

**Deep Learning**

# YOLO 아키텍처 분석 (for v5)

(You Only Look Once)

강사 양석환



# YOLO 아키텍처 분석 (for v5)



- YOLO v5는 기본적으로 Backbone과 Head로 구성됨

- Backbone

- 이미지로부터 Feature map을 추출하는 부분
- CSP-Darknet 사용
  - YOLO v4의 Backbone과 유사
  - YOLO v3의 Backbone은 Darknet53 → CSP 미적용
- YOLO v5-(s / m / l / x) 까지 총 4가지 버전의 Backbone이 존재함
  - 본 과정에서는 제일 작은 모델인 YOLO v5-s를 기준으로 함

YOLO v5 아키텍처의 정보  
~/yolov5/models/yolov5s.yaml  
파일을 통해서 확인할 수 있음

참고/출처 URL  
<https://ropiens.tistory.com/44>

- YOLO v5는 기본적으로 Backbone과 Head로 구성됨

- Head

- 추출된 Feature map을 바탕으로 물체의 위치를 찾는 부분
- Anchor Box(Default Box)를 처음에 설정하고 이를 이용하여 최종적인 Bounding Box를 생성함
- YOLO v3와 동일하게 3가지의 scale에서 바운딩 박스를 생성함
  - 8픽셀 정보를 가진 작은 물체, 16픽셀 정보를 가진 중간 물체, 32픽셀 정보를 가진 큰 물체를 인식 가능
  - 각 스케일에서 3개의 앵커 박스를 사용 → 총 9개의 앵커 박스가 있음

- 아키텍처 정보 파일

- `~/yolov5/models/` 경로의 `yolo.py`, `common.py` 의 코드가 중심

- `yolo.py`

- YOLO 아키텍처에 관한 코드
    - 이 코드를 통해 YOLO 아키텍처가 생성됨

- `common.py`

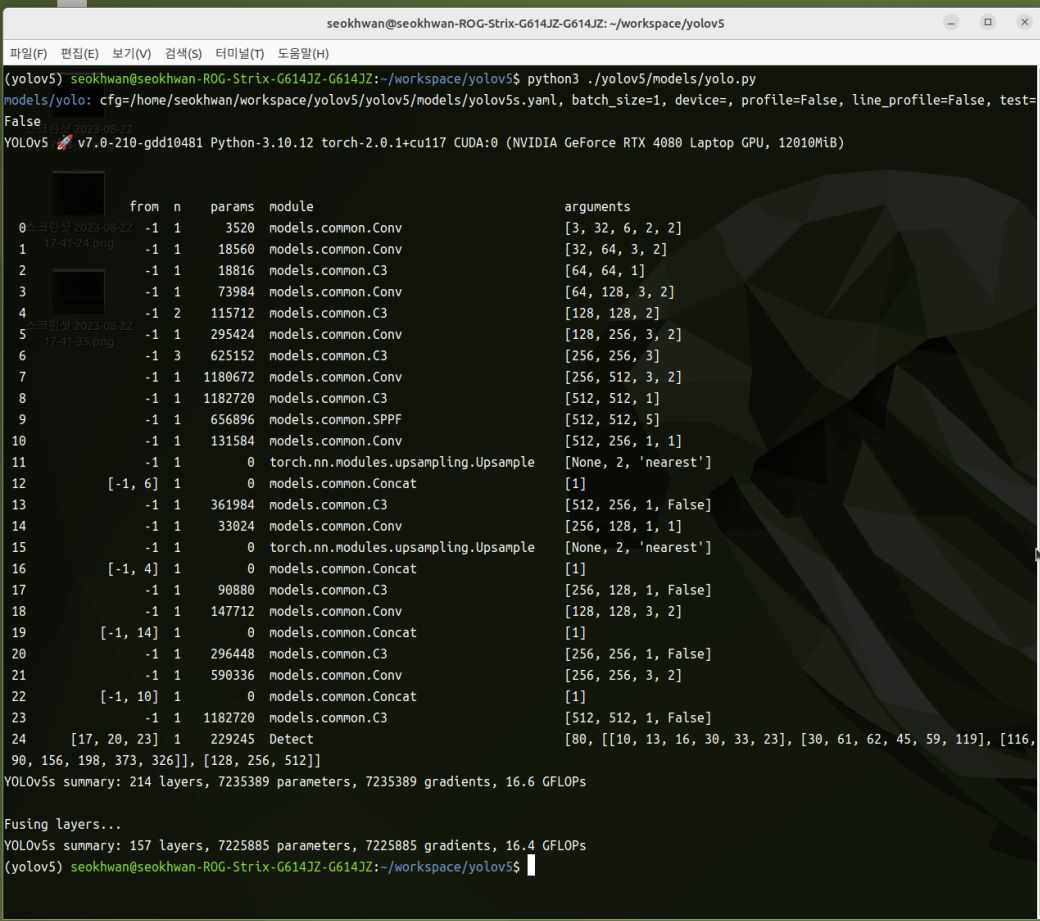
- YOLO 아키텍처를 구성하는 모듈(레이어)에 관한 코드
    - 이 코드에 `conv`, `BottleneckCSP` 등등 YOLO 모듈들이 구현되어 있음



