

# 모빌리티 빅데이터 실무

---

## 세부정보

---

- 담당 : 최혁두 교수님
  - 일시 : 화 14:00 ~ 17:00
  - 주 사용 플랫폼 : Flow
  - 주차 : 1주차
- 

## 참조

---

Reference : [lanLecture](#) (교수님 블로그)

---

## 1. 강의 소개

---

### 1.1. 강의 목표

- 강의 목표 : Deep Learning based Object Detector를 구현
  - 분류 (classification) : 영상에 들어있는 주요 사물의 종류를 확률적으로 출력, 10가지 사물이 있다면 한 영상에 대해 합이 1인 10개의 확률값이 나옴.
  - 검출(object detection) : 영상에서 사물의 영역을 감싸는 직사각형 경계상자 (bounding box)를 출력, 사물의 개수에 따라 출력의 양이 달라짐
  - 분리(image segmentation) : 영상에서 사물의 영역을 픽셀 단위로 분류하여 원본 영상과 같은 크기의 segmentation map을 출력

분류 모델 - 검출모델 - 분리모델 순으로 연구 대상으로 진행 되는 것 같다.

검출은 분리보다 출력 데이터 양이 훨씬 작기 때문에 출력 속도도 빠르고 후처리도 간편하다.

검출기(Detector)를 공부하는 이유는 검출기 구현을 통해서 텐서플로(Tensroflow)나 파이토치(pytorch) 같은 딥러닝 프레임 워크(Deep learning framework)의 다양한 사용법을 **깊이있게 배울 수 있기** 때문이다.

**검출기 구현을 통해 텐서플로와 파이토치를 모두 배우는 것을 목표로 한다.**

파이토치는 넘파이(Numpy) 다루듯이 데이터 일부를 수정할 수 있고 한줄씩 실행이 가능하지만 텐서플로는 미리 연산 그래프를 고정시켜 놓고 데이터가 고정된 연산과정을 통과하는 식이라 텐서연산에 대한 기술이 필요하다.

(대신 텐서플로는 상업적인 배포에 용이하다. 웹, 모바일 등에서 모델을 배포할 수 있고, 더 많은 프로그래밍 언어를 지원한다. 보통 학습은 python으로 돌리며 배포시에는 다른언어로도 배포가 가능하다.)

딥러닝 프로그래밍을 할 때 가장 필요한 것은 다차원 배열(Tensor, 텐서)을 다루는 능력이다. 텐서 연산에서는 가급적 for문을 사용하지 않고 (거의 금지되어 있음) 다차원 배열에서 한번에 처리하는 것이 유리하다.(텐서들을 하나로 합쳐가지고 한번에 처리하는 것이 유리하다.) 보통 4차원이나 5차원 텐서를 많이 다루는데 평소에 접하지 않았던 차원이라 텐서 연산에 대한 훈련을 많이 해야 여러 차원들을 자유자재로 다루며 원하는 연산을 할 수 있다.

또다른 목표는 **유연한 객체지향적인 프로그래밍 기법 습득**이다. 코드를 단순하게 구현하면 이해하기 쉬우나 실제로 개발을 할 때에는 다양한 옵션을 바꿔가면서 실험을 해야하기 때문에 단순한 구조를 유지하기 어렵다. 언제나 다양한 가능성을 열어두고 다양하게 확장가능한 프로그램을 지향한다.

텐서플로 모델을 먼저 배우고 유사한 파이토치 구현을 배운다.

CNN 모델 구현이 주된 내용일 것 같지만, 사실은 데이터 준비 과정이 가장 오래 걸리고 그 다음으로 손실 함수 구현이 오래걸린다. 모델구현은 프레임워크에서 편리한 API를 제공하므로 어렵지 않다.

## 1.2. 강의 계획

- 강의계획
  - 1. 강의 소개, YOLO v3 모델 설명
  - 2. pyenv 설치 및 가상환경 세팅, 텐서플로 및 파이토치 환경 세팅
  - 3. (TF,PT) 간단한 분류 모델 테스트
  - 4. (TF) 간단한 데이터 셋에서 tfrecord만들고 데이터 불러오기
  - 5. (PT) 파이토치 DataLoader 사용
  - 6. (TF) TfreordMaker 구조 설계, Cityscapes reader 구현
  - 7. Kitti reader 구현, TfreordReader 구현
  - 8. (TF) 학습 시스템 설계, Darknet53 구현
  - 9. (TF) 검출 헤드(Head) 구현, 손실 함수 구현
  - 10. (TF) 로깅(logging) 및 평가 시스템 구현
  - 11. (PT) DatasetLoader 구현
  - 12. (PT) Darknet53과 검출 헤드 구현, 학습 시스템에 적용
  - 13. (PT) 손실함수 구현
  - 14. (PT) 로깅(logging) 및 평가 시스템 구현

tfrecord : 텐서플로 전용 dataset 파일

많은 파일들을 하나씩 읽어가지고 학습하는 모델일 경우 많은 파일들이 있어 하드디스크에서 파일을 하나씩읽어 오게 되어 결국 읽어오는 속도가 느려져 전체적으로 학습시간이 느려질 것임. 따라서 사용할 데이터들을 **하나의 data recode라는 파일**에 넣어두고 읽어오는 것이 더 빠르다.

