**智能信息系统综合实践**

**实验报告**

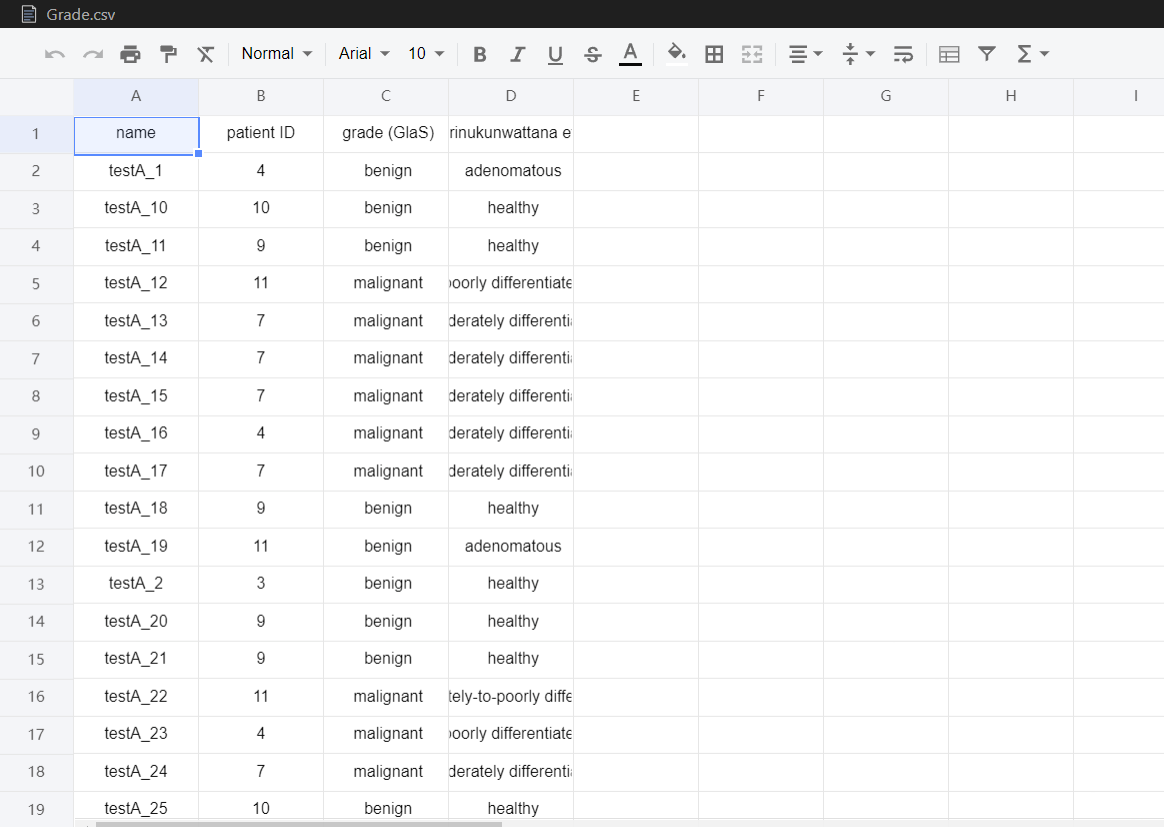
|  |  |
| --- | --- |
| **题 目：** | 目标分割 |
| **年 级：** | **2020级** |
| **专 业：** | **软件工程** |
| **姓 名：** | **庞晓宇** |