- YOLO v5의 아키텍처 구조

- yolo.py를 실행시키면 아키텍처의 구조를 볼 수 있음

- python3 ./yolov5/models/yolo.py



```
seokhwan@seokhwan-ROG-Strix-G614JZ-G614JZ: ~/workspace/yolov5
(yolov5) seokhwan@seokhwan-ROG-Strix-G614JZ-G614JZ:~/workspace/yolov5$ python3 ./yolov5/models/yolo.py
models/yolo: cfg=/home/seokhwan/workspace/yolov5/yolov5/models/yolov5s.yaml, batch_size=1, device=, profile=False, line_profile=False, test=
False
YOLOv5 v7.0-210-gdd10481 Python-3.10.12 torch-2.0.1+cu117 CUDA:0 (NVIDIA GeForce RTX 4080 Laptop GPU, 12010MiB)

  from  n  params module                                arguments
  ----  -  -
0  0  -1  1  3520  models.common.Conv                [3, 32, 6, 2, 2]
1  1  -1  1  18560 models.common.Conv                [32, 64, 3, 2]
2  2  -1  1  18816 models.common.C3                  [64, 64, 1]
3  3  -1  1  73984 models.common.Conv                [64, 128, 3, 2]
4  4  -1  2  115712 models.common.C3                  [128, 128, 2]
5  5  -1  1  295424 models.common.Conv                [128, 256, 3, 2]
6  6  -1  3  625152 models.common.C3                  [256, 256, 3]
7  7  -1  1  1180672 models.common.Conv                [256, 512, 3, 2]
8  8  -1  1  1182720 models.common.C3                  [512, 512, 1]
9  9  -1  1  656896 models.common.SPPF                [512, 512, 5]
10 10 -1  1  131584 models.common.Conv                [512, 256, 1, 1]
11 11 -1  1  0      torch.nn.modules.upsampling.Upsample [None, 2, 'nearest']
12 12 [-1, 6] 1  0      models.common.Concat              [1]
13 13 -1  1  361984 models.common.C3                  [512, 256, 1, False]
14 14 -1  1  33024  models.common.Conv                [256, 128, 1, 1]
15 15 -1  1  0      torch.nn.modules.upsampling.Upsample [None, 2, 'nearest']
16 16 [-1, 4] 1  0      models.common.Concat              [1]
17 17 -1  1  90880  models.common.C3                  [256, 128, 1, False]
18 18 -1  1  147712 models.common.Conv                [128, 128, 3, 2]
19 19 [-1, 14] 1  0      models.common.Concat              [1]
20 20 -1  1  296448 models.common.C3                  [256, 256, 1, False]
21 21 -1  1  590336 models.common.Conv                [256, 256, 3, 2]
22 22 [-1, 10] 1  0      models.common.Concat              [1]
23 23 -1  1  1182720 models.common.C3                  [512, 512, 1, False]
24 24 [17, 20, 23] 1  229245 Detect                    [80, [[10, 13, 16, 30, 33, 23], [30, 61, 62, 45, 59, 119], [116,
90, 156, 198, 373, 326]], [128, 256, 512]]

YOLOv5s summary: 214 layers, 7235389 parameters, 7235389 gradients, 16.6 GFLOPs

Fusing layers...
YOLOv5s summary: 157 layers, 7225885 parameters, 7225885 gradients, 16.4 GFLOPs
(yolov5) seokhwan@seokhwan-ROG-Strix-G614JZ-G614JZ:~/workspace/yolov5$
```