학습할 수 있는 구조를 짜고 back본을 만들어 검출 헤드 구현, 손실함수 구현...등

logging이 중요하다. (딥러닝은 잘되도 왜 잘된지, 안되면 왜 안되는지 잘 모르는 것이 문제다.)

---

## 2. YOLO (You Only Look Onece)

---

검출 모델 구현이라는 목표달성을 위해 **YOLO** 모델을 선택함

- 여러 모델을 비교하며 구현하면 좋지만 시간이 오래걸리고 하나를 구현하고 나면 다른 모델 또한 쉽게 구현할 수 있을 것이다.

YOLO를 선택한 이유는 가장 유명한 Single-stage 검출 모델이기 때문이다. Two-stage 검출 모델로 Faster R-CNN 등이 있는데 일반적으로 Two-stage 모델이 성능이 조금 더 나온다고 하지만 학습과정을 두 번 거쳐야 하여 구현상 번거로운 점이 있다.

## 2.1. YOLO 란?

YOLO는 Object Detector 계열에서 가장 유명한 모델 중 하나이다.  
2021.02.04기준 구글 학술 검색에서의 인용수이다.

1. Faster R-CNN : 19,934
2. YOLO (v1) : 13,806
3. Fast R-CNN : 13,255
4. SSD : 12,695

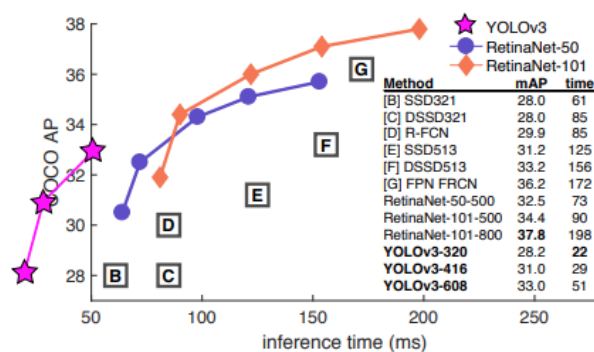
YOLO는 Joseph Redmon이라는 사람에 의해 개발되었다. [Joseph Redmon Homepage](#) (<< 이상한 사람)

남들은 텐서플로나 파이토치 등을 이용해서 개발하는 딥러닝 연구를 프레임워크부터 직접 만들어서 좋은 검출 모델까지 구현을 했다. (Nature 자매지를 능가하는 CVPR이라는 학회에서 최우수 논문상을 받기도 했다.)

Joseph에 의해 개발된 YOLO는 세 가지 버전이 있고 그이후 YOLO v4는 Joseph과는 관련이 없다.

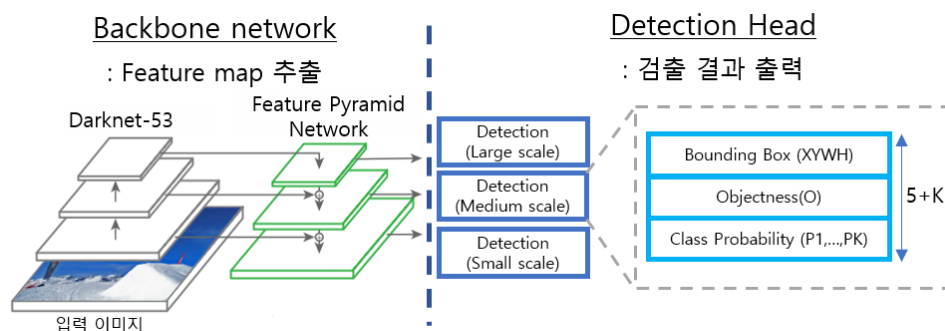
- Redmon, Joseph, et al. "You only look once: Unified, real-time object detection." *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016.
- Redmon, Joseph, and Ali Farhadi. "YOLO9000: better, faster, stronger." *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017.
- Redmon, Joseph, and Ali Farhadi. "Yolov3: An incremental improvement." *arXiv preprint arXiv:1804.02767* (2018).

YOLO모델의 장점은 **속도**다. 다른 RetinaNet에 비해 성능은 좀 떨어지지만 속도는 빠르다는 것을 볼 수 있으며 다른 모델과 비교해봐도 성능도 그다지 부족하지 않다.



