|  |
| --- |
| **2020년 2학기 캡스톤 디자인 2 최종 결과 보고서**  **실내 문서 전달 자율주행 카트**  **클라우드 기반 영상처리 모듈 개발**  **An Implementation of Object Recognition and**  **Collision Avoidance Modules for Indoor Self-driving Cart**  **Based on Cloud Image Processing Module**    **컴퓨터 공학과 이재빈**  **지도교수: 허의남**  INDEX  **1. 서론**  1.1연구배경  1.2 연구 목표  **2. 관련 연구**  2.1. 클라우드 환경에서의 분산 이미지 처리 프레임워크  2.2. AWS Lambda Serverless Computing 기술을 활용한 효율적인 딥러닝 기반 이미지 인식 서비스 시스템  2.3. 클라우드 서버 환경에서 Lidar 센서와 Yolo v3 라이브러리를 이용한 360도 영상 인식 시스템 구현  2.4. 관련 연구 분석  **3. 프로젝트 내용**  3.1. 주행 시나리오 및 모듈 별 기능  3.1.1. 구현 환경  3.1.2. 시나리오  3.1.2.1. 명패 인식 (Door Plate Number Detection) 시나리오  3.1.2.2. 사람 인지 (Human Object Recognition) 시나리오  3.2. 요구사항 및 해결 방안  3.2.1. 인식 시간  3.2.2. 객체 인지  3.2.3. 명패 (Door Plate Number Tag) 정보 인식  3.2.4. 명패 번호 (Door Plate Number) 인식용 데이터셋 활용  3.3. 시스템 설계  3.3.1. UML Diagram을 통한 시스템 모델링  3.3.1.1. Activity Diagram  3.3.1.2. Sequence Diagram  3.4 구현 과정  3.4.1. 개발 환경  3.4.2. Door Plate Crop모듈 개발  3.4.3. Noise Decision Model 구축  3.4.4. Number Recognition Model 구축  3.4.5. Door Plate Recognition 모듈 통합  3.4.6. Human Recognition 모듈 개발  4. 프로젝트 성능 개선  4.1. 분산환경 구축 전  4.2. 분상환경 구축 후  5. 결론 및 향후 연구  6. 참고 문헌  **초 록**  최근 들어, 고 성능, 저지연의 서비스 수요가 증가할 뿐 아니라 다양한 클라우드 환경이 구축되며 클라우드 컴퓨팅에 대한 관심이 높아지고 있다. 뿐만 아니라, 분산클라우드 환경은 네트워크 전체의 부하를 줄일 수 있음은 물론, 전체적인 서비스 퍼포먼스 향상을 이끌 수 있는만큼 효용성이 높다. 이에 따라, 실내 자율주행 카트의 위치파악을 위해, 영상처리 모듈을 엣지-코어 클라우드의 구조로 구현하며, 최종적으로 영상처리 모듈의 퍼포먼스 향상, 기존 연구에서의 병목현상 해결을 목표한다. |

**1. 서론**

**1.1. 연구배경**

5G network 환경과 더불어 머신 러닝과 같은 SW기술발전으로 인해 점차적으로 자율 주행 기술의 점진적 변화가 예상된다. 미국 자동차 학회(SAE)는 자율 주행의 구현 정도에 따라 자율주행 기준 발전단계를 마련하였고, 현대자동차는 2022년 까지 주행관련 정보를 판단하고 행동을 결정할 수 있는 레벨3단계를 대부분 구축할 것으로 전망하고 있다. 이처럼, 실외용 Autonomous Vehicle의 연구가 현재 활발히 연구 개발되고 있으나, 도로 위가 아닌 실내의 경우 또 다른 문제점에 직면하게 된다. 먼저 Guide Line인 차선이 없다는 점과 더불어, 비좁은 통로가 존재하고 사람들의 왕래가 실내에서는 더욱 잦다는 점을 감안하면 단순히 위치정보만을 활용한 자율주행카트 구현에는 무리가 있다. 때문에, 실내 자율 주행 카트의 경우 영상처리 모듈이 필수적이다. 하지만 영상처리 모듈의 경우 카트에 위치한 컴퓨팅 환경의 성능적 제약요건이 발생하게 되어, 실질적 주행에 있어서 어려움이 있다.

이에 따라 본 연구는 Core Cloud와 Edge Cloud간의 분산 컴퓨팅을 구현하여 전체적인 네트워크 부하 분산을 목표한다. 최종적으로 시간대비 영상처리 모듈의 퍼포먼스 향상, 기존 연구에서의 병목현상을 해소할 수 있도록 한다.

**1.2. 연구목표**

최근 네트워크 토폴로지의 규모가 점차적으로 확대되면서, 클라우드 컴퓨팅환경의 실시간, 저 지연, 분산 컴퓨팅 등의 서비스 요구사항이 점차 늘어나고 있다.

특히, 지난 연구의 경우, 카트에 위치한 에지 클라우드에서 영상처리를 수행할 경우 컴퓨팅 환경의 제약으로 부족한 퍼포먼스가 도출되었다.

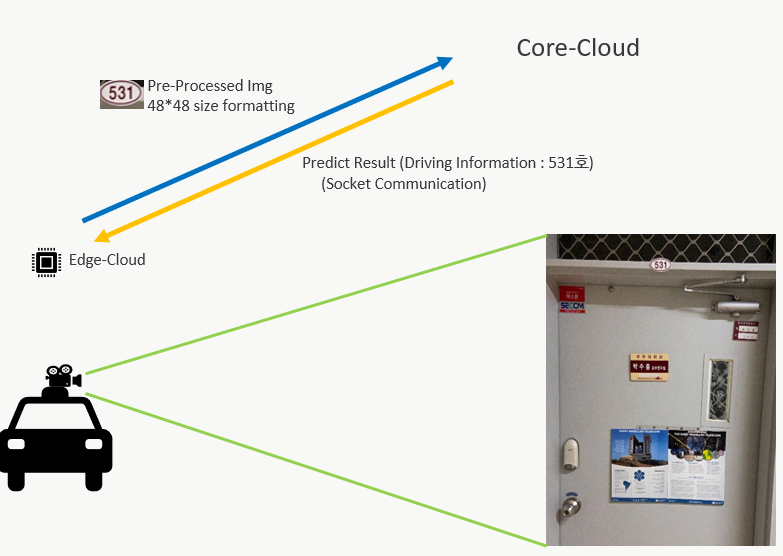
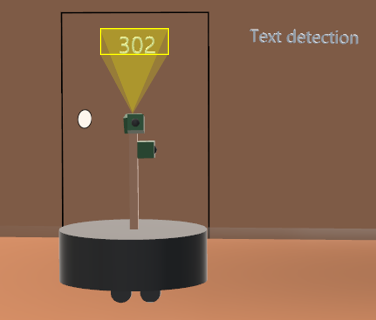
이에 따라 코어클라우드와 에지클라우드간 통신환경을 구축하였으나, 이분화된 모듈간의 성능 격차로 실시간 통신 과정에서 병목현상이 발생하였다. 이 두 점을 미뤄보아 모듈의 세분화, 단일 프로토콜을 통한 통신과정이 필요하였다.

따라서 본 연구에서는 카트에서 수집된 영상 원본 프레임을 기존 구현된 머신러닝 모델의 input형식에 맞추어 전처리 작업을 진행한다. 이후 전 처리된 데이터만을 Core Cloud에 송신하며, Core Cloud는 사전에 로드된 머신러닝 모델을 통해 예측값을 반환한다. 결과적으로 Edge-Cloud는 예측값만을 수신하게 된다.

이를 통해 최종적으로 네트워크 전체의 부하를 분산하여 고성능 컴퓨팅환경을 구축하고, 시간대비 영상 처리모듈의 퍼포먼스 향상, 기존 연구의 문제점인 병목현상 해소를 목표한다.

* 명패 인식[Text Recognition]모듈
* 명패인식모듈은 카트의 양 측면에 부착된 카메라를 통해 카트의 정확한 위치를 파악하는 기능을 담당한다. 실내의 특성상, GPS사용이 어렵다는점, 다양한 전파들의 간섭이 잦은만큼 클라우드에의해 구체적인 위치를 추정하기가 힘들다는 점을 고려하여, 양측면 카메라를 사용하여 카트의 구체적인 위치를 파악하도록 구현하였다.
* 구체적인 동작 프로세스는 다음과 같다.

1. 카트의 양측면에 달린 카메라를 통해 명패(Door Plate)를 추출하여 코어클라우드로 전송한다,
2. 코어클라우드는 사전 학습된 머신러닝 모델을 통해 현재 카트의 위치가 어디인지를 예측한 후 에지클라우드(카트)로 카트의 현재 주행 위치를 전송한다.
3. 에지클라우드(카트)는 예측된 값을 바탕으로 제어모듈에 주행정보를 전송한다.