## • YOLO v5의 모듈

### • Focus

- 입력 데이터  $x$ 를 출력 데이터  $y$ 의 형태로 변환시킴
- $x(b, c, w, h) \rightarrow y(b, 4c, w/2, h/2)$ 
  - $b$ : batch\_size,  $c$ : channel,  $w$ : width,  $h$ : height

### • Conv

- 일반적인 Conv + Batch\_Norm 레이어
- Conv 연산을 수행한 후 Batch Normalization 과정을 거침
- 활성화 함수로는 Hard Swish 함수를 사용

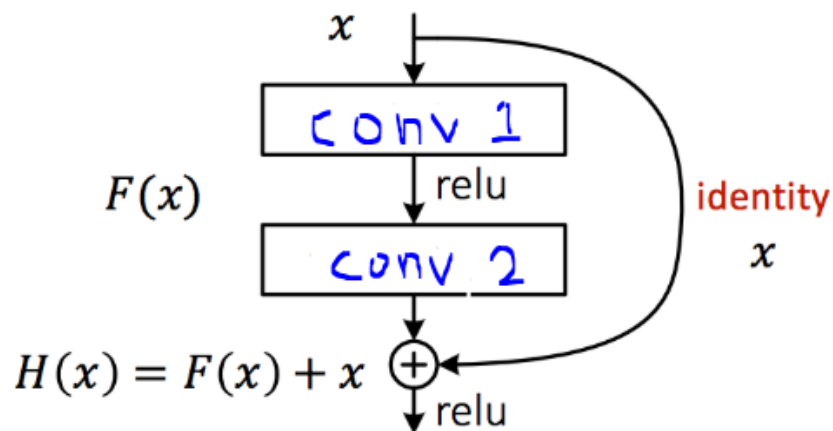
#### 분석 초점

common.py 파일을 기반으로 하여  
각 모듈 클래스의 forward 함수를 중심으로 분석함

- YOLO v5의 모듈

- Bottleneck

- ResNet에서도 사용된 Shortcut Connection이 적용된 블록





## • YOLO v5의 모듈

### • BottleneckCSP

- YOLO v5의 핵심

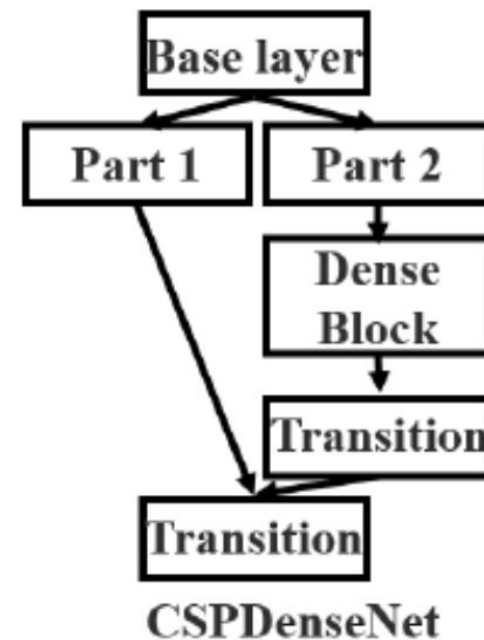
1. 4개의 Conv 레이어가 생성됨

- conv1, conv4: conv + batch\_norm 레이어
- conv2, conv3: conv 레이어 (batch\_norm 미적용)

2. CSP 구조에 따라 2개의 y 값을 생성함

- y1: Short-Connection으로 연결된 conv1 → conv3 연산 값
- y2 : 단순히 conv2를 연산한 값

3. 마지막으로 y1과 y2 값을 합치고, conv4 레이어를 통과하여 연산 수행



## • YOLO v5의 모듈

### • SPP

- YOLO v3-SPP에서 사용했던 Spatial Pyramid Pooling Layer
- Spatial bins로 5x5, 9x9, 13x13 크기의 Feature Map을 사용
  - 최종적으로  $5 + 9 + 13 = 27$ 의 크기로 고정된 1차원 형태의 배열을 생성
  - Fully Connected Layer에 입력으로 들어가도록 함

