

REPORT

기초인공지능
전기전자공학부
2017142136 이관희



연세대학교
YONSEI UNIVERSITY

1. 주제

의료 영상 segmentation의 한 예로 dcm 파일과 mat파일을 이용하여 학습과정부터 제작하여 lumbar과 spine을 class별로 segmentation을 진행한다.

2. neural network 없이

먼저 데이터를 분석해보자.



우리가 추출하고자 하는 데이터는 위의 왼쪽을 input으로 넣어 오른쪽 predict 결과이다. (원래는 L1,L2 등 등뼈의 이름도 분리해야 하지만 딥러닝 없이 뼈의 분리하기 위해서는 등뼈의 L part 말고 T part도 있어서 뼈의 네모난 모양을 추출하고 위에서 차례로 순서를 매기는 법은 불가능하다. 사용할 수 있는 factor는 각 뼈가 이루고 있는 각도 요소인데 이것도 뼈의 사진마다 혹은 디스크의 유무 또는 정도마다 이루는 각도가 달라서 그냥 구분없이 데이터를 뽑고자 하였다.) 딥러닝 없이 segmentation을 하기 위해서는 선택지가 많지 않다. 그 예로 조사 결과 thresholding, region growing, graph cut, active contour model, active shape model 이 있는데 내가 해본 건 오직 filtering과 thresholding 뿐이다. thresholding의 경우에는 일정 이상의 값만 추출해야 하는데 thresholding을 적용하는 것은 근처 비슷한 값을 가지고 있는 pixel값이 많아(다른 뼈 사진) 정확히 lumbar만 파악하는 값을 찾는 것이 불가능했다. 일단 해본 결과를 첨부해보자면 아래와 같다. 먼저 min max 값을 파악해야 한다. 그 후 적당한 값을 찾아보자.

```
>>> data.max()
255
>>> data.min()
0
```

코드 구현은 간단하다.

```
## 디버깅 없이
data = np.load('./datasets/test/input_000.npy')
data[data<=50] = 0
plt.imshow(data, cmap=plt.cm.bone)
```

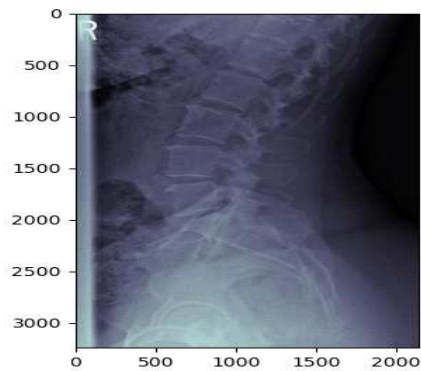


그림 9 원본

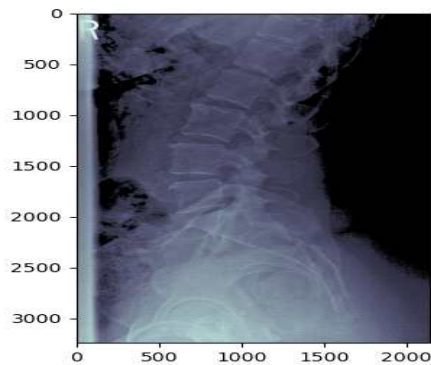


그림 10 threshold 50일때

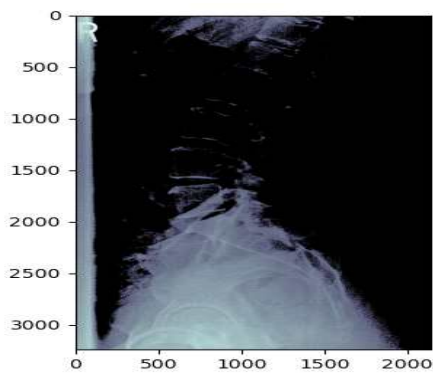
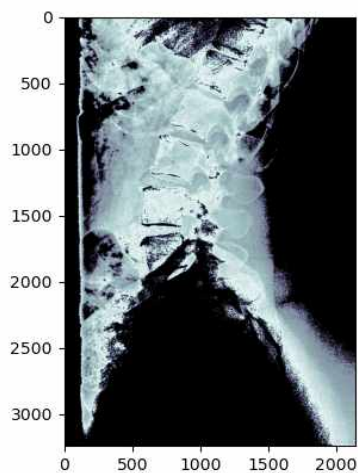


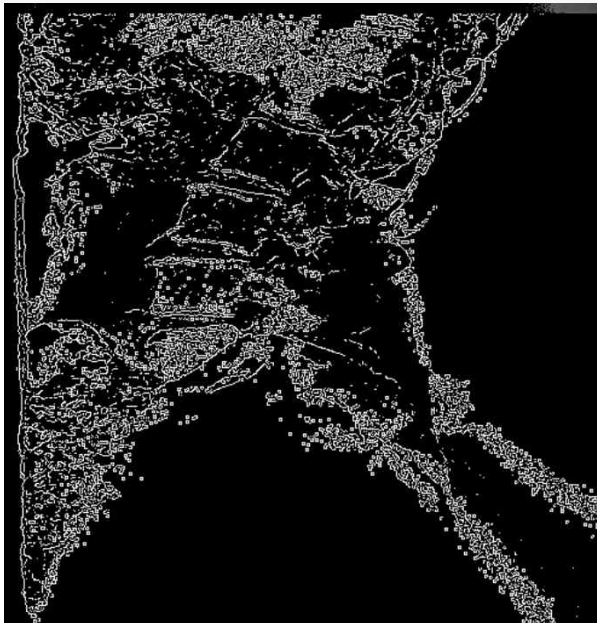
그림 11 threshold 100 일 때

이 결과를 보면 50과 100사이에 값이 lumbar로 의미 있다는 것을 알 수 있다.

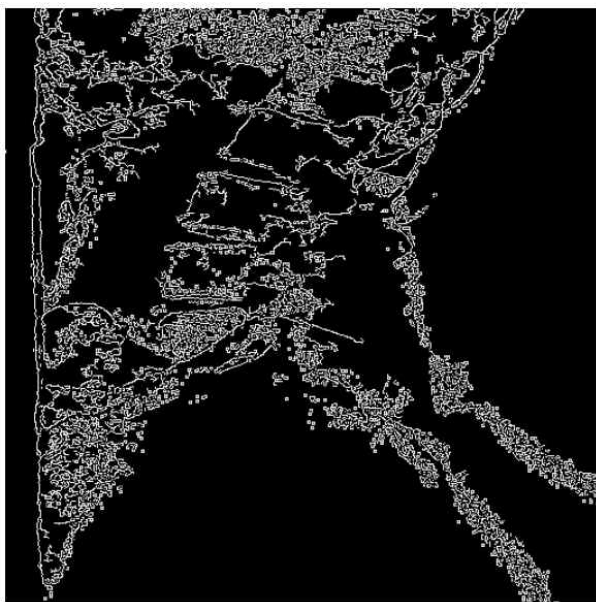


이 그림을 보면 어느정도로 lumbar이라고 파악되는 몇 개의 부분이 출력된 것을 확인할 수 있다. 이게 이걸 Canny edge filter에 통과 시켜서 결과를 파악해보자.

```
1 import numpy as np
2 import cv2
3 from google.colab.patches import cv2_imshow
4
5 ## 덤러닝 없이
6 img = np.load('/content/drive/Othercomputers/notebook/lumbar/datasets/test/input_000.npy')
7 img[img<=50] = 0
8 img[img>=100] = 0
9 #plt.imshow(img, cmap=plt.cm.bone)
10
11 ##
12 import cv2
13 img = cv2.resize(img, (512, 512))
14 edge1 = cv2.Canny(img,55,60)
15 cv2_imshow(img)
16 cv2_imshow(edge1)
17
18 cv2.waitKey(0)
19
```

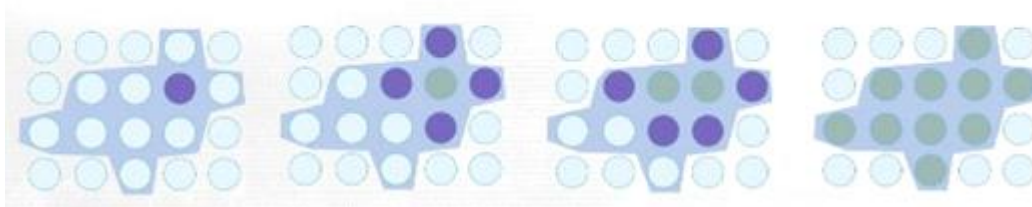


어느 정도 파악범위를 줄였다고 했지만 noise가 너무 많아 구분하기 힘들다. min max값 을 바꿔주면 noise를 아래처럼 조금 제거할 수 있긴 있는데 이 값이 hyper parameter이라서 적절한 filter 값을 직접구하기가 어렵다.



이렇게 해서 만약 네모난 뼈 모양만 얻을 수 있으면 내부 한 점을 기준으로 edge까지 region을 칠해줘야 하는데 이 때 region growing 기법을 사용하면 된다.

