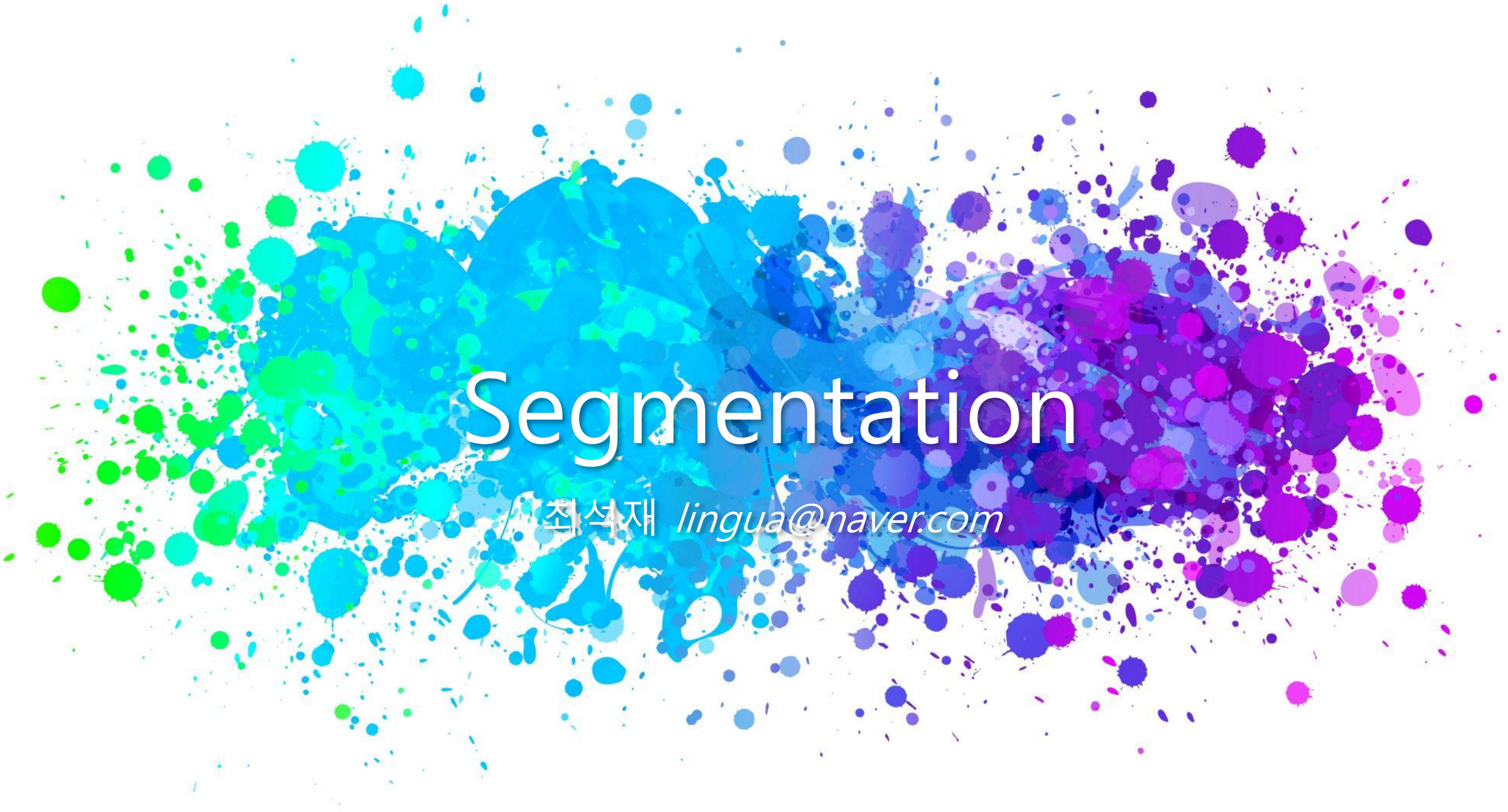


Segmentation

/ 최석재 *lingua@naver.com*

Image Segmentaion

- 이미지 분할은 이미지를 의미있는 여러 부분으로 나누는 것을 말한다
- 이미지에 있는 객체를 식별하는 것은 물론, 위치와 형태를 파악한다
- 의료 분야에서는 종양의 정확한 위치와 사이즈를 파악하는 데 사용할 수 있다

- 단점으로는 픽셀 단위에서 예측이 이루어지는 복잡한 계산을 수행하기 때문에
- 시간과 자원을 많이 소모하고,
- 지도학습 알고리즘으로서 상세한 레이블링이 있는 더 많은 데이터가 필요하다는 점이다
- 특히 훈련 데이터에 크게 의존하기 때문에 학습 데이터를 구축하는 비용이 많이 든다
  - Roboflow의 Semantic Segmentation에서 시범적으로 라벨링 서비스를 하고 있다

FCN *Fully Convolutional Networks*

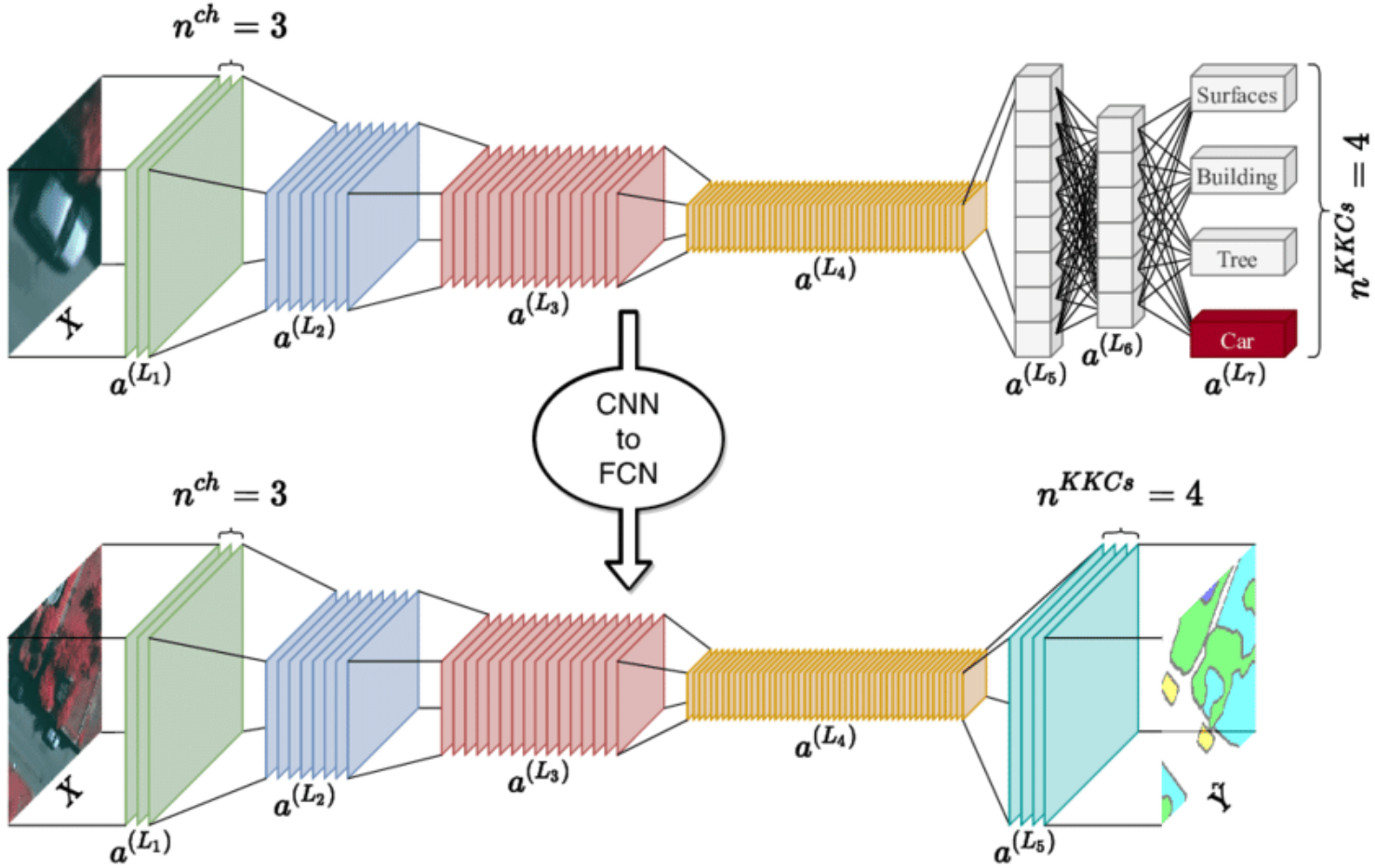
- CNN은 보통 분류를 위하여 마지막에 완전연결층을 사용하는데,
- 단점은 데이터를 하나의 덩어리로 처리한다는 것
- 공간적인 배열을 무시하고, 모든 픽셀 정보를 하나의 긴 벡터로 변환함으로써
- 전체적인 패턴을 인식하고 분류할 수는 있지만
- 이미지 내 객체의 정확한 위치나 형태를 파악하기 어렵게 된다

