机器学习大作业——医学 3D 图像检测分割

郑文卓 *519030910360* 王勤玮 *519030910378*

1. 问题描述和分析

1.1. 问题描述

在大作业的任务要求中，我们需要对医学 3D 图像进行检测与分割。具体而言，我们需要对数据集中的肋骨骨折部分进行检测与分割。通过使用神经网络的方法，返回出一个三维的 3D 图像的分割结果。并且最终使用 FROC 作为我们的监测评价指标。在这个问题中，我们将主要使用到 3D U-Net 神经网络。

链　接: `https://github.com/darkcorvushhh/res3dunet-RibFrac`

1.2. 问题分析

这个问题的要求以及难度相较于小作业 MedMNIST 而言，可以认为是提高了许多。

首先，从数据集本身来看，三维医学影像由于其三维的特性，最终其图像本身大小很大。但是图像中作为骨折的区域本身很小，并且我们也很难拥有较大规模的医学影像作为数据集。这也就意味着在实际训练的过程中，我们会遇到许多非骨折的部分，从数据量上来看是增大了不止一个数量级。

由以上的认识，我们本次实验中所需要的训练时间与对服务器设备的要求也上升了许多。这也就意味着我们在实际操作中所能拥有的测试机会相较于小作业来说，会减少很多。我们需要利用好宝贵的实验机会，进行我们的测试与调试。

当然，3D 医学图像本身就具有着他自己的特点。首先医疗影像语义较为简单、结构固定。因此语义信息相比自动驾驶等较为单一，因此并不需要去筛选过滤无用的信息。医疗影像的所有特征都很重要，因此低级特征（提供物体类别识别依据）和高级语义特征都很重要（提供精准分割定位依据），所以 U 型结构的 skip connection 结构（特征拼接）更好派上用场。

另外，医学影像的数据较少，获取难度大，数据量可能只有几百甚至不到 100，因此如果使用大型的网络例如 DeepLabv3+ 等模型，很容易过拟合。大型网络的优点是更强的图像表述能力，而较为简单、数量少的医学影像并没有那么多的内容需要表述。

医学影像往往是多模态的。比方说 ISLES 脑梗竞赛中，官方提供了 CBF，MTT，CBV 等多中模态的数据。因此医学影像任务中，往往需要自己设计网络去提取不同的模态特征，因此轻量结构简单的 Unet 可以有更大的操作空间。

这些 3D 医学图像的特性，都是我们可以使用 3D-UNet 作为本实验训练网络的原因。在实际操作这个问题时，我们同时也有很多细节需要确定下来。首先是关于数据集的处理，网络的设计以及最终需要输出的特征，使用怎样的训练方式以及测试方式，如何评估我们的训练模型等等，都是眼前的比较棘手的问题。

在代码的具体设计上，我们也需要根据给出的示例代码，想办法调节我们的数据输入。并且要充分理解示例代码中相应函数的功能以及输出，最终实现一个完整的工程过程。同时对于一些细节方面的设计，比如数据处理等方面，我们也不能落下，也需要进行详细的考虑。

综上所述，在这个任务中，我们既需要设计好一个神经网络，同时也要非常细致的实现这个网络从搭建到训练到测试全流程的各种细节，相较于笔者的水平而言，这可以称得上是一个不小的挑战。

2. 模型设计

对于模型的设计，在仔细研究了 ppt 上的网络 3D-Unet 后，一个灵感蹦入了我们的脑海。如果把我们小作业的思路利用到大作业里会怎么样？也就是说，如果把残差块加入到 3D-Unet 的训练

中会怎么样?

这个思路立刻启发了我们构思出 res3D-Unet，但光有灵感还不够，我们上网找了许多资料。功夫不负有心人，我们在 github 中找到了多篇关于 res3D-Unet 的代码，在参考了他们的结构后，结合 3D-Unet 的模型结构，我们设计出了如下的模型：

```
def forward(self,x) :
        l1=self.en1(x)+x
        s1=self.down1(l1)
        l2=self.en2(s1)+s1
        l2=F.dropout(l2,self.drop_rate,self.
    training)
        s2=self.down2(l2)
        l3=self.en3(s2)+s2
        l3=F.dropout(l3,self.drop_rate,self.
    training)
        s3=self.down3(l3)
        l4=self.en4(s3)+s3
        l4=F.dropout(l4,self.drop_rate,self.
    training)
        s4=self.down4(l4)
        out=self.de1(l4)+s4
        out=F.dropout(out,self.drop_rate,self.
    training)
        output1=self.map1(out)
        s6=self.up2(out)
        out=self.de2(torch.cat([s6, l3], dim=1))+
    s6
        out=F.dropout(out,self.drop_rate,self.
    training)
        output2=self.map2(out)
        s7=self.up3(out)
        out=self.de3(torch.cat([s7, l2], dim=1))+
    s7
        out=F.dropout(out,self.drop_rate,self.
    training)
        output3=self.map3(out)
        s8=self.up4(out)
        out=self.de4(torch.cat([s8, l1], dim=1))+
    s8
        output4=self.map4(out)
        # print(output1.shape)
        # print(output2.shape)
        # print(output3.shape)
        # print(output4.shape)
        if self.training is True:
            # return output1, output2, output3,
    output4
            return output4
        else:
            return output4
```

