상명대학교 컴퓨터 비전 기말 프로젝트 보고서

김준호

융합전자공학과

201710860

목차

I. 프로젝트의 개요 및 목표

II. 프로젝트 설계서

1. 전체 구성도 및 설명

2. 기능별 구성도 및 설명

III. 프로젝트 구현

1. 기능별 구현 내용 (프로그램 모듈별로 코드 설명)

IV. 실험 및 결과

1. 주어진 영상에 대해서 실험 결과를 수치와 graph, 이미지를 이용하여 표시 (성능을 나타낼 수 있는)

V. 결론

1. 프로젝트의 수행요약 및 결과

2. 프로젝트를 통해 배운 것

I. 프로젝트의 개요 및 목표

최근 자율주행에 대한 관심 및 수요가 급격히 증가하였다. 이에 따라서 첨단 운전자 보조 시스템, 일명 ADAS (Advanced Driver Assistance Systen) 에 대한 관심도 증가하고 있다.

ADAS란, 운전자가 안전하고 편리하게 주행할 수 있도록 도와주는 모든 기능을 말한다. 해당 수업에서는 ADAS 기능 중 아래와 같은 기능들을 구현한다.

1. 차선 검출
2. 차선 이탈 경고
3. 속도 검출
4. 급출발 / 급감속
5. 앞 차량과의 충돌 경고

제한 사항으로는 다음과 같은 것들이 있다.

먼저 딥러닝 기술을 사용하지 않고 전통적인 computer vision 기술로만 해결해야한다. 또한 구현한 시스템의 성능은 주어진 두 개의 영상에서 검증한다.

“highway\_D5\_trim.mp4” 및 “highway\_D6\_Trim.mp4”

II. 프로젝트 설계서

1. 전체 구성도 및 설명

먼저 카메라로부터 frame 이미지를 받아온다. 그리고 이들을 각각에 목적에 맞추어 적절히 이미지를 preprocessing 한 뒤, 먼저 detection을 한다.

Object Detection의 경우에는 Haar Cascade Algorithm을 사용할 예정이다. return 값으로 추정된 objects 들의 xywh 를 return 한다.

Lane Detection의 경우에는 직접 고안한 Algorithm을 사용할 예정이다. 이는 구현 설명 파트에서 자세히 설명하겠다.

Velocity Prediction의 경우에는 CCTV 영상으로 차량의 속도를 추정하는 기술에서 아이디어를

Diagram

Description automatically generated

Figure . 전체 구성도

얻었다. 점선 차선이 제대로 검출된다는 가정과 점선이 5m 라는 가정을 하면 영상 중간에 가로로 가상의 선을 긋고 점선 하나가 가상의 선을 지나가는데 몇 frame 이 걸리는지 계산하면 속도를 쉽게 추정 할 수 있다. 해당 Algorithm은 Lane Detection 에서 preprocessing 된 이미지를 사용하기 때문에 Lane Detection 의 로직 내부에서 수행된다.

Warning 에서는 현재 주행중인 차선을 이탈했는지에 대한 여부를 판단하고 충돌 위험의 여부를 판단한다. 또한 급출발, 급감속에 대한 판단을 한다.

이후 Visualize를 하는데 redundancy 를 줄이기 위해 이는 class 로 따로 구현하지 않고 필요한 정보가 있을때마다 수행할 계획이다.

2. 기능별 구성도 및 설명

2.1. frame acquisition

먼저 frame을 받아오는 구성도에 대해 설명한다. opencv 의 cv2.VideoCapture() class 의 instance 에서 read() method를 호출하면 frame의 존재 여부를 bool 변수로 나타내는 ret 과 frame을 return 한다. Figure2. 를 보면 받아온 frame을 어떻게 처리하는지에 대한 pipeline이 설명 돼있다. 초기에 받아온 frame은 1920x1080 크기이다. 여기서 도로 위가 아닌 하늘 부분은 전혀 사용되지 않으므로 하단의 2/3 부분의 이미지만 사용한다. 그리고 이를 256x128 로 resize 한다. 이때 opencv 의 cv2.resize 함수를 사용한다. 해당 이미지는 각 Detector 들의 결과를 그려내는데 사용 된다. (이하 template) 그리고 cv2.cvtColor 함수를 사용하여 이를 각각 gray, hsv scale 로 convert 한다. gray sclae 이미지는 하얀 선과 Object Detection을 위해 사용되며, hsv scale 이미지는 중앙선을 검출하기 위해 사용된다.

Graphical user interface, application

Description automatically generated

Figure . frame

2.2. Object Detection

input 으로 2.1. frame 에서 설명한 gray scale 이미지를 사용한다.

Haar Cascade Algorithm 을 사용하였다. pre-trained 된 haar cascade값을 사용하였으며 이는 다음 [github link](https://github.com/andrewssobral/vehicle_detection_haarcascades/blob/master/cars.xml)[[1]](#footnote-1) 의 open source file을 다운받아 사용하였다.

Detection 의 결과로 각 자동차들의 center point, x, y 값과 이미지에서의 width, height를 w, h 로 return 해준다. 이x, y, w, h 값들을 순서대로 (x, y, w, h) 로 담고 이를 list에 넣어 return 한다. Figure 3. 을 참고하라.

Graphical user interface

Description automatically generated

Figure . xywh

2.3. Lane Detection

input 으로 2.1. frame 에서 생성한 gray, hsv 이미지를 사용한다. 두 가지 이미지를 사용한 이유는 gray scale 만으로는 중앙선을 binarize 하기 어렵다고 판단되었기 때문이다. 따라서 흰색선들은 gray scale 로 검출하고, 중앙선을 hsv 이미지에서 cv2.inRange 함수를 사용하여 검출하였다. Figure 4. 를 참고하라.

Graphical user interface

Description automatically generated

Figure . lanes

그리고 이를 c2.warpPerspective 함수를 이용하여 Bird Eyed View (이하 BEV) 로 transform 한다. 이때 이미지의 size 는 128x32 이다. Figure 5. 를 참고하라.

Text

Description automatically generated

Figure . Bird Eyed View (BEV)

해당 이미지로 Lane Detection 과 Velocity Prediction 을 둘 다 수행한다. 우선은 Lane Detection을 먼저 설명하고 Velocity Prediction 은 이후에 설명하도록 하겠다.

먼저 차선은 왼쪽 차선과 오른쪽 차선만이 있다. 가운데 차선이나 3번째 차선은 없다. 이 말은 BEV 이미지에서도 모든 부분을 검토할 필요가 없고 양 옆 사이드에서만 차선을 검출해도 된다고 볼 수 있다. 이를 ROI (Region of Interest)로 보자. 그리고 이들을 여러 개의 window 로 나누어 각 윈도우 마다 차선으로 여겨질 pixel들의 좌표값들을 찾는다. 이때 이미지상 가장 가운데 있는 pixel이 차선이라고 생각할 수 있다. 그리고 차선의 좌표는 차량과 가장 가까이 있는 차선으로 결정하였다. Figure 6. 을 보자.

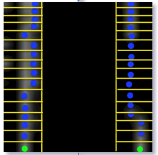


Figure . Finding lane pixels

노란색 박스는 각 왼쪽, 오른쪽 차선의 window 이다. 파란색은 각 윈도우마다의 이미지의 중앙에 가장 가까운 임의의 threshold의 값보다 큰 최초의 pixel 이다. 윈도우에 아무런 pixel 이 없다면 그 전 window의 pixel 의 위치를 따른다. 초록점은 최종적으로 Detector가 return 하는 왼쪽 차선과 오른쪽 차선의 좌표이다. 오른쪽 window들을 살펴보면 가운데는 선이 없음에도 점들이 찍혀 있는 것을 확인할 수 있다. 이는 차선이 보통 직선이기 때문에 차선으로 여전히 볼 수 있기 때문이다.

