차량지능기초
과제 1

자동차공학과

2017337 한별

깃허브 주소 : https://github.com/ByeolHan

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

- 데이터 설명, 양, 세부 요소 (Feature), 데이터 예시, 활용 예 등 데이터를 이해하는데 필요한 정보

1. PASCAL VOC07

1) 데이터 설명

PASCAL VOC는 Pattern Analysis, Statistical modeling and ComputAtional Learning Visual Object Classes의 약자로 classifcation, object detection, segmentation 성능을 평가할 수 있는 데이터셋이 포함되어 있다..

2) 양

9,963개의 시퀀스(sequence)의 학습 데이터로 구성되어 있다. train 데이터 2,50개, validation 데이터 2,501개, test 데이터 4,952개로 구성되어 있다. 총 24,640개의 주석이 달린 개체가 있다.

3) 클래스

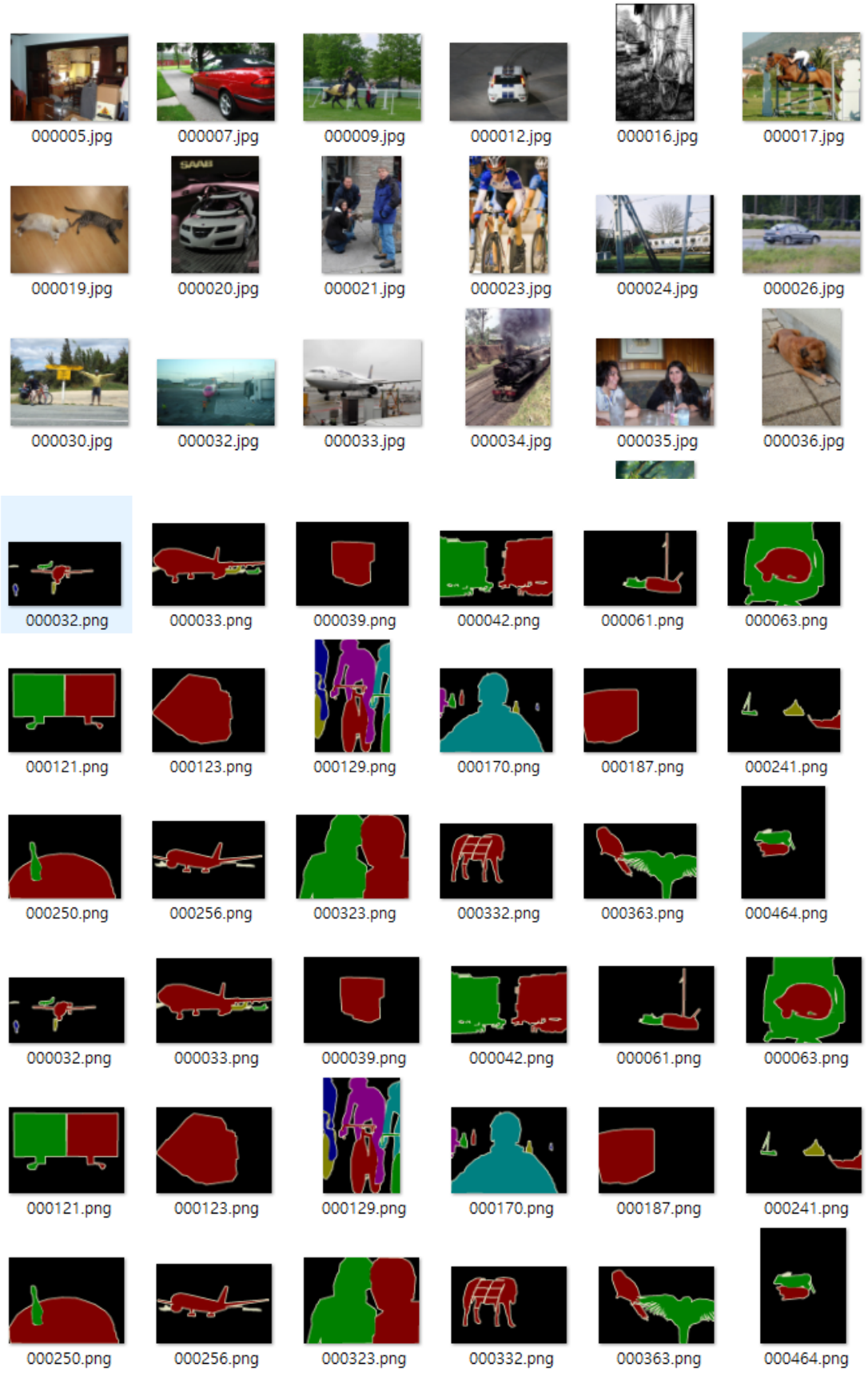
사람, 동물(새, 고양이, 소, 개, 말, 양), 운송체(비행기, 자전거, 보트, 버스, 승용차, 오토바이, 기차), 사물(병, 의자, 식탁, 화초, 소파, 모니터) 총 20개의 클래스로 나누어져 있다.

4) 세부요소 (Feature)

```
FeaturesDict({
    'image': Image(shape=(None, None, 3), dtype=tf.uint8),
    'image/filename': Text(shape=(), dtype=tf.string),
    'labels': Sequence(ClassLabel(shape=(), dtype=tf.int64,
num_classes=20)),
    'labels_no_difficult': Sequence(ClassLabel(shape=(),
dtype=tf.int64, num_classes=20)),
    'objects': Sequence({
        'bbox': BBoxFeature(shape=(4,), dtype=tf.float32),
        'is_difficult': tf.bool,
        'is_truncated': tf.bool,
        'label': ClassLabel(shape=(), dtype=tf.int64, num_classes=20),
        'pose': ClassLabel(shape=(), dtype=tf.int64, num_classes=5),
    }),
})
```

- image: 문장 단위 이미지 설명
- image/filename : 파일 이름
- name : class 이름 설명
- bbox : xmin, ymin, xmax, ymax 위치(float32 type)
- difficult, truncated 0,1 Bool 값으로 설명한다.
- label : 클래스 이름 설명
- pose : 객체 위치 설명 eg)right

5) 데이터 예시 (imagesets, segmentation class, segmentation object)



6) 활용 예

image classification, object detection, segmentation 평가 알고리즘을 구축하고 평가하는데 유용하게 사용된다. Pascal VOC challenge는 2005년 부터 2012년까지 매년 진행되기도 하였으며 입력 영상에서 특정 종류의 물체를 검출해내는 성능을 겨루는 대회였다. 물체 탐지 기술로서 대표적인 방법들이 이 대회에서 많이 탄생하였다. 센서의 실시간성 저하로 인한 객체 검출의 지연은 자율주행자동차의 안전성에 큰 위험원이 될 수 있다. 카메라를 이용한 딥러닝 기반 후, 측방 객체 검출과 차로 변경에 대해 딥러닝 알고리즘을 개발하고 객체 검출의 정확성과 실시간성을 VOC 데이터를 이용해 검증하는 등 단순한 객체 검출을 위한 데이터셋으로 사용될 뿐 아니라 안전한 자율주행 기술의 성능을 평가하는데에도 사용된다.

2. KITTI

1) 데이터 설명

KITTI Dataset은 RGB 카메라와 벨로다인 라이다(Velodyne Lidar) 등의 센서가 장착된 차량으로 독일 도시 지역에서 추출되었다. 데이터 셋은 난이도에 따라 easy, moderate, hard로 나누어진다. 전체 데이터 셋은 stereo, sceneflow, depth, obejct, tracking 등 많은 작업물이 포함되어있다.

2) 양

7,481개의 시퀀스(sequence)의 학습 데이터로 구성되어 있다. train 데이터 6,347개, validation 데이터 423개, test 데이터 711개로 구성되어 있다.

