3. 心室分割与心脏病分类

3.1 基本信息

3.1.1 成员及分工

队长：黄元通　　学号：20195063　　工作量：60%

组员：邱聖邦　　学号：20195263　　工作量：30%

组员：朱梓铭　　学号：20195237　　工作量：10%

分工：

黄元通：数据加载、分割网络及改进、分割分类联合学习网络；

邱聖邦：独立的分类；

朱梓铭：图像增强、图像预处理。

3.1.2 概要

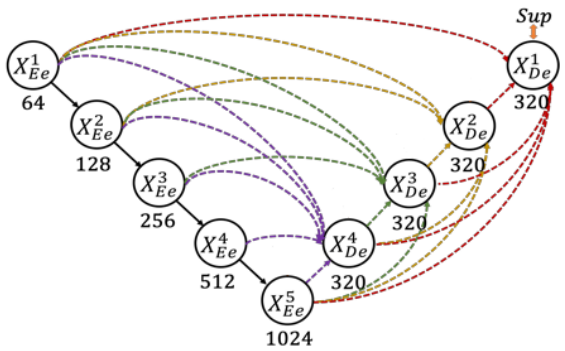
本次实验主要围绕“分割网络及改进”、“联合学习网络设计”、“模型训练”、“独立的分类方案”四大部分开展。

分割网络及改进部分介绍并测试了 UNet3+、使用 ResNet50 作为 backbone 的 UNet3+两种模型；联合学习网络设计部分提出和介绍了三种模型搭建方案及其相应实验；模型训练部分探究了在有限算力下的模型训练替代方案。

3.2 分割网络及改进

3.2.1 UNet3+网络的实现

模型以 UNet3+论文提出者的 Github 的代码为基础，对冗余代码进行了修改，主要为重写了解码器层、新参数初始化方式。基础 Unet3+结构如图 3.1 所示。



(c) UNet 3+

(Full-scale skip connections)

图 3.1 UNet3+网络结构

1. Decoder 类的实现：

因为原代码中解码器模块搭建采用的是在 UNet3+类定义时，对网络中共 20 条 skip connection 一条一条地连接，导致原代码不仅冗余度非常高，而且可读性极差。因此，我将解码器模块以解码器层（共 4 层）为单位，定义了新的类 Decoder(nn.Module)，以实现将 UNet3+类定义从原本 242 行的高冗余性代码缩减到了 55 行，且极大提高了类定义代码的可读性。

因为每个解码器层的输入都是 5 个尺寸各异的特征向量，skip connection 根据特征来自的层、和目标层间的特征维度差异，决定该线路是采用 MaxPool2d 下采样、Upsample 上采样、还是直接拼接。

以网络整体深度为 5 为例。记当前解码器层所在深度为 n_decoder，例如如果当前实例化的是X_{De}^{4}层，那么$n_decoder = 4$，将X_{De}^{4}层接收的来自$X_{Ee}^{1}, X_{Ee}^{2}, X_{Ee}^{3}, X_{Ee}^{4}, X_{Ee}^{5}$各层特征按照其上标顺序（也就是层所在深度，这也决定着该特征较原图的下采样率）分别记为$1, 2, 3, 4, 5$。那么前$n_decoder - 1$个 skip connection 为进行下采样，第$decoder$个 skip connection 为进行直接拼接，后$5 - n_decoder$个 skip connection 为进行上采样。其他所有解码器层同样符合这一规律。

因此在 Decoder 类中，只需执行为 5 次的循环，根据 scale 与 1 的关系即可判定改 skip connection 所需采取的方案及上/下采样所需倍率，其中 scale 计算方式为：

$$down\_cale = [1,2,4,8,16]$$

$$scale = down\_scale[n\_decoder - 1]/down\_scale[i]$$

该过程核心代码如下：

```
for i in range(5):
    # 确定上、下采样倍率差
    scale = int(down_scale[n_decoder - 1] / down_scale[i])
    re_scale = int(down_scale[i] / down_scale[n_decoder - 1])
    # 根据倍率差决定下采样、平接、上采样
    conv = nn.Sequential(nn.MaxPool2d(int(scale), ceil_mode=True),
                         N_Conv(filters[i], catChannel, 1), )\
        if scale > 1 else N_Conv(filters[i], catChannel, 1) \
        if scale == 1 else \
        nn.Sequential(nn.Upsample(scale_factor=int(re_scale),
                                  mode='bilinear', align_corners=True),
                      N_Conv(filters[i] if i == 4 else upChannel,
                                  catChannel, 1), )
    setattr(self, f'connection_{i + 1}', conv)
```

