CSC420
Final Project

Xiao Xuan Wang (wangx576) **1003422124**

Benli Wang(wangbenl) **1004156637**

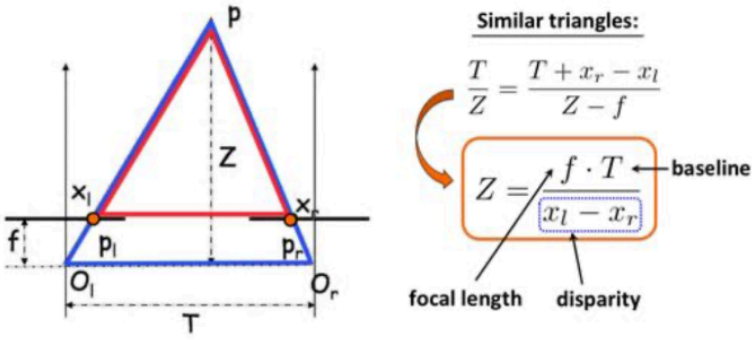
Road and object detection

Question 1

Compute disparity between the two stereo images.

We used two method for this question. Patch matching and SGBM.

For patch matching. Since we have two parallel cameras (call result image for left camera to be left image, result for right camera to be right image), thus in order to find the disparity of a patch on left image, we can check the corresponding rows of patch and find a best match in that row. After finding a best match, we can calculate the disparity and depth by compute



And the code I wrote to calculate the disparity is

```
def detect_match(img, template):
    """
    with help of documentation of matchTemplate in opencv
    """
    res = cv.matchTemplate(img,template, cv.TM_CCORR_NORMED)
    min_val, max_val, min_loc, max_loc = cv.minMaxLoc(res)
    top_left = max_loc[0] + (template.shape[1])/2
    return top_left
```

```
def compute_disparity(left_img,  right_img, patch_size = 5):
    radius = (patch_size-1)/2
    result = np.zeros(left_img.shape[:2])
    row,col,color = left_img.shape
    for cr in range(row):
        for cc in range(col):
            lr = int(max(0,cr-radius))
            rr = int(min(row,cr+radius+1))
            uc = int(max(0,cc-radius))
            dc = int(min(col,cc+radius+1))
            temp = left_img[lr:rr, uc:dc,:]
            xr = detect_match(right_img[int(max(0,lr-radius)):int(min(row,rr+radius)), :, :], temp)
            result[cr,cc] = cc-xr
    print(result.shape)
    return result
```

```
lpath = 'sync/image02/data/'
rpath = 'sync/image03/data/'

for i in range(107):
    imgnl = 'sync/image_02/data/{}.png'.format(str(i).zfill(10))
    imgnr = 'sync/image_03/data/{}.png'.format(str(i).zfill(10))
    left_img = cv.imread(imgnl)
    right_img = cv.imread(imgnr)
    disparity = compute_disparity(left_img, right_img, patch_size = 5)
    #disparity = gaussian_filter(disparity, sigma=1)
    disparity = (disparity // 10)*10
    plt.imsave(str(i).zfill(6)+".png", disparity, cmap='gray')
```

But however, the result is very noisy and inaccurate.

When left image is

I got result like

The second method we used is SGBM (Stereo Processing by Semiglobal Matching and Mutual Information) algorithm, the basic idea of this algorithm is making a disparity map by select the disparity value for each pixel. Then Set a energy function as following. Then try to minimize the E(D).

3

$$E(D) = \sum_p \left(C\left(p, D_p\right) + \sum_{q \in N_p} P_1 I\left[\left|D_p - D_q\right| = 1\right] + \sum_{q \in N_p} P_2 I\left[\left|D_p - D_q\right| > 1\right] \right)$$

p and q are the pixels in the image, Np stands for 8 pixels around p, C(p, Dp) is the cost when the disparity value for p is Dp, P1 is the penalty coefficient for pixels around p with difference of disparity value less and equal to 1, P2 is the penalty coefficient for pixels around p with difference of disparity value higher than 1, I[] will return 1 if and only if the statement inside is True. To optimize the solution is a NP-complete problem. However, it can be approximately solved by separate to many one-dimension problems and use dynamic programing to solve them. Thus, there are 8 one dimensional problems to solve.

In our project, we used the opencv package to implement the SGBM and generate the more accurate disparity images by following codes. The out put will have a black area on the left side and we filled up with the most left non-zero value column.

```
# SGBM Parameters -----------------
window_size = 3
# wsize default 3; 5; 7 for SGBM reduced size image; 15 for SGBM full size image (1300px and above); 5 Works nicely

left_matcher = cv2.StereoSGBM_create(
    minDisparity=0,
    numDisparities=112,
    blockSize=5,
    P1=8 * 3 * window_size ** 2,
    P2=32 * 3 * window_size ** 2,
    disp12MaxDiff=1,
    uniquenessRatio=15,
    speckleWindowSize=0,
    speckleRange=2,
    preFilterCap=63,
    mode=cv2.STEREO_SGBM_MODE_SGBM_3WAY
)

right_matcher = cv2.ximgproc.createRightMatcher(left_matcher)

# FILTER Parameters
lmbda = 80000
sigma = 1.2

wls_filter = cv2.ximgproc.createDisparityWLSFilter(matcher_left=left_matcher)
wls_filter.setLambda(lmbda)
wls_filter.setSigmaColor(sigma)

print('computing disparity...', index)
displ = left_matcher.compute(imgL, imgR)  # .astype(np.float32)/16
dispr = right_matcher.compute(imgR, imgL)  # .astype(np.float32)/16
displ = np.int16(displ)
dispr = np.int16(dispr)
filteredImg = wls_filter.filter(displ, imgL, None, dispr)  # important to put "imgL" here!!!

filteredImg = cv2.normalize(src=filteredImg, dst=filteredImg, beta=0, alpha=255, norm_type=cv2.NORM_MINMAX);
filteredImg = np.uint8(filteredImg)
cv2.imshow('Disparity Map', filteredImg)
filteredImg[:,:112] = np.tile(filteredImg[:,112], (112,1)).T
filteredImg = filteredImg / 2
cv2.imwrite('./data_road/disparity/depth_{}.png'.format(str(index).zfill(6)), filteredImg)
```

Here are some examples we get

Original image:

Disparity map:

Original image:

Disparity map:

Question 2

Compute depth of each pixel. Compute 3D location of each pixel.

Using method mentioned in patch match method in previous question, I can get the depth of a pixel using disparity by $depth = f*T/disparity$ where f is focal length, T is baseline.

As mentioned in set up, T = 0.54, and in calib_cam_to_cam.txt, the focal length f is 959.791.

```
def compute_depth(disparity, f, T):
    disparity = disparity
    disparity = cv.GaussianBlur(disparity, (21, 21), 1)
    disparity = np.maximum(1, disparity)
    print(disparity)
    result = f * T / (disparity)
    return result
```

To calculate 3D location, we need the information from calibration matrix of camera also from calib_cam_to_cam.txt. px = 696.0217 and py = 224.1806.

```
def calculate_3d(img, depth_map):
    scale_y = py / (depth_map.shape[0] / 2)
    scale_x = px / (depth_map.shape[1] / 2)
    Z = depth_map
    X = Z * (img[:, :, 1] * scale_x - px) / f
    Y = Z * ((depth_map.shape[0] - img[:, :, 0]) * scale_y - py) / f
    return np.stack((X, Y, Z), axis=-1)
```

```
depth_list = []
three_d_list = []
r,c = cv.imread('regular_disparity/'+str(0).zfill(6)+".png", cv.IMREAD_GRAYSCALE).shape
img = np.zeros((r, c,2))
for i in range(r):
    for j in range(c):
        img[i][j] = [i,j]
for i in range(100):
    disp_name = 'regular_disparity/'+str(i).zfill(6)+".png"
    disparity = cv.imread(disp_name, cv.IMREAD_GRAYSCALE)
    depth = compute_depth(disparity, f,T)
    three_d = calculate_3d(img,depth)
    depth_list.append(depth)
    three_d_list.append(three_d)
```

Since from previous question, the result of disparity using SGBM is much better than patch matching, so in this question and later question we will use SGBM to generate disparity.

Question 3

Train a road classifier on a set of annotated images, and compute road pixels in your

image. Which features would you use? Try to use both 2D and 3D features.

We tried two different machine learning approaches to solve this problem, SVM (support vector machine) and neural network. But, for both methods, we feed with same features. Here are nine features we give:

Red, Green, Blue value, Hue, Saturation and Value for the image, using the disparity map to calculate the 3D location (X, Y, Z) of each pixel.

