LeNet-5 网络实验报告

目录

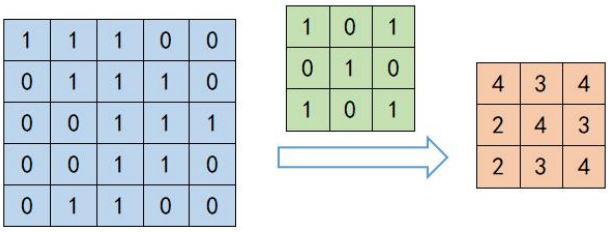
一、实验目的

实现 LeNet-5 在 MNIST 数据集上的训练和测试。

二、实验原理

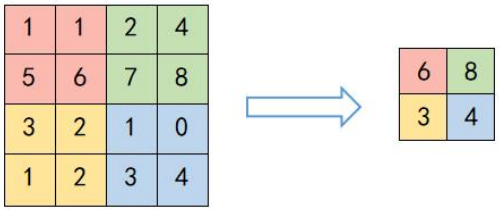
1、卷积层

图像识别里的卷积是二维卷积，即离散二维滤波器（也称作卷积核）与二维图像做卷积操作，二维滤波器滑动到二维图像上所有位置，并在每个位置上与该像素点及其领域像素点做内积。在深层卷积神经网络中，通过卷积操作可以提取出图像低级到复杂的特征。卷积层由若干卷积单元组成，负责输入图像的特征提取，卷积核，也称滤波器，其种类不同，所提取的特征也不同。通过卷积核过滤图像的各个小区域，从而得到这些区域的特征值。如图所示，原图输入为 5*5 的二维矩阵，卷积核为 3*3，最终输出的特征图为 3*3。



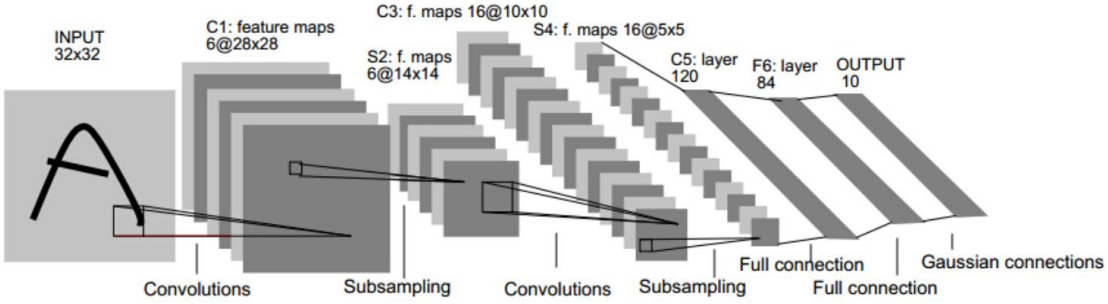
2、池化层

池化操作是取区域平均值或最大值，利于简化模型，提取鲁棒性更强的特征。如图所示，滤波器为 2 × 2，步长为 2，计算区域的最大值，最终得到 2 × 2 的结果。



3、LeNet-5 网络

LeNet-5 是一个经典的卷积神经网络模型，用于手写数字识别的深度神经网络模型，由两个卷积层和三个全连接层组成，其网络结构如下图。



3.1 INPUT 层-输入层

尺寸统一归一化为 32*32，输入 batchsize*32*32 的黑白分辨率图像；

3.2 C1 层-卷积层

输入图片：32*32；卷积核大小：5*5；卷积核种类：6；输出 featuremap 大小：28*28（32-5+1）=28；神经元数量：28*28*6；可训练参数：(5*5+1) * 6（每个滤波器 5*5 个 unit 参数和一个 bias 参数，一共 6 个滤波器）；连接数：（5*5+1）*6*28*28。