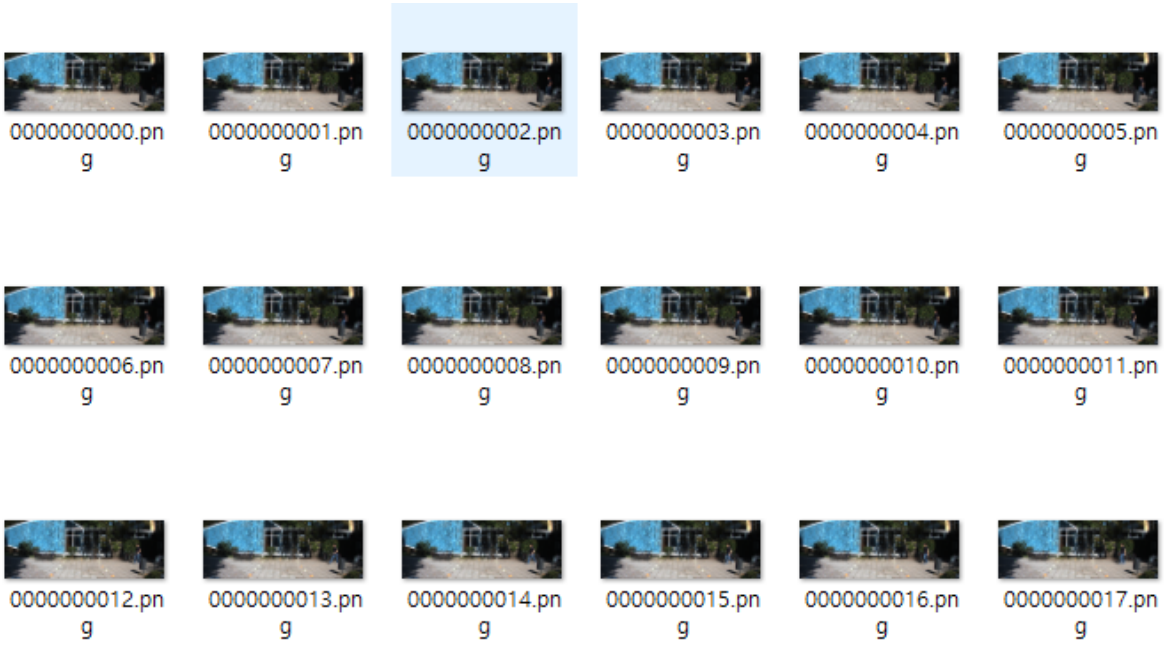
3) 클래스

학습 데이터는 9가지 객체의 종류(Car, Van, Truck, Pedestrian, Person_sitting, Cyclist, Tram, Misc, DontCare)와 51,867개의 라벨을 포함하고 있다.

4) 세부요소 (Feature)

```
FeaturesDict({
    'image': Image(shape=(None, None, 3), dtype=tf.uint8),
    'image/file_name': Text(shape=(), dtype=tf.string),
    'objects': Sequence({
        'alpha': tf.float32,
        'bbox': BBoxFeature(shape=(4,), dtype=tf.float32),
        'dimensions': Tensor(shape=(3,), dtype=tf.float32),
        'location': Tensor(shape=(3,), dtype=tf.float32),
        'occluded': ClassLabel(shape=(), dtype=tf.int64, num_classes=4),
        'rotation_y': tf.float32,
        'truncated': tf.float32,
        'type': ClassLabel(shape=(), dtype=tf.int64, num_classes=8),
    }),
})
```

5) 데이터 예시



6) 활용 예

딥러닝에 기반한 거리측정, 객체 탐지 기술, slam 알고리즘 모델의 학습데이터로 많이 사용되며 모델의 성능을 검증하는데에도 쓰인다. slam 기술이 상용화 될 경우
주변 환경과 주변 차량을 인식하여 정밀한 인식이 가능하고 자율주행 기술이 상용화 되기 이전에도 현재 차량 환경에서 운전자의 안전을 크게 높일 수 있다.
현실에서 수집된 공개 데이터셋은 특정한 나라의 교통 상황에만 국한되어 있고, 사계 변화가 뚜렷한 국내 사정에 적합하지 못한다는 단점이 있다. 또한 데이터가 모델 학습에 사용되려면 정답 데이터인 라벨도 필수적으로 확보가 되어야한다. 많은 데이터를 수집하고 데이터에 라벨을 수작업으로 붙이는 것은 많은 시간과 비용이 들어가며 인적요인으로 잘못 지정될 수 있다는 문제점 또한 있다. 네이버 랩스에서는 이러한 문제점들을 보완하기 위해 Virtual KITTI라는 가상의 데이터셋을 생성하고 발표하였다.
시뮬레이션 플랫폼에서는 악천우의 제약없이 국내 도로환경의 데이터 구현이 가능하다. 네이버 랩스에서는 유니티 게임 엔진을 사용하여 KITTI 데이터셋을 재현했다. 기계학습 모델을 이용하여 교육, 테스하기 위한 최초의 합성 데이터셋 중 하나이다. 가상 KITTI에는 다양한 이미징 및 기상 조건에서 도시환경의 5가지 가상세계에서 생성된 50 개의 고해상도 단안 동영상 (21,260 프레임)이 포함되어 있다. 현재 Virutal KITTI.ver2까지 출시되었다. 합성 데이터셋이 실제 데이터셋을 완전히 대체할 수는 없지만 예비 프로토타입을 평가하는데 유용한 것으로 입증됐으며, 실제 데이터셋과 함께 사용하면 성능을 개선할 수 있다.

3. COCO

1) 데이터 설명

대규모의 객체 탐지, 분류, 라벨링 데이터셋이다. coco 2014년, coco 2017년, coco 2017_panoptic 버전이 있으며 2014와 2017 버전은 동일한 이미지를 사용하지만 데이터셋 분할 비율이 다르다. 테스트 셋에는 주석이 없다. 91개의 클래스를 정의하지만 데이터에서는 80개의 클래스만 사용한다. Panoptic은 200개의 클래스를 정의하지만 133개만 사용한다.

2) 양

coco 2014년 버전은 train 데이터 82,783개, validation 데이터 40,504개, test 데이터 40,775개, test 2015 데이터 81,434개로 구성되어 있다.

coco 2017년 버전은 train 데이터 118,287개, validation 데이터 5,000개, test 데이터 40,670개로 구성되어 있다.

coco 2017_panoptic 버전은 train 데이터 118,287개, validation 데이터 5,000개로 구성되어 있다.

3) 클래스

2014년 버전과 2017년 버전은 91개의 클래스를 정의하지만 데이터에서는 80개의 클래스만 사용한다.

Panotptic은 200개의 클래스를 정의하지만 133개만 사용한다.

4) 세부요소 (Feature)

coco 2014년

```
FeaturesDict({
    'image': Image(shape=(None, None, 3), dtype=tf.uint8),
    'image/filename': Text(shape=(), dtype=tf.string),
    'image/id': tf.int64,
    'objects': Sequence({
        'area': tf.int64,
        'bbox': BBoxFeature(shape=(4,), dtype=tf.float32),
        'id': tf.int64,
        'is_crowd': tf.bool,
        'label': ClassLabel(shape=(), dtype=tf.int64, num_classes=80),
    }),
})
```

coco 2017년

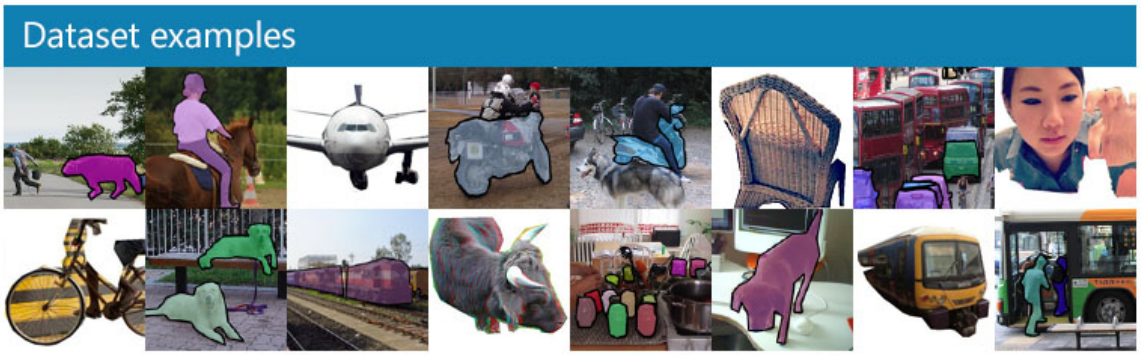
```
FeaturesDict({
    'image': Image(shape=(None, None, 3), dtype=tf.uint8),
    'image/filename': Text(shape=(), dtype=tf.string),
    'image/id': tf.int64,
    'objects': Sequence({
        'area': tf.int64,
        'bbox': BBoxFeature(shape=(4,), dtype=tf.float32),
        'id': tf.int64,
        'is_crowd': tf.bool,
        'label': ClassLabel(shape=(), dtype=tf.int64, num_classes=80),
    }),
```

```
})
```