위와 같이 특정한 pixel 들을 찾기 위해서는 중요한 pixel들의 위치가 특정한 점으로 결정 되어야 한다. 이를 위해 CNN 구조에서 Pooling layer 의 아이디어가 차용하였다. Pooling 은 feature map 의 size 를 줄일 뿐만 아니라 중요한 feature 들을 적절히 모아주는 역할을 한다. 이는 연산량을 줄여줄 수 있고, pixel들을 분석하기 좋게 모아준다. 따라서 BEV 이미지를 height 는 window 의 개수, 그리고 width 는 적절히 16개정도로 결정하였고 resize 를 하였다. 이후에 구현할 때 window의 개수를 16으로 결정 하였다. 따라서 최종적으로는 16x16 크기의 이미지를 사용한다. 위 Figure 6. 도 따라서 16x16 이미지이다.

최종적으로 위 결과들을 다시 2.1. 에서 template에 다시 역으로 mapping 하여 라인들을 그려낸다. 이때 lookup table 을 사용한다. Figure 7. 을 참고하라.



Figure . Lane result

2.4. Velocity Predict

속도를 측정할 때는 input 으로 2.3. 에서 사용하였던 BEV 이미지를 사용한다. 따라서 구현에서는 해당 기능을 구현하는 class 를 Lane Detector class 가 포함한다.

먼저 Optical Flow Algorithm 을 사용하고자 하였지만, 상당히 비싼 연산이었고 그렇다고 성능이 좋은 것이 아니었다. 따라서 해당 프로젝트의 목적에 맞는 Algorithm 을 따로 구현하였다. 해당 Algorithm 은 BEV 이미지에서 점선 차선이 잘 검출되는 것과 차량이 주행 중이라는 것을 가정한다. 또한 점선 차선의 점선이 5m 라는 사실을 바탕으로 구현되었다.

핵심 아이디어는 점선 차선이 ROI image의 우하단의 가상의 line 을 지나는데 몇 frame이

Chart, radar chart

Description automatically generated

Figure . 점선 차선을 지나는데 걸리는 시간 측정

걸리는지 count 하는 것이다. Figure 8. 을 보면 가장 위의 이미지에서는 아직 점선 차선이 이미지의 우하단에 걸치지 않았기 때문에 아직 line 이 아니다. 그러나 2번째부터 9번째 이미지는 이미지의 우하단 끝에 라인이 걸쳐 있기 때문에 line 이고, 이를 count 한다.

그렇다면 해당 주행중인 차량은 5m 를 8 frame 만큼 주행하였다는 의미이고, 영상은 30FPS 이므로 1 frame 당 33ms 인 것을 알 수 있다. 따라서 frame 수만 구할 수 있다면 아래와 같은 식으로 속도를 계산할 수 있다.

위와 같이 8 frame 이 걸렸다면 아래와 같이 계산될 수 있다.

이는 계산하면 약 68.18km/h 로 계산된다.

2.5. Lane Departure

Lane Departure 판단은 이미지의 정중앙을 기준으로 각 왼쪽 차선, 오른쪽 차선의 거리의 비율로 결정하였다.

Graphical user interface

Description automatically generated with medium confidence

Figure . Lane Departure criteria

이때 ratio 는 아래와 같이 계산될 수 있다.

이때 r 이 0.5 이상이면 lane departure, 그렇지 않으면 정상 주행으로 간주한다. 아래 Figure 10. 은 Lane Departure 의 예시 이미지이다.

A picture containing text, scene, way, road

Description automatically generated

Figure . Lane departure example

2.6. Collision

Collision은 Object Detection 에서 차량의 정보인 xywh 를 이용한다. 이를 통해 이미지상에서 하단의 두 개의 점의 좌표를 계산할 수 있다.



Figure . Collision

Figure 11. 이미지의 빨간 박스 하단에 보이는 두 개의 초록점이 이미지상에서 두 점이다. 이들이 만약 BEV 이미지상에서의 좌표에서 임의로 정해진 danger zone 에 들어오게 된다면 Warning 을 한다.



Figure . danger zone in BEV

Figure 12. 는 BEV 이미지에 빨간색으로 임의로 정한 danger zone을 시각화 한 것이다.

2.7. Rapid accelerate & break

이는 [한국교통안전공단의 위험운전행동 기준](https://etas.ts2020.kr/etas/frtl0401/pop/goList.do) [[2]](#footnote-2)표를 보고 그대로 구현하였다. 프로젝트에서 구현하는 것은 급출발과 급감속이다.

Graphical user interface, text

Description automatically generated

Figure . 위험운전행동 기준표

화물차를 기준으로 하였다. 속도는 2.4. 에서 구하였으므로 이를 그대로 사용하였다.

2.8. Visualize

필요에 따라 Detection 혹은 Warning 이후 곧바로 결과를 이미지에 출력하거나 터미널에 출력하였다. 성능 지표인 FPS과 속도만은 가장 마지막에 출력하였다.

3. 최종 flow chart

Diagram

Description automatically generated

Figure . 최종 flowchart

III. 프로젝트 구현

1. 기능별 구현 내용

1.1. main.py 함수

전체 코드

import cv2

from detections.LaneDetector import LaneDetector, SpeedDetector

from detections.ObjectDetector import ObjectDetectionHaar

from myWarnings.Warning import Warning

import time

import numpy as np

def main():

video\_name = "/Users/joono/Desktop/joono/ComputerVisionADASProject/videos/highway\_D6\_Trim.mp4"

fps = 30

cap = cv2.VideoCapture(video\_name)

w = int(cap.get(cv2.CAP\_PROP\_FRAME\_WIDTH))

h = int(cap.get(cv2.CAP\_PROP\_FRAME\_HEIGHT))

laneDetector = LaneDetector(video\_name)

objectDetector = ObjectDetectionHaar()

warner = Warning(*lane\_warn\_threshold*=0.5, *video\_name*=video\_name)

while True:

ret, frame = cap.read()

if not ret:

break

tt = time.perf\_counter()

template = frame[round(h\*(1/3)):, :, :]

template = cv2.resize(template, (256, 128))

gray = cv2.cvtColor(template, cv2.COLOR\_RGB2GRAY)

hsv = cv2.cvtColor(template, cv2.COLOR\_RGB2HSV)

l\_x, r\_x, speed, acceleration = laneDetector(gray, hsv, template)

detections = objectDetector(gray, template)

warner.is\_lane\_departure(l\_x, r\_x, template)

warner.is\_collision(detections, template)

warner.is\_rapid\_stop\_or\_start(speed, acceleration, template)

tt2 = time.perf\_counter()

fps = 1 / (tt2-tt)

cv2.putText(template, f"{speed:.2f}km", (100, 12), cv2.FONT\_HERSHEY\_SIMPLEX, 0.5, (200, 255, 200), 2)

cv2.putText(template, f"FPS:{fps:.2f}", (0, 12), cv2.FONT\_HERSHEY\_SIMPLEX, 0.5, (255, 200, 200), 2)

cv2.imshow(f"ADAS", template)

k = cv2.waitKey(0)

if 27 == k:

break

# Press the green button in the gutter to run the script.

if \_\_name\_\_ == '\_\_main\_\_':

main()

