目录

任务二 MNSTI 图像分割实验报告

一、图像分割

1. 任务介绍

图像分割就是把图像分成若干个特定的、具有独特性质的区域并提出感兴趣目标的技术和过程。它是由图像处理到图像分析的关键步骤。现有的图像分割方法主要分以下几类：基于阈值的分割方法、基于区域的分割方法、基于边缘的分割方法以及基于特定理论的分割方法等。从数学角度来看，图像分割是将数字图像划分成互不相交的区域的过程。图像分割的过程也是一个标记过程，即把属于同一区域的像素赋予相同的编号。

图 1.1　图像分割示意图

2. 方法

（1）基于阈值分割

阈值法的基本思想是基于图像的灰度特征来计算一个或多个灰度阈值，并将图像中每个像素的灰度值与阈值作比较，最后将像素根据比较结果分到合适的类别中。因此，该方法最为关键的一步就是按照某个准则函数来求解最佳灰度阈值。阈值法特别适用于目标和背景占据不同灰度级范围的图。图像若只有目标和背景两大类，那么只需要选取一个阈值进行分割，此方法成为单阈值分割；但是如果图像中有多个目标需要提取，单一阈值的分割就会出现作物，在这种情况下就需要选取多个阈值将每个目标分隔开，这种分割方法相应的成为多阈值分割。

图 4. 原始图像

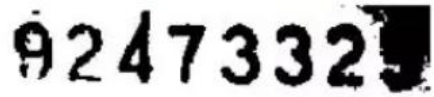


图 5. 阈值低，对亮区效果好，则暗区差

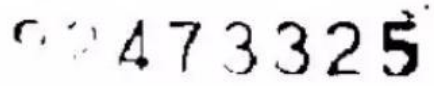


图 6. 阈值高，对暗区效果好，则亮区差

图 1.2　阈值分割法

（2） 基于边缘检测

基于边缘检测的图像分割算法试图通过检测包含不同区域的边缘来解决分割问题。它可以说是人们最先想到也是研究最多的方法之一。通常不同区域的边界上像素的灰度值变化比较剧烈，如果将图片从空间域通过傅里叶变换到频率域，边缘就对应着高频部分，这是一种非常简单的边缘检测算法。边缘检测技术通常可以按照处理的技术分为串行边缘检测和并行边缘检测。串行边缘检测是要想确定当前像素点是否属于检测边缘上的一点，取决于先前像素的验证结果。并行边缘检测是一个像素点是否属于检测边缘高尚的一点取决于当前正在检测的像素点以及与该像素点的一些临近像素点。最简单的边缘检测方法是并行微分算子法，它利用相邻区域的像素值不连续的性质，采用一阶或者二阶导数来检测边缘点。近年来还提出了基于曲面拟合的方法、基于边界曲线拟合的方法、基于反应-扩散方程的方法、串行边界查找、基于变形模型的方法。

(a) 梯度算法处理的结果　(b) Roberts 算法　(c) Sobel 算法

图 1.2　边缘检测法图像分割

（3）基于深度学习

基于深度学习的方法可以直接通过标注数据，将图像送入神经网络中让模型自动学习用于分割的特征，这里举个例子，FCN（Fully Convolutional Network）在 FCN 当中的反卷积-升采样结构中，图片会先进性上采样（扩大像素）；再进行卷积——通过学习获得权值。FCN 的网络结构如下图所示：

