视网膜血管检测分割

一、任务简述

本项目是基于医学图像的图像分割问题，视网膜的血管是最终要检测分割出来的前景，其余部分皆是背景，本质上是一个基于像素的二分类问题。本任务的数据集为 OCTA 数据集，包含 300 组视网膜的 OCTA 图片，每组视网膜图片包含 400 张 640*400 的以 bmp 储存的图片，在使用这些图片时需要将其拼凑成 640*400*400 的 volumn。血管分割的 ground truth 大小为 400*400。由于训练的图像是三维立体图像，而 gt 是二维图像，故需要对三维的图像经过一些投影变换，再在二维的平面上进行血管检测分割。本报告完成这个项目的大致流程为，直接将三维的立体图像投影到二维平面，经过一些数据增强后生成最终喂入模型的二维训练图像。模型采用 2D-Unet 的基本框架搭建，经过了四次下采样的编码层与四次上采样的解码层后利用 argmax 获得预测结果。训练模型时以前 160 组图片为训练集，161-180 组图片为验证集，181-200 组图片为测试集，采用 CrossEntropy 作为损失函数进行梯度下降，最终得到的模型在测试集上的 Dice 为 0.8304，BAC 为 0.9009。**模型在 200-300 组数据上的表现为：Dice 为 0.8125，BAC 为 0.8953.**

二、代码框架

```
├──Data
│    │    MakeDataset.py
│    │    MyDataset.py
│    │    readData.py
│    └──    TestDataset.py
│
├──lib
│    │    Evaluate.py
│    │    extract_patches.py
│    │    my_classes.py
│    │    utils.py
│    └──    visualize.py
│
├──options
│    │    base_options.py
│    │    Test_options.py
│    └──train_options.py
│
│    Model.py
│    read.txt
│    test.py
```

└──train.py

三、完成流程

1.数据预处理

1.1 直接投影

在数据预处理部分，本项目将 400 张 640×400 的图像拼接后得到了 640×400×400 的三维图像。在刚开始时尝试使用 IPN 直接对三维图像进行二维分割时发现三维图像的训练数据太大，由于设备的 GPU 资源不够难以进行训练，另外，由于设备的内存也不够充足，故只能将数据直接投影处理成二维图像后保存减小内存负担，因此只能放弃对三维图像进行训练。最终本项目将直接投影后的二维图像作为训练数据，将分割任务转换到了二维平面上进行。

1.2 直方图均衡化

观察投影后的图片发现图片整体偏暗，血管与周围的干扰信息之间的对比不是特别明显，故本项目还对直接投影后的图像做了直方图均衡化的处理，希望能够提高血管与干扰信息的对比度，从而更易区分。本项目分别对未做直方图均衡化的直接投影图像与直方图均衡化后的图像进行训练得到了两个模型。模型效果详见结果展示。

2. 模型搭建

2.1 Unet

本项目采用了 2D-Unet 的网络结构，经过四次下采样的编码层与四次上采样的解码层后在输出的通道维度上做 argmax 得到最终的预测结果。Unet 的模型打印如下：

```
==========================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================
├─Downsample_block: 1-1                  [-1, 64, 32, 32]          --
|    └─Conv2d: 2-1                       [-1, 64, 64, 64]          640
|    └─BatchNorm2d: 2-2                  [-1, 64, 64, 64]          128
|    └─Conv2d: 2-3                       [-1, 64, 64, 64]          36,928
|    └─BatchNorm2d: 2-4                  [-1, 64, 64, 64]          128
├─Downsample_block: 1-2                  [-1, 128, 16, 16]         --
|    └─Conv2d: 2-5                       [-1, 128, 32, 32]         73,856
|    └─BatchNorm2d: 2-6                  [-1, 128, 32, 32]         256
|    └─Conv2d: 2-7                       [-1, 128, 32, 32]         147,584
|    └─BatchNorm2d: 2-8                  [-1, 128, 32, 32]         256
├─Downsample_block: 1-3                  [-1, 256, 8, 8]           --
```