Region Growing



이 기법은 edge 안의 영역까지 색칠하는 방법을 말하는 것이다.

이외에도 실제 구현은 하지 못했지만 convolution layer을 통과시키면 filter가 네모난 형태와 비슷하기 때문에 네모난 모양에 값이 있는 lumbar에 위치에 조금 더 큰 값이 추출되는 것을 이용하는 방법도 있다.

딥러닝 없이 segmentation을 하는 방법 분석해보자면 다른 연구 결과들을 찾아봤을 때 deep learning을 사용한 결과보다는 정확도가 떨어지기는 하지만 의미있는 결과를 얻을 수 있긴 하다. 이런 deep learning없이 구현해본 결과 가장 중요한 것은 어떻게든 lumbar의 특징(feature)을 찾아 그것을 뽑아내는 작업을 해야한다. label 데이터를 학습 시키는 것이 아니기 때문에 사람이 직접 이 구분되는 feature를 찾아야하고 이것을 뽑아낼 수 있어야만 의미있는 결과를 얻을 수 있다. 위에 내가 사용하고자한 feature는 edge이다. lumbar의 특징이 네모나니 edge에서 네모난 것을 어떻게든 뽑아내서 이 결과를 이용하고자 하였다. 하지만 정확도를 올리기 위해서는 다른 feature들도 고려해야 한다. 다른 feature의 예시로는 lumbar의 번호는 위에서부터 순서대로 매겨진다는 점 혹은 이루는 각도가 차이가 난다는 것 등이 있을 수 있다. 하지만 위에도 볼 수 있듯이 noise가 관여하기에 정확한 데이터를 뽑는데 한계가 있다. 또한 실제 모든 feature들을 상호작용할 수 있게 구현해야 하기 때문에 코드의 양이 늘어날 뿐만 아니라 어렵다. 만약 딥러닝을 사용하게 되면 label 데이터를 보고

컴퓨터 자동적으로 feature을 파악하므로 lumbar을 구분하는 feature을 정확히 몰라도 좋은 결과를 얻을 수 있는 편리함이 있다. 물론 대표적인 featrue로 전처리할 수 있으면 더 좋은 결과를 얻을 수 있다.

3. neural network로 구현한 결과

먼저 코드 설명을 하자면 아래와 같다.

data_read_dcm.py -> dcm 파일을 읽어서 model에 들어갈 수 있게 npy파일로 바꾸는 역할

data_read_mat.py -> mat 파일을 읽어서 model에 label로 들어갈 수 있게 npy파일로 바꾸는 역할

dataset.py -> 데이터 로더 구현과 transform 구현한 파일

image.py -> 파일이 잘 만들어졌는지 확인하는 파일

model.py -> unet이 구현되어 있음.

resize.ipynb -> resize를 위해 연습한 것(필요없음)

results_data_name.py -> 만들어진 data는 output_000 폴로 되어 있는데 이를 처음 준 데이터 이름과 일치하게 바꾸어주는 파일

run.ipynb -> colab에서 돌리기 위해 만든 파일

train.py -> train시와 test 시의 과정이 저장되어 있음

util.py -> cheakpoint를 저장하고 불러오는 함수가 저장되어 있음(model weight)

train과 test 시 사용되는 코드

train -> data_read_dcm.py data_read_mat.py dataset.py image.py model.py run.ipynb
train.py util.py

test -> data_read_dcm.py dataset.py model.py results_data_name.py run.ipynb
train.py util.py

3.1 데이터 분석

먼저 살펴볼 것은 data_read_dcm.py와 data_read_mat.py 파일이다. 순서대로 dcm file의 raw 데이터만 읽어서 내가 사용할 numpy file로 새로 저장하거나 matlab 파일을 numpy file로 바꾸는 기능을 하고 있다. dcm 파일의 특징을 살펴보자면 이미지 파일 뿐만 아니라 이미지에 해당하는 정보가 포함되어있다. width나 height, depth(dimension)는 물론 환자의 정보가 포함되어 있는데 우리가 필요한 건 이미지를 나타내는 raw 정보뿐이다. 그러므로 사용하기 편리한 형태로 바꾸어줄 필요가 있다. dcm file을 본 김에 data_read_dcm.py 먼저 살펴보자.

```
1 ## 필요한 패키지 등록
2 import os
3 import numpy as np
4 import matplotlib.pyplot as plt
5 import SimpleITK as sitk
6 import glob # glob는 파일들의 리스트를 뽑을 때 사용하는데, 파일의 경로명을 이용해서 입맛대로 요리할 수 있습니다.
7 import re
```

위와 같이 필요한 패키지를 등록하고


```

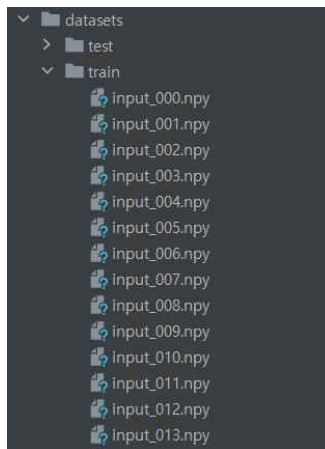
9  ▶  ## 데이터 불러오기 및 train set 저장 구분 train : val = 110 : 10
10  data_list = glob.glob('./train/img/*')
11
12  train_list= data_list[0:110]
13  dir_data = './datasets'
14  dir_save_train = os.path.join(dir_data, 'train')
15
16  if not os.path.exists(dir_save_train):
17      os.makedirs(dir_save_train)
18
19  for i,filename in enumerate(train_list):
20      images = sitk.ReadImage(filename)
21      images_array=sitk.GetArrayFromImage(images).astype('float32')
22      img=np.squeeze(images_array)
23      copy_img=img.copy()
24      min=np.min(copy_img)
25      max=np.max(copy_img)
26
27      copy_img1=copy_img-np.min(copy_img)
28      copy_img=copy_img1/np.max(copy_img1)
29      copy_img*=2**8-1
30      copy_img=copy_img.astype(np.uint8)
31
32  np.save(os.path.join(dir_save_train, 'input_%03d.npy' % i), copy_img)

```

위의 코드처럼 데이터를 불러와서 dcm 파일을 읽어야 한다. 이때 sitk을 사용했는데 이 패키지가 나한테 더 사용하기 용이했기 때문이다. 여기서 보면 단순히 readimage만을 사용하여 image 영역을 얻을 수 있다. 이렇게 얻은 데이터는 먼저 연산을 하기 위해 float32 데이터로 바꾼다. 여기서 dcm 파일의 특징은 uint16으로 되어 있기 때문에 그냥 uint8 형 변환(일반 이미지를 나타내는 pixel은 8bit를 사용한다.)을 사용하면 데이터 손실이 발생할 수 있다. 그래서 위와 같이 변환해줘야만 한다. 이 방식은 읽어온 dicom 파일을 0~1 사이 값으로 normalization해준 다음 255를 곱해주고 uint8로 바꾸어서 데이터 손실을 최대한 줄이는 것이 목적이다.

$$z = \frac{x - \min(x)}{\max(x) - \min(x)}$$

이런 방식을 사용해서 train dataset을 만들었다.



이걸 validation에도 적용하여 10개의 파일을 만든다.(처음에는 일반적으로 train과 val 비율이 8:2라는게 좋다는 이야기를 듣고 100:20비율로 했으나 정확도를 조금이라도 올리기 위해서 조금더 학습 데이터가 많으면 좋을 것 같아서 110:10으로 바꾸었다.)

Tissue	CT Number (HU)
Bone	+1000
Liver	40 - 60
Whiter mater	-20 to -30
Grey mater	-37 to -45
Blood	40
Muscle	10 - 40
Kidney	30
CSF	15
Water	0
Fat	-50 to -100
Air	-1000

추가적으로 조사한 것에 따르면 ct 촬영 값은 물체에 따라서 다르다.(위 그래프를 보면 전 처리가 필요한 이유가 있다. 범위가 -값과 255을 넘어가는 값이 있기 때문이다.) 즉 이 말은 bone중에서 특정한 값만을 설정하면 내가 원하는 영역을 볼 수 있다 그러나 이 방법을 사용해서 lumbar을 뽑으려고 ITK-SNAP toolkit을 설치해서 영역을 변경했지만 lumbar이 없어지는 값과 다른 뼈 들이 없어지는 값이 비슷하여 의미 있는 결과를 얻지 못해서 사용하지 않았다.