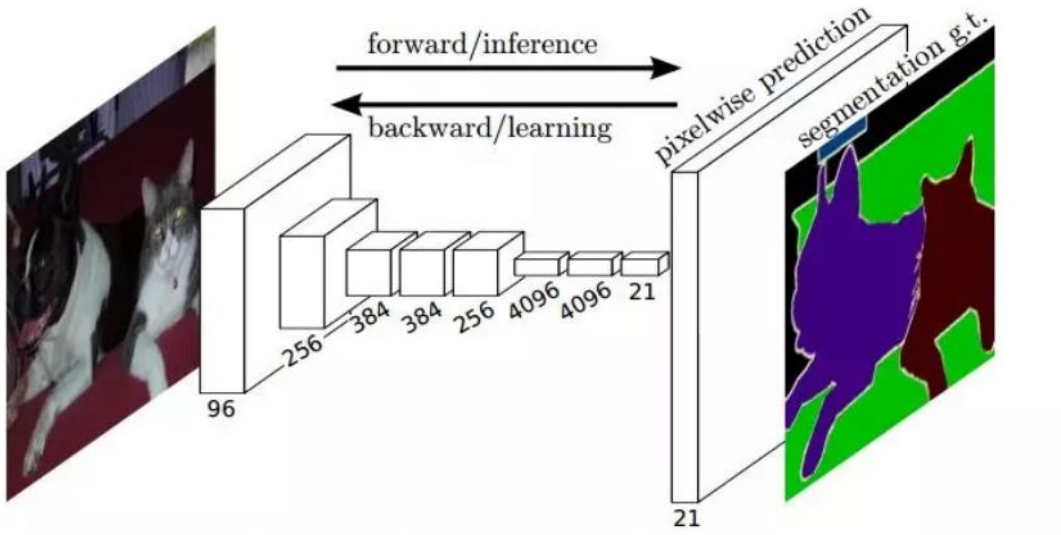


图 1.4　FCN 架构图

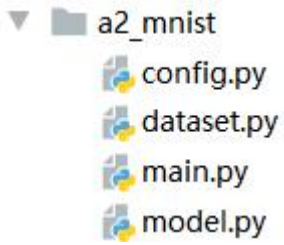
优缺点：

- FCN 对图像进行了像素级的分类，从而解决了语义级别的图像分割问题；
- FCN 可以接受任意尺寸的输入图像，可以保留下原始输入图像中的空间信息；
- 得到的结果由于上采样的原因比较模糊和平滑，对图像中的细节不敏感；
- 对各个像素分别进行分类，没有充分考虑像素与像素的关系，缺乏空间一致性。

二、建模（编程）过程

1. 架构

跟上次的 MNIST 图像分类一样，主要分成了数据处理模块 dataset、CNN 模型 model、配置文件 config 和主函数 main，我使用了 pytorch1.6 框架，在 Ubuntu16.04 上 GTX2080TI 训练模型。



2. 构建数据集

由于 pytorch 已经封装好了 MNIST 数据集，直接调用 API 即可：

```
import torch
from torchvision import datasets, transforms

def get_mnist_dataset(args, kwargs):

    train_loader = torch.utils.data.DataLoader(
        datasets.MNIST('./data', train=True, download=True,
                       transform=transforms.Compose([
                           transforms.ToTensor(),
                           transforms.Normalize((0.1307,), (0.3081,))])),
        batch_size=args.batch_size, shuffle=True, **kwargs)
    test_loader = torch.utils.data.DataLoader(
        datasets.MNIST('./data', train=False, transform=transforms.Compose([
            transforms.ToTensor(),
            transforms.Normalize((0.1307,), (0.3081,))])),
        batch_size=args.test_batch_size, shuffle=True, **kwargs)
    return train_loader, test_loader
```

但与上次不同的是，需要对背景添加噪声，并需要构建前后背景的标签。代码如下：

```
pixel_label, bg_mask = torch.zeros(data.size()).cuda().long(), torch.zeros(data.size()).cuda().long()
pixel_label[data > 0] = 1
pixel_label = pixel_label * (target.view(b, 1, 1, 1) + 1)
bg_mask[data == 0] = 1
data = (data - 0.1307) / 0.3081
noise = torch.randn(data.size()).cuda() * 1
noise *= bg_mask
data += noise
```

直接对原来像素值为 0 的区域添加高斯噪声，效果如下：

图 1.4　背景添加噪声前后对比图

3. 构建模型

在构建 LeNet 之前，我想先试试直接线性的 MLP 类似自编码器的结构看看性能，将原始像素映射到一个隐空间维度后再映射回去：

```python
class MLP(nn.Module):
    def __init__(self):
        super(MLP, self).__init__()
        self.fc = nn.Linear(28 * 28, 10)

    def forward(self, x):
        x = x.view(x.size(0), -1)
        x = self.fc(x)
        return F.log_softmax(x, dim=1)
```

