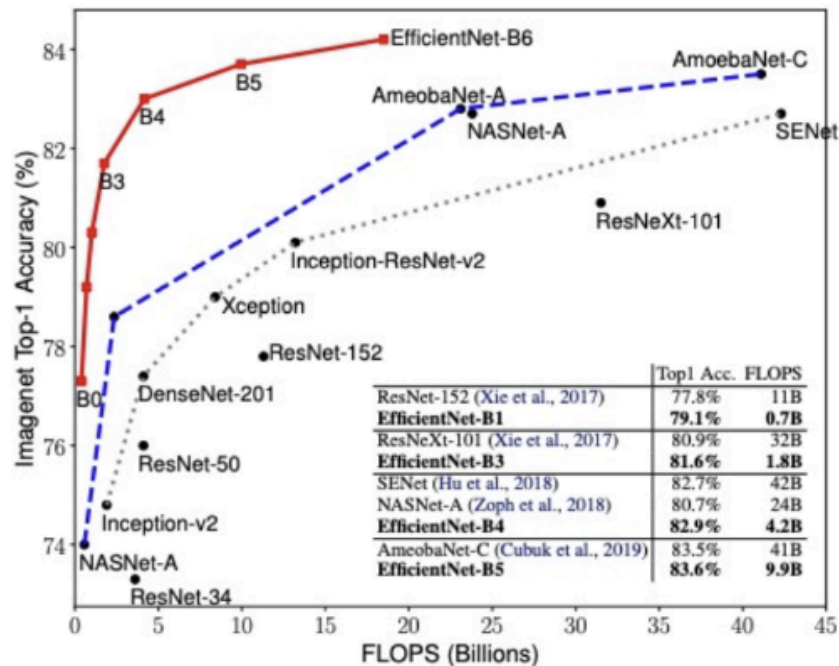


# EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks 논문 오픈소스 분석



- EfficientNet은 ImageNet 데이터셋에서 매우 높은 정확도를 기록했습니다.  
ImageNet에서 기존 ConvNet보다 8.4배 작으면서 6.1배 빠르고 더 높은 정확도를 얻었습니다. 예를 들어, EfficientNet-B4는 4.2B FLOPs(모델의 연산량)로 82.9%의 정확도를 기록하며, 이는 NASNet-A와 유사한 정확도이지만 연산량은 절반 이하입니다.
- EfficientNet은 Compound Scaling을 통해 모델의 크기, 깊이, 너비를 균형 있게 조절하여, 비슷한 정확도를 달성하면서도 다른 모델에 비해 훨씬 적은 연산량(FLOPs)과 파라미터를 사용합니다. EfficientNet-B0 모델은 5.3M 파라미터와 0.39B FLOPs를 사용하면서도 높은 성능을 보입니다.

Github link : <https://github.com/tensorflow/tpu/tree/master/models/official/efficientnet>

## 코드 실행 과정

0. 데이터셋을 불러오기 전에 Google Colab에 mount를 하고, 필요한 라이브러리를 import 하겠습니다.

```
[ ] from google.colab import drive
    drive.mount('efficientnet')
```

```
[ ]
    # import package

    # model
    import torch
    import torch.nn as nn
    import torch.nn.functional as F
    from torchsummary import summary
    from torch import optim

    # dataset and transformation
    from torchvision import datasets
    import torchvision.transforms as transforms
    from torch.utils.data import DataLoader
    from torchvision import models
    import os

    # display images
    from torchvision import utils
    import matplotlib.pyplot as plt
    %matplotlib inline

    # utils
    import numpy as np
    from torchsummary import summary
    import time
    import copy
```

## 1. 데이터셋 불러오기

- 데이터셋은 **torchvision** 패키지에서 제공하는 **STL10 dataset**을 이용하겠습니다.  
**STL10 dataset**은 10개의 **label**을 갖으며 **train dataset 5000개**, **test dataset 8000개**로 구성됩니다.

```
[ ] # specify path to data
path2data = '/content/efficientnet/MyDrive/data'

# if not exists the path, make the directory
if not os.path.exists(path2data):
    os.mkdir(path2data)

# load dataset
train_ds = datasets.STL10(path2data, split='train', download=True, transform=transforms.ToTensor())
val_ds = datasets.STL10(path2data, split='test', download=True, transform=transforms.ToTensor())

print(len(train_ds))
print(len(val_ds))
```

