머신러닝 프로그래밍 최종 프로젝트

< Yolov3를 활용한 객체인식 >

2021254018 김원우

목 차

1. 연구 배경

1-1 주제 선정 배경

> 자율주행 기술

최근 테슬라의 완전자율주행 출시가 또 미뤄졌다. 오토파일럿 관련 사고가 잇따르면서 아직까진 5단계의 완전자율주행자동차의 등장은 거리가 멀다는 의견이 많이 나오고 있다.
하지만 자율주행 기술은 그만큼 유망한 기술이며 발전가능성도 높은 기술인 것은 확실하다.

자율주행 기술 중, 가장 중요한 항목 중 하나는 주행 환경 인식이다. 특히 시각 정보의 처리가 가장 큰 비중을 차지하고 있다.
시각 정보에서는 객체를 탐지하고, 이를 바탕으로 이동 경로를 설계하는 기술들의 시작이 바로 객체 인식 기술이다. 본 프로젝트에서는 객체 인식 기술을 구현해보는 것을 주제로 선정하였다.

현재 이미지로부터 사물을 탐지하는 딥러닝 모델은 다양한 알고리즘으로 개발되었고, 누구나 다운받아서 사용할 수 있도록 제공되고 있다. 많은 모델 중 Yolov3 라는 모델을 사용하였다.

1-2 Yolo 란

> Yolov3

Yolo란 You Only Look Once의 약자로 2011년에 현재 자신의 행복을 가장 중시하고 소비하는 사람들의 생활 태도를 뜻하는 말로 유명해졌다. 하지만 인공지능 분야에서는 최첨단 실시간 객체 탐지 시스템의 이름으로 사용하고 있으며, Yolov3 모델은 많은 Yolo 버전 중 하나이다.
Yolov3 는 상당히 빠르고 정확한 모델이라고 평가 받고 있으며, 일반적으로 한 이미지에서 3번 객체인식을 진행한다. (모델 구조는 연구 과정에서 설명)
재교육 없이 모델 크기만 변경해도 속도와 정확성을 쉽게 바꿀 수 있는 장점이 있다.

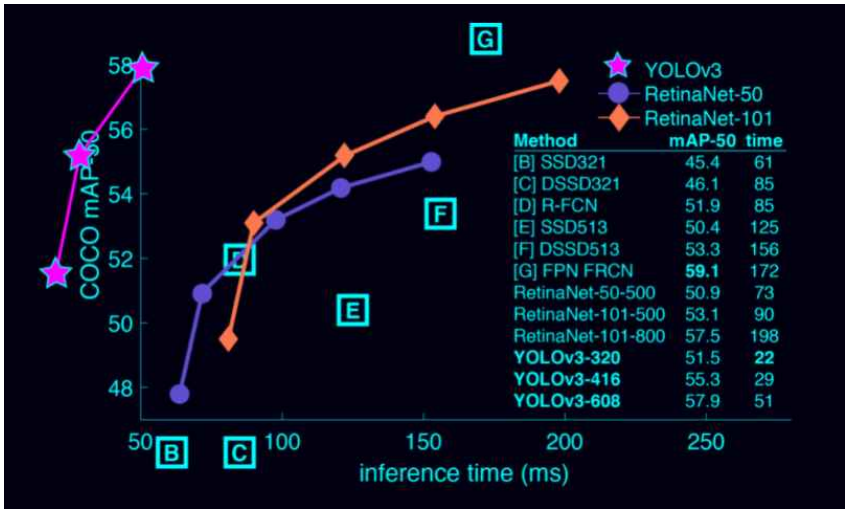


그림 1 모델 성능 비교

여러 모델들의 시간과 정확성을 비교한 그래프이다.
Yolov3 그래프가 축을 넘어가 있는데, 이는 오기가 아니라 실제로 축을 넘어갈 정도의 성능을 나타낸다.

2. 목표

2-1 yolov3 모델 분석 및 최적화

yolov3 모델 구조를 분석해보고 가지고 있는 데이터에 맞게 수정 및 최적화 한다.
최적화 시킨 모델을 직접 학습시킨 후, 모델 구조가 담겨있는 cfg 파일과 학습이 완료된 가중치 정보를 가지고 있는 weights 파일만을 사용하여 객체 인식을 수행한다.

2-2 소요시간 축소

자율주행 기술은 현재 일반적으로 차량에 탑재되어 사용된다. 때문에 객체를 인식하는 작업은 임베디드 시스템 환경에서도 돌아가야 할 것으로 생각된다. 가능한 빠른 시간 안에 1장의 이미지를 처리하는 방법을 고안한다.

2-3 최적의 객체 인식

주로 사용하는 환경, 장소, 카메라에 적합한 객체 인식 모델을 설계해 보는 것을 목표로 한다.
때문에 학습에 사용할 이미지 중 일부는 직접 도로에서 수집하였으며, 기존 장비가 학습하였던 이미지를 사용하였다.