2. 网络参数初始化：

将网络参数初始化统一到一个 weight_init 函数中，并通过 isinstance 函数判断当前层为 Linear、Conv2d 还是 BatchNorm2d 层，并相对应采用 xavier_normal_、kaiming_normal_和 constant_三种不同的参数初始化方案。

在网络实例化后，只需调用 net.apply(weight_init)函数，即可将 net 中所有网络层均按照 weight_init 函数中定义的方案初始化。

3.2.2 UNet3+网络的改进

实验中发现，使用基础 UNet3+网络时，在验证集上（选用的是数据集中的第 14、15 号病人的数据作为验证集）分割 F1 得分始终在 0.8 左右无法继续上升，因此尝试了将编码器部分换为 ResNet50、使用深监督、优化损失函数、图像增强、图像预处理等方面的实验。

1. 使用 ResNet50 作为编码器：

在 UNet3+论文中，作者使用了 ResNet101 作为网络的 backbone，因为实验平台的内存有限，为减小模型整体的内存消耗，实验了使用 ResNet50 作为 backbone 的网络结构。

基础 UNet3+编码器各层采用的是 DoubleConv：

$$DoubleConv = (convolution \Rightarrow [BatchNorm] \Rightarrow ReLU) * 2$$

具体实现时，采用的是将$X_{Ee}^{2}, X_{Ee}^{3}, X_{Ee}^{4}, X_{Ee}^{5}$各层中原本的采用的 DoubleConv 替换为实例化的一个 resnet50 中的 resnet.layer1、resnet.layer2、resnet.layer3、resnet.layer4 各模块。核心代码如下：

```
resnet = models.resnet50(False)
self.inc = N_Conv(n_channels, filters[0])
self.down1 = nn.Sequential(nn.MaxPool2d(2),
                           resnet.layer1, )
self.down2 = resnet.layer2
self.down3 = resnet.layer3
self.down4 = resnet.layer4
```

2. 使用深监督

Unet3+网络的深监督实现机制为从各解码器层取出特征向量，并上采样至原始图片尺寸。

具体实现时：

（1） 采用的是$convolution \Rightarrow UpSample$，其中卷积采用 3*3 卷积核，上采样为双线性插值，上采样倍率由原图尺寸除以特征向量尺寸计算得；

（2） 并在类定义中添加了 deep_supervise 参数，为 True、False 时分别表示“需要进行深监督”、“不需要进行深监督”；

（3） 为了统一两种情况的网络输出及损失计算，将 UNet3+网络输出改为列表，并将损失计算定义为对列表中各项依次计算损失并求和的过程，这样便可支持指定任意某几个层进行深监督（最后一层的输出默认放在第一个位置，其他层的深监督按照特征向量所在深度逆序排序）；

（4） 同时损失计算时添加 weight 参数（list），其各项为相应深监督层损失的占比（第一项为最后一层输出损失的占比，其他层的深监督按照特征向量所在深度逆序排序）。

例如，在X_{De}^{2}, X_{De}^{3}层进行深监督，那么在类定义中添加：

```
self.outconv3 = OutConv()
self.up3 = nn.Upsample(4, 'bilinear')
self.outconv2 = OutConv()
self.up2 = nn.Upsample(2, 'bilinear')
```

损失计算过程 Total_Loss 大致过程伪代码为：

```
def Total_Loss(y, outputs_list, weight):
    for i in range(len(outputs_list)):
        loss = weight[i] * (outputs_list[i]与 y[i]间的损失)
        total_loss = total_loss + loss
    return total_loss
```

那么便可通过设定 weight 的值，调整各深监督层损失影响的占比，例如设为 weight= [0.7, 0.2, 0.1]，那么$X_{De}^{3}, X_{De}^{2}, X_{De}^{1}$特征向量计算得的损失占比分别为 0.2、0.1、0.7。

通过控制参数实验，结果显示添加深监督确对提高模型准确率、增强训练稳定性有一定的改进作用，实验结果如表 3.1 所示：

表 3.1 无深监督、有深监督的网络在不同 epoch 下对比结果

类别	epoch	训练集 F1 得分	验证集 F1 得分
无深监督	10	71.87%	67.78%
	70	89.00%	83.69%
有深监督	50	88.30%	82.51%
	60	89.27%	83.15%
	70	**89.94%**	**84.40%**
	80	89.26%	83.88%

需要注意的是，每 10 个 epoch 保存一次模型，且保存的都是过去这 10 个

epoch 中总损失最低的 epoch 时的模型。从有深监督的实验可见，在 epoch 设定为 70 时，模型准确率达到这组实验中的最高，并且比无深监督的有十分微弱的提高，在训练集上提高了 0.94%，在验证集上提高了 0.71%。

