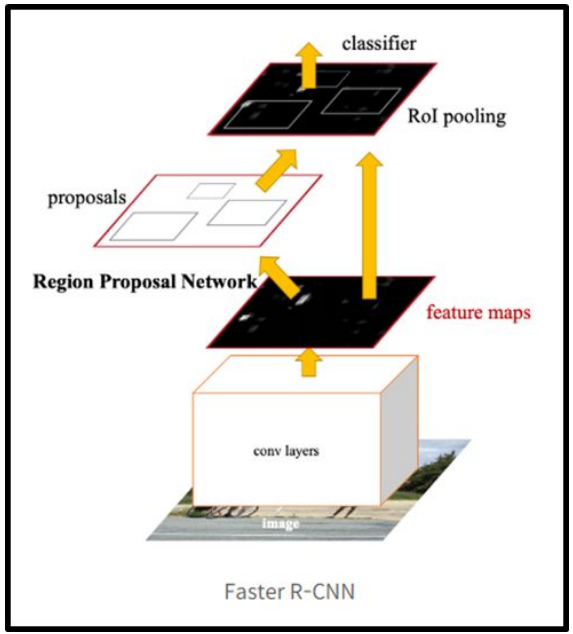
Faster R-CNN 실습

목차

Faster R-CNN



•네트워크 병목 현상 문제를 해결하기 위해 후보 영역 추출 작업을 수행하는 네트워크인 RPN(Region Proposal Network)을 도입합니다.

•RPN은 region proposals를 보다 정교하게 추출하기 위해 다양한 크기와 가로세로비를 가지는 bounding box인 Anchor Box를 도입합니다.

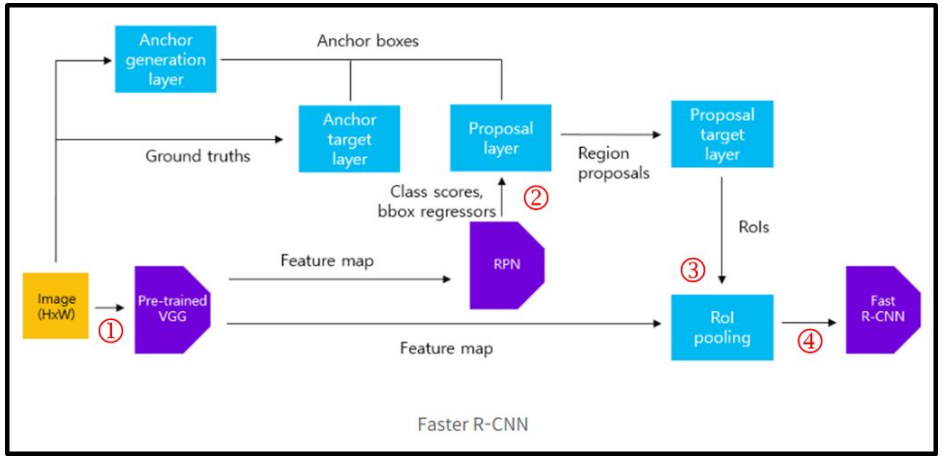
•Faster R-CNN 모델은 RPN과 Faster R-CNN 모델이 합쳐졌다고 볼 수 있습니다.

•RPN에서 region proposals를 추출하고 이를 Fast R-CNN 네트워크에 전달하여 객체의 Class와 위치를 예측합니다.

•이를 통해 모델의 전체적인 과정이 GPU 상에서 동작하여 병목 현상이 발생하지 않으며 end-to-end로 네트워크를 학습 시키는 것이 가능해집니다.

※ region proposal: 이미지 안에서 객체가 있을 만한 후보 영역을 먼저 찾아주는 방법

전체적인 동작 과정



1 원본 이미지를 pre-trained된 CNN 모델에 입력하여 feature map을 얻습니다.

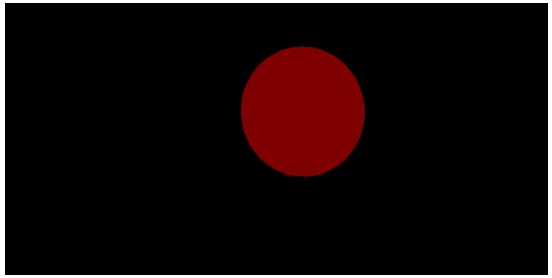
2 feature map은 RPN에 전달되어 적절한 region proposals을 산출합니다.

3 region proposals와 1) 과정에서 얻은 feature map을 통해 RoI pooling을 수행하여 고정된 크기의 feature map을 얻습니다.

4 Fast R-CNN 모델에 고정된 크기의 feature map을 입력하여 Classification과 Bounding box regression을 수행합니다.

Data Preprocessing

AC\train\image



AC\train\label

image 폴더에 있는 초음파 이미지에 빨간색으로 영역을 표시한 게 label 폴더에 있는 라벨 정보입니다.

먼저 이 label 폴더에 있는 라벨 정보를 이용하여 Bounding Box의 좌표를 알아내 csv 형식으로 저장해주었습니다.

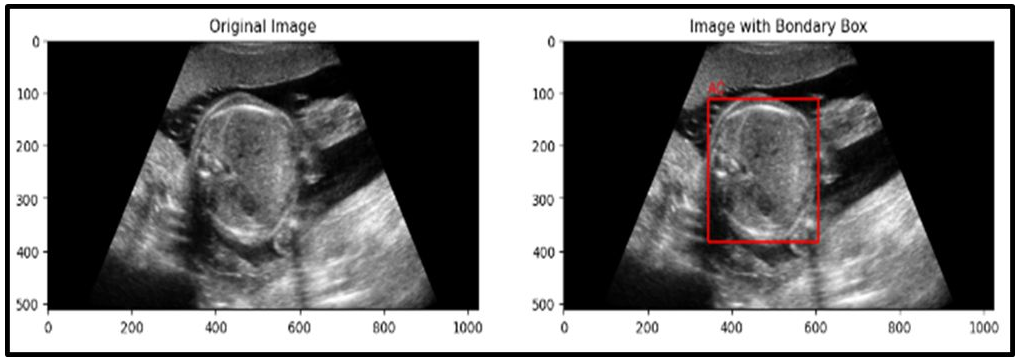
train.csv - Excel

name	x1	x2	y1	y2	classname
20151103_E0000056_I0004613.png	344	606	112	384	AC
20151103_E0000057_I0004695.png	454	699	91	336	AC
20151103_E0000057_I0004696.png	445	678	82	327	AC
20151103_E0000058_I0004790.png	403	579	236	419	AC
20151103_E0000059_I0004888.png	402	600	162	360	AC
20151103_E0000060_I0004997.png	364	535	174	340	AC
20151104_E0000049_I0004080.png	373	644	116	398	AC
20151104_E0000050_I0004170.png	340	576	106	328	AC
20151104_E0000052_I0004261.png	330	521	148	341	AC
20151104_E0000053_I0004361.png	446	659	129	354	AC
20151104_E0000054_I0004446.png	382	579	232	419	AC
20151104_E0000055_I0004531.png	391	633	157	402	AC
20151105_E0000003_I0003866.png	451	640	157	348	AC
20151105_E0000003_I0003893.png	360	612	207	433	AC
20151105_E0000037_I0003284.png	374	599	137	344	AC
20151105_E0000038_I0003387.png	350	653	124	378	AC
20151105_E0000039_I0003490.png	377	675	206	454	AC
20151105_E0000040_I0003585.png	438	725	137	426	AC
20151105_E0000042_I0003798.png	352	705	118	395	AC

Visualize image and bounding boxes

```
# RGB # 빨간색 : (128,0,0) /녹색 : (0,128,0)
```

```
folder_path = "C:/Users/JinaChoi/Desktop/Data/label/AC"

# 반복문
for file_name in os.listdir(folder_path):
    if file_name.endswith(".png"):
        file_path = os.path.join(folder_path, file_name)
        image = cv2.imread(file_path, cv2.IMREAD_UNCHANGED) # cv2.IMREAD_UNCHANGED -> PNG 파일

        gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

        # 빨간색과 연두색 영역을 기준으로 최소 바운딩 박스 생성
        red_lower = (0,0,100) # 빨간색 범위의 하한값(B, G, R)
        red_upper = (50, 50, 255)  # 빨간색 범위의 상한값 (B, G, R)
        green_lower = (0, 100, 0)  # 연두색 범위의 하한값 (B, G, R)
        green_upper = (50, 255, 50)  # 연두색 범위의 상한값 (B, G, R)

        # 빨간색 영역에 대한 마스크 생성
        red_mask = cv2.inRange(image, red_lower, red_upper)

        # 연두색 영역에 대한 마스크 생성
        green_mask = cv2.inRange(image, green_lower, green_upper)

        # 빨간색과 연두색 마스크 합침
        combined_mask = cv2.bitwise_or(red_mask, green_mask)

        # 마스크를 이용하여 객체 영역에 대한 contour 찾기
        contours, _ = cv2.findContours(combined_mask, cv2.RETR_EXTERNAL,
                                       cv2.CHAIN_APPROX_SIMPLE)


        # contour를 기반으로 최소 바운딩 박스 생성
        bounding_boxes = []
        for contour in contours:
            x, y, w, h = cv2.boundingRect(contour)
            bounding_boxes.append((x,y,w,h))

        # 원본 이미지에 바운딩 박스 그리기
        for (x, y, w, h) in bounding_boxes:
            cv2.rectangle(image, (x,y), (x+w, y+h), (0,255,0),2)

        print(file_name, x, y, x+w, y+h)
```



Original Image에 Bounding Box가 객체를 잘 감싸는 것을 확인할 수 있습니다.

