NUST COLLEGE OF ELECTRICAL

AND MECHANICAL ENGINEERING

SEMI-AUTOMATED ANNOTATION OF LIDAR BASED DATA

PROJECT REPORT

DE-38 (DEE)

SUBMITTED BY

AMAL SALEEM

MINHAH SALEEM

RABEEA FATMA KHAN

BACHELORS IN ELECTRICAL ENGINEERING 2020

PROJECT SUPERVISOR

DR SHAHZOR AHMAD

COLLEGE OF ELECTRICAL AND MECHANICAL ENGINEERING

PESHAWAR ROAD, RAWALPINDI

SEMI-AUTOMATED ANNOTATION OF LIDAR BASED DATA

PROJECT REPORT

DE-38 (DEE)

SUBMITTED BY

AMAL SALEEM

MINHAH SALEEM

RABEEA FATMA KHAN

BACHELORS IN ELECTRICAL ENGINEERING 2020

PROJECT SUPERVISOR

DR SHAHZOR AHMAD

COLLEGE OF ELECTRICAL AND MECHANICAL ENGINEERING

PESHAWAR ROAD, RAWALPINDI


بسم الله الرحمن الرحيم
١٤١٨

CERTIFICATE OF APPROVAL

It is to certify that the project "**Semi-Automated Annotation of LiDAR Based Data"** was done by **Amal Saleem**, **Minhah Saleem** and **Rabeea Fatma Khan** under supervision of **Dr. Shahzor Ahmad.**

This project is submitted to **Department of Electrical Engineering**, College of Electrical and Mechanical Engineering (Peshawar Road Rawalpindi), National University of Sciences and Technology, Pakistan in partial fulfilment of requirements for the degree of Bachelors of Engineering in Electrical Engineering.

Students:

1- **Amal Saleem**
NUST ID: ____________________________ Signature:____________________

2- **Minhah Saleem**
NUST ID: ____________________________ Signature:____________________

3- **Rabeea Fatma Khan**
NUST ID: ____________________________ Signature:____________________

APPROVED BY:

Project Supervisor: ______________ Date: __________________________
Dr. Shahzor Ahmad

Head Of Department:______________ Date: ________________________
Dr. Fahad Mumtaz Malik

DECLARATION

We hereby declare that no portion of the work referred to this Project Thesis has been submitted in support of an application for another degree or qualification of this of any other university or other institute of learning. If any act of plagiarism, we are fully responsible for every disciplinary action taken against us depending upon the seriousness of the proven offence, even the cancellation of our degree.

1- **Amal Saleem** ______________________________

2- **Minhah Saleem** ______________________________

3- **Rabeea Fatma Khan** ______________________________

COPYRIGHT STATEMENT

ACKNOWLEDGMENTS

First and foremost, thanks to Allah Almighty for his countless blessings. Who gave us the ability and courage to understand the things, by the grace of whom every obstacle was dissolved in the fulfillment of this arduous task.

Apart from the efforts of the group, the success of any project depends largely on the encouragement and guidelines of many others. We wish to express our gratitude to our supervisor **Dr. Shahzor Ahmad** who was abundantly helpful and offered invaluable assistance, support and guidance.

In addition nothing would have been possible without the help of our beloved parents. Their unrelenting support and patience was instrumental. They were always there for us in our time of need.

In the end, we can’t thank our department enough, **Department of Electrical Engineering**, for nurturing us over the years and helping us excel in our field. We are really appreciative of all the faculty members and staff, who have played their roles impeccably in ensuring that we are provided with the best facilities and treatment.

ABSTRACT

The purpose of this project is to design a semi-automated annotation tool for LiDAR based data of traffic. Mask R-CNN was trained and used for annotation of RGB Camera images. The annotated labels were then translated on to 3D point cloud. It predicts labels in subsequent frames using tracking algorithm. The web app displays RGB Camera Images with predicted masks, 3D point cloud with adjustable bounding boxes and labels, and cropped image of selected object. It can be very beneficial in annotating LiDAR images, since it only needs a single click on the cluster of object points in 3D point cloud, thus saving a lot of time required otherwise for fully manual annotation.

