**nnUnet**

由于源文件代码还挺多，故写一个整理文档。包含训练教程和代码逻辑梳理。

**训练使用教程**

以自己的数据为主，对于部分公开数据集（尤其是 [Medical Segmentation Decthlon](http://medicaldecathlon.com/) 十项全能数据集），作者有写好的代码

**1. 安装**

git clone https://github.com/MIC-DKFZ/nnUNet.git

cd nnUNet

pip install -e .

**硬性需求**

* 仅 linux 系统
* 需要 GPU
* inference 需要至少 4G 显存
* training 需要至少 10G 显存（RTX 2080ti，RTX 3080, RTX 3090 都可以）
* pytorch ，至少需要 cuDNN 8.0.2
* 建议 6 核 CPU
* 亲测 Nvidia 驱动版本也有限制，9000 不够

注：显存要求应该不是固定的，跟数据集有关，例如 meningoma 数据集的层厚比较大，切片数量比较小，实测训练最低只需要 7G，测试最低只需要 4G

**2. 数据集准备**

如果需要自行训练，首先需要按照他们的标准**规范数据集。**找一个地方建一个文件夹专门放 nnUNet 格式的数据，虽然作者说最好放在读取率高的盘上，但是考虑到中间会生成大量数据，还是放在 /data 下面吧，固态硬盘怕是吃不消，然后按照以下格式准备好几个文件夹：

* **nnUNet\_preprocessed**（用于存放预处理之后的数据，**训练期间**会从这个文件夹中读取数据）
* **nnUNet\_raw**（用于存放原数据和 cropped 数据）
* **nnUNet\_cropped\_data**（预处理程序会自动在这底下生成很多数据）
* **nnUNet\_raw\_data**（自己的原数据放在这）
* **nnUNet\_trained\_models**（用于存放模型参数，如果要下载预训练模型，也放在这）

存环境变量，因为后续操作都在命令行中以 nnUNet\_ 开头

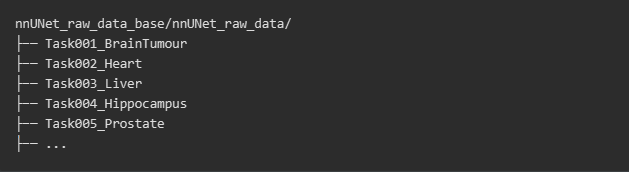
export nnUNet\_raw\_data\_base="/data/sjy/data/nnunet/nnUNet\_raw"

export nnUNet\_preprocessed="/data/sjy/data/nnunet/nnUNet\_preprocessed"

export RESULTS\_FOLDER="/data/sjy/data/nnunet/nnUNet\_trained\_models"

**硬性要求**

该框架支持多种任务，但必须按照一定的规则



* 不同的任务必须以 '**Task'** 和 **3 位数字编号**开头，如 'Task001\_'，后面的任务名称可以随便定（建议自己的数据集编号从 **100** 开始，不会和预训练模型冲突），对于十项全能数据，可以使用预设的指令去处理编号问题

nnUNet\_convert\_decathlon\_task -i /data/sjy/data/nnunet/nnUNet\_raw/nnUNet\_raw\_data/Task08\_HepaticVessel

* 每个 Task 底下需要包含以下文件（必须是 **.nii.gz** 格式存储）

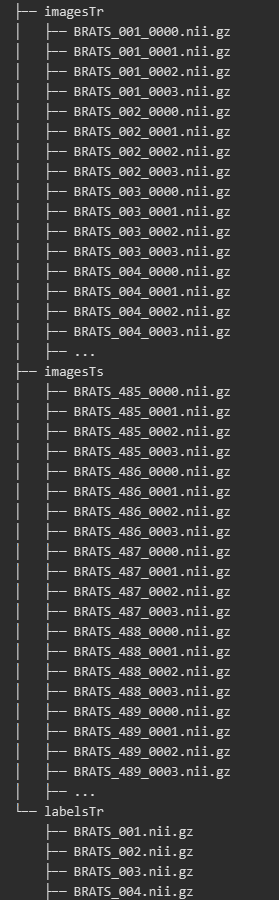


* dataset.json 文件是自己写的，以字典的方式存储指定数据集的一些字段。training 字段用于指明训练数据的**路径**，test 字段用于指明测试数据的路径，若没有测试集，则 "test": []。生成此文件的代码在 ./nnunet/data\_conversion 文件夹中，作者已经给出了一些公开数据集的处理 .py 文件，可以照着改：



* imagesTr 表示待训练数据，以交叉验证的方式训练（项目中是 5 折交叉验证，没有验证集），命名格式为 **case\_identifier\_XXXX.nii.gz**（必须！）
* case 保险起见，就以数据集的**名称**命名（可能需要和 dataset.json 中的 name 对应）
* identifier 是 **3 位数字编号**，所以最多只能有三位数个 cases
* XXXX 用来区分不同的模态，用 **4 位数字编号**区分，如 FLAIR (0000), T1w (0001), T1gd (0002) and T2w (0003)，存放的时候每个数据的物理大小必须对齐，图像必须**配准**
* imagesTs 表示待测试数据，可选
* labelsTr 是标签

注：对于多模态数据，存储数据的时候需要 \_XXXX 作为后缀，但是生成 json 文件的时候一定不要把 XXXX 写入路径，nnUNet 查找文件的时候，是根据 "modality" 字段自动补全多模态后缀。同时训练集数量应以 patient 数量为准，而不是文件数量为准（文件数 = patient 数量 \* 模态数）



**3. 预处理**

如果上述过程没有问题，则可以直接用现成的命令一步到位，它会生成很多数据存在 cropped 文件夹中

# xxx 是 task id

nnUNet\_plan\_and\_preprocess -t XXX --verify\_dataset\_integrity

--verify\_dataset\_integrity 不是必须的，而且亲测不太好用，它是比较绝对的 spacing 差距（多模态，标签），差 0.000.. 几都会报警告，所以差不多没啥问题可以不用这个选项

**4. 开始训练**

* 对于单卡训练

# 默认是在第一张卡上训练，如果要换卡，则需要先指定

export CUDA\_VISIBLE\_DEVICES=X

# XXX 是 task id，训练时使用 5 折交叉, FOLD 用于指定在哪一轴上进行训练，可选范围 [0,1,2,3,4]

# 2d

nnUNet\_train 2d nnUNetTrainerV2 XXX FOLD

# 3d 全分辨率

nnUNet\_train 3d\_fullres nnUNetTrainerV2 XXX FOLD

# 3d 级联（lowres + 全分辨率级联）

nnUNet\_train 3d\_lowres nnUNetTrainerV2 xxx FOLD

nnUNet\_train 3d\_cascade\_fullres nnUNetTrainerV2CascadeFullRes xxx FOLD

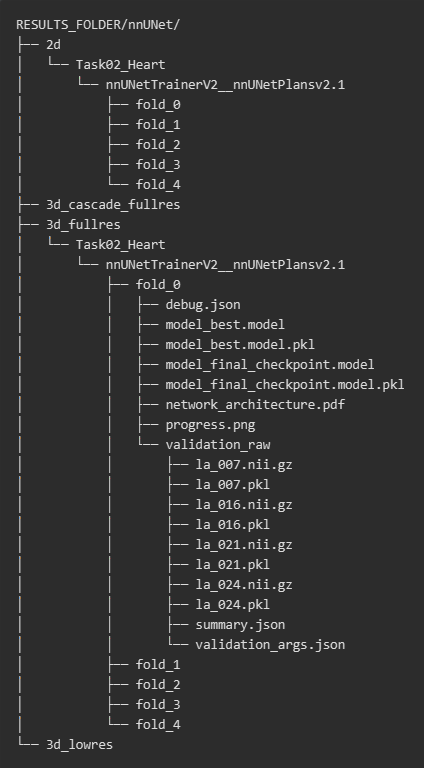
注：lowres 可以单独训练，级联模型必须要在 lowres 的所有 5 折训练结果基础上进行

* 对于多卡训练

export CUDA\_VISIBLE\_DEVICES=0,1

nnUNet\_train\_DP 3d\_fullres nnUNetTrainerV2\_DP 101 4 -gpus 2

输出模型自动存放在 RESULTS\_FOLDER 下



**5. 测试**

测试分两种情况

* 不带标签的测试集 imagesTs，估计是用来打榜的，需要**推理**（待补充）
* 待标签的测试集，可以直接包含在 imagesTr 中走**交叉验证**流程。该流程将数据集切成 5 份，用 4 份**完全训练完**之后会对剩下 1 份进行测试，会在 nnUNet\_trained\_models 底下自动生成相应的测试 label 统计结果和预测标签，非常智能，省去了自己测试的过程。但训练结束之后自动进行的测试是针对最后一个 epoch 的模型，不一定是最好的，推荐自己手动验证，最后的和最好的都看一下效果，命令是：

# valbest 时使用最好的模型，否则使用最后一个模型

nnUNet\_train 3d\_fullres nnUNetTrainerV2 XXX FOLD -val (--valbest)

**代码逻辑梳理**

**1. 预处理**

在训练之前首先需要用 nnUNet\_plan\_and\_preprocess 命令预处理以下数据，相应的代码在 ./nnunet/experiment\_planning/nnUNet\_plan\_and\_preprocess.py 文件中。代码还是比较多的，我这里只贴了关键部分说明思路。主要做了两件事，**crop**（1.1~1.2）和 **preprocess**（1.3~1.7）

**1.1 数据验证**

首先**确保数据干净**

* dataset.json 中规定的 train 和 test 文件都能找到实际的 .nii.gz 文件，label 和文件可以对应
* 所有的多模态文件均存在，并已经配准，pixel grids 已对齐

**1.2 数据裁剪**

对 volume 进行 crop，**裁出非零区域**（在所有模态上都是非零的，**只在 MRI 图像中进行**，CT 数据不进行这样的裁剪，因为 0 值在 CT 中表示的是水，裁剪没有意义）

