北京交通大学

“模式识别与机器学习”专题研讨

题目	肺部 MRI 图像肺实质分割		
姓名	钟军凯	学号	22211374
指导教师	李艳凤		
日期	2025 年 5 月 3 日		

电子信息工程学院

1. 研讨内容

- **内容**：肺部 MRI 图像肺实质分割；
- **要求**：采用基于 CNN 的分割网络实现，如 Unet，测试图像数量大于 10；
- **实验数据**：肺实质分割数据文件夹下包含三个文件夹，“img_train”为训练用肺部 T2W-MRI 图像，“lab_train”为“img_train”中每幅的分割结果，使用该两个文件夹用于训练分类器。“img_test”为测试用肺部图像，注意该文件夹中 IM 前的标号为病例号，IM 后的标号为层号，在得到每层图像的分割结果后，可通过同一个病例不同层的位置关系对分割结果进行约束。

2. 总体描述

本次研讨要求采用基于 CNN 的分割网络实现对肺部 MRI 图像的肺实质的分割。医学图像分割的一个特点是对位置的要求较高，需要非常精确地将目标分割出来。由于传统 CNN 在训练过程中有着池化这一步，而池化会对卷积出来的特征图进行压缩，具体一点，就是在一个区域内选出一个值来代表这个区域，比如最大值、平均值等。选出一个值来代表一个区域，那么这个区域内的值的位置信息都会丢失，也就是说，池化会导致特征图中像素点的位置信息的丢失，所以传统 CNN 不适合用于进行医学图像分割。而且，在医学图像分割中，医学图像的数据量是比较小的，所以在数据量小的情况下，使用传统 CNN 也不是一个合理的选择。

通过查阅资料，发现有一种名为 U-net 的基于 CNN 的网络结构，这个网络结构是适合用于进行生物医学图像分割的，所以决定采用 U-net 来完成本次研讨的内容。

U-net 设置了对称的编码器和解码器加上跳跃连接的结构，编码器中某一层在池化之前的输出将跳跃连接至与该层编码器对称的某层解码器的输入中，也就是这一层解码器的输入是对称的某层编码器池化前的输出和上一层解码器的输出的合并。如图 1 所示为 U-net 的结构示例。

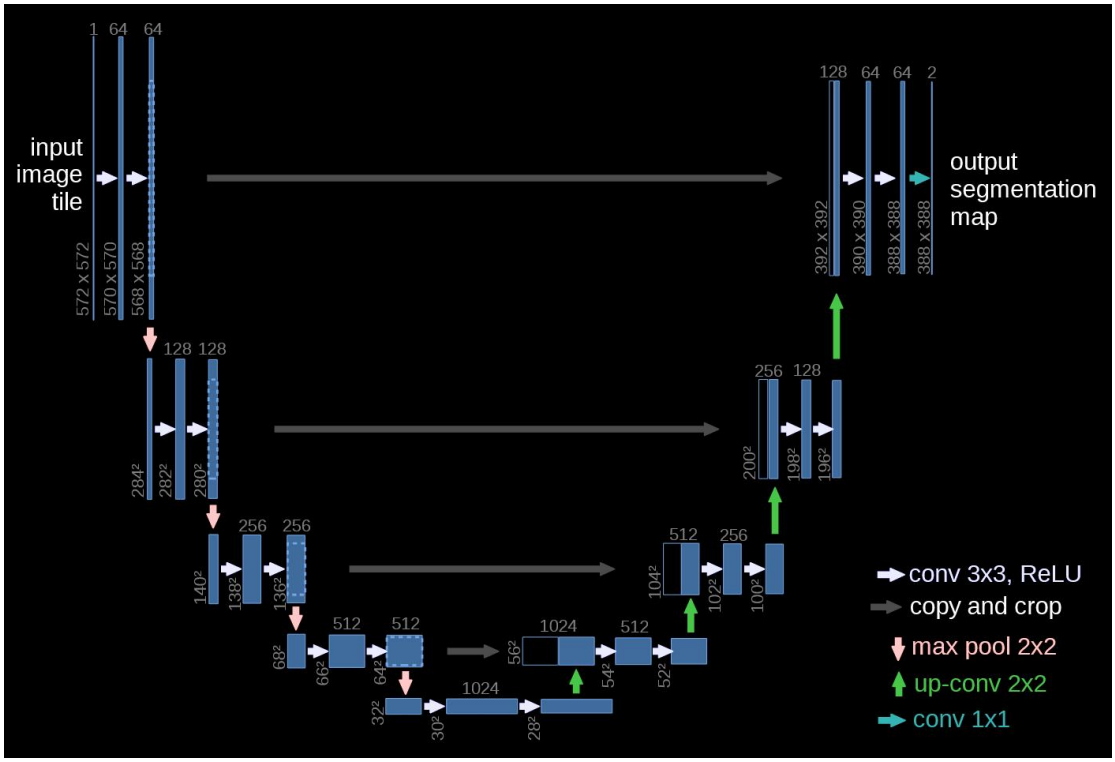

图 1：U-net 结构示例

本次研讨中，采用的网络结构与图 1 所示的网络结构相同，但是相关参数略有不同

3. 设计内容

在本次研讨中，我使用了 Pytorch 设计了如图 1 所示的网络结构，但是参数会不同。

因为肺部图像的训练集中，data 和 label 的大小是相同的，所以在进行卷积之前，需要对图像进行边缘填充，这样可以使得图像在卷积之后，图像的大小不会变化。在 Pytorch 中，可以使用 torch.nn.Conv2d()，来创建一个卷积层，其中的 padding 参数可以用来设置填充大小，填充大小由卷积核大小 kernel_size 决定，计算公式如（1）式所示：

$$\text{padding} = \frac{\text{kernel\_size} - 1}{2} \tag{1}$$

由于 U-net 中需要有跳跃连接，所以考虑这样设计：为编码器的一层和解码

器的一层定义一个类，也就是 UnetEncoderBlock 和 UnetDecoderBlock。在 UnetEncoderBlock 中，定义的结构为：两层卷积，每层卷积后面都有 ReLU 函数激活，然后放入 torch.nn.Sequential 中，池化单独定义，这样就可以得到池化之前的输出了，方便实现跳跃连接。解码器中的上采样和“卷积+激活”层也是分开定义的，方便接收跳跃连接。这两个类都会有 forward 函数。在 UnetEncoderBlock 类的 forward 函数中，会返回池化之前的结果和池化之后的结果，在 UnetDecoderBlock 类的 forward 函数中，其参数设置为上一层的输出和对应编码器池化前的输出，从而实现跳跃连接，然后返回最终的计算结果。

然后定义一个 Unet 类，然后使用 UnetEncoderBlock 和 UnetDecoderBlock 来创建编码器和编码器模块。在 Unet 的 forward 函数中，分别使用 UnetEncoderBlock 和 UnetDecoderBlock 实现每一层的计算和跳跃连接。

对于训练和测试，分别定义 train 函数和 test 函数。在 train 函数中，使用 Adam 优化器和 Pytorch 中的 StepLR 学习率规划器。使用的损失函数是 U-net 论文中使用的损失函数，这是一个基于交叉熵的损失函数，但是与交叉熵不相同，表达式如（2）式所示：

$$E=\sum_{x\in I} w(x)\log(p_{l(x)}(x)) \tag{2}$$

其实就是加入了权重的交叉熵，其中$w(x)$就是每一个像素的权重，权重的表达式如（3）式所示：

$$w(x)=w_c(x)+w_0\mathrm{e}^{-\frac{(d_1(x)+d_2(x))^2}{2\sigma^2}} \tag{3}$$

其中，$w_c(x)$每个像素所对应的类别（前景或背景）的出现频率；d_1是某个像素点和离它最近的边缘之间的距离；d_2是某个像素点和离它第二近的边缘之间的距离。w_0和σ是两个需要调整的参数。

d_1和d_2的获取思路：首先得到图像中所有的前景区域，可以使用 skimage.measure.label()得到，然后遍历所有前景区域，获取每个前景区域的边缘，

