



yolov5 Lane Detection 코드 리뷰

<https://github.com/jkd2021/YOLOv5-Model-with-Lane-Detection>

detect.py

라이브러리 및 모듈 관련 코드

```
1  import argparse
2  import time
3  from pathlib import Path
4
5  from PIL import Image
6  import cv2
7  import torch
8  import torch.backends.cudnn as cudnn
9  from numpy import random
10 import numpy as np
11
12 from LD import LaneDetection
13 from models.experimental import attempt_load
14 from utils.datasets import LoadStreams, LoadImages
15 from utils.general import check_img_size, check_requirements, check_imshow, non_max_suppression, apply_classifier, \
16     scale_coords, xyxy2xywh, strip_optimizer, set_logging, increment_path
17 from utils.plots import plot_one_box
18 from utils.torch_utils import select_device, load_classifier, time_synchronized
```

- argparse 는 머신러닝 모델의 하이퍼 파라미터를 관리하는 모듈
- time 은 시간과 관련된 기능을 제공하는 모듈
- pathlib 모듈을 사용하면 파일, 디렉토리의 경로를 객체로써 조작/처리
- PIL(Pillow) 모듈은 파이썬 이미지 처리 라이브러리로 PIL 이미지 작업은 다음과 같은 기능을 수행한다.
 - 픽셀 단위의 조작
 - 마스킹 및 투명도 제어

- 흐림, 윤곽 보정 다듬어 윤곽 검출 등의 이미지 필터
- 선명하게, 밝기 보정, 명암 보정, 색 보정 등의 화상 조정
- 이미지에 텍스트 추가
- cv2 는 OpenCV 모듈로 영상 처리에 사용할 수 있는 라이브러리
- torch 는 딥러닝 연산에 사용되는 라이브러리
- torch.backends.cudnn 은 GPU 계산을 가속화 하기 위해 CUDA를 지원하는 라이브러리
- LaneDetection은 차선감지를 위한 고유 모듈
- models.experimental.attempt_load 은 차선감지를 위해 사전 훈련된 모델을 로드하는 함수
- 이하 코드를 위한 기타 유틸리티 함수들

LD.py

차선 검출을 위한 모듈 `LD / Line Detection` 코드

```
import cv2
import numpy as np

class LaneDetection:
    # some of the codes is functional with other project, if there are some misu

    # 이미지가 제공되지 않았을 때 결과값을 0으로 반환한다.
    def LD(picture = None):

        # lines : np.array([])
        # lines_Hough_al : np.array([])

        left_lines_fitted, right_lines_fitted = np.array([[0, 0], [0, 0]]), np.array([[0,
        # 이미지가 제공된 경우
```