3.3 数据加载

3.3.1 构建所有图片路径列表

考虑到数据集文件夹结构较为复杂，在定义数据加载类 Dataset(Dataset)时，将构建所有图片绝对路径索引列表的功能单独置于函数 Locate_All 中。

另外，就本数据集而言，因为每一个 Heart Data\Image_XXX\png\Image\XX 文件夹下的所有图片均为同一个患者的，对于联合学习模型总体结构有以下两种构想：

（1） 将每一个患者文件夹下的图片作为一个 batch 进行输入，以让网络能在同一个 batch 间发现共同的规律，并且一个 batch 只做一次疾病分类（即用一个 batch 下所有图片一起作为分类模块的一个输入）；

（2） 分割、分类都以一张图片为单位，不考虑同一个患者数张图片间的相关性。

两种模式分别标记为“single”、“group”。因此，数据集加载需要就这两种不同方案进行适配。在两种模式下，Locate_All 函数流程为：

（1） 用 root, dirs, files 依次 os.walk 文件夹 Image_DCM、Image_HCM、Image_NOR 下的\png\Image 文件夹：

（2） 如果模式为“single”，遍历 files 项：

(1) 那么将图片绝对路径添加到列表 images 中；

(2) 将图片绝对路径中 Image 替换为 Label、image 替换为 label 即是该图片对应的标签的绝对路径，添加到列表 labels 中；

(3) 并且将该图片的分类标签添加到列表 classes 中；

（3） 如果模式为“group”：

(1) 将 files 中每一项在前追加 root，然后将该列表添加到 images；

(2) 将 files 中每一项中 Image 替换为 Label、image 替换为 label，然后将该列表添加到 files；

(3) 每添加包含一个患者所有图片路径的列表到 image 后，将该患者的分类标签添加到列表 classes 中；

3.3.2 构建数据集

数据集构建使用的是自定义 Pytorch 中的 Dataset 类。

训练集加载模块为 Train_Dataset(Dataset)，模式为“single”时，以单张图片为单位，每次返回的是(图片，分割目标，分类)；模式为“group”时，以患者为单位，返回的是([1, n 张图片], [1, n 张分割目标], 分类)。

在其__init__函数中，只许调用 Locate_All (image_dir, mode)函数，即可完成对 self.images, self.labels, self.classes 的初始化（Locate_All 中两参数分别为数据集路径、模式）。

在“single”模式下，__getitem__函数大致流程为：

（1） 根据传入的位置参数 idx，使用 Image.open 打开相应路径的图片 image、标签 label；

（2） 进行图片增强、图片预处理（分为 image 和 label 共同的和各自的）；

（3） 因为 Label 中图片不同的类使用的是 8 位颜色 0、85、170、255 表示，所以将 0、85、170、255 分别转化为类标签 0、1、2、3，这通过 label[label == 85] = 1 类似实现即可；

而当模式为“group”时，idx 对应的是因为患者的一组图片，与“single”模式下操作的不同为以下两点：

（1） 遍历 self.images[idx]，对其下所有图片和标签都进行与“single”模式下的操作流程，得到_image、_label；

（2） 然后使用 unsqueeze(0)，将_image、_label 在第一个维度前增加一维，然后再将_image、_label 添加到 images、labels 中，最后进行 torch.vstack(images 或 labels)，这样就实现了以患者为单位进行图片加载。

另外在“group”模式下，训练时的 batch size 将被固定为 1，即每次只获取一个患者图片，共 n 张（该实验中 n 为大于等于 7 小于等于 13 的整数），在训练时将数据输入网络之前，因为单个数据维度为[1,n,1,256,256]，还需使用 squeeze(0)将第一个维度去除，这样就实现了训练时一个 batch 为一个患者的数据。