- 이러한 방법은 이미지 분할과 같이 픽셀 단위의 예측을 수행할 때에는 한계가 있다
- 이에 완전연결 층을 전혀 사용하지 않고, 모든 층을 합성곱 층으로 구성한 것이 FCN.
- 합성곱 층만을 사용함으로 공간정보를 유지할 수 있게 되어 픽셀단위 예측이 가능해진다

- 기본 원리는 CNN의 여러 합성곱 레이어를 이용하여 특징을 추출하고,
- 마지막 CNN 레이어에는 Softmax 활성화 함수를 이용하여 분류하는 것이다

CNN vs FCN

- FCN은 CNN의 완전연결층을 제거하고, 해상도를 증가시키는 업샘플링 레이어를 둔다



U-Net

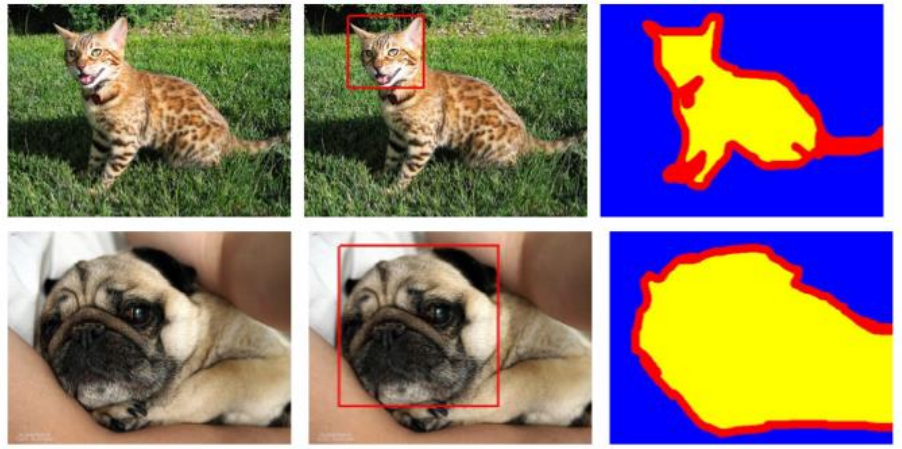
- 이미지 분할에서는 FCN과 함께 스킵 연결을 사용한다
- FCN은 업샘플링을 수행하는데, 이미지 분할은 여기에 인코더 – 디코더 구조를 만들어
- 인코더에서는 다운샘플링을 수행하게 하고, 디코더에서 업샘플링을 수행하게 한다
- 인코더 레이어의 출력을, 이를 입력으로 받을 수 있는 디코더의 대응 레이어에 직접 연결하는 것을 스킵 연결이라고 한다

- 스킵 연결을 사용하면 인코더에서 추출된 저수준 정보(예: 질감)와 디코더에서 추출된 고수준 정보(예: 구조)를 결합할 수 있다
- 또한 인코더에서 추출된 특징 맵을 디코더로 직접 전달함으로써, 손실된 공간적 정보를 업샘플링 과정에서 복원할 수 있다

- 이와 같이 스킵 연결은 정보의 손실을 줄이고, 더 풍부한 정보를 제공함으로써 모델의 학습 능력을 향상시킨다

Pix2Pix는 U-Net의 확장 버전으로,
GAN 구조를 추가하여 이미지를 변환하는 기능을 갖게한 것

파일 다운로드



https://www.robots.ox.ac.uk/~vgg/data/pets/

- 이번 실습은 Oxford-IIIT Pet Dataset로서 대량의 파일로서
- 구글 드라이브와 연결하여 파일 로딩을 하지 않고,
- Colab 상에서만 다운로드 하여 사용하는 방식을 취한다

파일 다운로드

- !wget https://www.robots.ox.ac.uk/~vgg/data/pets/data/images.tar.gz
- !wget https://www.robots.ox.ac.uk/~vgg/data/pets/data/annotations.tar.gz

압축 해제

- !tar -xf images.tar.gz
- !tar -xf annotations.tar.gz

여기서 사용된 코드는 텐서플로 문서를 참조하였음
https://keras.io/examples/vision/oxford_pets_image_segmentation/

데이터 확인

- import os

현재 작업 디렉토리 출력

- current_dir = os.getcwd()
- print("Current working directory:", current_dir)

- !ls # annotations, images 폴더가 있다

- print(os.listdir("images")) # images 폴더의 내용을 출력해본다

```
Current working directory: /content
annotations  annotations.tar.gz  images  images.tar.gz  sample_data
['Maine_Coon_204.jpg', 'japanese_chin_49.jpg', 'Egyptian_Mau_102.jpg', 'Maine_Coon_157.jpg', 'beagle_23.jpg', 'great_pyrenees_171.jpg', 'american_bulldog_14
```

마스크 이미지 확인

- from PIL import Image
- import numpy as np

- annotation_file = 'annotations/trimaps/Abyssinian_1.png'
- annotation_image = Image.open(annotation_file) # PIL 파일
- annotation_array = np.array(annotation_image) # 넘파이 배열로 변환

- unique_values = np.unique(annotation_array) # 고유한 픽셀 값 확인
- print("Unique values in the annotation image:", unique_values)

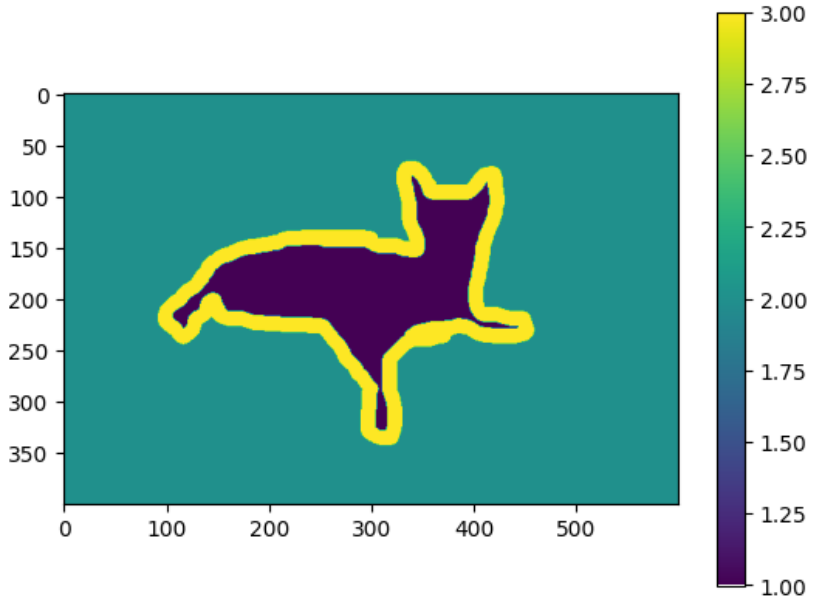
```
Unique values in the annotation image: [1 2 3]
```

