

# 《模式识别与机器学习》课程大作业

(2021-2022 春季学期)

姓 名	刘帅
学 院	人工智能学院
专 业	人工智能
班 级	2020219111
学 号	2020212267
班内序号	30

2022 年 6 月

# 一、任务描述

## 1.1 待解决任务的解释

### 1.1.1 任务导览

本次作业选择的Kaggle比赛为[TGS Salt Identification Challenge](#),进行地表下盐沉积物的分割任务。

### 1.1.2 任务背景

#### 1.1.2.1 需求背景

地球上石油和天然气聚集地区在地表之下具有大量的盐沉积物，然而人工确认盐分的位置以及盐体是较为困难的，并对石油天然气公司的钻井人员安全产生潜在影响。本题目希望参赛者根据图片及专家给出的盐体分割结果来判定盐的位置。

#### 1.1.2.2 数据集简介

目录结构：

—competition\_data

——train

描述：本次任务的训练数据集

———images

描述：该部分为真实盐层成像图所在文件夹

———4000张成像图

描述：该部分为为盐层成像图，共包含4000张图

片，图片大小为101x101

———masks

描述：该部分为盐层标签图所在文件夹

———4000张分割标签

描述：该部分为盐层标签图，与images中图片

一一对应，图片数量，大小均与images相同。

——test

描述：本次任务的测试数据集

———images

描述：盐层成像图片所在文件夹

———18000张成像图

描述：岩层真实成像图片，需要根据图片推理出

标签图

——depths.csv

描述：该文件描述盐层深度（本次实验中未使用）。

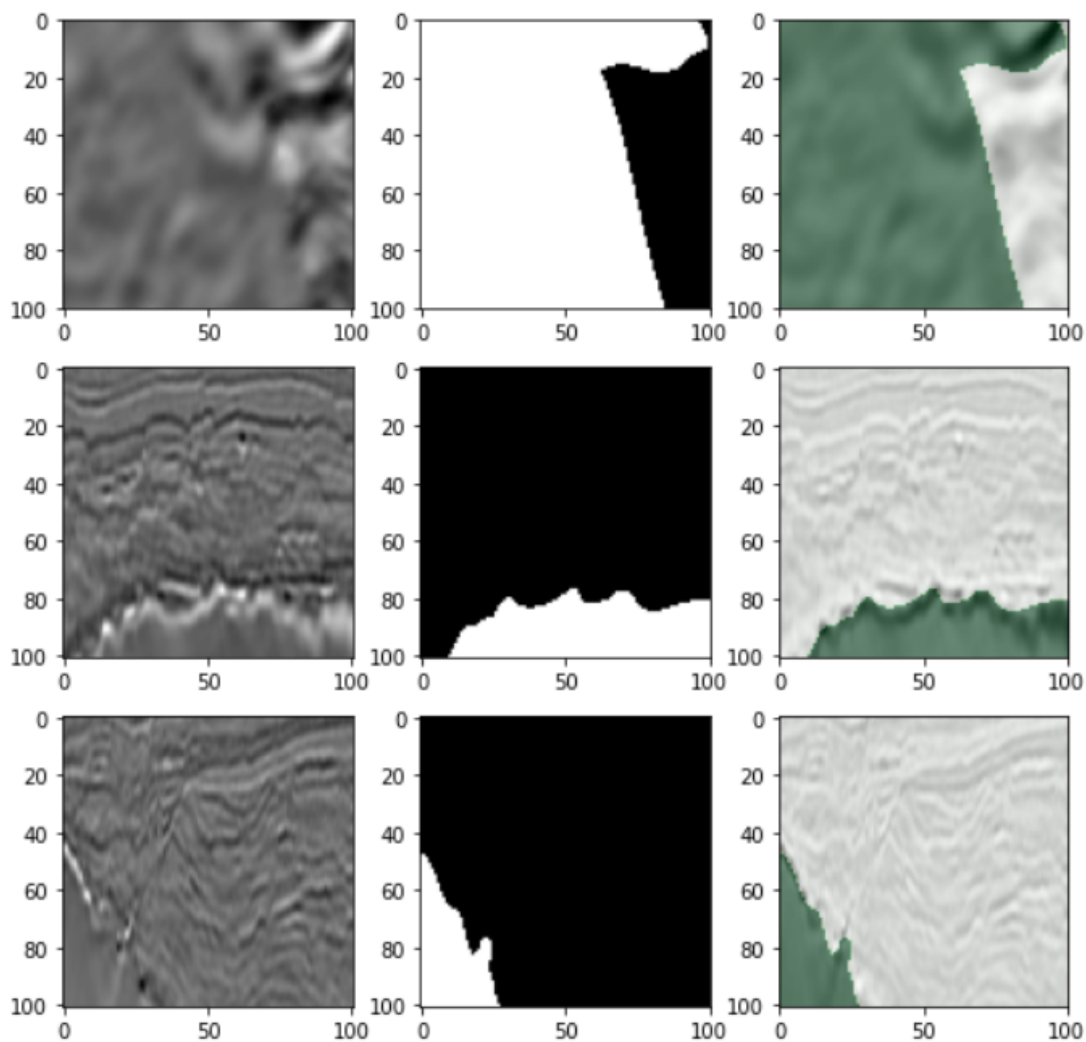
——sample\_submission.csv

描述：提交结果样例

——train.csv

描述：训练集的标签（主要用于检验，本次实

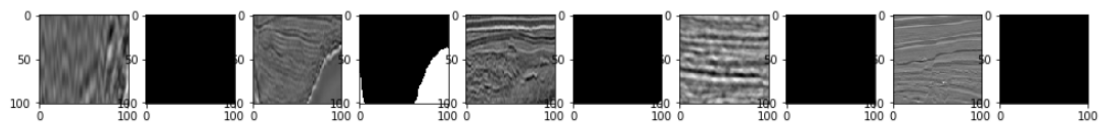
验中未使用)



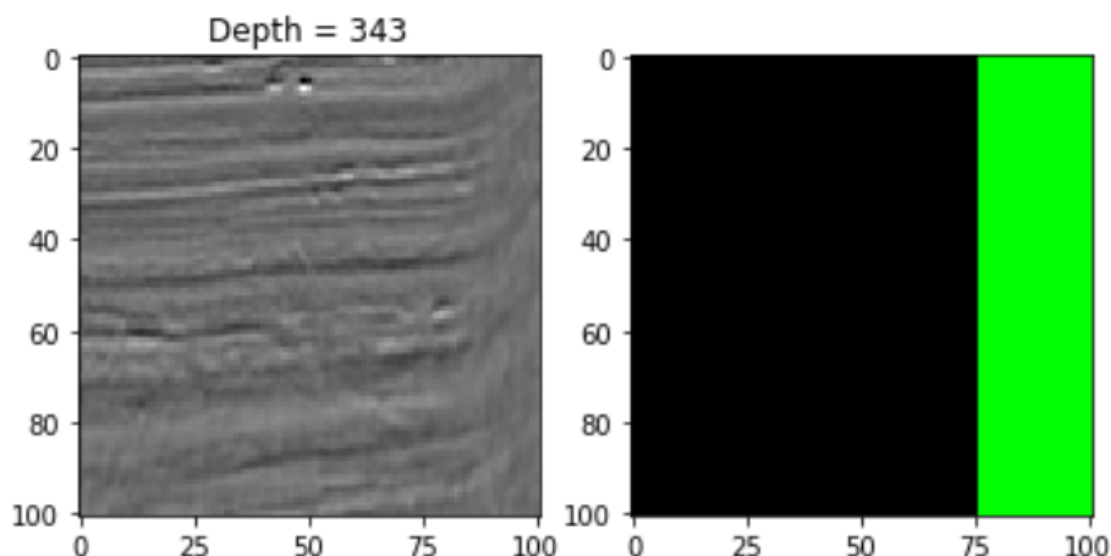
如图所示，第一列三张图取自images，表示真实盐层成像图。第二列三张图片表示其盐层分割结果，其中，白色代表识别为盐层。