3.4 联合学习网络设计

上文中提到，联合学习模型总体结构有“single”、“group”两种构想。因此，分别提出了以下三种网络连接方案：

（1） 方案 1：“group”模式，同一患者图片相关联，共同决定分类；

（2） 方案 2：“single”模式，使用最高层特征进行分类；

（3） 方案 3：“single”模式，使用 Attention 层构造分类输入。

3.4.1 方案 1：以患者为单位进行分类

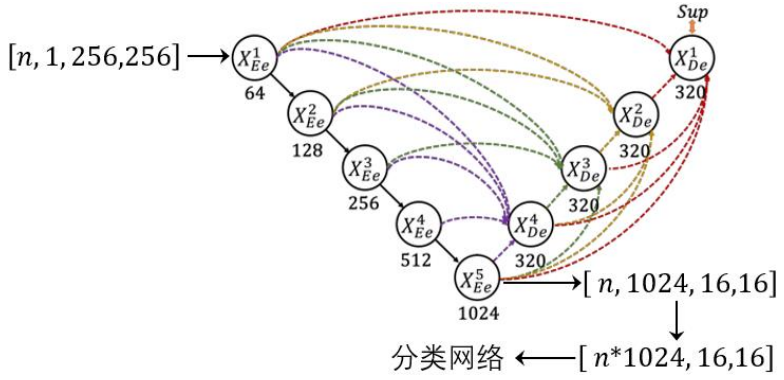


图 3.2 方案 1 示意图

以患者为单位进行分类的核心在于：网络连接处，将患者所有 n 张图片合并成一个整体，作为分类网络的一个输入，实现利用某患者的所有图片共同预测该患者的疾病类型。

但这也带来一个问题，那就是各个患者的图片数 n 不一定相同。显然这使得无法直接使用全连接、1*1 卷积等常规手段进行通道合并与降维。

为解决这一问题，采取的方案为使用 interpolate 将 n*1024 这一不确定的通道数全都下采样到一个事先选定的通道数 clsfct_input_size 上（例如取定 clsfct_input_size = 1024 * 5），这一数值便是分类网络输入的通道数。

具体代码实现如下：

```
logits, bottom = self.UNet(x)
# 使用双线性插值下（或上）采样
inputs = torch.vstack([i.squeeze() for i in bottom])  # [n*1024, 16, 16]
inputs = torch.permute(inputs, (1, 2, 0))  # [16, 16, n*1024]
# [16, 16, clsfct_input_size]
inputs = nn.functional.interpolate(inputs, size=self.clsfct_input_size, mode='linear', align_corners=True)
inputs = torch.permute(inputs, (2, 0, 1)).unsqueeze(0)  # [1, clsfct_input_size, 16, 16]
# 分类网络
outputs = self.net2(inputs).squeeze(2, 3)  # [1, 1024]
outputs = self.net3(outputs)  # [1,3]
return logits, outputs
```

即：

（1） 首先使用 vstack 将所有图片在 channel 维度上拼接；

（2） 然后使用 permute 将 channel 转置到最后；

（3） 便可开始使用 interpolate 在 channel 维度进行线性插值下采样；

（4） 最后使用 permute 和 unsqueeze 将 channel 恢复到正常位置。

这一方案顺利通过了正向传递，证明方案可行。

但由于实验平台内存不足（CG 总内存为 23GB），在 batch size 大于 5 时将

会出现内存不足报错导致无法正常训练。因此在实验条件限制下只能寻找其他替代方案。

3.4.2 方案 2：使用最高层特征分类

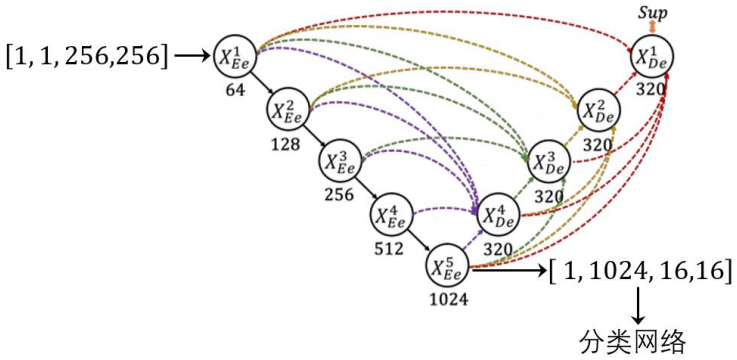


图 3.3 方案 2 示意图

将方案 1 的模式从“group”改为“single”即变成了方案 2。

使用以下简单网络（在分割网络冻结情况下）测试了一下方案 2 的预计效果：

```
self.net2 = nn.Sequential(Down(1024, 1024),    # [1024, 8, 8]
                          YoloBlock(1024, 512),    # [512, 8, 8]
                          Down(512, 1024),    # [1024, 4, 4]
                          nn.AdaptiveAvgPool2d((1, 1)),    # [1024, 1, 1]
                          )
self.net3 = nn.Sequential(nn.Linear(self.clsfct_input_size, 1000), nn.ReLU(),
                          nn.Linear(1000, 3), )
```