Files already downloaded and verified  
Files already downloaded and verified  
5000  
8000

```
[ ] # define transformation
transformation = transforms.Compose([
    transforms.ToTensor(),
    transforms.Resize(224)
])

# apply transformation to dataset
train_ds.transform = transformation
val_ds.transform = transformation

# make dataloader
train_dl = DataLoader(train_ds, batch_size=32, shuffle=True)
val_dl = DataLoader(val_ds, batch_size=32, shuffle=True)
```

- 랜덤하게 선택된 샘플 이미지를 그리드로 만들어 시각화하여 데이터의 형태와 레이블을 확인합니다.  
이 작업을 통해 데이터셋의 전처리가 제대로 되었는지, 이미지 크기가 맞는지 확인합니다.

```
[ ] # check sample images
def show(img, y=None):
    npimg = img.numpy()
    npimg_tr = np.transpose(npimg, (1, 2, 0))
    plt.imshow(npimg_tr)

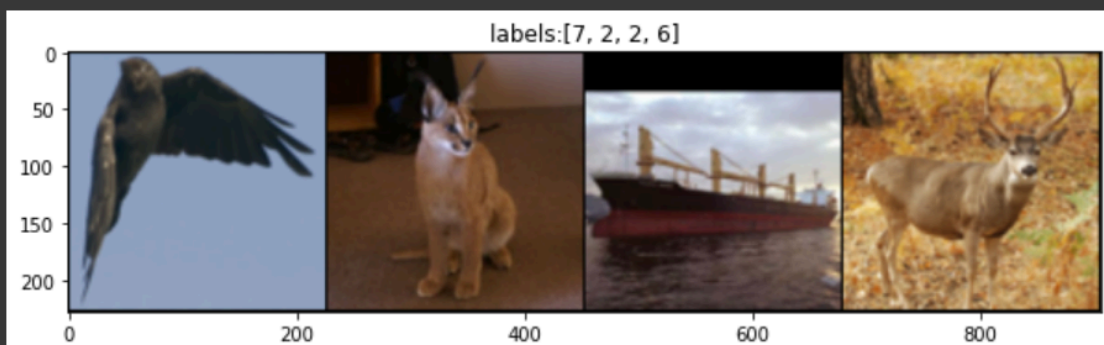
    if y is not None:
        plt.title('labels:' + str(y))

np.random.seed(10)
torch.manual_seed(0)

grid_size=4
rnd_ind = np.random.randint(0, len(train_ds), grid_size)

x_grid = [train_ds[i][0] for i in rnd_ind]
y_grid = [val_ds[i][1] for i in rnd_ind]

x_grid = utils.make_grid(x_grid, nrow=grid_size, padding=2)
plt.figure(figsize=(10,10))
show(x_grid, y_grid)
```



## 2. 모델 구축하기

코드는 <https://github.com/zsef123/EfficientNets-PyTorch/blob/master/models/effnet.py> 를 참고했습니다.

<https://github.com/katsura-jp/efficientnet-pytorch/blob/master/model/efficientnet.py>

### - Swish Activation

Swish 함수는 입력값에 sigmoid를 곱하여 비선형성을 적용하는 활성화 함수입니다.

### - SEBlock


SEBlock은 채널별 중요도를 학습하여 중요한 채널을 강조하고 덜 중요한 채널을

억제하는 모듈입니다. **Squeeze** 단계에서 채널별 평균값을 구하고, **Excitation** 단계에서 중요도를 계산해 입력 텐서에 적용합니다.

```
[ ] # Swish activation function
class Swish(nn.Module):
    def __init__(self):
        super().__init__()
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        return x * self.sigmoid(x)

# check
if __name__ == '__main__':
    x = torch.randn(3, 3, 224, 224)
    model = Swish()
    output = model(x)
    print('output size:', output.size())
```