### • Concat

- 2개의 conv 레이어 연산 값을 결합

### • `torch.nn.modules.upsampling.Upsample`

- 단순히 업샘플링하는 토치의 기본 라이브러리 함수
- 구조 값을 따르면 Feature Map 의 각 배열의 갯수를 2배로 올려줍니다.

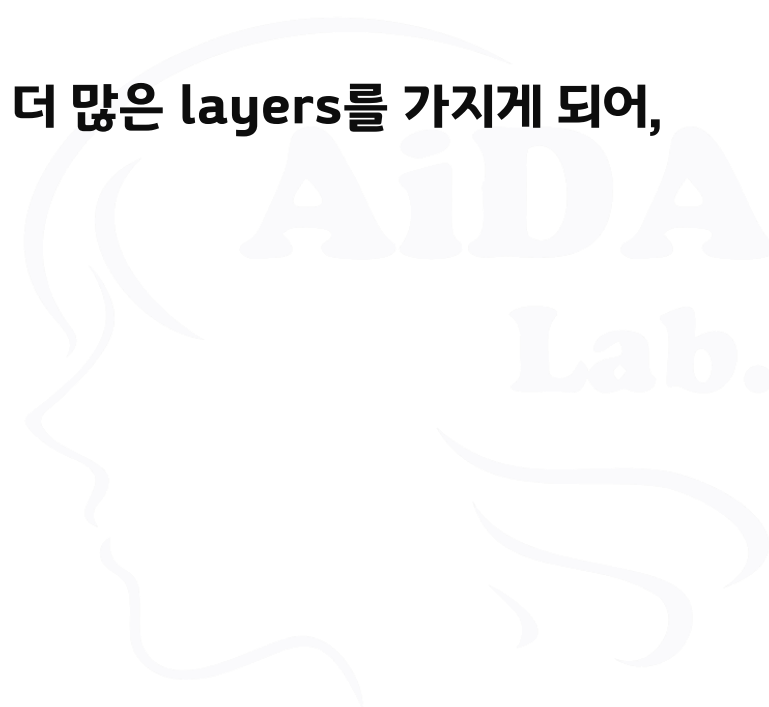
- s(small), m(medium), l(large), x(extra?)의 4가지가 있음
  - s가 가장 작고 빠르며, x가 가장 크고 느림
  - yaml 파일에 있는 "depth\_multiple" (model depth multiple)과 "width\_multiple" (layer channel multiple)의 두 가지 변수로 결정됨
  - YOLO v5-s의 depth&width\_multiple이 가장 작고(depth\_multiple : 0.33, width\_multiple : 0.50)
  - YOLO v5-x의 depth&width\_multiple이 가장 큼 (1.33, 1.25)

## • Depth\_Multiple

설명은 yolo.py 코드를 기반으로 진행

- 모델의 구조는 depth\_multi 값에 따라 변화함
- depth\_multiple 값이 클수록 BottleneckCSP 모듈(레이어)이 더 많이 반복되어, 더 깊은 모델이 됨
- yaml 파일에서 읽어온 depth\_multiple 값  $\rightarrow$  gd 변수에 저장, number 값  $\rightarrow$  n 변수
- $n(\text{depth gain}) = \max(\text{round}(n * gd), 1)$  if  $n > 1$  else  $n$   
 $\rightarrow$  만약  $n$ 이 1보다 크면  $n * gd$ 의 값을 반올림(소수점 둘째 자리)한 후, 1과 비교하여 큰 값을 선택. 그렇지 않으면 그냥  $n$ 을 사용
- Focus, Conv, SPP 모듈은 number 값이 1  $\rightarrow n * gd = 0.33 \rightarrow$  반올림해서 0.3  $\rightarrow$  max에 의해서 1
- BottleneckCSP 모듈은 number 값이 3, 9
  - $n(\text{number}) = 3$ 일 때:  $n * gd = 0.99 \rightarrow$  반올림해서 1  $\rightarrow$  Focus, Conv, SPP 의  $n(\text{depth gain}) = 1$
  - $n(\text{number}) = 9$ 일때:  $n * gd = 2.97 \rightarrow$  반올림해서 3  $\rightarrow$  BottleneckCSP의  $n(\text{depth gain}) = 3$

