과제

전진희

실습01. 복잡한 Augmentation 파이프라인 정리

```
import albumentations as A
import random

# 데이터 변환 파이프라인 정의
transform = A.Compose([
    A.HorizontalFlip(p=0.5),
    A.ShiftScaleRotate(p=0.5),
    A.RandomBrightnessContrast(p=0.3),
    A.RGBShift(r_shift_limit=30, g_shift_limit=30, b_shift_limit=30, p=0.3)
], bbox_params=A.BboxParams(format='coco', label_fields=['category_ids']))

random.seed(7)

# 변환된 데이터를 저장할 변수 선언
transformed = transform(image=image, bboxes=bboxes, category_ids=category_ids)

# 시각화 함수를 사용하여 변환된 데이터를 확인
visualize(
    transformed['image'],   # 변환된 이미지
    transformed['bboxes'],   # 변환된 bounding box
    transformed['category_ids'],   # 변환된 카테고리 ID
    category_id_to_name   # 카테고리 ID에 해당하는 이름을 저장한 딕셔너리
)
```

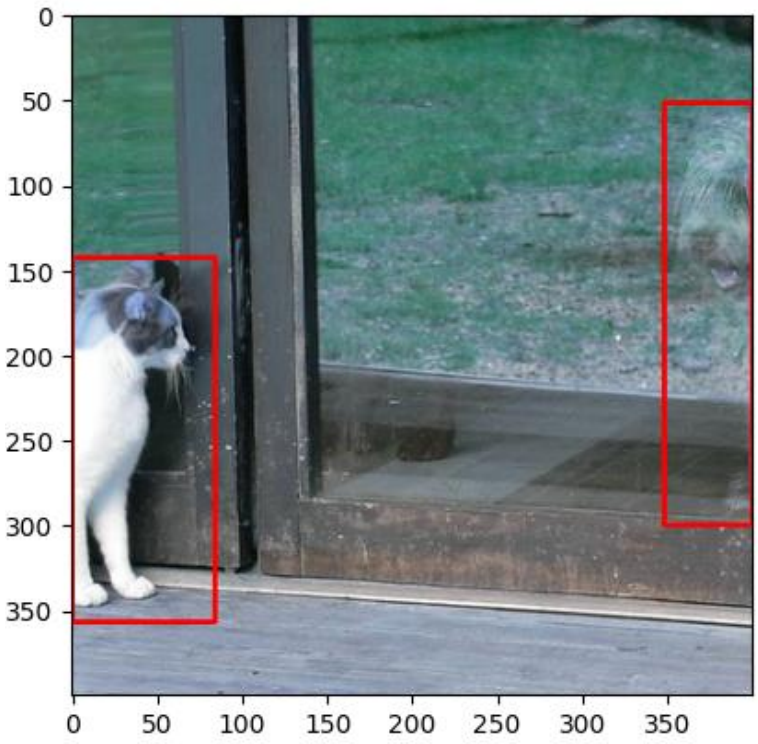
실습02. 공간적 보강

```
# 공간적 보강

transform_temp = A.Compose(
    [A.CenterCrop(height=400, width=400, p=1)],
    bbox_params=A.BboxParams(format='coco', min_area=4500, label_fields=['category_ids'])
# min_area는 픽셀값으로, 이 미니멈보다 픽셀이 적게 잡히면 라벨링한 객체가 잡히지 않습니다.
)

# 변환된 데이터를 저장할 변수 선언
transformed_01 = transform_temp(image=image, bboxes=bboxes, category_ids=category_ids)

# 시각화 함수를 사용하여 변환된 데이터를 확인
visualize(
    transformed_01['image'],   # 변환된 이미지
    transformed_01['bboxes'],   # 변환된 bounding box
    transformed_01['category_ids'],   # 변환된 카테고리 ID
    category_id_to_name   # 카테고리 ID에 해당하는 이름을 저장한 딕셔너리
```