Library

```
import os
import cv2
import pandas as pd
import numpy as np
import torch
import torchvision
from torch.utils.data import Dataset, DataLoader
from torchvision import transforms, datasets

from torchvision.models.detection import fasterrcnn_resnet50_fpn
from torchvision.transforms import functional as F
from torchvision.transforms import ToTensor
from tqdm import tqdm

from PIL import Image
import matplotlib.pyplot as plt
import matplotlib.patches as patches

import torch
import torch.nn as nn

from torchvision.transforms import ToTensor
import torch.nn.functional as F
```

학습에 사용될 targets은 list 형태,
targets 안에 있는 target은 dictionary 형태입니다.

boxes: bounding box 좌표
labels: 해당 객체의 클래스 라벨(AC: 1, FL: 2, HC: 3, HUM: 4)

```
targets
```

```
[{'boxes': tensor([[43, 28, 76, 96]]), 'labels': tensor([1])},
 {'boxes': tensor([[57, 23, 87, 84]]), 'labels': tensor([1])},
 {'boxes': tensor([[56, 20, 85, 82]]), 'labels': tensor([1])},
 {'boxes': tensor([[ 50,  59,  72, 105]]), 'labels': tensor([1])},
 {'boxes': tensor([[50, 40, 75, 90]]), 'labels': tensor([1])},
 {'boxes': tensor([[46, 44, 67, 85]]), 'labels': tensor([1])},
                                .
                                .
                                .
 {'boxes': tensor([[49, 42, 70, 58]]), 'labels': tensor([4])},
 {'boxes': tensor([[54, 42, 75, 57]]), 'labels': tensor([4])},
 {'boxes': tensor([[58, 34, 79, 41]]), 'labels': tensor([4])},
 {'boxes': tensor([[58, 40, 72, 54]]), 'labels': tensor([4])}]
```

1) Feature extraction by pre-trained VGG16

```
DEVICE = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print("Using device:", DEVICE)
```

```
imgTensor = torch.stack(images).to(DEVICE)
```

```
model = torchvision.models.vgg16(pretrained=True).to(DEVICE)
features = list(model.features)

# only collect layers with output feature map size (W, H) < 50
dummy_img = torch.zeros((1, 3, 128, 128)).float() # test image array

req_features = []
output = dummy_img.clone().to(DEVICE)

for feature in features:
    output = feature(output)
#     print(output.size()) => torch.Size([batch_size, channel, width, height])
    if output.size()[2] < 128//16: # 128/16=8
        break
    req_features.append(feature)
    out_channels = output.size()[1]

faster_rcnn_feature_extractor = nn.Sequential(*req_features)

output_map = faster_rcnn_feature_extractor(imgTensor)
```

```
print(dummy_img.shape)
print(out_channels)
# print(len(req_features))
```

```
torch.Size([1, 3, 128, 128])
512
```

이미지 크기: (1, 1050, 512) -> (3, 128, 128)
(VGG16 모델에 학습시키기 위해 3 channel로 변경, 이미지 크기 Resize)

이미지를 pre-trained된 VGG 모델에 입력하여 feature map을 얻습니다.

이미지의 크기가 128x128이며, sub-sampling ratio가 1/16이라고 했을 때, 8x8 크기의 feature map이 생성됩니다. (128 / 16 = 8)

out_channels: 512 (VGG16 모델의 마지막 feature map 채널 수)

※ sub-sampling ratio란 입력 이미지와 feature map의 비율을 말합니다. (입력 이미지에서 얼마나 축소되었는지를 나타내는 값)

1) 생성된 feature map 시각화

```
imgArray = output_map.data.cpu().numpy()
fig = plt.figure(figsize=(12, 4))
figNo = 1

for i in range(5):
    fig.add_subplot(1, 5, figNo)
    plt.imshow(imgArray[0, i], cmap='gray')
    figNo += 1

plt.show()
```

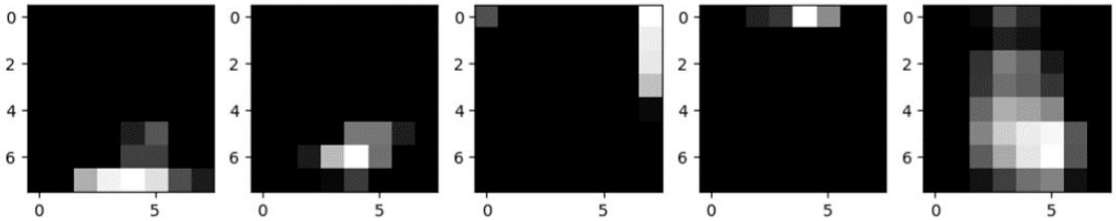


그림1. 8x8x512 feature map의 1~5까지의 채널 시각화

2)-1 Generate Anchors

```
feature_size = 128 // 16
ctr_x = np.arange(16, (feature_size + 1) * 16, 16)
ctr_y = np.arange(16, (feature_size + 1) * 16, 16)
print(len(ctr_x))
print(ctr_x)
```

```
index = 0
ctr = np.zeros((64, 2))

for i in range(len(ctr_x)):
    for j in range(len(ctr_y)):
        ctr[index, 1] = ctr_x[i] - 8
        ctr[index, 0] = ctr_y[j] - 8
        index += 1

# ctr => [[center x, center y], ...]
print(ctr.shape)
print(ctr[:10, :])
```

이제 anchor box를 생성합니다. 이미지의 크기가 128x128이며, sub-sampling ratio=1/16이므로, 총 64(=8x8)개의 anchor box를 생성해야 합니다.

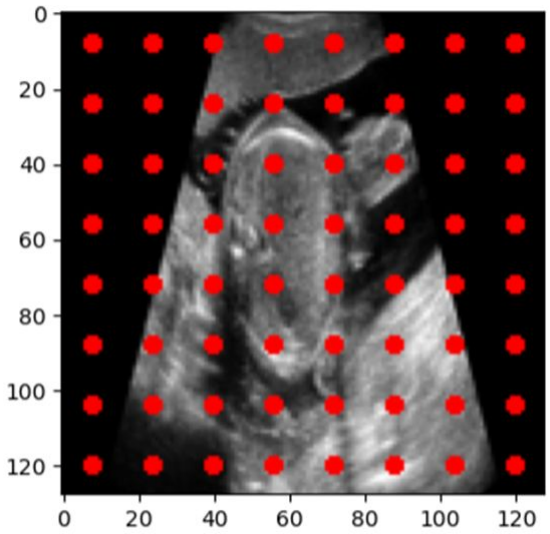


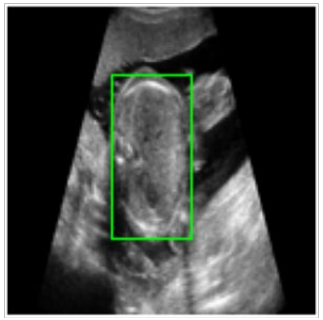
그림2. 이미지 내의 64개(8x8) anchor 표시

2)-2 Generate Anchors

객체를 잘 감쌀 수 있는 Anchor Box를 만들기 위해서는 ratios와 scales를 알맞게 설정해야 합니다.
따라서 AC, FL, HC, HUM의 Bounding Box의 높이와 너비(H, W) 비율의 평균값을 알아내고,
X축 시작점, Y축 시작점, X축 종점, Y축 종점의 평균 및 최소 최대값을 알아냅니다.

AC와 HC의 객체 모양이 비슷하고 FL과 HUM의 객체 모양이 비슷하여 두 그룹으로 묶어주었습니다.
이미지 사이즈가 128*128로 줄어들어 AC와 HC의 높이/너비 비율은 1.8:1, FL과 HUM의 높이/너비 비율이 0.5:1입니다.

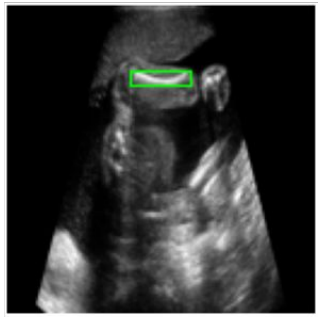
AC



1.8

1

HUM



0.5

1

평균	H/W(세로/가로)	비율
AC	1.899	1.9:1
FL	0.631	0.6:1
HC	1.6895	1.7:1
HUM	0.5335	0.5:1

평균	x축 시작점	y축 시작점	x축 종점	y축 종점
AC	48	35	80	95
FL	54	40	75	53
HC	46	32	82	94
HUM	55	40	76	50

최소,최대	x축 최소 시작점	y축 최소 시작점	x축 최대 종점	y축 최대 종점
AC	36	11	97	120
HC	35	15	95	120
FL	43	16	90	112
HUM	42	9	88	88