**1.2.1 CT（待补充）**

CT 数据这一步可以省略，但是需要从源文件提取专属的 **intensity** 属性，后续归一化需要

暂时没用到CT数据就没研究，以后再补上

**1.2.2 MR**

将裁剪完的图像和标签 concat 好存入 cropped 文件夹中，也就是一个 case 的维度是 [N+1, D, H, W]。对于没有标签的样本，程序会生成一个假的 "label"，过程如下：

def create\_nonzero\_mask(data):

""" 对一个 case 找一个所有模态均为非 0 区域的三维 mask """

from scipy.ndimage import binary\_fill\_holes

"data must have shape (C, X, Y, Z) or shape (C, X, Y)"

assert len(data.shape) == 4 or len(data.shape) == 3

nonzero\_mask = np.zeros(data.shape[1:], dtype=bool)

# 第一维是模态，遍历所有模态

for c in range(data.shape[0]):

this\_mask = data[c] != 0

nonzero\_mask = nonzero\_mask | this\_mask

nonzero\_mask = binary\_fill\_holes(nonzero\_mask)

return nonzero\_mask

def get\_bbox\_from\_mask(mask, outside\_value=0):

""" 在 xyz 三个维度上找到非零区域的最小 bbox """

mask\_voxel\_coords = np.where(mask != outside\_value)

minzidx = int(np.min(mask\_voxel\_coords[0]))

maxzidx = int(np.max(mask\_voxel\_coords[0])) + 1

minxidx = int(np.min(mask\_voxel\_coords[1]))

maxxidx = int(np.max(mask\_voxel\_coords[1])) + 1

minyidx = int(np.min(mask\_voxel\_coords[2]))

maxyidx = int(np.max(mask\_voxel\_coords[2])) + 1

return [[minzidx, maxzidx], [minxidx, maxxidx], [minyidx, maxyidx]]

def crop\_to\_nonzero(data, seg=None, nonzero\_label=-1):

""" 裁剪函数 """

# 在所有模态上找一个公共三维 nonzero\_mask

nonzero\_mask = create\_nonzero\_mask(data)

# 从公共 nonzero\_mask 得到最小长方体

bbox = get\_bbox\_from\_mask(nonzero\_mask, 0)

# 切原图，对模态分别切，切完再拼

cropped\_data = []

for c in range(data.shape[0]):

cropped = crop\_to\_bbox(data[c], bbox)

cropped\_data.append(cropped[None])

# [N,D,H,W]

data = np.vstack(cropped\_data)

if seg is not None:

cropped\_seg = []

for c in range(seg.shape[0]):

cropped = crop\_to\_bbox(seg[c], bbox)

cropped\_seg.append(cropped[None])

# [N,D,H,W]

seg = np.vstack(cropped\_seg)

# 切 nonzero\_mask，切完还是个长方体，[1,D,H,W]

nonzero\_mask = crop\_to\_bbox(nonzero\_mask, bbox)[None]

if seg is not None:

# 如果存在 label，对切完的 seg 和 nonzero\_mask 同为 0 的地方设置成 zonzero\_label=-1

# 这些区域既不含分割目标，在 MRI 图像中也属于无意义区域，此时 seg 包含 [-1,0,...]

# 这么做是为了后续归一化时的计算

seg[(seg == 0) & (nonzero\_mask == 0)] = nonzero\_label

else:

# 如果不存在 label，只按 nonzero\_mask 切

nonzero\_mask = nonzero\_mask.astype(int)

nonzero\_mask[nonzero\_mask == 0] = nonzero\_label

nonzero\_mask[nonzero\_mask > 0] = 0

# 把切完的 nonzero\_mask 作为 seg 使用，此时 seg 只含 [-1,0]，除了 MRI 图像中的无意义

# 区域，其他地方的 label 全为 0

seg = nonzero\_mask

return data, seg, bbox

**1.1 ~ 1.2 总结**

crop 做完之后会在 nnUNet\_cropped\_data 文件夹下生成以下文件：

nnUNet\_raw/nnUNet\_cropped\_data/Task101\_Meningoma/

—— gt\_segmentations # 直接从原始的 labelsTr 里拷过来

—— dataset.json # 直接复制原始的 dataset.json

—— dataset\_properties.pkl # 存储了整个数据集的信息

—— meningoma\_001.npz # 裁剪过的 volume 的 array，所有模态都已拼好，label 拼到最后一维

[N+1,D,H,W]

—— meningoma\_001.pkl # 该 case 的属性，细节如下

—— meningoma\_002.npz # 同 001，每个 case 都会有 npz 文件

—— meningoma\_002.pkl # 同 001，每个 case 都会有 pkl 文件

—— ...

整个数据集属性 dataset\_properties.pkl 存储属性如下：

dataset\_properties['all\_sizes'] = sizes # crop 之后的 sizes 集合

dataset\_properties['all\_spacings'] = spacings # crop 之后的 spacings 集合

dataset\_properties['all\_classes'] = all\_classes # 正整数，只列前景

dataset\_properties['modalities'] = modalities # {idx: modality name}

dataset\_properties['intensityproperties'] = intensityproperties # CT 有，MRI None

dataset\_properties['size\_reductions'] = size\_reductions # {patient\_id: size\_reduction}

裁剪率，现 size / 原 size

单个 case 存储的属性如下：

#============= 原始图像信息 =============

properties["original\_size\_of\_raw\_data"] = np.array(data\_itk[0].GetSize())[[2, 1, 0]]

properties["original\_spacing"] = np.array(data\_itk[0].GetSpacing())[[2, 1, 0]]

properties["list\_of\_data\_files"] = data\_files # 多个模态，路径

properties["seg\_file"] = seg\_file

properties["itk\_origin"] = data\_itk[0].GetOrigin()

properties["itk\_spacing"] = data\_itk[0].GetSpacing()

properties["itk\_direction"] = data\_itk[0].GetDirection()

#============= crop 部分加信息 =============

properties["crop\_bbox"] = bbox

properties['classes'] = np.unique(seg) # [-1,0,1...]

properties["size\_after\_cropping"] = data[0].shape

**1.3 数据归一化**

判断是否需要用 nonzero\_mask 来做归一化。仅 MRI。对于 MRI，仅 reductions 裁剪率（原 size / 现 size）中位数 < 0.75 才使用 nonzero\_mask 做归一化

modalities = self.dataset\_properties['modalities']

num\_modalities = len(list(modalities.keys()))

use\_nonzero\_mask\_for\_norm = OrderedDict()

# 以模态为单位判断是否要用 nonzero\_mask 做归一化，例如 {'t1':True, 't2':False}

for i in range(num\_modalities):

# CT 没有 nonzero\_mask

if "CT" in modalities[i]:

use\_nonzero\_mask\_for\_norm[i] = False

else:

all\_size\_reductions = []

# reductions 这个是在 cropped 时候就存好的一个属性，直接读出来

for k in self.dataset\_properties['size\_reductions'].keys():

all\_size\_reductions.append(self.dataset\_properties['size\_reductions'][k])

# 整个数据集裁剪率的中位数 < 0.75，就使用 nonzero\_mask 做归一化

if np.median(all\_size\_reductions) < 3 / 4.:

print("using nonzero mask for normalization")

use\_nonzero\_mask\_for\_norm[i] = True

else:

print("not using nonzero mask for normalization")

use\_nonzero\_mask\_for\_norm[i] = False

**1.4 数据重采样**

由于病人体型不同，而最后数字成像的分辨率是相同的，可能会导致失真。因此医学图像中都会带有 slice thickness 和 pixel spacing 这样的属性去还原真实物体。pixel spacing 对应的是 xy 轴，slice thickness 对应的是 z 轴。x 方向对应人体左右方向，y 方向对应人体前（胸）后（背）方向，z 方向对应人体上（头）下（脚）方向

* **各向异性（anisotropic）**：指 xyz 三个扫描方向的 pixel spacing 不一致，如 1mm\*1mm\*5mm
* **各项同性（isotropic）**：指 xyz 三个扫描方向的 pixel spacing 一致，如 1mm\*1mm\*1mm

通常 xy 方向的 spacing 较小，分辨率较高，z 方向的 spacing 较大，分辨率较低。预处理的时候一般通过像素插值调整 pixel spacing 为各向同性（从直觉上可以理解，但是理论依据我没找到）。通常存在两种方式：

* **上采样**：把 pixel spacing 统一向最小 spacing（最高分辨率）靠拢。能够最大程度保留图像信息，但生成的数据量大，训练较困难。同时，当 xy 和 z 的差距过大时，插值的 xy 层面不一定包含真正有价值的信息
* **下采样**：把 pixel spacing 统一向最大 spacing（最低分辨率）靠拢。损失不少图像信息，但数据量小，训练较容易

一般来讲，当 z 方向 spacing 过大时（>= 3mm）通常采用下采样更合适。而作者是**从数据集得到 spacing 基准（target\_spacing）**，所有数据向这个基准上下采样：

# 先以 50% 的四分位数（中位数）作为 pixel spacing 的基准

target\_spacing = np.percentile(np.vstack(spacings), 50, 0)

target\_size = np.percentile(np.vstack(sizes), 50, 0)

""" 但是由于某些数据集可能 pixel spacing 差异性比较大，直接用中位数作为基准可能会导致插值的时候

出现不良的伪像（见上采样的缺点），所以作者在这又细化了一下 target 的设置 """

# 找到 spacing 最大的轴

worst\_spacing\_axis = np.argmax(target\_spacing)

other\_axes = [i for i in range(len(target\_spacing)) if i != worst\_spacing\_axis]

other\_spacings = [target\_spacing[i] for i in other\_axes]

other\_sizes = [target\_size[i] for i in other\_axes]

# 当某个轴的分辨率远小于其他轴

has\_aniso\_spacing = target\_spacing[worst\_spacing\_axis] > (anisotropy\_threshold \* min(other\_spacings))