└─Conv2d: 2-9	[-1, 256, 16, 16]	295,168
	└─BatchNorm2d: 2-10	[-1, 256, 16, 16]	512
	└─Conv2d: 2-11	[-1, 256, 16, 16]	590,080
	└─BatchNorm2d: 2-12	[-1, 256, 16, 16]	512
├─Downsample_block: 1-4	[-1, 512, 4, 4]	--
	└─Conv2d: 2-13	[-1, 512, 8, 8]	1,180,160
	└─BatchNorm2d: 2-14	[-1, 512, 8, 8]	1,024
	└─Conv2d: 2-15	[-1, 512, 8, 8]	2,359,808
	└─BatchNorm2d: 2-16	[-1, 512, 8, 8]	1,024
├─Conv2d: 1-5	[-1, 1024, 4, 4]	4,719,616
├─BatchNorm2d: 1-6	[-1, 1024, 4, 4]	2,048
├─Conv2d: 1-7	[-1, 1024, 4, 4]	9,438,208
├─BatchNorm2d: 1-8	[-1, 1024, 4, 4]	2,048
├─Upsample_block: 1-9	[-1, 512, 8, 8]	--
	└─ConvTranspose2d: 2-17	[-1, 512, 8, 8]	8,389,120
	└─Conv2d: 2-18	[-1, 512, 8, 8]	4,719,104
	└─BatchNorm2d: 2-19	[-1, 512, 8, 8]	1,024
	└─Conv2d: 2-20	[-1, 512, 8, 8]	2,359,808
	└─BatchNorm2d: 2-21	[-1, 512, 8, 8]	1,024
├─Upsample_block: 1-10	[-1, 256, 16, 16]	--
	└─ConvTranspose2d: 2-22	[-1, 256, 16, 16]	2,097,408
	└─Conv2d: 2-23	[-1, 256, 16, 16]	1,179,904
	└─BatchNorm2d: 2-24	[-1, 256, 16, 16]	512
	└─Conv2d: 2-25	[-1, 256, 16, 16]	590,080
	└─BatchNorm2d: 2-26	[-1, 256, 16, 16]	512
├─Upsample_block: 1-11	[-1, 128, 32, 32]	--
	└─ConvTranspose2d: 2-27	[-1, 128, 32, 32]	524,416
	└─Conv2d: 2-28	[-1, 128, 32, 32]	295,040
	└─BatchNorm2d: 2-29	[-1, 128, 32, 32]	256
	└─Conv2d: 2-30	[-1, 128, 32, 32]	147,584
	└─BatchNorm2d: 2-31	[-1, 128, 32, 32]	256
├─Upsample_block: 1-12	[-1, 64, 64, 64]	--
	└─ConvTranspose2d: 2-32	[-1, 64, 64, 64]	131,136
	└─Conv2d: 2-33	[-1, 64, 64, 64]	73,792
	└─BatchNorm2d: 2-34	[-1, 64, 64, 64]	128
	└─Conv2d: 2-35	[-1, 64, 64, 64]	36,928
	└─BatchNorm2d: 2-36	[-1, 64, 64, 64]	128
├─Conv2d: 1-13	[-1, 2, 64, 64]	130

==========

Total params: 39,398,274

Trainable params: 39,398,274

Non-trainable params: 0

Total mult-adds (G): 5.04

==========

Input size (MB): 0.02
Forward/backward pass size (MB): 34.31
Params size (MB): 150.29
Estimated Total Size (MB): 184.62
==========

2.1 IPN

此外，其实本项目还搭建了从三维到二维的图像分割网络 IPN，具体实现一共搭建了五层结构。但是由于训练图像是三维的，数据太大难以在 GPU 上训练以及电脑内存不足等原因并未对此模型进行训练。IPN 的模型打印如下：

```
==========
Layer (type:depth-idx)                   Output Shape              Param #
==========
├─Sequential: 1-1                        [-1, 64, 32, 100, 100]    --
|    └─Conv3d: 2-1                       [-1, 64, 160, 100, 100]   1,792
|    └─ReLU: 2-2                         [-1, 64, 160, 100, 100]   --
|    └─Conv3d: 2-3                       [-1, 64, 160, 100, 100]   110,656
|    └─ReLU: 2-4                         [-1, 64, 160, 100, 100]   --
|    └─Conv3d: 2-5                       [-1, 64, 160, 100, 100]   110,656
|    └─ReLU: 2-6                         [-1, 64, 160, 100, 100]   --
|    └─MaxPool3d: 2-7                    [-1, 64, 32, 100, 100]    --
├─Sequential: 1-2                        [-1, 64, 8, 100, 100]     --
|    └─Conv3d: 2-8                       [-1, 64, 32, 100, 100]    110,656
|    └─ReLU: 2-9                         [-1, 64, 32, 100, 100]    --
|    └─Conv3d: 2-10                      [-1, 64, 32, 100, 100]    110,656
|    └─ReLU: 2-11                        [-1, 64, 32, 100, 100]    --
|    └─Conv3d: 2-12                      [-1, 64, 32, 100, 100]    110,656
|    └─ReLU: 2-13                        [-1, 64, 32, 100, 100]    --
|    └─MaxPool3d: 2-14                   [-1, 64, 8, 100, 100]     --
├─Sequential: 1-3                        [-1, 64, 4, 100, 100]     --
|    └─Conv3d: 2-15                      [-1, 64, 8, 100, 100]     110,656
|    └─ReLU: 2-16                        [-1, 64, 8, 100, 100]     --
|    └─Conv3d: 2-17                      [-1, 64, 8, 100, 100]     110,656
|    └─ReLU: 2-18                        [-1, 64, 8, 100, 100]     --
|    └─Conv3d: 2-19                      [-1, 64, 8, 100, 100]     110,656
|    └─ReLU: 2-20                        [-1, 64, 8, 100, 100]     --
|    └─MaxPool3d: 2-21                   [-1, 64, 4, 100, 100]     --
├─Sequential: 1-4                        [-1, 64, 2, 100, 100]     --
|    └─Conv3d: 2-22                      [-1, 64, 4, 100, 100]     110,656
|    └─ReLU: 2-23                        [-1, 64, 4, 100, 100]     --
|    └─Conv3d: 2-24                      [-1, 64, 4, 100, 100]     110,656
|    └─ReLU: 2-25                        [-1, 64, 4, 100, 100]     --
|    └─Conv3d: 2-26                      [-1, 64, 4, 100, 100]     110,656
```