对输入图像进行第一次卷积运算（使用 7 个大小为 5*5 的卷积核），得到 6 个 C1 特征图（6 个大小为 28*28 的 feature maps, 32-5+1=28）。卷积核的大小为 5*5，总共就有 6*（5*5+1）=156 个参数，其中+1 是表示一个核有一个 bias。卷积层 C1 内的每个像素都与输入图像中的 5*5 个像素和 1 个 bias 有连接，所以总共有 156*28*28 个连接（connection）。有这么多个连接，但是只需要学习 156 个参数，主要是通过权值共享实现的。

3.3 S2 层-池化层（下采样层）

输入：28*28；采样区域：2*2；采样方式：4 个输入相加，乘以一个可训练参数，再加上一个可训练偏置。结果通过 sigmoid；采样种类：6；输出 featureMap 大小：14*14（28/2）；神经元数量：14*14*6；连接数：（2*2+1）*6*14*14。S2 中每个特征图的大小是 C1 中特征

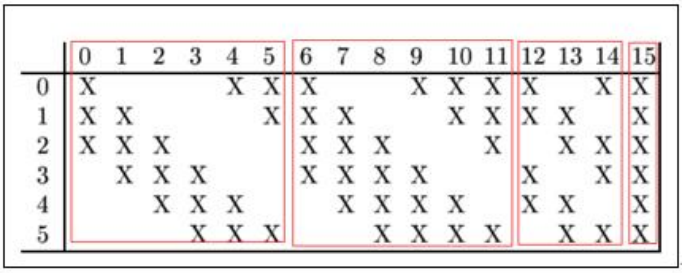
图大小的 1/4。

第一次卷积后紧接池化运算，使用 2*2 核进行池化，得到 6 个 14*14 的特征图（28/2=14）。S2 这个 pooling 层是对 C1 中的 2*2 区域内的像素求和乘以一个权值系数再加上一个偏置，然后将这个结果再做一次映射。同时有 5x14x14x6 个连接。

3.4 C3 层-卷积层

输入：S2 中几个特征 map 组合；卷积核大小：5*5；卷积核种类：16；输出 featureMap 大小：10*10 (14-5+1)；C3 中的每个特征 map 是连接到 S2 中几个特征 map 的，表示本层的特征 map 是上一层提取到的特征 map 的不同组合。可训练参数：1516。连接数：151600。

第一次池化后是第二次卷积，第二次卷积的输出是 16 个 10x10 的特征图，卷积核大小是 5*5。S2 有 6 个 14*14 的特征图，怎么从 6 个特征图得到 16 个特征图？ 这里是通过对 S2 的特征图特殊组合计算得到的 16 个特征图。具体如下：



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	X				X	X	X			X	X	X	X		X	X
1	X	X				X	X	X			X	X	X	X		X
2	X	X	X				X	X	X			X		X	X	X
3		X	X	X			X	X	X	X			X		X	X
4			X	X	X			X	X	X	X		X	X		X
5				X	X	X			X	X	X	X		X	X	X

C3 的前 6 个 feature map（对应上图第一个红框的 6 列）与 S2 层相连的 3 个 feature map 相连接（上图第一个红框），后面 6 个 feature map 与 S2 层相连的 4 个 feature map 相连接（上图第二个红框），后面 3 个 feature map 与 S2 层部分不相连的 4 个 feature map 相连接，最后一个与 S2 层的所有 feature map 相连。卷积核大小依然为 5*5，所以总共有 6*(3*5*5+1)+6*(4*5*5+1)+3*(4*5*5+1)+1*(6*5*5+1)=1516 个参数。图像大小为 10*10，共有 10*10*1516=151600 个连接。

3.5 S4 层-池化层（下采样层）

输入：10*10；采样区域：2*2；采样方式：4 个输入相加，乘以一个可训练参数，再加上一个可训练偏置。结果通过 sigmoid；采样种类：16；输出 featureMap 大小：5*5（10/2）；神经元数量：5*5*16=400；连接数：16*（2*2+1）*5*5=2000；S4 中每个特征图的大小是 C3 中特征图大小的 1/4。