### 1.1.2.3 数据集出现的问题

在进行盐体分割实验之前，我参照了该比赛中有关盐体识别和数据集背景的文章，[Intro to seismic, salt, and how to geophysics](#)，其中提到，超过39%的图片存在数据空标注或未标注的情况。



即4000张图片，仅有2400张图片是存在数据标注的。在存在数据标注的图片中，又存在许多错误标记的情况（也有人称是地质学家做标注时无法判定）



由此可见，此类分割任务训练集中存在较多脏数据，我们需要利用一定的算法对此类数据进行筛选。同时，避免丧失其泛化能力。

#### 1.1.2.4 为什么选择这个任务？

首先，本学期在学习时介绍到了感知机模型，全连接网络和卷积网络均利用了感知机模型的思想，同时适合解决XOR问题。其次，本学期在逻辑斯蒂回归模型章节提到了KL散度和交叉熵的相关知识，利用交叉熵原理进行参数估计可以很好的作为二分类问题的估计量。针对盐层分割问题是一个很好的二分类问题，交叉熵损失在图像类问题处理也较为适合。

同时，从大一下学期以来，我在吴铭老师的实验室已经实习近一年的时间，在一年的时间里，对于语义分割相关领域进行了较为全面的探索。因此，我希望借助此次kaggle比赛检验我目前对于语义分割任务的相关理解。同时，在我参与的课题研究中，也在数据处理和pipeline方面进行了创新。因此，遇到同类型的脏数据处理问题，我希望能够沿用自蒸馏（self-distilled）的方法进行数据的处理工作。

### 1.2任务的形式化描述

针对该任务，初步设定的算法流程为：

- 1、数据集读取，建立dataloader
- 2、搭建Unet网络
- 3、定义评估指标
- 4、利用训练集采用模型进行初步推理，进行数据筛选，并将筛选数据重新建立数据集。
- 5、利用重建数据集再次训练，得到推理模型
- 6、在测试集上进行数据推理
- 7、保存输出结果

## 二、介绍所选算法

### 2.1 网络综述

对于分割类任务，采用Unet网络具有很好的普适性，对于盐类分割这种数据量较小，且分辨率不高的问题，利用Unet网络训练可以更快的收敛并达到可观的实验效果

### 2.2 网络详细介绍

#### 2.2.1 Unet结构示意图

该图示为Unet论文所给出的网络结构示意图，Unet网络共分为三个部分：①左侧通道持续上升，而feature map图像大小不断减小的下采样过程。②右侧通道还原，feature map图像大小不断上升的上采样过程③不同维度的featuremap与上采样后的结果进行concat，作为上采样的featuremap。这样做有如下几个好处：首先，下采样部分类似于传统分类网络中的识别环节，其主要目的是获取图片的语义特征。上采样过程不仅可以将图片恢复至原有分辨率，还能够起到定位效果。而下采样的featuremap与上采样结果相连，可以减小因为上采样过程造成的语义信息丢失。

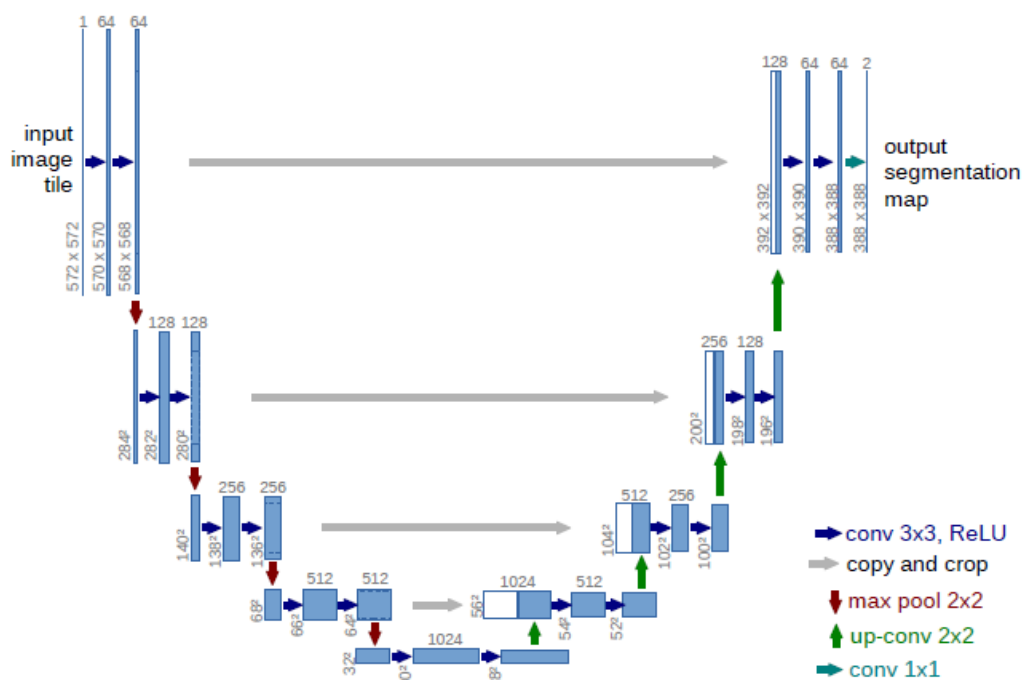


Fig. 1. U-net architecture (example for 32x32 pixels in the lowest resolution). Each blue box corresponds to a multi-channel feature map. The number of channels is denoted on top of the box. The x-y-size is provided at the lower left edge of the box. White boxes represent copied feature maps. The arrows denote the different operations.