coco 2017_panotptic

```
FeaturesDict({
    'image': Image(shape=(None, None, 3), dtype=tf.uint8),

 'image/filename': Text(shape=(), dtype=tf.string),
    'image/id': tf.int64,
    'panoptic_image': Image(shape=(None, None, 3), dtype=tf.uint8),
    'panoptic_image/filename': Text(shape=(), dtype=tf.string),
    'panoptic_objects': Sequence({
        'area': tf.int64,
        'bbox': BBoxFeature(shape=(4,), dtype=tf.float32),
        'id': tf.int64,
        'is_crowd': tf.bool,
        'label': ClassLabel(shape=(), dtype=tf.int64, num_classes=133),
    }),
})
```

5) 데이터 예시



6) 활용 예

이미지내의 사물의 경계를 탐지하는 Detection과 사람을 감지하고 몸을 분할하여 인체의 3d표면에 매핑하는 DensePose, 배경이 아닌 물체를 탐지하는 Stuff등 6개의 분야에서 대회를 진행한다.

Detection, Keypoints, Stuff, Panoptic, Captions의 성능을 확인하는데에도 사용된다.

COCO데이터셋의 경우 많은 클래스를 정의하고 있어 논문에서 COCO의 카테고리를 사용하기도 한다.

'U-Net 구조를 이용한 이미지에서의 보행자 분할'(김승택)에서는 자율주행 자동차에서의 보행자 인식 방법을 경계박스로 인해 정보 손실을 줄일 수 있는 보행자 분할 방법을 제시하였는데 COCO 데이터셋의 사람 카테고리를 이용하였다.

4. BDD100K

1) 데이터 설명

BDD100K는 'Berkeley Deep Drive'의 약자로 40초의 비디오 시퀀스, 720픽셀 해상도, 초당 30프레임 고화질로 취득된 10만개의 비디오 시퀀스로 구성된다. 거친 주행 환경 구현, GPS 정보, IMU 데이터 및 타임 스탬프가 포함되어 있다. 녹화된 비디오는 비 내리는 날씨, 흐린 날씨, 맑은 날씨, 안개와 같은 다양한 날씨 조건이 기록되어 있다. 데이터 세트는 낮과 밤이 적절한 비율로 기록되어 있다. image tagging(이미지 태그 지정), lane detection(차선 감지), drivable area segmentation(주행영역 분할), object detection(객체 감지), semantic segmentation(이미지 분할), instance segmentation(인스턴스 분할), multi-object detection tracking(다중 객체 감지 추적), multi-object segmentation tracking(다중 객체 분할 추적), domain adaptation(도메인 적응), imitation learning(모방 학습) 총 10가지 작업으로 구성된다.이미지에 쉽게 주석 처리하기 위해 수직 차선은 적색, 평행 차선은 청색으로 구분하였다. 적색 표시 있는 운전 경로와 청색 표시가 있는 대안 운전 경로로 주행 가능 구역을 구분한다. 차선 구분을 위한 데이터는 쉽게 주석처리하기 위해 수직 차선은 적색, 평행 차선은 청색으로 구분하였다. 도로 객체 탐지 데이터셋은 100,000개의 이미지에 주석이 달린 2D Bounding Box가 포함되어 있다.

2) 양

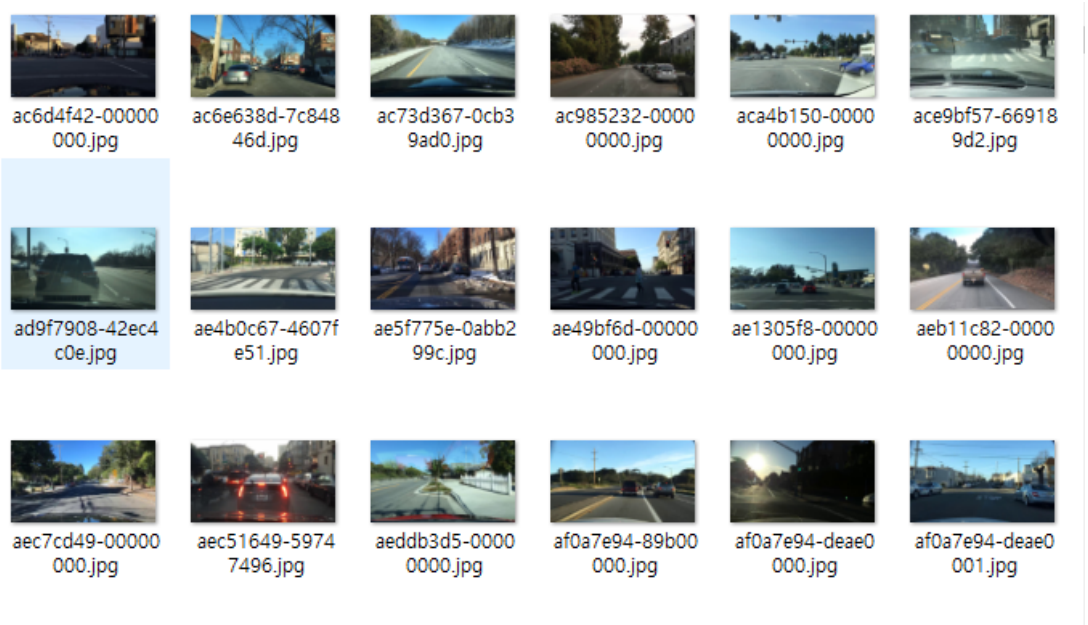
주석 처리된 10만개 이상의 다양한 비디오 클립에 주행 장면의 가장 큰 데이터셋을 수집하고 주석을 달았다.

