.9368, Train Accuracy: 0.6253, Test Loss: 0.9391, Test Accuracy: 0.6
Epoch 59/60: Train Loss: 0.9331, Train Accuracy: 0.6341, Test Loss: 0.9291, Test Accuracy: 0.6
Epoch 60/60: Train Loss: 0.9333, Train Accuracy: 0.6264, Test Loss: 0.9243, Test Accuracy: 0.6
```

不同学习率统一显示：

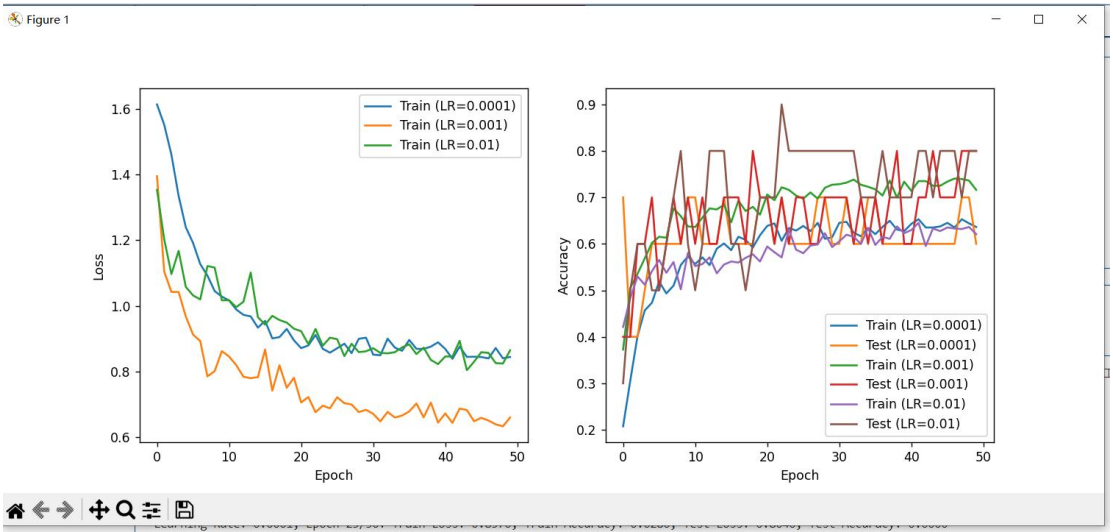



```
Learning Rate: 0.01, Epoch 50/60: Train Loss: 0.5499, Train Accuracy: 0.7938, Test Loss: 0.4945, Test Accuracy: 0.8
Learning Rate: 0.01, Epoch 51/60: Train Loss: 0.7947, Train Accuracy: 0.7761, Test Loss: 1.2597, Test Accuracy: 0.6
Learning Rate: 0.01, Epoch 52/60: Train Loss: 1.1253, Train Accuracy: 0.5743, Test Loss: 0.8710, Test Accuracy: 0.6
Learning Rate: 0.01, Epoch 53/60: Train Loss: 0.7010, Train Accuracy: 0.7206, Test Loss: 0.5141, Test Accuracy: 0.8
Learning Rate: 0.01, Epoch 54/60: Train Loss: 0.5005, Train Accuracy: 0.7960, Test Loss: 0.5022, Test Accuracy: 0.8
Learning Rate: 0.01, Epoch 55/60: Train Loss: 0.5222, Train Accuracy: 0.8038, Test Loss: 0.4114, Test Accuracy: 0.8
Learning Rate: 0.01, Epoch 56/60: Train Loss: 0.4779, Train Accuracy: 0.8315, Test Loss: 0.3964, Test Accuracy: 0.9
Learning Rate: 0.01, Epoch 57/60: Train Loss: 0.4004, Train Accuracy: 0.8492, Test Loss: 0.3471, Test Accuracy: 0.9
Learning Rate: 0.01, Epoch 58/60: Train Loss: 0.3560, Train Accuracy: 0.8559, Test Loss: 0.3344, Test Accuracy: 0.9
Learning Rate: 0.01, Epoch 59/60: Train Loss: 0.4373, Train Accuracy: 0.8559, Test Loss: 0.3186, Test Accuracy: 0.9
Learning Rate: 0.01, Epoch 60/60: Train Loss: 0.3987, Train Accuracy: 0.8570, Test Loss: 0.3344, Test Accuracy: 0.9
```

数据增强后显示：（可能模型还没达到过拟合的程度，所以体现不出优势）

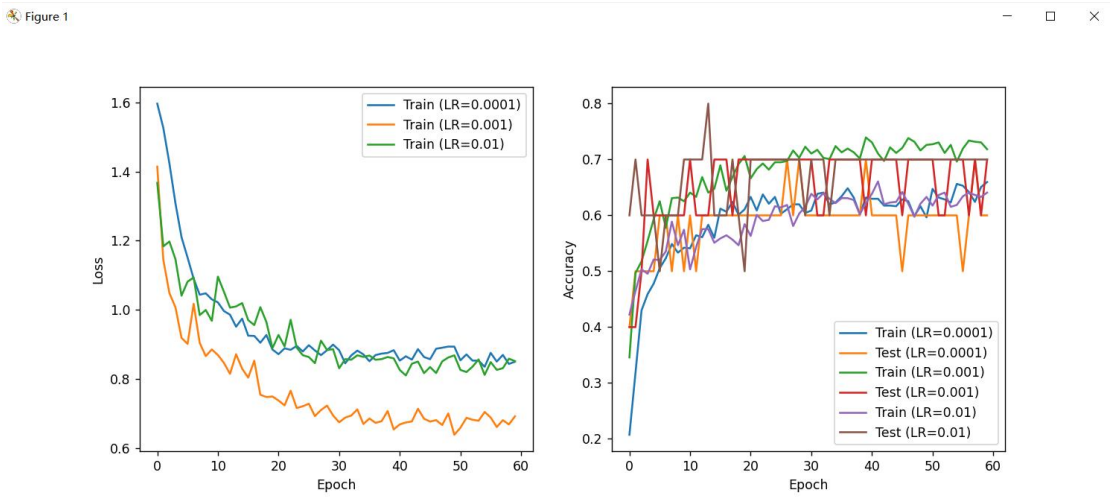



进行：数据增强，批归一化；在全连接层中添加 Dropout 层；和动态学习率调整后的显示：迭代 50 次：




