**Wrap Up Report**

**P – Stage 1**

**Image Classification**

**부스트캠프 AI 테크 P-Stage**

**선재우 T1098**

목차

[**1.** **최종 점수 및 순위** 4](#_Toc68860784)

[**2.** **목표** 4](#_Toc68860785)

[**[BaseLine 작성]** 4](#_Toc68860786)

[**[Data Processing]** 4](#_Toc68860787)

[**[Model]** 4](#_Toc68860788)

[**[Training]** 5](#_Toc68860789)

[**[Deploy]** 5](#_Toc68860790)

[**3.** **기술적인 도전** 6](#_Toc68860791)

[**[Data Processing]** 6](#_Toc68860792)

[1) Face 인식 6](#_Toc68860793)

[2) Generator의 초당 Batch 처리량 측정 8](#_Toc68860794)

[3) 데이터 불균형 해소 : imbalanced-dataset-sampler 9](#_Toc68860795)

[4) Validation data 12](#_Toc68860796)

[5) Data Augmentation 추가하기 12](#_Toc68860797)

[**[Model]** 15](#_Toc68860798)

[1) Pre-trained된 Efficient-Net 사용 15](#_Toc68860799)

[2) 다양한 pre-trained 모델 시도 15](#_Toc68860800)

[**[Training]** 19](#_Toc68860801)

[1) Learning Scheduler 19](#_Toc68860802)

[2) 데이터 불균형 해소 : Focal Loss 20](#_Toc68860803)

[3) nni (AutoML) 실행 22](#_Toc68860804)

[4) 모델 나누기 ( Mask -> Gender -> Age ) 23](#_Toc68860805)

[5) Soft voting앙상블 24](#_Toc68860806)

[**4. 평가** 28](#_Toc68860807)

[**1) 평가 지표 Table** 28](#_Toc68860808)

[**2) NNI로 최적의 Hyperparameter 찾기 + [ Crop / 일반 image ] 비교** 30](#_Toc68860809)

[**3) No Augmentation** 36](#_Toc68860810)

[**5.** **학습과정에서의 교훈** 38](#_Toc68860811)

[**6.** **마주한 한계와 도전숙제** 39](#_Toc68860812)

# **최종 점수 및 순위**





# **목표**

## **[BaseLine 작성] (기간 : 3/29 ~ 3/29)**

## **[Data Processing] - Face Recognition (기간 : 3/30 ~ 3/31) - Cross-validation 사용 -데이터 불균형 해소 / imbalanced Sampler, Focal Loss, OverSampling, Weighted loss (기간 : 4/1 ~ 4/8 ) - Data Augumentation (Affine, Gaussian 등) (기간 : 4/5 ~ 4/8) - Generator의 초당 Batch 처리량 측정 및 향상 (기간 : 3/31 ~ 3/31) - Cutmix 시도 - Repeated Agumentation - validation data 사용 (기간 : 4/1 ~ 4/1)**

## **[Model] - ResNet 152층 시도 - Efficient Net 시도 (기간 : 4/1 ~ 4/2) - YOLO 시도 - Pre-trained 모델에 Fine-tuning 하기 (기간 : 4/1 ~ 4/1) - Model의 초당 Batch 처리량 측정 및 향상 - dm\_nfnet 시도 (기간 : 4/7 ~ 4/7 )**

## **[Training] - 앙상블 시도 (기간 : 4/7 ~ 4/8)** **- Hyperparameter 변경 (기간 : 3/29 ~ 4/8 ) - Learning Schedular 사용 (기간 : 4/1 ~ 4/7 ) - Model의 초당 Batch 처리량 측정 - 좋은 위치에서 Checkpoint 만들기 // Adam으로 모든 minimum 찾고, SGD로 극소점 찾기 (새로운 baseline code) - Sex 분류 (1번 모델)  -> Age 분류 (2번 모델) -> Mask 분류 (3번 모델) // 모델 나누기 (기간 : 4/5 ~ 4/8 ) - Crop image (mask 분류)와 일반 image(age, sex 등 분류) 둘 다 사용 (기간 : 4/5 ~4/5 ) - batch size 작게 쓰면서, SGD 사용 (기간 : 4/8 ~ 4/8 ) - NNI (Auto ML) 사용 (기간 : 4/2 ~ 4/4)**

## **[Deploy] - Python 모듈화 (새로운 baseline code)**

# **기술적인 도전**

## **[Data Processing]**

### 1) Face 인식

haarcascade는 너무 error가 많아서 facenet-pytorch로 도전했습니다.

[github.com/timesler/facenet-pytorch](https://github.com/timesler/facenet-pytorch)

 facenet-pytorch를 사용하자 얼굴 부분만 crop 해서 나오는 것을 확인했습니다.

모든 train data들의 path를 불러옵니다.

imgs\_path = [filepath for filepath in glob.iglob(f'input/data/train/images/\*/\*', recursive=True)]

미리 pretrained 되어있는 vggface2를 가져다가 사용합니다.

LFW 정확도가 0.9965를 자랑한다고 써있습니다.

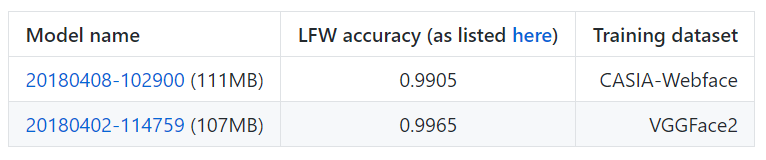


Figure . https://github.com/timesler/facenet-pytorch

Face Recognition의 강자인 MTCNN을 가져온 후 InceptionResnet의 face model을 평가(eval)용으로만 사용해 분류합니다.

*# If required, create a face detection pipeline using MTCNN:*

mtcnn = MTCNN()

resnet = InceptionResnetV1(pretrained='vggface2').eval()

새로운 image folder에다가 crop이미지를 저장합니다.

for imgpath in tqdm(imgs\_path):

try:

imgpath\_split = imgpath.split("/")

img = Image.open(imgpath)

new\_path\_name = os.path.join("input", "data", "train", "crop\_images", imgpath\_split[-2],imgpath\_split[-1])

*# Get cropped and prewhitened image tensor*

img\_cropped = mtcnn(img, save\_path=new\_path\_name)

*# Calculate embedding (unsqueeze to add batch dimension)*

img\_embedding = resnet(img\_cropped.unsqueeze(0))

*# Or, if using for VGGFace2 classification*

resnet.classify = True

img\_probs = resnet(img\_cropped.unsqueeze(0))

except:

print(imgpath+" is error")

총 crop 된 데이터는 18456개이며, 약 400개 정도의 data가 face를 못 찾은 것인지 오류가 발생했습니다.

모든 data는 운 좋게도 160x160x3 크기로 crop 되었습니다.

crop\_imgs\_path = [filepath for filepath in glob.iglob(f'input/data/train/crop\_images/\*/\*', recursive=True)]

>>> print(len(crop\_imgs\_path))

18456

### 2) Generator의 초당 Batch 처리량 측정

Face 인식을 통해 얼굴을 crop 한 후, 160x160x3 크기로 일정하게 사용하기 때문에 transform에서 resize를 할 필요가 사라졌습니다.

**crop 하기 전** batch/s 속도입니다.

start = time.time()

for batch\_in,batch\_out in tqdm(train\_iter):

pass

>>> print("time : ", time.time()-start)

time : 98.58156418800354

즉, 18,456개 / 98초 = **188 batch/s** 입니다.

**crop한 후** batch/s 속도입니다.

start = time.time()

for batch\_in,batch\_out in tqdm(train\_iter):

pass

>>> print("time : ", time.time()-start)

time : 21.3472101688385

즉, 18,456개 / 21초 =**878 batch/s** 입니다.

### 3) 데이터 불균형 해소 : imbalanced-dataset-sampler

데이터 불균형 문제를 해결하기 위해서 찾은 방법은 Focal Loss, Over and under sampling 입니다.

이번에는 Over and under sampling 방법을 섞은 imbalanced-dataset-sampler 방법을 사용했습니다.

원래 Over and under sampling은 데이터를 복사하거나 선별하여 만들지만 문제점이 존재합니다.

Over sampling은 소수 클래스를 복사하기 때문에 과적합이 발생할 수 있습니다.

Under sampling은 다수 클래스에서 선별하기 때문에 정보 손실이 발생합니다.

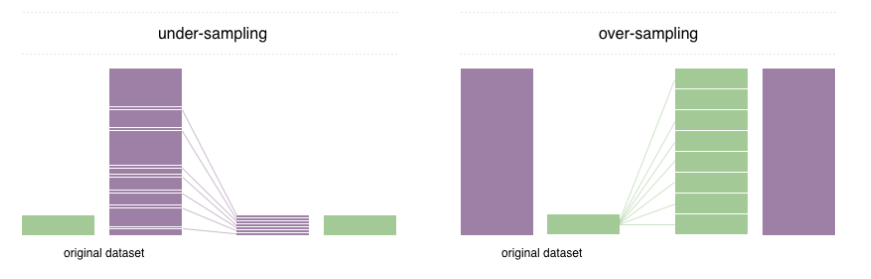


Figure . https://github.com/ufoym/imbalanced-dataset-sampler

imbalanced-dataset-sampler는 다음과 같은 방법으로 수행합니다.