## 2.2. YOLOv3 모델 구조

YOLO 모델은 크게 Backbone network(백본)와 Detection Head(헤드)로 구성이 되어있다. 백본에서는 영상의 다양한 정보를 포함한 Feature map을 만들고 헤드에서는 객체 정보를 추출한다.



백본에서는 이미지를 받아서 이미지에서 여러가지 특징을 추출하는 컨볼루션 레이어를 사용하여 피쳐맵을 만들어내고 Feature Pyramid network 상위 피쳐와 하위피쳐를 섞는 구조를 만들어 준다.

이 3가지 피쳐맵을 모두 사용하고 라지, 미디엄, 스몰 스켈일로 각각 사용되며 각각의 피쳐맵의 그리드 셀 마다 오른쪽과 같은 정보를 출력하게 된다.

Bounding Box(XYWH) : 사물(객체)을 감싸는 사각형 중심점의 좌표, 너비, 높이에 대한 정보

Objectness(O) : 물체가 있으면 1에 가까워지고 없다면 0에 가까워지는 사물을 판별하는 정보

Class Probability (P0, P1....Pk) : 어떤 사물인지 판별해주는 정보

그리드 셀 하나마다 3가지 앵커박스가 있어서 오른쪽 같은 정보가 3개씩 나온다. 최종적으로 출력의 채널 수는  $5+K$ 개가 된다.

## Backbone Network

YOLOv3에서 백본은 Darknet-53이라는 구조로 되어있다. ResNet과 비슷한 shortcut connection이 있어서 각 conv, block의 입력과 출력을 더해서 다음 레이어의 입력으로 들어간다.

Feature map을 1/2로 줄일때마다 채널을 두배로 늘린다. 마지막에 global average pooling은 분류에 사용되는 것이고 객체 검출을 할때는 필요하지 않다.

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

convolution 블럭과 아래쪽의 블럭의 출력이 더해지면서 그레디언트가 뒤에까지 잘 전파될 수 있도록 만듦

$256 - 128 - 64 - 32 - 16 - 8 \dots$  이런식으로 이미지의 해상도는 줄어드지만

$32 - 64 - 128 - 256 - 512 - 1024$  채널 수가 늘어난다.

중간에  $1 \times 1$  convolution 을 사용하는 이유는 연산을 줄이면서도 convolution을 여러번 해서 네트워크 전체적인 비선형성을 증가 시켜서 좀더 복잡한 문제를 해결할 수 있게 만들기 위함이다.

- $1 \times 1$  convolution을 사용하는 보통 피쳐맵을 압축하기 위하여 사용한다.

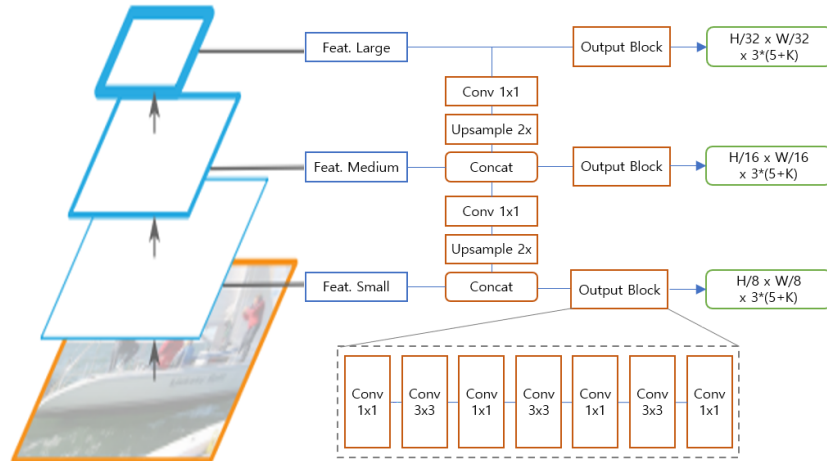
피쳐맵들이 많아도 전부다 사용하는 의미있는 피쳐맵이 아니기 때문에 압축과 확장을 반복하며 안좋은 정보는 빼내고 다양한 정보를 넣기 위해 사용한다.

밑에 있는 부분들은 최종적인 출력을 1000개로 한 이유는 이미지넷 프리 트레이닝 하게 만들기 위함이다.

디텍션을 사용할때는 Avgpool 부분은 사용하지 않으며 프리트레이닝을 할 때 사용된다.

## Detection head

Feature Pyramid network(FPN)구조는 "Feature Pyramid Networks for Object Detection"이란 논문에서 최초로 제안되었고 이후에 대부분의 검출 모델에서 이와 유사하거나 더 복잡한 구조가 사용되었다. YOLOv3의 자세한 구조는 다음과 같다.



Darknet에서는 large, medium, small 세 가지 스케일의 feature map을 출력한다. 여기서 주의할 점은 large feature map 혹은 large scale이라고 feature map 자체가 큰 것이 아니라 해당 feature map에서 검출되는 객체들의 크기가 크다는 뜻이다. Large feature map의 텐서 크기가 가장 작고 small feature map의 크기가 가장 크다.

