

# Object Detection

- 캡스톤 설계 및 실습 -



14조 모로

201900919 김영서

201901489 박서린

201903566 최서영

201903868 황가은

# 목차

- I. 서론
- II. 용어 정리
- III. One-stage Detection
  - 1. RetinaNet
  - 2. YOLO
  - 3. DETR
- IV. Two-stage Detection
  - 1. Cascade R-CNN
- V. 성능 비교
- VI. 결론

# I. 서론

항공기 활주로, 계류장, 공항 주변에는 지형 지물인 건물과 산, 지상 이동체인 항공기와 차량 등이 혼재한다. 이처럼 시각화된 3차원 공간을 정량적으로 평가하기 위해서는 시각 영상의 데이터화가 필요하다. 특히, 현재 항공 산업에 사용되는 object detection의 실제적인 사례가 없는 것으로 파악하였다.

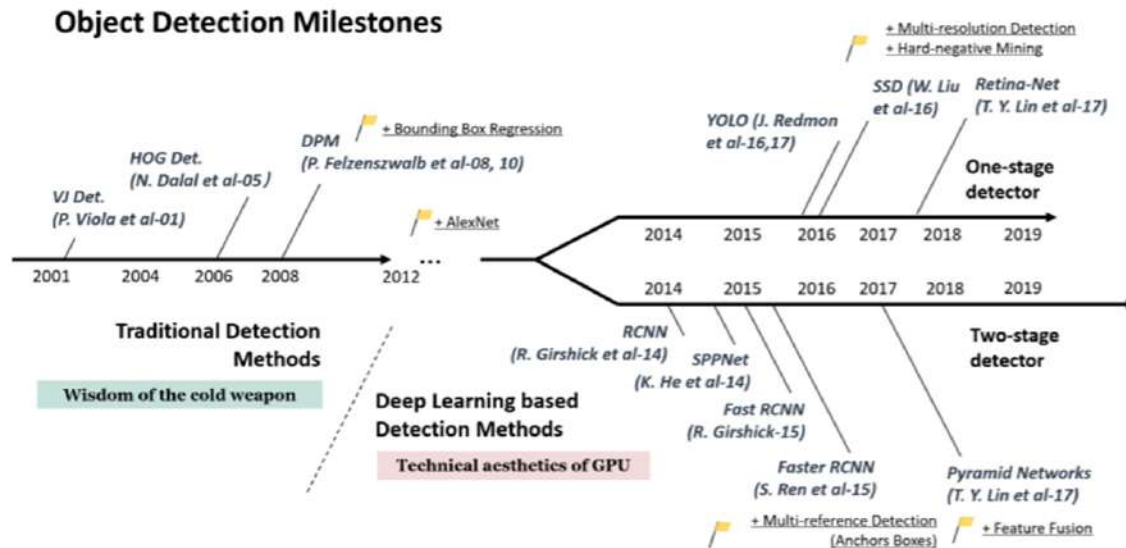
우선 일반적으로 시장에서 사용되는 open API를 조사하는 것으로 시작하였으나 이는 간편한 사용성과 이론으로 전문성이 부족하며 직접적으로 성능을 비교하기 위해 알아야 하는 API의 내부 구성을 알 수 없다는 한계점에 부딪혔다. 그리하여 이 한계점을 완화할 수 있도록 객체 탐지기의 open Source를 찾아보기로 결정하였다. 이후 조사한 오픈 소스 중 많이 등장하고 있는 model을 선택하여 이에 대해 심층적으로 조사하였고 각 모델의 등장 배경, 구조와 진행과정, 손실함수, 실행, 성능에 대해 정리하였다.[Reference 1~38 참고] 그 다음으로 모델 간의 성능 평가를 통해 계류장과 공항이라는 배경에서 목적에 맞는 최적의 모델을 선정하게 되었다.

## 객체 탐지기의 분류(One-Stage Detector, Two-Stage Detector)

객체 탐지기의 종류는 크게 One-Stage Detector와 Two-Stage Detector로 나뉜다. 이 둘 중 무엇이 더 발전된 모델인지를 나누는 것은 의미가 없다. 각자의 장단점이 있기 때문에 분야마다 선호하는 모델이 다르기 때문이다. 먼저 One-Stage Detector는 Regional proposal과 Classification이 동시에 이루어진다. 즉, Convolution Layer를 통해 Classification과 Bounding Box(경계 박스) 작업을 동시에 수행한다. 이에 속하는 모델은 YOLO계열, SSD계열(RetinaNet, SSD), DETR 등이 있다. Two-Stage Detector는 그와 반대로 순차적으로 이루어지는 구조를 가지고 있다. 이 구조에서는 Selective search 알고리즘을 통해 경계 박스를 생성하게 된다. 이에 속하는 모델은 R-CNN 계열이 있다.

이러한 구조의 차이로 One-Stage Detector는 비교적 검출 속도가 빠르지만 정확도가 낮고 그와 반대로 Two-Stage Detector는 검출 속도가 느리지만 정확도가 높다는 특징을 가지고 있다.

## 현재까지 객체 탐지 기술의 흐름



〈그림 1〉 객체 탐지 기술 흐름<sup>1</sup>

객체 탐지 분야는 지난 20년 동안 많은 발전을 이뤄왔다. 객체 탐지는 딥러닝의 도입 전후로 구분될 수 있다. 딥러닝 기반 검출기는 Anchor Free와 Anchor Based로 나뉘고, 다시 Anchor Based에서 One-Stage Detector와 Two-Stage Detector로 나뉜다. 2014년 이전의 Traditional Detection은 Sliding Window 방식을 사용한다. 이 방식은 좌측 상단에서부터 우측 하단으로 이동하면서 객체를 검출하는 방식이다. 다양한 형태의 Window를 각각 Sliding 시키는 방식이기 때문에 Window의 크기는 고정하고 크기를 변경한 여러 이미지를 사용하게 되는데 여기서 발생하는 문제는 객체가 없는 영역도 무조건 Sliding 하여야 하며 여러 형태의 Window와 여러 크기를 가진 이미지를 스캔해서 검출해야 하므로 수행 시간이 오래 걸리고 검출 성능이 상대적으로 낮다는데 있다.

2018년 이후의 객체 탐지기는 Anchor Based가 주를 이뤄 One-Stage 방식인지, Two-Stage 방식인지 구분한다.

<sup>1</sup> [Object Detection in 20 Years: A Survey, Zhengxia Zou, Zhenwei Shi, Member, IEEE, Yuhong Guo and Jieping Ye, Senior Member, IEEE]

## II. 용어 정리

### 1. 모델

조사한 모델은 다음과 같다. 괄호는 해당 모델의 조사를 맡은 팀원이다. Retinanet(박서린), Yolo v3(김영서), DETR(황가은), Faster R-CNN과 Cascade R-CNN(최서영) 모델 외에 일부 모델의 이전 버전이나 발전 배경이 된 몇 모델에 대한 설명도 포함한다.

#### a) R-CNN

R-CNN(Regions with CNN features)은 설정한 Region을 CNN의 input feature로 활용하여 object detection을 수행하는 신경망이다. R-CNN의 기본적인 구조는 물체의 위치를 찾는 region proposal과 물체를 분류하는 region classification으로 구성된다.

이미지에 있는 데이터와 레이블을 투입하여 물체의 영역을 찾는 region proposal 기법을 사용하여 bounding box를 생성한다. 생성된 bounding box를 CNN의 input으로 사용하여 classification을 수행한다. 분류한 뒤 bounding box를 조정하고 중복된 검출을 제거하며, 객체에 따라 box의 점수를 재산정하는 후처리 과정을 거친다.

#### b) Fast R-CNN

R-CNN의 느린 속도를 개선하기 위해 feature map을 공유한다. 기본적으로 GPU를 사용하지만, 영역 추정은 CPU에서 수행되기 때문에 영역 추정 단계에서 병목 현상이 발생한다.

### 2. 데이터셋

#### - COCO

객체 탐지에 쓰이는 데이터셋은 다양하다. 자주 쓰이는 데이터셋 중 하나인 COCO 데이터셋은 JPEG 이미지이며 JSON 포맷을 가지고 있다. COCO 데이터셋의 장점으로는 다양한 크기의 물체가 존재하며, 객체들이 혼잡하게 존재한다.

### 3. Backbone

input image를 feature map으로 변형하는 부분으로 feature extractor라고도 불린다. 이미지를 input으로 사용하여 이미지의 특징, feature map을 output으로 내보내는 모델이다.

## a) ResNet

총 152개의 레이어를 가진 매우 deep한 네트워크이다. Residual Learning을 통해 degradation problem<sup>2</sup>을 해결하는 방법 제시한다. residual learning을 통해 레이어를 깊게 쌓을 수 있도록 만들었다. 구현이 간단하며 학습 난이도가 매우 낮아지며 깊이가 깊어질수록 높은 정확도 향상을 보인다.

### - Residual Learning

Convolutional layer의 input에 해당하는 x값을 convolutional layer의 output에 다시 더해줌으로써, convolutional layer가 input 값과 output 사이의 차이(잔차, residual)를 학습하는데만 집중할 수 있도록 하는 기법이다. Residual learning을 통해 optimize가 더 쉬워지고 accuracy가 더 높아진다는 장점이 존재한다.

## b) CNN

CNN, Convolutional Neural Network는 인간의 시신경을 모방하여 만든 딥러닝 구조 중 하나이다. convolution 연산을 이용하여 이미지의 공간적인 정보를 유지하고, Fully Connected Neural Network 대비 연산 양을 획기적으로 줄여 이미지 분류에 좋은 성능을 보인다.

Fully Connected Neural Network 인공 신경망을 통해 3차원 데이터인 사진을 학습시키는 경우 1차원으로 평면화함으로써 공간 정보가 손실된다. 이처럼 특징을 추출하고 학습하는데 비효율적이며 정확도를 높이는데 한계를 CNN은 이러한 한계점을 극복하여 이미지의 공간 정보를 유지한 상태로 학습을 가능하게 한 모델이다. CNN 모델에는 Convolution Layer(합성곱 계층)와 Pooling Layer(풀링 계층)가 추가되며 마지막 부분에 이미지 분류를 위한 Fully Connected Layer가 더해진다.

### - Convolutional Layer(합성곱 계층)

특징 추출 신경망의 요소 중 하나로 입력 이미지에서 고유한 특징을 부각시킨 이미지를 새로 만들어내는 역할을 한다. 이렇게 생성된 이미지를 feature map이라 부른다. convolutional layer는 filter<sup>3</sup>를 이용해 이미지를 다른 이미지로 변환하고 입력 이미지를 처리하면 feature map을 얻는다. 이때 feature map의 수는 filter의 개수와 같다.

### - Pooling Layer

convolutional layer의 출력값을 입력으로 받아 출력 데이터인 feature map의 크기를 줄이거나 특정 데이터를 강조하는 용도로 사용된다. 원본 데이터의 차원을 줄이는 역할로

---

<sup>2</sup> 정확도가 어느 순간부터 정체되고 레이어가 더 깊어질수록 성능이 나빠지는 현상

<sup>3</sup> filter: 이미지의 특징을 찾아내기 위한 공용 파라미터로 kernel으로 불린다. 일반적으로 필터는 정사각 행렬로 정의된다.

이미지에서 특정 영역에 있는 픽셀들을 묶어 하나의 대푯값으로 축소한다. Pooling Layer를 처리하는 방법으로 Max Pooling, Average Pooling, Min Pooling이 있으며 CNN에서는 주로 Max Pooling<sup>4</sup> 기법을 사용한다

## 4. Neck

Backbone과 Head를 연결하는 부분으로 feature map을 refinement 정제, reconfiguration 재구성한다. 즉, Backbone에서 나온 feature map을 더욱 유의미하게 만들어주는 작업을 수행한다.

### a) FPN

FPN(Feature Pyramid Network)은 이미지를 컨볼루션 네트워크에 입력하여 다양한 크기의 특징 맵을 출력하는 네트워크이다. FPN은 새롭게 설계된 모델이 아니라 기존 모델에서 지정한 레이어 별로 특징 맵을 추출하여 수정하는 네트워크이다.

### b) FFN

FFN(Feed Forward Network)은 입력 층으로 데이터가 입력되고, 1개 이상으로 구성되는 은닉층을 거쳐서 마지막에 있는 출력층으로 출력 값을 내보내는 과정을 말한다.

## 5. Head

Backbone에서 추출한 feature map의 location 작업이 이루어지는 부분이다. 이미지의 bounding box를 구하고 classification(predict classes)을 수행한다.

Head는 크게 Dense Prediction과 Sparse Prediction으로 구분된다. Dense Prediction은 predict classes & bounding boxes 작업이 함께, Sparse Prediction은 각각 수행된다. Head가 Dense Prediction인 architecture는 One-Stage Detector가, Sparse Prediction인 경우 Two-Stage Detector가 된다.

## 6. 성능 지표

### a) AP (Average Precision)

Object Detection의 성능을 평가하는 방법 중 하나로 물체 인식 분야의 알고리즘 성능은 보통 AP로 평가한다. 우선 Precision-Recall 곡선(PR 곡선)으로 recall(재현율, 마땅히 검출해내야 하는

---

<sup>4</sup> Max Pooling: 정해진 픽셀 범위 내에서 최대값을 뽑아내는 방법

물체들 중에서 제대로 검출된 것의 비율) 값의 변화에 따른 precision(정밀도, 모든 검출 결과 중 제대로 검출된 비율) 값을 나타내고, 이 곡선의 아래쪽 면적을 계산한 것이 Average Precision(AP)이다. 보통 계산하기 전에 PR 곡선이 단조적으로 감소하는 그래프가 되도록 바뀌준 후 계산한다.