1. **题目（原题目）**

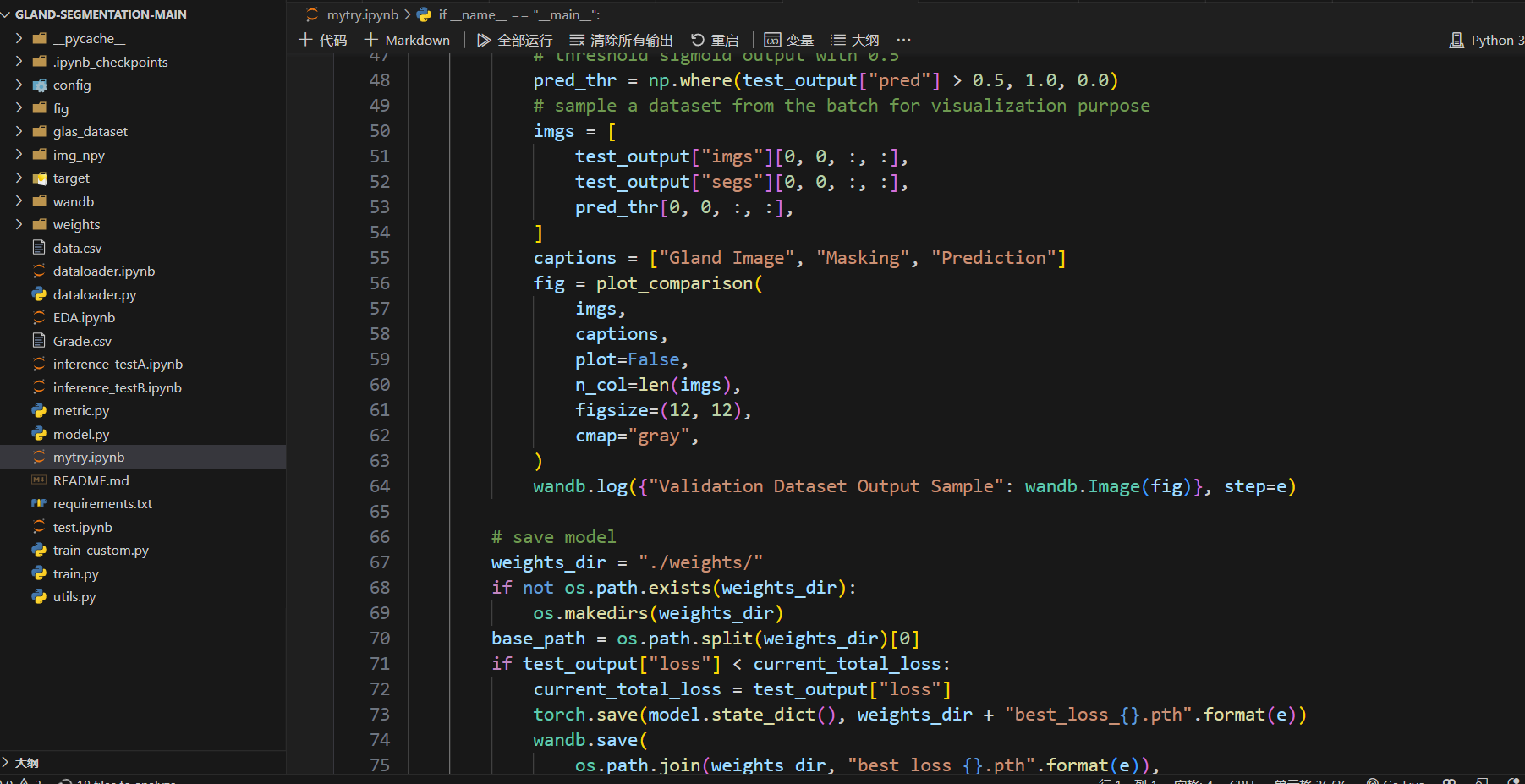
****

1. **解题步骤（思路+代码）**

观察数据集的分布特征：



项目架构及编码：



主要代码思路流程、步骤：

导入了一些常用的数据处理库和函数，为后续的数据加载和数据增强做准备。

import numpy as np # 用于科学计算  
from batchgenerators.dataloading.data\_loader import DataLoader # 用于加载训练数据和标签数据  
from batchgenerators.transforms.abstract\_transforms import Compose # 用于将多个数据变换组合在一起  
from batchgenerators.transforms.spatial\_transforms import (  
 MirrorTransform, # 镜像变换（左右镜像、上下镜像）  
 SpatialTransform, # 空间变换（旋转、缩放、平移）  
)  
from batchgenerators.transforms.color\_transforms import (  
 BrightnessMultiplicativeTransform, # 亮度变换（乘性）  
 ContrastAugmentationTransform, # 对比度变换（增强）（线性）（非线性）  
)  
from batchgenerators.transforms.noise\_transforms import (  
 GaussianNoiseTransform, # 高斯噪声变换  
 GaussianBlurTransform, # 高斯模糊变换  
)  
from batchgenerators.augmentations.crop\_and\_pad\_augmentations import crop # 裁剪和填充变换

定义函数 get\_split\_fold 接收一个 csv 文件的数据，该数据集已按照交叉验证的折叠进行了拆分，其中训练集为第 0 折，测试集 A 为第 1 折，测试集 B 为第 2 折。函数通过查找 "fold" 列的值来确定每个数据点所属的数据集，然后返回一个包含训练、测试 A 和测试 B 数据集的字典。

def get\_split\_fold(data):  
 """  
 如果数据集已经按照 [0,1,2] 分成了三个数据集  
 其中:  
 - 训练集 => 0  
 - 测试集A => 1  
 - 测试集B => 2  
 @param data: 存储数据集的 CSV 文件  
 @return: 训练集，测试集A，测试集B 的字典  
 """  
 # 折叠数据的返回索引  
 train\_idx = np.where(data["fold"] == 0)[0]  
 testA\_idx = np.where(data["fold"] == 1)[0]  
 testB\_idx = np.where(data["fold"] == 2)[0]  
  
 # 为每个数据集创建字典  
 train\_ds = {  
 "img\_npy": [data["img\_npy"].tolist()[i] for i in train\_idx],  
 "anno\_npy": [data["anno\_npy"].tolist()[i] for i in train\_idx],  
 "patient\_id": [data["patient ID"].tolist()[i] for i in train\_idx],  
 }  
 testA\_ds = {  
 "img\_npy": [data["img\_npy"].tolist()[i] for i in testA\_idx],  
 "anno\_npy": [data["anno\_npy"].tolist()[i] for i in testA\_idx],  
 "patient\_id": [data["patient ID"].tolist()[i] for i in testA\_idx],  
 }  
 testB\_ds = {  
 "img\_npy": [data["img\_npy"].tolist()[i] for i in testB\_idx],  
 "anno\_npy": [data["anno\_npy"].tolist()[i] for i in testB\_idx],  
 "patient\_id": [data["patient ID"].tolist()[i] for i in testB\_idx],  
 }  
  
 return {"train\_ds": train\_ds, "testA\_ds": testA\_ds, "testB\_ds": testB\_ds}

定义函数 get\_train\_transform(patch\_size, prob) ，参数1为输入图像的大小，输入2为变换的概率。该函数返回一个数据增强的变换列表，这些变换用于增强神经网络模型训练数据。这里采用的数据增强方法包括：弹性变形、镜像变换、亮度调整、高斯噪声、高斯模糊和对比度增强。其中弹性变形可以减少数据集的大小，并且不会引入边界伪影。