首先使用 net2 对特征进行进一步提取，然后使用 net3 的线性层进行分类。其中 YoloBlock 为 YOLO 作者在 DarckNet 中提出的基本单元，操作流程为：

$$\left(1x1conv(out_c) \Rightarrow [BN] \Rightarrow ReLU \Rightarrow 3x3conv(in_c)\right) * 2$$

$$\Rightarrow 1x1conv(out_c)$$

但发现效果并不理想，网络无法进行有效分类，会很快将输出全都变为 0。推测是由于分割网络编码器最底层特征是对图像的经可能详细的表征，无法集中有效地体现不同图片间的区别点（即不同图片的最底层特征大部分为相同内容），因此并不适合用于分类网络的输入。

3.4.3 方案 3：Attention 层构造分类输入

针对方案 2 中发现的问题，进一步提出了使用 Attention 层构造分类输入的方案 3：

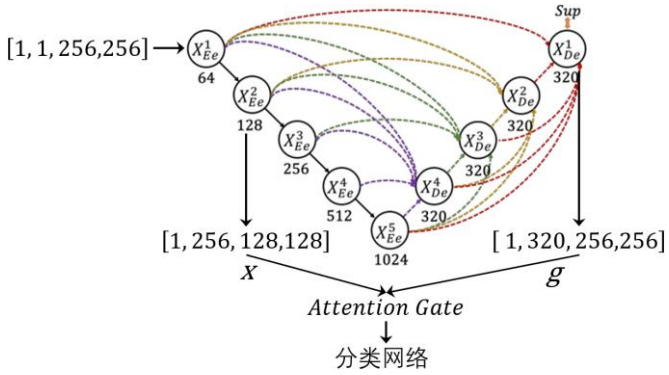


图 3.4 方案 3 示意图

其中注意力层使用的是 Attention UNet 作者提出的 Attention Gate：

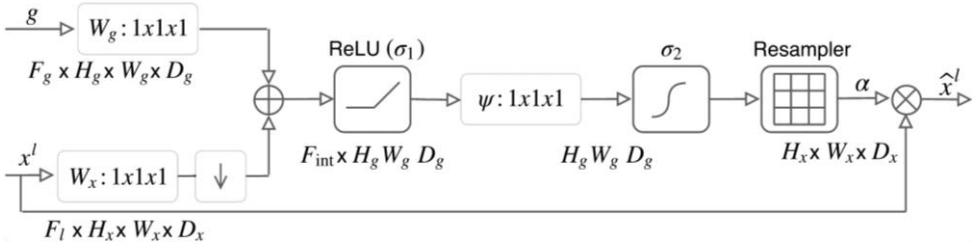


图 3.5 Attention Gate 流程图

分别从X_{Ee}^2, X_{De}^1层取出特征向量作为 Attention Gate 的输入 x 和 g，那么 x 和 g 便分别表示较底层的特征（蕴含分辨率较高的原始图片信息）和较高层的特征（蕴含较清晰的分割结果信息）。定性地理解，其含义就是重点关注原始图片信息中为前景信息的部分。

具体步骤为：

（1） 先将 g 进行一次卷积将 channel 数减为 256，并 average pool 下采样减小尺寸到与 x 相同；x 进行一次卷积，channel 数保持 256；

（2） 然后使用 Attention 层得到融合后的特征；

（3） 再次卷积和下采样以减小计算量，得到维度为[1,128,64,64]的特征作为分类网络（选用 DenseNet121）的输入；

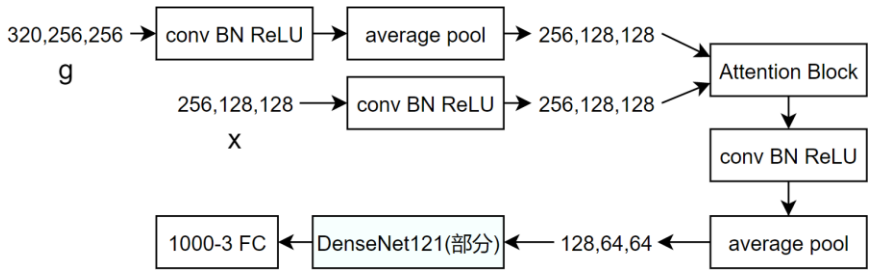


图 3.6 连接模块的完整网络结构

3.4.4 分类网络

分类网络选取的是 DenseNet121 的第二个 Dense Block 开始及其之后的全部

网络，也就是图 3.7 中红圈圈出的 6 个网络层，并在其后接一个 1000 至 3 的全连接层。

Layers	Output Size	DenseNet-121
Convolution	112 × 112	
Pooling	56 × 56	
Dense Block (1)	56 × 56	[1 × 1 conv, 3 × 3 conv] × 6
Transition Layer (1)	56 × 56	
	28 × 28	
Dense Block (2)	28 × 28	[1 × 1 conv, 3 × 3 conv] × 12
Transition Layer (2)	28 × 28	
	14 × 14	
Dense Block (3)	14 × 14	[1 × 1 conv, 3 × 3 conv] × 24
Transition Layer (3)	14 × 14	
	7 × 7	
Dense Block (4)	7 × 7	[1 × 1 conv, 3 × 3 conv] × 16
Classification Layer	1 × 1	

