차량지능기초 과제1(Basis of Vehicle Intelligence HW1)

학부: 소프트웨어학부 **학번:** 20191620 **이름:** 심혜린

GitHub Link: https://github.com/ShimHyerin/2021-VehicleIntelligence/tree/main/HW1

1) 자율주행 인지에 관련된 3종 이상의 공개 Data Set 조사, 정리

① BDD100K

Link: https://bdd-data.berkeley.edu/

Video: https://youtu.be/IGi9K9FY35Y

- **소개**

BBD100K는 Berkeley Deep Drive의 약자로 UC Berkeley 인공지능 Lab(BAIR)에서 연구되었다. 비디오는 약 40초의 길이 그리고 720px dpi 해상도와 30fps인 고화질로 취득되었으며, 100,000개 이상의 비디오로 구성되어 있다. 핸드폰 디바이스를 활용하여 거친 주행환경을 구현했고, 이것으로 기록된 GPS/IMU 데이터도 함께 제공된다. 다양한 날씨(비, 흐림, 맑음, 안개 등) 및 낮과 밤이 적절한 비율로 기록되어 있다.

- 데이터 양/특징

Sequences: 100,000

Images: 120,000,000

Multiple Cities: Yes

Multiple Weathers: Yes

Multiple Times of Day: Yes

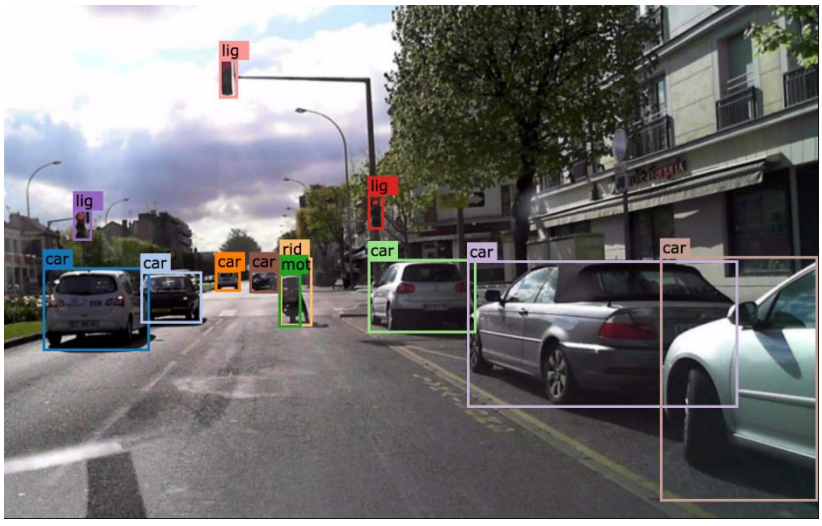
Multiple Scene types: Yes

위에서 서술한 바와 같이 10만개의 시퀀스와 1억 2천만 이미지라는 아주 방대한 양의 데이터를 가지고 있다. 그러나 라벨, 세그멘테이션, 맵 데이터가 딥러닝 모델 학습에 적합하도록 구조화되어 있기 때문에 데이터 전처리 과정에서 불필요한 반복이 필요로 하지 않다.

방대한 데이터를 가지고 있는 점과 그런 데이터 안에 여러 지역, 다양한 날씨, 낮과 밤을 모두 담고 있는 점이 가장 큰 특징이며 딥러닝 학습에 최적화된 구조로 되어있기 때문에 이 데이터 셋을 활용하여 다양한 응용을 쉽게 할 수 있다는 점을 특징으로 볼 수 있다.

- 데이터 활용/예시

1. 도로 객체 감지(Road Object Detection)

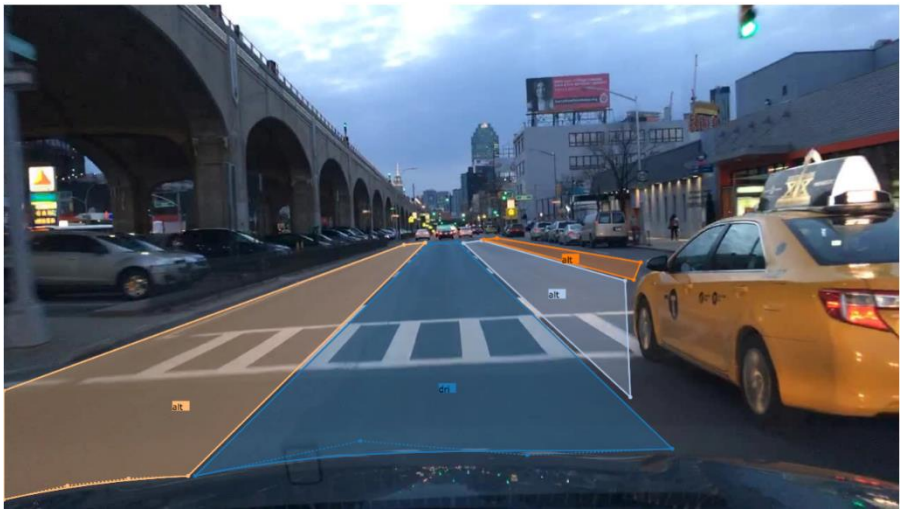


2D 바운딩 박스로 10만 개의 이미지 안에 버스, 신호등, 교통 표지판, 사람, 자전거, 트럭, 모터, 자동차, 기차, 그리고 탑승자를 표시하는 주석을 달았다.

2. 인스턴스 세그먼테이션(Instance Segmentation)

픽셀 레벨 및 풍부한 인스턴스 레벨 주석을 사용하여 10,000개 이상의 다양한 이미지를 탐색한다.

3. 운전 가능 지역(Driveable Area)



복잡한 10만 개의 이미지를 통해 운전이 가능한지를 학습한다. 또한 7만개의 훈련용, 만개의 검증용 pixel map 파일도 제공한다.

4. 차선 구분(Lane Markings)

주행 안내를 위해 100,000개의 영상에 여러 유형의 차선 표시 주석을 달았다.

② Waymo open dataset

Link: https://waymo.com/open/

Tutorial:https://colab.research.google.com/github/waymo-research/waymo-open-dataset/blob/master/tutorial/tutorial.ipynb

- **소개**

구글의 자회사인 Waymo에서 공개한 dataset이다. 비상업적인 용도면 모든 개발자에서 무료로 제공되고 있다. 19년도에 Perception dataset을 공개하였는데, 최근인 21년 3월에

Motion dataset를 공개하면서 dataset을 확장했다. 25개의 도시에서 약 1,000만 마일에 달하는 자율주행 테스트를 통해 데이터를 수집하였으며 그 데이터에는 다양한 기상조건과 보행자, 물체, 자전거 등의 조건이 함께 포함되어 있다.

- **데이터 양/특징**

Perception Dataset

인식 데이터는 다양한 지역(Multiple Cities)과 조건에서 10Hz(39만 프레임)으로 수집된 약 20초간의 주행 데이터로 구성되어 있다. 이 조건은 낮과 밤 등의 시간대, 날씨 등의 기상 조건 등의 다양한 주행 환경을 뜻한다.

센서 데이터

1 mid-range lidar

4 short-range lidars

5 cameras (front and sides)

Synchronized lidar and camera data

Lidar to camera projections

Sensor calibrations and vehicle poses

레이블이 있는 데이터

Labels for 4 object classes - Vehicles, Pedestrians, Cyclists, Signs

High-quality labels for lidar data in 1,200 segments

12.6M 3D bounding box labels with tracking IDs on lidar data

High-quality labels for camera data in 1,000 segments

11.8M 2D bounding box labels with tracking IDs on camera data

Motion Dataset

모션 데이터는 흥미로운 인터랙션을 위해 약 20초의 10Hz(20만 프레임 이상)로 채굴된 103,354개의 세그먼트로 구성되어 있다.

	Scenario proto	tf.Example
Segment length	9 seconds (1 history, 8 future)	9 seconds (1 history, 8 future)
Maps	Vector maps	Sampled as points
Representation	Single proto	Set of tensors

데이터는 시나리오 프로토콜 버퍼와 시나리오 프로토콜을 tf로 변환한 두가지의 형태로 제공된다.

