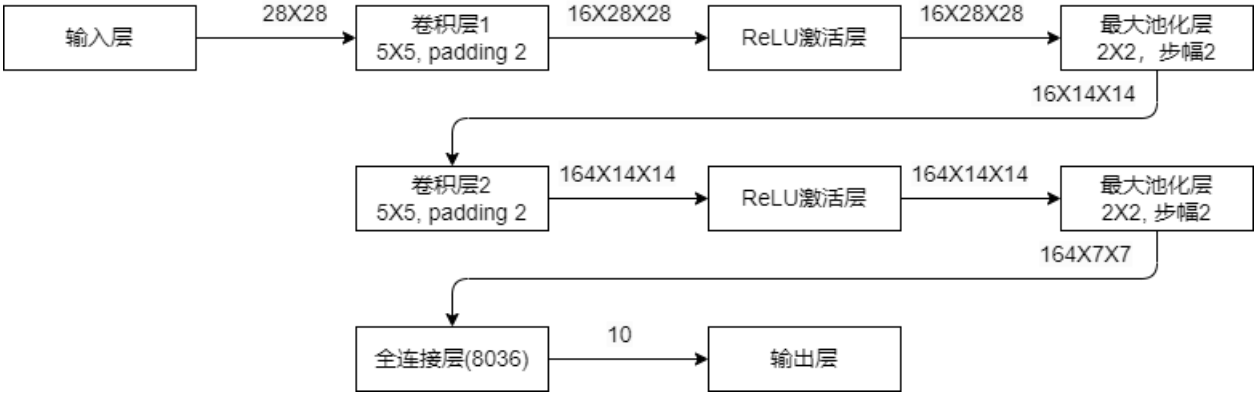
Final Project Report

200111524-龙浩然

2023/6/24

模型结构

本次实验，我采用LeNet的基本结构，在MNIST数据集上完成实验。



模型最大的特点是使用了双层卷积层来进行特征提取，双层结构中每一层都包含**卷积层+ReLU激活层+池化层。**
两个卷积层均使用5×5卷积核，$padding = 2, step = 1$以尽可能的保证卷积层不会对图像大小做出改变，方便后续池化层的计算（需要偶数维度）；
ReLU激活层均一致，使用pytorch提供的ReLU函数；两次池化层是同一个，均为$2 \times 2, step = 2$的最大池化层，会将图像的长宽缩小为一半$(28 \times 28 \rightarrow 14 \times 14 \rightarrow 7 \times 7)$。
最后是全连接层，首先要按照常规操作：将上面输出的向量展开，成为一维向量，然后输入全连接层中，进行计算，输出维度为10的结果（10分类器）
标准的LeNet结构实际上最后拥有多个全连接层，但是经过我的测试，针对MNIST数据集分类，**多层全连接层效果实际上和一层的效果几乎一致**，所以我将全连接层简化为一层，仍然能够达到不错的准确率。
代码如下：

```
class Net(nn.Module):
    conv_channel = 164  # 定义卷积层的输出通道数
    def __init__(self, classes=10):
        """定义初始化函数，classes为分类数，默认为10"""
        super(Net, self).__init__()  # 调用nn.Module的初始化函数
        self.classes = classes  # 将类别数保存在self.classes中
        # 第一个卷积层
        self.conv1 = nn.Conv2d(1, 16, kernel_size=5, stride=1, padding=2)
        # 第二个卷积层
        self.conv2 = nn.Conv2d(16, self.conv_channel, kernel_size=5, stride=1,
padding=2)
        # 定义一个平均池化层，池化核大小为2，步长为2
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)
        # # 定义全连接层，输入节点数为164*7*7，输出节点数为分类数
        self.fc = nn.Linear(self.conv_channel*7*7, self.classes)
    def forward(self, x):
        """前向传播函数"""
        # 对输入x进行第一次卷积，然后使用ReLU激活函数，再进行最大池化
        x = self.pool(nn.functional.relu(self.conv1(x)))
        # 对x进行第二次卷积，然后使用ReLU激活函数，再进行最大池化
        x = self.pool(nn.functional.relu(self.conv2(x)))
```

```
        # 将x展开成一维向量
        x = x.view(-1, self.conv_channel*7*7)
        # 对x使用ReLU激活函数，然后进行全连接层的计算
        x = self.fc((x))
        return x  # 返回输出结果
```

超参选择

模型内部参数：

参数名称	数值	原因
卷积层核大小	5x5	经过多次测试，发现5x5kernel的效果比7x7和3x3相对好
卷积层1输出通道	16	经过多次测试，卷积层1的输出通道数太多对性能损耗大，而且对准确率贡献不明显
卷积层2输出通道	164	经过多次测试，卷积层2的输出通道数与准确率上限上高度相关，所以在可接受范围内尽可能提高
池化核大小	2x2	标准的池化核大小，增大会导致图片维度变得太小；减小又会导致没有意义
池化层步长	2	标准的池化层步长，可以降低图片维度
全连接层输出节点	10	与分类数相关，本次项目分类10个数字，所以输出节点为10