接着我尝试了 LeNet 像 FCN 那样的结构，卷积池化后再上池化反卷积回去，用 channel 维度做分类，我尝试了两种，分别是一层和两次卷积池化再回去的。

```python
class LeNet_seg(nn.Module):
    def __init__(self):
        super(LeNet_seg, self).__init__()
        self.conv1 = nn.Conv2d(in_channels=1, out_channels=6, kernel_size=5, stride=1, padding=2)
        self.conv2 = nn.Conv2d(6, 16, 5, 1)  # 没padding
        self.fc1 = nn.Linear(5 * 5 * 16, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)
        self.drop = nn.Dropout(p=0.05)

        self.deconv1 = nn.ConvTranspose2d(in_channels=16, out_channels=6, kernel_size=5, stride=1, padding=0, output_padding=0, bias=True)
        self.deconv2 = nn.ConvTranspose2d(6, 11, 5, padding=2)
        self.unpool1 = nn.MaxUnpool2d(kernel_size=2, stride=2)
        self.cls = nn.Linear(11, 11)

    def forward(self, x):
        b, _, h, w = x.shape
        x = F.relu(self.conv1(x))  # [b,6,28,28]
        x, pool_idx1 = F.max_pool2d(x, 2, 2, return_indices=True)  # [b,6,14,14]
        # x = F.relu(self.conv2(x))  # [b,16,10,10]
        # x, pool_idx2 = F.max_pool2d(x, 2, 2, return_indices=True)  # [b,16,5,5]
        # # cls
        # x = x.view(b, -1)
        # x = F.relu(self.fc1(x))
        # x = F.relu(self.fc2(x))
        # x = self.fc3(x)
        # seg
        # x_ = self.unpool1(x, pool_idx2)  # [b,16,10,10]
        # x_ = self.deconv1(x)  # [b,6,14,14]
        x_ = self.unpool1(x, pool_idx1)  # [b,6,28,28]
        x_ = self.deconv2(x_)  # [b,11,28,28]
        x_ = x_.permute(0, 2, 3, 1)
        # x_ = self.cls(x_))
        return F.log_softmax(x, dim=-1), F.log_softmax(x_, dim=-1)  # , F.sigmoid(x_)
```

```python
class LeNet_seg_simple(nn.Module):
    def __init__(self):
        super(LeNet_seg_simple, self).__init__()
        self.conv1 = nn.Conv2d(in_channels=1, out_channels=11, kernel_size=5, stride=1, padding=2)

    def forward(self, x):
        b, _, h, w = x.shape
        x_ = self.conv1(x)  # [b,6,28,28]
        x_ = x_.permute(0, 2, 3, 1)

        return F.log_softmax(x, dim=-1), F.log_softmax(x_, dim=-1)  # , F.sigmoid(x_)
```

4. 模型训练与测试

就是把构建好的 dataloader 迭代器取出 bacth 数据送入模型接口，然后求交叉熵损失再反向传播更新参数即可。

与上次不同的是我使用了 11 分类的分割，将原始的数字只加 1，把 0 留给背景类，这样训练更加方便，统计二分类准确率也很简单。我进行测试 pixacc 的代码如下，count_bi 用来统计 2 分类准确率，即分别找到前后景的标签和预测结果，再统计即可，count 用来统计 10 分类准确率，这个直接和原先设定好的每个像素的 11 个类别比较即可，把相同的 sum 起来。

```
# TODO:pixel acc
count = torch.zeros(pixel_label.size()).long()
pred = pixel.argmax(dim=-1, keepdim=False)
count[pred == pixel_label] = 1
# print(pred[0])
count_bi = torch.zeros(pixel_label.size()).long()
tmp1 = torch.zeros(pixel_label.size()).long()
tmp2 = torch.zeros(pixel_label.size()).long()
tmp1[pred > 0] = 1
tmp2[pixel_label > 0] = 1
count_bi[tmp1 == tmp2] = 1

pixel_rate += count.sum().item()
pixel_binary_rate += count_bi.sum().item()
```

特别的我为了更好的观测训练过程，还使用了 tensorboard 还可视化模型与训练结果，将在下节展现。

三、实验

1. 实验设计与结果

主要就是有以下模型/方法，及其不同 epoch 时，在测试集的准确率指标，因为数据集比较干净简单，所以很快就可以到非常好的结果，我就只训练的 3epoch，且由趋势可以看出来第一个 epoch 就基本达到了性能的饱和点。

表 3.1　模型性能对比

模型/方法	**PixAcc(2 分类)**	**PixAcc(10 分类)**
MLP 一层	0.9908	**0.9723**
MLP 两层	**0.9914**	0.9719
LeNet 一层 conv+pooling	0.9702	0.8473
LeNet 两层 conv+pooling	0.9825	0.9168

其中 MLP 这种简单的模型反而胜过了 CNN 模型，可能是因为 MNIST 数据集确实太简单了，而 MLP 包含了巨大的参数量可能更好学习，但在更复杂的任务中 MLP 显然是不可能的，因为会爆显存而且性能也会因为任务复杂而下降。在 MLP 里，一层就已经足够好了，而在模仿 FCN 网络的 LeNet 在使用两层卷积池化后的效果更好。