각 코드 설명

ret, frame = cap.read()

if not ret:

break

tt = time.perf\_counter()

template = frame[round(h\*(1/3)):, :, :]

template = cv2.resize(template, (256, 128))

gray = cv2.cvtColor(template, cv2.COLOR\_RGB2GRAY)

hsv = cv2.cvtColor(template, cv2.COLOR\_RGB2HSV)

frame 을 받고 하단의 3/2 만을 사용하고 이를 256x128 로 resize 하고 gray, hsv scale 로 각각 convert 한다.

l\_x, r\_x, speed, acceleration = laneDetector(gray, hsv, template)

detections = objectDetector(gray, template)

lane detection 과 object detection 을 수행한다. speed 와 acceleration 은 Lane Detector 안에서 계산된다.

warner.is\_lane\_departure(l\_x, r\_x, template)

warner.is\_collision(detections, template)

warner.is\_rapid\_stop\_or\_start(speed, acceleration, template)

이후 warning 상황인지 아닌지를 판단한다.

tt2 = time.perf\_counter()

fps = 1 / (tt2-tt)

cv2.putText(template, f"{speed:.2f}km", (100, 12), cv2.FONT\_HERSHEY\_SIMPLEX, 0.5, (200, 255, 200), 2)

cv2.putText(template, f"FPS:{fps:.2f}", (0, 12), cv2.FONT\_HERSHEY\_SIMPLEX, 0.5, (255, 200, 200), 2)

cv2.imshow(f"ADAS", template)

현재 주행 속도와 FPS 를 계산하고 이를 256x128 이미지에 출력한다.

1.2. LaneDetector.py 코드 설명

import cv2

import numpy as np

class SpeedDetector:

def \_\_init\_\_(*self*) -> None:

*self*.*MAX\_TOLERANCE* = 3

*self*.*frame\_count* = 0

*self*.*tolerance* = 0

*self*.*coef\_of\_equ* = 3600 \* 5 / 33

def calc\_velocity(*self*, *frame\_count*):

if *frame\_count* == 0:

return 0

v = *self*.coef\_of\_equ / *frame\_count*

if v > 100:

return 0

else:

return v

def line\_frame\_count(*self*, *out*):

for i in range(LaneDetector.BEV\_WIDTH-LaneDetector.window\_width, LaneDetector.BEV\_WIDTH):

if *out*[LaneDetector.BEV\_HEIGHT-10][i] > 150:

# error condition

if *self*.frame\_count > 300000:

*self*.frame\_count = 0

*self*.frame\_count += 1

# velocity is not yet decided

return False, 0

else:

if *self*.tolerance < *self*.MAX\_TOLERANCE:

*self*.tolerance += 1

return False, 0

else:

*self*.tolerance = 0

frame\_count = *self*.frame\_count

if frame\_count != 0:

*self*.frame\_count = 0

return True, *self*.calc\_velocity(frame\_count)

else:

return False, 0

def \_\_call\_\_(*self*, *out*):

is\_calculated, velocity = *self*.line\_frame\_count(*out*)

if is\_calculated:

return velocity

class LaneDetector:

WARPAFFINE\_WIDTH = 256

WARPAFFINE\_HEIGHT = 128

BEV\_WIDTH = 32

BEV\_HEIGHT = 128

N\_WINDOWS = 16

window\_height = BEV\_HEIGHT // N\_WINDOWS

window\_width = 8

ROI\_WIDTH = 2

BEV\_POINTS = [

[0, 0],

[0, BEV\_HEIGHT],

[BEV\_WIDTH, 0],

[BEV\_WIDTH, BEV\_HEIGHT]

]

def \_\_init\_\_(*self*, *video\_name*: str):

# (x, y)순임. (y, x)순 아님

# LU -> LB -> RU -> RB

# 256 x 128 (W, H) image 기준

if *video\_name*.endswith("highway\_D6\_Trim.mp4"):

*self*.LANE\_ROI\_POINTS = [

[*self*.WARPAFFINE\_WIDTH // 2 - 25, 30],

[35, *self*.WARPAFFINE\_HEIGHT],

[*self*.WARPAFFINE\_WIDTH // 2 + 40, 30],

[*self*.WARPAFFINE\_WIDTH - 20, *self*.WARPAFFINE\_HEIGHT],

]

elif *video\_name*.endswith("highway\_D5\_Trim.mp4"):

*self*.LANE\_ROI\_POINTS = [

[*self*.WARPAFFINE\_WIDTH // 2 - 25, 30],

[30, *self*.WARPAFFINE\_HEIGHT],

[*self*.WARPAFFINE\_WIDTH // 2 + 25, 30],

[*self*.WARPAFFINE\_WIDTH - 30, *self*.WARPAFFINE\_HEIGHT],

]

*self*.M = cv2.getPerspectiveTransform(

np.array(*self*.LANE\_ROI\_POINTS, *dtype*=np.float32),

np.array(*self*.BEV\_POINTS, *dtype*=np.float32)

)

*self*.inv\_M = np.linalg.inv(*self*.M)

*self*.BEV2TEMPLATE\_LOOKUPTBL = []

# HSV yello

*self*.low\_yellow = np.array([0, 50, 0])

*self*.upper\_yellow = np.array([120, 255, 255])

for x in range(*self*.BEV\_WIDTH+1):

*self*.BEV2TEMPLATE\_LOOKUPTBL.append([])

for y in range(*self*.BEV\_HEIGHT+1):

bef\_coor = np.array([x, y, 1])

aft\_coor = np.matmul(*self*.inv\_M, np.transpose(bef\_coor))

*self*.BEV2TEMPLATE\_LOOKUPTBL[x].append(

[int(aft\_coor[0] \* (1 / aft\_coor[2])),

int(aft\_coor[1] \* (1 / aft\_coor[2]))]

)

*self*.BEV\_color = np.zeros((*self*.BEV\_HEIGHT, *self*.BEV\_WIDTH, 3))

*self*.closing\_kernel = cv2.getStructuringElement(cv2.MORPH\_ELLIPSE, (5, 5))

*self*.roi\_to\_BEV\_table = []

for row in range(*self*.N\_WINDOWS+1):

\_ = list()

\_row = 8 \* row

for l\_idx in range(0, *self*.BEV\_WIDTH // 2):

\_.append((2 \* l\_idx, \_row))

for r\_idx in range(*self*.BEV\_WIDTH // 2, *self*.BEV\_WIDTH+1):

\_.append((2 \* r\_idx + 1, \_row))

*self*.roi\_to\_BEV\_table.append(\_)

*self*.previous\_roi\_result = None

*self*.previous\_out = None

*self*.num\_pix\_of\_short\_line = 37

# length of white short line is 5m

*self*.meter\_per\_pixel = 5 / 37

*self*.previous\_optical\_line = *self*.BEV\_HEIGHT // 2

*self*.previous\_speed = 60

# self.speedMeter = SpeedMeter(speed=/60, threshold=10)

*self*.speedDetector = SpeedDetector()

*self*.cur\_speed = 0

*self*.acc\_frame\_count = 0

def roi\_lane\_detect(*self*, *roi\_points*, *template*=None):

roi\_result = np.zeros\_like(*roi\_points*)

l\_x, r\_x = 0, 0

l\_pts, r\_pts = [], []

# num of rois in a window

n\_rois = *self*.window\_width // *self*.ROI\_WIDTH

# total num of roi in a row

roi\_width = *self*.BEV\_WIDTH // *self*.ROI\_WIDTH

# find most inner lane

for i in range(*self*.N\_WINDOWS): # 16: N\_WINDOW

for l\_idx in range(n\_rois - 1, -1, -1): # 3, 2, 1, 0

if *roi\_points*[i][l\_idx] > 0:

roi\_result[i][l\_idx] = 1

x\_bev, y\_bev = *self*.roi\_to\_BEV\_table[i][l\_idx]

l\_pts.append(*self*.BEV2TEMPLATE\_LOOKUPTBL[x\_bev][y\_bev])

cv2.circle(*self*.BEV\_color, *self*.roi\_to\_BEV\_table[i][l\_idx], 2, (0, 255, 255), -1)