도로 위의 주요 객체인 차량과 사람을 더 잘 인식하는 모델을 설계한다.

3. 연구 과정

coco 데이터셋과 보유하고 있는 데이터 중에서 선별한 300장의 이미지를 학습에 사용하였다. 추가로 사용한 300장은 각각 차량과 사람의 이미지를 가지고 있다.
각 이미지 별로 직접 라벨링 한 후, 학습시킨다. 학습이 완료되면 가중치를 담고 있는 weights 파일이 생성 되는데 이를 객체 인식에 사용하였다

3-1 학습 방법 및 과정

1) 학습 데이터 이미지 라벨링

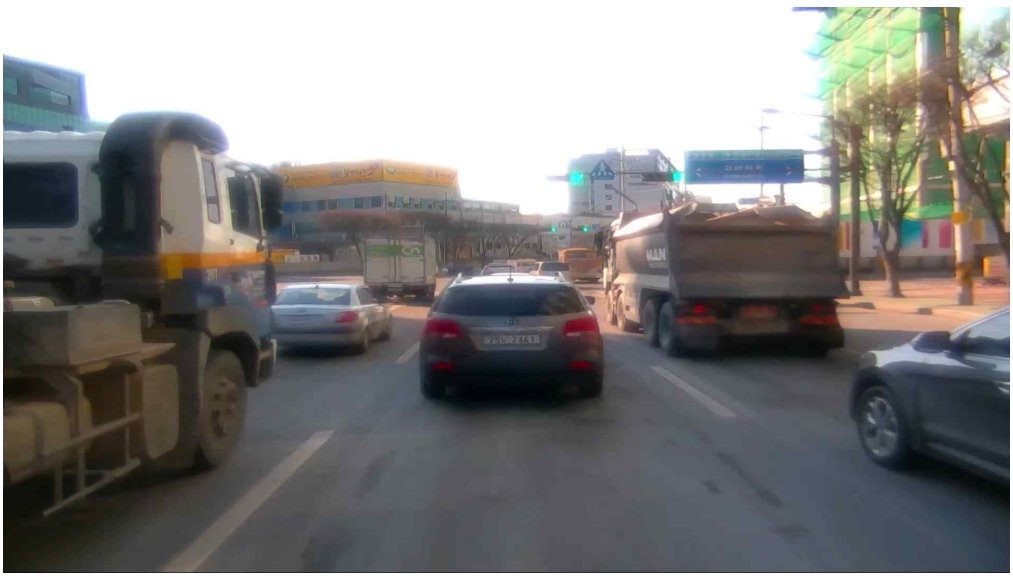


그림 2 학습 이미지-1

그림 3 학습 이미지-2

학습에 사용한 이미지의 예시이다. 특히 300장의 이미지는 사용 환경에 맞춰서 선별하였는데 이 이미지들은 라벨링이 추가로 필요하다.
이미지를 학습하기 위해서는 식별하고자 하는 객체를 지정해 줄 필요가 있었다.
Yolo mark 라는 도구를 사용하여 객체의 라벨을 지정하였다.

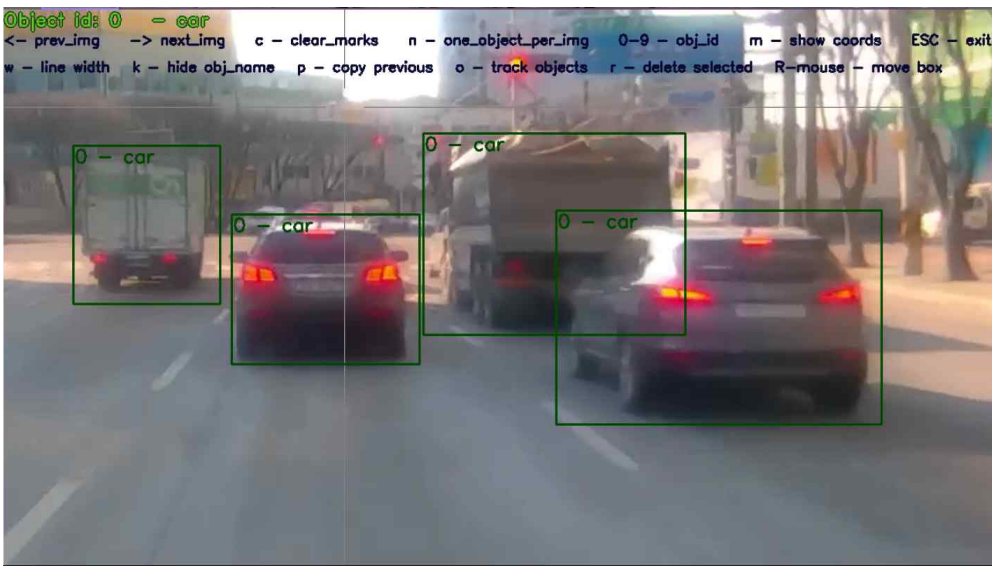


그림 4 학습 이미지 라벨링(차량)

