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	信息与计算科学班	专业（方向）	信息与计算科学
学号	21311359	姓名	何凯迪

一、 实验题目

1、运用 pytorch 框架完成中药图片分类。

2、需要画出 loss、准确率曲线图。

二、 实验内容

1. 算法原理

卷积神经网络（CNN）

卷积即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。在图像识别或分类任务中，卷积神经网络便是一种利用卷积层提取图像特征并进行相应的处理的神经网络，网络经过计算模型，由大量的神经元以及层与层之间的激活函数组成。

输入层：CNN 的输入通常是一个二维矩阵，如图像。图像的每个像素通常由红、绿、蓝通道的值组成。

卷积层：卷积层是 CNN 的核心部分。它由一组可学习的卷积核组成。每个卷积核都是一个小的二维矩阵，通过在输入数据上滑动并与之进行卷积操作来提取特征。卷积操作是通过计算输入数据和卷积核之间的乘积和来完成的。通过在不同位置应用不同的卷积核，卷积层能够提取不同的特征。

激活函数：在卷积层的输出上应用一个非线性激活函数（如 ReLU）可以引入非线性特性，增加模型的表达能力。

池化层：池化层用于减小特征图的空间尺寸，同时保留主要特征。常用的池化操作是最大池化，它在每个区域中选择最大值作为池化后的值。

全连接层：在经过多个卷积层和池化层后，通常会连接一个或多个全连接层。全连接层中的每个神经元与前一层的所有神经元都连接起来。全连接层可以捕捉到全局特征，并产生最终的分类结果。

输出层：输出层是最后一层全连接层，它的神经元数量为任务的类别数量。使用一个适当的激活函数来将神经元的输出转换为概率分布，表示每个类别的概率。

损失函数：损失函数用于衡量模型预测结果与真实标签之间的差距。常用的损失函数包括交叉熵损失函数。

反向传播：通过使用反向传播算法，CNN 可以根据损失函数的梯度信息来更新模型中的参数，以使预测结果逐渐接近真实标签。
训练和优化：CNN 使用训练数据来学习特征的表示，并通过不断迭代的优化过程来最小化损失函数。常用的优化算法包括随机梯度下降（SGD）和其变种。
预测：在训练完成后，CNN 可以用于对新的未见数据进行预测。通过将输入数据传递到 CNN 中并根据输出层的概率分布进行分类，可以得到模型的预测结果。

2. 伪代码

（以下为具有普遍性的伪代码，实际实验代码做了一定改动）

训练：

```
def Train(epochs_num, model, criterion, train_loader):
    # 对整个训练集进行 epochs_num 次训练
    for epoch in range(epochs_num):
        total_loss = 0.0

        # 在每个 epoch 中遍历训练集
        for images, labels in train_loader:
            # 将图像转换为符合输入的格式
            samples = images.reshape(-1, input_size)
            # 输入图像并获取模型输出的标签
            output = model(samples)
            # 计算损失函数交叉熵
            loss = criterion(output, labels)
            # 梯度清零
            model.zero_grad()
            # 反向传播
            loss.backward()
            # 更新参数
            optimizer.step()
            total_loss += loss.item()

        # 输出每个 epoch 的平均损失
        avg_loss = total_loss / len(train_loader)
        print(f"Epoch {epoch+1}/{epochs_num}, Average Loss: {avg_loss:.4f}")
    return model
```

测试：

```
def Test(test_loader, model):
    correct = 0
    types = []
    # 禁用梯度计算，因为在测试阶段不需要更新参数
    with torch.no_grad():
        # 在测试集上进行推断
        for images, labels in test_loader:
            # 输入图像并获取模型输出的标签
            output = model(images)
            _, predicted = torch.max(output.data, 1)
            types.extend(predicted.tolist())
            # 统计预测正确的样本数
            correct += (predicted == labels).sum().item()
    # 计算正确率
    rate = correct / len(test_loader.dataset)
    return types, rate
```

损失函数及优化器：

```
# 定义损失函数和优化器
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

3. 关键代码展示（带注释）

处理数据：

```
# 数据预处理和转换
transform = transforms.Compose([
    transforms.Resize((32, 32)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
])

# 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)
```

损失函数采用交叉熵，优化器选用 Adam：

```
# 定义损失函数和优化器
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001) # 学习率
```

搭建神经网络：

（仿照 LeNet 框架）