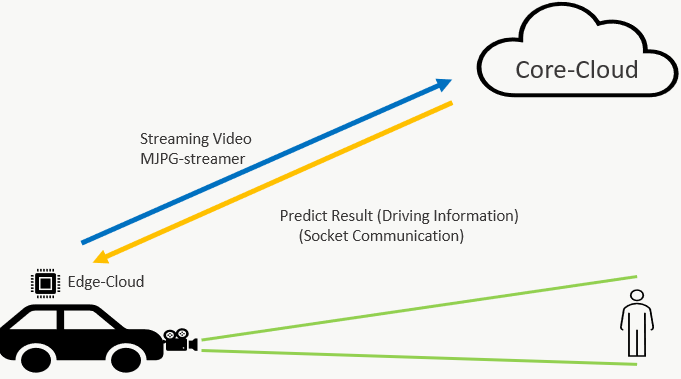
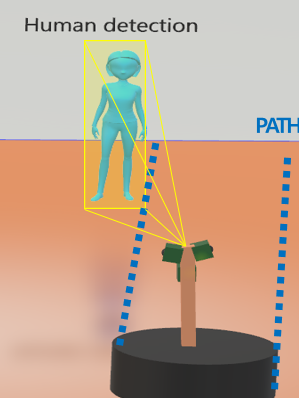
[그림1 명패인식모듈 개념도]

* 사람 인지[Human Recognition]모듈
* 사람인지모듈은 카트 전방에 부착된 카메라를 통해 사람객체를 인식할 수 있도록 구현된다. 또한 사람을 회피안 예상 경로를 출력하고, 이를 1차원 형태의 배열로 변환, 제어모듈로 주행정보를 전송한다.
* 구체적인 동작 프로세스는 다음과 같다.

1. 카트 전방의 카메라영상을 코어클라우드로 스트리밍한다.

(이때 감지 최소범위는 5m이상으로, 5m이내의 사람, 장애물은 LIDAR센서를 활용하여 탐지함)

1. 코어클라우드는 수신받은 영상을 바탕으로 사람을 인지하며, 카트 정면기준 화면을 10등분 하여, 현재 장애물의 위치를 포함한 주행정보를 에지클라우드(카트)로 반환한다.
2. 장애물을 회피하는 예상 주행경로를 화면에 출력한다.



[그림 2 사람 인지 모듈 개념도]

**2. 관련 연구**

**2.1. 클라우드 환경에서의 분산 이미지 처리 프레임워크**

Google Cloud Platform을 활용하여 분산 환경을 기반으로 이미지를 분산 저장하고 Map Reduce 프레임 워크를 구축하였다. 또한 객체 식별 알고리즘중 하나인 SLIC를 적용하여 고해상도 이미지의 분산처리를 구현하였으며 객체 식별과 같은 이미지 처리 알고리즘에도 해당 프레임워크의 적용가능성을 검증하였다. 그러나 실시간 처리가 중요한 자율주행 카트의 경우 빠른 송수신 시스템이 보장되어야 한다는 점, 해당 연구가 Google Cloud Platform에 의존성이 있다는 점을 고려할 때, 본 연구를 자율주행카트에 직접 적용하기에는 한계가 있다.

**2.2. AWS Lambda Serverless Computing 기술을 활용한 효율적인 딥러닝 기반 이미지 인식 서비스 시스템**

AWS Lambda Serverless Computing 기술을 활용하여 비용 절감뿐만 아니라 처리 시간 및 용량제한 문제를 해결하여 대형 신경망 모델을 서비스하기에 효율적인 성능을 확인하였다.

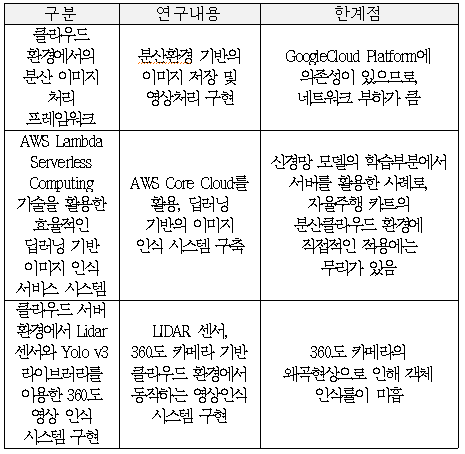
AWS의 Core Cloud서버를 활용하여 보다 효율적인 서비스를 제시하여 인프라 설계 단계의 비용, 처리 성능 등을 향상하였지만 신경망 모델의 학습부분에서 서버를 활용한 사례로, 자율주행 카트의 분산클라우드 환경에 직접적인 적용에는 무리가 있다.

**2.3. 클라우드 서버 환경에서 Lidar 센서와 Yolo v3 라이브러리를 이용한 360도 영상 인식 시스템 구현**

LIDAR 센서와 360도 카메라를 활용하여 클라우드 환경에서 동작하는 영상인식 시스템을 구현하였다. LIDAR센서 에서 움직임이 감지될 경우 녹화를 진행하는 방식으로 시스템 부하를 줄였으며, AWS EC2서버를 활용하여 클라우드 환경을 구축하였다. 다만, 영상 밝기의 영향이나 360도 카메라의 왜곡현상으로 인해 객체 인식률이 미흡하다는 한계점이 잔존한다.

**2.4. 관련 연구 분석**

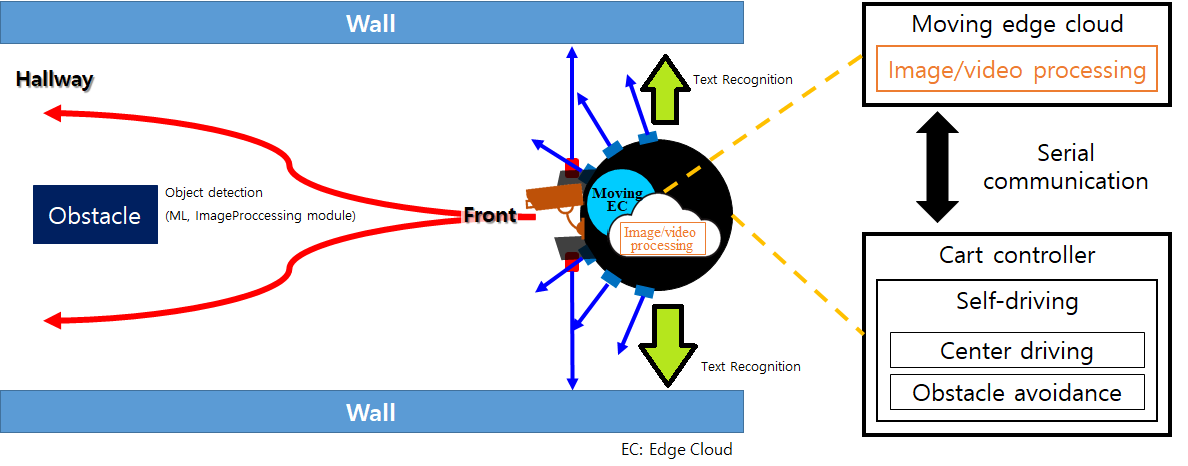
앞서 보듯, 관련 기존 연구는 Google Cloud Platform, AWS Core Cloud Service와 같이 기존 플랫폼에 의존성을 갖고 있어 실시간 영상처리에 있어서 높은 지연율을 보였다. 따라서 본 연구는 별도의 플랫폼이 아닌 기존 구현된 Core Cloud를 활용하여 전체 네트워크의 부하를 줄임과 동시에, Core Cloud와 Edge Cloud간의 병목현상을 최소화 하여 실내 환경에 특화된 보다 효율적인 자율주행 카트를 제시한다.