def get\_train\_transform(patch\_size, prob=0.5):  
 tr\_transforms = [] # 创建一个空列表，用于存储变换  
 # 使用SpatialTransform进行空间变换  
 tr\_transforms.append(  
 SpatialTransform(  
 patch\_size, # 输入图像的大小  
 [i // 2 for i in patch\_size], # 中心点  
 do\_elastic\_deform=True, # 弹性变形  
 alpha=(0.0, 300.0), # 弹性变形的强度  
 sigma=(20.0, 40.0), # 弹性变形的平滑度  
 do\_rotation=True, # 旋转  
 angle\_x=(-np.pi / 15.0, np.pi / 15.0), # 旋转角度  
 angle\_y=(-np.pi / 15.0, np.pi / 15.0), # 旋转角度  
 angle\_z=(0.0, 0.0), # 旋转角度  
 do\_scale=True, # 缩放  
 scale=(1 / 1.15, 1.15), # 缩放比例  
 random\_crop=False, # 随机裁剪  
 border\_mode\_data="constant", # 边界模式：常数  
 border\_cval\_data=0, # 边界值  
 order\_data=3, # 数据的阶数  
 p\_el\_per\_sample=prob, # 弹性变形的概率  
 p\_rot\_per\_sample=prob, # 旋转的概率  
 p\_scale\_per\_sample=prob, # 缩放的概率  
 )  
 )  
 # 使用MirrorTransform进行镜像变换，这里只进行左右镜像  
 tr\_transforms.append(MirrorTransform(axes=(1,)))  
 # 使用BrightnessMultiplicativeTransform对图像的亮度进行调整  
 tr\_transforms.append(  
 # 下面的三个参数分别是：亮度的乘性因子，是否对每个通道进行变换，每个样本的变换概率  
 BrightnessMultiplicativeTransform(  
 (0.7, 1.5), per\_channel=True, p\_per\_sample=prob  
 )  
 )  
 # 使用GaussianNoiseTransform对图像添加高斯噪声，噪声的方差在 [0, 0.5] 之间  
 tr\_transforms.append(  
 GaussianNoiseTransform(noise\_variance=(0, 0.5), p\_per\_sample=prob)  
 )  
 # 使用GaussianBlurTransform对图像进行高斯模糊，模糊的程度在 [0.5, 2.0] 之间  
 tr\_transforms.append(  
 GaussianBlurTransform(  
 blur\_sigma=(0.5, 2.0),  
 different\_sigma\_per\_channel=True,  
 p\_per\_channel=prob,  
 p\_per\_sample=prob,  
 )  
 )  
 # 使用ContrastAugmentationTransform对图像进行对比度增强，增强的程度在 [0.75, 1.25] 之间  
 tr\_transforms.append(  
 ContrastAugmentationTransform(contrast\_range=(0.75, 1.25), p\_per\_sample=prob)  
 )  
 # 使用Compose将这些变换组合在一起  
 tr\_transforms = Compose(tr\_transforms)  
 return tr\_transforms # 返回变换

继承 batchgenerators.dataloading.data\_loader.DataLoader 定义自己的 DataLoader 类，用于加载训练数据，并生成训练批次。构造函数接收训练数据、批量大小、裁剪大小、线程数量等参数，并可选择是否启用数据洗牌、数据增强等功能。

class DataLoader(DataLoader): # batchgenerators.dataloading.data\_loader.DataLoader  
 def \_\_init\_\_(  
 self,  
 data, # 数据集：必须是由get\_list\_of\_patients返回的患者列表（并由get\_split\_deterministic拆分）  
 batch\_size, # 批次大小  
 patch\_size, # 批具有的空间大小  
 num\_threads\_in\_multithreaded, # 多线程  
 crop\_status=False, # 是否裁剪：默认不裁剪  
 crop\_type="center", # 裁剪类型：中心裁剪  
 seed\_for\_shuffle=1234, # 随机种子  
 return\_incomplete=False, # 默认不返回不完整的批次  
 shuffle=True, # 默认打乱  
 infinite=True, # 默认无限循环  
 margins=(0, 0, 0), # 边距0  
 ):  
 super().\_\_init\_\_(  
 data,  
 batch\_size,  
 num\_threads\_in\_multithreaded,  
 seed\_for\_shuffle,  
 return\_incomplete,  
 shuffle,  
 infinite,  
 )  
 self.patch\_size = patch\_size # 批具有的空间大小  
 self.n\_channel = 3 # 通道数  
 self.indices = list(range(len(data["img\_npy"]))) # 索引  
 self.crop\_status = crop\_status # 是否裁剪  
 self.crop\_type = crop\_type # 裁剪类型  
 self.margins = margins # 边距  
  
 @staticmethod # 静态方法：加载患者  
 def load\_patient(img\_path):  
 img = np.load(img\_path, mmap\_mode="r") # 以只读模式加载图像  
 return img  
  
 def generate\_train\_batch(self):  
 idx = self.get\_indices() # 调用父类的方法获取下一个批次中要使用的病人的索引  
 gland\_img = [self.\_data["img\_npy"][i] for i in idx] # 根据索引获取数据图像  
 img\_seg = [self.\_data["anno\_npy"][i] for i in idx] # 根据索引获取标注图像  
 patient\_id = [self.\_data["patient\_id"][i] for i in idx] # 根据索引获取病人ID  
 # 初始化空数组用于存储数据和标注  
 img = np.zeros(  
 (len(gland\_img), self.n\_channel, \*self.patch\_size), dtype=np.float32  
 )  
 seg = np.zeros(  
 (len(img\_seg), self.n\_channel, \*self.patch\_size), dtype=np.float32  
 )  
 # 迭代patients\_for\_batch并将其包含在批次中  
 for i, (j, k) in enumerate(zip(gland\_img, img\_seg)):  
 img\_data = self.load\_patient(j) # 加载数据图像  
 seg\_data = self.load\_patient(k) # 加载标注图像  
 # 根据文档要求，输入图像应该以通道为首的顺序输入，因此我们使用张量操作来转换为通道为首的格式  
 img\_data = np.einsum("hwc->chw", img\_data)  
 seg\_data = np.einsum("hwc->chw", seg\_data)  
 # 现在随机裁剪到self.patch\_size大小  
 # crop期望数据为(b, c, x, y, z)，但patient\_data的形状为(c, x, y, z)，因此我们需要添加一个虚拟维度，以便它能够工作（@Todo，可以改进）  
 if self.crop\_status:  
 img\_data, seg\_data = crop(  
 img\_data[None],  
 seg=seg\_data[None],  
 crop\_size=self.patch\_size,  
 margins=self.margins,  
 crop\_type=self.crop\_type,  
 )  
 img[i] = img\_data[0]  
 seg[i] = seg\_data[0]  
 else:  
 img[i] = img\_data  
 seg[i] = seg\_data  
 return {"data": img, "seg": seg, "patient\_id": patient\_id}

定义一系列工具函数，用于图像处理和分割。

# 导入matplotlib和skimage模块以支持图像处理和分割。  
import matplotlib.pyplot as plt  
from skimage import color  
from skimage import segmentation

