자율주행인지 REPORT

자동차IT융합학과

20173416 이민지

GITHUB ADDRESS : https://github.com/minji09/self-driving

1. 자율주행 인지에 관련 Data Set

1) KAIST Multispectral Pedestrian Detection Benchmark

다중 스펙트럼 보행자 인지 벤치마킹

- **목표** :
  1. 정렬된 다중 스펙트럼 보행자 데이터 셋의 컬러, 열 이미지 쌍 제시
  2. 컬러와 열 채널 간 상호 보완 관계 분석 및 제안
  3. 확장 ACF의 몇 가지 조합 제안

- **소개 :**

  국내 공개 데이터 셋으로 HD급의 화질로 촬영된 9만 5,000여 장의 보행자 데이터 셋입니다.

  정렬된 다중 스펙트럼 (RGB 컬러 + 열) 영상을 포착하기 위해 컬러 카메라, 열 카메라 및 빔 스플리터로 촬영되었습니다.

  빛의 변화를 고려하기 위해 낮과 밤 시간대의 다양한 정규 교통 장면들을 포착하였습니다.

  주석(라벨)은 경계 상자 사이의 시간적 대응을 포함하고 있습니다.

- **크기 및 범위 :**

  이 데이터 셋는 95k개의 색 열 쌍(640x480, 20Hz)으로 구성되며 모든 쌍은 총 103,128개의 상세한 라벨과 1,182명의 고유 보행자에 대해 라벨이 달려있습니다.

  데이터 셋에서 사람은 평균적으로 74.80 프레임으로 나타납니다. (3.74초)

- **훈련(Train), 검증(Test) 데이터 셋 설정 :**

  무작위 구분과 비교하여 특정 장면에서의 데이터 편향과 오버피팅을 피하는 것에 도움을 줍니다.

  컬러, 열 이미지 쌍을 train, test 데이터셋으로 나눈 기준은 아래와 같습니다.

  1. 데이터 셋에서 나타난 보행자 수 유사
  2. 데이터 셋에서 주간/야간 이미지의 프레임 수 유사
  3. 두 데이터 셋이 일치하지 않음

2) Waymo Open Dataset

Sharing our self-driving data for research

- **소개 :**

  25개의 도시에서 1000만 마일이 넘는 자율주행 거리를 수집한 자율주행 데이터셋입니다.

- **크기 및 범위:**

  1000 개의 주행 세그먼트들이 수록되어 있으며 각 세그먼트들은 센서당 10Hz에서 20만 프레임에 해당하는 연속 주행 20초를 포착합니다.

- **다양한 주행 환경 :**

  이 데이터셋은 여러 장소에서 광범위한 주행 조건(낮과 밤, 새벽과 황혼, 햇빛과 빛)을 포착한 밀집한 도시 및 교외 환경을 포함하고 있습니다.

- 센서 종류 **:**

  각 조각들은 1개의 중거리 라이더, 4개의 단거리 라이더, 5개의 전면 및 측면 카메라의 센서 데이터를 포함하고 있습니다

- **정밀한 라벨 :**

  차량, 보행자, 자전거 탄 사람 등 정밀하게 라벨을 부착한 라이더 프레임과 이미지를 포함합니다.

  총 1200만개의 3D 라벨링과 120만개의 2D 라벨링이 되어있습니다.

  차량, 보행자, 자전거, 표지판 객체 등급을 4단계로 나누어 라벨링을 했습니다.

3) BDD100K (Berkely Deep Drive)

A Diverse Driving Dataset for Heterogeneous Multitask Learning

- **소개 :**

  BDD100K는 가상 주행 장면에 대한 새롭고 다양한 대규모 데이터셋입니다.

  현실적인 주행 시나리오를 다루며 다양한 환경 영역에서 관심 있는 범주의 위치 배열과 모습 변화를 더 많이 포착하였습니다.