与 3D-Unet 相比，其核心就是在每个 encoder 阶段构造一个残差块，在 decoder 阶段加上 dropout，同时把 conv+bn+relu 作为一个卷积块，提高收敛速度。

但是在实际训练时我们发现如果用上了前面三个 output 计算 loss 并反传，训练效果反而不太理想，可能对于此次项目来说 res3D-Unet 的前三个 output 并不能很好的表征骨折的特征，故此注释掉。

参数量大概为 950 万。

3. 数据预处理与模型训练

3.1. 数据集读取

在数据集的读取上，我们需要对数据集的具体内容有一定的认识。首先，训练集，验证集以及测试集中，所有的数据均是以.nii 形式存储的，我们需要使用到 python 中的 SimpleITK 或者 Nibabel 库进行读取。在这里，我们使用的是 SimpleITK 库进行的数据读取。

对读取得到的数据，我们需要做一些基本处理。由于 SimpleITK 库中得到的数据均是每一个点的像素值，我们在这里需要做一些简单的变换，使得每一个点的数据尽可能的小，这样当数据输入到网络中之后，不会出现数值爆炸的情况。所以在这里我们选择了让每一个点的值除以一个固定的归一化系数，这在我们的代码中有体现。

在此之后，我们需要对标签文件做一些了解。对测试集和验证集，其标签输出为 array 之后的结果是如原网站中所说的 label-code。即-1 到 4 的表示是否骨折，以及是什么类型的骨折的标签。0 代表原图像中此处没有骨折，其他数字分别代表着不同类型的骨折。在这里，为了前期处理的方便，以及测试网络模型的，我们将所有非零的标签置为了 1，即我们先测试我们的网络能否成功实现将骨折与非骨折部位做区分的一个二分类问题。在二分类问题调试完成之后，我们希望能在原来的基础之上实现可以达成目标的多分类问题。

以下的内容实现，都是基于我们简化问题而完成的步骤。即**先将问题转化为二分类问题**。具体实现简化如下。

```
ct_array = ct_array / self.args.norm_factor
ct_array = ct_array.astype(np.float32)
label[label[:, :, :] != 0] = 1
```

考虑到我们需要使用 Dataloader 函数进行分 batch 的处理，但是分 batch 之后，我们不太方便将一张图的许多信息加入到网络中进行训练。所以我们在处理的时候，选择了将同一张图的多个骨折部分重新拼接到一个维度之上，相当于是手动的实现了分 batch 的操作。具体实现如下。在这里，我们考虑到训练的速度，只考虑将骨折处丢入至网络中进行训练。对占比很大的背景部分，我们暂时不做考虑。

```
    for i in range(int(input.shape[2] // 64)):
            for j in range(int(input.shape[3] // 64)):
                for k in range(int(input.shape[4] // 64)):
                    input_new = input[:, :, 64 * i:64 * (i + 1), 64 * j:64 * (j + 1), 64 * k:64 * (k + 1)].float()
                    label_new = label[:, :, 64 * i:64 * (i + 1), 64 * j:64 * (j + 1), 64 * k:64 * (k + 1)].float()
                    if torch.sum(label_new).item() == 0:
                        continue
                    else:
                        input_final = torch.cat((input_final, input_new), 0)
                        label_final = torch.cat((label_final, label_new), 0)
```

最后我们是实现了一个可以返回出图像数据的 Tensor 以及标签信息的 Tensor 的数据集。返回出的数据的大小与原图相同，我们将在之后的过程中对数据进行一些手动 Reshape 操作，即如上代码中的操作，这相当于是舍弃掉了一部分数据，然后将剩余的骨折部分绑在一起。

3.2. 预处理选择

在将数据丢入网络之前，我们有考虑过采用一些更加具体的归一化操作对原始图像数据进行处理，比如将图像数据限制在 [-1,1] 范围之内。同时也考虑采用一些数据增强的手段，比如随机旋转，随机翻转等等。在之后的实验中，我们会根据所得到的结果对比数据增强之后的实验结果的差异。