# last idx is left x

l\_x = l\_idx

break

for r\_idx in range(roi\_width - n\_rois, roi\_width): # 13, 12, 11, 10:

if *roi\_points*[i][r\_idx] > 0:

roi\_result[i][r\_idx] = 1

x\_bev, y\_bev = *self*.roi\_to\_BEV\_table[i][r\_idx]

r\_pts.append(*self*.BEV2TEMPLATE\_LOOKUPTBL[x\_bev][y\_bev])

cv2.circle(*self*.BEV\_color, *self*.roi\_to\_BEV\_table[i][r\_idx], 2, (0, 255, 255), -1)

# last idx is right x

r\_x = r\_idx

break

*self*.previous\_roi\_result = roi\_result

# two short lane has length of 3points (two short lane is a line)

if len(l\_pts) > 4:

x\_bev, y\_bev = *self*.roi\_to\_BEV\_table[*self*.N\_WINDOWS][l\_x]

pts = *self*.BEV2TEMPLATE\_LOOKUPTBL[x\_bev][y\_bev]

l\_pts.append(pts)

l\_x = pts[0]

else:

l\_x = None

if len(r\_pts) > 4:

x\_bev, y\_bev = *self*.roi\_to\_BEV\_table[*self*.N\_WINDOWS][r\_x]

pts = *self*.BEV2TEMPLATE\_LOOKUPTBL[x\_bev][y\_bev]

r\_pts.append(pts)

r\_x = pts[0]

else:

r\_x = None

if not *template* is None:

cv2.polylines(*template*, [np.array(l\_pts)], False, (255, 0 ,255), 2)

cv2.polylines(*template*, [np.array(r\_pts)], False, (255, 0 ,255), 2)

return l\_x, r\_x

def \_\_call\_\_(*self*, *gray*, *hsv*, *template*=None):

*self*.BEV\_color = np.zeros\_like(*self*.BEV\_color)

# center line (yellow line)

center\_line\_mask = cv2.inRange(*hsv*, *self*.low\_yellow, *self*.upper\_yellow)

# white line image

ret, thresh = cv2.threshold(*gray*, 180, 255, cv2.THRESH\_BINARY)

# combine yellow, white line image

out = cv2.add(thresh, center\_line\_mask)

# BEV image (256, 1024) (W, H)

out = cv2.warpPerspective(out, *self*.M, (*self*.BEV\_WIDTH, *self*.BEV\_HEIGHT))

out = cv2.normalize(out, None, 255, 0, cv2.NORM\_MINMAX, cv2.CV\_8UC1)

cv2.imshow("BEV", out)

# roi points temp

roi\_points = cv2.resize(out, (*self*.BEV\_WIDTH // *self*.ROI\_WIDTH, *self*.N\_WINDOWS))

l\_x, r\_x = *self*.roi\_lane\_detect(roi\_points, *template*)

speed = *self*.speedDetector(out)

if speed:

v\_delta = speed - *self*.cur\_speed

acceleration = v\_delta / (0.033 \* *self*.acc\_frame\_count)

*self*.cur\_speed = speed

*self*.acc\_frame\_count = 0

else:

acceleration = 0

*self*.acc\_frame\_count += 1

return l\_x, r\_x, *self*.cur\_speed, acceleration

def show\_BEV(*self*):

cv2.imshow("BEV", *self*.BEV\_color)

**SpeedDetector class 설명**

def \_\_init\_\_(*self*) -> None:

*self*.*MAX\_TOLERANCE* = 3

*self*.*frame\_count* = 0

*self*.*tolerance* = 0

*self*.*coef\_of\_equ* = 3600 \* 5 / 33

실제적으로 이미지에서 차선이 중간중간 끊겨 있는 부분이 있을 수 있다. 이를 몇 번까지 참는지에 대한 값이 MAX\_TOLERANCE 변수에 저장 돼있고, frame\_count 는 count 된 frame 을 나타낸다. tolerance 는 함수 내부적으로 몇 번째 참는 것인지에 대한 변수이다. coef\_of\_equ 는 frame 값만 가지고 바로 속도를 계산할 수 있도록 미리 계산해둔 값이다.

def line\_frame\_count(*self*, *out*):

for i in range(LaneDetector.BEV\_WIDTH-LaneDetector.window\_width, LaneDetector.BEV\_WIDTH):

if *out*[LaneDetector.BEV\_HEIGHT-10][i] > 150:

# error condition

if *self*.frame\_count > 300000:

*self*.frame\_count = 0

*self*.frame\_count += 1

# velocity is not yet decided

return False, 0

else:

if *self*.tolerance < *self*.MAX\_TOLERANCE:

*self*.tolerance += 1

return False, 0

else:

*self*.tolerance = 0

frame\_count = *self*.frame\_count

if frame\_count != 0:

*self*.frame\_count = 0

return True, *self*.calc\_velocity(frame\_count)

else:

return False, 0

해당 method 는 return 값으로 ret, velocity 를 return 한다. ret 은 속도가 계산되어 준비됐다는 변수이다.

오른쪽 window 의 pixel을 돌아가며 threshod (150) 이상의 값을 갖는 pixel 을 찾는다. 만약 있다면 frame\_count 를 1 증가시킨다. 그렇지 않으면 MAX\_TOLERANCE 만큼 참아보고 그렇지 않으면 해당 점선 차선이 끝났다고 판단하여 속도를 계산한다.

def calc\_velocity(*self*, *frame\_count*):

if *frame\_count* == 0:

return 0

v = *self*.coef\_of\_equ / *frame\_count*

if v > 100:

return 0

else:

return v

위에서 미리 계산해 둔 coef\_of\_equ 변수 값을 frame\_count 로 나누어 속도를 계산한다. 가끔 속도가 튀는 경우가 많은데 100 이 넘어가면 보통 에러 상황이므로 무시하도록 설정하였다.

**LaneDetector class 설명**

WARPAFFINE\_WIDTH = 256

WARPAFFINE\_HEIGHT = 128

BEV\_WIDTH = 32

BEV\_HEIGHT = 128

N\_WINDOWS = 16

window\_height = BEV\_HEIGHT // N\_WINDOWS

window\_width = 8

ROI\_WIDTH = 2

BEV\_POINTS = [

[0, 0],

[0, BEV\_HEIGHT],

[BEV\_WIDTH, 0],

[BEV\_WIDTH, BEV\_HEIGHT]

]

WAPAFFINE\_WIDTH, HEIGHT 는 template 으로 사용할 이미지의 크기를 정의해두었다. BEV\_WIDTH, HEIGHT 는 BEV 이미지의 width, height 값이다. N\_WINDOW 는 window 의 갯수이다. window\_height, width도 미리 계산해둔다. ROI\_WIDTH 는 window 를 다시 쪼개는데 이때 묶을 갯수를 의미한다. BEV\_POINTS 는 warpPerspective 에 사용할 Matrix 를 계산하는데 사용된다.

def \_\_init\_\_(*self*, *video\_name*: str):

# (x, y)순임. (y, x)순 아님

# LU -> LB -> RU -> RB

# 256 x 128 (W, H) image 기준

if *video\_name*.endswith("highway\_D6\_Trim.mp4"):