## b) mAP (mean Average Precision)

mAP가 높을수록 정확하고 작을수록 부정확하다.

1. 모델이 예측한 모든 바운딩 박스를 가져온다.
2. 바운딩 박스들을 내림차순으로 정렬한다.
3. 모든 결과값에 대해 precision과 recall을 계산한다.
4. 위에서 계산한 값으로 PR곡선을 그린다.
5. PR곡선의 아래 영역을 계산한다(=AP)
6. 모든 클래스의 PR 곡선 아래 영역 계산
7. 6번의 값을 모두 합한 다음 물체 클래스의 개수로 나눠준다.

# 7. 기타

## a) Anchor Box

입력 영상에 대해서 객체가 있을 법한 곳에 설정한 박스로, 특정 영역을 포괄하는 객체가 있는지 네트워크 학습을 통해 판단한다. 입력 영상에 anchor box가 생성되는 위치는 샘플링된 feature map으로 결정된다.

## b) Bounding Box (BB)

Detection에서는 박스 형태로 위치를 표시한다. 이때 사용하는 박스는 네 변이 이미지 상에서 수직/수평 방향(axis-aligned)을 향한 직사각형 모양의 박스이고, 이를 Bounding Box라고 한다.  
→ Bounding Box가 네트워크가 predict한 object의 위치가 표현된 네트워크의 출력이라면, Anchor Box는 네트워크가 detect해야 할 object의 shape에 대한 가정인 네트워크의 입력이다.

## c) End-to-End

End-to-End란 학습 알고리즘에 직접 입력을 넣고 그에 대한 출력 또한 직접 얻는 것을 말한다. 학습 알고리즘 입력의 끝과 출력의 끝에 직접적으로 연결되어 있는 것이다.

‘입력 -> 학습 알고리즘 -> 출력’



#### d) IoU

Intersection over Union의 약자로 object detector의 정확도를 측정하는데 이용되는 평가 지표이다. IoU는 Ground Truth와 Prediction Box의 교집합 ÷ 합집합으로 계산한다. 이때 ground-truth는 testing set에서 object 위치를 라벨링 한 것이며 predicted bounding box는 모델이 예측한 box를 말한다.

IoU는 모델이 예측한 바운딩 박스가 ground-truth bounding box와 겹치는 부분이 많으면 많을수록 rewards를 준다. IoU를 사용함으로써 x,y좌표가 정확히 일치하는지를 보는게 아니라 예측한 바운딩 박스가 정답과 최대한 가까워지도록 학습하는 것을 의미한다.

#### e) NMS

Non-Max Suppression의 약자로, object detector 가 예측한 여러 개의 bounding box 중 정확한 box를 선택하도록 하는 기법이다. 객체 검출 알고리즘은 object가 존재하는 위치 주변에 여러 개의 bounding box를 만든다는 문제가 발생한다. 이 문제를 해결하기 위해 하나의 bounding box를 선택하는데 적용되는 기법이다. NMS의 간단한 과정은 다음과 같다.

1. 각 bounding box가 갖는 confidence score에 대해 threshold 이하의 값을 갖는 bounding box를 제거한다.
2. 남은 bounding box들을 confidence score 기준으로 내림차순 정렬을 한 후, 가장 앞에 있는 box를 기준으로 다른 box와의 IoU 값을 구한다. IoU 값이 threshold 이상인 box들을 제거한다. (bounding box 사이의 IoU가 높을수록 동일한 object를 검출하고 있다고 판단한다.)
3. 앞의 과정을 모든 bounding box에 반복적으로 적용하여 비교, 제거한다.

#### f) RoI

Region of Interest의 약자로, 영상이나 이미지 내에서 관심 있는 영역을 의미한다.

- RoI Pooling

RoI 영역에 해당되는 부분만 Max Pooling을 통해 feature map으로부터 고정된 길이의 저차원 벡터로 축소한다. RoI Pooling을 사용하면 입력 이미지 크기와 상관없이 고정된 feature map을 얻을 수 있다.

## g) RPN

Region Proposal Networks의 약자로, 영역 추정 네트워크를 말한다. 원본 이미지에서 region proposals를 추출하는데, anchor box를 생성하면 수많은 region proposals가 만들어진다. 이때 RPN은 region proposals에 대하여 class score를 매기고, bounding box coefficient를 출력하는 기능을 한다.

### - 동작 과정

- 1) 원본 이미지를 pre-trained된 VGG 모델에 입력하여 feature map을 얻는다.
- 2) 위에서 얻은 feature map에 대하여 3x3 conv 연산을 적용한다. 이때 feature map의 크기가 유지될 수 있도록 padding을 추가한다.
- 3) class score를 매기기 위해 feature map에 대해 1x1 conv 연산을 적용한다. 이때 출력하는 feature map의 channel 수가 2x9가 되도록 설정한다.
- 4) bounding box regressor를 얻기 위해 feature map에 대해 1x1 conv 연산을 적용한다. 이때 출력하는 feature map의 channel 수가 4(bounding box regressor)x9(anchor box 9개)가 되도록 설정한다.

## h) Selective Search

R-CNN 계열의 모델에서 이미지 후보 영역을 추천할 때 사용하는 알고리즘이다. segmentation(이미지에서 pixel 단위로 객체 추출)과 Exhaustive search(모든 객체의 위치를 찾아냄) 두가지 방법을 결합하여 후보영역을 추천한다. Object Detection 분야에서 R-CNN, Fast R-CNN 등의 성능을 개선시키는 방법으로 활용되었으나 end-to-end 방식으로 학습하기 어렵고 실시간 탐지가 불가능해 Faster R-CNN부터는 사용되지 않거나 변형된 형태로 사용된다.

## i) 손실 함수

손실 함수는 라벨값과 예측값의 차이를 손실(Loss)라고 말하며 이들을 계산하기 위한 함수를 의미한다. 그리고 학습은 해당 손실을 최소화하는 방향으로 가중치를 업데이트하며 진행된다. 이런 방향으로 진행되기 때문에 손실 함수를 제대로 정하는 것은 매우 중요한 일이다. 같은 네트워크를 사용한다고 하더라도 손실 함수가 바뀌면 가중치는 다른 값으로 업데이트가 되고 최종적으로 신경망의 성능이 달라지게 된다.

# III. One-stage Detection

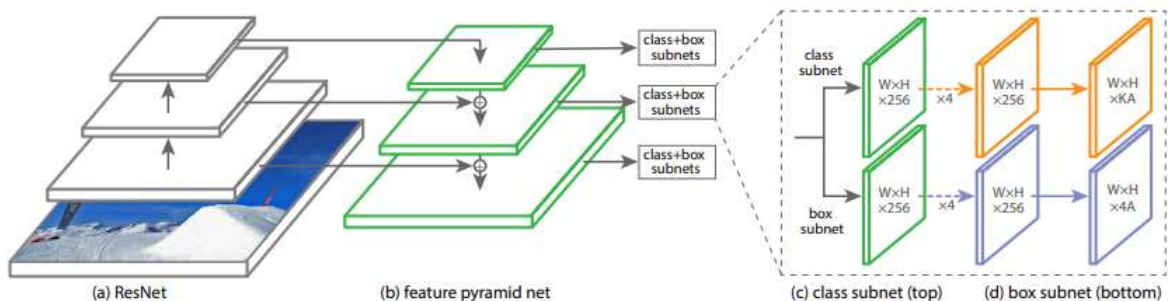
## 1. RetinaNet

### a) 등장 배경

One-stage detector는 Two-stage detector보다 속도는 빠르지만 성능은 떨어진다는 단점이 존재한다. Retinanet 저자는 one-stage detector의 낮은 정확도의 원인은 객체와 배경 클래스 불균형이 원인이라는 것을 발견하였다. Focal Loss는 one-stage detector에서도 클래스 불균형 문제를 해결할 수 있도록 한다. Loss function을 수정하여 예측하기 쉬운 example에는 0에 가까운 loss를 부여하고 예측하기 어려운 negative example에는 기존보다 높은 loss를 부여한다. Focal Loss의 효과를 실험하기 위해 one-stage detector인 RetinaNet을 설계했다.

### b) 구조 및 진행 과정

하나의 backbone network와 classification과 bounding box regression을 수행하는 2개의 subnetwork로 구성한다. Backbone network는 convolutional feature map을 추출한다. 첫번째 subnet은 object classification을 수행한다. 두번째 subnet은 bounding box regression을 수행한다. RetinaNet은 모델이 예측하기 어려운 hard example에 집중하도록 하는 Focal Loss를 제안한다. ResNet과 FPN을 활용하여 구축된 one-stage 모델인 RetinaNet은 focal loss를 사용하여 two stage 모델 Faster R-CNN의 정확도를 능가한다. 이를 통해 One-stage detector의 빠른 detection 시간의 장점을 가지면서 One-stage detector의 detection 성능 저하 문제 개선하였다.



〈그림 2〉 RetinaNet Architecture<sup>5</sup>

<sup>5</sup>T. -Y. Lin, P. Goyal, R. Girshick, K. He and P. Dollár "Focal Loss for Dense Object Detection"

Feedforward(순방향 신경망)<sup>6</sup>로 ResNet을 사용. <그림 2>를 보면 ResNet 상단에서 FPN backbone을 사용했고 multi-scale convolutional feature pyramid를 생성 -> Anchor box를 생성

## ● FPN

RetinaNet은 P3에서 P7의 pyramid level들을 사용한다. 모든 pyramid level에서 채널의 개수  $C=256$ 이다.

## ● Anchor

각 pyramid level에 총 9개의 anchor를 할당한다. 각 anchor는 one-hot K vector(K개의 class 중 해당하는 class는 1, 나머지는 0)와 바운딩 박스 offset 4개를 할당한다. 따라서 하나의 anchor에는  $K \times 4$ 의 vector가 할당된다.

## ● Classification Subnet

각 위치에서 A개의 anchor들의 K개의 object class들의 존재 확률을 예측한다. Subnet의 parameter들은 모든 pyramid level들에서 공유된다. 각 pyramid level에  $KA$ 개의 filter를 지닌  $3 \times 3$  conv layer가 4개로 구성된 conv layer를 부착한다.( $K$ =class 수,  $A$ =anchor 수)  
Classification subnet의 출력값에 Focal Loss를 적용한다.

## ● Box Subnet

Anchor와 ground-truth의 offset을 계산하는 network이다. Classification subnet과 동일하지만 마지막에  $4A$  길이를 출력한다. 각 anchor마다 offset 4개의 값을 출력하는 것이다.. 2개로 나뉜 subnet을 통해 classification과 box regression은 유사한 구조를 가지지만 다른 parameters로 학습된다.

---

<sup>6</sup> FeedForward(순방향 신경망) : 입력 층(Input Layer)으로 데이터가 입력되고 1개 이상으로 구성되는 은닉 층(hidden layer)을 거쳐서 마지막에 있는 출력 층(output layer)으로 출력 값을 내보내는 과정(feedback을 가지지 않거나 loop 연결을 가지지 않는 network)

### c) 손실 함수

Retinanet에서는 손실 함수 중에서 Cross entropy loss, Focal loss를 다룬다.

#### - Cross entropy Loss

$$CE(p, y) = \begin{cases} -\log(p) & \text{if } y = 1 \\ -\log(1 - p) & \text{otherwise.} \end{cases} \quad 7$$

위 식은 이진 분류 문제에서 사용하는 CE(Cross Entropy) loss function으로  $y$ 는 ground-truth class이고  $p$ 는 모델이 예측한 값이다.

$$p_t = \begin{cases} p & \text{if } y = 1 \\ 1 - p & \text{otherwise,} \end{cases}$$

and rewrite  $CE(p, y) = CE(p_t) = -\log(p_t)$ . <sup>8</sup>

$p_t$ =해당 class가 존재할 확률로  $p_t$ 가 0.5이상이면 분류하기 쉬운 easy example이다. easy example은 loss값이 작지만 엄청나게 많아지면 대부분의 loss를 차지하게 되어 hard example의 영향을 감소시킨다. CE loss는 클래스 불균형이 존재할 때 좋은 선택이 아니다.