图 3.7 选用的分类网络：DenseNet121 的部分

具体实现是由 torchvision.models. densenet121 创建一个 DenseNet121，然后从该 DenseNet121. features 中取出 denseblock2、transition2、denseblock3、transition3、denseblock4、norm5 层，全都放入新建的一个 self.net_classify = nn.Sequential 中即可。

注意，按照 DenseNet 的源码，在 norm5 和后面的 classifier 间，还有一个将最终特征图全局平均池化、再展平的操作，然后再输入分类层。即在 forward 中，需添加以下代码：

```
x = self.net_classify(inputs)    # [1, 1024, 16, 16]
x = functional.relu(x, inplace=True)
x = functional.adaptive_avg_pool2d(x, (1, 1))    # [1, 1024, 1, 1]
x = torch.flatten(x, 1)    # [1, 1024]
outputs_classify = self.classifier(x)    # [1, 4]
```

3.5 模型训练

1. 迁移学习训练方案及适用性探究

又一次，联合模型也出现了实验平台内存不足的现象，导致无法有效地直接训练分割和分类的联合模型，于是只好采用的类似迁移学习的训练方案作为替代品。

迁移学习的训练方法就是先训练好一个完整的分割网络，然后将该分割网络的参数载入联合模型并将其完全冻结，第二轮训练中单独对连接模块及分类网络

进行训练。主要对在这一过程中遇到的以下两点做出说明：

（1） 若需要对已保存好的网络的 forward 函数进行修改（例如，此实验中，分割网络单独训练时、和在放入联合模型中后，需要返回的是不同的层的特征向量），则可使用 partial 函数进行修改：

要修改的函数 = partial(新的函数, 要修改的对象)

这样达到仅修改这一个对象中该函数的目的，而类定义、其他类对象均不受影响；

（2） 若网络中有部分冻结的参数，那么在训练时，可使用 filter 配合 lambda 函数将仅未被冻结的参数传入优化器中：

params=filter(lambda p: p.requires_grad, net.parameters())

虽然迁移学习达到了训练模型的目的，但在实验中发现这种方式还是无法达到联合学习对分割和分类准确率同时提高的促进作用。下表中，分割模型的训练集是各疾病中的 1-13 号患者数据；为更好的考察迁移学习的有效性，分割模型的训练集是各疾病中的 1-10 号患者数据；记使用 11-13 号患者数据作为验证集为 val-1、使用 14-15 号患者数据作为验证集为 val-2。实验结果如表 3.2 所示：

表 3.2 加载不同分割模型在不同 epoch 下分类正确率对比结果

分割模型 F1 得分	epoch	val-1 F1 得分	val-2 F1 得分
测试集 89.94%	10	91.86%	78.17%
验证集 84.40%	**40**	**93.74%**	**82.84%**
	20	88.40%	68.30%
测试集 90.35%	30	89.59%	71.70%
验证集 85.18%	40	81.84%	78.25%
	20-2nd	**92.30%**	**79.10%**

可见，在其他情况相同条件下，加载两个不同的分割模型时，分类所能达到最高正确率是不同的。

如表中的第二个虽然比第一个在分割上准确率更高，但在分类上却更低；并且 20-2nd 一项是第二次进行一个总 epoch=50 的训练，其仅在 10~20epoch 内达到分类准确率最高之后都不再出现更高的正确率；更甚者，加载准确率最高的分割模型（在验证集 F1 得分为 85.90%）时，分类所能达到的最高 F1 得分仅有 77.66%。

所以得到结论：

（1） 本数据集上，分割和分类准确率并不是成正比关系；

（2） 这也意味着迁移学习在本数据集上是不适用的。

2. 损失函数改进

分割任务的损失初步采用 CrossEntropyLoss、dice_loss、ms_ssim_loss 进行联合评价。通过调参实验，发现以下参数能获得更好的准确率、及更快的收敛速度：