2)-2 Generate Anchor boxes

```
ratios = [1.8, 0.5] # [1.7, 1.9, 0.5]
scales = [1, 2, 3] # [1, 2, 4]

sub_sample = 128 // 16

num_anchors = len(ratios) * len(scales)
print(num_anchors)
```

```
6
```

```
# AC와 HC 객체용 앵커 박스 설정
#ac_hc_scale = 2.0  # 2.0
ac_hc_ratio = 1.8
ac_hc_ctr_x =  ac_hc_x_start_mean #35
ac_hc_ctr_y = ac_hc_y_start_mean #11

# FL와 HUM 객체용 앵커 박스 설정
#fl_hum_scale = 2.0  # 2.0
fl_hum_ratio = 0.5
fl_hum_ctr_x = fl_hum_x_start_mean #42
fl_hum_ctr_y = fl_hum_y_start_mean #9

# Calculate the number of anchor boxes
num_anchors = len(ratios) * len(scales)

# Initialize anchor boxes array
anchor_boxes = np.zeros(((feature_size * feature_size * num_anchors), 4))
index = 0

for i in range(feature_size):
    for j in range(feature_size):
        ctr_x = sub_sample * j + sub_sample // 2
        ctr_y = sub_sample * i + sub_sample // 2

        # Calculate the actual center coordinates based on object means
        ctr_x = ctr_x + ac_hc_ctr_x if i < feature_size // 2 else ctr_x + fl_hum_ctr_x
        ctr_y = ctr_y + ac_hc_ctr_y if j < feature_size // 2 else ctr_y + fl_hum_ctr_y


        for ratio in ratios:
            for scale in scales:
                # Anchor box height, width
                h = sub_sample * scale * np.sqrt(ratio)
                w = sub_sample * scale * np.sqrt(1. / ratio)

                # Anchor box [x1, y1, x2, y2]
                anchor_boxes[index, 0] = ctr_x - w / 2. # ctr_x - w / 2.
                anchor_boxes[index, 1] =  ctr_y - h / 2. # ctr_y - h / 2.
                anchor_boxes[index, 2] = ctr_x + w * 2. #anchor_boxes[index, 2] = ctr_x + w / 2.
                anchor_boxes[index, 3] = ctr_y + h * 2. #anchor_boxes[index, 3] = ctr_y + h / 2.

                # Ensure anchor boxes are within the image boundaries
                anchor_boxes[index, 0] = max(anchor_boxes[index, 0], 0)
                anchor_boxes[index, 1] = max(anchor_boxes[index, 1], 0)
                anchor_boxes[index, 2] = min(anchor_boxes[index, 2], imgTensor.size(3) - 1)
                anchor_boxes[index, 3] = min(anchor_boxes[index, 3], imgTensor.size(2) - 1)
                index += 1
```

ratios를 통해 1.8:1, 0.5:1, 가로 세로 비율을 가진 앵커 박스 생성.
scales는 각각 기본 앵커 박스의 가로 세로 크기에 1, 2, 3를 곱한 크기의 앵커 박스 생성.
sub_sample: 이미지 특성맵의 크기는 원본 이미지의 1/16로 조정.

8x8 간격의 grid마다 서로 다른 ratios, scales를 가지는 6개의 anchor 박스 생성해줍니다.

반복문을 통해 anchor_boxes 변수에 전체 anchor box의 좌표(x1, y1, x2, y2)를 저장합니다

anchor_boxes 변수의 크기는 (384, 4)입니다.
※ 384: 64(anchor 개수) x 6(box 개수)

2)-2 Generate Anchor boxes

```
print(anchor_boxes.shape)
print(anchor_boxes[:14, :])
```

```
(384, 4)
[[ 48.01857603  31.63343685  62.92569588  58.46625258]
 [ 45.03715206  26.26687371  74.85139176  79.93250517]
 [ 42.05572809  20.90031056  86.77708764 101.39875775]
 [ 45.34314575  34.17157288  73.627417    48.3137085 ]
 [ 39.6862915   31.34314575  96.254834    59.627417  ]
 [ 34.02943725  28.51471863 118.88225099  70.9411255 ]
 [ 56.01857603  31.63343685  70.92569588  58.46625258]
 [ 53.03715206  26.26687371  82.85139176  79.93250517]
 [ 50.05572809  20.90031056  94.77708764 101.39875775]
 [ 53.34314575  34.17157288  81.627417    48.3137085 ]
 [ 47.6862915   31.34314575 104.254834    59.627417  ]
 [ 42.02943725  28.51471863 126.88225099  70.9411255 ]
 [ 64.01857603  31.63343685  78.92569588  58.46625258]
 [ 61.03715206  26.26687371  90.85139176  79.93250517]]
```

```
img_clone = np.copy(img)
plt.axis('off')
# draw random anchor boxes
for i in range(0, 5):
    x1 = int(anchor_boxes[i][0])
    y1 = int(anchor_boxes[i][1])
    x2 = int(anchor_boxes[i][2])
    y2 = int(anchor_boxes[i][3])

    cv2.rectangle(img_clone, (x1, y1), (x2, y2), color=(255, 0, 0),
                  thickness=1)

# draw ground truth boxes
for target in targets[:1]:
    bbox = target['boxes'].cpu().numpy()
    for i in range(len(bbox)):
        cv2.rectangle(img_clone, (bbox[i][0], bbox[i][1]),
                                 (bbox[i][2], bbox[i][3]),
                      color=(0, 255, 0), thickness=1)

plt.imshow(img_clone)
plt.show()
```

왼쪽은 생성된 앵커박스 좌표입니다.
순서대로 [x1, y1, x2, y2]

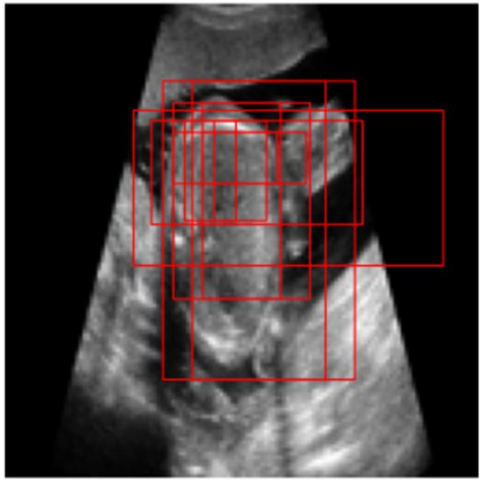


그림3. anchor box시각화

3) Anchor Target layer

```
index_inside_ac_hc = np.where(
    (anchor_boxes[:, 0] >= 35) &
    (anchor_boxes[:, 1] >= 11) &
    (anchor_boxes[:, 2] <= 97) &
    (anchor_boxes[:, 3] <= 120)
)[0]

index_inside_fl_hum = np.where(
    (anchor_boxes[:, 0] >= 42) &
    (anchor_boxes[:, 1] >= 9) &
    (anchor_boxes[:, 2] <= 90) &
    (anchor_boxes[:, 3] <= 112)
)[0]

index_inside = np.concatenate((index_inside_ac_hc, index_inside_fl_hum))
print(index_inside.shape)

valid_anchor_boxes = anchor_boxes[index_inside]
print(valid_anchor_boxes.shape)
```

```
(135,)
(135, 4)
```

RPN을 학습시키기 위해 적절한 anchor box를 선택하는 작업을 수행합니다.

Bounding Box의 X축 시작점, Y축 시작점, X축 종점, Y축 최소, 최대값을 이용하여 bounding box 경계 내부에 있는 anchor box만을 선택합니다.

384개의 anchor box 중 135개의 유효한 anchor box를 얻었습니다.

```
# ious 배열을 (앵커 박스 개수, 객체 개수) 크기로 수정
ious = np.empty((len(valid_anchor_boxes), len(bbox)), dtype=np.float32)
ious.fill(0)

# anchor boxes
for i, anchor_box in enumerate(valid_anchor_boxes):
    xa1, ya1, xa2, ya2 = anchor_box
    anchor_area = (xa2 - xa1) * (ya2 - ya1)

    # ground truth boxes
    for j, gt_box in enumerate(bbox):
        xb1, yb1, xb2, yb2 = gt_box
        box_area = (xb2 - xb1) * (yb2 - yb1)

        inter_x1 = max([xb1, xa1])
        inter_y1 = max([yb1, ya1])
        inter_x2 = min([xb2, xa2])
        inter_y2 = min([yb2, ya2])

        if (inter_x1 < inter_x2) and (inter_y1 < inter_y2):
            inter_area = (inter_x2 - inter_x1) * (inter_y2 - inter_y1)
            iou = inter_area / (anchor_area + box_area - inter_area)
        else:
            iou = 0

        ious[i, j] = iou


print(ious.shape)
print(ious[1:10, :])
```

```
[[0.67445534 0.38754395 0.42394498 ... 0.19520983 0.07241058 0.1225    ]
 [0.17825311 0.11787696 0.12792526 ... 0.2096538  0.2423649  0.24268131]
 [0.32068124 0.32869935 0.31820074 ... 0.196875   0.091875   0.1225    ]
 ...
 [0.13842614 0.21857923 0.2224694  ... 0.29934174 0.23573884 0.23074365]
 [0.2533658  0.68410724 0.6088873  ... 0.12280016 0.07755692 0.0934417 ]
 [0.08506482 0.19432895 0.17953794 ... 0.13713244 0.2827384  0.17461102]]
```

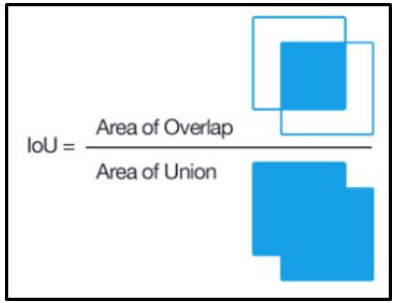


그림4. 유효한 앵커 박스의 IoU

※ IoU: 각 anchor box와 ground truth bounding box가 overlab되는 정도를 비교합니다.

