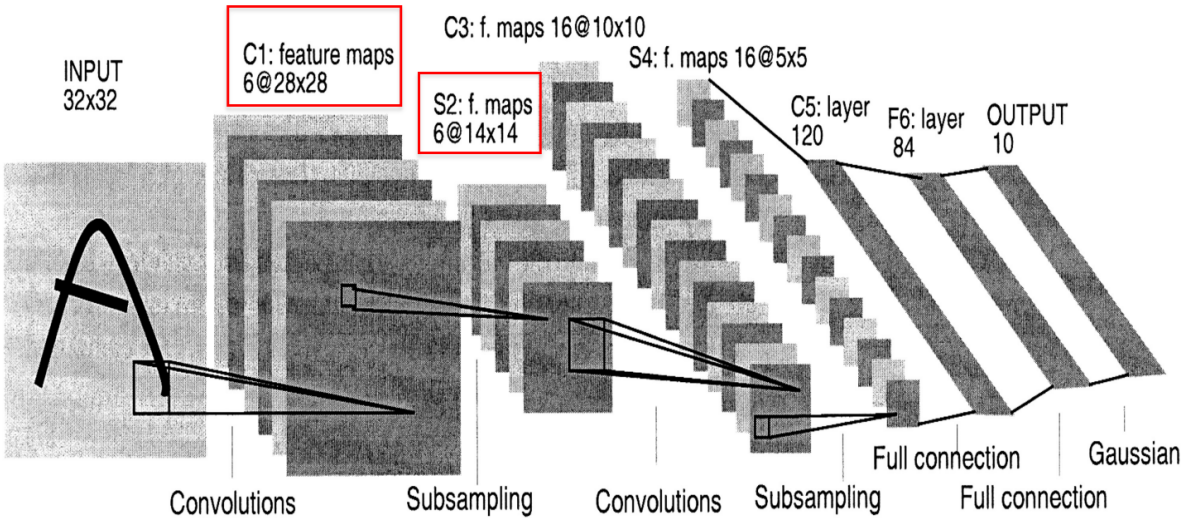
Part2实验报告

卷积神经网络

CNN框架结构与理解

相比part1的bp神经网络，卷积神经网络的结构更加复杂，它包括输入层、卷积层、激活层、池化层、全连接层、输出层，结构图如下：



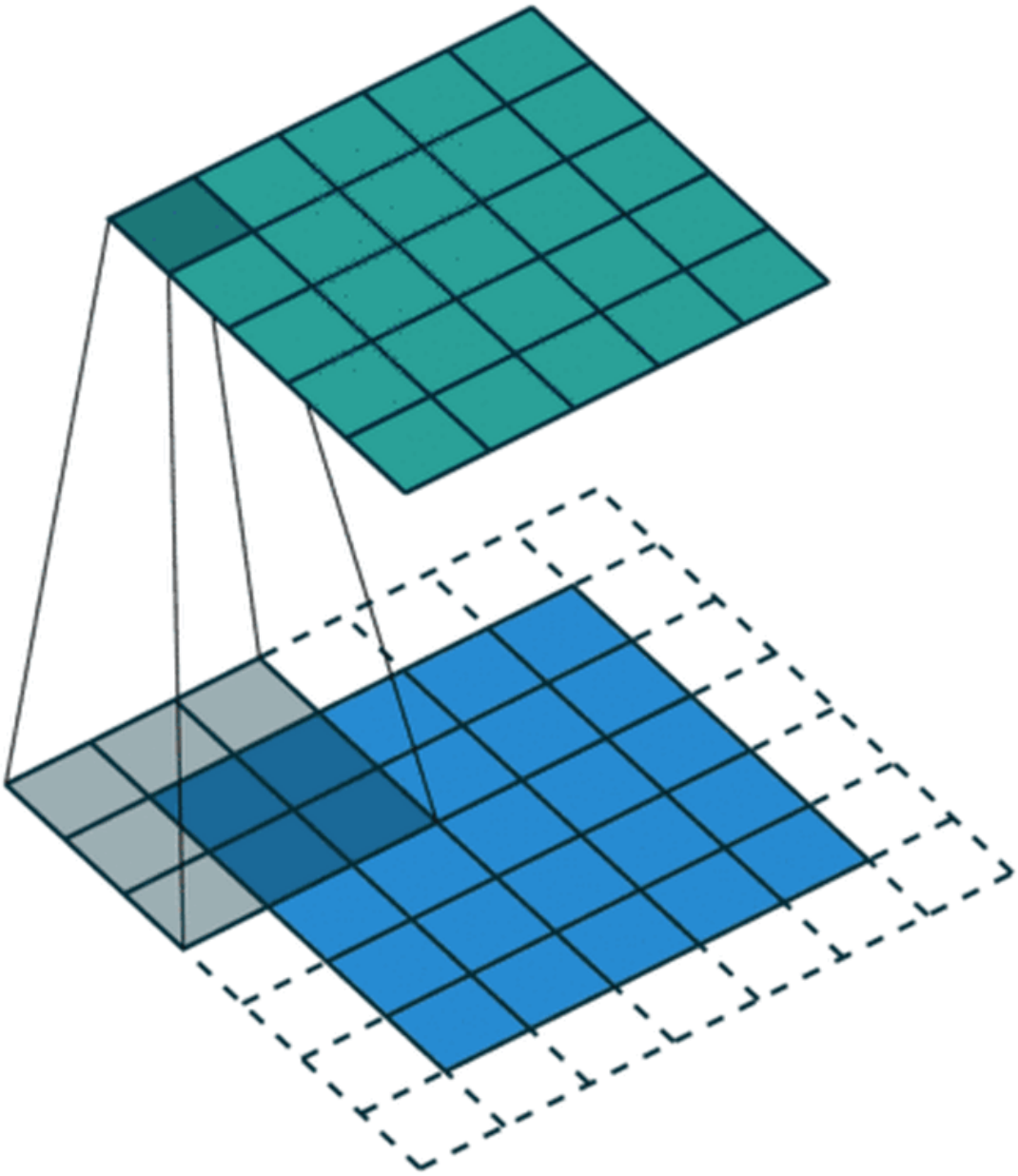
- 输入层

以上图为例，图片就是输入层，但是在输入之前，我们要对图像进行预处理，比如转化为灰度图、归一化，将图像转换为数字矩阵，以适应后续层的计算。

- 卷积层

先介绍**局部感知**：人的大脑识别图片的过程中，并不是一下子整张图同时识别，而是对于图片中的每一个特征首先局部感知，然后更高层次对局部进行综合操作，从而得到全局信息。

