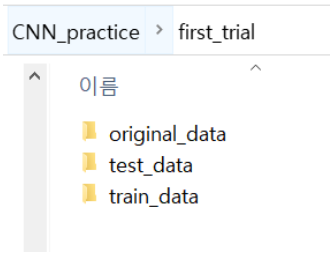
20.05.25

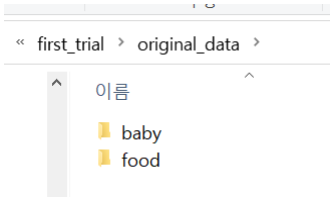
주제 **VGG (Visual Geometry Group)**

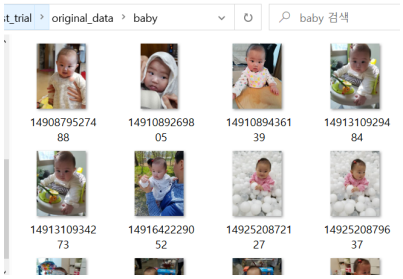
자신의 사진을 로드해서 학습에 사용하는 방법!

Google Drive 이용하기 (Colab 환경)

Google Drive에 나의 파일 올리기

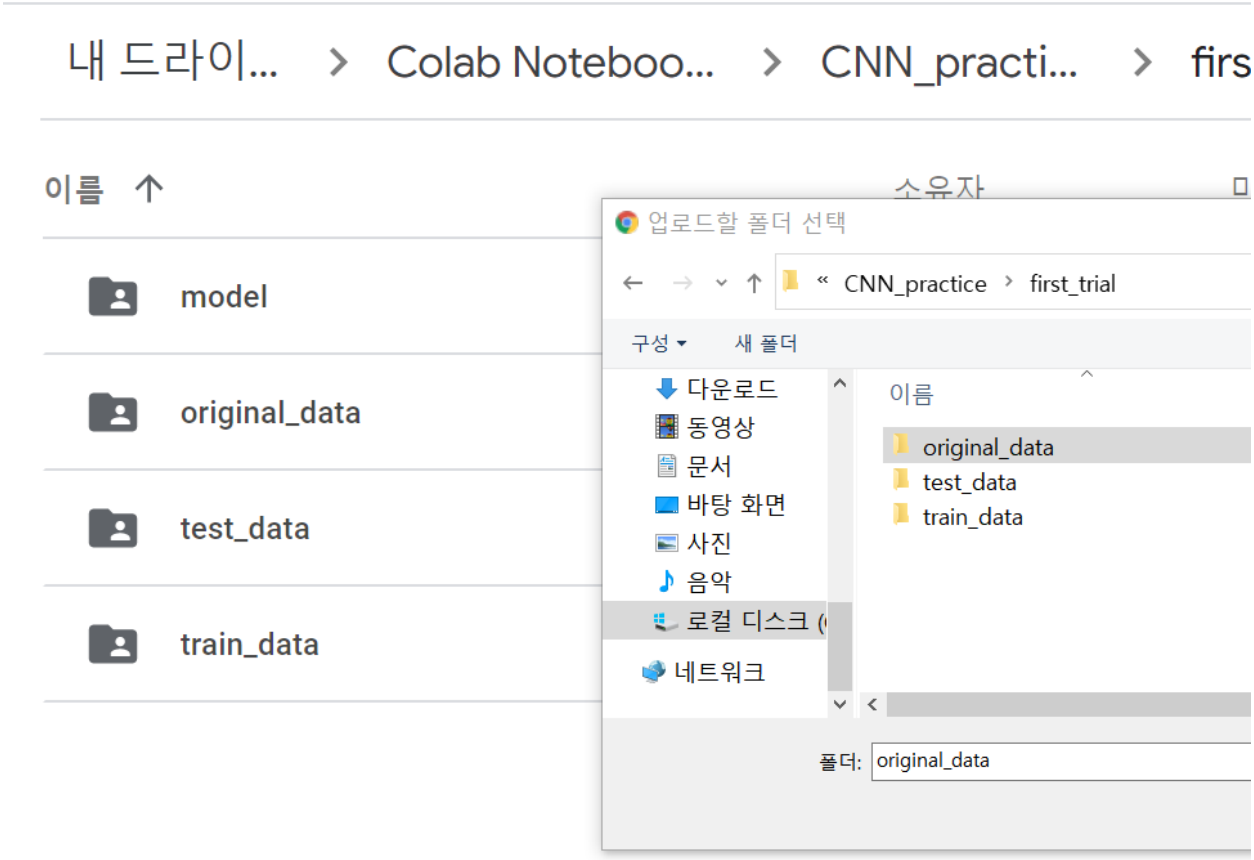






프로젝트 폴더 > data 폴더를 train_data, test_data로 나누어 각각 생성합니다.