객체 데이터

10.8M objects with tracking IDs

Labels for 3 object classes - Vehicles, Pedestrians, Cyclists

3D bounding boxes for each object

Mined for interesting behaviors and scenarios for behavior prediction research, such as unprotected turns, merges, lane changes, and intersections

3D bounding boxes are generated by a model trained on the Perception Dataset

지도 데이터

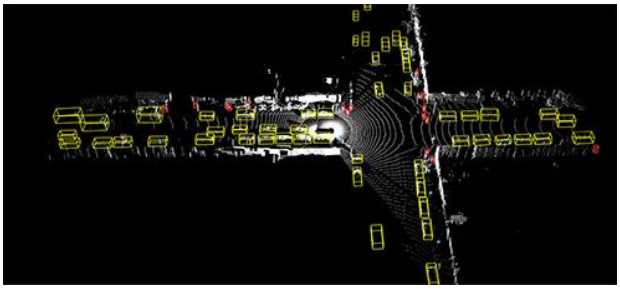
3D map data for each segment

Locations include: San Francisco, Phoenix, Mountain View, Los Angeles, Detroit, and Seattle

- **데이터 활용/예시**

Perception Dataset

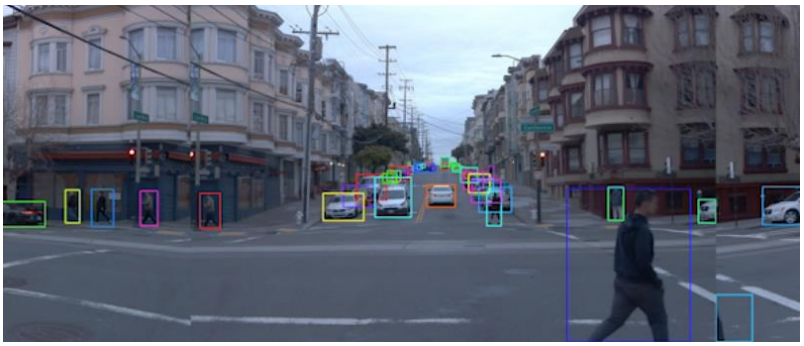
데이터 레이블 – 단순이 투영이 아닌 Lidar 및 Camera 데이터에 대해 독립적으로 생성된 레이블이 포함됨



[3D Lidar 레이블]

Lidar 데이터에 3D 경계 상자 레이블을 제공한다. 차량, 보행자, 자전거, 표지판에 3D 라벨이 있다. 원리는 다음과 같다.

> The bounding boxes have zero pitch and zero roll. Heading is the angle (in radians, normalized to [-π, π]) needed to rotate the vehicle frame +X axis about the Z axis to align with the vehicle's forward axis.
>
> Each scene may include an area that is not labeled, which is called a “No Label Zone” (NLZ). These capture areas such as the opposite side of a highway. See our label specifications document for details. NLZs are represented as polygons in the global frame. These polygons are not necessarily convex. In addition to these polygons, each lidar point is annotated with a boolean to indicate whether it is in an NLZ or not.
>
> Our metrics computation code requires the user to provide information about whether the prediction result is overlapping with any NLZ. Users can get this information by checking whether their prediction overlaps with any NLZ-annotated lidar points (on both 1st and 2nd returns).
>
> * https://waymo.com/open/data/perception/

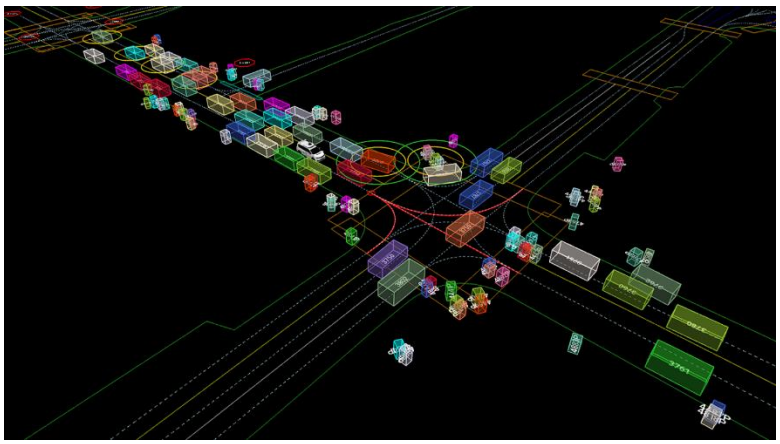


[2D 카메라 라벨]

차량, 보행자, 자전거 타는 사람에는 2D 라벨이 있다. 그러나 카메라 전체에 걸쳐 객체 추적 통신을 제공하지 않는다.

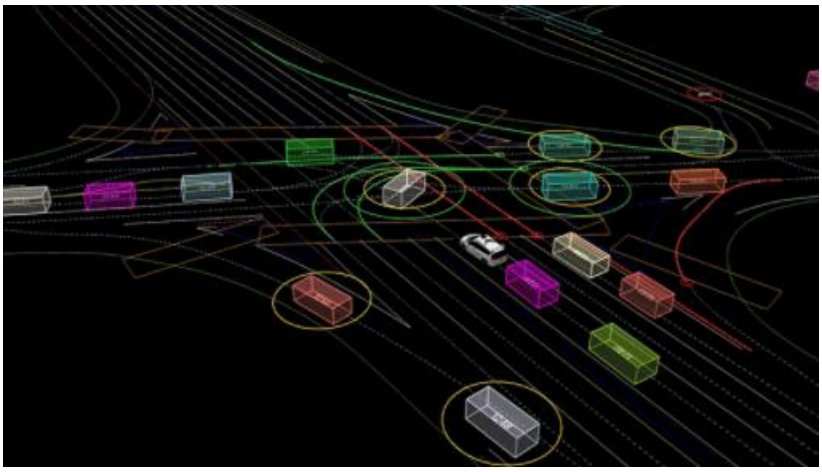
Motion Dataset

데이터 샘플링



훈련 또는 검증 세트의 각 9초 시퀀스는 1초의 기록 데이터, 현재 시간에 대한 1개의 샘플, 10Hz 샘플링에서 8초의 미래 데이터를 포함한다. 이는 총 91개 표본에 대해 10개의 이력 표본, 1개의 현재 시간 표본 및 80개의 미래 표본에 해당한다. 테스트 세트는 총 11개 샘플(이력 10개 및 현재 시간 샘플 1개)에 대한 실제 미래 데이터를 숨긴다.

시나리오 프로토콜 버퍼/tf 형식

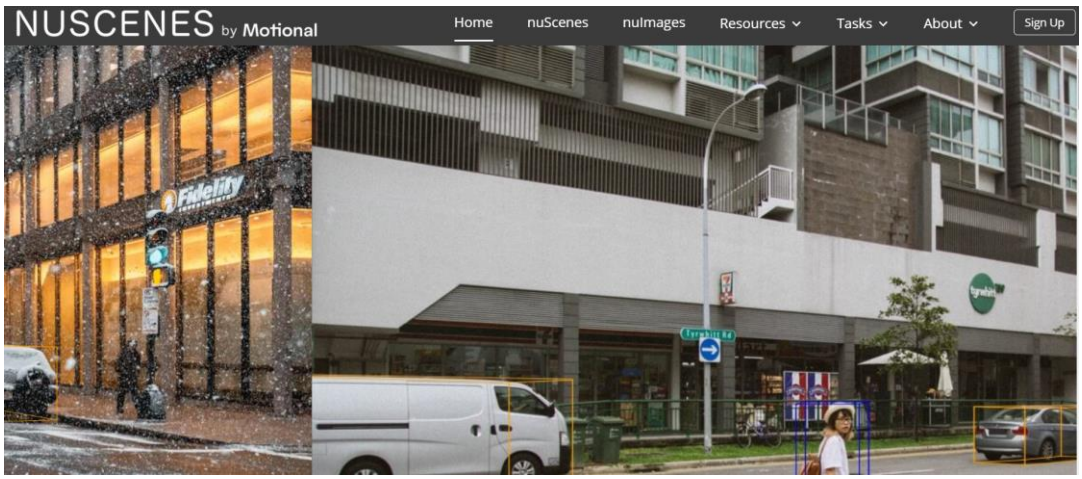


시나리오 프로토에는 시나리오의 각 시간 단계에 대한 객체 상태를 포함하는 객체 트랙 세트가 포함되어 있다. 또한 정적 맵 기능과 각 시간 단계에 대한 동적 맵 기능(예: 트래픽 신호)을 포함한다. 보다 자세한 내용은 밑의 주소에 서술되어 있다.