#### - Focal Loss

Focal Loss는 cross entropy loss에 인자를 하나 추가한 것. 추가한 인자는 modulating factor라고 하고 easy example의 영향을 감소시키고 hard example에 집중하도록 함. 앞에 인자를 추가함으로써 easy example이 loss에 미치는 영향을 감소시킬 수 있다.

$$FL(p_t) = -(1 - p_t)^\gamma \log(p_t). \quad 9$$

$\gamma$ 는 하이퍼파라미터. 감마값으로 easy/hard example의 가중치를 조절한다.

example이 잘못 분류되었으면  $p_t$ 는 낮은 값을 가짐.  $p_t$ 가 낮으면 modulating factor는 1에 가까운 값을 갖게 된다. 따라서 loss는 가중치에 영향을 받지 않는다. example이 예측하기 쉽다면  $p_t$ 는 큰 값을 갖게 될 것이다.  $\gamma=2$ ,  $p_t=0.9$ 인 경우에 CE loss보다 100배 낮은 loss값을 가진다.  $p_t$ 가 0.5이하면 CE loss보다 4배

---

<sup>7</sup>T. -Y. Lin, P. Goyal, R. Girshick, K. He and P. Dollár "Focal Loss for Dense Object Detection"

<sup>8</sup> T. -Y. Lin, P. Goyal, R. Girshick, K. He and P. Dollár "Focal Loss for Dense Object Detection"

<sup>9</sup>T. -Y. Lin, P. Goyal, R. Girshick, K. He and P. Dollár "Focal Loss for Dense Object Detection"

더 큰 loss를 가진다. 이처럼 pt에 따라 loss 값을 조절하여 easy example의 영향을 낮추고 hard example의 영향을 높이는 것이다.

#### d) 성능

|                            | backbone                 | AP          | AP <sub>50</sub> | AP <sub>75</sub> | AP <sub>S</sub> | AP <sub>M</sub> | AP <sub>L</sub> |
|----------------------------|--------------------------|-------------|------------------|------------------|-----------------|-----------------|-----------------|
| <i>Two-stage methods</i>   |                          |             |                  |                  |                 |                 |                 |
| Faster R-CNN+++ [16]       | ResNet-101-C4            | 34.9        | 55.7             | 37.4             | 15.6            | 38.7            | 50.9            |
| Faster R-CNN w FPN [20]    | ResNet-101-FPN           | 36.2        | 59.1             | 39.0             | 18.2            | 39.0            | 48.2            |
| Faster R-CNN by G-RMI [17] | Inception-ResNet-v2 [34] | 34.7        | 55.5             | 36.7             | 13.5            | 38.1            | 52.0            |
| Faster R-CNN w TDM [32]    | Inception-ResNet-v2-TDM  | 36.8        | 57.7             | 39.2             | 16.2            | 39.8            | <b>52.1</b>     |
| <i>One-stage methods</i>   |                          |             |                  |                  |                 |                 |                 |
| YOLOv2 [27]                | DarkNet-19 [27]          | 21.6        | 44.0             | 19.2             | 5.0             | 22.4            | 35.5            |
| SSD513 [22, 9]             | ResNet-101-SSD           | 31.2        | 50.4             | 33.3             | 10.2            | 34.5            | 49.8            |
| DSSD513 [9]                | ResNet-101-DSSD          | 33.2        | 53.3             | 35.2             | 13.0            | 35.4            | 51.1            |
| <b>RetinaNet (ours)</b>    | ResNet-101-FPN           | 39.1        | 59.1             | 42.3             | 21.8            | 42.7            | 50.2            |
| <b>RetinaNet (ours)</b>    | ResNeXt-101-FPN          | <b>40.8</b> | <b>61.1</b>      | <b>44.1</b>      | <b>24.1</b>     | <b>44.2</b>     | 51.2            |

〈그림 3〉<sup>10</sup>

RetinaNet의 성능은 기존 Two-Stage Detector, One-Stage Detector 모델들과 비교해보았을 때 성능이 우수하다.

## 2. YOLO v3

#### a) 등장 배경

기존 Object Detection 분야에서 딥러닝 모델을 이용하여 객체 탐지를 수행하는 모델이 대표적으로 R-CNN(2014)이 존재한다. YOLO(2016)는 기존 모델들과 근접한 정확도를 가지며 더 많은 양의 이미지를 처리할 수 있는 실시간 객체 탐지를 위해 등장하였다.

R-CNN은 이미지에서 일정한 규칙으로 이미지를 여러 장으로 쪼개 CNN 모델을 통과시킨다. 한 장의 이미지에서 객체 탐지를 수행해도 실제로는 수 천장의 이미지가 모델에 통과되는 것이다. 반면에 YOLO는 이미지 전체를 단 하나로 본다. 이미지의 픽셀로부터 bounding box의 위치, class probabilities를 구하기까지의 절차를 단 하나의 회귀 문제로 재정의하였다. 기존에 다양한 전처리 모델과 인공 신경망을 결합하여 사용하는 방법과 달리 하나의 인공신경망에서 처리한다.

Joseph Redmon의 YOLO는 2016년에 처음 발표되었고 버전 3(YOLO v3)까지 공개되었다. 이후 다른 저자에 의해 현재 Yolov5까지 개발되었다. 처음 등장한 YOLO v1은 정확도가 낮다는 문제가 있었고

<sup>10</sup> T. -Y. Lin, P. Goyal, R. Girshick, K. He and P. Dollár "Focal Loss for Dense Object Detection"

2017년 YOLO v2(Yolo9000)를 발표하였다. 9,000개의 이미지를 탐지하면서 분류할 수 있는 모델이며 Batch Normalization<sup>11</sup>, Direct Location Prediction<sup>12</sup>, Multi-Scale Training<sup>13</sup> 기법 등을 도입하여 FPS와 mAP를 높였다. 그러나 YOLO v2까지도 정확도 문제를 완벽히 해결하지 못하였고 이후 ResNet에 사용된 FPN 방식을 도입하여 정확도를 높인 YOLO v3 모델이 등장하였다.

## b) 구조 및 진행 과정

YOLO v3는 입력 이미지를  $S \times S$  그리드(grid)로 분할한 후 각 그리드 셀에 대해 CNN(Convolutional Neural Network)을 실행한다. 각 구간에 따라 bounding box와 class probability를 생성하여 해당 구역의 객체를 인식한다. 후보 영역을 추출하기 위해 별도의 네트워크를 적용하지 않기 때문에 R-CNN보다 처리 시간 측면에서 월등한 성능을 보인다.

YOLO v3의 전체적인 구조를 살펴보면 Backbone을 'Darknet 53'을 사용하여 3개의 feature map을 출력한다. Neck에서는 FPN을 통해 크기가 다른 3개의 feature map을 upsampling, concatenation 과정에 거쳐 유의미한 특징을 부여한다. 이후 이들을 Head 부분으로 전달하며 3개의 feature map에 각각 anchor box 기법을 적용하여 bounding box를 구한 후 classification을 수행한다.

### ● Darknet 53

YOLO v3에서는 'Darknet 53' 모델을 Backbone으로 사용하였다. Darknet 53은 23개의 residual units<sup>14</sup>를 포함하며 각 residual unit은  $3 \times 3$ ,  $1 \times 1$  convolutional layer를 한 개씩 포함한다. 각 residual unit의 끝에는 입력 벡터와 출력 벡터 사이의 element-wise 덧셈이 수행된다. downsampling<sup>15</sup> 단계는 stride 값을 2로 갖는 5개의 분리된 convolutional layers에서 수행된다. residual units 사이에 있는 convolutional layer size를 보면 모두 '2'로 나뉘지는 걸 볼 수 있다.

---

<sup>11</sup> Batch Normalization: 신경망 안에서 batch 단위로 각 feature들의 평균과 표준편차를 이용해 normalization하는 방법. internal covariate shift 현상을 해결하기 위해 사용하였다. -> 2% mAP 향상

<sup>12</sup> Direct Location Prediction: 기존 anchor box들의 loss에는 제한이 없어 중심 셀에서 떨어진 위치에 bounding box를 예측하는 일이 발생하였다. 이에 대한 제약사항을 추가하기 위해 sigmoid를 적용하였다. -> 5% 성능 향상

<sup>13</sup> Multi-Scale Training: Fully Convolutional Network 방식을 도입하여 다양한 해상도로 입력 이미지를 취하였으며 최종 feature map의 크기를 홀수가 되도록 하였다.

<sup>14</sup> residual unit(=block) has two  $3 \times 3$  convolutional layers with the same number of output channels (ResNet 네트워크 구조에서 사용되는 형태)

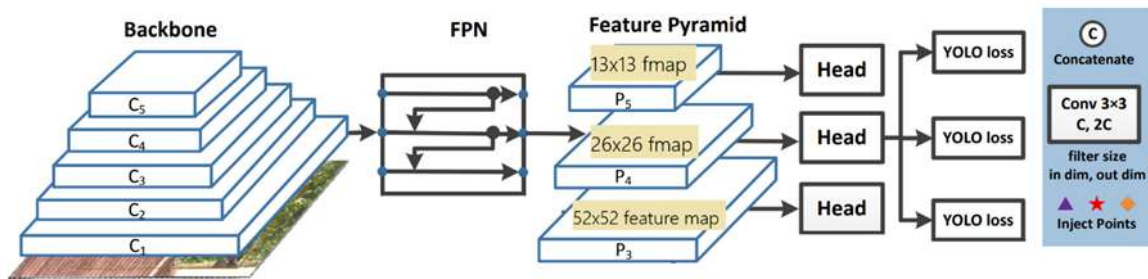
<sup>15</sup> downsampling: a reduction of the feature maps size, 이미지의 차원을 줄여 적은 메모리로 깊은 convolution을 할 수 있도록 한다. 이 과정을 통해 feature 정보(resolution 해상도)를 잃게 된다.

|    | Type          | Filters | Size      | Output    |
|----|---------------|---------|-----------|-----------|
|    | Convolutional | 32      | 3 × 3     | 256 × 256 |
|    | Convolutional | 64      | 3 × 3 / 2 | 128 × 128 |
| 1x | Convolutional | 32      | 1 × 1     | 128 × 128 |
|    | Convolutional | 64      | 3 × 3     |           |
|    | Residual      |         |           |           |
|    | Residual      |         |           |           |
| 2x | Convolutional | 128     | 3 × 3 / 2 | 64 × 64   |
|    | Convolutional | 64      | 1 × 1     | 64 × 64   |
|    | Convolutional | 128     | 3 × 3     |           |
|    | Residual      |         |           |           |
| 8x | Convolutional | 256     | 3 × 3 / 2 | 32 × 32   |
|    | Convolutional | 128     | 1 × 1     | 32 × 32   |
|    | Convolutional | 256     | 3 × 3     |           |
|    | Residual      |         |           |           |
| 8x | Convolutional | 512     | 3 × 3 / 2 | 16 × 16   |
|    | Convolutional | 256     | 1 × 1     | 16 × 16   |
|    | Convolutional | 512     | 3 × 3     |           |
|    | Residual      |         |           |           |
| 4x | Convolutional | 1024    | 3 × 3 / 2 | 8 × 8     |
|    | Convolutional | 512     | 1 × 1     | 8 × 8     |
|    | Convolutional | 1024    | 3 × 3     |           |
|    | Residual      |         |           |           |
|    | Avgpool       |         | Global    |           |
|    | Connected     |         | 1000      |           |
|    | Softmax       |         |           |           |

〈그림 4〉 Architecture of Darknet 53<sup>16</sup>

## ● FPN

YOLO v3는 FPN(Feature Pyramid Network) 구조를 통해 3개의 feature map에 upsampling<sup>17</sup>과 concatenation<sup>18</sup>과정을 거쳐 유의미한 특징을 부여한다. 보통 backbone에서 단계를 거쳐 최종 feature map을 출력하나 YOLO v3는 중간에 3개의 서로 다른 scale을 가진 feature map을 FCN을 이용하여 feature pyramid를 설계한다. 이후, 3개의 feature map을 Head 부분으로 전달한다.



〈그림 5〉 Backbone-Neck-Head architecture of YOLOv3<sup>19</sup>

## ● Output

bounding box를 구하고 classification을 하면 하단의 〈그림8〉과 같은 output이 나온다. tx,ty,tw,th는 각 bounding box의 좌표, Po는 Object일 확률, P1,P2,...,Pc는 클래스일 확률이다. 예를 들어, ‘강아지와

<sup>16</sup> Joseph Redmon, Ali Farhadi, “YOLOv3: An Incremental Improvement”, (2018.04)

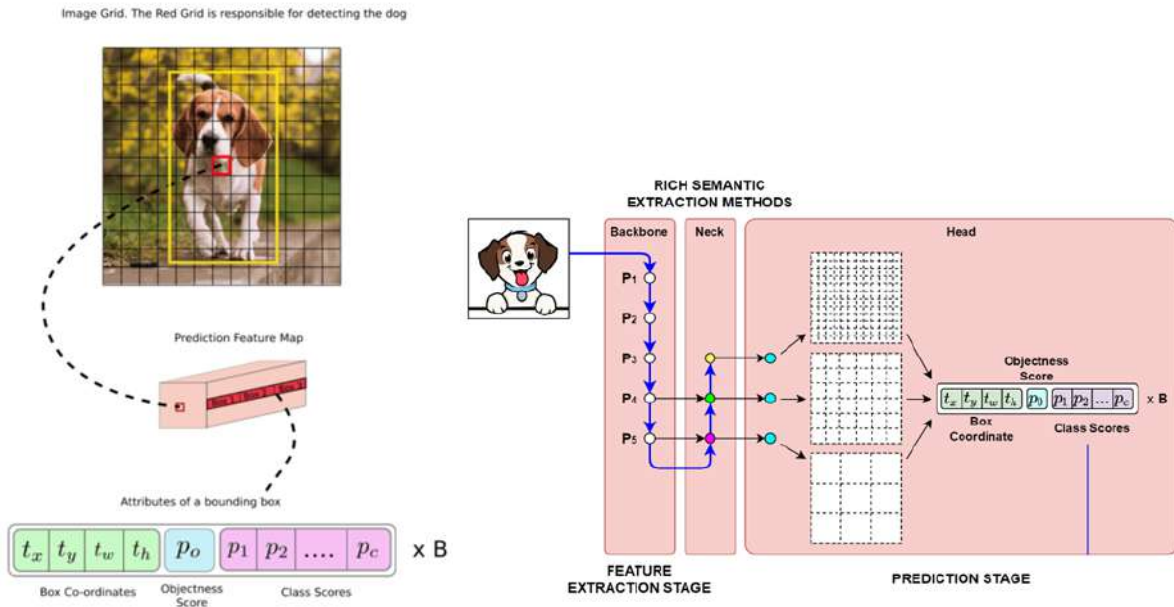
<sup>17</sup> upsampling: 축소된 feature map을 원본 이미지의 크기로 되돌리기 위해 늘려가는 방법

<sup>18</sup> concatenation: 여러 feature map을 결합시키는 연산, feature map의 크기가 다르면 불가능하다.