 output size: torch.Size([3, 3, 224, 224])

```
[ ] # SE Block
class SEBlock(nn.Module):
    def __init__(self, in_channels, r=4):
        super().__init__()

        self.squeeze = nn.AdaptiveAvgPool2d((1,1))
        self.excitation = nn.Sequential(
            nn.Linear(in_channels, in_channels * r),
            Swish(),
            nn.Linear(in_channels * r, in_channels),
            nn.Sigmoid()
        )

    def forward(self, x):
        x = self.squeeze(x)
        x = x.view(x.size(0), -1)
        x = self.excitation(x)
        x = x.view(x.size(0), x.size(1), 1, 1)
        return x

# check
if __name__ == '__main__':
    x = torch.randn(3, 56, 17, 17)
    model = SEBlock(x.size(1))
    output = model(x)
    print('output size:', output.size())
```

 output size: torch.Size([3, 56, 1, 1])

- MBConv

MBConv는 EfficientNet의 기본 빌딩 블록으로, Depthwise Convolution과 SEBlock을 결합하여 채널별 정보를 강화합니다. 확장 비율로 채널을 늘린 뒤, SEBlock을 적용하고 최종적으로 입력 크기로 다시 축소합니다.

```
1 class MBConv(nn.Module):
2     expand = 6
3     def __init__(self, in_channels, out_channels, kernel_size, stride=1, se_scale=4, p=0.5):
4         super().__init__()
5         # first MBConv is not using stochastic depth
6         self.p = torch.tensor(p).float() if (in_channels == out_channels) else torch.tensor(1).float()
7
8         self.residual = nn.Sequential(
9             nn.Conv2d(in_channels, in_channels * MBConv.expand, 1, stride=stride, padding=0, bias=False),
10            nn.BatchNorm2d(in_channels * MBConv.expand, momentum=0.99, eps=1e-3),
11            Swish(),
12            nn.Conv2d(in_channels * MBConv.expand, in_channels * MBConv.expand, kernel_size=kernel_size,
13                    stride=1, padding=kernel_size//2, bias=False, groups=in_channels*MBConv.expand),
14            nn.BatchNorm2d(in_channels * MBConv.expand, momentum=0.99, eps=1e-3),
15            Swish()
16        )
17
18        self.se = SEBlock(in_channels * MBConv.expand, se_scale)
19
20        self.project = nn.Sequential(
21            nn.Conv2d(in_channels*MBConv.expand, out_channels, kernel_size=1, stride=1, padding=0, bias=False),
22            nn.BatchNorm2d(out_channels, momentum=0.99, eps=1e-3)
23        )
24
25        self.shortcut = (stride == 1) and (in_channels == out_channels)
26
27    def forward(self, x):
28        # stochastic depth
29        if self.training:
30            if not torch.bernoulli(self.p):
31                return x
32
33        x_shortcut = x
34        x_residual = self.residual(x)
35        x_se = self.se(x_residual)
36
37        x = x_se * x_residual
38        x = self.project(x)
39
40        if self.shortcut:
41            x = x_shortcut + x
42
43        return x
44
45    # check
46    if __name__ == '__main__':
47        x = torch.randn(3, 16, 24, 24)
48        model = MBConv(x.size(1), x.size(1), 3, stride=1, p=1)
49        model.train()
50        output = model(x)
51        x = (output == x)
52        print('output size:', output.size(), 'Stochastic depth:', x[1,0,0,0])
53
```

출력값 : output size: torch.Size([3, 16, 24, 24]) Stochastic depth: tensor(False)

## - SepConv

SepConv는 Depthwise Convolution과 SEBlock을 결합해 채널별 정보를 학습하는 모듈입니다.