Link1: https://waymo.com/open/data/motion/

Link2:https://github.com/waymo-research/waymo-open-dataset/blob/master/waymo_open_dataset/protos/scenario.proto

③ nuScenes



Link: https://www.nuscenes.org/

GitHub: https://github.com/nutonomy/nuscenes-devkit

- **소개**

nuScenes dataset은 3D 객체(물체) 주석이 포함된 대규모 자율 주행 데이터 세트다. 약 20초 분량의 scene이 1000개 있고 1,400,000개의 카메라 이미지와 390,000개의 라이더 스윕이 있다. nuScenes 또한 Multiple Cites 데이터를 지원한다(보스턴, 싱가포르).

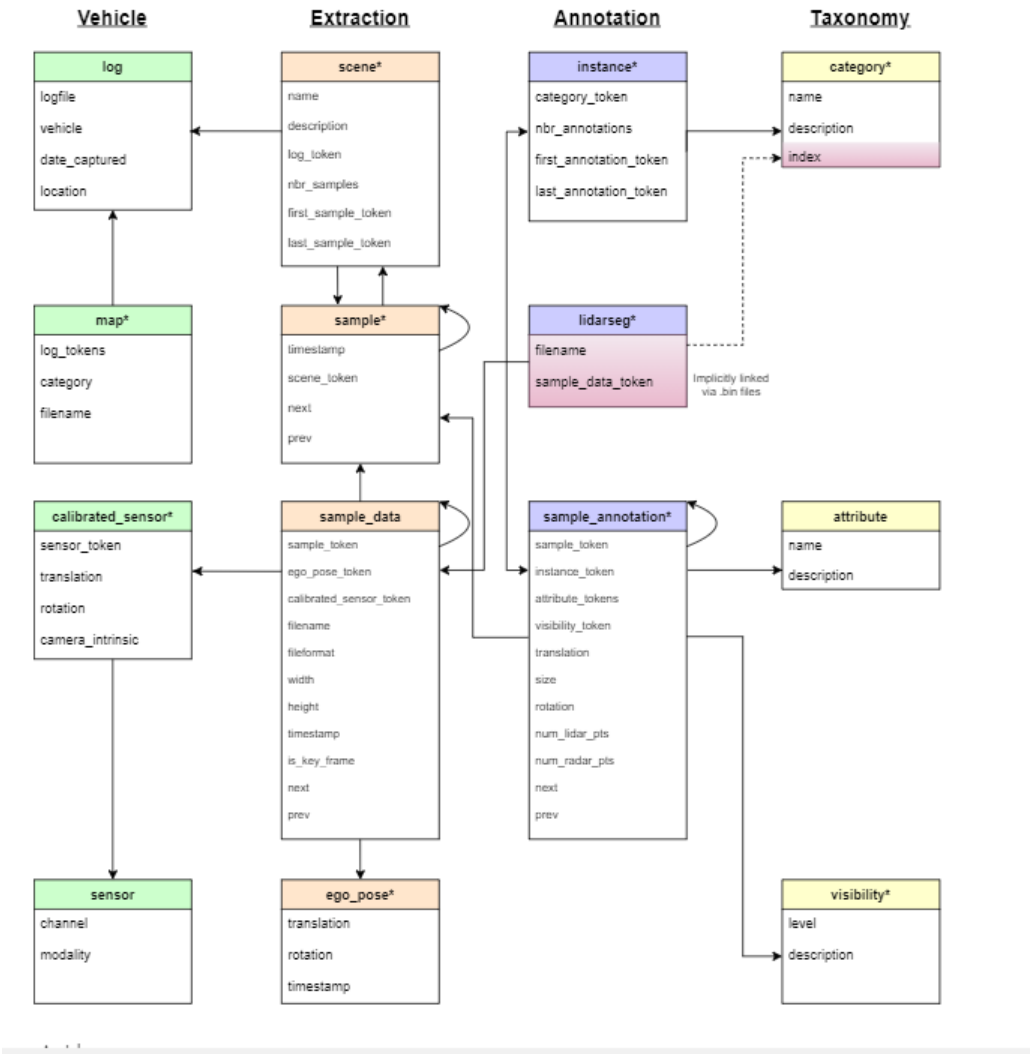
- **데이터 양/특징**

nuScences dataset 사이트에 서술되어 있는 dataset의 특징은 다음과 같다.

- Full sensor suite (1x LIDAR, 5x RADAR, 6x camera, IMU, GPS)
- 1000 scenes of 20s each
- 1,400,000 camera images
- 390,000 lidar sweeps
- Two diverse cities: Boston and Singapore
- Left versus right hand traffic
- Detailed map information
- 1.4M 3D bounding boxes manually annotated for 23 object classes
- Attributes such as visibility, activity and pose
- New: 1.1B lidar points manually annotated for 32 classes

- New: Explore nuScenes on SiaSearch
- Free to use for non-commercial use

데이터 스키마는 다음과 같다.



Tutorial: https://www.nuscenes.org/nuimages#tutorial

Attribute: 속성은 범주가 동일하게 유지되는 동안 변경 될 수 있는 인스턴스의 속성이다.

Ex) 주차 / 정지 / 이동중인 차량 및 자전거에 탑승자가 있는지 여부.

calibrated_sensor: 특정 차량에서 보정 된 특정 센서 (라이더 / 레이더 / 카메라)의 정의이다. 모든 외부 매개 변수는 자아 차체 프레임과 관련하여 제공된다. 모든 카메라 이미지는 왜곡되지 않고 수정된다.

Category: 객체 범주의 분류이다 (ex: 차량, 사람).

Ego_pose: 특정 타임 스탬프에서 차량의 ego pose

Instance: 객체 인스턴스(예: 특정 차량) 이것은 관찰한 모든 개체 인스턴스의 열거이다. 인스턴스는 여러 장면에서 추적되지 않는다.

Lidarseg: keyfram 과 연결된 lidar point cloud 에 해당하는 nuScene-lidarseg 주석과 sample_datas 간의 매핑이다.

Log: keyframe 과 연결된 lidar point cloud 에 해당하는 nuScene-lidarseg 주석과 sample_datas 간의 매핑이다.

Map: 하향식 보기에서 바이너리 시멘틱 마스크로 저장된 데이터이다.

Sample: 샘플은 2Hz 에서 주석이 달린 키 프레임입니다. 데이터는 단일 LIDAR 스위프의 일부와 동일한 타임스탬프에서 수집된다.

Sample_annontation: 표본에 표시된 개체의 위치를 정의하는 경계 상자. 모든 위치 데이터는 글로벌 좌표계와 관련하여 제공된다.

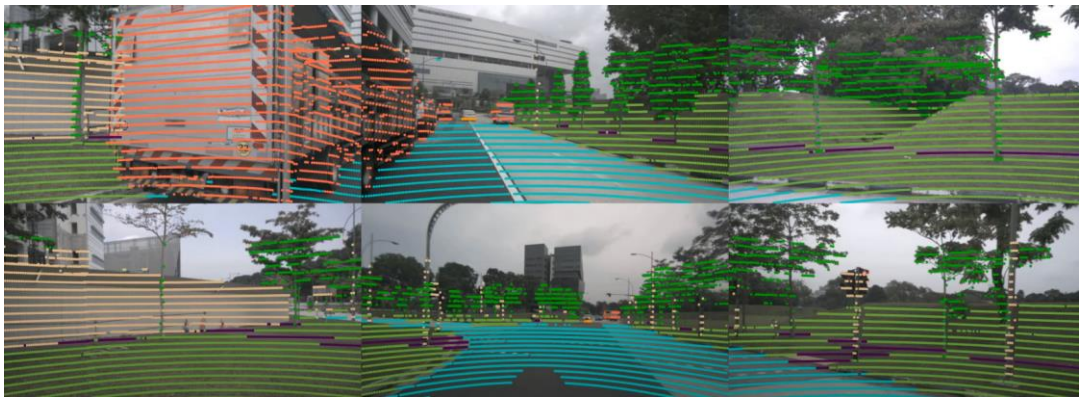
Sample_data: 이미지, 포인트 클라우드 또는 레이더 리턴과 같은 센서 데이터. is_key_frame=인 sample_data 의 경우사실, 타임스탬프는 해당 타임스탬프가 가리키는 표본과 매우 가까워야 한다. 키 프레임이 아닌 경우 sample_data 는 시간에 가장 가까운 샘플을 가리킨다.

