Project1第二部分实验报告

22307130158 吴优

实验内容

使用卷积神经网络实现12个手写汉字分类

整体架构

1. **定义 CNN 模型**：建立卷积神经网络模型，包括卷积层、池化层和全连接层。
2. **自定义数据集类**：`CustomDataset` 自定义数据集的加载和预处理。
3. **主要执行**：
    - **数据预处理**：`transforms.Compose` 组合了数据转换操作，包括转换为灰度图像和转换为张量。
    - **构建数据集**：`datasets.ImageFolder` 加载数据集，然后将数据划分为训练集和测试集。
    - **数据加载器**：`DataLoader` 创建了用于训练和测试的数据加载器。
    - **模型初始化**：初始化了 `CNN` 模型。
    - **定义损失函数和优化器**：使用交叉熵损失和 `Adam` 优化器。
    - **训练与测试**：遍历训练集进行训练，使用测试集进行测试。

代码分析

cnn类

- init初始化函数

```
    def __init__(self):
        # init初始化函数
        super(CNN, self).__init__()  # 调用父类 `nn.Module` 的构造函数，确保正确地初始化
        self.conv1 = nn.Conv2d(in_channels=1, out_channels=32, kernel_size=3, padding=1)  #
卷积层
        # 输入通道为1，输出通道为32，卷积核大小为3x3，填充为1，以保持输入输出大小一致
        self.conv2 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, padding=1)
# 另一个卷积层
        # 输入通道为上一层的输出通道数（即32），输出通道数为64，其余参数与第一个卷积层相同
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)  # 最大池化层
        # 池化核大小为2x2，步长为2，这将图像尺寸减小一半
        self.fc1 = nn.Linear(64 * 7 * 7, 128)  # 全连接层
        # 输入大小为64x7x7（经过两次池化操作后的大小），输出大小为128
        self.fc2 = nn.Linear(128, 12)  # 另一个全连接层
        # 输入大小为128，输出大小为12，表示网络最终输出的类别数量为12
```

- 前向传播forward函数：

```
    def forward(self, x):
        x = self.pool(torch.relu(self.conv1(x)))
        x = self.pool(torch.relu(self.conv2(x)))
        x = x.view(-1, 64 * 7 * 7)
        x = torch.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

将输入 x 通过第一个卷积层，然后应用 ReLU 激活函数，最后通过最大池化层。将上一层的输出再次通过另一个卷积层、ReLU激活函数和最大池化层。将上一层的输出展平成一维向量，以便输入到全连接层。通过第一个全连接层，并应用 ReLU 激活函数。通过最后一个全连接层，得到最终的输出。

数据集类CustomDataset

```
class CustomDataset(torch.utils.data.Dataset):
    def __init__(self, data, transform=None):
        self.data = data
        self.transform = transform
    def __len__(self):
        return len(self.data)
    def __getitem__(self, idx):
        img_path, label = self.data[idx]
        img = Image.open(img_path)
        if self.transform:
            img = self.transform(img)
        return img, label
```

1. 构造函数 data：包含图像路径和对应标签的数据列表 transform：用于指定数据转换操作，图像预处理操作。不提供则默认为 None。
2. __len__(self)：返回数据集的长度，即数据样本的数量。
3. __getitem__(self, idx)：用于根据给定索引 idx 获取数据集中样本。从 self.data 中获取索引为 idx 的图像路径 img_path 和标签 label。使用 Image.open(img_path) 打开图像文件，将其加载为 PIL 图像对象。最后返回处理后的图像数据和对应的标签。

数据的预处理：

```
if __name__ == '__main__':
    # 数据预处理
    transform = transforms.Compose([
        transforms.Grayscale(),  # 转换为灰度图像
        transforms.ToTensor()    # 转换为Tensor
    ])

    # 数据集根目录
    root_dir = 'train_data/train'
```

```
    # 构建数据集
    all_data = datasets.ImageFolder(root=root_dir, transform=transform)

    # 获取所有类别及其对应的文件夹路径
    class_folders = all_data.class_to_idx.items()
    # 打乱类别顺序

    # 划分训练集和测试集
    train_data = []
    test_data = []

    for class_name, class_folder in class_folders:
        class_path = os.path.join(root_dir, class_name)
        images = [img_name for img_name in os.listdir(class_path) if
img_name.endswith('.bmp')]
        random.shuffle(images)

        num_images = len(images)
        num_train = int(0.8 * num_images)

        # 将数据划分为训练集和测试集
        for i, img_name in enumerate(images):
            img_path = os.path.join(class_path, img_name)
            img_data = (img_path, class_folder)
            if i < num_train:
                train_data.append(img_data)
            else:
                test_data.append(img_data)

    # 创建训练集和测试集数据加载器
    train_loader = torch.utils.data.DataLoader(
        CustomDataset(train_data, transform=transform),
        batch_size=4, shuffle=True, num_workers=2
    )

    test_loader = torch.utils.data.DataLoader(
        CustomDataset(test_data, transform=transform),
        batch_size=4, shuffle=False, num_workers=2
    )
```

对图像数据进行预处理，包括转换为灰度图像和转换为 `Tensor` 格式。加载图像数据集，并构建数据集对象 `all_data`，使用 `datasets.ImageFolder` 类来处理文件夹结构的数据集。获取数据集中所有类别及其对应的文件夹路径。将数据集划分为训练集和测试集，其中80%的数据用于训练，20%的数据用于测试。创建训练集和测试集的数据加载器，使用 `torch.utils.data.DataLoader` 类加载自定义数据集 `CustomDataset` 对象，并指定批量大小、是否打乱数据和多线程加载数据的工作进程数量。

训练以及测试