- 원하는 이미지 분류에 따라 하부 파일로 나누어 주어야 합니다.



원하는 디렉토리로 가서 우클릭 > 폴더 업로드를 선택합니다.

구글 드라이브에 해당 파일을 모두 올렸다면, 사용할 준비가 끝났습니다.

Google Drive에 저장한 파일을 Colab에 연동시키기 위한 준비

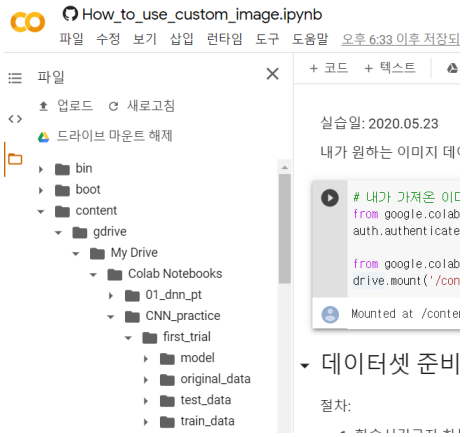
```
from google.colab import auth
auth.authenticate_user()

from google.colab import drive
drive.mount('/content/gdrive', force_remount=True)
```

위 코드블럭을 실행해 colab과 G-Drive를 연동시킵니다.

Google Drive에서 Path 구하기

Colab의 파일 쪽에서 gdrive 경로를 찾아준 다음, 폴더 옆에 커서를 갖다 대면 점 세 개가 뜨는데,

그걸 누르고 **'경로복사'** 를 선택해주면 파일까지의 경로를 알아낼 수 있습니다.



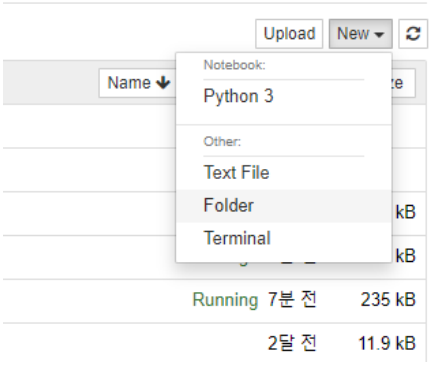
ImageFolder 함수의 **root** 인자에 방금 복사한 경로를 Ctrl+V로 붙여넣기 하면 해당 폴더에서 데이터를 가져오도록 할 수 있습니다.

data를 불러올 때는 사진을 분류한 하위 폴더가 아닌, 그 모든 분류를 묶어 준 폴더를 경로로 지정해야 합니다.

PyTorch 데이터를 가져올 때 사진을 폴더 순서대로 자동으로 label을 매기기 때문입니다.

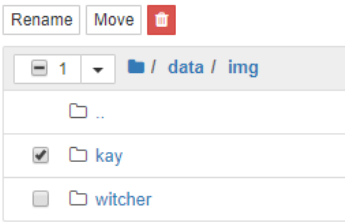
```
project_path = '/content/gdrive/My Drive/Colab Notebooks/CNN_practice/first_trial/original_data'
train_data = torchvision.datasets.ImageFolder(root = project_path , transform=trans)
```