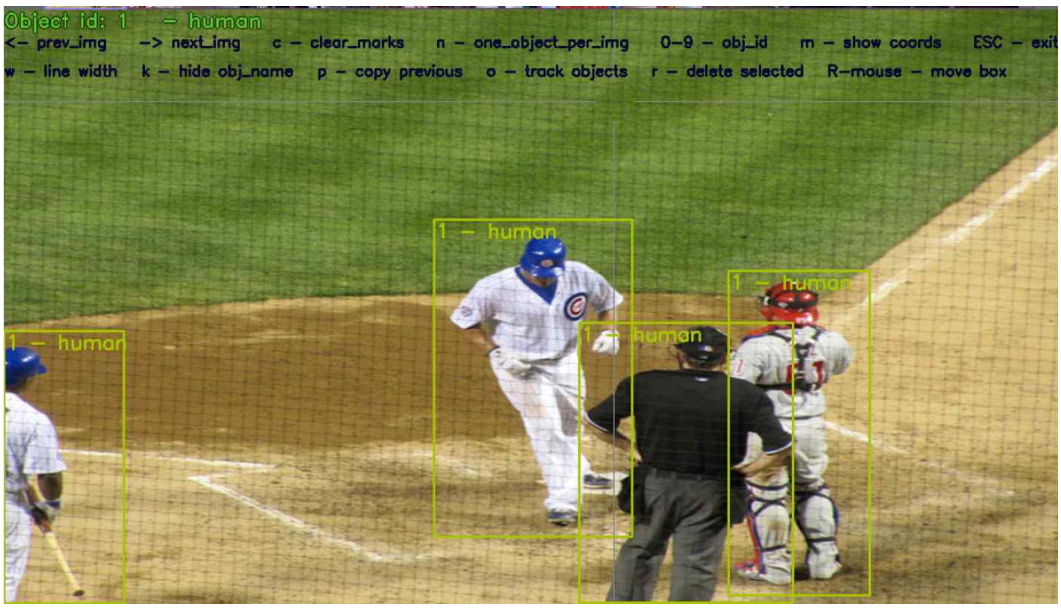


그림 5 학습 이미지 라벨링(사람)

각 이미지 별로 객체로 인식하고자 하는 부분에 직접 박스를 그렸다.
박스를 그리게 되면 각 꼭지점의 좌표와 class를 담은 txt 파일이 생성된다.

1) obj.names

```
1   human
2   bicycle
3   car
.
.
.
76  vase
77  scissors
78  teddy bear
79  hair drier
80  toothbrush
```

그림 6 객체 이름 파일

분류하고자 하는 객체의 이름을 가지고 있는 파일이다.
총 80개 객체의 이름이 저장되어 있다.

2) obj.data

```
1  classes= 80
2  train  = data/train.txt
3  #valid  = data/train.txt
4  names = data/obj.names
5  backup = backup/
```

그림 7 obj.data

학습 시 훈련 데이터의 위치와 class 개수, class 이름(human, bicycle, human 등등)을 가지고 있는 파일이다. backup 에는 학습이 진행되면서 가중치를 가지고 있는 weights 파일이 저장된다

3) cfg
cfg 란 모델의 구조를 블록 형식으로 가지고 있는 파일이다.

- 모델 구조

```
layer     filters    size              input                output
    0 conv     16  3 x 3 / 1   640 x 480 x   3   ->   640 x 480 x  16  0.265 BFLOPs
    1 max          2 x 2 / 2   640 x 480 x  16   ->   320 x 240 x  16
    2 conv     32  3 x 3 / 1   320 x 240 x  16   ->   320 x 240 x  32  0.708 BFLOPs
    3 max          2 x 2 / 2   320 x 240 x  32   ->   160 x 120 x  32
    4 conv     64  3 x 3 / 1   160 x 120 x  32   ->   160 x 120 x  64  0.708 BFLOPs
    5 max          2 x 2 / 2   160 x 120 x  64   ->    80 x  60 x  64
    6 conv    128  3 x 3 / 1    80 x  60 x  64   ->    80 x  60 x 128  0.708 BFLOPs
    7 max          2 x 2 / 2    80 x  60 x 128   ->    40 x  30 x 128
    8 conv    256  3 x 3 / 1    40 x  30 x 128   ->    40 x  30 x 256  0.708 BFLOPs
    9 max          2 x 2 / 2    40 x  30 x 256   ->    20 x  15 x 256
   10 conv    512  3 x 3 / 1    20 x  15 x 256   ->    20 x  15 x 512  0.708 BFLOPs
   11 max          2 x 2 / 1    20 x  15 x 512   ->    20 x  15 x 512
   12 conv   1024  3 x 3 / 1    20 x  15 x 512   ->    20 x  15 x1024  2.831 BFLOPs
   13 conv    256  1 x 1 / 1    20 x  15 x1024   ->    20 x  15 x 256  0.157 BFLOPs
   14 conv    512  3 x 3 / 1    20 x  15 x 256   ->    20 x  15 x 512  0.708 BFLOPs
   15 conv    255  1 x 1 / 1    20 x  15 x 512   ->    20 x  15 x 255  0.078 BFLOPs
   16 yolo
   17 route  13
   18 conv    128  1 x 1 / 1    20 x  15 x 256   ->    20 x  15 x 128  0.020 BFLOPs
   19 upsample           2x    20 x  15 x 128   ->    40 x  30 x 128
   20 route  19 8
   21 conv    256  3 x 3 / 1    40 x  30 x 384   ->    40 x  30 x 256  2.123 BFLOPs
   22 conv    255  1 x 1 / 1    40 x  30 x 256   ->    40 x  30 x 255  0.157 BFLOPs
   23 yolo
```