```
model = CNN()
```

```
    # 定义损失函数和优化器
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters(), lr=0.0001)

    # 训练模型
    num_epochs = 20
    for epoch in range(num_epochs):
        running_loss = 0.0
        for i, data in enumerate(train_loader, 0):
            inputs, labels = data
            optimizer.zero_grad()
            outputs = model(inputs)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()
            running_loss += loss.item()
            if i % 100 == 99:
                print('[%d, %5d] loss: %.7f' %
                      (epoch + 1, i + 1, running_loss / 100))
                running_loss = 0.0

    print('Finished Training')

    # 测试模型
    correct = 0
    total = 0
    with torch.no_grad():
        for data in test_loader:
            images, labels = data
            outputs = model(images)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

    print('Accuracy of the network on the test images: %.5f' % (
            100 * correct / total))
```

定义了损失函数 `criterion`，这里使用的是交叉熵损失函数（`nn.CrossEntropyLoss`），适用于多分类任务。定义优化器 `optimizer`，这里使用 `Adam` 优化器（`optim.Adam`）对模型参数进行优化，学习率设为 `0.0001`。开始训练模型，循环 `num_epochs` 次，每次循环遍历训练数据集 `train_loader` 将优化器的梯度清零。数据输入模型，得到输出，计算预测值与真实标签之间的损失，反向传播，更新模型参数。 记录当前损失值并打印，每处理 `100` 个 `mini-batch` 打印一次 在测试阶段，用测试数据集 `test_loader` 进行模型测试

对于cnn的理解

（1）卷积层的作用是对输入数据进行卷积操作，一个卷积核就是一个窗口滤波器，在网络训练过程中，使用自定义大小的卷积核作为一个滑动窗口对输入数据进行卷积。卷积运算的目的是提取输入的不同特征，底层的只能提取到最简单的特征，随着层数变深，提取的特征逐渐具体与清楚。 偏置向量的作用是对卷积后的数据进行简单线性的加法，就是卷积后的数据加上偏置向量中的数据 （2）Relu函数是一个线性函数，即 `f(x)=max(0,x)`，由于经过Relu函数激活后的数据0值一下都变成0，所以为了尽可能的降低损失，卷积层的后面加上一个偏置向量，对数据进行一次简单的线性加法，使得数据的值产生一个横向的偏移，避免被激活函数过

滤掉更多的信息。 （3）池化层，通常在卷积层之后会得到维度很大的特征，将特征切成几个区域，取其最大值，得到新的、维度较小的特征。池化的过程也是一个移动窗口在输入矩阵上滑动，滑动过程中去这个窗口中数据矩阵上最大值作为输出，池化层的大小一般为2*2，步长为1。 池化层夹在连续的卷积层中间， 用于压缩数据和参数的量，减小过拟合。池化层的作用是对数据进行降维处理，把冗余信息去除，把最重要的特征抽取出来，这也是池化操作作用。在一定程度上防止过拟合，更方便优化。 （4）全连接层，把所有局部特征结合变成全局特征，用来计算最后每一类的得分。全连接层在分类问题中用作网络的最后层，作用主要为将数据矩阵进行全连接，然后按照分类数量输出数据

网络结构的调整

主要调整以下部分的参数

```
self.conv1 = nn.Conv2d(in_channels=1, out_channels=32, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, padding=1)
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)
        self.fc1 = nn.Linear(64 * 7 * 7, 128)
        self.fc2 = nn.Linear(128, 12)  # 输出12个类别
```

首先是conv1与conv2的参数调整，主要调整conv1的输出通道数、conv2的输出通道数以及两者的卷积核大小。

conv1Outputchannel	conv2Outputchannel	kernel_size	正确率
16	32	3*3	97.5~98
32	64	3*3	98.5~99
32	64	5*5	98.5~99

分析：

1. 输出通道大小的设置 在卷积层中，输出通道决定了该层提取的特征数量。增加输出通道可以增加模型的表示能力，因为每个输出通道可以学习到不同的特征表示。减少输出通道可以降低模型的复杂度和参数数量，减少过拟合的风险。但在另一方面，减少输出通道可能会限制网络学习到足够丰富的特征表示，从而降低模型的性能。
2. 卷积核大小的设置 增大卷积核可以覆盖更大的局部区域，使网络能够捕获更广泛的上下文信息，有助于提取更全局的特征。但也会增加每个卷积操作的计算量，导致网络的计算成本增加，尤其是在处理大型图像时。 在另一方面，较大的卷积核可能会导致信息丢失或模糊化，特别是当感受野超过目标对象的尺度时，网络可能会捕捉到过多的背景信息而忽略目标细节。增大卷积核还可能增加模型的参数数量，增加过拟合的风险。

防止过拟合

dropout方法 从隐藏层神经元中随机选择一个一半大小的子集临时删除掉。对一批训练样本，先前向传播然后反向传播损失，梯度下降法更新参数。不断重复这一过程这样可以使得神经网络泛化性更强，防止过于依赖于局部的特征。