입력과 출력 크기가 같으면 잔차 연결(residual connection)을 통해 성능을 보강합니다.

```
1 class SepConv(nn.Module):
2     expand = 1
3     def __init__(self, in_channels, out_channels, kernel_size, stride=1, se_scale=4, p=0.5):
4         super().__init__()
5         # first SepConv is not using stochastic depth
6         self.p = torch.tensor(p).float() if (in_channels == out_channels) else torch.tensor(1).float()
7
8         self.residual = nn.Sequential(
9             nn.Conv2d(in_channels * SepConv.expand, in_channels * SepConv.expand, kernel_size=kernel_size,
10                      stride=1, padding=kernel_size//2, bias=False, groups=in_channels*SepConv.expand),
11             nn.BatchNorm2d(in_channels * SepConv.expand, momentum=0.99, eps=1e-3),
12             Swish()
13         )
14
15         self.se = SEBlock(in_channels * SepConv.expand, se_scale)
16
17         self.project = nn.Sequential(
18             nn.Conv2d(in_channels*SepConv.expand, out_channels, kernel_size=1, stride=1, padding=0, bias=False),
19             nn.BatchNorm2d(out_channels, momentum=0.99, eps=1e-3)
20         )
21
22         self.shortcut = (stride == 1) and (in_channels == out_channels)
23
24     def forward(self, x):
25         # stochastic depth
26         if self.training:
27             if not torch.bernoulli(self.p):
28                 return x
29
30         x_shortcut = x
31         x_residual = self.residual(x)
32         x_se = self.se(x_residual)
33
34         x = x_se * x_residual
35         x = self.project(x)
36
37         if self.shortcut:
38             x = x_shortcut + x
39
40         return x
41
42 # check
43 if __name__ == '__main__':
44     x = torch.randn(3, 16, 24, 24)
45     model = SepConv(x.size(1), x.size(1), 3, stride=1, p=1)
46     model.train()
47     output = model(x)
48     # stochastic depth check
49     x = (output == x)
50     print('output size:', output.size(), 'Stochastic depth:', x[1,0,0,0])
```

출력값 : output size: torch.Size([3, 16, 24, 24]) Stochastic depth: tensor(False)

## EfficientNet 클래스 요약 설명

- 클래스 개요

이 클래스는 EfficientNet 모델을 구현하며, 사용자 지정 파라미터(num\_classes, width\_coef, depth\_coef, 등)를 통해 모델의 폭, 깊이, 확장 비율을 조정할 수 있습니다.
- 초기화 함수 (`__init__`)
  - 채널, 반복 횟수 및 스트라이드 설정

각 스테이지별 채널 크기, 반복 횟수, 스트라이드, 커널 크기를 정의하고, width\_coef와 depth\_coef로 이를 조정합니다.
  - **Stochastic Depth**