3)-1 Sample positive/negative anchor boxes

```
pos_iou_threshold = 0.7
neg_iou_threshold = 0.3

label[gt_argmax_ious] = 1
label[max_ious >= pos_iou_threshold] = 1
label[max_ious < neg_iou_threshold] = 0

n_sample = 40
pos_ratio = 0.5
n_pos = int(pos_ratio * n_sample)

pos_index = np.where(label == 1)[0]

if len(pos_index) > n_pos:
    disable_index = np.random.choice(pos_index,
                                     size = (len(pos_index) - n_pos),
                                     replace=False)
    label[disable_index] = -1

n_neg = n_sample * np.sum(label == 1)
neg_index = np.where(label == 0)[0]

if len(neg_index) > n_neg:
    disable_index = np.random.choice(neg_index,
                                     size = (len(neg_index) - n_neg),
                                     replace = False)
    label[disable_index] = -1
```

```
argmax_ious = ious.argmax(axis=1)
print(argmax_ious.shape)
print(argmax_ious)

max_ious = ious[np.arange(len(index_inside)), argmax_ious]
print(max_ious)
```

```
[0.57701576 0.9069559  0.7672299
```

max_iou는 각 유효한 앵커 박스와 연결된 가장 높은 IoU값들입니다.

IoU > 0.7인 anchor box를 positive sample,

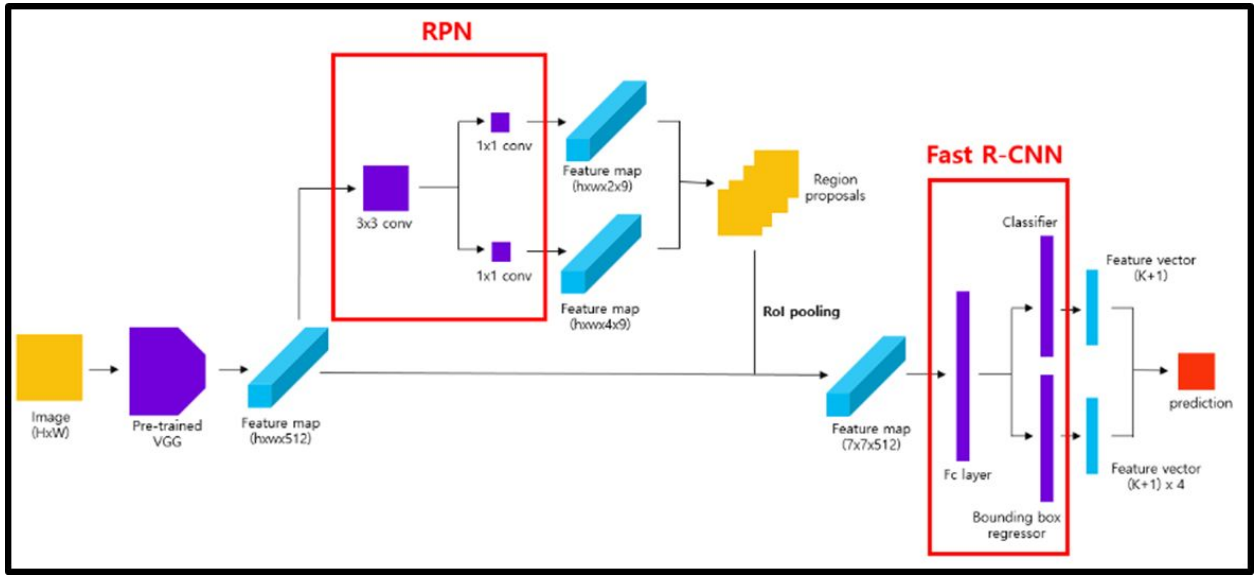
IoU < 0.3 미만인 anchor box는 negative sample로 저장합니다. (positive sample과 negative sample의 비율을 1:1로 지정)

label 변수에 positive sample일 경우 1, negative sample일 경우 0으로 저장합니다. (0.3<IoU<0.7는 -1로 비교 대상에서 제외)

```
label[max_ious >= pos_iou_threshold] = 1
print("label after positive assignment:", label)

label[max_ious < neg_iou_threshold] = 0
print("label after negative assignment:", label)
```

```
label after negative assignment: [-1  1  1  1 -1  1  1 -1  1 -1 -1  0 -1
  1  1  1 -1  1  1  1  1  1  1 -1  0 -1  1 -1  1 -1  1 -1 -1  1 -1 -1  0
 -1  1 -1 -1  1 -1 -1 -1  0 -1  1 -1 -1  1 -1  0  0  0 -1  1 -1  0  0  0
  0 -1 -1 -1  0  0  0  0  1  1  1 -1  1  1 -1 -1 -1 -1  1  1  1  1  1 -1
 -1 -1  1  1  1  1  1  1  1  1 -1 -1  1 -1 -1  1 -1 -1 -1  1 -1  1 -1 -1
  1 -1 -1  1 -1 -1 -1  0 -1 -1 -1  0  0  1  0]
```

4) RPN




```
in_channels = 512
mid_channels = 512
n_anchor = 6

num_classes = 2

# bounding box regressor 출력 채널 수
reg_output_channels = n_anchor * 4 # 36

# Objectness 출력 채널 수
obj_output_channels = n_anchor * num_classes # 18

conv1 = nn.Conv2d(in_channels, mid_channels, 3, 1, 1).to(DEVICE)
conv1.weight.data.normal_(0, 0.01)
conv1.bias.data.zero_()

# bounding box regressor
reg_layer = nn.Conv2d(mid_channels, reg_output_channels, 1, 1, 0).to(DEVICE)
reg_layer.weight.data.normal_(0, 0.01)
reg_layer.bias.data.zero_()

# Objectness
obj_layer = nn.Conv2d(mid_channels, obj_output_channels, 1, 1, 0).to(DEVICE)
obj_layer.weight.data.normal_(0, 0.01)
obj_layer.bias.data.zero_()
```

RPN에서는 후보 영역이 어떤 class에 해당하는지까지 구체적인 분류를 하지 않고 객체가 포함되어 있는지 여부만을 분류합니다.

앞선 1)과정을 통해 얻은 feature map에 3x3 conv 연산을 적용하는 layer를 정의하고,
Objectness와 Bounding Box Regressor를 얻기 위해 feature map에 대하여 1x1 conv 연산을 적용합니다.

※ 각 Channel 수

•bounding box regressor: 6x4(anchor box의 수 x bounding box coordinates)

•Objectness: 6x2(anchor box의 수 x Object/Non-Object)

4) RPN

```
# 모델의 출력 얻기
x = conv1(output_map.to(DEVICE))
pred_anchor_locs = reg_layer(x)
pred_obj_scores = obj_layer(x)

print(pred_anchor_locs.shape, pred_obj_scores.shape)
```

```
torch.Size([800, 24, 8, 8]) torch.Size([800, 12, 8, 8])
```

8x8x512 크기의 feature map을 Bounding box regressor, Objectness에 입력하여 각각 bounding box 위치 (=**pred_anchor_locs**)와 objectness score(=**pred_obj_scores**)를 얻습니다. 이를 target 값과 비교하기 위해 적절하게 resize해줍니다.

```
pred_anchor_locs = pred_anchor_locs.view(800, -1, 4)
print("pred_anchor_locs.shape:", pred_anchor_locs.shape)

# 분류
pred_obj_scores = pred_obj_scores.view(800, -1, 2)
print("pred_obj_scores.shape:",pred_obj_scores.shape)

objectness_score = pred_obj_scores.view(800, 8, 8, 6, 2)[:, :, :, :, 1].contiguous().view(800, -1)
print("objectness_score.shape:", objectness_score.shape)
print("anchor_locations.shape:", anchor_locations.shape)
print("anchor_labels.shape:", anchor_labels.shape)
```

```
pred_anchor_locs.shape: torch.Size([800, 384, 4])
pred_obj_scores.shape: torch.Size([800, 384, 2])
objectness_score.shape: torch.Size([800, 384])
anchor_locations.shape: (384, 4)
anchor_labels.shape: (384,)
```

※ bounding box regressor: [800, 24(6*4), 8, 8] => [800, 384(8*8*6), 4] (dy, dx, dh, dw)

※ Objectness: [800, 12(6*2), 8, 8] => [800, 384, 2] (object/non-object)

4)-1 RPN loss 계산

Objectness loss 계산

```
# 모델의 손실 함수 계산
rpn_loc = pred_anchor_locs[0]
rpn_score = pred_obj_scores[0]

gt_rpn_loc = torch.from_numpy(anchor_locations)
gt_rpn_score = torch.from_numpy(anchor_labels)

print(rpn_loc.shape,
      rpn_score.shape,
      gt_rpn_loc.shape,
      gt_rpn_score.shape)
```

```
rpn_obj_loss = F.cross_entropy(rpn_score, gt_rpn_score.long().to(DEVICE), ignore_index = -1)
print(rpn_obj_loss)
```

```
torch.Size([384, 4]) torch.Size([384, 2]) torch.Size([384, 4]) torch.Size([384])
tensor(0.6967, grad_fn=<NllLossBackward0>)
```