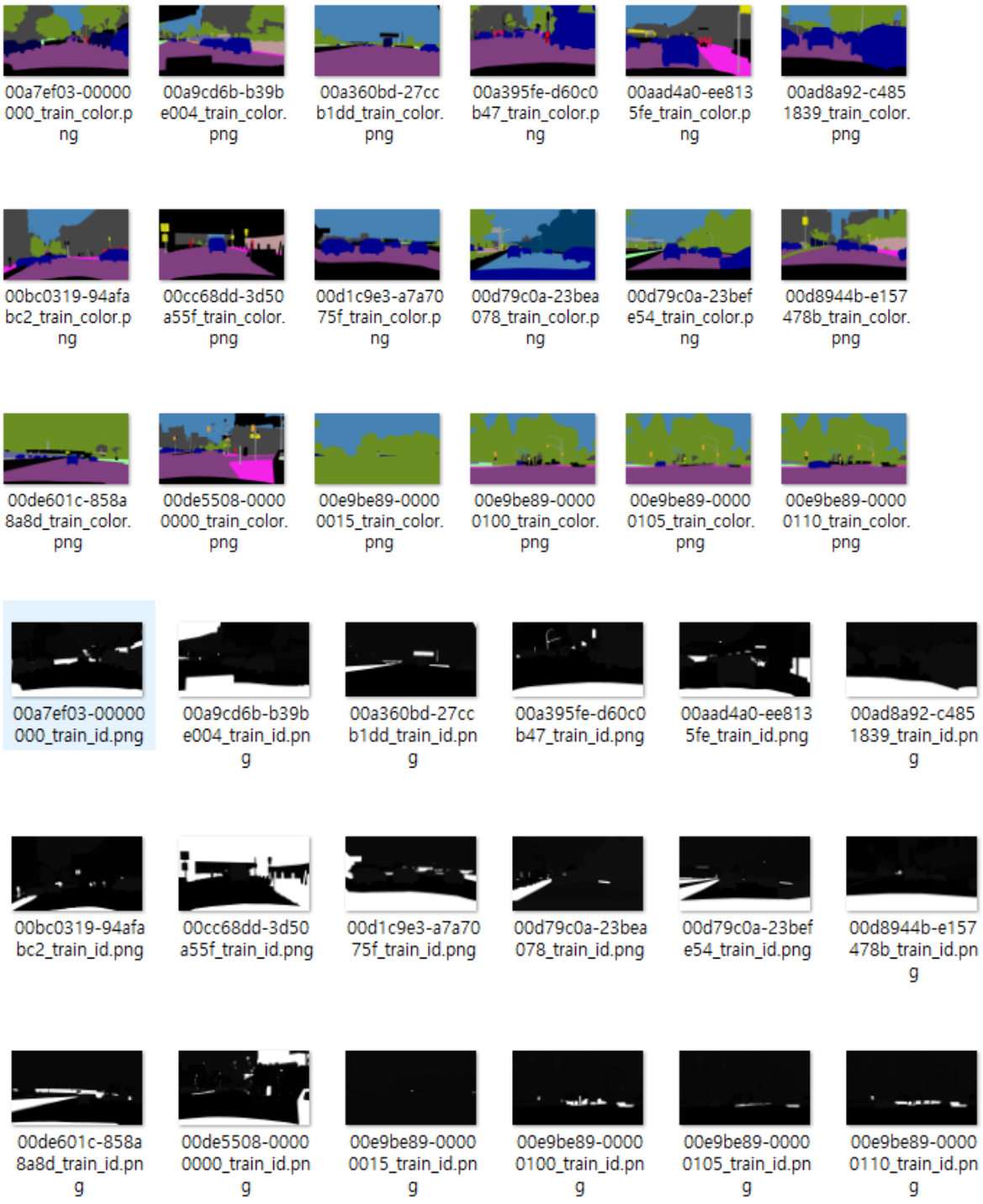
3) 클래스

pedestrian, rider, other person, car, bus, truck, train, trailer, other vehicle, motorcycle, bicycle의 클래스로 이루어져 있다.

주행 영역 분류를 위해 70,000개 훈련용, 10,000개 검증용 픽셀맵 파일도 제공한다. 객체 분류은 7,000개 훈련용 데이터셋과 1,000개의 검증용 데이터셋을 제공하고 있다.

4) 데이터 예시 (images, colo_labels, labels)





5) 활용 예

도로/ 포장도로의 보행자 탐지의 컴퓨터 비전 알고리즘 구현에 유용하며 보행자 뿐만 아니라 도로 객체 감지, 객체 분류, 차선 구분 등의 머신러닝 학습, 시뮬레이션에 사용될 수 있다. 눈, 비, 흐림 등 다양한 날씨를 포함하고, 야간과 터널 등 조도가 낮은 상황에서도 수집이 됐기 때문에 객체 인지 모델을 일반화가 가능하도록 학습시키는 것이 가능하다. 다양한 상황에서 객체를 정확하게 인식하여 자율주행의 기술의 정확성, 안전성을 높일 수 있다.

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

- 코드 설명, 구성, 활용 및 결과 등 코드를 이해하는데 필요한 정리

1. faster R-CNN

[모델 설명]

: 2-Stage Object Detector로 Region Proposal (탐색 영역 찾기)과 Detection(해당 영역 분류) 두 가지 과정이 순차적으로 수행되는 방법론이다. R-CNN은 Region Proposal마다 하나의 CNN을 돌리고 Bounding Box 모델을 동시에 학습해야 하니 학습에 걸리는 시간이 엄청났다. 이 문제를 해결하기 위해 Fast R-CNN이 개발되었다. Feature을 이미지로부터 뽑아내는 것이 아니라 CNN을 거친 Feature Map상에서 RoI Pooling을 사용하여 Feature을 뽑아낸다. 하나의 CNN만 돌아가면 되므로 object Detection의 수행속도가 빨라졌다. 여기에 Region Proposal을 생성하는 방식을 뉴럴 네트워크로 해결한 Faster R-CNN이 등장했다. Faster R-CNN은 Region Proposal을 생성하는 방법 자체를 CNN 내부에 네트워크 구조로 넣어놓은 모델이다. 이 네트워크를 RPN(Region Proposal Network) 이라고 하며, RPN을 통해서, RoI Pooling을 수행하는 레이어와 Bounding Box를 추출하는 레이어가 같은 특징 맵을 공유할 수 있다.

[Faster R-CNN의 구조]

: Convolution Layer를 거쳐 입력 이미지의 특징 추출 → RPN이 feature map(출력 특징 맵)에서 Region Proposal(탐색 영역) 추출 → RoI Pooling Layer에서 Region Proposal(탐색영역)들에 대해 RoI 풀링 수행

[RPN의 작동 방식]

: 출력 특징 맵 위를 지정한 크기의 window가 슬라이딩하면서 각 지점마다 지정된 크기의 anchor를 지정한 개수만큼 생성한다. 모든 anchor에 대해 가능한 Bounding Box의 좌표와 그 안에 물체가 들어있을 확률을 계산한다. k개의 anchor box에 대해 각각의 위치, 크기, Bounding Box로써의 점수들을 계산한다.

- cls layer : 해당 박스 안에 물체의 존재 여부를 분류
  2k개의 점수들은 해당 anchor 내에 물체의 존재 여부 확률을 계산한다.
- reg layer : Bounding box의 정확한 위치를 예측한다.
  4k개의 좌표 값들은 해당 anchor의 x,y좌표와 Width, Height값을 가지고 있다.

두 layer들의 학습을 통해 물체가 들어있는 정확한 Bounding Box = RoI들을 추출할 수 있다.

[RoI Pooling]

: RPN이 생성한 RoI들에 대해 각 Region에 대한 특징 맵이 모두 동일하게 고정된 사이즈로 생성되도록 함. 이를 통해 각 RoI 내 물체들의 분류를 시행할 수 있다.

- Multi-task Loss를 사용하여 모든 트레이닝을 하나의 Loss함수로 진행. RPN을 학습하기 위한 Loss와 분류기를 학습하기 위한 Loss를 더한 모습. 학습을 더 빠르게, 통합적으로 진행할 수 있다.

[코드 구성 활용 및 결과]

- 설치되어야하는 환경

  cython

  ciffi

  opencv-python

  scipy

  msgpack

  easydict

  matplotlib

  pyyalm

  tensorboardX

- 결과

  다음은 모델을 구현한 결과의 일부이다.

  1). PASCAL VOC 2007 (Train/Test: 07trainval/07test, scale=600, ROI Align)

모델	#GPU	배치크기	lr	lr-decay	max-epoch	시간/epoch	mem/GPU	mAP
VGG-16	1	1	1e-3	5	6	0.76hr	3265MB	70.1

  2). COCO (Train/Test: coco_train+coco_val-minival/minival, scale=600, max_size=1000, ROI Align)

모델	#GPU	배치크기	lr	lr-decay	max-epoch	시간/epoch	mem/GPU	mAP
Res-101	8	24	1e-2	4	6	5.4hr	10659MB	33.9

- 실행하기 위한 전제 조건
  - Python 2.7 or 3.6
  - Pytorch 0.4.0
  - CUDA 8.0 or higher
- 학습을 위한 데이터 셋
  - PASCAL_VOC 07 + 12
  - COCO
  - Visual Genome

: kit으로 연결되는 url 제공

- pre-trained 모델
  - VGG16과 ResNet101을 pre-trained 모델 사용.
- train 학습

```
CUDA_VISIBLE_DEVICES=$GPU_ID python trainval_net.py \
                   --dataset pascal_voc --net vgg16 \
                   --bs $BATCH_SIZE --nw $WORKER_NUMBER \
                   --lr $LEARNING_RATE --lr_decay_step $DECAY_STEP \
                   --cuda
```

- 다음 코드를 실행해서 faster R-CNN 모델을 훈련할 수 있다.
- vgg16 대신 resnet101 대입 가능하다.
- bs(batch size)의 기본 값은 1이며 batch_size와 worker_number는 GPU메모리 크기에 따라 적절하게 설정할 수있다. 12G 메모리의 Titan Xp에서는 최대 4개까지 가능하다.

  batch_size = forward와 backward를 한번에 얼만큼의 데이터씩 진행할 것인지를 의미한다. 모델의 가중치를 한번 업데이트 시킬 때마다 사용되는 샘플들의 묶음을 의미한다. 1000개의 샘플이 있는데 batchsize가 50이면 20번의 가중치 업데이트가 일어난다.