其他参数：

参数名称	数值	原因
batch_size：batch样本数	32	随便设的
epoch：迭代次数	2	1次其实就够了，但是2次才能勉强达到99%
lr：学习率	0.0008	经测试0.0008效果较好

训练过程

训练前需要加载测试集，对训练数据、测试数据进行归一化、数据增强，对训练集进行打乱

```
def data_loader():
    """load MNIST dataset\n
    对数据进行预处理:归一化、数据增强、打乱数据"""
    # 定义数据转换，将图像数据转换为tensor，并将像素值归一化
    # 利用transform模块进行数据增强，包括随机水平翻转、随机竖直翻转、随机旋转、随机裁剪等
    transform = transforms.Compose(
    [transforms.ToTensor(),transforms.Normalize((0.5,), (0.5,))])
    # 加载训练数据集，并对数据进行预处理
    trainset = torchvision.datasets.MNIST(root='./data', train=True,download=True, transform=transform)
    # 定义训练数据集的数据加载器，batch_size为每个batch的样本数，shuffle为是否打乱数据顺序
    train_loader = torch.utils.data.DataLoader(trainset,
```

```
batch_size=32,shuffle=True, num_workers=2)
    # 加载测试数据集，并对数据进行预处理
    testset = torchvision.datasets.MNIST(root='./data',
train=False,download=True, transform=transform)
    # 定义测试数据集的数据加载器，batch_size为每个batch的样本数，shuffle为是否打乱数
据顺序
    test_loader = torch.utils.data.DataLoader(testset,
batch_size=32,shuffle=False, num_workers=2)
    return train_loader, test_loader
```

然后需要设置损失函数与更新函数，获取cuda：

```
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.Adam(net.parameters(), lr=0.0008)
    device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
```

最后进行训练：

```
def train(train_loader, net, criterion, optimizer, device):
    """train the model, print loss and accuracy every 200 mini-batches"""
    net.to(device)
    # loop over the dataset
    for epoch in range(2):
        running_loss = 0.0
        correct = 0
        total = 0
        for i, data in enumerate(train_loader, 0):
            inputs, labels = data
            inputs, labels = inputs.to(device), labels.to(device)
            # zero the parameter gradients
            optimizer.zero_grad()
            outputs = net(inputs)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct += (predicted == labels).sum().item()
            loss = criterion(outputs, labels)
            loss.backward()
            # update the parameters
            optimizer.step()
            running_loss += loss.item()
            if i % 200 == 199:
                print('\t[%d][%5d] loss: %.3f accuracy: %.3f' % (epoch+1, i +
1, running_loss / 100, 100 * correct / total))
                running_loss = 0.0
                correct = 0
                total = 0
```

训练结果如下：

```
[NOTICE] Start training
        [1][  200] loss: 0.841 accuracy: 87.641
        [1][  400] loss: 0.262 accuracy: 95.984
        [1][  600] loss: 0.166 accuracy: 97.656
        [1][  800] loss: 0.156 accuracy: 97.500
        [1][ 1000] loss: 0.129 accuracy: 97.984
        [1][ 1200] loss: 0.117 accuracy: 98.266
        [1][ 1400] loss: 0.108 accuracy: 98.172
        [1][ 1600] loss: 0.114 accuracy: 98.172
        [1][ 1800] loss: 0.130 accuracy: 98.109
        [2][  200] loss: 0.062 accuracy: 99.125
        [2][ 1800] loss: 0.071 accuracy: 98.953
[NOTICE] Finish train
```

性能

使用测试集与测试函数进行测试：

```
def test(test_loader, net, device):
    """test the model, print accuracy of each class and total accuracy"""
    correct_hole = 0
    correct_list = [0] * 10
    hole_list = [0] * 10
    total = 0
    net.to(device)
    with torch.no_grad():
        for data in test_loader:
            images, labels = data
            images, labels = images.to(device), labels.to(device)
            outputs = net(images)
            _, predicted = torch.max(outputs.data, 1)
            total += labels.size(0)
            correct_hole += (predicted == labels).sum().item()
            for i in range(len(labels)):
                hole_list[labels[i]] += 1
                if predicted[i] == labels[i]:
                    correct_list[labels[i]] += 1
    print('\tAccuracy of the network on the 10000 test images: %.3f %%' % (100
* correct_hole / total))
    for i in range(10):
        print('\tAccuracy of %d: %.3f %%' % (i, 100 * correct_list[i] /
hole_list[i]))
```

测试结果如下：

```
[NOTICE] Start testing
        Accuracy of the network on the 10000 test images: 99.080 %
        Accuracy of 0: 99.592 %
        Accuracy of 1: 99.207 %
        Accuracy of 2: 99.322 %
        Accuracy of 3: 99.703 %
        Accuracy of 4: 99.796 %
        Accuracy of 5: 99.103 %
        Accuracy of 6: 98.956 %
        Accuracy of 7: 97.568 %
        Accuracy of 8: 99.589 %
        Accuracy of 9: 98.018 %
[NOTICE] Finish test
```

实际上经过我多次测试，最终测试准确率稳定在89.9%上下，整个训练时间（含加载数据、测试）稳定在40s左右