그림 8 학습 모델 구조

layer 순으로 설계 되어있는 모델이며, size의 / 부분은 stride를 의미한다.
conv 연산을 수행할 때 filters 의 개수만큼 output이 잘 출력되는 것을 알 수 있으며, yolo 수행 전 output의 filters 개수는 255이 들어간다. (classes 가 80개이기 때문)
max 는 max pooling을 의미하며, 16, 23 layer인 yolo layer에서 객체를 인식하게 된다
기본적으로 yolov3 모델은 yolo layer가 3번 들어가 있고 conv layer의 수도 보다 많지만, 수행시간을 줄이기 위한 경량화 모델을 사용하였다.

3-2 객체 인식 수행

1) 수행 방법
회사 내에서 사용되는 이미지 데이터 셋을 사용하여 객체 인식을 수행해보았다.
데이터 셋의 이미지는 640*480 사이즈이며, 사내 특정 용도로 인해 동일한 장면을 Cmos 카메라와 열상 카메라로 촬영한 이미지들로 구성되어 있다.

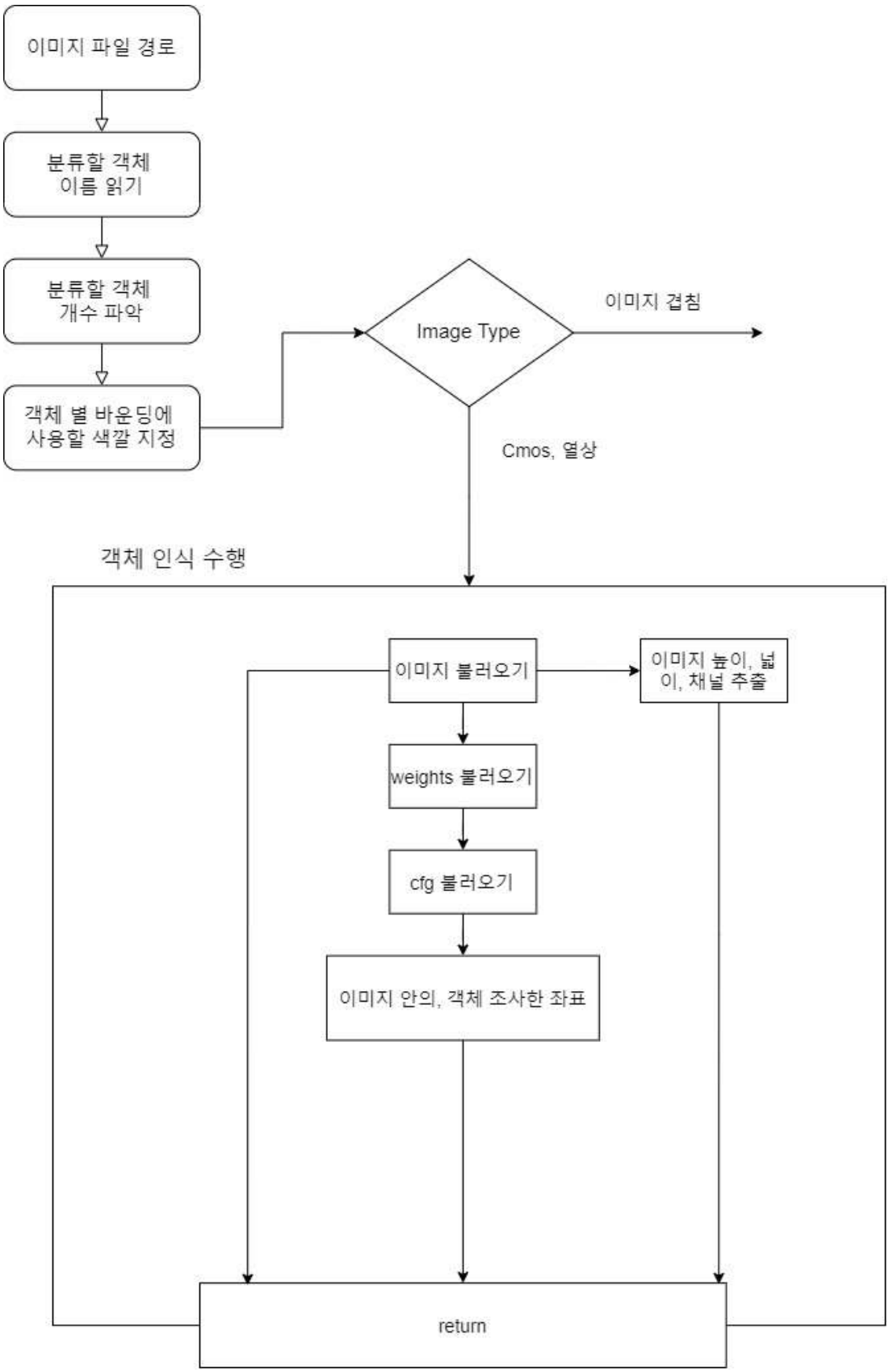
그림 9 Cmos 결과 이미지

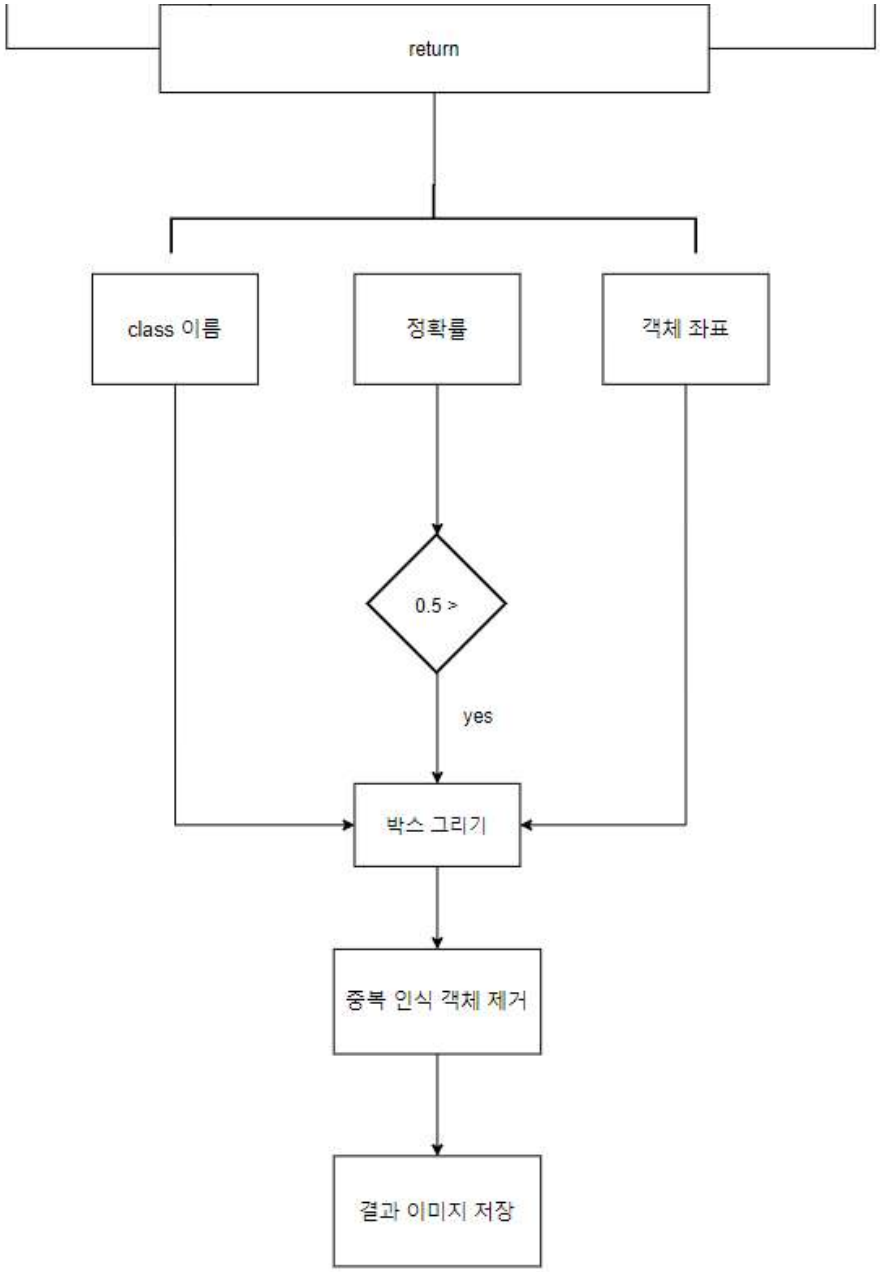
그림 10 열상 결과 이미지

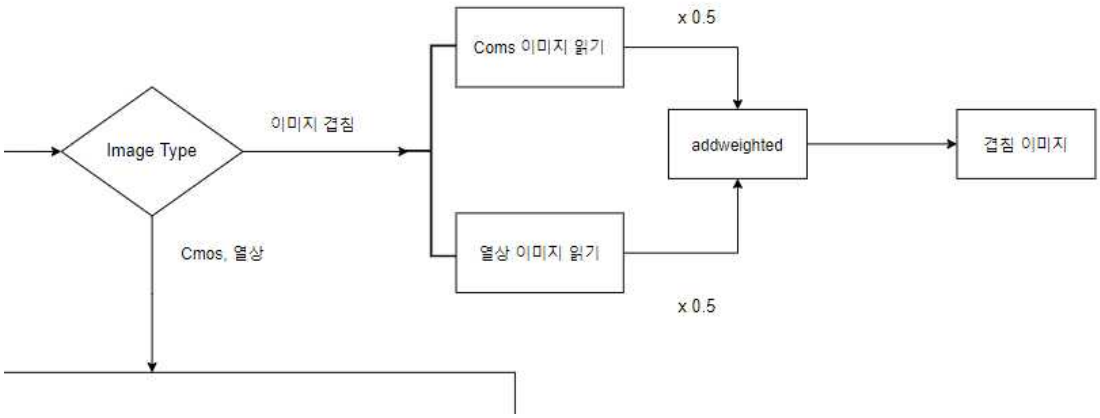
위의 그림들은 동일한 장면의 Cmos와 열상 이미지를 각각 객체 인식을 수행한 결과이다.
Cmos 이미지를 객체 인식 한 결과 자동차만 인식이 되었으며, 열상 이미지는 자전거만 인식이 된 경우이다.
초기 계획은 Cmos 이미지만을 사용하여 본 프로젝트를 진행하고자 하였으나, 호기심으로 데이터 셋에 있는 열상 이미지로도 객체 인식을 해보았는데 위와 같은 결과를 얻게 되었다.