Scene: 씬(scene)은 로그에서 추출한 연속 프레임의 20 초 길이의 시퀀스다. 동일한 로그에서 여러 장면이 나올 수 있다. 개체 ID(인스턴스 토큰)는 여러 장면에서 보존되지 않는다.

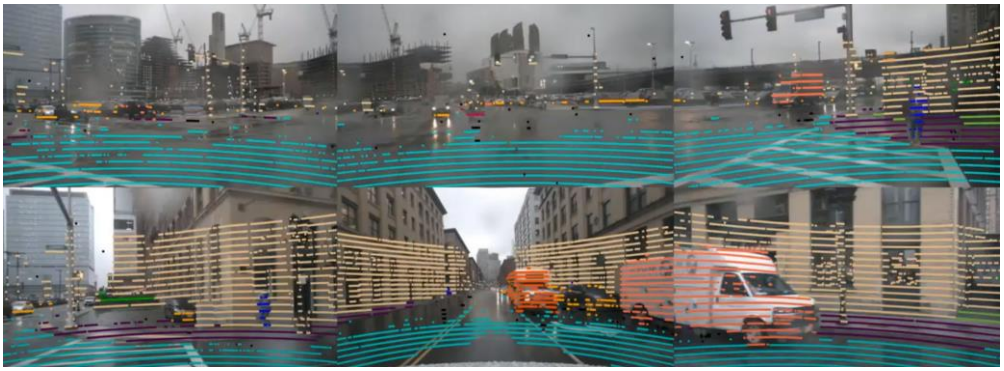
Sensor: 특정 센서 유형.

Visibility: 인스턴스의 가시성은 모든 6 개 영상에 표시되는 주석의 일부다. 4 개의 빈으로 0-40%, 40-60%, 60-80%, 80-100%로 결합된다.

- **데이터 활용/예시**



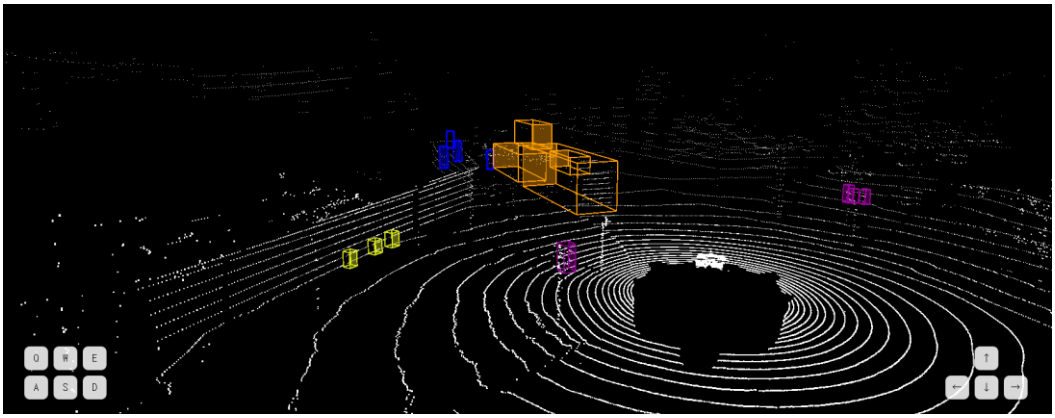
Video: https://youtu.be/NMpqXYXORvs

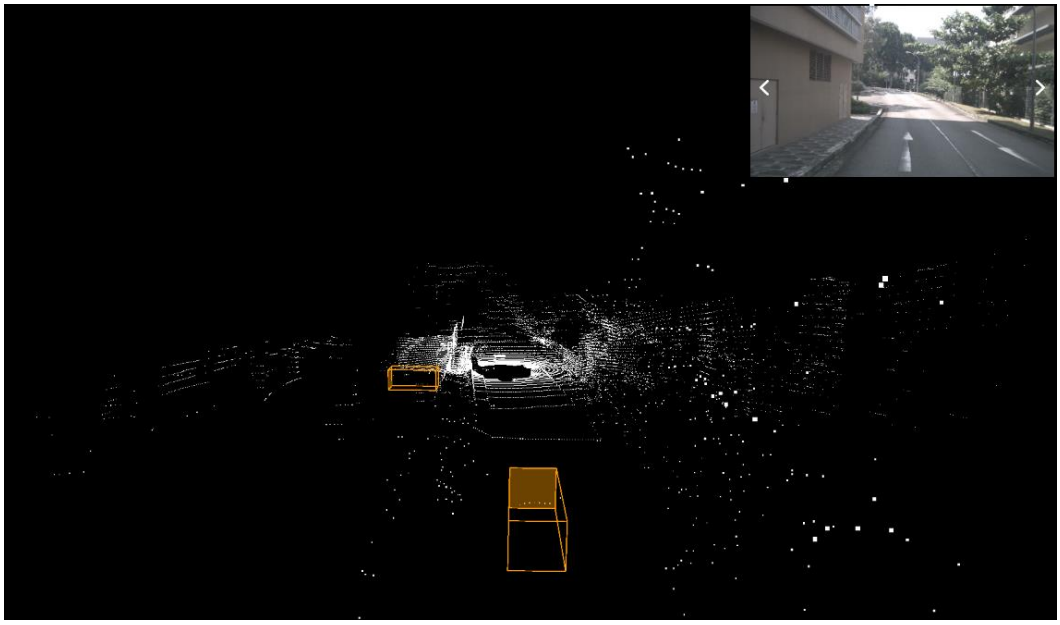


Video: https://youtu.be/Y-lTpvYhc0A

[NuScence – Lidarseg]

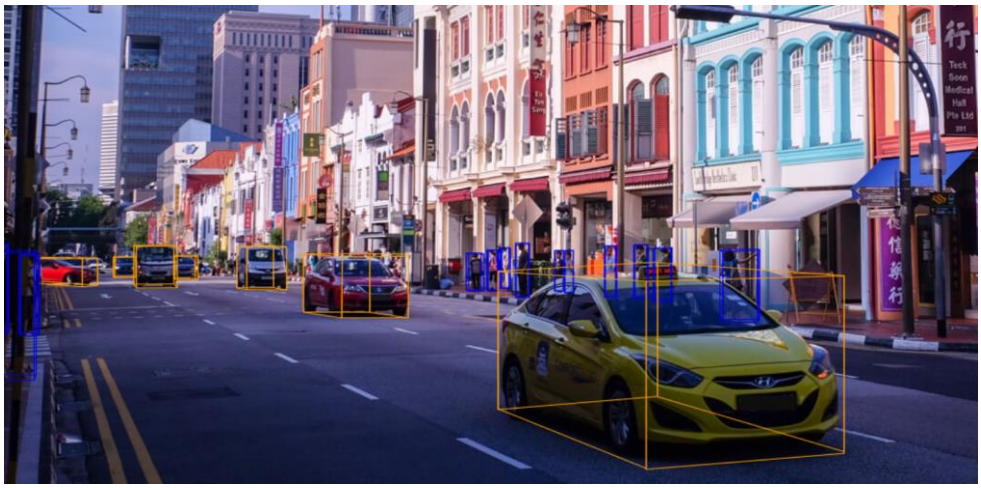
NuScence의 Lidarseg는 Bounding Box 또는 큐보이드(cuboid)가 3D 객체를 나타내는 데 사용된다. nuScene-lidarseg는 32개의 레이블 중 하나로 주석이 달린 nuScene 데이터 세트의 40,000 키 프레임에 있는 모든 leadar 포인트에 대한 주석을 포함함으로써 더 높은 수준의 세분성을 가진다. nuScene의 23개 포그라운드 클래스(사물) 외에도 9개의 백그라운드 클래스(사물)가 포함되어 있다. 이를 활용하여 라이더 포인트 클라우드 분할, 전경 추출, 센서 보정 및 매핑과 같은 것들을 정량화 할 수 있다.