- 해당 모듈들은  $n(\text{depth gain})$ 값 만큼 반복 수행
  - $n(\text{number}) = 3$ 을 가지는 BottleneckCSP  $\rightarrow (0)$ : Bottleneck 하나만 반복  $\rightarrow 1$ 번
  - $n(\text{number}) = 9$ 를 가지는 BottleneckCSP  $\rightarrow (0)$ : Bottleneck  $\sim (2)$ : Bottleneck를 반복  $\rightarrow 3$ 번
- 따라서  $\text{depth\_multiple}$  값이 큰 YOLO v5-x는 s에 비해 당연히 더 많은 layers를 가지게 되어, 더 깊은 모델이 됨



## • Width\_Multiple

YOLO v5-s 기준으로 설명함

- yaml 파일의 첫번째 args 값과 width\_multiple 값을 곱한 값이 해당 모듈의 채널 값으로 사용됨  
→ Width\_Multiple 값이 증가할 수록 해당 레이어의 conv 필터 수가 증가함
- YOLO v5-s의 width\_multiple 값은 0.5 → 변수 gw에 저장
- yaml 파일의 args의 첫번째 값 → c2
- 해당 모듈의 채널 수 =  $c2 * gw$
- 따라서 YOLO v5-x는 s에 비해 큰 width\_multiple 값을 지니므로, 각 모듈의 채널 수가 가장 많음

- **yaml 파일에 따르면**
  - Head는 [from, number, module, args] 으로 구성
  - {Conv, Upsample, Concat, BottleneckCSP}이 한 블록으로 구성 → 이러한 블록들이 총 4개
  - 마지막의 Detect 부분에서 모두 연결됨
- **Head에서는**
  - BottleneckCSP 만이 number 값이 3으로, depth\_multiple 값에 따라 더 많이 반복 가능  
→ 즉 YOLO v5-x가 s에 비해 Head 층도 더 깊음
- **Head에서 주의 깊게 봐야할 모듈은 Concat과 Detect**

- Concat 모듈

- yaml 파일에서 Concat 모듈을 보면

```
[[ -1, 6], 1, Concat, [1]], # cat backbone P4
```

- 해당 블록을 가져오면

```
[[ -1, 1, Conv, [512, 1, 1]], # head p5 (N 10)  
[ -1, 1, nn.Upsample, [None, 2, 'nearest']],  
[[ -1, 6], 1, Concat, [1]], # cat backbone P4  
[ -1, 3, BottleneckCSP, [512, False]], # 13
```

- **[[ -1, 6], 1, Concat, [1]]**의 구성

- Concat의 의미: Concat의 바로 직전 층인 nn.Upsample 층과 **i=6**인 BottleneckCSP 층의 결합을 의미



- **Concat의 정리**

- 첫 번째 블록의 Concat 부분 : 백본의 P4와 결합 (yolo.py 기준 i=6인 BottleneckCSP)
- 두 번째 블록의 Concat 부분 : 백본의 P3와 결합 -> 작은 물체 검출 (yolo.py 기준 i=4인 BottleneckCSP)
- 세 번째 블록의 concat 부분 : 헤드의 P4와 결합 -> 중간 물체 검출 (yolo.py 기준 i=14인 Conv)
- 네 번째 블록의 concat 부분 : 헤드의 P5와 결합 -> 큰 물체 검출 (yolo.py 기준 i=10인 Conv)