if picture is not None:

```
# frame은 LaneDetection.app_canny 를 이용해 picture 을 grayscale 이미  
# 흑백으로 변환된 이미지는 명암 정보를 통해 차선과 같은 선분 추출이 더 쉬워진  
frame = LaneDetection.app_canny(picture)
```

```
# LaneDetection.mask 는 frame 이미지에 Canny 엣지 검출 알고리즘을 적용  
# 마스크를 통해 관심 영역인 차선 부분의 가장자리(윤곽선)를 "추출"  
mask = LaneDetection.mask(frame)
```

```
# 즉, frame은 Canny 엣지 검출이 적용된 그레이 스케일 이미지이고  
# mask 는 그 frame에서 차선 인식 영역을 나타내는 마스크 이미지이다.
```

```
# cv2.bitwise_and 를 이용해 이미지의 비트 연산을 수행  
# how ? - frame과 maske 두 이미지를 비트 단위로 AND 연산  
# masked_edge_img dpsms 두 이미지의 비트 연산 결과가 저장됨
```

```
masked_edge_img = cv2.bitwise_and(frame, mask)
```

```
# 결과적으로 masked_edge_img에는 원래 frame 이미지에서 마스크 이미지여  
# 마스크 외부 영역의 정보는 제거된다.
```

```
# 허프변환을 사용해 차선 후보 선분리를 수행  
# 기울기를 이용하여 왼쪽 차선과 오른쪽 차선을 분리하는 과정을 수행  
lines = cv2.HoughLinesP(masked_edge_img, 1, np.pi / 100, 15, minLine
```

```
# 위 조건을 만족하는 선분이 존재한다면  
if lines is not None:
```

```
    # 기울기가 -0.5보다 작을때 왼쪽 차선 후보로 간주하여 left_lines 리스트0  
    left_lines = [line for line in lines if -0.5 > LaneDetection.calculate_slo  
    # 기울기가 0.5보다 크면 오른쪽 차선 후보로 간주하여 right_lines 리스트에  
    right_lines = [line for line in lines if LaneDetection.calculate_slope(lir
```

```
# Remove noisy lines (이상치 제거)  
# 각각의 리스트에서 노이즈로 판단되는 선분을 제거한다.  
# 평균 기울기와 차이가 큰 선분을 제거하는 방식으로 구현
```

```

left_lines = LaneDetection.reject_abnormal_lines(left_lines, 0.1)
right_lines = LaneDetection.reject_abnormal_lines(right_lines, 0.1)

# Fit the left and right lane lines separately (최소 제곱 근사)
# 선분들의 좌표 정보를 이용하여 차선을 근사하는 직선의 시작점과 끝점 좌표
left_lines_fitted = LaneDetection.least_squares_fit(left_lines)
right_lines_fitted = LaneDetection.least_squares_fit(right_lines)

# 근사 결과가 없을 때 즉, 차선이 검출되지 않은 경우 왼쪽/오른쪽 도
if left_lines_fitted is None:
    left_lines_fitted = np.array([[0, 0], [0, 0]])
if right_lines_fitted is None:
    right_lines_fitted = np.array([[0, 0], [0, 0]])

# 근사 결과가 있을 때, 즉 차선이 검출되면 시작점과 끝점 좌표로 구성된
return left_lines_fitted, right_lines_fitted

return left_lines_fitted, right_lines_fitted

```

```

# slope_calculation (기울기 계산)
def calculate_slope(line):
    # line은 선분정보를 담고 있는 리스트이고, 시작점과 끝점의 x/y좌표값으로 구성
    x_1, y_1, x_2, y_2 = line[0] #리스트의 첫번째 요소인 가장 처음 검출된 선분만을 가

    # 우리가 흔히 아는 기울기 계산법 수행하여 반환 (y의 증가율/x의 증가율)
    # 이때, 0.01은 0으로 나눌때의 오류를 방지하기 위해 사용됨.
    return (y_2 - y_1) / (x_2 - x_1 + 0.01)

# Canny 엣지 검출 알고리즘을 적용하여 이미지에서 가장자리 정보를 검출
# Canny 엣지 검출을 적용할 이미지 데이터는 numpy 배열 또는 cv 이미지 객체 형태
def app_canny(picture):

```

```

img = picture

# 엣지 검출의 민감도를 조절하는 파라미터
minThreshold = 60 # 약한 엣지 검출
maxThreshold = 130 # 강한 엣지 검출

# 입력 이미지와 두 개의 임계 값을 받는다.
# cv2.Canny 를 이용하여 가장자리 정보만 남기도록 이미지 필터링
edges = cv2.Canny(img, minThreshold, maxThreshold)

#가장자리값을 리턴
return edges

# 자동차 전방 사다리꼴 모양의 마스크를 생성
def mask(frame):

    # np.zeros_like(frame) 함수를 이용하여 프레임과 동일한 크기의 검정색 이미지를
    mask = np.zeros_like(frame)

    # 그 마스크의 첫번째 원소가 검정색 이미지의 높이이고
    # 사다리꼴의 윗변 높이는 이미지 높이의 75%가 된다.
    height = mask.shape[0]
    height_bevel = int(0.75 * height)

    # 마스크의 두번째 원소가 검정색 이미지의 너비이고
    # 사다리꼴의 왼쪽/오른쪽 선분의 길이는 이미지 너비의 60%가 된다.
    # 사다리꼴의 아래 변 너비는 이미지 너비의 20%가 된다.
    length = mask.shape[1]
    length_bevel = int(0.6 * length)
    length_under = int(0.2 * length)

    # 위 연산을 통해 얻은 네 점 좌표를 이용하여 사다리꼴 모양의 윤곽선을 정의
    # cv2.fillPoly 함수를 이용하여 사다리꼴 모양으로 마스크 이미지를 채움
    mask = cv2.fillPoly(mask, np.array([[[length_under, height], [length - length_under, height],
                                         [length_bevel, height_bevel], [length - length_bevel, height_bevel],
                                         [length_under, height]]])),