<sup>19</sup> 직접 편집한 이미지 (원본 출처: Xiang Long 외 10, “PP-YOLO: An Effective and Efficient Implementation of Object Detector”, (2020.08))



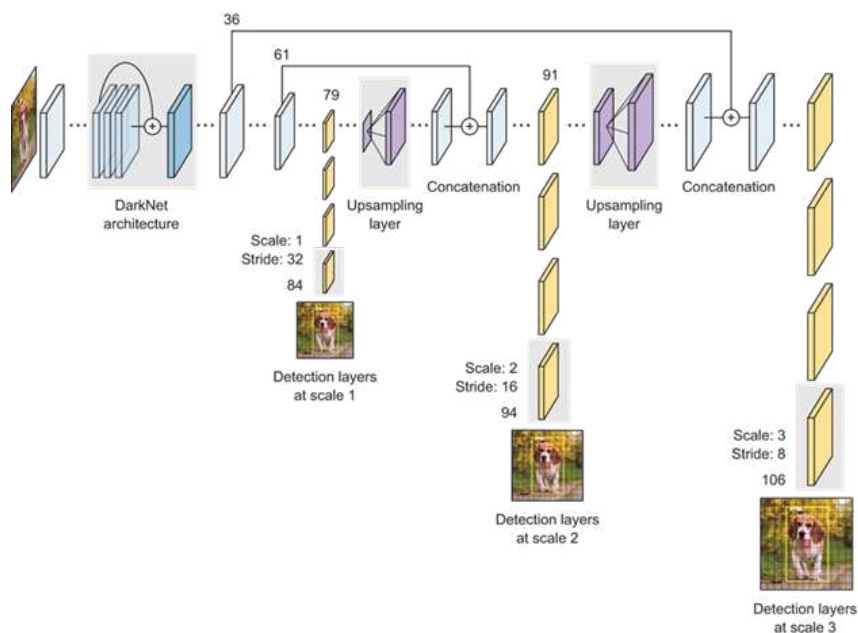
고양이의 분류' 문제라면 클래스 확률은 P1(강아지), P2(고양이)로 2가지이다. YOLO v3에서는 1 cell 당 3개의 anchor box를 가지며 각 3가지의 scale을 가지므로 1 cell 당  $3 \times 3 = 9$ 개의 anchor box를 갖는다. 따라서 B값이 9이다. 최종적으로 이미지를 input으로 넣어 Backbone, Neck, Head를 거쳐 output이 나오는 과정을 간단히 나타내면 하단의 <그림9>와 같다.



<그림 6> example of output from YOLOv3 <sup>20</sup>

<그림 7> 전체적인 YOLOv3 Architecture and Output <sup>21</sup>

## ● 진행 과정 및 예시



<sup>20</sup> Ayoosh Kathuria, "How to implement a YOLO (v3) object detector from scratch in PyTorch", (2018)

<https://blog.paperspace.com/how-to-implement-a-yolo-object-detector-in-pytorch/>

<sup>21</sup> Minsoo Go, Deep Learning Bible", <https://wikidocs.net/163583>, (2022.03)

- 1) 이미지를 입력하고 표준 크기로 이미지를 조정한다.
- 2) 입력 영상을 3개의 척도 13\*13, 26\*26, 52\*52 그리드로 나눈다. (객체의 중심점이 그리드 단위로 떨어지는 경우 그리드 단위는 객체를 예측한다.)
- 3) k-means clustering를 사용하여 각 그리드 단위에 대한 경계 상자를 결정한다. 각 grid unit에는 3 개의 클러스터가 있습니다. 3개의 척도가 있으므로 grid unit 당 총 9개의 클러스터가 있다.
- 4) feature extraction을 위해 이미지를 네트워크에 입력한다. 모델은 먼저 작은 규모의 13\*13의 feature map을 생성한다.
- 5) 13\*13 소규모 feature map은 먼저 convolutional set와 2배 upsampling을 거친 후, 26\*26 feature map에 연결하여 예측 결과를 출력한다.
- 6) (5)단계에서 출력된 26\*26 feature map은 convolution set와 2배의 upsampling을 거친 후, 52\*52 feature map에 연결하여 예측 결과를 출력한다.
- 7) 3가지 스케일 예측 출력의 특징들을 합한 후, probability score를 threshold로 사용하여 낮은 점수를 가진 대부분의 anchor를 필터링한다. 그런 다음 처리에 NMS(Non Maximum Suppression)를 사용하여 더 정확한 최상의 경계 상자를 남긴다.

### c) 손실 함수

앞서 YOLO v3 network를 통해 얻은 multi-scale feature maps를 loss function을 통해 학습시킨다.

---

<sup>22</sup> Robertson Davies, 『Deep Learning for Vision Systems』 7. Object Detection with R-CNN, SSD, and YOLO, (2020.10)

### Regression Loss

$$\lambda_{\text{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^B \mathbb{1}_{ij}^{\text{obj}} \left[ (x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2 \right]$$

$$+ \lambda_{\text{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^B \mathbb{1}_{ij}^{\text{obj}} \left[ \left( \sqrt{w_i} - \sqrt{\hat{w}_i} \right)^2 + \left( \sqrt{h_i} - \sqrt{\hat{h}_i} \right)^2 \right]$$

$$+ \sum_{i=0}^{S^2} \sum_{j=0}^B \mathbb{1}_{ij}^{\text{obj}} (C_i - \hat{C}_i)^2$$

### Confidence Loss

$$+ \lambda_{\text{noobj}} \sum_{i=0}^{S^2} \sum_{j=0}^B \mathbb{1}_{ij}^{\text{noobj}} (C_i - \hat{C}_i)^2$$

$$+ \sum_{i=0}^{S^2} \mathbb{1}_i^{\text{obj}} \sum_{c \in \text{classes}} (p_i(c) - \hat{p}_i(c))^2$$

### Classification Loss

23

#### - Box Regression Loss

예측 상자에 객체가 포함된 경우에만 계산이 적용된다. 예측 확률과 실제값의 오차의 제곱을 적용한 값으로 객체를 책임지는 box만 대상으로 학습한다.

#### - Confidence Loss

예측된 경계 상자에 개체가 있는지 확인한다. 이 손실 함수는 모델이 background와 foreground 영역을 구별하는 데 도움을 준다. 예측 상자와 모든 실제 상자들의 IoU가 특정 threshold 미만이면 background로, 그렇지 않으면 foreground(혹은 객체)으로 결정한다. 예측된 confidence score와 ground truth의 IoU 오차를 기반으로 학습하며 물체가 없는 영역에 대한 값을 덜 반영한다.

#### - Classification Loss

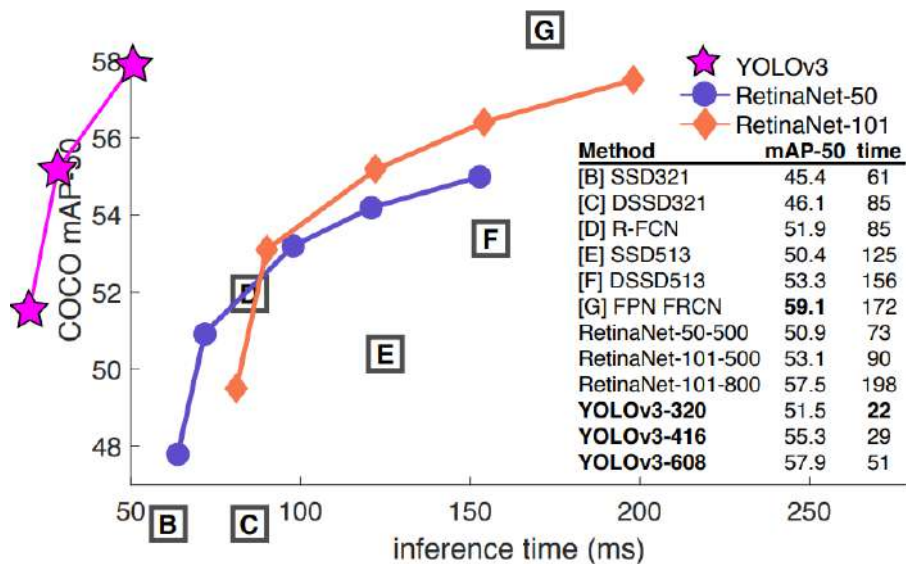
예측 상자의 개체가 속한 범주를 결정한다. class의 mutual exclusion<sup>24</sup>을 배제할 수 있고 여러 범주의 객체가 중첩되어 감지되지 않는 문제를 해결할 수 있다.

<sup>23</sup> 직접 편집한 이미지, (원본 사진 - Joseph Redmon 외, "You Only Look Once: Unified, Real-Time Object Detection", 2018.04)

<sup>24</sup> mutual exclusion: 상호 배제, 둘 이상의 프로그램이 동시에 공유 불가능한 자원을 사용하는 것을 방지하는 것. 객체 검출의 흐름 상 2개 이상의 class에 속하는 결과를 여기는 것으로 보인다.

#### d) 성능

COCO 50 benchmark, 즉, 예측값과 실제값 사이의 IoU가 0.5보다 작으면 예측이 false positive로 판정되는 경우 RetinaNet과 비교하였을 때 유사한 mAP 성능을 보이거나 속도가 상당히 빠른 것을 확인할 수 있다. 하단의 그림을 살펴보면 기존 RetinaNet 논문에서 YOLO v3을 추가한 표를 볼 수 있다. 이는 기존 모델들과 유사한 mAP를 지니지만, 그래프에서 벗어날 정도로 빠른 속도로 작동한다.



<그림 9> YOLO v3 성능 비교 <sup>25</sup>

<sup>25</sup> Joseph Redmon, Ali Farhadi, "YOLOv3: An Incremental Improvement", (2018.04)

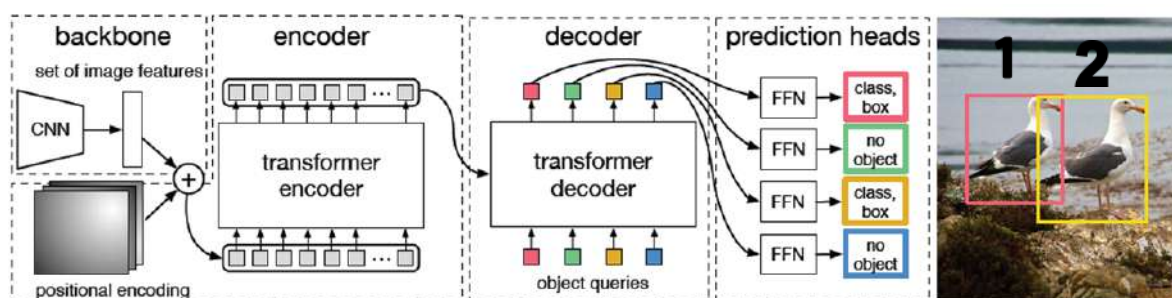
### 3. DETR

#### a) 등장 배경

현존하고 있는 객체 탐지 모델들을 설명하기 위해 NMS, Region proposal, Anchor box 등 새로운 개념들이 등장했지만, 이러한 개념들은 이해하기 어려울 뿐더러 구현하기 어렵고 코드가 길어진다는 단점을 가지고 있다. 그래서 이러한 개념을 효과적으로 제거하여 간소화된 파이프 라인을 가지며 단순하면서 end-to-end 학습이 가능한 새로운 프레임 워크 DETection TRansformer(DETR)을 만들게 되었다.

#### b) 구조와 진행흐름

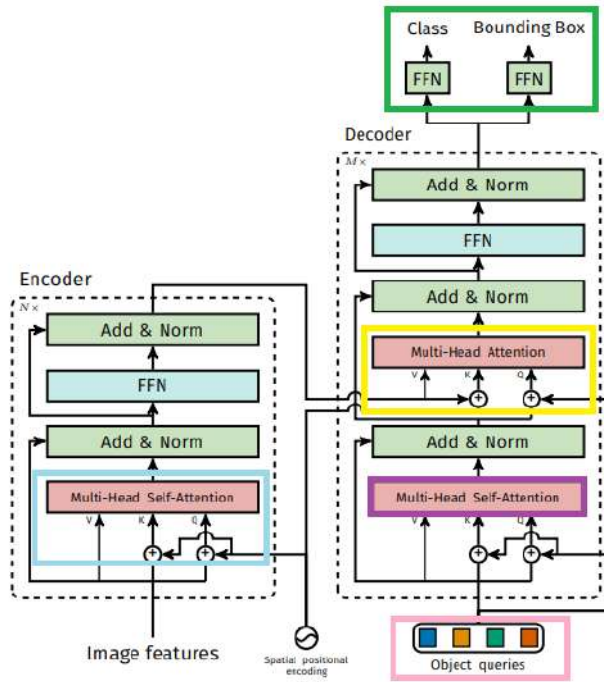
DETR 모델의 구조는 CNN Backbone(ResNet50) + Transformer(encoder-decoder) + FFN으로 되어있다.