S4 是 pooling 层，窗口大小 2*2，共计 16 个 feature map，C3 层的 16 个 10x10 的图分别进行以 2x2 为单位的池化得到 16 个 5x5 的特征图。2000 个连接的连接方式与 S2 层类似。

3.6 C5 层-卷积层

输入：S4 层的 16 个单元特征 map（与 s4 全相连）；卷积核大小：5*5；卷积核种类：120；输出 featureMap 大小：1*1（5-5+1）；可训练参数/连接：120*（16*5*5+1）=48120。

C5 层是一个卷积层。由于 S4 层的 16 个图的大小为 5x5，与卷积核的大小相同，所以卷积后形成的图的大小为 1x1。这里形成 120 个卷积结果。每个都与上一层的 16 个图相连。所以共有(5x5x16+1)x120 = 48120 个参数，同样有 48120 个连接。

3.7 F6 层-全连接层

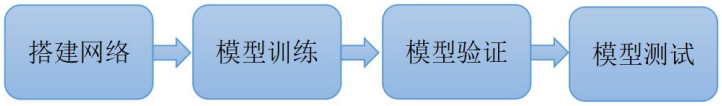
输入：c5 的 120 维向量；计算方式：计算输入向量和权重向量之间的点积，再加上一个偏置，结果通过 sigmoid 函数输出；可训练参数:84*(120+1)。

F6 层有 84 个节点，对应一个 7x12 的比特图，-1 表示白色，1 表示黑色，这样每个符号的比特图的黑白色就对应于一个编码。该层的训练参数和连接数是(120 + 1)x84。

3.8 Output 层-全连接层

Output 层也是全连接层，共有 10 个节点，分别代表数字 0 到 9，且如果节点 i 的值为 0，则网络识别的结果是数字 i。采用的是径向基函数（RBF）的网络连接方式。

三．实验步骤



1、下载并加载数据，并做出一定的预先处理

由于 MNIST 数据集图片尺寸是 28x28 单通道的，而 LeNet-5 网络输入 Input 图片尺寸是 32x32，因此使用 transforms.Resize 将输入图片尺寸调整为 32x32。

MNIST 数据集采用标准化 transforms.Normalize((0.1307,), (0.3081,))。标准化：样本减去均值，再除以标准差，最终样本呈现均值为 0 方差为 1 的数据分布。

神经网络模型偏爱标准化数据，因为这种数据在 sigmoid、tanh 经过激活函数后求导得到的导数很大，反之原始数据不仅分布不均（噪声大）而且数值通常都很大（本例中数值范围是 0~255），激活函数后求导得到的导数则接近与 0，这也称为梯度消失。所以数据的标准化有利于加快神经网络的训练。

除此之外，还需要保持 train_set、val_set 和 test_set 标准化系数的一致性。标准化系数就是计算要用到的均值和标准差，在本例中是((0.1307,), (0.3081,))，均值是 0.1307，标准差是 0.3081，这些系数都是数据集提供方计算好的数据。不同数据集有不同的标准化系数，例如([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])是 ImageNet dataset 的标准化系数（RGB 三个通道对应三组系数），当需要将 Imagenet 预训练的参数迁移到另一神经网络时，被迁移的神经网络就需要使用 Imagene 的系数，否则预训练不仅无法起到应有的作用甚至还会帮倒忙。

```
pipline_train = transforms.Compose([
    # 随机旋转图片
    transforms.RandomHorizontalFlip(),
    # 将图片尺寸resize到32x32
    transforms.Resize((32, 32)),
    # 将图片转化为Tensor格式
    transforms.ToTensor(),
    # 正则化(当模型出现过拟合的情况时，用来降低模型的复杂度)
    transforms.Normalize((0.1307,), (0.3081,))
])
pipline_test = transforms.Compose([
    # 将图片尺寸resize到32x32
    transforms.Resize((32, 32)),
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,))
])
# 下载数据集
train_set = datasets.MNIST(root="./MNIST", train=True, download=False, transform=pipline_train)
test_set = datasets.MNIST(root="./MNIST", train=False, download=False, transform=pipline_test)
# 加载数据集
trainloader = torch.utils.data.DataLoader(train_set, batch_size=128, shuffle=True)
testloader = torch.utils.data.DataLoader(test_set, batch_size=128, shuffle=False)
```