# 当这个最小分辨率轴包含的 voxels 数量（我觉得 voxels 可以和 size 和分辨率等同）远小于其他轴

has\_aniso\_voxels = target\_size[worst\_spacing\_axis] \* anisotropy\_threshold < min(other\_sizes)

# 其实说白了就是 xyz spacing 差距过大了，通常 xy spacing 差不多，那也就是 xy 和 z 的差距太大了

# 同时符合两种情况，说明该数据集内差异很大，要小心选择 target spacing，而不是直接用中位数

# 这里作者列了一个数据集，它的 spacing 大多是 10mm\*1.5mm\*1.5mm 的，但是也出现了一些 case，它们

# 的 spacing 是 5mm\*1.5mm\*1.5mm 或者 6mm\*1.5mm\*1.5mm 的，作者就是要处理这种情况（中位数不合适）

if has\_aniso\_spacing and has\_aniso\_voxels:

spacings\_of\_that\_axis = np.vstack(spacings)[:, worst\_spacing\_axis]

# 对于这种情况，使用 10% 四分位数作为基准，相对于中位数，选择了一个更小的值

# 但是为什么是 10 也没说

target\_spacing\_of\_that\_axis = np.percentile(spacings\_of\_that\_axis, 10)

# 该轴的 spacing 就算再小也不能小过其他轴

if target\_spacing\_of\_that\_axis < min(other\_spacings):

target\_spacing\_of\_that\_axis = max(min(other\_spacings), target\_spacing\_of\_that\_axis) + 1e-5

target[worst\_spacing\_axis] = target\_spacing\_of\_that\_axis

""" shape/分辨率/size 直接从数据集选用 median 即可，shape 并不参与重采样，只是在 1.5 会用到 """

# 根据 target\_spacing 算新的 shapes（sizes）

new\_shapes = [np.array(i) / target\_spacing \* np.array(j) for i, j in zip(spacings, sizes)]

# 在新的 shapes 里面找到中位数

median\_shape = np.median(np.vstack(new\_shapes), 0)

**1.5 设置网络参数**

根据 1.4 得到的 **target\_spacing** 和 **median\_shape** 计算**网络参数**，包括 patch size，卷积核大小，池化核大小等。整个代码见文件 ./nnunet/experiment\_planning/experiment\_planner\_baseline\_3DUNet\_v21.py 中的 get\_properties\_for\_stage 函数，最终生成的就是一个 'plan' 字典，包含了一个 **stage** 中的参数（stage 是什么见 1.6）：

plan = {'batch\_size': batch\_size, # 1.5.5

'num\_pool\_per\_axis': network\_num\_pool\_per\_axis, # 1.5.2 + 1.5.4

'patch\_size': input\_patch\_size, # 1.5.1 + 1.5.3 + 1.5.4

'median\_patient\_size\_in\_voxels': new\_median\_shape, # 1.5.1

'current\_spacing': current\_spacing, # 1.6，区分 fullres 和 lowres

'original\_spacing': original\_spacing, # 1.4 原图的 spacing

'do\_dummy\_2D\_data\_aug': do\_dummy\_2D\_data\_aug, # 1.5.5

'pool\_op\_kernel\_sizes': pool\_op\_kernel\_sizes, # 1.5.2 + 1.5.4

'conv\_kernel\_sizes': conv\_kernel\_sizes, # 1.5.2 + 1.5.4

}

注：从 1.5 开始，所有的 target\_spacing 和 median\_shape 均是转置过的，spacing 最大的轴调到第一位，这个属于代码实现的细节，为了代码的完整我放在这了

max\_spacing\_axis = np.argmax(target\_spacing)

remaining\_axes = [i for i in list(range(3)) if i != max\_spacing\_axis]

# spacing 最大的轴调到第一位，索引

self.transpose\_forward = [max\_spacing\_axis] + remaining\_axes

# forward 之后再倒回原来的顺序，索引

self.transpose\_backward = [np.argwhere(np.array(self.transpose\_forward) == i)[0][0]

for i in range(3)]

target\_spacing = np.array(target\_spacing)[self.transpose\_forward]

median\_shape = np.array(median\_shape)[self.transpose\_forward]

为了更好地梳理作者处理数据的逻辑，我把这个函数拆成以下几个点来写。先申明这个函数需要的参数，之后不做赘述：

"""

:param current\_spacing: 当前 stage 的 target\_spacing，不同 stage 的 current\_spacing 是不同的

:param original\_spacing: 数据集的 target\_spacing

:param original\_shape: 数据集的 median\_shape

:param num\_cases: 样本数量

:param num\_modalities: 模态数量

:param num\_classes: 类别数量

:return: 上面的 plan 字典

"""

**1.5.1 初始 patch size**

计算一个适合数据集的 input patch size 作为初始，之后还会根据其他的条件来更新这个 patch size

# 分辨率应该是整型

new\_median\_shape = np.round(original\_spacing / current\_spacing \* original\_shape).astype(int)

# 单位体积（1mm\*1mm\*1mm）的 size

input\_patch\_size = 1 / np.array(current\_spacing)

# 归一化（三个轴均值为 1，其实我没明白为什么要这么做）

input\_patch\_size /= input\_patch\_size.mean()

# 初始化 patch size（功能上相当于最大 size）

input\_patch\_size \*= 1 / min(input\_patch\_size) \* 512

input\_patch\_size = np.round(input\_patch\_size).astype(int)

# 若该 size 比 median\_shape 还大，就截断到 median\_shape，因为再大就没有意义了

input\_patch\_size = [min(i, j) for i, j in zip(input\_patch\_size, new\_median\_shape)]

注：为什么作者初始化 patch size 的时候从 median\_shape 开始绕了一圈，而不是直接用 median\_shape？

（其实我没看懂，但感觉很有道理的样子）

The rationale behind this is that for some organ of interest the acquisition method will most likely be chosen such that the field of view and voxel resolution go hand in hand to show the doctor what they need to see. This assumption may be violated for some modalities with anisotropy (cine MRI) but we will have t live with that. In future experiments I will try to 1) base input patch size match aspect ratio of input size in mm (instead of voxels) and 2) to try to enforce that we see the same 'distance' in all directions (try to maintain equal size in mm of patch)

**1.5.2 卷积核与池化核**

这两个参数是跟 **spacing** 有关的，如果某一个轴的 spacing 相对其他轴特别大/小，该轴的卷积核大小为 1；对于spacing 相似的轴，卷积核大小为 3。举个例子，最常见的情况是 xy 差不多大，但 z 差距较大，如 6mm\*0.5mm\*0.5mm，假设对应的分辨率为 [24, 512, 512]，那么得到的卷积核大小为 1\*3\*3，相当于 xy 面的二维卷积。实际上由于 z 轴对应的 voxels 跟 xy 轴相比实在太少，voxels 之间差距太大用 kernel=3 卷出来效果也不一定好，所以这么设置是合理的。池化核也是类似的操作，大小只在 1 和 2 中选。这个过程会**循环迭代多次直到已经没有可以池化的轴**，生成的是一个参数列表，对应着 unet 的结构。代码如下：

while True:

""" 对应 unet，迭代生成的参数列表，每池化一次分辨率减半，spacing 增一倍，直到无法池化 """

min\_spacing = min(current\_spacing)

valid\_axes\_for\_pool = [i for i in range(dim) if current\_spacing[i] / min\_spacing < 2]

axes = []

for a in range(dim):

my\_spacing = current\_spacing[a]

# 遍历每个轴，找和它 spacing 差不多（a<2b 且 b<2a）的轴，这种“轴对”叫 partner

# partner 并不是严格成对的，可以只含它本身，如 [[0],[1,2]]

partners = [i for i in range(dim)

if current\_spacing[i] / my\_spacing < 2

and my\_spacing / current\_spacing[i] < 2]

# 找一个最大 partner

if len(partners) > len(axes):

axes = partners

# 对于 partner 中的轴，卷积核大小为 3，否则为 1

conv\_kernel\_size = [3 if i in axes else 1 for i in range(dim)]

# 并不是所有轴都可以池化的，用最小特征图大小和最大池化次数做限制

valid\_axes\_for\_pool = [i for i in valid\_axes\_for\_pool

if current\_size[i] >= 2\*min\_feature\_map\_size] # 4

valid\_axes\_for\_pool = [i for i in valid\_axes\_for\_pool

if num\_pool\_per\_axis[i] < max\_numpool] # 999

# 如果没有可以 pool 的轴就跳出

if len(valid\_axes\_for\_pool) == 0:

break

other\_axes = [i for i in range(dim) if i not in valid\_axes\_for\_pool]

pool\_kernel\_sizes = [0] \* dim

# 可以池化的轴 pool kernel size = 2

for v in valid\_axes\_for\_pool:

pool\_kernel\_sizes[v] = 2

num\_pool\_per\_axis[v] += 1

# pool 一次相当于分辨率减半，spacing\*2

current\_spacing[v] \*= 2

current\_size[v] = np.ceil(current\_size[v] / 2)

# 不可以池化的轴 pool kernel size = 1

for nv in other\_axes:

pool\_kernel\_sizes[nv] = 1

pool\_op\_kernel\_sizes.append(pool\_kernel\_sizes)

conv\_kernel\_sizes.append(conv\_kernel\_size)

# 最后多加了一个 3\*3\*3 的卷积核，是给 bottleneck 的，也就是 unet 最底部的那个卷积块

conv\_kernel\_sizes.append([3]\*dim)

**1.5.3 更新 patch size**

1.5.1 中得到的 input\_patch\_size 并不一定适合 **CNN** 的操作，在 CNN 下采样的时候一定是要保证可以被 2 整除，所以还需要将这个 size 规范为最接近原值的能被下采样整除的大小

# 根据池化的次数得到下采样倍率

must\_be\_divisible\_by = 2 \*\* np.array(net\_numpool\_per\_axis)

# 重新计算 patch size，如 [22,510,447]--pool[4,128,64]-->[24,512,448]

new\_shp = [input\_patch\_size[i] + must\_be\_divisible\_by[i] - input\_patch\_size[i]