3.3. 模型训练

在本部分中，我们使用的显卡是 NVIDIA GeForce RTX 2080Ti，11G 显存。根据实验中的测试结果，我们的 batchsize 最多可以设置为 8，所以在之后的实验过程中，我们都是使用了 8 作为 batchsize 进行试验。

我们的模型使用的优化器是 pytorch 中自带的 Adam 函数，设置了初始学习率为 0.001，同时每训练 10 个 epoch，学习率会下降到原来的 0.1，同时我们使用的 loss 函数是示例代码中已经给出的 DiceLoss 函数，在这里，我们需要计算的是网络输出结果的 DiceLoss，也即是 batchsize*64*64*64 的 Tensor 的 DiceLoss 的均值。DiceLoss 的实现如下；

```
class DiceLoss(nn.Module):
    def __init__(self):
        super().__init__()

    def forward(self, pred, target):
        num = target.size(0)

        smooth = 1
        pred = pred.view(num, -1)
        target = target.view(num, -1)
        inter = pred * target

        score = 2. * (inter.sum(1) + smooth) / (pred.sum(1) + target.sum(1) + smooth)
        score = 1 - score.sum() / num

        return score
```

在实验的过程中，resize 的操作已经在之前的模块中给出。同时我们还考虑加入了无监督学习的更新参数 α。我们的训练函数最终实现如下：

```
def train(model, train_loader, optimizer, loss_f, args):
    model.train()
    print('It is the training CT process')

    for idx, (input, label) in tqdm(enumerate(train_loader), total=len(train_loader)):
        input, label = input.to(device), label.unsqueeze(0).to(device)

        input_final = torch.tensor([]).to(device)
        label_final = torch.tensor([]).to(device)
```

```
        optimizer.zero_grad()

        for i in range(int(input.shape[2] // 64)):
        for j in range(int(input.shape[3] // 64)):
        for k in range(int(input.shape[4] // 64)):
            input_new = input[:, :, 64 * i:64 * (i + 1), 64 * j:64 * (j + 1), 64 * k:64 * (k + 1)].float()
            label_new = label[:, :, 64 * i:64 * (i + 1), 64 * j:64 * (j + 1), 64 * k:64 * (k + 1)].float()
            if torch.sum(label_new).item() == 0:
                continue
            else:
                input_final = torch.cat((input_final, input_new), 0)
                label_final = torch.cat((label_final, label_new), 0)

            if input_final.shape[0] >= 8:
                F = nn.Sigmoid()
                output0 = F(output0).to(device)
                F = nn.Sigmoid()
                output1= F(output1).to(device)
                F = nn.Sigmoid()
                output2 = F(output2).to(device)
                F = nn.Sigmoid()
                output3 = F(output3).to(device)

                loss0 = loss_f(output0, label_final)
                loss1 = loss_f(output1, label_final)
                loss2 = loss_f(output2, label_final)
                loss3 = loss_f(output3, label_final)
                loss = loss3 + args.alpha * (loss0 + loss1 + loss2)


                loss.backward()
                optimizer.step()

                input_final = torch.tensor([]).to(device)
                label_final = torch.tensor([]).to(device)

        if input_final.shape[0] != 0:
            output0, output1, output2, output3 = model(input_final)
            F = nn.Sigmoid()
            output0 = F(output0).to(device)
            F = nn.Sigmoid()
            output1 = F(output1).to(device)
            F = nn.Sigmoid()
            output2 = F(output2).to(device)
            F = nn.Sigmoid()
            output3 = F(output3).to(device)
            loss0 = loss_f(output0, label_final)
            loss1 = loss_f(output1, label_final)
            loss2 = loss_f(output2, label_final)
            loss3 = loss_f(output3, label_final)
            loss = loss3 + args.alpha*(loss0 + loss1+ loss2)
            loss.backward()
            optimizer.step()
```

3.4. 模型测试模块

同时对于我们的训练模型，在实际操作的过程中，我们需要对已经训练的模型进行测试。在这里，我们为了保证训练的模型能够顺利的保存，选择了在每一个 Epoch 运行结束之后，保存已经训练完毕的模型参数。这样，在我们最后进行测试时，我们可以有目的性的选择我们需要进行测试的模型。具体模型保存的实现如下：

```
class DiceLoss(nn.Module):
state = {'net': model.state_dict(), 'optimizer': optimizer.state_dict(), 'epoch': epoch, 'loss': loss}
print('%d training epochs have finished'%(epoch))
torch.save(state, os.path.join(save_path, 'trainmodel_info.pth'))
torch.save(state, os.path.join(save_path, 'tempmodel_%d_info.pth'%epoch))
print('on the go of saving %d epoch model\n'%epoch)
```