2、搭建 LeNet-5 神经网络结构，并定义前向传播的过程

```python
# 搭建 LeNet-5 神经网络结构，并定义前向传播的过程
class LeNet(nn.Module):
    def __init__(self):
        super(LeNet, self).__init__()
        self.conv1 = nn.Conv2d(1, 6, 5)
        self.relu = nn.ReLU()
        self.maxpool1 = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.maxpool2 = nn.MaxPool2d(2, 2)
        self.fc1 = nn.Linear(16*5*5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.conv1(x)
        x = self.relu(x)
        x = self.maxpool1(x)
        x = self.conv2(x)
        x = self.maxpool2(x)
        x = x.view(-1, 16*5*5)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        output = F.log_softmax(x, dim=1)
        return output
```

3、将定义好的网络结构搭载到 GPU/CPU，并定义优化器

```python
# 创建模型，部署gpu
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = LeNet().to(device)
# 定义优化器
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

4、定义训练过程

```python
def train_runner(model, device, trainloader, optimizer, epoch):
    # 训练模型，启用 BatchNormalization 和 Dropout，将BatchNormalization和Dropout置为True
    model.train()
    total = 0
    correct = 0.0

    # enumerate迭代已加载的数据集,同时获取数据和数据下标
    for i, data in enumerate(trainloader, 0):
        inputs, labels = data
        # 把模型部署到device上
        inputs, labels = inputs.to(device), labels.to(device)
        # 初始化梯度
        optimizer.zero_grad()
        # 保存训练结果
        outputs = model(inputs)
        # 计算损失和
        # 多分类情况通常使用cross_entropy(交叉熵损失函数)，而对于二分类问题，通常使用sigmod
        loss = F.cross_entropy(outputs, labels)
        # 获取最大概率的预测结果
        # dim=1表示返回每一行的最大值对应的列下标
        predict = outputs.argmax(dim=1)
        total += labels.size(0)
        correct += (predict == labels).sum().item()
        # 反向传播
        loss.backward()
        # 更新参数
        optimizer.step()
        if i % 1000 == 0:
            # loss.item()表示当前loss的数值
            print("Train Epoch{} \t Loss: {:.6f}, accuracy: {:.6f}%".format(epoch, loss.item(), 100*(correct/total)))
            Loss.append(loss.item())
            Accuracy.append(correct/total)
    return loss.item(), correct/total
```

5、定义验证过程

```python
def test_runner(model, device, testloader):
    # 模型验证, 必须要写, 否则只要有输入数据, 即使不训练, 它也会改变权值
    # 因为调用eval()将不启用 BatchNormalization 和 Dropout, BatchNormalization和Dropout置为False
    model.eval()
    # 统计模型正确率, 设置初始值
    correct = 0.0
    test_loss = 0.0
    total = 0
    # torch.no_grad将不会计算梯度, 也不会进行反向传播
    with torch.no_grad():
        for data, label in testloader:
            data, label = data.to(device), label.to(device)
            output = model(data)
            test_loss += F.cross_entropy(output, label).item()
            predict = output.argmax(dim=1)
            # 计算正确数量
            total += label.size(0)
            correct += (predict == label).sum().item()
        # 计算损失值
        print("test_avarage_loss: {:.6f}, accuracy: {:.6f}%".format(test_loss/total, 100*(correct/total)))
```

6、运行