[nuScences - Scene]

싱가포르와 보스턴의 도시 및 교외 장면에서 나온 조밀한 데이터를 특징으로 하여 360도 뷰 전체를 포괄하는 다양한 dataset이며 상업적으로 사용할 수 있다.



차량 검출 및 물체 검출, 도로 검출 등 자율 주행에 필요한 물체 인식에 활용할 수 있는

Dataset이다.

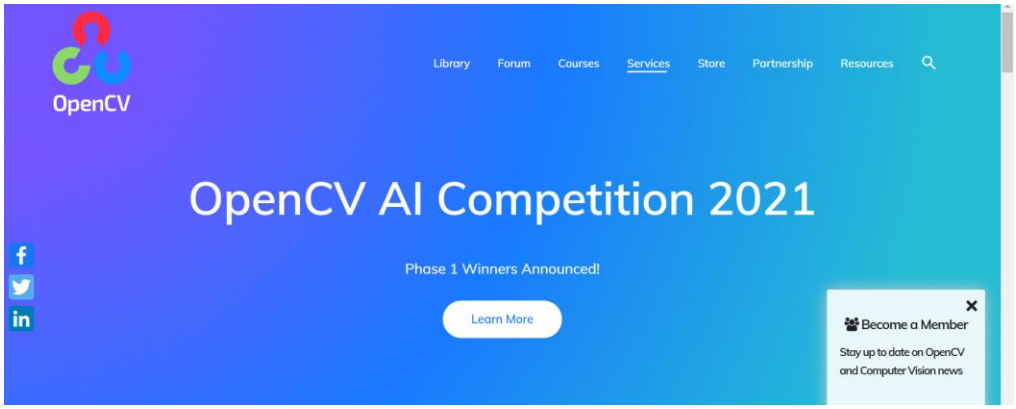
2) 자율주행 인지에 관련된 2종 이상 Open Source 조사, 정리

① 물체 인식 오픈소스

자율주행을 위해서는 자동차, 사람, 자전거 등 물체 인식이 필수적이다. 조사한 코드는 오픈소스 컴퓨터 비전 라이브러리인 OpenCV와 cvlib를 사용하여 물체를 검출하는 코드다. 이 코드는 물체를 인식한 후 인식한 물체를 2D Bounding 박스로 표시해주고 그렇게 검출한 물체의 종류를 박스 위에 텍스트로 표시해주는 것이 목표이다.

먼저 이 코드에 사용될 오픈소스 라이브러리 및 모델에 대해 설명하겠다.

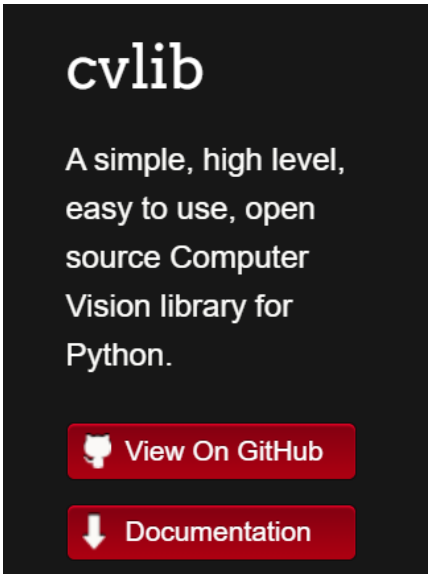
OpenCV



Link: http://opencv.org

컴퓨터 비전 라이브러리로 크로스플랫폼과 실시간 이미지 프로세싱에 중점을 둔 라이브러리다. 영상처리가 필요할 때 거의 필수적으로 사용하고 있으며, 이 코드에서는 이미지를 읽고 쓰는 기능을 위해 사용될 것이다.

Cvlib



Link1: Information Site

Link2: GitHub

파이썬에서 객체 혹은 얼굴 인식을 위해 사용하기 쉬운 컴퓨터 비전 라이브러리다. 이것은 OpenCV와 tensorflow를 사용하고 있기 때문에 cvlib를 쓰기 위해선 필수로 설치해야 한다. 이 코드에서는 cvlib의 함수 detect_common_object()를 사용하기 위해 필요하다. 이 함수가 실질적인 물체를 검출하는 역할을 하며 약 80종류의 물체 검출이 가능하다.

yolo

물체 검출을 하는 cvlib의 함수 detect_common_object()에서 COCO dataset으로 학습된 yolov3 모델을 지원한다. yolo모델은 물체 인식을 수행하기 위해 고안된 유명한 심층 신경망이며 현재 yolov5까지 공개되었다.

전체 코드는 아래와 같다.

```
import cv2
import cvlib as cv
from cvlib.object_detection import draw_bbox

img_path = './tmp.jpg'
img = cv2.imread(img_path)

bbox, label, conf = cv.detect_common_objects(img)
print(bbox, label, conf)
img = draw_bbox(img, bbox, label, conf)
```

```
cv2.imwrite('result.jpg', img)
```

코드 설명

- 라이브러리 import

```
import cv2
import cvlib as cv
from cvlib.object_detection import draw_bbox
```

OpenCV를 import한다.

cvlib를 import한다.

cvlib의 object_detection 모듈 안의 draw_bbox를 import한다.

- 이미지 읽기

```
img_path = './tmp.jpg'
img = cv2.imread(img_path)
```

물체를 검출할 이미지의 경로를 작성하여 이미지를 가져온다.

openCV의 imread함수를 사용해 이미지를 읽는다.

- 물체 인식

```
bbox, label, conf = cv.detect_common_objects(img)
print(bbox, label, conf)
img = draw_bbox(img, bbox, label, conf)
```

cvlib의 detect_common_object() 함수를 사용하여 아까 읽은 이미지에서 물체를 인식한다.

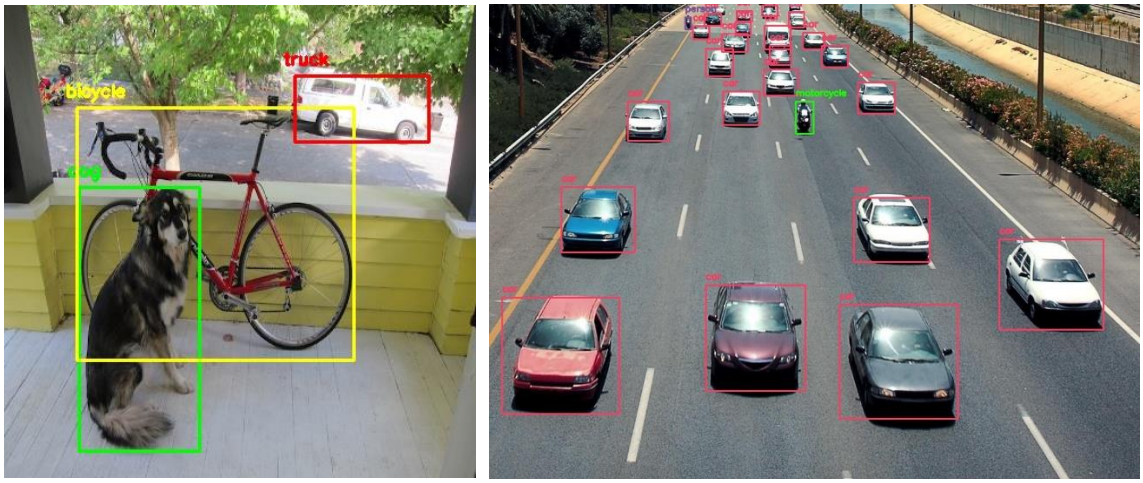
읽은 이미지에 물체를 검출한 데이터로 덮는다.