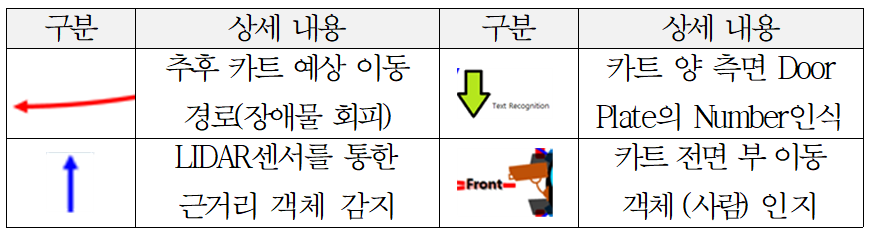
[표1] 기존 연구 분석

**3. 프로젝트 내용**

**3.1. 주행 시나리오 및 모듈 별 기능**



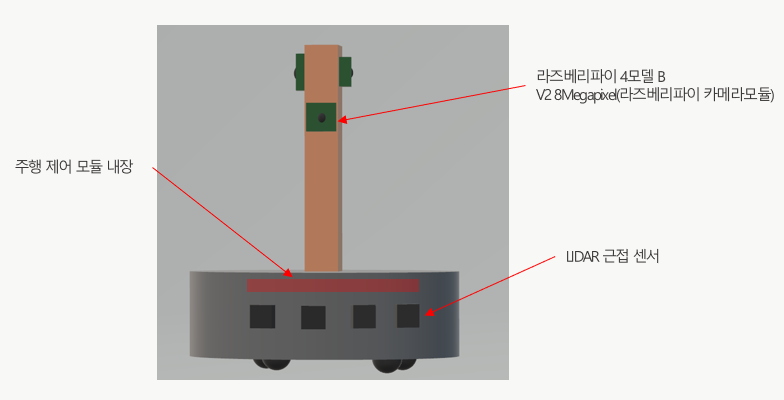
[그림 3] 주행 개념도 (위)



[표 2] 주행 개념도 기호

1. 기기 구성

자율주행 카트에 LIDAR센서를 전방, 양 측면에 부착하여 근거리에 대한 물체감지가 이루어진다. 또한, 카트의 위치 정보는 각 클라우드의 엣지에 의해 세밀화 된다. 다만 클라우드의 엣지만으로는 카트의 실내 위치를 정확히 추론할 수 없음과 LIDAR센서의 감지 거리 한계를 고려하여, 다음 두 모듈을 통해 실내의 위치를 정확하게 추론하고 원거리의 물체를 회피할 수 있도록 한다.



[그림 4] 기기 구성도

1. Edge Cloud 영상처리 모듈

영상처리 모듈은 카트에 부착된 전면카메라와 양 측면 카메라를 활용한다. Edge Cloud 영상처리 모듈은 전면 카메라, 양 측면 카메라의 영상을 코어클라우드로 전송하며. 전면카메라는 사람인지를, 양 측면 카메라는 실내의 정보파악을 위한 Door Plate Number 추출 기능을 위해 동작한다.

1. Core Cloud Prediction 모듈

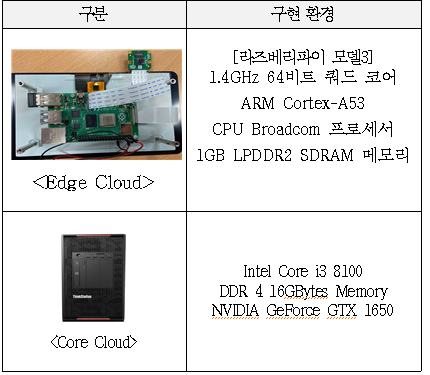
사람 인지의 경우, 스트리밍 되는 카트의 영상을 바탕으로 원거리에 사람이 있는지를 예측한다. 이후, 예측된 결과값을 카트로 전송하여 카트 주행이 원활하도록 한다.

명패인지모듈의 경우, 카트 양측면으로부터 수신받은 전처리 데이터를 바탕으로 노이즈를 필터링하며, 실내 호수를 파악하여 현재 카트의 위치를 송신하며, 이는 카트제어에 이용된다.

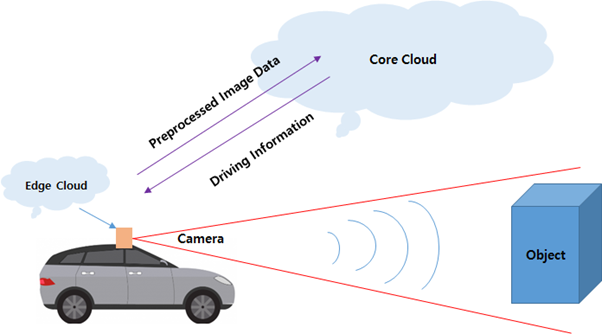
**3.1.1. 구현 환경**

Edge Cloud의 Object Recognition and Collision Avoidance 모듈의 구현 환경은 라즈베리파이모델3에 웹캠을 부착하여 구성되었으며 라즈베리파이의 하드웨어 사양은 다음과 같다. 또한, 카트의 동작 범위는 좁은 복도가 있고, 명패(Door Plate Number)가 있는 실내 환경으로 한정한다. Core Cloud의 구현 환경은 다음과 같으며, 원격지에 위치하여 Edge Cloud로부터 수신 받는 데이터를 해독하여, 주행정보를 판단, Edge Cloud로 주행정보를 송신한다

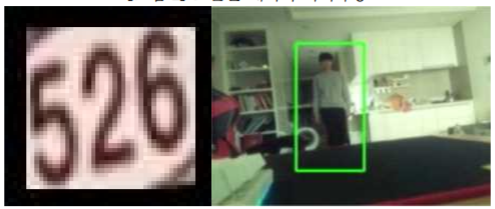
[표 3] 주요 모듈 개발 환경



**3.1.2. 시나리오**



[그림 5] Edge Cloud, Core Cloud 간 데이터 처리 과정

****

[그림 6, 7] 실내 명패 크롭 예시 (좌) 및 사람 인식 (우)

**3.1.2.1. 명패 인식 (Door Plate Number Detection) 시나리오**

양 측면 카메라에서 전달받은 실시간 RGB 영상에서 Door plate에 해당하는 이미지를 48\*48 사이즈로 크롭한다. 크롭된 이미지는 Grayscale로 변환되어 Core Cloud로 전송된다. 이후, 송신받은 이미지를 바탕으로 Core Cloud는 CNN모델에 해당 데이터를 입력하며, 6개의 결과 값을 도출한다.

이 때 6개의 결과 값은 이미지가 숫자를 포함하는지 여부, 인식된 숫자의 길이, 최대 4자리의 숫자 값으로, 이 결과 값을 통해 실내 정보 데이터베이스와 비교하여 현재 카트의 정확한 실내 위치를 카트의 Edge Cloud로 전송한다.

**3.1.2.2. 사람 인지 (Human Object Recognition) 시나리오**

전방 카메라에서 받은 실시간 영상으로부터 OpenCV의 Hog디스크립터를 이용하여 사람을 인식한다. 이후 카트의 예상 동선을 10개요소를 가진 1차원배열로 작성하고 카트의 너비를 고려, 가장 이상적인 경로 정보를 1차원 배열 형태로 제어모듈에게 전송한다.

**3.2. 요구사항 및 해결 방안**

**3.2.1. Core Cloud ML 모델 Latency 요구사항**

기존 연구에서 측정된 Door Plate Recognition 전처리 과정은 1초당 약 30프레임을 반환하였으며 머신러닝 예측모듈은 약 10프레임으로 병목현상이 발생하였다. 이에 따라 분산 클라우드 환경에서는, Core Cloud상에 위치한 머신러닝 모델의 예측 모듈이 최소 25~30프레임을 유지해야 하며, 이를 바탕으로 두 컴퓨팅 환경 간의 간극차를 해소하여야 한다.

**3.2.2. 프로토콜 요구사항**

에지클라우드와 코어클라우드 간의 통신을 위해, 프로토콜을 통일하여야 한다. 다만 이는 전체적인 서비스 모듈의 성능을 방해하지 않는 한에서 시행되어야 하며, 주행에 필요한 최소한의 정보만을 포함하여 구성함으로써 네트워크 부하를 줄여야한다.

**3.2.3. Critical Section 요구사항**

데이터를 송신, 수신하는 과정에서 Critical Section이 명확히 준수되어야 한다. Write 이후 Read가 수행되어야 하며 Core Cloud와 Edge Cloud간의 Signal을 통하여 서버를 동기화할 필요가 있다. 즉, Edge Cloud에서의 전처리 수행 -> 결과 값 Write -> 전송모듈의 Read, 전송 -> Core Cloud 수신모듈, 수신 -> 예측모듈 수행, 예측값 반환 -> 주행정보 도출 알고리즘 수행 -> 전송모듈에 의해 주행정보 전송의 절차가 명확히 준수되어야 한다.

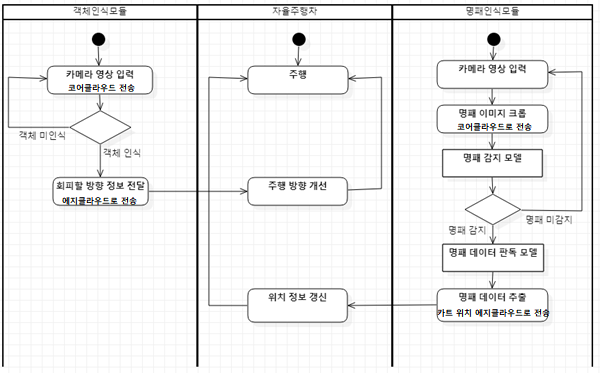
**3.2.4. 명패 번호 (Door Plate Number) 인식용 데이터셋 활용**

Number Detection의 용도로 사용될 데이터셋은 3~4 자리의 숫자로 이루어진 이미지 집합을 요구한다. 이 경우 기존 연구에서 사용되었던 SVHN 데이터셋만으로는 높은 정확도를 기대하기 어려우므로 기존 SVHN 데이터셋에 실내 환경에 특화된 데이터를 포함하여 데이터셋을 새로 구축할 필요가 있다.