* imbalanced dataset에서 샘플링을 할 때 class의 분포 재조정
* 샘플링 가중치를 자동으로 추정
* avoid creating a new balanced dataset (?)
* Over sampling에서 발생하는 overfitting을 방지

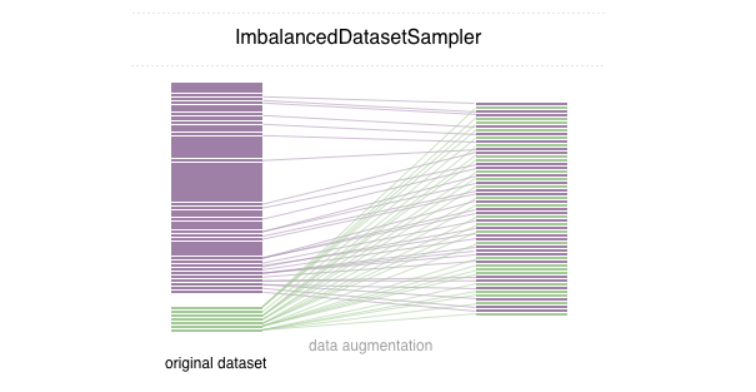


Figure . https://github.com/ufoym/imbalanced-dataset-sampler

적용하기 위해서는 git에서 clone 후 setup.py를 install합니다. (또는 setup.py가 들어있는 폴더에서 pip instll .을 실행합니다.)

!python setup.py install

!pip install .

torchsampler에서 ImbalancedDatasetSampler를 import합니다.

from torchsampler import ImbalancedDatasetSampler

DataLoader에서 sampler=ImbalancedDatasetSampler(train\_dst)를 추가해줍니다.

(여기서 suffle=True를 빼주어야하는데 SubsetRandomSampler가 동작하기 때문이라고 합니다.

ref. [stackoverflow.com/questions/61033726/valueerror-sampler-option-is-mutually-exclusive-with-shuffle-pytorch](https://stackoverflow.com/questions/61033726/valueerror-sampler-option-is-mutually-exclusive-with-shuffle-pytorch))

train\_iter = torch.utils.data.DataLoader(train\_dst, sampler=ImbalancedDatasetSampler(train\_dst), batch\_size=BATCH\_SIZE)

valid\_iter = torch.utils.data.DataLoader(valid\_dst, sampler=ImbalancedDatasetSampler(valid\_dst), batch\_size=BATCH\_SIZE)

이때, 오류가 나서 torchsampler의 imbalanced.py의 \_get\_label() 함수를 약간 수정해주었습니다.

def \_get\_label(self, dataset, idx):

...

elif isinstance(dataset, torch.utils.data.Subset):

return dataset.dataset.imgs[idx][1]

else:

return dataset[idx][1] *# custom dataset이기 때문에 여기로 들어갑니다.*

그냥 DataLoader를 할 때보다 동작하는데 훨씬 오래 걸립니다.

이후 데이터를 찍어보면 800개 후반대로 모두 맞춰집니다. (차후에 사진 첨부)

### 4) Validation data

원래 K-fold Cross-Validation을 사용하려고 했지만, 시간이 오래 걸린다고해서, 간단한 Validation data를 구현했습니다.

모듈은 sklearn을 사용했습니다.

*#validation data*

train\_image, validation\_image, train\_target, validation\_target = train\_test\_split(train\_data\_list, train\_label\_list, test\_size=0.2, shuffle=True, stratify=train\_label\_list)

valid\_dst = TrainDataset(validation\_image, validation\_target, transform)

valid\_iter = torch.utils.data.DataLoader(valid\_dst, sampler=ImbalancedDatasetSampler(valid\_dst), batch\_size=BATCH\_SIZE)

### 5) Data Augmentation 추가하기

Model을 좀 더 Robust하게 만들기 위해서 augmentation에 Affine Augmentation과 Gaussian Augmentation을 추가합니다.

Affine Augmentation을 Class로 만들고, transforms.Compos에 RandomAffine을 추가합니다.

class AffineAugmentation:

def \_\_init\_\_(self, resize, mean, std, \*\*args):

self.transform = transforms.Compose([

transforms.CenterCrop([350, 350]),

Resize(resize, Image.BILINEAR),

transforms.RandomAffine(0, shear=10, scale=(0.8, 1.2)),

ColorJitter(0.1, 0.1, 0.1, 0.1),

ToTensor(),

Normalize(mean=mean, std=std)

])

def \_\_call\_\_(self, image):

return self.transform(image)

Gaussian Augmentation도 transform으로 만들고 Gaussian Noise를 추가해줍니다.

class AddGaussianNoise(object):

"""

transform 에 없는 기능들은 이런식으로 \_\_init\_\_, \_\_call\_\_, \_\_repr\_\_ 부분을

직접 구현하여 사용할 수 있습니다.

"""

def \_\_init\_\_(self, mean=0., std=1.):

self.std = std

self.mean = mean

def \_\_call\_\_(self, tensor):

return tensor + torch.randn(tensor.size()) \* self.std + self.mean

def \_\_repr\_\_(self):

return self.\_\_class\_\_.\_\_name\_\_ + '(mean={0}, std={1})'.format(self.mean, self.std)

class GaussianAugmentation:

def \_\_init\_\_(self, resize, mean, std, \*\*args):

self.transform = transforms.Compose([

transforms.CenterCrop([350, 350]),

Resize(resize, Image.BILINEAR),

ColorJitter(0.1, 0.1, 0.1, 0.1),

ToTensor(),

Normalize(mean=mean, std=std),

AddGaussianNoise()

])

def \_\_call\_\_(self, image):

return self.transform(image)

## **[Model]**

### 1) Pre-trained된 Efficient-Net 사용

현재 (2021년 4월 기준) 가장 좋은 NFNet이 나왔지만 Parameter 수가 어마어마해서 Efficient-net-b6 모델을 사용했습니다.

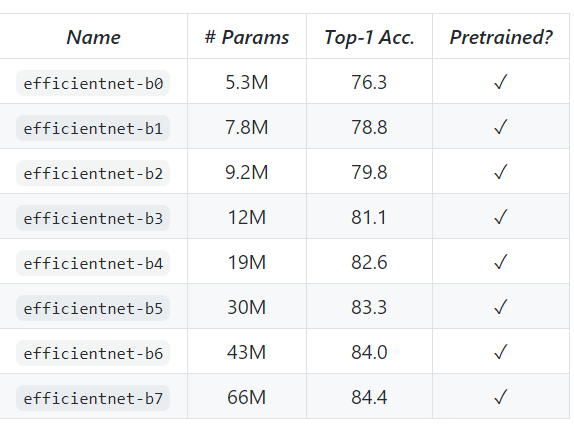
Pre-trained를 불러오는 과정은 정말 간단했습니다.

from efficientnet\_pytorch import EfficientNet

C = EfficientNet.from\_pretrained('efficientnet-b6', num\_classes=18).to(device)

### 2) 다양한 pre-trained 모델 시도

* **Efficientnet\_b6**43M parameter로 확실히 적은 parameter입니다.  
    
  train도 확실히 빨라서 pretrained 사용할 때 기준 1 epoch에 78% Accuracy를 보여주었습니다.



class Efficientnet\_b6(nn.Module):

def \_\_init\_\_(self, num\_classes, pretrained=True):

super().\_\_init\_\_()

if pretrained == True:

self.model = EfficientNet.from\_pretrained('efficientnet-b6', num\_classes=num\_classes)

else:

self.model = EfficientNet.from\_name('efficientnet-b6', num\_classes=num\_classes)

def forward(self, x):

return self.model(x)

* **Efficientnet\_b4**b6보다 가벼운 모델

class Efficientnet\_b4(nn.Module):

def \_\_init\_\_(self, num\_classes, pretrained=True):

super().\_\_init\_\_()

if pretrained == True:

self.model = EfficientNet.from\_pretrained('efficientnet-b4', num\_classes=num\_classes)

else:

self.model = EfficientNet.from\_name('efficientnet-b4', num\_classes=num\_classes)

def forward(self, x):

return self.model(x)

* **ECAresnet50t**  
    
  82.35 top-1 @ 320x320, 81.52 @ 256x256 (Feb 8, 2021)  
    
  사용해보니 확실히 빠르게 Accuracy에 도달했고, 모델도 가벼웠습니다.  
    
  ECA Attention과 함께 여러 ResNet 가중치를 추가한 모델입니다.  
    
  ref) [github.com/rwightman/pytorch-image-models](https://github.com/rwightman/pytorch-image-models)

class ecaresnet50t(nn.Module):

def \_\_init\_\_(self, num\_classes, pretrained=True):

super().\_\_init\_\_()

if pretrained == True:

self.model = timm.create\_model('ecaresnet50t', pretrained=True)

else:

self.model = timm.create\_model('ecaresnet50t', pretrained=False)

self.model.fc = nn.Linear(in\_features=2048, out\_features=num\_classes, bias=True)

def forward(self, x):

return self.model(x)

* **SE - Resnet152d**  
    
  256x256 val, 0.94 crop top-1 - 83.75, 320x320 val, 1.0 crop - 84.36 (Jan 3, 2021)  
    
  Squeeze-and-Excitation Networks  
    
  ref) [arxiv.org/abs/1709.01507](https://arxiv.org/abs/1709.01507)

class seresnet152d(nn.Module):

def \_\_init\_\_(self, num\_classes, pretrained=True):

super().\_\_init\_\_()

if pretrained == True:

self.model = timm.create\_model('seresnet152d', pretrained=True)

else:

self.model = timm.create\_model('seresnet152d', pretrained=False)

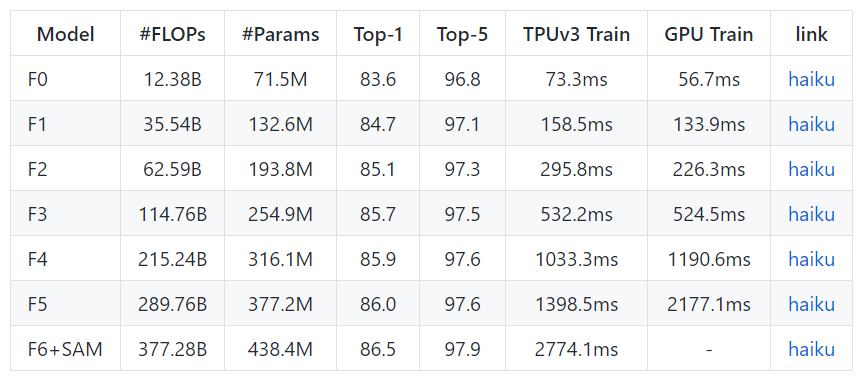
self.model.fc = nn.Linear(in\_features=2048, out\_features=num\_classes, bias=True)

def forward(self, x):

return self.model(x)

* **dm\_nfnet\_f0**

Model의 다양성을 주기 위해서 timm에 pre-trained 되어있는 dm\_nfnet\_0을 사용했습니다.



