

# **TEAM 5**

2022 Spring KHU Competition

# 목차

- Model select
- Training-validation Ratio
- Data Augmentation
- Learning rate
- Loss function
- Optimizer
- Fine-tuning

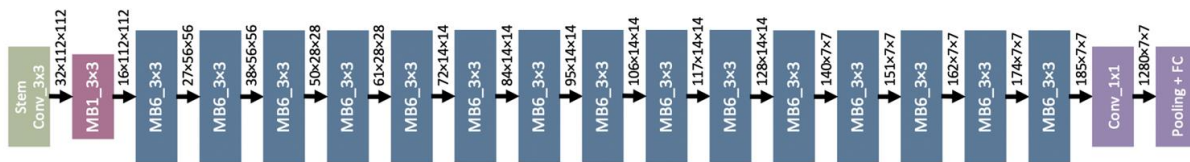
# ReXNet (Rank eXpansion Networks)

Rethinking Channel Dimensions for Efficient Model design (2021)

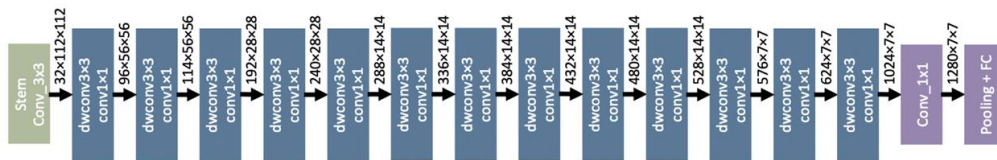
Representational bottleneck 현상을 줄이기 위해 Rank expansion의 개념 도입

NAS-based method (Neural Architecture Search) 를 통해 찾아낸 모델 구조.

MobileNetV2를 베이스 모델로 각 block의 output channel을 파라미터화 해서 훈련을 통해 찾아냄.



(a) ReXNet (x1.0)



(b) ReXNet (plain)

# 기존 연구로부터의 insight

1. inverted bottleneck은 첫 1x1 convolution에서 expansion ratio가 6 이하여야 한다.
2. 각 inverted bottleneck은 더 큰 dimension ratio를 필요로 한다.
3. 1x1 convolution과 3x3 convolution 뒤에는 복잡한 nonlinearity (ELU, SiLU)가 필요하다. (depthwise convolution은 제외)

→ReXNet은 제일 성능이 좋았던 SiLU 사용.

CIFAR-100으로 성능 측정 - insight 확인.

Network	FLOPs	Top-1	Nuc. norm
Baseline	103M	48.0%	5997.5
+ Increase DR of 1x1 conv (1/20→1/6)	99M	52.1%	6655.9
+ Increase DR of IB (0.22→0.8)	105M	53.8%	6703.2
+ Replace ReLU6 with SiLU	105M	54.6%	6895.9

Nonlinearity	Top-1 (%)	Top-5 (%)	FLOPs	Params.
ReLU6 [47]	77.3	93.5	0.40B	4.8M
Leaky ReLU [37]	77.4	93.6	0.40B	4.8M
Softplus [12]	77.6	93.8	0.40B	4.8M
ELU [8]	77.6	93.7	0.40B	4.8M
SiLU [18, 43]	<b>77.9</b>	<b>93.9</b>	0.40B	4.8M

Table A3. Nonlinear functions and ImageNet accuracy.

# Designing with Channel Configuration

channel configuration을 찾기 위한  
formulation

$$\max_{c_i, i=1 \dots d} \text{Acc}(N(c_1, \dots, c_d))$$

$$\text{s.t. } c_1 \leq c_2 \leq \dots \leq c_{d-1} \leq c_d,$$

$$\text{Params}(N) \leq P, \text{ FLOPs}(N) \leq F,$$

a, b로 channel dimension을 파라미터화

$$c_i = af(i) + b,$$

parameter를 찾기 위해 NAS method 를  
사용함.

Search constraints	Ranking	Acc (%)	Params (M)	FLOPs (M)	Best and worst models among searched configurations
(a) Models with 5 inverted bot., # Params $\simeq$ 0.2M, FLOPs $\simeq$ 30M	Top-10% Mid-10% Bot-10%	63.1 $\pm$ 0.1 62.2 $\pm$ 0.0 61.4 $\pm$ 0.2	0.2 $\pm$ 0.0 0.2 $\pm$ 0.0 0.2 $\pm$ 0.0	30 $\pm$ 1 30 $\pm$ 1 30 $\pm$ 1	Best: 34-34-45-55-66 (Acc: 63.4%) Worst: 36-36-36-36-83 (Acc: 61.1%)
(b) Models with 9 inverted bot., # Params $\simeq$ 0.5M, FLOPs $\simeq$ 100M	Top-10% Mid-10% Bot-10%	68.8 $\pm$ 0.1 68.0 $\pm$ 0.1 67.4 $\pm$ 0.3	0.5 $\pm$ 0.0 0.5 $\pm$ 0.0 0.5 $\pm$ 0.0	101 $\pm$ 5 100 $\pm$ 6 97 $\pm$ 8	Best: 24-33-42-50-59-68-77-85-94 (Acc: 68.9%) Worst: 39-39-39-39-39-39-39-87-158 (Acc: 67.6%)
(c) Models with 9 inverted bot., # Params $\simeq$ 1.0M, FLOPs $\simeq$ 200M	Top-10% Mid-10% Bot-10%	71.0 $\pm$ 0.1 70.3 $\pm$ 0.0 69.9 $\pm$ 0.2	1.0 $\pm$ 0.0 1.0 $\pm$ 0.0 1.0 $\pm$ 0.0	210 $\pm$ 15 198 $\pm$ 18 200 $\pm$ 15	Best: 30-45-59-74-88-103-117-132-146 (Acc: 71.1%) Worst: 47-47-70-70-70-70-70-364 (Acc: 69.6%)
(d) Models with 13 inverted bot., # Params $\simeq$ 3.0M, FLOPs $\simeq$ 300M	Top-10% Mid-10% Bot-10%	73.0 $\pm$ 0.1 72.1 $\pm$ 0.1 71.7 $\pm$ 0.1	3.0 $\pm$ 0.0 3.0 $\pm$ 0.0 3.0 $\pm$ 0.0	351 $\pm$ 5 351 $\pm$ 5 351 $\pm$ 6	Best: 34-34-34-40-64-88-112-136-160-184-208-232-256 (Acc: 73.2%) Worst: 52-52-52-52-52-52-52-52-115-263-263-412 (Acc: 71.6%)

# 모델 선정 이유

parameter 수가 5M으로 제한되어 있기 때문에 가볍고 성능이 좋은 모델을 선정.