상의 feature map은 1x1 conv를 거쳐 2배로 확장된 다음 하위 feature map과 합쳐진다. 이는 고수준의 의미를 포함한 상의 feature와 국소적 디테일을 포함한 하의 feature를 결합하여, 객체 검출에 다양한 수준의 feature를 활용하기 위함이다.

격합된 feature map은 몇 번의 convolution을 거쳐 feature map 형태로 최종 출력을 내보낸다.

출력 feature map의 크기는 large, vedium, small 각각 원래 이미지에 비해 1/32, 1/16, 1/8 크기다. 출력 feature map을 통해 해당 위치에 객체가 있는지 (objectness), 어느범위에 있는지(bounding box : XYWH), 어떤 종류의 객체인지(class probability, p1...pK)를 알려준다.

FC-layer를 사용하지 않고 convolution 연산만 사용했기 때문에 영상의 해상도가 늘어나도 같은 구조를 그대로 사용할 수 있다. 다만 학습시 Anchor box의 크기는 조절해야 한다.

FPN 구조는 해상도는 줄이고 채널을 늘리는 피쳐맵을 만든다.

마지막 피쳐맵에서만 바운딩 박스를 찾게 되면 작은 물체들을 못찾기 때문에 3개의 피쳐맵을 같이 사용한다.

ex : (13, 13) + (26, 26) + (52, 52)의 피쳐맵을 다 사용한다.

이런 멀티 스케일 피쳐맵을 사용하게 되는데 찾아야 하는 객체들의 크기가 모두 다르기 때문에 각각에 맞는 스케일을 사용하기 위해 사용한다.

Output Block이 있는데 여러개의 컨볼루션이 반복되었다.

라지 스케일 피쳐맵과 미디엄 스케일 피쳐맵이 합쳐지는데 1\*1컨볼루션을 진행 한 뒤 upsample 2x를 통하여 늘려서 concat을 진행한다. 이후 small 스케일 피쳐맵과 합쳐질때도 동일한 과정을 거친다.

출력의 차원은 피쳐맵의 크기 예다가 5+K의 채널이 나와야 하는데 각각의 그리드 셀마다 사용되는 앵커가 3개씩 있기 때문에 \* 3을 해준다.

large스케일은 검출되는 물체가 크기 때문에 라지스케일이라고 하는 것.

하이레벨 피쳐와 로우레벨 피쳐를 합쳐가지고 큰단위의 특징과 작은 단위의 특징도 잘 볼 수 있도록 피쳐를 섞는 과정을 가진다.

거의 모든 연산을 컨볼루션으로만 진행하므로 피쳐맵의 크기가 커지면 컨볼루션을 더하면 되기 때문에 해상도가 늘어나도 같은 구조를 사용할 수 있다.

## Anchor Box

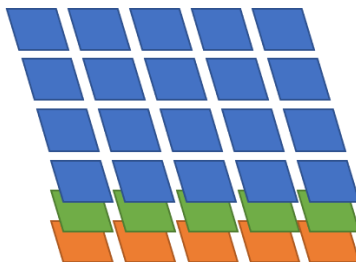
YOLO에서는 bounding box의 중심점과 너비 높이를 출력하는데 사전에 정한 anchor box를 기준으로 상대적인 위치와 크기 변화만을 학습한다. 최종 결과를 보여줄 때는 이를 픽셀좌표로 변환하여 보여준다.

YOLO구조에서 보이듯 YOLO는 large, medium, small 세가지 스케일로 검출 결과를 출력한다. Small feature map은 작은 물체에 대한 학습데이터를 할당받아 작은 물체를 검출하도록 학습하고 Large feature map은 큰 물체에 대한 학습데이터를 할당받아 큰 물체를 검출한다. 각 스케일의 feature map은 내부적으로 세 가지의 크기의 anchor box를 가져서 총 9가지의 anchor box가 있다. 아래는 각 anchor box의 픽셀 단위 크기다.

YOLO에서는 COCO 데이터셋에서 bounding box의 크기에 대해 K-means clustering 기법을 이용해 가장 빈번하게 사용되는 box 크기를 anchor box의 크기로 사용하였다.

```
[(10, 13), (16, 30), (33, 23), (30, 61), (62, 45), (59, 119), (116, 90), (156, 198), (373, 326)]
```

가령 medium feature map은 모델에 (416, 416) 영상을 입력했을때 (26, 26)해상도를 가진다. 각 픽셀마다 세가지 크기의 anchor box가 할당되므로 medium 스케일에서는 총  $26 * 26 * 3$  개의 bounding box를 출력할 수 있다. 다음그림은  $4 * 5 * 3$  feature map의 예시를 그린것이다.