데이터 셋을 최대한 활용하고자 2가지 종류의 이미지를 모두 사용하였고, 각각의 결과 이미지를 겹쳐서 최종 결과 이미지를 출력하였다.

그림 11 최종 결과 이미지

2) Flowchart


이미지 파일 경로
분류할 객체 이름 읽기
분류할 객체 개수 파악
객체 별 바운딩에 사용할 색깔 지정
Image Type
이미지 겹침
Cmos, 열상
객체 인식 수행
이미지 불러오기
이미지 높이, 넓이, 채널 추출
weights 불러오기
cfg 불러오기
이미지 안의, 객체 조사한 좌표
return


return
class 이름
정확률
객체 좌표
0.5 >
yes
박스 그리기
중복 인식 객체 제거
결과 이미지 저장


Coms 이미지 읽기
x 0.5
Image Type
이미지 겹침
addweighted
겹침 이미지
Cmos, 열상
열상 이미지 읽기
x 0.5

3) 코드 구조

객체 인식을 위한 코드는 교재의 예제를 참고하여 작성하였다. (311page, 프로그램 6-12)

```
import numpy as np
import cv2
import timeit
from openpyxl import load_workbook
```

필요한 라이브러리를 import 하는 부분이다
numpy 는 랜덤변수를 활용하고자 사용하였으며,
cv2 는 이미지 불러오기, 사이즈 변환, 학습된 모델 적용을 위해 import 하였다.
timeit 은 실행시간을 측정하는데 사용되었다.
또한 많은 이미지들을 효과적으로 테스트해보기 위해서 이미지 데이터 셋의 위치와 이름이 기입된 엑셀파일을 활용하였다.

```
def preprocessing(num, q):
```

함수 실행 부분

```
    exdata = load_ws.cell(num, 1).value

    if q == 0:
        img_path = 'D:/AI/EO/' + str(exdata[6:]) + '.png'