*self*.LANE\_ROI\_POINTS = [

[*self*.WARPAFFINE\_WIDTH // 2 - 25, 30],

[35, *self*.WARPAFFINE\_HEIGHT],

[*self*.WARPAFFINE\_WIDTH // 2 + 40, 30],

[*self*.WARPAFFINE\_WIDTH - 20, *self*.WARPAFFINE\_HEIGHT],

]

elif *video\_name*.endswith("highway\_D5\_Trim.mp4"):

*self*.LANE\_ROI\_POINTS = [

[*self*.WARPAFFINE\_WIDTH // 2 - 25, 30],

[30, *self*.WARPAFFINE\_HEIGHT],

[*self*.WARPAFFINE\_WIDTH // 2 + 25, 30],

[*self*.WARPAFFINE\_WIDTH - 30, *self*.WARPAFFINE\_HEIGHT],

]

*self*.M = cv2.getPerspectiveTransform(

np.array(*self*.LANE\_ROI\_POINTS, *dtype*=np.float32),

np.array(*self*.BEV\_POINTS, *dtype*=np.float32)

)

*self*.inv\_M = np.linalg.inv(*self*.M)

*self*.BEV2TEMPLATE\_LOOKUPTBL = []

# HSV yello

*self*.low\_yellow = np.array([0, 50, 0])

*self*.upper\_yellow = np.array([120, 255, 255])

for x in range(*self*.BEV\_WIDTH+1):

*self*.BEV2TEMPLATE\_LOOKUPTBL.append([])

for y in range(*self*.BEV\_HEIGHT+1):

bef\_coor = np.array([x, y, 1])

aft\_coor = np.matmul(*self*.inv\_M, np.transpose(bef\_coor))

*self*.BEV2TEMPLATE\_LOOKUPTBL[x].append(

[int(aft\_coor[0] \* (1 / aft\_coor[2])),

int(aft\_coor[1] \* (1 / aft\_coor[2]))]

)

*self*.roi\_to\_BEV\_table = []

for row in range(*self*.N\_WINDOWS+1):

\_ = list()

\_row = 8 \* row

for l\_idx in range(0, *self*.BEV\_WIDTH // 2):

\_.append((2 \* l\_idx, \_row))

for r\_idx in range(*self*.BEV\_WIDTH // 2, *self*.BEV\_WIDTH+1):

\_.append((2 \* r\_idx + 1, \_row))

*self*.roi\_to\_BEV\_table.append(\_)

# self.speedMeter = SpeedMeter(speed=/60, threshold=10)

*self*.speedDetector = SpeedDetector()

*self*.cur\_speed = 0

*self*.acc\_frame\_count = 0

영상마다 ROI 의 좌표가 조금씩 달라 영상마다 ROI 를 직접 설정해주었다. LANE\_ROI\_POINTS 가 해당 변수이다.

M 은 원본 이미지 -> BEV 이미지로 변환하기 위한 Matrix 이며, inv\_M 은 이의 inverse matrix 이다. BEV2TEMPLATE\_LOOBUPTBL 은 BEV 에서 다시 원본 이미지로 좌표를 mapping 하기 위한 lookup table 을 저장한다.

low\_yellow, upper\_yellow 는 hsv 이미지에서 중앙선을 검출하기 위한 범위를 정의한 것이다.

roi\_to\_BEV\_table 은 16x16 이미지에서 BEV 이미지로 mapping 하기 위한 lookup table 이다.

speedDetector는 speedDetector의 instance 이며, cur\_speed 는 현재 속도, acc\_frame\_count 는 가속도를 구할 때 사용할 count 할 변수이다.

def roi\_lane\_detect(*self*, *roi\_points*, *template*=None):

l\_x, r\_x = 0, 0

l\_pts, r\_pts = [], []

# num of rois in a window

n\_rois = *self*.window\_width // *self*.ROI\_WIDTH

# total num of roi in a row

roi\_width = *self*.BEV\_WIDTH // *self*.ROI\_WIDTH

# find most inner lane

for i in range(*self*.N\_WINDOWS): # 16: N\_WINDOW

for l\_idx in range(n\_rois - 1, -1, -1): # 3, 2, 1, 0

if *roi\_points*[i][l\_idx] > 0:

x\_bev, y\_bev = *self*.roi\_to\_BEV\_table[i][l\_idx]

l\_pts.append(*self*.BEV2TEMPLATE\_LOOKUPTBL[x\_bev][y\_bev])

# last idx is left x

l\_x = l\_idx

break

for r\_idx in range(roi\_width - n\_rois, roi\_width): # 13, 12, 11, 10:

if *roi\_points*[i][r\_idx] > 0:

x\_bev, y\_bev = *self*.roi\_to\_BEV\_table[i][r\_idx]

r\_pts.append(*self*.BEV2TEMPLATE\_LOOKUPTBL[x\_bev][y\_bev])

# last idx is right x

r\_x = r\_idx

break

# two short lane has length of 3points (two short lane is a line)

if len(l\_pts) > 4:

x\_bev, y\_bev = *self*.roi\_to\_BEV\_table[*self*.N\_WINDOWS][l\_x]

pts = *self*.BEV2TEMPLATE\_LOOKUPTBL[x\_bev][y\_bev]

l\_pts.append(pts)

l\_x = pts[0]

else:

l\_x = None

if len(r\_pts) > 4:

x\_bev, y\_bev = *self*.roi\_to\_BEV\_table[*self*.N\_WINDOWS][r\_x]

pts = *self*.BEV2TEMPLATE\_LOOKUPTBL[x\_bev][y\_bev]

r\_pts.append(pts)

r\_x = pts[0]

else:

r\_x = None

if not *template* is None:

cv2.polylines(*template*, [np.array(l\_pts)], False, (255, 0 ,255), 2)

cv2.polylines(*template*, [np.array(r\_pts)], False, (255, 0 ,255), 2)

return l\_x, r\_x

for 문에서는 각 window 를 돌며 차선이라고 예상되는 pixel 들을 찾는다. 찾은 pixel 들의 갯수가 4개가 넘지 않으면 차선을 검출하지 못했다고 판단한다. 16x16 이미지에서 점선 차선의 pixel 갯수는 3개이기 때문이다.

def \_\_call\_\_(*self*, *gray*, *hsv*, *template*=None):

# center line (yellow line)

center\_line\_mask = cv2.inRange(*hsv*, *self*.low\_yellow, *self*.upper\_yellow)

# white line image

ret, thresh = cv2.threshold(*gray*, 180, 255, cv2.THRESH\_BINARY)

# combine yellow, white line image

out = cv2.add(thresh, center\_line\_mask)

# BEV image (256, 1024) (W, H)

out = cv2.warpPerspective(out, *self*.M, (*self*.BEV\_WIDTH, *self*.BEV\_HEIGHT))

out = cv2.normalize(out, None, 255, 0, cv2.NORM\_MINMAX, cv2.CV\_8UC1)

cv2.imshow("BEV", out)

# roi points temp

roi\_points = cv2.resize(out, (*self*.BEV\_WIDTH // *self*.ROI\_WIDTH, *self*.N\_WINDOWS))

l\_x, r\_x = *self*.roi\_lane\_detect(roi\_points, *template*)

speed = *self*.speedDetector(out)

if speed:

v\_delta = speed - *self*.cur\_speed

acceleration = v\_delta / (0.033 \* *self*.acc\_frame\_count)

*self*.cur\_speed = speed

*self*.acc\_frame\_count = 0

else:

acceleration = 0

*self*.acc\_frame\_count += 1

return l\_x, r\_x, *self*.cur\_speed, acceleration

LaneDetector 를 호출하면 위와 같은 method 를 수행한다. 먼저 중앙선을 검출하고 이후에 하얀 차선을 검출한다. 이들을 더하고 BEV 로 transform 한 뒤 normalize 한다. 이후 16x16 이미지로 줄인 뒤 roi\_lane\_detect() method 를 호출하여 차선을 검출한다. 그리고 속도와 가속도를 계산한다.