1, 2, 3 의 3개 값이 있다
1은 경계, 2는 배경, 3은 객체

마스크 이미지 시각화

- import matplotlib.pyplot as plt

- plt.imshow(annotation_array, cmap='viridis')
- plt.colorbar()
- plt.show()



고양이 이미지

배열 출력

- 앞의 넘파이 배열을 값으로 출력해본다

\# numpy 배열 전체를 출력할 수 있도록 설정

- np.set_printoptions(threshold=np.inf, linewidth=np.inf)
- print(annotation_array)

```
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
3 3 3 3 3 3 1 1 1 1 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2

2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
3 3 3 3 3 3 1 1 1 1 1 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2

2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
3 3 3 3 3 3 1 1 1 1 1 1 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2

2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
```

배경(2)이 많은 부분을 차지하고 있다

경로 설정

- input_dir = "images/" # 입력 이미지 저장 경로
- target_dir = "annotations/trimaps/" # 타겟인 마스크 이미지 저장 경로
- img_size = (160, 160) # 결과 이미지 사이즈
- num_classes = 3 # 분류 클래스의 수
- batch_size = 32

Dataset Statistics

Breed	Count
American Bulldog	200
American Pit Bull Terrier	200
Basset Hound	200
Beagle	200
Boxer	199
Chihuahua	200
English Cocker Spaniel	196
English Setter	200
German Shorthaired	200
Great Pyrenees	200
Havanese	200
Japanese Chin	200
Keeshond	199
Leonberger	200
Miniature Pinscher	200
Newfoundland	196
Pomeranian	200
Pug	200
Saint Bernard	200
Samyoed	200
Scottish Terrier	199
Shiba Inu	200
Staffordshire Bull Terrier	189
Wheaten Terrier	200
Yorkshire Terrier	200
Total	4978

1.Dog Breeds

Breed	Count
Abyssinian	198
Bengal	200
Birman	200
Bombay	200
British Shorthair	184
Egyptian Mau	200
Main Coon	190
Persian	200
Ragdoll	200
Russian Blue	200
Siamese	199
Sphynx	200
Total	2371

2.Cat Breeds

Family	Count
Cat	2371
Dog	4978
Total	7349

3.Total Pets

Oxford-IIIT Pet Dataset은 2개 대분류, 37개 소분류로 되어 있다
여기서는 대분류 Cat, Dog만 사용하며, 추가하여 배경(2, Background)도 분류 종류에 속하므로 총 3 분류이다

경로 설정

```
input_img_paths = sorted(                                   # ④ 리스트 컴프리헨션으로 생성된 파일 리스트를 오름차순으로 정렬
    [                                                       # 리스트 컴프리헨션 시작
        os.path.join(input_dir, fname)                      # ③ 조건을 만족하는 fname에 대해 input_dir와 결합하여 전체 경로 생성
        for fname in os.listdir(input_dir)                  # ① 경로 내 모든 파일 이름을 리스트로 만듦
        if fname.endswith(".jpg")                           # ② 파일 이름이 ".jpg"로 끝나는 파일만 선택
    ]
)
target_img_paths = sorted(
    [
        os.path.join(target_dir, fname)
        for fname in os.listdir(target_dir)
        if fname.endswith(".png") and not fname.startswith(".")     # ".png"로 끝나면서 "."으로 시작하지 않는 파일
    ]
)
```

타겟인 마스크 이미지는 png 타입으로 되어 있다
"."으로 시작하는 파일은 숨김 파일이므로 제외한다

경로 설정

- 샘플의 수를 확인한다
- 입력 이미지와 마스크 이미지의 수는 동일하다

- print("입력 샘플의 수:", len(input_img_paths)) # 7390
- print("타겟 샘플의 수:", len(target_img_paths)) # 7390

이미지 확인

- 입력 이미지와 마스크 이미지를 확인한다

- from IPython.display import Image, display
- from keras.utils import load_img
- from PIL import ImageOps

- display(Image(filename=input_img_paths[9])) # 하나의 입력 이미지를 출력한다

위의 입력 이미지에 대응되는 마스크 이미지를 출력한다
마스크 이미지의 데이터가 1, 2, 3처럼 비슷한 값으로 되어 있어 display()로 바로 그리면 형태가 구분되지 않는다
autocontrast()로 명암을 자동 조정하여 형태를 잘 구분할 수 있도록 한다

- img = ImageOps.autocontrast(load_img(target_img_paths[9]))
- display(img)

데이터셋 생성 (1/3)

- 입력 이미지와 마스크 이미지를 로드하고,
- 전처리한 후 배치 형태로 변환한다

- import keras
- import numpy as np
- from tensorflow import data as tf_data
- from tensorflow import image as tf_image
- from tensorflow import io as tf_io

데이터셋 생성 (2/3)

- def get_dataset(batch_size, img_size, input_img_paths, target_img_paths, max_dataset_len=None):

-
```
    def load_img_masks(input_img_path, target_img_path):
        input_img = tf_io.read_file(input_img_path)                          # 파일 읽기
        input_img = tf_io.decode_jpeg(input_img, channels=3)                 # RGB 변환
        input_img = tf_image.resize(input_img, img_size)                     # 이미지 사이즈 변환
        input_img = tf_image.convert_image_dtype(input_img, "float32")       # float32 형으로 변환

        target_img = tf_io.read_file(target_img_path)
        target_img = tf_io.decode_png(target_img, channels=1)                # 마스크 이미지는 단일 채널로 되어 있음
        target_img = tf_image.resize(target_img, img_size, method="nearest") # 최근접 이웃 보간법을 이용한 이미지 리사이징
        target_img = tf_image.convert_image_dtype(target_img, "uint8")       # uint8 형 유지

        target_img -= 1                                                      # 딥러닝 모델에서는 클래스 레이블이 일반적으로 0부터
        return input_img, target_img                                         # 시작하므로 이에 맞추기 위하여 1, 2, 3으로 되어 있는 레이블에서
                                                                             # 1을 빼어 0, 1, 2로 만들어준다
```

tf.io.read_file()은 이미지 파일을 읽어 바이트 문자열로 구성된 텐서플로의 텐서로 변환한다
tf.io.decode_jpeg/png()는 인코딩된 바이트 문자열 텐서를 디코딩하여 (높이, 너비, 채널) 형식의 3차원 이미지 텐서로 변환한다

데이터셋 생성 (3/3)

```
if max_dataset_len:                                  # 디버깅 시에는 데이터 셋의 크기를 제한하여 빠른 진행이 가능하게 한다
    input_img_paths = input_img_paths[:max_dataset_len]
    target_img_paths = target_img_paths[:max_dataset_len]
# 입력 이미지 경로와 마스크 이미지 경로 리스트를 받아 텐서 슬라이스로 데이터셋을 생성
dataset = tf_data.Dataset.from_tensor_slices((input_img_paths, target_img_paths))
dataset = dataset.map(load_img_masks, num_parallel_calls=tf_data.AUTOTUNE)
return dataset.batch(batch_size)     # 데이터셋을 배치 형태로 묶어 반환
```