    else:
        img_path = 'D:/AI/IR/' + str(exdata[6:13]) + 'IR.png'

    img = cv2.imread(img_path)
    img = cv2.resize(img, dsize=(640, 480), interpolation=cv2.INTER_AREA)

    height, width, channels = img.shape
```

엑셀 파일을 활용하여 이미지를 불러온다.
불러온 이미지의 사이즈를 640*480 으로 설정한다.
그 후 이미지의 높이, 넓이, 채널을 저장한다.

```
blob = cv2.dnn.blobFromImage(img, 1.0 / 256, (640, 480), (0, 0, 0), swapRB=True, crop=False)

yolo_model = cv2.dnn.readNet('D:/yolov3/yolov3-tiny.weights', 'D:/yolov3/1/tiny.cfg')
layer_names = yolo_model.getLayerNames()
out_layers = [layer_names[i[0] - 1] for i in yolo_model.getUnconnectedOutLayers()]
```

blobFromImage 함수를 이용해 전처리를 수행한다. 이미지의 값을 0~1 로 정규화하고, BGR로 되어있는 영상 포맷을 RGB로 바꾼다.
그 후 weights 파일과 cfg 파일을 불러온다.
출력을 담당하는 yolo 층을 알아낸다. (yolo 층은 2개)

```
yolo_model.setInput(blob)
output3 = yolo_model.forward(out_layers)

return output3, height, width, channels, img
```