在具体实现模型的测试的时候，我们目前所采用的模型测试方式简述为：将输入的完整图像按滑窗分割为若干个大小为 64*64*64 的图像块，将图像块输入至网络中，然后对网络输出的结果，使用 Sigmoid 函数转化为预测概率之后。将最终输出的预测概率与设定的判断阈值进行比较。

然后，我们需要将输出得到的预测概率转化为能够进行评测的.nii 文件以及.csv 文件。这一步骤，我们是参考了 Github 中 FracNet 代码中的相关函数，他们使用了 scikit-learn 中的 regionprops 类进行了分块的处理以及输出，并且最终转化为可以最后输出的 pandas.DataFrame 形式。函数代码实现如下：

```
def _make_submission_files(pred, image_id, affine):
```

```
    pred_label = label(pred > 0).astype(np.int16)
    pred_regions = regionprops(pred_label, pred)
    pred_index = [0] + [region.label for region in
     pred_regions]
    pred_proba = [0.0] + [region.mean_intensity
    for region in pred_regions]
    # placeholder for label class since
    classifaction isn't included
    pred_label_code = [0] + [1] * int(pred_label.
    max())
    pred_image = nib.Nifti1Image(pred_label,
    affine)
    pred_info = pd.DataFrame({
        "public_id": [image_id] * len(pred_index),
        "label_id": pred_index,
        "confidence": pred_proba,
        "label_code": pred_label_code
    })

    return pred_image, pred_info
```

目前我们暂时采用的滑窗的 stride 大小为 32，判断阈值为 0.9，test 的主体代码如下:

```
model.eval()
print('It is the Testing CT')
with torch.no_grad():
    for idx, (input, filename, channel) in tqdm(
    enumerate(test_loader), total=len(test_loader
    )):
        [B, X, Y, Z] = input.shape
        channel = channel.item()
        input = input.unsqueeze(1).to(device)
        pred_final = torch.empty(B, X, Y, Z).to(
    device)
        filename = filename[0]
        filename_last = filename.split('-')[0].
    split('/Data/val')[0] + '/labels/test' +
    filename.split('-')[0].split('/Data/val')[1]
    + '.nii.gz'
        image_id = filename_last.split('.')[0].
    split('/')[7]

        for i in range(int(input.shape[2] // 32 -
    1)):
        for j in range(int(input.shape[3] // 32 -
    1)):
        for k in range(int(input.shape[4] // 32 -
    1)):
            input_new = input[:, :, 32 * i:32 * (i
     + 2), 32 * j:32 * (j + 2), 32 * k:32 * (k +
    2)].float()
            output = model(input_new)

            # Using sigmoid to denote the
    probability
            F = nn.Sigmoid()
            output = F(output).to(device)
            pred_final[:, 32 * i:32 * (i + 2), 32
    * j:32 * (j + 2), 32 * k:32 * (k + 2)] =
    output

        pred_final[:, X:channel] = pred_final[:,
    2*X-channel:X]
        pred_final = pred_final.squeeze(0).cpu().
    numpy()
        pred_final[pred_final[:, :, :] < 0.9] = 0
        pred_final = pred_final.transpose((1, 2,
    0))
        pred_final_new = pred_final[:, :, :channel
    ]

        pred_img, pred_info =
    _make_submission_files(pred_final_new,
    image_id, np.eye(4))
        nib.save(pred_img, filename_last)
        pred_info_list.append(pred_info)

    pred_info = pd.concat(pred_info_list,
    ignore_index=True)
    pred_info.to_csv(os.path.join(args.root_path,
    'labels/test/ribfrac-val-pred.csv'), index=
    False)
```

在此之后，我们完成了训练的部分，并且将测试之后得到的标签图像以及输出的.csv 文件做了保存之后。可以进行最终的结果测试。

4. 性能分析

4.1. 初始实验结果

对我们已经进行过处理的简化的二分类问题，我们采用的训练集是原训练集中所有的 420 张图片，测试集是原验证集中的 80 张图像。我们所采用的训练 Epoch 数是 20 个，最终得到了如下的结果，如图 1 所示: 以上的实验结果说明我们的实现过程中存在着一些问题。我们相信我们的代码的主体部分还是十分正确的，在实现的过程中，有一些必要重要的思路上的问题仍然需要我们解决。