% must\_be\_divisible\_by[i] for i in range(len(input\_patch\_size))]

for i in range(len(input\_patch\_size)):

if input\_patch\_size[i] % must\_be\_divisible\_by[i] == 0:

new\_shp[i] -= must\_be\_divisible\_by[i]

new\_shp = np.array(new\_shp).astype(int)

**1.5.4 GPU 相关**

1.5.3 中更新的 patch size 符合数据集也符合 CNN，但是不一定符合 **GPU**，所以这里还要根据实际的资源再调整 patch size。如果 patch size 过大，就减少分辨率，一次减一个轴，从分辨率最大的轴开始减，按照 must\_be\_divisible\_by 减，代码如下：

def compute\_approx\_vram\_consumption(patch\_size, num\_pool\_per\_axis, base\_num\_features,

max\_num\_features,num\_modalities, num\_classes,

pool\_op\_kernel\_sizes, deep\_supervision=False,

conv\_per\_stage=2):

""" 根据参数统计整个网络中所有特征图的存储量，不是真正的显存占用量，但是和显存占用成正比 """

if not isinstance(num\_pool\_per\_axis, np.ndarray):

num\_pool\_per\_axis = np.array(num\_pool\_per\_axis)

npool = len(pool\_op\_kernel\_sizes)

map\_size = np.array(patch\_size)

# conv\_per\_stage \* 2 + 1 是算上了 encoder, decoder 和 transposed conv

tmp = np.int64((conv\_per\_stage \* 2 + 1) \* np.prod(map\_size, dtype=np.int64) \* base\_num\_features +

num\_modalities \* np.prod(map\_size, dtype=np.int64) +

num\_classes \* np.prod(map\_size, dtype=np.int64))

num\_feat = base\_num\_features

# 把经过池化的所有特征图都统计上

for p in range(npool):

for pi in range(len(num\_pool\_per\_axis)):

map\_size[pi] /= pool\_op\_kernel\_sizes[p][pi]

num\_feat = min(num\_feat \* 2, max\_num\_features)

# 到达最后一个池化的时候，已经到达了 unet 的最底部，这个地方只需要一个 conv 块，

# 没有 encoder / decoder / transposed conv

num\_blocks = (conv\_per\_stage \* 2 + 1) if p < (npool - 1) else conv\_per\_stage

tmp += num\_blocks \* np.prod(map\_size, dtype=np.int64) \* num\_feat

if deep\_supervision and p < (npool - 2):

tmp += np.prod(map\_size, dtype=np.int64) \* num\_classes

return tmp

# 对照值，对于这里的参数我也很好奇，可能这就是作者要求显存的原因？

ref = Generic\_UNet.use\_this\_for\_batch\_size\_computation\_3D \

\* self.unet\_base\_num\_features \

/ Generic\_UNet.BASE\_NUM\_FEATURES\_3D # 520000000\*32/30

# 当前 input patch size 实际需要使用的存储量

here = Generic\_UNet.compute\_approx\_vram\_consumption(new\_shp, network\_num\_pool\_per\_axis,

self.unet\_base\_num\_features,

self.unet\_max\_num\_filters,

num\_modalities, num\_classes,

pool\_op\_kernel\_sizes,

conv\_per\_stage=self.conv\_per\_stage)

while here > ref:

# 各个轴的分辨率从小到大排序，返回索引，[-1] 是最大的轴，从最大的开始降

axis\_to\_be\_reduced = np.argsort(new\_shp / new\_median\_shape)[-1]

tmp = deepcopy(new\_shp)

# 降的时候按照 must\_be\_divisible\_by 来降

tmp[axis\_to\_be\_reduced] -= shape\_must\_be\_divisible\_by[axis\_to\_be\_reduced]

# 根据新的分辨率重新计算 must\_be\_divisible\_by

\_, \_, \_, \_, shape\_must\_be\_divisible\_by\_new = \

get\_pool\_and\_conv\_props(current\_spacing, tmp,

self.unet\_featuremap\_min\_edge\_length,

self.unet\_max\_numpool,

)

new\_shp[axis\_to\_be\_reduced] -= shape\_must\_be\_divisible\_by\_new[axis\_to\_be\_reduced]

# 分辨率降完之后连带着之前的卷积核池化核等等参数都需要更新

network\_num\_pool\_per\_axis, pool\_op\_kernel\_sizes, conv\_kernel\_sizes, new\_shp, \

shape\_must\_be\_divisible\_by = get\_pool\_and\_conv\_props(current\_spacing, new\_shp,

self.unet\_featuremap\_min\_edge\_length,

self.unet\_max\_numpool,

)

# 重新计算存储量，不符合就继续减分辨率

here = Generic\_UNet.compute\_approx\_vram\_consumption(new\_shp, network\_num\_pool\_per\_axis,

self.unet\_base\_num\_features,

self.unet\_max\_num\_filters,

num\_modalities, num\_classes,

pool\_op\_kernel\_sizes,

conv\_per\_stage=self.conv\_per\_stage

)

**1.5.5 其余参数**

* batch size

batch\_size = Generic\_UNet.DEFAULT\_BATCH\_SIZE\_3D # 2

batch\_size = int(np.floor(max(ref / here, 1) \* batch\_size))

# batch size 不能太大

max\_batch\_size = np.round(self.batch\_size\_covers\_max\_percent\_of\_dataset \* dataset\_num\_voxels /

np.prod(input\_patch\_size, dtype=np.int64)).astype(int)

max\_batch\_size = max(max\_batch\_size, self.unet\_min\_batch\_size) # unet\_min\_batch\_size=2

batch\_size = min(batch\_size, max\_batch\_size)

* 如果存在轴的分辨率差距太大，就做一下 2d 数据扩充（对于 3D 情况来说，一般就是 z 和 xy 相差太大）

# 这里 input\_patch\_size[0] 对应的是分辨率最小、spacing 最大的轴，见 1.5 中的 transpose

do\_dummy\_2D\_data\_aug = (max(input\_patch\_size) / input\_patch\_size[0])

> self.anisotropy\_threshold

**1.6 多阶段**

如果 1.5 中最终得到的 input patch size 占原图（1.4 得到的 median\_shape）的比例比较小，就使用**两阶段（stage）**。1.5 中的实验参数都是针对 3d fullres stage 的，加一个阶段 3d lowres stage 做级联。分辨率是通过慢慢放大 spacing 迭代的：

# lowres spacing 从原 spacing（target\_spacing）开始往上升

lowres\_stage\_spacing = deepcopy(target\_spacing)

# 计算原图有多少 voxels

num\_voxels = np.prod(median\_shape, dtype=np.int64)

# 1.5 plan 中的 patch size 有多少 voxels

architecture\_input\_voxels\_here = np.prod(self.plans\_per\_stage[-1]['patch\_size'], dtype=np.int64)

""" 迭代放大 lowres stage 的 spacing，直到 patch size 占原图比例足够大，按照 1.01 倍慢慢放大 """

while num\_voxels > how\_much\_of\_a\_patient\_must\_the\_network\_see\_at\_stage0 \* architecture\_input\_voxels\_here:

max\_spacing = max(lowres\_stage\_spacing)

# 如果有一个轴的 spacing 特别大，先放大这个轴

if np.any((max\_spacing / lowres\_stage\_spacing) > 2):

lowres\_stage\_spacing[(max\_spacing / lowres\_stage\_spacing) > 2] \*= 1.01

# 否则就大家一起放大

else:

lowres\_stage\_spacing \*= 1.01

# 由于是针对 lowres 做的，num\_voxels 也应该跟着 lowres 的 spacing 调整

num\_voxels = np.prod(target\_spacing / lowres\_stage\_spacing \* median\_shape, dtype=np.int64)

# 这个转置就是一个代码上的细节操作，我之前都省略了，不影响逻辑，这里为了代码的完整我补上了

lowres\_stage\_spacing\_transposed = np.array(lowres\_stage\_spacing)[self.transpose\_forward]

# get\_proterties\_for\_stage 干的事情就是 1.5 的东西，# 算 lowres 的网络参数 'plan'

new = self.get\_properties\_for\_stage(lowres\_stage\_spacing\_transposed,

target\_spacing\_transposed, median\_shape\_transposed,

len(self.list\_of\_cropped\_npz\_files),

num\_modalities, len(all\_classes) + 1)

# 算 lowres 的 patch size 的 voxels

architecture\_input\_voxels\_here = np.prod(new['patch\_size'], dtype=np.int64)

整个大 plan 中包含的参数为

plans = {'num\_stages': len(list(self.plans\_per\_stage.keys())),

'num\_modalities': num\_modalities,

'modalities': modalities, # ['t1','t2',...]

'normalization\_schemes': normalization\_schemes, # 区分 CT / MR，['CT','nonCT']

'dataset\_properties': self.dataset\_properties, # dataset\_properties.pkl

'list\_of\_npz\_files': self.list\_of\_cropped\_npz\_files, # cropped 文件路径

'original\_spacings': spacings, # 数据集所有的 spacings

'original\_sizes': sizes, # 数据集所有的 sizes（分辨率）

'preprocessed\_data\_folder': self.preprocessed\_output\_folder,

'num\_classes': len(all\_classes),

'all\_classes': all\_classes,

'base\_num\_features': self.unet\_base\_num\_features, # =30，用于计算显存占用

'use\_mask\_for\_norm': use\_nonzero\_mask\_for\_normalization, # 1.3, {0:False,...}

'keep\_only\_largest\_region': only\_keep\_largest\_connected\_component, # None

'min\_region\_size\_per\_class': min\_region\_size\_per\_class, # None

'min\_size\_per\_class': min\_size\_per\_class, # None

'transpose\_forward': self.transpose\_forward, # 把 spacing 最大的轴放在第一位

'transpose\_backward': self.transpose\_backward, # 把 spacing 倒回原来的顺序

'data\_identifier': self.data\_identifier,

'plans\_per\_stage': self.plans\_per\_stage, # stage: plan，1.5

'preprocessor\_name': self.preprocessor\_name,

'conv\_per\_stage': self.conv\_per\_stage, # unet 中的一个 block 包含几个 conv

}

注：这里的 'keep\_only\_largest\_region'，'min\_region\_size\_per\_class'，'min\_size\_per\_class' 作者没有任何处理直接设置成了 None，感觉是还没开发完的功能

**1.7 预处理实现**

1.3~1.6 只是生成了配置文件，并没有真正对数据进行操作，这部分才是将之前的预处理设置真正地作用在数据上。相关的代码在 ./nnunet/preprocessing/preprocessing.py 文件中，默认使用 GenericPreprocessor 类进行处理（1.6 plan 中的 '**preprocessor\_name**' tag）。处理的时候从 stage 配置文件里面读参数，以**一个病人为单位**进行 操作，生成的数据存入以下文件夹：

nnUNet\_preprocessed/Task101\_Meningoma/nnUNetData\_plans\_v2.1\_stage0/

—— meningoma\_001.npz

—— meningoma\_001.pkl

—— ...