박스의 위치와 너비 높이를 아웃풋에서 직접 출력되는게 아니고 아웃풋에서 나오는 것은 사전에 정의된 앵커 박스를 기준으로 하여 상대적으로 이동, 크기변화, 중심점의 이동 등 상대적인 정보만 출력이 되게 된다.

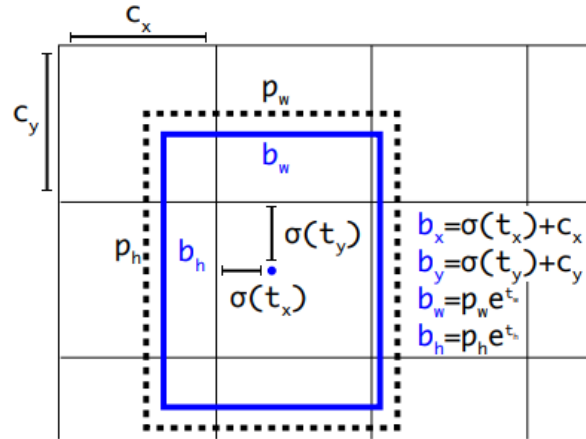
small 스케일은 작은 물체를 학습하도록 할당이 되어 학습이 되고 large 스케일은 큰 물체를 학습하도록 할당이 되어 학습이 진행 된다.

각각의 스케일에 3가지 앵커박스가 있으므로 총 9개의 앵커 박스에서 출력이 나올 수 있다.

각 앵커박스의 픽셀단위의 크기데이터에서 3개씩 나누어서 스몰, 미디엄, 라지 순의 바운딩 박스이다.

COCO dataset에서 K-means clustering 기법을 사용해서 많이 쓰이는 영역을 사용하였고 (데이터를 기반으로 설정, 다른곳에서는 직접 지정하기도 하였음) 416, 416으로 이미지를 축소시킨 다음에 입력을 하였을때 각각에 그리드셀 피쳐맵에 따라 앵커박스가 3개 출력이 된다. 최종적으로 스코어 값이 높은 값만 출력이 되는 것이다.

Feature map에서 바로 박스의 위치와 크기를 출력하지 않고 아래 그림과 같은 후처리 과정을 거쳐 박스 위치와 크기를 만든다.



- $t_x, t_y, t_w, t_h$  : feature map에서 바로 출력된 박스의 원시 정보
- $c_x, c_y$  : 각 그리드 셀(grid cell)의 좌표 (0, 0) ~ (GW-1, GH-1)
- $p_w, p_h$  : anchor box의 크기
- $b_x, b_y, b_w, b_h$  : 후처리를 통해 출력된 최종적인 bounding box 정보
  - $b_x, b_y$  : sigmoid 함수를 써서  $t_x, t_y$ 를 0~1 값으로 변환하여 cell 내부에서 박스의 중심점 위치를 표시, 거기에 cell의 좌표  $c_x, c_y$ 를 더하여 feature map 왼쪽 위 구석을 기준으로 한 좌표 출력
  - $b_w, b_h$  : exponential 함수를 써서 크기가 음수가 나오지 않게 함. 박스 크기를 직접 학습하지 않고 정해진 anchor box 크기  $p_w, p_h$ 에 대한 비율만 학습

위 그림의 수식에서는 모든 크기의 기준이 그리드 스케일로 수식이 적혀있다. (모든 좌표와 크기가 [0, 0] ~ [GW-1, GH-1] 범위) 하지만 실제 구현에서는 그리드의 크기가 feature map마다 다르고 그리드 크기에 따라 손실 함수의 크기도 달라질 수 있으므로 모든 좌표와 크기를 [0, 0] ~ [1, 1] 사이의 범위로 정규화하여 사용한다.

416크기 이미지를 기준으로 feature map의 크기는 'large'가 (13, 13), 'small'이 (52, 52)이므로 그리드 스케일로 좌표나 크기의 손실 함수를 구하면 'large'는 손실 함수가 작게 나오고 'small'은 크게 나와서 균형있게 학습할 수 없다. 모든 좌표와 크기를 0 ~ 1 범위로 정규화하면 어떤 feature map이든 동일한 비중으로 학습할 수 있고 나중에 다시 영상에 bounding box를 표시할 때도 원본 영상의 크기만 곱해주면 된다.

오른쪽에  $b_x, b_y$ 를 구하는 식에 붙어있는 건 시그모이드 함수이다.

즉 시그모이드( $t_x$ ) +  $c_x$ , 시그모이드( $t_y$ ) +  $c_y$  이다.

0~1사이의 수로 정규화 하기위해 사용된다.

시그모이드를 씌움으로 인해서 그리드 셀을 넘어가지 않게 된다.

$b_w, b_h$ 또한  $h, w$ 값을 어떤 값이 나오더라도 항상 양수가 나오도록 할 수 있는 Exponential(지수) 함수를 사용하여 계산해준다.