stochastic\_depth를 활성화하면, 레이어 드롭 확률을 단계적으로 계산하여 적용합니다.
  - 모듈 생성
    - 초기 업샘플링 레이어 (Upsample)
    - 첫 번째 스테이지: 일반 컨볼루션
    - 나머지 스테이지: SepConv 또는 MBConv를 사용하는 블록 반복 구조
    - 최종 스테이지: 1x1 컨볼루션과 활성화 함수 Swish
    - 풀링, 드롭아웃, 선형 계층을 통해 출력 클래스 수에 맞게 매핑
- **Forward** 함수
  - 입력 이미지를 업샘플링 후, 9개의 스테이지를 거쳐 최종 출력 확률을 계산합니다.
- **\_make\_Block** 함수
  - block 타입(SepConv 또는 MBConv)의 레이어를 반복하여 스테이지를 생성합니다.
  - 첫 번째 레이어만 지정된 스트라이드를 적용하고, 이후에는 1로 설정.



```

1 class EfficientNet(nn.Module):
2     def __init__(self, num_classes=10, width_coef=1., depth_coef=1., scale=1., dropout=0.2, se_scale=4, stochastic_depth=False, p=0.5):
3         super().__init__()
4         channels = [32, 16, 24, 40, 80, 112, 192, 320, 1280]
5         repeats = [1, 2, 2, 3, 3, 4, 1]
6         strides = [1, 2, 2, 2, 1, 2, 1]
7         kernel_size = [3, 3, 5, 3, 5, 5, 3]
8         depth = depth_coef
9         width = width_coef
10
11         channels = [int(x*width) for x in channels]
12         repeats = [int(x*depth) for x in repeats]
13
14         # stochastic depth
15         if stochastic_depth:
16             self.p = p
17             self.step = (1 - 0.5) / (sum(repeats) - 1)
18         else:
19             self.p = 1
20             self.step = 0
21
22         # efficient net
23         self.upsample = nn.Upsample(scale_factor=scale, mode='bilinear', align_corners=False)
24
25         self.stage1 = nn.Sequential(
26             nn.Conv2d(3, channels[0], 3, stride=2, padding=1, bias=False),
27             nn.BatchNorm2d(channels[0], momentum=0.99, eps=1e-3)
28         )
29
30         self.stage2 = self._make_Block(SepConv, repeats[0], channels[0], channels[1], kernel_size[0], strides[0], se_scale)
31
32         self.stage3 = self._make_Block(MBConv, repeats[1], channels[1], channels[2], kernel_size[1], strides[1], se_scale)
33
34         self.stage4 = self._make_Block(MBConv, repeats[2], channels[2], channels[3], kernel_size[2], strides[2], se_scale)
35
36         self.stage5 = self._make_Block(MBConv, repeats[3], channels[3], channels[4], kernel_size[3], strides[3], se_scale)
37
38         self.stage6 = self._make_Block(MBConv, repeats[4], channels[4], channels[5], kernel_size[4], strides[4], se_scale)
39
40         self.stage7 = self._make_Block(MBConv, repeats[5], channels[5], channels[6], kernel_size[5], strides[5], se_scale)
41
42         self.stage8 = self._make_Block(MBConv, repeats[6], channels[6], channels[7], kernel_size[6], strides[6], se_scale)
43
44         self.stage9 = nn.Sequential(
45             nn.Conv2d(channels[7], channels[8], 1, stride=1, bias=False),
46             nn.BatchNorm2d(channels[8], momentum=0.99, eps=1e-3),
47             Swish()
48         )
49
50         self.avgpool = nn.AdaptiveAvgPool2d((1,1))
51         self.dropout = nn.Dropout(p=dropout)
52         self.linear = nn.Linear(channels[8], num_classes)
53
54     def forward(self, x):
55         x = self.upsample(x)
56         x = self.stage1(x)
57         x = self.stage2(x)
58         x = self.stage3(x)
59         x = self.stage4(x)
60         x = self.stage5(x)
61         x = self.stage6(x)
62         x = self.stage7(x)
63         x = self.stage8(x)
64         x = self.stage9(x)
65         x = self.avgpool(x)
66         x = x.view(x.size(0), -1)
67         x = self.dropout(x)
68         x = self.linear(x)
69         return x
70
71     def _make_Block(self, block, repeats, in_channels, out_channels, kernel_size, stride, se_scale):
72
73         strides = [stride] + [1] * (repeats - 1)
74         layers = []
75         for stride in strides:
76             layers.append(block(in_channels, out_channels, kernel_size, stride, se_scale, self.p))
77             in_channels = out_channels
78             self.p -= self.step
79
80         return nn.Sequential(*layers)
81
82

```

## - EfficientNet Variants 함수

efficientnet\_b0부터 efficientnet\_b7까지 EfficientNet의 8가지 버전을 생성합니다.

각 함수는 width\_coef, depth\_coef, scale, dropout 등 다양한 파라미터로 모델의 폭, 깊이, 입력 크기 스케일링을 조정합니다.