#### 2.2.2 Unet网络结构解析

##### 2.2.2.1 导入torch相应的库

```
import torch
import torch.nn as nn
import torch.nn.functional as F
```

### 2.2.2.2 定义卷积层

其中包含一个卷积层和一个batchnorm层，并进行一步激活操作。

```
class convBlock(nn.Module):
    def __init__(self, in_channels, filters, size, stride=1, activation=True):
        super(convBlock, self).__init__()
        self.activation = activation
        self.conv = nn.Conv2d(in_channels, filters, size, stride=stride,
padding=size // 2)
        self.norm = nn.BatchNorm2d(filters)

    def forward(self, x):
        x = self.conv(x)
        x = self.norm(x)
        if self.activation:
            return F.relu(x)
        else:
            return x
```

### 2.2.2.3 定义残差模块

本次实验中，Unet网络中的下采样部分选用resnet作为backbone，首先记录原始输入作为residual部分，而后经过两层定义的convblock，将输出结果与初始输入x进行跳连，实现resnet中的跳连。

```
class residualBlock(nn.Module):
    def __init__(self, in_channels, filters, size=3):
        super(residualBlock, self).__init__()

        self.norm = nn.BatchNorm2d(in_channels)
        self.conv1 = convBlock(in_channels, filters, size)
        self.conv2 = convBlock(filters, filters, size, activation=False)

    def forward(self, x):
        residual = x
        x = F.relu(x)
        x = self.norm(x)
        x = self.conv1(x)
        x = self.conv2(x)
        x += residual
        return x
```

### 2.2.2.4 定义上采样块

采用转置卷积，将输入x1进行升维操作，并定义下采样部分的feature map为x2,将下采样部分与反卷积得到的结果进行连接，作为上采样的输出。

```

class deconvBlock(nn.Module):
    def __init__(self, in_channels, out_channels, kernel_size=2, stride=2):
        super(deconvBlock, self).__init__()

        self.deconv = nn.ConvTranspose2d(in_channels, out_channels,
kernel_size=kernel_size, stride=stride)

    def forward(self, x1, x2):
        xd = self.deconv(x1)
        x = torch.cat([xd, x2], dim=1)
        return x

```

### 2.2.2.5 定义unet网络

对在下采样部分的结构类似，即为池化-dropout-卷积层-残差层-激活函数，每经过一个下采样模块，通道数翻倍，同时图片长宽减半。

利用公式 $N = [(W - F + 2P) / Stride] + 1$ 对网络进行搭建。

```

class UnetModel(nn.Module):
    def __init__(self, filters=16, dropout=0.5):
        super(UnetModel, self).__init__()
        self.conv1 = nn.Sequential(
            nn.Conv2d(1, filters, 3, padding=1),
            residualBlock(filters, filters),
            residualBlock(filters, filters),
            nn.ReLU()
        )
        self.conv2 = nn.Sequential(
            nn.MaxPool2d(2, 2),
            nn.Dropout(dropout / 2),
            nn.Conv2d(filters, filters * 2, 3, padding=1),
            residualBlock(filters * 2, filters * 2),
            residualBlock(filters * 2, filters * 2),
            nn.ReLU()
        )
        self.conv3 = nn.Sequential(
            nn.MaxPool2d(2, 2),
            nn.Dropout(dropout),
            nn.Conv2d(filters * 2, filters * 4, 3, padding=1),
            residualBlock(filters * 4, filters * 4),
            residualBlock(filters * 4, filters * 4),
            nn.ReLU()
        )
        self.conv4 = nn.Sequential(
            nn.MaxPool2d(2, 2),
            nn.Dropout(dropout),
            nn.Conv2d(filters * 4, filters * 8, 3, padding=1),
            residualBlock(filters * 8, filters * 8),
            residualBlock(filters * 8, filters * 8),
            nn.ReLU()
        )
        self.middle = nn.Sequential(
            nn.MaxPool2d(2, 2),
            nn.Dropout(dropout),

```

```

        nn.Conv2d(filters * 8, filters * 16, 3, padding=3 // 2),
        residualBlock(filters * 16, filters * 16),
        residualBlock(filters * 16, filters * 16),
        nn.ReLU()
    )
    self.deconv4 = deconvBlock(filters * 16, filters * 8, 2)
    self.upconv4 = nn.Sequential(
        nn.Dropout(dropout),
        nn.Conv2d(filters * 16, filters * 8, 3, padding=1),
        residualBlock(filters * 8, filters * 8),
        residualBlock(filters * 8, filters * 8),
        nn.ReLU()
    )
    self.deconv3 = deconvBlock(filters * 8, filters * 4, 3)
    self.upconv3 = nn.Sequential(
        nn.Dropout(dropout),
        nn.Conv2d(filters * 8, filters * 4, 3, padding=1),
        residualBlock(filters * 4, filters * 4),
        residualBlock(filters * 4, filters * 4),
        nn.ReLU()
    )
    self.deconv2 = deconvBlock(filters * 4, filters * 2, 2)
    self.upconv2 = nn.Sequential(
        nn.Dropout(dropout),
        nn.Conv2d(filters * 4, filters * 2, 3, padding=1),
        residualBlock(filters * 2, filters * 2),
        residualBlock(filters * 2, filters * 2),
        nn.ReLU()
    )
    self.deconv1 = deconvBlock(filters * 2, filters, 3)
    self.upconv1 = nn.Sequential(
        nn.Dropout(dropout),
        nn.Conv2d(filters * 2, filters, 3, padding=1),
        residualBlock(filters, filters),
        residualBlock(filters, filters),
        nn.ReLU(),
        nn.Dropout(dropout / 2),
        nn.Conv2d(filters, 1, 3, padding=1)
    )
    def forward(self, x):
        conv1 = self.conv1(x) # 101 -> 101
        conv2 = self.conv2(conv1) # 50 -> 25
        conv3 = self.conv3(conv2) # 25 -> 12
        conv4 = self.conv4(conv3) # 12 - 6
        x = self.middle(conv4) # 6 -> 12
        x = self.deconv4(x, conv4) #与上采样结果进行跳连
        x = self.upconv4(x) # 12 -> 25
        x = self.deconv3(x, conv3) #与上采样结果进行跳连
        x = self.upconv3(x) # 25 -> 50
        x = self.deconv2(x, conv2) #与上采样结果进行跳连
        x = self.upconv2(x) # 50 -> 101
        x = self.deconv1(x, conv1) #与上采样结果进行跳连
        x = self.upconv1(x)

    return x

```



设batch大小为16，每一层输出结果如图所示：

```
conv1: torch.Size([16, 16, 101, 101])
conv2: torch.Size([16, 32, 50, 50])
conv3: torch.Size([16, 64, 25, 25])
conv4: torch.Size([16, 128, 12, 12])
middle: torch.Size([16, 256, 6, 6])
deconv4: torch.Size([16, 256, 12, 12])
upconv4: torch.Size([16, 128, 12, 12])
deconv3: torch.Size([16, 128, 25, 25])
upconv3: torch.Size([16, 64, 25, 25])
deconv2: torch.Size([16, 64, 50, 50])
upconv2: torch.Size([16, 32, 50, 50])
deconv1: torch.Size([16, 32, 101, 101])
output: torch.Size([16, 1, 101, 101])
```