1.3. ObjectDetector class 설명

**전체 코드**

import cv2

class ObjectDetectionHaar:

def \_\_init\_\_(*self*) -> None:

*self*.*haar\_xml* = "/Users/joono/Desktop/joono/ComputerVisionADASProject/detections/cars.xml"

*self*.*car\_cascade* = cv2.CascadeClassifier(*self*.haar\_xml)

def \_\_call\_\_(*self*, *gray*, *template*):

cars = *self*.car\_cascade.detectMultiScale(

*gray*,

*scaleFactor*=1.05,

*minNeighbors*=5,

*minSize*=(20, 20),

*flags*=cv2.CASCADE\_SCALE\_IMAGE

)

for (x,y,w,h) in cars:

cv2.rectangle(*template*, (x,y),(x+w,y+h),(0,0,255),2)

return cars

**코드 설명**

def \_\_init\_\_(*self*) -> None:

*self*.*haar\_xml* = "/Users/joono/Desktop/joono/ComputerVisionADASProject/detections/cars.xml"

*self*.*car\_cascade* = cv2.CascadeClassifier(*self*.haar\_xml)

haar\_xml 은 cars.xml file 의 경로를 저장한다. car\_cascade 는 opencv 의 CascadeClassifier instance 이다.

def \_\_call\_\_(*self*, *gray*, *template*):

cars = *self*.car\_cascade.detectMultiScale(

*gray*,

*scaleFactor*=1.05,

*minNeighbors*=5,

*minSize*=(20, 20),

*flags*=cv2.CASCADE\_SCALE\_IMAGE

)

for (x,y,w,h) in cars:

cv2.rectangle(*template*, (x,y),(x+w,y+h),(0,0,255),2)

return cars

gray scale 의 이미지가 input 으로 들어오면 haar cascade algotirhm 을 수행하고 그 결과를 xywh 의 list 로 return 하는데 이를 그대로 return 한다.

1.4. Warning class 설명

**전체 코드**

import cv2

import numpy as np

class Warning:

WARPAFFINE\_WIDTH = 256

WARPAFFINE\_HEIGHT = 128

BEV\_WIDTH = 32

BEV\_HEIGHT = 128

N\_WINDOWS = 16

window\_height = BEV\_HEIGHT // N\_WINDOWS

window\_width = 8

ROI\_WIDTH = 2

BEV\_POINTS = [

[0, 0],

[0, BEV\_HEIGHT],

[BEV\_WIDTH, 0],

[BEV\_WIDTH, BEV\_HEIGHT]

]

def \_\_init\_\_(*self*, *lane\_warn\_threshold*, *video\_name*) -> None:

*self*.*center\_of\_template* = 128

*self*.*maximum\_of\_distance\_mul* = *self*.center\_of\_template \* *self*.center\_of\_template

*self*.*lane\_warn\_threshold* = *lane\_warn\_threshold*

if *video\_name*.endswith("highway\_D6\_Trim.mp4"):

*self*.LANE\_ROI\_POINTS = [

[*self*.WARPAFFINE\_WIDTH // 2 - 35, 30],

[25, *self*.WARPAFFINE\_HEIGHT],

[*self*.WARPAFFINE\_WIDTH // 2 + 40, 30],

[*self*.WARPAFFINE\_WIDTH - 20, *self*.WARPAFFINE\_HEIGHT],

]

elif *video\_name*.endswith("highway\_D5\_Trim.mp4"):

*self*.LANE\_ROI\_POINTS = [

[*self*.WARPAFFINE\_WIDTH // 2 - 35, 30],

[20, *self*.WARPAFFINE\_HEIGHT],

[*self*.WARPAFFINE\_WIDTH // 2 + 30, 30],

[*self*.WARPAFFINE\_WIDTH - 20, *self*.WARPAFFINE\_HEIGHT],

]

# self.lr = linear\_model.RANSACRegressor()

*self*.M = cv2.getPerspectiveTransform(

np.array(*self*.LANE\_ROI\_POINTS, *dtype*=np.float32),

np.array(*self*.BEV\_POINTS, *dtype*=np.float32)

)

*self*.TEMPLATE2BEV\_LOOKUPTBL = []

for x in range(256 + 1):

*self*.TEMPLATE2BEV\_LOOKUPTBL.append([])

for y in range(128 + 1):

bef\_coor = np.array([x, y, 1])

aft\_coor = np.matmul(*self*.M, np.transpose(bef\_coor))

*self*.TEMPLATE2BEV\_LOOKUPTBL[x].append(

[int(aft\_coor[0] \* (1 / aft\_coor[2])),

int(aft\_coor[1] \* (1 / aft\_coor[2]))]

)

*self*.collision\_front\_danger\_distance = 0.5 \* *self*.BEV\_HEIGHT

*self*.collision\_side\_margin = 10

*self*.MAX\_RAPID\_TOLERANCE = 3

*self*.rapid\_start\_tolerance = 0

*self*.rapid\_stop\_tolerance = 0

# @staticmethod

def is\_lane\_departure(*self*, *l\_x*, *r\_x*, *template*):

if not *l\_x* is None and not *r\_x* is None:

left\_dist = abs(*l\_x* - *self*.center\_of\_template)

right\_dist = abs(*r\_x* - *self*.center\_of\_template)

if left\_dist > right\_dist:

ratio = right\_dist / left\_dist

else:

ratio = left\_dist / right\_dist

if ratio < *self*.lane\_warn\_threshold:

cv2.putText(

*template*,

"Lane!", (170, 25),

cv2.FONT\_HERSHEY\_SIMPLEX, 0.8, (0, 0, 255), 2)

return True

else:

return False

# @staticmethod

def is\_collision(*self*, *detections*, *template*):

for x, y, w, h in *detections*:

try:

x, y, w, h = int(x), int(y), int(w), int(h)

left\_p, right\_p = (x, y+h), (x+w, y+h)

left\_p\_BEV = *self*.TEMPLATE2BEV\_LOOKUPTBL[left\_p[0]][left\_p[1]]

right\_p\_BEV = *self*.TEMPLATE2BEV\_LOOKUPTBL[right\_p[0]][right\_p[1]]

x, y = left\_p\_BEV[0], left\_p\_BEV[1]

if *self*.collision\_side\_margin < x < *self*.BEV\_WIDTH - *self*.collision\_side\_margin \

and *self*.collision\_front\_danger\_distance < y:

cv2.putText(

*template*,

"Warn!", (170, 50),

cv2.FONT\_HERSHEY\_SIMPLEX, 0.8, (0, 0, 255), 2)

return True

else:

False

x, y = right\_p\_BEV[0], right\_p\_BEV[1]

if *self*.collision\_side\_margin < x < *self*.BEV\_WIDTH + *self*.collision\_side\_margin \

and *self*.collision\_front\_danger\_distance < y:

cv2.putText(

*template*,

"Warn!", (170, 50),

cv2.FONT\_HERSHEY\_SIMPLEX, 0.8, (0, 0, 255), 2)

return True

else:

False

except Exception as e:

print(e)

continue

else:

return False

def is\_rapid\_stop\_or\_start(*self*, *speed*, *acceleration*, *template*):