Table of Contents

LIST OF FIGURES

LIST OF TABLES

LIST OF SYMBOLS

Acronyms

LiDAR = Light Detection And Ranging
CNN = Convolutional Neural Network
R-CNN = Regional Convolutional Neural Network
MS COCO = Microsoft Common Objects in Context
RMSE = Root Mean Square Error
RPN =Risk Priority Number
IoU = Intersection over Union

Shorthand (Formulae)

G_o = Sampled data matrix
C_o = Co-variance matrix
$\bar{p}$ = mean of p

1 INTRODUCTION

1.1 Project Background

LiDAR (Light Detection And Ranging) sensor uses a light beam to detect, track and monitor objects. Its main application is in autonomous vehicles of Level 4 and Level 5 as it is robust to illumination changes and delivers accurate distance measurement to close objects [1]. This makes it an efficient sensor when it comes to self-driving vehicles. The scope of this sensor in years to come has been the real motivation for us to take on this project.

For object detection using LiDAR point cloud, plenty of research has been done to provide an effective and efficient method. Using deep learning to achieve this has been the rage in recent years and many works have successfully achieved this goal [1], [2]. As deep learning required plenty of dataset for training, it is necessary that an annotation tool that efficiently annotates LiDAR data is present.

1.2 Problem Statement

Annotating camera images is a remarkably advanced technology with various software working towards providing impressive results. However, the problem arises when LiDAR point cloud is annotated as bare minimum work has been done in this field. Progress in this aspect is important to ensure the adoption of LiDAR sensors in Intelligent Transport System. Problems arising with LiDAR point clouds are:

1. Low resolution (sparse points)
2. Complexity in obtaining 3D bounding boxes and segmentation
3. Annotation of sequential frames

We aim to develop an annotation tool that caters to all of these issues, is open sourced so that it could be used by everyone and hence provide any betterments that could come its way. As LiDAR sensors are a novel technology, there are high chances of advancement in the algorithms, equipment etc. Our aim is to provide a user friendly interface and algorithm that can be updated at any instance.

1.3 Objectives

We aim to come forth with a state-of-the-art annotation tool that would deploy one-click annotation to accurately determine points belonging to an object and to create tight 3D bounding boxes. Labels obtained from camera image annotation will be inferred with the obtained bounding boxes. These results will be tracked on to the consequent frames as LiDAR data is obtained over sequential frames. This tool enables human annotators to easily annotate data and also create new datasets for deep learning, as these algorithms require a large amount of data for training.

1.4 Scope

The future scope of LiDAR sensors is what motivated us to pursue this project. Currently, LiDAR is a new technology whose advantages have been seen by many leading companies in regard to autonomous driving. Companies like Ford, Audi, Porsche and Volvo have either integrated the sensor into their vehicles or invested in the LiDAR sensor industry [3]. Further, according to Wijeyasinghe et al. [4] in ten years the global market for these sensors will increase to $5.4 billion and is one of the emerging industries in terms of venture capital (VC) dollars.

The numbers show that in recent future the demand and research in LiDAR sensors is bound to increase. To be a part of a dynamic change in autonomous vehicles, it is essential that an effectual annotation tool be developed to ease the production of datasets.

2 LITERATURE REVIEW

This project drew inspiration from various related works that focused on object detection through point cloud and LiDAR based annotation. The project works on an effective manner of annotation – one-click annotation. One-click annotation requires the human annotator to click only once on the point cloud and get the bounding box for the entire object. This is an intelligent and efficient method that considerably reduces the time required to annotate image or video data.

For this, it is essential that the bounding box tightly cover the object and correctly predict labels for each point cloud, known as semantic segmentation [1].

Recent approach towards detection utilizes deep learning algorithms. This gives accurate results than previous approaches.

2.1 Object Detection Using Mask RCNN

Mask RCNN [5] performs instance-level detection of objects which is essential when working with autonomous vehicles. It accurately detects object in an image and generates a segmentation mask. This is done for every instance. Mask RCNN is an extension of Faster 54xRCNN as it delivers masks as well as bounding boxes for each object. Faster RCNN provides with bounding boxes around the object.

Mask RCNN works with Region Proposal Network’s first stage to obtain anchors.