다음은 matlab 파일을 numpy로 바꾸는 방법이다.


```

▶ ## 필요한 패키지 등록
import os
import numpy as np
import matplotlib.pyplot as plt
import scipy.io # matlab 파일 읽을 때
import glob #모든 glob는 파일들의 리스트를 뽑을 때 사용하는데, 파일의 경로명을 이용해서 입맛대로 요리할 수 있습니다.

▶ ## label 불러오기 및 train set 저장 구문
label_list = glob.glob('./train/label/*')

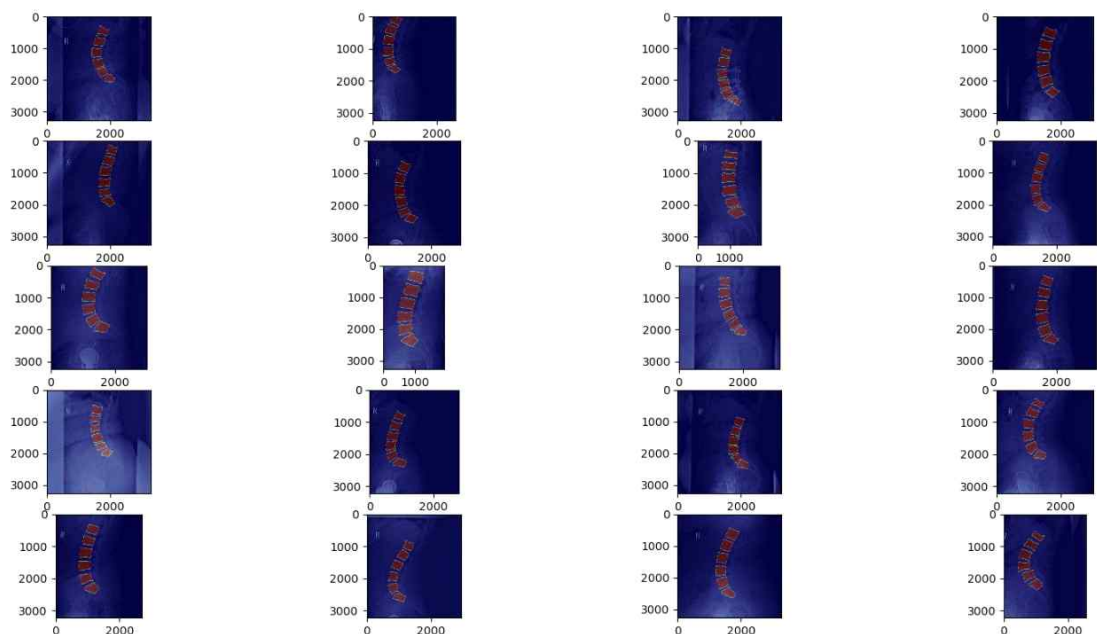
train_list = label_list[0:110]
dir_data = './datasets'
dir_save_train = os.path.join(dir_data, 'train')

if not os.path.exists(dir_save_train):
    os.makedirs(dir_save_train)

for i, filename in enumerate(train_list):
    mat_file = scipy.io.loadmat(filename)
    mat_file_value = mat_file['label_separated']
    np.save(os.path.join(dir_save_train, 'label_%03d.npy' % i), mat_file_value)

```

사용한 패키지는 scipy.io로 matlab 파일을 읽을 때 용이하다. matlab 파일을 읽을 때 pixel label만 읽지 않고 다른 정보들도 일부 일어난다. 여기서 label부분만 보기 위해서는 `mat_file_value = mat_file['label_separated']`이것처럼 사용하여 읽어주면된다. 결국 이 파트에서 중요한 것은 우리가 필요한 것은 img 데이터에 해당하는 pixel값만이라 적절히 분리하여 전처리 하는 것이 중요하다. label 데이터는 0 1로만 되어 있어야 하지만 7번 label(배경 label)이 일부 0과 255로 이루어져 있다는 것을 발견했다.(BCE loss를 사용한 결과 값이 -가 나왔다) 그래서 이 값을 바꾸어 주는 작업도 진행했다. for문으로 0과 1만으로 구성되지 않은 파일을 찾아서 데이터의 max와 min을 파악하여 0 1로만 되도록 바꾸어주었다. 이 결과 얻은 numpy파일을 input data와 label data를 overlay 시키면 아래와 같다. 학습시킬 데이터를 파악하기 위해서 모든 데이터를 살펴보았는데 단순히 R(오른쪽에서 찍은 영상) 사진만 있었고 각기 size가 다르다는 것을 확인할 수 있었다. (resize가 필요함)



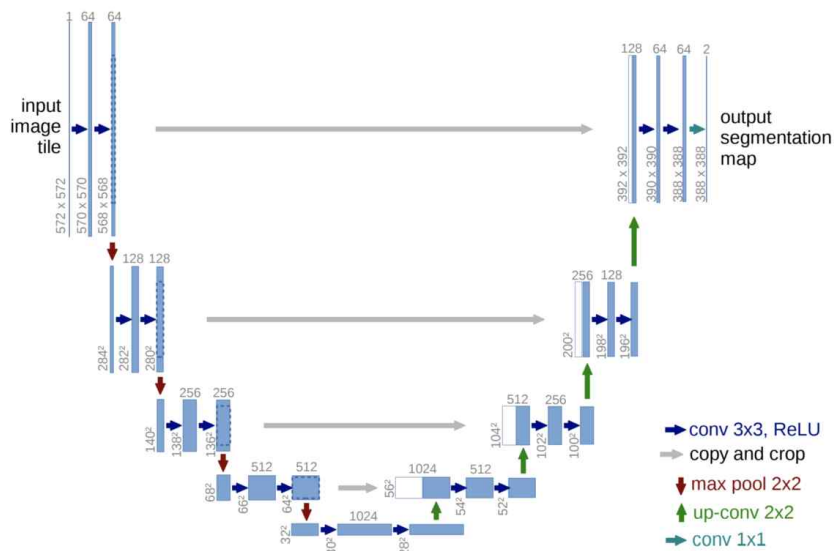
3.2 모델 (model.py)

모델은 캐글 사이트에서 현재 진행하는 챌린지에 사용하는 model들 중 가장 정확도 높은 것으로 사용하려고 했지만 계속 발전되고 있는 deeplab의 경우 일반 영상에서는 많이 사용되지만 의료 영상의 경우 정확도가 unet에 비해 크게 다르지 않아서 unet을 사용하기로 결정했다. 다른 RCNN에 다른 모델을 결합한 모델도 있었지만 모델이 이해가 되지 않아서 수업시간에 배운 unet으로 최종 결정한 것이다. 처음에는 unet의 parameter(30M정도이다.)이 크게 문제되지 않을 것이라고 생각했는데 학습과정에서 gpu의 성능의 한계를 보고 다양한 unet의 parameter을 줄인 모델도 찾아보았다.(squeeze unet) 그러나 parameter이 줄면 gpu 사용에서 out 오류가 나지 않지만 정확도가 확연히 떨어져서 우리가 해야하는 task에는 정확도가 생명이라 사용하지 않았다.

Model Architecture	#Params	Model Size (MB)	Kernel Size
Squeeze U-Net	2.59M	32	3×3 2×2 1×1
U-Net	30M	386	3×3 2×2 1×1
SegNet	30M	117	3×3
Deep Lab	20M	83	3×3 – 1×1
FCN -8s	132M	539	3×3–1×1– 4×4 7×7–16×16
DeconvNet	143M	877	1×1 – 3×3 7×7

Table 7: Comparison of Squeeze U-Net (1) and U-Net (2) regarding intersection over union (IoU) and false positive to true positive for test set

	Building		Tree		Sky		Car		Road		Average	
	(1)	(2)	(1)	(2)	(1)	(2)	(1)	(2)	(1)	(2)	(1)	(2)
Percent True positive pixels												
Average	78.5%	83.3%	51.1%	72.6%	93.7%	96.9%	71.1%	84.5%	95.6%	97.4%	78.0%	86.9%
Max	86.3%	90.1%	73.8%	92.4%	99.4%	99.4%	99.3%	97.8%	98.2%	99.2%	91.4%	95.8%
Min	58.8%	73.3%	18.0%	51.7%	76.4%	91.1%	46.0%	49.0%	90.0%	93.0%	57.8%	71.6%
Intersection over Union (IoU)												
Average	0.689	0.639	0.370	0.403	0.842	0.879	0.427	0.628	0.751	0.800	0.616	0.670
Max	0.808	0.734	0.575	0.650	0.928	0.945	0.667	0.812	0.913	0.937	0.778	0.816
Min	0.395	0.443	0.113	0.220	0.676	0.753	0.011	0.020	0.575	0.686	0.354	0.424
False Positive pixels relative to true positive pixels												
Average	22.1%	39.3%	106.3%	130.1%	12.5%	11.0%	688.2%	354.1%	31.3%	23.4%	172.1%	111.6%
Max	83.1%	89.4%	331.0%	280.4%	24.8%	23.1%	8870.8%	4752.1%	62.8%	39.4%	1874.5%	1036.9%
Min	6.0%	23.4%	18.6%	20.1%	6.3%	4.5%	26.2%	10.7%	7.7%	5.6%	13.0%	12.8%



unet 모델의 모양은 위와 같다. 이것은 contracting path(아래와 가는 path) 와 expansive path(위로 가는 path)을 나누어 설계하였다.

```
## 네트워크 구축하기
class UNet(nn.Module):
    def __init__(self):
        super(UNet, self).__init__()

        def CBR2d(in_channels, out_channels, kernel_size=3, stride=1, padding=1, bias=True):
            layers = []
            layers += [nn.Conv2d(in_channels=in_channels, out_channels=out_channels,
                                kernel_size=kernel_size, stride=stride, padding=padding,
                                bias=bias)]
            layers += [nn.BatchNorm2d(num_features=out_channels)]
            layers += [nn.ReLU()]

            cbr = nn.Sequential(*layers)

            return cbr

        # Contracting path
        self.enc1_1 = CBR2d(in_channels=1, out_channels=64)
        self.enc1_2 = CBR2d(in_channels=64, out_channels=64)

        self.pool1 = nn.MaxPool2d(kernel_size=2)

        self.enc2_1 = CBR2d(in_channels=64, out_channels=128)
        self.enc2_2 = CBR2d(in_channels=128, out_channels=128)

        self.pool2 = nn.MaxPool2d(kernel_size=2)

        self.enc3_1 = CBR2d(in_channels=128, out_channels=256)
        self.enc3_2 = CBR2d(in_channels=256, out_channels=256)
```

