Final Project Report

20190067 DaeseongKim

CoE202 Final Project

1. Data augmentation1

- 가장 먼저 Data를 전처리 시키기 위하여 Data Augmentation을 진행하였다. pytorch의 CenterCrop, RandomCrop, Resize, RandomHorizontalFlip, RandomRotation 등의 함수를 적절히 조합하여 skeleton code로 주어진 unet모델을 가지고 시험해보았다. 이 때, RandomVerticalFlip은 이미지 인식에서 바다 위 배와 하늘 위 비행기 등을 서로 착각하여 잘못된 학습을 할 수 있다 판단하여 시험 대상에서 제외시켰다.

- Random함수 같은 경우에는 dataset 속 image와 label에 동일한 seed값으로 random 함수들에 적용되어야 하는데, 그렇지 않아, 학습의 정답지여야 하는 label이 문제인 image의 정답이 아니게 되는 상황이 발생하였다. 구글 검색을 통하여 아래 그림1)과 같이 CoE_Dataset 클래스의 getitem 부분에서 시드를 고정해줌으로써, 이 문제를 해결할 수 있었다.

- 그렇게 다양한 종류의 transforms를 조합하여 실험해본 결과, 그림 2)와 같이 RandomCrop과 RandomHorizontal을 적용해주는 것이 가장 효과적이었다. 이때, RandomCrop 같은경우는 사이즈를 (256,256)으로 적용해주었는데, 이보다 작은 사이즈의 data가 존재하여 오류가 나기도 했지만, pad_if_needed=True를 인자로 넘겨줌으로써 해결할 수 있었다.

```
seed = np.random.randint(2147483647)
random.seed(seed)
torch.cuda.manual_seed(seed)
torch.manual_seed(seed)
if self.input_transform is not None:
    image = self.input_transform(image)
random.seed(seed)
torch.cuda.manual_seed(seed)
torch.manual_seed(seed)
if self.target_transform is not None:
    label = self.target_transform(label)
```

그림1) 시드 고정 Code

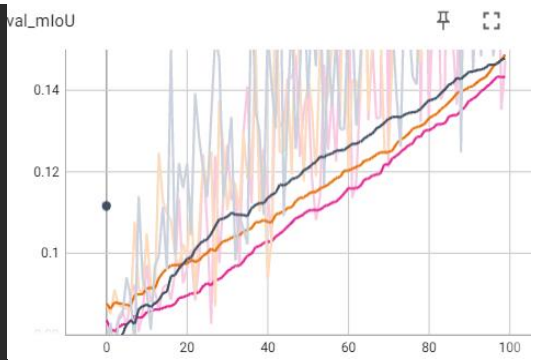


그림2) Black : RancomCrop + Random Horizontal Flip

- Resize나 Rotaition과 달리 Crop의 경우에는 Validation dataset에 적용이 되더라도 큰 영향이 없을 것이라 판단하여, 굳이 dataset을 수동으로 분리할 필요가 없다고 생각하였기에, 이 dataset에 transform을 지정해 준 후, RandomSplit 함수를 이용하여 이 dataset을 subset 2개(train, validation)으로 나누어 주었다. 기대만큼, 큰 향상은 없었고 많은 시도를 해보았지만 성공적인 data augmentation을 해내지 못했다고 판단하였고, 모델을 개선시킨 후, 데이터 개수를 직접 늘리는 방법으로 이를 해결하였다. 이는 보고서 마지막 부분에서 다룰 것이다.

2. Optimizer 선정 및 배치 정규화

- 다음으로 optimizer를 선정했다. 기존의 Adam, 그리고 SGD, RMSprop 세 가지를 테스트해보았는데, 기존의 unet의 성능이 낮아, 큰 차이가 나지 않았다. unet 모델에 각 encoder와

decoder 그리고 center 함수에 배치 정규화를 적용해줌으로써, 성능을 향상시킨 이후, optimizer를 서로 비교해보았다. 나머지가 모두 같은 상황에서 그림3)과 같이 현 모델에 가장 적합한 optimizer는 Adam이었고, 이에 따라 Adam을 optimizer로 그대로 사용해 주었다.