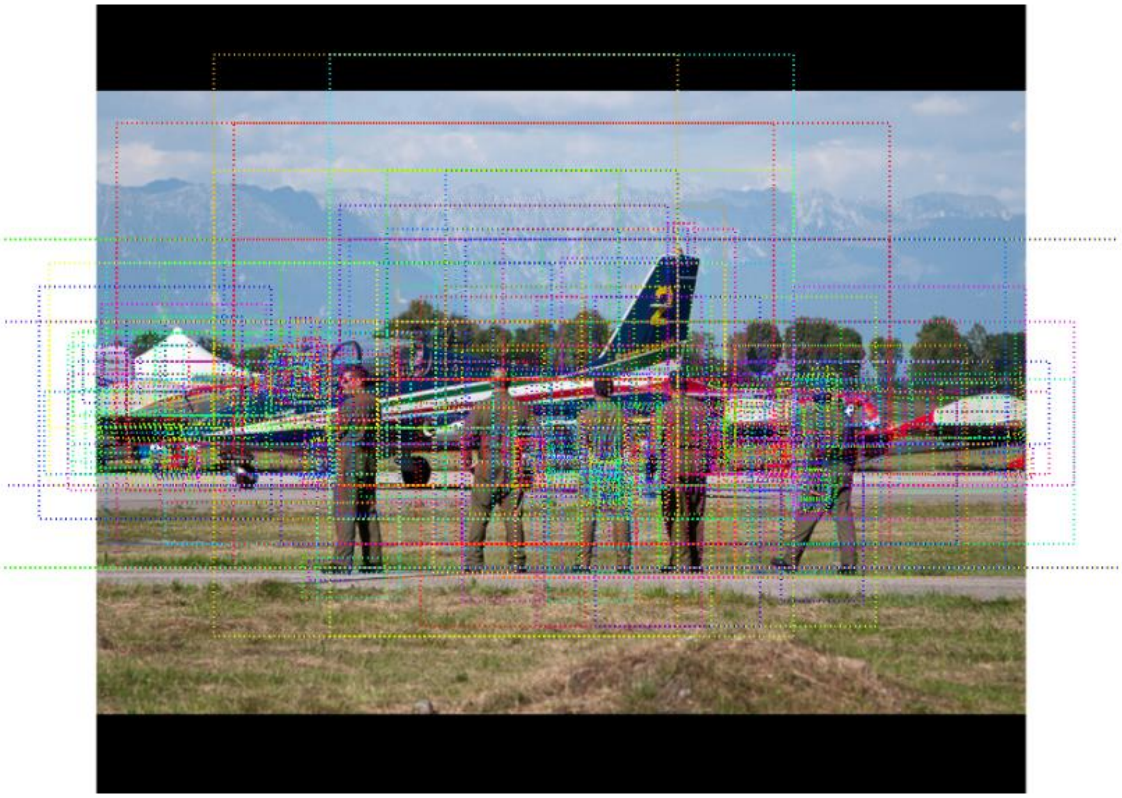


Figure 1: Visualization of every step in RPN

Then these are refined to obtain tight bounding boxes around every object and its label.

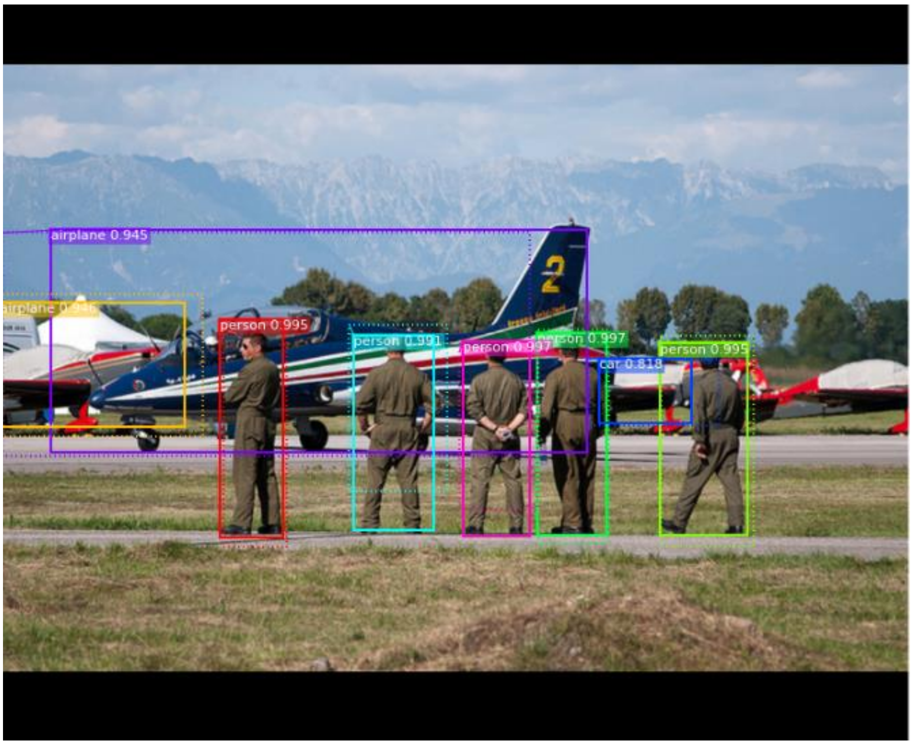


Figure 2: Refinement and final bounding boxes

From this, masks are generated that are then placed on the object in the bounding box.