→ ReXNet, EfficientNet-B0 선정.

ReXNet을 제안한 논문에서 비교한 결과에 따르면 EfficientNet에 비해 더 적은 파라미터로 조금 더 높은 성능을 보임.

fine-grained classification task에도 ResNet50, EfficientNet-B0보다 더 좋은 성능을 보였기 때문에 ReXNet (x1.0) 을 사용하기로 결정.

ReXNet vs. EfficientNet

Network	Top-1	Top-5	FLOPs	Params	CPU	GPU
ReXNet (x0.9)	<b>77.2%</b>	<b>93.5%</b>	0.35B	4.1M	45ms	20ms
Eff-B0 [51]	77.3%	93.5%	0.39B	5.3M	47ms	23ms
ReXNet (x1.0)	<b>77.9%</b>	<b>93.9%</b>	0.40B	4.8M	47ms	21ms
Eff-B1 [51]	79.2%	94.5%	0.70B	7.8M	70ms	37ms
ReXNet (x1.3)	<b>79.5%</b>	<b>94.7%</b>	0.66B	7.6M	55ms	28ms
Eff-B2 [51]	80.3%	95.0%	1.0B	9.2M	77ms	48ms
ReXNet (x1.5)	<b>80.3%</b>	<b>95.2%</b>	0.9B	9.7M	59ms	31ms
Eff-B3 [51]	<b>81.7%</b>	95.6%	1.8B	12M	100ms	78ms
ReXNet (x2.0)	81.6%	<b>95.7%</b>	1.5B	16M	69ms	40ms

Fine-grained classifications

Dataset	Network	Top-1	FLOPs	Params
Food-101 [2]	ResNet50 [16]	87.0%	4.1B	25.6M
	EfficientNet-B0 [51]	87.5%	0.4B	5.3M
	<b>ReXNet (x1.0)</b>	<b>88.4%</b>	0.4B	4.8M
Stanford Cars [28]	ResNet50 [16]	<b>92.6%</b>	4.1B	25.6M
	EfficientNet-B0 [51]	90.7%	0.4B	5.3M
	<b>ReXNet (x1.0)</b>	91.5%	0.4B	4.8M
Aircraft [38]	ResNet50 [16]	89.4%	4.1B	25.6M
	EfficientNet-B0 [51]	87.1%	0.4B	5.3M
	<b>ReXNet (x1.0)</b>	<b>89.5%</b>	0.4B	4.8M
Flowers-102 [41]	ResNet50 [16]	97.7%	4.1B	25.6M
	EfficientNet-B0 [51]	97.3%	0.4B	5.3M
	<b>ReXNet (x1.0)</b>	<b>97.8%</b>	0.4B	4.8M

# 모델 선정 이유

기존의 배웠던 ResNet과 DenseNet을 주어진 조건에 맞게 하이퍼 파라미터를 변경해서 학습을 시도함.

→ 비교적 작은 파라미터 수인 5M에서는 두 모델은 충분한 성능을 보이지 못했음.

위 슬라이드에서 비교대상이었던 EfficientNet 모델을 이용해서 학습을 진행함.

→ Adam과의 조합에 있어 RexNetV1보다 성능이 좋지 않았음.

성능을 끌어 올리기 위한 다양한 시도를 하지 않았음에도 RexNetV1의 평균 성능이 가장 뛰어남.

학습에 있어 무엇보다도 기본 모델을 빠르게 선정하는 것이 좋다고 팀원 간의 합의가 이뤄짐.

→ RexNetV1을 기반으로 모델을 학습하기로 결정.

→ 모델의 train 성능은 충분하다고 판단. 오버피팅을 최대한 줄이는 방법들을 고민하기 시작.

# RexNetV1 변경 시도

```
class ReXNetV1(nn.Module):  
    def __init__(self, input_ch=16, final_ch=180, width_mult=1.0, depth_mult=1.0, classes=400,  
                  use_se=True,  
                  se_ratio=12,  
                  dropout_ratio=0.2,  
                  bn_momentum=0.9):  
        super(ReXNetV1, self).__init__()
```

parameter 수가 5M 이하인 범위에서, input\_ch와 width\_mult, depth\_mult, dropout\_ratio 모두 변경 시도.

→ 대부분의 결과에서 오히려 일반화 성능이 하락함.

→ RexNet의 default 값 자체가 학습이 굉장히 잘 되도록 설정되어 있었기 때문이라고 판단.

dropout\_ratio를 증가시켰을 때 약간의 일반화 성능 향상을 보였지만, weight\_decay를 함께 높였을 때 train\_data에 대한 학습이 잘 이뤄지지 않았음.



## RexNetV1 변경 시도

```
self.fc = nn.Sequential(  
    nn.Conv2d(in_channels, channels // se_ratio, kernel_size=1, padding=0),  
    nn.BatchNorm2d(channels // se_ratio),  
    nn.ReLU(inplace=True),  
    nn.Conv2d(channels // se_ratio, channels, kernel_size=1, padding=0),  
    nn.Sigmoid()  
)
```

RexNetV1 내부에 임의의 층을 추가하거나, 제거 하는 방향으로 모델 성능 확인.

→ 그 어떤 변화도 기존보다 좋아지지 않았음...

주어진 조건과 시간에 있어 모델을 세세하게 이해하고, 내부를 변경해서 성능을 증가시킨다는 것이 현실적으로 불가능함을 깨달음.

→ dropout ratio를 포함, 모델 내부를 변경하지 않기로 결정!