定义函数 plot\_comparison 用于绘制多个图像的比较，但仅在列方向上进行。输入参数包括图像列表、标题列表、行数、列数、绘图标识、保存路径等。如果绘图标识为True，函数将绘制图像；否则，它将返回一个图像对象。

def plot\_comparison(  
 input\_img, # 输入图像  
 caption=None, # 标题  
 plot=True, # 是否绘制  
 save\_path=None, # 保存路径  
 save\_name=None, # 保存名称  
 save\_as="png", # 保存格式  
 save\_dpi=300, # 保存分辨率  
 captions\_font=20, # 标题字体大小  
 n\_row=1, # 行数  
 n\_col=2, # 列数  
 figsize=(5, 5), # 图像大小  
 cmap="gray", # 颜色映射：灰度  
):  
 print()  
 if caption is not None:  
 assert len(caption) == len(  
 input\_img  
 ), "Caption length and input image length does not match"  
 assert len(input\_img) == n\_col, "Error of input images or number of columns!"  
  
 fig, axes = plt.subplots(n\_row, n\_col, figsize=figsize)  
 fig.subplots\_adjust(hspace=0.4, wspace=0.4, right=0.7)  
  
 for i in range(n\_col):  
 axes[i].imshow(np.squeeze(input\_img[i]), cmap=cmap)  
 if caption is not None:  
 axes[i].set\_xlabel(caption[i], fontsize=captions\_font)  
 axes[i].set\_xticks([])  
 axes[i].set\_yticks([])  
  
 plt.tight\_layout()  
 if save\_path is not None:  
 plt.savefig(save\_path + "{}.{}".format(save\_name, save\_as), save\_dpi=save\_dpi)  
 if plot:  
 plt.show()  
 else:  
 return fig

定义函数 plot\_hist 用于绘制两个图像的直方图，这两个图像在同一行中，并且标题显示在图像下方。输入参数包括图像列表、标题列表、行数、列数、bin数、像素值范围等。

def plot\_hist(  
 inp\_img, titles, n\_row=1, n\_col=2, n\_bin=20, ranges=[0, 1], figsize=(5, 5)  
):  
 """输入图像的绘制直方图  
 Args:  
 inp\_img (\_type\_): 图像列表  
 titles (\_type\_): 标题列表  
 n\_row (int, optional): 行数  
 n\_col (int, optional): 列数  
 n\_bin (int, optional): bin数  
 ranges (list, optional): 范围  
 figsize (tuple, optional): 图像大小  
 """  
 assert len(titles) == len(  
 inp\_img  
 ), "Caption length and input image length does not match"  
 assert len(inp\_img) == n\_col, "Error of input images or number of columns!"  
  
 fig, axes = plt.subplots(n\_row, n\_col, figsize=figsize)  
 fig.subplots\_adjust(hspace=0.4, wspace=0.4, right=0.7)  
  
 for i in range(n\_col):  
 inp = np.squeeze(inp\_img[i])  
 axes[i].hist(inp.ravel(), n\_bin, ranges)  
 axes[i].set\_title(titles[i])  
 axes[i].set\_xlabel("Pixel Value")  
 axes[i].set\_ylabel("Frequency")  
  
 plt.tight\_layout()  
 plt.show()

定义函数 overlay\_mask 将分割掩模覆盖在原始图像上。输入参数包括图像、分割掩模、颜色和掩模不透明度。

def overlay\_mask(image, mask, colors=[(0, 1.0, 0)], alpha=0.12):  
 # 图像归一化  
 if np.max(image) > 1.0:  
 image = image / 255.0  
 # 灰度图像  
 if mask.ndim == 3:  
 mask = mask[:, :, 0]  
 mask\_image = color.label2rgb(mask, image, colors=colors, alpha=alpha, bg\_label=0)  
 return mask\_image

overlay\_boundary 函数在原始图像上绘制分割边界线。输入参数包括图像、分割掩模、颜色和边界线模式。

def overlay\_boundary(image, mask, color=(0, 1.0, 0), mode="thick"):  
 if np.max(image) > 1.0:  
 image = image / 255.0  
 if mask.ndim == 3:  
 mask = mask[:, :, 0]  
 boundary\_image = segmentation.mark\_boundaries(image, mask, color=color, mode=mode)  
 return boundary\_image

plot\_labels\_color 通过循环使用matplotlib定义的颜色映射可视化分割标签。输入参数包括标签、颜色映射等。

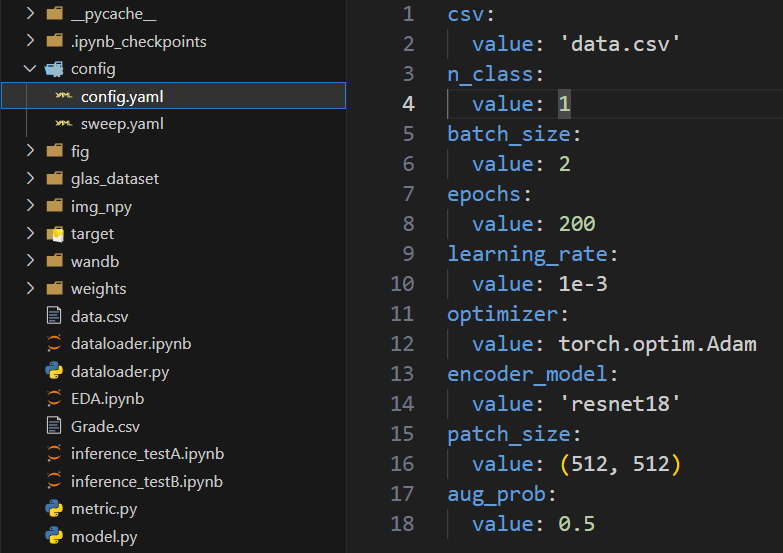
def plot\_labels\_color(label\_im, cmap="tab20c"):  
 # 构造彩色图像以叠加  
 color\_mask = np.zeros(label\_im.shape)  
 get\_cmap = plt.cm.get\_cmap(cmap)  
 # 循环通过cmap为每种颜色是相关联的标签  
 for i in range(np.max(label\_im)):  
 color\_mask[label\_im[:, :, 0] == i + 1] = list(get\_cmap(i))[:-1]  
 return color\_mask