  UC 버클리와 외부 연구진들은 주석 처리된 10만개 이상의 다양한 비디오 클립에 주행 장면의 가장 큰 데이터셋을 수집하고 주석을 달았습니다.

- **평가방법 :**

이미지 태그 지정, 차선 감지, 주행 영역 분할, 도로 객체 감지, 이미지 분할, 인스턴스 분할, 다중 객체 감지 추적, 다중 객체 분할 추적, 도메인 적응 및 모방 학습의 10가지 작업으로 구성되어집니다.

- **크기 및 범위 :**

  각 비디오는 길이는 40초이며 높은 해상도(100,000 HD) 를 가지고 있습니다.

  비디오에는 궤적 정보를 위한 GPU/IMC 데이터가 함께 제공된니다.

  데이터셋은 1억개 이상의 프레임으로 된 1100시간 이상의 주행 경험을 가지고 만들어졌습니다.

- **다양한 주행 환경 :**

  데이터셋은 훈련 모델에 유용한 지리적, 환경적, 환경 다양성을 보유하고 있습니다.

- **도로 객체 감지 :**

  2D bounding box 는 버스, 신호동, 교통 표지판, 사람, 자전거, 트럭, 모터, 자동차, 기차 그리고 탑승자를 위한 10만개의 이미지에 주석을 달았습니다.

- **인스턴스 분할 :**

  pixel level 및 풍부한 인스턴스 레벨 주석을 사용하여 10,000개 이상의 다양한 이미지를 탐색합니다.

- **주행 영역 :**

  100,000개의 이미지로부터 복잡한 주행 결정방법을 학습합니다.

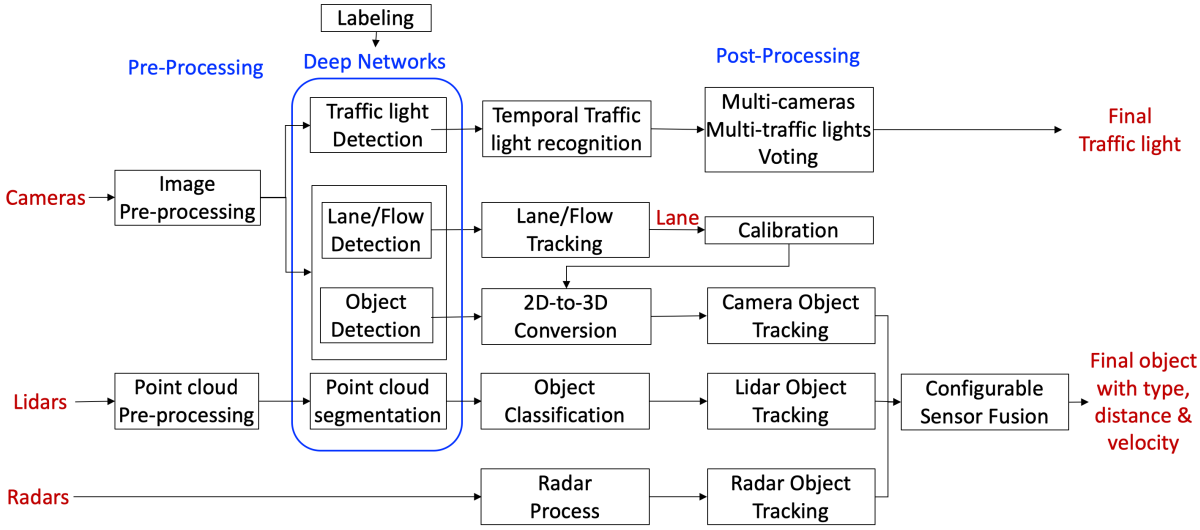
- **차선 표시 :**

  주행 안내를 위해 100,000개의 영상에 여러 유형의 차선 표시 주석을 달았습니다.

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

Apollo perception

인지 모듈을 살펴보면 아래와 같습니다.