*# model.py*

class nfnet\_f0(nn.Module):

def \_\_init\_\_(self, num\_classes, pretrained=True):

super().\_\_init\_\_()

if pretrained == True:

self.model = timm.create\_model('dm\_nfnet\_f0', pretrained=True)

else:

self.model = timm.create\_model('dm\_nfnet\_f0', pretrained=False)

self.model.head.fc = nn.Linear(in\_features=3072, out\_features=num\_classes, bias=True)

def forward(self, x):

return self.model(x)

## **[Training]**

### 1) Learning Scheduler

CosineAnnealingLR을 사용했습니다.

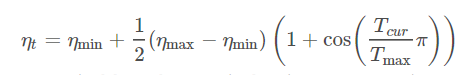


Figure . https://www.kaggle.com/isbhargav/guide-to-pytorch-learning-rate-scheduling

parameter로 Tmax (최대 iteration 횟수)와 eta\_min을 조정하는데 learning rate가 cos함수를 기반으로 eta\_min까지 갔다가 다시 초기 learning rate까지 올라갑니다.

scheduler = optim.lr\_scheduler.CosineAnnealingLR(optm, T\_max=10, eta\_min=0)

### 2) 데이터 불균형 해소 : Focal Loss

focal loss를 가지고 온 github입니다.

https://github.com/clcarwin/focal\_loss\_pytorch

전에 사용했던 imbalanced-dataset-sampler를 테스트했을 때 사용했을 때와 비슷한 결과가 나와서 다른 방법을 시도했습니다.

사용해볼 방법은 다음과 같습니다.

* **Focal Loss**  
    
  gamma 값을 바꿔가면서 사용
* **Over Sampling**  
    
  소수 class의 data를 복사하는 방법으로 사용
* **Focal Loss & Over Sampling**  
    
  둘을 섞으면 성능이 더 나아지지 않을까...
* **Weighted loss**loss에 weight을 주는 방법  
    
  ref) [discuss.pytorch.org/t/weights-in-weighted-loss-nn-crossentropyloss/69514](https://discuss.pytorch.org/t/weights-in-weighted-loss-nn-crossentropyloss/69514)

Focal Loss는 다음과 같습니다.

class FocalLoss(nn.Module):

def \_\_init\_\_(self, weight=None,

gamma=2., reduction='mean'):

nn.Module.\_\_init\_\_(self)

self.weight = weight

self.gamma = gamma

self.reduction = reduction

def forward(self, input\_tensor, target\_tensor):

log\_prob = F.log\_softmax(input\_tensor, dim=-1)

prob = torch.exp(log\_prob)

return F.nll\_loss(

((1 - prob) \*\* self.gamma) \* log\_prob,

target\_tensor,

weight=self.weight,

reduction=self.reduction

)

사용은 다른 loss와 똑같이 사용하면 됩니다.

loss = FocalLoss(gamma=2)

γ에 따라 class의 imbalance를 조절할 수 있다고 합니다.

가장 좋은 성능은 γ=2 일 때라고 하는데 확인해 보아야 할 것 같습니다.

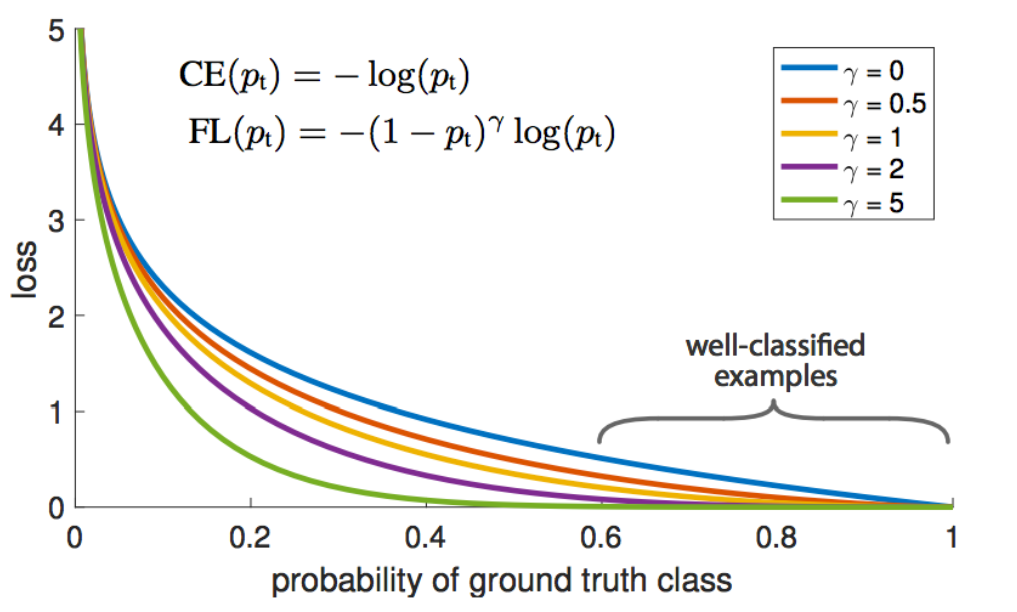
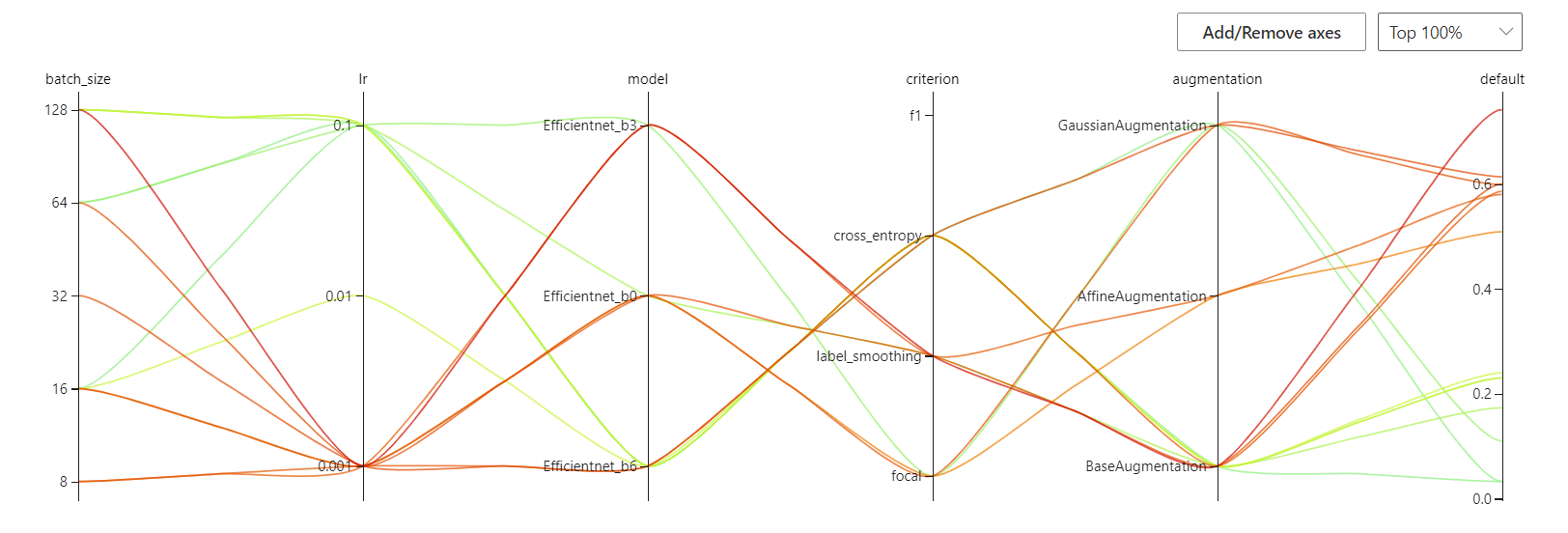
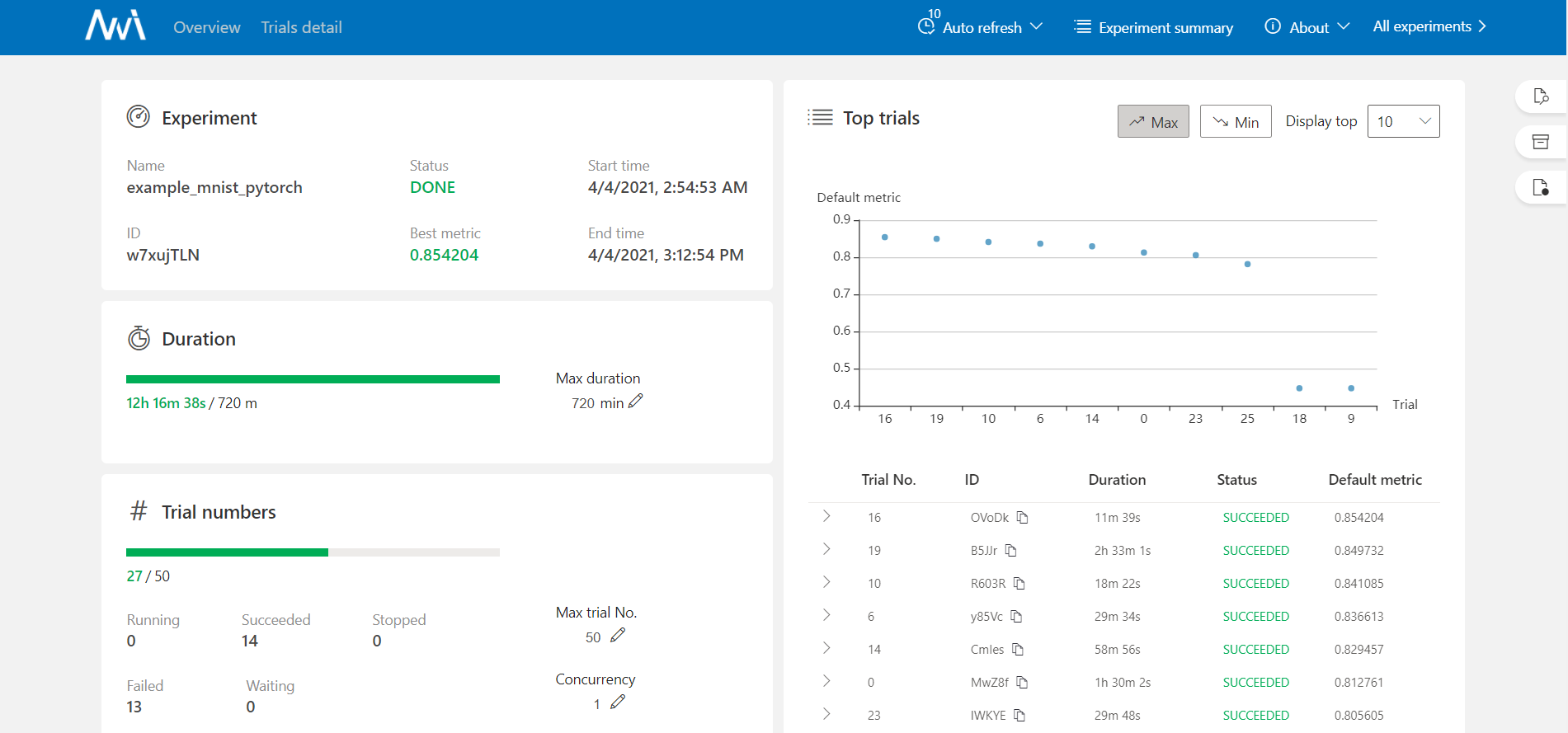


