실시간 보행자 인식 및 위치 표시

Homography를 이용한 영상 속 대상 위치 표시

상반기 베스텔라랩 인턴 장병희

목차

서론

1. 연구 목적

주차장 내에 실시간으로 동작 중인 CCTV가 여러 대를 이용하여 실시간으로 보행자를 검출하고 검출한 보행자 위치를 지도 위에 표시하여 자율 주행 자동차에게 전달한다. 이를 통해서 자율 주행 자동차는 주행 전에 주차장 내에 보행자 위치를 인식하고 사전에 보행자에게 가해질 위험을 최소화 할 수 있게 있도록 하는 개발 연구이다.

2.Homography

실시간 영상을 통해서 보행자를 인식하는 것은 yolov5를 사용한다. 이 때 사용되는 학습 가중치는 기본적으로 제공하는 가중치를 사용한다. 그 후 보행자를 인식한 결과를 이용하여 지도 위에 표시하기 위해서 Homography를 이용한다. Homography란 한 평면을 다른 평면에 투영시켰을 때 대응되는 점들 사이에 대응되는 일정한 변환 관계를 의미한다. 우리는 이를 이용하여 각 영상에서 표현되는 대응 점과 지도의 대응 점을 매칭하여 변환 관계를 찾아 보행자 위치를 지도 위에 표시 하였다.

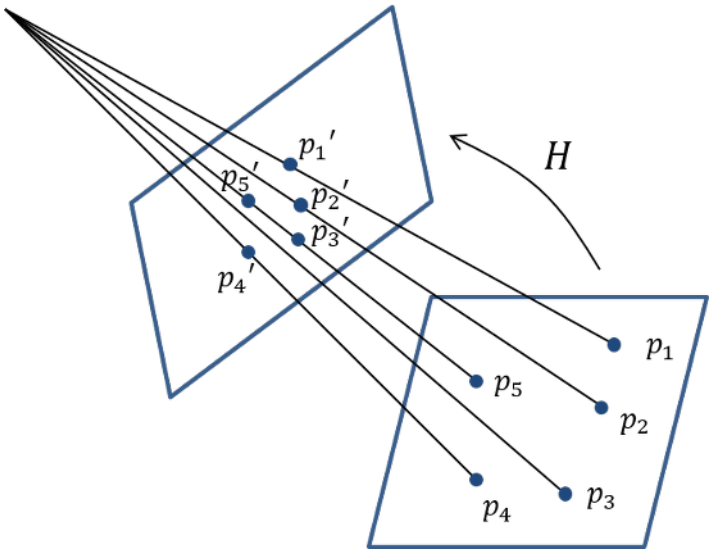

그림 1. Homography

따라서, 본 개발에서 23개의 CCTV와 하나의 지도에 대해서 대응점을 매칭하여 각 영상 별로 변환 관계를 계산하였으며, 영상별 변환 행렬은 config_hd_2.py에 정리되어 있다.

본론

1. 개발환경

OS	CPU	RAM	GPU
Linux Mint 20.2	AMD Ryzen 5 5600X 6-Core Processor	32GB	RTX 3080ti 12GB

개발 환경은 운영 체제는 우분투 기반에 리눅스 민트를 사용하였으며 CPU는 라이젠 5 5600X 메모리는 32기가를 사용하였으며 GPU는 3080ti를 사용하였다. 그래픽 카드를 제외한다면은 컴퓨터의 성능은 일반 가정용 PC와 차이가 없는 하드웨어 제원에서 개발하였다.

1. 개발코드분석

1) config_hd_2.py

```
cams = {
   'CCTV02': {
      'road': [[228, 64], [354, 62], [601, 570], [7, 551]],
      'src': "rtsp://admin:123456@172.10.14.12/",
      'uuid': [
         "a68712cb-0a2b-11ec-9c5c-4616e5ba6420",
         "a6b65ad6-0a2b-11ec-9c5c-4616e5ba6420",
         "a6b3847b-0a2b-11ec-9c5c-4616e5ba6420",
      ],
      'pos': [
         "1-y2-x6-1",
         "1-y3-x5-2",
         "1-y3-x5-1"
      ],
      'roi': [[648, 116, 775, 180], [850, 165, 950, 232], [1035, 174, 1167, 246]],
'homoMat':[[-0.3073809669587126, 7.99707279067324, 2702.7232410911593],[0.11904321736998807, 4.8732445695710584, 2445.109893443374],[8.205149899555664e-05, 0.003500582969557371, 1.0]]
   }
}
```

코드 1. config_hd_2.py(일부)

Config_hd_2.py 안에는 각 CCTV 별로 필요로 한 정보들을 저장하고 있다. cams 내에 각각 CCTV이름으로 key값이 존재하고 key 값에는 CCTV주소를 가지고 있는 'src' 및 CCTV별 Homography 변환 행렬을 가지고 있는 'homoMat'이 존재한다. 본 개발에서는 두 값을 이용하여 CCTV에 접속하고 CCTV에서 검출한 보행자 위치를 변환 행렬을 통해서 변환한다.