# 用于输入图像的通道方向最小最大归一化  
def min\_max\_norm(img, axis=(1, 2)):  
 inp\_shape = img.shape  
 img\_min = np.broadcast\_to(img.min(axis=axis, keepdims=True), inp\_shape)  
 img\_max = np.broadcast\_to(img.max(axis=axis, keepdims=True), inp\_shape)  
 x = (img - img\_min) / (img\_max - img\_min + float(1e-18))  
 return x

import warnings  
import os  
import pandas as pd  
from batchgenerators.dataloading.multi\_threaded\_augmenter import (  
 MultiThreadedAugmenter,  
) # 多线程数据增强数据加载器  
import wandb  
from tqdm import tqdm # 显示进度条  
import torch # PyTorch深度学习库  
from torch import nn # 神经网络模块  
import segmentation\_models\_pytorch as smp # 用于图像分割的PyTorch中的模型  
from torchsummary import summary # 打印模型概要信息  
import math  
import time

c:\Software\Python311\Lib\site-packages\tqdm\auto.py:21: TqdmWarning: IProgress not found. Please update jupyter and ipywidgets. See https://ipywidgets.readthedocs.io/en/stable/user\_install.html  
 from .autonotebook import tqdm as notebook\_tqdm

1. 忽略Python警告，以确保代码更加简洁、可读和易于维护。
2. 使用WandB库初始化一个新的项目并将其与Gland\_Seg项目中的glaseg实体关联起来。（首先应当前往wandb注册账号获取apikey并在命令啊行运行wandb login）
3. 从指定的CSV文件中读取数据，并将其转化为数据集字典形式。
4. 根据配置文件中指定的参数，获取patch\_size、batch\_size和epochs。
5. 获取训练数据集的数据增强变换和使用MultiThreadedAugmenter类对数据进行扩增。
6. 使用DataLoader类对训练集和验证集进行分批处理，设置batch\_size、patch\_size、num\_threads\_in\_multithreaded等参数，并使用了pin\_memory=False命令禁用固定内存功能。



warnings.filterwarnings("ignore")  
# wandb.init(project="Gland\_Seg", entity="glaseg", config="config/config.yaml")  
wandb.init(project="Gland\_Seg", config="config/config.yaml")  
config = wandb.config  
tabular\_data = pd.read\_csv(config.csv)  
ds\_dict = get\_split\_fold(tabular\_data)  
patch\_size = eval(config.patch\_size)  
batch\_size = config.batch\_size  
epochs = config.epochs  
tr\_transforms = get\_train\_transform(patch\_size, prob=config.aug\_prob)  
train\_dl = DataLoader(  
 data=ds\_dict["train\_ds"],  
 batch\_size=batch\_size,  
 patch\_size=patch\_size,  
 num\_threads\_in\_multithreaded=4,  
 seed\_for\_shuffle=5243,  
 return\_incomplete=False,  
 shuffle=True,  
 infinite=True,  
)  
train\_gen = MultiThreadedAugmenter(  
 train\_dl,  
 tr\_transforms,  
 num\_processes=4,  
 num\_cached\_per\_queue=2,  
 seeds=None,  
 pin\_memory=False,  
)  
val\_dl = DataLoader(  
 data=ds\_dict["testA\_ds"],  
 batch\_size=batch\_size,  
 patch\_size=patch\_size,  
 num\_threads\_in\_multithreaded=1,  
 seed\_for\_shuffle=5243,  
 return\_incomplete=False,  
 shuffle=True,  
 infinite=True,  
)

Failed to detect the name of this notebook, you can set it manually with the WANDB\_NOTEBOOK\_NAME environment variable to enable code saving.  
wandb: Currently logged in as: xftxyz2001. Use `wandb login --relogin` to force relogin

<IPython.core.display.HTML object>

<IPython.core.display.HTML object>

<IPython.core.display.HTML object>

<IPython.core.display.HTML object>

<IPython.core.display.HTML object>

1. 使用 smp.Unet 模型定义一个深度学习模型，并配置所需参数，然后定义一个优化器并指定学习率。
2. 定义一个学习率调度器，用于在训练过程中动态调整学习率。（调度器在验证集上监测到模型性能没有提高时，就将学习率减少一个倍数。）
3. 使用标签平滑技术定义一个损失函数。

# 获取用于训练的 CPU 或 GPU 设备  
device = "cuda" if torch.cuda.is\_available() else "cpu"  
  
# 定义模型  
model = smp.Unet(  
 encoder\_name=config.encoder\_model,  
 decoder\_use\_batchnorm=True,  
 in\_channels=3,  
 classes=config.n\_class,  
).to(device)  
  
# 打印模型概述  
summary(model, (3, 512, 512))  
# 定义优化器  
optimizer = eval(config.optimizer)(model.parameters(), lr=float(config.learning\_rate))  
  
# 定义学习率调度器  
scheduler = torch.optim.lr\_scheduler.ReduceLROnPlateau(  
 optimizer, patience=20, factor=0.1  
)  
  
# 定义 Dice 损失函数  
dice\_loss = smp.losses.DiceLoss(smp.losses.BINARY\_MODE, from\_logits=True)  
  
# 定义带有标签平滑的交叉熵损失函数  
xent = smp.losses.SoftBCEWithLogitsLoss(smooth\_factor=0.1)

----------------------------------------------------------------  
 Layer (type) Output Shape Param #  
================================================================  
 Conv2d-1 [-1, 64, 256, 256] 9,408  
 BatchNorm2d-2 [-1, 64, 256, 256] 128  
 ReLU-3 [-1, 64, 256, 256] 0  
 MaxPool2d-4 [-1, 64, 128, 128] 0  
 Conv2d-5 [-1, 64, 128, 128] 36,864