# Train-validation Ratio

```
[ ] data = datasets.ImageFolder(data_dir)
    train_size = int(len(data)*0.95)
    val_size = int((len(data)-train_size))
    train_data, val_data = random_split(data, [train_size, val_size])
    torch.manual_seed(3334)
    print(f'train size: {len(train_data)}\nval size: {len(val_data)}')

    train_data.dataset.transform = train_transform
    val_data.dataset.transform = val_transform
    batch_size = 128
    train_loader = DataLoader(train_data, batch_size=batch_size, shuffle=True, num_workers=2)
    val_loader = DataLoader(val_data, batch_size=batch_size, shuffle=True, num_workers=2)
```

train size: 55468

val size: 2920

val accuracy 의 신뢰성이 낮아지더라도, 더 높은 일반화 성능을 위해 train\_size를 증가.

# Data Augmentation

```
[ ] train_transform = transforms.Compose([transforms.Resize((64,64)),transforms.RandomRotation(45),transforms.RandomHorizontalFlip(),
                                         transforms.RandomVerticalFlip(),transforms.ToTensor(),
                                         transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])])
val_transform = transforms.Compose([transforms.Resize((64,64)),transforms.RandomRotation(45),transforms.RandomHorizontalFlip(),
                                    transforms.RandomVerticalFlip(),transforms.ToTensor(),
                                    transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])])
```

- Resize()를 제거.

→ 학습 시간은 오래 소요되지만 전체적인 성능 향상을 보였음.

- RandomVerticalFlip()을 제거.

새 data의 특성상 새가 거꾸로 된 상태로 사진이 찍힐 확률이 낮다고 판단함.

→ 제거 시 전체적인 accuracy가 증가함. 앞으로의 augmentation 정책에 있어, 높은 각도의 변경을 자제하는 것이 좋다고 판단함.

# Data Augmentation

```
[ ] train_transform = transforms.Compose([transforms.RandomErasing(),
                                          transforms.AutoAugment(),
                                          transforms.ToTensor(),
                                          transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])])
val_transform = transforms.Compose([transforms.ToTensor(),
                                    transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])])
```



- AutoAugment()를 사용.

→ ImageNet에 기반한 augmentation 방법으로, 다양한 이미지에 대해 사용되고 있으며, 무엇보다 우리 데이터에도 성능이 좋았음.

- RandomErasing()을 사용.

배경이나 새가 물체에 가려졌을 때를 대비. 일반화 성능이 향상할 것으로 기대함.

→ 실제로 적용 시, 전체적인 accuracy가 증가함.

# Data Augmentation

```
train_transform = transforms.Compose([transforms.RandomHorizontalFlip(),
                                      transforms.RandomRotation((-20, 20)),
                                      transforms.RandomRotation((-40, 40)),
                                      transforms.RandomErasing(),
                                      transforms.ToTensor(),
                                      transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])])
val_transform = transforms.Compose([transforms.ToTensor(),
                                    transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])])
```

- 두 번의 RandomRotation()을 사용.

→ VerticalFlip시 성능이 줄어드는 것을 반영.

→ 비교적 작은 각도의 Rotation 들을 사용. 회전을 통한 이미지에 대한 다양성을 높여 일반화 성능이 상승함!

## 최종 Data Augmentation

```
train_transform = transforms.Compose([transforms.RandomHorizontalFlip(),
                                      transforms.RandomRotation((-15, 15)),
                                      transforms.RandomRotation((-25, 25)),
                                      transforms.RandomErasing(),
                                      transforms.ToTensor(),
                                      transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])])
val_transform = transforms.Compose([transforms.ToTensor(),
                                    transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])])
```

- 두 번의 RandomRotation()의 범위를 축소.

20도와 40도가 합쳐진 각도는 비교적 커, 학습에 방해가 될 수 있다고 판단.

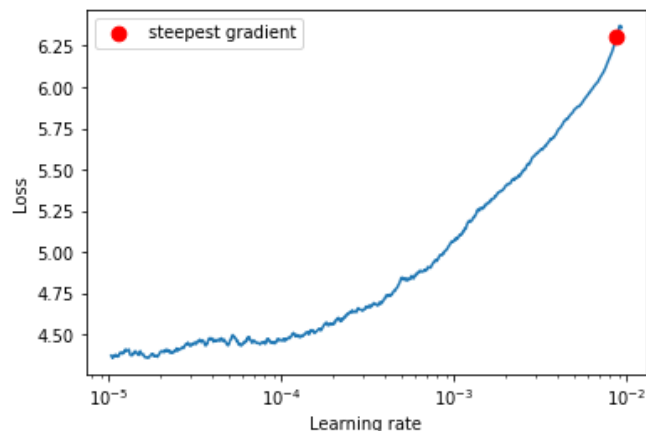
20도를 15도로, 40도를 25도로 줄임.

→ 여러 조합의 Data Augmentation 중 위 결과가 가장 일반화 성능이 좋았음.

# Learning rate - lr\_finder ?

```
!pip install torch_lr_finder
from torch_lr_finder import LRFinder

criterion = nn.CrossEntropyLoss(label_smoothing=0.06)
optimizer = optim.AdamW(model.parameters(), lr=1e-2)
lr_finder = LRFinder(model, optimizer, criterion)
lr_finder.range_test(train_loader, end_lr=1e-5, num_iter=1000, step_mode="exp")
lr_finder.plot() # to inspect the loss-learning rate graph
lr_finder.reset() # to reset the model and optimizer to their initial state
```



Typically, a good static learning rate can be found half-way on the descending loss curve.

lr\_finder의 개발자 의도와 반대의 그래프가 나왔지만, 해당 방법으로 나온 lr을 그냥 사용함.

→ 성능이 나쁘지 않다?

→ 초기 lr로 8.77E-03을 학습에 계속 사용했음.

# Learning rate - Scheduler ?

```
scheduler = torch.optim.lr_scheduler.StepLR(optimizer, step_size=1, gamma= 0.95)
```

scheduler는 일정한 epoch마다 Learning rate를 변경해주는 기법.