서브 함수인 load_img_masks()를 dataset의 각 요소에 적용
시스템 리소스에 따라 병렬 처리 수준을 자동으로 조정하여 수행한다

```python
def get_dataset(batch_size, img_size, input_img_paths, target_img_paths, max_dataset_len=None):

    def load_img_masks(input_img_path, target_img_path):
        input_img = tf_io.read_file(input_img_path)
        input_img = tf_io.decode_png(input_img, channels=3)
        input_img = tf_image.resize(input_img, img_size)
        input_img = tf_image.convert_image_dtype(input_img, "float32")

        target_img = tf_io.read_file(target_img_path)
        target_img = tf_io.decode_png(target_img, channels=1)
        target_img = tf_image.resize(target_img, img_size, method="nearest")
        target_img = tf_image.convert_image_dtype(target_img, "uint8")

        target_img -= 1
        print(f"Shape of target_img: {target_img.shape}")
        return input_img, target_img

    if max_dataset_len:
        input_img_paths = input_img_paths[:max_dataset_len]
        target_img_paths = target_img_paths[:max_dataset_len]
    dataset = tf_data.Dataset.from_tensor_slices((input_img_paths, target_img_paths))
    dataset = dataset.map(load_img_masks, num_parallel_calls=tf_data.AUTOTUNE)
    return dataset.batch(batch_size)
```

tf.data.Dataset.from_tensor_slices()

- tf.data.Dataset.from_tensor_slices()는 배열, 텐서, 리스트, 파일 경로 리스트를 입력으로 받아
- 개별 요소로 슬라이싱하여 텐서플로의 Dataset 타입으로 만든다

- import tensorflow as tf
- import numpy as np

- data = np.array([1, 2, 3, 4, 5])

numpy 배열 생성

- dataset = tf.data.Dataset.from_tensor_slices(data)
- print(type(dataset))

TensorFlow 데이터셋 생성

데이터셋의 각 요소 출력

- for element in dataset:
    print(element.numpy())

```
<class 'tensorflow.python.data.ops.from_tensor_slices_op._TensorSliceDataset'>
1
2
3
4
5
```

개별 원소가 Dataset이 되었다

모델 함수 (1/4)

- 모델을 생성하는 함수를 정의한다

- from keras import layers

- def get_model(img_size, num_classes):

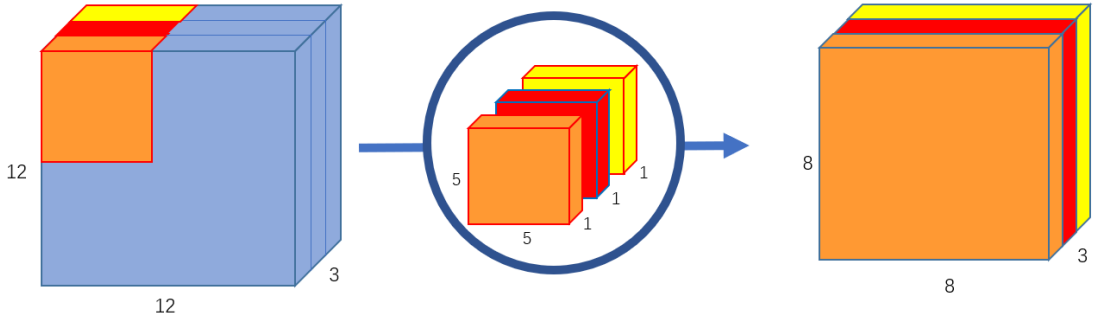
```
    inputs = keras.Input(shape=img_size + (3,))          # input은 이미지의 사이즈 + 채널 수가 되게 한다
                                                         # 이미지의 사이즈가 (150, 150)일 경우 + (3,)은 (150, 150, 3)이 된다

    # 다운샘플링 과정
    x = layers.Conv2D(32, 3, strides=2, padding="same")(inputs)   # 3x3 크기의 32개 필터, stride=2, padding='same'
    x = layers.BatchNormalization()(x)                            # 배치 정규화
    x = layers.Activation("relu")(x)                              # relu 활성화 함수 사용

    previous_block_activation = x                                 # 잔차 연결에 사용할 수 있도록 임시 저장
```

모델 함수 (2/4)

```
# 3번의 순회를 통해 3개의 블록을 만든다
for filters in [64, 128, 256]:                    # 첫 번째 블록(순회)은 64개 필터, 두 번째는 128개 필터, 세 번째는 256개 필터 사용
    x = layers.Activation("relu")(x)
    x = layers.SeparableConv2D(filters, 3, padding="same")(x)
    x = layers.BatchNormalization()(x)

    x = layers.Activation("relu")(x)
    x = layers.SeparableConv2D(filters, 3, padding="same")(x)          # 3x3 커널 사용
    x = layers.BatchNormalization()(x)

    x = layers.MaxPooling2D(3, strides=2, padding= " same " )(x)       # 3x3 풀링 창과 stride=2 사용

    residual = layers.Conv2D(filters, 1, strides=2, padding="same")(   # 1x1 필터를 stride=2로 사용
        previous_block_activation                                      # 잔차 연결을 할 수 있도록 임시 저장한 내용을 합성곱 연산하여
    )                                                                  # 크기를 맞춤
    x = layers.add([x, residual])                                      # 크기가 맞추어진 임시 저장 내용을 잔차 연결
    previous_block_activation = x                                      # 현재 블록의 출력을 저장하여 다음 블록의 잔차 연결에 사용
```

SeparableConv2D()

깊이별 분리 합성곱 (Depthwise Separable Convolution)



- 깊이별 분리 합성곱을 사용하는 이유는 비슷한 효과를 가지면서도 연산량을 줄일 수 있기 때문
- 깊이별 분리 합성곱은 다음의 두 단계를 수행한다

기본 합성곱에서 3x3 사이즈의 필터를 사용한다고 하면
필터도 3개의 채널이 있으므로 (3, 3, 3) 필터가 있는 것이나,

깊이별 합성곱에서는 (3, 3, 1) 필터가 있는 것
이 형상의 독립적인 필터를 채널별로 적용한다

1. 깊이별 합성곱 (Depthwise Convolution)

- 이 단계는 각 입력 채널별로 필터를 적용하여 별도의 2D 합성곱을 수행한다
- 따라서 채널 간의 상호 작용은 고려하지 않고, 공간적 정보만 추출한다

2. 점별 합성곱 (1x1 합성곱. Pointwise Convolution)

- 깊이별 합성곱의 출력을 1x1 합성곱을 사용하여 결합한다
- (1, 1, 3) 필터를 사용하므로 이 단계는 채널 간의 상호 작용이 고려된다

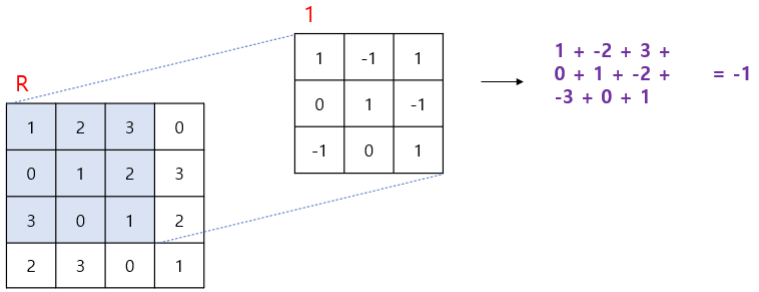
- 기본 합성곱에서는 각 필터가 각 채널에 대하여 합성곱 연산 후 더하는 과정을 갖지만,
- 깊이별 합성곱에서는 채널별로 2D 합성곱만 수행하기 때문에 연산량이 줄어든다
- 채널 간 상호 작용 효과는 두 번째 단계인 점별 합성곱을 통해 보완한다

※ 채널별 합성곱 연산은 Convolution 구현의 3차원 데이터의 합성곱 연산 참조

SeparableConv2D()

깊이별 분리 합성곱 (Depthwise Separable Convolution)