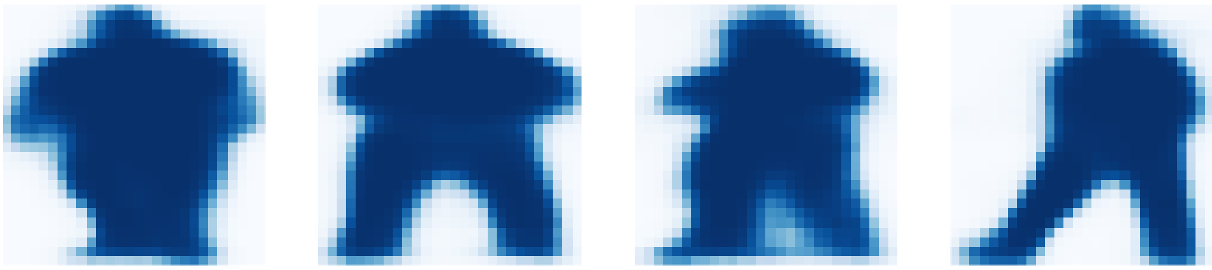


Figure 3: Mask Generation

To avoid any issues, it is important to keep random noise and/or all zeros in mind. To prevent such a nuisance, multiple tools can be applied:

- *Layer activation* is when the activation of multiple layers is inspected.

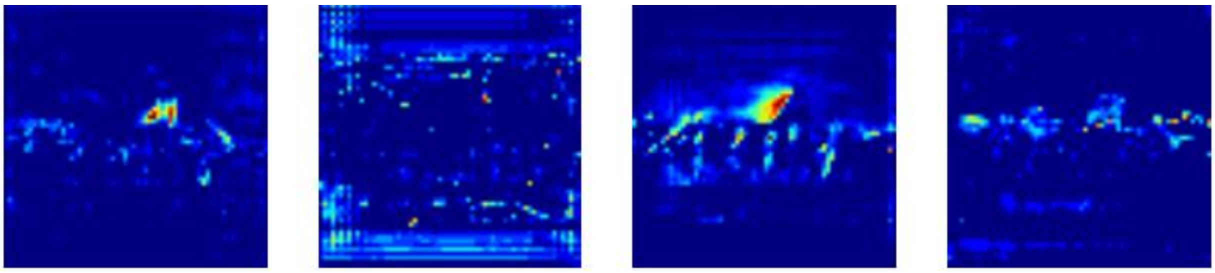


Figure 4: Layer Activation Visualization

- *Weight histograms* which inspect the weights of histograms for each object.

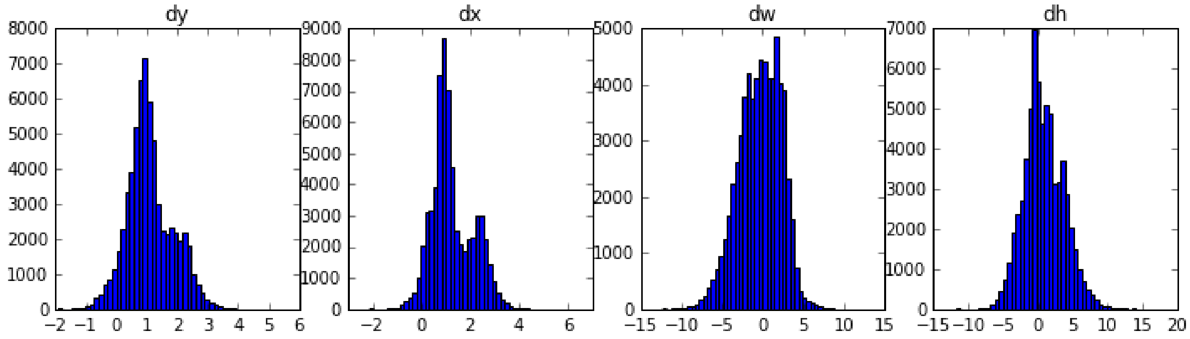


Figure 5: Weight Histograms

- *TensorBoard* is a widely adopted tool in machine learning. It tracks loss and accuracy as well as show histograms of weights, biases.

