实验一 手写数字识别

一、 实验目的

1. 掌握卷积神经网络基本原理；
2. 掌握 PyTorch（或其他框架）的基本用法以及构建卷积网络的基本操作；
3. 了解 PyTorch（或其他框架）在 GPU 上的使用方法。

二、 实验要求

1. 搭建 PyTorch（或其他框架）环境；
2. 构建一个规范的卷积神经网络组织结构；
3. 在 MNIST 手写数字数据集上进行训练和评估，实现测试集准确率达到 98%及以上；
4. 按规定时间在课程网站提交实验报告、代码以及 PPT。

三、 实验原理（以 PyTorch 为例）

1. PyTorch 基本用法：

使用 PyTorch，必须了解 PyTorch：

- 张量的创建与使用
- 数据创建和数据加载
- 数据增强
- 网络模型创建
- 使用 torch.autograd 自动求梯度
- 模型参数优化
- 模型加载与保存

PyTorch 的前身是 Torch，其底层和 Torch 框架一样，但是使用 Python 重新写了很多内容，不仅更加灵活，支持动态图，而且提供了 Python 接口。它是由 Torch7 团队开发，是一个以 Python 优先的深度学习框架，不仅能够实现强大的 GPU 加速，同时还支持动态神经网络。

2. 卷积神经网络：

典型的卷积神经网络由卷积层、池化层、激活函数层交替组合构成，因此可将其视为一种层次模型，形象地体现了深度学习中“深度”之所在。

- 卷积操作

卷积运算是卷积神经网络的核心操作，给定二维的图像 I 作为输入，二维卷积核 K， 卷积运算可表示为：

$$S(i,j)=(I*K)(i,j)=\sum_m\sum_n I(i+m,j+n)K(m,n) \tag{1}$$

给定 5×5 输入矩阵、3×3 卷积核，相应的卷积操作如图 1 所示。

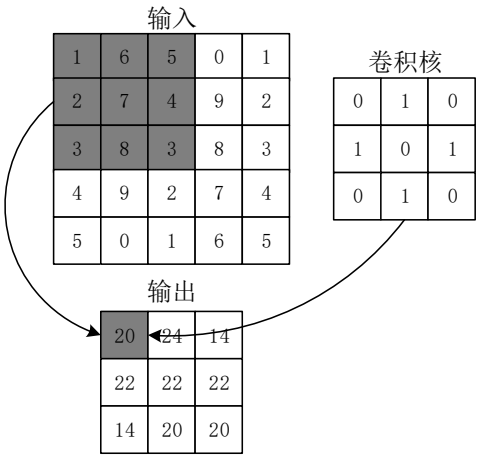

图 1 卷积运算

在使用 TensorFlow 等深度学习框架时，卷积层会有 padding 参数，常用的有两种选择，一个是“valid”，一个是“same”。前者是不进行填充，后者则是进行数据填充并保证输出与输入具有相同尺寸。

构建卷积或池化神经网络时，卷积步长也是一个很重要的基本参数。它控制了每个操作在特征图上的执行间隔。

- 池化操作

池化操作使用某位置相邻输出的总体统计特征作为该位置的输出，常用最大池化（max-pooling）和均值池化（average-pooling）。池化层不包含需要训练学习的参数，仅需指定池化操作的核大小、操作步长以及池化类型。池化操作示意如图 2 所示。

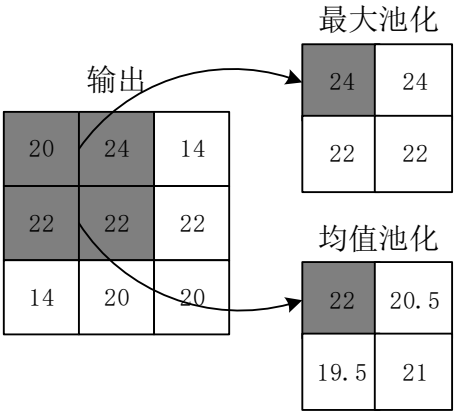

图 2 池化操作

- 激活函数层

卷积操作可视为对输入数值进行线性计算发挥线性映射的作用。激活函数的引入，则增强了深度网络的非线性表达能力，从而提高了模型的学习能力。常用的激活函数有 sigmoid、tanh 和 ReLU 函数。