다양한 scheduler가 존재. 그 중에서 많이 흔히 사용되는 StepLR 방법은 매 step\_size마다, 기존의 lr에 gamma를 곱해주는 방식.

→ 사용했을 때보다 성능이 향상되지 않음.

→ 어떤 scheduler를 사용할 지, StepLR을 사용한다고 하더라도 step\_size와 gamma 등 고려할 파라미터가 증가해서 부담.

→ Scheduler를 사용하지 않기로 결정!



# Loss function - CrossEntropyLoss

$$\ell(x, y) = L = \{l_1, \dots, l_N\}^\top, \quad l_n = -w_{y_n} \log \frac{\exp(x_{n,y_n})}{\sum_{c=1}^C \exp(x_{n,c})} \cdot 1\{y_n \neq \text{ignore\_index}\}$$

where  $x$  is the input,  $y$  is the target,  $w$  is the weight,  $C$  is the number of classes, and  $N$  spans the minibatch dimension as well as  $d_1, \dots, d_k$  for the  $K$ -dimensional case. If `reduction` is not `'none'` (default `'mean'`), then

$$\ell(x, y) = \begin{cases} \sum_{n=1}^N \frac{1}{\sum_{n=1}^N w_{y_n} \cdot 1\{y_n \neq \text{ignore\_index}\}} l_n, & \text{if reduction = 'mean';} \\ \sum_{n=1}^N l_n, & \text{if reduction = 'sum'}. \end{cases}$$

Note that this case is equivalent to the combination of `LogSoftmax` and `NLLLoss`.

CrossEntropyLoss 말고 다른 손실 함수를 고려하지 않았음.  
label\_smoothing을 제외한 다른 파라미터 수정하지 않고, default 값을 사용.

# Label smoothing

$$q'(k) = (1 - \epsilon)\delta_{k,y} + \frac{\epsilon}{K}.$$

$$H(q', p) = - \sum_{k=1}^K \log p(k) q'(k) = (1 - \epsilon)H(q, p) + \epsilon H(u, p)$$

Table 1: Survey of literature label smoothing results on three supervised learning tasks.

DATA SET	ARCHITECTURE	METRIC	VALUE w/o LS	VALUE w/ LS
IMAGENET	INCEPTION-V2 [6]	TOP-1 ERROR	23.1	<b>22.8</b>
		TOP-5 ERROR	6.3	<b>6.1</b>

Table 2: Top-1 classification accuracies of networks trained with and without label smoothing used in visualizations.

DATA SET	ARCHITECTURE	ACCURACY ( $\alpha = 0.0$ )	ACCURACY ( $\alpha = 0.1$ )
CIFAR-10	ALEXNET	$86.8 \pm 0.2$	$86.7 \pm 0.3$
CIFAR-100	RESNET-56	$72.1 \pm 0.3$	$72.7 \pm 0.3$

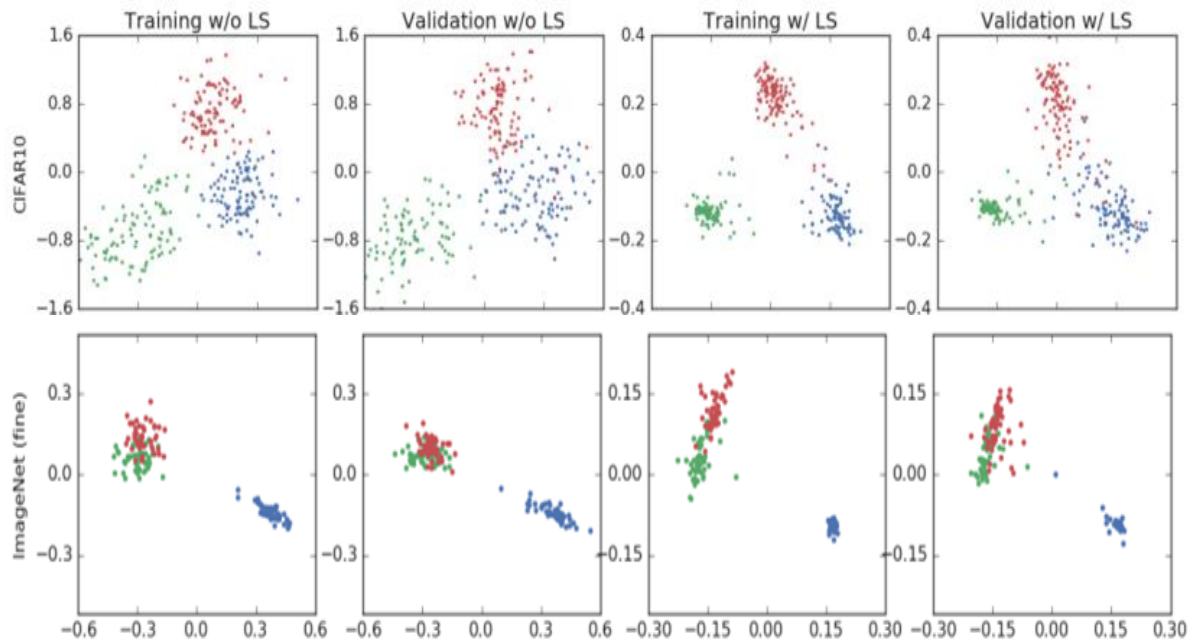
Label smoothing이란?

$\epsilon/K$  만큼의 값을 K개의 모든 class의 label의 예측 확률에 나누어 주어 soft한 예측이 이루어지게 하는 기법.

CrossEntropyLoss에서 선택적으로 적용 가능.

논문 '[When Does Label Smoothing Help?](#)' 에 따르면 label smoothing은 정답에 대한 과잉 확신을 방지하기 때문에 일반화 성능을 향상시킬 수 있다는 실험 결과를 근거로 본 학습에 사용하기로 결정함.