- 깊이별 분리 합성곱의 기본 합성곱과의 가장 큰 차이점은 점별 합성곱에만 필터의 수가 고려된다는 점이다
- 전체 영역이 아닌, 한 개 영역에 대한 연산만 생각해본다
- 3채널을 갖는 이미지에 3x3 사이즈의 필터를 32개 사용한다고 하였을 때,



- <u>기본 합성곱에서는</u> 첫 번째 채널에 3x3=9번의 연산이 이루어진다
- 이것이 RGB 채널별로 있게 되므로 9x3=27번의 연산이 있게 된다
- 필터의 수가 32개이므로, 이러한 연산이 32번 일어나 총 연산량은 32x27=**864** 이다

- <u>깊이별 분리 합성곱은</u> 깊이별 합성곱과 점별 합성곱의 두 단계로 나뉜다
- [1] 깊이별 합성곱에서는 (3, 3, 1)의 필터를 채널 수만큼 준비한다. 첫 번째 채널에 3x3=9번의 연산이 일어나고
- 이것이 RGB 채널별로 있게 되므로 9x3=27번의 연산이 있게 된다(여기까지는 연산량 동일). 깊이별 합성곱에서는 3채널 이미지에는 항상 3개의 필터가 있게 된다

- [2] 점별 합성곱 단계로 넘어간다. (1, 1, 3) 필터를 사용하여 합성곱하므로
- 첫 번째 채널에 1회 연산, 3채널 모두에 3번의 연산이 일어난다. (1, 1, 3) 필터이므로 3채널의 결과를 모두 더하므로 채널간 상호작용이 고려된다
- 이러한 (1, 1, 3) 필터를 이용한 점별 합성곱 연산을 32회 수행하여 3x32=96번의 연산이 일어난다
- 총 연산량은 27+96=**123** 이다

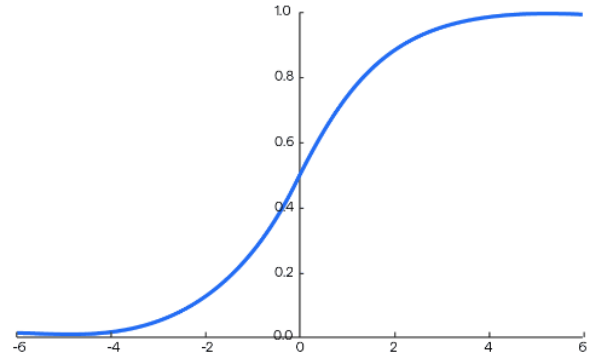
※ 필터가 (3, 3, 3) 사이즈일 때 이것은 3채널을 갖는 한 개 필터이다
깊이별 합성곱에서는 (3, 3, 1) 사이즈의 필터를 채널마다 준비하여 총 3개의 필터가 있게 된다

모델 함수 (3/4)

```
# 업샘플링 과정
for filters in [256, 128, 64, 32]:                                  # 업샘플링 단계에서 사용할 필터 수 설정 및 순회
    x = layers.Activation("relu")(x)
    x = layers.Conv2DTranspose(filters, 3, padding="same")(x)       # 역합성곱 함수를 통해 업샘플링
    x = layers.BatchNormalization()(x)

    x = layers.Activation("relu")(x)
    x = layers.Conv2DTranspose(filters, 3, padding="same")(x)
    x = layers.BatchNormalization()(x)
    x = layers.UpSampling2D(2)(x)                                   # 단순히 픽셀을 파라미터 size 배수로 복사하여 업샘플링

    # 잔차 연결
    residual = layers.UpSampling2D(2)(previous_block_activation)    # 이전 블록의 출력을 업샘플링
    residual = layers.Conv2D(filters, 1, padding="same")(residual)  # 1x1 합성곱을 사용하여 공간 정보 유지
    x = layers.add([x, residual])                                   # 현재 블록의 출력과 residual을 더하여 손실을 보완
    previous_block_activation = x                                   # 현재 블록의 출력을 저장하여 다음 블록의 잔차 연결에 사용
```

※ Conv2DTranspose()는 새로운 픽셀을 예측하는 방식으로 생성하여 이미지 품질이 유지되나,
UpSampling2D()는 새로운 정보를 생성없이, 각 픽셀값을 반복하여 크기를 늘리므로 이미지 품질이 떨어질 수 있지만, 계산 비용이 낮다

모델 함수 (4/4)




```
# Softmax 활성화 함수로 각 픽셀 위치마다 출력 채널의 값을 확률 분포로 변환한다
# 0, 1, 2 중 속할 확률이 계산된다
outputs = layers.Conv2D(num_classes, 3, activation="softmax", padding="same")(x)


# 모델 정의
model = keras.Model(inputs, outputs)
return model
```

완전연결층은 입력 데이터의 모든 요소가 출력 노드와 직접 연결되는 구조로 복잡한 패턴을 학습하는 데 유리하다
따라서 이미지 전체를 분류하는 데 좋은 성능을 보인다
하지만 데이터를 벡터로 푸는 과정에서 공간적인 정보가 사라진다 (예: 32x32 2D 데이터는 32x32=1024 벡터로 변환)

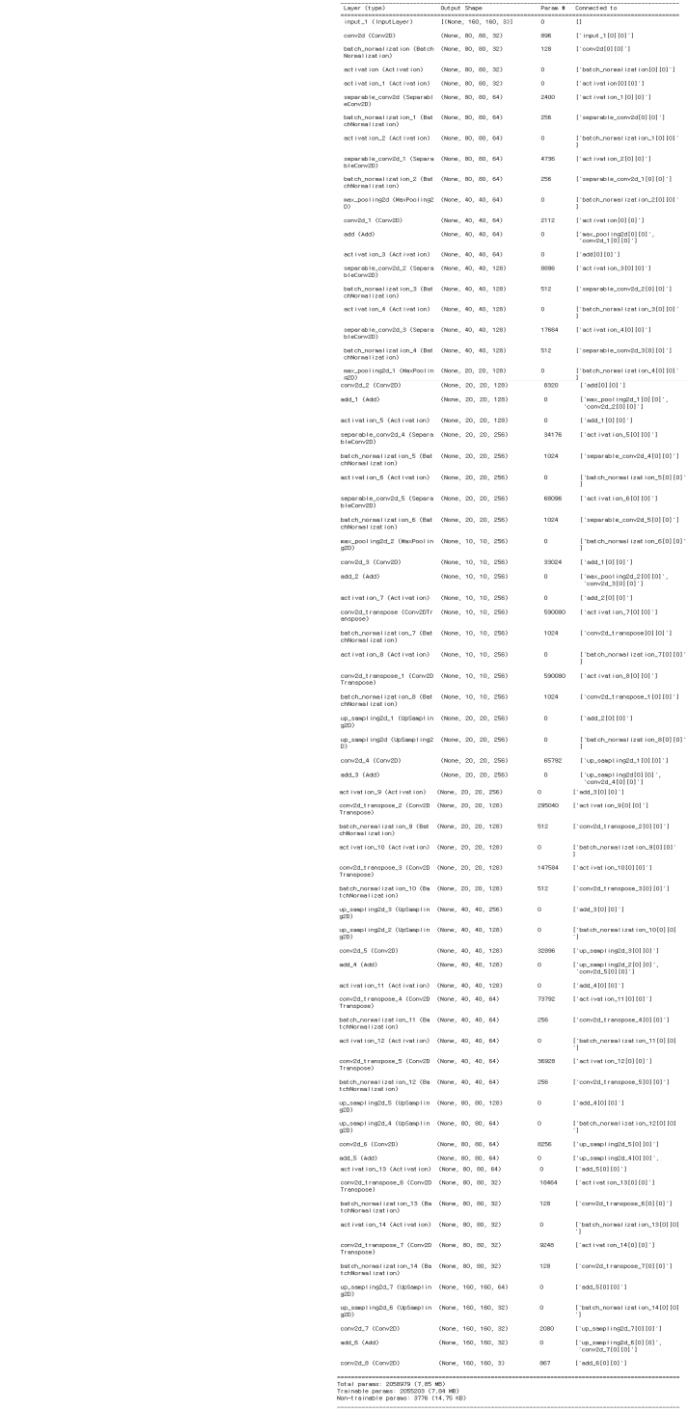
CNN은 합성곱 연산을 통해 필터 크기만큼의 입력 이미지만 처리되지만 공간적인 구조를 유지한다
따라서 이미지의 형태 정보를 파악하는 이미지 분할 작업에 더 좋은 성능을 보인다