$t_w, t_h$ 가 작은값(음수)이면은 앵커박스에 비해 작은 박스라고 할 수 있고 양수면은 원래 앵커박스보다 큰 박스가 나올 것이다.

이 과정을 통해 크기와 위치의 변화를 출력을 하여 bounding box를 만들어 내게 된다.

모든 박스들은 그리드 셀을 기준으로 계산을 하여 box를 구하게 된다.

하지만 실제로 이미지에 바운딩 박스를 그리더라도 그리드 셀, 피쳐맵이 원본 이미지보다는 작기 때문에  $b_w, b_h, b_x, b_y$ 를 이미지의 크기에 맞춰서 스케일링 하여 늘려주거나 해야 원본 이미지에 맞춰서 바운딩 박스를 그릴 수 있는데 피쳐맵의 크기가 여러가지가 나올 수 있고, 3가지 스케일이 나올 수 있는 문제, 입력 사이즈가 달라지면 피쳐맵의 크기도 달라지기 때문에 여러가지 크기에 작업을 하기 힘들다.

따라서 bx, by같은 그리드 셀의 스케일 보다는 전체 범위를 0과 1 사이에 넣어가지고 0 ~ 1 scale로 바운딩 박스를 스케일링 해주면은 다른 스케일이라고 따로 처리해줄 필요가 없고 그냥 이미지의 크기만 곱해주면 된다.


원본이미지의 픽셀단위의 바운딩 박스를 그릴 수 있게 되고, 피쳐맵의 크기에 따라서 로스값의 전체적인 크기가 달라지거나 하는 것을 막을 수 있다.

라지 스케일 - 피쳐맵 작음, 스몰 스케일 - 피쳐맵 큼에 따라 로스값이 달라지면 라지스케일은 학습이 덜 되는 것이기 때문에 loss값의 크기가 비슷해야 골고루 학습이 되기 때문에 0~ 1사이의 값으로 만들어 줘야 좋다.

## 2.3. 학습 및 예측

### IoU (Intersection over Union)

IoU는 검출 모델에서 자주 사용되는 용어인데 두 개의 bounding box 사이의 겹치는 부분의 비율이다. 보통 실제 사물의 범위를 나타내는 GT(ground truth) box가 있고 검출 모델에서 출력한 predicted box가 있을 때 둘 사이에 IoU를 계산하여 검출이 잘 되었는지를 판단한다. 보통 이 비율이 50%가 넘어야 검출이 제대로 되었다고 본다. IoU 계산 방법은 아래 그림을 보면 직관적으로 이해할 수 있다. IoU는 두 박스를 합친 영역 대비 중복 영역의 넓이의 비율이다.

$$\text{IoU} = \frac{\text{Area of Overlap}}{\text{Area of Union}}$$


IoU 원래 나와야 하는 박스와 모델에서 예측한 박스가 얼마나 정확한지 알기 위한 지표 즉 예측하여 나온 결과와 GT(정답)값의 박스가 어느정도 일치해있는지 판단하는 방법.

보통 50%이상은 일치되어야 검출이 제대로 되었다고 한다.

(사람이 보기에는 30%이상만 되어도 위치가 삐딱하긴 해도 대~강 맞다고 보인다)

## 학습(Training)

모델을 학습시킬 때는 학습 데이터에 들어있는 각 GT box에 대해 어느 위치의 어떤 크기의 anchor에서 학습시킬 것인지를 미리 정해야 한다. 다른 모델에서는 무작위로 bounding box를 출력한 뒤 그 중에서 GT box와 가장 IoU가 높은 box에 학습 데이터를 할당하기도 한다. YOLO에서는 GT box의 위치와 크기를 이용하여 정확한 계산으로 어느 feature map의 어느 위치의 어떤 anchor box에 학습 데이터를 할당할지를 결정한다. 모든 feature map의 모든 anchor box에서 박스 예측 결과가 나오지만 학습에 사용되는 박스는 GT box의 개수와 같다.

학습에는 다음과 같은 손실함수가 사용된다.

- Objectness : anchor box에 사물이 있는지 없는지에 대한 확률로 표시 (0 ~ 1의 값이 나옴)
- Bounding box : GT와 모델 예측 사이의 (XYHW) 차이



- Category : Multi-labeling이 가능하도록 각각의 클래스에 대해 "있다, 없다"를 binary crossentropy로 학습

모든 스케일을 다 학습시키는 것이 아닌 실제로 물체가 발견된 곳에서만 학습을 하게 되는데 물체가 하나라도 스케일이 3개이므로 3가지 위치가 나올 텐데 그중에서 물체의 크기에 맞는 스케일에 학습을 할당한다.

그리드와 똑같은 크기의 텐서를 만들어가지고 데이터가 있는 범위만 의미있는 값을 넣어 놓고 나머지는 0으로 만들어서 만들어 놓으면 좀더 편리하게 계산을 할 수 있다.