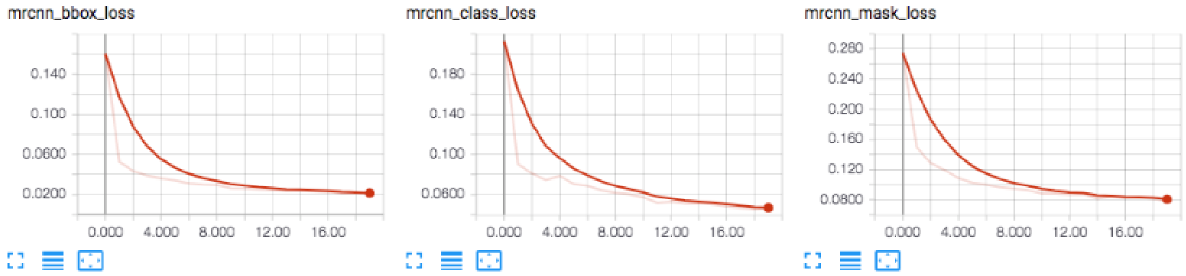


Figure 6: TensorBoard Visualization of Loss

Hence, what we get are exact masks on objects as well as their bounding boxes.

Figure 7: Final Predictions (with masks and bounding boxes)

YOLO has taken a newer and faster approach towards object detection. It integrates the parts of object detection into a single neural network [6]. It does so by first dividing the image into regions and then seeing if the center of an object lies within that grid. It then predicts bounding boxes for that object as well as confidence scores. The predictions contain five elements. The *(x,y)* coordinates of the bounding box center with respect to the grid boundaries, the width and height with respect to the entire image and lastly, the confidence score with respect to the ground truth. These predicted probabilities determine the weight of the bounding box.

Instead of using thousands of networks to detect objects for a single image like its predecessors, YOLO uses a single network. This improves its operational and computational time by 1000 times for Fast R-CNN and 100 times for Fast R-CNN, with state-of-the-art results. Hence making it one of the most popular detector.

2.2 3-D Object Detection using Point Clouds

Plenty research has been done to efficiently detect 3D object from point clouds. These point clouds are obtained via LiDAR, stereo etc. Most researches focus on transforming the point cloud data to 3D voxel grids which are regularly spaced. Earlier works used handcrafted geometrical feature representation of the scattered point clouds as in [7]. VoxelNet removes the manual feature representation as demonstrated in [8], achieving a better operational time and can learn various shapes thus giving better results in detecting pedestrians etc. than its predecessors. It divides the point cloud into regularly distanced 3D voxel grid, and then converts the points within each to represent a feature vector. These vectors are scored by RPN for object detection.

This approach, however, holds large amounts of data and can cause computational issues. Hence, a better approach to this is shown in [9], where variance of the points in the input point cloud data is calculated. This work not only focuses on object identification but also in scene segmentation.

Another notable mention, which is similar to our project is vehicle detection using LiDAR point cloud in [2]. The point cloud data is projected on to a 2D point map where the confidence score is measured using a 2D end-to-end CNN. Simultaneously, bounding boxes are also created. This work successfully predicts 3D bounding boxes using a 2D CNN.

2.3 Dataset

Training a deep learning network requires significant amount of labelled data, specifically LiDAR point cloud data. KITTI dataset [10], which comprises of 15,000 frames of 3D bounding box annotations is one of these datasets. For our project we have used pre-trained COCO weights for training and the KITTI dataset [10] for testing.

Another dataset, the Apolloscape dataset [11] comprises of 140,000 frames of point-wise background annotation. The two datasets mentioned are public datasets. It is essential that a large amount of data is used to train and test the product to ensure safe operation in real world installation. As in the real world, the sensor configuration, position etc. could differ from train and test data which could lead to detection issues. Hence, it is vital to produce more datasets for calibration of sensor in different real life scenarios.

2.4 Annotation tools

An optimal annotation tool for LiDAR point clouds for intelligent transport system must be compatible with the following specifications:

1. Be compatible with LiDAR data
2. Be efficient with video annotation (be able to work on sequence frames)

To create huge amount of data required by deep learning networks for LiDAR sensor based detection, an effectual annotation tool is required. However, most work has been done on annotation of images like PolygonRNN [12] and an improved version [13], which helps accelerate the annotation process by using humans-in-the-loop and works on high resolution images to detect high resolution objects. In addition to this, VATIC [14] is a video annotator that tracks consecutive frames (through linear interpolation) to create custom video datasets.

As shown above, annotation tools that deal directly with LiDAR based data are not readily available. The ones available are not open sourced as in [7]. Hence, our main motivation is the work done by Wang et al. [15]. It concentrates on developing an open sourced efficient annotation tool for LiDAR point cloud data, which takes into consideration the recent research work done in LiDAR based detection and how it could be used in L4-L5 levels of autonomous vehicles as well as in an intelligent transport and surveillance system. It focuses on the main issues related to LiDAR sensors:

4. Low resolution (sparse points)
5. Complexity in obtaining 3D bounding boxes and segmentation
6. Annotation of sequential frames

The resolution of LiDAR sensor translates to the density of the point cloud obtained, where a 64 line LiDAR will provide a better resolution than a 32 line one. Consequently, a 64 line LiDAR sensor will provide a significantly lower Root Mean Square Error (RMSE) [16] as compared to a 32 one, however this improvement in resolution also renders the increased cost. In [12], researchers work towards estimating a dense depth image from a sparse depth image by developing a deep regression model. However, this is would still not cater to object identification. The difficulty in catering to sparse point clouds is emphasized in [11] effectively.