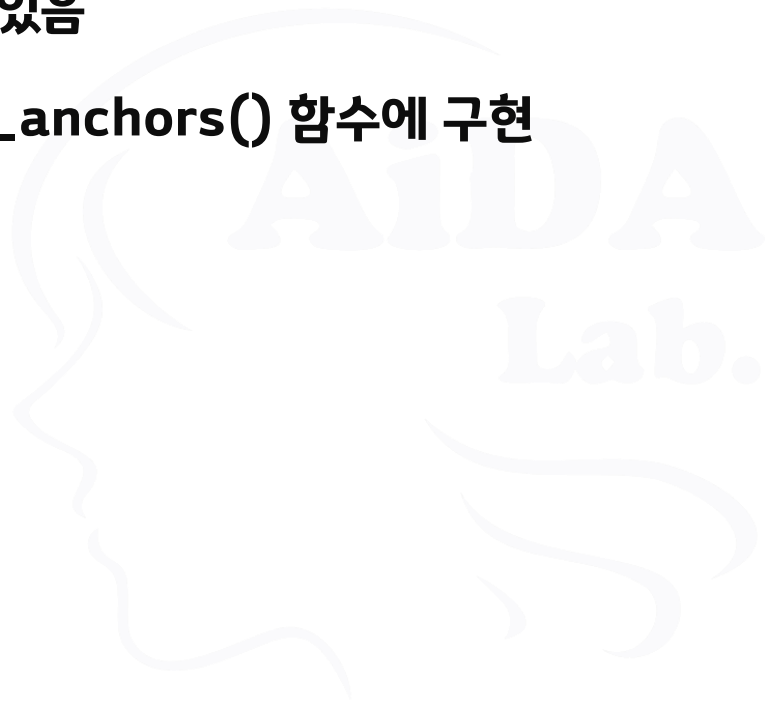
- **Detect 모듈**

```
[[17, 20, 23], 1, Detect, [nc, anchors]], # Detect(P3, P4, P5)
```

- i=17, 20, 23인 레이어를 종합하여 Detect

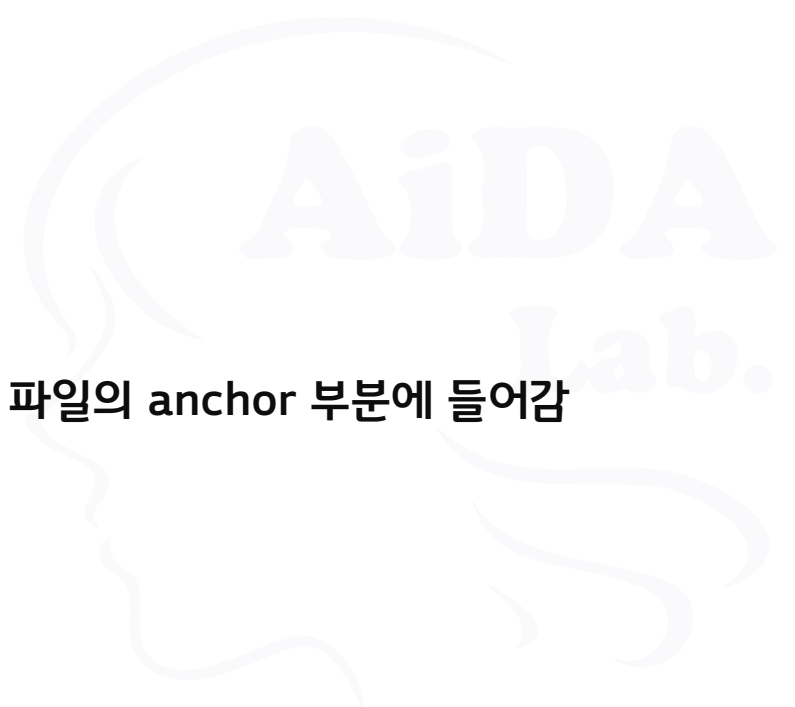
- 앵커 박스 값 계산하기

- YOLO v5에서 default로 사용하는 앵커 박스는 코코 데이터 기반의 값
  - 커스텀 데이터에서는 앵커 박스 값이 적절하지 않을 수 있음
  - 커스텀 데이터셋에 알맞는 앵커 박스 값을 계산해야 할 필요가 있음
- 앵커 박스의 값 계산은 `~/yolov5/utils/general.py`의 `kmean_anchors()` 함수에 구현