실습03. 정형 데이터 셋 나누기

```
import pandas as pd
from sklearn.model_selection import train_test_split

# 데이터 다운로드
url = 'https://raw.githubusercontent.com/mGalarnyk/Tutorial_Data/master/King_County/kingCountyHouseData.csv'
df = pd.read_csv(url)

print(df)
```

```
features_data = ['bedrooms', 'bathrooms', 'sqft_living', 'sqft_lot', 'floors'] # 학습 데이터
target_data = ['price' ] # 정답지

x_data = df.loc[:,features_data] # 학습 데이터
y_data = df.loc[:,target_data] # 정답지

print(x_data)
print(y_data)
```

```
       bedrooms  bathrooms  sqft_living  sqft_lot  floors
0             3       1.00         1180      5650     1.0
1             3       2.25         2570      7242     2.0
2             2       1.00          770     10000     1.0
3             4       3.00         1960      5000     1.0
4             3       2.00         1680      8080     1.0
...         ...        ...          ...       ...     ...
21608         3       2.50         1530      1131     3.0
21609         4       2.50         2310      5813     2.0
21610         2       0.75         1020      1350     2.0
21611         3       2.50         1600      2388     2.0
21612         2       0.75         1020      1076     2.0

[21613 rows x 5 columns]
          price
0      221900.0
1      538000.0
2      180000.0
3      604000.0
4      510000.0
...         ...
21608  360000.0
21609  400000.0
21610  402101.0
21611  400000.0
21612  325000.0

[21613 rows x 1 columns]
```

데이터 나누기 train val test

```
x_train, x_val_list, y_train, y_val_list = train_test_split(x_data, y_data, random_state=777, train_size=0.8) # 훈련 세트와 검증+테스트 세트로 분할
x_val, x_test, y_val, y_test = train_test_split(x_val_list, y_val_list, random_state=777, test_size=0.5)
 # 검증+테스트 세트를 검증 세트와 테스트 세트로 분할

print("------------변경되기 전 데이터 양-------------")
print("x_data 크기 : ", x_data.shape)
print("y_data 크기 : ", y_data.shape)
print("------------변경 후 데이터 양-------------")
print("x_train >> : ", x_train.shape)
print("y_train >> : ", y_train.shape)
print("x_val >> : ", x_val.shape)
print("y_val >> : ", y_val.shape)
print("x_test >> : ", x_test.shape)
print("y_test >> : ", y_test.shape)
```

```
------------변경되기 전 데이터 양-------------
x_data 크기 :  (21613, 5)
y_data 크기 :  (21613, 1)
------------변경 후 데이터 양-------------
x_train >> :  (17290, 5)
y_train >> :  (17290, 1)
x_val >> :  (2161, 5)
y_val >> :  (2161, 1)
x_test >> :  (2162, 5)
y_test >> :  (2162, 1)
```

실습04. 텐서 기본 실습

```
import torch
import numpy as np
```

텐서 초기화하기 데이터로부터 직접 텐서를 생성할 수 있다

```
# 1. torch 이용해서 만든 텐서
data = [[1,2],[3,4]]
x_data = torch.tensor(data)       # 리스트 data를 텐서로 변환

print(x_data)

# 2. numpy를 이용해서 만든 텐서
np_array = np.array(data)
x_np = torch.from_numpy(np_array)
print(x_np)
```

```
tensor([[1, 2],
        [3, 4]])
tensor([[1, 2],
        [3, 4]], dtype=torch.int32)
```