Jupyter Notebook 폴더에 직접 업로드

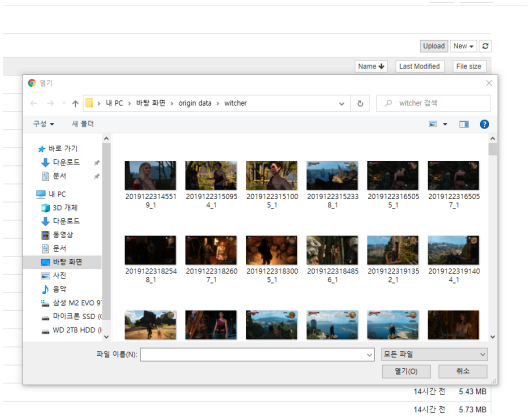


주피터 노트북 우측 상단

new > folder 로 폴더 생성

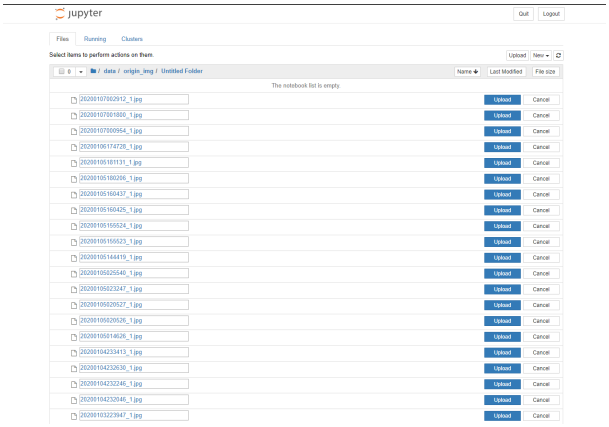


폴더를 체크하고 좌측상단의 Rename을 클릭,폴더 이름을 변경.



폴더로 들어가

우측상단 upload 클릭하여 이미지 업로드



근데 업로드 버튼을 일일이 눌러줘야 된다;

\ 0ㅁ0 /

+) 검색해보니 폴더 전체를 업로드하려면 zip 파일로 압축해준 후 업로드하고 따로 풀어야 한다고 하네요.

ㄴ 역시 머리가 나쁘면 몸이 고생하는구만유..

VGG (Visual Geometry Group)

VGG Net의 구조

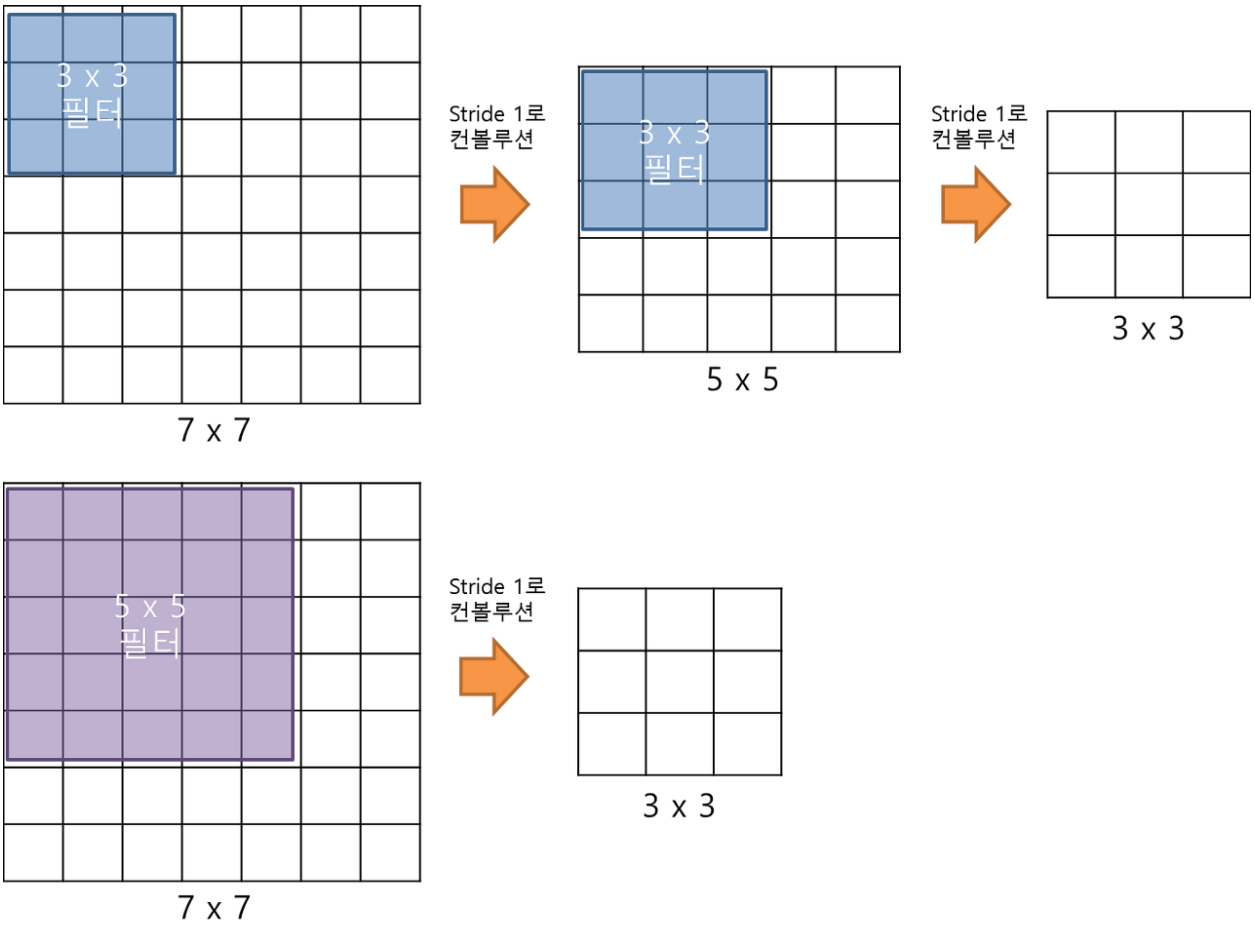
torchvision.models.vgg는 vgg11~19까지 만들 수 있도록 되어있으며, **3×224×224 입력을 기준**으로 만들도록 되어있다.