```
class LeNet(nn.Module):
    def __init__(self, num_classes):
        super(LeNet, self).__init__()
        # 定义特征提取部分
        self.features = nn.Sequential(
            nn.Conv2d(3, 6, kernel_size=5),  # 输入通道数 3，输出通道数 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 2x2，步幅为 2
            nn.Conv2d(6, 16, kernel_size=5),  # 输入通道数 6，输出通道数 16，卷积核大小 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 2x2，步幅为 2
        )

        # 定义分类部分
        self.classifier = nn.Sequential(
            nn.Linear(16 * 5 * 5, 120),  # 全连接层，输入大小为 16x5x5，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
        )

    def forward(self, x):
        x = self.features(x)
        x = torch.flatten(x, 1)  # 将特征展平为一维向量
        x = self.classifier(x)
        return x
```

训练：

```python
def train(model, dataloader, criterion, optimizer):
    model.train()  # 设置为训练模式
    running_loss = 0.0  # 用于累计每个 batch 的损失值
    correct = 0
    total = 0

    for images, labels in dataloader:
        images = images.to(device)
        labels = labels.to(device)

        optimizer.zero_grad()  # 清除梯度
        outputs = model(images)  # 通过模型进行前向传播，得到预测结果
        loss = criterion(outputs, labels)  # 计算损失函数值
        loss.backward()  # 反向传播，计算梯度
        optimizer.step()

        running_loss += loss.item()  # 累加损失函数值
        _, predicted = outputs.max(1)  # 获取预测结果中概率最高的类别
        total += labels.size(0)
        correct += predicted.eq(labels).sum().item()

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

    return epoch_loss, accuracy
```

测试：（测试无需计算梯度）

```python
# 定义测试函数
def test(model, dataloader, criterion):
    model.eval()  # 设置为评估模式
    running_loss = 0.0  # 用于累计每个 batch 的损失值
    correct = 0
    total = 0

    with torch.no_grad():  # 不需要计算梯度
        for images, labels in dataloader:
            images = images.to(device)
            labels = labels.to(device)
            outputs = model(images)  # 通过模型进行前向传播，得到预测结果
            loss = criterion(outputs, labels)  # 计算损失函数值
```

```
            running_loss += loss.item()  # 累加损失函数值
            _, predicted = outputs.max(1)  # 获取预测结果中概率最高的类别
            total += labels.size(0)
            correct += predicted.eq(labels).sum().item()

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

    return epoch_loss, accuracy
```

4. 创新点&优化（如果有）

比较了多种不同的模型（手动实现的 LeNet、手动实现的 ResNet、下载 pytorch 自带的 ResNet18）对结果的影响，最后选择了比较典型的 LeNet 模型。

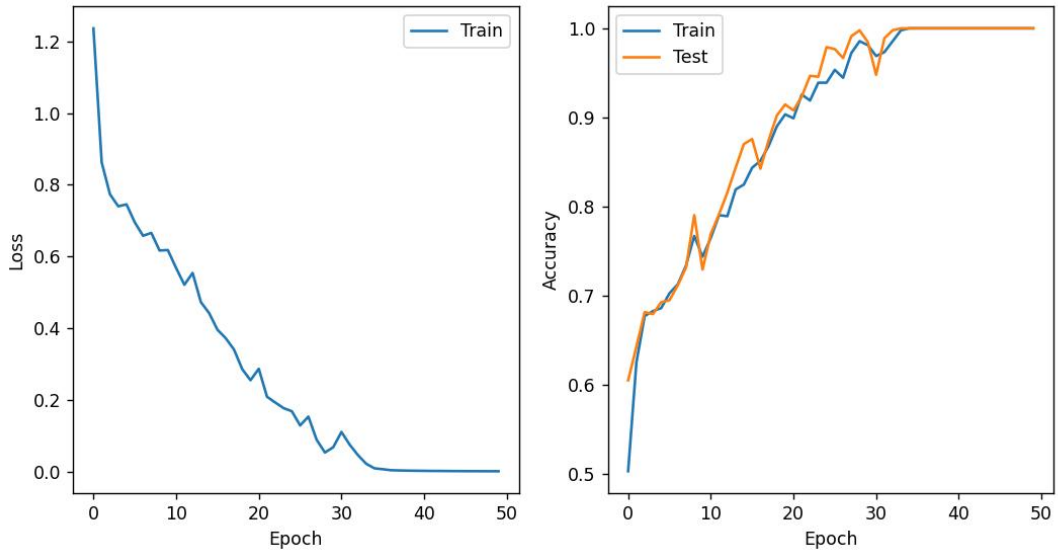
三、 实验结果及分析

1. 实验结果展示示例（可图可表可文字，尽量可视化）

输出：