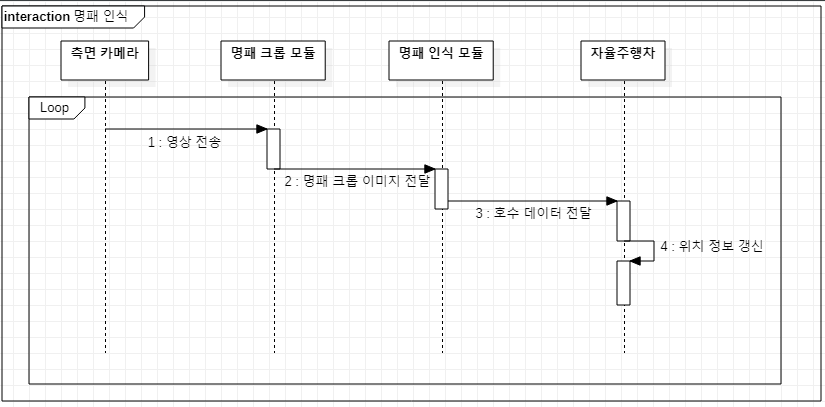
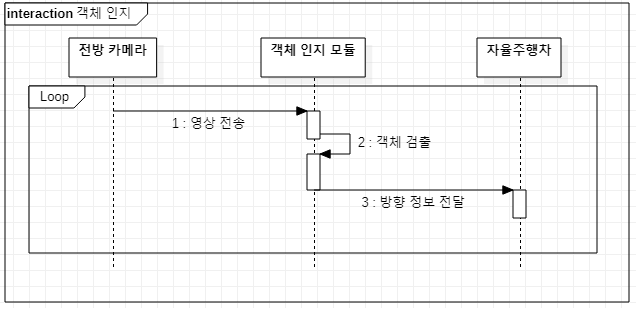
**3.3. 시스템 설계**

**3.3.1. UML Diagram을 통한 시스템 모델링**

**3.3.1.1. Activity Diagram**

****

**3.3.1.2. Sequence Diagram**



* 1. **구현 과정**
     1. **개발 환경**

구현하려는 명패인지(Door Plate Recognition), 사람 인지(Human Recognition)모듈은 라즈베리 파이의 python3환경 상에서 개발 및 구축되었으며 이때 사용된 라이브러리 목록은 다음과 같다.

* OpenCV – Python [ 실시간 컴퓨터 비전을 목적으로 한 프로그래밍 라이브러리]
* Numpy [행렬이나 다차원 배열을 쉽게 처리 할 수 있도록 지원하는 라이브러리]
  + 1. **Door Plate Crop모듈 개발**

좁은 복도가 존재하고 양 옆에 명패가 부착되어 있는 실내환경에서 카트의 현재 주행위치를 보다 정확하게 파악하기 위해 우선적으로 명패영역 이미지를 추출하는 것이 우선적으로 필요하였다. 그러나 실내의 반사율이 높은 환경, 라즈베리파이 카메라모듈의 해상도 및 컴퓨팅 성능 제약, 각도에 따른 형태 왜곡 등의 문제로 초창기 명패가 아닌 다른 이미지가 크롭되는 잡음이 상당수 발생하였다. 이에 따라, 잡음을 낮추는 방법으로, Canny Edge검출법과 평균색 추출 함수를 별도로 작성하여 해당 이미지 크롭 정확도를 개선하였다.



[그림 8] 정확한 명패 크롭이지미(좌)와 잘못된 이미지 크롭 예시(우측 3개이미지)

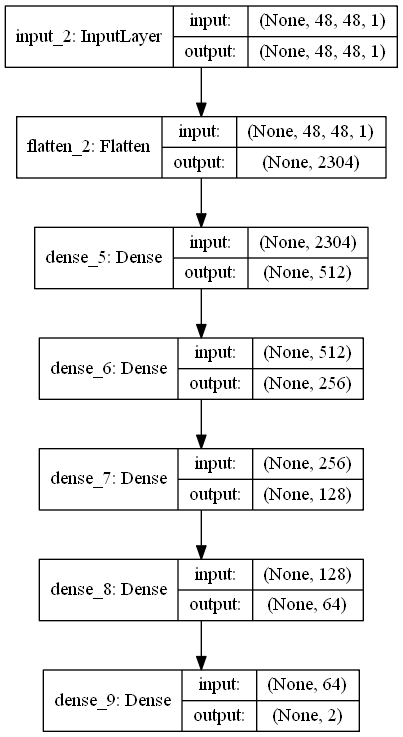
이미지 크롭과정에서 작성된 소스코드와 세부 설명은 다음과 같다.

|  |
| --- |
| def filter\_img(img):  #이미지의 RGB값을 분석하여 찾는 실내 Tag가 맞는지 판별  img = cv2.resize(img, (10,10))  first = [0,0,0]  for x\_loc in range(0, 10):  for y\_loc in range(0, 10):  bgr\_value = img[x\_loc,y\_loc]  first=first+bgr\_value  first[0] = first[0]/100  first[1] = first[1]/100  first[2] = first[2]/100  print(first)  blue = first[0]<200 and first[0]>120  green = first[1]>120 and first[1]<210  red = first[2]>130 and first[2]<230  if(blue and green and red):  return False  else:  return True |
| 먼저 크롭된 이미지에 대해 평균색상을 추출 및 비교하는 함수이다. 해당 함수는 크롭된 영역의 이미지를 10\*10픽셀로 resize하고 이후 해당 이미지에 대해 평균 색상을 BGR순서로 추출한다. 이후 사전에 조사한 명패의 색상 임계값을 토대로 해당 이미지를 필터링하였다.  해당 프로젝트에서 사용된 실내환경의 색상 임계값은 다음과 같다.  R : 130~230  G : 120~210  B : 120~200 |
| def bboxes(inp,prevTime):  #Frame을 인자로 전달받음  img = inp  index = prevTime  start = time.time()  curTime = time.time()  img\_final = inp  img2gray = cv2.cvtColor(inp, cv2.COLOR\_BGR2GRAY)  #GRAY Image 8bit per pixel  img\_canny = cv2.Canny(img2gray, 50, 150)  contours, \_ = cv2.findContours(img\_canny, cv2.RETR\_CCOMP, cv2.CHAIN\_APPROX\_SIMPLE)  # get contours  #RETR CCOMP : 이미지에서 모든 윤곽선 추출  #APPROX\_SIMPLE : Contour를 구성하는 끝점만을 저장  #APPROX\_NONE :: Contour를 구성하는 모든 점 저장  for contour in contours:  if w > 50 or h > 35 or w<13:  continue  if h / w > 1.0 or w / h > 2.0:  continue  if h>40 or w>70:  continue  if y>150:  continue  cropped = img\_final[y :y + h , x : x + w]  if(filter\_img(cropped)):  #cv2.rectangle(img, (x-10, y-20), (x + w + 10, y + h+10), (0, 0, 255), 3)  cropped = img\_final[y :y + h , x : x + w]  cropped = cv2.cvtColor(cropped, cv2.COLOR\_BGR2RGB)  cropped = cv2.resize(cropped, (48,48))  index = index + 1  else:  continue  sec = curTime - prevTime  prevTime = curTime  try:  fps = 1/(sec)  except ZeroDivisionError:  fps = 0  str1 = ("FPS : {0}".format(int(fps)))  cv2.putText(img, str1, (0, 40), cv2.FONT\_HERSHEY\_COMPLEX\_SMALL, 0.8, (0, 255, 0),1)  cv2.imshow('captcha\_result', img)  return index |
| 이미지를 크롭하는 bboxes 함수의 경우 프레임을 그레이스케일로 변환, 변환한 이미지를 바탕으로 Canny Edge Detection을 적용하였다.  Canny Edge검출 알고리즘은 노이즈제거, Gradient값이 높은 부분을 찾고 최대값을 가진 픽셀만을 검출하여 최종적으로 Thresholding을 적용하는 다단계 알고리즘으로 구성되어있다.  구체적인 내용은 다음과 같다.   1. 5X5 가우시안 필터를 적용, 노이즈 제거 2. Gradient 값이 높은 부분 찾기 3. Gradient최대값이 아닌 픽셀의 값을 0으로 만들기 4. Hyteresis Thresholding  * 3번의 도출된 엣지가 실제 엣지인지 판단하는 단계로 임계값을 기준으로 엣지 검출   위와 같은 과정으로 도출된 Edge를 기준으로 윤곽선을 검출하였다. 이때, Open CV의 Find Contour함수를 통해 윤곽선을 검출할 수 있도록 구현하였으며 적용한 옵션은 다음과 같다.   1. Find Contour의 추출모드 옵션 RETR\_CCOMP 적용   해당 옵션은 이미지에서 모든 윤곽선을 검출하는 옵션인데, 실내 명패의 이미지가 전체 프레임 크기와 비교하여 지나치게 작아 해당 옵션을 통해 면밀한 윤곽선을 검출할 수 있도록 하였다.   1. Contour 근사 방법 APPROX\_SIMPLE옵션 적용   주어진 Edge에 대해서 윤곽선의 외곽지점만을 검출하는 옵션이다.  이후, 해당 작업을 통해 윤곽선의 가장 외곽지점 4점이 검출된 이미지 x,y,w,h값을 통해 추출된다. 이때 x,y,w,h값은 아래 그림의 픽셀 위치와 같다.    [그림9] 크롭이미지 좌표  추출된 이미지는 명패의 위치가 지정 영역을 벗어나지 않는지 검증하게 된다. 해당 조건을 부합하게 되면 최종적으로 기존에 작성한 filter\_img함수에 의해 이미지의 평균 색상이 임계값 기준에 부합하는지 판단하게 되며 해당 조건을 만족하면 해당 이미지는 크롭되어 48\*48 size로 변환된다.  만약 조건을 만족하지 못했을 경우 for문 반복구조에 의해 다음 Edge를 기준으로 해당 알고리즘을 반복하게 된다.  해당 소스코드에 대한 대략적인 플로우차트는 다음과 같다. |