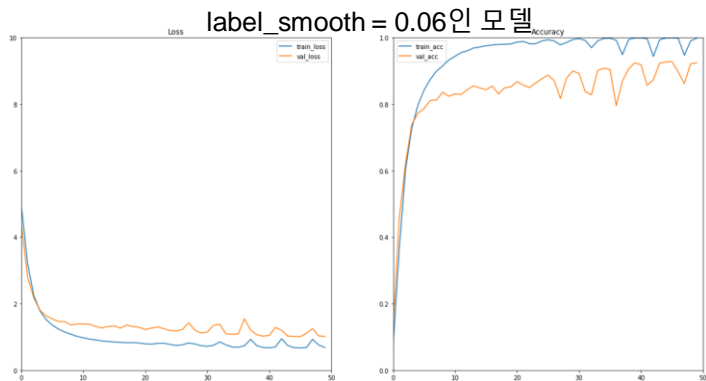
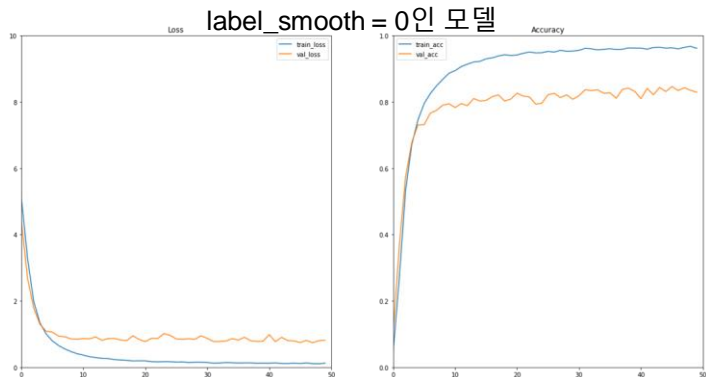
# Label smoothing과 fine grained classification



샘플을 오답 class에서 일정한 거리만큼 떨어지도록 만들기 때문에 class들끼리 서로 분리가 잘 되도록 함.

fine grained classification 에서 label smoothing을 사용했을 때 class들끼리 서로 분리가 잘 되는 게 실험적으로 밝혀짐.

# label smooth 값( $\epsilon$ ) 변화에 따른 성능 변화



label_smooth	val accuracy
label_smooth=0	0.846
label_smooth=0.02, 0.04	so low
label_smooth=0.06	0.928
label_smooth=0.08	0.827
label_smooth=0.1	0.9

다양한 값으로 label smooth를 시험해 보았더니,  
 $\epsilon=0.06$ 일 때 가장 일반화 성능이 좋게 나옴.

→ 0.06을 사용할 수치로 결정.

```
CLASS torch.optim.Adam(params, lr=0.001, betas=(0.9, 0.999), eps=1e-08, weight_decay=0,
    amsgrad=False, *, maximize=False) [SOURCE]
```



Implements Adam algorithm.

---

**input** :  $\gamma$  (lr),  $\beta_1, \beta_2$  (betas),  $\theta_0$  (params),  $f(\theta)$  (objective)  
           $\lambda$  (weight decay), *amsgrad*, *maximize*  
**initialize** :  $m_0 \leftarrow 0$  (first moment),  $v_0 \leftarrow 0$  (second moment),  $\widehat{v}_0^{max} \leftarrow 0$

---

**for**  $t = 1$  **to** ... **do**  
  **if** *maximize* :  
     $g_t \leftarrow -\nabla_{\theta} f_t(\theta_{t-1})$   
  **else**  
     $g_t \leftarrow \nabla_{\theta} f_t(\theta_{t-1})$   
  **if**  $\lambda \neq 0$   
     $g_t \leftarrow g_t + \lambda \theta_{t-1}$   
     $m_t \leftarrow \beta_1 m_{t-1} + (1 - \beta_1) g_t$   
     $v_t \leftarrow \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$   
     $\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$   
     $\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$   
  **if** *amsgrad*  
     $\widehat{v}_t^{max} \leftarrow \max(\widehat{v}_t^{max}, \widehat{v}_t)$   
     $\theta_t \leftarrow \theta_{t-1} - \gamma \widehat{m}_t / (\sqrt{\widehat{v}_t^{max}} + \epsilon)$   
  **else**  
     $\theta_t \leftarrow \theta_{t-1} - \gamma \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$

---

**return**  $\theta_t$

---

## Adam의 한계 ?

lr과 weight\_decay를 제외한 파라미터는 default 값을 사용.

lr은 앞서 정한 8.77E-03으로 설정.

weight\_decay는 기존의 5e-4에서 점점 키워가며 학습을 진행함.


→ 값을 크게 증가하였음에도 오버피팅 문제를 해결하지 못했음.

→ 값이 너무 크게 되면 train\_data 학습도 제대로 진행이 되지 않음.

다른 optimizer도 고민하기 시작함.

# Adam의 weight decay가 의도와 다르게 동작한다는 이슈

## Parameters

- **params** (*iterable*) – iterable of parameters to optimize or dicts defining parameter groups
- **lr** (*float, optional*) – learning rate (default: 1e-3)
- **betas** (*Tuple[float, float], optional*) – coefficients used for computing running averages of gradient and its square (default: (0.9, 0.999))
- **eps** (*float, optional*) – term added to the denominator to improve numerical stability (default: 1e-8)
- **weight\_decay** (*float, optional*) – weight decay (L2 penalty) (default: 0) 
- **amsgrad** (*boolean, optional*) – whether to use the AMSGrad variant of this algorithm from the paper [On the Convergence of Adam and Beyond](#) (default: False)
- **maximize** (*bool, optional*) – maximize the params based on the objective, instead of minimizing (default: False)

일반적으로 사용하는 Pytorch의 Adam이 weight decay로 L2 penalty를 사용.

하지만, AdamW를 소개하는 논문인

[Decoupled Weight Decay Regularization](#)에서는

Adam이 L2 regularization를 사용하여 weight decay를 부여할 때, 주어진 역할을 다하지 못한다고 함.

이는 적응적 학습률(adaptive learning rate)을 사용하는 optimizer들이 weight decay 효과를 덜 받게 되는 문제가 있기 때문.