可以采用 skimage.segmentation.find_boundaries()得到，然后计算像素点到该边缘的距离，可以使用 scipy.ndimage.distance_transform_edt()计算。将所有计算得到的距离合成为一个 numpy 数组，使用 numpy.sort()进行排序，得到d_1和d_2。

观察（2）式，因为不是正常的交叉熵，所以在计算损失函数的时候，不能只是单纯的使用 torch.nn.CrossEntropyLoss()，需要将 torch.nn.CrossEntropyLoss 的 reduction 设置为 none，这样 torch.nn.CrossEntropyLoss 输出的才不会是一个值，而是一个张量，然后将输出的张量和计算出来的权重进行元素级乘法，得到一个加权张量，最终对该加权张量求均值，得到最终的损失函数计算值。

最后，为了充分利用 Pytorch 的特性，将训练集和测试集都定义为一个类，分别定义__len__方法和__getitem__方法。

4. 算法分析

4.1 算法优缺点

以下是 U-net 的优点：U-net 使用对称编解码的形式。编码器通过多次卷积和池化（下采样），多次提取图像特征，提取上下文信息；解码器通过多次上采样，可以逐步恢复分辨率。对称的结构保证了每一次下采样都有一次上采样对应，保证了不同尺度下都可以进行信息融合。U-net 还使用了跳跃连接的方式，也就是每一层编码器在池化之前的输出会与对应的解码器的上采样的输出拼接，得到最终解码器“卷积+激活”层的输入，这样就可在恢复分辨率的时候，得到因池化而丢失的位置信息。与此同时，U-net 也可以在少量数据的情况下达到比较优秀的性能，这与医学图像数据较少的特点契合。

以下是 U-net 的缺点：由于拥有对称编解码器的结构和跳跃连接，所以 U-net 中有着较多的网络参数，从而导致计算与内存的开销会比较大，在实验过程中，当 batch_size 大于 8，程序就会出现 cuda 的 out of memory 的错误。

4.2 算法参数选取

对于计算权重中w_0和σ的选取，论文中给出的推荐值是w_0等于 10 和σ等于

5，但是论文中的场景是对细胞进行分割，本次研讨是对肺实质进行分割，所以不能使用论文中的推荐值。在调试中发现，d_1和d_2的取值最大可以到 400，最小是 0，所以对于σ，考虑使用 0 到 400 的中间值，也就是σ等于 200。当σ太小，那么网络会更加聚焦于局部，此时对于边界的分割将会更加精确，所以会出现整个肺部都被分割出来了，这时为了体现出这一点，背景有可能会变为白色，但是我们需要的只是肺实质，所以σ不能太小；当σ太大，对于边缘的提取将会变得模糊，也就是可能所有边缘都分割不出来了。因为，最终σ取了一个中间值。对于w_0参数的选取，因为如果w_0太大，边缘元素的权重将会远大于其他元素，这将会使得边缘分割的更加精确，导致一些不需要的边缘也被分割出来；如果w_0太小，则$w_c(x)$的影响将会增大，而空间位置上的信息对权重的影响将会减小。最终，经过测试，选取了w_0等于 1。

对于 epoch 和 batch_size，batch_size 选取了 4，也就是一次处理 4 张图像，此时的训练效果相对于其他 batch_size 来说是较好的，而且当 batch_size 超过了 8，就会出现 cuda out of memory 的报错。epoch 则是选取了 50，因为训练集数据量较小，所以选择较大 epoch 会比较好，但是在训练过程中发现，训练到一定程度，损失函数就很难下降了，所以最终选择了 epoch 等于 50。

使用了 Adam 优化器和 StepLR 学习率规划器，所以对于学习率也要有相应的设置。在训练过程中，当学习率为 0.001 的时候，损失函数值的振荡程度较高，当学习率为 0.001 的时候，损失函数的变化较为平稳，所以最终的 Adam 的初始学习率设置为了 0.0001。对于 StepLR，我设置为了每 10 个 epoch，学习率下降一半，这样可以使得在学习到一定程度之后，可以进行更加精细的学习。