VGG 연구팀은 깊이의 영향만을 최대한 확인하고자 컨볼루션 필터커널의 사이즈는 가장 작은 3 x 3으로 고정했다. 필터커널의 사이즈가 크면 그만큼 이미지의 사이즈가 금방 축소되기 때문에 네트워크의 깊이를 충분히 깊게 만들기 불가능하기 때문이다.

VGG 연구팀은 original 논문에서 총 6개의 구조(A, A-LRN, B, C, D, E)를 만들어 성능을 비교했다. 여러 구조를 만든 이유는 기본적으로 깊이의 따른 성능 변화를 비교하기 위함이다. 이중 D 구조가 VGG16이고 E 구조가 VGG19라고 보면 된다.

ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

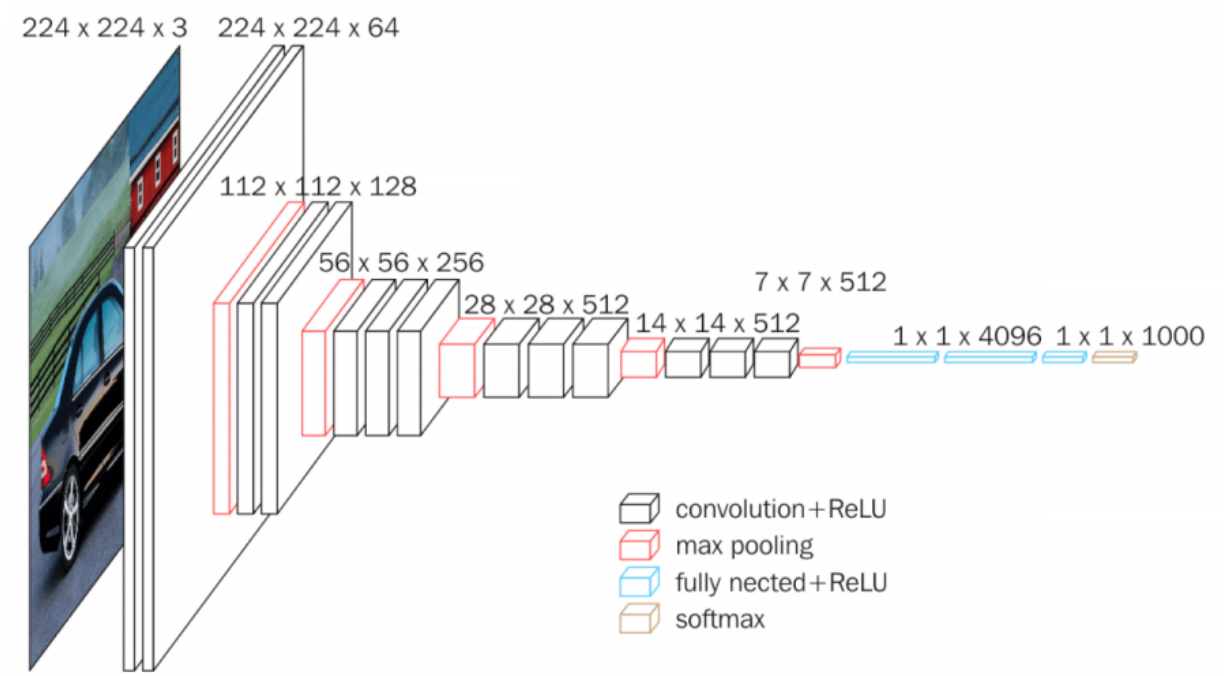
VGGNet의 구조를 깊이 들여다보기에 앞서 먼저 집고 넘어가야할 것이 있다. 그것은 바로 3 x 3 필터로 두 차례 컨볼루션을 하는 것과 5 x 5 필터로 한 번 컨볼루션을 하는 것이 결과적으로 동일한 사이즈의 특성맵을 산출한다는 것이다(아래 그림 참고). 3 x 3 필터로 세 차례 컨볼루션 하는 것은 7 x 7 필터로 한 번 컨볼루션 하는 것과 대응된다.