4.2. 代码的修改思路及结果

我们重新回顾了一下代码中遇到的各种问题，再经过各种检查与尝试之后。我们主要对两个地方进行了修改以及测试。

1. 我们对数据集读取的函数进行了更换，并

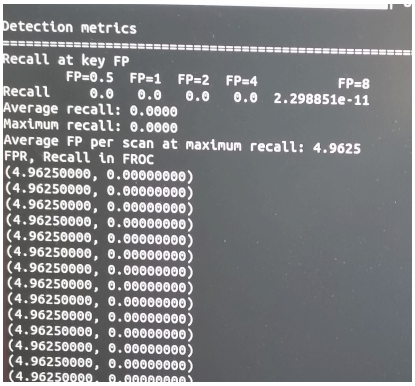

图 1: 初始实验结果

且在读取的时候考虑以图像的像素值作为我们所读取的内容。修改之后大致如下

```
ct = sitk.ReadImage(self.filename_list[index])
ct = sitk.Cast(sitk.Rescaleintensity(ct), sitk.
    sitkUInt8)
label_orient = sitk.ReadImage(self.label_list[
    index])
ct_array = sitk.GetArrayFromImage(ct)
label = sitk.GetArrayFromImage(label_orient)
```

2. 我们对丢入网络进行训练的图片块进行了修改。最开始我们的代码思路中是将标签值为均为 0 的背景块舍弃，但是实际上我们可能需要舍弃的是像素值全部为 0 的像素块，所以我们需要对训练集中的代码进行一定的修改。

3. 我们的均一化需要在对全部为 0 的像素块进行挑选之后进行。基于以上两点，我们可以将我们的训练函数的对应代码修改成以下的代码。

```
#为排版方便，我们将i，j，k没有缩进的放在一起，实际
    代码中是有相应缩进的三层循环
input_final = torch.tensor([]).to(device)
label_final = torch.tensor([]).to(device)
zero_tensor = torch.zeros([1,1,64,64,64], dtype=
    torch.int)
optimizer.zero_grad()
for i in range(int(input.shape[2] // 64)):
for j in range(int(input.shape[3] // 64)):
for k in range(int(input.shape[4] // 64)):
    input_new = input[:, :, 64 * i:64 * (i + 1),
    64 * j:64 * (j + 1), 64 * k:64 * (k + 1)].
    float()
    label_new = label[:, :, 64 * i:64 * (i + 1),
    64 * j:64 * (j + 1), 64 * k:64 * (k + 1)].
    float()
    if (input_new == zero_tensor).all() == True:
        continue

    else:
        input = normalize(input)
        input = input_new = input[:, :, 64 * i:64
    * (i + 1), 64 * j:64 * (j + 1), 64 * k:64 * (
    k + 1)]
        input_final = torch.cat((input_final,
    input_new), 0)
        label_final = torch.cat((label_final,
    label_new), 0)
```

但是在对以上的修改进行了测试之后，我们发现我们的测试结果依旧没有明显的改观。所以我们认为是之前的代码中的基本结构和思路可能存在一些问题。我们打算重新整理我们的思路，从一个新的角度尝试一段代码，得到一个可以让自己相对满意的实验结果。

5. 对基本结果的改进思路与改进结果

5.1. 新的更改的选择

我们重新回顾了自己的代码之后，认为自己的代码在两个方面上存在着比较巨大的漏洞：首先，对于图片分块的选择上，我们选择了不再以每一块作为主要的对象，而是选择找到每个 64*64*64 块的中心点，并且将其从数据集中返回出来。同时在对训练集的处理中，我们不能简单的无选择性或是简单地将带标签的图片分块之后加入网络。在训练的过程中，我们需要找到那些我们真正需要的中心点（ROI Centroids），它包括了正样本以及负样本的中心点。

对训练与测试的过程上，我们认为还是要尽量简单，避免出现一些可能遇到的 bug，所以我们希望将主要的操作都留在数据集中。

与此同时，我们需要对标签图片的相同部位做同样的操作，于是我们可以得到我们的数据集，包括了图片数据与以中心点为中心的 64*64*64 的图片块。

我们在模型的最后加入了 Sigmoid/Softmax，输出的类别的概率。于是我们可以根据网络的结果计算指标。我们的 loss 函数依旧是 Dice+B CEloss 并加上了一定的权值，优化器依旧是 Adam 函数。同时在自行测试代码的时候，我们使用的评测指标包括了 accuracy，precision，recall 等。

对测试集，我们所需要的修改包括适应于训练集的取中心点的处理。其他步骤则与前文所述类似.