- test 시험

```
python test_net.py --dataset pascal_voc --net vgg16 \
                   --checksession $SESSION --checkepoch $EPOCH
--checkpoint $CHECKPOINT \
                   --cuda
```

- vgg16 대신 resnet101 대입 가능하다.
- session, epoch, checkpoint를 설정한다. (예 : SESSION = 1, EPOCH = 6, CHECKPOINT = 416).

  epoch : 모든 트레이닝 데이터에 대해서 forward와 backward pass를 진행한 상태를 의미한다. 학습의 횟수를 의미한다. 20번의 가중치 업데이트가 있을 때 epoch가 6번이면 120번 가중치 업데이트된다.
  check point : 재개를 위한 저장 포인트

- demo code

```
python demo.py --net vgg16 \
               --checksession $SESSION --checkepoch $EPOCH --checkpoint
$CHECKPOINT \
               --cuda --load_dir path/to/model/directoy
```

  - ROOT/ images 폴더에서 탐지 결과를 찾을 수 있다.
  - 데모 코드는 VOC 데이터셋 카테고리만 지원하므로 라인을 수정해서 적용하면 된다.

- 탐지 결과

- 활용분야

딥러닝이 등장하면서 카메라 영상을 이용하여 차량, 보행자, 도로등을 인지하는 자율주행 기술이 발전하고 있다. Faster R-CNN은 물체 탐지 기법중 빠른 처리 속도를 보여주며 조도에 강하다고 알려져있다. 따라서 전방 끼어들기 차량에 대한 감지를 위해 Faster R-CNN을 이용하여 차량 종류, 끼어들기 상태에 따른 차량 부분 형상, 색, 환경 인자(조도, 배경 등)의 변화에 대한 차량 감지 성능 영향을 분석한 연구가 진행되었다. 도로 상황에서 빈번하게 검출되는 신호등, 사람, 교통 표지판, 차량을 실시간으로 Embdded Board 환경에서 검출할 수 있도록 경량화하는 연구 등에서도 정확도가 높은 Faster R-CNN이 객체 탐지를 위해 사용되었다.

2. YOLOv5

[모델 설명]

R-CNN 계열의 네트워크들은 영상에 물체가 있을 것 같은 후보를 먼저 뽑는다. 후보로 뽑힌 영역들은 classifier의 많은 레이어들을 통과하며 어떤 클래스인지 검열된다. 분류가 끝난 후, Bounding Box를 다시 정의하고, 중복 검출을 지운 후 박스안을 다시 평가하는 작업을 진행한다. Proposal의 수도 많고 오버헤드도 크기 때문에 속도가 느리다. 이미지를 여러장으로 분할하고 분석하는 R-CNN 모델과 달리 YOLO는 이미지 전체를 한번만 보게 된다. 또한 1-Stage Object Detector로 Region Proposal (탐색 영역 찾기)과 Detection(해당 영역 분류) 두가지 과정을 한번에 처리하는 방법론을 사용한다. 전체 이미지에서 추출한 특징을 Bounding Box를 예측하기 위해 사용한다. 동시에 네트워크는 이미지의 클래스를 예측한다. 영상에서 물체를 찾는 방법에는 두가지가 있는데 Proposal방식과 Grid 방식이다. Proposal 방식은 수많은 제안을 하는 selective search로 시간이 오래 걸린다. YOLO는 grid 방식을 사용하여 속도를 현저하게 높일 수 있었다. S X S 크기의 grid로 나눈다. grid cell의 개수가 곧 proposal의 수이며 오버헤드가 전혀 없다. 각각의 grid 셀은 Bounding Box와 그 박스들에 대한 점수들을 반환한다. 이 점수는 박스 안에 물체를 포함하고 있는지와 분류한 클래스가 얼마나 맞는지를 반영한다. 각 Bounding Box는 x,y. width, height, 신뢰도 값을 갖는다. 신뢰도는 예측 box와 실제 정답 사이의 IOU를 의미한다. IOU(Intersection Over Union)는 교집합/ 합집합 이라는 뜻으로 모델이 예측한 결과와 GT의 교집합, 합집합을 통해 측정한다. 교집합/합집합으로 계산한다. 이 값들은 모두 (0,1)의 범위로 정규화된다. 하나의 그리드 셀당 클래스 가능성과 박스 가능성 예측을 곱해 셀 하나의 클래스 가능성 세트를 예측한다. 얇은 경계 박스들은 threshold(보통 0.5)보다 작으면 지워진다. 남은 후보 박스들은 NMS알고리즘을 선별하여 걸러진다. YOLO는 7*7개의 그리드 셀에서 각각 2개의 경계박스 후보를 예측한다. 클래스의 신뢰도 점수가 thresh보다 낮은 것은 모두 0으로 셋팅한다. 신뢰도 점수가 낮다면 안에 무엇이 들었을지는 모르지만 최소한 그 클래스는 아닐 것이라고 판단한다. 경계박스 신뢰도 순으로 내림차순 한 다음 두 경계박스의 겹치는 부분이 크다면 신뢰도가 낮은 것을 0으로 지워준다. 겹치는 부분이 별로 없을 경우 다른 물체가 있을 수도 있으므로 그냥 넘어간다. 나머지 클래스에 대해서도 한개씩 같은 작업을 반복해준다. 최종적으로 신뢰점수가 threshold보다 낮은 값은 나중에 지워준다. 아래 수식은 multi-loss function으로 모든 bounding box에 대해 얼마나 오차가 큰지 계산한다. 신버전들은 YOLO모델에 여러 알고리즘을 추가하여 정확도와 속도를 높여 나오고 있다.

Bag of Freebies와 같이 training cost를 증가시켜 정확도를 높이는 방법들을이 있으며 Data Augmentation, Regularization, Loss Function을 추가, 변형한다. Bag of Specials와 같이 오로지 Inference cost만 증가시켜서 정확도를 높이는 방법들이 있으며 Enhancement of Receptive Field, Feature Intergration, Activation Function, Attention Module, Normalization, Post Processing을 추가, 변형한다.

[코드 구성 활용 및 결과]

- 설치되어야 하는 환경
- python 3.8

  -베이스

  3.2.2ver matplotlib

  1.18.5ver numpy

  4.1.2ver opencv-python

  5.3.1ver PyYAML

  1.4.1ver scipy

  1.7.0ver torch

  0.8.1ver torchvision

  4.41.0ver tqdm

  -로깅

  2.4.1ver tensorboard

  -플로팅

  0.11.ver 0searborm

  -내보내기

  4.1ver coremltools

  1.8.1ver onnx

  0.19.2ver scikit-learn

- 모델 사이즈에 따른 결과

  사이즈가 작을 수록 속도가 빠르나 정확도는 떨어진다.

model	size	AP(val)	AP(test)	AP(50)	Speed (v100)	FPS (v100)	params	GFLOPS
YOLOv5s	640	36.8	36.8	55.6	2.2ms	455	7.3M	17.0
YOLOv5m	640	44.5	44.5	63.1	2.9ms	345	21.4M	51.3
YOLOv5l	640	48.1	48.1	66.4	3.8ms	264	47.0M	115.4
YOLOv5x	640	50.1	50.1	68.7	6.0ms	167	87.7M	218.8

*AP = 하나의 Class 마다 11가지 recall 값에 따른 Precision의 평균을 낸 것.

Precision과 Recall은 반비례 관계를 갖기 때문에 Object Detection에서는 **AP**(Average Precision) 지표를 사용한다. Recall을 0에서 1까지 0.1씩 증가시킬 때 Precision은 필연적으로 감소하는데 각 단위마다 Precision 값을 계산하여 평균을 낸다.