그러면 3 x 3 필터로 세 차례 컨볼루션을 하는 것이 7 x 7 필터로 한 번 컨볼루션하는 것보다 나은 점은 무엇일까? 일단 **가중치 또는 파라미터의 갯수의 차이**다. 3 x 3 필터가 3개면 총 27개의 가중치를 갖는다. 반면 7 x 7 필터는 49개의 가중치를 갖는다. CNN에서 가중치는 모두 훈련이 필요한 것들이므로, 가중치가 적다는 것은 그만큼 훈련시켜야할 것의 갯수가 작아진다. 따라서 학습의 속도가 빨라진다. 동시에 층의 갯수가 늘어나면서 특성에 **비선형성을 더 증가**시키기 때문에 특성이 점점 더 유용해진다.

VGG16 구조 분석

그러면 이제 VGG16(D 구조)를 예시로, 각 층마다 어떻게 특성맵이 생성되고 변화되는지 자세하게 살펴보자. 아래 구조도와 함께 각 층의 세부사항을 읽어나가면 이해하기가 그렇게 어렵지 않을 것이다.



0) 인풋: 224 x 224 x 3 이미지(224 x 224 RGB 이미지)를 입력받을 수 있다.

1) 1층(conv1_1): 64개의 3 x 3 x 3 필터커널로 입력이미지를 컨볼루션해준다. zero padding은 1만큼 해줬고, 컨볼루션 보폭(stride)는 1로 설정해준다. zero padding과 컨볼루션 stride에 대한 설정은 모든 컨볼루션층에서 모두 동일하니 다음 층부터는 설명을 생략하겠다. 결과적으로 64장의 224 x 224 특성맵(224 x 224 x 64)들이 생성된다. 활성화시키기 위해 ReLU 함수가 적용된다. ReLU함수는 마지막 16층을 제외하고는 항상 적용되니 이 또한 다음 층부터는 설명을 생략하겠다.

2) 2층(conv1_2): 64개의 3 x 3 x 64 필터커널로 특성맵을 컨볼루션해준다. 결과적으로 64장의 224 x 224 특성맵들(224 x 224 x 64)이 생성된다. 그 다음에 2 x 2 최대 풀링을 stride 2로 적용함으로 특성맵의 사이즈를 112 x 112 x 64로 줄인다.

*conv1_1, conv1_2와 conv2_1, conv2_2등으로 표현한 이유는 해상도를 줄여주는 최대 풀링 전까지의 층등을 한 모듈로 볼 수 있기 때문이다.

3) 3층(conv2_1): 128개의 3 x 3 x 64 필터커널로 특성맵을 컨볼루션해준다. 결과적으로 128장의 112 x 112 특성맵들(112 x 112 x 128)이 산출된다.

4) 4층(conv2_2): 128개의 3 x 3 x 128 필터커널로 특성맵을 컨볼루션해준다. 결과적으로 128장의 112 x 112 특성맵들(112 x 112 x 128)이 산출된다. 그 다음에 2 x 2 최대 풀링을 stride 2로 적용해준다. 특성맵의 사이즈가 56 x 56 x 128로 줄어들었다.

5) 5층(conv3_1): 256개의 3 x 3 x 128 필터커널로 특성맵을 컨볼루션한다. 결과적으로 256장의 56 x 56 특성맵들(56 x 56 x 256)이 생성된다.

6) 6층(conv3_2): 256개의 3 x 3 x 256 필터커널로 특성맵을 컨볼루션한다. 결과적으로 256장의 56 x 56 특성맵들(56 x 56 x 256)이 생성된다.