※ rpn_obj_loss: RPN의 분류 손실값 (Object일 확률) 작을수록 더 정확한 분류를 의미

위치(Bounding Box) loss 계산

SmoothL1(x): ((x < 1).float() * 0.5 * x ** 2) + ((x >= 1).float() * (x - 0.5)) https://wikidocs.net/148635

```
# only positive samples(객체가 있는 것만)
pos = gt_rpn_score > 0
mask = pos.unsqueeze(1).expand_as(rpn_loc)
print(mask.shape)

# take those bounding boxes which have positive labels
mask_loc_preds = rpn_loc[mask].view(-1, 4)
mask_loc_targets = gt_rpn_loc[mask].view(-1, 4)
print(mask_loc_preds.shape, mask_loc_targets.shape)

x = torch.abs(mask_loc_targets.cpu() - mask_loc_preds.cpu())
rpn_loc_loss = ((x < 1).float() * 0.5 * x ** 2) + ((x >= 1).float() * (x - 0.5))
#print(rpn_loc_loss)
print(rpn_loc_loss.sum())
```

```
torch.Size([384, 4])
torch.Size([9, 4]) torch.Size([9, 4])
tensor(3.2920, dtype=torch.float64, grad_fn=<SumBackward0>)
```

※ rpn_loc_loss: RPN의 위치 손실값, (객체가있는) 각 앵커박스의 예측된 위치와 실제 위치 간의 차이를 계산

```
#Combining both the rpn_obj_loss and rpn_reg_loss

rpn_lambda = 10
N_reg = (gt_rpn_score > 0).float().sum()
rpn_loc_loss = rpn_loc_loss.sum() / N_reg
rpn_loss = rpn_obj_loss + (rpn_lambda * rpn_loc_loss)
print(rpn_loss)
```

※ rpn_loss: RPN의 총 손실값

```
tensor(4.3545, dtype=torch.float64, grad_fn=<AddBackward0>)
```

< RPN의 loss를 계산 과정 >

Objectness loss는 cross entropy loss를 활용하여 구합니다.

Bounding box regression loss는 오직 positive에 해당하는 sample에 대해서만 loss를 계산하므로, positive/negative 여부를 저장하는 배열인 mask를 생성해줍니다.

이를 활용하여 Smooth L1 loss를 구해줍니다. Objectness loss와 Bounding box regression loss 사이를 조정하는 balancing parameter λ=10으로 지정해주고 두 loss를 더해 multi-task loss를 구합니다.

5) Proposal layer

Proposal layer에서는 Anchor generation layer에서 생성된 anchor boxes와 RPN에서 반환한 class scores와 bounding box regressor를 사용하여 region proposals를 추출하는 작업을 수행합니다.

```
# Send the 384 ROIs predicted by RPN to Fast RCNN to predict bbox + classifications
# First use NMS (Non-maximum supression) to reduce 384 ROI to 300 to 200

nms_thresh = 0.7  # non-maximum supression (NMS)

n_train_pre_nms = 250 # NMS 이전에 사용되는 훈련용 RPN 후보군
n_train_post_nms = 150 # NMS 이후에 사용되는 훈련용 RPN 후보군(최종 훈련)

n_test_pre_nms = 150 # NMS 이전에 사용되는 RPN 후보군

n_test_post_nms = 50 # NMS 이후에 사용되는 테스트용 (최종 테스트)
min_size = 16 # 후보군을 선택할 때 고려되는 최소한의 박스 크기 이 값보다 작은 크기의 박스는 제외함.
```

```
# clip the predcited boxes to the image

img_size = (128, 128)
roi[:, slice(0, 4, 2)] = np.clip(roi[:, slice(0, 4, 2)], 0, img_size[0]) # [:, 0, 2]
roi[:, slice(1, 4, 2)] = np.clip(roi[:, slice(1, 4, 2)], 0, img_size[1]) # [:, 1, 3]

print(roi.shape)
print(np.max(roi))
print(np.min(roi))
```

```
(384, 4)
128.0
15.573631187333568
```

RoI 형태는 (384, 4), 최대값: 128, 최소값: 15.57

※ RoI는 Region of Interest, 이미지에서 관심 영역을 의미함. 객체 후보 상자들에서 실제로 객체가 있는 것으로 추정되는 영역.

1. score 변수에 저장된 objectness score를 내림차순으로 정렬한 후 min_size(16)보다 큰 박스를 선택합니다. 384 -> 272

```
# remove predicted boxes with either height or width < threshold

hs = roi[:, 3] - roi[:, 1]
ws = roi[:, 2] - roi[:, 0]

keep = np.where((hs >= min_size) & (ws >= min_size))[0]
roi = roi[keep, :]
score = objectness_score_numpy[keep]
print(keep.shape, roi.shape, score.shape)
```

```
(272,) (272, 4) (272,)
```

2. 상위 N(n_train_pre_nms=250)개의 anchor box에 대하여 Non maximum suppression 알고리즘을 수행합니다.

272 -> 250

```
# take top pre_nms_topN (e.g.
order = order[:n_train_pre_nms]
roi = roi[order, :]
print(order.shape, roi.shape)
```

```
(250,) (250, 4)
```

3. NMS 수행 후 anchor box 중 상위 N(n_train_post_nms=150)개의 region proposals를 학습에 사용합니다. 250 -> 125

```
# take the indexes of order the probability score in descending order
# non maximum suppression

order = order.argsort()[::-1]
keep = []

while (order.size > 0):
  i = order[0] # take the 1st elt in roder and append to keep
  keep.append(i)

  xx1 = np.maximum(x1[i], x1[order[1:]])
  yy1 = np.maximum(y1[i], y1[order[1:]])
  xx2 = np.minimum(x2[i], x2[order[1:]])
  yy2 = np.minimum(y2[i], y2[order[1:]])

  w = np.maximum(0.0, xx2 - xx1 + 1)
  h = np.maximum(0.0, yy2 - yy1 + 1)

  inter = w * h
  ovr = inter / (areas[i] + areas[order[1:]] - inter)
  inds = np.where(ovr <= nms_thresh)[0]
  order = order[inds + 1]

keep = keep[:n_train_post_nms] # while training/testing, use accordingly
roi = roi[keep]
print(len(keep), roi.shape)
```

```
140 (140, 4)
```

※ 140: 150개의 RoI가 NMS에서 중복된 일부 RoI가 제거 되기 때문에 140이 출력됩니다.

각 RoI는 4개의 좌표값으로 표현되고 총 140개의 RoI가 최종 학습에 선택됩니다.

6) Proposal Target layer

```
n_sample = 50 #학습 데이터에서 n개의 RoI를 랜덤하게 선택하여 사용.
pos_ratio = 0.5 # positive 비율
pos_iou_thresh = 0.5 # positive으로 분류하기 위한 IoU 임계값(0.5)

# 0 < iou < 0.5 이면 negative으로 분류.
neg_iou_thresh_hi = 0.5
neg_iou_thresh_lo = 0.0
```

```
# fine the iou of each ground truth object with the region proposals

ious = np.empty((len(roi), bbox.shape[0]), dtype = np.float32)
ious.fill(0)

for num1, i in enumerate(roi):
  ya1, xa1, ya2, xa2 = i
  anchor_area = (ya2 - ya1) * (xa2 - xa1)

  for num2, j in enumerate(bbox):
    yb1, xb1, yb2, xb2 = j
    box_area = (yb2 - yb1) * (xb2 - xb1)
    inter_x1 = max([xb1, xa1])
    inter_y1 = max([yb1, ya1])
    inter_x2 = min([xb2, xa2])
    inter_y2 = min([yb2, ya2])

    if (inter_x1 < inter_x2) and (inter_y1 < inter_y2):
      inter_area = (inter_y2 - inter_y1) * (inter_x2 - inter_x1)
      iou = inter_area / (anchor_area + box_area - inter_area)
    else:
      iou = 0
    ious[num1, num2] = iou

print(ious.shape)
```

```
(140, 800)
```

```
gt_assignment = ious.argmax(axis=1)
max_iou = ious.max(axis=1)

print(gt_assignment)
print(max_iou)

# assign the labels to each proposal
gt_roi_label = labels[gt_assignment]
print(gt_roi_label)
```

•ious: RoI(140개)와 Ground Truth Bounding Box(800개) 간의 region proposal를 추출.(=IoU값)

•

•**max_iou**: 각 RoI가 가장 높은 IoU값을 갖는 Ground truth bounding box와의 IoU값.

```
[0.02429135 0.         0.         0.         0.00917011 0.02921512
 0.05185021 0.13401784 0.10419289 0.04583239 0.32181886 0.12677275
 0.12027325 0.22848383 0.26104292 0.18703261 0.27990898 0.40251598
 0.26103383 0.3713286  0.41013017 0.         0.         0.
```

첫번째 RoI의 경우 가장 높은 IoU가 0.02이며 두번째 RoI는 어떤 Ground Truth Bounding Box와도 IoU가 0이다.

이러한 IoU값들은 RoI들과 Ground Truth 간의 겹치는 정도를 나타내며, 이를 기반으로 RoI를 양성(max_iou≥0.5)과 음성(0≤max_ioi<0.5)으로 분류합니다.

•gt_assignmet: 각 RoI에 가장 높은 IoU를 가지는 Ground Truth Bounding Box의 인덱스를 저장.