Using RGB value because we want to distinguish the road and other object by color, it is a intuitive idea since human recognize the object by color first. Then applying HSV value will also help us to differentiate object in the image. The feature of 3D location is also added because we know in the ideal case, the y value for road will always be consistent. Besides, for all the training image we get, the road is in the center of the image, thus, 3D location should be an efficacious feature.

Here are our codes and some examples.

SVM

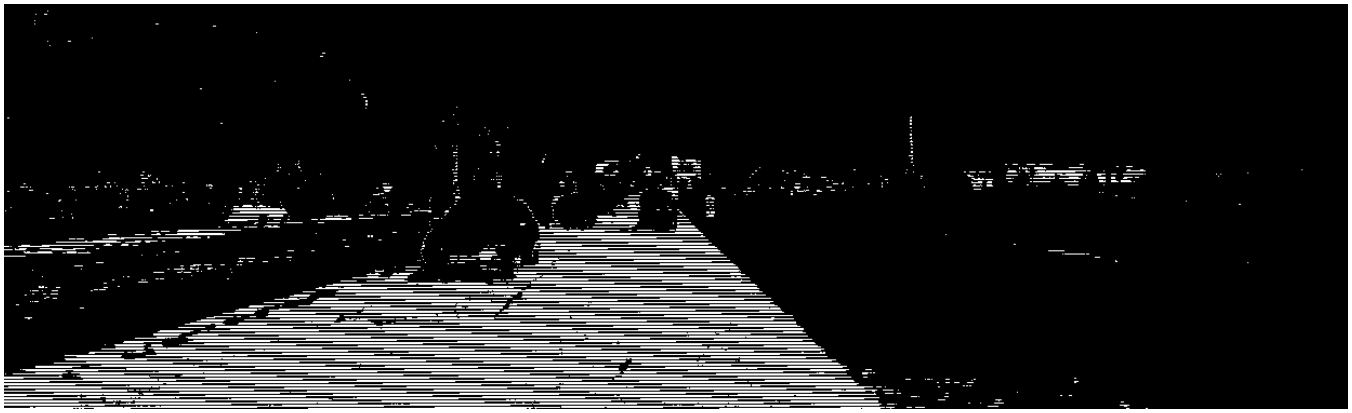
```
def train_detector():
    #     print(len(validate[0]), validate[0][0])
    #     train_image, validate_image = train[0], validate[0]
    img_lst = []
    val_lst = []
    depth_lst = []
    for i in range(80):
        im_1 = cv.imread("data_road/data_road_left/training/image_2/um_{}.png".format(str(i).zfill(6)))
        im_2 = cv.imread("data_road/data_road_left/training/gt_image_2/um_road_{}.png".format(str(i).zfill(6)))
        im_3 = cv.imread("data_road/disparity/depth_{}.png".format(str(i).zfill(6)))

        img_lst.append(cv.blur(im_1, (5,5)))
        val_lst.append(cv.blur(im_2, (5,5)))
        depth_lst.append(cv.blur(im_3, (5,5)))
    train_image = img_lst[0]

    validate_image = val_lst[0]
    depth = depth_lst[0]

    M, N, _ = train_image.shape
    m, n, _ = validate_image.shape
    locs = get_3d_location(depth)

    bayes = np.zeros((M,N,1))
    for val_img in val_lst:
        bayes = bayes + val_img[:,:,0].reshape(M,N,1) / 255

    hsv = cv.cvtColor(train_image, cv.COLOR_BGR2HSV)
    hsv = np.insert(hsv, 3, train_image[:, :, 0], axis=2)
    hsv = np.insert(hsv, 4, train_image[:, :, 1], axis=2)
    hsv = np.insert(hsv, 5, train_image[:, :, 2], axis=2)


    X = np.reshape(hsv, (M * N, 6))
    X = X / 255
```

```
X = np.reshape(hsv, (M * N, 6))
X = X / 255

loc_flat1 = locs[:,0].flatten()
loc_flat1.sort()
max_loc_1 = loc_flat1[int(0.9 * loc_flat1.shape[0])]
loc_flat2 = locs[:, 1].flatten()
loc_flat2.sort()
max_loc_2 = loc_flat2[int(0.9 * loc_flat2.shape[0])]
loc_flat3 = locs[:, 2].flatten()
loc_flat3.sort()
max_loc_3 = loc_flat3[int(0.9 * loc_flat3.shape[0])]

X = np.insert(X, 6, locs[:, 0] / max_loc_1, axis=1)
X = np.insert(X, 7, locs[:, 1] / max_loc_2, axis=1)
X = np.insert(X, 8, locs[:, 2] / max_loc_3, axis=1)

y = np.reshape(validate_image, (M * N, 3))
y = y[:, 0]
y[y > 125] = 255
y[y <= 125] = 0
```

```
for i in range(1, 80):
    train_image, validate_image, depth = img_lst[i], val_lst[i], depth_lst[i]
    M, N, _ = train_image.shape
    m, n, _ = validate_image.shape
    hsv = cv.cvtColor(train_image, cv.COLOR_BGR2HSV)
    hsv = np.insert(hsv, 3, train_image[:, :, 0], axis=2)
    hsv = np.insert(hsv, 4, train_image[:, :, 1], axis=2)
    hsv = np.insert(hsv, 5, train_image[:, :, 2], axis=2)
    new_x = np.reshape(hsv, (M * N, 6))
    new_x = new_x /255

    locs = get_3d_location(depth)

    # normalize the 3d location
    loc_flat1 = locs[:, 0].flatten()
    loc_flat1.sort()
    max_loc_1 = loc_flat1[int(0.9 * loc_flat1.shape[0])]
    loc_flat2 = locs[:, 1].flatten()
    loc_flat2.sort()
    max_loc_2 = loc_flat2[int(0.9 * loc_flat2.shape[0])]
    loc_flat3 = locs[:, 2].flatten()
    loc_flat3.sort()
    max_loc_3 = loc_flat3[int(0.9 * loc_flat3.shape[0])]

    new_x = np.insert(new_x, 6, locs[:, 0] / max_loc_1, axis=1)
    new_x = np.insert(new_x, 7, locs[:, 1] / max_loc_2, axis=1)
    new_x = np.insert(new_x, 8, locs[:, 2] / max_loc_3, axis=1)
    new_y = np.reshape(validate_image, (m * n, 3))
    new_y = new_y[:, 0]
    new_y[new_y > 125] = 255
    new_y[new_y <= 125] = 0
    X = np.vstack((X, new_x))
    y = np.concatenate((y, new_y))
    print(i)
return X, y
```

```
if __name__ == "__main__":
    X, y = train_detector()
    clf = LinearSVC(random_state=0, tol=1e-05)
    print('start')
    clf.fit(X, y)
    pkl_filename = "road_model.pkl"
    with open(pkl_filename, 'wb') as file:
        pickle.dump(clf, file)
```

Example



The output can recognize the road location but is not perfect. Thus, I applied some filter on the output image to get a more accurate result.

```
for i in range(3):
    image = cv.blur(image, (3,3))
    image[image[:,:,0] > 160] = [255,255,255]
    image[image[:,:,0] < 40] = [0,0,0]
    kernel = cv.getStructuringElement(cv.MORPH_ELLIPSE, (5, 5))
    image = cv.morphologyEx(image, cv.MORPH_OPEN, kernel)

image[image[:, :, 0] > 0] = [255, 255, 255]
return image
```

Then the new output is

Moreover, the pros by choosing SVM is it can solve high dimension problems well and get the relation of the data and features. However, the training takes longer time and the worst case will take too long if the data set is large.