Figure . [Tsung-Yi Lin el al. arXiv 2018]

## 

### 3) nni (AutoML) 실행

nnictl을 실행하고 ngrok을 통해 monitoring하면 다음과 같이 자유롭게 hyperparameter를 넣어서 실행할 수 있습니다.



### 4) 모델 나누기 ( Mask -> Gender -> Age )

nni에서 한 번에 돌리기 위해 Dataset의 num\_classes와 model의 num\_classes 부분들을 수정해서 search\_space.json에 변수로 choice하게 만들었습니다.

*// search\_space.json*

{

"batch\_size": {"\_type":"choice", "\_value": [8, 16, 32]},

"model": {"\_type" : "choice", "\_value" : ["Efficientnet\_b6", "Efficientnet\_b4", "ecaresnet50t", "seresnet152d"]},

"criterion": {"\_type": "choice", "\_value" : ["focal", "label\_smoothing", "cross\_entropy", "f1"]},

"classification": {"\_type": "choice", "\_value": ["mask", "gender", "age", "multi"]}

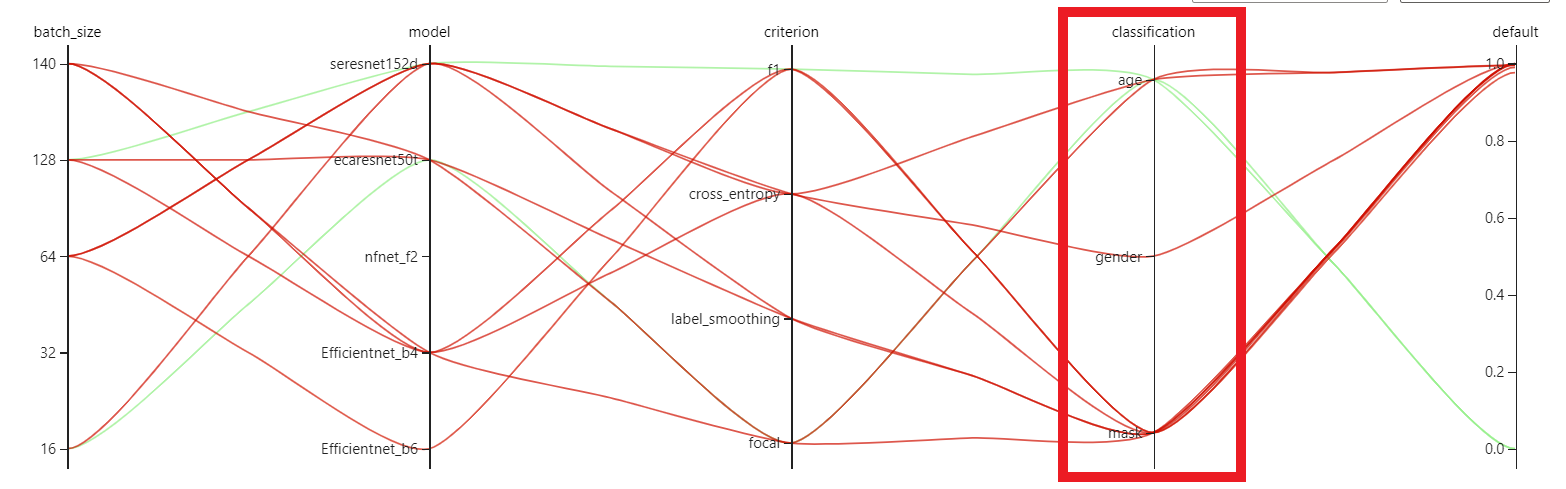
}

train.py에는 parser를 추가했습니다.

*# train.py*

parser.add\_argument('--classification', type=str, default='multi', help='classification type (default: multi)')

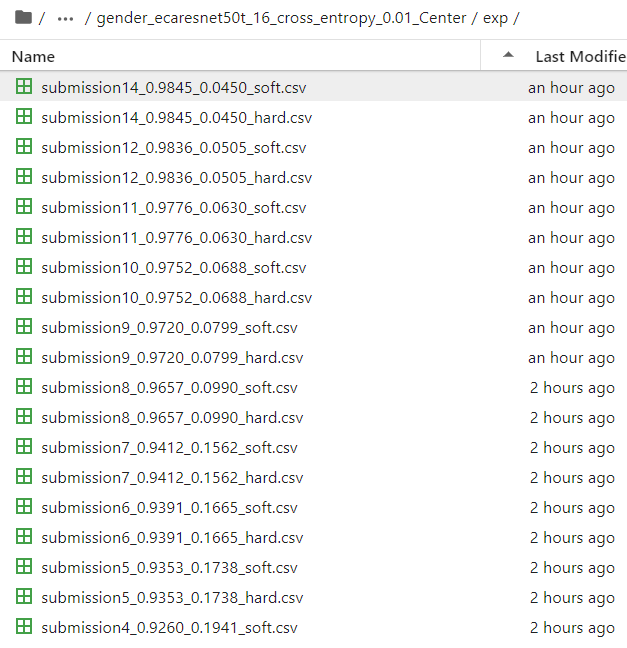
nni를 통해 mask, gender, age의 classification에 대한 여러 hyperparameter를 실험할 수 있습니다.



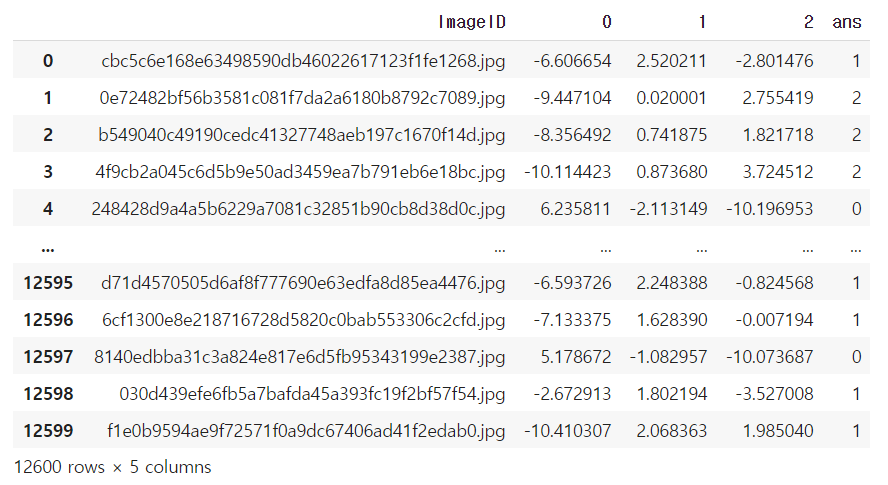
### 5) Soft voting앙상블

앙상블은 **Soft predictions votes**를 사용했습니다.

따라서 NNI를 통해 "train\_accuracy가 0.9 이상이면서 loss가 줄어든" epoch에서 soft prediction csv를 출력했습니다.