```python
# 调用
epoch = 30
Loss = []
Accuracy = []
for epoch in range(1, epoch+1):
    print("start_time", time.strftime('%Y-%m-%d %H:%M:%S', time.localtime(time.time())))
    loss, acc = train_runner(model, device, trainloader, optimizer, epoch)
    Loss.append(loss)
    Accuracy.append(acc)
    test_runner(model, device, testloader)
    print("end time: ", time.strftime('%Y-%m-%d %H:%M:%S', time.localtime(time.time())), '\n')
```

7、模型输出及保存

```python
print(model)
torch.save(model, './models/model-mnist.pth')  # 保存模型
```

四、实验结果

1、数据集 MNIST

MNIST 是一个手写体数字的图片数据集，该数据集来由美国国家标准与技术研究所（National Institute of Standards and Technology (NIST)）发起整理，一共统计了来自 250 个不同的人手写数字图片，其中 50%是高中生，50%来自人口普查局的工作人员。该数据集的收

集目的是希望通过算法，实现对手写数字的识别。1998 年，Yan LeCun 等人首次提出 LeNet-5 网络，利用上述数据集实现了手写字体的识别。

数据集主要包括以下四个文件。训练集一共包含 60,000 张图像和标签，测试集一共包含 10,000 张图像和标签。测试集中前 5000 个来自最初 NIST 项目的训练集，后 5000 个来自最初 NIST 项目的测试集。因为前 5000 个数据来自于美国人口普查局的员工，而后 5000 个来自于大学生，所以前 5000 个更规整。

该数据集自 1998 年起，被广泛地应用于机器学习和深度学习领域，用来测试算法的效果，例如线性分类器（Linear Classifiers）、K-近邻算法（K-Nearest Neighbors）、支持向量机（SVMs）、神经网络（Neural Nets）、卷积神经网络（Convolutional nets）等等。

文件下载	文件用途
train-images-idx3-ubyte.gz	训练集图像
train-labels-idx1-ubyte.gz	训练集标签
t10k-images-idx3-ubyte.gz	测试集图像
t10k-labels-idx1-ubyte.gz	测试集标签

解压得到的并不是一系列图片，而是.idx1-ubyte 和.idx3-ubyte 格式的文件。这是一种 IDX 数据格式，其基本格式如下。

```
magic number
size in dimension 0
size in dimension 1
size in dimension 2
.....
size in dimension N
data
```

```
0x08: unsigned byte
0x09: signed byte
0x0B: short (2 bytes)
0x0C: int (4 bytes)
0x0D: float (4 bytes)
0x0E: double (8 bytes)
```

其中 magic number 为 4 字节，前 2 字节永远是 0，第 3 字节代表数据的格式。

第 4 字节的含义表示维度的数量（dimensions）: 1 表示一维（比如 vectors）， 2 表示二维（比如 matrices），3 表示三维（比如 numpy 表示的图像：高，宽，通道数）。

训练集和测试集的标签文件的格式(train-labels-idx1-ubyte 和 t10k-labels-idx1-ubyte) idx1-ubtype 的文件数据格式如下：

```
[offset] [type]          [value]          [description]
0000     32 bit integer  0x00000801(2049) magic number (MSB first)
0004     32 bit integer  60000            number of items
0008     unsigned byte   ??               label
0009     unsigned byte   ??               label
........
xxxx     unsigned byte   ??               label
```

第 0 ~ 3 字节是 32 位整型数据，取值为 0x00000801(2049)，即用幻数 2049 记录文件数据格式，这里的格式为文本格式。

第 4～7 个字节是 32 位整型数据，取值为 60000（训练集时）或 10000（测试集时），用来记录标签数据的个数；

第 8 个字节 ~ ，是一个无符号型的数，取值为对应 0~9 之间的标签数字，用来记录样本的标签。

训练集和测试集的图像文件的格式(train-images-idx3-ubyte 和 t10k-images-idx3-ubyte) idx3-ubtype 的文件数据格式如下：