Neural Network

```
if __name__ == "__main__":
    BATCH_SIZE = 10
    SHUFFLE_BUFFER_SIZE = 100
    print("Num GPUs Available: ", len(tf.config.experimental.list_physical_devices('GPU')))

    with tf.device('/gpu:1'):

        x_train, y_train = train_detector()
        test_x, test_y = get_test()
        train_dataset = tf.data.Dataset.from_tensor_slices((x_train, y_train))
        test_dataset = tf.data.Dataset.from_tensor_slices((test_x, test_y))
        # Building a 3-layer DNN with 50 units each.
        model = tf.keras.Sequential([
            tf.keras.layers.Flatten(),
            tf.keras.layers.conv2d(),
            tf.keras.layers.Dense(1024, activation='relu'),
            tf.keras.layers.Dense(108, activation='relu'),
            tf.keras.layers.Dense(2, activation='softmax')
        ])

        train_dataset = train_dataset.shuffle(SHUFFLE_BUFFER_SIZE).batch(BATCH_SIZE)
        test_dataset = test_dataset.batch(BATCH_SIZE)
        # Use the train data to train this classifier

        model.compile(optimizer=tf.keras.optimizers.RMSprop(),
                              loss=tf.keras.losses.SparseCategoricalCrossentropy(),
                              metrics=[tf.keras.metrics.SparseCategoricalAccuracy()])
        model.fit(train_dataset, epochs=2, verbose=1)
        print('testing')
        model.evaluate(test_dataset)
        model.save('saved_model', save_format='tf')
```

The labels on the road and the lines are not recognize as road might because when we trained model, the most road don't have the white color and make the model to mis predict the road

Question 4

Fit a plane in 3D to the road pixels by using the depth of the pixels. Make sure your algorithm is robust to outliers

By using the pixel labeled as rode and their 3D location calculated in question 2, I can perform RANSAC to find a optimal plane. Since we know three 3d point can find a unique plane, so I can randomly pick three point in the road, and fit a plane using three point, calculate the difference between other road pixel and the plane, count the outliers(pixel that has a difference greater than threshold), and find the optimal plane that has the smallest outliers.

```
def calculate_error(three_d, road, a, b, c, d, threshold=0.5):
    # get 3d point for road pixel
    three_d_road = three_d[road[:,0], road[:,1]]
    predict = (d - a*three_d_road[:,0]-b*three_d_road[:,1])/c
    diff = np.abs(predict - three_d_road[:,2])
    return (diff>1).sum()
```

```
label_name = 'data_road/training/gt_image_2/um_lane_000000.png'
label = cv.imread(label_name)
label = np.sum(label,axis = 2)/255
road = np.asarray(np.where(label == 2))
road = road.T
three_d = three_d_list[0]
min_outlier = 1000000
optimal = None
for i in range(1000):
    pick3 = np.random.choice(len(road), 3)
    p1 = road[pick3[0]]
    p2 = road[pick3[1]]
    p3 = road[pick3[2]]
    p1 = three_d[p1[0],p1[1]]
    p2 = three_d[p2[0],p2[1]]
    p3 = three_d[p3[0],p3[1]]
    v1 = p3 - p1
    v2 = p2 - p1
    cp = np.cross(v1, v2)
    a, b, c = cp
    v2 = p2 - p1
    d = np.dot(cp, p3)
    outlier = calculate_error(three_d, road, a, b, c, d)
    if outlier < min_outlier:
        min_outlier = outlier
        optimal = [a,b,c,d]
print(p1,p2,p3,min_outlier,optimal)
```

For this iteration, result is 0.2534022227414456x-33.70033785527126y+ 0.7452026089080129z= 42.45503035930558

Set x,z = 0, we have y =-1.26, close to the true value since the camera is 1.65m high.

However since ransac is randomly choosing point, so there is a chance that it fits the point with inaccurate 3d location or that is false positively classified as a road. So we choose the one with less outliers(the vertical distance between the point and the plane is larger than 0.5) and apply a bigger iteration times.

Another way to do this is using MSE but since this is not robust to the points that has been classify as road but it is not road, and assume the calculation for 3d location is correct, then because of those point, the result plane will be higher than what we expected. So I choose to use RANSAC.

Question 5

Plot each pixel in 3D (we call this a 3D point cloud). On the same plot, show also the estimated ground plane.

We use open3d to plot. With help of 3d location calculated by previous question, and the pixel color is it's own image, normalized by dividing 255.

For showing the plane, I generate 100000 value for X and Z(X is -150 to 150, Z is 0-300), and use the plane to fit Y. plot XYZ using red color on the same graph with the image.

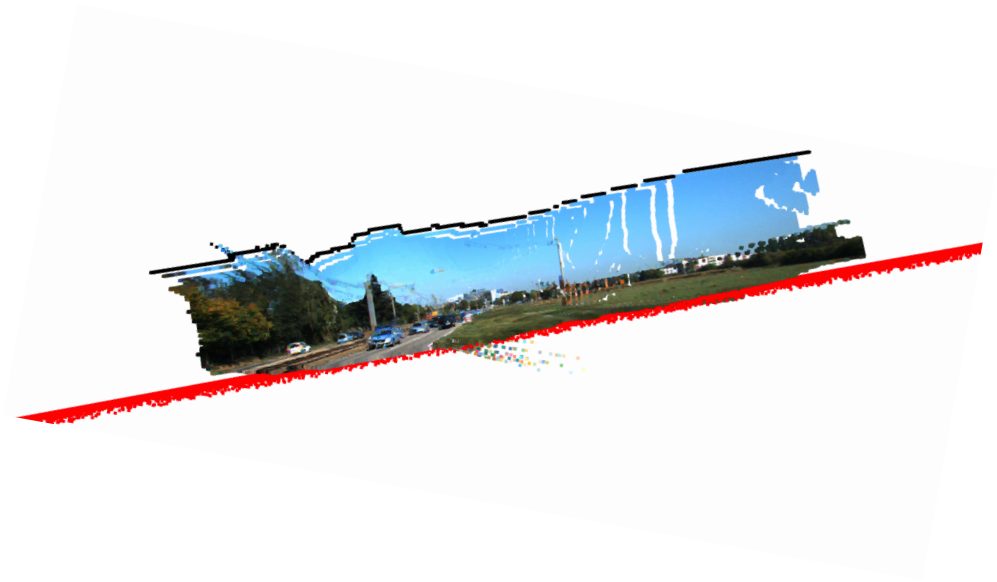
Code:

```
x = ((np.random.rand(100000))-0.5)*300
z = (np.random.rand(100000))*300
y = (optimal[3] - x*optimal[0] - z*optimal[2])/optimal[1]

pxyz = np.stack((x, y, z), axis=-1)
pxyzc1 = np.zeros((100000))
pxyzc2 = np.zeros((100000))
pxyzc3 = np.ones((100000))
pxyzc= np.stack((pxyzc1, pxyzc2, pxyzc3), axis=-1)

xyz = three_d.reshape((-1, 3))

xyz = np.vstack((xyz, pxyz))

pcd = o3d.geometry.PointCloud()
pcd.points = o3d.utility.Vector3dVector(xyz)
imgnl = './dis/road_038/um_000038.png'
left_img = cv.imread(imgnl).reshape((-1, 3))
C = left_img / 255
C = np.vstack((C, pxyzc))
C = C[:, ::-1]
pcd.colors = o3d.utility.Vector3dVector(C)
o3d.visualization.draw_geometries([pcd])
```

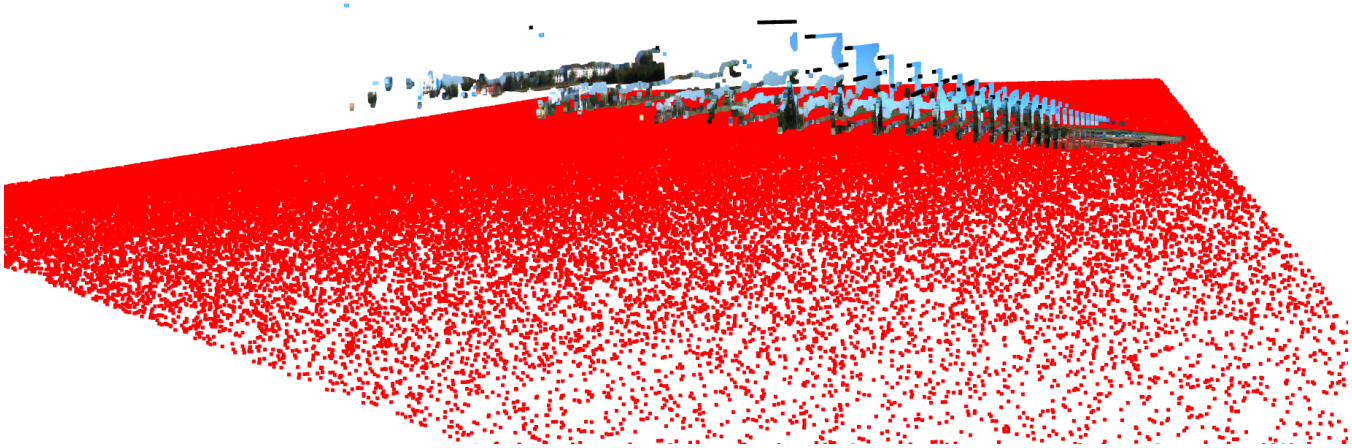
Result:

Input:

Capture from front:

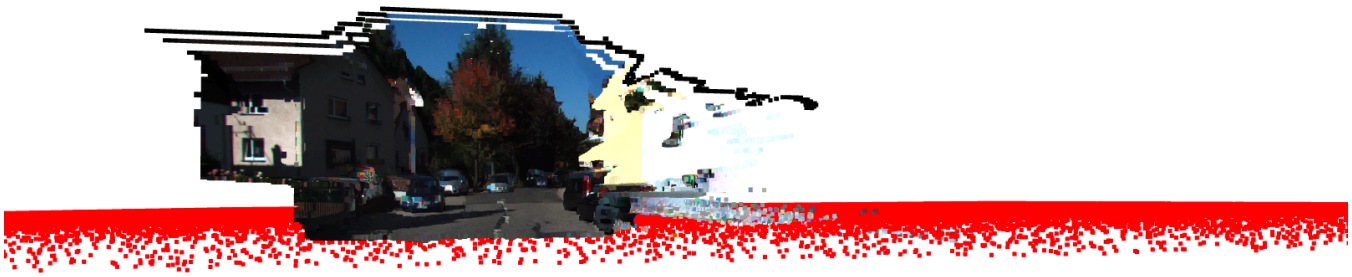


Capture from side:



input:

Result:



Question 6

Detect cars in the image

To detect car in the image. I used YOLO v3. Which give the image as input then apply the model, then return the detected object's coordinate.