四、 实验所用工具及数据集

1. 工具

Anaconda、PyTorch

（PyTorch 安装教程参考：PyTorch 官网：https://pytorch.org/）

2. 数据集

MNIST 手写数字数据集

（下载地址及相关介绍：http://yann.lecun.com/exdb/mnist/）

五、 实验步骤与方法（以 PyTorch 为例）

1）安装实验环境，包括 Anaconda、PyTorch，若使用 GPU 还需要安装 cuda、cudnn；

2）下载 MNIST 手写数字数据集；

3）加载 MNIST 数据；

```
# 下载mnist手写数据集
train_data = torchvision.datasets.MNIST(
    root='./data/',  # 保存或提取的位置，会放在当前文件夹中
    train=True,  # true说明是用于训练的数据，false说明是用于测试的数据
    transform=torchvision.transforms.ToTensor(),  # 转换PIL.Image or numpy.ndarray为Tensor
)

test_data = torchvision.datasets.MNIST(
    root='./data/',
    train=False  # 表明是测试集
)

# 批训练 50个samples,  1  channel, 28x28 (50,1,28,28)
# Torch中的DataLoader是用来包装数据的工具，它能帮我们有效迭代数据，这样就可以进行批训练
train_loader = Data.DataLoader(
    dataset=train_data,
    batch_size=BATCH_SIZE,
    shuffle=True  # 是否打乱数据
)

test_loader = Data.DataLoader(
    dataset=test_data,
    batch_size=BATCH_SIZE,
    shuffle=False  # 是否打乱数据
)
```

4）构建模型；

用 class 类来建立 CNN 模型

CNN 流程：　　卷积(Conv2d)-> 激励函数(ReLU)->池化(MaxPooling)->

卷积(Conv2d)-> 激励函数(ReLU)->池化(MaxPooling)->

展平多维的卷积成的特征图->接入全连接层(Linear)->输出

```
class CNN(nn.Module):  # 我们建立的CNN继承nn.Module这个模块
    def __init__(self):
        super(CNN, self).__init__()
        # 建立第一个卷积(Conv2d)-> 激励函数(ReLU)->池化(MaxPooling)
        self.conv1 = nn.Sequential(
            # 第一个卷积con2d
            nn.Conv2d(  # 输入图像大小(1,28,28)
                in_channels=1,  # 输入图片的高度，因为minist数据集是灰度图像只有一个通道
                out_channels=16,  # n_filters 卷积核的高度
                kernel_size=5,  # filter size 卷积核的大小 也就是长x宽=5x5
                stride=1,  # 步长
                padding=2,  # 想要con2d输出的图片长宽不变，就进行补零操作 padding = (kernel_size-1)/2
            ),  # 输出图像大小(16,28,28)
            # 激活函数
            nn.ReLU(),
            # 池化，下采样
            nn.MaxPool2d(kernel_size=2),  # 在2x2空间下采样
            # 输出图像大小(16,14,14)
        )
        # 建立第二个卷积(Conv2d)-> 激励函数(ReLU)->池化(MaxPooling)
        self.conv2 = nn.Sequential(
            # 输入图像大小(16,14,14)
            nn.Conv2d(  # 也可以直接简化写成nn.Conv2d(16,32,5,1,2)
                in_channels=16,
                out_channels=32,
                kernel_size=5,
                stride=1,
                padding=2
            ),
            # 输出图像大小 (32,14,14)
            nn.ReLU(),
            nn.MaxPool2d(2),
            # 输出图像大小(32,7,7)
        )
        # 建立全卷积连接层
        self.out = nn.Linear(32 * 7 * 7, 10)  # 输出是10个类

    # 下面定义x的传播路线
    def forward(self, x):
        x = self.conv1(x)  # x先通过conv1
        x = self.conv2(x)  # 再通过conv2
        # 把每一个批次的每一个输入都拉成一个维度，即(batch_size,32*7*7)
        # 因为pytorch里特征的形式是[bs,channel,h,w]，所以x.size(0)就是batchsize
        x = x.view(x.size(0), -1)  # view就是把x弄成batchsize行个tensor
        output = self.out(x)
        return output
```

5）创建优化器和损失函数；

```
# 优化器选择Adam
optimizer = torch.optim.Adam(cnn.parameters(), lr=LR)
# 损失函数
loss_func = nn.CrossEntropyLoss()  # 目标标签是one-hotted
```

6）训练和评估模型。

```
# 开始训练
for epoch in range(EPOCH):
    for step, (b_x, b_y) in enumerate(train_loader):  # 分配batch data
        output = cnn(b_x)  # 先将数据放到cnn中计算output
        loss = loss_func(output, b_y)  # 输出和真实标签的loss，二者位置不可颠倒
        optimizer.zero_grad()  # 清除之前学到的梯度的参数
        loss.backward()  # 反向传播，计算梯度
        optimizer.step()  # 应用梯度

        if step % 50 == 0:
            correct = total = 0
            with torch.no_grad():
                for _, (b_x, b_y) in enumerate(test_loader):
                    output = cnn(b_x)
                    pred_y = torch.max(output, 1)[1].detach().numpy()
                    correct += float((pred_y == b_y.detach().numpy()).astype(int).sum())
                    total += float(b_y.size(0))
            accuracy = correct / total
            print('Epoch: ', epoch, '| train loss: %.4f' % loss.data.numpy(), '| test accuracy: %.2f' % accuracy)

torch.save(cnn.state_dict(), 'cnn.pth')#保存模型
```