5. 测试结果

以下列出部分测试结果。

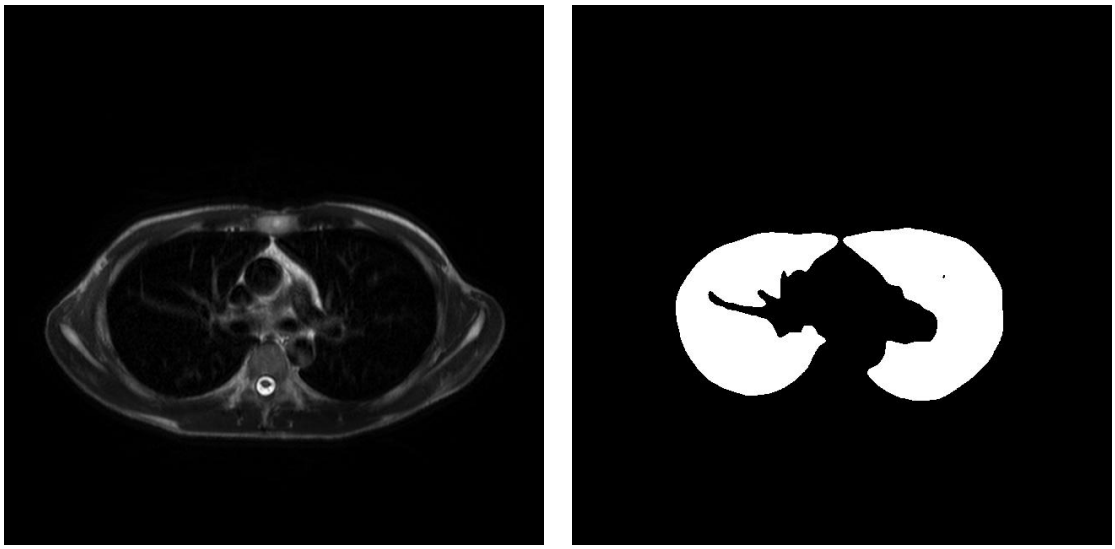

图 2：测试结果 1

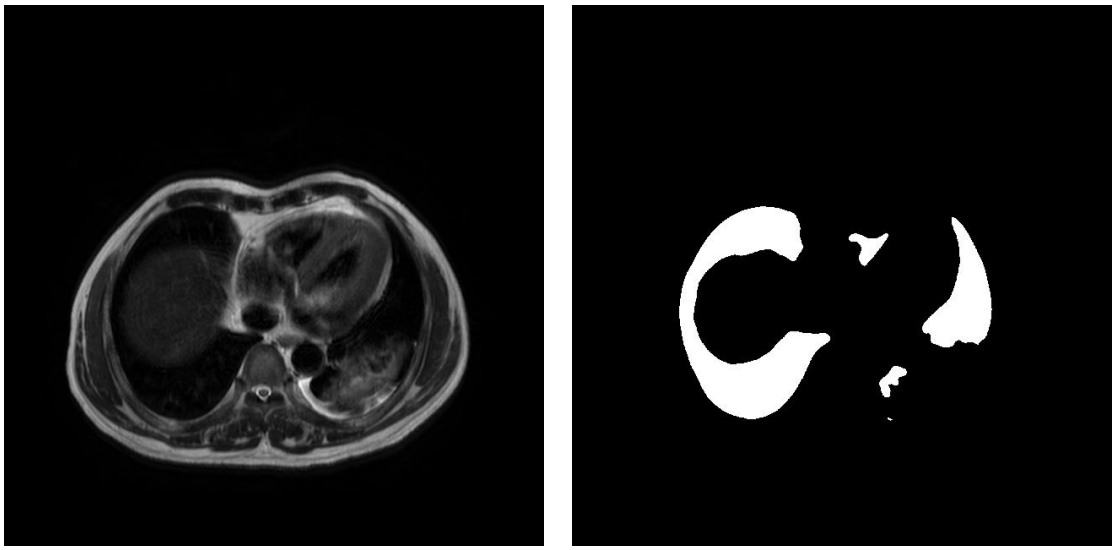

图 3：测试结果 2

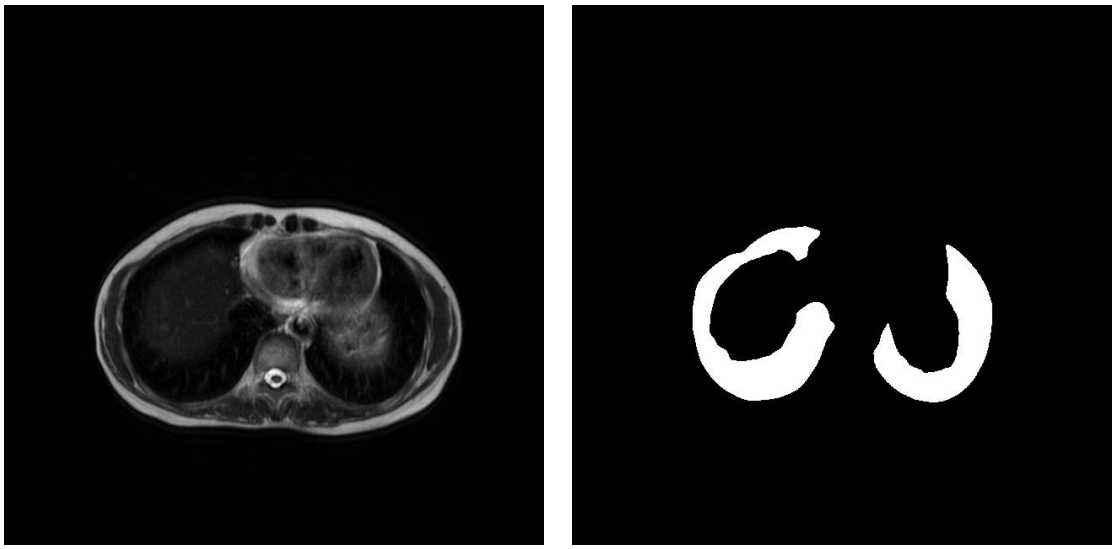

图 4：测试结果 3

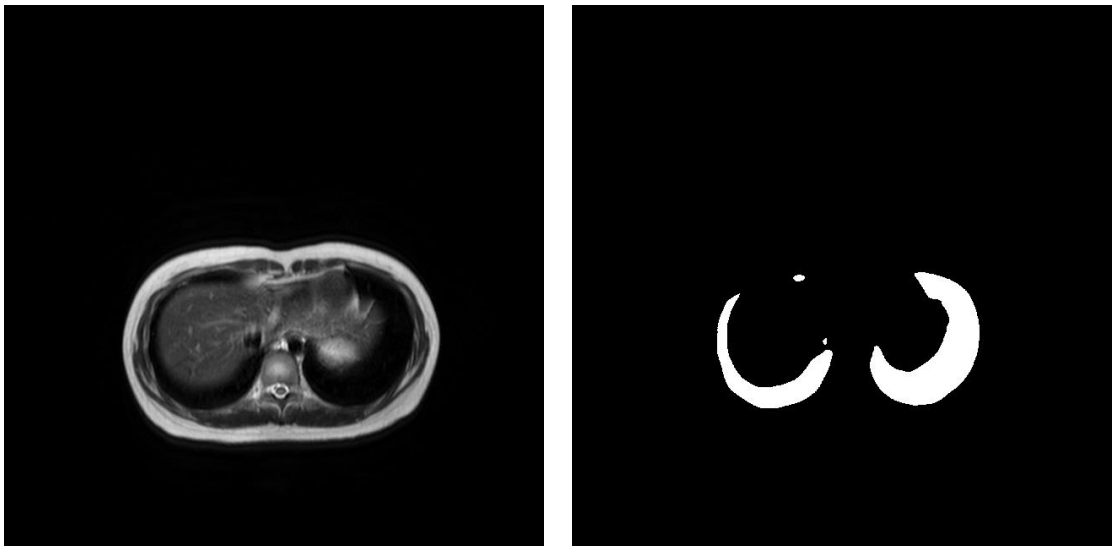

图 5：测试结果 4