In my implementation, I will reshape all my input image to be size 416*416*3, Then apply multiple convolution.

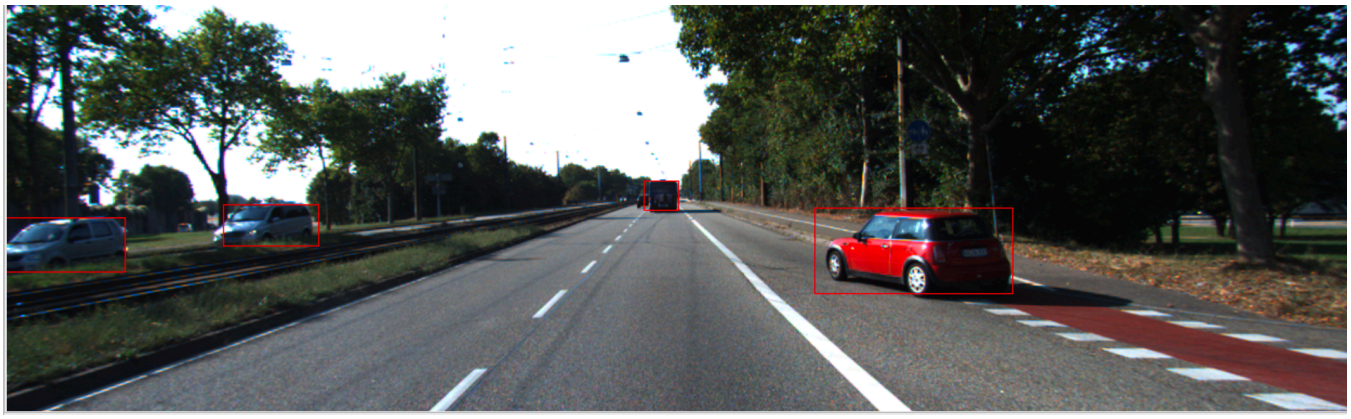
Using the last layer's information, We can get the position of bounding box using logistic regression with non_max_suppression. Then I write the x,y,w,h in to a mat file for later reference and then draw a box around bounding box.

```
import cv2 as cv
import scipy.io as sio
import numpy as np

# Initialize the parameters
confThreshold = 0.5  #Confidence threshold
nmsThreshold = 0.4   #Non-maximum suppression threshold
inpWidth = 416       #Width of network's input image
inpHeight = 416      #Height of network's input image


# Load names of classes
classesFile = "coco.names"
classes = None
with open(classesFile, 'rt') as f:
    classes = f.read().rstrip('\n').split('\n')

# Give the configuration and weight files for the model and load the network using them.
modelConfiguration = "yolov3.cfg"
modelWeights = "yolov3.weights"
frame_name = 'um_000087.png'
frame = cv.imread(frame_name)
```

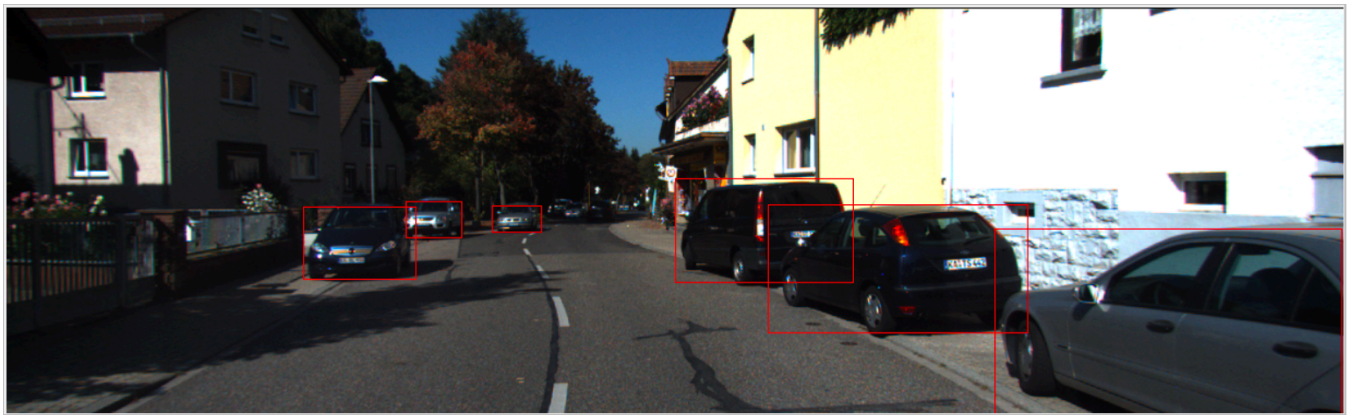
```python
def postprocess(frame, outs):
    frameHeight = frame.shape[0]
    frameWidth = frame.shape[1]

    # Scan through all the bounding boxes output from the network and keep only the
    # ones with high confidence scores. Assign the box's class label as the class with
    # the highest score.
    classIds = []
    confidences = []
    boxes = []
    for out in outs:
        for detection in out:
            scores = detection[5:]
            classId = np.argmax(scores)
            confidence = scores[classId]
            if confidence > confThreshold:
                center_x = int(detection[0] * frameWidth)
                center_y = int(detection[1] * frameHeight)
                width = int(detection[2] * frameWidth)
                height = int(detection[3] * frameHeight)
                left = int(center_x - width / 2)
                top = int(center_y - height / 2)
                classIds.append(classId)
                confidences.append(float(confidence))
                boxes.append([left, top, width, height])

    # Perform non maximum suppression to eliminate redundant overlapping boxes with
    # lower confidences.
    indices = cv.dnn.NMSBoxes(boxes, confidences, confThreshold, nmsThreshold)
    result_box = []
    for i in indices:
        i = i[0]
        box = boxes[i]
        left = box[0]
        top = box[1]
        width = box[2]
        height = box[3]
        cv.rectangle(frame, (left, top), (left + width, top + height), (0, 0, 255))
        result_box.append(box)
    cv.imwrite("bbbbb.png",frame)
    return result box
```

```
net = cv.dnn.readNetFromDarknet(modelConfiguration, modelWeights)
net.setPreferableBackend(cv.dnn.DNN_BACKEND_OPENCV)
net.setPreferableTarget(cv.dnn.DNN_TARGET_CPU)

blob = cv.dnn.blobFromImage(frame, 1 / 255, (inpWidth, inpHeight), [0, 0, 0], 1,
                            crop=False)
net.setInput(blob)

# Runs the forward pass to get output of the output layers
layersNames = net.getLayerNames()
out_name = [layersNames[i[0] - 1] for i in net.getUnconnectedOutLayers()]
outs = net.forward(out_name)

# Remove the bounding boxes with low confidence
output = postprocess(frame, outs)
file_name = 'r{}.mat'.format(frame_name[:-4].zfill(6))
sio.savemat(file_name, {'ds': []})
for (left, top, width, height) in output:
    right = left + width
    bottom = top + height
    cur = sio.loadmat(file_name)['ds']
    if len(cur) == 0:
        sio.savemat(file_name,
                    {'ds': [left, right, top, bottom]})
    else:
        cur = np.append(cur, [[left, right, top, bottom]], axis=0)
        sio.savemat(file_name, {'ds': cur})
```

Result image(good)



Bad result (only few car detected, but smaller car near vanishing point is not detected)



However with YOLO, the accuracy rate is not as good as model like faster RCNN, especially with smaller item, but the speed is very fast. But since we solving problem for automatic driving, smaller car(which in reality is usually far car) is not very important, but the speed for car detection is very important, so I chose to use YOLO.

Question 7

Train a classifier that predicts viewpoint for each car. The viewpoint labels are in 30◦

increments, thus train a 12-class classifier.