### 三、基于Unet网络 and 自蒸馏算法的盐体分割任务

#### 3.1 方法框图

# 盐体分割

建立dataloader, 读取数据

建立Unet网络

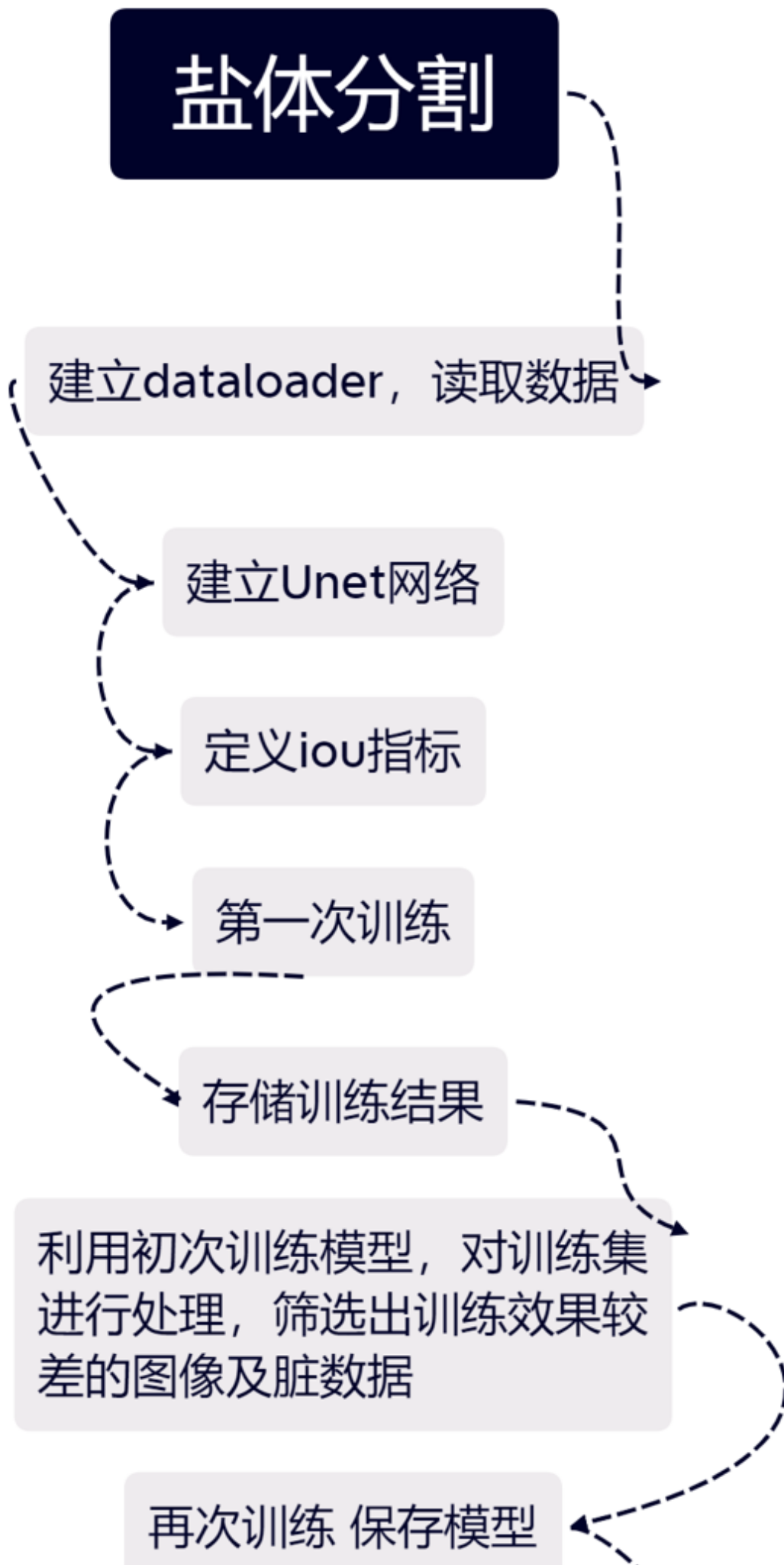
定义iou指标


第一次训练

存储训练结果

利用初次训练模型, 对训练集  
进行处理, 筛选出训练效果较  
差的图像及脏数据

再次训练 保存模型





在测试集上进行推理，将标注信息导出

## 3.2 代码介绍

### 3.2.1 导入可视化、pytorch相关库

定义时间戳，对训练过程用时进行记录，利用kaggle上的gpu进行训练，设置device为“cuda”类型

```
import os
import time
import math
import glob
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from PIL import Image

import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import Dataset, DataLoader
import torchvision
import torchvision.transforms as T
from torchvision import models

def timeSince(since):
    now = time.time()
    s = now - since
    m = math.floor(s / 60)
    s -= m * 60
    return '%dm %ds' % (m, s)

device = torch.device("cuda")
```

### 3.2.2 文件解压缩，定义图像路径

由于图像存储在input中的zip文件中，先利用命令将文件进行解压缩，并设置图片和标签的路径

```
! unzip -q /kaggle/input/tgs-salt-identification-challenge/competition_data.zip
image_path = "/kaggle/working/competition_data/train/images"
mask_path = "/kaggle/working/competition_data/train/masks"
```

### 3.2.3 展示训练集中的部分结果

验证训练集读入正确，图片能通过Image读出至PIL格式，方便后续对于数据转化成tensor类型。

```
names = ['0a19821a16', '0df53ae04c', '1544a0e952']
images = [Image.open(os.path.join(image_path, name+'.png')) for name in names]
masks = [Image.open(os.path.join(mask_path, name+'.png')) for name in names]

transforms = T.Compose([T.Grayscale(), T.ToTensor()])
x = torch.stack([transforms(image) for image in images])
y = torch.stack([transforms(mask) for mask in masks])

fig = plt.figure( figsize=(9, 9))

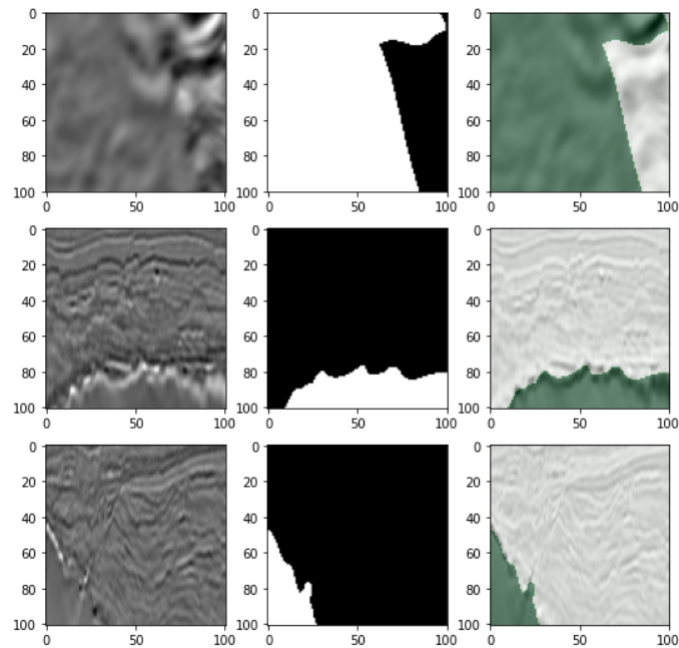
ax = fig.add_subplot(331)
plt.imshow(images[0])
ax = fig.add_subplot(332)
plt.imshow(masks[0])
ax = fig.add_subplot(333)
ax.imshow(x[0].squeeze(), cmap="Greys")
ax.imshow(y[0].squeeze(), alpha=0.5, cmap="Greens")

ax = fig.add_subplot(334)
plt.imshow(images[1])
ax = fig.add_subplot(335)
plt.imshow(masks[1])
ax = fig.add_subplot(336)
ax.imshow(x[1].squeeze(), cmap="Greys")
ax.imshow(y[1].squeeze(), alpha=0.5, cmap="Greens")

ax = fig.add_subplot(337)
plt.imshow(images[2])
ax = fig.add_subplot(338)
plt.imshow(masks[2])
ax = fig.add_subplot(339)
ax.imshow(x[2].squeeze(), cmap="Greys")
ax.imshow(y[2].squeeze(), alpha=0.5, cmap="Greens")

plt.show()
```