* + 1. **Noise Decision Model 구축**

Number Recognition Model은 48\*48 이미지에서 숫자를 3-4개 감지하는 모델로 복잡하고 많은 파라미터를 가지고 있기 때문에 라즈베리 파이의 하드웨어 한계 상 모든 크롭 이미지를 모델에 넣을 수 없다. 따라서 본 연구에서는 경량화된 노이즈 판별 모델을 추가로 구축, 크롭된 이미지를 한 번 더 걸러내어 숫자가 포함되었다고 강하게 추측되는 이미지만 선별하는 로직을 추가했다. 여기서Noise Decision Model이 해당 역할을 하는 경량 모델이다.

Noise Decision Model로는, 미리 훈련된 VGG-16 모델을 기반으로 제작된 Number Recognition Model에 비해 훨씬 경량화된 모델이 요구되어 최대한 적은 파라미터 수를 갖게 모델을 구성했다.



[그림10] Noise Decision Model 구성

|  |
| --- |
| def designed\_model():  inp = Input(shape=(48, 48, 1))  x = Flatten()(inp)  x = Dense(512, activation='relu')(x)  x = Dense(256, activation='relu')(x)  x = Dense(128, activation='relu')(x)  x = Dense(64, activation='relu')(x)  out = Dense(2, activation='softmax')(x)    model = keras.Model(inputs=inp, outputs=out)  optim = optimizers.Adam(lr=0.001, beta\_1=0.9, beta\_2=0.999, epsilon=None, decay=0.001, amsgrad=True)  model.compile(loss='categorical\_crossentropy', optimizer=optim, metrics=['accuracy'])  return model |
| Grayscale로 미리 처리된 48\*48\*1 이미지 텐서가 입력으로 주어지고, 5개의 Fully Connected Layer를 배치했다. 숫자가 포함되었는지 확인하는 것이 목적이기 때문에, CNN을 이용해 모델을 제작해보기도 했지만 파라미터 수가 줄어든 대신 오히려 복잡성이 증가하여 만족스러운 결과를 얻지 못했다.  출력은 softmax 함수를 통과해 [0, 1]의 확률을 나타낸 배열로, 0일 확률이 높으면 이미지에 숫자가 들어있지 않은 것으로 판별, 1일 확률이 높으면 이미지에 숫자가 들어있는 것으로 판별한다.  optimizer로 Adam을 사용하였고, 하이퍼 파라미터는 SVHN을 이용한 기존 연구의 파라미터와 같은 값을 사용하였다. |

* + 1. **Number Recognition Model 구축**

Number Recognition Model에는 opencv를 이용한 크롭 이미지 전략과, Noise Decision Model을 통과한 이미지만이 입력으로 주어지므로, 숫자가 들어있지 않을 확률은 매우 낮다. 다만, 위의 전략과 모델이 완벽하지 않으므로 본 모델에도 숫자가 들어있는지 판별하는 기능이 포함되어야 한다. 따라서, 본 모델에게 요구되는 기능은 숫자 판별, 숫자의 개수 판별, 1번째 자리 ~ 4번째 자리의 숫자 판별로 총 3개이다.

본 모델에는 입력으로 48\*48\*1 Grayscale 이미지가 주어지고, 이를 미리 훈련된 VGG-16 모델에 넣은 후, VGG-16 모델의 출력을 4개의 Fully Connected Layer를 거치게 해서 출력을 만든다.

다만, VGG-16 모델은 RGB 이미지를 기준으로 제작되었기에 3개의 채널이 있어야하므로 48\*48\*1 크롭 이미지를 3번 중첩하는 방식으로 48\*48\*3 텐서를 구성하여 입력하였다.

VGG-16 모델에서 1\*1\*512 텐서가 출력되는데, 이를 512사이즈의 텐서로 변환 후, 3개의 Fully Connected Layer를 거친다. 이 때, 출력된 1024 사이즈의 텐서를 6개의 FC Layer의 입력으로 받아 최종적인 결과가 나온다.

말단 출력층의 Layer들의 구성은 다음과 같다.

1. [0, 1]의 출력, 숫자 포함 판별 기능.

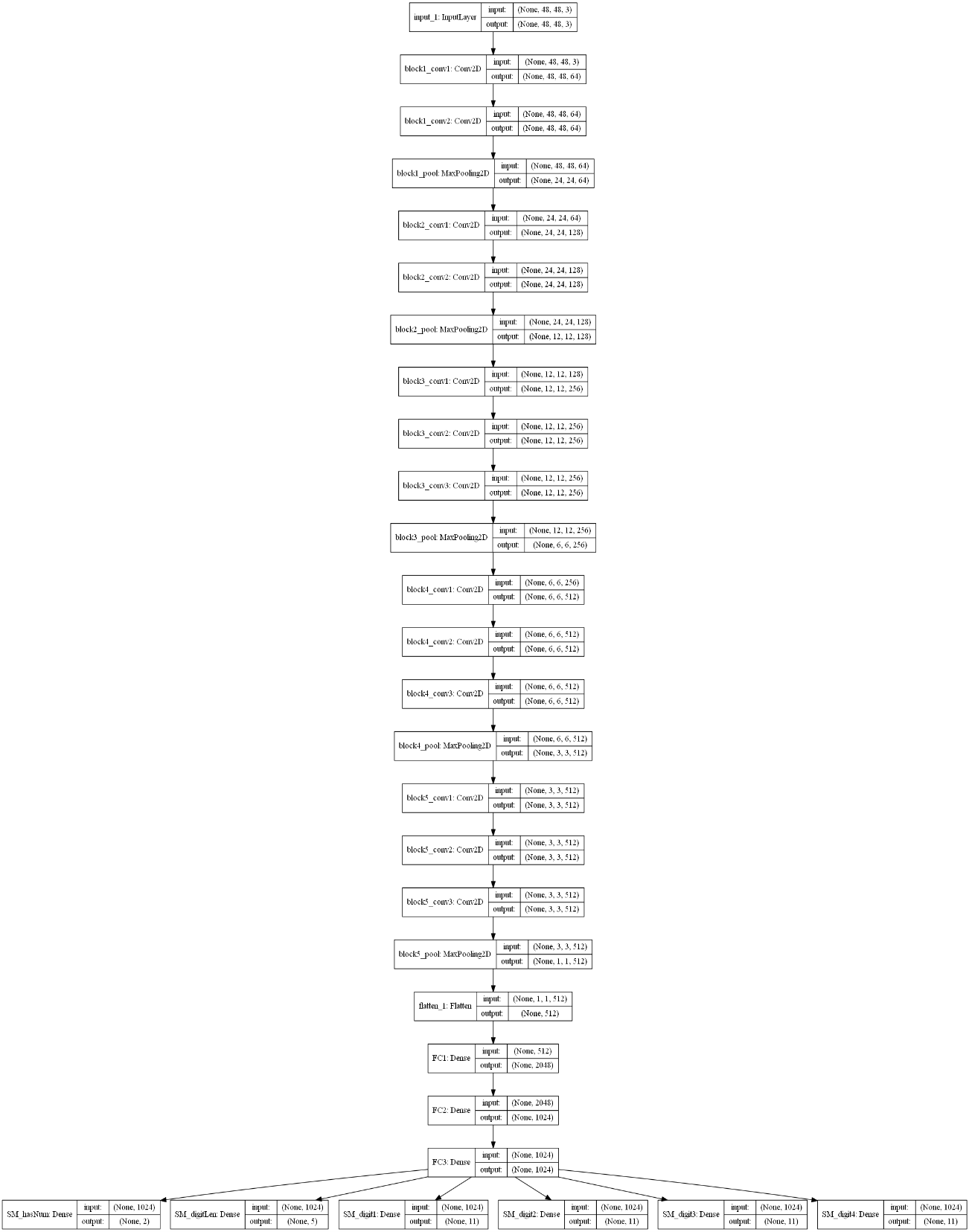
2. [0, 1, 2, 3, 4]의 출력, 이미지에 포함된 숫자의 개수 판별 기능. 주로 0, 3, 4의 값이 많이 예측된다.