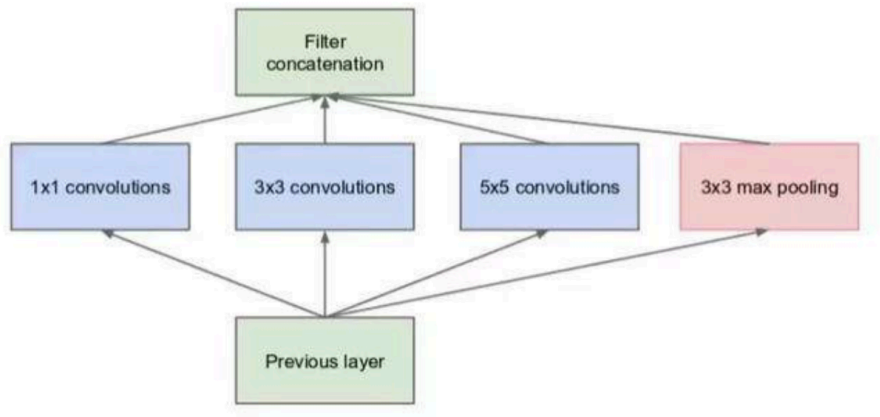
In this question, to train the classifier, I first used the bounding box data from training set to cut the car segment out and put them in their corresponding folder.

```
7  import numpy as np
8  import os
9  import pandas
10 from os import listdir
11 from os.path import isfile, join
12 from scipy import misc
```

```
if __name__ == "__main__":
    # PATHS
    DETECTIONS_PATH = './training/label_2'
    OUTPUT_PATH = './result'
    IMAGES_PATH = './data_object/training/image_2'

    # Get filenames
    detection_files = [DETECTIONS_PATH + os.sep + f for f in
                       listdir(DETECTIONS_PATH) if isfile(join(DETECTIONS_PATH, f))]

    image_files = [IMAGES_PATH + os.sep + f for f in listdir(IMAGES_PATH) if
                   isfile(join(IMAGES_PATH, f))]

    image_files.sort()
    detection_files.sort()

    counter = 0
    bins = np.arange(-180, 181, 30)
    for i, f in zip(image_files, detection_files):
        img = misc.imread(i)
        dataframe = pandas.read_csv(f, header=0, sep=' ')
        dataset = dataframe.values
        cars = []
        for d in dataset:
            # If the object is a car.
            if d[0] == 'Car':
                # Segment it.
                angle = float(d[14])
                bin = np.digitize(np.degrees(angle), bins)
                segmented = img[int(float(d[5])):int(float(d[7])),
                                int(float(d[4])):int(float(d[6])), :]
                segmented = misc.imresize(segmented, (128, 128), interp='bilinear',
                                          mode=None)
                os.makedirs(OUTPUT_PATH + os.sep + str(bin) + os.sep, exist_ok=True)
                misc.imsave(OUTPUT_PATH + os.sep + str(bin) + os.sep + str(
                    counter) + '.png', segmented)
                counter += 1
```

Then using the separated car segments, I used keras's InceptionV3 to train the model.

InceptionV3 is still a model that using convolution. The basic idea for Inception is



And This is very computational costly. In order to decrease running time, we choose to add some 1*1 convolution layer to reduce dimension. Also separate 5*5 convolution matrix to two 3*3 convolution matrix to decrease running time, so intotal we have

type	patch size/stride or remarks	input size
conv	3×3/2	299×299×3
conv	3×3/1	149×149×32
conv padded	3×3/1	147×147×32
pool	3×3/2	147×147×64
conv	3×3/1	73×73×64
conv	3×3/2	71×71×80
conv	3×3/1	35×35×192
3×Inception	As in figure 5	35×35×288
5×Inception	As in figure 6	17×17×768
2×Inception	As in figure 7	8×8×1280
pool	8 × 8	8 × 8 × 2048
linear	logits	1 × 1 × 2048
softmax	classifier	1 × 1 × 1000

Table 1. The outline of the proposed network architecture. The output size of each module is the input size of the next one. We are using variations of reduction technique depicted Figure 10 to reduce the grid sizes between the Inception blocks whenever applicable. We have marked the convolution with 0-padding, which is used to maintain the grid size. 0-padding is also used inside those Inception modules that do not reduce the grid size. All other layers do not use padding. The various filter bank sizes are chosen to observe principle 4 from Section 2.

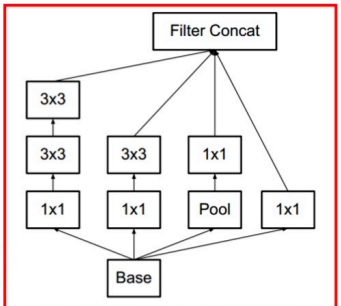


Figure 5. Inception modules where each 5×5 convolution is replaced by two 3×3 convolution, as suggested by principle 3 of Section 2.

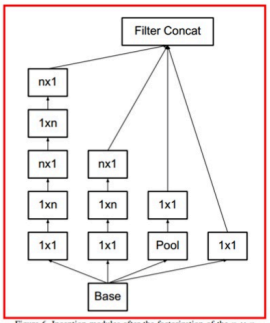


Figure 6. Inception modules after the factorization of the $n \times n$ convolutions. In our proposed architecture, we chose $n = 7$ for the 17×17 grid. (The filter sizes are picked using principle 3)

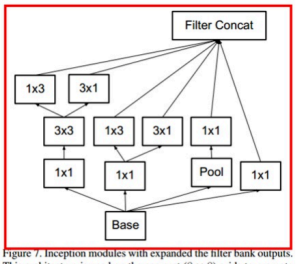


Figure 7. Inception modules with expanded the filter bank outputs. This architecture is used on the coarsest (8×8) grids to promote high dimensional representations, as suggested by principle 2 of Section 2. We are using this solution only on the coarsest grid, since that is the place where producing high dimensional sparse representation is the most critical as the ratio of local processing (by 1×1 convolutions) is increased compared to the spatial aggregation.

And also in InceptionV3, it factorized 7*7 convolution layer and add BatchNorm to surport classification.

In my implementation, I put the entire segment in and resize it to 128*128*3 and divide by 255 to normalize it to be with in 0-1.

```
train_datagen = ImageDataGenerator(
    rescale=1. / 255,
    shear_range=0.2,
    zoom_range=0.2,
    horizontal_flip=False)

test_datagen = ImageDataGenerator(rescale=1. / 255)


train_generator = train_datagen.flow_from_directory(
    './result',  # this is the target directory
    target_size=(128, 128),  # all images will be resized to 128x128
    batch_size=32,
    class_mode='categorical')

input_tensor = Input(shape=(128, 128, 3))
```

Then add a HOG feature to the data set. Since the change direction for a patch is very important for viewpoint classification.

```
data_list = []
batch_index = 0
y =[]

while batch_index <= train_generator.batch_index:
    data = train_generator.next()
    data_list.extend(data[0])
    y.extend(data[1])
    batch_index = batch_index + 1

data_array = np.asarray(data_list)
y_array = np.asarray(y)
his = np.empty((0,128,128,1))

for i in range(data_array.shape[0]):
    fd, hog_image = hog(data_array[i,:,:,:], orientations=8, pixels_per_cell=(16, 16),
                        cells_per_block=(1, 1), visualize=True,
                        multichannel=True)
    hog_image=np.expand_dims(hog_image, axis=2)
    hog_image=np.expand_dims(hog_image, axis=0)
    his = np.vstack((his,hog_image))


data = np.append(data_array, his, 3)
```

```
# We use the InceptionV3 /GoogLeNet model but retrain it to classify out datset.
base_model = InceptionV3(input_tensor=input_tensor, weights='imagenet', include_top=False)
# Add a global spatial average pooling layer.
x = base_model.output
x = GlobalAveragePooling2D()(x)
# Add a fully-connected layer.
x = Dense(1024, activation='relu')(x)
# Add a HEAVY dropout.
x = Dropout(0.7)(x)
predictions = Dense(12, activation='softmax')(x)

model = Model(input=base_model.input, output=predictions)
```

The base mode doesn’t have top layer when build it, but we add a Rectified Linear Unit on top. Also I add a average pooling layer before that and I use Dropout to avoid overfitting.

```
for layer in base_model.layers:
    layer.trainable = False

# compile the model (should be done *after* setting layers to non-trainable)
model.compile(optimizer='rmsprop', loss='categorical_crossentropy', metrics=['accuracy'])

model.fit(data, y_array, epochs=10)
```

First we froze all layers for base model to capture correct bottleneck,

```
for layer in model.layers[:172]:
    layer.trainable = False
for layer in model.layers[172:]:
    layer.trainable = True

model.compile(optimizer=SGD(lr=0.01, decay=1e-6, momentum=0.9))
model.fit(data, y_array, epochs=15)

model.save('object_classifierfit2.h5')  # Save the final model.