피쳐맵과 똑같은 모양을 만들고 내가 가지고 있는 학습데이터 GT를 피쳐맵에다가 하나씩 할당을 하여 학습데이터를 만들어야 한다는 것이다.

조건을 가지고 한두개씩 처리하는 것 보다는 전체영역에 대해 똑같은 연산을 동시에 수행하는 것이 더 좋다. (GPU가 계산하는 효율이...)

**텐서 연산은 텐서끼리 한번에 연산하는 것이 더 간단하고 좋다.**

Objectness : 0~1사이로 나오는 값을 binary crossentropy값의 loss값을 구해 계산을 시켜 학습을 진행  
Bounding box : GT와 프레딕션 사이의 차이를 가지고 차이가 줄어들도록 학습

Category : 어느 종류의 객체인지 분류하는 classification loss를 사용하여 binary crossentropy로 학습을 진행.

만약 10가지의 종류가 있다면 10에 출력에 대해서 소프트 맥스와 크로스 엔트로피를 계산하지만 yolo에서는 각각 따로따로 하게 만들었다.

즉 차냐 사람이냐 가 아니라 차인가 아닌가, 사람인가 아닌가, 의 식으로 binary crossentropy를 여러개 사용하여 classification을 진행한다.

## 예측(Prediction) feat. NMS

학습 후 검출 결과를 출력할 때는 모든 feature map에서 나오는 bounding box를 출력하지 않고 검출된 객체만 출력한다. 일단 각 anchor box위치에서 objectness > t\_obj일때 해당 위치에서 출력되는 bounding box에 객체가 있다고 볼 수 있는데 objectness만 볼 경우 비슷한 위치에서 다수의 객체가 중복으로 검출되는 문제가 발생한다. 객체 하나에 검출 결과를 하나만 출력하기 위해서는 NMS(Non-Maximum Suppression)를 써서 중복된 객체들 중 가장 score가 높은 것만 출력하도록 한다.

객체의 존재 여부를 판단하는 score는 (objectness \* class prob)을 사용한다. 즉 어떤 anchor 위치에서 objectness가 0.8, person 클래스에 대한 확률이 0.6이라면 그 위치에서 사람 검출에 대한 score는 0.48이다.

객체가 있는 것들만 걸러져야 하는데 앵커박스의 위치가 objectness 보다 스레스 홀드가 높을때 거기에 물체가 있을 수가 있다 고 생각한다. 물체가 있으면은 그 물체 근처의 피쳐맵에서 해당 물체를 찾으려고 박스를 출력한다.

물체가 하나 있어도 objectness 높게 반응하는 앵커박스가 동시에 여러개가 나올 수 있다.

그것들 중에 가장 잘 된것만 뽑고 나머지는 버리기 위해 **중복된 객체들 중 가장 score가 높은 것만 출력하도록 하는 NMS를 사용한다.**

NMS 알고리즘은 다음과 같다. 이것은 하나의 feature map에 대해 nms를 하는것이고 YOLO에서는 세 가지 스케일의 feature map이 있으므로 이를 세 번 반복해야 한다.

```
def nms(bboxes, scores, score_thresh, iou_thresh):
    """Non-Maximum suppression.
    bboxes: numpy array [N, 4(xywh)]
    scores: numpy array [N, K] (N=num boxes, K=num categories)
```

```

    score_thresh: a box can be a detection candidate if its score is larger than
    score_thresh
    iou_thresh: two boxes are considered overlapping if their IOU is larger than
    iou_thresh
    """
    N, K = scores.shape
    # sort box indices in decending order, [N, K]
    order = scores.argsort(axis=0)[::-1]
    # ignore boxes with score < score_thresh
    scores[scores < score_thresh] = 0

    for k in range(K): # for each category
        for i in range(N-1): # for each highest score box
            if scores[order[i,k], k] == 0: # skip ignored box
                continue
            # iou between i-th rank box and lower rank boxes [N-i-1]
            ious = batch_iou(boxes[order[i,k]], boxes[order[i+1:,k]])
            # ignore boxes with iou > iou_thresh
            for j, iou in enumerate(ious):
                if iou > iou_thresh:
                    scores[order[j+i+1,k], k] = 0

    # box category [N], maximum category scores per box [N]
    box_categories = scores.argmax(axis=1)
    max_score_per_box = scores.max(axis=1)
    # remove ignored box info [M]
    box_categories = box_categories[max_score_per_box > 0]
    boxes = boxes[max_score_per_box > 0]
    return boxes, box_categories

```

```

def nms(boxes, scores, score_thresh, iou_thresh):
    """ Non-Maximum supression.
        boxes: numpy array [N, 4(xywh)]
        scores: numpy array [N, K] (N=num boxes, K=num categories)
        score_thresh: a box can be a detection candidate if its score is larger than
        score_thresh
        iou_thresh: two boxes are considered overlapping if their IOU is larger than
        iou_thresh
    """

```

boxes에서 N은 전체 앵커박스의 갯수 ex) 4 \* 5 \* 3 feature map 이라고 하면 60 개의 박스가 나온다고 할 수 있다.

scores 의 N개의 박스가 K의 score을 가지고 있다

score\_thresh는 스코어가 쓰레시 홀드보다 낮은 것들을 걸러주는 단계이다.