여기서 CBR2d는 unet의 오른쪽으로 가는 반복구조를 나타내는 것이다.

```

35
36     self.pool3 = nn.MaxPool2d(kernel_size=2)
37
38     self.enc4_1 = CBR2d(in_channels=256, out_channels=512)
39     self.enc4_2 = CBR2d(in_channels=512, out_channels=512)
40
41     self.pool4 = nn.MaxPool2d(kernel_size=2)
42
43     self.enc5_1 = CBR2d(in_channels=512, out_channels=1024)
44
45     # Expansive path
46     self.dec5_1 = CBR2d(in_channels=1024, out_channels=512)
47
48     self.unpool4 = nn.ConvTranspose2d(in_channels=512, out_channels=512,
49                                       kernel_size=2, stride=2, padding=0, bias=True)
50
51     self.dec4_2 = CBR2d(in_channels=2 * 512, out_channels=512)
52     self.dec4_1 = CBR2d(in_channels=512, out_channels=256)
53
54     self.unpool3 = nn.ConvTranspose2d(in_channels=256, out_channels=256,
55                                       kernel_size=2, stride=2, padding=0, bias=True)
56
57     self.dec3_2 = CBR2d(in_channels=2 * 256, out_channels=256)
58     self.dec3_1 = CBR2d(in_channels=256, out_channels=128)
59
60     self.unpool2 = nn.ConvTranspose2d(in_channels=128, out_channels=128,
61                                       kernel_size=2, stride=2, padding=0, bias=True)
62
63     self.dec2_2 = CBR2d(in_channels=2 * 128, out_channels=128)
64     self.dec2_1 = CBR2d(in_channels=128, out_channels=64)
65
66     self.unpool1 = nn.ConvTranspose2d(in_channels=64, out_channels=64,
67                                       kernel_size=2, stride=2, padding=0, bias=True)
68
69     self.dec1_2 = CBR2d(in_channels=2 * 64, out_channels=64)
70     self.dec1_1 = CBR2d(in_channels=64, out_channels=64)
71
72     self.fc = nn.Conv2d(in_channels=64, out_channels=7, kernel_size=1, stride=1, padding=0, bias=True)
73
74     def forward(self, x): # x is input
75         enc1_1 = self.enc1_1(x)
76         enc1_2 = self.enc1_2(enc1_1)
77         pool1 = self.pool1(enc1_2)
78
79         enc2_1 = self.enc2_1(pool1)
80         enc2_2 = self.enc2_2(enc2_1)
81         pool2 = self.pool2(enc2_2)
82
83         enc3_1 = self.enc3_1(pool2)
84         enc3_2 = self.enc3_2(enc3_1)
85         pool3 = self.pool3(enc3_2)
86
87         enc4_1 = self.enc4_1(pool3)
88         enc4_2 = self.enc4_2(enc4_1)

```

```

89         pool4 = self.pool4(enc4_2)
90
91         enc5_1 = self.enc5_1(pool4)
92
93         dec5_1 = self.dec5_1(enc5_1)
94
95         unpool4 = self.unpool4(dec5_1)
96         cat4 = torch.cat((enc4_2, unpool4), dim=1)
97         dec4_2 = self.dec4_2(cat4)
98         dec4_1 = self.dec4_1(dec4_2)
99
100        unpool3 = self.unpool3(dec4_1)
101        cat3 = torch.cat((enc3_2, unpool3), dim=1)
102        dec3_2 = self.dec3_2(cat3)
103        dec3_1 = self.dec3_1(dec3_2)
104
105        unpool2 = self.unpool2(dec3_1)
106        cat2 = torch.cat((enc2_2, unpool2), dim=1)
107        dec2_2 = self.dec2_2(cat2)
108        dec2_1 = self.dec2_1(dec2_2)
109
110        unpool1 = self.unpool1(dec2_1)
111        cat1 = torch.cat((enc1_2, unpool1), dim=1)
112        dec1_2 = self.dec1_2(cat1)
113        dec1_1 = self.dec1_1(dec1_2)
114
115        x = self.fc(dec1_1)
116
117        return x

```

굉장히 길지만 사실 단순 반복형태이다. 코드의 설명은 수업시간에 배운 그대로를 펼친모양이므로 생략한다.

3.3 데이터 셋 (dataset.py)

여기서는 데이터 로더를 구현하고 제한된 데이터의 data augment와 overfitting을 막기 위해 일종의 regression을 첨가한다. 우리에게 주어진 dataset에서 test는 label이 없기 때문에 test와 val과는 다른 방식으로 정의해야한다. (그냥 label만 없애주면 된다.)

```

10 class Dataset(torch.utils.data.Dataset):
11     def __init__(self, data_dir, transform=None):
12         self.data_dir = data_dir
13         self.transform = transform
14
15         lst_data = os.listdir(self.data_dir)
16
17         lst_label = [f for f in lst_data if f.startswith('label')]
18         lst_input = [f for f in lst_data if f.startswith('input')]
19         lst_label.sort()
20         lst_input.sort()
21         self.lst_label = lst_label
22         self.lst_input = lst_input
23
24     def __len__(self):
25         return len(self.lst_input)
26
27     def __getitem__(self, index):
28         label = np.load(os.path.join(self.data_dir, self.lst_label[index]))
29         input = np.load(os.path.join(self.data_dir, self.lst_input[index]))
30
31         #0~1사이로 normalize
32         label = label
33         input = input/255.0
34
35         if label.ndim == 2:
36             label = label[:, :, np.newaxis]
37         if input.ndim == 2:
38             input = input[:, :, np.newaxis]
39
40         data = {'input': input, 'label': label}
41
42         if self.transform:
43             data = self.transform(data)
44
45         return data

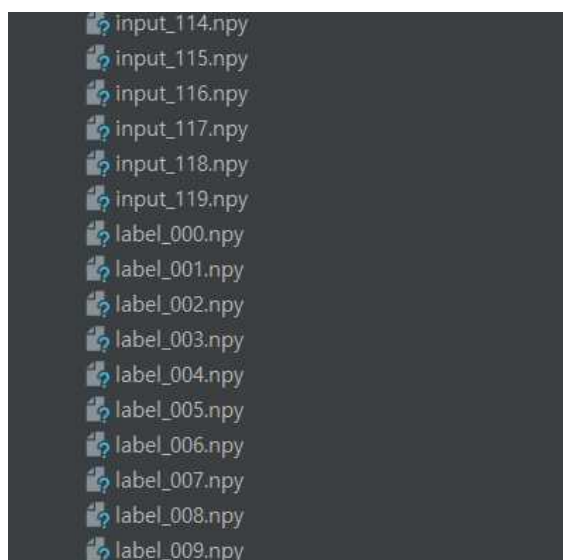
```

데이터로더 구현은 __init__에서 상속받고자 하는 데이터 __len__에서 dataset의 길이 __getitem__에서 내가 사용하고자 하는 데이터의 모양을 정의하면 된다. 우리는 dcm 파일과 label 파일을 분리해서 저장해둔상태이다. 그래서 이를 모아줄 필요성이 있다. 그래서 같은 directory의 input과 label의 파일을 같은 순서로 가져와야 된다. label의 값은 이미 위에서 모두 0과 1로만 만들어주었기 때문에 0과 1사이로 normalize된 반면 input은 uint8 데이터 이므로 255까지 값을 가지고 있어 normalize가 필요하다. if문에 추가된건 나중에 tensor로