→ 이를 해결하기 위해 AdamW를 제안.

**Algorithm 1:** *Adam*, our proposed algorithm for stochastic optimization. See section 2 for details, and for a slightly more efficient (but less clear) order of computation.  $g_t^2$  indicates the elementwise square  $g_t \odot g_t$ . Good default settings for the tested machine learning problems are  $\alpha = 0.001$ ,  $\beta_1 = 0.9$ ,  $\beta_2 = 0.999$  and  $\epsilon = 10^{-8}$ . All operations on vectors are element-wise. With  $\beta_1^t$  and  $\beta_2^t$  we denote  $\beta_1$  and  $\beta_2$  to the power  $t$ .

**Require:**  $\alpha$ : Stepsize

**Require:**  $\beta_1, \beta_2 \in [0, 1)$ : Exponential decay rates for the moment estimates

**Require:**  $f(\theta)$ : Stochastic objective function with parameters  $\theta$

**Require:**  $\theta_0$ : Initial parameter vector

$m_0 \leftarrow 0$  (Initialize 1<sup>st</sup> moment vector)

$v_0 \leftarrow 0$  (Initialize 2<sup>nd</sup> moment vector)

$t \leftarrow 0$  (Initialize timestep)

**while**  $\theta_t$  not converged **do**

$t \leftarrow t + 1$

$g_t \leftarrow \nabla_{\theta} f_t(\theta_{t-1})$  (Get gradients w.r.t. stochastic objective at timestep  $t$ )

$m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$  (Update biased first moment estimate)

$v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$  (Update biased second raw moment estimate)

$\hat{m}_t \leftarrow m_t / (1 - \beta_1^t)$  (Compute bias-corrected first moment estimate)

$\hat{v}_t \leftarrow v_t / (1 - \beta_2^t)$  (Compute bias-corrected second raw moment estimate)

$\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \hat{m}_t / (\sqrt{\hat{v}_t} + \epsilon)$  (Update parameters)

**end while**

**return**  $\theta_t$  (Resulting parameters)

**Algorithm 2** Adam with L<sub>2</sub> regularization and Adam with decoupled weight decay (AdamW)

1: **given**  $\alpha = 0.001, \beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 10^{-8}, \lambda \in \mathbb{R}$

2: **initialize** time step  $t \leftarrow 0$ , parameter vector  $\theta_{t=0} \in \mathbb{R}^n$ , first moment vector  $m_{t=0} \leftarrow \mathbf{0}$ , second moment vector  $v_{t=0} \leftarrow \mathbf{0}$ , schedule multiplier  $\eta_{t=0} \in \mathbb{R}$

3: **repeat**

4:  $t \leftarrow t + 1$

5:  $\nabla f_t(\theta_{t-1}) \leftarrow \text{SelectBatch}(\theta_{t-1})$  ▷ select batch and return the corresponding gradient

6:  $g_t \leftarrow \nabla f_t(\theta_{t-1}) + \lambda \theta_{t-1}$

7:  $m_t \leftarrow \beta_1 m_{t-1} + (1 - \beta_1) g_t$  ▷ here and below all operations are element-wise

8:  $v_t \leftarrow \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$

9:  $\hat{m}_t \leftarrow m_t / (1 - \beta_1^t)$  ▷  $\beta_1$  is taken to the power of  $t$

10:  $\hat{v}_t \leftarrow v_t / (1 - \beta_2^t)$  ▷  $\beta_2$  is taken to the power of  $t$

11:  $\eta_t \leftarrow \text{SetScheduleMultiplier}(t)$  ▷ can be fixed, decay, or also be used for warm restarts

12:  $\theta_t \leftarrow \theta_{t-1} - \eta_t (\alpha \hat{m}_t / (\sqrt{\hat{v}_t} + \epsilon) + \lambda \theta_{t-1})$

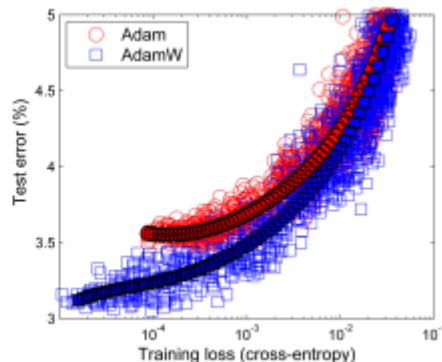
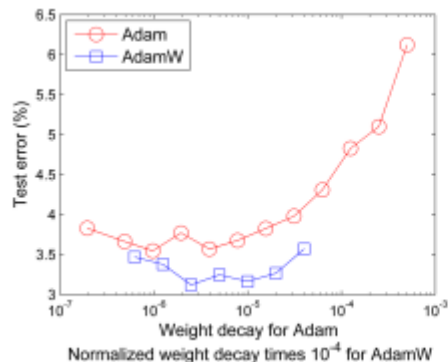
13: **until** stopping criterion is met

14: **return** optimized parameters  $\theta_t$

6 line에서 L2 regularization이 적용된 모습을 확인할 수 있음. L2-norm을 미분했을 때  $\lambda\theta$ 가 나옴.

7~10 line에서 모멘텀과 적응적 학습율을 적용하면 6 line의 weight decay의 결과가 점점 작아지기 때문에, 12 line에서 직접적으로 L2 regularization을 적용한 모습을 볼 수 있었음.

# Optimizer - AdamW



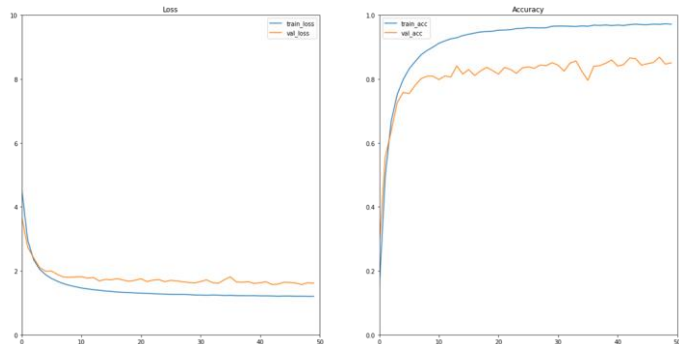
논문상에서 'AdamW가 Adam에 비해 전체적으로 낮은 test loss를 가진다'는 사실이 실험적으로 확인되었기 때문에, AdamW를 코드에 적용시켜 보았고, 그 결과 val accuracy가 증가함.