- 이미지 쓰기

```
cv2.imwrite('result.jpg', img)
```

OpenCV의 imwrite() 함수를 사용해 검출 결과 이미지를 생성한다.

위 코드를 실행 시 이런 결과가 나올 것이다.



[실행 예시 사진]

② 차선 인식 오픈소스

자율주행에서 차선 인식은 굉장히 중요하다. 차선 인식은 주행시 큰 역할을 한다. 자율 주행에서 차선 인식은 예를들어 차도의 중앙에 위치하기 위해 양 옆 차선을 인식하고 그 중간 위치를 center로 지정해 주행을 하는 상황 또는 차선변경 시 점선 혹은 직선인지 판별하거나 다른 차량이 차선을 변경하고 있을 때 차선이 가려지는 정도를 판별하는 상황 등에 사용된다.

조사한 코드는 도로에 있는 "직선"차선을 검출하여 도로 이미지에 검출한 차선을 일정 색상으로 표시해주는 것이 목표다.

이 코드에서는 위에서 서술한 OpenCV 오픈소스 라이브러리와 수학연산을 위한 Numpy 패키지를 사용한다.

Numpy

Link: NumPy

수학 및 과학 연산을 위한 파이썬 패키지다. 편리하게 수치해석, 통계 관련 기능을 실행할 수 있으며 실행 속도도 꽤 빠른 편에 속한다. 기본적으로 array를 생성하고 색인, 처리, 연산 등의 기능을 수행한다. 보통 아래와 같은 코드로 위 모듈을 호출한다.

import numpy as np

차선 검출의 단계

차선 검출을 위해 총 5번의 단계를 거쳐 결과값을 얻을 수 있다. 필요한 단계는 다음과 같다.

0. 기본 도로 이미지/영상

1 Grayscle(흑백 처리)

2 Blur(노이즈 제거)

3 Canny edge

4 ROI(관심영역)

5 Hough transform(허프변환)

6. Results

전체 코드는 다음과 같다.

```
import cv2
import numpy as np

def grayscale(img):
    return cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)
```

```
def gaussian_blur(img, kernel_size):
    return cv2.GaussianBlur(img, (kernel_size, kernel_size), 0)

def canny(img, low_threshold, high_threshold):
    return cv2.Canny(img, low_threshold, high_threshold)

def region_of_interest(img, vertices, color1=255):

    mask = np.zeros_like(img)
    color = color1

    cv2.fillPoly(mask, vertices, color)

    ROI_image = cv2.bitwise_and(img, mask)
    return ROI_image

def draw_lines(img, lines, color=[0, 0, 255], thickness=2):
    for line in lines:
        for x1,y1,x2,y2 in line:
            cv2.line(img, (x1, y1), (x2, y2), color, thickness)

def hough_lines(img, rho, theta, threshold, min_line_len, max_line_gap):
    lines = cv2.HoughLinesP(img, rho, theta, threshold, np.array([]), minLineL
ength=min_line_len, maxLineGap=max_line_gap)
    line_img = np.zeros((img.shape[0], img.shape[1], 3), dtype=np.uint8)
    draw_lines(line_img, lines)

    return line_img

def weighted_img(img, initial_img, α=1, β=1., λ=0.):
    return cv2.addWeighted(initial_img, α, img, β, λ)

### 이미지의 경우
image = cv2.imread('tmp.jpg')
height, width = image.shape[:2]

gray_img = grayscale(image)
blur_img = gaussian_blur(gray_img, 3)
canny_img = canny(blur_img, 70, 210)

vertices = np.array([[(50,height),(width/2-
45, height/2+60), (width/2+45, height/2+60), (width-
50,height)]], dtype=np.int32)

ROI_img = region_of_interest(canny_img, vertices)
hough_img = hough_lines(ROI_img, 1, 1 * np.pi/180, 30, 10, 20)
```

```
result = weighted_img(hough_img, image)
cv2.imshow('result',result)
cv2.waitKey(0)

###

### 동영상의 경우
cap = cv2.VideoCapture('tmp.mp4')

while(cap.isOpened()):
    ret, image = cap.read()

    height, width = image.shape[:2]

    gray_img = grayscale(image)

    blur_img = gaussian_blur(gray_img, 3)

    canny_img = canny(blur_img, 70, 210)

    vertices = np.array([[(50,height),(width/2-
45, height/2+60), (width/2+45, height/2+60), (width-
50,height)]], dtype=np.int32)
    ROI_img = region_of_interest(canny_img, vertices)

    hough_img = hough_lines(ROI_img, 1, 1 * np.pi/180, 30, 10, 20)

    result = weighted_img(hough_img, image)
    cv2.imshow('result',result)
    if cv2.waitKey(1) & 0xFF == ord('q'):
        break
cap.release()
cv2.destroyAllWindows()

###
```

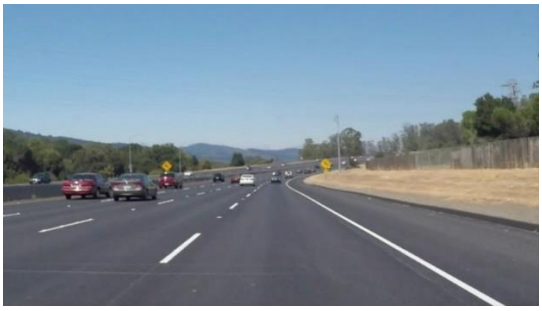
참고: https://m.blog.naver.com/windowsub0406/220894462409

코드 설명

차선 검출 단계별 코드의 설명과 함께 해당 단계를 왜 수행해야 하는지에 대한 원리도 함께 서술하였다.

0. 기본 도로 이미지/영상

자율주행시 cam에 찍히는 기본 데이터이다. 눈으로 보이는 실제 도로와 유사하다.



```
image = cv2.imread('tmp.jpg')
height, width = image.shape[:2]
```

[이미지의 경우]

```
cap = cv2.VideoCapture('tmp.mp4')
```

[동영상의 경우]

1. grayscle(흑백 처리)

컴퓨터의 연산량을 줄이기 위해 사용한다. 필수는 아니며 상황에따라 컬러 이미지 데이터가 edge 검출에 유리할 수도 있다. 그러나 이 경우 RGB 3가지 채널 각각에 대한 모든 edge를 연산하면서 속도저하나 연산량 증가라는 결과가 나타나게 된다.

```
def grayscale(img):
    return cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)

gray_img = grayscale(image)
```

OpenCV의 cvtColor(이미지, cv2.COLOR_RGB2GRAY) 함수를 통해 흑백으로 사진을 전환한다. 함수의 용어 그대로 색상 이미지(RGB)를 GRAY로 색상변환(color convert)을 하는 것이다.

2. blur(노이즈 제거)

이미지 데이터의 노이즈를 없애고 불필요한 gradient를 제거하기 위해 사용한다. 필수는 아니지만 실제 canny edge 알고리즘 단계에는 노이즈 제거를 위한 blur작업을 수행하지만 OpenCV에 내장되어있는 Canny edge 함수는 blur를 수행하지 않는다. 노이즈 제거를 위해 blur를 수행하는 단계를 직접 추가시켜 실제 Canny edge 알고리즘과 비슷하게 구현하는 것이 좋을 것 같다는 생각이 든다.

```
def gaussian_blur(img, kernel_size):
    return cv2.GaussianBlur(img, (kernel_size, kernel_size), 0)

blur_img = gaussian_blur(gray_img, 3)
```

openCV의 GaussianBlur() 함수를 사용한다. 가우시안 블러는 여러가지의 번짐(blur) 효과 중 하나의 방법이다. Kernel 사이즈를 크게 키울수록 Blur 효과가 강해진다.

3. canny edge

여러 단계를 거쳐 edge를 검출하는 방법이다. 검출한 edge가 너무 두꺼워 object를 식별하기 어렵게 하거나 중요한 edge를 모두 검출하기 위한 명확한 경계 값을 찾기 불가능할 때 Canny 알고리즘을 사용해 해결한다. 간단한 단계로 설명하면 아래와 같다.