```

```
color=255)
```

```
##### the size of mask and the parameters in cv2.fillPoly() needs
##### for different scenarios (camera pointing angle and orientati
##### to achieve the effect of recognition of the specified area in
# 코드를 다양한 상황에 적용하기 위해서는 마스크 크기(높이와 너비)와 cv2.fillPol
return mask
```

```
# line과 threshold를 인수로 받는 함수 정의
```

```
def reject_abnormal_lines(lines, threshold):
```

```
    # lines 리스트의 각 선분을 반복하여
```

```
    # slopes라는 새로운 리스트를 생성
```

```
    # 각 선분마다 LaneDetection.calculate_slope(line) 함수를 호출하여 기울기를 :
```

```
    # 그 계산값이 slope 리스트의 값으로 들어감
```

```
    slopes = [LaneDetection.calculate_slope(line) for line in lines]
```

```
# -----
```

```
    # slopes 리스트에 있는 기울기 값들을 반복적으로 확인하며 이상치를 제거하는 가
```

```
    # i = 0
```

```
    # while True:
```

```
    #     if i + 1 > len(slopes):
```

```
        # these codes equals to the p
```

```
    #         break
```

```
        # tuning in line 69 / 70
```

```
    #     if 0.5 > abs(slopes[i]):
```

```
    #         slopes.pop(i)
```

```
    #         lines.pop(i)
```

```
    #         i = i
```

```
    #     else:
```

```
    #         i += 1
```

```
# -----
```

```
    while len(slopes) > 0:
```

```
        # slopes 리스트에 있는 모든 기울기 값의 평균을 계산
```

```
        mean = np.mean(slopes)
```

```
        # slopes 리스트에 있는 각 기울기 값과 평균 기울기 값의 절대차를 계산
```

```

diff = [abs(slope - mean) for slope in slopes]

# 평균 기울기와 가장 차이가 큰 기울기 값이 위치한 인덱스
max_slope = np.argmax(diff)

# diff 리스트의 max_slope 인덱스 값이 설정된 임계값 즉, threshold 보다 큰지
# 만약 차이 값이 임계값보다 크다면, 해당 기울기를 이상치로 판단
# 이상치라고 판단되는 경우, slopes 리스트에서 해당 기울기 값을 제거
# 또, lines.pop(max_slope) 를 통해 해당 기울기 값에 연결된 선분도 lines 리스트에서 제거
if diff[max_slope] > threshold:
    slopes.pop(max_slope)
    lines.pop(max_slope)
else:
    break # 더이상 이상치가 없을때 종료

# 이상치가 제거된 lines를 반환
return lines

#즉, 위 코드는 차선 검출 알고리즘에서 line 리스트로부터 평균 기울기와 크게 차이

```

```

# least square fitting
# 입력된 여러 라인 세그먼트를 기반으로 최소 제곱 회귀를 사용하여 단일 라인의 끝점
def least_squares_fit(lines):
    """
    :param (line in lines): set of lines, [np.array([x_1, y_1, x_2, y_2]), [np.array
    :return: end points of a line, np.array([[xmin, ymin], [xmax, ymax]])
    """

    # 1. 라인 세그먼트들로부터 모든 좌표점 가져오기
    # np.ravel 함수를 사용하여 line 리스트의 값을 1차원 배열로 평평하게 만든다.
    # 이때, line 는 허프 변환을 통해 검출된 선분 정보를 담고 있는 리스트로,
    # [x1, y1, x2, y2] 에서 x1, y1는 선분의 시작점 좌표이고, x2, y2는 선분의 끝점 좌

    # 선분의 시작점 x 좌표와 끝점 x 좌표를 추출 -> 1차원 배열 x_coords에 순서대로
    x_coords = np.ravel([[line[0][0], line[0][2]] for line in lines])
    # 모든 선분의 시작점 y 좌표와 끝점 y 좌표를 추출 -> 1차원 배열 y_coords에 순서