→ 사용할 optimizer로 결정.

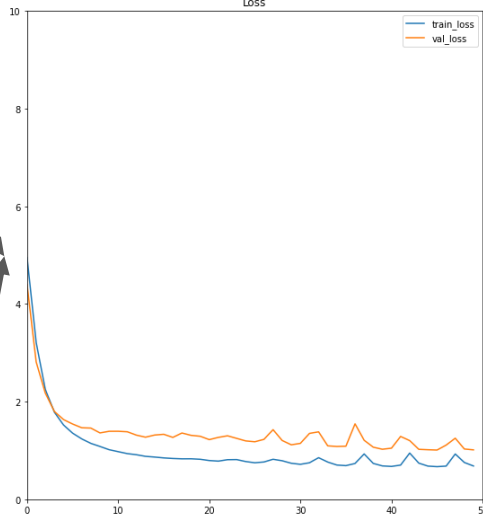


# label smoothing과 AdamW의 혼합

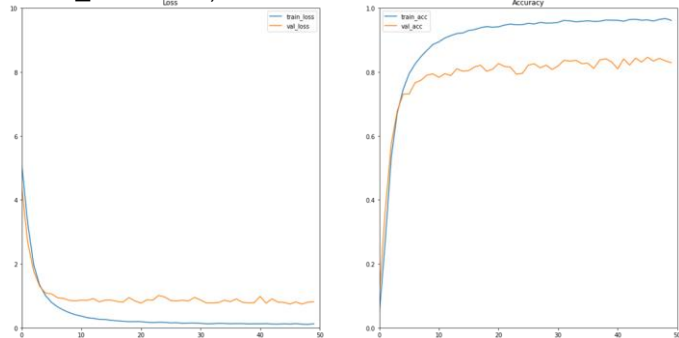
label\_smooth=0.06, Adam인 모델



label\_smooth=0.06, AdamW인 모델



label\_smooth=0, AdamW인 모델



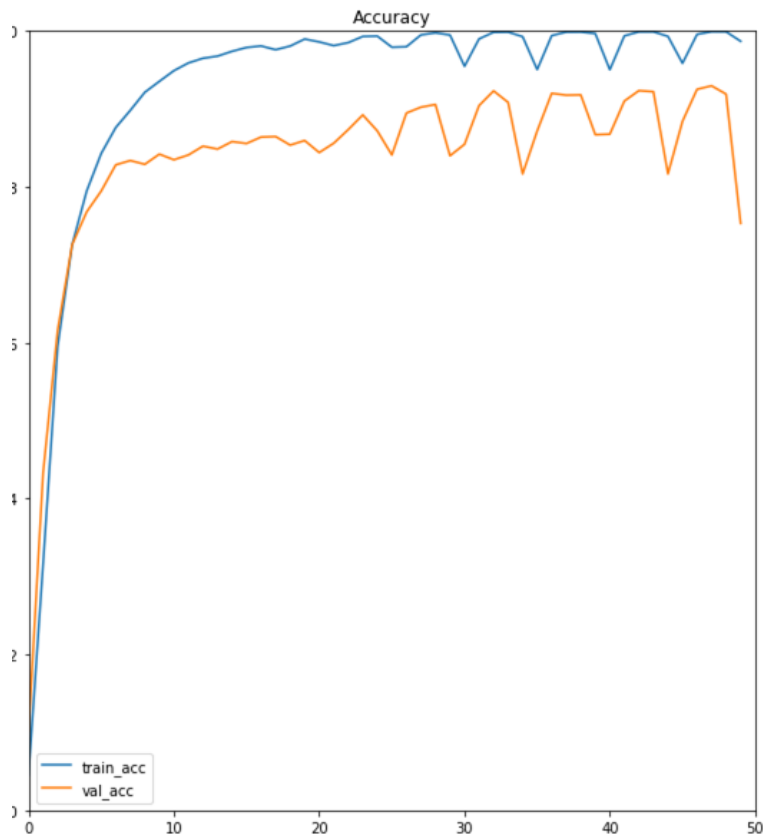
label smoothing과 AdamW를 같이 사용했을 때 진폭이 더해져 더 좋은 best parameters를 찾을 확률이 커짐.

# AdamW parameters

```
model = ReXNetV1(width_mult=1.0, classes=400).cuda()  
criterion = nn.CrossEntropyLoss(label_smoothing=0.06)  
optimizer = torch.optim.AdamW(model.parameters(), lr=5.77E-03, weight_decay=0.015)
```

- AdamW의 default weight\_decay 값(0.01) 을 사용했지만 오버피팅이 계속 발생.  
→ 오버피팅을 줄이기 위해 weight\_decay 값을 0.015로 변경. 값을 0.02로 변경했을 때는 전체적인 학습을 저하함.
- 기존의 lr로는 label smoothing과 함께 사용했을 시, 오버피팅이 너무 잘 일어난다고 판단.  
→ lr을 8.77E-03에서 5.77E-03으로 낮추고 학습 진행하기로 결정.

# 1차 모델 학습 결과 분석



1. 수 많은 모델 학습 결과 best\_model은 대부분 40 ~ 46 epoch에서 생성.

→ 굳이 50 epoch까지 같은 방법으로 학습할 필요가 있을까?

2. 학습이 계속 진행되어도, train\_acc와 val\_acc의 차이가 획기적으로 줄지 않음.

→ val\_acc의 신뢰도가 다소 낮다는 점을 고려하더라도 오버피팅이 계속 존재.

3. 남은 나머지 4 epoch를 어떻게 활용하면 좋을까?