```
Learning Rate: 0.0001, Epoch 26/50: Train Loss: 0.8709, Train Accuracy: 0.6386, Test Loss: 0.7139, Test Accuracy: 0.6000
Learning Rate: 0.0001, Epoch 27/50: Train Loss: 0.8849, Train Accuracy: 0.6275, Test Loss: 0.7838, Test Accuracy: 0.6000
Learning Rate: 0.0001, Epoch 28/50: Train Loss: 0.8562, Train Accuracy: 0.6452, Test Loss: 0.6633, Test Accuracy: 0.7000
Learning Rate: 0.0001, Epoch 29/50: Train Loss: 0.8997, Train Accuracy: 0.6109, Test Loss: 0.7703, Test Accuracy: 0.7000
Learning Rate: 0.0001, Epoch 30/50: Train Loss: 0.9032, Train Accuracy: 0.6142, Test Loss: 0.7646, Test Accuracy: 0.6000
Learning Rate: 0.0001, Epoch 31/50: Train Loss: 0.8515, Train Accuracy: 0.6452, Test Loss: 0.7492, Test Accuracy: 0.6000
Learning Rate: 0.0001, Epoch 32/50: Train Loss: 0.8498, Train Accuracy: 0.6475, Test Loss: 0.7526, Test Accuracy: 0.7000
Learning Rate: 0.0001, Epoch 33/50: Train Loss: 0.9001, Train Accuracy: 0.6242, Test Loss: 0.7463, Test Accuracy: 0.6000
Learning Rate: 0.0001, Epoch 34/50: Train Loss: 0.8732, Train Accuracy: 0.6164, Test Loss: 0.7543, Test Accuracy: 0.6000
Learning Rate: 0.0001, Epoch 35/50: Train Loss: 0.8631, Train Accuracy: 0.6341, Test Loss: 0.7010, Test Accuracy: 0.7000
```

迭代 60 次：




```
Learning Rate: 0.01, Epoch 49/60: Train Loss: 0.8620, Train Accuracy: 0.6208, Test Loss: 0.6841, Test Accuracy: 0.7000
Learning Rate: 0.01, Epoch 50/60: Train Loss: 0.8681, Train Accuracy: 0.6330, Test Loss: 0.7140, Test Accuracy: 0.7000
Learning Rate: 0.01, Epoch 51/60: Train Loss: 0.8260, Train Accuracy: 0.6175, Test Loss: 0.6983, Test Accuracy: 0.7000
Learning Rate: 0.01, Epoch 52/60: Train Loss: 0.8198, Train Accuracy: 0.6364, Test Loss: 0.6876, Test Accuracy: 0.7000
Learning Rate: 0.01, Epoch 53/60: Train Loss: 0.8353, Train Accuracy: 0.6408, Test Loss: 0.7234, Test Accuracy: 0.7000
Learning Rate: 0.01, Epoch 54/60: Train Loss: 0.8563, Train Accuracy: 0.6153, Test Loss: 0.6588, Test Accuracy: 0.7000
Learning Rate: 0.01, Epoch 55/60: Train Loss: 0.8116, Train Accuracy: 0.6186, Test Loss: 0.7077, Test Accuracy: 0.7000
Learning Rate: 0.01, Epoch 56/60: Train Loss: 0.8486, Train Accuracy: 0.6341, Test Loss: 0.7439, Test Accuracy: 0.7000
Learning Rate: 0.01, Epoch 57/60: Train Loss: 0.8261, Train Accuracy: 0.6408, Test Loss: 0.6937, Test Accuracy: 0.7000
Learning Rate: 0.01, Epoch 58/60: Train Loss: 0.8309, Train Accuracy: 0.6364, Test Loss: 0.6774, Test Accuracy: 0.7000
Learning Rate: 0.01, Epoch 59/60: Train Loss: 0.8584, Train Accuracy: 0.6330, Test Loss: 0.7474, Test Accuracy: 0.7000
Learning Rate: 0.01, Epoch 60/60: Train Loss: 0.8514, Train Accuracy: 0.6408, Test Loss: 0.7016, Test Accuracy: 0.7000
```

可以看到 loss 函数基本都收敛，准确率更为稳定。

## 2. 评测指标展示及分析（机器学习实验必须有此项，其它可分析运行时间等）

• 可以看到在学习率不同的 4 个模型中，学习率为 0.01 时，学习率过大，学习速度也较快但是，难以收敛；而当学习率过低，为 0.00001 时，会导致学习速度太慢

• 而合适的学习率能够加速训练拟合的同时又能更好的逼近最优解