```

Then we turn on trainable for some inception blocks, refit it and save the model.

The accuracy rate is around 0.88 after training.

Then we can use the trained classification model to classify viewpoint.

```
import cv2
import os
import scipy.io as sio
from keras.models import load_model
import numpy as np
from scipy.misc import imsave, imread
import csv

# Set paths

# Detections in .mat format from DPM / matlab
DETECTIONS_DIR = './mat_result'
# Image Files for the detections - must have the same basename
IMAGES_DIR = './data_object/testing/image_2'
# Path to output everything to
OUTPUT_DIR = './angle_re'

```

```

# Get filepaths for images
image_files = [os.path.join(IMAGES_DIR, f) for f in os.listdir(IMAGES_DIR) if f.endswith('.png')]

# Get filepaths for detections
detection_files = [os.path.join(DETECTIONS_DIR, f) for f in
                   os.listdir(DETECTIONS_DIR) if
                   f.endswith('.mat')]

# Sort them so that they're in the same order
image_files.sort()
detection_files.sort()
```

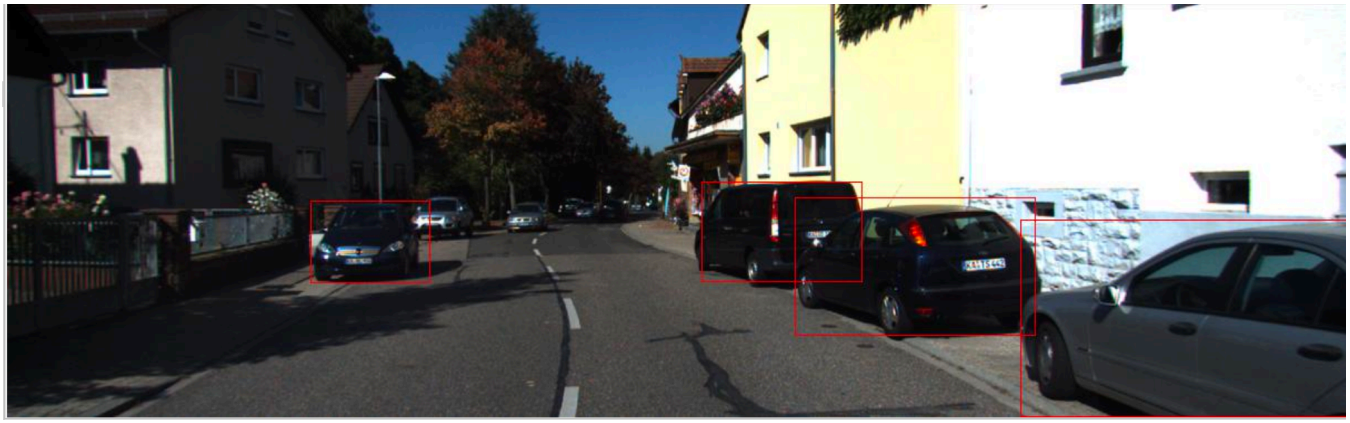
```
model = load_model('object_classifierfit2.h5')
# Iterate through images and dpm detections
for i, d in zip(image_files, detection_files):
    angles = []
    # Filter detections
    filtered = []
    for m in sio.loadmat(d)['ds']:
        filtered.append(m)
    # Load image
    image = imread(i)
    # ITerate through all detected objects
    for d in filtered:
        segment = image[int(d[2]):int(d[3]), int(d[0]):int(d[1]),:]
        if (segment.shape[0] == 0 or segment.shape[1] == 0):
            continue
        segment = cv2.resize(segment, (128, 128))
        segment = np.array([segment])
        segment = segment / 255.
        # Generate a prediction
        preditction = model.predict(segment)
        angle = (np.argmax(preditction, axis=1)) * 30 - 180
        x = d.tolist()
        x.append(angle[0])
        angles.append(x)
    # Output to a csv
    with open(os.path.basename(i).split('.')[0] + '.csv', 'w') as f:
        w = csv.writer(f)
        w.writerows(angles)
```

For input we need the image's name and .mat file that contains the detection to be in the same sorted order. In .mat file we have the order per each row to be left, right, up, down of bounding box, same as the output in csv. And the last element is the angle.

Result

Input image and the corresponding detect element is

The returned csv is 754,927,189,268,-30 Which is accurate.



The result csv file is

```
709,925,174,297,-30
911,1220,193,371,0
271,382,175,252,150
621,773,156,253,-30
```

We can see that the right most car has should have a negative viewpoint angle but since the image didn't capture the entire car, it is classified as 0. So this is a limitation for this method. It depend on the detection box. When the car is not fully showed, there might be classification error.

The reason why I choose InceptionV3 with GAP(global average pooling) is because First CNN is good for segment classification and InceptionV3 is very good model under CNN. It has increased accuracy and speed than regular CNN. The reason why I choose GAP but not GMP(global maximum pooling) and FC(fully connected) is because some data shows that FS's accuracy decreased after several epochs, might caused by over fitting, and GMP has a lower accuracy than GAP.

(code reference : https://github.com/Colliflower/Computer-Vision-Project)

Question 8

Show a test image with the detected car bounding boxes and show the estimated viewpoints by plotting an arrow in the appropriate direction.

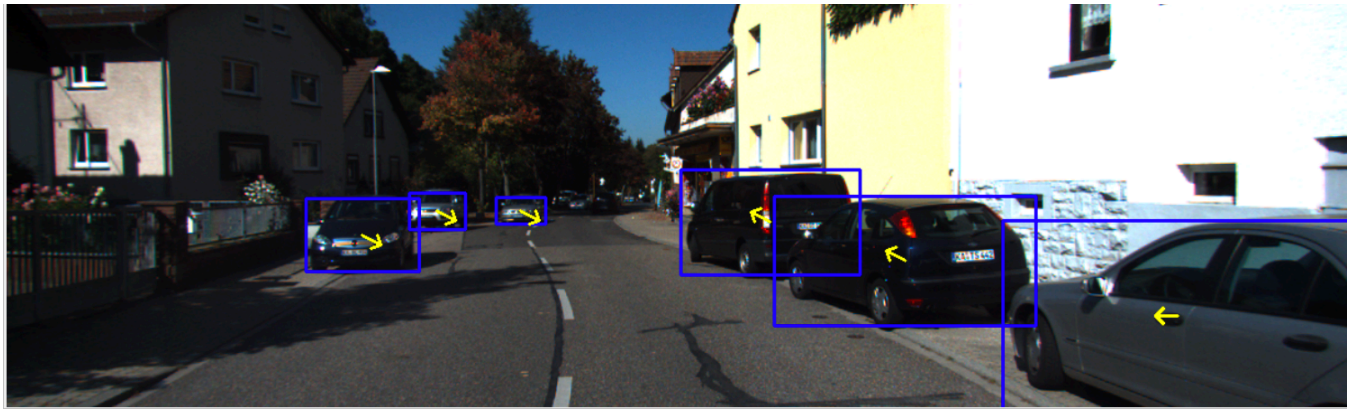
The arrow can be generate by the viewpoint and detect box but need to calibrate.

When angle = 0, the arrow is vertical. Thus change in vertical is -sin(angle+pi/2), change in horizontal is cos(angle+pi/2),

```python
import csv
import cv2
import math

img = './data_object/testing/image_2/um_000087.png'
image = cv2.imread(img)
angle = 'um_000087.csv'
```