변환할 때 총 3개의 요소가 있어야하는데 만약 2개의 요소만 있음(높이와 넓이) 오류가 나서 이를 방지하고자 넣은 요소이다.

```
47 # test dataset
48 class Dataset_test(torch.utils.data.Dataset):
49     def __init__(self, data_dir, transform=None):
50         self.data_dir = data_dir
51         self.transform = transform
52
53         lst_data = os.listdir(self.data_dir)
54
55         lst_input = [f for f in lst_data if f.startswith('input')]
56         lst_input.sort()
57         self.lst_input = lst_input
58
59     def __len__(self):
60         return len(self.lst_input)
61
62     def __getitem__(self, index):
63         input = np.load(os.path.join(self.data_dir, self.lst_input[index]))
64
65         #0~1사이로 normalize
66         input = input/255.0
67
68         if input.ndim == 2:
69             input = input[:, :, np.newaxis]
70
71         data = {'input': input}
72
73         if self.transform:
74             data = self.transform(data)
75
76         return data
```

test dataset의 경우 label이 없기 때문에 같은 데이터를 사용하면 transform 과정에서 오류가 생긴다. 그래서 따로 구현해주었다.



```
input_114.npy
input_115.npy
input_116.npy
input_117.npy
input_118.npy
input_119.npy
label_000.npy
label_001.npy
label_002.npy
label_003.npy
label_004.npy
label_005.npy
label_006.npy
label_007.npy
label_008.npy
label_009.npy
```

지금 train 데이터 파일 안에는 위의 그림의 형태가 되어있다. 넘버링이 같은 것 끼리 같은 데이터를 가리킨다. 즉 input_000.npy와 label_000.npy와 match되어야 한다. 그래서 sort를

이용해 순서를 일치시키고 각각의 데이터를 mapping 시켜주었다.

transform의 경우 단순히 하나의 이미지 file이라면 pytorch에서 제공하는 transform을 사용해도 되지만 여기서는 dataset을 새로 정의했기 때문에 transform을 내가 구현한 dataset과 일치하게 정의해야한다. 그래서 아래와 같이 구현해주었다.

```
78 ▶ ## 트랜스폼 구현하기
79 class ToTensor(object):
80     def __call__(self, data):
81         label, input = data['label'], data['input']
82
83         input = input.transpose((2, 0, 1)).astype(np.float32)
84         label = label.transpose((2, 0, 1)).astype(np.float32)
85
86         data = {'input': torch.from_numpy(input), 'label': torch.from_numpy(label)}
87
88         return data
89
90 class Normalization(object):
91     def __init__(self, mean=0.5, std=0.5):
92         self.mean = mean
93         self.std = std
94
95     def __call__(self, data):
96         label, input = data['label'], data['input']
97
98         input = (input - self.mean) / self.std
99
100         data = {'input': input, 'label': label}
101
102         return data
```

totensor는 GPU에서 학습을 진행하기 위해서 channel과 width와 height의 순서를 변경하는 것으로 tensor의 순서는 channel x row의 수 x column의 수 이기 때문에 numpy의 형태인 row수 x column 수 x channel 의 순서를 바꾸어주고 torch.from_numpy를 이용해서 tensor로 바꾸어 주면 된다. normalization의 경우에는 학습시 좀더 정형화된 데이터를 만들기 위해서 사용하는데 지금 0~1 값을 가진 input 데이터이므로 mean을 0.5 std=0.5로 정의하고 구현했다.

```
104 class Resize(object):
105     def __init__(self, size=(512, 512)):
106         self.size = size
107     def __call__(self, data):
108         label, input = data['label'], data['input']
109
110         #input=resize(input, self.size)
111         #label=resize(label, self.size)
112         input = cv2.resize(input, dsize=self.size, interpolation=cv2.INTER_CUBIC)[: , :, np.newaxis]
113         label = cv2.resize(label, dsize=self.size, interpolation=cv2.INTER_CUBIC)
114
115         data = {'input': input, 'label': label}
116
117         return data
```

resize의 경우에는 unet을 통과시키려면 같은 데이터 사이즈가 되어야 하므로 반드시 필요한 것이다. 나중에도 설명하겠지만 처음에는 256x256으로 GPU 메모리를 최대한 신경 써주었

으나 어짜피 나중에 다시 원래 사이즈로 resize시켜야 하기 때문에 이 때 불가피한 오차가 발생할 수 있어서 512로 키워주었다. 더 키우면 colab에서 돌릴 때 batch size가 매우 작아져야 하므로 (아니면 gpu out이 뜸) 여기가 키울수 있는 최대치라고 보면된다.

```
119 class RandomFlip(object):
120     def __call__(self, data):
121         label, input = data['label'], data['input']
122
123         if np.random.rand() > 0.7:
124             label = np.fliplr(label)
125             input = np.fliplr(input)
126
127         elif np.random.rand() > 0.5:
128             label = np.flipud(label)
129             input = np.flipud(input)
130
131         data = {'label': label, 'input': input}
132
133         return data
```

```
135 class RandomRotation(object):
136     def __call__(self, data):
137         label, input = data['label'], data['input']
138
139         if np.random.rand() > 0.7:
140             label = np.rot90(label, k=1, axes=(0, 1))
141             input = np.rot90(input, k=1)
142
143         elif np.random.rand() > 0.5:
144             label = np.rot90(label, k=3, axes=(0, 1))
145             input = np.rot90(input, k=3)
146
147         data = {'label': label, 'input': input}
148
149         return data
```

데이터의 overfitting도 방지할 겸 final test dataset이 뭔지 몰라 filp과 rotation을 구현했다. 아쉬운점은 실제 final data에서는 filp과 rotation 된 데이터는 없어서 이런 데이터도 구분할 수 있다는 것을 못 보여준 것 같아서 아쉬움이 남는다. 아쉬운 점은 데이터의 shift도 구현했으면 결과가 좋지 않았을까 생각이 든다는 점이다. shift도 구현하기는 사실 어렵지 않았지만 shift 구현시 끝쪽에 몰린 데이터가 잘릴 것 같아서 구현하다가 지웠다. 지금 생각해보면 if

문을 잘 설계했으면 끝쪽에 있는 lumbar 데이터도 잘리기 않았을거고 조금 더 결과가 좋지 않았을까 생각이 된다.

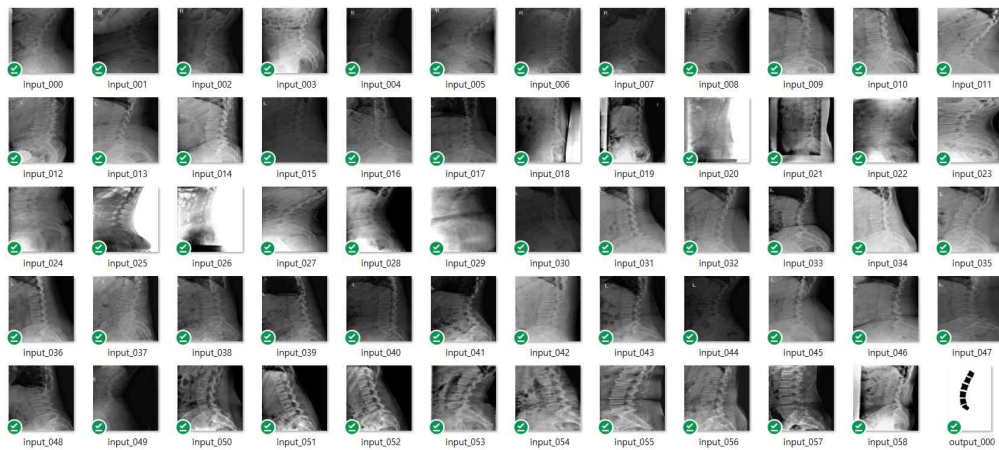


그림 39 final dataset

이제는 test dataset의 transform을 구현해보자. 단지 차이는 label만 없고 test에서는 rotation과 flip이 필요없다.

```
151 ▶ ## test 트랜스폼 구현하기 label 이 없음
152 class ToTensor_test(object):
153     def __call__(self, data):
154         input = data['input']
155
156         input = input.transpose((2, 0, 1)).astype(np.float32)
157
158         data = {'input': torch.from_numpy(input)}
159
160         return data
161
162 class Normalization_test(object):
163     def __init__(self, mean=0.5, std=0.5):
164         self.mean = mean
165         self.std = std
166
167     def __call__(self, data):
168         input = data['input']
169
170         input = (input - self.mean) / self.std
171
172         data = {'input': input}
173
174         return data
```

```

176 class Resize_test(object):
177     def __init__(self, size=(512,512)):
178         self.size = size
179     def __call__(self, data):
180         input = data['input']
181
182         input = cv2.resize(input, dsize=self.size, interpolation=cv2.INTER_CUBIC)[: , :, np.newaxis]
183
184         data = {'input': input}
185
186         return data

```

3.4 util 함수 (util.py)

여기서는 checkpoint를 저장하는 방식이 구현되어 있다. 데이터의 양이 적어 transform 된 데이터의 epoch를 많이 돌려야 결과가 좋아지는데 학습의 연속성을 위해 중간 model을 저장하기 위해서 구현하였다.

```

2 import os
3 import torch
4
5 ## 네트워크 저장하기
6 def save(ckpt_dir, net, optim, epoch):
7     if not os.path.exists(ckpt_dir):
8         os.makedirs(ckpt_dir)
9
10    torch.save({'net': net.state_dict(), 'optim': optim.state_dict()},
11              "%s/model_epoch%d.pth" % (ckpt_dir, epoch))
12
13 ## 네트워크 불러오기
14 def load(ckpt_dir, net, optim):
15     if not os.path.exists(ckpt_dir):
16         epoch = 0
17         return net, optim, epoch
18
19     ckpt_lst = os.listdir(ckpt_dir)
20     ckpt_lst.sort(key=lambda f: int(''.join(filter(str.isdigit, f))))
21
22     dict_model = torch.load('%s/%s' % (ckpt_dir, ckpt_lst[-1]))
23
24     net.load_state_dict(dict_model['net'])
25     optim.load_state_dict(dict_model['optim'])
26     epoch = int(ckpt_lst[-1].split('epoch')[1].split('.pth')[0])
27
28     return net, optim, epoch

```

save는 model을 저장하는 것이고 학습을 연속성있게 진행하기 위해 load를 구현한 것이다. 이걸 save에서는 net, optim, epoch를 저장하고 불러올 때는 checkpoint가 저장되어 있는 directory에서 가장 마지막에 저장되어 있는 것(가장 epoch가 많이 돌아 간 것)을 부르기 위해 index -1을 설정하였다. 이를 조절하려면 -1를 수정하거나 사용할 model 빼고는 다

directory에서 지우면 된다.