3. L2 Regularization

- L2 regularization을 위하여 Adam optimizer에 weight_decay를 0, 1e-2, 1e-5, 1e-8로 적용해 보았는데, 그림 4)와 같이 0과 1e-5이 val miou가 가장 높았는데 그 둘의 성능에 큰 차이를 보이지는 않았지만 weight_decay를 1e-5로 설정해 주었다. 이는 weight_decay를 1e-5 로 설정한 것이 val loss 부분에서는 현저히 낮았기 때문이다. Weight decay 설정을 다음과 같이 하였다. (optim = torch.optim.Adam(model.parameters(), lr = learning rate, **weight_decay = 1e-5**)

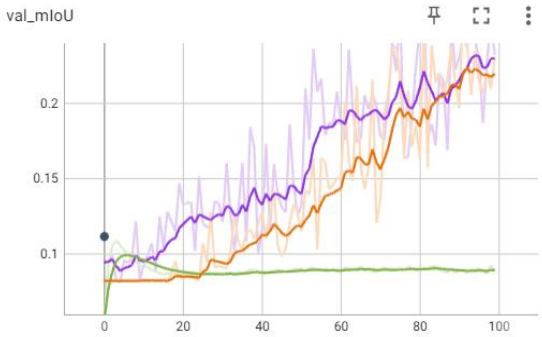


그림3) 보라색 : Adam, 주황색 : RMSprop, 녹색 : SGD

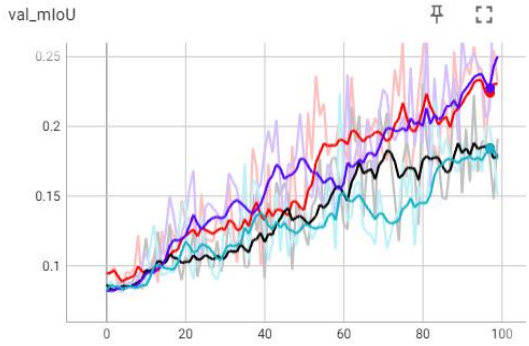


그림 4) 보라색 : 1e-5, 빨간색 : 0 (L2)

4. 모델 수정

- 다음으로는, 모델을 향상시켰다. 기존의 UNet의 구조는 그대로 유지하되, 앞에서 배치정규화를 추가해 주었고, 기존에 존재했던 Dropout은 그대로 유지하였다. encoder와 decoder 수를 2개씩 더 늘려주어 5개로 만들었다. 기존의 UNet 모델이 조금 더 복잡해져야 좋은 성능을 낼 것이라 판단하였기 때문이다. 기존의 3->64->128->256 으로 가던 채널 수를 3->32->64->128->256->512 로 모델을 복잡화 시켰고, 그 결과 그림5)와 같은 성능 향상을 얻을 수 있었다. 그림 7)은 수정된 UNet 구조 코드이다.

- 이후, 층이 깊어져 나타난 degradation 현상을 해결하기 위하여 Encoder와 Decoder의 Double Conv2d 사이에 residual block을 추가해주었다. 이를 위해 UNetEnc와 UNetDec 클래스를 그림8) 과 같이 수정해 주었다. 즉, encoder와 decoder를 한 번 거칠 때마다 residual block을 각각 한 번씩 거치게 되도록 설계하였다. 이 때, residual block에서 x와 F(x)의 channel 수가 달라져 단순 덧셈을 할 수 없는 경우가 발생할 것이라 생각하여, x의 채널 수를 F(x)의 채널 수로 조정해주기 위해 downsample 함수를 1*1 ConV2D로 정의하여

적용해줌으로써 성공적으로 residual block을 구현할 수 있었다.

- 그림 6)과 같이 Encoder에만 추가했을 때에도 큰 성능 향상을 보였고, 이후 Decoder에도 residual block을 추가해주었더니 더 큰 향상을 보였다. center 부분에는 오히려 성능이 저하되는 결과가 나와 추가해주지 않았다.