2. 可视化部分

（1）LeNet 网络

图 3.1 可以看出网络中的各模块，以及数据在每个模型中是如何流动的

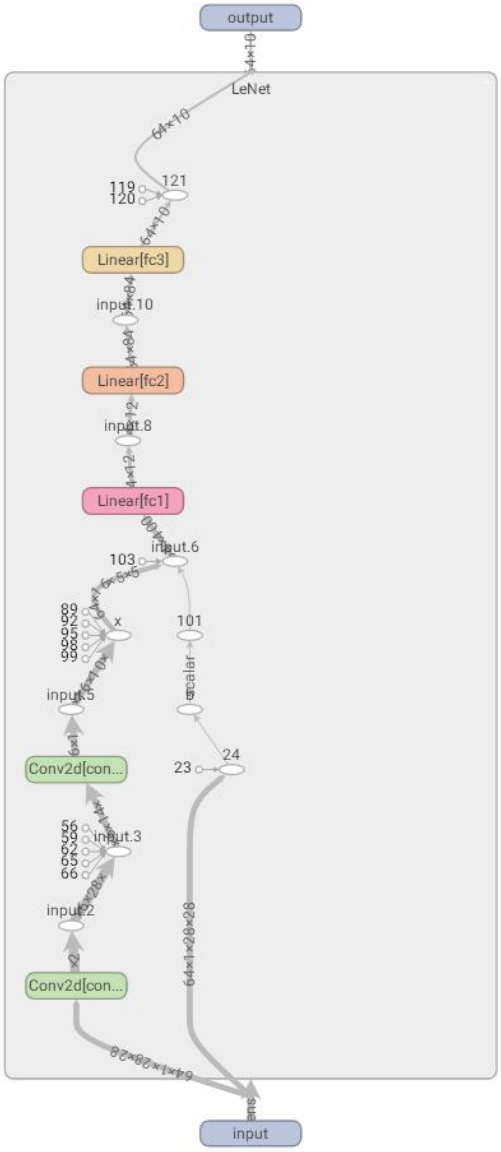


图 3.2　tensorboard 可视化的 LeNet 网络

（2）损失函数与准确率曲线

这里只可视化了 LeNet 网络的结果，如图 3.2 和 3.3 所示。由于 MNIST 上做分割非常简单，在第一个 epoch 就可以达到基本饱和的性能，我只可视化了第一个 epoch，可以看到整体 loss 收敛的不错，有一定的震荡是正常现象，因为美国 batch 内的学习难度本就不同，而且因为是在第一个 epoch，每次遇到的都是模型从未遇到的图片。