**1.7.1 重采样**

根据当前病人的 **original\_spacing** 和 shape，和当前 stage 的 **target\_spacing**，重新计算此病人的 new\_shape。从原始 shape 到 new\_shape 用的是 skimage 中的 resize 函数。如果 original\_spacing 或 target\_spacing 存在各向异性的轴，要 separate 出来单独处理，inplane 平面先常规 resize，separate 轴再 resize；如果不存在各向异性，就直接常规 resize。

from scipy.ndimage.interpolation import map\_coordinates

new\_shape = np.round(((np.array(original\_spacing) / np.array(target\_spacing))

.astype(float) \* shape)).astype(int)

# 如果轴 axis 为各向异性，且需要 resize

if shape[axis] != new\_shape[axis]:

""" 以下 resize 方法是从 skimage 的 resize() 方法改进而来（这个就很有灵性了）"""

rows, cols, dim = new\_shape[0], new\_shape[1], new\_shape[2]

orig\_rows, orig\_cols, orig\_dim = reshaped\_data.shape

row\_scale = float(orig\_rows) / rows

col\_scale = float(orig\_cols) / cols

dim\_scale = float(orig\_dim) / dim

map\_rows, map\_cols, map\_dims = np.mgrid[:rows, :cols, :dim]

map\_rows = row\_scale \* (map\_rows + 0.5) - 0.5

map\_cols = col\_scale \* (map\_cols + 0.5) - 0.5

map\_dims = dim\_scale \* (map\_dims + 0.5) - 0.5

coord\_map = np.array([map\_rows, map\_cols, map\_dims])

# 如果 resize 的对象是 image

if not is\_seg or order\_z == 0:

reshaped\_final\_data.append(map\_coordinates(reshaped\_data, coord\_map,

order=order\_z, cval=cval,

mode='nearest')[None])

# 如果 resize 的对象是 label，做成 multihot 的形式

else:

unique\_labels = np.unique(reshaped\_data)

reshaped = np.zeros(new\_shape, dtype=dtype\_data)

for i, cl in enumerate(unique\_labels):

reshaped\_multihot = np.round(map\_coordinates((reshaped\_data == cl)

.astype(float), coord\_map,

order=order\_z, cval=cval,

mode='nearest'))

reshaped[reshaped\_multihot > 0.5] = cl

reshaped\_final\_data.append(reshaped[None])

注：对比了一下发现在 resize 的时候 inplane 的 mode 为 '**edge**'，separate 轴的 mode 为 '**nearest**'。

**1.7.2 归一化**

根据 1.5 plan 中的 '**normalization\_schemes**' tag，对 CT/CT2 和 MR 模态区分处理

* CT/CT2：使用数据集的 intensity 属性
* MR：如果使用 nonzero\_mask，计算均值和方差的时候，只计算 label 中 >= 0 的区域，排除值为 -1 的区域，也就是只统计有意义的区域的像素值（1.2 中做 -1 标签的目的就在这）

for c in range(len(data)):

scheme = self.normalization\_scheme\_per\_modality[c]

# CT 模态

if scheme == "CT":

# clip to lb and ub from train data foreground and use foreground mn and sd from training data

assert self.intensityproperties is not None, \

"ERROR: if there is a CT then we need intensity properties"

mean\_intensity = self.intensityproperties[c]['mean']

std\_intensity = self.intensityproperties[c]['sd']

lower\_bound = self.intensityproperties[c]['percentile\_00\_5']

upper\_bound = self.intensityproperties[c]['percentile\_99\_5']

data[c] = np.clip(data[c], lower\_bound, upper\_bound)

# 归一化

data[c] = (data[c] - mean\_intensity) / std\_intensity

# 实际上 CT 的 nonzero\_mask 就是 False

if use\_nonzero\_mask[c]:

data[c][seg[-1] < 0] = 0

# CT2 模态

elif scheme == "CT2":

# clip to lb and ub from train data foreground, use mn and sd form each case for normalization

assert self.intensityproperties is not None, \

"ERROR: if there is a CT then we need intensity properties"

lower\_bound = self.intensityproperties[c]['percentile\_00\_5']

upper\_bound = self.intensityproperties[c]['percentile\_99\_5']

mask = (data[c] > lower\_bound) & (data[c] < upper\_bound)

data[c] = np.clip(data[c], lower\_bound, upper\_bound)

mn = data[c][mask].mean()

sd = data[c][mask].std()

# 归一化

data[c] = (data[c] - mn) / sd

if use\_nonzero\_mask[c]:

data[c][seg[-1] < 0] = 0

# MR 模态

else:

if use\_nonzero\_mask[c]:

mask = seg[-1] >= 0

else:

mask = np.ones(seg.shape[1:], dtype=bool)

# 只对有意义的区域做归一化

data[c][mask] = (data[c][mask] - data[c][mask].mean()) / (data[c][mask].std()

+ 1e-8)

data[c][mask == 0] = 0

**1.7.3 前景位置**

对于每个类别，在该类别的前景位置上随机采样一些点，记录这些**采样的前景点的位置**，存入该 case 的 properties 文件

num\_samples = 10000

# 至少要有 1% 的前景被采样到，否则会很稀疏

min\_percent\_coverage = 0.01 # at least 1% of the class voxels need to be selected, otherwise it may be too sparse

rndst = np.random.RandomState(1234)

class\_locs = {}

# 对于所有前景类别

for c in all\_classes:

# all\_data[-1] 是标签 seg

all\_locs = np.argwhere(all\_data[-1] == c)

if len(all\_locs) == 0:

class\_locs[c] = []

continue

target\_num\_samples = min(num\_samples, len(all\_locs))

# 采样个数至少要包含1%的前景量

target\_num\_samples = max(target\_num\_samples, int(np.ceil(len(all\_locs)

\* min\_percent\_coverage)))

selected = all\_locs[rndst.choice(len(all\_locs), target\_num\_samples, replace=False)]

class\_locs[c] = selected

**1.3~1.7 总结**

根据 cropped 文件夹内的数据生成 nnUNet\_preprocessed 文件夹：

nnUNet\_preprocessed/Task101\_Meningoma/

—— gt\_segmentations # 从 cropped 复制

—— nnUNetData\_plans\_v2.1\_2D\_stage0 # 1.7 生成

—— nnUNetData\_plans\_v2.1\_stage0 # 1.7 生成

—— dataset.json # 从 cropped 复制

—— dataset\_properties.pkl # 从 cropped 复制

—— nnUNetPlansv2.1\_plans\_2D.pkl # 1.6 生成，1.6 只写了 3D 版，但 2D 大概也差不多

—— nnUNetPlansv2.1\_plans\_3D.pkl # 1.6 生成

同时对原 case 存储的属性进行补充：

#============= 原始图像信息 =============

properties["original\_size\_of\_raw\_data"] = np.array(data\_itk[0].GetSize())[[2, 1, 0]]

properties["original\_spacing"] = np.array(data\_itk[0].GetSpacing())[[2, 1, 0]]

properties["list\_of\_data\_files"] = data\_files # 多个模态，路径

properties["seg\_file"] = seg\_file

properties["itk\_origin"] = data\_itk[0].GetOrigin()

properties["itk\_spacing"] = data\_itk[0].GetSpacing()

properties["itk\_direction"] = data\_itk[0].GetDirection()

#=========== crop 部分加信息 ===========

properties["crop\_bbox"] = bbox

properties['classes'] = np.unique(seg) # [-1,0,1...]

properties["size\_after\_cropping"] = data[0].shape

#======== preprocess 部分加信息 ========

properties['use\_nonzero\_mask\_for\_norm'] = use\_nonzero\_mask\_for\_norm

properties["size\_after\_resampling"] = data[0].shape # 重采样之后的 size

properties["spacing\_after\_resampling"] = target\_spacing # 重采样之后的 spacing

properties['class\_locations'] = class\_locs # 采样一些前景的位置

**2. 训练**

训练的命令为 nnUNet\_train（单卡）/nnUNet\_train\_DP（多卡），相应代码在 ./nnunet/run/run\_training/run\_training.py 和 ./nnunet/run/run\_training/run\_training\_DP.py 文件中，本节以单卡为例进行梳理，使用的 trainer 为 nnUNetTrainerV2，代码见 ./nnunet/training/network\_training 文件夹。 继承路线为 nnUNetTrainerV2 -> nnUNetTrainer -> NetworkTrainer

注：一些细节设置

* deterministic：会使网络具有确定性，但是作者认为这不是必要的，因为这可能导致网络过拟合某些随机种子

if deterministic:

np.random.seed(12345)

torch.manual\_seed(12345)

if torch.cuda.is\_available():

torch.cuda.manual\_seed\_all(12345)

cudnn.deterministic = True

torch.backends.cudnn.benchmark = False

* loss：
* 2D 和 3D cascade 网络：**batch** dice + CE loss
* 3D fullres 网络：**sample** dice + CE loss