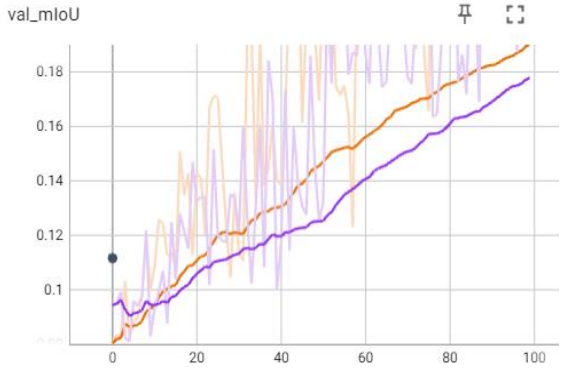


그림 5) 주황색 : 5개 보라색 : 3개 (enc, dec 수)

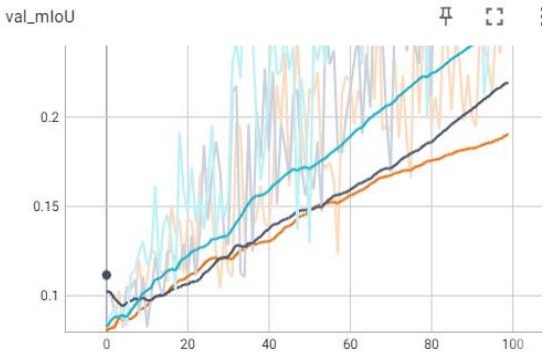


그림 6) 하늘색 : 둘다 추가, 검은색 : enc에만 추가

```
def forward(self, x):
    enc0 = self.enc0(x)
    enc1 = self.enc1(enc0)
    enc2 = self.enc2(enc1)
    enc3 = self.enc3(enc2)
    enc4 = self.enc4(enc3)
    center = self.center(enc4)
    center = self.after_center(center)
    dec4 = self.dec4(torch.cat([center, F.interpolate(enc4, size=center.size()[2:], mode='bilinear')], 1))
    dec3 = self.dec3(torch.cat([dec4, F.interpolate(enc3, size=dec4.size()[2:], mode='bilinear')], 1))
    dec2 = self.dec2(torch.cat([dec3, F.interpolate(enc2, size=dec3.size()[2:], mode='bilinear')], 1))
    dec1 = self.dec1(torch.cat([dec2, F.interpolate(enc1, size=dec2.size()[2:], mode='bilinear')], 1))
    dec0 = self.dec0(torch.cat([dec1, F.interpolate(enc0, size=dec1.size()[2:], mode='bilinear')], 1))

    return F.interpolate(self.final(dec0), size=x.size()[2:], mode='bilinear')
```

그림 7) 수정 후 UNet 구조 (enc, dec 2개씩 추가)

```
    if in_channels != out_channels:
        conv = nn.Conv2d(in_channels, out_channels, kernel_size=1, stride=1, bias=False)
        bn = nn.BatchNorm2d(out_channels)
        self.downsample = nn.Sequential(conv, bn)
    else:
        self.downsample = nn.Sequential()

def forward(self, x):
    out = self.down(x)
    out = out + self.downsample(x)
    out = self.relu(out)
    out = self.maxpool_dropout(out)
    return out
```

그림 8) Residual block 구현 코드 (in encoder)

5. 실패한 시도들

1) Atrous Convolution : 다양한 Convolution 기법들을 찾아보던 중, Atrous Convolution이라는 기법을 알게 되었다. Convolution을 진행할 때, 중간에 구멍을 두고 Convolution을 진행하는 기법인데, 사람이 이미지를 인식할 때 주변의 여러 점들과의 해당 픽셀의 상호 관계를

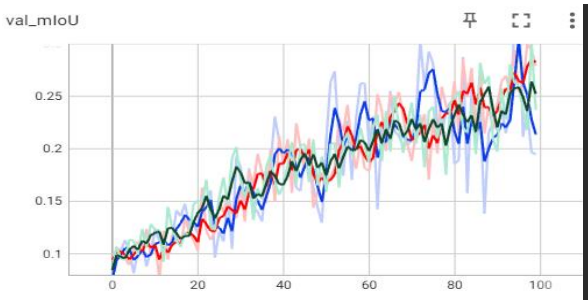
기반으로 그 해당 픽셀을 분류해 내는 것을 생각해보면 이 Atrous Convolution 기법이 semantic segmentation에 상당한 효과가 있을 것이라 생각하였다.