<그림 10> DETR 구조<sup>26</sup>

현재까지의 객체 탐지 구조에서 볼 수 없었던 Transformer는 순서적인 데이터 간의 연관성을 병렬적으로 파악하여 자연어 처리 및 음성 처리 등의 분야에서 활발히 활용되고 있었다. DETR은 다차원 행렬의 형태를 띤 이미지를 순차적인 형태로 변경하여 Transformer에 넣어주면 픽셀 간의 연관성 및 유사도를 거시적으로 파악할 수 있을 것이라는 관점에서 고안되었다.

<sup>26</sup> Object Detection, Xizhou Zhu, Weiye Su, Lewei Lu, Xiaogang Wang, Jifeng Dai, arXiv 2020, <https://arxiv.org/pdf/2005.12872.pdf>



〈그림 11〉 Transformer 구조<sup>27</sup>

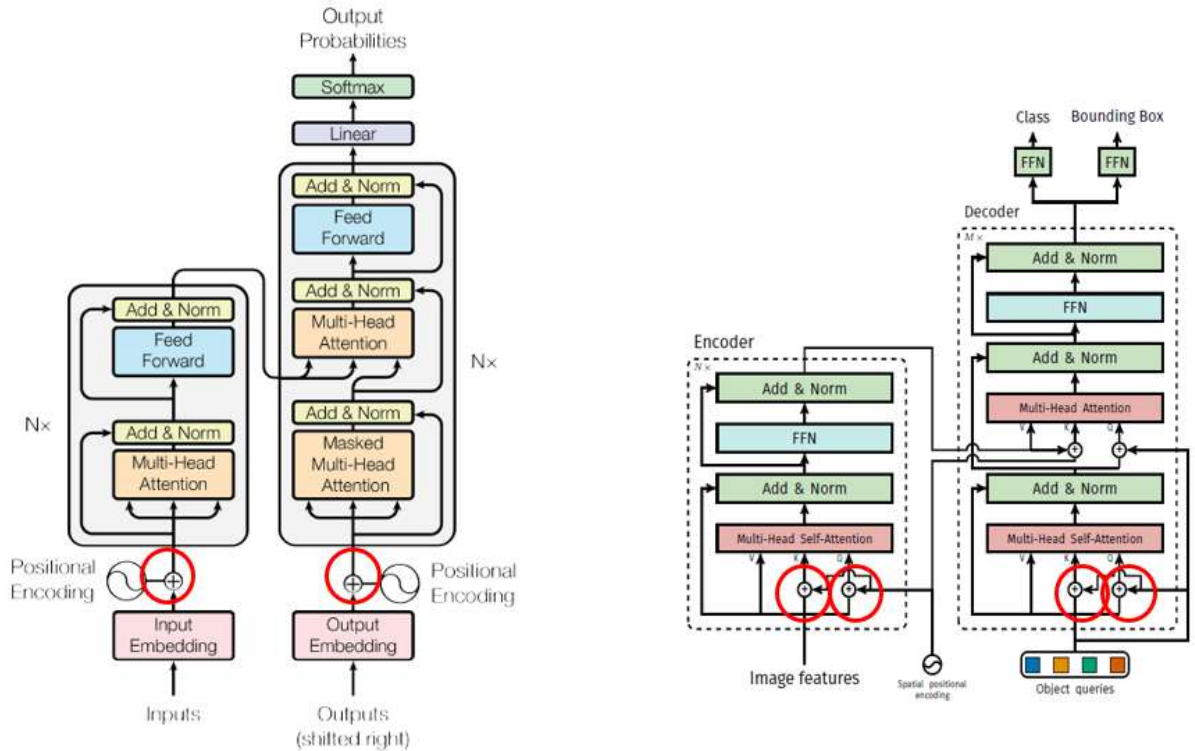
Transformer에서 하는 일을 간단히 설명해보면 다음과 같다. Encoder 부분(파란색 박스)에서는  $d * HW$ 의 특징 맵에 Positional encoding 정보를 더한 행렬을 multi-head self-attention에 통과시킨다. Decoder 부분 중 분홍색 박스는 N개의 경계 박스에 대해 N개의 오브젝트 쿼리를 생성하고, 초기 쿼리는 0으로 설정되어 있다. 보라색 박스에서 Decoder는 앞서 말한 N개의 오브젝트 쿼리를 입력 받아 multi-head self-attention을 거쳐 가공된 N개의 unit을 출력한다. 노란색 박스에서는 이 N개의 유닛들이 쿼리로, 그리고 Encoder의 출력 유닛들이 Key와 Value로 작동한다. 초록색 박스에서는 최종적으로 N개의 유닛들이 각각 FFN을 거쳐 클래스와 경계 박스 정보를 출력한다.

#### - Original Transformer와의 차이점

Original Transformer(“Attention is All You Need” 논문)와 DETR의 Transformer 구조는 비슷하면서도 약간 다르다.

1. Positional encoding 하는 위치가 다르다. CNN 백본으로 뽑아낸 특징맵에는 위치 정보가 소실되어 있다. 기존의 Transformer도 이와 같은 문제점을 해결하기 위해 Positional encoding을 더해주었다. DETR도 마찬가지로 Positional encoding을 더해주는데 위치가 다르다.

<sup>27</sup> Object Detection, Xizhou Zhu, Weiye Su, Lewei Lu, Xiaogang Wang, Jifeng Dai, arXiv 2020, <https://arxiv.org/pdf/2005.12872.pdf>



<그림 12><sup>28</sup>

2. Autoregression이 아닌 Parallel 방식으로 출력한다. 기존 Transformer는 단어 한 개씩 순차적으로 출력값을 내놓는다. Autoregression은 현재 출력값을 출력하기 위해 이전 단계까지 출력한 출력값을 참고하는 방식이다. 반면 DETR에서 사용한 Transformer는 Parallel 방식으로, 즉 모든 출력값을 한번에 출력하는 방식이다.

DETR은 충분히 큰 수의 경계 박스를 N개 설정하고 이에 대해서 클래스와 경계 박스의 크기 및 위치를 예측한다. Class 집합에는 사전에 정의한 Class 외에 No-object class를 추가하여 출력값 중 객체가 없는 경우 No-object class에 배정하도록 한다.

<그림 12>을 보면 입력 이미지에 2개의 객체만 존재한다면 2개의 경계 박스에 대해서는 클래스와 경계 박스의 크기 및 위치를 예측하고 나머지 2개에 대해서는 No object를 출력하게 된다. 하지만 앞서 말했듯 DETR은 N개의 경계 박스가 동시에 출력된다. 즉 N개의 경계 박스가 어떤 객체를 검출하고 있는지 알 수 없다는 것이다. 그렇다면 만약 그림에서 분홍색 경계 박스가 1번 갈매기에 대한 경계 박스라면 손실(loss) 값은 작겠지만, 2번 갈매기에 대한 경계 박스라면 손실 값이 클 것이다. 따라서 경계 박스가 어떤 객체를 검출하고 있는지 1:1로 매칭을 해주는 과정이 필요하며 이를 이분 매칭(bipartite matching)이라고 한다. 그래프 탐색으로 모든 경우에 대한 매칭 손실을 계산하여 비교하게 되면 총  $O(n! * n)$ 의 복잡성을 가지지만 본문에서는 헝가리안(Hungarian) 알고리즘을 활용하여 복잡성을  $O(n^3)$ 으로 향상시켰다.

<sup>28</sup> 한뫼컴비 외 6명, “한뫼한뫼 딥러닝 컴퓨터 비전 백과사전”, <https://wikidocs.net/145910>



## - 이분매칭

이분매칭은 A집단이 B집단을 선택하는 방법에 대한 알고리즘이다. 예시를 들어 살펴보자면 ‘사람’이라는 집단과 ‘알파벳’이라는 두 개의 집단이 있다고 가정한다.

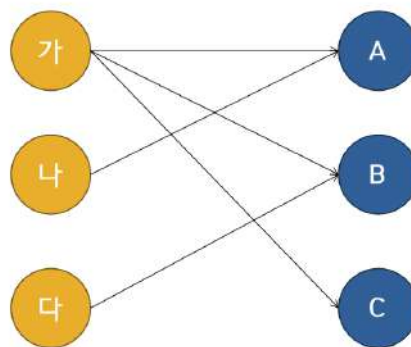
**A | B | C**

가 : A, B, C 다 좋아.

나 : A가 좋아.

다 : B가 좋아.

위와 같이 ‘사람’ 집단에 존재하는 각 인원이 원하는 항목이 정해져 있을 때 가장 효과적으로 매칭시켜 줄 수 있는 경우는 바로 그래프 형태로 표현할 수 있다.



〈그림 13〉

효과적으로 매칭시킨다는 것은 곧 ‘최대 매칭(Max matching)’을 의미한다. 일반적인 이분 그래프에서 최대 매칭은 최대 유량을 구하는 알고리즘을 통해 구할 수 있다.

## - 헝가리안 알고리즘

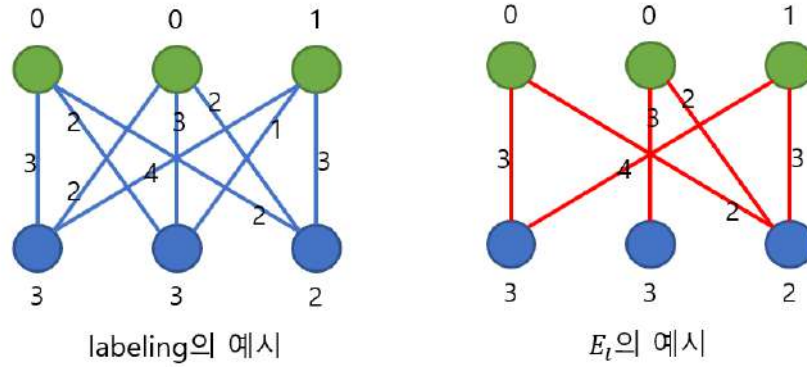
이 알고리즘 또한 가중치가 있는 이분 그래프에서 최대 유량 매칭을 찾기 위한 알고리즘이다.

헝가리안 알고리즘은 각 요소마다 적당한 수를 라벨링한다. 집합 X, Y에 대해서

$l(x) + l(y) \geq w(x, y)$ 를 만족할 때, 이러한 라벨링은 실현 가능하다고 한다.

$l(x) + l(y) = w(x, y)$ 인 에지들을 모아놓은 것을 Equality Graph라고 한다. 예를 들어 아래와 같은 그래프에서 하나의 실현가능한 라벨링과 그 때의 Equality Graph를 나타낸 것이다.





〈그림 14〉<sup>29</sup>

라벨링 1(초록색 원)과 라벨링2(파란색 원)의 매칭을 보면,  $l(1) + l(2) = w(1,2) = 3$ 이 된다. 매칭의 가중치도 3이므로 이 매칭은 실현가능하다는 것이다. 하지만 라벨링 1(초록색 원)과 라벨링 3(두번째 파란색 원)의 매칭을 보면  $l(1) + l(3) = w(1,3) = 4$ 가 되어야 한다. 하지만 매칭의 가중치는 1이므로 실현가능하지 않다. 그러므로 오른쪽의 Equality Graph에서 해당 매칭은 존재하지 않는다.

또한 하나의 객체에 대해 여러 출력을 생성하는 Near-duplicate prediction 문제도 이분 매칭을 사용하게 되면 결국 하나의 쌍으로 매칭되기 때문에 학습과정에서 어느정도 해결된다고 볼 수 있다.

### c) 손실 함수

- $\sigma$  : Ground truth의 object set의 순열
- $\sigma_{\text{hat}}$  :  $L_{\text{match}}$ 를 최소로 하는 예측 bounding box set의 순열
- $y$  : Ground truth의 object set //  $y_{\text{hat}}$ : 예측한 N object set
- $c$  : class label //  $p(c)$ : 해당 class에 속할 확률
- $b$  : bounding box의 위치와 크기 (x, y, w, h)

#### - $L_{\text{match}}$

실제 경계 박스와 예측 경계 박스가 잘 매칭되었을 때 낮은 값을 가지도록 한다. 해당 과정은 다른 객체 탐지와 다르게 중복되는 예측이 나오지 않는다.

$$-\mathbf{1}_{\{c_i \neq \emptyset\}} \hat{p}_{\sigma(i)}(c_i) + \mathbf{1}_{\{c_i \neq \emptyset\}} \mathcal{L}_{\text{box}}(b_i, \hat{b}_{\sigma(i)})_{30}$$

<sup>29</sup> 삼성 소프트웨어 멤버십 블로그, <http://www.secmem.org/blog/2021/04/18/hungarian-algorithm/>

<sup>30</sup> 한때컴비 외 6명, “한땀한땀 딥러닝 컴퓨터 비전 백과사전”, <https://wikidocs.net/145910>

- $p(c)$  : 해당 클래스로 예측한 확률이다. 앞에 마이너스가 붙어 있으므로 해당 확률이 높을 수록  $L_{match}$ 가 작아진다.
- $L_{box}$  : 실제 경계 박스와 예측 경계 박스 사이의 손실값이다. 두 박스가 비슷할 수록  $L_{match}$ 가 작아진다.