3.5 train 함수 (train.py)

여기서는 train과 test가 구현되어 있다. 먼저 변수를 file 외부에서도 수정할 수 있도록 parser를 구현하였다. 이게 필요한 이유는 노트북의 gpu 사양이 좋지 않아 colab에서 돌리기 위해서이다.

```
14 ▶ ## Parser 설정하기
15 parser = argparse.ArgumentParser(description="Train the UNet",
16                                 formatter_class=argparse.ArgumentDefaultsHelpFormatter)
17
18 parser.add_argument("--lr", default=1e-3, type=float, dest="lr")
19 parser.add_argument("--batch_size", default=32, type=int, dest="batch_size")
20 parser.add_argument("--num_epoch", default=5, type=int, dest="num_epoch")
21
22 parser.add_argument("--data_dir", default="./datasets", type=str, dest="data_dir")
23 parser.add_argument("--ckpt_dir", default="./checkpoint", type=str, dest="ckpt_dir")
24 parser.add_argument("--log_dir", default="./log", type=str, dest="log_dir")
25 parser.add_argument("--result_dir", default="./result", type=str, dest="result_dir")
26
27 parser.add_argument("--mode", default="train", type=str, dest="mode")
28 parser.add_argument("--train_continue", default="on", type=str, dest="train_continue")
29 parser.add_argument("--port", default=52162)
30
31 args = parser.parse_args()
32
```

```
33 ▶ ## 트레이닝 파라미터 설정하기
34 lr = args.lr
35 batch_size = args.batch_size
36 num_epoch = args.num_epoch
37
38 data_dir = args.data_dir
39 ckpt_dir = args.ckpt_dir
40 log_dir = args.log_dir
41 result_dir = args.result_dir
42
43 mode = args.mode
44 train_continue = args.train_continue
45
46 device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
47
48 print("learning rate: %.4e" % lr)
49 print("batch size: %d" % batch_size)
50 print("number of epoch: %d" % num_epoch)
51 print("data dir: %s" % data_dir)
52 print("ckpt dir: %s" % ckpt_dir)
53 print("log_dir: %s" % log_dir)
54 print("result dir: %s" % result_dir)
55 print("mode: %s" % mode)
56
57 ▶ ## 디렉토리 생성하기
58 if not os.path.exists(result_dir):
59     os.makedirs(os.path.join(result_dir, 'png'))
60     os.makedirs(os.path.join(result_dir, 'numpy'))
61
```



```

62 ▶ ## 네트워크 학습하기
63 if mode == 'train':
64     transform = transforms.Compose([Normalization(mean=0.5, std=0.5), Resize(), RandomFlip(), RandomRotation(), ToTensor()])
65
66     dataset_train = Dataset(data_dir=os.path.join(data_dir, 'train'), transform=transform)
67     loader_train = DataLoader(dataset_train, batch_size=batch_size, shuffle=True, num_workers=8, pin_memory=True)
68
69     dataset_val = Dataset(data_dir=os.path.join(data_dir, 'val'), transform=transform)
70     loader_val = DataLoader(dataset_val, batch_size=batch_size, shuffle=False, num_workers=8, pin_memory=True)
71
72     # 그밖에 부수적인 variables 설정하기
73     num_data_train = len(dataset_train)
74     num_data_val = len(dataset_val)
75
76     num_batch_train = np.ceil(num_data_train / batch_size)
77     num_batch_val = np.ceil(num_data_val / batch_size)
78 else: # test
79     transform = transforms.Compose([Normalization_test(mean=0.5, std=0.5), Resize_test(), ToTensor_test()])
80
81     dataset_test = Dataset_test(data_dir=os.path.join(data_dir, 'test'), transform=transform)
82     loader_test = DataLoader(dataset_test, batch_size=batch_size, shuffle=False, num_workers=8, pin_memory=True)
83
84     # 그밖에 부수적인 variables 설정하기
85     num_data_test = len(dataset_test)
86
87     num_batch_test = np.ceil(num_data_test / batch_size)
88

```

여기서는 위에서 구현한 custom Dataset과 transform을 사용하여 train val test set을 load 한다. 여기서 몇가지를 살펴보자면 num_workers의 경우 GPU의 개수 x4를 사용해야 좋다고 해서 8로 설정하였고 pin_memory=True에 경우에는 GPU를 효율적으로 사용하기 위해서 정의하였다.

```

89 ▶ ## 네트워크 생성하기
90 net = UNet().to(device)
91
92 ▶ ## 손실함수 정의하기
93 fn_loss = nn.BCEWithLogitsLoss().to(device)
94 fn_loss = nn.MultiLabelSoftMarginLoss().to(device) # 다중 클래스 이기 때문에
95 ▶ ## Optimizer 설정하기
96 optim = torch.optim.Adam(net.parameters(), lr=lr)
97
98 ▶ ## 그밖에 부수적인 functions 설정하기 output 저장용 위한 함수
99 fn_tonumpy = lambda x: x.to('cpu').detach().numpy().transpose(0, 2, 3, 1) # from tensor to numpy
100 fn_denorm = lambda x, mean, std: (x * std) + mean #denormalize
101 fn_class = lambda x: 1.0 * (x > 0.5) #classification() using thresholding(p=0.5) 네트워크 output을 binary class로 변환
102
103 ▶ ## Tensorboard 를 사용하기 위한 SummaryWriter 설정
104 writer_train = SummaryWriter(log_dir=os.path.join(log_dir, 'train'))
105 writer_val = SummaryWriter(log_dir=os.path.join(log_dir, 'val'))
106

```

그 외 필요한 것을 선언하였는데 tensorboard를 이용하면 나중에 결과를 보면서 파악하겠지만 loss의 추이를 살펴볼 수 있다. loss의 경우는 multi class일 경우 위의 loss를 많이 사용한다고 해서 가져왔다. dice loss를 사용하지 않은 이유는 의료 영상 segmentation의 경우 dice loss보다 BCEWithLogitsLoss를 사용하는 것이 좋다는 글을 봐서 이걸 사용했다. MutiLabeSoftMarginLoss()는 단순히 BCEWithLogitsLoss의 multi class 버전이라고 한다.

```

107 > ## 네트워크 학습시키기
108 st_epoch = 0
109
110 # TRAIN MODE
111 if mode == 'train':
112     if train_continue == "on":
113         net, optim, st_epoch = load(ckpt_dir=ckpt_dir, net=net, optim=optim)
114
115     for epoch in range(st_epoch + 1, num_epoch + 1):
116         net.train()
117         loss_arr = []
118
119         for batch, data in enumerate(loader_train, 1):
120             # forward pass
121             label = data['label'].to(device=device)
122             input = data['input'].to(device=device, dtype=torch.float32)
123             output = net(input)
124             # backward pass
125             optim.zero_grad()
126
127             loss = fn_loss(output, label)
128             loss.backward()
129
130             optim.step()
131
132             # 손실함수 계산
133             loss_arr += [loss.item()]
134

```

```

135     print("TRAIN: EPOCH %04d / %04d | BATCH %04d / %04d | LOSS %.4f" %
136           (epoch, num_epoch, batch, num_batch_train, hp.mean(loss_arr)))
137
138     # Tensorboard 저장하기
139     # label = fn_tonumpy(label)
140     # input = fn_tonumpy(fn_denorm(input, mean=0.5, std=0.5))
141     # output = fn_tonumpy(fn_class(output))
142     #
143     # writer_train.add_image('label', label[... , 0:1], num_batch_train * (epoch - 1) + batch, dataformats='NHWC')
144     # writer_train.add_image('input', input, num_batch_train * (epoch - 1) + batch, dataformats='NHWC')
145     # writer_train.add_image('output', output[... , 0:1], num_batch_train * (epoch - 1) + batch, dataformats='NHWC')
146     #
147     # writer_train.add_image('label_back', label[... , 6:7], num_batch_train * (epoch - 1) + batch, dataformats='NHWC')
148     # writer_train.add_image('output_back', output[... , 6:7], num_batch_train * (epoch - 1) + batch, dataformats='NHWC')
149
150
151     writer_train.add_scalar('loss', np.mean(loss_arr), epoch)
152
153     with torch.no_grad():
154         net.eval()
155         loss_arr = []
156
157         for batch, data in enumerate(loader_val, 1):
158             # forward pass
159             label = data['label'].to(device=device)
160             input = data['input'].to(device=device, dtype=torch.float32)
161             output = net(input)
162