이는 nn.conv2d의 padding과 dilation을 함께 변화시킴으로써 수행할 수 있었다. 기존의 encoder의 double Conv에서 1(default)으로 설정 되어있던 dilation 값을 여러 변화를 줘가며 실험을 해보았다. Encoder class에 dilation 인자를 만들어 주고, 각각의 encoder의 dilation을 순서대로 1, 1, 1, 2, 2(그림10)로 설정해 둘 때 가장 좋은 결과를 낼 수 있었는데, 이것은 기존의 모두 1로 설정할 때와 크게 차이가 없었다. 그래서 뒤에 나오는 새로운 Data augmentation을 적용한 뒤 기존의 모두 1로 설정한 ConV와 이 1,1,1,2,2로 dilation을 설정한 ConV를 비교해보았고, 그 결과 기존의 Convolution에 비해 큰 차이로 성능이 안좋아진 것을 확인할 수 있었다. 따라서, 기존의 Convolution을 그대로 사용하였다. 그림 9)에서 알 수 있듯, ASPP와 함께 적용해도 성능에 큰 변화는 없었다.

2) Atrous Spatial Pyramid Pooling(ASPP) : 이는 1)의 Astrous Convolution과 비슷하게 dilation의 변화를 주어가며 여러 개의 layer를 만들고 이들을 병합하고 다시 원래 channel로 1*1 ConV 시키는 Pooling 기법이다. 이 또한, 1)과 마찬가지의 이유로 semantic segmentation을 다룰 때, 좋은 기법이라 생각하여 그림 11)과 같은 코드로 시험해보았다. 기존의 Center 대신해 이를 적용해 주었는데, 성능은 그대로였다. Dilation의 rate들을 (4,8,12), (6,12,18), (12,24,36) 등 다양하게 시험해보았지만 모두 성능 향상은 없었고 오히려 저하되는 경우도 많았다. 이 Aspp를 적용하기 위하여 다음과 같은 시도를 해보았다.

- 기존의 UNet 구조의 Center를 Aspp로 대신하였다. 하지만 성능이 오히려 저하되었다.
- Decoder를 없애고, Center대신 Aspp를 거친 후, 중간에 enc2를 가져와 torch cat을 한 후, decoder를 대신하는 역할을 구현해주었지만, 이전보단 향상이 있었지만 unet의 decoder보다는 성능이 저하되었다.
- 다시 decoder를 복원하고, Encoder마다 maxpooling 대신 이 Aspp를 적용해보기도 하였는데, 이 또한 실패하였다.

➔ Astrous Convolution과 ASPP를 시도해보았지만, 결국 성능이 같거나 저하되었고, 기존의 UNet 모델을 그대로 사용하게 되었다.




```
self.enc0 = UNetEnc(3, 32, dilation = 1)
self.enc1 = UNetEnc(32, 64, dilation = 1)
self.enc2 = UNetEnc(64, 128, dilation = 1)
self.enc3 = UNetEnc(128, 256, dilation = 2)
self.enc4 = UNetEnc(256, 512, dilation = 2, dropout=True)
```

그림 9) 파란색 : 기존 모델, 빨간색 : ASPP + Dilation(1,1,1,2,2), 검은색 : Dilation(1,1,1,2,2), 그림 10) Dilation 추가

```python
class aspp(nn.Module):
    def __init__(self, in_channel, out_channel,dilation=1):
        super(aspp, self).__init__()
        self.avgpool = nn.AdaptiveAvgPool2d(1)
        self.conv = nn.Conv2d(in_channel, out_channel, 1, 1)
        self.conv_1_1 = nn.Conv2d(in_channel, out_channel, 1, 1)
        self.conv_3_1 = nn.Conv2d(in_channel, out_channel, 3, 1, padding=dilation*1, dilation=dilation*1)
        self.conv_3_2 = nn.Conv2d(in_channel, out_channel, 3, 1, padding=dilation*2, dilation=dilation*2)
        self.conv_3_3 = nn.Conv2d(in_channel, out_channel, 3, 1, padding=dilation*3, dilation=dilation*3)
        self.conv_1_2 = nn.Conv2d(in_channel, out_channel, 1, 1)
        self.BatchNorm = nn.BatchNorm2d(out_channel)
        self.conv_output = nn.Conv2d(out_channel * 5, out_channel, 1, 1)
        self.relu = nn.ReLU()

    def forward(self, x):
        size = x.shape[2:]
        out1 = self.conv_1_1(x)
        out1 = self.BatchNorm(out1)
        out1 = self.relu(out1)
        out2 = self.conv_3_1(x)
        out2 = self.BatchNorm(out2)
        out2 = self.relu(out2)
        out3 = self.conv_3_2(x)
        out3 = self.BatchNorm(out3)
        out3 = self.relu(out3)
        out4 = self.conv_3_3(x)
        out4 = self.BatchNorm(out4)
        out4 = self.relu(out4)
        out5 = self.avgpool(x)
        out5 = self.conv_1_2(out5)
        out5 = self.BatchNorm(out5)
        out5 = self.relu(out5)
        out5 = F.interpolate(out5, size=size, mode="bilinear")
        out = torch.cat([out1, out2, out3,out4,out5],1)
        out = self.conv_output(out)
        return out
```