3. [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]의 출력. 첫번째 자리의 숫자 판별 기능. (10은 Null을 의미한다.)

4. [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]의 출력. 두번째 자리의 숫자 판별 기능.

5. [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]의 출력. 세번째 자리의 숫자 판별 기능.

6. [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]의 출력. 네번째 자리의 숫자 판별 기능.

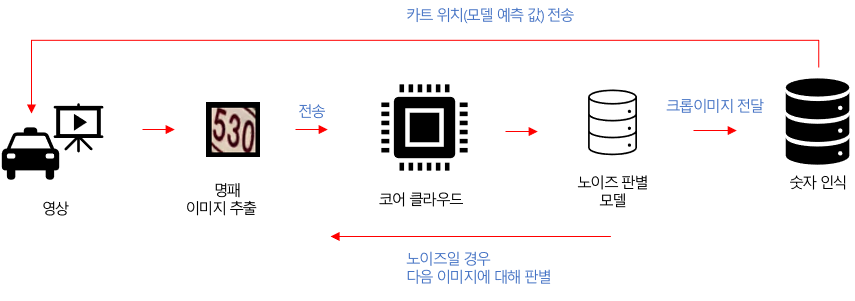


[그림 11] Number Recognition Model 구성

* + 1. **Door Plate Recognition 모듈 통합**

Door Plate Recognition 모듈은 다음 소스코드를 통해 구현하였다.

모듈의 개념적 진행 내용은 다음과 같다.



[그림12] Door Plate Recognition모듈 구성도

|  |
| --- |
| labeling\_module.py |
| # plaidml  # import plaidml.keras  # plaidml.keras.install\_backend()  # packages  from keras.models import load\_model  from keras.preprocessing import image  # import queue  import datetime  import numpy as np  from queue import Full, Empty  from multiprocessing import Process, Queue  import socket  import cv2  fname = 'croppedimg/{}.png'  save\_imgs = False  HOST = '192.168.35.87'  PORT = 9999  class LabelingModule:  def \_\_init\_\_(self):  self.model1 = load\_model('checker\_model.h5')  self.model2 = load\_model('svhn\_model.h5')  self.image\_queue = Queue(maxsize=3000)  self.label\_queue = Queue(maxsize=10)  self.signal\_queue = Queue()  self.predict\_process = Process(target=\_predict, \  args=(  self.model1, self.model2, self.image\_queue, self.label\_queue, self.signal\_queue))  def run(self):  self.predict\_process.start()  def close(self):  self.signal\_queue.put\_nowait('stop')  self.image\_queue.close()  self.label\_queue.close()  def new\_tensor(self, tensor):  try:  self.image\_queue.put(tensor)  except Full:  print('[LabelingModule] image\_queue is full')  def new\_image(self, filename):  tensor = self.\_img\_to\_tensor(filename)  try:  self.image\_queue.put(tensor)  except Full:  print('[LabelingModule] image\_queue is full')  def \_img\_to\_tensor(self, filename):  img = image.load\_img(filename, target\_size=(48, 48))  img\_tensor = image.img\_to\_array(img)  img\_tensor = np.squeeze(img\_tensor)  img\_tensor /= 255.  img\_tensor = img\_tensor - img\_tensor.mean()  return img\_tensor  def decode(output):  if(output[0]==0):  return 'Noise'  else:  if(output[1] == 3):  return str(output[2])+str(output[3])+str(output[4])  elif (output[1] == 4):  return str(output[2]) + str(output[3]) + str(output[4])+'-'+ str(output[5])  def send\_predict\_result(HOST, PORT,message):  # (address family) IPv4, TCP  client\_socket = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)  # raspberry pi addr  client\_socket.connect((HOST, PORT))  client\_socket.sendall('Door Plate Detected : '+message.encode('utf-8'))  print("Send : ", message)  def \_predict(model1, model2, input\_queue, output\_queue, signal\_queue):  print('predict process started.')  index = 0  while True:  try:  signal = signal\_queue.get\_nowait()  if signal == 'stop':  break  except Empty:  pass  try:  tensor = input\_queue.get(timeout=-1)  except Empty:  continue  tensor = np.array([tensor])  has\_number = model1.predict(tensor)[0]  if int(has\_number[0]) == 1:  continue  if save\_imgs:  img = cv2.cvtColor(tensor[0], cv2.COLOR\_RGB2BGR)  cv2.imwrite(fname.format(index), img)  index += 1  label\_data = model2.predict(tensor)  o1 = np.argmax(label\_data[0])  o2 = np.argmax(label\_data[1])  o3 = np.argmax(label\_data[2])  o4 = np.argmax(label\_data[3])  o5 = np.argmax(label\_data[4])  o6 = np.argmax(label\_data[5])  output = [o1, o2, o3, o4, o5, o6]  print('[LabelingModule] predict result :', decode(output))  send\_predict\_result(HOST,PORT) |

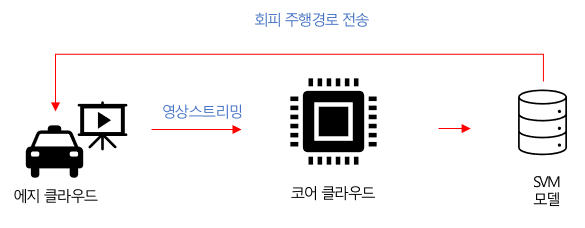
|  |
| --- |
| Core\_cloud.py |
| import socket  import cv2  import numpy as np  #from queue import Queue  from \_thread import \*  from multiprocessing import Queue  from labeling\_module import LabelingModule  enclose\_q = Queue()  lm = LabelingModule()  #socket에서 수신한 버퍼를 반환  def recvall(sock, count):  # 바이트 문자열  buf = b''  while count:  newbuf = sock.recv(count)  if not newbuf: return None  buf += newbuf  count -= len(newbuf)  return buf    if \_\_name\_\_ == "\_\_main\_\_":  lm.predict\_process.start()  RECV\_HOST=''  RECV\_PORT=9999 #RECV PORT    #TCP 사용  s=socket.socket(socket.AF\_INET,socket.SOCK\_STREAM)  print('Socket created')    #CoreCloud IP, PortNumber set  s.bind((RECV\_HOST,RECV\_PORT))  print('Socket bind complete')  s.listen(1)  print('Socket now listening')    #conn , addr socket binded addr  conn,addr=s.accept  while True:  # client에서 받은 stringData length (==(str(len(stringData))).encode().ljust(16))  length = recvall(conn, 16)  stringData = recvall(conn, int(length))  data = np.fromstring(stringData, dtype = 'uint8')  #data decode  cropped = cv2.imdecode(data, cv2.IMREAD\_COLOR)  cropped = cv2.resize(cropped, (48,48)) #Crop Image Resize  result = lm.new\_tensor(cropped) # Predict result  lm.predict\_process.join() # thread join  edge\_socket, addr = server\_socket.accept()  enclose\_q.put(result)  if(conn): #연결 끊어질 경우 loop 탈출  break |