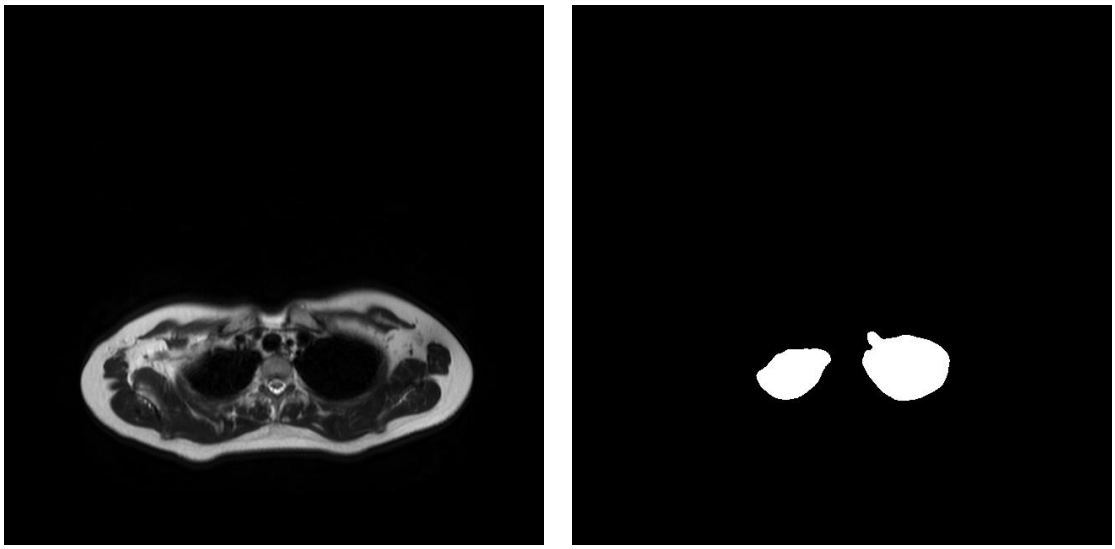

图 6：测试结果 5

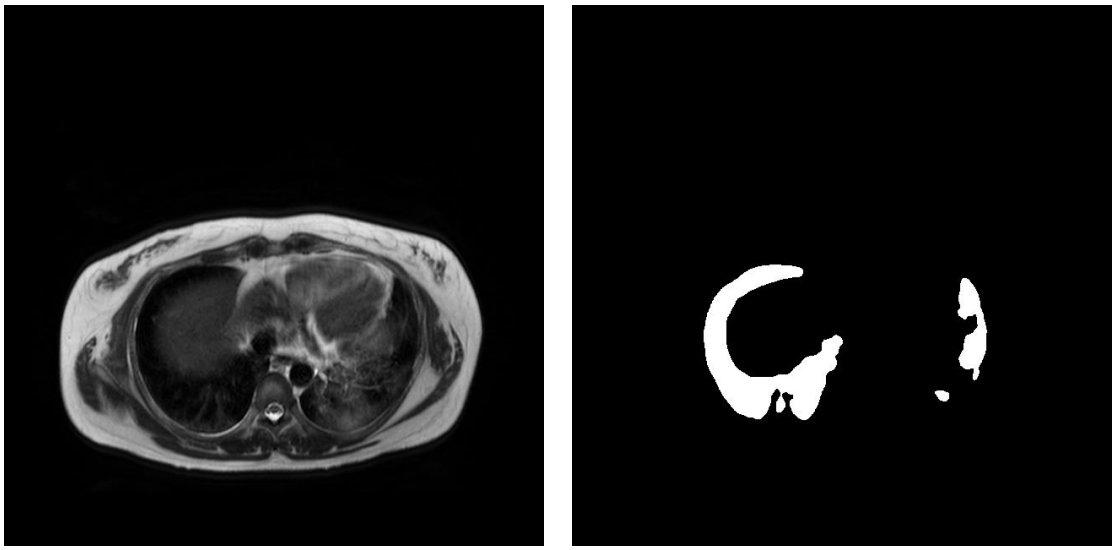

图 7：测试结果 6

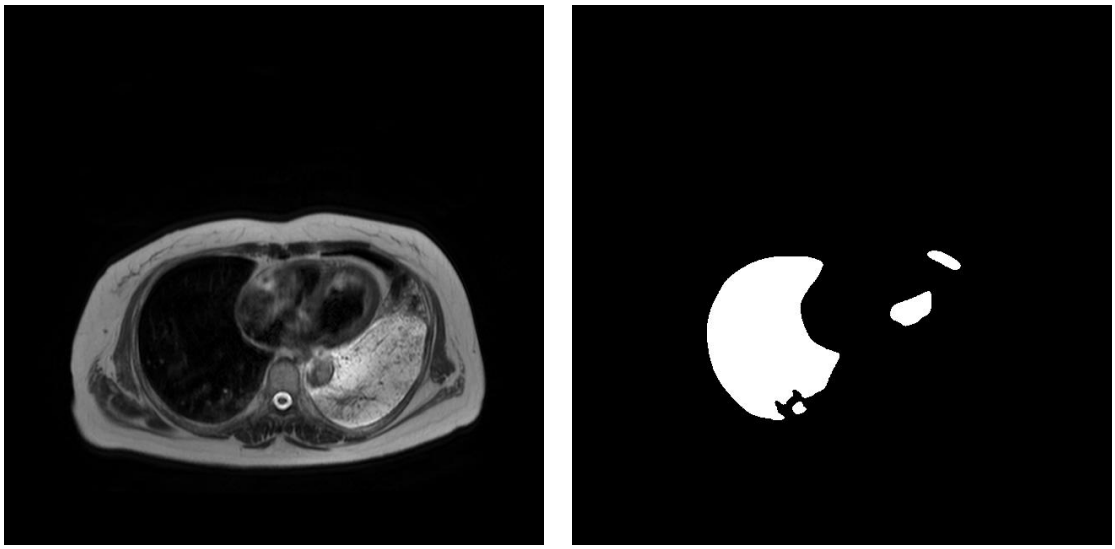

图 8：测试结果 7

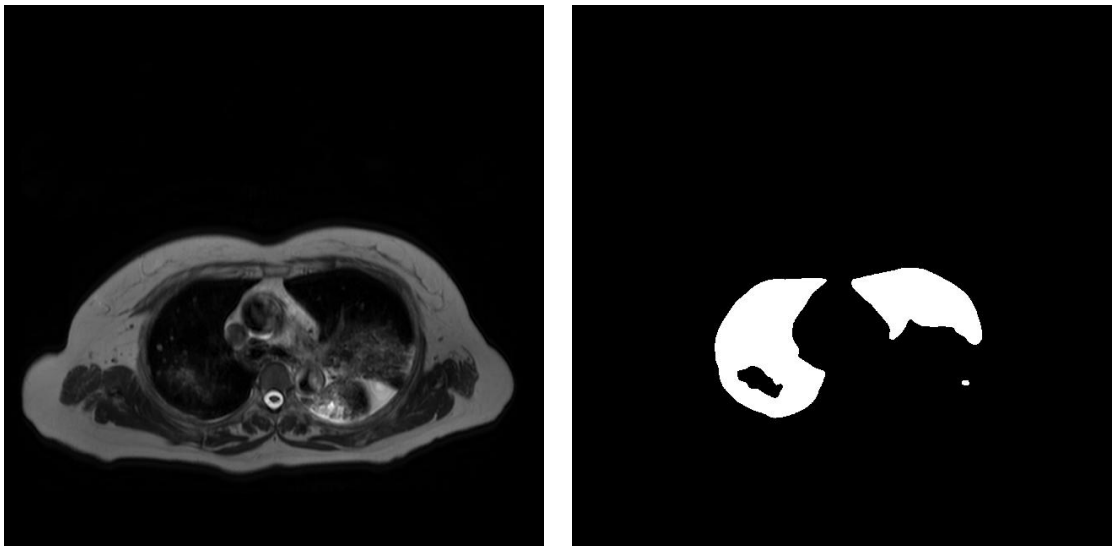

图 9：测试结果 8

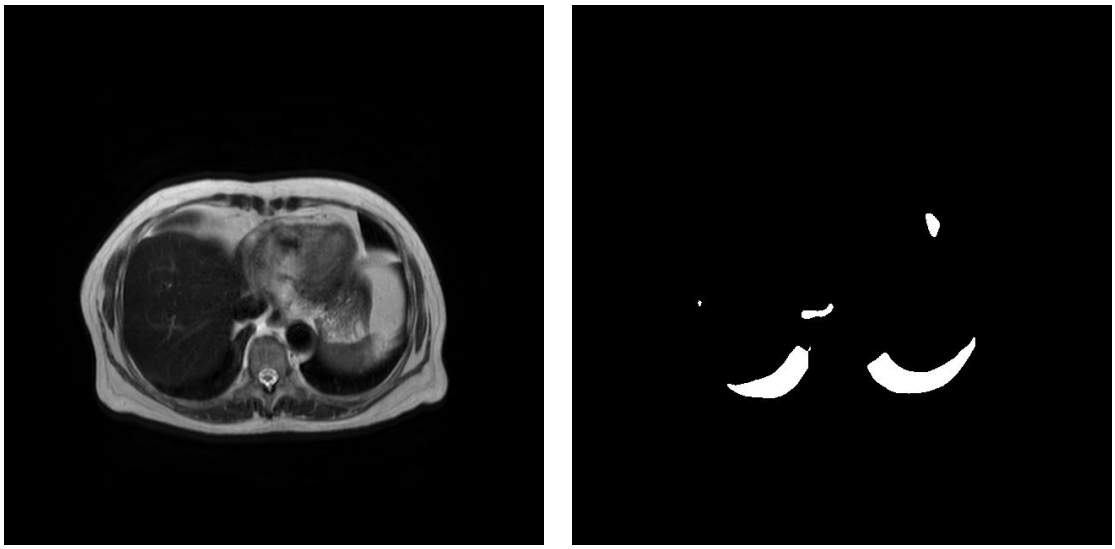