일반적으로 카메라, 라이다, 레이더 세개의 센서를 활용합니다.

가공하기 이전에 전처리 과정을 진행을 합니다. 후에 딥러닝 기법을 이용하여 교통신호, 차선, 장애물등의 감지를 하고 라벨링 작업을 실시하게 됩니다.

여러 가공과정들을 거친 후 최종 신호등 인식, 차선 추적, 장애물을 결정합니다.

- CIPO(경로 내 가장 가까운 개체) 감지 가까운 개체) 감지

  CIPO에는 세로 방향 제어를 위한 도로의 주요 물체 감지가 포함됩니다.

  차량의 자기 운동 예측을 사용하여 가상의 차선을 생성하고 속도 및 각속도 변환을 사용한 차량 모델에 기초하여 차선의 길이와 곡률을 결정하게 됩니다.

- 소멸 지점 감지

  소멸 지점을 감지하기 위해 차선 인코더 끝에 추가적인 네트워크 분기가 부착됩니다 .

  이 분기는 합성곱 층과 완전히 연결된 층으로 구성되며, 여기서 합성곱 층은 소멸 지점 작업을 위한 차선 특징을 변환하고 완전히 연결된 층은 전체 이미지를 전역 요약하여 소멸 지점의 위치를 출력합니다.

  출력은 x, y 좌표에서 직접 제공하는 대신, 소멸점의 출력은 x, y 좌표에서 영상 중심까지의 거리를 나타내는 dx, dy의 형태가 됩니다.

YOLOv3 : Darknet

YOLO 는 You Only Look Once의 약자로, 이미지를 한 번 보는 것으로 물체의 종류와 위치를 추측할 수 있습니다.

단일 네트워크를 통해 여러개의 bounding box에 대한 객체 확률을 계산합니다.

물체인식을 수행하기 위해 고안된 심층 신경망이며 bounding box 조정과 분류를 신경망 구조를 통해 동시에 실행하여 구현하는 것이 특징입니다.

기타 환경 (옵션으로 없어도 무방) 으로는 OpenCV / GPU, Cudnn, CUDA 입니다.

`convert.py` : 변환명령 수행 파일입니다.

`yolov3.cfg` : Darknet에서 사용하는 모델 구조 정의 파일입니다.

`yolo3.weight` : Darknet으로 학습된 모델 파일입니다.

makefile을 확인해보겠습니다.

```
GPU=0
CUDNN=0
OPENCV=0
OPENMP=0
DEBUG=0

ARCH= -gencode arch=compute_30,code=sm_30 \
      -gencode arch=compute_35,code=sm_35 \
      -gencode arch=compute_50,code=[sm_50,compute_50] \
      -gencode arch=compute_52,code=[sm_52,compute_52]
#      -gencode arch=compute_20,code=[sm_20,sm_21] \ This one is deprecated?

# This is what I use, uncomment if you know your arch and want to specify
# ARCH= -gencode arch=compute_52,code=compute_52

VPATH=./src/:./examples
SLIB=libdarknet.so
ALIB=libdarknet.a
EXEC=darknet
OBJDIR=./obj/

CC=gcc
CPP=g++
NVCC=nvcc
AR=ar
ARFLAGS=rcs
OPTS=-Ofast
LDFLAGS= -lm -pthread
COMMON= -Iinclude/ -Isrc/
CFLAGS=-Wall -Wno-unused-result -Wno-unknown-pragmas -Wfatal-errors -fPIC

```

경로 설정이 `VPATH = ./src/:/examples` 로 되어있는 것을 확인할 수 있습니다.