Annotating a LiDAR point cloud would require predicting a 3D bounding box around the object. This should tightly cover the object. And to correctly predict the point cloud belonging to the object. Annotating consecutive frames is vital to creating new datasets without time-consuming processes. In [11] identified objects in previous frame are tracks on to the next one. These challenges pose a significant challenge. To overcome these challenges, [11] performs sensor fusion, where labels from an annotated image are projected onto a 3D point cloud. This approach is highly effective as image annotators are remarkably advanced in detection and annotation. The obtained labels are projected on the point cloud. This makes sense as the LiDAR sensor and camera are calibrated.

One-click annotation uses clustering algorithms to obtain the points belonging to an image. Any point need to be clicked only once, the algorithms cluster and estimate the object, consequently drawing the 2D bounding box. For each bounding box, the points in it are considered as the object and the labels obtained from image annotation are inferred to it.

Tracking a detected object to the next image is important to reduce computation and operational time of annotation. According to Wang et al. [11, p. 2], these steps resulted in a "6.2x reduction in annotation time while delivering better label quality, as measured by 23.6% and 2.2% higher instance level precision and recall, and 2.0% higher bounding box IoU".

The methodology followed in this work is firstly, to calibrate the camera and LiDAR sensor. This is to ensure that during inferring, every pixel of camera image has a label for every pixel of LiDAR data. Then comes the pre-labeling of LiDAR data. Annotation of camera images is performed, using Mask R-CNN. In one-click annotation, the LiDAR data points are clustered to give the object, on which the bounding boxes are fitted. Tracking is done though application of Kalman filter to determine the center of a bounding box. For the consequent frame, acceleration and velocity of the center are used to give the bounding box.

3 GROUND REMOVAL ALGORITHM

3.1 Motivation

As part of our One Click Annotation strategy it is imperative that the ground be removed from our point cloud in order to assist in getting clear clusters of objects after we click on them.

3.2 Objective

To make a good estimation of ground segment in LiDAR point cloud in order to remove the ground data from our point cloud.

3.3 Algorithm

Start off by modeling the ground as a segment of planes where 'n' is the normal vector approximation to the ground plane with,

$$n = [a; b; c]^T$$

'p' is any LiDAR data point,

$$p = [x; y; z]^T$$

and 'd' is point 'p's distance from the ground.
The aim is to find an approximation of' 'n' for which all points on the ground plane have a minimized distance 'd'.

Note: The data in the bin file of the point cloud is a flattened Numpy array which has to be reshaped into a matrix with 4 columns: x, y, z, intensity. This is done by using the command:
Data.reshape((-1, 4))
Where the '4' depicts the no. of columns and '-1' tells the function that the number of rows is an unknown dimension. Numpy will figure out this unknown dimension at runtime. For our algorithm we are not concerned with column 4 since the algorithm uses only data co-ordinates.

To approximate 'n':

STEP 1: Collect a sample of points called "data" with the lowest height 'z' or in our case points with 'z' lower than a threshold.
Threshold = The mean of 'z' of all data points.
Stack all these sample data points in a matrix G.

```
for i in range(data.shape[0]):
    if data[i,2] < mean:
        G = np.vstack((G,data[i]))
        a = a+1
```

STEP 2: Compute the Co-variance matrix 'C' of sampled data "G".
This can be done by using the formula:

$$\bar{\mathrm{p}} = \frac{1}{|G_0|} \sum_{i=1}^{|G_0|} \mathrm{p}_i,$$

$$C_0 = \sum_{i=1}^{|G_0|} (\mathrm{p}_i - \bar{\mathrm{p}})(\mathrm{p}_i - \bar{\mathrm{p}})^T.$$

Where 'p-bar' is the mean of the sampled data 'G' and 'pi' are consecutively all the points in matrix 'G'.

STEP 3: Check the variations in the Co-variance matrix, large variations correspond to the direction of plane and small variations correspond to the direction of normal vector.
Thus calculate the Single Value Decomposition (SVD) of the co-variance matrix 'C' and

model the Eigen vector corresponding to the smallest Eigen value of ‘C’ as a good approximation of the normal vector ‘n’.