|  |
| --- |
| Edge\_cloud.py |
| # -\*- coding: euc-kr -\*-  import socket  import cv2  import numpy as np  from queue import Queue  from \_thread import \*  enclose\_q = Queue()  recv\_enclose\_q = Queue()  def filter\_img(img):  img = cv2.resize(img, (10,10))  first = [0,0,0]  for x\_loc in range(0, 10):  for y\_loc in range(0, 10):  bgr\_value = img[x\_loc,y\_loc]  first=first+bgr\_value  first[0] = first[0]/100  first[1] = first[1]/100  first[2] = first[2]/100  blue = first[0]<200 and first[0]>120  green = first[1]>120 and first[1]<210  red = first[2]>130 and first[2]<230  if(blue and green and red):  return True  else:  return False  def bboxes(inp):  img = inp  start = time.time()  curTime = time.time()  img\_final = inp  img2gray = cv2.cvtColor(inp, cv2.COLOR\_BGR2GRAY) #GRAY Image 8bit per pixel  ret, mask = cv2.threshold(img2gray, 180, 255, cv2.THRESH\_BINARY) #threshold : distinguish background, object  image\_final = cv2.bitwise\_and(img2gray, img2gray, mask=mask) #bitwise  ret, new\_img = cv2.threshold(img\_final, 180, 255, cv2.THRESH\_BINARY) # Nfor black text , cv.THRESH\_BINARY\_IV  newimg = cv2.cvtColor(new\_img, cv2.COLOR\_BGR2GRAY) #Gray Image converting  \_,contours, \_ = cv2.findContours(newimg, cv2.RETR\_CCOMP, cv2.CHAIN\_APPROX\_NONE) # get contours  #cv2.CHAIN\_APPROX\_NONE: All of contour point  for contour in contours:  [x, y, w, h] = cv2.boundingRect(contour)  if h / w > 1.0 or w / h > 2.0:  continue  #if h>40 or w>70:  #continue  if y>150:  continue  cropped = img\_final[y :y + h , x : x + w]  if(filter\_img(cropped)):  cv2.rectangle(img, (x, y), (x + w, y + h), (0, 0, 255), 3)  cv2.putText(img,"cropped", (x-50,y-10), cv2.FONT\_HERSHEY\_COMPLEX\_SMALL, 1, (0,0,255), 1)  else:  continue  return cropped  def send\_threaded(Client\_socket, addr, queue):  print("Connected by : ", addr[0], " : ", addr[1])  while True:  try :  data = Client\_socket.recv(1024)  if not data:  print("Disconnected")  break  StringData = queue.get()  Client\_socket.send(str(len(StringData)).ljust(16).encode())  Client\_socket.send(StringData)  except ConnectionResetError as e:  print("Disconnected")  Client\_socket.close()  def webcam(queue):  capture = cv2.VideoCapture(0)  while True:  ret, frame = capture.read()  if ret == False:  continue  frame = bboxes(frame)  encode\_param = [int(cv2.IMWRITE\_JPEG\_QUALITY), 90]  result, imgencode = cv2.imencode('.jpg', frame, encode\_param)  data = np.array(imgencode)  stringData = data.tostring()  queue.put(stringData)  cv2.imshow('image', frame)  key = cv2.waitKey(1)  if key == 27:  break  def recvall(sock, count):  # byte string  buf = b''  while count:  newbuf = sock.recv(count)  if not newbuf: return None  buf += newbuf  count -= len(newbuf)  return buf  if \_\_name\_\_ == '\_\_main\_\_':  SEND\_HOST = '192.168.35.227' #CORE CLOUD  SEND\_PORT = 9999  server\_socket = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)  server\_socket.setsockopt(socket.SOL\_SOCKET, socket.SO\_REUSEADDR, 1)  server\_socket.bind((SEND\_HOST, SEND\_PORT))  server\_socket.listen()  RECV\_HOST = '192.168.35.87' #EDGE CLOUD  RECV\_PORT = 9998  recv\_server\_socket = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)  recv\_server\_socket.setsockopt(socket.SOL\_SOCKET, socket.SO\_REUSEADDR, 1)  recv\_server\_socket.bind((RECV\_HOST, RECV\_PORT))  recv\_server\_socket.listen()  print('server start')  start\_new\_thread(webcam, (enclose\_q,))  while True:  print('wait')  client\_socket, addr = server\_socket.accept()  start\_new\_thread(send\_threaded, (client\_socket, addr, enclose\_q,))  #preprocessed data recieve.  conn,addr = recv\_server\_socket.accept() #waiting receiving...  if(conn):  length = recvall(conn, 16)  stringData = recvall(conn, int(length))  data = np.fromstring(stringData, dtype = 'uint8')  #receive driving info.  if(conn): #loop ex  break  else:  pass  server\_socket.close() |

* + 1. **Human Recognition 모듈 개발**

사람의 왕래가 잦고, 장애물의 간섭이 빈번한 실내 환경의 특성과 좁은 복도가 존재하는 물리적 환경을 고려하여 Human Recognition을 통해 카트의 예상 주행 경로를 파악할 수 있도록 설계하였다.

최종적으로 근거리의 경우, LIDAR센서를 통해 우발적 대처를 가능케 하였으며 원거리(5m이상)의 경우 사람 객체를 인식하여 회피주행 할 수 있도록 해당 모듈을 개발하였다. Human Recognition은 OpenCV의 Hog디스크립터와 SVM모델을 기반으로 설계하였으며 예상 경로를 1차원 배열로 반환하고, 화면상에 실시간으로 예상 경로를 그릴 수 있도록 구현하였다.



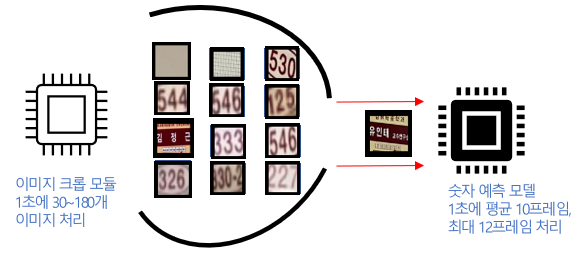
해당 모듈의 상세 코드는 다음과 같다.