스코어가 높은들 중에서 중복되는 것을 걸러주는게 (score가 가장 높은 것을 걸러주는 것이) NMS  
IoU는 여러개의 물체들을 겹쳐있으면 중복되는 것을 제거해 줄 때 얼마나 겹쳐있으면 볼 것인지에  
대한 기준 값.(두개의 박스가 중복되어 있을 때 얼마나 겹쳐있다고 판단을 할것인지 에 대한...?)

"""

```

    N, K = scores.shape
    # sort box indices in decending order, [N, K]
    order = scores.argsort(axis=0)[::-1]
    """

```

argsort : index를 정렬하는 함수.

```
> argsort(axis=0)[::-1]
```

> 여기서 argsort를 하면 0 1 2 순으로 정렬되지만 [::-1]이 있어 2 1 0 순으로 정렬된다.

이렇게 해주는 이유는 **score**가 높은 것을 찾아야 하기 때문에 순서를 뒤집기 위함이다. (가장 높은게 인덱스가 가장 뒤에 할당되어있음.)

```
.....

score
0.2 0.6
0.5 0.3
0.7 0.4

order
2 0
1 2
0 1
.....

"""
    # ignore boxes with score < score_thresh
    scores[scores < score_thresh] = 0

# threshold보다 작은 것을 0처리 해준다. 여기에서 0.5라고 한다면 0.3, 같은 0.5보다 작은 값
은 다 0으로 만들어 준다.

    for k in range(K): # for each category
"""
카테고리를 무시하고 계산을 한다면 두개가 겹쳐있다고 해서 둘중에 하나만 뭉뚱거릴 수 있으니 카
테고리 별로 구분을 해주어서 NMS를 해줘야 한다.
NMS가 자신과 겹치는지 확인해서 어떤것이 score가 높은지 알아야 하므로 n:n으로 매칭을 해주어
야 한다.
즉 각각의 박스마다 for문을 돌아야한다.
"""

    for i in range(N-1): # for each highest score box
        if scores[order[i,k], k] == 0: # skip ignored box
            continue
        # score가 0인 항목들은 바로 pass해도 된다는 뜻이다.
        # i가 아닌 order을 넣은 이유는 score가 높은 것부터 검사하기 위하여 order을 넣어
        준 것이다.
        # 즉 i번째로 score가 높은 i의 score을 계산하게 되는 것이다.

        # iou between i-th rank box and lower rank boxes [N-i-1]
        ious = batch_iou(boxes[order[i,k]], boxes[order[i+1:,k]])
"""
IoU를 계산하게 되는데 k번째 카테고리에서 i번째 스코어가 높은 값의 box와 스코어가 낮은 나머
지 박스들과 일관적으로 IoU를 계산을 하는 것이다. (for문_)
"""

        # ignore boxes with iou > iou_thresh
        for j, iou in enumerate(ious):
            if iou > iou_thresh:
                scores[order[j+i+1,k], k] = 0

"""
스코어가 높은 박스들 부터 처리를 하고 있으므로 IoU가 높은 것이 있다면은 그 박스의 값을 0으로
만들어 버리는 것이다.
즉 높은 score을 가진 박스를 가지고 다른 박스들과 비교해 본뒤 IoU가 높다면 겹치는것이 많은
의미이기 때문에 스코어를 지워 score값이 높은 1가지 박스만 나오도록 해주는 작업이다
"""
```

```
# box category [N], maximum category scores per box [N]
box_categories = scores.argmax(axis=1)
max_score_per_box = scores.max(axis=1)
"""
```

위 과정을 통하여 대부분 0으로 만들어버린 **score map**을 가지고 **argmax**를 하면 **box** 카테고리가 나오게 된다.

**axis = 1** (가로차원이) 카테고리를 의미하므로 카테고리를 거를 수 있다.

박스 카테고리를 구하고 **max**값 자체도 구하는 작업을 진행한다.

```
"""
```

```
# remove ignored box info [M]
box_categories = box_categories[max_score_per_box > 0]
boxes = boxes[max_score_per_box > 0]
return boxes, box_categories
"""
```

맥스 스코어 0인것들을 지워버리는 작업이다.

즉 박스 안에 모든 스코어가 0이되는 경우를 제거해주는 작업이다.

박스과 박스의 카테고리가(인덱스) 출력이 된다.

```
"""
```

---