另外的，我们的数据增强操作可以更改到在数据集读取的时候进行，这也避免了一些不可知的错误的发生。

我们的代码更改的参考来自于 Github，主要参考自 FracNet，同时更改后，我们希望能够快速的得到一个比较好的结果，我们依旧是如之前一样，**做的是一个二分类的问题**。

数据集的处理如下，这里包括了训练以及测试两部分：

```python
class FracNetTrainDataset(Dataset):

    def __init__(self, image_dir, label_dir=None, crop_size=64,
                 transforms=None, num_samples=4, train=True):
        self.image_dir = image_dir
        self.label_dir = label_dir
        self.public_id_list = sorted([x.split("-")[0]
                                      for x in os.listdir(image_dir)])
        self.crop_size = crop_size
        self.transforms = transforms
        self.num_samples = num_samples
        self.train = train

    def __len__(self):
        return len(self.public_id_list)

    @staticmethod
    def _get_pos_centroids(label_arr):
        centroids = [tuple([round(x) for x in prop.centroid])
                     for prop in regionprops(label_arr)]

        return centroids

    @staticmethod
    def _get_symmetric_neg_centroids(pos_centroids, x_size):
        sym_neg_centroids = [(x_size - x, y, z)
                             for x, y, z in pos_centroids]

        return sym_neg_centroids

    @staticmethod
    def _get_spine_neg_centroids(shape, crop_size, num_samples):
        x_min, x_max = shape[0] // 2 - 40, shape[0] // 2 + 40
        y_min, y_max = 300, 400
        z_min, z_max = crop_size // 2, shape[2] - crop_size // 2
        spine_neg_centroids = [(
            np.random.randint(x_min, x_max),
            np.random.randint(y_min, y_max),
            np.random.randint(z_min, z_max)
        ) for _ in range(num_samples)]

        return spine_neg_centroids

    def _get_neg_centroids(self, pos_centroids, image_shape):
        num_pos = len(pos_centroids)
        sym_neg_centroids = self._get_symmetric_neg_centroids(
            pos_centroids, image_shape[0])

        if num_pos < self.num_samples // 2:
            spine_neg_centroids = self._get_spine_neg_centroids(image_shape,
                self.crop_size, self.num_samples - 2 * num_pos)
        else:
            spine_neg_centroids = self._get_spine_neg_centroids(image_shape,
                self.crop_size, num_pos)

        return sym_neg_centroids + spine_neg_centroids

    def _get_roi_centroids(self, label_arr):
        if self.train:
            # generate positive samples' centroids
            pos_centroids = self._get_pos_centroids(label_arr)

            # generate negative samples' centroids
            neg_centroids = self._get_neg_centroids(pos_centroids,
                label_arr.shape)

            # sample positives and negatives when necessary
            num_pos = len(pos_centroids)
            num_neg = len(neg_centroids)
            if num_pos >= self.num_samples:
                num_pos = self.num_samples // 2
                num_neg = self.num_samples // 2
            elif num_pos >= self.num_samples // 2:
                num_neg = self.num_samples - num_pos

            if num_pos < len(pos_centroids):
                pos_centroids = [pos_centroids[i] for i in np.random.choice(
                    range(0, len(pos_centroids)),
```

```
    size=num_pos, replace=False)]
            if num_neg < len(neg_centroids):
                neg_centroids = [neg_centroids[i]
    for i in np.random.choice(
                    range(0, len(neg_centroids)),
    size=num_neg, replace=False)]

            roi_centroids = pos_centroids +
    neg_centroids
        else:
            roi_centroids = [list(range(0, x, y //
     2))[1:-1] + [x - y // 2]
                for x, y in zip(label_arr.shape,
    self.crop_size)]
            roi_centroids = list(product(*
    roi_centroids))

        roi_centroids = [tuple([int(x) for x in
    centroid])
            for centroid in roi_centroids]

        return roi_centroids

    def _crop_roi(self, arr, centroid):
        roi = np.ones(tuple([self.crop_size] * 3))
     * (-1024)

        src_beg = [max(0, centroid[i] - self.
    crop_size // 2)
            for i in range(len(centroid))]
        src_end = [min(arr.shape[i], centroid[i] +
     self.crop_size // 2)
            for i in range(len(centroid))]
        dst_beg = [max(0, self.crop_size // 2 -
    centroid[i])
            for i in range(len(centroid))]
        dst_end = [min(arr.shape[i] - (centroid[i]
     - self.crop_size // 2),