图 10：测试结果 9

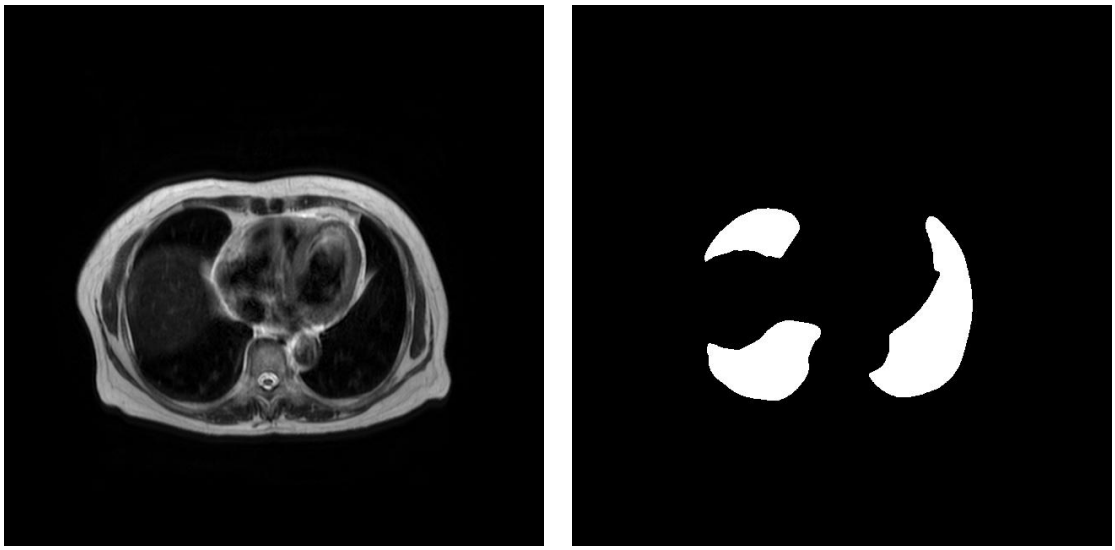

图 11：测试结果 10

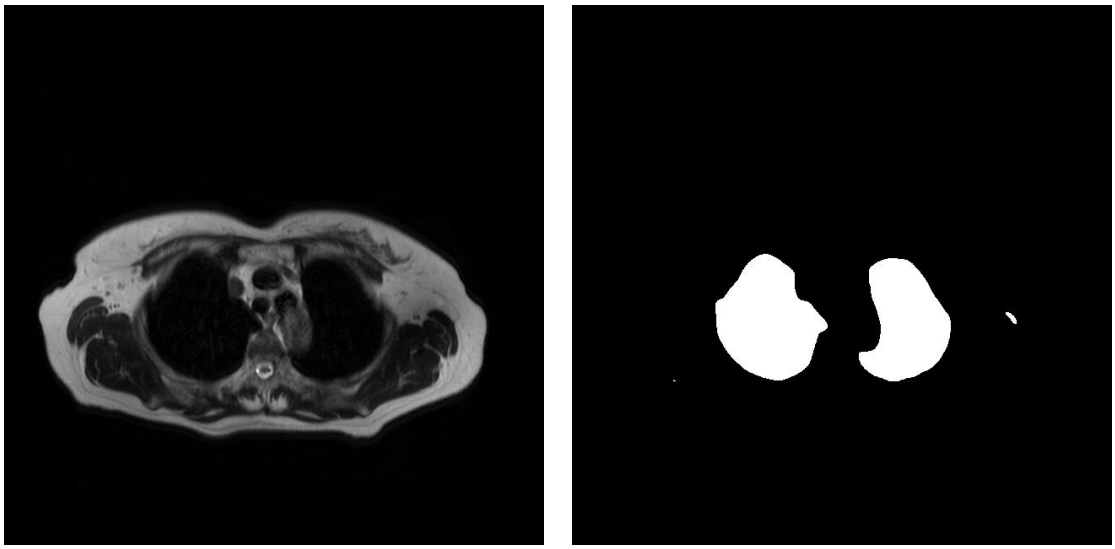

图 12：测试结果 11

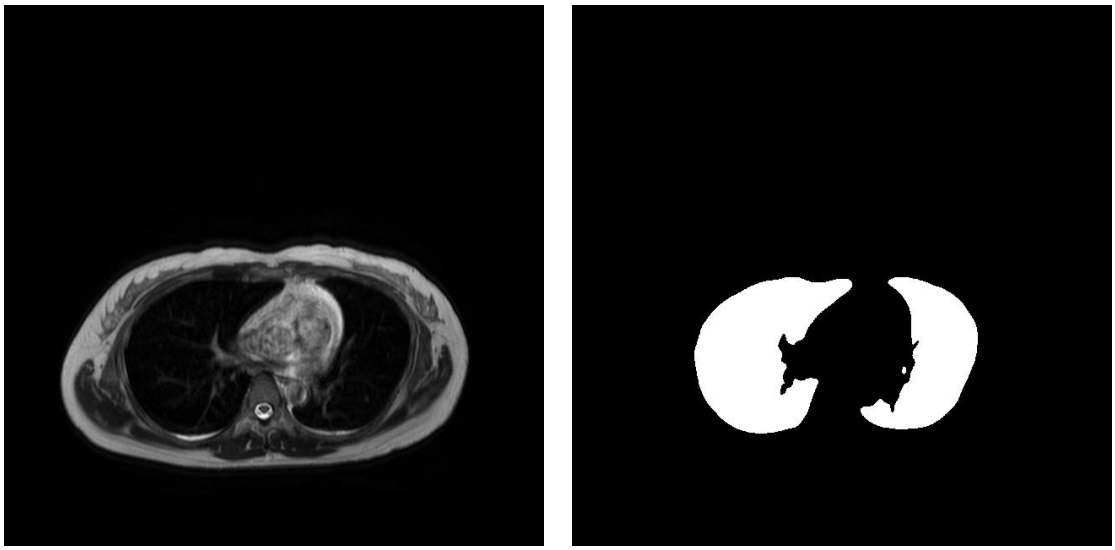

图 13：测试结果 12

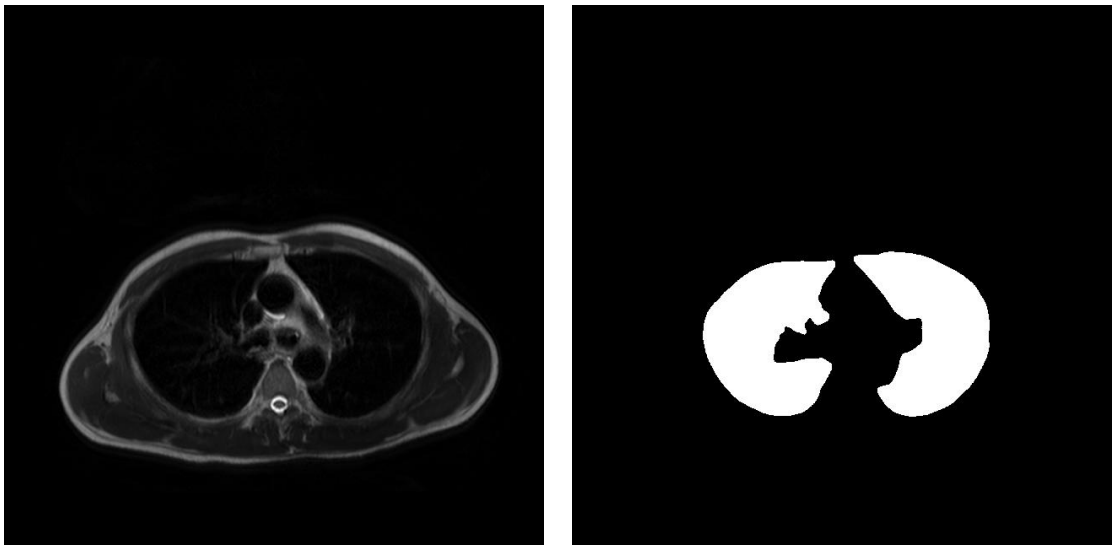

图 14：测试结果 13