•**gt_roi_label**: 각 RoI에 할당된 레이블.(Ground Truth Bounding Box의 레이블 값 할당.)

```
[1 1 1 1 1 1 1 1 1 2 1 1 1 2 2 1 1 1 1 1 1 1 2 2 1 2 2 2
 1 2 1 1 2 1 1 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 1 2 4
 1 1 1 1 1 1 2 2 2 4 2 4 1 1 1 1 1 1 1 1 2 1 1 1 2 3 1 1 2
 1 1 1 2 1 1 2 2 1 1 1 1 1 1 1 1 1 1 2 1 1 2 1 2 4 2 2 3
```

6) Proposal Target layer

```
# select the foreground rois as per the pos_iou_thresh
# and n_sample x pos_ratio (100 x 0.25 = 25) foreground samples

pos_roi_per_image = n_sample * pos_ratio # 50
pos_index = np.where(max_iou >= pos_iou_thresh)[0]
pos_roi_per_this_image = int(min(pos_roi_per_image, pos_index.size))

if pos_index.size > 0:
  pos_index = np.random.choice(
      pos_index, size=pos_roi_per_this_image, replace=False)

print(pos_roi_per_this_image)
print(pos_index)
```

```
24
[ 26 103 134  29  64 117  96 101  93  60  97  63  45  68  67 119 132 100
  27 138 139 133 104  81]
```

```
# similarly we do for negative(background) region proposals

neg_index = np.where((max_iou < neg_iou_thresh_hi) &
                     (max_iou >= neg_iou_thresh_lo))[0]

neg_roi_per_this_image = n_sample - pos_roi_per_this_image
neg_roi_per_this_image = int(min(neg_roi_per_this_image, neg_index.size))

if neg_index.size > 0:
 neg_index = np.random.choice(
    neg_index, size = neg_roi_per_this_image, replace=False)

print(neg_roi_per_this_image)
print(neg_index)
```

```
26
[ 32 126  16  91  62   3  30  37  82 107  21  43  86 137  94 123  95  73
  75 112  23 115  46  20  44 108]
```

n_sample: 학습 데이터 140개에서
50개의 RoI를 랜덤하게 선택하여 사용.

positive RoI가 24개, negative RoI가 26개

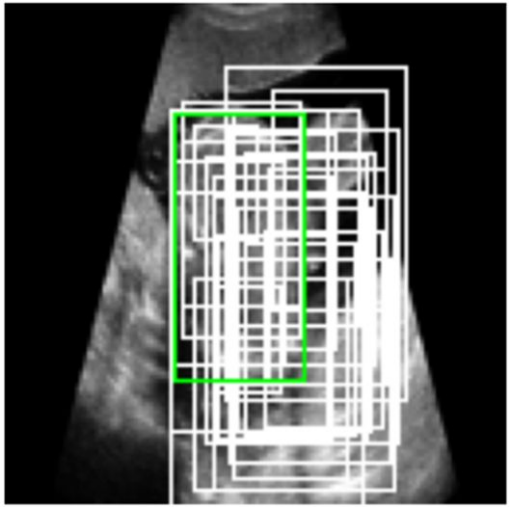


그림4. display RoI samples with **positive**

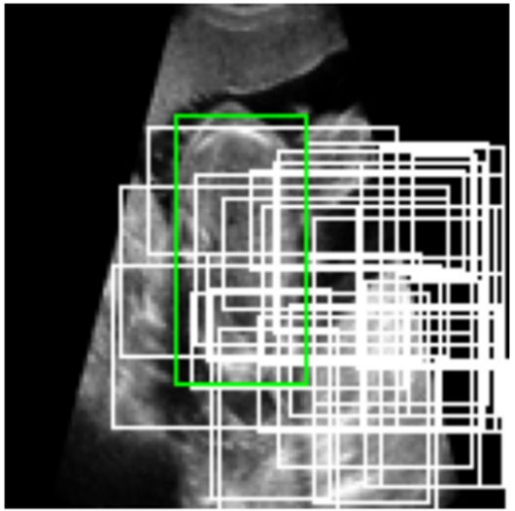


그림5. display RoI samples with **negative**

7) RoI pooling

```
# Bounding Box
rois = torch.from_numpy(sample_roi).float()
roi_indices = 0 * np.ones((len(rois),), dtype=np.int32)
roi_indices = torch.from_numpy(roi_indices).float()
print(rois.shape, roi_indices.shape)
```

```
torch.Size([50, 4]) torch.Size([50])
```

```
indices_and_rois = torch.cat([roi_indices[:, None], rois], dim=1)
xy_indices_and_rois = indices_and_rois[:, [0, 2, 1, 4, 3]]
indices_and_rois = xy_indices_and_rois.contiguous()
print(xy_indices_and_rois.shape)
```

```
torch.Size([50, 5])
```

```
size = (7, 7)
adaptive_max_pool = nn.AdaptiveMaxPool2d(size[0], size[1])
```

```
output = []
rois = indices_and_rois.data.float()
rois[:, 1:].mul_(1/16.0) # sub-sampling ratio
rois = rois.long()
num_rois = rois.size(0)
print(num_rois)

for i in range(num_rois):
  roi = rois[i]
  im_idx = roi[0]
  im = output_map.narrow(0, im_idx, 1)[..., roi[1]:(roi[3]+1), roi[2]:(roi[4]+1)]
  tmp = adaptive_max_pool(im)
  output.append(tmp[0])

output = torch.cat(output, 0)

print(output.size())
```

```
50
torch.Size([50, 512, 7, 7])
```

```
k = output.view(output.size(0), -1)
print(k.shape) # 25088 = 7*7*512
```

```
torch.Size([50, 25088])
```

RoI 개수는 50 즉, feature map의 수는 50, feature map의 크기는 (7, 7)

이렇게 생성된 feature map들은 이후에 객체 감지 모델의 입력으로 사용되어 객체의 특징을 추출하는 데 활용됩니다.

k에는 feature map들이 1차원으로 펼쳐진 결과가 저장되어 있음. 각 RoI마다 25088개의 특징을 가지고 있습니다.

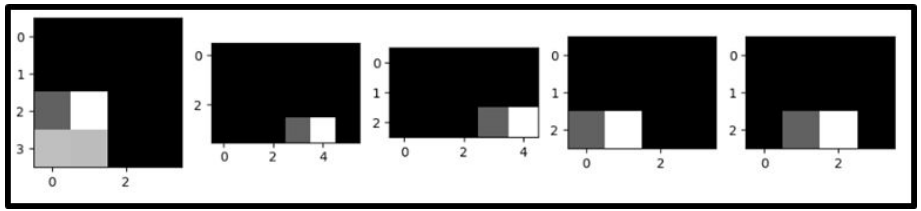


그림6. 처음 5개의 RoI의 Feature map을 시각화

8) Fast R-CNN

```
roi_head_classifier = nn.Sequential(*[nn.Linear(25088, 1000), nn.Linear(1000, 512)]).to(DEVICE) # 512

# Bounding Box regressor
loc_2 = nn.Linear(512, 5*4).to(DEVICE) # 5개의 클래스 (AC, FL, HC, HW, 배경), 각 클래스당 4개의 좌표
loc_2.weight.data.normal_(0, 0.01)
loc_2.bias.data.zero_()

# Classifier
classifier = nn.Linear(512, 5).to(DEVICE) # AC, FL, HC, HW, 배경에 대한 점수, 각 클래스는 1, 2, 3, 4, 0에 해당합니다

# k의 크기가 (배치 크기, 25088)이므로, 512로 변경하기 위해 이전 레이어의 출력 크기가 1000이 맞는지 확인
k = roi_head_classifier(k.to(DEVICE))
roi_loc_2 = loc_2(k)
roi_classifier = classifier(k)
print(roi_loc_2.shape, roi_classifier.shape)
```

```
torch.Size([50, 20]) torch.Size([50, 5])
```

```
print(roi_loc_2.shape)
print(roi_classifier.shape)

#actual
print(gt_roi_locs.shape)
print(gt_roi_labels.shape)
```

```
torch.Size([50, 20])
torch.Size([50, 5])
(50, 4)
(50,)
```

ROI Classification loss

```
gt_roi_loc = torch.from_numpy(gt_roi_locs)
gt_roi_label = torch.from_numpy(np.float32(gt_roi_labels)).long()
print(gt_roi_loc.shape, gt_roi_label.shape)

roi_cls_loss = F.cross_entropy(roi_classifier.cpu(), gt_roi_label.cpu(), ignore_index=-1)
print(roi_cls_loss)
```

```
torch.Size([50, 4]) torch.Size([50])
tensor(1.6613, grad_fn=<NllLossBackward0>)
```

※ roi_cls_loss: 클래스 판별, 각 클래스에 해당할 확률

```
n_sample = roi_loc_2.shape[0]
roi_loc = roi_loc_2.view(n_sample, -1, 4)
print(roi_loc.shape)
```

```
torch.Size([50, 5, 4])
```

```
roi_loc = roi_loc[torch.arange(0, n_sample).long(), gt_roi_label]
print(roi_loc.shape)
```

- roi_head_classifier: FC Layer (RoI Pooling에서 추출한 K를 FC Layer에 전달한다.)