```
[offset] [type]          [value]          [description]
0000     32 bit integer  0x00000803(2051) magic number
0004     32 bit integer  60000            number of images
0008     32 bit integer  28               number of rows
0012     32 bit integer  28               number of columns
0016     unsigned byte   ??               pixel
0017     unsigned byte   ??               pixel
........
xxxx     unsigned byte   ??               pixel
```

第 0 ~ 3 字节，是 32 位整型数据，取值为 0x00000803(2051)，即用幻数 2051 记录文件数据格式，这里的格式为图片格式。

第 4～7 个字节，是 32 位整型数据，取值为 60000（训练集时）或 10000（测试集时），用来记录图片数据的个数。

第 8～11 个字节，是 32 位整型数据，取值为 28，用来记录图片数据的高度。

第 12～15 个字节，是 32 位整型数据，取值为 28，用来记录图片数据的宽度。

第 16 个字节 ~ ，是一个无符号型的数，取值为 0~255 之间的灰度值，用来记录图片按行展开后得到的灰度值数据，其中 0 表示背景（白色），255 表示前景（黑色）。

2、训练

训练一共迭代 30 轮，下面是前三轮和后三轮的截图示意。

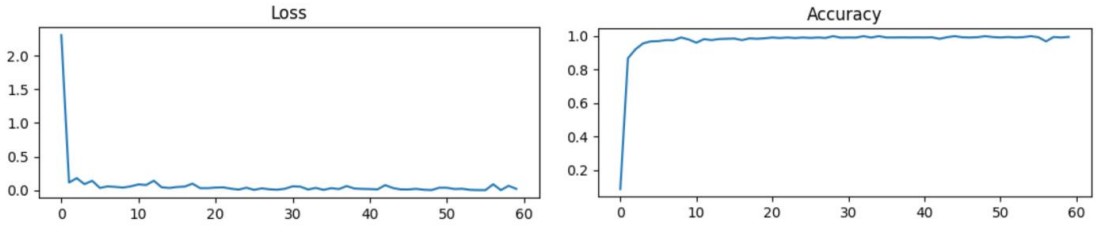
```
start_time 2023-04-15 13:17:52
Train Epoch1     Loss: 2.307597, accuracy: 8.593750%
test_avarage_loss: 0.001237, accuracy: 95.100000%
end_time:  2023-04-15 13:18:03

start_time 2023-04-15 13:18:03
Train Epoch2     Loss: 0.178003, accuracy: 92.187500%
test_avarage_loss: 0.000705, accuracy: 97.060000%
end_time:  2023-04-15 13:18:12

start_time 2023-04-15 13:18:12
Train Epoch3     Loss: 0.138998, accuracy: 96.875000%
test_avarage_loss: 0.000578, accuracy: 97.610000%
end_time:  2023-04-15 13:18:21
```

(a) 前三轮

```
start_time 2023-04-15 13:22:06
Train Epoch28    Loss: 0.001054, accuracy: 100.000000%
test_avarage_loss: 0.000491, accuracy: 98.370000%
end_time:  2023-04-15 13:22:16

start_time 2023-04-15 13:22:16
Train Epoch29    Loss: 0.087192, accuracy: 96.875000%
test_avarage_loss: 0.000498, accuracy: 98.550000%
end_time:  2023-04-15 13:22:25

start_time 2023-04-15 13:22:25
Train Epoch30    Loss: 0.064486, accuracy: 99.218750%
test_avarage_loss: 0.000502, accuracy: 98.590000%
end_time:  2023-04-15 13:22:34

Finished Training
```

(b) 后三轮

经历 30 次 epoch 的 loss 和 accuracy 曲线如下。



LeNet-5 的模型输出如下。