STEP 4: Calculate the distance ‘d’ of all the data points ‘p’ with approximated normal vector using the equation:

$$n^T.\text{p} = \text{d}$$

where ‘d’ is the dot product of the normal vector and point.

STEP 5: Resample points on the plane by choosing those points with a distance value ‘d’ less than the threshold value.
Threshold for distance = 0.5 (We tried with several different threshold values and this gave the optimum result). Stack these sample points again in an empty matrix ‘G’.

STEP 6: Repeat steps 1 to 5

- Until normal vector ‘n’ converges
- For a fixed number of iterations

We chose a fixed number of iteration.
Iterations = 25
After approximating normal vector ‘n’, all the data points on the ground plane (points with ‘d’< 0.5 to approximated ‘n’) are automatically in the matrix G at the end of the last iteration.

STEP 7: Delete these points from the original point cloud.

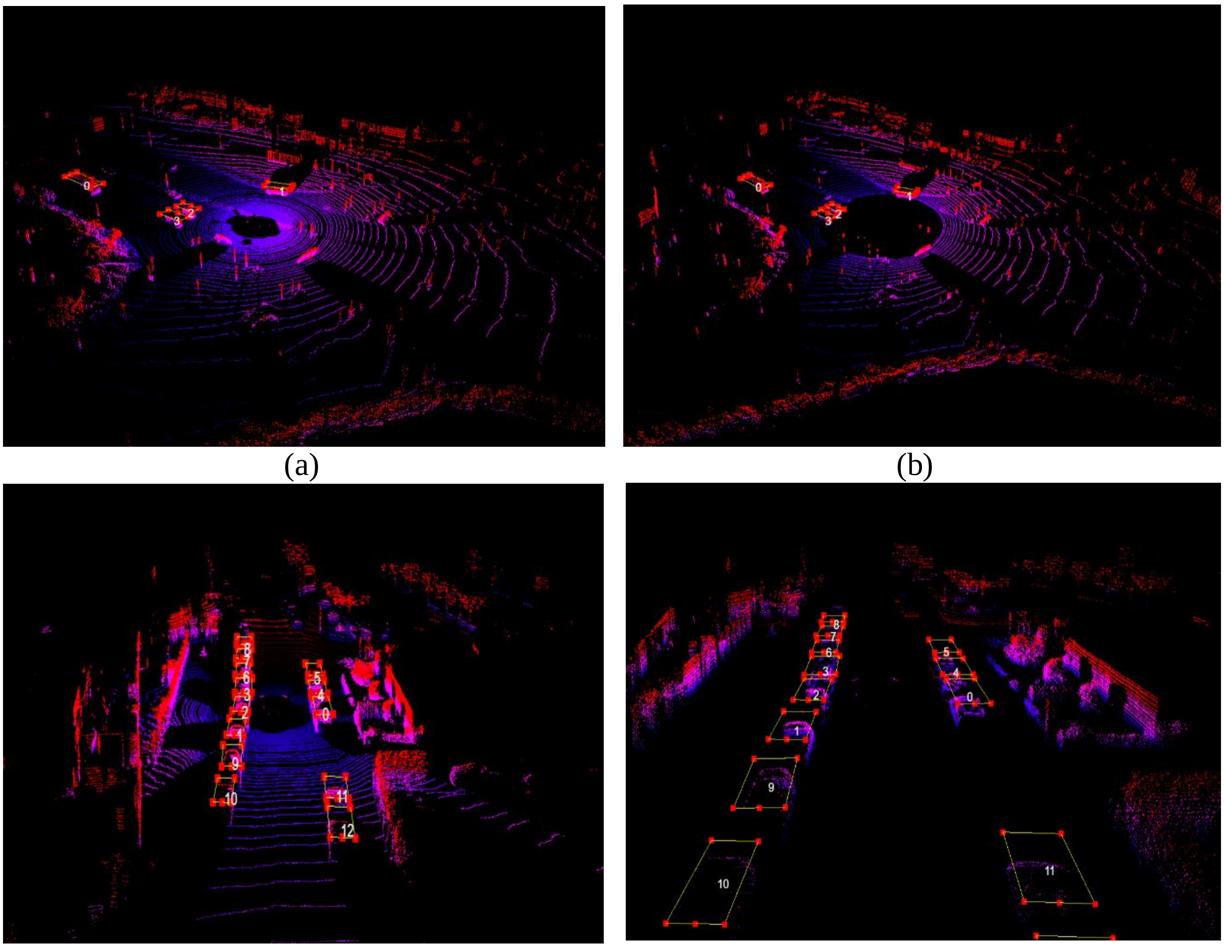
```
def groundremoved(G,data):
    i=0
    while (i< data.shape[0]) and (G.shape[0]!=0):
        if (data[i,0] == G[0,0]) and (data[i,1]==G[0,1]) and (data[i,2]==G[0,2]):
            data = np.delete(data,i,axis = 0)
            G = np.delete(G,0,axis = 0)
            i = i-1
        i=i+1
```

‘G’ matrix, composed of all the points on the ground plane, is deleted from original point cloud ‘data’.