모델에 전처리 이미지를 입력하고 그 출력 값을 output3에 저장한다.
output3 와 이미지 정보를 리턴한다

```
filename = 'D:/AI/image_data.xlsx'

load_wb = load_workbook(filename, data_only=True)
load_ws = load_wb['sheet1']

f=open('D:/yolov3/1/names.txt', 'r')

classes = [line.strip() for line in f.readlines()]

colors=np.random.uniform(0,255,size=(80,3))
time0 = timeit.default_timer()

index_img = 2 # 2 ~ 1607
step = [0,1,2]
```

이미지 데이터 셋의 경로와 이미지 파일 이름이 기입되어 있는 엑셀 파일을 활용한다.
names.txt 파일에는 분류하고자 하는 사물들의 이름이 기입되어 있다.
분류하고자 하는 객체의 이름을 classes 리스트에 저장한다
colors 변수에는 80가지의 색깔을 랜덤으로 설정한다
time0 변수에는 timeit 기능을 사용하여 현재의 시간을 저장한다
index_img 는 이미지 데이터 셋의 개수로 Cmos와 열상 각각 1607 장씩 사용할 수 있다
step 은 Cmos 이미지의 객체인식 수행, 열상 이미지의 객체인식 수행, 2 가지의 결과를 겹치는 작업 등을 구별하는 역할을 수행한다.

```
for st in step:
    for num in range(1,index_img):

        if st == 2:
            img1 = cv2.imread('D:/yolov3/1/EO/EO_r' + str(num) + '.png')
            img2 = cv2.imread('D:/yolov3/1/IR/IR_r' + str(num) + '.png')

            a = 0.5
            b = 1.0 - a
            dst = cv2.addWeighted(img1, a, img2, b, 0)
            cv2.imwrite('D:/yolov3/1/RESULT/RESULT_r' + str(num) + '.png', dst)
```

step 이 2라면 Cmos 이미지와 열상이미지의 결과를 겹치는 작업을 수행한다.

겹치는 작업은 addWeighted 함수가 수행하며 각각 이미지의 투명도를 50%로 설정하고 더하는 방법으로 이미지를 겹친다. 그 후 결과 이미지를 파일로 저장한다.

```
    else:
        output3, height, width, channels, img = preprocessing(num, st)

        class_ids,confidences,boxes=[],[],[]

        for output in output3:
            for vec85 in output:
                scores=vec85[5:]
                class_id=np.argmax(scores)
                confidence=scores[class_id]

                if confidence > 0.5:
                    centerx,centery=int(vec85[0]*width),int(vec85[1]*height)
                    w,h=int(vec85[2]*width),int(vec85[3]*height)
                    x,y=int(centerx-w/2),int(centery-h/2)
                    boxes.append([x,y,w,h])
                    confidences.append(float(confidence))
                    class_ids.append(class_id)
```

Cmos 이미지와 열상 이미지의 객체 인식을 수행하는 코드이다.
preprocessing 함수를 수행하여 리턴받는다.
class_ids 는 인식한 객체의 이름, confidences 에는 정확률, boxes 에는 인식한 객체의 주위 일정 영역에 박스를 표시하기 위하여 사용된다.
output3은 객체인식을 수행했던 이미지의 좌표와 인식 점수, class별 정확률을 가지고 있다.

class_id 의 정확률(confidence)이 0.5가 넘는다면, 객체가 인식된 것으로 판단하며, 해당 객체의 주변에 칠 박스의 꼭지점의 좌표를 저장한다.
이 과정을 반복하면서 객체인식을 수행했던 이미지의 좌표 중 정확률이 0.5가 넘는 객체들은 모두 박스가 표시되고 어떠한 객체로 인식된 것인지 그 이름을 class_id에 저장하고 정확률도 저장한다.
객체인식을 수행했던 이미지의 좌표를 모두 탐색하면, 반복문을 빠져나가게 된다

```
indexes=cv2.dnn.NMSBoxes(boxes,confidences,0.5,0.4)

for i in range(len(boxes)):
    if i in indexes:
        x,y,w,h=boxes[i]
        text=str(classes[class_ids[i]])+'%.3f'%confidences[i]
        cv2.rectangle(img,(x,y),(x+w,y+h),colors[class_ids[i]],2)
        cv2.putText(img,text,(x,y+30),cv2.FONT_HERSHEY_PLAIN,2,colors[class_ids[i]],2)
```

indexes 는 NMS 알고리즘을 적용해 주위 바운딩 박스에 비해 최대를 유지한 것만 남기며, 반복문에서는 살아남은 박스의 class와 정확률 정보를 이미지위에 그린다

```
        if st == 0:
            img_name = 'D:/yolov3/1/EO/EO_r'+str(num)+'.png'
            cv2.imwrite(img_name, img)
            time1 = timeit.default_timer()

        elif st == 1 :
            img_name = 'D:/yolov3/1/IR/IR_r' + str(num) + '.png'
            cv2.imwrite(img_name, img)
            time2 = timeit.default_timer()


time6 = timeit.default_timer()
```