**2.1 数据加载**

**2.1.1 构造数据集**

nnunet 使用 5 折交叉切分数据，使用 sklearn 的 KFold 函数，对 case 名称进行切分，在 preprocessed 文件夹下生成 splits\_final.pkl。也可以使用别的折去训练，但是需要自己做一个 splits\_final.pkl 文件。**训练**的时候，根据 **FOLD** 参数 load 某一折的数据用于训练和测试，见训练使用教程第 3 点。自己做的训练集要注意 FOLD 对应。

**2.1.2 DataLoader**

作者重新了 DataLoader 函数，见文件 ./nnunet/training/dataloading/dataset\_loading.py 的 DataLoader3D 类（-> SlimDataLoaderBase，该类在 batchgenerators 包内，需要 pip install ）。在进行这一步之前，作者设置了数据扩充的参数，并且根据**旋转**和**缩放**的参数得到了一个新的 patch size，跟网络所需输入 patch size（1.5）有些差别，称为 **basic\_generator\_patch\_size**（这个变量得和 2.1.3 连起来看）

# 只绕 x 轴旋转 [-15,15]

rot\_x = (-15. / 360 \* 2. \* np.pi, 15. / 360 \* 2. \* np.pi)

rot\_y = (-0. / 360 \* 2. \* np.pi, 0. / 360 \* 2. \* np.pi)

rot\_z = (-0. / 360 \* 2. \* np.pi, 0. / 360 \* 2. \* np.pi)

# 最大不超过 90 度

rot\_x = min(90 / 360 \* 2. \* np.pi, rot\_x)

rot\_y = min(90 / 360 \* 2. \* np.pi, rot\_y)

rot\_z = min(90 / 360 \* 2. \* np.pi, rot\_z)

from batchgenerators.augmentations.utils import rotate\_coords\_3d, rotate\_coords\_2d

# 1.5 的 patch size

coords = np.array(final\_patch\_size)

final\_shape = np.copy(coords)

# 计算旋转矩阵，有点像旋转坐标系？

if len(coords) == 3:

final\_shape = np.max(np.vstack((np.abs(rotate\_coords\_3d(coords, rot\_x, 0, 0)), final\_shape)), 0)

final\_shape = np.max(np.vstack((np.abs(rotate\_coords\_3d(coords, 0, rot\_y, 0)), final\_shape)), 0)

final\_shape = np.max(np.vstack((np.abs(rotate\_coords\_3d(coords, 0, 0, rot\_z)), final\_shape)), 0)

elif len(coords) == 2:

final\_shape = np.max(np.vstack((np.abs(rotate\_coords\_2d(coords, rot\_x)), final\_shape)), 0)

# 这个 final\_shape 和原始的 patch size 是不一样的

final\_shape /= min(scale\_range)

* 由于 case 的原始大小（1.7.1）和 dataloader 需要的 basic\_generator\_patch\_size 不一样，在输入网络的时候需要对 case 进行 **crop**。crop 时采用**过采样前景**策略，即强制采样的一个 batch 里有 **1/3** 的数据含有前景，这样会使训练更加稳定
* 如果使**多阶段**的，把上一个阶段的 **seg** 结果加载进来在这一阶段进行优化，同样需要 crop
* 最后输出的还是一个大小为 basic\_generator\_patch\_size 的 shape

# 如果当前样本需要过采样前景

if force\_fg:

# 提取该 case 的所有前景位置（1.7.3）

foreground\_classes = np.array([i for i in properties['class\_locations'].keys()

if len(properties['class\_locations'][i]) != 0])

foreground\_classes = foreground\_classes[foreground\_classes > 0]

# 如果该 case 不存在任何前景

if len(foreground\_classes) == 0:

selected\_class = None

voxels\_of\_that\_class = None

# 如果该 case 存在某些前景

else:

# 随机选择一个前景类

selected\_class = np.random.choice(foreground\_classes)

# 找到该类的前景坐标

voxels\_of\_that\_class = properties['class\_locations'][selected\_class]

if voxels\_of\_that\_class is not None:

# 如果含有前景，随机选择一个 voxel，以这个选择的 voxel 为中心，在合理的边界内进行裁剪

selected\_voxel = voxels\_of\_that\_class[np.random.choice(len(voxels\_of\_that\_class))]

bbox\_x\_lb = max(lb\_x, selected\_voxel[0] - self.patch\_size[0] // 2)

bbox\_y\_lb = max(lb\_y, selected\_voxel[1] - self.patch\_size[1] // 2)

bbox\_z\_lb = max(lb\_z, selected\_voxel[2] - self.patch\_size[2] // 2)

else:

# 不含前景就随机裁

bbox\_x\_lb = np.random.randint(lb\_x, ub\_x + 1)

bbox\_y\_lb = np.random.randint(lb\_y, ub\_y + 1)

bbox\_z\_lb = np.random.randint(lb\_z, ub\_z + 1)

**2.1.3 数据扩充（待补充）**

这里作者设置了太多的方法了，我只列出 meningoma 实验中用到的部分

* **DataChannelSelectionTransform**

选择数据的某些 channels，默认为 None，不做此操作

* **SegChannelSelectionTransform**

选择标签的某些 channels，默认为 [0]，只有一种任务的标签（这种标签是可以多目标的，即取值 [-1,0,1,2,...]，不太清楚这个选择还有什么样的情况）

* **Convert3DTo2DTransform**

如果需要对 3D 数据做 2D 数据扩充，需要先把 3D 转 2D，对于 [B, N, D, H, W] 的数据，把 N 和 D 维度暂时先做成一个维度 N\*D，[B, N\*D, H, W]

data\_dict['data'] = data\_dict['data'].reshape((shp[0], shp[1] \* shp[2], shp[3], shp[4]))

data\_dict['seg'] = data\_dict['seg'].reshape((shp[0], shp[1] \* shp[2], shp[3], shp[4]))

* **SpatialTransform**

“几乎包含了你能想象到的所有空间数据扩充方式”，是在原始 patch size（1.5）上做的

* **Convert2DTo3DTransform**

如果对 3D 数据做了 2D 扩充，扩充完之后需要再转回去

data\_dict['data'] = data\_dict['data'].reshape((shp[0], shp[1], shp[2],

current\_shape[-2],

current\_shape[-1]))

data\_dict['seg'] = data\_dict['seg'].reshape((shp[0], shp[1], shp[2],

current\_shape[-2],

current\_shape[-1]))

* **GaussianNoiseTransform**
* **GaussianBlurTransform**
* **BrightnessMultiplicativeTransform**
* **ContrastAugmentationTransform**
* **SimulateLowResolutionTransform**
* **GammaTransform**
* **MirrorTransform**
* **MaskTransform**
* **RemoveLabelTransform**
* **RenameTransform**

在 Dataloader 的时候返回的是字典，如 Convert3DTo2DTransform 代码中的 data\_dict['data'] 和 data\_dict['seg']，这个 transform 就是改变某个 value  的 key 值，实际在做的时候是把 'seg' 这个 key 换成了 'target'（所以干嘛不在 Dataloader 的时候就用 'target' 呢？还要多此一举？）

* **DownsampleSegForDSTransform2**
* **NumpyToTensor**

以上为训练集使用的所有方法，验证集只使用了 **RemoveLabelTransform，DataChannelSelectionTransform，SegChannelSelectionTransform，RenameTransform，DownsampleSegForDSTransform2，NumpyToTensor**

**2.2 网络结构**

默认使用 Generic\_UNet 结构，在 ./nnunet/network\_architecture/generic\_UNet.py 文件中

* Basic block：**Conv2/3d + Dropout**（可选） **+ InstanceNorm2/3d + LeakyReLU**，卷积的参数（pad 和 stride）是根据 1.5 的 '**conv\_kernel\_sizes**' 和 '**pool\_op\_kernel\_sizes**' 计算得到的，Dropout 在 Encoder 中用到了，在 Decoder 中未用到
* Encoder 阶段一个 conv\_blocks\_context（level） 由几个 Basic block 组成是通过 '**conv\_per\_stage**'（1.6）得到的
* Decoder 阶段一个 conv\_blocks\_localization（level）的卷积组成同上，每个 level 后面会接一个额外的普通卷积（3\*3\*3）做分割，用于深度监督，深度监督算 loss 的时候需要 upsample 到原图大小

self.conv\_pad\_sizes = []

for krnl in self.conv\_kernel\_sizes:

# kernel\_size=3 时 pad=1，否则 pad=0

self.conv\_pad\_sizes.append([1 if i == 3 else 0 for i in krnl])

# 这个 first\_stride 跟下采样卷积有关

# 如果不做下采样卷积，则 first\_stride 为 None（conv 默认 stride=1 ）且之后要单独做池化

# 如果做下采样卷积，第一个 level 的 conv stride 为 1（第一层不需要下采样），其余按下面方式计算

# 之后做上采样使用的 stride 也是这个

for d in range(num\_pool):

if d != 0 and self.convolutional\_pooling:

first\_stride = pool\_op\_kernel\_sizes[d - 1]

else:

first\_stride = None

# 深度监督

self.upscale\_logits\_ops = []

# 每个level的下采样倍数

cum\_upsample = np.cumprod(np.vstack(pool\_op\_kernel\_sizes), axis=0)[::-1]

for usl in range(num\_pool - 1):

if self.upscale\_logits:

self.upscale\_logits\_ops.append(

Upsample(scale\_factor=tuple([int(i) for i in cum\_upsample[usl + 1]]),

mode=upsample\_mode))

else:

self.upscale\_logits\_ops.append(lambda x: x)

**2.3 训练参数**

* **SGD**（lr=1e-2，momentum=0.99，nesterov=True，weight\_decay=3e-05）
* momentum 默认为 0.99，但是很偶尔会发生在某些数据集上，在一定量 epoch（如 100/1000）之后验证集的 dice 仍然为 0 的情况，此时将 momentum 的值降为 **0.95**
* 每次 epoch 使用 **poly** 学习率更新策略
* 设置了 **patience** 参数控制训练是否需要**提前结束训练**，一般发生在训练 moving average 曲线太过**震荡**且长时间 **loss 未改善**，否则就正常训练至 max\_epoch 结束。通过两个变量 '**val\_eval\_criterion\_MA**' 和 '**train\_loss\_MA**' 来控制，这两个变量在更新的时候都是以 alpha \* old + (1 - alpha) \* new 的方式进行：