노이즈 제거

gradient값이 높은 부분 찾기

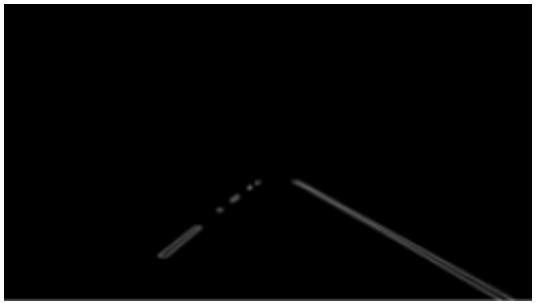
edge가 아닌 픽셀 제거하기

hyteresis thresholdin(진짜 edge가 맞는지 판단하기)- max 값과 min값 사이에 위치한 약한 edge가 강한 edge와 연결되어 있으면edge로 판단, 반대의 경우 edge 제거

```
def canny(img, low_threshold, high_threshold):
    return cv2.Canny(img, low_threshold, high_threshold)

canny_img = canny(blur_img, 70, 210)
```

openCV에 내장된 Canny Edge 알고리즘인 Canny() 함수를 사용한다. low_threshold, high_threshold의 비율은 1:2 혹은 1:3이 이상적이다.

4. ROI(관심영역)



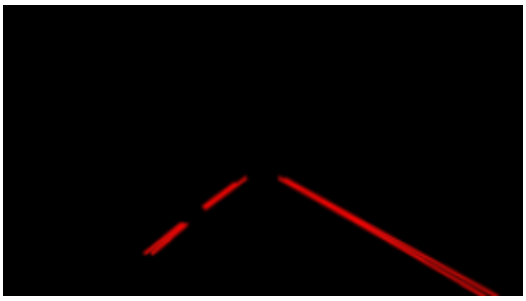
Region of Interest의 약자다. 원하는 부분을 따로 분리하여 관심영역으로 지정하고 이를 영상처리(차선검출) 할 때 사용된다.

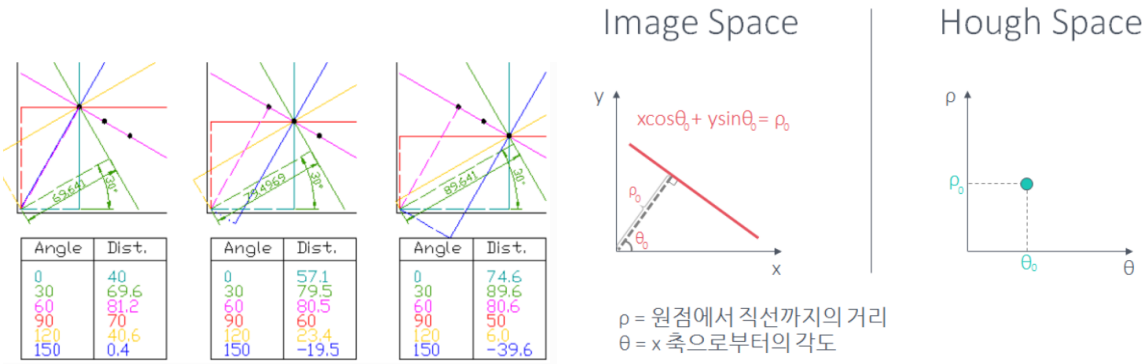
```
def region_of_interest(img, vertices, color3=(255,255,255), color1=255):


    mask = np.zeros_like(img)
    if len(img.shape) > 2: # 컬러 이미지 -> R, G, B 채널 3 개

        color = color3
    else: # 흑백 이미지 -> 채널 하나
        color = color1

    cv2.fillPoly(mask, vertices, color)

    ROI_image = cv2.bitwise_and(img, mask)
    return ROI_image
```

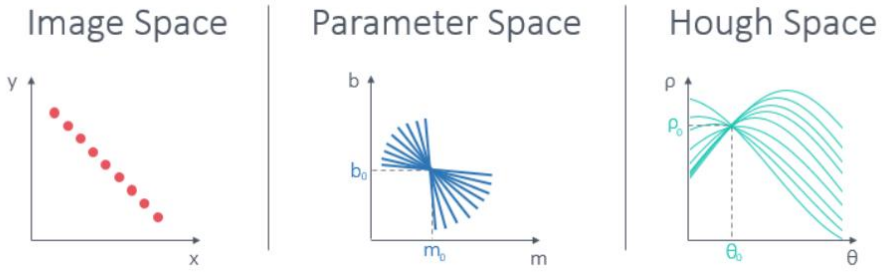
```
vertices = np.array([[(50,height),(width/2-
45, height/2+60), (width/2+45, height/2+60), (width-
50,height)]], dtype=np.int32)

ROI_img = region_of_interest(canny_img, vertices)
```

수학적 연산을 위해 numpy 모듈을 사용하고, openCV의 fillPoly(이미지, 좌표, 색상) 함수와 bitwise_and() 함수를 사용한다. fillPoly() 함수는 채워진 다각형을 그려준다. bitwise_and(이미지 1, 이미지2) 함수는 이미지 비트연산을 위해 사용된다. 당연하게도 AND 연산을 수행한다.

5. hough transform(허프변환)







직선을 표현하는 방법을 x,y (Image space) -> m,b(Parameter space) -> ρ,θ(Hough space) 이렇게 바꾼 것이다. Parameter space로 바꾸면 기존 이미지에 있는 수많은 직선들을 하나의 점으로 표현할 수 있다. 이런 Parameter space의 단점인 기울기가 무한대로 가는 문제점을 해결하기 위해

직선이 아닌 곡선으로 표현한 게 Hough Space다.

```
def hough_lines(img, rho, theta, threshold, min_line_len, max_line_gap):
    lines = cv2.HoughLinesP(img, rho, theta, threshold, np.array([]), minLineL
ength=min_line_len, maxLineGap=max_line_gap)
    line_img = np.zeros((img.shape[0], img.shape[1], 3), dtype=np.uint8)
    draw_lines(line_img, lines)

hough_img = hough_lines(ROI_img, 1, 1 * np.pi/180, 30, 10, 20)
```

openCV의 함수 HoughLinesP() 함수를 사용한다. HoughLinesP()는 HoughLines() 함수를 최적화한 것으로 확률 허프변환이다. 간단하게 설명하면 HoughLines() 는 직선을, HoughLinesP()는 선분릉 출력한다는 차이가 있다. 모든 점이 아닌 임의의 점을 이용하여 직선을 찾는다. 따라서 임계값을 작게 설정해야 한다. 선의 시작과 끝점을 return 해주어 화면에 쉽게 표시할 수 있다는 장점이 있다. HoughLinesP() 함수의 자세한 사용 법은 다음과 같다.

HoughLinesP(이미지, rho(r값의 법위 0~1), theta(0~180), threshold(클수록 정확도 상승), minLineLength(선의 최소 길이), maxLineGap(선과 선 사이의 최대 허용 간격))

6. 결과

위 알고리즘 수행 후 openCV의 weight_img() 함수를 통해 검출된 선을 덮어쓰고(overlap), imshow() 함수를 통해 결과값이 담긴 이미지를 출력하면 차선이 검출된 이미지를 얻을 수 있다.

```
result = weighted_img(hough_img, image)
cv2.imshow('result',result)
```

[이미지의 경우]

3) 2)의 정리한 코드 중 하나 실행해서 결과 확인

두 가지의 코드 중 차선 인식 코드를 직접 실행시켜 보았다. 이 코드는 개발 환경 설정이 필요하다. 설치가 필요한 라이브러리는 OpenCV며, 설치 방법은 다음과 같다.

```
pip install opencv-python
```

python 버전 2.7과 충돌이 나는 경우 python의 기본 버전을 python3으로 변경하는 방법이 있다. 파이썬 버전은 다음과 같은 명령어를 통해 확인이 가능하다.

```
python –version
```

python3 –version 을 하면 python3 버전의 현재 버전이 출력된다.