```
LeNet(
  (conv1): Conv2d(1, 6, kernel_size=(5, 5), stride=(1, 1))
  (relu): ReLU()
  (maxpool1): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (conv2): Conv2d(6, 16, kernel_size=(5, 5), stride=(1, 1))
  (maxpool2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (fc1): Linear(in_features=400, out_features=120, bias=True)
  (fc2): Linear(in_features=120, out_features=84, bias=True)
  (fc3): Linear(in_features=84, out_features=10, bias=True)
)
```

3、测试

本轮一共选取了数字 0-9 各四张图片进行测试，选取的图像部分来自网络，部分为手写拍摄获取。以下为部分成功案例。

概率： tensor([[8.0041e-01, 1.4927e-11, 7.4975e-04, 2.5605e-04, 3.0897e-10, 3.2973e-09,
1.3918e-09, 5.1159e-10, 1.8756e-01, 1.1023e-02]], device='cuda:0',
grad_fn=<SoftmaxBackward0>)
预测类别： 0

(a) 原图　　(b) 预测结果

概率： tensor([[0.1788, 0.0910, 0.4645, 0.0474, 0.0131, 0.0440, 0.0097, 0.0461, 0.0449,
0.0607]], device='cuda:0', grad_fn=<SoftmaxBackward0>)
预测类别： 2

(a) 原图　　(b) 预测结果

概率： tensor([[1.6915e-07, 3.8951e-13, 1.9495e-02, 9.5139e-05, 5.5756e-16, 1.9471e-08,
6.1631e-09, 1.7010e-10, 9.8041e-01, 1.1536e-07]], device='cuda:0',
grad_fn=<SoftmaxBackward0>)
预测类别： 8

(a) 原图　　(b) 预测结果

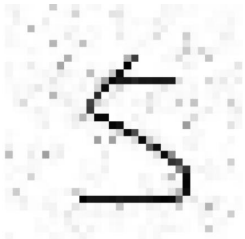
概率： tensor([[3.1302e-02, 1.0975e-10, 1.0817e-05, 1.6120e-04, 7.0600e-05, 7.0803e-08,
2.0188e-05, 4.5537e-08, 1.5930e-03, 9.6684e-01]], device='cuda:0',
grad_fn=<SoftmaxBackward0>)
预测类别： 9

(a) 原图　　(b) 预测结果

以下为部分失败案例。由于 MNIST 数据集是黑底白字，测试的几张图片都不是，所以精度有所下降。数字 5 所处背景噪声较多，数字 6 本身数字不是很清晰，这些都是导致无法正确识别的原因所在。

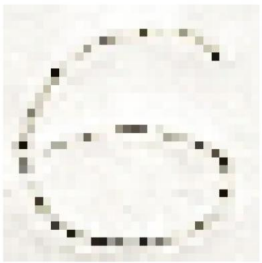
概率： tensor([[9.9960e-01, 6.8513e-08, 1.3222e-04, 3.6210e-07, 5.5841e-07, 5.7102e-07,
3.0467e-07, 4.8045e-08, 2.0954e-04, 5.8454e-05]], device='cuda:0',
grad_fn=<SoftmaxBackward0>)
预测类别： 0

(a) 原图　　(b) 预测结果



概率： tensor([[7.9028e-05, 1.9656e-09, 3.3934e-02, 3.1296e-04, 1.1247e-10, 3.0363e-05,
2.2327e-02, 6.4500e-08, 9.4332e-01, 9.3492e-08]], device='cuda:0',
grad_fn=<SoftmaxBackward0>)
预测类别： 8

(a) 原图　　(b) 预测结果



概率： tensor([[7.9759e-01, 4.8451e-05, 1.5233e-06, 1.8686e-06, 1.0501e-08, 1.4053e-07,
2.0231e-01, 3.8651e-06, 3.9724e-05, 5.4767e-08]], device='cuda:0',
grad_fn=<SoftmaxBackward0>)
预测类别： 0

(a) 原图　　(b) 预测结果