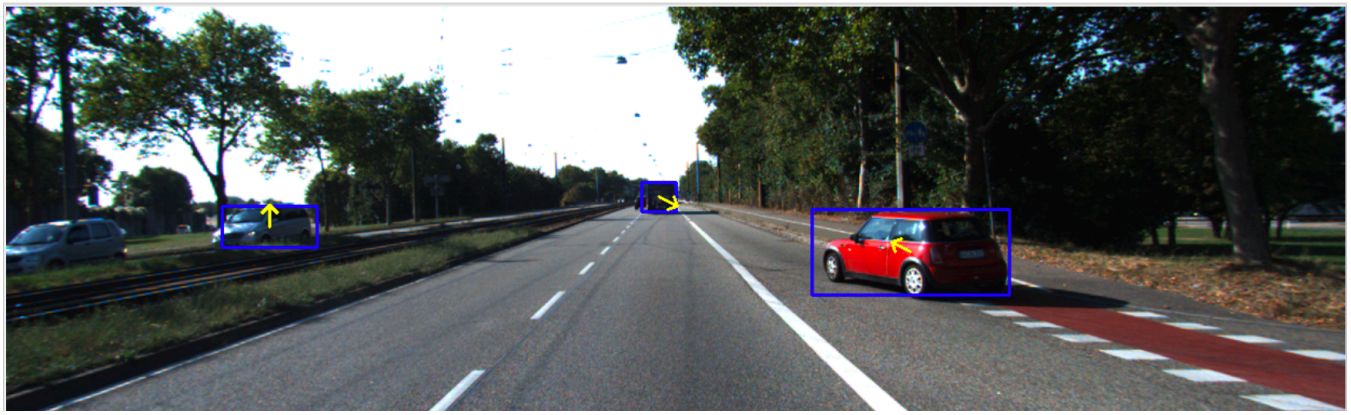
```python
detect = []
with open(angle) as csv_file:
    csv_reader = csv.reader(csv_file, delimiter=',')
    for row in csv_reader:
        detect.append(row)

for data in detect:
    data = [int(x) for x in data]
    color = (255, 0, 0)
    thickness = 2
    sp = (data[0], data[2])
    ep = (data[1], data[3])
    cv2.rectangle(image,sp, ep, color, thickness)
    start_point = (int((data[2]+data[3])/2), int((data[0]+data[1])/2))

    # End coordinate
    data[4] = data[4]+ math.pi/2
    end_point = (start_point[0]-int(20*math.sin(data[4])),start_point[1]+int(20*math.cos(data[4])))

    print(end_point)
    # Yellow color in BGR
    color = (0, 255, 255)
    # Line thickness of 9 px
    thickness = 2
    # Using cv2.arrowedLine() method
    # Draw a red arrow line
    # with thickness of 9 px and tipLength = 0.5
    image = cv2.arrowedLine(image, start_point[::-1], end_point[::-1],
                            color, thickness, tipLength=0.5)

cv2.imwrite("arror87.png", image)
```

Result



Since we only have 12 classifier, and thus when we have a angle in between it will be less accurate.

Bad result



When the segment is very small, since we will resize it to same size with others, it will lost a lot of feature. But how ever in autonomous driving, car with small segment are very far away and we don't need to analysis it yet.

Question 9

Given the ground plane, estimated depth, and the location of the car's bounding box,

how would you compute a 3D bounding box around each detected car? Add the 3D

bounding boxes to your plot from 5.

To solve this question, first we need to get some data from previous parts. Including 2D box location, orientation of the car (the more accurate, the better result. So, I fixed some values manually) and the 3D location of the car. Besides, we also need the size of the car (I assume each car have similar size) and the camera rotation matrices provided to compute 3D value to 2D.

Assume we are given the size of each car, then we can easily compute 8 points to generate a 3D box, for example, the front left upper corner can be -1/2*width, height, -1/2*length to the center of the car. with the 8 points are calculated then rotate it by the given rotation angle, we next can add the car

center position to it and get the relative location of each point in real world to the camera. The last step is applying camera rotation matrices to each of those points and get the 2D positions in the image. Connect pair of points to draw the 3D bounding boxes.

As for the 3d Box in the cloud points, it will follow the same steps above instead of transferring the points back to 2D image, this time we plot the points in the cloud with the calculated 3D location, then draw the box.

Code for 3D box in 2d image:

```
def project_to_image(P, pts):
    n_pts = pts.shape[1]
#     print('pts:', pts)
    #print('ones', np.ones((1, n_pts)))
    X = np.concatenate((pts, np.ones((1, n_pts))), axis=0)
    x = np.dot(P, X)
    x = x[:2, :] / x[2:3, :]
    return x
```

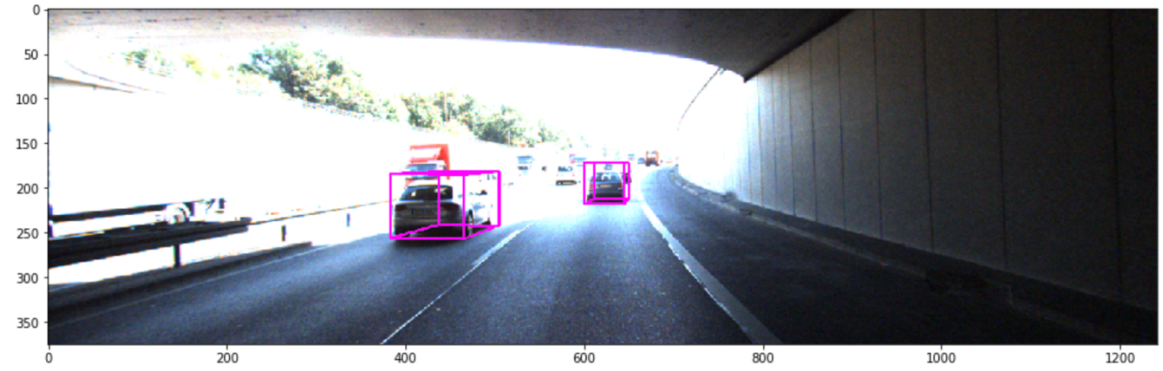
```
def kitti_plot_3d_bbox(img, label:KittiLabel, cam_to_img):

    colors = {
        'green': (0, 255, 0),
        'pink': (255, 0, 255),
        'blue': (0, 0, 255)
    }

    color = colors['pink']


    # Object dimensions
    h, w, l = label.dimensions

    # Coordinate of object centers
    # Note that label.location is location of the point at the bottom of the object
    # Need to add half object height to get object center
    tx, ty, tz = label.location

    theta_ray = np.arctan2(tx, tz)
    theta_local = label.alpha
    theta = theta_ray + theta_local

    ry = theta

    obj2body_R = R.from_euler('y', ry).as_dcm()
    obj2body_t = np.array([tx, ty, tz], dtype=float).reshape(-1, 1)
```

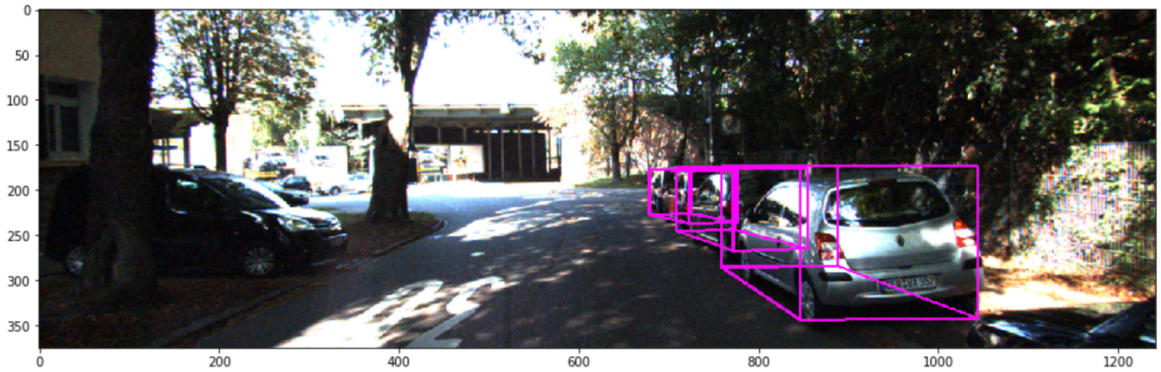
```
# Box corners
# x-axis on object (car) points from back to front of car
x_corners = [l/2, l/2, -l/2, -l/2, l/2, l/2, -l/2, -l/2]
y_corners = [0,0,0,0,-h,-h,-h,-h]
z_corners = [w/2, -w/2, -w/2, w/2, w/2, -w/2, -w/2, w/2]

corners_in_obj = np.asarray([x_corners, y_corners, z_corners], dtype=float)
corners_in_body = np.dot(obj2body_R, corners_in_obj) + obj2body_t

# Project to image
corners_in_img = project_to_image(cam_to_img, corners_in_body)
face_idxes = [[0, 1, 5, 4], # front face
         [1, 2, 6, 5], #left face
         [2, 3, 7, 6], # back face
         [3, 0, 4, 7], #right face
           ]

for idx in face_idxes:
    p1 = corners_in_img[:, idx].astype(int)
    p2 = np.roll(p1, 1, axis=1)
    for i in range(p1.shape[1]):
        cv2.line(img, (p1[0, i], p1[1, i]), (p2[0, i], p2[1, i]), color, 2 )
return img
```