그림 11) ASPP class 코드 (in model_baseline.py , 분량 관계로 크기를 줄였습니다.)

6. Data Augmentation2

- 기존의 1의 Data augmentation에서 성능 향상이 기대만큼 나오지 않아, 이 transform 작업이 잘 안되었다고 판단하였다. 그래서 직접 1000개의 이미지를 변형하여 데이터 수 자체를 늘리자는 생각을 하게 되었다. 1000개의 기존 이미지와 라벨의 pair를 임의로 900개와 100개로 나누어 주고(수동) 이들을 각각 train, valid라는 폴더로 업로드하여 dataset을 따로 불러왔다. train set같은 경우에는 세 번 불러왔는데, 각각 transform을 아래와 같이 다르게 함으로써, 데이터를 증폭시켜 2700개의 train data를 만들었다. validation set은 수동으로 나눈 100개의 데이터 이미지와 라벨 pair를 불러왔는데, 이 때 학습 과정에서 validation data의 크기가 모두 같지 않으면 오류가 생겨, 원래는 transform을 적용하면 안되지만, 단순 RandomCrop transform을 적용하여 size만 모두 통일 시켜주었다. 처음에는 기존의 dataset을 나누지 않고 1000개의 변형 data set들 3개를 병합해서 3000개의 데이터를 만든 후, train과 validation set으로 RandomSplit하였는데, 이는 validation set에 train set의 변형 이미지가 들어가 잘못된 validation loss와 miou를 야기할 가능성이 크다고 판단하여 수동으로 나누어 주었다. 그 결과 최종 test miou는 **0.5~0.6**의 값을 얻을 수 있었다..

```python
custom_transform1 = [transforms.Resize((300,400))]
custom_transform2 = [transforms.RandomCrop((300,400), pad_if_needed=True),]
custom_transform3 = [transforms.Resize((320,480)),transforms.RandomHorizontalFlip(p=0.8),transforms.CenterCrop((300,400)),]
```

그림 12) 최종 train 데이터 augmentation에 적용한 transform.

```python
dataset1 = CoE_Dataset(root=f"{filepath}/train",
                       input_transform=input_transform1,
                       target_transform=target_transform1)
dataset2 = CoE_Dataset(root=f"{filepath}/train",
                       input_transform=input_transform2,
                       target_transform=target_transform2)
dataset3 = CoE_Dataset(root=f"{filepath}/train",
                       input_transform=input_transform3,
                       target_transform=target_transform3)
train_dataset = torch.utils.data.ConcatDataset([dataset1,dataset2,dataset3])
valid_dataset = CoE_Dataset(root=f"{filepath}/valid",
                       input_transform=valid_input_transform,
                       target_transform=valid_target_transform)
```