Cmos , 열상 이미지 종류에 따라 이름을 다르게 하여 저장한다.
또한 각각 저장 시간을 측정한다
모든 작업이 끝나면 (이미지 겹치기 작업까지 완료) 현재 시간을 측정한다.

```
print("cmos %f초 " %(time1 - time0))
print("열상 %f초 " %(time2 - time1))
print("결과 %f초 " %(time6 - time0))
```

마지막으로 각 작업마다 측정한 시간으로부터 Cmos 이미지의 객체인식에 걸린 시간, 열상 이미지의 객체 인식에 걸린 시간, 최종 겹치는 이미지를 생성하기 까지 걸린 시간을 계산하여 출력한다.

4. 결과

코드를 수행한 결과 위와 같이 최종 결과 이미지를 얻을 수 있었다.
이미지 내의 차량(car), 트럭(truck), 사람(human)을 인식하여 출력한 모습이다.

위의 결과는 이미지 내의 모든 객체를 인식하였지만, 한 명의 사람을 여러 번 인식하는 문제가 발생하였다. 학습 모델 layer 중 yolo의 구성이 원인일 것으로 생각된다.

다른 장면의 결과 이미지이다. 위의 결과는 수많은 자전거가 있음에도 불구하고 단 한 개의 자전거만 인식에 성공하였으며, 차량은 인식하였지만 사람은 인식하지 못하였다.

대체적으로 작은 물체부터 어느 정도 큰 물체까지 잘 식별이 되었지만 대체로 성능이 매우 뛰어나지는 않았다

```
cmos 0.134342초
열상 0.121959초
결과 0.292299초

Process finished with exit code 0
```

그림 28 객체인식 수행 시간

코드 실행 시 출력되는 객체 인식에 걸린 수행 시간을 출력하였다.

Cmos, 열상 이미지의 경우 즉 한 장의 이미지 처리는 0.13 sec,
겹친 이미지는 약 0.30 sec 정도 소요되었다.

5. 문제점 및 보완

5-1 성능

결과 이미지는 대부분 객체 인식 성능이 좋지 않았다.
대체적으로 큰 물체나 흐릿한(해상도가 좋지않은) 물체는 대부분 식별 되지 않았다.
이러한 현상을 대비해서 낮은 해상도의 이미지도 학습시켰는데, 학습 이미지의 개수가 매우 적어서인지 제대로 학습되지 않은 듯 하다.

상황에 따라 인식하지 못하는 물체의 특징도 매우 달랐다. 밤과 같이 어두운 장면에서는 객체를 많이 인식하지 못하였다.
어두운 장면에서는 열상 카메라의 이미지가 훨씬 성능이 좋게 나타났다.

5-2 학습 이미지의 다양성 필요

객체를 인식하지 못한 경우는 대부분 물체의 일부가 가려졌거나, 흐릿한 객체였다.
모델을 학습시킨 이미지는 대부분 coco 데이터 셋이었다. 내 주위 환경에 적합한 모델을 설계하기 위해서는 직접 촬영한 데이터를 많이 늘려서 학습시켜야 한다.
이번 프로젝트에서는 300장의 이미지를 직접 학습시켰는데 터무니없이 부족해 보인다.

5-3 수행 시간

Cmos, 열상 이미지의 경우 즉 한 장의 이미지 처리는 0.13 sec,
겹친 이미지는 약 0.30 sec 정도 소요되었다.

수행 시간의 목표치는 0.033 sec 이었는데, 설계 모델은 1장당 0.13 sec 가 소요되었다. 약 4배의 차이가 발생하였는데, 이와 같은 소요 시간을 줄이기 위해서는 모델 layer 구조개선 또는 이미지의 사이즈를 줄이는 방법을 검토해봐야 한다.

수행시간은 RTX2060이 탑재된 윈도우10(i7) 노트북에서 측정하였다.

모델 layer 구조를 개선함에 있어서,
현재 인식 성능이 좋지 않음에도 불구하고 모델의 layer를 축소하기만 한다면. 인식이 안 될 수도 있다.
성능을 유지하기 위해서는 filters 와 같은 다른 파라미터 값의 조정이 필요해 보인다.

수행 시간 뿐 만 아니라 성능과도 연관이 있는 항목은 학습 데이터의 선별이다. 실제로 학습에 사용된 대부분의 이미지인 coco 데이터 셋은 작동 환경인 도로와는 어울리지 않았으며, 이미지 목적도 많이 달랐다.

사용 목적에 맞게 모델 수정 및 학습 데이터의 선별이 모델 개선에 큰 도움이 될 것으로 생각된다