경계 박스의 크기가 클수록 loss가 커지므로 아래와 같이 경계 박스 간 IoU loss를 더하여 이를 보정해준다.

$$\lambda_{iou} \mathcal{L}_{iou}(b_i, \hat{b}_{\sigma(i)}) + \lambda_{L1} ||b_i - \hat{b}_{\sigma(i)}||_1^{31}$$

손실 함수로 IOU loss를 사용하는 이유는 DETR이 작은 객체 검출에 대한 성능이 부족하기 때문에 이를 보완하기 위해서입니다. DETR은 CNN backbone에서 하나의 feature map만을 추출하여 객체 검출에 활용하기 때문에, Feature Pyramid Network(FPN)와 같이 여러 scale의 feature map을 추출하여 객체를 검출하는 방식과 비교해서 작은 객체에 대한 탐지 성능이 떨어진다. IOU 값은 작은 객체든 큰 객체든 면적이 겹치는 비율에 따라 결정되기 때문에 IOU loss를 이용하면 작은 객체도 손실 함수에 크게 기여할 수 있게 된다.

#### - $\sigma_{hat}$

$L_{match}$ 를 최소로 하는 예측 경계 박스 순서  $\sigma_{hat}$ 를 찾는다.

$$\hat{\sigma} = \arg \min_{\sigma \in \mathfrak{S}_N} \sum_i^N \mathcal{L}_{match}(y_i, \hat{y}_{\sigma(i)})^{32}$$

이후 Hungarian 알고리즘을 통해 최적의 cost 값을 가지는 output-target 조합을 정하고 나서 손실 함수를 계산하게 된다.

#### - Hungarian loss

$\sigma_{hat}$ 을 찾았으면 이분 매칭이 완료된 것이므로 loss를 구할 수 있다. loss는 loss를 최소화하는 방식으로 학습이 진행되도록 Hungarian loss를 계산한다.

$$\mathcal{L}_{Hungarian}(y, \hat{y}) = \sum_{i=1}^N \left[ -\log \hat{p}_{\hat{\sigma}(i)}(c_i) + \mathbf{1}_{\{c_i \neq \emptyset\}} \mathcal{L}_{box}(b_i, \hat{b}_{\hat{\sigma}(i)}) \right]^{33}$$

<sup>31</sup> 한땀컴비 외 6명, “한땀한땀 딥러닝 컴퓨터 비전 백과사전”, <https://wikidocs.net/145910>

<sup>32</sup> 한땀컴비 외 6명, “한땀한땀 딥러닝 컴퓨터 비전 백과사전”, <https://wikidocs.net/145910>

<sup>33</sup> 한땀컴비 외 6명, “한땀한땀 딥러닝 컴퓨터 비전 백과사전”, <https://wikidocs.net/145910>

#### d) 성능

DETR 논문에서 Faster R-CNN을 비교 모델로 COCO minival 데이터셋 을 이용하여 동일한 조건 아래 성능을 비교했다. 훈련시간은 총 72시간이다.

Table 1: Comparison with Faster R-CNN with a ResNet-50 and ResNet-101 backbones on the COCO validation set. The top section shows results for Faster R-CNN models in Detectron2 [50], the middle section shows results for Faster R-CNN models with Glou [38], random crops train-time augmentation, and the long 9x training schedule. DETR models achieve comparable results to heavily tuned Faster R-CNN baselines, having lower AP<sub>S</sub> but greatly improved AP<sub>L</sub>. We use torchscript Faster R-CNN and DETR models to measure FLOPS and FPS. Results without R101 in the name correspond to ResNet-50.

| Model                 | GFLOPS/FPS | #params | AP          | AP <sub>50</sub> | AP <sub>75</sub> | AP <sub>S</sub> | AP <sub>M</sub> | AP <sub>L</sub> |
|-----------------------|------------|---------|-------------|------------------|------------------|-----------------|-----------------|-----------------|
| Faster RCNN-DC5       | 320/16     | 166M    | 39.0        | 60.5             | 42.3             | 21.4            | 43.5            | 52.5            |
| Faster RCNN-FPN       | 180/26     | 42M     | 40.2        | 61.0             | 43.8             | 24.2            | 43.5            | 52.0            |
| Faster RCNN-R101-FPN  | 246/20     | 60M     | 42.0        | 62.5             | 45.9             | 25.2            | 45.6            | 54.6            |
| Faster RCNN-DC5+      | 320/16     | 166M    | 41.1        | 61.4             | 44.3             | 22.9            | 45.9            | 55.0            |
| Faster RCNN-FPN+      | 180/26     | 42M     | 42.0        | 62.1             | 45.5             | 26.6            | 45.4            | 53.4            |
| Faster RCNN-R101-FPN+ | 246/20     | 60M     | 44.0        | 63.9             | <b>47.8</b>      | <b>27.2</b>     | 48.1            | 56.0            |
| DETR                  | 86/28      | 41M     | 42.0        | 62.4             | 44.2             | 20.5            | 45.8            | 61.1            |
| DETR-DC5              | 187/12     | 41M     | 43.3        | 63.1             | 45.9             | 22.5            | 47.3            | 61.1            |
| DETR-R101             | 152/20     | 60M     | 43.5        | 63.8             | 46.4             | 21.9            | 48.0            | 61.8            |
| DETR-DC5-R101         | 253/10     | 60M     | <b>44.9</b> | <b>64.7</b>      | 47.7             | 23.7            | <b>49.5</b>     | <b>62.3</b>     |

<그림 15><sup>34</sup>

표를 보면 AP75, APs에서 Faster R-CNN이 높고 나머지는 DETR이 높다. 이는 DETR이 큰 객체 검출에는 좋은 성능을 보이지만 작은 객체 검출에는 그렇지 않다는 것을 의미한다. 연산량 측면에서 보면 DETR이, FPS에서는 Faster R-CNN이 좋은 성능을 보인다. ‘+’ 가 붙은 Faster R-CNN은 3배의 시간을 더 훈련시켰다고 한다. DETR의 객체 검출 속도와 정확성에 관해서는 성능 평가 부분에서 더 자세히 다룰 예정이다.

딥러닝 분야별로 알고리즘을 볼 수 있는 SOTA(State-Of-The-Art)에서 COCO minival 데이터셋을 기준으로 검색해본 결과는 다음과 같다. 현재를 기준으로 DETR은 해당 위치이다.

<sup>34</sup> Object Detection, Xizhou Zhu, Weiye Su, Lewei Lu, Xiaogang Wang, Jifeng Dai, arXiv 2020, <https://arxiv.org/pdf/2005.12872.pdf>



<그림 16> COCO-minval 기준으로 현존하는 모델 중 DETR의 성능적 측면에서의 위치 <sup>35</sup>

<sup>35</sup> SOTA, <https://paperswithcode.com/sota/object-detection-on-coco-minival>

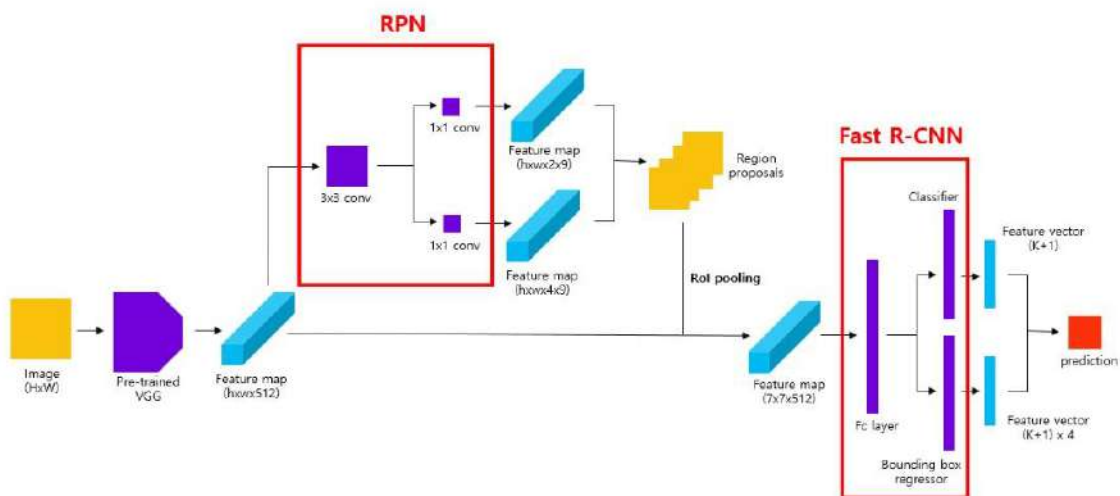
# IV. Two-stage Detection

## 1. Faster R-CNN

### a) 등장 배경

기존 Fast-RCNN 모델은 selective search 알고리즘을 통해 region proposals을 추출하기 때문에 학습 및 detection 속도를 향상시키는데 한계가 있다. 또한 detection 과정을 end-to-end 방법으로 수행하지 못한다는 문제가 있다. 이러한 문제를 해결하기 위해 처리 속도와 모델의 완성도를 개선한 Faster R-CNN 모델이 등장하였다.

### b) 구조



<그림 17> Faster R-CNN 구조<sup>36</sup>

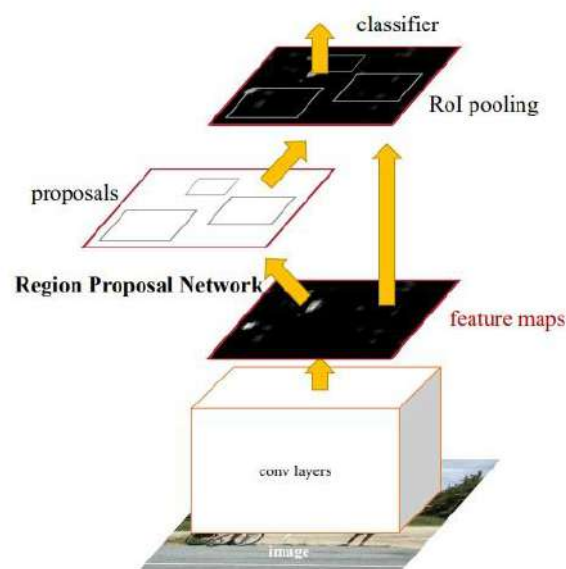
Faster R-CNN 모델을 간략하게 보면 RPN+Fast R-CNN이라고 볼 수 있다. 그동안 Selective Search를 사용하여 계산해왔던 Region Proposal 단계를 Neural Network 안으로 끌어와서 진정한 의미의 end-to-end Object Detection 모델을 제시하였으며, 기존 Fast R-CNN 구조를 계승하면서 selective search를 대신 RPN을 통해서 RoI를 계산한다. 이로 인해 GPU를 통한 RoI 계산이 가능해졌으며, RoI 계산

<sup>36</sup> herbwood, "Faster R-CNN 논문(Faster R-CNN: Towards Real-Time ObjectDetection with Region Proposal Networks) 리뷰"(2020), <https://hhnam.tistory.com/11>

역시도 학습시켜 정확도를 높였다. 또한, RPN은 Selective Search가 2000개의 RoI를 계산하는 데 반해 800개 정도의 RoI를 계산하면서도 더 높은 정확도를 보인다.

Faster R-CNN은 하나의 Backbone network에 해당하는 커다란 네트워크가 있는데, 이 네트워크는 주어진 영상의 features를 CNN을 통해 추출하는 데 사용된다. 이렇게 추출된 features는 1차 적으로 Region proposal을 얻는 데 사용되고, 이렇게 주어진 region proposal을 기반으로 앞서 주어진 feature에서 해당 영역들의 feature를 모아 classification 단계를 거친 후 최종 결과를 도출한다.

### c) 진행 과정



〈그림 18〉 Faster R-CNN 진행 과정<sup>37</sup>

RPN에 region proposals를 추출하고 이를 Fast R-CNN 네트워크에 전달하여 객체의 class와 위치를 예측한다. 이를 통해 모델의 전체 과정이 GPU상에서 동작하여 병목 현상이 발생하지 않으며, end-to-end 네트워크로 학습시키는 것이 가능해진다. 전체적인 동작 순서는 아래와 같다.

1. 원본 이미지를 pre-trained된 CNN 모델에 입력하여 feature map을 얻는다.
2. feature map을 RPN에 전달하여 적절한 region proposals를 산출한다.
3. region proposals 와 1에서 얻은 feature map을 통해 RoI Pooling을 수행하여 고정된 크기의 feature map을 얻는다.
4. Fast R-CNN 모델에 고정된 크기의 feature map을 입력하여 classification과 Bounding box regression을 수행한다.