|  |
| --- |
| HumanRecognition\_Core.py |
| 먼저 라즈베리 파이 LCD해상도(640\*480)에 맞게 해당 경로를 출력해주는 함수를 작성하였다. 각 함수는 인식된 객체가 화면의 왼쪽 혹은 오른쪽에 위치할 때 호출되며, 객체의 좌표를 기준으로 카트의 회피경로를 계산하여 화면에 실시간으로 그려준다. |
| import numpy as np  import cv2  import time  import socket  HOST = '192.168.35.87'  PORT = 9999  # (address family) IPv4, TCP  client\_socket = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)  # raspberry pi addr  client\_socket.connect((HOST, PORT))  hog = cv2.HOGDescriptor()  hog.setSVMDetector(cv2.HOGDescriptor\_getDefaultPeopleDetector())  cv2.startWindowThread()  fname = "./croppedimg/human/"  # open webcam video stream  cap = cv2.VideoCapture("http://192.168.35.87:8091/?action=stream")  print("Capture From Raspberry pi dev/Camera0")  isFirst = True  i = 0  weight\_list = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]  prevTime = 0  def check\_obstacle(weight\_list, xA, xB):  for i in range(0, len(weight\_list)):  weight\_list[i] = 0  xA = int(xA / 32)  xB = int(xB / 32)  for i in range(xA, xB + 1):  if (weight\_list[i] == 0):  weight\_list[i] = weight\_list[i] - 1  return weight\_list  def draw\_left\_path(img, x, y, w, h):  start\_point = x + w  cv2.line(img, (160 - 2 \* w, 240), (start\_point, y + int(h)), (255, 0, 0), 8) # 4픽셀 선 그리기  cv2.line(img, (start\_point, y + int(h)), (start\_point + int(w / 5), y + int(h - 20)), (255, 0, 0), 10)  start\_point = x + 2 \* w  cv2.line(img, (250, 240), (start\_point, y + int(h)), (255, 0, 0), 8) # 4픽셀 선 그리기  # cv2.line(img, (start\_point, y+int(h)), (start\_point+int(w/5),y+int(h-20)), (255, 0, 0), 10)  return img  def draw\_right\_path(img, x, y, w, h):  start\_point = x  cv2.line(img, (160 + 2 \* w, 240), (start\_point, y + int(h)), (255, 0, 0), 8) # 8픽셀 선 그리기  cv2.line(img, (start\_point, y + int(h)), (start\_point - int(w / 5), y + int(h - 20)), (255, 0, 0), 8)  start\_point = abs(x - w)  cv2.line(img, (20, 240), (start\_point, y + int(h)), (255, 0, 0), 8) # 8픽셀 선 그리기  # cv2.line(img, (start\_point, y + int(h)), (start\_point - int(w / 5), y + int(h-20)), (255, 0, 0), 8)  return img  # initialize the HOG descriptor/person detector  avg = 0  count = 1  while (True):  # Capture frame-by-frame  start = time.time()  curTime = time.time()  ret, frame = cap.read()  # resizing for faster detection[240,160] [320 \* 240]  frame = cv2.resize(frame, (320, 160))  # using a greyscale picture, also for faster detection  # detect people in the image  # returns the bounding boxes for the detected objects  boxes, weights = hog.detectMultiScale(frame, winStride=(8, 8))  detectCount = 0  boxes = np.array([[x, y, x + w, y + h] for (x, y, w, h) in boxes])  for (xA, yA, xB, yB) in boxes:  # display the detected boxes in the colour picture  w = xB - xA  h = yB - yA  cv2.rectangle(frame, (xA, yA), (xB, yB),  (0, 255, 0), 2)  cv2.putText(frame, "Detect", (xA - 50, yA - 10), cv2.FONT\_HERSHEY\_COMPLEX\_SMALL, 1, (0, 0, 255), 1)  detectCount = detectCount + 1  if (detectCount > 1):  print("Waiting...")  client\_socket.sendall('Detecting...'.encode('utf-8'))  else:  if (i % 10 == 0):  cropped = frame[yA:yB, xA:xB]  # print("xA : {0}, xB : {1}, yA : {2}, yB : {3}".format(xA, xB,yA,yB)) # Print Width, Height of Cropped Area  i = 0  if (xB < 190 and xA < 130):  #print("Left Side Detect.")  try:  frame = draw\_left\_path(frame, xA, yA, xB - xA, yB - yA)  client\_socket.sendall('Human Detected : Left'.encode('utf-8'))  except:  pass  elif (xA > 130 and xB > 190):  #print("Right Side Detect")  try:  frame = draw\_right\_path(frame, xA, yA, xB - xA, yB - yA)  client\_socket.sendall('Human Detected : Right '.encode('utf-8'))  except:  pass  else:  try:  frame = draw\_right\_path(frame, xA, yA, xB - xA, yB - yA)  except:  pass  #print("Center Side Detect")  # s = fname + str(i)+'.jpg'  # cv2.imwrite(s, cropped) # IMG File Write  #print("time :", time.time() - start)  # Display the resulting frame  # frame = cv2.resize(frame, (480,320))  sec = curTime - prevTime  prevTime = curTime  fps = 1 / (sec)  if(fps+30>70):  fps = 70  else:  fps = fps+30  str1 = ("FPS : {0}".format(int(fps)))  avg = (avg + int(fps)) / count  cv2.putText(frame, str1, (0, 40), cv2.FONT\_HERSHEY\_COMPLEX\_SMALL, 0.8, (0, 255, 0), 1)  frame = cv2.resize(frame, (680, 480), interpolation=cv2.INTER\_CUBIC)  cv2.imshow('frame', frame)  count = count+1  if cv2.waitKey(1) & 0xFF == ord('q'):  break  client\_socket.close()  # When everything done, release the capture  cap.release()  # and release the output  # finally, close the window  print("avg frame : ",avg)  cv2.destroyAllWindows()  cv2.waitKey(1) |
| 사람 객체의 인지를 위해 Open CV의 Hog 디스크립터를 사용하였다. 코드 전체의 대략적인 내용은 다음과 같다.   1. 프레임 사이즈 조정 및 그레이 스케일 변환 2. Hog 디스크립터를 사용하여 객체 탐지 3. 탐지된 객체에 대해 bbox(bounding box) Drawing. 4. 회피 주행 경로 출력 및 1차원 배열 반환   [사람이 여러명이 있고 복도의 공간이 여유롭지 않을 경우 대기]  해당 모듈에서 사용된 Hog디스크립터는 보행자 검출 목적의 디스크립터로, 대상 객체의 일반화에 매우 적합한 디스크립터이다.  Hog디스크립터의 동작과정은 다음과 같다.    [그림12] Hog 디스크립터를 통한 사람 객체 검출 과정  Hog디스크립터는 단순히 윤곽선이 아닌, window크기에 따른 8방향의 히스토그램을 계산한다. 객체를 인지하는 순서는 다음과 같다.   1. 영상으로부터 Gradient를 계산한다. 2. Gradient를 이용하여 Local Histogram을 생성한다. 3. Local Histogram을 이어붙여 1차원의 Vector(HOG Feature)를 생성한다. 4. 검출된 객체를 학습된 SVM모델의 입력값으로 반환한다. 5. 사람 객체 검출   위의 과정을 통한 HOG 디스크립터의 경우 조명 변화의 영향을 적게 받으며 대상을 일반화하여 검출할 수 있다는 장점이 있다. 또한, 대상객체의 방향에 구애받지 않고 사람객체를 인지할 수 있어 보다 정확한 예측 및 인지가 가능하다.  해당 코드에 대한 개략적인 플로우차트는 다음과 같다. |
| Human\_Recognition\_Edge.py [예측결과 수신모듈] |
| import socket  PORT = 9999  server\_socket = socket.socket(socket.AF\_INET, socket.SOCK\_STREAM)  server\_socket.setsockopt(socket.SOL\_SOCKET, socket.SO\_REUSEADDR,1)  server\_socket.bind(('',PORT))  server\_socket.listen(1)  client\_socket,addr = server\_socket.accept()  print('Connected by', addr)  while True:  data = client\_socket.recv(1024)  if not data:  pass  print('Recieved from', addr, data.decode('utf-8')) |

**4. 프로젝트 성능 개선**

**4.1. 분산환경 구축 전**

기존 Edge Cloud에서만 모든 영상처리를 수행할 경우, Door Plate Number Detection은 초당 10프레임, Human Object Recognition은 초당 20-25 프레임의 성능을 보여주었다. 뿐만 아니라, 당시 모듈간의 처리속도 차이에 의해서, 병목현상이 발생하여 오랫동안 모듈을 실행될 경우 버퍼 오버플로우가 발생, 강제 종료되는 경우가 있어 실사용에 어려움이 있었다.

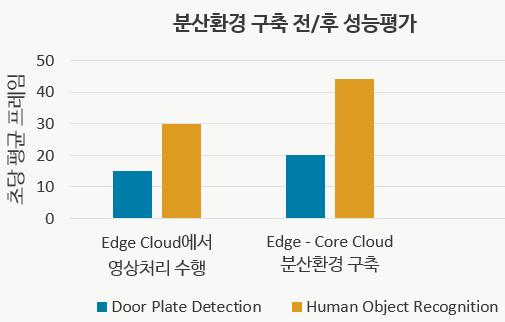


[그림 13] 분산환경 구축 전 병목현상 이슈

**4.2. 분산환경 구축 후**

에지클라우드의 컴퓨팅성능 제약으로 인한 한계점을 극복하기 위해, 코어클라우드와의 소켓통신, MJPG스트리머를 통한 스트리밍통신을 바탕으로 모듈을 분산처리할 수 있도록 구현하였다.

때문에, 복잡한 영상처리 혹은 딥러닝 모델을 통한 예측과정을 상대적으로 컴퓨팅성능이 좋은 코어클라우드에서 처리함으로써, 빠른 퍼포먼스를 기대할 수 있었다.



[그림 14] 분산환경 구축 전/후 성능평가 결과

* **명패인식모듈**

1프레임 평균 처리속도가 0.15초에 그치던것에 반해, 분산환경 구축 후 1프레임당 평균 0.03초의 처리속도를 보였다. 즉, 초당 30프레임 이상을 유지할 수 있으며, 에지클라우드의 전처리모듈이 30-40프레임의 성능을 감안하면 병목현상 역시 해결할 수 있음을 알 수 있다.

* **사람인지모듈**

기존 라즈베리파이에서 약 20-25프레임에 그치던 성능에 비해, 현재 평균 44프레임으로 개선되었음을 볼 수 있다. 때문에, 사람의 우발적인 행동에도 보다 빠르게 대처할 수 있으며, 에지클라우드의 부하가 크게 줄어 빠른 퍼포먼스를 보여주었다.

**5. 결론 및 향후 연구**

실내용 자율 주행 카트를 개발함에 있어서, 이미 저장된 지도정보가 아닌, 상황에 따라 유연한 대응을 하는 자율주행카트를 개발하고, 실내 환경에 최적화된 자율주행 시스템을 연구하였다. 또한 근거리의 경우 센서를 활용한 Object Detection과, 원거리의 경우 영상처리 및 머신러닝 모델에 의한 예측으로 우발적인 상황에는 하드웨어적인 명령으로 빠른 대처가 가능하며 실시간 영상처리를 기반으로 향후 이동경로를 미리 예상할 수 있다.

뿐만 아니라, 영상처리 모듈의 경우 Edge Cloud환경을 기반으로 분산처리를 구현하여, 카트상의 컴퓨팅 성능 제약을 회피할 수 있으며 최종적으로 전처리 모듈과 예측모듈간의 병목현상을 줄이고, 네트워크 전체의 부하를 줄일 수 있었다.

**6. 참고 문헌**

[1] 박강민, 허용석, and 권혁윤. "클라우드 환경에서의 분산 이미지 처리 프레임워크." 한국정보과학회 학술발표논문집 2019.12 (2019): 158-60. Web.

[2] 이현철, Hyunchul Lee, 이성민, Sungmin Lee, 김강석, and Kangseok Kim. "AWS Lambda Serverless Computing 기술을 활용한 효율적인 딥러닝 기반 이미지 인식 서비스 시스템." 정보처리학회논문지. 소프트웨어 및 데이터 공학 9.6 (2020): 177-86. Web.

[3] 조국한, 조현우, and 송영준. "클라우드 서버 환경에서 Lidar 센서와 Yolo V3 라이브러리를 이용한 360도 영상 인식 시스템 구현." 한국정보기술학회논문지 18.2

[4] 이재빈, 김명현, 김윤곤, 허의남. (2020). 실내 문서 전달 자율주행 카트 개발을 위한 객체 인지 및 충돌 방지 모듈 개발. 한국정보과학회 학술발표논문집, 1483-1485.