아래 사진은 csv로 출력된 soft predictions 입니다. 차례대로 0, 1, 2번 label이며, 이 중 ans는 argmax로 구합니다.



따라서 **mask, age, gender에 관한 각각의 csv들을 모아서 평균을 내줍니다.**

아래는 age ensemble에 대한 예시입니다. ensemble하고 싶은 soft predictions csv path를 list로 가지고 옵니다.

\_age\_list = [

'/content/csv\_file/age\_ecaresnet50t\_140\_focal\_0.01/exp/submission29\_0.9947\_0.0061\_soft.csv',

'/content/csv\_file/age\_seresnet152d\_64\_cross\_entropy\_0.01/exp/submission34\_0.9955\_0.0188\_soft.csv',

]

age = []

 age에 pd.read\_csv로 넣어줍니다.

for idx, path in enumerate(\_age\_list):

age.append(pd.read\_csv(path))

\_age\_list의 각각의 column '0', '1', '2' (label)들을 각 행에 맞춰서 더해줍니다.

\_0 = 0

\_1 = 0

\_2 = 0

for idx in range(len(\_age\_list)):

\_0 += age[idx]['0']

\_1 += age[idx]['1']

\_2 += age[idx]['2']

ensemble\_age라는 새로운 dataframe을 만들고, 각 label (column)의 평균을 넣어줍니다.

ensemble\_age = pd.DataFrame({'ImageID':age[0]['ImageID'],

'0':\_0/len(\_age\_list),

'1':\_1/len(\_age\_list),

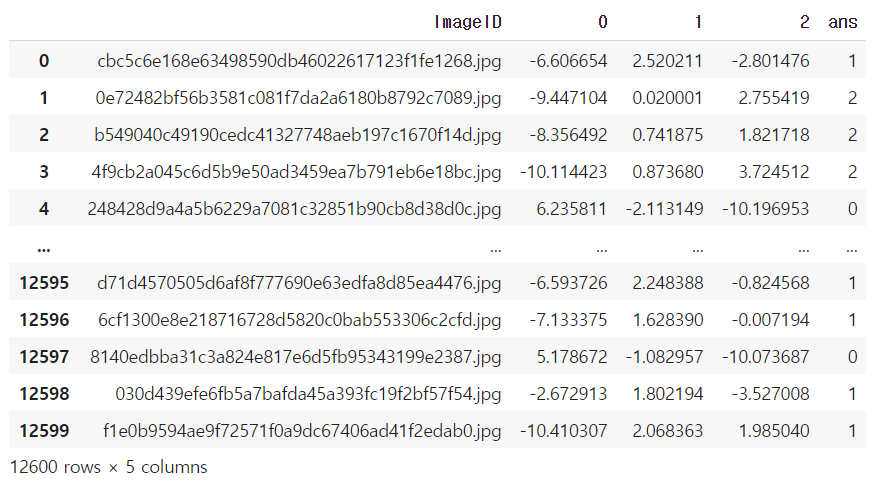
'2':\_2/len(\_age\_list),

})

ensemble \_age의 각 label들의 argmax (pandas에서는 idxmax)를 구해서 ensemble\_age의 'ans' 열에 넣어줍니다.

ensemble\_age['ans'] = ensemble\_age[['0', '1', '2']].idxmax(axis=1)

>>> ensemble\_age

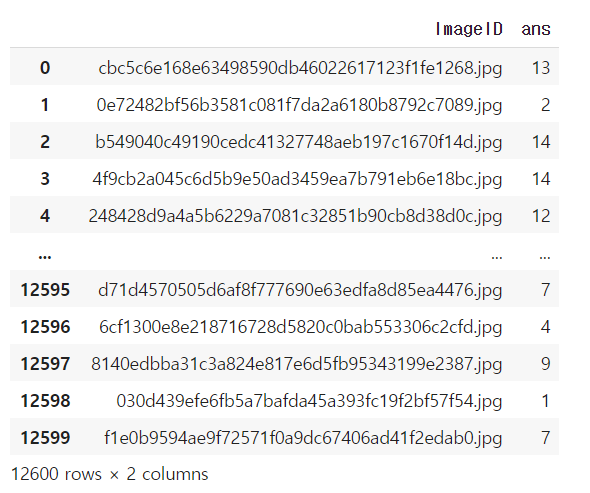


나머지 mask와 gender도 같은 방식으로 출력합니다.

이후, submission 행에 넣어서 'Image ID'와 'ans'를 출력합니다.

submission['ans'] = pd.to\_numeric(ensemble\_mask['ans']) \*6 + pd.to\_numeric(ensemble\_gender['ans'])\*3 + pd.to\_numeric(ensemble\_age['ans'])

>>> submission

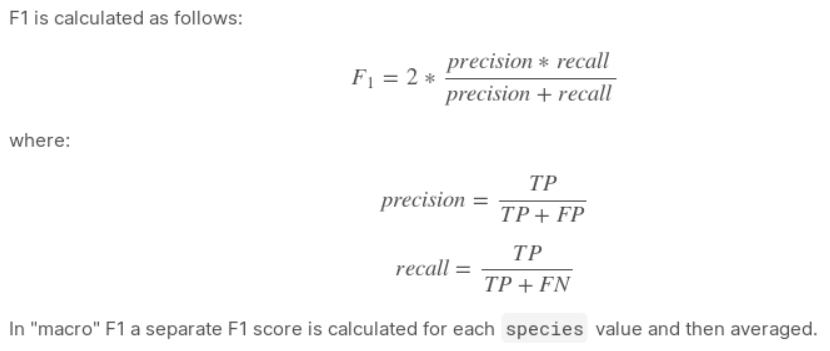


# **4. 평가**

## **1) 평가 지표 Table**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **날짜** | Data processing | Model | Training | Time | Accuracy | F1 |
| 3/29 |  | - ResNet-50 - | - Hyperparmeter 설정 - | 7h 32m | 61.87% | 0.52% |
| 3/31 | - Face Recognition - |  |  | 36m | 65.05% | 0.56% |
| 4/1 |  | - (pre-trained) Efficient-Net-b6 - | - epoch : 20, batch size : 128 - | 1h 35m | 73.67% | 0.66% |
| 4/1 |  | - (pre-trained) Efficient-Net-b7 - | - epoch : 20, batch size : 100 - | 1h 31m | 73.25% | 0.66% |
| 4/1 |  |  | - learning scheduler :  CosineAnnealingLR - epoch : 21 - | 1h 41m | 68.49% | 0.60% |
| 4/2 | -  imbalanced-dataset-sampler - |  | - learning scheduler X - epoch : 26 - | 1h 39m | 72.38% | 0.64% |
| 4/2 |  |  | - loss : Focal loss  (γ = 2) - epoch : 16 - | 1h 29m | 74.43% | 0.68% |
| 4/8 |  | - (pre-trained) Efficient-net b6 Efficient-net b4 ecaresnet50t seresnet152d - | - Ensemble (Soft voting) - | 17h 24m | 74.46% | 0.68% |
| 4/8 | - No Augmentation - | - (pre-trained) ecaresnet50t - | - batch size : 8 epoch : 15 - | 1h 47m | 78.9841% | 0.7389% |
| 4/8 |  | - (pre-trained) Efficient-net b4 - | - batch size : 16 epoch : 12 - | 1h 36m | 75.6190% | 0.6919% |
| 4/8 |  | - (pre-trained) ecaresnet50t Efficient-net b4 - | - Ensemble (Soft voting) - |  | 79.4921% | 0.7404% |

**\* F1 Score**



## **2) NNI로 최적의 Hyperparameter 찾기 + [ Crop / 일반 image ] 비교**

찾고 싶은 hyperparameter의 종류는 다음과 같습니다.

* **batch\_size :** 최적의 batch size 조합
* **model :** 모두 Efficientnet만 사용했는데 다양성이 떨어지네요..
* **criterion :** focal, label smoothing, cross entropy, f1 사용 (다음에는 weighted loss도 사용)
* **data dir :** crop image와 일반 image를 가지고 훈련시켰을 때 성능이 차이가 나는지 궁금했습니다.

*# search\_space.json*

{

"batch\_size": {"\_type":"choice", "\_value": [8, 16, 32, 64, 128, 140]},

"model": {"\_type" : "choice", "\_value" : ["Efficientnet\_b6", "Efficientnet\_b0", "Efficientnet\_b3"]},

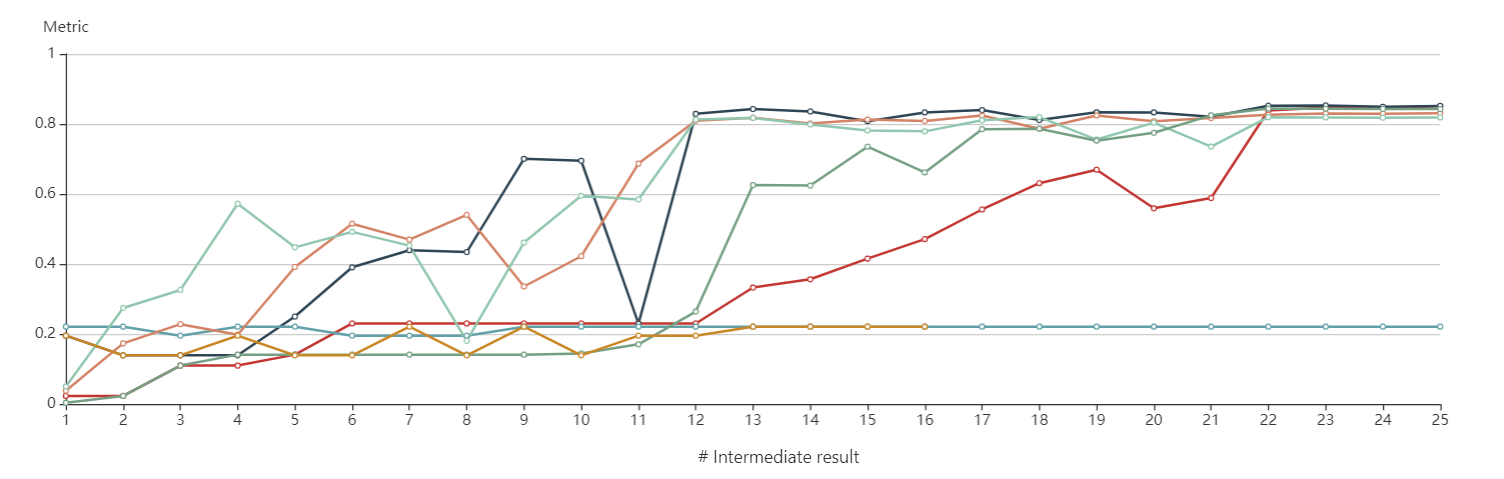
"criterion": {"\_type": "choice", "\_value" : ["focal", "label\_smoothing", "cross\_entropy", "f1"]},