卷积层利用人脑的这种机制来处理图像，卷积层使用**卷积核**来进行局部感知，卷积核（kernel）是一个权重矩阵，卷积计算过程中，卷积核会按照一定的步长（stride）扫描输入矩阵，对照应的区域的数进行加权求和，从而提取局部特征，有时为了得到特定大小的卷积结果，需要对原矩阵边缘进行填充（padding），一般填充0。经过卷积操作之后，矩阵尺寸会变小，即模型参数会减少。卷积计算的示意图如下：



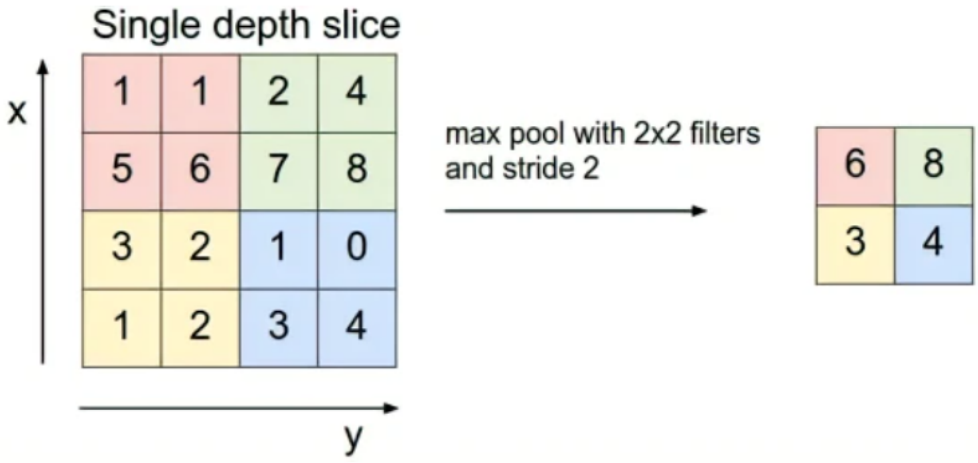
- 激励层

激励层做的事情本质上是对上一层的输出做一次非线性的映射，常用的激励函数有Relu、Sigmoid、Tanx等。那么为什么要有激励层？因为如果不用激励函数，那每一层的输出都是上一层输入的线性函数，这种情况下无论有多少神经网络层，输出都是输入的线性组合，和没有隐层的效果是一样的，所以要有激励层。