""" 更新 train\_loss\_MA """

# 如果设置了自带的 lr\_schedule，train\_loss\_MA 也可以作为参数传入 lr\_scheduler.step()

# 但 nnUNetTrainerV2 使用的学习率更新策略是 poly，不涉及 step 的问题，故此变量对 lr 无实际影响

if self.train\_loss\_MA is None:

# 初始化

self.train\_loss\_MA = self.all\_tr\_losses[-1]

else:

# 更新

self.train\_loss\_MA = self.train\_loss\_MA\_alpha \* self.train\_loss\_MA \

+ (1 - self.train\_loss\_MA\_alpha) \* self.all\_tr\_losses[-1]

""" 更新 val\_eval\_criterion\_MA """

if self.val\_eval\_criterion\_MA is None:

# 初始化

if len(self.all\_val\_eval\_metrics) == 0:

# 如果未计算验证 metrics，则转而使用验证 loss 作为标准

# 且注意 metrics 我们希望越大越好，而 loss 刚好相反，所以要对 loss 取负

self.val\_eval\_criterion\_MA = - self.all\_val\_losses[-1]

else:

self.val\_eval\_criterion\_MA = self.all\_val\_eval\_metrics[-1]

else:

# 更新

if len(self.all\_val\_eval\_metrics) == 0:

# alpha \* old - (1 - alpha) \* new

# 跟初始化的时候相同，这里也要对 new 的部分取负

self.val\_eval\_criterion\_MA = self.val\_eval\_criterion\_alpha \

\* self.val\_eval\_criterion\_MA \

- (1 - self.val\_eval\_criterion\_alpha) \

\* self.all\_val\_losses[-1]

else:

self.val\_eval\_criterion\_MA = self.val\_eval\_criterion\_alpha \

\* self.val\_eval\_criterion\_MA \

+ (1 - self.val\_eval\_criterion\_alpha) \* \

self.all\_val\_eval\_metrics[-1]

""" 提前停止训练 """

# 初始化

self.best\_MA\_tr\_loss\_for\_patience = self.train\_loss\_MA

self.best\_epoch\_based\_on\_MA\_tr\_loss = self.epoch

self.best\_val\_eval\_criterion\_MA = self.val\_eval\_criterion\_MA

# 当前 epoch 和最好的 epoch 之间相差太多（很多个 epoch 内 loss 都没有改善），可能需要提前停止

if self.epoch - self.best\_epoch\_based\_on\_MA\_tr\_loss > self.patience:

# 虽然已经到了“忍耐的极限”，但是学习率仍然高于 1e-6，继续训练

if self.optimizer.param\_groups[0]['lr'] > self.lr\_threshold: # 1e-6

self.best\_epoch\_based\_on\_MA\_tr\_loss = self.epoch - self.patience // 2

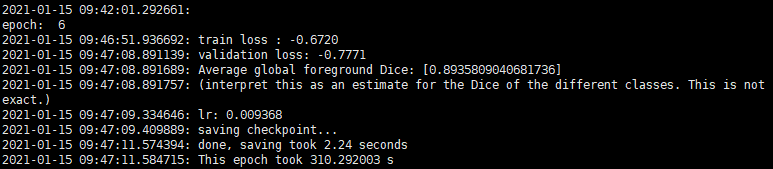
else:

# “忍到极限”且学习率过低，早停

continue\_training = False

**2.4 在线评估**

训练集每完成一个 epoch，对训练集和验证集分别计算 loss，评估前景 tp，fp，fn，dice，对训练集+验证集（如果我没有看错的话）做平均 dice 实时监控，print 出来的 Average global foreground Dice 并不是验证集真正的 dice，仅作参考，验证集在验证阶段会单独做，会有一些特殊的处理方式



一个 epoch 之后会保存最新的模型和一些相关的信息

# 模型

save\_this = {'epoch': self.epoch + 1,

'state\_dict': state\_dict,

'optimizer\_state\_dict': optimizer\_state\_dict,

'lr\_scheduler\_state\_dict': lr\_sched\_state\_dct,

'plot\_stuff': (self.all\_tr\_losses, self.all\_val\_losses,

self.all\_val\_losses\_tr\_mode, self.all\_val\_eval\_metrics)}

# 模型相关信息

info['init'] = self.init\_args # 一些参数 (plans\_file, fold, output\_folder,

# dataset\_directory, batch\_dice, stage, unpack\_data,

# deterministic, fp16)

info['name'] = self.\_\_class\_\_.\_\_name\_\_ # 使用训练器的名称，如 nnUNetTrainerV2

info['class'] = str(self.\_\_class\_\_) # 训练器所在类的文件路径

info['plans'] = self.plans # plan 的细节, 1.6

**2.5 Loss**

所有 loss 函数均在 ./nnunet/training/loss\_functions 文件夹下，分三个部分：

* **交叉熵 loss**：这个没什么好说的
* **dice loss**：直接对 dice 取负，所以这个网络在训练的时候是会出现 loss < 0 的情况的
* **深度监督**：对 unet 每个 level（池化）的输出都计算 loss，加入最终的 loss 函数，不同分辨率阶段的 loss 使用不同的权重，分辨率越小权重越低：

net\_numpool = len(self.net\_num\_pool\_op\_kernel\_sizes)

weights = np.array([1 / (2 \*\* i) for i in range(net\_numpool)])

# 忽略最后一次池化的输出（分辨率过于小了）

mask = np.array([True] + [True if i < net\_numpool - 1 else False for i in range(1, net\_numpool)])

weights[~mask] = 0

weights = weights / weights.sum()

**2.6 交叉验证**

这种方式适合数据集完全带标签，直接全部放到 imagesTr 中走**交叉验证**的流程。一般 in-house 数据集这么做就够了。这个过程是在 4 折数据集**训练结束后**顺带完成的，会在保存 model 的文件夹内把验证集的预测结果存好，metrics 也顺带全部计算好存成 json，统计的时候直接读取即可

* 首先处理 seg，MR seg 中值为 -1 的地方全部改为 0
* 在训练的时候，是把样本 crop 出 patch size，但在测试的时候肯定不能 crop，最好使用 **sliding window**。使用 sliding window 的话输入图像至少要和 patch size 一样大，必要时需要 pad。sliding window 的 step 取值范围 (0,1] 的浮点数，步长的最大限制为 step\_width = patch\_size \* step，可以更小。则 step 数量的计算公式为 num\_steps = ceil((image\_size - patch\_size) / step\_width))

""" 计算滑动窗口的 step """

target\_step\_sizes\_in\_voxels = [i \* step\_size for i in patch\_size]

# 只有当 image\_size = patch\_size 时，num\_steps = 1，否则 > 1

num\_steps = [int(np.ceil((i - k) / j)) + 1

for i, j, k in zip(image\_size, target\_step\_sizes\_in\_voxels, patch\_size)]

# 记录每一步的起始位置

steps = []

for dim in range(len(patch\_size)):

# the highest step value for this dimension is

# 最后一步到达的位置（最大步）

max\_step\_value = image\_size[dim] - patch\_size[dim]

if num\_steps[dim] > 1:

# 实际能走多少步

actual\_step\_size = max\_step\_value / (num\_steps[dim] - 1)

else:

# 不影响，当 num\_step = 1 时，下面的 i 只能等于 0

actual\_step\_size = 99999999999

steps\_here = [int(np.round(actual\_step\_size \* i)) for i in range(num\_steps[dim])]

steps.append(steps\_here)

""" 把对每个窗口分别做预测 """

aggregated\_results = np.zeros([self.num\_classes] + list(data.shape[1:]),

dtype=np.float32)

aggregated\_nb\_of\_predictions = np.zeros([self.num\_classes] + list(data.shape[1:]),

dtype=np.float32)

for x in steps[0]:

# 根据 steps 把框框出来

lb\_x = x

ub\_x = x + patch\_size[0]

for y in steps[1]:

lb\_y = y

ub\_y = y + patch\_size[1]

for z in steps[2]:

lb\_z = z

ub\_z = z + patch\_size[2]

# data 第一个轴 None 是补了一个 batch size 的轴

# predict() 是我简化的一个函数，实际上对应下下一点 mirror TTA

predicted\_patch = predict(data[None, :, lb\_x:ub\_x, lb\_y:ub\_y, lb\_z:ub\_z])[0]

# 把裁出来的预测结果加到原图对应的位置

aggregated\_results[:, lb\_x:ub\_x, lb\_y:ub\_y, lb\_z:ub\_z] += predicted\_patch

# 如果用到了 gaussion（下一点中会讲），就加上 gaussion\_importance\_map

# 否则加 torch.ones()

aggregated\_nb\_of\_predictions[:, lb\_x:ub\_x, lb\_y:ub\_y, lb\_z:ub\_z] \

+= add\_for\_nb\_of\_preds

* 算 **gaussian importance map**（待补充：为什么要算这个？）

from scipy.ndimage.filters import gaussian\_filter

def \_get\_gaussian(patch\_size, sigma\_scale=1. / 8) -> np.ndarray:

tmp = np.zeros(patch\_size)

center\_coords = [i // 2 for i in patch\_size]

sigmas = [i \* sigma\_scale for i in patch\_size]

tmp[tuple(center\_coords)] = 1

gaussian\_importance\_map = gaussian\_filter(tmp, sigmas, 0, mode='constant', cval=0)

gaussian\_importance\_map = gaussian\_importance\_map / np.max(gaussian\_importance\_map) \* 1

gaussian\_importance\_map = gaussian\_importance\_map.astype(np.float32)