7) 7층(conv3_3): 256개의 3 x 3 x 256 필터커널로 특성맵을 컨볼루션한다. 결과적으로 256장의 56 x 56 특성맵들(56 x 56 x 256)이 생성된다. 그 다음에 2 x 2 최대 풀링을 stride 2로 적용한다. 특성맵의 사이즈가 28 x 28 x 256으로 줄어들었다.

8) 8층(conv4_1): 512개의 3 x 3 x 256 필터커널로 특성맵을 컨볼루션한다. 결과적으로 512장의 28 x 28 특성맵들(28 x 28 x 512)이 생성된다.

9) 9층(conv4_2): 512개의 3 x 3 x 512 필터커널로 특성맵을 컨볼루션한다. 결과적으로 512장의 28 x 28 특성맵들(28 x 28 x 512)이 생성된다.

10) 10층(conv4_3): 512개의 3 x 3 x 512 필터커널로 특성맵을 컨볼루션한다. 결과적으로 512장의 28 x 28 특성맵들(28 x 28 x 512)이 생성된다. 그 다음에 2 x 2 최대 풀링을 stride 2로 적용한다. 특성맵의 사이즈가 14 x 14 x 512로 줄어든다.

11) 11층(conv5_1): 512개의 3 x 3 x 512 필터커널로 특성맵을 컨볼루션한다. 결과적으로 512장의 14 x 14 특성맵들(14 x 14 x 512)이 생성된다.

12) 12층(conv5_2): 512개의 3 x 3 x 512 필터커널로 특성맵을 컨볼루션한다. 결과적으로 512장의 14 x 14 특성맵들(14 x 14 x 512)이 생성된다.

13) 13층(conv5-3): 512개의 3 x 3 x 512 필터커널로 특성맵을 컨볼루션한다. 결과적으로 512장의 14 x 14 특성맵들(14 x 14 x 512)이 생성된다. 그 다음에 2 x 2 최대 풀링을 stride 2로 적용한다. 특성맵의 사이즈가 7 x 7 x 512로 줄어든다.

14) 14층(fc1): 7 x 7 x 512의 특성맵을 flatten 해준다. flatten이라는 것은 전 층의 출력을 받아서 단순히 1차원의 벡터로 펼쳐주는 것을 의미한다. 결과적으로 7 x 7 x 512 = 25088개의 뉴런이 되고, fc1층의 4096개의 뉴런과 fully connected 된다. 훈련시 dropout이 적용된다.

15) 15층(fc2): 4096개의 뉴런으로 구성해준다. fc1층의 4096개의 뉴런과 fully connected 된다. 훈련시 dropout이 적용된다.