- 池化层

池化（Pooling）：也称为**欠采样**或**下采样。**主要用于特征降维，压缩数据和参数的数量，**减小过拟合**，同时提高模型的容错性。主要有：

- Max Pooling：最大池化（常用）
- Average Pooling：平均池化



通过池化层，原来的矩阵会变小，特征维度会降低。

- 全连接层

经过前面若干次卷积+激励+池化后，终于来到了全连接层，其实在全连接层之前，如果神经元数目过大，学习能力强，有可能出现过拟合。因此，可以引入dropout操作，来随机删除神经网络中的部分神经元，来解决此问题。还可以进行局部归一化（LRN）、数据增强等操作，来增加鲁棒性。全连接层部分，可以理解为一个简单的多分类神经网络（如：BP神经网络），通过softmax函数得到最终的输出。整个模型训练完毕。

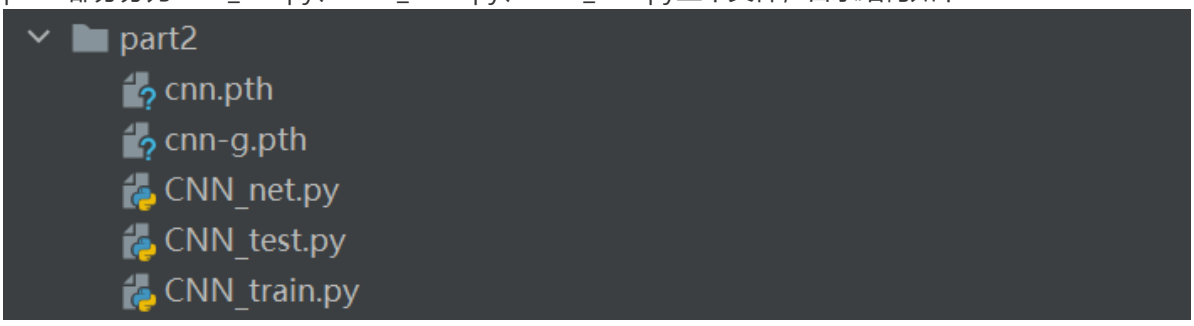
- 输出层

输出层是指全连接层的最后一层输出，对于多分类问题通过softmax函数得到最终的输出。

代码结构

项目目录结构

part2部分分为CNN_net.py、CNN_train.py、CNN_test.py三个文件，目录结构如下：



CNN网络类定义

CNN网络类定义在CNN_net.py中，包含网络结构定义和前向传播函数，代码如下：

```
class CNNNet(nn.Module):

    def __init__(self):
        super(CNNNet, self).__init__()        # 父类初始化
        # 输入图像是单通道，conv1 kenrnel size=5*5，输出通道 6
        self.conv1 = nn.Conv2d(in_channels=1, out_channels=16, kernel_size=3)
        # conv2 kernel size=5*5, 输出通道 16
        self.conv2 = nn.Conv2d(16, 32, 3)
        # 池化层
        self.pool = nn.MaxPool2d(2)
        # 激活层
        self.relu = nn.Relu()
```

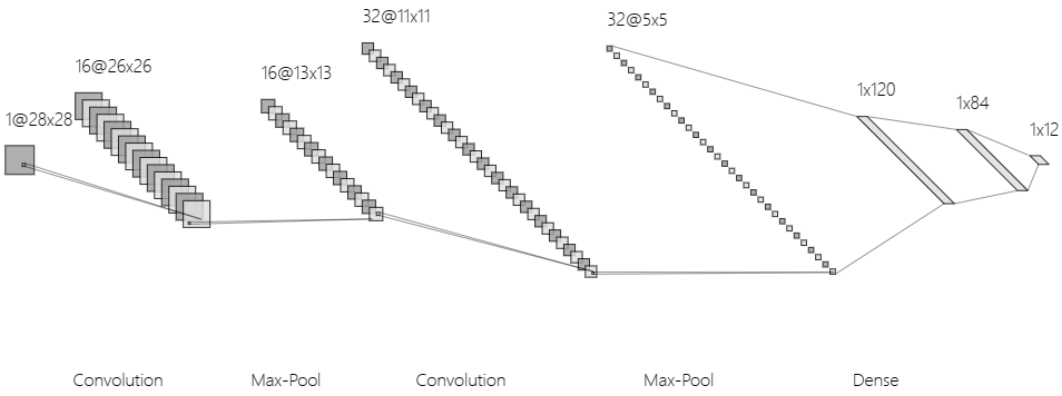
```
        # 全连接层
        self.fc1 = nn.Linear(32*5*5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 12)

    def forward(self, x):
        # max-pooling 采用一个 (2,2) 的滑动窗口
        x = F.max_pool2d(F.relu(self.conv1(x)), (2, 2))
        # 核(kernel)大小是方形的话，可仅定义一个数字，如 (2,2) 用 2 即可
        x = F.max_pool2d(F.relu(self.conv2(x)), 2)
        x = x.view(-1,self.num_flat_features(x))
        x = self.fc1(x)
        x = F.relu(x)
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

    def num_flat_features(self, x):
        # 除了 batch 维度外的所有维度
        size = x.size()[1:]
        num_features = 1
        for s in size:
            num_features *= s
        return num_features
```