…

Identity-125 [-1, 1, 512, 512] 0  
 Activation-126 [-1, 1, 512, 512] 0  
================================================================  
Total params: 14,328,209  
Trainable params: 14,328,209  
Non-trainable params: 0  
----------------------------------------------------------------  
Input size (MB): 3.00  
Forward/backward pass size (MB): 1139.00  
Params size (MB): 54.66  
Estimated Total Size (MB): 1196.66  
----------------------------------------------------------------

自定义一个损失函数 custom\_loss 接受模型的预测值和目标值作为输入，通过计算二分类交叉熵和Dice loss来计算总损失，并返回总损失、二分类交叉熵和Dice loss三个值。

def custom\_loss(pred, target):  
 xent\_l = xent(pred, target)  
 dice\_l = dice\_loss(pred, target)  
 loss = xent\_l + dice\_l  
 return loss, xent\_l, dice\_l

接受模型和优化器作为输入，通过训练集数据计算总损失、二分类交叉熵和Dice loss，并返回这些值，以及输入图像、目标掩模和模型的预测掩模。

def train(model, optimizer):  
 # total number of training batches  
 num\_batches = math.ceil(len(ds\_dict["train\_ds"]["img\_npy"]) / batch\_size)  
 model.train()  
 batch\_xent\_l = []  
 batch\_dice\_l = []  
 batch\_loss = []  
 print("Training...")  
 for i in tqdm(range(num\_batches)):  
 train\_batch = next(train\_gen)  
 imgs = train\_batch["data"]  
 segs = train\_batch["seg"]  
 # normalization  
 imgs = min\_max\_norm(imgs)  
 # binarisation  
 segs = np.where(segs > 0.0, 1.0, 0.0).astype("float32")  
 segs = np.expand\_dims(segs[:, 0, :, :], 1)  
 imgs, segs = torch.from\_numpy(imgs).to(device), torch.from\_numpy(segs).to(  
 device  
 )  
 # Compute loss  
 pred = model(imgs)  
 loss, xent\_l, dice\_l = custom\_loss(pred, segs)  
 # Backpropagation  
 optimizer.zero\_grad()  
 loss.backward()  
 optimizer.step()  
 # batch losses  
 batch\_xent\_l.append(xent\_l)  
 batch\_dice\_l.append(dice\_l)  
 batch\_loss.append(loss)  
 # apply sigmoid to masking  
 segs = nn.Sigmoid()(segs)  
 # taking the average along the batch  
 loss = torch.mean(torch.as\_tensor(batch\_loss)).item()  
 avg\_xent\_l = torch.mean(torch.as\_tensor(batch\_xent\_l)).item()  
 avg\_dice\_l = torch.mean(torch.as\_tensor(batch\_dice\_l)).item()  
  
 return {  
 "loss": loss,  
 "xent\_l": avg\_xent\_l,  
 "dice\_l": avg\_dice\_l,  
 "imgs": imgs.cpu().detach().numpy(),  
 "segs": segs.cpu().detach().numpy(),  
 "pred": pred.cpu().detach().numpy(),  
 }

接受模型作为输入，通过测试集数据计算总损失、二分类交叉熵和Dice loss，并返回这些值，以及输入图像、目标掩模和模型的预测掩模。

def test(model):  
 num\_batches = math.ceil(len(ds\_dict["testA\_ds"]["img\_npy"]) / batch\_size)  
 model.eval()  
 # no need back prop for testing set  
 batch\_xent\_l = []  
 batch\_dice\_l = []  
 batch\_loss = []  
 print("Testing...")  
 with torch.no\_grad():  
 for i in tqdm(range(num\_batches)):  
 val\_batch = next(val\_dl)  
 imgs = val\_batch["data"]  
 segs = val\_batch["seg"]  
 # normalization  
 imgs = min\_max\_norm(imgs)  
 # binarisation  
 segs = np.where(segs > 0.0, 1.0, 0.0).astype("float32")  
 segs = np.expand\_dims(segs[:, 0, :, :], 1)  
 imgs, segs = torch.from\_numpy(imgs).to(device), torch.from\_numpy(segs).to(  
 device  
 )  
 # Compute loss  
 pred = model(imgs)  
 loss, xent\_l, dice\_l = custom\_loss(pred, segs)  
 # batch losses  
 batch\_xent\_l.append(xent\_l)  
 batch\_dice\_l.append(dice\_l)  
 batch\_loss.append(loss)  
 # apply sigmoid to masking  
 segs = nn.Sigmoid()(segs)  
 # taking the average along the batch  
 loss = torch.mean(torch.as\_tensor(batch\_loss)).item()  
 avg\_xent\_l = torch.mean(torch.as\_tensor(batch\_xent\_l)).item()  
 avg\_dice\_l = torch.mean(torch.as\_tensor(batch\_dice\_l)).item()  
 return {  
 "loss": loss,  
 "xent\_l": avg\_xent\_l,  
 "dice\_l": avg\_dice\_l,  
 "imgs": imgs.cpu().detach().numpy(),  
 "segs": segs.cpu().detach().numpy(),  
 "pred": pred.cpu().detach().numpy(),  
 }

在训练和测试函数中，首先将输入数据进行归一化和二值化处理，然后将处理后的数据转换为PyTorch张量，将其输入到模型中进行预测，并计算损失。然后使用反向传播算法计算梯度，更新模型参数。最后，将损失和其他指标取平均值，并将所有张量转换为numpy数组，以便进行可视化和进一步分析。