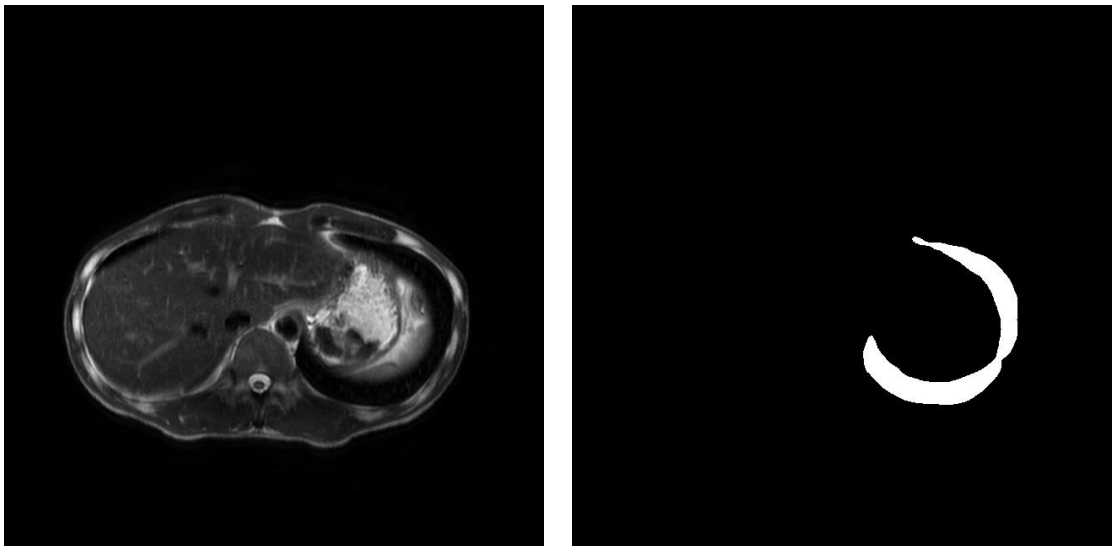

图 15：测试结果 14

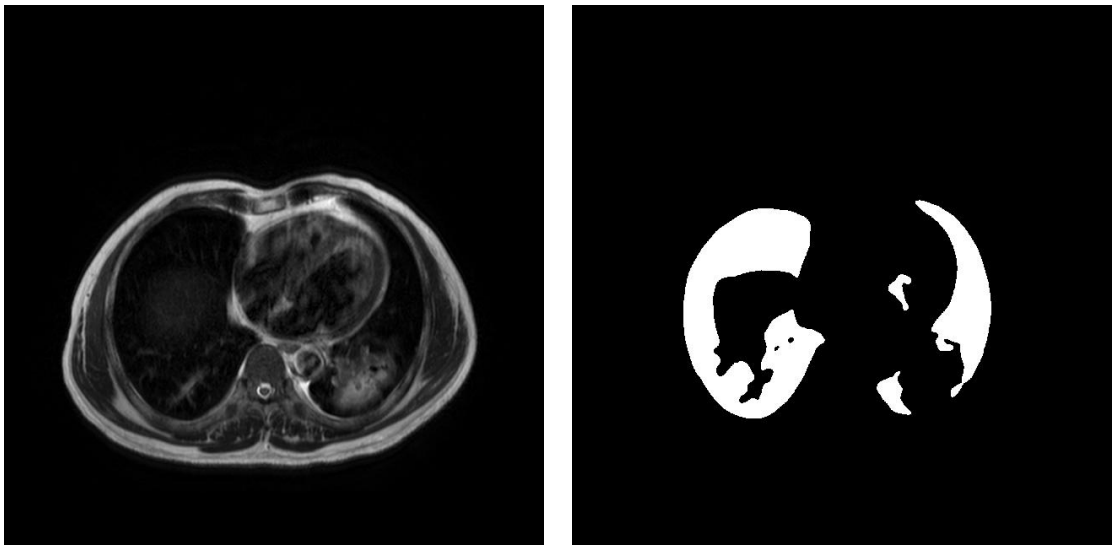

图 16：测试结果 15

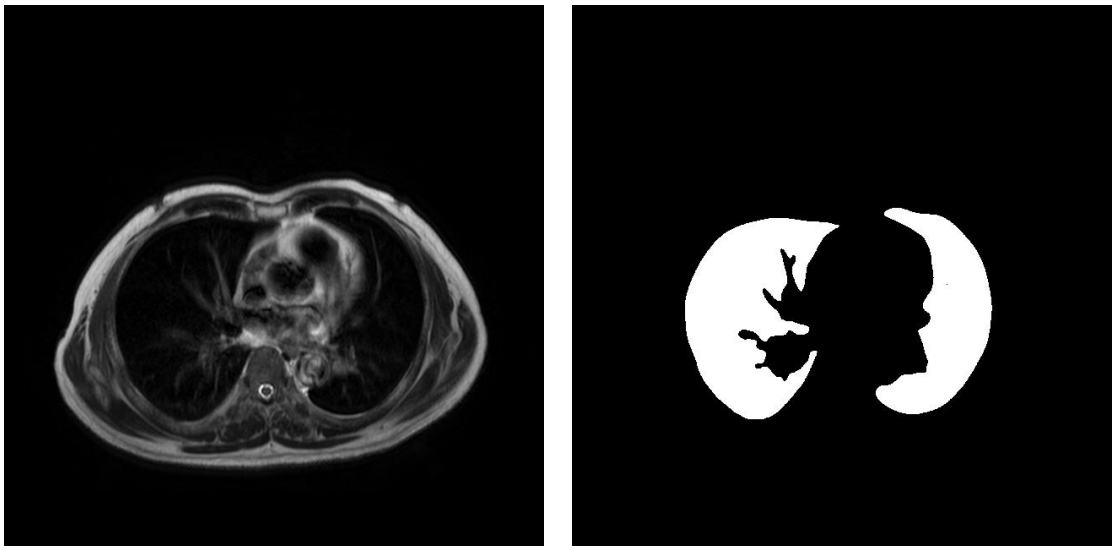

图 17：测试结果 16

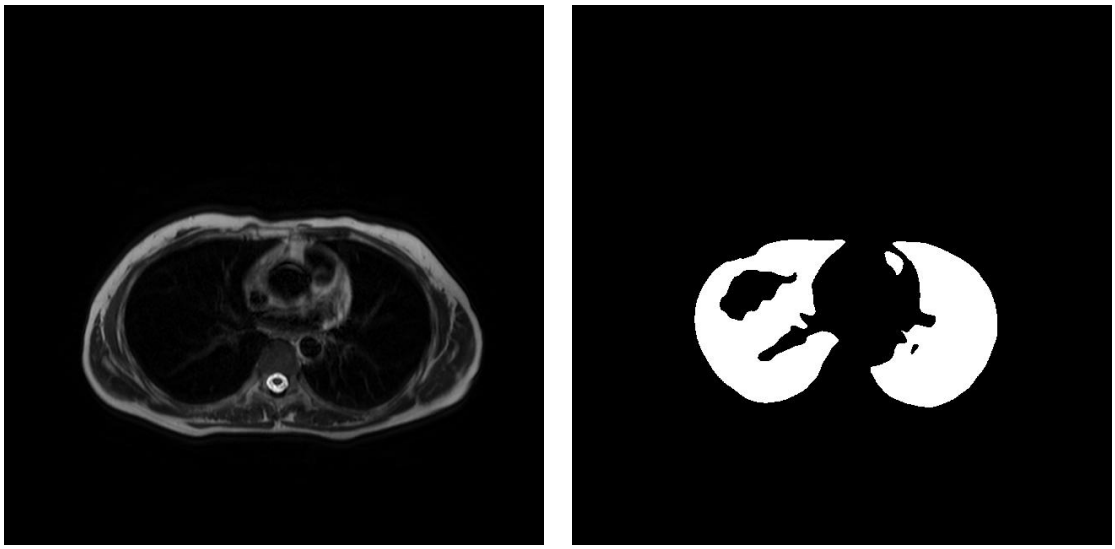

图 18：测试结果 17

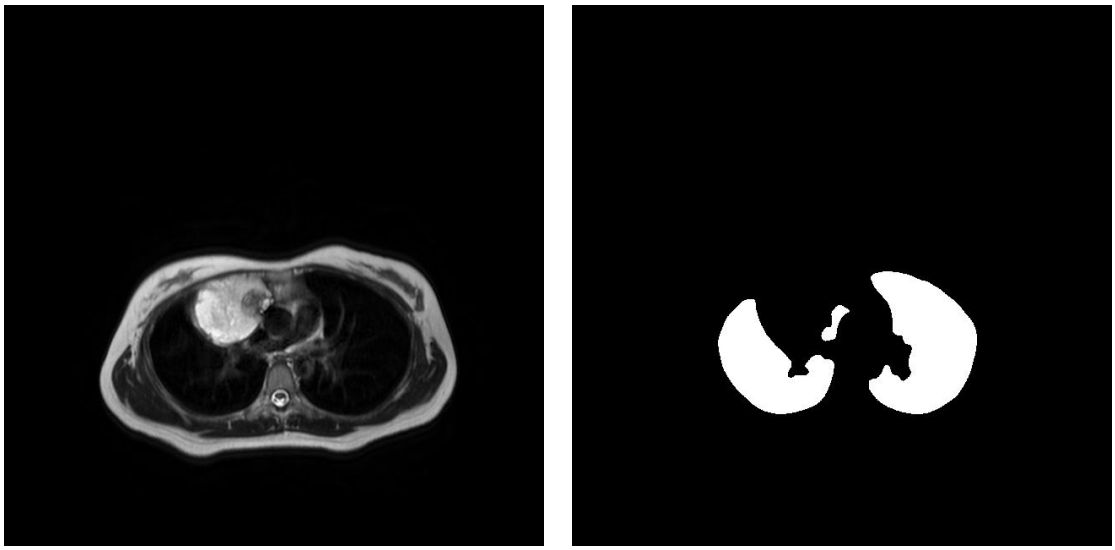

图 19：测试结果 18

6. 测试结果分析

大部分测试结果都能分割出肺实质的部分，但是有些分割结果会带有一些噪声，如图 3、8、12、16 所示，图像中与肺实质位置相近且颜色也相近的部分也会被当作肺实质。有些图像中，分割出来的肺实质会有缺失的部分，如图 18 所示，而且如果图像中的肺实质又被“遮住”，那么被“遮住”的部分可能不会被当作肺实质，如图 2、4、7、9、11、17 所示，所以我猜测图 18 也是由该原因导致，但是训练集中也有类似情况，所以该情况应该是正常的。