- B0: 기본 모델 (폭: 1.0, 깊이: 1.0)
- B7: 가장 큰 모델 (폭: 2.0, 깊이: 3.1, 더 큰 입력 크기, 더 높은 드롭아웃)

```
1 def efficientnet_b0(num_classes=10):
2     return EfficientNet(num_classes=num_classes, width_coef=1.0, depth_coef=1.0, scale=1.0, dropout=0.2, se_scale=4)
3
4 def efficientnet_b1(num_classes=10):
5     return EfficientNet(num_classes=num_classes, width_coef=1.0, depth_coef=1.1, scale=240/224, dropout=0.2, se_scale=4)
6
7 def efficientnet_b2(num_classes=10):
8     return EfficientNet(num_classes=num_classes, width_coef=1.1, depth_coef=1.2, scale=260/224., dropout=0.3, se_scale=4)
9
10 def efficientnet_b3(num_classes=10):
11     return EfficientNet(num_classes=num_classes, width_coef=1.2, depth_coef=1.4, scale=300/224, dropout=0.3, se_scale=4)
12
13 def efficientnet_b4(num_classes=10):
14     return EfficientNet(num_classes=num_classes, width_coef=1.4, depth_coef=1.8, scale=380/224, dropout=0.4, se_scale=4)
15
16 def efficientnet_b5(num_classes=10):
17     return EfficientNet(num_classes=num_classes, width_coef=1.6, depth_coef=2.2, scale=456/224, dropout=0.4, se_scale=4)
18
19 def efficientnet_b6(num_classes=10):
20     return EfficientNet(num_classes=num_classes, width_coef=1.8, depth_coef=2.6, scale=528/224, dropout=0.5, se_scale=4)
21
22 def efficientnet_b7(num_classes=10):
23     return EfficientNet(num_classes=num_classes, width_coef=2.0, depth_coef=3.1, scale=600/224, dropout=0.5, se_scale=4)
24
25
26 # check
27 if __name__ == '__main__':
28     device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
29     x = torch.randn(3, 3, 224, 224).to(device)
30     model = efficientnet_b0().to(device)
31     output = model(x)
32     print('output size:', output.size())
```

출력값 : output size: torch.Size([3, 10])

- 모델 초기화 및 장치 설정

`efficientnet_b0()` 모델을 초기화하고, 학습에 사용할 장치(CPU/GPU)로 모델을 이동시킵니다.

- 모델 구조 출력

`summary()` 함수를 사용하여 입력 크기 (3, 224, 224)에 대한 모델의 구조, 계층별 파라미터 수 및 총 파라미터 수를 출력합니다.

이를 통해 모델의 구조와 복잡도를 확인할 수 있습니다.

```
[ ] # print model summary
model = efficientnet_b0().to(device)
summary(model, (3,224,224), device=device.type)
```



Layer (type)	Output Shape	Param #
-----	-----	-----
Upsample-1	[-1, 3, 224, 224]	0
Conv2d-2	[-1, 32, 112, 112]	864
BatchNorm2d-3	[-1, 32, 112, 112]	64
Conv2d-4	[-1, 32, 112, 112]	288
BatchNorm2d-5	[-1, 32, 112, 112]	64
Sigmoid-6	[-1, 32, 112, 112]	0
Swish-7	[-1, 32, 112, 112]	0
AdaptiveAvgPool2d-8	[-1, 32, 1, 1]	0
Linear-9	[-1, 128]	4,224
Sigmoid-10	[-1, 128]	0
Swish-11	[-1, 128]	0
Linear-12	[-1, 32]	4,128
Sigmoid-13	[-1, 32]	0
SEBlock-14	[-1, 32, 1, 1]	0
Conv2d-15	[-1, 16, 112, 112]	512
BatchNorm2d-16	[-1, 16, 112, 112]	32
SepConv-17	[-1, 16, 112, 112]	0
Conv2d-18	[-1, 96, 56, 56]	1,536
BatchNorm2d-19	[-1, 96, 56, 56]	192
Sigmoid-20	[-1, 96, 56, 56]	0
Swish-21	[-1, 96, 56, 56]	0
Conv2d-22	[-1, 96, 56, 56]	864
BatchNorm2d-23	[-1, 96, 56, 56]	192
Sigmoid-24	[-1, 96, 56, 56]	0

### 3. 학습하기

- 손실 함수 및 최적화 정의

`CrossEntropyLoss`를 사용하여 손실을 계산하며, **Adam** 최적화 기법과 학습률 조정 스케줄러(`ReduceLROnPlateau`)를 적용합니다.

- 미니배치 손실 및 정확도 계산

미니배치 단위로 손실을 계산하고 역전파를 통해 가중치를 업데이트하며, 예측 정확도를 계산합니다.

- 에포크 단위 손실 및 검증

에포크 전체에 대해 훈련 및 검증 데이터를 통해 손실과 정확도를 계산합니다.