메인 소스 파일이 구동되는 곳은 `./example/darknet.c`라는 것을 알 수 있습니다.

```
int main(int argc, char **argv)
{
    //test_resize("data/bad.jpg");
    //test_box();
    //test_convolutional_layer();
    if(argc < 2){
        fprintf(stderr, "usage: %s <function>\n", argv[0]);
        return 0;
    }
    gpu_index = find_int_arg(argc, argv, "-i", 0);
    if(find_arg(argc, argv, "-nogpu")) {
        gpu_index = -1;
    }
```

```
    if (0 == strcmp(argv[1], "average")){
        average(argc, argv);
    } else if (0 == strcmp(argv[1], "yolo")){
        run_yolo(argc, argv);
    } else if (0 == strcmp(argv[1], "super")){
        run_super(argc, argv);
    } else if (0 == strcmp(argv[1], "lsd")){
        run_lsd(argc, argv);
    } else if (0 == strcmp(argv[1], "detector")){
        run_detector(argc, argv);
    } else if (0 == strcmp(argv[1], "detect")){
        float thresh = find_float_arg(argc, argv, "-thresh", .5);
        char *filename = (argc > 4) ? argv[4]: 0;
        char *outfile = find_char_arg(argc, argv, "-out", 0);
        int fullscreen = find_arg(argc, argv, "-fullscreen");
        test_detector("cfg/coco.data", argv[2], argv[3], filename, thresh, .5, outfile, fullscreen);
```

main문의 일부분입니다.

```
else if (0 == strcmp(argv[1], "detect")){
        float thresh = find_float_arg(argc, argv, "-thresh", .5);
        char *filename = (argc > 4) ? argv[4]: 0;
        char *outfile = find_char_arg(argc, argv, "-out", 0);
        int fullscreen = find_arg(argc, argv, "-fullscreen");
        test_detector("cfg/coco.data", argv[2], argv[3], filename, thresh, .5, outfile, fullscreen);
```

이미지일 경우 인자 "detect"에 걸려 조건문이 실행되고 test_detector 함수를 통해 인식을 할 것을 알 수 있습니다.

LaneNet : Lane-Detection

실시간 차선 감지를 위한 심층 신경망을 구현하기위해 tensorflow를 사용합니다.

이 모델은 실시간 차선 감지 작업에 차별적 손실 함수를 사용하는 인코더-디코더 단계, 이진 의미 분할 단계 및 인스턴스 의미 분할로 구성됩니다.

코드 분석

`python tools/train_lanenet_tusimple.py`을 통해 수행이 시작됩니다.

```
#...

def train_model():

    if CFG.TRAIN.MULTI_GPU.ENABLE:
        LOG.info('Using multi gpu trainner ...')
        worker = multi_gpu_trainner.LaneNetTusimpleMultiTrainer(cfg=CFG)
    else:
        LOG.info('Using single gpu trainner ...')
        worker = single_gpu_trainner.LaneNetTusimpleTrainer(cfg=CFG)

    worker.train()
    return
```