<sup>37</sup> Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun, "Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks" (2016)

#### d) 손실 함수

우선 RPN을 훈련하기 위해 각 anchor box마다 이진 분류를 수행한다. 분류를 하려면 anchor box에 positive label이 있어야 하는데, (i) 실제 경계 박스(ground-truth box)와 IoU가 가장 큰 앵커, 또는 (ii) 실제 경계 박스(ground-truth box)와 IoU가 0.7이 넘는 앵커이다. 이때 실제 경계 박스 하나마다 여러 앵커 박스를 positive label로 할당할 수 있는데, 객체 위치를 정확히 나타내는 앵커 박스 단 하나만 positive label로 사용하는 게 아니라는 것이다. 실제 경계 박스와 IoU가 높다면 여러 앵커 박스를 positive label로 간주한다. 보통은 (ii) 조건으로 앵커 박스를 찾지만, (ii) 조건을 만족하는 앵커 박스가 없으면 (i) 조건으로 positive label 앵커 박스를 찾는다. 또한, IoU가 0.3보다 작은 앵커 박스는 negative label로 간주하고 positive도 negative도 아닌 앵커 박스는 훈련에서 제외하기 때문에 객체 탐지에 영향을 미치지 않는다.

#### - Multi-task Loss

$$L(\{p_i\}, \{t_i\}) = \frac{1}{N_{cls}} \sum_i L_{cls}(p_i, p_i^*) + \lambda \frac{1}{N_{reg}} \sum_i p_i^* L_{reg}(t_i, t_i^*)$$

RPN과 Fast R-CNN을 학습시키기 위해 Multi-task loss(Classification Loss + Regression Loss)를 사용한다.  $i$ 는 앵커 박스 인덱스를 뜻하며  $p_i$ 는  $i$ 번째 앵커 박스가 객체일 확률을 말한다. ground truth label인  $p_i^*$ 는 앵커가 positive이면 1, negative이면 0이다. 이때 negative라는 건 배경을 의미한다. 그리고  $t_i$ 는 예측 경계 박스의 4가지 좌표값이고,  $t_i^*$ 는 실제 경계 박스의 4가지 좌표값이다. 분류 손실을 나타내는  $L_{cls}$ 는 두 가지 클래스(객체 또는 객체가 아님)에 대한 로그 손실이고  $L_{reg}$ 는 경계 박스 회귀 손실을 뜻한다. 회귀 손실값은 positive 앵커 박스일 때만(객체 일 때만) 활성화된다. negative일 때는 배경일 때 이므로 경계 박스를 구할 필요가 없기 때문이다. 또한, 회귀 손실에  $p_i^* \times L_{reg}$ 가 포함되므로, 앵커 박스가 negative일 때는 이 값이 0이 된다( $p_i^* =$  앵커가 positive이면 1, negative이면 0). 마지막으로 분류 손실과 회귀 손실 모두  $N_{cls}$ 와  $N_{reg}$ 로 나눠서 정규화했으며 분류 손실, 회귀 손실 간 균형을 맞추기 위해  $\lambda$  파라미터를 두었다.

Classification은 log loss를 통해서 계산하고, regression loss의 경우 smoothL1 함수를 사용한다.

RPN에서는 객체의 존재 여부만을 분류하는 반면, Fast R-CNN에서는 배경을 포함을 class를 분류한다는 점에서 차이가 있다.



## e) 성능

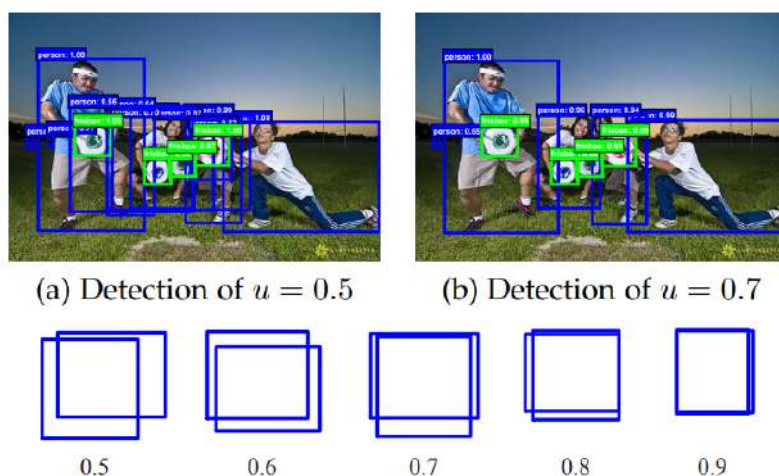
| method                           | proposals | training data | COCO val |               | COCO test-dev |               |
|----------------------------------|-----------|---------------|----------|---------------|---------------|---------------|
|                                  |           |               | mAP@.5   | mAP@[.5, .95] | mAP@.5        | mAP@[.5, .95] |
| Fast R-CNN [2]                   | SS, 2000  | COCO train    | -        | -             | 35.9          | 19.7          |
| Fast R-CNN [impl. in this paper] | SS, 2000  | COCO train    | 38.6     | 18.9          | 39.3          | 19.3          |
| Faster R-CNN                     | RPN, 300  | COCO train    | 41.5     | 21.2          | 42.1          | 21.5          |
| Faster R-CNN                     | RPN, 300  | COCO trainval | -        | -             | <b>42.7</b>   | <b>21.9</b>   |

<그림 19> Faster R-CNN 성능<sup>38</sup>

MS COCO 데이터셋을 사용하여 학습한 모델의 결과 Faster R-CNN이 Fast R-CNN보다 더 높은 성능을 보임을 확인할 수 있다.

## 2. Cascade R-CNN

### a) 등장 배경



<그림 20> threshold 값에 따른 Bounding Box<sup>39</sup>

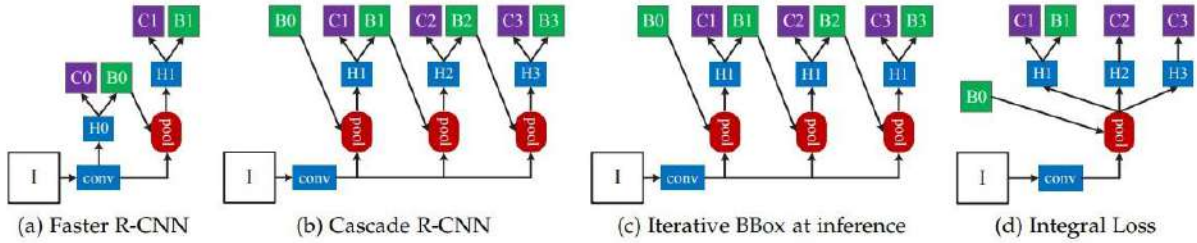
기존의 R-CNN 계열의 object detector들은 positive/negative를 정의하는 IoU의 threshold  $u$ 를 0.5로 설정한다. 그 결과 <그림 20>의 (a)와 같이 잡음이 많은 BB(Bounding Box)를 출력한다.  $u$ 를 0.7로 설정하면 BB의 결과 값이 정확한 대신 detection의 전체 성능이 감소한다. 많은 proposals를 검출하지 못하기 때문에 recall이 감소해 AP가 감소하기 때문이다. Zhaowei Cai, Nuno Vasconcelos는 이러한 문제점을 지적하며 간단한 방법으로 개선하여 R-CNN의 성능을 크게 높였는데, 이것이 Cascade R-CNN이다.

<sup>38</sup> Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun, "Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks" (2016)

<sup>39</sup> Zhaowei Cai, Nuno Vasconcelos, "Cascade R-CNN: High Quality Object Detection and Instance Segmentation", (2019)



## b) 구조 및 진행 과정



〈그림 21〉 'I' = Input image, 'conv' = Backbone convolutions, 'pool' = region-wise feature extraction, 'H' = Network head, 'B' = bounding box, 'C' = classification. (B0는 proposals in all architectures.)<sup>40</sup>

Faster R-CNN이 1개의 classifier만 사용했다면, Cascade R-CNN은 추가적인  $n$ 개(보통은 2개 추가)의 classifier 사용한다. 〈그림 21〉의 (b)를 보면 영상  $I$ 가 들어왔을 때 conv단계에서 영상의 features를 추출하고 RPN이 물체가 있을 만한 지역들을 BB로 추천해 준다. 그리고 해당 BB가 지정하는 위치의 features를 RoI pooling해서 해당 위치의 물체가 어떤 물체인지를 classifier가 분간하고 한 번 더 정확한 BB를 추정한다. 해당 classifier는 이전의 classifier가 만들어낸 BB를 받아 새로운 classification task를 수행하고 거기서 만들어지는 BB를 그다음의 classifier에게 넘긴다. 이러한 과정에서 각각의 단계에서 만들어내는 BB는 점점 더 정확할 것이라고 가정하고, 그다음 단계의 classifier는 이전 단계보다 더 높은 IoU를 기준으로 학습시킴으로써 성능을 극대화 시킨다.

## c) 손실 함수

$$L(x^t, g) = L_{cls}(h_t(x^t), y^t) + \lambda[y^t \geq 1]L_{loc}(f_t(x^t, b^t), g)$$

IoU threshold를 적용하면 할 수록 output IoU가 높아지고, 각 stage를 진행할 수록 IoU를 더 높여가면 더 좋은 output IoU를 얻을 수 있다. 그래서 다음 stage로 넘어갈 수록 높은 input 이미지가 남게 되고, IoU threshold를 다음 stage에서 높여도 output IoU가 크게 떨어지지 않는 것이다. 이렇게 진행하면 overfitting이 일어나지 않아 deeper stage에서도 높은 IoU threshold를 적용할 수 있고 자연스럽게 각 stage를 진행할 수록 outlier들이 제외된다.

<sup>40</sup> Zhaowei Cai, Nuno Vasconcelos, "Cascade R-CNN: High Quality Object Detection and Instance Segmentation", (2019)

#### d) 성능

|                       | AP          | AP <sub>50</sub> | AP <sub>60</sub> | AP <sub>70</sub> | AP <sub>80</sub> | AP <sub>90</sub> |
|-----------------------|-------------|------------------|------------------|------------------|------------------|------------------|
| FPN+ baseline         | 34.9        | 57.0             | 51.9             | 43.6             | 29.7             | 7.1              |
| <i>Iterative BBox</i> | 35.4        | 57.2             | 52.1             | 44.2             | 30.4             | 8.1              |
| <i>Integral Loss</i>  | 35.4        | 57.3             | 52.5             | 44.4             | 29.9             | 6.9              |
| Cascade R-CNN         | <b>38.9</b> | <b>57.8</b>      | <b>53.4</b>      | <b>46.9</b>      | <b>35.8</b>      | <b>15.8</b>      |

〈그림 22〉 다른 모델 기법과의 성능 비교

〈그림 22〉는 Cascade R-CNN의 효과를 비교하기 위해 이전에 비슷한 직관을 가지고 있었지만 Cascade R-CNN과는 다른 접근 방법을 제안했던 방법 2가지와 성능을 비교한 것이다.

첫번째 방법은 Iterative BBox at inference 모델로, faster R-CNN과 마찬가지로 하나의 classifier만을 학습시켜 두고 이를 test 시에 iterative하게 활용하는 방법이다. 이는 〈그림 21〉의 (C)에 표현되어 있는데, input 영상이 들어가고 RPN을 거쳐 classifier가 BB를 만들어내면 이 BB는 RPN이 만들어낸 BB보다 더 정확하게 예측한 BB라고 기대할 수 있다. 그래서 동일한 classifier에 이전에 본인이 예측한 BB를 이용해 다시 한번 classification을 수행하면 좀 더 정확하게 classification task를 수행 할 수 있을 것이라고 기대할 수 있다. 이 방법은 실제로 성능향상을 보여주는가 하나, 상대적으로 극적인 효과를 보여주지는 못한다. 또한, 2번 이상에 iteration은 성능 향상에 도움을 주지 못했다.

두번째 방법은 〈그림 21〉에서 (d)에 해당하는 방법이다. RPN이 한번만 proposal을 하는데 그때의 proposal이 어떤때는 더 정확했을 수 있고, 어떤 때는 상대적으로 부정확했을 수도 있기 때문에, 서로 다른 classifier를 각각의 IoU 기준마다 학습을 시키고 test시에는 최종적으로 모든 classifier가 만들어낸 결과들을 ensemble하는 방법이다. 이 역시도 성능 향상을 보여주는가 했으나, 여전히 proposal이 정확하지 않을 수 있다는 문제점은 그대로 남아있었기에 성능향상이 두드러 지지는 않았다.