# 국토교통부 위험운전행동 기준

# 5km/h 이하 속도에서 출발하여 초당 6km/h 이상 가속 운행하는 경우

if *acceleration* >= 5 and *speed* <= 5:

*self*.rapid\_stop\_tolerance = 0

*self*.rapid\_start\_tolerance += 1

if *self*.rapid\_start\_tolerance >= *self*.MAX\_RAPID\_TOLERANCE:

print("Warn : Rapid Accelerate!")

cv2.putText(

*template*,

"RapStart", (170, 75),

cv2.FONT\_HERSHEY\_SIMPLEX, 0.8, (100, 255, 255), 2)

# 초당 8km/h 이상 감속 운행하고 속도가 6km/h 이상인경우

if *acceleration* <= -8 and *speed* >= 6:

*self*.rapid\_start\_tolerance = 0

*self*.rapid\_stop\_tolerance += 1

if *self*.rapid\_stop\_tolerance >= *self*.MAX\_RAPID\_TOLERANCE:

print("Warn! : Rapid Stop!")

cv2.putText(

*template*,

"RapStop", (170, 75),

cv2.FONT\_HERSHEY\_SIMPLEX, 0.8, (100, 255, 255), 2)

**코드 설명**

WARPAFFINE\_WIDTH, HEIGHT, BEV\_WIDTH, HEIGHT, N\_WINDOW, window\_heith, width, ROI\_WIDTH, BEV\_POINTS, video\_name, M 은 LaneDetection 에서 사용한 변수들인데 이들을 그대로 다시 사용하였다. 새롭게 추가된 코드들만 설명한다.

*self*.*center\_of\_template* = 128

*self*.*lane\_warn\_threshold* = *lane\_warn\_threshold*

*self*.TEMPLATE2BEV\_LOOKUPTBL = []

for x in range(256 + 1):

*self*.TEMPLATE2BEV\_LOOKUPTBL.append([])

for y in range(128 + 1):

bef\_coor = np.array([x, y, 1])

aft\_coor = np.matmul(*self*.M, np.transpose(bef\_coor))

*self*.TEMPLATE2BEV\_LOOKUPTBL[x].append(

[int(aft\_coor[0] \* (1 / aft\_coor[2])),

int(aft\_coor[1] \* (1 / aft\_coor[2]))]

)

*self*.collision\_front\_danger\_distance = 0.5 \* *self*.BEV\_HEIGHT

*self*.collision\_side\_margin = 10

*self*.MAX\_RAPID\_TOLERANCE = 3

*self*.rapid\_start\_tolerance = 0

*self*.rapid\_stop\_tolerance = 0

center\_of\_template 은 lane departure 에 사용될 중간 point 에 해당한다. Figure 9. 를 참고하라. lane\_warn\_threshold 는 ratio 가 해당 값보다 작아지면 warning 을 한다. TEMPLATE2BEV\_LOOKUPTBL 은 원본 이미지에서 검출한 image의 좌표를 BEV 로 mapping 하는 lookup table 이다.

collision\_front\_danger\_distance 는 danger zone 의 거리를 의미한다. collision\_side\_margin 은 danger zone 이 BEV에 딱 맞지 않게 여유를 주기 위한 변수이다.

MAX\_RAPID\_TOLERANCE 는 급감속, 급출발등을 판단할 때 frame 단위로 판단 되기 때문에 에러를 잡는 것이 필요하다. 따라서 급감속, 급출발을 MAX\_RAPID\_TOLERANCE 이상해야 급감속, 급출발을 했다고 판단한다. rapid\_start\_tolerance, rapid\_stop\_tolerance 는 해당 tolerance 를 count 하는 변수이다.

def is\_lane\_departure(*self*, *l\_x*, *r\_x*, *template*):

if not *l\_x* is None and not *r\_x* is None:

left\_dist = abs(*l\_x* - *self*.center\_of\_template)

right\_dist = abs(*r\_x* - *self*.center\_of\_template)

if left\_dist > right\_dist:

ratio = right\_dist / left\_dist

else:

ratio = left\_dist / right\_dist

if ratio < *self*.lane\_warn\_threshold:

cv2.putText(

*template*,

"Lane!", (170, 25),

cv2.FONT\_HERSHEY\_SIMPLEX, 0.8, (0, 0, 255), 2)

return True

else:

return False

lane departure 를 계산하는 method 이다. ratio 를 위에서 설명 했던 바와 같이 계산한다. 그리고 이를 영상에 출력한다.

def is\_collision(*self*, *detections*, *template*):

for x, y, w, h in *detections*:

try:

x, y, w, h = int(x), int(y), int(w), int(h)

left\_p, right\_p = (x, y+h), (x+w, y+h)

left\_p\_BEV = *self*.TEMPLATE2BEV\_LOOKUPTBL[left\_p[0]][left\_p[1]]

right\_p\_BEV = *self*.TEMPLATE2BEV\_LOOKUPTBL[right\_p[0]][right\_p[1]]

x, y = left\_p\_BEV[0], left\_p\_BEV[1]

if *self*.collision\_side\_margin < x < *self*.BEV\_WIDTH - *self*.collision\_side\_margin \

and *self*.collision\_front\_danger\_distance < y:

cv2.putText(

*template*,

"Warn!", (170, 50),

cv2.FONT\_HERSHEY\_SIMPLEX, 0.8, (0, 0, 255), 2)

return True

else:

False

x, y = right\_p\_BEV[0], right\_p\_BEV[1]

if *self*.collision\_side\_margin < x < *self*.BEV\_WIDTH + *self*.collision\_side\_margin \

and *self*.collision\_front\_danger\_distance < y:

cv2.putText(

*template*,

"Warn!", (170, 50),

cv2.FONT\_HERSHEY\_SIMPLEX, 0.8, (0, 0, 255), 2)

return True

else:

False

except Exception as e:

print(e)

continue

else:

return False

input 으로 들어온 detections list 들을 순회하며 각 좌표들을 lookup table 에 대응할 수 있도록 int 로 변환한다. left\_p, right\_p 는 각 detect 된 object 들의 bbox 의 하단 points 들이다. left\_p\_BEV, right\_p\_BEV 는 BEV 로 mapping 된 points 들이다. 그리고 이들이 임의로 정의해둔 danger zone 에 들어있는지 아닌지를 판단하고 그렇다면 영상에 Warn! 을 출력한다.

def is\_rapid\_stop\_or\_start(*self*, *speed*, *acceleration*, *template*):

# 국토교통부 위험운전행동 기준

# 5km/h 이하 속도에서 출발하여 초당 6km/h 이상 가속 운행하는 경우

if *acceleration* >= 5 and *speed* <= 5:

*self*.rapid\_stop\_tolerance = 0

*self*.rapid\_start\_tolerance += 1

if *self*.rapid\_start\_tolerance >= *self*.MAX\_RAPID\_TOLERANCE:

print("Warn : Rapid Accelerate!")

cv2.putText(

*template*,

"RapStart", (170, 75),

cv2.FONT\_HERSHEY\_SIMPLEX, 0.8, (100, 255, 255), 2)

# 초당 8km/h 이상 감속 운행하고 속도가 6km/h 이상인경우

if *acceleration* <= -8 and *speed* >= 6:

*self*.rapid\_start\_tolerance = 0

*self*.rapid\_stop\_tolerance += 1

if *self*.rapid\_stop\_tolerance >= *self*.MAX\_RAPID\_TOLERANCE:

print("Warn! : Rapid Stop!")

cv2.putText(

*template*,

"RapStop", (170, 75),

cv2.FONT\_HERSHEY\_SIMPLEX, 0.8, (100, 255, 255), 2)

만약 급출발에 해당 한다면 rapid\_stop\_tolerance 를 0으로 초기화한다. 반대로 급감속에 해당 하면 rapid\_start\_tolerance 를 0으로 초기화한다.