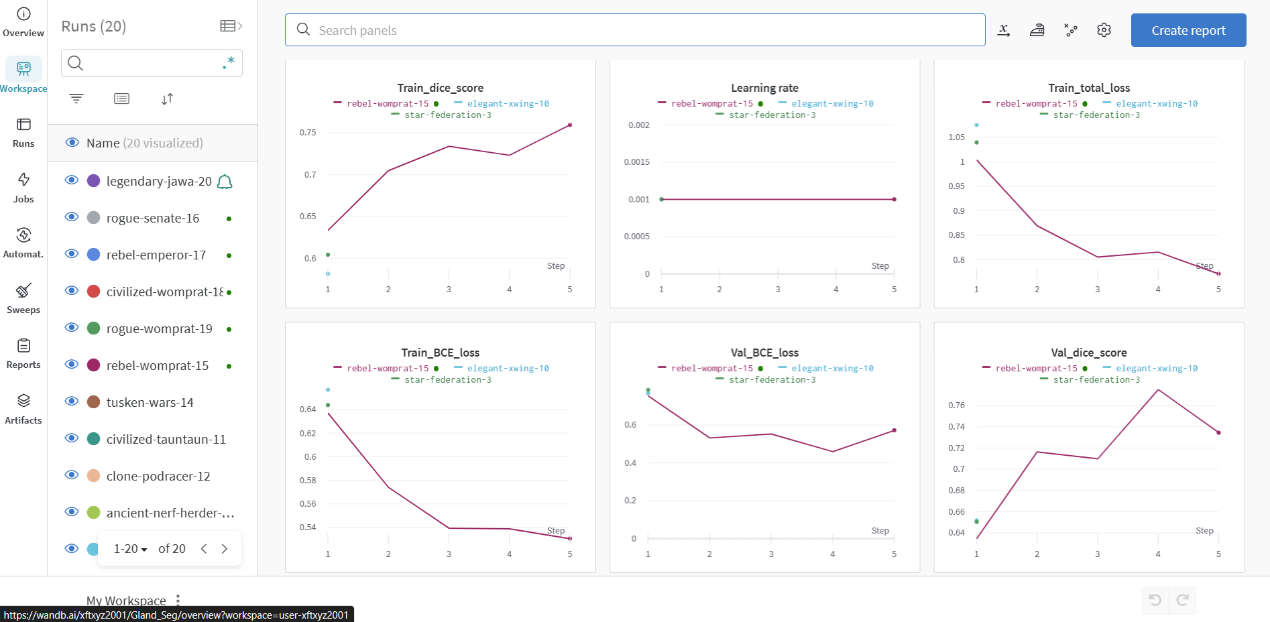
主函数逻辑：

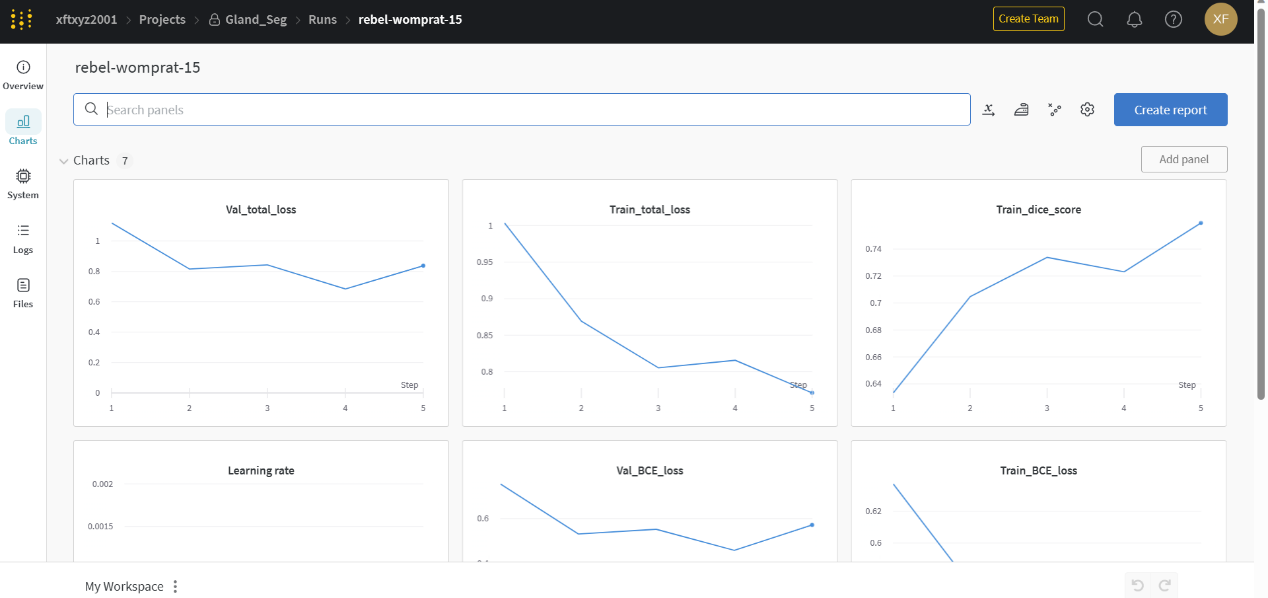
1. 定义两个变量current\_total\_loss和current\_dice\_score，分别用于记录当前最佳的总损失值和dice得分。
2. 对于每个epoch进行循环，从1到epochs+2。
3. 输出当前epoch的编号。
4. 调用train()函数进行模型训练，返回train\_output，包括总损失值、二元交叉熵损失值和dice得分。
5. 调用test()函数对模型进行测试，返回test\_output，包括总损失值、二元交叉熵损失值、dice得分、模型预测输出和原始图像和标签。
6. 调用scheduler.step()函数来更新学习率，scheduler是一个torch.optim.lr\_scheduler.ReduceLROnPlateau类型的对象，用于动态调整学习率。
7. 打印训练输出结果和验证输出结果，分别包括总损失值、二元交叉熵损失值和dice得分。
8. 使用wandb.log()函数记录日志，包括训练集和验证集的总损失值、二元交叉熵损失值和dice得分，以及当前的学习率。
9. 如果当前epoch是10的倍数，则进行预测可视化并使用wandb.log()函数记录可视化结果。
10. 如果当前的验证集总损失值小于current\_total\_loss，则保存当前模型的参数，并将current\_total\_loss更新为当前验证集总损失值。同样的，如果当前dice得分大于current\_dice\_score，则保存当前模型的参数，并将current\_dice\_score更新为当前dice得分。
11. 循环结束后，打印模型的训练时间。