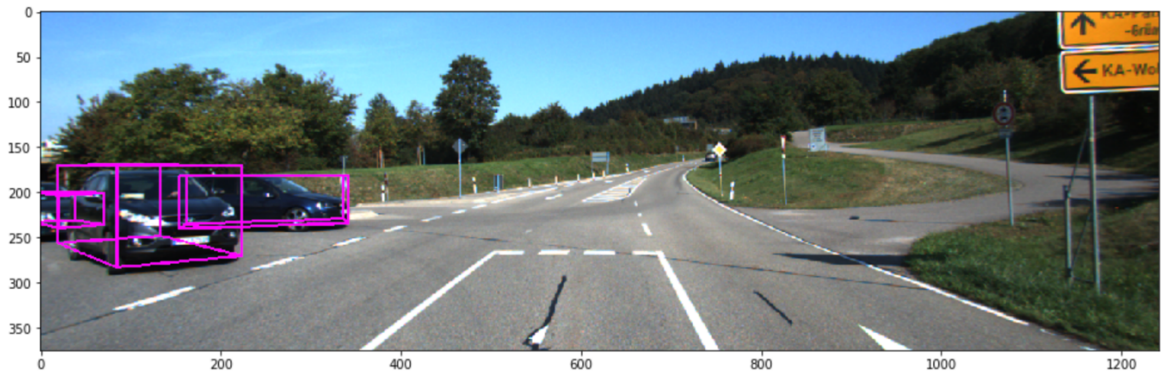
```
def main():
    img_file = './kitti_2d/training/image_2/000001.png'
    label_file = './kitti_2d/training/label_2/000001.txt'
    calib_file = './kitti_2d/training/calib/000001.txt'
    image = cv2.imread(img_file, cv2.COLOR_BGR2RGB)
    image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)

    labels = kitti_read_label_file(label_file)
    labels = [l for l in labels if l.type != 'DontCare']

    cam_to_img = kitti_get_calibration_cam_to_image(calib_file)
    print(cam_to_img)

    for label_info in labels:
        if label_info.truncated > 0:
            continue


        # Overlay 3D box using computed location
        image = kitti_plot_3d_bbox(image, label_info, cam_to_img)
    return image

img = main()
plt.figure(figsize=(16, 16))
plt.imshow(img)
plt.savefig('img_with_box3d.png')
```

Reference: https://github.com/nhonth/visualize-3D-BoundingBox.git

Examples:







Code for 3D box in cloud:

Generate 8 points by given car size

```python
def get_box_points(location, angle):

    tx, ty, tz = location
    theta_ray = np.arctan2(tx, tz)
    theta_local = angle
    ry = theta_ray + theta_local

    obj2body_R = R.from_euler('y', ry).as_dcm()
    obj2body_t = np.array([tx, ty, tz], dtype=float).reshape(-1, 1)
    l, h, w = car_size
    x_corners = [l / 2, l / 2, -l / 2, -l / 2, l / 2, l / 2, -l / 2, -l / 2]
    y_corners = [0, 0, 0, 0, -h, -h, -h, -h]
    z_corners = [w / 2, -w / 2, -w / 2, w / 2, w / 2, -w / 2, -w / 2, w / 2]
    corners_in_obj = np.asarray([x_corners, y_corners, z_corners], dtype=float)
    corners_in_body = np.dot(obj2body_R, corners_in_obj) + obj2body_t
    return corners_in_body.T
```

```python
def draw_box(points):

    lines = [[0, 1], [1, 5], [2, 3], [4, 5], [1, 2], [3, 0], [6, 7],
             [0, 4], [1, 5], [2, 6], [3, 7], [4, 7], [5, 6]]
    colors = [[0, 255, 255] for i in range(len(lines))]
    line_set = o3d.geometry.LineSet()
    line_set.lines = o3d.utility.Vector2iVector(lines)
    line_set.colors = o3d.utility.Vector3dVector(colors)
    print(points)
    line_set.points = o3d.utility.Vector3dVector(points)
    return line_set
```

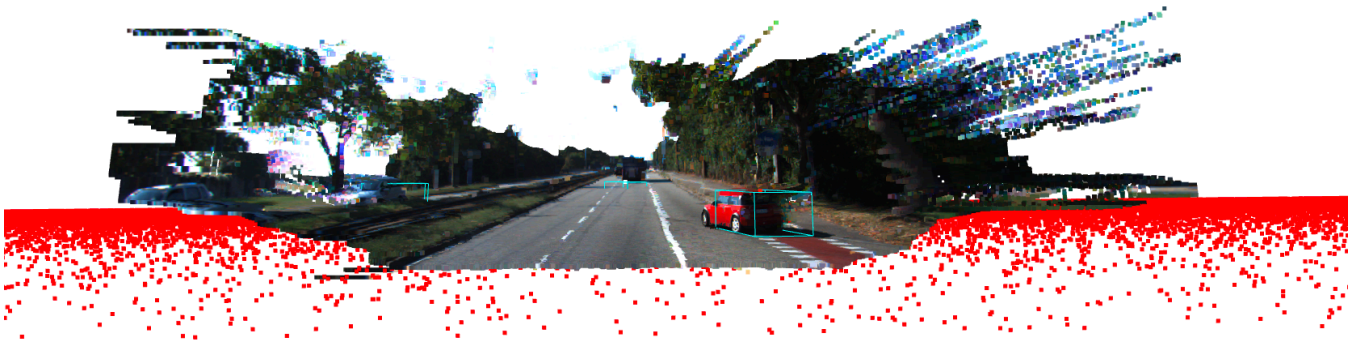
To get the 3D location of the car, first I get the center part of the car patch and get its 3D location. Then apply sin and cos function to calculate how far from that point to the car center point. Then add them up to get the final location for the center point of the car. However, it will not accurate enough sometimes.

```python
def get_location(center, ori):
    angle = -ori / 180 * np.pi
    # new3d_img = calculate_3d(imgnl, depth)
    location = three_d[center[0], center[1], :]
    location[0] = location[0] + np.sin(angle) * 1 / 2 * car_size[0]
    location[1] = 0
    location[2] = location[2] + np.cos(angle) * 1 / 2 * car_size[2]
    return location, angle
```

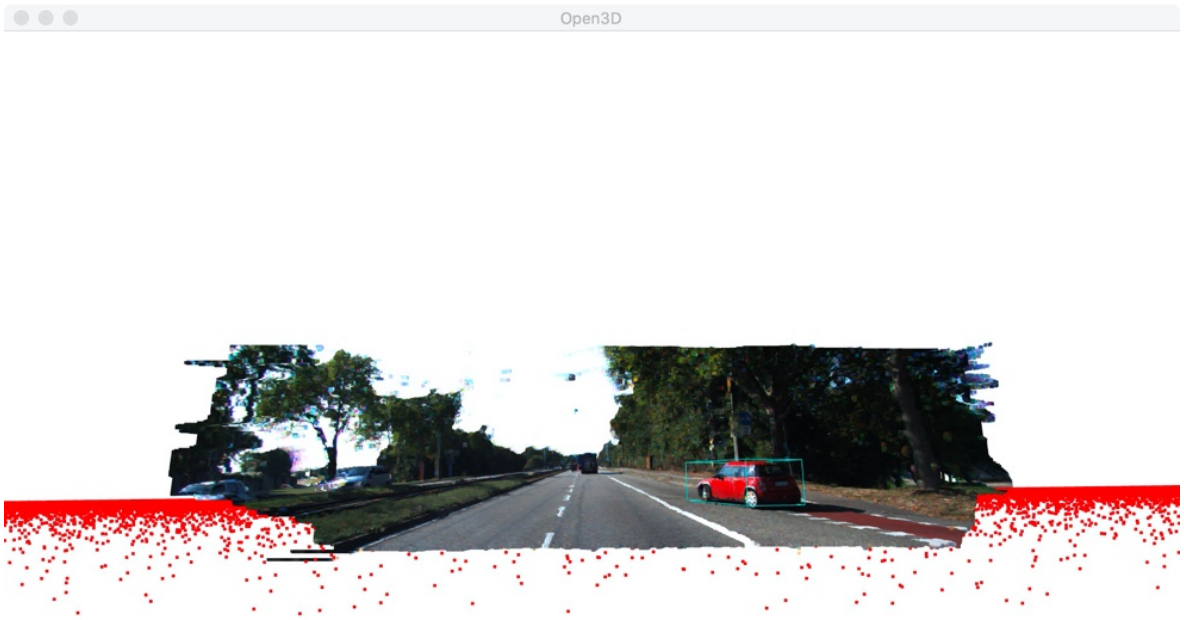
```python
location, angle = get_location(center, angle)
line_set = draw_box(get_box_points(location, angle))
print('detail',location, angle)
```

Example:

By using the data from previous parts, I draw the boxes for the detected cars. However, the data is definitely not accurate enough, thus the error in the result 3D cloud is huge.



After I fixed the data manually by giving a more accurate center point for the red car on the right side, the result performs better for the red car.


Open3D