```
Epoch 38/50: Train Loss: 0.0029, Train Accuracy: 1.0000, Test Loss: 0.0023, Test Accuracy: 1.0
Epoch 39/50: Train Loss: 0.0024, Train Accuracy: 1.0000, Test Loss: 0.0020, Test Accuracy: 1.0
Epoch 40/50: Train Loss: 0.0021, Train Accuracy: 1.0000, Test Loss: 0.0018, Test Accuracy: 1.0
Epoch 41/50: Train Loss: 0.0018, Train Accuracy: 1.0000, Test Loss: 0.0015, Test Accuracy: 1.0
Epoch 42/50: Train Loss: 0.0015, Train Accuracy: 1.0000, Test Loss: 0.0014, Test Accuracy: 1.0
Epoch 43/50: Train Loss: 0.0014, Train Accuracy: 1.0000, Test Loss: 0.0012, Test Accuracy: 1.0
Epoch 44/50: Train Loss: 0.0012, Train Accuracy: 1.0000, Test Loss: 0.0011, Test Accuracy: 1.0
Epoch 45/50: Train Loss: 0.0011, Train Accuracy: 1.0000, Test Loss: 0.0009, Test Accuracy: 1.0
Epoch 46/50: Train Loss: 0.0009, Train Accuracy: 1.0000, Test Loss: 0.0008, Test Accuracy: 1.0
Epoch 47/50: Train Loss: 0.0009, Train Accuracy: 1.0000, Test Loss: 0.0007, Test Accuracy: 1.0
Epoch 48/50: Train Loss: 0.0008, Train Accuracy: 1.0000, Test Loss: 0.0007, Test Accuracy: 1.0
Epoch 49/50: Train Loss: 0.0007, Train Accuracy: 1.0000, Test Loss: 0.0006, Test Accuracy: 1.0
Epoch 50/50: Train Loss: 0.0007, Train Accuracy: 1.0000, Test Loss: 0.0005, Test Accuracy: 1.0
```

图表：

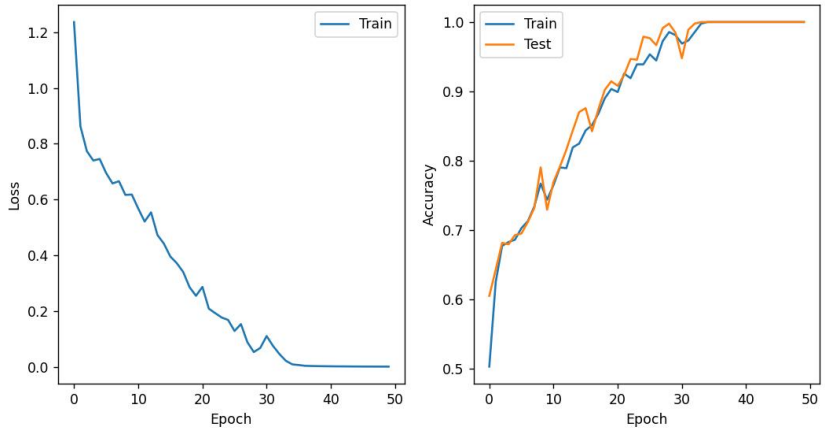


2. 评测指标展示及分析（机器学习实验必须有此项，其它可分析运行时间等）

做了三种不同的模型（手动实现的 LeNet、手动实现的 ResNet、下载 pytorch 自带的 ResNet18）对结果的影响，得到结果如下：

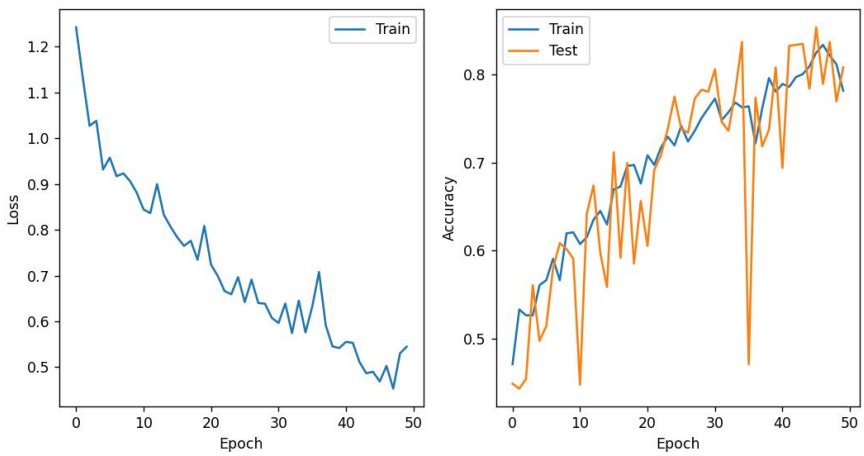
1、手动实现的 LeNet：