여기서의 모든 성능 검증은 MS COCO dataset을 활용하였고, Cascade R-CNN은 3개의 classifier를 사용해서 학습되었다. 모든 학습과정은 end-to-end로 한번에 학습되었으며 각각의 classifier는 각각 IoU 기준 0.5, 0.6, 0.7 로 학습되었다. Integral Loss 방법도 마찬가지로 3개의 classifiers를 학습시킬때 위와 동일한 IoU 기준을 사용하였다. 〈그림 22〉의 결과를 살펴보면 기존의 방법들은 근소한 수준으로 성능을 올렸지만 Cascade R-CNN의 경우에는 훨씬더 큰 폭의 성능 향상을 보여줌을 확인 할 수 있고, 특히 AP80 이상에서의 성능향상이 크게 나타남을 알 수 있다.

| # stages | test stage   | AP          | AP <sub>50</sub> | AP <sub>60</sub> | AP <sub>70</sub> | AP <sub>80</sub> | AP <sub>90</sub> |
|----------|--------------|-------------|------------------|------------------|------------------|------------------|------------------|
| 1        | 1            | 34.9        | 57.0             | 51.9             | 43.6             | 29.7             | 7.1              |
| 2        | <u>1 ~ 2</u> | 38.2        | <b>58.0</b>      | <b>53.6</b>      | 46.7             | 34.6             | 13.6             |
| 3        | <u>1 ~ 3</u> | <b>38.9</b> | 57.8             | 53.4             | <b>46.9</b>      | 35.8             | 15.8             |
| 4        | <u>1 ~ 3</u> | <b>38.9</b> | 57.4             | 53.2             | 46.8             | <b>36.0</b>      | 16.0             |
| 4        | <u>1 ~ 4</u> | 38.6        | 57.2             | 52.8             | 46.2             | 35.5             | <b>16.3</b>      |

〈그림 23〉 Cascade R-CNN 성능<sup>41</sup>

Cascade R-CNN의 각각의 classifier가 만들어내는 결과를 살펴보면 classifier를 거치면 거칠수록 AP80이상에서 확실히 성능이 올라감을 볼 수 있다. 그러나 3번째 stage를 살펴보면 오히려 AP50의 경우에는 성능이 약간 저하되는 것을 볼 수 있는데, 이는 1번째 stage의 경우에는 다양한 객체들을 모두 검출하려 하는 반면에 3번째는 더 정확하게 검출하려 하다보니 경우에 따라서는 놓치는 때가 생겨서 그런 것이라고 추측할 수 있다. 그렇기 때문에 3번째 Cascade R-CNN을 통해 가장 좋은 성능을 얻기 위해서는 최종 stage의 결과만을 사용 하는 것이 아니라, 첫번째부터 마지막까지의 stage에서 만들어낸 결과를 ensemble하면 가장 좋은 성능을 얻을 수 있다. 그러나 4번째 classifier를 IoU 기준 0.75를 학습시켰을 때는 큰 성능 향상을 보이지 않았고, 3개 초과는 큰 성능 향상을 기대할 수 없다는 것을 알 수 있다.

<sup>41</sup>Zhaowei Cai, Nuno Vasconcelos, "Cascade R-CNN: High Quality Object Detection and Instance Segmentation", (2019)

## V. 성능 비교

| Model         | Backbone   | Memory(GB) | Box AP |
|---------------|------------|------------|--------|
| RetinaNet     | R-50-FPN   | 3.8        | 36.5   |
| YOLO v3       | Darknet-53 | 3.8        | 30.9   |
| DETR          | R-50       | 7.9        | 40.1   |
| Faster R-CNN  | R-50-FPN   | 4.0        | 37.4   |
| Cascade R-CNN | R-50-FPN   | 4.4        | 40.3   |

〈표 1〉 Object Detection model 성능 비교

### a) 예시 ①

- 원본 이미지



| Model     | Output   |
|-----------|--|
| RetinaNet |  <p>The image shows an airfield with several aircraft. Green bounding boxes are drawn around four aircraft in the middle ground. Each box is labeled with 'airplane' and a confidence score: 0.74, 0.74, 0.74, and 0.73. The text 'result' is at the top center, and 'time : 1.2382967472076416' is at the bottom left.</p>    |
| YOLO v3   |  <p>The image shows the same airfield as the RetinaNet output. Green bounding boxes are drawn around three aircraft in the middle ground. Each box is labeled with 'airplane' and a confidence score: 0.43, 0.69, and 0.51. The text 'result' is at the top center, and 'time : 0.717247436114562' is at the bottom left.</p> |



Cascade  
R-CNN





<표 2> 예시 1의 실행 결과

b) 예시 ②



- 원본 이미지





| Model     | Output  |
|-----------|---|
| RetinaNet |  <p>The RetinaNet output shows a scene with an airplane and several trucks. Bounding boxes and confidence scores are as follows:</p> <ul style="list-style-type: none"> <li>airplane 0.74 (green box)</li> <li>airplane 0.50 (green box)</li> <li>truck 0.58 (purple box)</li> <li>truck 0.66 (purple box)</li> <li>truck 0.61 (purple box)</li> </ul> <p>Time: 1.299360990524292</p> |
| YOLO v3   |  <p>The YOLO v3 output shows the same scene with different bounding boxes and confidence scores:</p> <ul style="list-style-type: none"> <li>airplane 0.54 (green box)</li> <li>truck 0.59 (purple box)</li> <li>truck 0.61 (purple box)</li> </ul> <p>Time: 0.8172638416290283</p>   |



|                     |   |
|---------------------|---|
| <p>DETR</p>         |   |
| <p>Faster R-CNN</p> |  |



<표 3> 예시 2의 실행 결과

위의 <표 1>의 결과를 따라 각 모델마다 직접 실행해본 결과가 <표 2>, <표 3>이다. 실험 이미지는 이미지 안의 객체가 많은 것을 선정했다. 예시1과 예시2의 실행속도를 표로 정리하자면 다음과 같다.

| Model         | 예시 1  | 예시 2  | 평균 실행 시간 |
|---------------|-------|-------|----------|
| RetinaNet     | 1.238 | 1.299 | 1.2685   |
| YOLO v3       | 0.717 | 0.817 | 0.767    |
| DETR          | 1.024 | 1.158 | 1.091    |
| Faster R-CNN  | 1.407 | 1.284 | 1.3455   |
| Cascade R-CNN | 1.346 | 1.420 | 1.383    |

<표 4> Object Detection 모델 실행 시간

두 실험 이미지에 대해 평균 실행 시간은 **YOLO v3 > DETR > RetinaNet > Faster R-CNN > Cascade R-CNN** 이다. 하지만 검출 정확도의 경우 **Cascade R-CNN > Faster R-CNN > DETR > RetinaNet > YOLO v3** 이다. 시간 부분에서 타 모델보다 독보적으로 빠른 실행 시간을 가지고 있는 모델은 YOLO v3인데, 해당 모델을 제외하면 타 모델들의 평균 실행 시간의 편차가 0.0375 ~ 0.292 사이라는 점을 보아 크게 차이가 나지 않다는 것을 알 수 있다.

## VI. 결론

이번 연구의 주제는 항공 산업에서 활용될 객체 검출이라는 특정한 주제이다. 따라서 시각화된 영상을 데이터화하는데 항공기 활주로, 계류장, 공항에서 인식해야 할 중요한 대상의 기준이 비행기, 차량, 사람 등이라고 생각하였다. 실제로 여러 이미지를 실행한 결과 Cascade R-CNN 모델이 기준에 포함된 주요 대상을 가장 잘 검출하는 것으로 파악하였다. 특히 두 번째 예시의 이미지를 기준으로 DETR과 Cascade R-CNN의 결과를 비교하면 알 수 있다.

먼저 DETR의 객체 검출 정확도는 매우 높으나 동일한 객체에 대해 박스의 개수, 즉, 하나의 객체에 대해 중복적으로 검출되는 현상을 확인할 수 있다. 그러나 Cascade R-CNN의 결과는 하나의 객체에 대해 정확히 하나의 결과 box가 출력됨을 확인할 수 있다.

추가적으로 항공기 환경에 있어 실시간으로 객체를 검출해내는 능력이 매우 중요하다. 그런데 YOLO v3를 제외하고는 실행 시간이 비슷하므로 종합적으로 판단하였을 때 Cascade R-CNN이 가장 적합하다는 결과를 도출하였다.

## Reference

- [1] Object Detection, Xizhou Zhu, Weiye Su, Lewei Lu, Xiaogang Wang, Jifeng Dai, arXiv 2020  
인간지능이 인공지능을 공부하는 장소, <https://keyog.tistory.com/32> 2020
- [2] 한땀컴비 외 6명, “한땀한땀 딥러닝 컴퓨터 비전 백과사전”, <https://wikidocs.net/145910>
- [3] RE Tech Achive, <https://rauleun.github.io/DETR> 2020
- [4] sjinu, [\[논문리뷰\] DETR: End-to-End Object Detection with Transformers](#) 2021
- [5] DSBA Seminar,  
<http://dsba.korea.ac.kr/seminar/?mod=document&uid=1784>(<http://dsba.korea.ac.kr/seminar/?mod=document&uid=1784>) 2021
- [6] 정리는 습관, <https://powerofsummary.tistory.com/205> 2021
- [7] KP's blog, <https://kp1994.tistory.com/15> 2020
- [8] 삼성 소프트웨어 멤버십 블로그, <http://www.secmem.org/blog/2021/04/18/hungarian-algorithm/> 2021
- [9] 딥러닝 공부방, [\[논문 읽기\] RetinaNet\(2017\) 리뷰 Focal Loss for Dense Object Detection \(tistory.com\)](#) 2021
- [10] Sangne's log, [RetinaNet 논문 리뷰 \(velog.io\)](#) 2021
- [11] T. -Y. Lin, P. Goyal, R. Girshick, K. He and P. Dollár. Focal Loss for Dense Object Detection.arXiv 2017
- [12] Vision.log [ResNet 논문리뷰 \(velog.io\)](#) 2022
- [13] 참신러닝 <https://leechamin.tistory.com/184> 2019
- [14] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. “Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks”. *arXiv:1506.01497*, 2016.
- [15] herbwood. “Faster R-CNN 논문(Faster R-CNN: Towards Real-Time ObjectDetection with Region Proposal Networks) 리뷰”. <https://hhnam.tistory.com/11>, 2020.
- [16] “2. faster R-CNN”. <https://welcome-to-dewy-world.tistory.com/110>. Nov, 2020.
- [17] Zhaowei Cai, Nuno Vasconcelos. “Cascade R-CNN: High Quality Object Detection and Instance Segmentation”. *arXiv:1712.00726*, 2019.
- [18] brianjaum. “Cascade R-CNN: Delving into High Quality Object Detection”.  
<https://blog.lunit.io/2018/08/13/cascade-r-cnn-delving-into-high-quality-object-detection/>, Aug 2018.
- [19] 동산. “Cascade R-CNN: Delving into High Quality Object Detection”.  
<https://m.blog.naver.com/jinyuri303/221853527701>. Mar, 2020
- [20] M\_AI. “[RetinaNet] 1. Anchor Box”. <https://yhu0409.tistory.com/2>, May 2021.
- [21] bskyvision. “물체 검출 알고리즘 성능 평가방법 AP(Average Precision)의 이해”. <https://bskyvision.com/465>, Apr 2019.
- [22] graygreat. “[ML] Bounding Box”. <https://oopsys.tistory.com/229>, Oct 2020.
- [23] 유니디니. “[객체 탐지] Selective Search for Object Recognition”. <https://go-hard.tistory.com/33>, Jan 2020.
- [24] *Leadtek AI Forum*,
- [25] Ross Girshick, Jeff Donahue, Trevor Darrell, Jitendra Malik, UC Berkeley, “Rich feature hierarchies for accurate object detection and semantic segmentation Tech report (v5)”, (2014.10)
- [26] Pooja Mahajan, “Fully Connected vs Convolutional Neural Networks”,  
<https://medium.com/swlh/fully-connected-vs-convolutional-neural-networks-813ca7bc6ee5>
- [27] Taewan Kim, “CNN, Convolutional Neural Network 요약”, (2018.02), <http://taewan.kim/post/cnn/>
- [28] 한땀컴비 외 6명, “한땀한땀 딥러닝 컴퓨터 비전 백과사전”, <https://wikidocs.net/142645>
- [29] 이도현, “You Only Look Once”,,  
<https://medium.com/curg/you-only-look-once-%EB%8B%A4-%EB%8B%A8%EC%A7%80-%ED%95%9C-%EB%B2%88%EB%A7%8C-%EB%B3%B4%EC%95%98%EC%9D%84-%EB%BF%90%EC%9D%B4%EB%9D%BC%EA%B5%AC-bddc8e6238e2>
- [30] Joseph Redmon, Ali Farhadi, “YOLOv3: An Incremental Improvement “, (2018.04)
- [31] QI-CHAO MAO, HONG-MEI SUN, YAN-BO LIU, RUI-SHENG JIA, “Mini-YOLOv3: Real-Time Object Detector for Embedded Applications”
- [32] PaperspaceBlog Ayoosh Kathuria, “Series: YOLO object detector in PyTorch”,  
<https://blog.paperspace.com/tag/series-yolo/>
- [33] PyLessons, “TensorFlow 2 YOLOv3 Mnist detection training tutorial”, (2020.05),  
<https://pylessons.com/YOLOv3-TF2-mnist>
- [34] 한땀컴비 외 6명, “한땀한땀 딥러닝 컴퓨터 비전 백과사전”, <https://wikidocs.net/139127>
- [35] 한땀컴비 외 6명, “한땀한땀 딥러닝 컴퓨터 비전 백과사전”, <https://wikidocs.net/145910>

- [36] Enough is not enough, “[Object Detection] Feature Pyramid Network (FPN)”, (2020.4.6), <https://eehoeskrap.tistory.com/300>
- [37] 생각많은 소심남, “[MLY] end-to-end 학습의 장단점”, (2018.10.12), <https://talkingaboutme.tistory.com/entry/MLY-end-to-end-%ED%95%99%EC%8A%B5%EC%9D%98-%EC%9E%A5%EB%8B%A8%EC%A0%90>
- [38] 한땀컴비 외 6명, “한땀한땀 딥러닝 컴퓨터 비전 백과사전”, <https://wikidocs.net/152766>