7. 程序源码

```
import os
import cv2 as cv
import numpy as np
import torch
import torch.nn as nn
import torch.optim as optim
from torch.optim.lr_scheduler import StepLR
from torch.utils.data import Dataset
from torch.utils.data import DataLoader
from matplotlib import pyplot as plt
import scipy
import skimage

def compute_weight (label, w_0 = 10, std = 5):
    # 统计前景背景元素数量
    label = label / 255
    label = label.astype(np.uint8)
    fg_num = np.count_nonzero(label == 1)  # 前景元素数量
    bg_num = label.size - fg_num  # 背景元素数量

    # 计算前景和背景的出现频率
    w_c = np.copy(label)
    w_c[w_c == 0] = 1 / bg_num
    w_c[w_c == 1] = 1 / fg_num

    # 得到不同的前景连通
    label_map = skimage.measure.label(label_image = label, connectivity = 2)

    # 计算像素到不同边缘的距离
    d_list = []
    for i in range(label_map.max()):
        fg_region = (label_map == (i + 1))
        edge = skimage.segmentation.find_boundaries(fg_region, mode = "inner")  # 得到边缘
        distance = scipy.ndimage.distance_transform_edt(edge == 0)  # 计算到边缘的距离
        d_list.append(distance)

    # 得到 d1 和 d2
    d = np.stack(d_list, axis = 0)
    if d.shape[0] == 1:  # 如果只有一个边缘，则 d1 和 d2 相等
        d1 = d[0, :, :]
```

```
            d2 = d[0, :, :]
        else:
            d = np.sort(d, axis = 0)   # 升序排序
            d1 = d[0, :, :]
            d2 = d[1, :, :]

        w = w_c + w_0 * np.exp(-((d1 + d2) ** 2) / (2 * std ** 2))
        w = w / (np.mean(w) + 1e-12)   # 使用均值归一化，让其均值保持为 1，加上 1e-12
防止出现除以 0
        return w


'''
@brief 训练集的类
'''
class TrainDataset (Dataset):
    def __init__ (self, data_train_dir, label_train_dir):
        super().__init__()

        self.data_train_dir = data_train_dir
        self.label_train_dir = label_train_dir

        self.data_train_list = os.listdir(data_train_dir)
        self.label_train_list = os.listdir(label_train_dir)

    def __len__ (self):
        # 返回训练集长度
        return len(self.data_train_list)

    def __getitem__ (self, index):
        # 得到图像路径
        data_train_path = os.path.join(self.data_train_dir, self.data_train_list[index])
        label_train_path = os.path.join(self.label_train_dir, self.label_train_list[index])

        # 读取图像，得到 numpy 数组
        data_train = cv.imread(data_train_path, cv.IMREAD_GRAYSCALE)
        label_train = cv.imread(label_train_path, cv.IMREAD_GRAYSCALE)

        # 统一大小
        data_train = cv.resize(data_train, (512, 512))
        label_train = cv.resize(label_train, (512, 512))

        weight = compute_weight(label_train, w_0 = 1, std = 200)
```

```
        # 归一化
        data_train = data_train / 255.0
        label_train = label_train / 255.0

        # 转换为 pytorch 的张量形式
        data_train_tensor = torch.from_numpy(data_train).float().unsqueeze(0)
        label_train_tensor = torch.from_numpy(label_train).long()
        weight_tensor = torch.from_numpy(weight).float()

        return data_train_tensor, label_train_tensor, weight_tensor

'''
@brief 测试集的类
'''
class TestDataset (Dataset):
    def __init__ (self, data_test_dir):
        super().__init__()

        self.data_test_dir = data_test_dir
        self.data_test_list = os.listdir(data_test_dir)

    def __len__ (self):
        # 返回测试集长度
        return len(self.data_test_list)

    def __getitem__ (self, index):
        data_test_path = os.path.join(self.data_test_dir, self.data_test_list[index])  # 得到图像路径
        data_test = cv.imread(data_test_path, cv.IMREAD_GRAYSCALE)  # 读取图像，得到 numpy 数组
        data_test = data_test / 255.0  # 归一化
        data_test_tensor = torch.from_numpy(data_test).float().unsqueeze(0)  # 转换为 pytorch 的张量形式

        return data_test_tensor, self.data_test_list[index]

'''
@brief Unet 编码器的一层的类
'''
class UnetEncoderBlock (nn.Module):
    def __init__ (self, in_channels, out_channels, conv_size = 3, conv_stride = 1):
        super().__init__()
        self.block = nn.Sequential(
            nn.Conv2d(
```

```
                    in_channels = in_channels,
                    out_channels = out_channels,
                    kernel_size = conv_size,
                    stride = conv_stride,
                    padding = int((conv_size - 1) / 2)
                ),
                nn.ReLU(),
                nn.Conv2d(
                    in_channels = out_channels,
                    out_channels = out_channels,
                    kernel_size = conv_size,
                    stride = conv_stride,
                    padding = int((conv_size - 1) / 2)
                ),
                nn.ReLU()
            )

            self.pool = nn.MaxPool2d(kernel_size = 2)

        def forward (self, input):
            output = self.block(input)
            output_pool = self.pool(output)
            return output, output_pool   # 前者用于跳跃连接，后者用于下一层的输入

'''
@brief Unet 解码器的一层的类
'''
class UnetDecoderBlock (nn.Module):
    def __init__ (self, in_channels, out_channels, conv_size = 3, conv_stride = 1):
        super().__init__()
        self.up_conv = nn.ConvTranspose2d(
            in_channels = in_channels,
            out_channels = out_channels,
            kernel_size = 2,
            stride = 2
        )

        self.block = nn.Sequential(
            nn.Conv2d(
                in_channels = in_channels,
                out_channels = out_channels,
                kernel_size = conv_size,
                stride = conv_stride,
                padding = int((conv_size - 1) / 2)
```

```
                ),
                nn.ReLU(),
                nn.Conv2d(
                    in_channels = out_channels,
                    out_channels = out_channels,
                    kernel_size = conv_size,
                    stride = conv_stride,
                    padding = int((conv_size - 1) / 2)
                ),
                nn.ReLU()
            )

    def forward (self, input, encoder_output):
        input_up_conv = self.up_conv(input)
        connect = torch.cat([input_up_conv, encoder_output], dim = 1)
        output = self.block(connect)
        return output

'''
@brief Unet 类
'''
class Unet (nn.Module):
    def __init__ (self):
        super().__init__()

        self.encoder1 = UnetEncoderBlock(1, 64)
        self.encoder2 = UnetEncoderBlock(64, 128)
        self.encoder3 = UnetEncoderBlock(128, 256)
        self.encoder4 = UnetEncoderBlock(256, 512)

        self.mid_trans = nn.Sequential(
            nn.Conv2d(512, 1024, kernel_size = 3, padding = 1),
            nn.ReLU(),
            nn.Conv2d(1024, 1024, kernel_size = 3, padding = 1),
            nn.ReLU()
        )

        self.decoder1 = UnetDecoderBlock(1024, 512)
        self.decoder2 = UnetDecoderBlock(512, 256)
        self.decoder3 = UnetDecoderBlock(256, 128)
        self.decoder4 = UnetDecoderBlock(128, 64)

        self.final_trans = nn.Conv2d(64, 2, kernel_size = 1)
```

```
    def forward (self, input):
        en_output_1, en_output_1_pool = self.encoder1(input)
        en_output_2, en_output_2_pool = self.encoder2(en_output_1_pool)
        en_output_3, en_output_3_pool = self.encoder3(en_output_2_pool)
        en_output_4, en_output_4_pool = self.encoder4(en_output_3_pool)

        mid_trans_output = self.mid_trans(en_output_4_pool)

        de_output_1 = self.decoder1(mid_trans_output, en_output_4)
        de_output_2 = self.decoder2(de_output_1, en_output_3)
        de_output_3 = self.decoder3(de_output_2, en_output_2)
        de_output_4 = self.decoder4(de_output_3, en_output_1)

        final_trans_output = self.final_trans(de_output_4)

        return final_trans_output

'''
@brief 模型训练
@param model: 需要训练的模型
@param dataloader: 数据加载器
@param optimizer: 优化器
@param device: 运行设备
@return loss_list: 一轮 epoch 的损失函数列表
'''
def train (model, dataloader, optimizer, device):
    model.to(device)
    loss_func = nn.CrossEntropyLoss(reduction = 'none')  # 损失函数为交叉熵
    loss_list = []

    for inputs, targets, weight in dataloader:
        # 将数据放到 device 上
        inputs = inputs.to(device)
        targets = targets.to(device)
        weight = weight.to(device)

        # 实际训练过程
        optimizer.zero_grad()
        outputs = model(inputs)
        loss = loss_func(outputs, targets)
        loss = weight * loss
        loss = loss.mean()
        loss.backward()  # 反向传播
        optimizer.step()
```

```
            print(f"当前学习率：{scheduler.get_last_lr()[0]}")
            print(f"损失函数计算值：{loss.item()}")
            loss_list.append(loss.item())

        return loss_list

'''
@brief 模型测试
@param model: 需要测试的模型
@param dataloader: 数据加载器
@param device: 运行设备
@return outputs: 测试结果列表
@return file_names: 测试文件的文件名
'''
def test (model, dataloader, device):
        model.eval()  # 测试模式
        outputs = []  # 输出值列表
        file_names = []  # 用于测试的文件名的列表

        with torch.no_grad():  # 没有梯度计算
            for inputs, file_name in dataloader:
                inputs = inputs.to(device)
                output = model(inputs)
                outputs.append(output)
                file_names.append(file_name)

        return outputs, file_names

if __name__ == '__main__':
        epoches = 50  # 训练次数
        batch_size = 4  # 批大小
        loss_lists = []  # 保存每轮 epoch 的 loss 列表的列表

        # 创建训练集的加载器
        data_train_dir = "./lung_seg_data/img_train"
        label_train_dir = "./lung_seg_data/lab_train"
        lung_data = TrainDataset(data_train_dir, label_train_dir)
        lung_dataloader = DataLoader(lung_data, batch_size = batch_size, shuffle = True)

        # 创建测试集的加载器
        data_test_dir = "./lung_seg_data/img_test"
        test_data = TestDataset(data_test_dir)
        test_dataloader = DataLoader(test_data, batch_size = batch_size)
```

```
    # 创建一个 Unet
    u_net = Unet()

    # Adam 优化器
    optimizer = optim.Adam(params = u_net.parameters(), lr = 0.0001)

    # 学习率规划器
    scheduler = StepLR(optimizer, 10, 0.5)

    # 模型运行设备
    device = "cuda" if torch.cuda.is_available() else "cpu"

    # 训练
    for epoch in range(epoches):
        loss_list = train(u_net, lung_dataloader, optimizer, device)
        scheduler.step()  # 规划学习率
        loss_lists.append(loss_list)
        torch.cuda.empty_cache()  # 清空缓存
        print(f"完成第{epoch + 1}次训练")

    # 测试
    print("正在测试")
    test_output, test_file_name = test(u_net, test_dataloader, device)
    print("完成测试")

    # 对测试结果进行处理，得到分割结果
    test_output = torch.cat(test_output, dim = 0)
    test_result = torch.argmax(test_output, dim = 1)
    output_array = test_result.cpu().numpy()
    output_array = output_array * 255
    output_array = output_array.astype(np.uint8)

    # 测试结果保存的路径
    test_result_dir = "./test_result"
    os.makedirs(test_result_dir, exist_ok = True)

    # 由于 batch_size 是 4，test 函数返回的列表的元素是一个长度为 4 的元组，现在将元素改为单个文件名
    test_file_name = [name for batch_size in test_file_name for name in batch_size]

    # 写入测试结果
    print("正在写入图片")
    for i in range(output_array.shape[0]):
```

```
        path = os.path.join(test_result_dir, test_file_name[i])
        cv.imwrite(path, output_array[i, :, :])

print("写入完成")

# 保存模型
save_path = "./unet_model.pth"
torch.save(u_net.state_dict(), save_path)

# 绘制每个 epoch 的 loss 曲线，画在一张图中
for i in range(epoches):
        t = range(len(loss_lists[i]))
        plt.plot(t, loss_lists[i])

plt.show()
```