- torch.tensor()는 입력 텐서를 복사하여 새로운 텐서를 만듭니다. 이 함수는 항상 새로운 메모리를 할당하므로, 원본 데이터와의 메모리 공유가 이루어지지 않습니다.
- torch.from_numpy() 함수는 NumPy 배열을 PyTorch 텐서로 변환할 때, 원본 데이터와의 메모리 공유를 유지합니다.

```
x_ones = torch.ones_like(x_data)
print(f"ones Tensor:₩n{x_ones}")
# 주어진 입력 텐서와 동일한 크기의 텐서를 생성하고 모든 요소를 1로 채우면됩니다



x_rand = torch.rand_like(x_data, dtype=torch.float)  # x_data 속성을 덮어쓴다
print(f"Random Tensor:₩n{x_rand}")
# torch.rand_like() 주어진 입력 텐서와 동일한 크기의 텐서를 생성하고 모든 요소를 랜덤한 값으로 채운다. 그리고 타입 지정하면 그 타입으로 변경된다
# 0과 1사이의 랜덤한 값으로 초기화 되고 데이터 타입 유형은 dtype=torch.float 지정된다
```

```
ones Tensor:
tensor([[1, 1],
        [1, 1]])
Random Tensor:
tensor([[0.8425, 0.7634],
        [0.8828, 0.5931]])
```

```
# 무작위 또는 상수 값을 사용하기
shape = (2,3,) #  마지막 콤마(,) 다음에 비워둔 이유는 파이썬에서 튜플을 정의할 때 원소가 하나인 경우에도 쉼표(,)를 사용하기 때문 / (행,렬)
rand_tensor = torch.rand(shape) * 10  # 10을 곱하여 범위를 10까지 늘렸다 (0 ~ 10)
ones_tensor = torch.ones(shape)
zeros_tensor = torch.zeros(shape)

print("rand_tesnsor  ₩n", rand_tensor)
print("ones_tesnsor  ₩n", ones_tensor)
print("zeros_tesnsor  ₩n", zeros_tensor)



# 유효 범위를 최소값 얼마 부터 ~ 최대값 얼마까지 6 ~10
shape_temp = (5,6)
min_val = 6
max_val = 10

rand_tensor_temp = torch.rand(shape_temp) * (max_val - min_val) + min_val
print(rand_tensor_temp)
```

```
rand_tesnsor
 tensor([[0.5216, 1.3128, 6.3806],
        [2.7914, 4.8000, 5.5472]])
ones_tesnsor
 tensor([[1., 1., 1.],
        [1., 1., 1.]])
zeros_tesnsor
 tensor([[0., 0., 0.],
        [0., 0., 0.]])
tensor([[8.8684, 7.6697, 8.8303, 8.5580, 8.4632, 6.2413],
        [8.3459, 8.5465, 6.1430, 7.6811, 9.2791, 7.5726],
        [9.1309, 8.2938, 6.5598, 6.1945, 7.9910, 6.2117],
        [9.9174, 8.6974, 9.6860, 7.1775, 9.0970, 9.6585],
        [9.6091, 7.9616, 9.3550, 7.7338, 7.0788, 7.7244]])
```