2) main_seq.py - getFrame

```
def getFrame(cctv_addr,cctv_name,return_dict):
    font = cv2.FONT_HERSHEY_SIMPLEX  # 글씨 폰트
    cap = cv2.VideoCapture(cctv_addr)
    while True:
        start_time = time.time()
        ret,frame = cap.read()
        end_time = time.time()
        print(f'Get {cctv_name} a frame - {round(end_time - start_time, 3)} s')
        if ret:
            return_dict['img'][cctv_name] = frame
        else:
            Error_image = np.zeros((720, 1920, 3), np.uint8)
            cv2.putText(Error_image, "Video Not Found!", (20, 70), font, 1, (0, 0, 255), 3)  # 비디오 접속 끊어짐 표시
            return_dict['img'][cctv_name] = Error_image
            #retry
            cap = cv2.VideoCapture(cctv_addr)
        k = cv2.waitKey(1) & 0xff
        if k == 27:
            cap.release()
            break
```

코드 2. main_seq.py – getFrame

소스 코드 내에서 main.py와 main_seq.py가 존재 하는데 본 개발에서는 main_seq.py를 다룬다. 그 이유는 최초 개발시 실시간성을 위해서 CCTV에서 영상을 얻어 yolo를 통해 보행자를 인식하는 것까지 하나의 프로세스로 생성해서 실행하였다. 그러나 본 개발에서는 사용할 수 있는 그래픽 카드(RTX 3080ti)가 하나이기에 프로세스가 증가함에 따라서 그래픽카드 메모리 부족을 초래한다. 그래서 영상에서 프레임만 병렬적으로 가져오고 추론은 sequential 하게 실행함으로서 그래픽 카드 메모리 부족 문제를 해결하였다. 따라서, 소스 코드 이름에 seq가 붙어 있다.

그래서 getFrame에 의해 CCTV영상 수 만큼 프로세스를 생성해서 병렬적으로 계속적으로 프레임을 가져온다. 그 후 return_dict[‘img’][‘저장하고자 하는 CCTV 이름’]안에 프레임을 저장한다. return_dict는 공유 변수로서 다른 프로세스들도 공유하는 변수이다. 이를 통해서 계속해서 새로운 프레임을 저장하여 실시간성 있는 추론을 가능하도록 한다. 또한, getFrame에서는 자신에게 주어진 변수에만 접근을 하기 때문에 비동기적으로 실행하더라도 문제가 발생하지 않는다.

3) main_seq.py - detect

```
def detect(return_dict):
    font = cv2.FONT_HERSHEY_SIMPLEX  # 글씨 폰트
    # yolov5
    # 로컬 레포에서 모델 로드(yolov5s.pt 가중치 사용, 추후 학습후 path에 변경할 가중치 경로 입력)
    model = torch.hub.load('yolov5', 'custom', path='yolov5s.pt', source='local', device=0)
    # 검출하고자 하는 객체는 사람이기 때문에 coco data에서 검출할 객체를 사람으로만 특정(yolov5s.pt 사용시)
    model.classes = [0]
    model.conf = 0.7
    window_width=320
    window_height=270
    # CCTV 화면 정렬
    for num,cctv_name in enumerate(cams.keys()):
```

```
        cv2.namedWindow(cctv_name)
        cv2.moveWindow(cctv_name,320*(num%6),270*(num//6))
    # CCTV 화면 추론
    while True:
        for cctv_name in cams.keys():
            # 추론
            img = return_dict['img'][cctv_name]
            start_time=time.time()
            bodys = model(img, size=640)
            end_time=time.time()
            print(f'yolov5 {cctv_name} img 추론 시간 - {round(end_time - start_time, 3)} s')
            flag = False
            points = []
            # yolo5
            for i in bodys.pandas().xyxy[0].values.tolist():
                # 결과
                x1, y1, x2, y2, conf, cls, name = int(i[0]), int(i[1]), int(i[2]), int(i[3]), i[4], i[5], i[6]
                cv2.rectangle(img, (x1, y1), (x2, y2), (0, 255, 0), 2)  # bounding box
                cv2.putText(img, name, (x1 - 5, y1 - 5), font, 0.5, (255, 0, 0), 1)  # class 이름
                cv2.putText(img, "{:.2f}".format(conf), (x1 + 5, y1 - 5), font, 0.5, (255, 0, 0), 1)  # 정확도
                # 보행자 좌표 표시
                target_x = int((x1 + x2) / 2)  # 보행자 중심 x 좌표
                target_y = int(y2)  # 보행자 하단 좌표
                # 보행자 픽셀 위치 표시
                img = cv2.circle(img, (target_x, target_y), 10, (255, 0, 0), -1)
                cv2.putText(img, "X:{} y:{}".format(target_x + 5, target_y + 5), (target_x + 10, target_y + 10), font,
                            0.5, (255, 0, 255), 1)
                # homography 변환
                target_point = np.array([target_x, target_y, 1], dtype=int)
                target_point.T
                H = np.array(cams[cctv_name]['homoMat'])
                target_point = H @ target_point
                target_point = target_point / target_point[2]
                target_point = list(target_point)
                target_point[0] = round(int(target_point[0]), 0)  # x -> left
                target_point[1] = round(int(target_point[1]), 0)  # y -> top
                points.append((target_point[0], target_point[1]))
                flag = True  # 변환된 정보 저장

            # 변환된 보행자 픽셀 위치 저장
            if flag:
                return_dict[cctv_name] = (flag, points)
            else:
                return_dict[cctv_name] = (False, [])
            temp_img = cv2.resize(img, dsize=(window_width, window_height))
            cv2.imshow(cctv_name, temp_img)
        send2server(return_dict)
        k = cv2.waitKey(1) & 0xff
        if k == 27:
            break
```