输出结果表示：



### 3.2.4 定义dataloader

定义dataloader，将数据读入pytorch中，为后续训练做准备。利用glob将图像信息获取，后续在一个batch中，没读到一个索引，返回该图片在灰度转化后以tensor表示其结果。

```
class segmentDataset(Dataset):
    def __init__(self, image_path, mask_path):
        self.image_path = image_path
        self.mask_path = mask_path

        image_list= glob.glob(image_path + '/*.png')
        sample_names = []
        for file in image_list:
            sample_names.append(file.split('/')[1].split('.')[0])

        self.sample_names = sample_names

        self.transforms = T.Compose([T.Grayscale(), T.ToTensor()])#compose实例化，
        转化为灰度图，并以tensor形式表示。

        def __getitem__(self, idx):
            image = Image.open(os.path.join(self.image_path,
            self.sample_names[idx]+'*.png'))
            mask = Image.open(os.path.join(self.mask_path,
            self.sample_names[idx]+'*.png'))
            return self.transforms(image), self.transforms(mask)

        def __len__(self):
            return len(self.sample_names)
```

### 3.2.5 实例化train dataloader

```
train_dataset = segmentDataset(image_path, mask_path)
```

### 3.2.6 Unet网络定义

该部分内容已经在**2.2.2 Unet网络结构解析**进行介绍，此处不再过多赘述。

### 3.2.7 训练指标定义

语义分割任务，训练指标为IOU（Intersection over Union），即 $\frac{A \cap B}{A \cup B}$ ，表示推理图片与原始标签的相似程度，对于二分类问题，输出结果经归一化后为[-1,1]之间的一个数，如果<0,表示代表标签0，即非盐体，若>0,则表示识别该像素属于盐体。

同时，定义训练函数，进行损失函数的反向传播和梯度更新。

```
def get_iou_score(outputs, labels):
    A = labels.squeeze().bool()
    pred = torch.where(outputs<0., torch.zeros_like(outputs),
torch.ones_like(outputs))
    B = pred.squeeze().bool()
    intersection = (A & B).float().sum((0,1))
    union = (A | B).float().sum((0,1))
    iou = intersection / union
    return iou

def train_one_batch(model, x, y):
    x, y = x.to(device), y.to(device)

    outputs = model(x)
    loss = loss_fn(outputs, y)
    iou = get_iou_score(outputs, y).mean()

    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
    return loss.item(), iou.item()
```

### 3.2.7 网络训练

训练70个轮次，每轮读取数据的大小为64个，记录每一轮训练时间及训练指标IOU，定义梯度更新的优化器为Adam，损失函数采用带有sigmoid归一化的二分类交叉熵损失(即loss进行归一化，使样本在0-1之间)

```
NUM_EPOCHS = 70
BATCH_SIZE = 64

model = UnetModel().to(device)
model.train()
optimizer = torch.optim.Adam(model.parameters())
scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer, 'max')

loss_fn = nn.BCEWithLogitsLoss()

train_dataloader = DataLoader(train_dataset, batch_size = BATCH_SIZE, shuffle =
True, drop_last = True)
steps = train_dataset.__len__()// BATCH_SIZE
print(steps,"steps per epoch")
```

```

start = time.time()
train_losses = []
train_iious = []
for epoch in range(1, NUM_EPOCHS + 1):
    print('-' * 10)
    print('Epoch {}/{}'.format(epoch, NUM_EPOCHS))
    running_iou = []
    running_loss = []
    for step, (x, y) in enumerate(train_data_loader):
        loss, iou = train_one_batch(model, x, y)
        running_iou.append(iou)
        running_loss.append(loss)
        print('\r{:6.1f} %\tloss {:8.4f}\tIoU {:8.4f}'.format(100*
(step+1)/steps, loss, iou), end = "")

    print('\r{:6.1f} %\tloss {:8.4f}\tIoU {:8.4f}\tt{}'.format(100*
(step+1)/steps, np.mean(running_loss), np.mean(running_iou), timesince(start)))
    scheduler.step(np.mean(running_iou))

train_losses.append(loss)
train_iious.append(iou)

```

训练过程如图所示：

```

-----
Epoch 1/70
100.0 %
-----
Epoch 2/70
100.0 %
-----
Epoch 3/70
100.0 %
-----
Epoch 4/70
100.0 %
-----
Epoch 5/70
100.0 %
-----
Epoch 6/70
100.0 %
-----
Epoch 7/70
100.0 %
-----
Epoch 8/70
100.0 %
-----
Epoch 9/70
100.0 %
-----
Epoch 10/70
100.0 %
-----
Epoch 11/70
100.0 %
-----
Epoch 12/70
100.0 %
-----
Epoch 13/70
100.0 %
-----
Epoch 14/70
100.0 %
-----
Epoch 15/70
100.0 %
-----
Epoch 16/70
100.0 %
-----
Epoch 17/70
100.0 %
-----

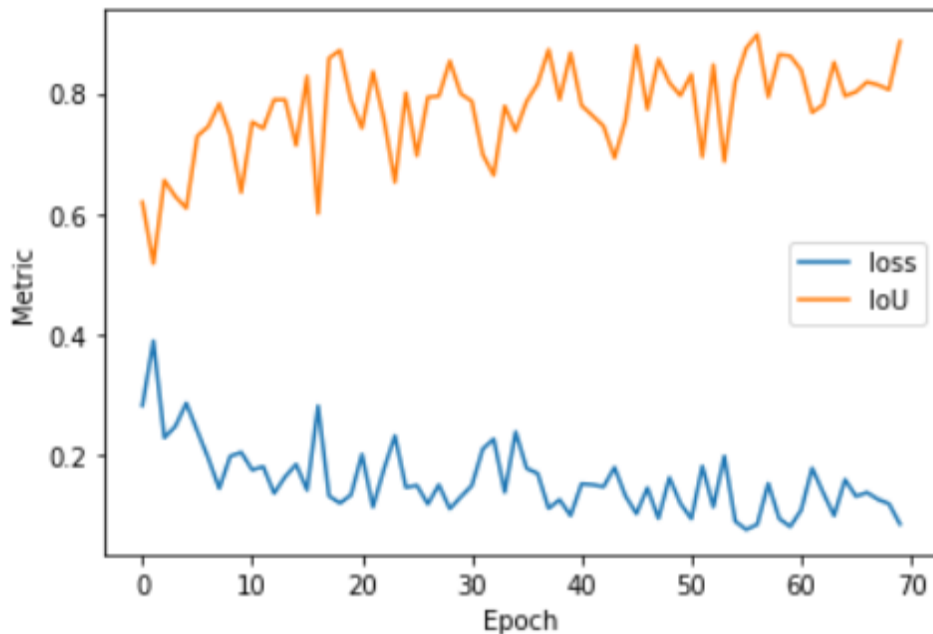
```