```

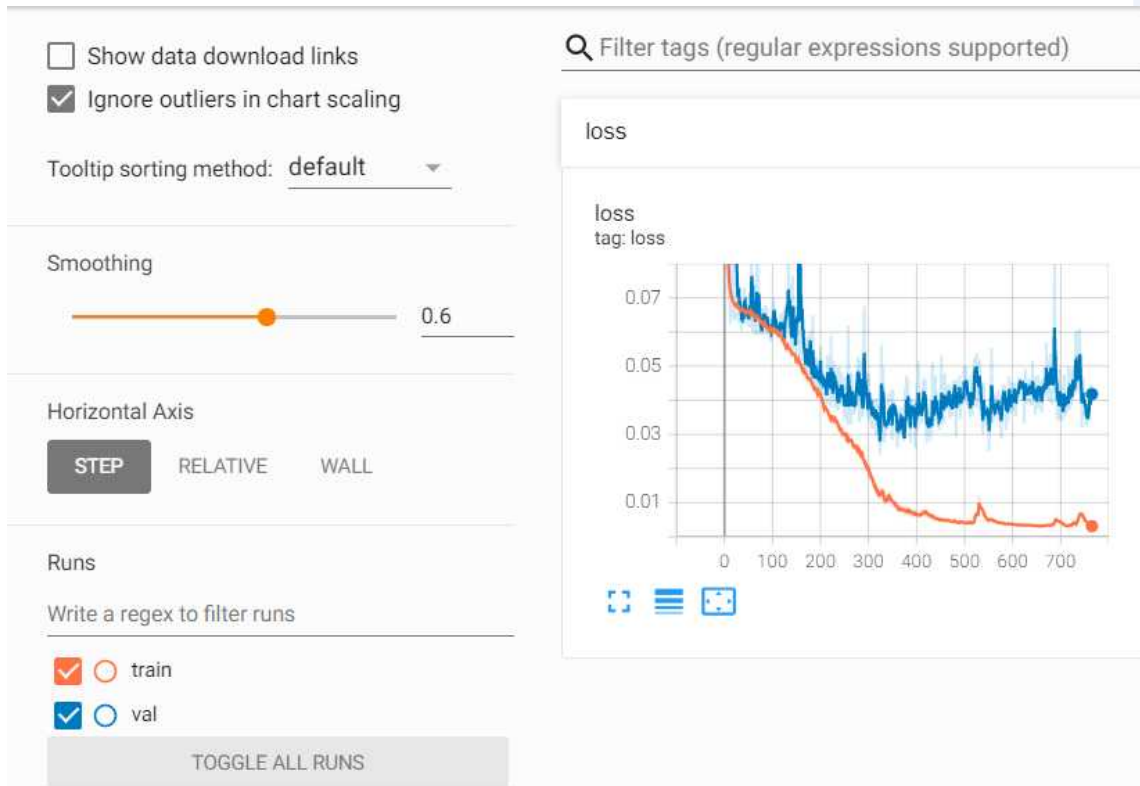


```

163     # 손실함수 계산하기
164     loss = fn_loss(output, label)
165
166     loss_arr += [loss.item()]
167
168     print("VALID: EPOCH %04d / %04d | BATCH %04d / %04d | LOSS %.4f" %
169           (epoch, num_epoch, batch, num_batch_val, np.mean(loss_arr)))
170
171     # Tensorboard 저장하기
172     # label = fn_tonumpy(label)
173     # input = fn_tonumpy(fn_denorm(input, mean=0.5, std=0.5))
174     # output = fn_tonumpy(fn_class(output))
175
176     # writer_val.add_image('label', label[... , 0:1], num_batch_train * (epoch - 1) + batch, dataformats='NHWC')
177     # writer_val.add_image('input', input, num_batch_train * (epoch - 1) + batch, dataformats='NHWC')
178     # writer_val.add_image('output', output[... , 0:1], num_batch_train * (epoch - 1) + batch, dataformats='NHWC')
179     #
180     # writer_val.add_image('label_back', label[... , 6:7], num_batch_train * (epoch - 1) + batch, dataformats='NHWC')
181     # writer_val.add_image('output_back', output[... , 6:7], num_batch_train * (epoch - 1) + batch, dataformats='NHWC')
182
183     writer_val.add_scalar('loss', np.mean(loss_arr), epoch)
184
185     if epoch % 50 == 0:
186         save(ckpt_dir=ckpt_dir, net=net, optim=optim, epoch=epoch)
187
188     writer_train.close()
189     writer_val.close()
190

```

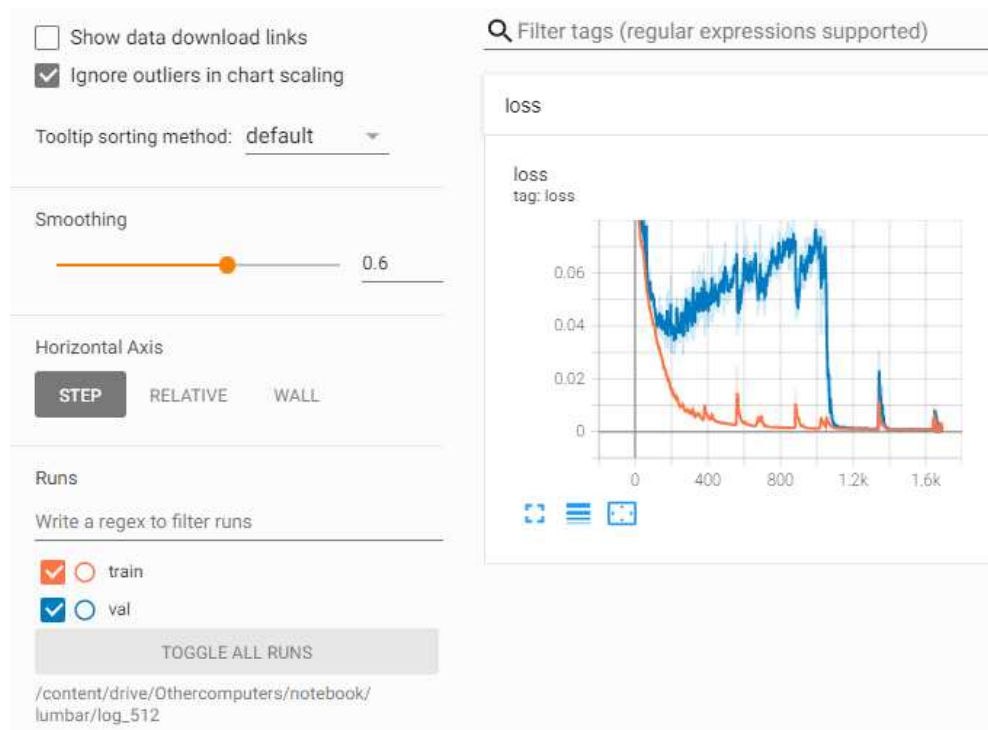
여기까지가 train mode의 전체 코드인데 주석처리 된 것은 이미지를 저장하는 부분으로 학습의 속도 향상을 위해서 빼준 부분다. 그 외에는 epoch 50마다 model을 저장하는거 loss만 tensorboard에 저장하는거만 빼면 실습시간에 했던 내용이다. (train data에서는 backpropagation이 필요하지만 val에서는 필요없다.는 등) 이렇게 얻은 결과를 살펴보자. 먼저 loss값의 변화만 살펴보면 주황색이 train 파란색이 val이다. 이 결과는 resize를 256x256으로 batchsize를 16으로 했을 때의 결과이다. (learning rate는 처음에는 10^{-2} 였다가 학습이 진행이 꽤 된뒤 10^{-3} , 10^{-4} 으로 바꾸어 주었다.)



2	492353	0.92584	0.83149	0.87866	600
2	492353	0.92782	0.83318	0.8805	650
2	492353	0.92726	0.83207	0.87967	700
2	492353	0.9294	0.82299	0.87619	750
3	492353	0.92901	0.8324	0.88071	1025

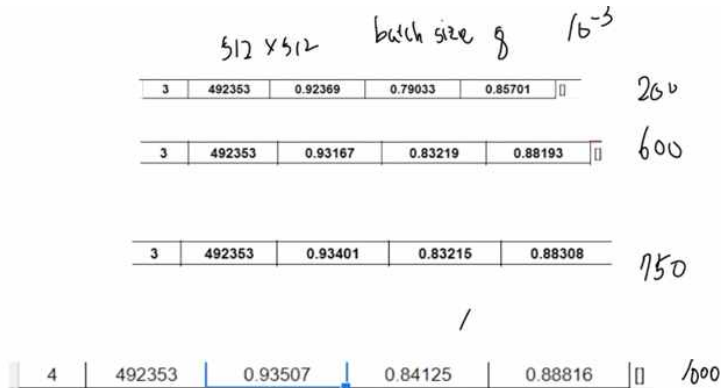
그 결과 연습으로 올릴 수 있는 리더보드의 경우 정확도가 위와 같았다. 1025의 경우 overfitting이 일어나서 그 결과를 삭제하고 750까지만 저장해두었다. 보면 결과가 0.87~0.88사이로 유지되는데 그중 650에서 가장 좋은 결과를 얻었다. 리더보드용 데이터가 20개 뿐이 되지 않아서 결과가 0.01단위로 차이 나는 것은 의미가 없다고 판단되지만 그럼에도 650에서의 모델이 가장 정확도가 높았다.