위의 코드에서 학습 train은 `/trainner/tusimple_lanenet_single_gpu_trainner.py`에서 수행됩니다.

```
def train(self):
        """
        :return:
        """
        self._sess.run(tf.global_variables_initializer())
        self._sess.run(tf.local_variables_initializer())
        if self._cfg.TRAIN.RESTORE_FROM_SNAPSHOT.ENABLE:
            try:
                LOG.info('=> Restoring weights from: {:s} ... '.format(self._initial_weight))
                self._loader.restore(self._sess, self._initial_weight)
                global_step_value = self._sess.run(self._global_step)
                remain_epoch_nums = self._train_epoch_nums - math.floor(global_step_value / self._steps_per_epoch)
                epoch_start_pt = self._train_epoch_nums - remain_epoch_nums

        else:
            LOG.info('=> Starts to train LaneNet from scratch ...')
            epoch_start_pt = 1

        for epoch in range(epoch_start_pt, self._train_epoch_nums):
            train_epoch_losses = []
            train_epoch_mious = []
            traindataset_pbar = tqdm.tqdm(range(1, self._steps_per_epoch))

            for _ in traindataset_pbar:

                if self._enable_miou and epoch % self._record_miou_epoch == 0:
                    _, _, summary, train_step_loss, train_step_binary_loss, \
                        train_step_instance_loss, global_step_val = \
                        self._sess.run(
                            fetches=[
                                self._train_op, self._miou_update_op,
                                self._write_summary_op_with_miou,
                                self._loss, self._binary_seg_loss, self._disc_loss,
                                self._global_step
                            ]
                        )
                    train_step_miou = self._sess.run(
                        fetches=self._miou
                    )
                    train_epoch_losses.append(train_step_loss)
                    train_epoch_mious.append(train_step_miou)
                    self._summary_writer.add_summary(summary, global_step=global_step_val)
                    traindataset_pbar.set_description(
                        'train loss: {:.5f}, b_loss: {:.5f}, i_loss: {:.5f}, miou: {:.5f}'.format(
                            train_step_loss, train_step_binary_loss, train_step_instance_loss, train_step_miou
                        )
                    )
                else:
                    _, summary, train_step_loss, train_step_binary_loss, \
                        train_step_instance_loss, global_step_val = self._sess.run(
                            fetches=[
                                self._train_op, self._write_summary_op,
                                self._loss, self._binary_seg_loss, self._disc_loss,
                                self._global_step
                            ]
                        )
                    train_epoch_losses.append(train_step_loss)
                    self._summary_writer.add_summary(summary, global_step=global_step_val)
                    traindataset_pbar.set_description(
                        'train loss: {:.5f}, b_loss: {:.5f}, i_loss: {:.5f}'.format(
                            train_step_loss, train_step_binary_loss, train_step_instance_loss
                        )
                    )

            train_epoch_losses = np.mean(train_epoch_losses)
            if self._enable_miou and epoch % self._record_miou_epoch == 0:
                train_epoch_mious = np.mean(train_epoch_mious)
```

```
            if epoch % self._snapshot_epoch == 0:
                if self._enable_miou:
                    snapshot_model_name = 'tusimple_train_miou={:.4f}.ckpt'.format(train_epoch_mious)
                    snapshot_model_path = ops.join(self._model_save_dir, snapshot_model_name)
                    os.makedirs(self._model_save_dir, exist_ok=True)
                    self._saver.save(self._sess, snapshot_model_path, global_step=epoch)
                else:
                    snapshot_model_name = 'tusimple_train_loss={:.4f}.ckpt'.format(train_epoch_losses)
                    snapshot_model_path = ops.join(self._model_save_dir, snapshot_model_name)
                    os.makedirs(self._model_save_dir, exist_ok=True)
                    self._saver.save(self._sess, snapshot_model_path, global_step=epoch)

            log_time = time.strftime('%Y-%m-%d %H:%M:%S', time.localtime(time.time()))
            if self._enable_miou and epoch % self._record_miou_epoch == 0:
                LOG.info(
                    '=> Epoch: {:d} Time: {:s} Train loss: {:.5f} '
                    'Train miou: {:.5f} ...'.format(
                        epoch, log_time,
                        train_epoch_losses,
                        train_epoch_mious,
                    )
                )
            else:
                LOG.info(
                    '=> Epoch: {:d} Time: {:s} Train loss: {:.5f} ...'.format(
                        epoch, log_time,
                        train_epoch_losses,
                    )
                )
        LOG.info('Complete training process good luck!!')

        return
```

모델의 학습과정을 보겠습니다.

```
try:
    LOG.info('=> Restoring weights from: {:s} ... '.format(self._initial_weight))
    self._loader.restore(self._sess, self._initial_weight)
    global_step_value = self._sess.run(self._global_step)
    remain_epoch_nums = self._train_epoch_nums - math.floor(global_step_value / self._steps_per_epoch)
    epoch_start_pt = self._train_epoch_nums - remain_epoch_nums
```