그림 13) data 수동으로 나누어 불러온 코드

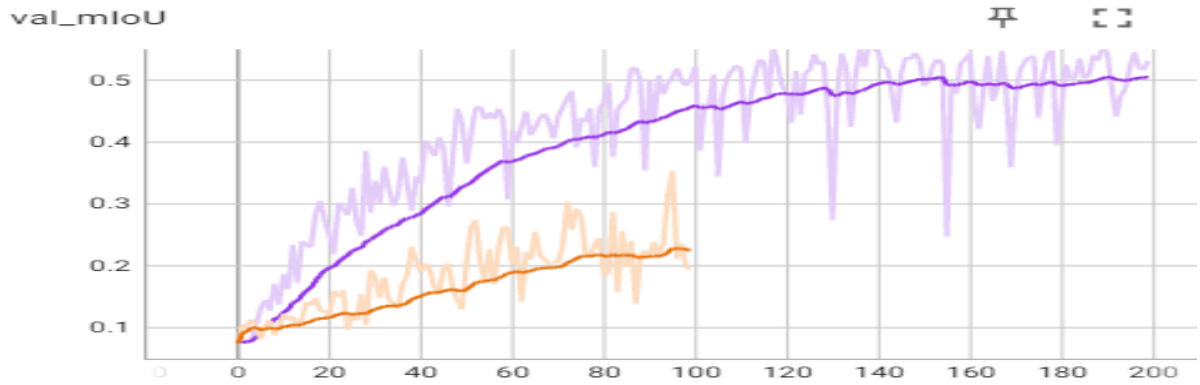


그림 14) 최종 결과 : 데이터 augmentation2의 효과 (보라색 : 2버전(2700개로 학습), 주황색 : 1버전(900개로 학습))

7. Hyperparameter 조정 과 learning scheduler.

1) Epoch : 그림 14)를 보면 학습 횟수 140 부근 이후로는 더 이상 val miou가 향상되지 않음을 알 수 있었다. 그래서 최종 모델을 학습시킬 때 epoch를 200으로 설정하였다. **(Colab Pro를 사용하였는데도, 학습 시간이 8시간 이상 걸려, 더 이상 늘릴 수 없었다.)**

2) Batch size : 실험 결과 배치 사이즈가 커질수록 더 빠르게 성능이 향상됨을 알 수 있었고, 최종 결과에는 큰 영향은 없었다. 하지만 너무 16보다 크게 하니 CUDA 메모리 문제가 생겨 16로 설정하였다.

3) Crop_size : 크롭 사이즈는 dataset을 보니 대부분 사진들의 크기가 300 * 500 정도라 판단하였고, 이와 비슷한 비율로 crop 해주기 위하여 따로 변수를 사용하지 않고 직접 숫자를 써주었다. (i.e. RandomCrop((300,500)))

4) Learning rate : 기존의 1e-3과 1e-4, 1e-2, 5*1e-3 등을 시험해보았지만 기존의 1e-3이 가장 성능이 좋았다. 1e-2와 5*1e-3 같은 경우에는 오히려 성능이 많이 저하되었고, 1e-4 같은 경우에는 성능 향상이 너무 느려 1e-3 정도의 사이즈가 적당하다고 판단하였고, 이후 최종 버전 모델을 학습시킬 때, 1.3 * 1e-3 로 설정한 결과 성능 향상을 볼 수 있었다.

5) Learning Scheduler : torch.optim.lr_scheduler.StepLR을 이용하여 25step마다 다양한 gamma의 변화와 초기 learning rate 변화를 주며 실험해본 결과 그림 15)와 같이 그 어떤 실험에서도 성능이 향상되지 않았고, 이를 사용하지 않은 것이 가장 성능이 좋았기에 사용하지 않았다.

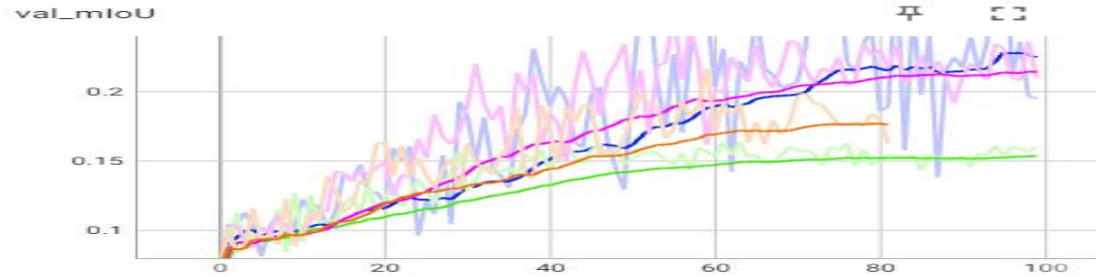


그림 15) 파란색 : learning scheduler X