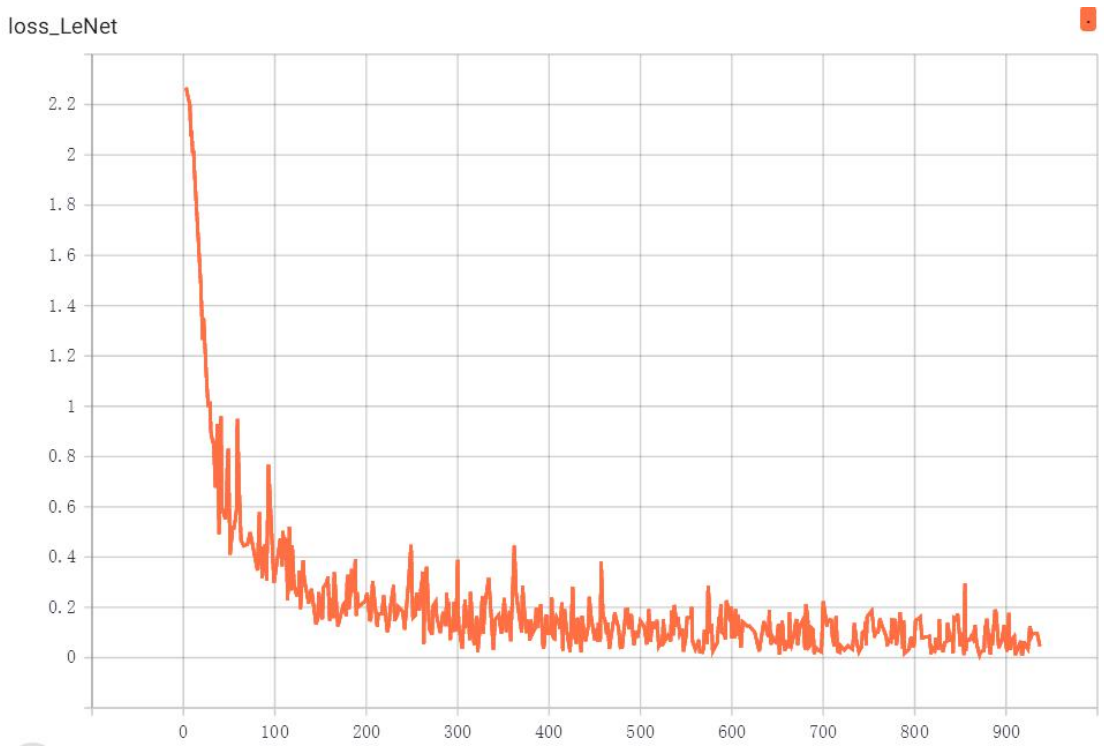


图 3.2 第一个 epoch 内 LeNet 在训练集上 loss 变化曲线

而准确率曲线就更加稳定了，因为每次训练过程中的学习，模型都会强化自身，从而在测试集达到更优秀的性能，而且可以看到非常之快，100 次迭代训练 10%左右的数量数据就可以达到 90%的测试集性能。

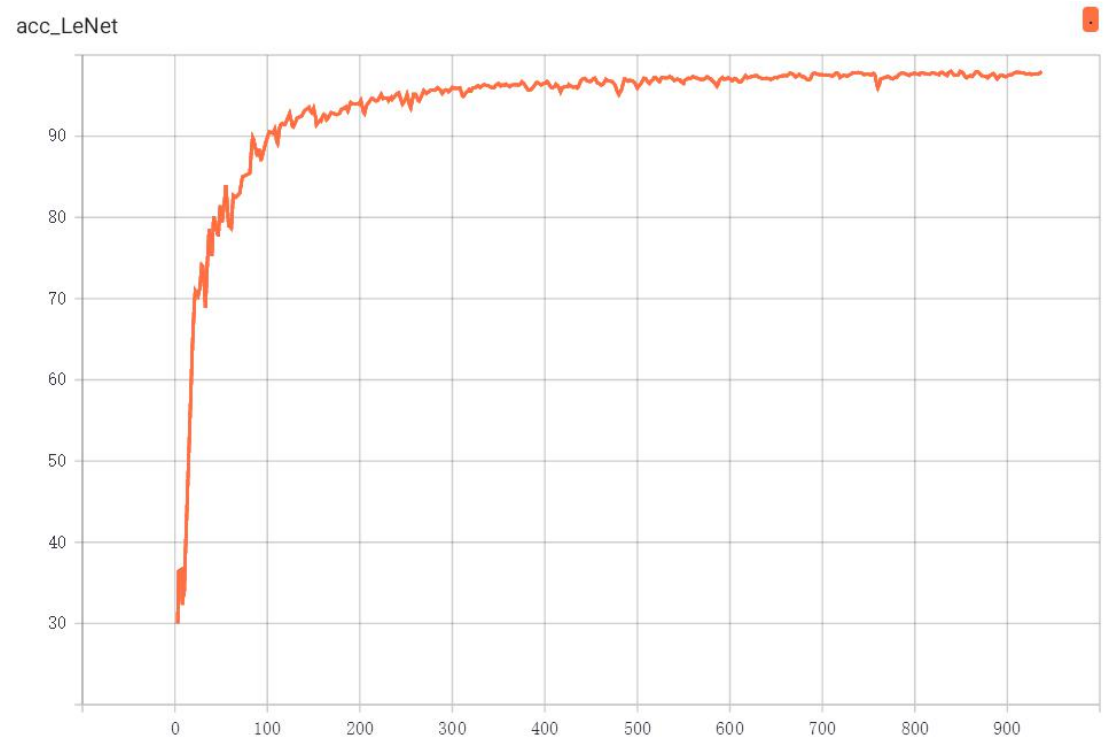


图 3.3 第一个 epoch 内 LeNet 在测试集上准确率变化曲线

参考文献

[1] http://yann.lecun.com/exdb/mnist/
[2] http://yann.lecun.com/exdb/lenet/
[3] https://blog.csdn.net/bigbennyguo/article/details/87956434
[4] https://pytorch.org/tutorials/
[5] https://zhuanlan.zhihu.com/p/41736894
[6]https://www.zhihu.com/search?type=content&q=%E5%9B%BE%E5%83%8F%E5%88%86%E5%89%B2