모델 함수 전체 코드

```python
def get_model(img_size, num_classes):
    inputs = keras.Input(shape=img_size + (3,))

    # 다운샘플링 과정
    x = layers.Conv2D(32, 3, strides=2, padding="same")(inputs)
    x = layers.BatchNormalization()(x)
    x = layers.Activation("relu")(x)
    previous_block_activation = x

    # 3번의 순회를 통해 3개의 블록을 만든다
    for filters in [64, 128, 256]:
        x = layers.Activation("relu")(x)
        x = layers.SeparableConv2D(filters, 3, padding="same")(x)
        x = layers.BatchNormalization()(x)

        x = layers.Activation("relu")(x)
        x = layers.SeparableConv2D(filters, 3, padding="same")(x)
        x = layers.BatchNormalization()(x)

        x = layers.MaxPooling2D(3, strides=2, padding="same")(x)

        residual = layers.Conv2D(filters, 1, strides=2, padding="same")(
            previous_block_activation
        )
        x = layers.add([x, residual])
        previous_block_activation = x

    # 업샘플링 과정
    for filters in [256, 128, 64, 32]:
        x = layers.Activation("relu")(x)
        x = layers.Conv2DTranspose(filters, 3, padding="same")(x)
        x = layers.BatchNormalization()(x)

        x = layers.Activation("relu")(x)
        x = layers.Conv2DTranspose(filters, 3, padding="same")(x)
        x = layers.BatchNormalization()(x)

        x = layers.UpSampling2D(2)(x)

        residual = layers.UpSampling2D(2)(previous_block_activation)
        residual = layers.Conv2D(filters, 1, padding="same")(residual)
        x = layers.add([x, residual])
        previous_block_activation = x

    outputs = layers.Conv2D(num_classes, 3, activation="softmax", padding="same")(x)

    model = keras.Model(inputs, outputs)
    return model
```

모델 생성

- model = get_model(img_size, num_classes)
- model.summary()