| └─ReLU: 2-27 [-1, 64, 4, 100, 100] --
| └─MaxPool3d: 2-28 [-1, 64, 2, 100, 100] --
├─Sequential: 1-5 [-1, 64, 1, 100, 100] --
| └─Conv3d: 2-29 [-1, 64, 2, 100, 100] 110,656
| └─ReLU: 2-30 [-1, 64, 2, 100, 100] --
| └─Conv3d: 2-31 [-1, 64, 2, 100, 100] 110,656
| └─ReLU: 2-32 [-1, 64, 2, 100, 100] --
| └─Conv3d: 2-33 [-1, 64, 2, 100, 100] 110,656
| └─ReLU: 2-34 [-1, 64, 2, 100, 100] --
| └─MaxPool3d: 2-35 [-1, 64, 1, 100, 100] --
├─Sequential: 1-6 [-1, 2, 1, 100, 100] --
| └─Conv3d: 2-36 [-1, 2, 1, 100, 100] 3,458
| └─ReLU: 2-37 [-1, 2, 1, 100, 100] --
==========
Total params: 1,554,434
Trainable params: 1,554,434
Non-trainable params: 0
Total mult-adds (G): 509.31
==========
Input size (MB): 6.10
Forward/backward pass size (MB): 3017.73
Params size (MB): 5.93
Estimated Total Size (MB): 3029.76
==========

3. 模型训练

3.1 训练设备

本项目的代码部分基于 pytorch 实现，安装了 CUDA 与 GPU 版本的 torch，使得模型得以使用设备本身的 GPU 资源。与在此之前一直使用的 CPU 版本的 torch 的训练速度相比有非常明显的提升。

3.2 数据分配

在训练模型时由于测试集的 gt 并未给出，因此便对已有数据进行了 8:1:1 的划分，即 1-160 组的数据作为训练集，161-180 组的数据作为验证集，181-200 组的数据作为测试集。

3.3 Minibatch

由于电脑的 GPU 资源实在是少的可怜，训练时采用 32 的 batch_size 就是本机的极限了，故在训练时的 batch_size 取为 32。

3.4 数据增强

本项目对直接投影的图像与直方图均衡化的图像分别进行了训练。在将这两种图像数据输入模型进行训练时均进行了裁剪(64×64)，旋转与翻转等数据增强处理，在这个过程中，为了保持训练图像与 gt 之间的一致性，本项目使用了 albumentations 这个库，达到了对训练图像与 gt 进行同步处理的目的。

3.5 损失函数

3.5.1 1-IoU

刚开始本项目采用自行编写的 1-IoU 作为 loss 函数进行梯度下降，但不知道为什么 loss 的值一直没有变化。在请教过助教后发现是由于直接利用 argmax 取得预测值导致无法计算梯度，导致模型无法更新造成的。故将网络输出经过 sigmoid 函数，通道数改为 1，将输出直接变为像素值为前景的概率。将此输出输入自行编写的 loss 函数中进行模型训练，发现 loss 开始下降，模型不断更新。

3.5.2 CrossEntropy

除了自行编写的 loss 函数之外，还用了 torch.nn.CrossEntropy()作为损失函数进行模型训练，经过测试，两种损失函数训练出的模型效果相似，具体详见结果展示。