*FPS : 초당 몇 장의 이미지가 처리 가능한지를 나타낸다. 어떤 하드웨어를 사용하였는지, 어떤 size의 이미지를 사용하였는지에 따라 수치가 달라지기 때문에 상대적인 비교만 가능하다는 단점이 있다. 정확도만큼이나 중요한 성능 지표이다.

* GFLOPS : 컴퓨터의 1초당 부동 소수점 연산의 실행 횟수를 10억(=109) 단위로 표현한 것.

- 환경

CUDA/CUDNN, Python and PyTorch가 installed 되어있는 아래 환경에서 구현가능하다.

- Google Colab and Kaggle
- Google Cloud Deep Learning VM
- Amazon Deep Learning AMI
- Docker Image.

- 구현

: 해당 소스코드를 이용하여 image, video의 객체 탐지가 가능하다.

```
$ python detect.py --source 0  # webcam
                            file.jpg  # image
                            file.mp4  # video
                            path/  # directory
                            path/*.jpg  # glob

rtsp://170.93.143.139/rtplive/470011e600ef003a004ee33696235daa  # rtsp stream
                            rtmp://192.168.1.105/live/test  # rtmp stream

http://112.50.243.8/PLTV/88888888/224/3221225900/1.m3u8  # http stream
```

: 이미지를 이용하여 추론하면 아래 예시와 같은 결과가 나온다.

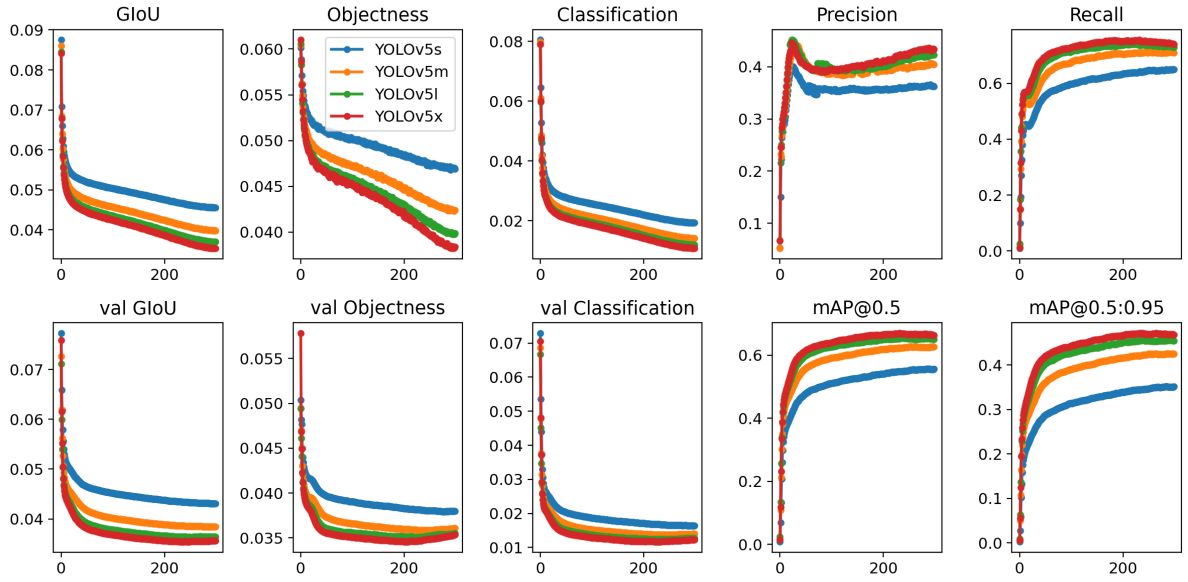
```
$ python detect.py --source data/images --weights yolov5s.pt --conf 0.25

Namespace(agnostic_nms=False, augment=False, classes=None, conf_thres=0.25,
device='', exist_ok=False, img_size=640, iou_thres=0.45, name='exp',
project='runs/detect', save_conf=False, save_txt=False, source='data/images/',
update=False, view_img=False, weights=['yolov5s.pt'])
YOLOv5 v4.0-96-g83dc1b4 torch 1.7.0+cu101 CUDA:0 (Tesla V100-SXM2-16GB,
16160.5MB)

Fusing layers...
Model Summary: 224 layers, 7266973 parameters, 0 gradients, 17.0 GFLOPS
```

```
image 1/2 /content/yolov5/data/images/bus.jpg: 640x480 4 persons, 1 bus, Done.
(0.010s)
image 2/2 /content/yolov5/data/images/zidane.jpg: 384x640 2 persons, 1 tie,
Done. (0.011s)
Results saved to runs/detect/exp2
Done. (0.103s)
```

- 아래 명령을 실행하여 repository에 준비된 COCO 데이터셋에 대한 결과 재현이 가능하다.
- s,m,l,x의 교육시간은 단일 테슬라 V100에서 2/4/6/8일 소요된다.
- 다중 GPU의 속도가 더 빠르다.

```
$ python train.py --data coco.yaml --cfg yolov5s.yaml --weights '' --batch-size 64
                                        yolov5m                                40
                                        yolov5l                                24
                                        yolov5x                                16
```

- 모델 사이즈에 따른 훈련 결과



- 활용 예

  ADAS 기능인 차선 유지 운전에 대해 차선 이탈에 대한 성능을 향상 시키고자 YOLO의 빠른 인식을 사용하고 카메라로 부터 주변 환경으로부터 영향을 받는 상황을 인지하고 주행 데이터를 수집하여 관심 영역을 추출하는 차선검출 시스템을 구현할 수 있다. 주행환경 이미지를 수집하여 차등 간격의 조향각을 결정하고 자율주행의 안정성을 높이는데 사용된다.

3) 2)의 정리한 코드 중 하나 실행해서 결과 확인

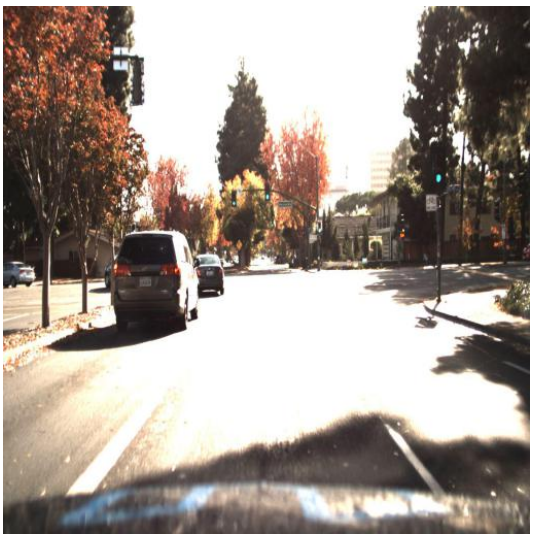
- 구현 환경 (시스템, lib, docker 등 실행 환경) 및 실행에 관련된 코드 설명 등

- 코랩을 이용해 구현

- 해당 오픈 소스에서 제안한 COCO데이터셋은 사이즈가 너무 크고 시간이 많이 소요되므로 roboflow에서 Udacity Self Driving Car Dataset을 다운로드하여 학습 데이터로 사용하였다.

```
# 코랩 폴더에 파일 다운로드
!curl -L "https://public.roboflow.com/ds/de2NaM2tt2?key=Lgp4k9JMFo" >
roboflow.zip; unzip roboflow.zip; rm roboflow.zip
```

[이미지 예시]



```
2 0.20416666666666666 0.5 0.009375 0.02666666666666667
10 0.3375 0.48833333333333334 0.03125 0.04666666666666667
1 0.3458333333333333 0.4991666666666667 0.025 0.035
1 0.4088541666666667 0.4941666666666667 0.028125 0.03833333333333334
1 0.9052083333333333 0.44583333333333336 0.18541666666666667 0.13166666666666668
```

[label 데이터 예시]