# gaussian\_importance\_map 不能为 0，否则会 nans

gaussian\_importance\_map[gaussian\_importance\_map == 0] = np.min(

gaussian\_importance\_map[gaussian\_importance\_map != 0])

return gaussian\_importance\_map

# 这个东西在上一点和下一点中都会用到

gaussian\_importance\_map = self.\_get\_gaussian(patch\_size, sigma\_scale=1. / 8)

* **mirror** 的 TTA（Test Time Augmentation）：测试时先将样本做 flip 的扩充，将不同的扩充的平均预测结果作为此样本的最终预测结果

"""

假设输入 data 的 shape 为 [1,n,x,y,z]，对应 [batch\_size, modalities, ...]，

mirror\_axes 为 (x,y,z)，对 volume 的三个维度都进行镜面翻转，

则一共需要 8 次 inference，即 (x,y,z) 的子集个数（要算上不翻转的原图本身），

作者采用了类似模型融合的方式，对 8 个结果的输出做平均作为最终的预测结果

"""

if do\_mirroring:

mirror\_idx = 8

num\_results = 2 \*\* len(mirror\_axes)

for m in range(mirror\_idx):

# [1,n,x,y,z]

# mirror\_axes=(0,1,2) 对应 (x,y,z)

if m == 0:

# 这个 self(x) 对应 model(x)，也就是 inference 的过程

# 不翻转

pred = self.inference\_apply\_nonlin(self(x))

result\_torch += 1 / num\_results \* pred

if m == 1 and (2 in mirror\_axes):

# 以 2 轴翻转，2 轴对应dim=4

pred = self.inference\_apply\_nonlin(self(torch.flip(x, (4, ))))

# 翻转完之后，相加的时候需要再翻回去

result\_torch += 1 / num\_results \* torch.flip(pred, (4,))

if m == 2 and (1 in mirror\_axes):

# 以1轴翻转，对应dim=3

pred = self.inference\_apply\_nonlin(self(torch.flip(x, (3, ))))

result\_torch += 1 / num\_results \* torch.flip(pred, (3,))

if m == 3 and (2 in mirror\_axes) and (1 in mirror\_axes):

pred = self.inference\_apply\_nonlin(self(torch.flip(x, (4, 3))))

result\_torch += 1 / num\_results \* torch.flip(pred, (4, 3))

if m == 4 and (0 in mirror\_axes):

pred = self.inference\_apply\_nonlin(self(torch.flip(x, (2, ))))

result\_torch += 1 / num\_results \* torch.flip(pred, (2,))

if m == 5 and (0 in mirror\_axes) and (2 in mirror\_axes):

pred = self.inference\_apply\_nonlin(self(torch.flip(x, (4, 2))))

result\_torch += 1 / num\_results \* torch.flip(pred, (4, 2))

if m == 6 and (0 in mirror\_axes) and (1 in mirror\_axes):

pred = self.inference\_apply\_nonlin(self(torch.flip(x, (3, 2))))

result\_torch += 1 / num\_results \* torch.flip(pred, (3, 2))

if m == 7 and (0 in mirror\_axes) and (1 in mirror\_axes) and (2 in mirror\_axes):

pred = self.inference\_apply\_nonlin(self(torch.flip(x, (4, 3, 2))))

result\_torch += 1 / num\_results \* torch.flip(pred, (4, 3, 2))

result\_torch[:, :] \*= gaussian\_importance\_map

* 计算预测结果的**类别概率**，作者用的方法很神奇（没看懂）。注意在 sliding window 那一步对数据进行了 pad，因此要对预测结果首先进行**反 pad**

# 这是什么操作？等同于 softmax 的作用？

class\_probabilities = aggregated\_results / aggregated\_nb\_of\_predictions

# argmax 得到 pred 常规操作

predicted\_segmentation = class\_probabilities.argmax(0)

* 上一步得到的预测结果是重采样之后的结果，要得到最终结果还需要把样本的 spacing 和 size **反重采样**回去（original\_spacing + cropped\_size），然后把 cropped 数据贴回原图，再根据 data\_properties.pkl 中事先保存的三维数据存成 nii 文件

# 回忆 new\_shape = original\_spacing / target\_spacing \* original\_shape

# segmentation\_softmax 是 new\_shape 预测概率图

# shape\_original\_after\_cropping 是 original\_spacing，original\_shape 的原始 cropped 图像

# 现在就是要把 new\_shape 的概率图 resize 回 original\_shape，得到 original\_spacing 下的概率图

seg\_old\_spacing = resample\_data\_or\_seg(segmentation\_softmax, shape\_original\_after\_cropping,

is\_seg=False, axis=lowres\_axis, order=order,

do\_separate\_z=do\_separate\_z, cval=0,

order\_z=interpolation\_order\_z)

# 根据 bbox 的框，把 cropped 数据贴回原图即 raw\_data

bbox = properties\_dict.get('crop\_bbox')

if bbox is not None:

# raw\_shape

seg\_old\_size = np.zeros(shape\_original\_before\_cropping)

for c in range(3):

bbox[c][1] = np.min((bbox[c][0] + seg\_old\_spacing.shape[c],

shape\_original\_before\_cropping[c]))

seg\_old\_size[bbox[0][0]:bbox[0][1],

bbox[1][0]:bbox[1][1],

bbox[2][0]:bbox[2][1]] = seg\_old\_spacing

else:

# 没 crop 就不需要

seg\_old\_size = seg\_old\_spacing

# 存 nii

seg\_resized\_itk = sitk.GetImageFromArray(seg\_old\_size\_postprocessed.astype(np.uint8))

seg\_resized\_itk.SetSpacing(properties\_dict['itk\_spacing'])

seg\_resized\_itk.SetOrigin(properties\_dict['itk\_origin'])

seg\_resized\_itk.SetDirection(properties\_dict['itk\_direction'])

sitk.WriteImage(seg\_resized\_itk, out\_fname)

* **评估结果**，在文件夹内保存 summary.json，源文件为 ./nnunet/evaluation/evaluator.py

**2.7 后处理**

在 2.6 初步验证结束后，作者对预测结果进行了后处理又生成了一个文件夹叫 validation\_raw\_postprocessed（对比 2.6 生成的是 validation\_raw），相应代码在 ./nnunet/postprocessing/connected\_components.py 中

* 对每个类，除了**最大连通区域**，其他的区域全部消除，连通区域的判定是通过 scipy.ndimage 包中的 label 函数

"""

使用范例

>>> from scipy.ndimage import label, generate\_binary\_structure

>>> a = np.array([[0,0,1,1,0,0],

[0,0,0,1,0,0],

[1,1,0,0,1,0],

[0,0,0,1,0,0]])

>>> labeled\_array, num\_features = label(a)

>>> num\_features

4

>>> labeled\_array

array([[0, 0, 1, 1, 0, 0],

[0, 0, 0, 1, 0, 0],

[2, 2, 0, 0, 3, 0],

[0, 0, 0, 4, 0, 0]])

"""

lmap, num\_objects = label(mask.astype(int))

# 计算每个连通区域的物理大小

object\_sizes = {}

for object\_id in range(1, num\_objects + 1):

object\_sizes[object\_id] = (lmap == object\_id).sum() \* volume\_per\_voxel

# 保存最大被消除的区域

largest\_removed[c] = None

kept\_size[c] = None

if num\_objects > 0:

# 一般情况下最大的区域会被保留

maximum\_size = max(object\_sizes.values())

kept\_size[c] = maximum\_size

for object\_id in range(1, num\_objects + 1):

# 对于其他的小区域，消除

if object\_sizes[object\_id] != maximum\_size:

remove = True

# 对于小区域，只删除比阈值小的那些

if minimum\_valid\_object\_size is not None:

remove = object\_sizes[object\_id] < minimum\_valid\_object\_size[c]

if remove:

image[(lmap == object\_id) & mask] = 0

if largest\_removed[c] is None:

largest\_removed[c] = object\_sizes[object\_id]

else:

largest\_removed[c] = max(largest\_removed[c], object\_sizes[object\_id])

* **评估**后处理之后的结果，如果相对 validation\_raw 负优化了，则不保存上述结果

**2.8 总结**

整个训练过程完全跑完之后会在 nnUNet\_trained\_models 底下生成以下文件

nnUNet\_trained\_models/nnUNet/3d\_fullres/Task102\_T1CMeningoma \

/nnUNetTrainerV2\_\_nnUNetPlansv2.1

—— gt\_niftis # 直接从 preprocessed 里面复制过来，1.2

—— plans.pkl # 直接从 preprocessed 里面复制过来，1.6

—— fold\_4 # 使用第 4 折数据进行训练，fold\_ 后面的数字是执行训练时的命令，训练教程 4.

———— validation\_raw # 做普通测试（无后处理）

—————— summary.json # 每个样本的预测 metrics 统计值

—————— validation\_args.json # 测试时用到的一些参数

—————— meningoma\_006.nii.gz # 预测结果，已 resample 至原图 spacing

—————— ...

———— validation\_raw\_postprocessed # 后处理基础上做测试

—————— summary.json # 后处理基础上每个样本的预测 metrics 统计值

—————— meningoma\_006.nii.gz # 预测结果，已 resample 至原图 spacing

—————— ...

———— debug.json # 训练中使用的一些参数，有很多

———— model\_best.model # 验证 metrics 最好的模型

———— model\_best.model.pkl # 模型相关信息，如训练时的参数等等，2.4

———— model\_final\_checkpoint.model # 训练完成后产生的最后一个模型

———— model\_final\_checkpoint.model.pkl

———— (model\_latest.model) # 训练过程中自动保存最新 epoch 的模型，训练结束后删除

———— (model\_latest.model.pkl)

———— postprocessing.json # 后处理使用的一些参数

———— progress.png # 训练集、验证集的 loss 曲线，和实时监控的平均 dice 曲线

———— train\_log\_2021\_1\_14\_12\_02\_24.txt # 记录训练时后台打印的信息

**3 无标签推理（待补充）**

这种方式适合打榜的公开数据集，即测试集数据已提供，但不含标签，需要离线测试好上传至网站进行评分，以后有时间再补上