텐서 속성

```
# 텐서의 어트리뷰트 보기

tensor_val = torch.rand(3,4)

# 디바이스 정보 가져오기
device = 'cuda' if torch.cuda.is_available() else 'cpu'
print(device)
```

```
tensor_val.to(device)

# 디바이스 변경하고자 하는경우
# 텐서의 디바이스를 변경하려면 to() 메서드를 사용할 수 있습니다. 이 메서드는 새로운 디바이스로 텐서를 이동시킵니다.
# EX) model.to(device)

print(f"Shape of tensor : {tensor_val.shape}")
print(f"Data Type of tensor : {tensor_val.dtype}")
print(f"Device tendor is stored on : {tensor_val.device}")
```

```
cpu
Shape of tensor : torch.Size([3, 4])
Data Type of tensor : torch.float32
Device tendor is stored on : cpu
```

```
# 표준 인덱싱과 슬라이싱
tensor_1 = torch.ones(4,4)
tensor_1[:,3] = 0
print(tensor_1)

tensor_2 = torch.ones(4,4)
tensor_2[:,2] =2
print(tensor_2)
```

```
tensor([[1., 1., 1., 0.],
        [1., 1., 1., 0.],
        [1., 1., 1., 0.],
        [1., 1., 1., 0.]])
tensor([[1., 1., 2., 1.],
        [1., 1., 2., 1.],
        [1., 1., 2., 1.],
        [1., 1., 2., 1.]])
```

```
# 텐서 합치기

t1 = torch.cat([tensor_1, tensor_1, tensor_1], dim=1)
print(t1)
```

```
tensor([[1., 1., 1., 0., 1., 1., 1., 0., 1., 1., 1., 0.],
        [1., 1., 1., 0., 1., 1., 1., 0., 1., 1., 1., 0.],
        [1., 1., 1., 0., 1., 1., 1., 0., 1., 1., 1., 0.],
        [1., 1., 1., 0., 1., 1., 1., 0., 1., 1., 1., 0.]])
```

```
# 텐서 곱하기

t_mult = tensor_1.mul(tensor_2) # 곱하기
print(t_mult)

print(tensor_1 * tensor_2) # 곱하기2
```

```
tensor([[1., 1., 2., 0.],
        [1., 1., 2., 0.],
        [1., 1., 2., 0.],
        [1., 1., 2., 0.]])
tensor([[1., 1., 2., 0.],
        [1., 1., 2., 0.],
        [1., 1., 2., 0.],
        [1., 1., 2., 0.]])
```

```
# 행렬 곱
#  행렬 곱셈은 두 개의 행렬을 곱하여 새로운 행렬을 생성하는 연산

print(tensor_2.matmul(tensor_2.T))
print(tensor_2 @ tensor_2.T)
```

```
tensor([[7., 7., 7., 7.],
        [7., 7., 7., 7.],
        [7., 7., 7., 7.],
        [7., 7., 7., 7.]])
tensor([[7., 7., 7., 7.],
        [7., 7., 7., 7.],
        [7., 7., 7., 7.],
        [7., 7., 7., 7.]])
```

```
# Tensor -> NumPy 배열로 변환

t = torch.ones(5)
print(t)
n = t.numpy()
print(n)

t.add_(1)
print(t)
print(n)

# 메모리 공간을 공유해서 텐서가 바뀌면 해당 넘파이도 바뀜
```

```
tensor([1., 1., 1., 1., 1.])
[1. 1. 1. 1. 1.]
tensor([2., 2., 2., 2., 2.])
[2. 2. 2. 2. 2.]
```

실습05. View

```
"""
파이토치 텐서의 뷰는 넘파이의 Reshape 와 같은 역할
Reshape > 텐서의 크기를 변경해주는 역할
"""

# 3차원 데이터 생성
t_temp = np.array([[[0, 1, 2], [3, 4, 5], [6, 7, 8], [9, 10, 11]]])
ft = torch.FloatTensor(t_temp)
print(ft)
print(ft.shape)

# ft view -> 2차원 텐서로 변경
# -1 : 나는 그 값을 모르니 파이토치 니가 알아서 해!! 두번째 차원의 길이는 3 가지도록 해라.
print(ft.view([-1, 3])) # (?, 3)
print(ft.view([-1, 3]).shape)

"""
 view() 메서드를 사용하여 텐서의 차원을 변경하면,
 ->  데이터를 복사하여 새로운 텐서를 생성하고
 이 새로운 텐서는 원래 텐서와 메모리를 공유안함 !!

"""
```

```
tensor([[[ 0.,  1.,  2.],
         [ 3.,  4.,  5.],
         [ 6.,  7.,  8.],
         [ 9., 10., 11.]]])
torch.Size([1, 4, 3])
tensor([[ 0.,  1.,  2.],
        [ 3.,  4.,  5.],
        [ 6.,  7.,  8.],
        [ 9., 10., 11.]])
torch.Size([4, 3])
```

```
print(ft.view([-1, 1, 3]))
print(ft.view([-1, 1, 3]).shape)
```

```
tensor([0., 1., 2.])
torch.Size([3])
```

```
"""
언스퀴즈 - 특정 위치에서 1인 차원을 추가합니다.
"""


ft_temp = torch.Tensor([0,1,2])
print(ft_temp.shape)
```

```
torch.Size([3])
```

```
# 첫번째 차원에서 1차원 추가
# 인덱스 0
print(ft_temp.unsqueeze(0))
print(ft_temp.unsqueeze(0).shape)
```

```
tensor([[0., 1., 2.]])
torch.Size([1, 3])
```

```
# view로 1차원 추가
print(ft_temp.view(1, -1))
print(ft_temp.view(1, -1).shape)
```

```
tensor([[0., 1., 2.]])
torch.Size([1, 3])
```