            self.crop_size) for i in range(len(
    centroid))]
        roi[
            dst_beg[0]:dst_end[0],
            dst_beg[1]:dst_end[1],
            dst_beg[2]:dst_end[2],
        ] = arr[
            src_beg[0]:src_end[0],
            src_beg[1]:src_end[1],
            src_beg[2]:src_end[2],
        ]

        return roi

    def _apply_transforms(self, image):
        for t in self.transforms:
            image = t(image)

        return image


    def __getitem__(self, idx):
        # read image and label
        public_id = self.public_id_list[idx]
        image_path = os.path.join(self.image_dir,
    f"{public_id}-image.nii.gz")
        label_path = os.path.join(self.label_dir,
    f"{public_id}-label.nii.gz")
        image = nib.load(image_path)
        label = nib.load(label_path)
        image_arr = image.get_fdata().astype(np.
    float)
        label_arr = label.get_fdata().astype(np.
    uint8)

        # calculate rois' centroids
        roi_centroids = self._get_roi_centroids(
    label_arr)

        # crop rois
        image_rois = [self._crop_roi(image_arr,
    centroid)
            for centroid in roi_centroids]
        label_rois = [self._crop_roi(label_arr,
    centroid)
            for centroid in roi_centroids]

        if self.transforms is not None:
            image_rois = [self._apply_transforms(
    image_roi)
                for image_roi in image_rois]

        image_rois = torch.tensor(np.stack(
    image_rois)[:, np.newaxis],
            dtype=torch.float)
        label_rois = (np.stack(label_rois) > 0).
    astype(np.float)
        label_rois = torch.tensor(label_rois[:, np
    .newaxis],
            dtype=torch.float)

        return image_rois, label_rois

    @staticmethod
    def collate_fn(samples):
        image_rois = torch.cat([x[0] for x in
    samples])
        label_rois = torch.cat([x[1] for x in
    samples])

        return image_rois, label_rois

    @staticmethod
    def get_dataloader(dataset, batch_size,
    shuffle=False, num_workers=0):
        return DataLoader(dataset, batch_size,
    shuffle,
```

```
            num_workers=num_workers, collate_fn=
    FracNetTrainDataset.collate_fn)


class FracNetInferenceDataset(Dataset):

    def __init__(self, image_path, crop_size=64,
    transforms=None):
        image = nib.load(image_path)
        self.image_affine = image.affine
        self.image = image.get_fdata().astype(np.
    int16)
        self.crop_size = crop_size
        self.transforms = transforms
        self.centers = self._get_centers()

    def _get_centers(self):
        dim_coords = [list(range(0, dim, self.
    crop_size // 2))[1:-1]\
            + [dim - self.crop_size // 2] for dim
    in self.image.shape]
        centers = list(product(*dim_coords))

        return centers

    def __len__(self):
        return len(self.centers)

    def _crop_patch(self, idx):
        center_x, center_y, center_z = self.
    centers[idx]
        patch = self.image[
            center_x - self.crop_size // 2:
    center_x + self.crop_size // 2,
            center_y - self.crop_size // 2:
    center_y + self.crop_size // 2,
            center_z - self.crop_size // 2:
    center_z + self.crop_size // 2
        ]

        return patch

    def _apply_transforms(self, image):
        for t in self.transforms:
            image = t(image)

        return image

    def __getitem__(self, idx):
        image = self._crop_patch(idx)
        center = self.centers[idx]

        if self.transforms is not None:
            image = self._apply_transforms(image)

        image = torch.tensor(image[np.newaxis],
    dtype=torch.float)
```

```
Detection metrics
============================================================
Recall at key FP
          FP=0.5      FP=1      FP=2      FP=4      FP=8
Recall  0.216667  0.216667  0.216667  0.216667  0.216667
Average recall: 0.2167
Maximum recall: 0.2167
Average FP per scan at maximum recall: 0.4375
FPR, Recall in FROC
(0.00000000, 0.00000000)
(0.00000000, 0.00000000)
(0.00000000, 0.00000000)
(0.00000000, 0.00000000)
(0.00000000, 0.00000000)
(0.00000000, 0.00000000)
(0.00000000, 0.00000000)
(0.00000000, 0.00000000)
(0.00000000, 0.00000000)
(0.00000000, 0.00000000)
(0.00000000, 0.00000000)
(0.00000000, 0.00000000)
(0.00000000, 0.00000000)
(0.00000000, 0.00000000)
(0.00000000, 0.00000000)
```