"data\_dir": {"\_type": "choice", "\_value" : ["/opt/ml/input/data/train/crop\_images", "/opt/ml/input/data/train/images"]}

}

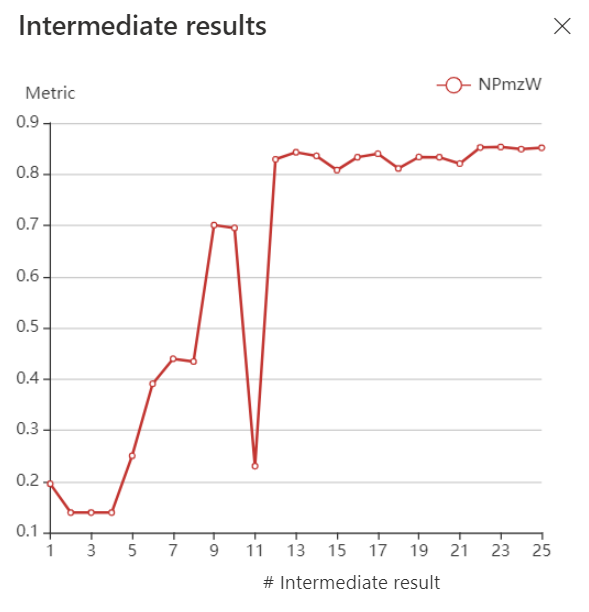
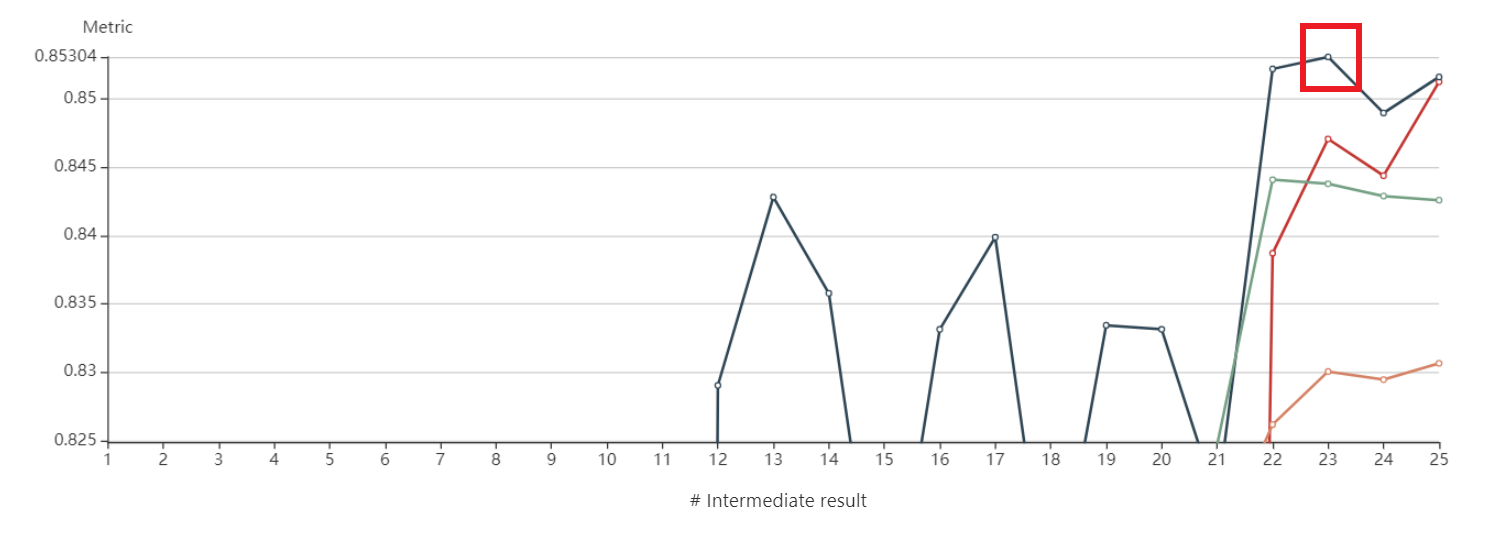
space\_search의 parameter를 가지고 돌려보면 다음과 같습니다.

* **Duration :** 8h
* **Trial Number :** Succeeded 7 / Failed 1 (아마 cuda out of memory)



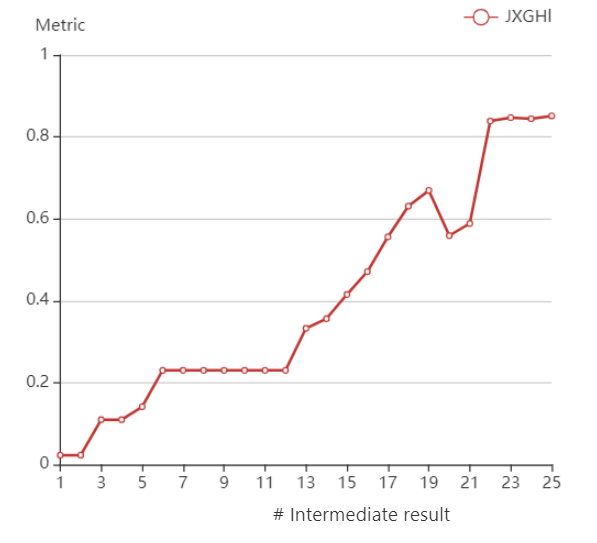
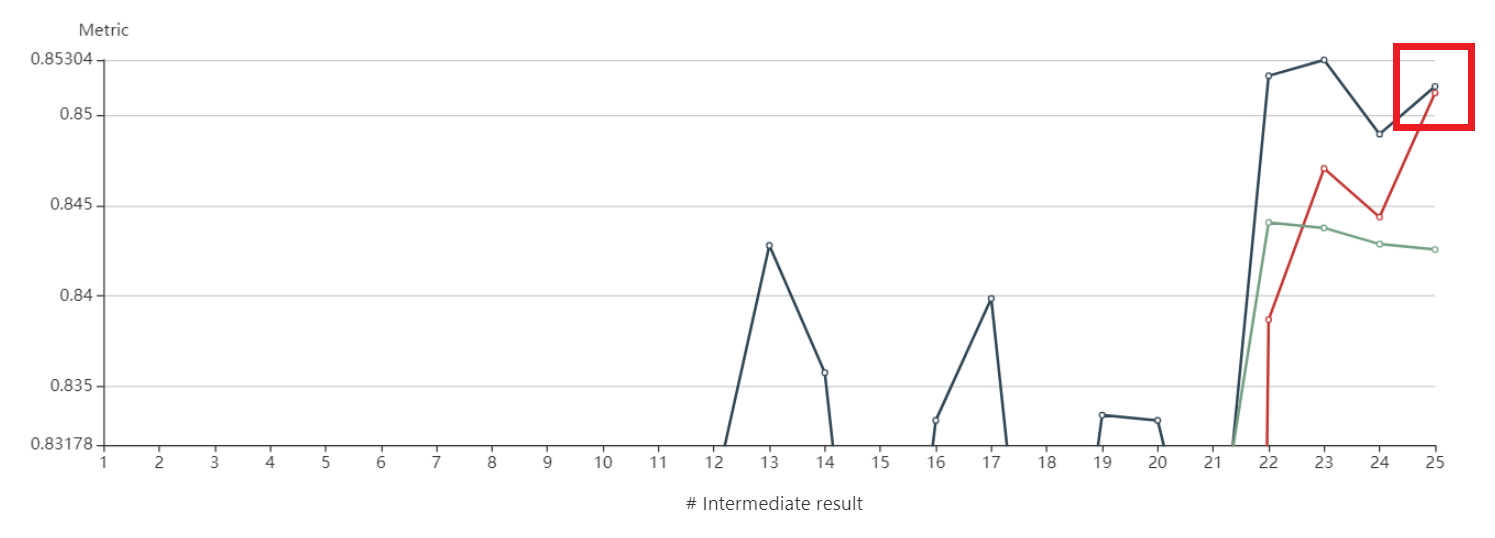
**(1) 성능 : 0.8530% (남색)**

* batch size : 64
* model : Efficientnet\_b6
* criterion : cross\_entropy
* data : 일반 images
* epochs : 23