충돌 해결

```
sudo update-alternatives –config python
```

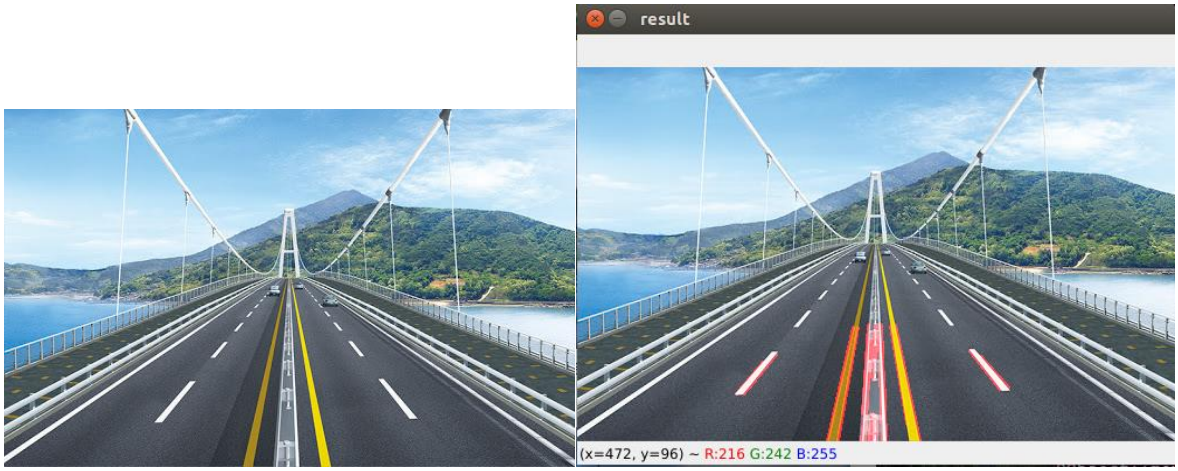
만약 여기서 에러가 난다면

```
sudo update-alternatives –install /usr/bin/python python /usr/bin/python3.7(현재버전) 1
```

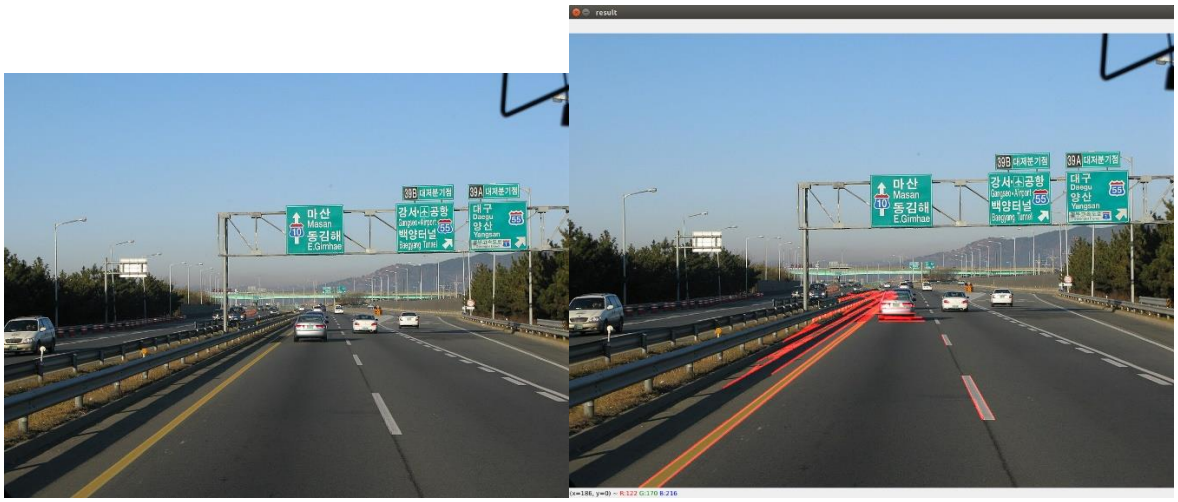
이렇게 python의 버전을 변경하여도 python2.7로 실행이 된다면 ROS(python2.7)과 충돌이 나는 것을 의심해 볼 수 있다. 이 경우 터미널을 실행시킬 때마다 다음 명령어를 입력하면 정상적으로 작동된다.

```
unset PYTHONPATH
```

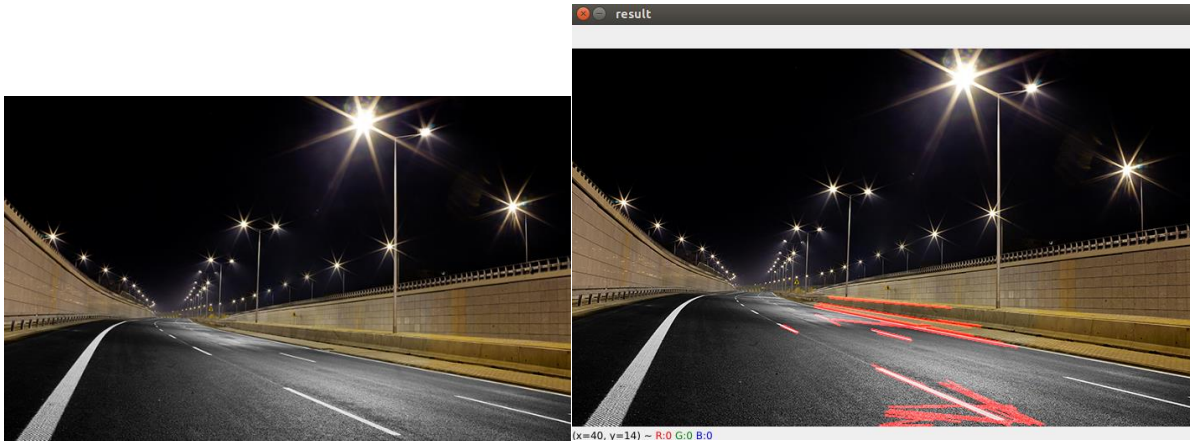
다음은 차선인식 코드의 실행 결과이다.



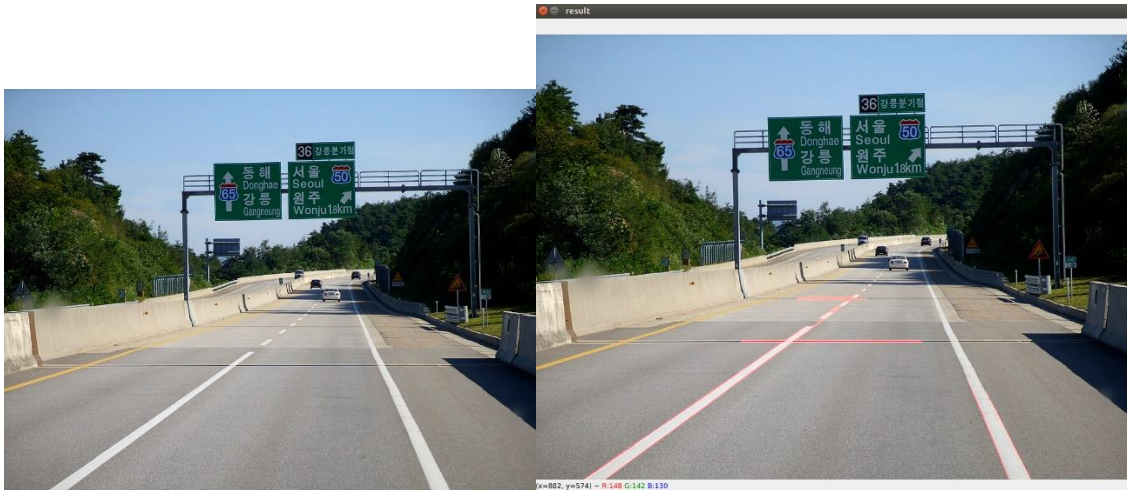
[결과 1]



[결과 2]



[결과 3]



[결과 4]

정확도가 높지는 않지만 무난하게 차선이 인식되는 결과를 확인할 수 있다. 추후에 곡선이나 그림자가 있는 곳에서도 차선인식이 가능한 코드를 구현해 보는 것도 좋을 것 같다.