- 모델 훈련 및 저장

가장 낮은 검증 손실 기준으로 최적의 가중치를 저장하며, 학습률 조정을 통해 성능을 최적화합니다.

```

1 # define loss function, optimizer, lr_scheduler
2 loss_func = nn.CrossEntropyLoss(reduction='sum')
3 opt = optim.Adam(model.parameters(), lr=0.01)
4
5 from torch.optim.lr_scheduler import ReduceLROnPlateau
6 lr_scheduler = ReduceLROnPlateau(opt, mode='min', factor=0.1, patience=10)
7
8
9 # get current lr
10 def get_lr(opt):
11     for param_group in opt.param_groups:
12         return param_group['lr']
13
14
15 # calculate the metric per mini-batch
16 def metric_batch(output, target):
17     pred = output.argmax(1, keepdim=True)
18     corrects = pred.eq(target.view_as(pred)).sum().item()
19     return corrects
20
21
22 # calculate the loss per mini-batch
23 def loss_batch(loss_func, output, target, opt=None):
24     loss_b = loss_func(output, target)
25     metric_b = metric_batch(output, target)
26
27     if opt is not None:
28         opt.zero_grad()
29         loss_b.backward()
30         opt.step()
31
32     return loss_b.item(), metric_b
33
34
35 # calculate the loss per epochs
36 def loss_epoch(model, loss_func, dataset_dl, sanity_check=False, opt=None):
37     running_loss = 0.0
38     running_metric = 0.0
39     len_data = len(dataset_dl.dataset)
40
41     for xb, yb in dataset_dl:
42         xb = xb.to(device)
43         yb = yb.to(device)
44         output = model(xb)
45
46         loss_b, metric_b = loss_batch(loss_func, output, yb, opt)
47
48         running_loss += loss_b
49
50         if metric_b is not None:
51             running_metric += metric_b
52
53         if sanity_check is True:
54             break
55
56     loss = running_loss / len_data
57     metric = running_metric / len_data
58     return loss, metric
59
60
61 # function to start training
62 def train_val(model, params):
63     num_epochs=params['num_epochs']
64     loss_func=params['loss_func']
65     opt=params['optimizer']
66     train_dl=params['train_dl']
67     val_dl=params['val_dl']
68     sanity_check=params['sanity_check']
69     lr_scheduler=params['lr_scheduler']
70     path2weights=params['path2weights']
71
72     loss_history = {'train': [], 'val': []}
73     metric_history = {'train': [], 'val': []}
74
75     best_loss = float('inf')
76     best_model_wts = copy.deepcopy(model.state_dict())
77     start_time = time.time()
78
79     for epoch in range(num_epochs):
80         current_lr = get_lr(opt)
81         print('Epoch {}/{}'.format(epoch, num_epochs-1, current_lr))
82
83         model.train()
84         train_loss, train_metric = loss_epoch(model, loss_func, train_dl, sanity_check, opt)
85         loss_history['train'].append(train_loss)
86         metric_history['train'].append(train_metric)
87
88         model.eval()
89         with torch.no_grad():
90             val_loss, val_metric = loss_epoch(model, loss_func, val_dl, sanity_check)
91         loss_history['val'].append(val_loss)
92         metric_history['val'].append(val_metric)
93
94         if val_loss < best_loss:
95             best_loss = val_loss
96             best_model_wts = copy.deepcopy(model.state_dict())
97             torch.save(model.state_dict(), path2weights)
98             print('Copied best model weights!')
99
100         lr_scheduler.step(val_loss)
101         if current_lr != get_lr(opt):
102             print('Loading best model weights!')
103             model.load_state_dict(best_model_wts)
104
105     print('train loss: %.6f, val loss: %.6f, accuracy: %.2f, time: %.4f min' % (train_loss, val_loss, 100*val_metric, (time.time()-start_time)/60))
106     print('-'*10)
107
108     model.load_state_dict(best_model_wts)
109     return model, loss_history, metric_history