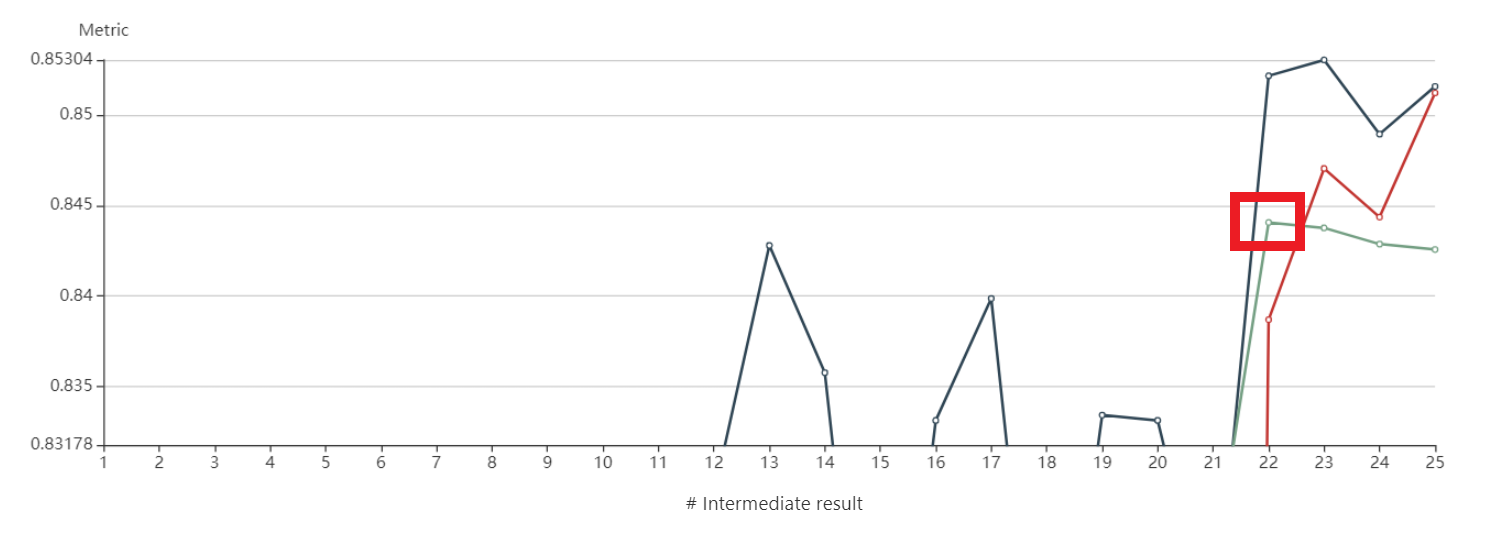
**(2) 성능 : 0.8512% (빨강)**

* batch size : 140
* model : Efficientnet\_b3
* criterion : focal
* data : crop images
* epochs : 25



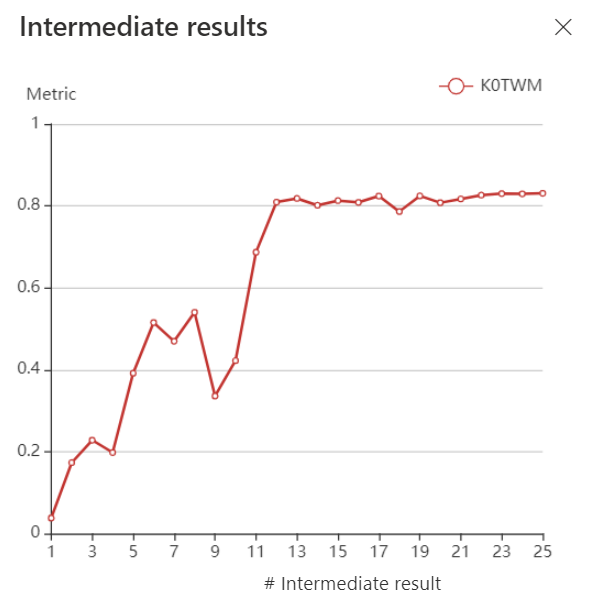
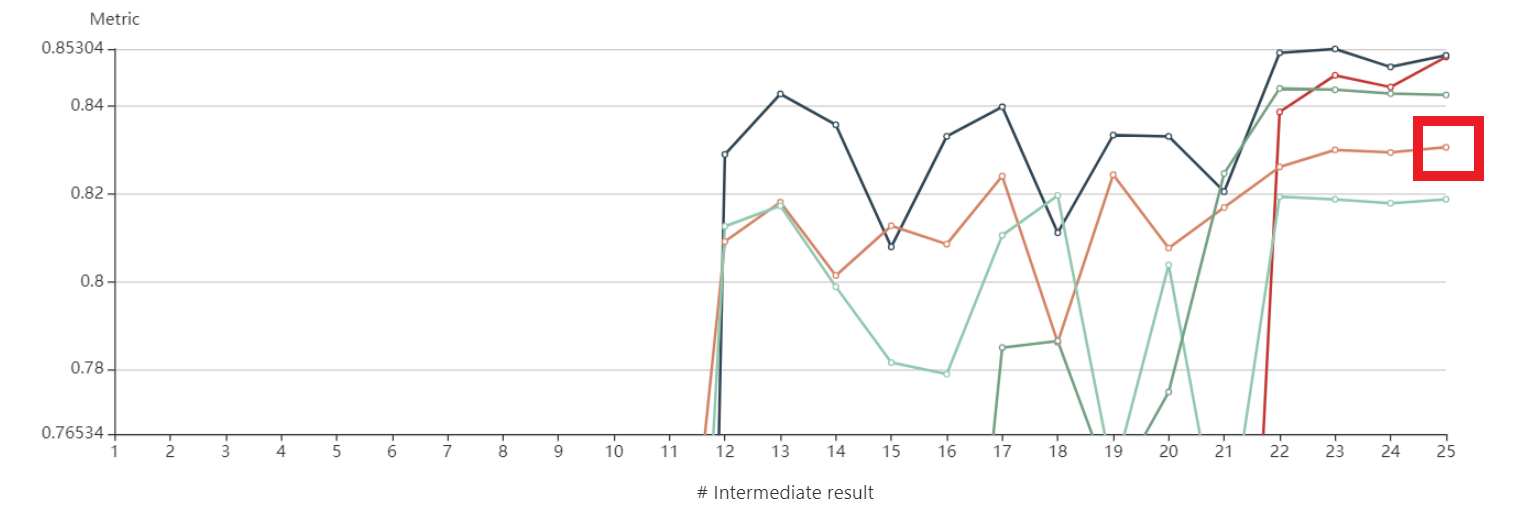
**(3) 성능 : 0.8440% (초록)**

* batch size : 140
* model : Efficientnet\_b3
* criterion : label\_smoothing
* data : crop images
* epochs : 22



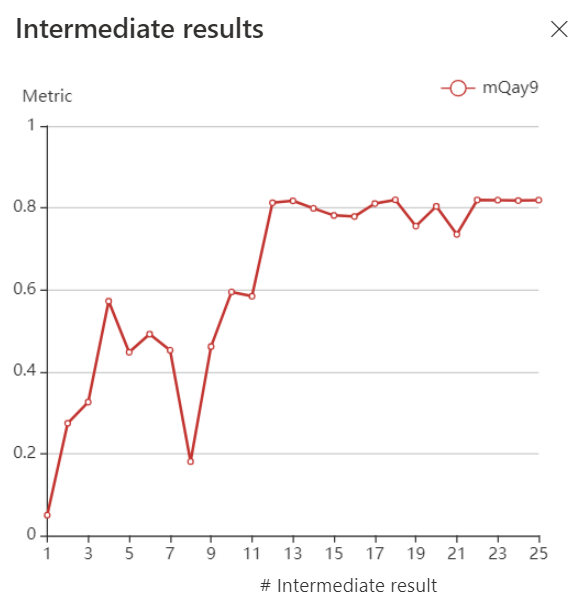
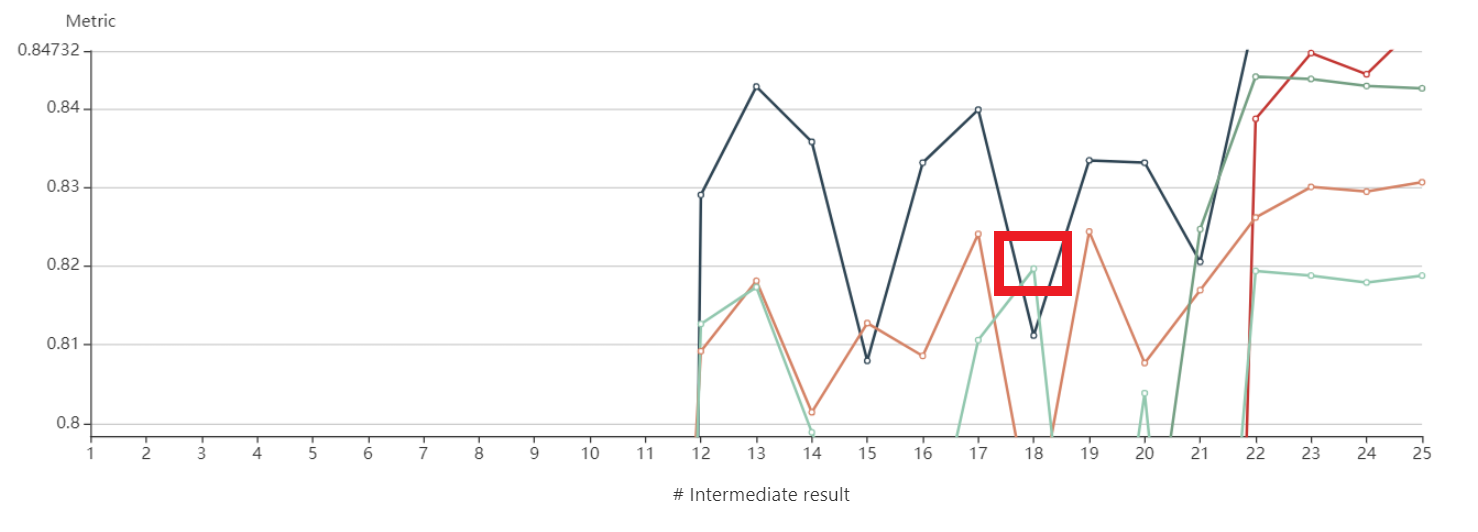
**(4) 성능 : 0.8306% (주황)**