- class x y width height 순으로 정보가 기재되어 있음

```
# yolov5 모델 깃허브에서 clone
%cd /content
!git clone https://github.com/ByeolHan/yolov5.git

# 환경 구현
```

```
%cd /content/yolov5/
!pip install -r  requirements.txt
```

```
# Udacity Self Driving Car Dataset class 개수, 내용 확인
%cat /content/dataset/data.yaml

# 데이터 개수 확인 15000개
%cd /
from glob import glob

img_list = glob('/content/dataset/export/images/*.jpg')

print(len(img_list))

#트레이닝 셋 80%, 테스트 셋 20%로 나누기트레이닝 셋 80%, 테스트 셋 20%로 나누기
12000개/3000개
from sklearn.model_selection import train_test_split
train_img_list, val_img_list = train_test_split(img_list, test_size = 0.2,
random_state = 2000)

print(len(train_img_list), len(val_img_list))

# train, val 이미지를 txt 파일로 저장한 후 train 데이터 셋을 train.txt로
val테스트 데이터 셋을 val.txt로 train 데이터 셋을 train.txt로 val테스트 데이터
셋을 val.txt로
with open('/content/dataset/train.txt','w') as f:
  f.write('\n'.join(train_img_list) + '\n')

with open('/content/dataset/val.txt','w') as f:
  f.write('\n'.join(val_img_list) + '\n')

import yaml

with open('/content/dataset/data.yaml','r') as f:
  data = yaml.load(f)

  print(data)

  data['train'] = '/content/dataset/train.txt'
  data['val'] = '/content/dataset/val.txt'

import yaml

with open('/content/dataset/data.yaml','r') as f:
```

```
  data = yaml.load(f)

print(data)

# 데이터 설정
data['train'] = '/content/dataset/train.txt'
data['val'] = '/content/dataset/val.txt'

with open('/content/dataset/data.yaml','w') as f:
  yaml.dump(data,f)

print(data)

# 학습하기
%cd /content/yolov5/

!python train.py --img 416 --batch 16 --epochs 5 --data
/content/dataset/export/data.yaml --cfg ./models/yolov5s.yaml --weights
yolov5s.pt --name selfdrivingyolov5s_results
```

```
/content/yolov5
github: up to date with https://github.com/ByeolHan/yolov5 ✅
YOLOv5 🚀 v4.0-188-gc8c8da6 torch 1.8.1+cu101 CUDA:0 (Tesla K80, 11441.1875MB)
```

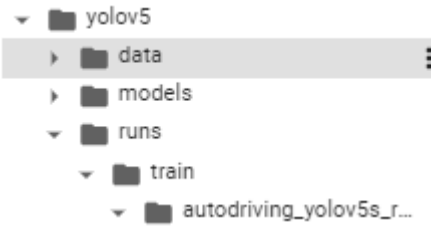
: Tesla K80, 11기가 정도의 GPU를 사용하고 있다는 걸 알 수 있다.

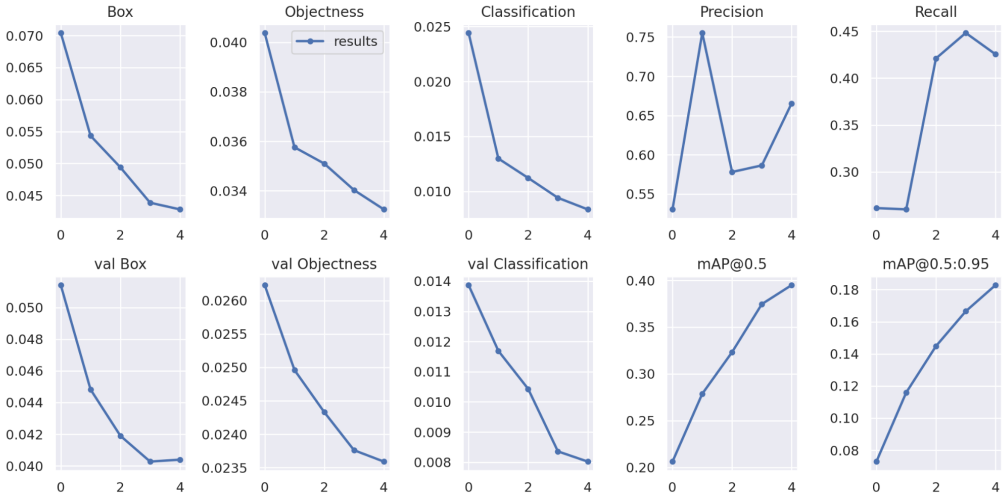
```
     Epoch   gpu_mem       box       obj       cls     total    labels  img_size
       0/4     6.44G   0.07041   0.04038   0.02441    0.1352        93       640: 100% 750/750 [10:24<00:00,  1.20it/s]
                 Class     Images     Labels          P          R     mAP@.5 mAP@.5:.95: 100% 94/94 [01:13<00:00,  1.27it/s]
                   all       3000      19549      0.531      0.261      0.206      0.073

     Epoch   gpu_mem       box       obj       cls     total    labels  img_size
       1/4     6.45G   0.05434   0.03576   0.01299    0.1031        81       640: 100% 750/750 [10:00<00:00,  1.25it/s]
                 Class     Images     Labels          P          R     mAP@.5 mAP@.5:.95: 100% 94/94 [01:06<00:00,  1.41it/s]
                   all       3000      19549      0.755       0.26      0.279      0.116

     Epoch   gpu_mem       box       obj       cls     total    labels  img_size
       2/4     6.45G   0.04941    0.0351   0.01123   0.09574       131       640: 100% 750/750 [09:54<00:00,  1.26it/s]
                 Class     Images     Labels          P          R     mAP@.5 mAP@.5:.95: 100% 94/94 [01:06<00:00,  1.42it/s]
                   all       3000      19549      0.578      0.421      0.323      0.145

     Epoch   gpu_mem       box       obj       cls     total    labels  img_size
       3/4     6.45G   0.04388   0.03403  0.009421   0.08733       125       640: 100% 750/750 [09:55<00:00,  1.26it/s]
                 Class     Images     Labels          P          R     mAP@.5 mAP@.5:.95: 100% 94/94 [01:05<00:00,  1.43it/s]
                   all       3000      19549      0.586      0.448      0.375      0.167

     Epoch   gpu_mem       box       obj       cls     total    labels  img_size
       4/4     6.45G   0.04284   0.03325  0.008372   0.08446       142       640: 100% 750/750 [09:54<00:00,  1.26it/s]
                 Class     Images     Labels          P          R     mAP@.5 mAP@.5:.95: 100% 94/94 [01:13<00:00,  1.28it/s]
                   all       3000      19549      0.666      0.425      0.395      0.183
                 biker       3000        428      0.389      0.521      0.416      0.162
                   car       3000      12785      0.742      0.793      0.819       0.46
            pedestrian       3000       2207      0.576      0.479      0.493      0.181
          trafficLight       3000        522       0.39      0.553      0.388      0.183
    trafficLight-Green       3000       1122      0.533      0.614       0.52      0.183
trafficLight-GreenLeft       3000         58          1          0     0.0416     0.0182
    trafficLight-Red       3000       1320      0.631      0.692      0.689      0.294
trafficLight-RedLeft       3000        328      0.434      0.268      0.261      0.119
 trafficLight-Yellow       3000         36          1          0    0.00495     0.0019
trafficLight-YellowLeft       3000          7          1          0   0.000173   5.69e-05
              truck       3000        736      0.625       0.76      0.711       0.41
5 epochs completed in 0.936 hours.
```

: epoch마다 image 개수 label 개수 prediction recall mAP0.5 mAP0.95 순으로 보여진다.