在CNNNet的初始化函数中我定义了网络的层，包括卷积层、激活层、池化层和全连接层，然后在前向传播函数forward里面写每一层对数据的处理，在forward中，直接调用torch.nn.fuctional中的池化函数和激活函数更加方便。函数num_flat_features是为了将经过第二个卷积层，并经过激活和池化后的中间数据展开成数组，使得数据形状适应后面全连接层的计算。

我用绘图工具绘制了自己的CNN网络的结构图，其参数标注与实际参数一致，如下：



我的CNN网络结构图

数据集导入&初始化

这部分代码在CNN_train.py中

- 数据集的导入和数据初始化使用torchvision库，使用torchvision.datasets.ImageFolder读取训练数据，对数据的处理采取 **灰度图+归一化** 处理，使用torchvision.transforms的函数来实现。为了观察训练的效果，我把数据集按照 `9：1` 的比例分割成**训练集**和**验证集**代码如下：

```
data_transforms = {
        'train': transforms.Compose([
            transforms.Grayscale(1),
            transforms.ToTensor(),
            transforms.Normalize((0.5),(0.5))    # 归一化，加快收敛速度
        ]),
        'val': transforms.Compose([
            transforms.Grayscale(1),
            transforms.ToTensor(),
            transforms.Normalize((0.5),(0.5))    # 归一化，加快收敛速度
        ])
    }

    dataset =
torchvision.datasets.ImageFolder("../train/train",transform=data_transforms["tra
in"])  # 读取数据集


    # 划分训练集和验证集合
    train_dataset,val_dataset = data.random_split(dataset,
[int(train_ratio*len(dataset)),len(dataset)-int(train_ratio*len(dataset))])
    train_loader = data.DataLoader(dataset=train_dataset, batch_size=BATCH_SIZE,
shuffle=True)
    val_loader =
data.DataLoader(dataset=val_dataset,batch_size=BATCH_SIZE,shuffle=False)
```

训练过程

这部分代码在CNN_train.py中

训练中的几个重要参数（可方便地调整）：

- EPOCH：训练的轮数
- LR：学习率
- BATCH_SIZE：数据批量的大小

训练过程中优化器选用Adam，损失函数选用CE损失函数。对于每个轮次，对训练集的数据按照batch的大小批量进行前向传播，计算损失并反向传播更新参数，代码如下：

```
    cnnnet = CNNNet()
    print("网络结构如下：\n"+cnnnet)
    optimizer = optim.Adam(cnnnet.parameters(),lr = LR)
    criterion = nn.CrossEntropyLoss()   # CE Loss

    start = time.time()    # 记录训练开始的时间
    for epoch in range(EPOCH):
        cnnnet.train()
        for i, x_y in tqdm(list(enumerate(train_loader))):
            batch_x = Variable(x_y[0])
            batch_y = Variable(x_y[1])
            output = cnnnet.forward(batch_x)
            loss = criterion(output, batch_y)
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
        print('Train Epoch: {} Loss: {:.6f}'.format(epoch, loss.item()))
```

```
    print('\nFinished Training! Total cost time: ', time.time()-start)
    torch.save(cnnnet,"cnn-g.pth")
```