3.6 学习率调整

在机器学习的学习过程中以及在完成项目找资料的过程中，了解到学习率在深度学习和其它一些循环迭代算法中，都非常重要。在效率上，它几乎是与算力同等重要的因素；在效果上，它也决定着模型的准确率。如果设置太小，则收敛缓慢，也有可能收敛到局部最优解；设置太大又导致上下摆动，甚至无法收敛。因此本项目在训练过程中对学习率进行更新，每隔 20 epochs 调整学习率为 lr=lr*0.9。

3.7 防止过拟合

为了防止模型过度训练造成过拟合，本项目在训练模型时加入了早停机制，当验证集的指标在一定的 epoch 内没有提升就终止训练。

4. 模型评估

4.1 指标计算

模型测试集为 181-200 组数据，对所有像素预测的 IoU 以及 BAC 求平均值即为模型最后的指标结果。

4.2 预测结果可视化

将待预测图像裁剪成训练数据的大小(64×64)，按照一定的顺序与 Height_stride、Width_stride 裁剪图像进行预测。将裁剪预测后的图像按顺序叠加拼接即可得到原来一整张图像的分割结果。将该结果保存为图片形式即可得到血管分割结果。裁剪后进行预测的图像会有重叠，即同一像素会有几组预测值，最终该像素的预测值由多组预测值的平均值决定，

大于等于 0.5 预测为前景，否则预测为背景。

5. 结果展示

5.1 性能指标

	Dice	BAC
直接投影（CrossEntropy）	0.8188	0.9037
直接投影（IoU）	0.8167	0.8998
直方图均衡化（CrossEntropy）	0.8304	0.9009