- **roi_loc_2**: Bounding Box Regressor를 거쳐서 클래스별 bounding box의 예측값 (50, 20)
- **roi_classifier**: Classifier를 거쳐서 얻은 클래스별 점수 (50, 5)

- gt_roi_locs: Ground truth Bounding Box 좌표
- gt_roi_labels: Ground truth RoI의 클래스 레이블

8) Fast R-CNN

ROI Bounding Box Regressor loss, mask로 학습

```
# for regression we use smooth l1 loss as defined in the Fast R-CNN paper
pos = gt_roi_label > 0
mask = pos.unsqueeze(1).expand_as(roi_loc)
print(mask.shape)
```

```
torch.Size([50, 4])
```

```
# take those bounding boxes which have positive labels
mask_loc_preds = roi_loc[mask].view(-1, 4)
mask_loc_targets = gt_roi_loc[mask].view(-1, 4)
print(mask_loc_preds.shape, mask_loc_targets.shape)

x = torch.abs(mask_loc_targets.cpu() - mask_loc_preds.cpu())
roi_loc_loss = ((x < 1).float() * 0.5 * x ** 2) + ((x >= 1).float() * (x - 0.5))
#print(roi_loc_loss.shape)
# print(roi_loc_loss)
print(roi_loc_loss.sum())
```

```
torch.Size([24, 4]) torch.Size([24, 4])
tensor(1.2964, dtype=torch.float64, grad_fn=<SumBackward0>)
```

24: 선택된 Positive RoI 개수
4: Positive RoI에 대한 실제 바운딩 박스 좌표

```
roi_lambda = 10.
roi_loss = roi_cls_loss + (roi_lambda * roi_loc_loss)
print(roi_loss)
```

※ roi_loc_loss: Positive RoI들의 Bounding Box Regressor 손실값

```
tensor([[1.7084, 1.8641, 2.2574, 1.6351],
        [1.6243, 1.6226, 1.6764, 1.6576],
        [1.6628, 1.6226, 1.6573, 1.6496],
        [1.6230, 1.9677, 1.6496, 1.9487],
        [1.6770, 1.6849, 1.6388, 1.6657],
        [1.7657, 1.6257, 1.7384, 1.6535],
        [1.6975, 1.6239, 1.8295, 1.6873],
        [1.6226, 1.6280, 1.6248, 1.6243],
        [1.8555, 1.6518, 2.0551, 1.6851],
        [2.0242, 1.6417, 1.6274, 1.6458],
        [1.6540, 1.6232, 1.6560, 1.6239],
        [1.7300, 1.6301, 1.7522, 1.6504],
        [1.7144, 1.9522, 1.8196, 2.6416],
        [1.6432, 1.6420, 1.6850, 1.6442],
        [1.6347, 1.6331, 1.8600, 1.7299],
        [1.6228, 1.6244, 1.6487, 1.6237],
        [1.7110, 1.7187, 2.5699, 1.7693],
        [1.6285, 1.6270, 1.7393, 1.6255],
        [1.6230, 2.1197, 1.6226, 1.6446],
        [1.6227, 1.6250, 1.6439, 1.6261],
        [1.8458, 1.6801, 2.3018, 1.6667],
        [1.9068, 1.6248, 1.6227, 1.6344],
        [1.6309, 1.6235, 1.7527, 1.6238],
        [1.6834, 1.6239, 1.7909, 1.6329]], dtype=torch.float64,
       grad_fn=<AddBackward0>)
```

roi_loss: roi_loc_loss(Bounding Bounding Box Regressor)와 roi_cls_loss(Classifier loss)의 총 손실
형태는 [24, 4]로 각 행은 클래스 AC, HC, FL, HUM에 대한 분류 손실과 각 양성 RoI의 바운딩 박스 좌표에 대한 회귀손실을 포함합니다.

rpn_loss는 λ =10으로 설정하고 rpn_obj_loss, rpn_loc_loss를 더해 multi task loss를 구함.
roi_loss는 λ =10으로 설정하고 roi_cls_loss, roi_loc_loss를 더해 multi task loss를 구함.
∴ total_loss = rpn_loss + roi_loss
total_loss가 최소화 되는 것이 모델의 목표이며, 이를 위해 네트워크의 가중치가 업데이트되고 모델이 학습됩니다.

```
total_loss = rpn_loss + roi_loss
print(total_loss)
```

```
tensor([[6.0628, 6.2186, 6.6119, 5.9896],
        [5.9787, 5.9771, 6.0308, 6.0120],
        [6.0172, 5.9771, 6.0118, 6.0041],
        [5.9775, 6.3222, 6.0041, 6.3032],
        [6.0315, 6.0394, 5.9932, 6.0202],
        [6.1201, 5.9802, 6.0929, 6.0079],
        [6.0520, 5.9784, 6.1839, 6.0418],
        [5.9771, 5.9824, 5.9793, 5.9788],
        [6.2099, 6.0063, 6.4095, 6.0395],
        [6.3787, 5.9961, 5.9818, 6.0002],
        [6.0084, 5.9777, 6.0105, 5.9784],
        [6.0844, 5.9845, 6.1066, 6.0049],
        [6.0689, 6.3067, 6.1740, 6.9960],
        [5.9976, 5.9964, 6.0395, 5.9986],
        [5.9891, 5.9875, 6.2145, 6.0844],
        [5.9772, 5.9788, 6.0031, 5.9781],
        [6.0655, 6.0731, 6.9244, 6.1237],
        [5.9830, 5.9815, 6.0937, 5.9799],
        [5.9775, 6.4742, 5.9771, 5.9991],
        [5.9772, 5.9795, 5.9984, 5.9805],
        [6.2002, 6.0346, 6.6563, 6.0212],
        [6.2612, 5.9793, 5.9771, 5.9889],
        [5.9853, 5.9780, 6.1072, 5.9783],
        [6.0378, 5.9783, 6.1454, 5.9874]], dtype=torch.float64,
       grad_fn=<AddBackward0>)
```

9) Training

PyTorch

```
>>> model =
torchvision.models.detection.fasterrcnn_resnet50_fpn(weights=FasterRCNN_ResNet50_FPN_Weights
.DEFAULT)
>>> # For training
>>> images, boxes = torch.rand(4, 3, 600, 1200), torch.rand(4, 11, 4)
>>> boxes[:, :, 2:4] = boxes[:, :, 0:2] + boxes[:, :, 2:4]
>>> labels = torch.randint(1, 91, (4, 11))
>>> images = list(image for image in images)
>>> targets = []
>>> for i in range(len(images)):
>>>     d = {}
>>>     d['boxes'] = boxes[i]
>>>     d['labels'] = labels[i]
>>>     targets.append(d)
>>> output = model(images, targets)
>>> # For inference
>>> model.eval()
>>> x = [torch.rand(3, 300, 400), torch.rand(3, 500, 400)]
>>> predictions = model(x)
>>>
>>> # optionally, if you want to export the model to ONNX:
>>> torch.onnx.export(model, x, "faster_rcnn.onnx", opset_version = 11)
```

images

(4, 3, 600, 1200) ->(이미지 개수, C, H, W)

boxes

(4, 11, 4) -> (이미지 개수, 클래스 개수, 좌표)

labels

(1, 91, (4, 11)) -> (1, anchor box 개수, (이미지 개수, 클래스 개수))

targets은 list 형태 (bounding box 좌표와 해당 객체의 클래스 라벨 정보를 담고 있음)

boxes와 labels는 dictionary 형태

```
[{'boxes': tensor([[ 48,  42,  80, 100]]), 'labels': tensor([2])},
 {'boxes': tensor([[55, 30, 77, 46]]), 'labels': tensor([4])},
 {'boxes': tensor([[ 45,  48,  85, 114]]), 'labels': tensor([2])},
 {'boxes': tensor([[50, 21, 78, 28]]), 'labels': tensor([3])},
 {'boxes': tensor([[ 55,  34,  91, 106]]), 'labels': tensor([1])},
 {'boxes': tensor([[56, 55, 79, 61]]), 'labels': tensor([4])},
 {'boxes': tensor([[58, 46, 76, 62]]), 'labels': tensor([4])},
 {'boxes': tensor([[59, 55, 83, 66]]), 'labels': tensor([4])}]
```

X (test 이미지)

[(3, 300, 400), ..] -> [(C, H, W), …]

model.train( )

model(images, targets)를 통해 학습을 진행합니다.

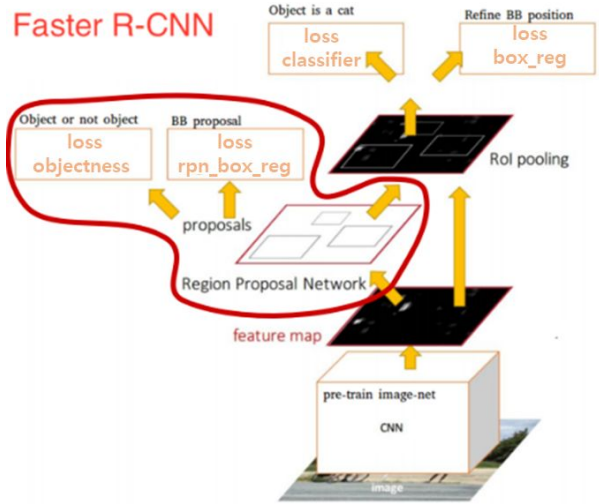
model.eval( )

model(x)를 통해 학습 시킨 모델로 새로운 이미지를 넣었을 때의 예측결과를 확인합니다.