* batch size : 32
* model : Efficientnet\_b6
* criterion : label\_smoothing
* data : crop images
* epochs : 25



**(5) 성능 : 0.8196% (연두)**

* batch size : 16
* model : Efficientnet\_b3
* criterion : cross\_entropy
* data : 일반 images
* epochs : 18

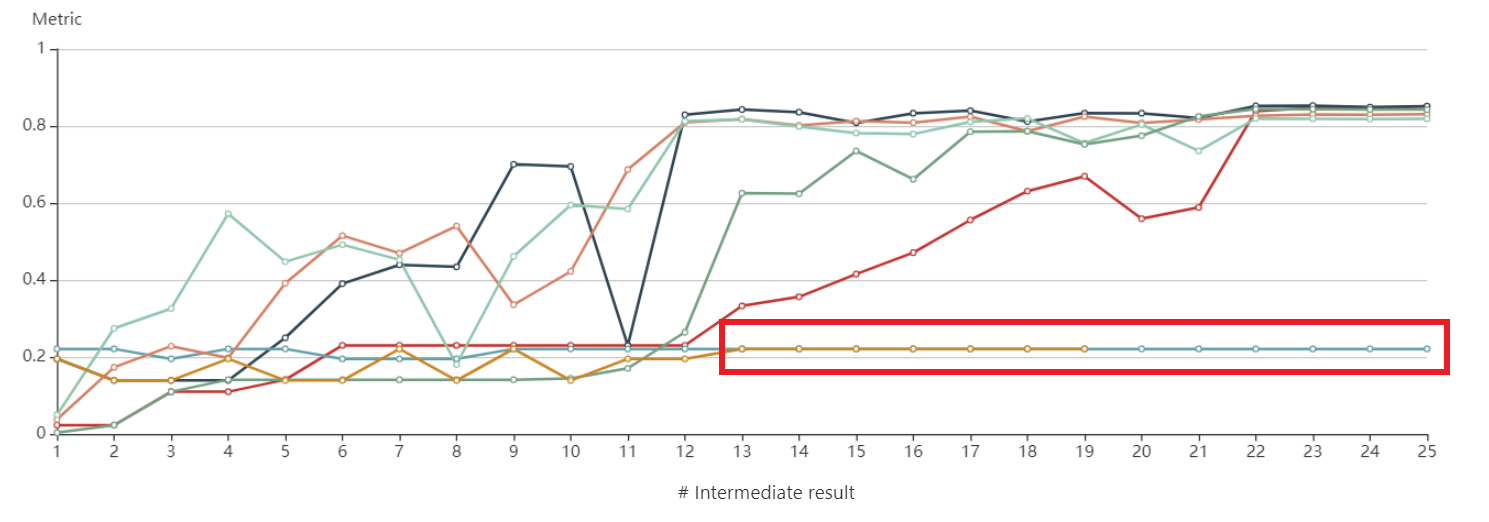


## 

**(6) 성능이 안나오는 hyperparameter**

* batch size : 8
* model : Efficientnet\_b0
* criterion : f1
* data : 일반 images
* epochs : 25
* batch size : 8
* model : Efficientnet\_b6
* criterion : f1
* data : 일반 images
* epochs : 25

모두**batch size 8, criterion f1**을 사용하고 있습니다.



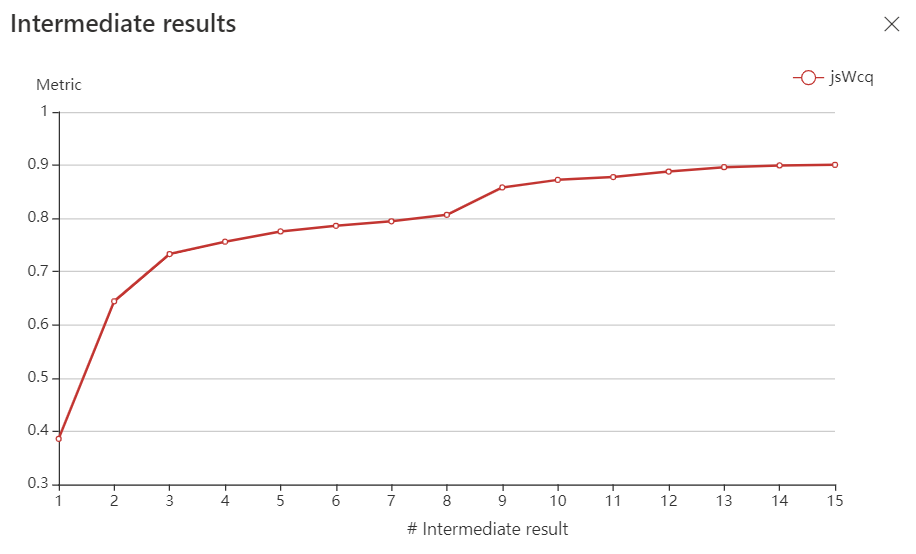
## **3) No Augmentation**

Stage 1을 하면서 Augmentation을 사용하면 오히려 성능이 하락한다는 말이 많아서 Augmentation을 아예 뺐습니다.

Data Processing 단계의 Center Crop, MTCNN Crop 모두 사용하지 않고 온전한 image를 input으로 넣었습니다.

**(1) 첫 번째 시도**

* batch\_size:8
* model:"ecaresnet50t"
* criterion:"cross\_entropy"
* classification:"multi"
* epoch : 14



효과가 있었고,**test Accuracy 4% 정도, F1-score 0.05%** 정도의 성능 향상이 있었습니다.

처음에는 image 뒤의 noise 때문에 걱정했지만, background의 noise들이 Robust 한 모델을 만들어주는 역할을 하는 것 같습니다.

**(2) 두 번째 시도**

* batch\_size: 16
* model:"Efficientnet\_b4"
* criterion:"cross\_entropy"
* classification:"multi"
* epoch : 12

더 많은 epoch을 돌렸어야 하는데 제출 시간문제로 epoch을 12까지밖에 돌리지 못했습니다.

그래도 기존**test Accuracy 1% 정도, F1-score 0.01%** 정도의 성능 향상이 있었습니다.

# **학습과정에서의 교훈**

* 피어세션을 진행하면서 개인의 구글 Searching보다 집단 지성이 더 효과적이라는 것을 배웠습니다.   
    
  물론 기본적인 구글 Searching은 중요한 요소 중 하나지만, 피어세션과 토론 게시판을 통해 사람들과 함께 지식을 공유하고 소통하면서 전체적으로 더욱 성장하는 시간을 가졌습니다.
* 강의에서 Optimizer로 대부분 Adam만 사용한다는 말을 듣고, 프로젝트에서 Optimizer를 변경할 생각을 전혀 하지 못했습니다.   
    
  하지만 피어세션과 토론게시판을 통해 더 다양한 Optimizer와 기본적인 SGD를 사용해서 성능 효과를 봤다는 말을 들었고, 이를 계기로 다시 한 번 Optimizer에 대해 공부하는 계기가 되었습니다.  
    
  기초를 탄탄히 하는 것이 중요하는 것을 느꼈습니다.
* 이론과 실제 프로젝트는 다릅니다.  
    
  프로젝트를 하면서 정말 많이 느낀 부분입니다. 이론적으로 “어떤 Model이 좋다”, “어떤 Loss가 좋다”라는 글을 많이 봤지만, 실제로 프로젝트를 진행하면서 큰 모델보다 작은 모델에서 성능이 좋게 나온 경우도 있었고 Focal Loss(Class imbalance 문제 해결 때문)보다 Cross Entropy가 성능이 더 높게 나온 경우도 있었습니다.
* 토론 게시판의 공유 문화가 좋았습니다.  
    
  피어님들이 사용한 데이터, 모델, 하이퍼파라미터 공유와 그에 따른 결과를 알게 되어, 프로젝트를 하면서 탄력을 받았던 부분들이 있었습니다. 또한, 자잘한 팁 공유를 통해 많은 도움을 받았던 것 같습니다.

# **마주한 한계와 도전숙제**

* **아쉬웠던 점들**

1. No Augmentation 방법에서 성능 향상이 있었는데 마감시간 때문에 epoch을 많이 돌리지 못하고 제출한 것
2. 더 다양한 Loss를 사용해지 못한 것
3. Data Processing 단계에서 다양한 Augmentation 수행을 못해본 것
4. K-Fold Validation을 통한 Ensemble을 못해본 것
5. 어떻게 동작하는지 파헤치기보다는 성능 향상에 급급해 모듈을 가져다 쓰는 경우가 많았던 것 같습니다. 가장 경계해야할 부분입니다.
6. 직접 코드로 작성해보지 못한 pre-trained 모델들
7. Baseline 모듈화를 직접 코드로 작성해보지 못한 것

* **한계/교훈을 바탕으로 다음 스테이지에서 새롭게 시도해볼 것**

1. 이론적인 내용을 정확히 알고 모듈을 갖다 쓰기
2. Residual block 직접 코드로 짜기
3. 다양한 Loss와 Optimizer를 사용해보기
4. Python 모듈화 해보기