Epoch	loss	IoU	time
1/70	0.4454	0.4640	0m 22s
2/70	0.3242	0.5660	0m 39s
3/70	0.2873	0.6118	0m 56s
4/70	0.2517	0.6510	1m 13s
5/70	0.2440	0.6574	1m 29s
6/70	0.2418	0.6672	1m 46s
7/70	0.2334	0.6830	2m 2s
8/70	0.2110	0.7105	2m 19s
9/70	0.2075	0.7088	2m 36s
10/70	0.2013	0.7065	2m 52s
11/70	0.1959	0.7275	3m 9s
12/70	0.2022	0.7156	3m 25s
13/70	0.1957	0.7281	3m 42s
14/70	0.1931	0.7267	3m 59s
15/70	0.1875	0.7363	4m 16s
16/70	0.1925	0.7313	4m 32s
17/70			

### 3.2.8 训练结果可视化

```
plt.plot(train_losses, label = 'loss')
plt.plot(train_iou, label = 'IoU')
plt.xlabel('Epoch')
plt.ylabel('Metric')
plt.legend()
plt.show()
```

可视化结果如图所示，可见，loss在训练过程中下降，IoU指标在训练过程中上升。



### 3.2.9 筛选指标较好数据

在初次训练后，我们利用此时训练的中间模型遍历训练集，分别求得IOU，并与标签进行比较，若IOU数值小于0.4（阈值大小需要根据数据集和训练情况进行调整），说明该模型可能不利于学习（或数据本身就存在问题），对于这些存在问题的数据，我们将其丢弃，并检测train数据集中的全黑图片，该类图片可能由于标注不当导致，也应进行剔除，将剔除后的结果存入new\_train与new\_mask文件夹中。

```
os.mkdir('./new_train')
os.mkdir('./new_mask')
imglist=os.listdir(image_path)
masklist=os.listdir(mask_path)
i=0
import shutil
for ele in imglist:
    i+=1
    img=Image.open(image_path+'/'+ele)
    mask=Image.open(mask_path+'/'+ele)
    tensor = T.Compose([T.Grayscale(), T.ToTensor()])
    img=tensor(img).unsqueeze(0)
    mask=tensor(mask).unsqueeze(0)
    output=model(img)
    # print(get_iou_score(output,mask))
    if(get_iou_score(output,mask)>0.4) and
tensor(Image.open(mask_path+'/'+ele)).sum(1).sum(1)!=0 :
        shutil.copy(image_path+'/'+ele,"/kaggle/working/new_train/"+ele)
        shutil.copy(mask_path+'/'+ele,"/kaggle/working/new_mask/"+ele)
```



### 3.2.10 查看筛选后数据集的数据量

```
print(len(os.listdir("/kaggle/working/new_train/")))
print(len(os.listdir("/kaggle/working/new_mask/")))
```

输出结果为：

```
1827
1827
```

说明此时剔除出近60%的数据，我们在前文曾提到，由于40%的数据都属于全黑的数据，因此可用数据仅有2400张，同时，在学习过程中，由于初步训练的模型包含了上述脏数据，可能对某些不方便学习的图片的推理过程也产生了影响，由于实验时间问题，本实验仅仅将所有数据进行剔除。（原本需要进行多次对比试验、交叉验证等方法进行筛选）。

### 3.2.11 对筛选后数据再次训练

对于筛选后的数据再次进行70轮训练，此训练流程体现了自蒸馏的训练思想，即训练前后未改变任何实验参数，仅仅通过数据对于训练集本身的学习进行二次优化。

```
new_dataset = segmentDataset("/kaggle/working/new_train/",
                              "/kaggle/working/new_mask/")
NUM_EPOCHS = 70
BATCH_SIZE = 64

# model = torch.load('./unet_gpu.pth')
# model.train()
optimizer = torch.optim.Adam(model.parameters())
scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer, 'max')
model=model.to(device)
loss_fn = nn.BCEWithLogitsLoss()

train_dataloader = DataLoader(new_dataset, batch_size = BATCH_SIZE, shuffle =
True, drop_last = True)
steps = train_dataset.__len__()// BATCH_SIZE
print(steps,"steps per epoch")

start = time.time()
train_losses = []
train_iious = []
for epoch in range(1, NUM_EPOCHS + 1):
    print('-' * 10)
    print('Epoch {}/{}'.format(epoch, NUM_EPOCHS))
    running_iou = []
    running_loss = []
    for step, (x, y) in enumerate(train_dataloader):
        loss, iou = train_one_batch(model, x, y)
        running_iou.append(iou)
        running_loss.append(loss)
        print('\r{:6.1f} %\tloss {:8.4f}\tIoU {:8.4f}'.format(100*
(step+1)/steps, loss,iou), end = "")

    print('\r{:6.1f} %\tloss {:8.4f}\tIoU {:8.4f}\t{t}'.format(100*
(step+1)/steps,np.mean(running_loss),np.mean(running_iou), timesince(start)))
```

```
scheduler.step(np.mean(running_iou))
```

```
train_losses.append(loss)
```

```
train_iou.append(iou)
```

训练过程截图：

```
-----
Epoch 1/70
24.2 %      loss  0.1129  IoU   0.8834   0m 4s
-----
Epoch 2/70
24.2 %      loss  0.0852  IoU   0.9126   0m 8s
-----
Epoch 3/70
24.2 %      loss  0.0786  IoU   0.9178   0m 12s
-----
Epoch 4/70
24.2 %      loss  0.0743  IoU   0.9226   0m 16s
-----
Epoch 5/70
24.2 %      loss  0.0739  IoU   0.9225   0m 20s
-----
Epoch 6/70
24.2 %      loss  0.0673  IoU   0.9287   0m 24s
-----
Epoch 7/70
24.2 %      loss  0.0665  IoU   0.9290   0m 28s
-----
Epoch 8/70
24.2 %      loss  0.0696  IoU   0.9245   0m 32s
-----
Epoch 9/70
24.2 %      loss  0.0655  IoU   0.9312   0m 36s
-----
Epoch 10/70
24.2 %     loss  0.0622  IoU   0.9332   0m 40s
-----
Epoch 11/70
24.2 %     loss  0.0633  IoU   0.9317   0m 44s
-----
Epoch 12/70
24.2 %     loss  0.0616  IoU   0.9333   0m 48s
-----
Epoch 13/70
24.2 %     loss  0.0689  IoU   0.9266   0m 52s
-----
Epoch 14/70
24.2 %     loss  0.0624  IoU   0.9335   0m 56s
-----
Epoch 15/70
```

### 3.2.12 模型保存

```
model_path = 'unet_cpu.pth'
gpumodel_path = 'unet_gpu.pth'
## save weights
torch.save(model.cpu().state_dict(), model_path)
torch.save(model.state_dict(), gpumodel_path)
```