9) Training

```
for epoch in range(num_epochs):
    model.train()
    total_loss = 0.0


    for i, data in enumerate(train_data_loader):
        images = data[0]
        images = list(image.to(device) for image in images)
        targets =  data[1]

        # 'labels'를 하나의 텐서로 합칩니다.
        labels = torch.cat([t['labels'] for t in targets], dim=0).to(device)

        optimizer.zero_grad()
        loss_dict = model(images, targets)
        print("loss_dict:", loss_dict)
        losses = sum(loss for loss in loss_dict.values())
        print("losses:", losses)

        losses.backward()
        optimizer.step()
        total_loss += losses.item()

    print('Epoch %d, loss: %.3f' % (epoch + 1, total_loss / len(train_data_loader)))
```

```
loss_dict: {'loss_classifier': tensor(0.1364, grad_fn=<NllLossBackward0>), 'loss_box_reg': tensor
(0.0224, grad_fn=<DivBackward0>), 'loss_objectness': tensor(0.1036, grad_fn=<BinaryCrossEntropyWith
LogitsBackward0>), 'loss_rpn_box_reg': tensor(0.0060, grad_fn=<DivBackward0>)}
losses: tensor(0.2684, grad_fn=<AddBackward0>)
loss_dict: {'loss_classifier': tensor(0.2428, grad_fn=<NllLossBackward0>), 'loss_box_reg': tensor
(0.0706, grad_fn=<DivBackward0>), 'loss_objectness': tensor(0.1850, grad_fn=<BinaryCrossEntropyWith
LogitsBackward0>), 'loss_rpn_box_reg': tensor(0.0156, grad_fn=<DivBackward0>)}
losses: tensor(0.5141, grad_fn=<AddBackward0>)
loss_dict: {'loss_classifier': tensor(0.2613, grad_fn=<NllLossBackward0>), 'loss_box_reg': tensor
(0.0760, grad_fn=<DivBackward0>), 'loss_objectness': tensor(0.0465, grad_fn=<BinaryCrossEntropyWith
LogitsBackward0>), 'loss_rpn_box_reg': tensor(0.0055, grad_fn=<DivBackward0>)}
losses: tensor(0.3893, grad_fn=<AddBackward0>)
```



※ loss_dict
loss_classifier(RoI)
loss_box_reg(RoI)
loss_objectness(RPN)
loss_rpn_box_reg(RPN)

※ losses: 네개의 loss값을 합산한 total loss

```
Epoch 5, loss: 0.055
```

 # 1
train 결과
epoch = 5일 때, train_data_loader가
loss: 0.055 값을 최종적으로 얻었습니다.

```
Valid_loss: 0.079
```

 # 2
학습시킨 모델에 valid_data_loader을 넣었을 때
Validation loss값은 0.079가 나오는 것을 확인할 수 있습니다.

출력값 설명

< train, model.train() >

outputs = model(images, targets)

```
outputs {'loss_classifier': tensor(0.0978, grad_fn=<NllLossBackward0>),
  'loss_box_reg': tensor(0.0260, grad_fn=<DivBackward0>),
  'loss_objectness': tensor(0.1393,
grad_fn=<BinaryCrossEntropyWithLogitsBackward0>),
  'loss_rpn_box_reg': tensor(0.0042, grad_fn=<DivBackward0>)}
```

※ train에서의 outputs은
4개의 loss값을 출력하고 이를 통해 최종 loss값을
얻어 모델을 학습시킵니다.

< eval, model.eval() >

outputs = model(images)

```
outputs [{'boxes': tensor([[48.0128, 25.1149, 85.8751, 87.2272],
            [50.6432, 24.1795, 83.9450, 80.5928],
            [49.9133, 25.1338, 85.8551, 78.0996]], grad_fn=<StackBackward0>),
              'labels': tensor([2, 1, 3]),
              'scores': tensor([0.9919, 0.1356, 0.0671], grad_fn=<IndexBackward0>)},
```

※ eval에서의 outputs은
하나의 이미지 당 여러 개의 bounding box 좌표,
label, score가 출력됩니다.

이 중 score가 가장 높은 값의 label을 해당
이미지의 class로 선정하고, class에 해당하는
boxes선택하여 객체를 예측합니다.

10) Eval

1. 새로운 이미지를 가지고 학습된 모델에 적용하여 예측되는 score, label, bounding box를 확인합니다.

2. 각각의 이미지에서 score(0~1)가 가장 높은 label을 해당 class로 지정하고 예측된 class의 bounding box 좌표를 predicted_labels & predicted_bounding_boxes 리스트에 넣어줍니다.

3. 예측 class와 정답 class를 비교하여 classification 정확도를 알아내고

4. predicted_bounding_boxes와 Ground Truth를 시각화하여 객체를 잘 감싸는지 확인합니다.

scores: 예측된 라벨에 대한 신뢰도 점수,
아래 그림은 가장 높은 신뢰도 값을 추출한 것

```
scores tensor(0.9974, grad_fn=<SelectBackward0>)
scores tensor(0.9058, grad_fn=<SelectBackward0>)
scores tensor(0.9791, grad_fn=<SelectBackward0>)
scores tensor(0.9130, grad_fn=<SelectBackward0>)
scores tensor(0.9919, grad_fn=<SelectBackward0>)
scores tensor(0.6439, grad_fn=<SelectBackward0>)
scores tensor(0.9558, grad_fn=<SelectBackward0>)
scores tensor(0.6531, grad_fn=<SelectBackward0>)
scores tensor(0.7080, grad_fn=<SelectBackward0>)
scores tensor(0.8820, grad_fn=<SelectBackward0>)
scores tensor(0.9984, grad_fn=<SelectBackward0>)
scores tensor(0.9937, grad_fn=<SelectBackward0>)
scores tensor(0.9892, grad_fn=<SelectBackward0>)
scores tensor(0.9987, grad_fn=<SelectBackward0>)
scores tensor(0.9988, grad_fn=<SelectBackward0>)
scores tensor(0.7477, grad_fn=<SelectBackward0>)
scores tensor(0.0780, grad_fn=<SelectBackward0>)
scores tensor(0.2516, grad_fn=<SelectBackward0>)
scores tensor(0.8902, grad_fn=<SelectBackward0>)
scores tensor(0.7463, grad_fn=<SelectBackward0>)
```

predicted_bounding_boxes: 예측된 클래스에 대한 바운딩 박스 모음

```
[[48.17478942871094, 32.84360885620117, 78.96611785888672, 93.57556152343751, [49.82625961303711,
6.264883041381836, 75.9918212890625, 82.81349182128906], [45.10122299194336, 26.005495071411133,
6.39801788330078, 88.78578948974611, [64.70689392089844, 31.614200592041016, 104.41425323486328,
```

predicted_labels: 예측된 클래스 모음

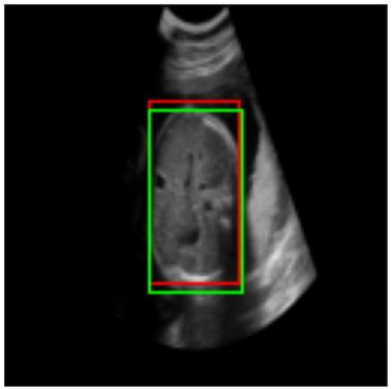
```
[1, 1, 1, 1, 1, 2, 2, 2, 4, 2, 3, 3, 3, 3, 3, 2, 2, 1, 4, 4]
```

20개 中 16개 정답 -> classification accuracy: 80%

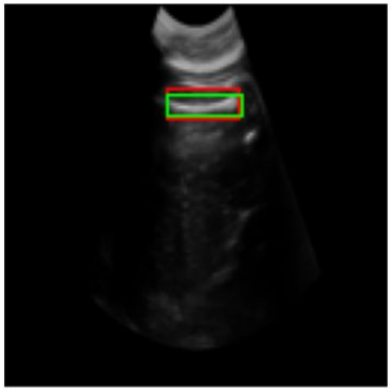
AC와 HC의 classification 성능은 good
FL과 HUM이 비슷하게 생겨 classification에서 엇갈리게 분류함.
ex) 2를 4로 분류하고, 4를 2로 분류함.

10) Eval

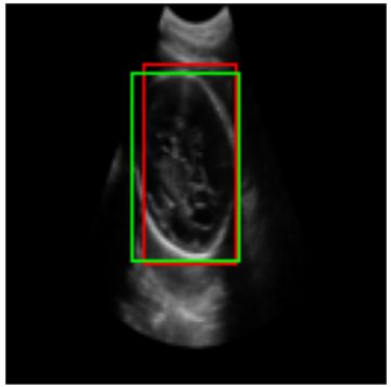
그림7. score가 가장 높은 predicted_bounding box와 Ground Truth Box 시각화 (차례대로 AC, FL, HC, HUM)



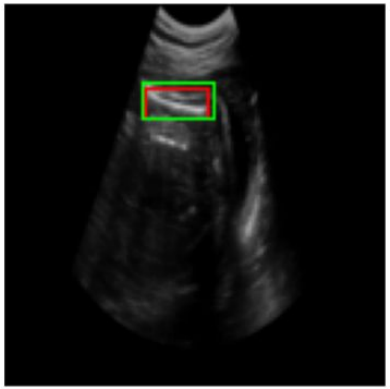
AC₩20170620_E0000924_I0033178.png



FL₩20170620_E0000924_I0033179.png



FL₩20170620_E0000924_I0033179.png



FL₩20170620_E0000924_I0033179.png

< 결론 >

Faster R-CNN모델을 활용해 Object Detection을 수행했을 때, Predicted bounding box가 객체를 잘 감싸는 것을 확인할 수 있고, 객체의 모양이 비슷할수록 Classification 정확도는 떨어지는 것을 확인할 수 있습니다.

< 개선점 >

이미지 개수 ↑, epoch ↑, 이미지 사이즈 (224, 224), batch = 16 or 32로 조정. (메모리 부족으로 인해 모두 downsize함.)