그리고 MAX\_RAPID\_TOLERANCE 이상으로 해당 event 가 반복되면 이를 영상에 출력한다.  
그러나 이는 굉장히 짧은 순간에 출력되기 때문에 terminal 에도 다시 출력한다.

IV. 실험 및 결과

1. 주어진 영상에 대해서 실험 결과를 수치와 graph, 이미지를 이용하여 표시

**실험 영상 링크**

[ADAS on jetson nano Trim\_D5](https://youtu.be/DohUFEVVo4o)

[ADAS on jetson nano Trim\_D6](https://youtu.be/gOn0Grd-rxw)

**Trim\_D5, Trim\_D6 에 대한 영상 내에서의 FPS 변화**

Chart, line chart

Description automatically generated

Figure . Trim\_D5 FPS

Chart

Description automatically generated

Figure . Trim\_D6 FPS

**Resources 사용량**

A screen shot of a computer

Description automatically generated with low confidence

Figure . 수행 시작 전 일부 usage

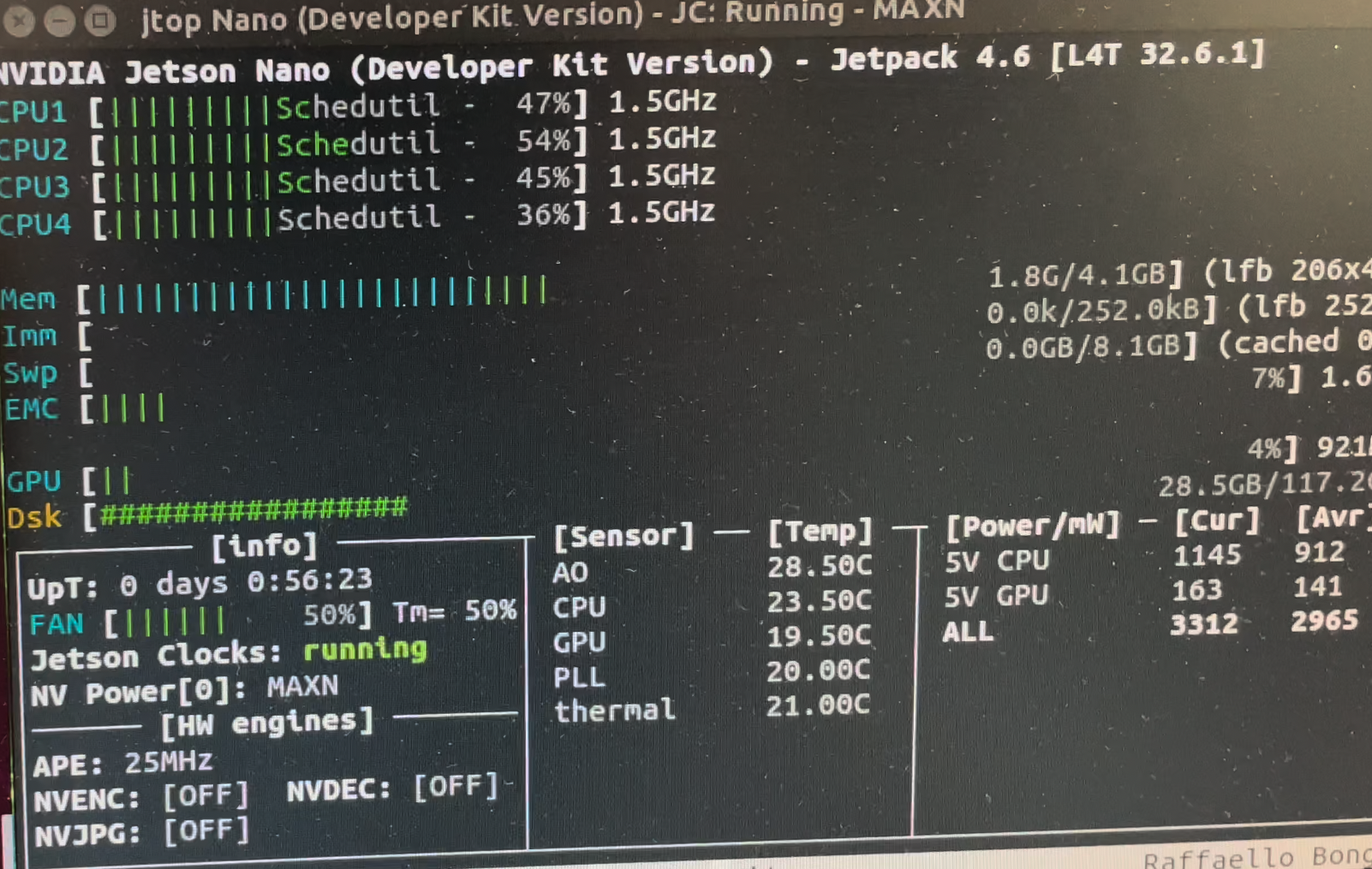


Figure . 수행 후 일부 usage

V. 결론

1. 프로젝트의 수행요약 및 결과

시스템에 대한 성능을 파악할 GT 가 존재하지 않으므로 실험 수행 영상으로 성능을 대체한다. 직선 구간, 속도가 어느 정도 빠른 경우에는 Lane Detection, Object Detection, Velocity Prediction 등 잘 수행하는 것으로 보인다. 그러나 ROI 를 벗어나는 곡선 구간, 교차로 구간, 혹은 빛 바랜 중앙선 등 예측 범위에서 벗어나는 경우를 만나면 예측이 틀리는 경우가 많다. 또한 속도를 측정하는 주기도 빠르게 업데이트되지 않아 급출발, 급감속을 제대로 검출하지 못한다. 또한 Trim\_D6 영상 에서 정지한 차 뒤로 바싹 붙는 경우가 있었는데 여기서는 Lane, Object, Velocity 그 어느 것도 제대로 제대로 탐지하지 못하였다.

위와 같은 것들이 해당 프로젝트를 진행하면서 아쉬웠던 점들이다.

하지만 두 영상의 FPS 를 기록한 Figure 15, 16. 을 보면 대부분 30 FPS 이상을 유지하였다. 두 영상에서 평균적으로 FPS 가 40을 넘겼다. Realtime 에 적합한 시스템을 구현하였다고 생각한다. 또한 memory도 100MB 정도만을 사용하였으며, GPU 는 전혀 사용하지 않았고 CPU usage 도 대부분 40-60% 정도를 사용하며 준수하게 수행되었다.

2. 프로젝트를 통해 배운 것

AI 만이 아니라 기존의 Computer Vision 기술로도 충분히 case 가 specific 하다면 ADAS 같은 시스템을 구축할 수 있을 거라 생각되었다. 오히려 충분한 preprocessing 과 더욱 엄격한 rules 등이 존재한다면 훨씬 빠르고 예측 가능하며 적은 resource 로도 주어진 task 를 해결할 수 있다고 생각되었다.

Computer Vision 과제를 수행할 때는 rule based 로 해결할지, model based 로 해결할지를 정할 때 이번 프로젝트를 수행한 경험이 큰 도움이 될 것 같다.

1. https://github.com/andrewssobral/vehicle\_detection\_haarcascades/blob/master/cars.xml [↑](#footnote-ref-1)
2. https://etas.ts2020.kr/etas/frtl0401/pop/goList.do [↑](#footnote-ref-2)