**\*3.2.13 再次定义iou指标**（由于下面测试的时候封装成了一个batch，因此交运算和并运算的维度从(0,1)改变成了(1,2)，其余部分完全相同。

```
def demo_get_iou_score(outputs, labels):
    A = labels.squeeze().bool()
    pred = torch.where(outputs<0., torch.zeros_like(outputs),
torch.ones_like(outputs))
    B = pred.squeeze().bool()
    intersection = (A & B).float().sum((1,2))
    union = (A| B).float().sum((1,2))
    iou = (intersection + 1e-6) / (union + 1e-6)
    return iou

def demo_train_one_batch(model, x, y):
    x, y = x.to(device), y.to(device)

    outputs = model(x)
```

```

loss = loss_fn(outputs, y)
iou = demo_get_iou_score(outputs, y).mean()

optimizer.zero_grad()
loss.backward()
optimizer.step()
return loss.item(), iou.item()

```

### 3.2.14 推理结果展示

```

model.eval()
names = ['6caec01e67', '2bfa664017', '1544a0e952']
images = [Image.open(os.path.join(image_path, name+'.png')) for name in names]
masks = [Image.open(os.path.join(mask_path, name+'.png')) for name in names]

transforms = T.Compose([T.Grayscale(), T.ToTensor()])
x = torch.stack([transforms(image) for image in images]).cuda()
y = torch.stack([transforms(mask) for mask in masks]).cuda()
outputs = model(x)
preds = torch.where(outputs<0., torch.zeros_like(outputs),
torch.ones_like(outputs)).cuda()
ious = demo_get_iou_score(outputs, y).cuda()
print(outputs.shape)
fig = plt.figure( figsize=(9, 12))

ax = fig.add_subplot(331)
plt.imshow(images[0])
ax = fig.add_subplot(332)
x=x.cpu().numpy()
y=y.cpu().numpy()
preds=preds.cpu().numpy()
ious=ious.cpu().numpy()
ax.imshow(x[0].squeeze(), cmap="Greys")
ax.imshow(y[0].squeeze(), alpha=0.5, cmap="Greens")
ax = fig.add_subplot(333)
ax.imshow(x[0].squeeze(), cmap="Greys")
ax.imshow(preds[0].squeeze(), alpha=0.5, cmap="OrRd")
ax.set_title("IoU: " + str(round(ious[0].item(), 2)), loc = 'left')

ax = fig.add_subplot(334)
plt.imshow(images[1])
ax = fig.add_subplot(335)
ax.imshow(x[1].squeeze(), cmap="Greys")
ax.imshow(y[1].squeeze(), alpha=0.5, cmap="Greens")
ax = fig.add_subplot(336)
ax.imshow(x[1].squeeze(), cmap="Greys")
ax.imshow(preds[1].squeeze(), alpha=0.5, cmap="OrRd")
ax.set_title("IoU: " + str(round(ious[1].item(), 2)), loc = 'left')

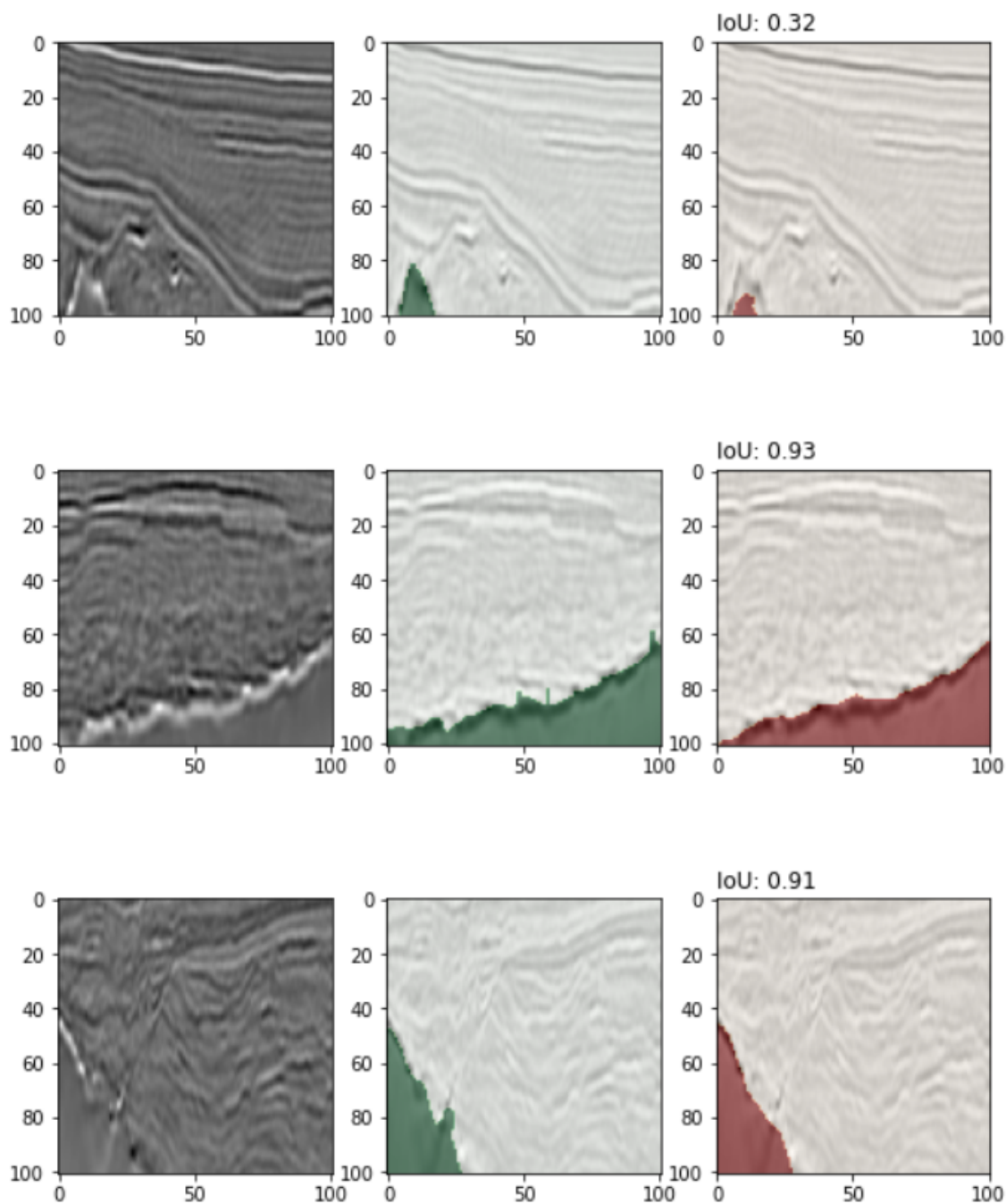
ax = fig.add_subplot(337)
plt.imshow(images[2])
ax = fig.add_subplot(338)
ax.imshow(x[2].squeeze(), cmap="Greys")
ax.imshow(y[2].squeeze(), alpha=0.5, cmap="Greens")
ax = fig.add_subplot(339)

```

```
ax.imshow(x[2].squeeze(), cmap="Greys")
ax.imshow(preds[2].squeeze(), alpha=0.5, cmap="OrRd")
ax.set_title("IoU: " + str(round(ious[2].item(), 2)), loc = 'left')

plt.show()
```

```
torch.Size([3, 1, 101, 101])
```