验证部分

这部分在CNN_test.py中。

验证部分首先打开训练时保存的模型，然后读取数据集（和上面训练过程中的读取一样，只是把全部数据都当成验证集合），然后把数据输入模型做预测，并计算损失和准确率，和训练过程相比只是没有反向传播：

```
    model = torch.load("cnn-g.pth")
    BATCH_SIZE = 100
    data_transforms = {
        'val': transforms.Compose([
            transforms.Grayscale(1),
            transforms.ToTensor(),
            transforms.Normalize((0.5),(0.5))    # 归一化，加快收敛速度
        ])
    }
    valid_set =
torchvision.datasets.ImageFolder("../train/train",transform=data_transforms["val
"])  # 读取数据集
    train_ratio = 0   # 全是验证集
    val_loader =
data.DataLoader(dataset=valid_set,batch_size=BATCH_SIZE,shuffle=False)

    loss_func = nn.CrossEntropyLoss()
    model.eval()
    val_loss = 0
    correct = 0
    with torch.no_grad():
        for data, target in val_loader:
            output = model.forward(data)
            loss = loss_func(output, target)
            val_loss += loss.item()  # sum up batch loss
            pred = output.argmax(dim=1, keepdim=True)  # get the index of the
max log-probability
            correct += pred.eq(target.view_as(pred)).sum().item()
            val_loss /= len(val_loader.dataset)
    print('\nval set: Average loss: {:.8f}, Accuracy: {}/{} ({:.0f}%)\n'.format(
        val_loss, correct, len(val_loader.dataset),
        100. * correct / len(val_loader.dataset)))
```

优化方法

归一化：加快收敛速度

在导入数据的时候，我们先将图片转为灰度图，然后调用torchvision.transforms.ToTensor将像素值映射到0-1（除255），然后我们采用transforms.Normalize((0.5),(0.5)) 来把数据归一化，参数分别表示均值和标准差，通过归一化的处理，**可以提高参数收敛速度**，而且使用了归一化以后，**准确率也有一定提高，从96%提高到了98%**，在我自己分出来的验证集上大概多识别正确将近20个图片。

批量计算修改参数

训练过程中我们选择训练批量数据（BATCH_SIZE个）修改一次参数，这样可以**提高收敛速度**。如果每次训练都修改一次的话，可能会在最优点附近晃来晃去，得不到最优点。两次参数的更新也有可能互相抵消掉，造成目标函数震荡的比较剧烈。而使用**小批梯度下降（mini-batch gradient decent）**的方法，把数据分为若干个批，按批来更新参数，这样，一个批中的一组数据共同决定了本次梯度的方向，下降起来就不容易跑偏，减少了随机性，收敛加快。另一方面因为批的样本数与整个数据集相比小了很多，计算量也不是很大，计算减慢的影响较小。

使用较小的卷积核

实验中发现**使用3×3的卷积核效果好于5×5的卷积核**，我觉得原因可能是我们的图片本身比较小（28×28），而且汉字是黑色的，其余是白色的，边缘部分像素差距较大，用小一点的卷积核更能捕捉边缘特征。

训练结果

在训练过程中，我自己按`9：1`的比例把训练数据分成了训练集和验证集合，在训练过程中，每个epoch会打印损失loss，训练结束会在自己分出的验证集上验证。

每个epoch的损失变化情况如下：

```
100%|██████████| 67/67 [00:01<00:00, 41.95it/s]
Train Epoch: 0 Loss: 0.501605
100%|██████████| 67/67 [00:02<00:00, 32.15it/s]
Train Epoch: 1 Loss: 0.170968
100%|██████████| 67/67 [00:02<00:00, 32.06it/s]
Train Epoch: 2 Loss: 0.180579
100%|██████████| 67/67 [00:02<00:00, 33.40it/s]
Train Epoch: 3 Loss: 0.081595
100%|██████████| 67/67 [00:01<00:00, 33.67it/s]
Train Epoch: 4 Loss: 0.168008
100%|██████████| 67/67 [00:02<00:00, 33.50it/s]
Train Epoch: 5 Loss: 0.063775
100%|██████████| 67/67 [00:01<00:00, 34.86it/s]
Train Epoch: 6 Loss: 0.071517
100%|██████████| 67/67 [00:01<00:00, 34.81it/s]
Train Epoch: 7 Loss: 0.033412
100%|██████████| 67/67 [00:01<00:00, 33.58it/s]
Train Epoch: 8 Loss: 0.020760
100%|██████████| 67/67 [00:02<00:00, 32.88it/s]
Train Epoch: 9 Loss: 0.038850
```

在自己分出的验证集上的正确率到达98%：

```
val set: Average loss: 0.0001, Accuracy: 731/744 (98%)
```