```

```

y_coords = np.ravel([[line[0][1], line[0][3]] for line in lines])

# 2. 다항식 계수를 얻기 위해 직선 피팅을 수행
# 허프 변환을 통해 검출한 선분이 있을 때만 수행
if lines != None:

    # 모든 선분의 x좌표를 담은 x_coords 배열 길이가 1이상일 때
    # 즉, 선분에 대한 충분한 정보가 있을때
    if len(x_coords) >= 1:

        # np.polyfit 함수를 사용하여 x좌표,y좌표 데이터에 대하여 1차 다항식 최적화
        # 최적화된 다항식의 계수들이 poly에 저장
        poly = np.polyfit(x_coords, y_coords, deg=1)

        # 3. 다항식 계수를 기반으로 두 직선의 점을 계산하여 직선을 고유하게 결정
        # x_coords 배열의 최소값과 해당 x 좌표에서 다항식 poly의 값을 사용하여
        # 따라서, point_min 변수에는 직선의 시작점 (x,y) 좌표가 저장
        point_min = (np.min(x_coords), np.polyval(poly, np.min(x_coords)))

        # x_coords 배열의 최대값과 해당 x 좌표에서 다항식 poly의 값을 사용하여
        # 따라서, point_max 에는 직선의 끝점 (x,y) 좌표가 저장
        point_max = (np.max(x_coords), np.polyval(poly, np.max(x_coords)))

        # 위에서 얻은 직선의 시작점과 끝점 좌표를 numpy 배열로 저장
        return np.array([point_min, point_max], dtype=np.int)
    else:
        pass
else:
    pass

# 위 코드는 검출된 선분 정보를 기반으로 1차 다항식을 최적화하고
# 이 다항식을 사용하여 직선의 시작점과 끝점 좌표를 계산
# 반환된 numpy 배열은 차선 시각화 등에 사용될 수 있다.

```

```

## error report

```



```
class CustomError(Exception):
    def __init__(self,ErrorInfo):
        super().__init__(self)
        self.errorinfo=ErrorInfo
    def __str__(self):
        return self.errorinfo
```

- Canny 엣지 검출이란 1. 정확한 검출, 2. 정확한 위치, 3. 단일 에지 이 세가지 조건을 만족하는 에지 검출기로 윤곽선을 검출하는 알고리즘이다

원본



캐니 엣지 검출



- 허프 변환 하이퍼파라미터 설명 (`cv2.HoughLinesP` 함수)

`Hough Line Transform` 은 이미지에서 직선을 찾기 위해 사용되는 알고리즘으로 함수에 포함되는 하이퍼 파라미터는 다음과 같다.

- 1) `rho` : 허용 오차 거리를 픽셀 단위로 나타낸다.
- 2) `theta` : 허용 오차 각도를 라디안 단위로 나타낸다.
- 3) `threshold` : 직선으로 판단할 최소한의 동일 개수, 같은 직선에 몇개의 점이 등장해야 직선으로 판단할지를 나타내는 최소한의 개수
`threshold` 가 큰 값이면 정확도가 증가하지만 직선 검출 개수가 감소하고, 작은 값이면 정확도가 감소하지만 직선 검출 개수는 많아진다.
- 4) `minLineLength` : 선분으로 인정할 최소의 길이
- 5) `maxLineGap` : 동일 직선상의 선분들이 얼마 이하 떨어져있으면 연결할 것 인지 (즉, 어느 정도의 끊김일 때 동일 선분이라고 볼건지)

```
lines = cv2.HoughLinesP(masked_edge_img, 1, np.pi / 100,  
15, minLineLength=50, maxLineGap=10)
```

```
#허용 오차 거리 1픽셀  
#허용 오차 각도 1/100 = 0.01 라디안  
#최소 15개의 픽셀이 동일해야 선분으로 인정  
#50 픽셀 이상인 선분만 선분으로 검출  
#10픽셀 이상의 끊임이 있으면 선분으로 인정 X
```