```
Epoch 38/50: Train Loss: 0.0029, Train Accuracy: 1.0000, Test Loss: 0.0023, Test Accuracy: 1.0
Epoch 39/50: Train Loss: 0.0024, Train Accuracy: 1.0000, Test Loss: 0.0020, Test Accuracy: 1.0
Epoch 40/50: Train Loss: 0.0021, Train Accuracy: 1.0000, Test Loss: 0.0018, Test Accuracy: 1.0
Epoch 41/50: Train Loss: 0.0018, Train Accuracy: 1.0000, Test Loss: 0.0015, Test Accuracy: 1.0
Epoch 42/50: Train Loss: 0.0015, Train Accuracy: 1.0000, Test Loss: 0.0014, Test Accuracy: 1.0
Epoch 43/50: Train Loss: 0.0014, Train Accuracy: 1.0000, Test Loss: 0.0012, Test Accuracy: 1.0
Epoch 44/50: Train Loss: 0.0012, Train Accuracy: 1.0000, Test Loss: 0.0011, Test Accuracy: 1.0
Epoch 45/50: Train Loss: 0.0011, Train Accuracy: 1.0000, Test Loss: 0.0009, Test Accuracy: 1.0
Epoch 46/50: Train Loss: 0.0009, Train Accuracy: 1.0000, Test Loss: 0.0008, Test Accuracy: 1.0
Epoch 47/50: Train Loss: 0.0009, Train Accuracy: 1.0000, Test Loss: 0.0007, Test Accuracy: 1.0
Epoch 48/50: Train Loss: 0.0008, Train Accuracy: 1.0000, Test Loss: 0.0007, Test Accuracy: 1.0
Epoch 49/50: Train Loss: 0.0007, Train Accuracy: 1.0000, Test Loss: 0.0006, Test Accuracy: 1.0
Epoch 50/50: Train Loss: 0.0007, Train Accuracy: 1.0000, Test Loss: 0.0005, Test Accuracy: 1.0
```



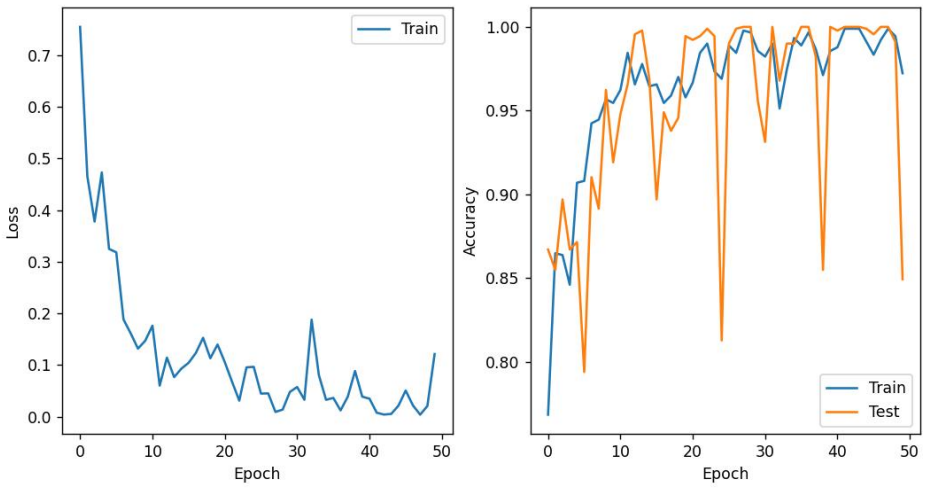
2、手动实现的 ResNet：