- 학습 결과

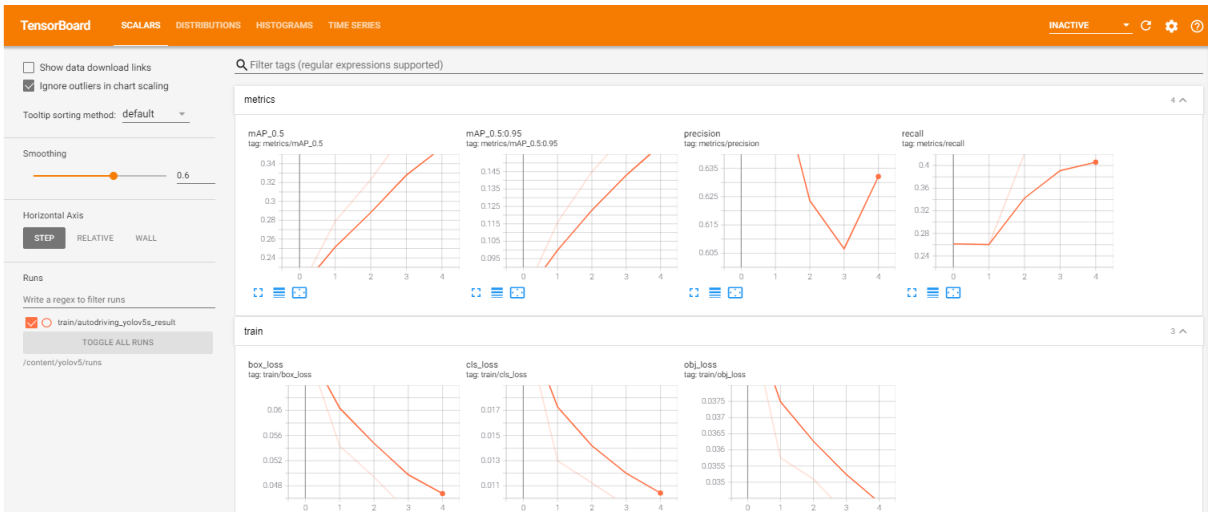
  학습이 끝나면 지정한 `autodriving_yolov5s_result` 폴더에서 결과를 확인할 수 있다.





- 텐서보드 학습 결과

```
%load_ext tensorboard
%tensorboard --logdir /content/yolov5/runs
```



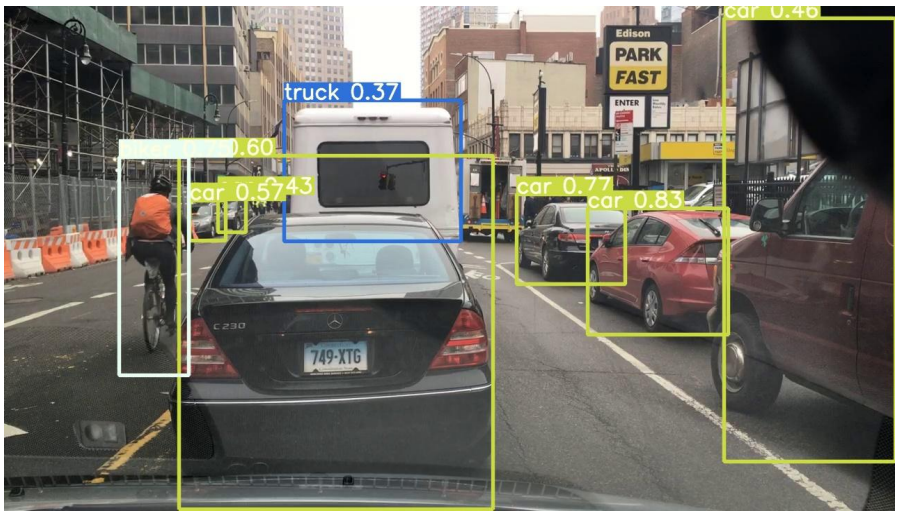
- 테스트셋 학습 결과 확인하기

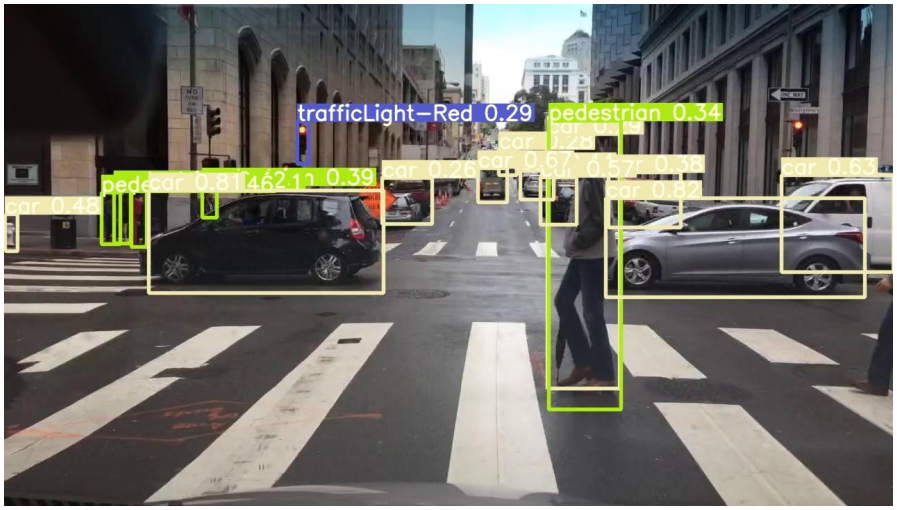
```
from IPython.display import Image
import os

val_img_path = val_img_list[0]

!python detect.py --weights
/content/yolov5/runs/train/autodriving_yolov5s_result/weights/best.pt --img
416 --conf 0.5 --source "{val_img_path}"
```

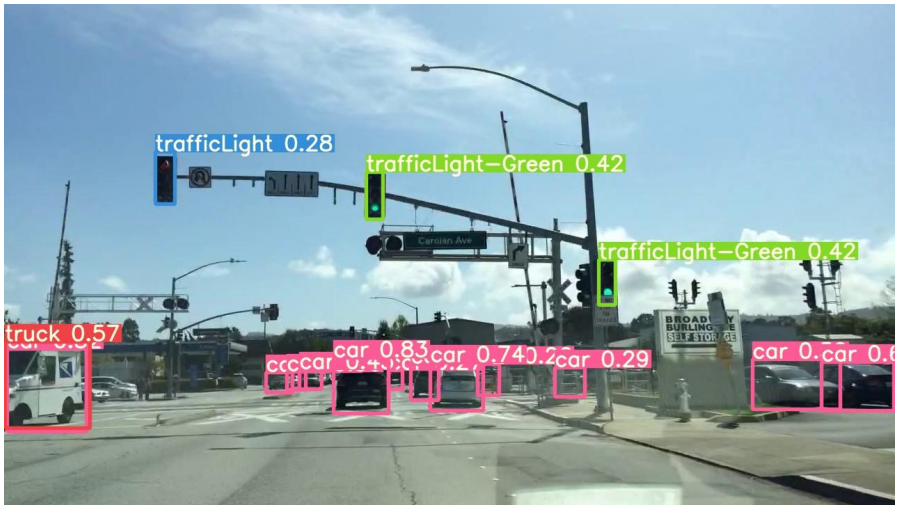
: 차량을 인식하고 있는 것을 확인할 수 있다

- 오픈데이터셋(bdd100k)를 넣어 모델 성능 확인하기

```
# 파일 업로드
from google.colab import files
uploaded = files.upload()

# best weight 경로와 파일 경로를 지정해서 객체 탐지
!python detect.py --weights
/content/yolov5/runs/train/autodriving_yolov5s_result/weights/best.pt
--source /content/yolov5/e52cac9f-87266c6f.jpg
```







: 차, 트럭, 사람, 신호등을 인지하는 모습을 볼 수 있다.

참고 문헌

https://cocodataset.org/#detection-2020

https://europe.naverlabs.com/research/computer-vision/proxy-virtual-worlds/

http://www.cvlibs.net/datasets/kitti/

https://pjreddie.com/projects/pascal-voc-dataset-mirror/

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

https://ctkim.tistory.com/91

김승택, & 이효종. (2019). U-Net 구조를 이용한 이미지에서의 보행자 분할. 한국정보처리학회 학술대회논문집, *26*(1), 519-521.

이선영, 김민구, & 김정하. (2018). 카메라를 활용한 딥러닝 기반 자율주행자동차의 사각 지대 객체 검출 및 경고 시스템에 관한 연구. 한국자동차공학회 추계학술대회 및 전시회, 615-619.

박상배, & 김정하. (2020). 도심지 자율주행을 위한 머신러닝 기반의 실시간 다 객체 인식 방법. 제어로봇시스템학회 논문지, *26*(6), 499-505.

전현기, & 송봉섭. (2017). 전방 끼어들기 차량 감지를 위한 Faster R-CNN 적용 연구. 한국자동차공학회 춘계학술대회, 726-727.