### 3.2.15 定义推理结果函数

将推理结果进行展平，并返回其在101x101图像下分割结果的位置

```
def rle_encode(im):
    pixels = im.flatten(order = 'F')
    pixels = np.concatenate([[0], pixels, [0]])
    runs = np.where(pixels[1:] != pixels[:-1])[0] + 1
    runs[1::2] -= runs[:-1]
    return ' '.join(str(x) for x in runs)

transforms = T.Compose([T.Grayscale(), T.ToTensor()])
```

### 3.2.16 推理结果生成

```
image_path = "/kaggle/working/competition_data/test/images"
sub_df = pd.read_csv('/kaggle/working/competition_data/sample_submission.csv')
n = sub_df.shape[0]
rle_mask = []
for idx in range(n):
    ## 加载测试集图像
    sample_name = sub_df['id'][idx]
    image = Image.open(os.path.join(image_path, sample_name+'.png'))
    image = transforms(image)
    image=image.cuda()
    ## 测试集预测
    out = model(image.unsqueeze(0)).squeeze()
    pred = torch.where(out<0., torch.zeros_like(out),
    torch.ones_like(out)).cpu()
    ## 写入mask
    rle_mask.append(rle_encode(pred.numpy()))
    print("\rprogress {}/{}".format(idx+1, n), end = "")

sub_df['rle_mask'] = rle_mask
sub_df.to_csv('submission.csv', index = False)#结果写入csv
```

## 四、实验结果分析

对于实验结果，着重讨论如下几点内容：

### 1、epoch选择问题

对于epoch选择，在200个epoch上进行实验是较为合适的，而考虑到实验时间、大作业完成时间等因素，本次实验任务采用的epoch数为140，已经有了初步拟合效果，该方面仍可能进行一部分提升。

### 2、对于自蒸馏算法的解释

利用自蒸馏，两次进行数据集筛选的方法，在同样的epoch次数下可以达到更好的效果，以图为例

Submission and Description	Private Score	Public Score	Use for Final Score
<a href="#">submission (4).csv</a> a few seconds ago by Shuai Liu <a href="#">add submission details</a>	0.65313	0.63579	<input type="checkbox"/>
<a href="#">self-distilled Unet for TGS competition (version 3/3)</a> 25 minutes ago by Shuai Liu Notebook self-distilled Unet for TGS competition   Version 3	Error	Error	<input type="checkbox"/>
<a href="#">submission (3).csv</a> 9 hours ago by Shuai Liu <a href="#">add submission details</a>	0.61604	0.58989	<input type="checkbox"/>

下图submission3为在原数据集训练140个epoch的结果，而submission (4) 则利用数据筛选的方法，前后训练了70个epoch，加起来总训练轮数一致，而产生了较好的实验效果，提高了近4个百分点。说明在训练过程中可能存在脏数据干扰。

Epoch 62/70					
100.0 %	loss	0.1190	IoU	0.8211	17m 15s
-----					
Epoch 63/70					
100.0 %	loss	0.1188	IoU	0.8233	17m 32s
-----					
Epoch 64/70					
100.0 %	loss	0.1207	IoU	0.8262	17m 48s
-----					
Epoch 65/70					
100.0 %	loss	0.1213	IoU	0.8261	18m 5s
-----					
Epoch 66/70					
100.0 %	loss	0.1259	IoU	0.8131	18m 21s
-----					
Epoch 67/70					
100.0 %	loss	0.1142	IoU	0.8282	18m 38s
-----					
Epoch 68/70					
100.0 %	loss	0.1149	IoU	0.8354	18m 54s
-----					
Epoch 69/70					
100.0 %	loss	0.1162	IoU	0.8319	19m 11s
-----					
Epoch 70/70					
100.0 %	loss	0.1111	IoU	0.8395	19m 27s

+ Code

+ Markdown

Epoch 1/70					
24.2 %	loss	0.1129	IoU	0.8834	0m 4s
Epoch 2/70					
24.2 %	loss	0.0852	IoU	0.9126	0m 8s
Epoch 3/70					
24.2 %	loss	0.0786	IoU	0.9178	0m 12s
Epoch 4/70					
24.2 %	loss	0.0743	IoU	0.9226	0m 16s
Epoch 5/70					
24.2 %	loss	0.0739	IoU	0.9225	0m 20s
Epoch 6/70					
24.2 %	loss	0.0673	IoU	0.9287	0m 24s
Epoch 7/70					
24.2 %	loss	0.0665	IoU	0.9290	0m 28s
Epoch 8/70					
24.2 %	loss	0.0696	IoU	0.9245	0m 32s
Epoch 9/70					
24.2 %	loss	0.0655	IoU	0.9312	0m 36s
Epoch 10/70					
24.2 %	loss	0.0622	IoU	0.9332	0m 40s
Epoch 11/70					
24.2 %	loss	0.0633	IoU	0.9317	0m 44s
Epoch 12/70					
24.2 %	loss	0.0616	IoU	0.9333	0m 48s

同时，观察两轮训练的最末与起始状态，可以发现，在数据经处理后的训练过程中，IOU出现了明显的提升。该过程可以解释为，第二次对于“较好的数据”再次进行了训练，也就是在训练数据的分布上，为“好数据”增加了一些权重，使得那些有利于分割的图像细节得到了放大，因此，数据在这些图像细节上进行推理时将更容易标注出推理结果。

### 3、调参问题

在本次实验中，adam的相关参数、epoch数、batchsize等超参数仍然具有一定的优化空间，鉴于时间的问题，调参过程我将在完成本次大作业后继续进行尝试。

## 五、总结与展望

本次实验利用Unet网络。成功训练出了盐体分割的模型，利用Unet网络和自蒸馏算法对于模型进行训练，并初步解释了自蒸馏筛选过程的有效性。然而，本次实验过程仍然具有较大实验潜力，后续具有较大实践空间。

### 5.1 模型潜力

本次实验选择了Unet模型，然而，目前语义分割模型中存在分割更为精准、训练速度更快的模型（如deeplab系列、HRnet、Linknet等），在本次实验之前，我利用Linknet对于该任务进行了简单的预实验，却很失望的发现模型存在无法收敛的情况，结合后续所讨论的数据集问题，可能因为脏数据数量较多影响了训练过程。在后续可以通过对数据集进一步筛选从而得到其他语义分割网络的模型

## 5.2 数据筛选潜力

由于实验时间较短，我仅仅将脏数据进行“一刀切”处理，然而，以那些没有识别出盐体的标签为例，这些标签可能有一部分确实是因为没有盐体存在，而这类数据对于模型学习过程是具有一定用处的。同时，也需要添加更多的对比实验对于数据筛选方法的可靠性进行证实。由于此比赛归根到底还是一项比赛，以指标作为唯一参照，因此指标的提升在一定程度上存在可解释性。同时在第四章**自蒸馏算法的解释**中进行了解释。

## 5.3 参数调节潜力

针对上文所提到的调参问题，仍然可以调整参数进行比赛指标的优化。

参考文献：

Self-Distilled StyleGAN: Towards Generation from Internet Photos 借鉴了其中自蒸馏的训练思想

U-Net: Convolutional Networks for Biomedical Image Segmentation 借鉴了Unet的网络结构

<https://www.kaggle.com/jesperdramsch/intro-to-seismic-salt-and-how-to-geophysics> 针对比赛背景和baseline有了一定了解