（1） CrossEntropyLoss 中，考虑到背景的像素数量远远多于前景，因此将 0（背景）、1、2、3 中，类 0 背景的权重下调为 0.6；

（2） ms_ssim_loss 更多的是衡量图像级别的相似度，且损失值通常较高，权重下调到 0.2。

（3） dice_loss：（A、B 中前景交集面积/前景面积之和）*2

表 3.3 损失函数调参前后模型最优性能对比结果

类别	到最优所需 epoch	训练集 F1 得分	验证集 F1 得分
默认参数	70	89.94%	84.40%
调整后	60	90.44%	85.90%

3.6 交互式界面

1. 界面实现

使用 PyQt5 实现了一个简单的交互式界面（仅用于模型使用即 detect，不包括训练 train 过程）。大致步骤如下：

（1） 使用 Qt Designer 设计好页面（MainWindow.ui 文件）；

（2） 使用 PyUIC5 工具将.ui 文件转为.py 文件（MainWindow.py），即得到一个可以进行页面初始化的类 Ui_MainWindow；

（3） 创建 My_MainWindow 类，其同时继承于 QtWidgets.QWidget 和 Ui_MainWindow，调用父类 Ui_MainWindow 中的 setupUi 函数，即完成了页面创建；

（4） 在 My_MainWindow 类添加槽函数和信号槽连接即可。

运行效果如图 3.8 所示，路径选择既可采用直接在输入框输入、也可通过点击左侧的按钮以进行浏览：

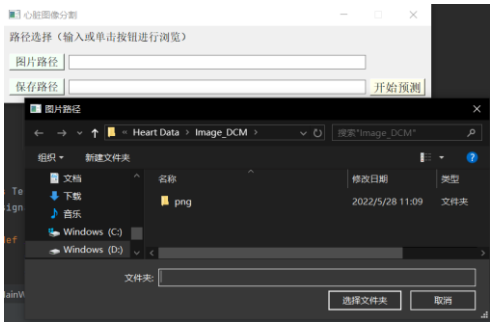


图 3.8 交互界面及路径选择

2. QThread 的使用

因为使用模型进行检测的过程耗时较长，在此期间让界面做出相应提示、并不再接收新的指令显然是很重要的，否则多次重复点击会导致检测的重复执行。因此使用 QThread 类，通过新建子进程运行检测，并在子进程中使用自定义信号 pyqtSignal 以实现与主进程（交互界面进程）间的同步。

具体实现方法为：

（1） 构建类 Test(QThread)，在 My_MainWindow 中实例化对象 self.Test = Test()；将模型运行所需的 image_dir、save_dir、batch_size 等都定义于类 Test 中，而 My_MainWindow 中可直接访问对象 self.Test 中所有变量，因此就实现了交互式界面进程向检测进程传输参数的功能；

（2） 然后将 self.Test.start()、self.setEnabled(False)等绑定到“开始预测”按钮的 clicked 信号，从而实现了点击“开始预测”按钮后开始模型预测、并关闭交互界面中所有按钮的功能；

（3） Test 进程中的 pyqtSignal(int)型数据 signal_buttonEnable 在预测结束时发送信号 0，将 setEnabled(True)绑定到该信号的 0 上，就实现了 Test 进程结束将交互式界面所有按钮重新启用；另外将模型检测全过程在 try 下进行，若捕获到异常（如路径无效、文件缺失）可让 signal_buttonEnable 发送不同值信号，这样实现了在有运行错误情况下让交互式界面显示错误原因的功能，并防止了程序错误导致崩溃。

运行过程、运行完成情况截图如图 3.9 所示：

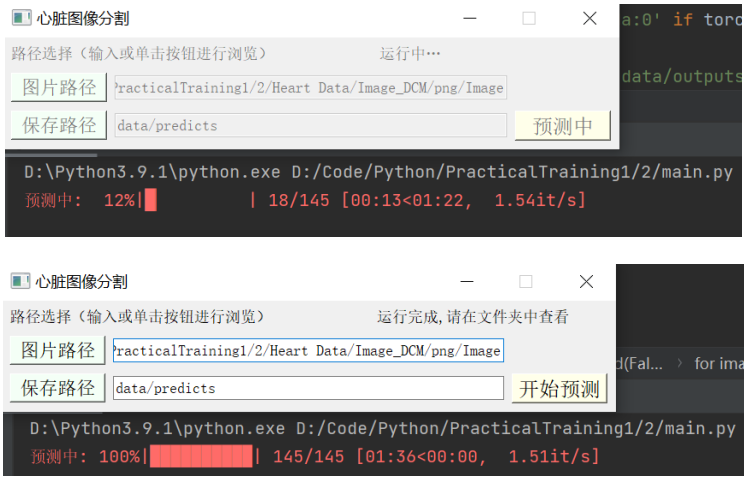


图 3.9 运行中、运行完成后界面

3.7 图像增强与预处理

通过观察数据集，可发现有些较异常的数据表现在亮度低、对比度低、较模糊等方面，因此设计了以下图像增强和预处理方案，二者均是添加到数据集加载类中即可。

1. 图像增强：

通过实验，发现采用“先将平均亮度统一为 10，然后直方图均衡化”的方案，从主观上认为可消除亮度低、对比度低的影响，并且减小了分割难度。该图像增强对比图如图 3.8 所示。左边上下分别为正常亮度的原图、亮度过低的原图；右边上下分别为两原图经图像增强后的结果：