- **kmean\_anchors()** → 앵커 박스 계산의 핵심

- path : data yaml 파일 경로
- n : 생성할 앵커박스 갯수 (우리는 9개의 앵커박스를 생성)
- img\_size : 이미지 크기
- thr : 매개변수로 제어되는 Threshold(임계값)
- gen : 유전자 알고리즘의 진화적 반복 횟수(돌연변이+선택)
- verbose : 진행 내용을 어느 정도 출력할 것인지 결정하는 변수
- return k → 유전자 알고리즘 진화 후 K-평균 + 앵커 → model.yaml 파일의 anchor 부분에 들어감



- **LOSS 함수분석**

- **GIoU (giou\_loss)**

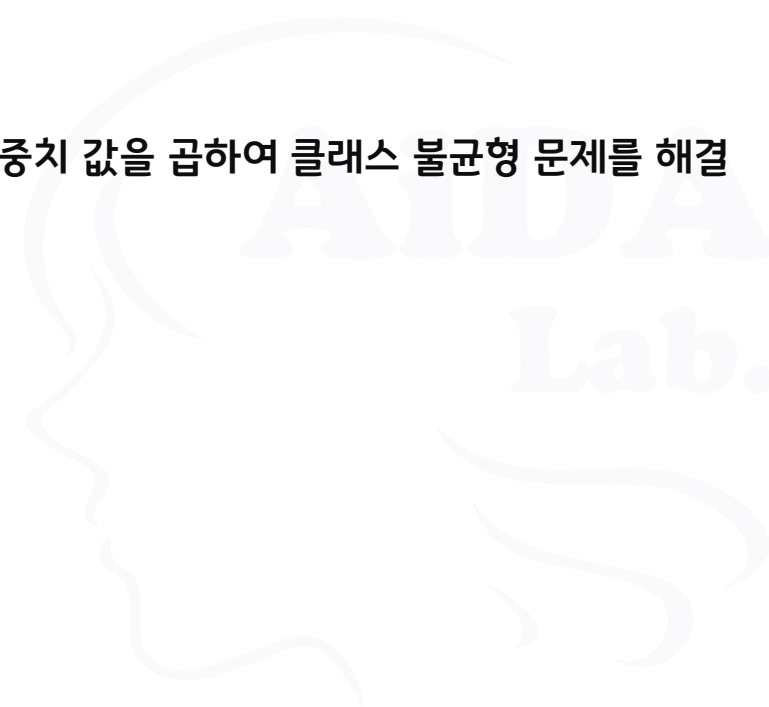
- bounding box에 관한 loss 함수.  $1 - \text{giou 값} = \text{giou loss}$
    - giou loss는 utils/general.py의 compute\_loss 함수에 구현

- **obj (objectness loss) 및 cls (classification loss)**

- BCEwithLogitsLoss 사용 (general.py에서 확인)
      - BCEwithLogitsLoss는 class가 2개인 경우에 사용하는 loss function
      - BCE(Binary Cross Entropy)에 sigmoid layer를 추가한 것



- classification loss
  - 객체가 탐지되었을 때, 탐지된 객체의 class가 맞는지에 대한 loss
  - MSE와 유사하게 (판단값 - 실제값)<sup>2</sup> 으로 계산
- objectness loss
  - class에 구분 없이 객체 탐지 자체에 대한 loss
  - 객체가 있을 경우의 loss, 없을 경우의 loss를 따로 계산 후, 각 loss에 가중치 값을 곱하여 클래스 불균형 문제를 해결



- **Optimizer 분석**

- optimizer의 default 값은 SGD
- 추가 설정으로 Adam으로 변경 가능

- **mAP 분석**

train.py 에서 구현

- **mAP\_0.5**
  - mAP의 평균을 IoU Threshold = 0.5로 구한 값
- **mAP\_0.5:0.95**
  - mAP의 평균을 다음의 IoU Threshold 값으로 구한 것
  - 즉 0.5~0.95 사이의 IOU threshold 값을 0.05 씩 값을 변경해서 측정한 mAP의 평균값
  - IoU의 threshold 값이 mAP\_0.5보다 높기 때문에, 수치는 mAP\_0.5보다 낮게 나옴

THANK  
YOU