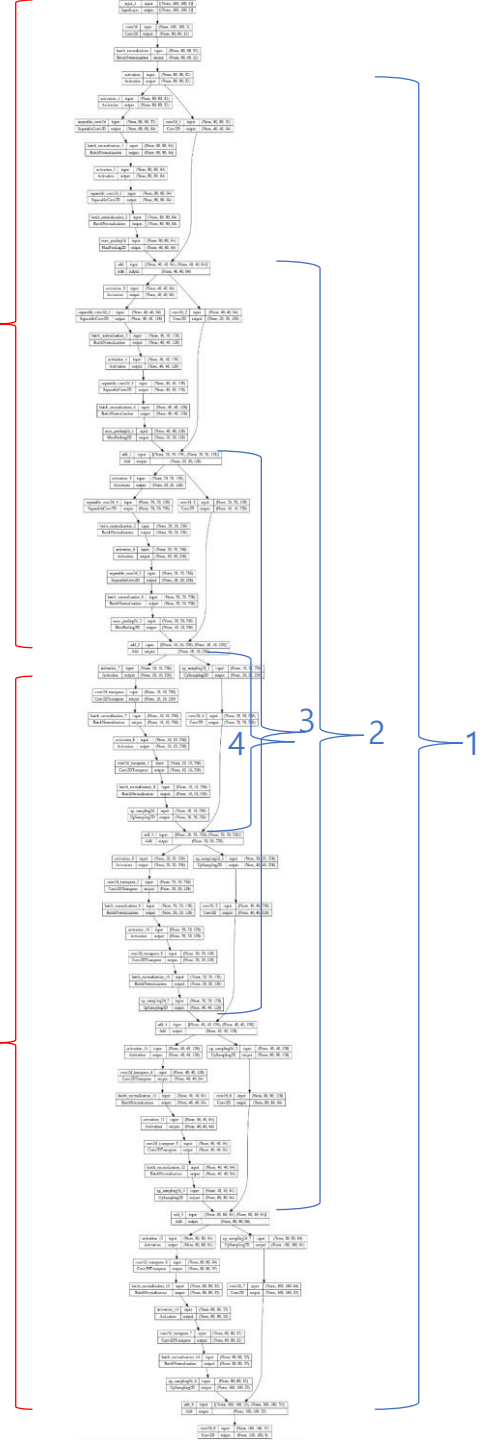
```
Layer (type)                    Output Shape         Param #   Connected to
==================================================================================================
input_1 (InputLayer)            [(None, 160, 160, 3)]  0         []

conv2d (Conv2D)                 (None, 80, 80, 32)   896       ['input_1[0][0]']

batch_normalization (BatchNormalization)  (None, 80, 80, 32)   128       ['conv2d[0][0]']

activation (Activation)         (None, 80, 80, 32)   0         ['batch_normalization[0][0]']

activation_1 (Activation)       (None, 80, 80, 32)   0         ['activation30[0][0]']

separable_conv2d (SeparableConv2D)  (None, 80, 80, 64)   2400      ['activation_1[0][0]']

batch_normalization_1 (BatchNormalization)  (None, 80, 80, 64)   256       ['separable_conv2d[0][0]']

activation_2 (Activation)       (None, 80, 80, 64)   0         ['batch_normalization_1[0][0]']

separable_conv2d_1 (SeparableConv2D)  (None, 80, 80, 64)   4736      ['activation_2[0][0]']

batch_normalization_2 (BatchNormalization)  (None, 80, 80, 64)   256       ['separable_conv2d_1[0][0]']

max_pooling2d (MaxPooling2D)    (None, 40, 40, 64)   0         ['batch_normalization_2[0][0]']

conv2d_1 (Conv2D)               (None, 40, 40, 64)   2112      ['activation[0][0]']

add (Add)                       (None, 40, 40, 64)   0         ['max_pooling2d[0][0]',
                                                                 'conv2d_1[0][0]']

activation_3 (Activation)       (None, 40, 40, 64)   0         ['add[0][0]']

separable_conv2d_2 (SeparableConv2D)  (None, 40, 40, 128)  8896      ['activation_3[0][0]']

batch_normalization_3 (BatchNormalization)  (None, 40, 40, 128)  512       ['separable_conv2d_2[0][0]']

activation_4 (Activation)       (None, 40, 40, 128)  0         ['batch_normalization_3[0][0]']

separable_conv2d_3 (SeparableConv2D)  (None, 40, 40, 128)  17664     ['activation_4[0][0]']

batch_normalization_4 (BatchNormalization)  (None, 40, 40, 128)  512       ['separable_conv2d_3[0][0]']

max_pooling2d_1 (MaxPooling2D)  (None, 20, 20, 128)  0         ['batch_normalization_4[0][0]']

conv2d_2 (Conv2D)               (None, 20, 20, 128)  8320      ['add[0][0]']

add_1 (Add)                     (None, 20, 20, 128)  0         ['max_pooling2d_1[0][0]',
                                                                 'conv2d_2[0][0]']

activation_5 (Activation)       (None, 20, 20, 128)  0         ['add_1[0][0]']

separable_conv2d_4 (SeparableConv2D)  (None, 20, 20, 256)  34176     ['activation_5[0][0]']

batch_normalization_5 (BatchNormalization)  (None, 20, 20, 256)  1024      ['separable_conv2d_4[0][0]']

activation_6 (Activation)       (None, 20, 20, 256)  0         ['batch_normalization_5[0][0]']

separable_conv2d_5 (SeparableConv2D)  (None, 20, 20, 256)  68096     ['activation_6[0][0]']

batch_normalization_6 (BatchNormalization)  (None, 20, 20, 256)  1024      ['separable_conv2d_5[0][0]']

max_pooling2d_2 (MaxPooling2D)  (None, 10, 10, 256)  0         ['batch_normalization_6[0][0]']

conv2d_3 (Conv2D)               (None, 10, 10, 256)  33024     ['add_1[0][0]']

add_2 (Add)                     (None, 10, 10, 256)  0         ['max_pooling2d_2[0][0]',
                                                                 'conv2d_3[0][0]']

activation_7 (Activation)       (None, 10, 10, 256)  0         ['add_2[0][0]']

conv2d_transpose (Conv2DTranspose)  (None, 10, 10, 256)  590080    ['activation_7[0][0]']

batch_normalization_7 (BatchNormalization)  (None, 10, 10, 256)  1024      ['conv2d_transpose[0][0]']

activation_8 (Activation)       (None, 10, 10, 256)  0         ['batch_normalization_7[0][0]']

conv2d_transpose_1 (Conv2DTranspose)  (None, 10, 10, 256)  590080    ['activation_8[0][0]']

batch_normalization_8 (BatchNormalization)  (None, 10, 10, 256)  1024      ['conv2d_transpose_1[0][0]']

up_sampling2d_1 (UpSampling2D)  (None, 20, 20, 256)  0         ['add_2[0][0]']

up_sampling2d (UpSampling2D)    (None, 20, 20, 256)  0         ['batch_normalization_8[0][0]']

conv2d_4 (Conv2D)               (None, 20, 20, 256)  65792     ['up_sampling2d_1[0][0]']

add_3 (Add)                     (None, 20, 20, 256)  0         ['up_sampling2d[0][0]',
                                                                 'conv2d_4[0][0]']

activation_9 (Activation)       (None, 20, 20, 256)  0         ['add_3[0][0]']

conv2d_transpose_2 (Conv2DTranspose)  (None, 20, 20, 128)  295040    ['activation_9[0][0]']

batch_normalization_9 (BatchNormalization)  (None, 20, 20, 128)  512       ['conv2d_transpose_2[0][0]']

activation_10 (Activation)      (None, 20, 20, 128)  0         ['batch_normalization_9[0][0]']

conv2d_transpose_3 (Conv2DTranspose)  (None, 20, 20, 128)  147584    ['activation_10[0][0]']

batch_normalization_10 (BatchNormalization)  (None, 20, 20, 128)  512       ['conv2d_transpose_3[0][0]']

up_sampling2d_3 (UpSampling2D)  (None, 40, 40, 256)  0         ['add_3[0][0]']

up_sampling2d_2 (UpSampling2D)  (None, 40, 40, 128)  0         ['batch_normalization_10[0][0]']

conv2d_5 (Conv2D)               (None, 40, 40, 128)  32896     ['up_sampling2d_3[0][0]']

add_4 (Add)                     (None, 40, 40, 128)  0         ['up_sampling2d_2[0][0]',
                                                                 'conv2d_5[0][0]']

activation_11 (Activation)      (None, 40, 40, 128)  0         ['add_4[0][0]']

conv2d_transpose_4 (Conv2DTranspose)  (None, 40, 40, 64)   73792     ['activation_11[0][0]']

batch_normalization_11 (BatchNormalization)  (None, 40, 40, 64)   256       ['conv2d_transpose_4[0][0]']

activation_12 (Activation)      (None, 40, 40, 64)   0         ['batch_normalization_11[0][0]']

conv2d_transpose_5 (Conv2DTranspose)  (None, 40, 40, 64)   36928     ['activation_12[0][0]']

batch_normalization_12 (BatchNormalization)  (None, 40, 40, 64)   256       ['conv2d_transpose_5[0][0]']

up_sampling2d_5 (UpSampling2D)  (None, 80, 80, 128)  0         ['add_4[0][0]']

up_sampling2d_4 (UpSampling2D)  (None, 80, 80, 64)   0         ['batch_normalization_12[0][0]']

conv2d_6 (Conv2D)               (None, 80, 80, 64)   8256      ['up_sampling2d_5[0][0]']

add_5 (Add)                     (None, 80, 80, 64)   0         ['up_sampling2d_4[0][0]',
                                                                 'conv2d_6[0][0]']

activation_13 (Activation)      (None, 80, 80, 64)   0         ['add_5[0][0]']

conv2d_transpose_6 (Conv2DTranspose)  (None, 80, 80, 32)   18464     ['activation_13[0][0]']

batch_normalization_13 (BatchNormalization)  (None, 80, 80, 32)   128       ['conv2d_transpose_6[0][0]']

activation_14 (Activation)      (None, 80, 80, 32)   0         ['batch_normalization_13[0][0]']

conv2d_transpose_7 (Conv2DTranspose)  (None, 80, 80, 32)   9248      ['activation_14[0][0]']

batch_normalization_14 (BatchNormalization)  (None, 80, 80, 32)   128       ['conv2d_transpose_7[0][0]']

up_sampling2d_7 (UpSampling2D)  (None, 160, 160, 64)  0         ['add_5[0][0]']

up_sampling2d_6 (UpSampling2D)  (None, 160, 160, 32)  0         ['batch_normalization_14[0][0]']

conv2d_7 (Conv2D)               (None, 160, 160, 32)  2080      ['up_sampling2d_7[0][0]']

add_6 (Add)                     (None, 160, 160, 32)  0         ['up_sampling2d_6[0][0]',
                                                                 'conv2d_7[0][0]']

conv2d_8 (Conv2D)               (None, 160, 160, 3)  867       ['add_6[0][0]']

==================================================================================================
Total params: 2058379 (7.85 MB)
Trainable params: 2055203 (7.84 MB)
Non-trainable params: 3176 (14.75 KB)
```

아키텍처 시각화

- tf.keras.utils.plot_model(model, show_shapes=True, dpi=64)

Skip Connection:
스킵연결 1이 스킵연결 2, 3, 4를 내포하고 있는 U-Net 구조
같은 식으로 스킵연결 2는 스킵연결 3, 4를,
스킵연결 3은 스킵연결 4를 내포하고 있다 (중첩 스킵 연결)

- **스킵연결 1**: activation (4번째 레이어) ~ add6 (마지막에서 2번째 레이어)
- **스킵연결 2**: add ~ add_5
- **스킵연결 3**: add_1 ~ add_4
- **스킵연결 4**: add_2 ~ add_3



Pix2Pix에서처럼 U자형 구조가 명확히 그려지지는 않았으나, 업샘플링, 다운샘플링, 스킵 연결을 갖춘 U-Net 구조
중첩 스킵 연결은 다른 레벨 사이의 정보를 잘 통합하여 전반적인 구조와 세밀한 디테일을 동시에 잘 파악할 수 있다

다양한 스케일에서의 정보 학습

- 다양한 스케일에서 다운샘플링이 이루어지기 때문에
- 텍스처, 색상, 밝기 등의 저수준 정보가 다양한 수준에서 학습될 수 있다

- 이렇게 학습된 저수준 정보는 해당 수준의 업샘플링과 연결되어
- 이미지를 본래 사이즈로 복원시킬 때 추가되어 정보의 손실을 최소화한다

- 또한 다운샘플링을 통해 계산량을 줄여서 많은 레이어를 쌓을 수 있도록 한다

get_dataset() 사용

- import random

- val_samples = 1000 # 1000개만 사용
- random.Random(1337).shuffle(input_img_paths) # seed 1337로 입력 데이터 셔플
- random.Random(1337).shuffle(target_img_paths) # 마스크 이미지도 셔플