```

- 모델 학습을 위한 설정을 정의하고 필요한 디렉토리를 생성하여 학습을 준비하는 코드입니다.

```
[ ] # define the training parameters
params_train = {
    'num_epochs':100,
    'optimizer':opt,
    'loss_func':loss_func,
    'train_dl':train_dl,
    'val_dl':val_dl,
    'sanity_check':False,
    'lr_scheduler':lr_scheduler,
    'path2weights':'./models/weights.pt',
}

# check the directory to save weights.pt
def createFolder(directory):
    try:
        if not os.path.exists(directory):
            os.makedirs(directory)
    except OSError:
        print('Error')
createFolder('./models')

[ ] model, loss_hist, metric_hist = train_val(model, params_train)
```

```
model, loss_hist, metric_hist = train_val(model, params_train)

Epoch 0/19, current lr= 0.01
Copied best model weights!
train loss: 1.507311, val loss: 2.022092, accuracy: 29.46, time: 0.5537 min
-----

Epoch 1/19, current lr= 0.01
Copied best model weights!
train loss: 1.502268, val loss: 1.713452, accuracy: 33.74, time: 1.1052 min
-----

Epoch 2/19, current lr= 0.01
train loss: 1.433284, val loss: 1.850826, accuracy: 34.46, time: 1.6430 min
-----

Epoch 3/19, current lr= 0.01
Copied best model weights!
train loss: 1.410226, val loss: 1.549302, accuracy: 38.41, time: 2.1956 min
-----

Epoch 4/19, current lr= 0.01
train loss: 1.381581, val loss: 1.884781, accuracy: 36.20, time: 2.7360 min
-----

Epoch 5/19, current lr= 0.01
train loss: 1.331858, val loss: 1.987151, accuracy: 38.57, time: 3.2770 min
-----

Epoch 6/19, current lr= 0.01
train loss: 1.282874, val loss: 1.943976, accuracy: 40.51, time: 3.8162 min
-----

Epoch 7/19, current lr= 0.01
Loading best model weights!
train loss: 1.238939, val loss: 2.035488, accuracy: 41.15, time: 4.3620 min
-----

Epoch 8/19, current lr= 0.001
train loss: 1.261815, val loss: 1.815406, accuracy: 38.90, time: 4.9073 min
```

```
Epoch 8/19, current lr= 0.001
train loss: 1.261815, val loss: 1.815406, accuracy: 38.90, time: 4.9073 min
-----

Epoch 9/19, current lr= 0.001
Copied best model weights!
train loss: 1.160707, val loss: 1.412762, accuracy: 47.58, time: 5.4557 min
-----

Epoch 10/19, current lr= 0.001
train loss: 1.115870, val loss: 1.814790, accuracy: 43.15, time: 5.9984 min
-----

Epoch 11/19, current lr= 0.001
Copied best model weights!
train loss: 1.093127, val loss: 1.326927, accuracy: 49.76, time: 6.5489 min
-----

Epoch 12/19, current lr= 0.001
train loss: 1.061364, val loss: 1.632945, accuracy: 44.77, time: 7.0912 min
-----

Epoch 13/19, current lr= 0.001
train loss: 1.039094, val loss: 1.539156, accuracy: 46.35, time: 7.6331 min
-----

Epoch 14/19, current lr= 0.001
train loss: 1.001268, val loss: 1.504423, accuracy: 47.33, time: 8.1757 min
-----

Epoch 15/19, current lr= 0.001
train loss: 0.966988, val loss: 1.548535, accuracy: 45.30, time: 8.7132 min
-----

Epoch 16/19, current lr= 0.001
train loss: 0.926754, val loss: 1.696836, accuracy: 47.27, time: 9.2518 min
-----

Epoch 17/19, current lr= 0.001
train loss: 0.870007, val loss: 1.743425, accuracy: 44.82, time: 9.7910 min
-----

Epoch 18/19, current lr= 0.001
train loss: 0.838045, val loss: 1.575955, accuracy: 47.94, time: 10.3278 min
```

#### 4. 성능 평가하기

Train-Val Loss는 초기 몇 Epoch 동안 빠르게 감소하며 모델이 학습됨을 보여줍니다.

하지만 Epoch 12 이후 검증 손실이 증가하는 과적합 경향이 나타납니다. 이는 모델이 훈련 데이터에 적합했지만 검증 데이터에 대한 일반화가 부족함을 의미합니다.