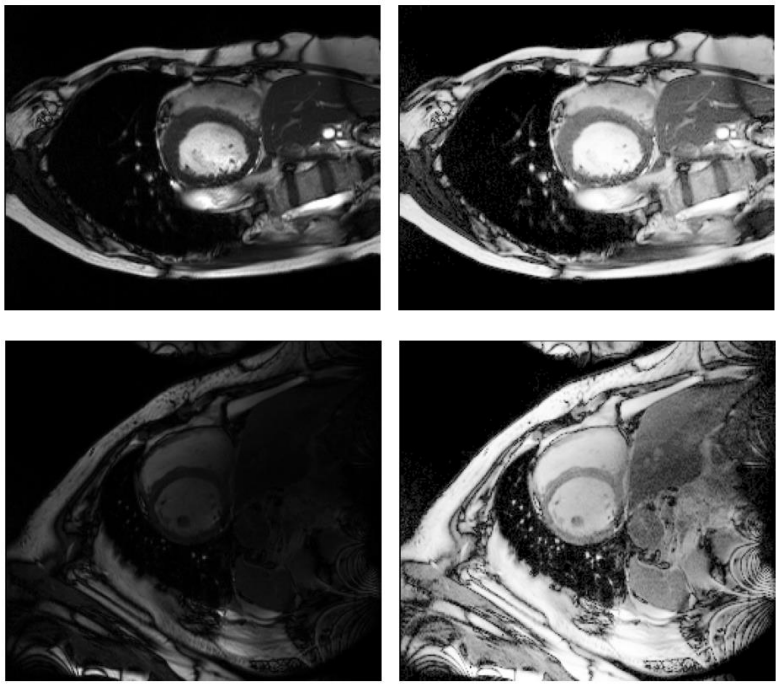


图 3.8 正常亮度、亮度过低图片的图像增强效果

核心代码如下：

```
temp = numpy.array(image)
temp = ((10 / temp.mean()) * temp).astype(numpy.uint8)
temp = cv2.equalizeHist(temp)
image = Image.fromarray(temp)
```

另外还尝试了使用 Image 库添加边缘增强、锐化增强、细节增强等 filter，但发现均会导致分割准确率出现较明显的下降。

2. 预处理：

通过对数据集的观察，发现不同患者的图像中躯干截面并不都是水平的，有会出现竖、倾斜等情况，所以添加了随机水平翻转 RandomHorizontalFlip、随机

竖直反转 RandomVerticalFlip、随机仿射变换 RandomAffine(degrees=(0, 15), scale=(0.5, 1.5),)等形变操作。

值得注意的是：

（1） 分割任务中任何涉及到形变的预处理操作都需要同时作用到图像和标签（分割目标）上。实验中通过将所有形变操作放置于图像和标签共同的 transform：transform_share 中实现；

（2） 必须保证涉及随机性的预处理，在二者上必须相同。这一点需要通过在执行 transform_share(x)、(y)前将随机数种子置为相同实现；

（3） 且为保证不同图像间的预处理仍具有随机性，每张图片都需要通过随机数生成新的种子（如使用 randint）。

核心代码如下：

```
def __getitem__(self, idx):
    ……
    # X、Y 共同
    if self.randomCrop:
        #  在执行 transform_share(x)、(y)前将 seed 置为相同
        seed = randint(0, 2147483647)
        torch.manual_seed(seed)
        image = self.transform_share(image)
        torch.manual_seed(seed)
        annotation = self.transform_share(annotation)
    # X/Y 单独
    image = self.transform(image)
    annotation = self.target_transform(annotation)
```

将 torch.manual_seed 设为相同值后，进行随机操作时就会产生相同的结果。而同时为保障随机裁剪具有随机性，每次都将 seed 设为不同值（randint 中的 2147483647 为随机输入的一个很大的值，以确保 seed 值的随机性）。

而 image 单独的预处理采用了常规的高斯模糊 GaussianBlur(kernel_size=3)、随机降低锐度 RandomAdjustSharpness(0, p=0.2)操作。