초기 가중치(weight) 설정, 학습률 설정, 에포크 설정 등을 해줍니다.

```
 for epoch in range(epoch_start_pt, self._train_epoch_nums):
            train_epoch_losses = []
            train_epoch_mious = []
            traindataset_pbar = tqdm.tqdm(range(1, self._steps_per_epoch))

            for _ in traindataset_pbar:

                if self._enable_miou and epoch % self._record_miou_epoch == 0:
                    _, _, summary, train_step_loss, train_step_binary_loss, \
                        train_step_instance_loss, global_step_val = \
                        self._sess.run(
                            fetches=[
                                self._train_op, self._miou_update_op,
                                self._write_summary_op_with_miou,
                                self._loss, self._binary_seg_loss, self._disc_loss,
                                self._global_step
```

```
                        ]
                    )
                train_step_miou = self._sess.run(
                    fetches=self._miou
                )
                train_epoch_losses.append(train_step_loss)
                train_epoch_mious.append(train_step_miou)
                self._summary_writer.add_summary(summary, global_step=global_step_val)
                traindataset_pbar.set_description(
                    'train loss: {:.5f}, b_loss: {:.5f}, i_loss: {:.5f}, miou: {:.5f}'.format(
                        train_step_loss, train_step_binary_loss, train_step_instance_loss, train_step_miou
                    )
                )
```

에포크로 학습의 횟수를 정해주었고 학습데이터셋을 기반으로 예측한 값과 실제 값의 차이인 손실(loss)와 모델 성능 지표인(Mlou), 학습률을 계산해줍니다.

이 손실을 최소화하는 방향으로 값을 갱신해가며 최적화하게 됩니다.

딥러닝 알고리즘을 학습시키는 방법은 `/trainner/tusimple_lanenet_single_gpu_trainner.py` 의 경우 warm-up 방식의 학습 방법으로 (learning rate를 높이고 낮추고의 반복) 가중치(weight)를 갱신하고 있으며 `/trainner/tusimple_lanenet_multi_gpu_trainner.py`의 경우 average_gradient 경사하강법을 추가로 학습시켜 가중치를 갱신하는 것을 알 수 있습니다..

3) Darknet 실행

통합환경은 Colab을 통해 진행하였습니다.

```
!git clone https://github.com/AlexeyAB/darknet.git
```

저장소를 복제해옵니다.

```
!wget https://pjreddie.com/media/files/yolov3.weights
```

`yolov3.weights`라는 weights(가중치)를 다운받습는다. 이미 공개되어진 weights 파일입니다.

```
!./darknet detect cfg/yolov3.cfg yolov3.weights data/road.jpg
```

위의 코드는 단일 이미지 인식 명령어입니다.

변환된 모델이 잘 작동하는지 확인하기 위해 임의의 sample 이미지 (road.jpg)를 이용하겠습니다.

이미지의 경우 detect를 비디오의 경우 detector를 사용합니다.

```
import cv2
import matplotlib.pyplot as plt

image = cv2.imread('predictions.jpg')

plt.imshow(cv2.cvtColor(image, cv2.COLOR_BGR2RGB))
plt.show()
```

openCV 모듈을 import하였습니다.

`cv2.imread()` 함수를 이용하여 jpg 파일을 읽었습니다.

예측되어진 predictions.jpg 파일입니다.

이 예측 이미지를 bgr에서 rgv 이미지로 변환하여 보여줍니다.

적절하게 bounding box가 조정되었고 물체의 종류를 추측해내었습니다.

대부분의 차량들을 거의 100% 확률로 파악해낸 것을 보았습니다. 간혹 거의 가려진 버스의 경우 50%의 확률로 트럭으로 오인, 하나의 차량을 두개의 bounding box로 조정되기도 하는 것을 보았습니다. 좀 더 많은 훈련을 통해 정확한 가중치값을 갱신해야 할 필요가 있어보입니다.