코드 3. main_seq.py – detect

본 개발 코드에서 제일 핵심이 되는 코드이다. 앞서 말한 것과 같이 cctv 영상은 병렬적으로 가져오지만 추론은 sequential하게 한다고 언급하였다. 위에 코드에서 보다시피 파이토치를 통해

로드 된 yolo모델 안에 img가 순차적으로 들어감을 확인할 수 있다. 그러나 입력되는 img는 추론되기 전까지는 계속해서 최신화 되어 입력 되어진다. 그 이유는 병렬적으로 이미지를 가져오는 프로세스들이 return_dict내에 계속해서 저장하고 있고 detect는 cctv별로 순차적으로 추론하지만 그 때 입력되는 이미지들은 return_dict안에 저장된 이미지를 사용하기 때문에 과거의 이미지를 사용하지 않는다. 그래서 sequtential하게 실행되더라도 최대한 실시간성을 보장할 수 있도록 하였다.
그 후 yolo에 의해 보행자를 인식하면 보행자를 인식한 bounding box 하단 중심점을 homography를 사용하여 지도 좌표로 변환한다. 그래서 이 값을 retun_dict에 저장하여 구해진 모든 좌표 정보를 지도 위에 표시하고 서버로 전송한다.

3) 조건별 성능 분석

현재 개발된 알고리즘을 통해서 한 영상을 전체적으로 걸리는 시간을 계산 하면은 아래에 수식과 같다. 하나의 CCTV 정보를 전달한다고 가정하였을 때 T_{cctv}에 대한 시간만 필요로 하다. T_{server}는 종합적으로 모든 데이터는 mqtt 방식으로 서버로 전송할 때 소요되는 시간이다. T_{yolo} 는 yolo에 의해 소요되는 시간이고 모든 CCTV에 대해서 추론을 하여야함으로 각각 CCTV 수에 따라서, 시간이 커지게 된다. 그래서 전체 시간은 CCTV에 의해 크게 의존적이라고 판단할 수 있다. 그래서 CCTV 변화에 따라서, 성능을 평가하였다.

$$T_{total} = T_{cctv} + T_{server} + \sum_{k=1}^{n} T_{yolo\_k}$$

CCTV영상은 RTSP방식을 통해서 가져오게 된다. RTSP는 TCP 방식을 사용하기 때문에 연결 상태를 계속해서 확인해야 함으로 여러 영상을 동시에 얻을 시 약간의 지연이 발생할 수 있다. 그래서 이 문제를 해결하기 위해 컴퓨터를 CCTV와 같은 네트워크 상에 설치하여 물리적인 지연을 최소화하여 기존 다른 서버를 거쳐 영상을 가져올 때에 비해서 빠른 영상 읽기가 가능하였다. 그럼에도 불구하고 CCTV 지연이 가장 큰 문제라고 판단되어 CCTV영상 수에 따른, 성능 평가를 한다.