- train_input_img_paths = input_img_paths[:-val_samples] # 마지막 1000개를 제외한 훈련 데이터
- train_target_img_paths = target_img_paths[:-val_samples] # 같은 인덱스의 마스크 이미지

- val_input_img_paths = input_img_paths[-val_samples:] # 뒤에서부터 1000개의 검증 데이터
- val_target_img_paths = target_img_paths[-val_samples:] # 같은 인덱스의 마스크 이미지

훈련과 검증 Dataset 생성
- train_dataset = get_dataset(batch_size, img_size, train_input_img_paths, train_target_img_paths, max_dataset_len=1000)
- valid_dataset = get_dataset(batch_size, img_size, val_input_img_paths, val_target_img_paths)

모델 컴파일 및 훈련

- model.compile(optimizer=keras.optimizers.Adam(1e-4), loss="sparse_categorical_crossentropy")

- callbacks = [keras.callbacks.ModelCheckpoint("oxford_segmentation.keras", save_best_only=True)]

- epochs = 50
- model.fit(
    train_dataset,
    epochs=epochs,
    validation_data=valid_dataset,
    callbacks=callbacks,
    verbose=2,
)

```
Epoch 40/50
32/32 - 3s - loss: 0.2062 - val_loss: 1.1924 - 3s/epoch - 108ms/step
Epoch 41/50
32/32 - 3s - loss: 0.2098 - val_loss: 1.2151 - 3s/epoch - 108ms/step
Epoch 42/50
32/32 - 3s - loss: 0.2222 - val_loss: 1.0242 - 3s/epoch - 109ms/step
Epoch 43/50
32/32 - 3s - loss: 0.2320 - val_loss: 1.0486 - 3s/epoch - 109ms/step
Epoch 44/50
32/32 - 3s - loss: 0.2260 - val_loss: 1.0301 - 3s/epoch - 109ms/step
Epoch 45/50
32/32 - 3s - loss: 0.2141 - val_loss: 1.0686 - 3s/epoch - 108ms/step
Epoch 46/50
32/32 - 3s - loss: 0.2144 - val_loss: 1.2530 - 3s/epoch - 109ms/step
Epoch 47/50
32/32 - 3s - loss: 0.2093 - val_loss: 1.2007 - 3s/epoch - 108ms/step
Epoch 48/50
32/32 - 3s - loss: 0.2026 - val_loss: 1.1486 - 3s/epoch - 108ms/step
Epoch 49/50
32/32 - 3s - loss: 0.2030 - val_loss: 1.0255 - 3s/epoch - 108ms/step
Epoch 50/50
32/32 - 3s - loss: 0.2035 - val_loss: 1.0790 - 3s/epoch - 108ms/step
<keras.src.callbacks.History at 0x7d7960339270>
```

결과 시각화

테스트 데이터셋 생성

앞에서 만든 valid_dataset과 동일하여, 하나의 데이터를 두 번 사용하고 있다

- val_dataset = get_dataset(batch_size, img_size, val_input_img_paths, val_target_img_paths)
- val_preds = model.predict(val_dataset) # 예측 수행

- print(val_preds.shape)

```
32/32 [==============================] - 1s 14ms/step
(1000, 160, 160, 3)
```

- print(val_preds[0]) # 0 번 데이터의 각 픽셀에서의 예측 결과 출력 →

Label0 (경계) Label1 (배경) Label2 (객체)

```
[5.99469524e-04 9.98258889e-01 1.41567167e-03]
[5.62698347e-04 9.98129427e-01 1.30784628e-03]
[5.49412973e-04 9.98262346e-01 1.18825072e-03]
[5.63316687e-04 9.97898340e-01 1.53841940e-03]
[7.18983123e-04 9.97218370e-01 2.06263713e-03]
[7.75314518e-04 9.96092975e-01 3.13171139e-03]
[7.62805517e-04 9.95283186e-01 3.95401055e-03]
[5.39462827e-03 9.81268227e-01 1.33371865e-02]
[1.38380635e-03 9.93836224e-01 4.77994047e-03]
[1.76181132e-03 9.88962114e-01 9.27608740e-03]
[1.94505206e-03 9.85695660e-01 1.23593435e-02]
[3.78117966e-03 9.76994753e-01 1.92239806e-02]
[4.69718548e-03 9.72203434e-01 2.30994504e-02]
[6.03599846e-03 9.65397477e-01 2.85665356e-02]
[5.97888790e-03 9.67058420e-01 2.69626901e-02]
[9.97321960e-03 9.54266906e-01 3.57598849e-02]
[8.46546888e-03 9.58432674e-01 3.31018753e-02]
[1.02566862e-02 9.58145082e-01 3.15982588e-02]
[1.00863250e-02 9.60580587e-01 2.93331295e-02]
[1.20762037e-02 9.64902103e-01 2.30217427e-02]
[9.03843995e-03 9.67589796e-01 2.33716462e-02]
[1.24409590e-02 9.59345162e-01 2.82138437e-02]
[1.48097230e-02 9.47441936e-01 3.77482772e-02]
[6.28565699e-02 9.05447185e-01 3.16962264e-02]]]
```

결과 시각화

결과 시각화 함수 정의

- def display_mask(i):

```
mask = np.argmax(val_preds[i], axis=-1)          # 예측 결과 분류 차원에서 가장 높은 값을 갖는 인덱스 선택
mask = np.expand_dims(mask, axis=-1)             # (height, width) → (height, width, 1)의 3D로 변환
img = ImageOps.autocontrast(keras.utils.array_to_img(mask))   # 마스크 배열을 이미지로 변환 후
                                                              # 명암 대비가 잘 드러나게 자동 조정
display(img)                                     # 화면에 출력
```

결과 시각화

- i = 10 # 10번째 이미지를 출력해본다
- display(Image(filename=val_input_img_paths[i]))

정답 마스크 이미지 출력

- img = ImageOps.autocontrast(load_img(val_target_img_paths[i]))
- display(img)

모델이 예측한 마스크 이미지 출력

- display_mask(i)

※ 학습이 충분히 이루어지지는 않았다

결과 이미지 (160x160)

픽셀 포지션

- 경계(0), 배경(1), 객체(2)에 대하여 모델이 예측한 픽셀 위치를 확인한다

- predicted_classes = np.argmax(val_preds[10], axis=-1)   # 각 픽셀에서 예측된 클래스
- mask_positions = np.where(predicted_classes == 0)   # 경계(0) 픽셀 위치

- print("경계 픽셀의 (행, 열) 포지션:", mask_positions)

```
경계 픽셀의 (행, 열) 포지션: (array([ 16,  17,  17,  17,  17,  18,  18,  18,
18,  19,  19,  19,  19,  19,  19,  19,  19,  20,  20,  20,  20,  20,  21,  21,
21,  21,  21,  21,  21,  21,  21,  21,  21,  21,  21,  21,  21,  21,  21,  21,
21,  21,  21,  21,  22,  22,  22,  22,  22,  22,  22,  22,  22,  22,  22,  22,
22,  22,  22,  22,  22,  22,  22,  22,  ...
                    ..................................................
 ... 141, 143]), array([111, 111, 112, 120, 122, 111, 120, 121, 122, 111, 112,
120, 121, 122, 123, 124, 125, 111, 112, 120, 121, 122,  89,  90,  91,  95,
104, 105, 106, 107, 108, 109, 110, 111, 112, 115, 119, 120, 121, 122, 123,
124, 125, 126,  89,  90,  91,  95, 104, 105, 106, 107, 108, 109, 110, 111,
112, 115, 116, 117, 119, 120, 121, 122, 123, 124, 125, 126,  79,  8
```

0으로 예측한 곳의 행의 인덱스

0으로 예측한 곳의 열 인덱스