def main():  
 start = time.time()  
 current\_total\_loss = 1000  
 current\_dice\_score = 0  
 for e in range(1, epochs + 2):  
 print("Epcohs:", e)  
 train\_output = train(model, optimizer)  
 test\_output = test(model)  
 scheduler.step(test\_output["loss"])  
 print("Training Outputs: ")  
 print(  
 "Total loss: {:.2f}, BCE: {:.2f}, Dice Score: {:.2f}".format(  
 train\_output["loss"], train\_output["xent\_l"], 1 - train\_output["dice\_l"]  
 )  
 )  
 print("-" \* 100)  
 print("Validation Outputs: ")  
 print(  
 "Total loss: {:.2f}, BCE: {:.2f}, Dice Score: {:.2f}".format(  
 test\_output["loss"], test\_output["xent\_l"], 1 - test\_output["dice\_l"]  
 )  
 )  
 # logging  
 wandb.log(  
 {  
 "Train\_total\_loss": train\_output["loss"],  
 "Val\_total\_loss": test\_output["loss"],  
 },  
 step=e,  
 )  
 wandb.log(  
 {  
 "Train\_BCE\_loss": train\_output["xent\_l"],  
 "Val\_BCE\_loss": test\_output["xent\_l"],  
 },  
 step=e,  
 )  
 wandb.log(  
 {  
 "Train\_dice\_score": 1 - train\_output["dice\_l"],  
 "Val\_dice\_score": 1 - test\_output["dice\_l"],  
 },  
 step=e,  
 )  
 wandb.log({"Learning rate": optimizer.param\_groups[0]["lr"]}, step=e)  
 if e % 10 == 0:  
 # threshold sigmoid output with 0.5  
 pred\_thr = np.where(test\_output["pred"] > 0.5, 1.0, 0.0)  
 # sample a dataset from the batch for visualization purpose  
 imgs = [  
 test\_output["imgs"][0, 0, :, :],  
 test\_output["segs"][0, 0, :, :],  
 pred\_thr[0, 0, :, :],  
 ]  
 captions = ["Gland Image", "Masking", "Prediction"]  
 fig = plot\_comparison(  
 imgs,  
 captions,  
 plot=False,  
 n\_col=len(imgs),  
 figsize=(12, 12),  
 cmap="gray",  
 )  
 wandb.log({"Validation Dataset Output Sample": wandb.Image(fig)}, step=e)  
  
 # save model  
 weights\_dir = "./weights/"  
 if not os.path.exists(weights\_dir):  
 os.makedirs(weights\_dir)  
 base\_path = os.path.split(weights\_dir)[0]  
 if test\_output["loss"] < current\_total\_loss:  
 current\_total\_loss = test\_output["loss"]  
 torch.save(model.state\_dict(), weights\_dir + "best\_loss\_{}.pth".format(e))  
 wandb.save(  
 os.path.join(weights\_dir, "best\_loss\_{}.pth".format(e)),  
 base\_path=base\_path,  
 )  
 if (1 - test\_output["dice\_l"]) > current\_dice\_score:  
 current\_dice\_score = 1 - test\_output["dice\_l"]  
 torch.save(model.state\_dict(), weights\_dir + "best\_dice\_{}.pth".format(e))  
 wandb.save(  
 os.path.join(weights\_dir, "best\_dice\_{}.pth".format(e)),  
 base\_path=base\_path,  
 )  
 print()  
  
 print("Model training runtime: {} mins".format((time.time() - start) / 60.0))

if \_\_name\_\_ == "\_\_main\_\_":  
 main()

受限于设备性能，最终没有完成模型的下面是部分wandb的图表：





1. **总结（心得体会）**

语义分割是一种计算机视觉任务，旨在将图像中的每个像素分配到预定义的语义类别中。与对象检测任务不同，语义分割要求对图像中每个像素进行分类，而不仅仅是对对象进行检测和定位。

因此，语义分割是一种像素级别的图像分析技术，可以用于许多应用程序，如自动驾驶、医疗图像分析和图像编辑等。在语义分割中，每个像素都被分配到特定的类别中，这通常通过在图像上绘制彩色掩膜来可视化。掩膜中的每个颜色代表一种类别。

在本次实践中，我们导入必要的模块，建立训练和验证数据加载器，初始化UNet模型及其优化器，并定义模型的损失函数。定义了一个定制的训练函数，该函数通过训练集运行模型，计算损失，并使用反向传播更新模型的权重。类似地，定义了一个测试函数来计算测试集上的损失，而不更新模型的权重。

上面的Python脚本是进行腺体分割。首先我们导入必要的库，如PyTorch、NumPy和Pandas。之后引入一些自定义模块，如utils、dataloader和segmentation\_models\_pytorch。

初始化权重和偏差(wandb)项目，并根据位于config/config. YAML中的YAML配置文件配置一些参数。YAML文件用于配置模型超参数，如学习率、epoch数、批处理大小、补丁大小和编码器模型架构。

定义自定义损失函数以及训练和测试函数。训练函数使用训练数据集训练模型，而测试函数在验证数据集上评估训练后的模型。

使用来自segmentation\_models\_pytorch的Unet架构，这是用于图像分割任务的类似于u - net的架构。该模型采用带标记平滑的二元交叉熵损失函数和一个骰子损失函数。用于训练的优化器在YAML配置文件中指定，学习率也由配置文件设置。

训练函数计算损失，反向传播，并使用指定的优化器更新模型的权重。测试函数通过计算损失来评估验证数据集上的训练模型，并且在评估期间不执行反向传播或权重更新。

使用batchgenerators库中的multithreadadaugmenter类使用多线程数据增强。这个库为图像分割任务提供了一些有用的数据增强函数和实用程序。

最后，将训练函数的一些重要输出保存为numpy数组，以供进一步分析，例如损失、二进制交叉熵损失、骰子损失以及预测的和真实的分割掩码。