CCTV 대수	영상 가져오기 (중앙 값)	영상 가져오기 (최대 값)	Yolo 추론 시간 (중앙값)	Yolo 추론 시간 (최대 값)
1	0.025s	0.102s	0.009s	0.013s
5	0.039s	0.286s	0.01s	0.017s
10	0.028	30.039s	0.009s	0.03s
15	0.004s	0.478s	0.011s	0.024s
23	0.004s	0.792s	0.014s	0.029s

표 1. CCTV 대수별 성능 시간 측정

성능을 평가하기 위한 시간 측정은 평균 값이 아닌 중앙 값을 사용하였다. 평균 값을 사용한다면 특정하게 너무 길어진 시간에 의해서 평균 시간이 너무 길어질 수 있기 때문이다. 따라서, 중앙 값으로 CCTV 대수별에 따른 시간 측정을 비교하였다. 비교 결과 CCTV 영상 가져오는데 연결된 CCTV대수가 증가함에 따라서 중앙 값으로는 더욱 짧은 처리 시간을 보였다. 또한, YOLO 같은 경우 중앙 값과 최대 값을 비교하였을 때 최대 0.02s 정도에 차이만 보였다. 그러나 영상을 가져오는 부분에서 최대 걸리는 시간이 30s까지 걸리는 문제가 발견하였다. 모델이 보행자를 인식하는데 최대 0.03s가 소요되는 반면 영상 자체를 가지고 오는 것에 지연이 되는 시간이 너무 큼을 알 수 있었다. 즉, 실시간으로 영상을 처리하기 위해서는 안정적인 영상 스트리밍이 관건이라고 확인 했다.

3.결론

1) 고찰

보행자 인식를 인식하고 이를 지도상에 표시하기 위해서 yolov5와 homography를 사용하였다. 또한, 실시간성을 위해서 영상을 읽고 이를 추론 과정을 병렬적으로 실행하여 최대한 실시간성을 보장하였다. 또한, 이 결과를 서버로 전송하여 어플리케이션에 표시하는 것까지 모두 끝마쳤으며 비교적 원활히 동작함을 확인 할 수 있었다. 이후 안정성에 대해서는 조금 더 관찰할 필요성이 보여진다.

그러나 기존에 실시간성을 위해 yolo의 추론 시간이 느리다고 판단하여 Tensorrt등 하드웨어를 가속화하여 최대한 시간을 단축 시키고자 하였다. 그러나 이번 보행자 인식을 개발하면서 성능을 평가함으로서 yolo에 추론 시간보다는 CCTV를 스트리밍 해오는 시간이 가장 큰 bottleneck임을 수치적으로 확인할 수 있었다. 그래서 이후 원활하고 실시간성을 보완하고자 한다면은 스트리밍 지연 시간을 최소화는 것이 필요로 해보인다.

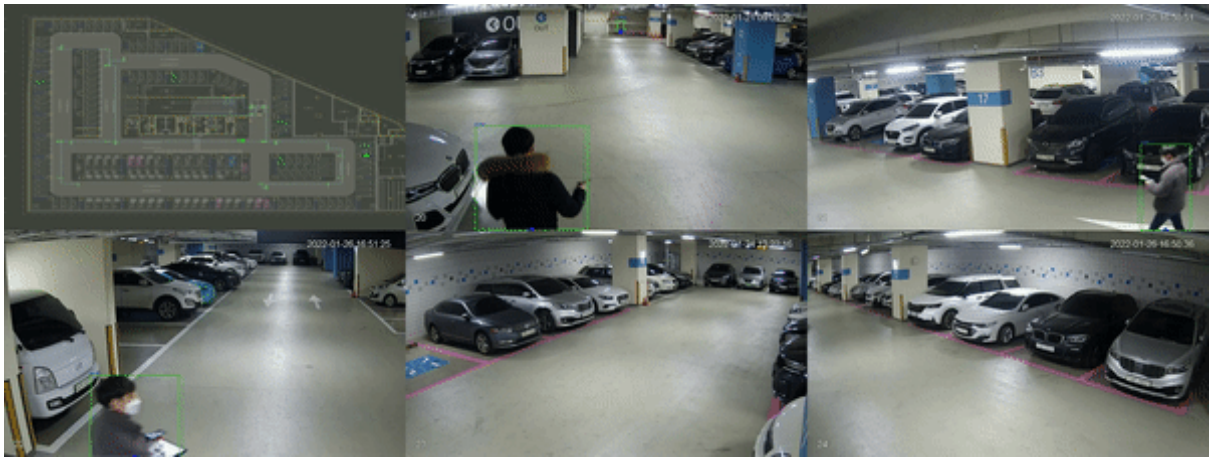

그림 1, 구현 결과