정확도를 향상시키기 위해서 주어진 모델에서 바꿀 수 있는 parameter을 바꿔보자. 먼저 resize의 단위를 512x512로 바꾸었고 gpu out이 일어나지 않도록 batchsize를 8로 learning rate는 10^{-3} 으로 쥘 유지하였다. 또한 validation set을 10개로 줄이고 train set을 110개로 늘렸다. (원래는 train:val = 100 :20 이었음) 그 결과 아래와 같다 (이 학습 방법을 최종에서 사용하였다.)

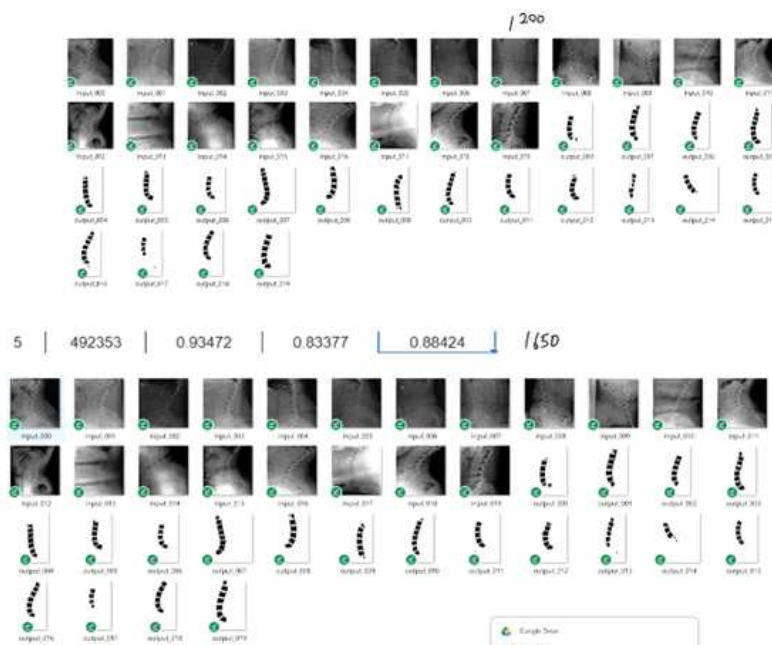


여기서 1000epoch까지만 파란색 val이 의미를 가진다. 왜냐하면 리더보드를 보니 진짜 사소한 차이로 순위가 바뀌는 것 같아서 validation도 모두 train set에 사용하였기 때문이다. 결

과는 아래와 같다.



1650 epoch 에서의 리더보드 결과가 빠져 있는데 그 결과가 아래에 있다. 정확도가 큰 변화는 없어 정석대로 학습시킨 1000epoch 모델을 final에서 사용하였다. 아쉬운점은 data가 작아서 이 결과가 overfitting인지 아닌지 파악할 수 없었다. 오직 알 수 있는건 리더보드의 결과인데 리더보드용 데이터도 적고 값이 드라마틱하게 바뀌지 않아서 단순히 정확도가 조금 낮아져서 overfitting이 발생했다고 판단하기 무리가 있었다.



이건 1200에서의 결과와 1650에서의 결과이다. 1200 epoch의 리더보드 결과는 시간이 촉박해서 돌리지 못했는데 결과를 보면 거의 유사하다. 위의 결과는 7번 label인 배경 라벨만을 때서 본 것이다. 1600epoch의 정확도를 보면 1000epoch보다 0.004정도 낮다. 이게 의미있는 차이는 아닐 것 같다. test 데이터를 뽑는 코드는 아래와 같다.

```

191 # TEST MODE
192 else:
193     net, optim, st_epoch = load(ckpt_dir=ckpt_dir, net=net, optim=optim)
194     print("epoch: %.4e" % st_epoch)
195
196     shape_lst = []
197     dir_save_test = os.path.join(data_dir, 'test')
198     lst_data = os.listdir(dir_save_test)
199
200     for i in range(len(lst_data)):
201         k = np.load(os.path.join(dir_save_test, 'input_%03d.npy' % i))
202         revered_list = list(reversed(k.shape))
203         b = tuple(revered_list)
204         shape_lst.append(b)
205
206     with torch.no_grad():
207         net.eval()
208
209         for batch, data in enumerate(loader_test, 1):
210             # forward pass
211             input = data['input'].to(device)
212             output = net(input)
213
214             print("TEST: BATCH %04d / %04d !" % (batch, num_batch_test))
215
216             input = fn_tonumpy(fn_denorm(input, mean=0.5, std=0.5))
217             output = fn_tonumpy(fn_class(output))
218
219             for j in range(input.shape[0]):
220
221                 id = batch_size * (batch - 1) + j
222
223                 output_new = []
224                 # 이미지 resize
225                 for ii in range(output.shape[-1]):
226                     ex_new = cv2.resize(output[j][..., ii:ii+1], dsize=shape_lst[id], interpolation=cv2.INTER_NEAREST)
227                     output_new.append(ex_new) # list
228                 a = np.array(output_new) # numpy array로 변경
229                 b = a.transpose(2, 0, 1) # 축이 channel x row x column -> row x column x channel
230
231                 plt.imsave(os.path.join(result_dir, 'png', 'input_%03d.png' % id), input[j].squeeze(), cmap='gray')
232                 plt.imsave(os.path.join(result_dir, 'png', 'output_%03d.png' % id), b[...,:6:7].squeeze(), cmap='gray')
233
234                 np.save(os.path.join(result_dir, 'numpy', 'output_%03d.npy' % id), b.astype(np.uint8))

```

test data set은 처리할 것이 하나있다. 그건 input데이터의 사이즈로 output 데이터를 바꾸는 것이다. 그래서 원래 데이터의 size를 저장하고 있는 list가 필요하다. 여기서 각 index는 Dastaset의 getitem의 index와 동일하다. 200~204 줄까지가 input 데이터의 사이즈를 저장하는 부분인데 resize 시킬 때 cv2.resize를 사용한다면 이 list를 그대로 사용하면 column과 row가 반대로 나온다. 그래서 shape을 reversed 시켜주는 것이다. 224에서 226은 이미지를 resize하는 부분이다. 여기서는 interpolation에서 Inter_nearest를 사용했는데 이걸 사용해야 unit8가 유지되어서 이를 사용하였다.

이 코드를 colab에 돌리기 위해서 필요한 코드는

```
1 !python3 '/content/drive/Othercomputers/notebook/lumbar/train.py' #
2 --lr 1e-3 --batch_size 8 --num_epoch 2000 #
3 --data_dir "/content/drive/Othercomputers/notebook/lumbar/datasets" #
4 --ckpt_dir "/content/drive/Othercomputers/notebook/lumbar/cheakpoint_512" #
5 --log_dir "/content/drive/Othercomputers/notebook/lumbar/log_512" #
6 --result_dir "/content/drive/Othercomputers/notebook/lumbar/results" #
7 --mode "train" #
8 --train_continue "on"
```

이거면 충분하다. test일 경우 mode를 test로 바꿔주면 된다.

3.6 결론

실험 결과 아쉬운 점은 filp과 rotation의 transform을 차라리 하지 않았으면 결과가 조금 더 좋지 않았을까하는 점과 그 무엇보다 학습시킬 때 pretrain말고는 한게 없다는 점이다. 물론 이것만으로도 많은 시간이 소모됐지만 사실 이걸 구현하기 이전에 몇가지 관련 논문을 읽었는데 코드 구현이 오픈소스로 공유되지 않아서 사용할 수가 없었다. 그래서 segment의 정확도를 높이기 위해서 lumbar의 꼭짓점을 detection한다거나 하는 창의성 있는 방법을 적용하지 못하고 unet에 transform 몇 개만을 구현해서 결과를 도출한게 아쉽다. 역량이 거기까지 안됐지만 다른 학우의 최종 결과를 보니 그런 lumbar에 적합한 모델을 설계한 것을 같아서 아쉬움이 남는다.

압축 파일참고

log 데이터를 뽑아라하길래 tensorboard에 저장한 log 데이터를 첨부합니다. 또한 이미지 파일도 첨부하고 있고 중간에 왜 val loss가 준 이유는 이 보고서에 설명되어 있습니다.