16) 16층(fc3): 1000개의 뉴런으로 구성된다. fc2층의 4096개의 뉴런과 fully connected된다. 출력값들은 softmax 함수로 활성화된다. 1000개의 뉴런으로 구성되었다는 것은 1000개의 클래스로 분류하는 목적으로 만들어진 네트워크란 뜻이다.

```
import torch.nn as nn
import torch.utils.model_zoo as model_zoo
```

```
__all__ = [
    'VGG', 'vgg11', 'vgg11_bn', 'vgg13', 'vgg13_bn', 'vgg16', 'vgg16_bn',
    'vgg19_bn', 'vgg19',
]


model_urls = {
    'vgg11': 'https://download.pytorch.org/models/vgg11-bbd30ac9.pth',
    'vgg13': 'https://download.pytorch.org/models/vgg13-c768596a.pth',
    'vgg16': 'https://download.pytorch.org/models/vgg16-397923af.pth',
    'vgg19': 'https://download.pytorch.org/models/vgg19-dcbb9e9d.pth',
    'vgg11_bn': 'https://download.pytorch.org/models/vgg11_bn-6002323d.pth',
    'vgg13_bn': 'https://download.pytorch.org/models/vgg13_bn-abd245e5.pth',
    'vgg16_bn': 'https://download.pytorch.org/models/vgg16_bn-6c64b313.pth',
    'vgg19_bn': 'https://download.pytorch.org/models/vgg19_bn-c79401a0.pth',
}
```

```
class VGG(nn.Module):
    def __init__(self, features, num_classes=1000, init_weights=True):
        super(VGG, self).__init__()

        self.features = features # 쌓아나가야 될 VGG-net의 convolution layer들

        self.avgpool = nn.AdaptiveAvgPool2d((7, 7))

        # 3개의 fully connected layer에 대한 내용
        self.classifier = nn.Sequential(
            nn.Linear(512 * 7 * 7, 4096),# image size가 다른상태에서 적용하려면 이부분을 꼭 수정하기!!!!
            nn.ReLU(True),
            nn.Dropout(),
            nn.Linear(4096, 4096),
            nn.ReLU(True),
            nn.Dropout(),
            nn.Linear(4096, num_classes),
        )

        if init_weights:
            self._initialize_weights()

    def forward(self, x):
        x = self.features(x) #Convolution
        x = self.avgpool(x) # avgpool
        x = x.view(x.size(0), -1) #
        x = self.classifier(x) #FC layer
        return x

    def _initialize_weights(self):
        for m in self.modules():
```

```
            # 만약 m 이 nn.Conv2d일 때
            if isinstance(m, nn.Conv2d): #위에 feature가 넘겨줬던 layer를 m에게 하나씩 return
                # filter 의 weight값을 kaiming(=He) nomalization
                nn.init.kaiming_normal_(m.weight, mode='fan_out', nonlinearity='relu')
                if m.bias is not None:
                    # VGG-net에서 bias 값은 0
                    nn.init.constant_(m.bias, 0)
            elif isinstance(m, nn.BatchNorm2d):
                # m 이 BatchNorm2d일 경우
                nn.init.constant_(m.weight, 1)
                nn.init.constant_(m.bias, 0)
            elif isinstance(m, nn.Linear):
                # m이 Linear function일 경우
                nn.init.normal_(m.weight, 0, 0.01)
                nn.init.constant_(m.bias, 0)
```

```
# 'A': [64, 'M', 128, 'M', 256, 256, 'M', 512, 512, 'M', 512, 512, 'M']

def make_layers(cfg, batch_norm=False):
    layers = []
    in_channels = 3

    for v in cfg:
        if v == 'M':
            layers += [nn.MaxPool2d(kernel_size=2, stride=2)]
        else:
            conv2d = nn.Conv2d(in_channels, v, kernel_size=3, padding=1)
            if batch_norm:# 만약 True일 경우 실행
                layers += [conv2d, nn.BatchNorm2d(v), nn.ReLU(inplace=True)]
            else:# False면 아래 layer값이 추가
                layers += [conv2d, nn.ReLU(inplace=True)]
            ############## 중요!! ############
            in_channels = v

    return nn.Sequential(*layers)
```

in_channels = v가 중요한 이유는 다음과 같다. 첫 실행에서 v는 64고, conv2d = nn.Conv2d(3, 64, kernel_size=3, padding=1)이걸 실행하게 된다. 이 Conv2d를 통과하면 Conv2d를 통과하면 channel 수가 64로 변경된다. 다음 채널에서 in_channel을 64로 받아야하기 때문데 위의 in_channel을 v로 바꿔주는 것이다.

```
cfg = {
    'A': [64, 'M', 128, 'M', 256, 256, 'M', 512, 512, 'M', 512, 512, 'M'], # conv layer 8 + fc 3 =11 == vgg11
    'B': [64, 64, 'M', 128, 128, 'M', 256, 256, 'M', 512, 512, 'M', 512, 512, 'M'], # 10 + 3 = vgg 13
    'D': [64, 64, 'M', 128, 128, 'M', 256, 256, 256, 'M', 512, 512, 512, 'M', 512, 512, 512, 'M'], #13 + 3 = vgg 16
    'E': [64, 64, 'M', 128, 128, 'M', 256, 256, 256, 256, 'M', 512, 512, 512, 512, 'M', 512, 512, 512, 512, 'M'], # 16 +3 =vgg 19
    # 내가 원하는 대로 작성
    'custom' : [64,64,64,'M',128,128,128,'M',256,256,256,'M']
}
```

▼ 'A'로 레이어를 생성한 결과

Sequential((0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)) (1): ReLU(inplace=True) (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False) (3): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)) (4): ReLU(inplace=True) (5): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False) (6): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)) (7): ReLU(inplace=True) (8): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)) (9): ReLU(inplace=True) (10): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False) (11): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)) (12): ReLU(inplace=True) (13): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)) (14): ReLU(inplace=True) (15): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False) (16): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)) (17): ReLU(inplace=True) (18): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)) (19): ReLU(inplace=True) (20): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
)