图 2: 修改后的实验结果

```
        return image, center

    @staticmethod
    def _collate_fn(samples):
        images = torch.stack([x[0] for x in
    samples])
        centers = [x[1] for x in samples]

        return images, centers

    @staticmethod
    def get_dataloader(dataset, batch_size,
    num_workers=0):
        return DataLoader(dataset, batch_size,
    num_workers=num_workers,
            collate_fn=FracNetInferenceDataset.
    _collate_fn)
```

主要的训练流程如下:

```
    for epoch in range(num_epochs):
        num = epoch+1
        print("At Epoch"+str(num)+"\n")
        train_loop(model, train_loader, loss_func,
     optimizer, device)
        save_path = model_path + "/epoch_" +str(
    epoch+1) + "_model.pth.tar"
        torch.save(model.state_dict(), save_path)
        val_loop(model, val_loader, loss_func,
    thresh=0.1, batch_size=batch_size, device=
    device)
        torch.cuda.empty_cache()
```

最终我们得到了如图 2 的初步结果:

在判断时，我们手动的将我们原数据集中的.csv 文件的标签改成了仅有 0 和 1 的二分类问题。从图中看到，我们的 FROC 来到了将近 0.22 的区段。作为一个初步的结果，它已经非常的让人振奋了。

```
Detection metrics
============================================================
Recall at key FP
          FP=0.5   FP=1   FP=2   FP=4   FP=8
Recall   0.468085  0.55   0.55   0.55   0.55
Average recall: 0.5336
Maximum recall: 0.5500
Average FP per scan at maximum recall: 0.5875
FPR, Recall in FROC
```

图 3: 最好的实验结果

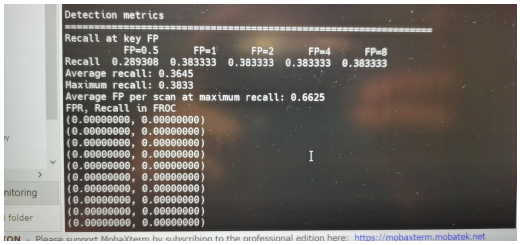

图 4: *100epoch* 的实验结果

5.2. 其他参数的更改

在此之后，我们进行了其他参数的更改，考虑到对参数影响最大的可能是最后标签的判断阈值。在这里，我们主要更改的是代码中这部分中，label 内的参数：

```
def _make_submission_files(pred, image_id, affine):
    pred_label = label(pred > 0).astype(np.int16)
```

我们考虑将 pred 的判断条件（即置信度）线性地从 0.9 向 0 递减，最后对比一下实验结果。在这里，模型的训练 Epoch 均为 50 个。我们最后得到的实验结果见 Github：（由于不可预知的错误，我们没有办法在报告中制表）

其中结果最好的如图 3，50epoch 训练出的模型，参数为 0.1：从以上的结果中，我们可以看到的是，在 50 个 Epoch 的训练下，我们的模型对预测结果中的背景部分的排除能力很强。

而当训练模型为 100epoch 时，参数设置为 0.1，结果为 0.3645，原因应该是模型过拟合。结果如图 4.

我们最终 test 集生成时选择了 100epoch 训练出的模型，所以 froc 指标估计在 0.35 左右。如果选择了 50epoch 训练的模型，froc 估计在 0.55 左右。

6. 思考与体会

6.1. 结论与总结

从以上的实验结果可以看出，我们的模型对二分类的任务的完成依旧是能够完成一定的识别。但是看起来一些部分作为骨折的判断并不是完全的拥有较高的置信度。这一方面是因为我们的结果是基于二分类而实现的，最后的实现结果不会是完全的好。

受时间原因等限制，我们最终没有能够完成全部 5 种骨折的具体分类，所以最后的结果中，也仅显示了二分类的结果。

我们还训练了一个 100Epoch 的模型，这将作为我们最后测试集提交数据时使用的模型。

6.2. 心得与体会

在经过了整个实验的流程之后，虽然在一开始我们自己构建模型编写代码，进行测试的时候遇到了各种各样的困难，经常会因为一些实验中出现的问题与不好的实验结果而感到焦虑与不安。但是回顾整个实验中的流程，我们在自己编写代码的过程中对代码的调试的工作的认识得到了加深。同时，我们也认识到了一个完整的神经网络工程中，需要实现的工作也有很多，最后结果的实现是需要几部分共同协力相互配合着完成的。

从整个的实现流程来看，我们也了解到了神经网络从搭建到实现的一段完整的流程所需要做的具体工作是什么。从数据集读取，到网络设计，再到训练与测试的设计；包括中间的一些细节化的数据处理，如数据增强操作等，这些都是在整个神经网络的构建当中所需要的步骤。在实现的过程中，我们也参阅了需要有用的文档与资料，包括各种工具包的官方文档，和一些论坛等等。这些知识让我们对代码的具体实现以及错误的处理有了更加全面的认识。

最后，我们还是要认识到的一点是，作为一门实践类的课程，我们在实现代码的过程中，从无到有的思路始终应该是属于我们自己的创造。由于最终的实现时间有限，我们可能需要参考一些现成的网络代码来达到迅速的调整我们的思路的目

的。但是对于我们来说，如何能够完全掌握一段现有代码的思路并且进行修改也是我们需要掌握的能力，这是一种学习他人的体现。我们依旧要提高自己独立自主思考的能力，对于日后可能的科研来说，这是极其重要的一项能力。

在本次任务中，最遗憾的一点是我们没能完全坚持下自己最初的思路，进行完善的修改，用我们纯粹的自己编写的代码完成这项任务。但是我们依旧是在他人的一个比较完善的代码的基础之上做出了更改，这是我们本任务比较好的一点。