# Fine-tuning idea

```
new_transform = transforms.Compose([transforms.RandomHorizontalFlip(),
                                    transforms.ToTensor(),
                                    transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])])
```

```
criterion_Re = nn.CrossEntropyLoss(label_smoothing = 0.06)
optimizer_Re = torch.optim.Adam(model_Re.parameters(), lr=0.001)
```

```
for epoch in range(4):
    model_Re.train()
    for inputs, labels in val_loader:
        inputs = inputs.to(device)
        labels = labels.to(device)
        optimizer_Re.zero_grad()
        with torch.set_grad_enabled(True):
            outputs = model_Re(inputs)
            loss = criterion_Re(outputs, labels)
            loss.backward()
            optimizer_Re.step()
    torch.save(model_Re.state_dict(), 'model_final.pt')
```

1. 일반화 성능을 더 올릴 수 있는 방법이 있을까?

→ 퀄리티 좋은 val\_data을 이용하여, 학습의 거의 완료된 모델을 추가로 학습하자!

2. val\_data까지 학습에 사용하면 성능 측정은 어떻게?

→ 적절한 lr과 optimizer을 이용하면, 퀄리티 좋은 data의 증가는 일반화 성능을 올릴 가능성이 높다!

→ 성능을 측정할 data set이 남아 있지 않기 때문에 최종적인 모델로는 latest\_model을 사용.

## Fine-tuning시 사용할 data

1. best\_model을 선택 시 train accuracy는 0.999 근접할 정도로 매우 높음.

→ train\_data를 추가 학습에 사용하는 것은 시간 낭비.

2. val\_data에 어떤 Augmentation을 적용할 것인가?

→ 모델의 안정성과 최소한의 데이터 증강을 위해 RandomHorizontalFlip만을 Augmentation 정책으로 선택함.

```
new_transform = transforms.Compose([transforms.RandomHorizontalFlip(),
                                    transforms.ToTensor(),
                                    transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])])
```

```
val_data.dataset.transform = new_transform
```

```
batch_size = 128
```

```
val_loader = DataLoader(val_data, batch_size=batch_size, shuffle=False, num_workers=2)
```

## Fine-tuning에 사용할 lr에 대한 많은 고민

1. 높은 lr은 모델의 안정성을 훼손할 가능성이 높다.
2. 낮은 lr은 거의 학습이 완료된 모델이 변화를 주기 힘들다.

### 고려사항

1. 4 epoch라는 낮은 반복 수.
2. 이미 학습이 거의 완료된 현재 모델의 학습 상태.
3. 전체 데이터셋의 5%에 해당하는 작은 val\_size.
4. batch size: 128
5. RandomHorizontalFlip만 적용한 최소한의 augmentation 정책.

→ lr를 0.001라는 임의의 수로 결정. (Adam의 default lr 값)

## Dropout ratio에 대한 고민

```
model_Re = ReXNetV1(width_mult=1.0, classes=400, dropout_ratio = 0.16).cuda()  
model_Re.load_state_dict(torch.load('./model_best.pt'))
```

현재 모델은 학습이 거의 완료된 상태에서 작은 크기의 val\_data를 이용하여, 적은 반복 횟수와 작은 lr로 추가 학습을 진행하려고 함.

val\_data 학습을 통해 조금이라도 train\_data에 오버피팅된 모델 파라미터 값을 변경하기 위해서, dropout ratio을 조금 낮추는 방식으로 수정함.

→ 기존 수치와 크게 차이가 나지 않는 수치인 0.16로 최종 결정함.

## Fine tuning을 위한 세부 설정

### optimizer - Adam

Adam, SGD, AdamW 등을 포함한 다양한 optimizer을 초기에 고민.

weight\_decay가 fine tuning 과정에 큰 도움이 되지 않을 것이라 생각.

→ default Adam을 사용하기로 결정.

### label smoothing ?

Adam과 함께 사용했을 때 안정성에 큰 문제가 없었음.

label smoothing이 fine grained classification에서 일반화 성능을 끌어 올릴 수 있음.

→ 기존에 사용했던 값인 0.06으로 계속 사용하기로 결정!

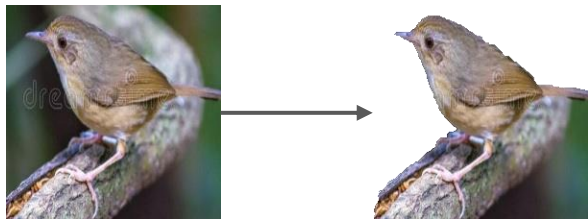


# 성능향상을 위한 기타 노력들

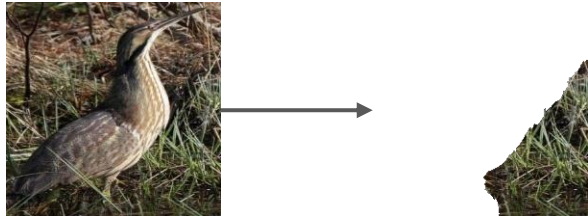
opencv의 grabCut()을 사용하여 일괄적으로 배경 제거 시도.

제거가 적절하게 되지 않은 경우 학습에 악영향.

배경 제거가 잘 된 경우



배경 제거가 잘 되지 않은 경우



Validation set을 나눌 때 stratify 시도.

유의미한 val accuracy의 신뢰성 증가는 없었음.

```
from sklearn.model_selection import train_test_split

data = datasets.ImageFolder(data_dir)

train_indices, val_indices = train_test_split(list(range(len(data.targets))), test_size=0.05, stratify=data.targets)

train_data = torch.utils.data.Subset(data, train_indices)
val_data = torch.utils.data.Subset(data, val_indices)

torch.manual_seed(3334)
print(f'train size: {len(train_data)}\nval size: {len(val_data)}')

train_data.dataset.transform = train_transform
val_data.dataset.transform = val_transform
batch_size = 128
train_loader = DataLoader(train_data, batch_size=batch_size, shuffle=True)
val_loader = DataLoader(val_data, batch_size=batch_size, shuffle=True)
```

Python

train size: 55468  
val size: 2920

# 감사합니다.

결과는 아쉽지만, 팀원 한 명도 빠짐없이 최선을 다했습니다.