可以发现三个模型的性能指标相差不大。

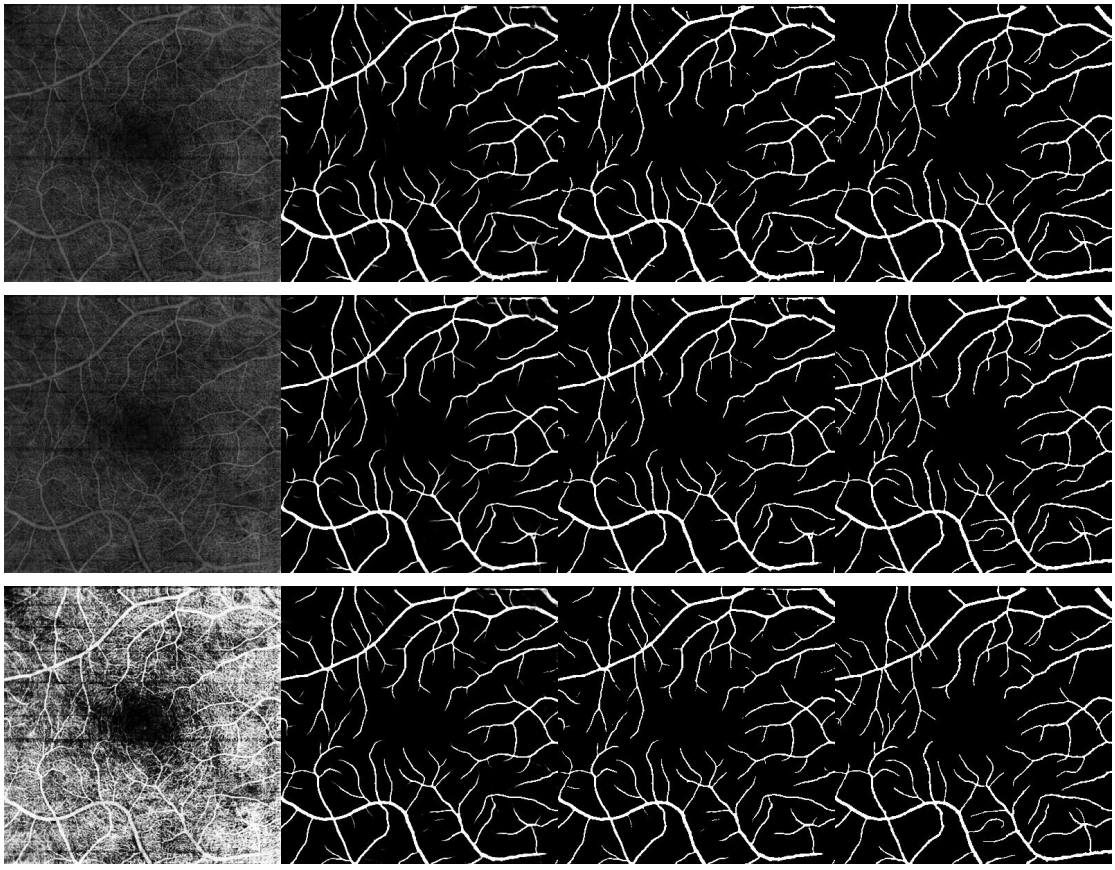
5.2 可视化

以下三张图片左边第一张为原始图片，中间两张为检测分割结果，右边为 gt。

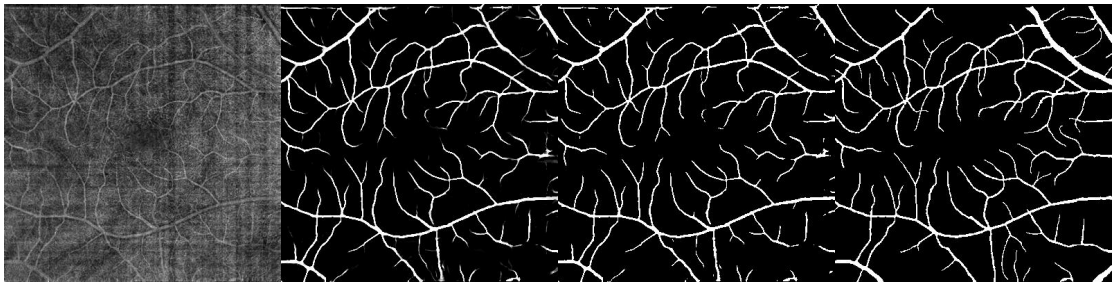
图一为直接投影后用 CrossEntropy 训练的模型。

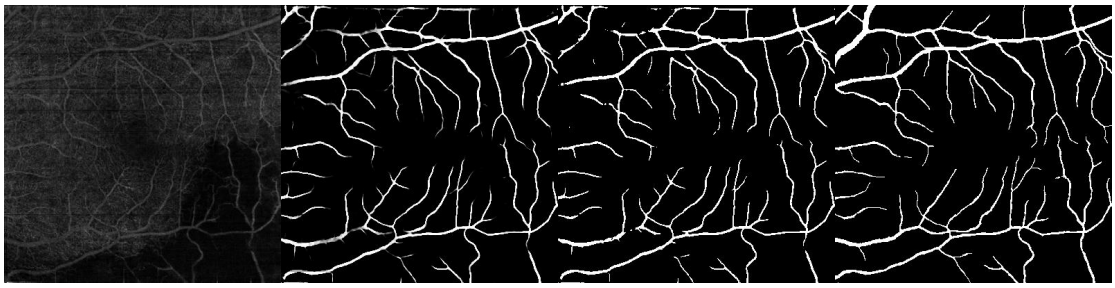
图二为直接投影后用 1-IoU 训练的模型。

图三为直接投影经直方图均衡化后用 CrossEntropy 训练的模型。



图四、五是在最后测试集上的结果展示：





四、项目感想

耗时一个多月终于完成了机器学习工程实践这个项目，只能说是收获良多。一个月以来，我查了许多资料，看了许多代码，踩了许多坑，有了许多深度学习上的第一次。我第一次自己成功的配置了所有需要的环境，我第一次从数据处理到结果可视化地搭建并成功训练了一个深度学习的模型（虽然效果确实一般）。我知道了怎么去给自己的数据写一个 Dataset 然后传入 Dataloader 在每次训练时生成数据，我知道了搭建模型的常用方法，我知道了训练模型时如何自己实现损失函数（一定要考虑到梯度反传的问题），我知道了有了模型之后如何利用模型完整地完成自己的预测任务……在实践过程中我知道了很多理论的实现细节，熟悉了 pytorch 这个工具的基本用法，遇到了许多问题，也尝试着自己去解决了许多的问题。当代码终于运行起来，终于看到正常的结果之后，真的会非常开心。很感谢这门两学分但是只上过开头四节课的实践课，我觉得它带给我的收获是比大一两门人工智能的课加起来还要多的。非常感谢倪老师每周认真负责地组织答疑和疫情期间贴心的减负，非常感谢荣学长每次及时的回复与帮助。虽然由于能力和精力问题，我只能在基础任务的基础上做了一些没有什么大用的调整，但这次算是自己一步步实现的项目还是让我挺快乐的。

五、参考资料

代码主要借鉴了以下两个网址：
https://github.com/chaosallen/IPN_tensorflow
https://github.com/lee-zq/VesselSeg-Pytorch