```
Epoch 38/50: Train Loss: 0.5914, Train Accuracy: 0.7616, Test Loss: 0.6062, Test Accuracy: 0.7
Epoch 39/50: Train Loss: 0.5453, Train Accuracy: 0.7960, Test Loss: 0.5998, Test Accuracy: 0.7
Epoch 40/50: Train Loss: 0.5419, Train Accuracy: 0.7805, Test Loss: 0.4748, Test Accuracy: 0.8
Epoch 41/50: Train Loss: 0.5552, Train Accuracy: 0.7894, Test Loss: 0.6954, Test Accuracy: 0.7
Epoch 42/50: Train Loss: 0.5532, Train Accuracy: 0.7860, Test Loss: 0.4441, Test Accuracy: 0.8
Epoch 43/50: Train Loss: 0.5113, Train Accuracy: 0.7971, Test Loss: 0.4170, Test Accuracy: 0.8
Epoch 44/50: Train Loss: 0.4868, Train Accuracy: 0.8004, Test Loss: 0.4359, Test Accuracy: 0.8
Epoch 45/50: Train Loss: 0.4901, Train Accuracy: 0.8093, Test Loss: 0.4870, Test Accuracy: 0.8
Epoch 46/50: Train Loss: 0.4689, Train Accuracy: 0.8248, Test Loss: 0.3702, Test Accuracy: 0.9
Epoch 47/50: Train Loss: 0.5028, Train Accuracy: 0.8337, Test Loss: 0.5620, Test Accuracy: 0.8
Epoch 48/50: Train Loss: 0.4533, Train Accuracy: 0.8215, Test Loss: 0.4118, Test Accuracy: 0.8
Epoch 49/50: Train Loss: 0.5302, Train Accuracy: 0.8115, Test Loss: 0.5638, Test Accuracy: 0.8
Epoch 50/50: Train Loss: 0.5448, Train Accuracy: 0.7816, Test Loss: 0.4841, Test Accuracy: 0.8
```



3、自带的 ResNet18：

```
Epoch 38/50: Train Loss: 0.0387, Train Accuracy: 0.9867, Test Loss: 0.0796, Test Accuracy: 1.0
Epoch 39/50: Train Loss: 0.0883, Train Accuracy: 0.9712, Test Loss: 0.4446, Test Accuracy: 0.9
Epoch 40/50: Train Loss: 0.0387, Train Accuracy: 0.9856, Test Loss: 0.0021, Test Accuracy: 1.0
Epoch 41/50: Train Loss: 0.0348, Train Accuracy: 0.9878, Test Loss: 0.0118, Test Accuracy: 1.0
Epoch 42/50: Train Loss: 0.0076, Train Accuracy: 0.9989, Test Loss: 0.0008, Test Accuracy: 1.0
Epoch 43/50: Train Loss: 0.0040, Train Accuracy: 0.9989, Test Loss: 0.0003, Test Accuracy: 1.0
Epoch 44/50: Train Loss: 0.0053, Train Accuracy: 0.9989, Test Loss: 0.0005, Test Accuracy: 1.0
Epoch 45/50: Train Loss: 0.0212, Train Accuracy: 0.9911, Test Loss: 0.0053, Test Accuracy: 1.0
Epoch 46/50: Train Loss: 0.0505, Train Accuracy: 0.9834, Test Loss: 0.0198, Test Accuracy: 1.0
Epoch 47/50: Train Loss: 0.0219, Train Accuracy: 0.9922, Test Loss: 0.0009, Test Accuracy: 1.0
Epoch 48/50: Train Loss: 0.0038, Train Accuracy: 0.9989, Test Loss: 0.0006, Test Accuracy: 1.0
Epoch 49/50: Train Loss: 0.0206, Train Accuracy: 0.9945, Test Loss: 0.0248, Test Accuracy: 1.0
Epoch 50/50: Train Loss: 0.1212, Train Accuracy: 0.9723, Test Loss: 0.7391, Test Accuracy: 0.8
```



可以看到三种模型中，第一种的损失函数下降较为平滑，波动较小；第三种的损失函数下降较快，在第十次迭代时基本降到了 0.1 附近。第一种的训练及测试的准确率呈现稳步上升的趋势，大约在第 32 次迭代之后几乎稳定在 100%；第二种的准确率不但偏低且波动极大；第三种的准确率从第一次迭代起便处于较高的指标，总体几乎控制在 80%~100%，但是相比之下波动较大。

四、 参考资料

https://blog.csdn.net/CltCj/article/details/120060543?ops_request_misc=%257B%2522request%255Fid%2522%253A%252216870505341680018858515 6%2522%252C%2522scm%2522%253A%252220140713.130102334..%2522%257D&request_id=168705053416800188585156&biz_id=0&utm_medium=distribute.pc_search_result.none-task-blog-2~all~top_positive~default-1-120060543-null-null.142^v88^control_2,239^v2^insert_chatgpt&utm_term=pytorch&spm=1018.2226.3001.4187