3.4 Results

Dataset: **9_drive_0017_sync**
Point cloud file: 0000000000.bin

Normal vector approximation:	[-0.0497478 -0.056985 0.99713483]
Total data points in point cloud	(117765, 4)
Ground points approximated	(7313, 4)
Ground removed point cloud	(110452, 4)



(a) (b)

(c) (d)

Figure 8: ('a' and 'c') before ground removal ('b' and 'd') after implementing the ground removal algorithm.

A thing to note is that the point cloud available for viewing by user on our web app is one with its ground intact (this can be changed by setting 'ground-removed = True' when loading point clouds in function 'getFramePointCloud()' of our app.py to view ground removed point cloud). The ground removed point cloud is only used by our application after we've clicked on a cluster of points, to assist in the one click annotation process.

4 WEB APP GUI

The designed app is hosted on localhost and can be accessed using following web link after running the app:

http://127.0.0.1:5000/

JavaScript was used to write the template for webpage.

4.1 Loading Images

A binary file of the full point cloud, a binary file of the point cloud with the ground removed, and an image for which we need annotations to be done are placed in app/test_dataset folder. Batches of frames loaded from app/test_dataset folder can be seen in left pane along with object IDs and visible 3D point cloud which can be rotated and translated.

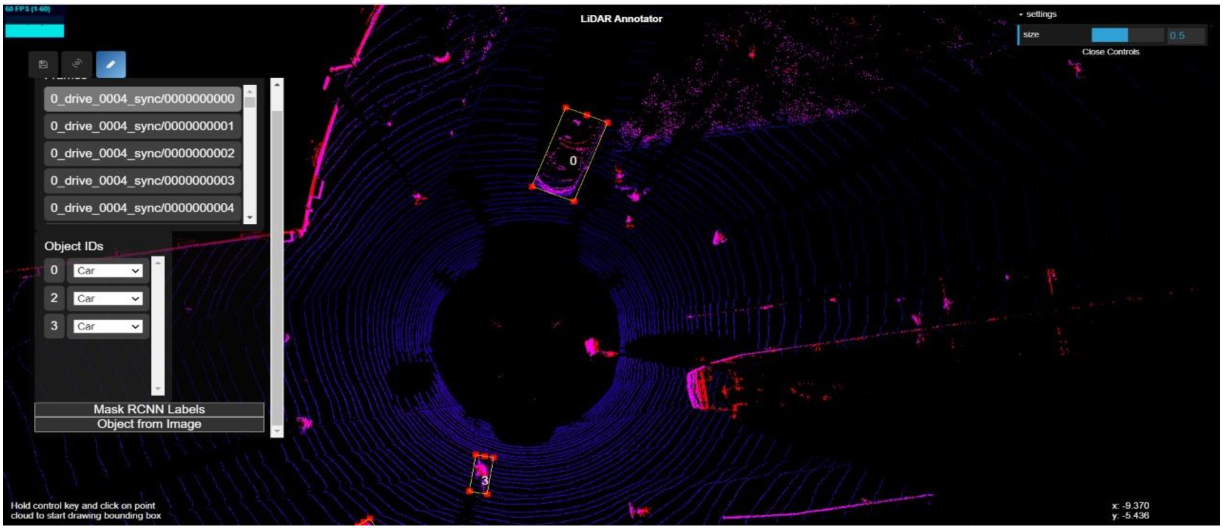


Figure 9: webpage interface showing image frames, point cloud, bounding box and object IDs.

4.2 One-click bounding box

Bounding box will be generated for a cluster by clicking on a single point on the cluster while holding the "a" key. This auto-generated bounding box can be adjusted according to requirement.

4.3 Bounding box labels

Predicted labels for bounding boxes can be seen in Object IDs table on the left of point cloud. Labels are given in front of respective bounding box index and can be changed by simply clicking on label and selective another one from the dropdown menu.

4.4 Mask RCNN labels

RGB images with segmented masks and bounding boxes corresponding to loaded frame can be seen on clicking "Mask RCNN labels" button.

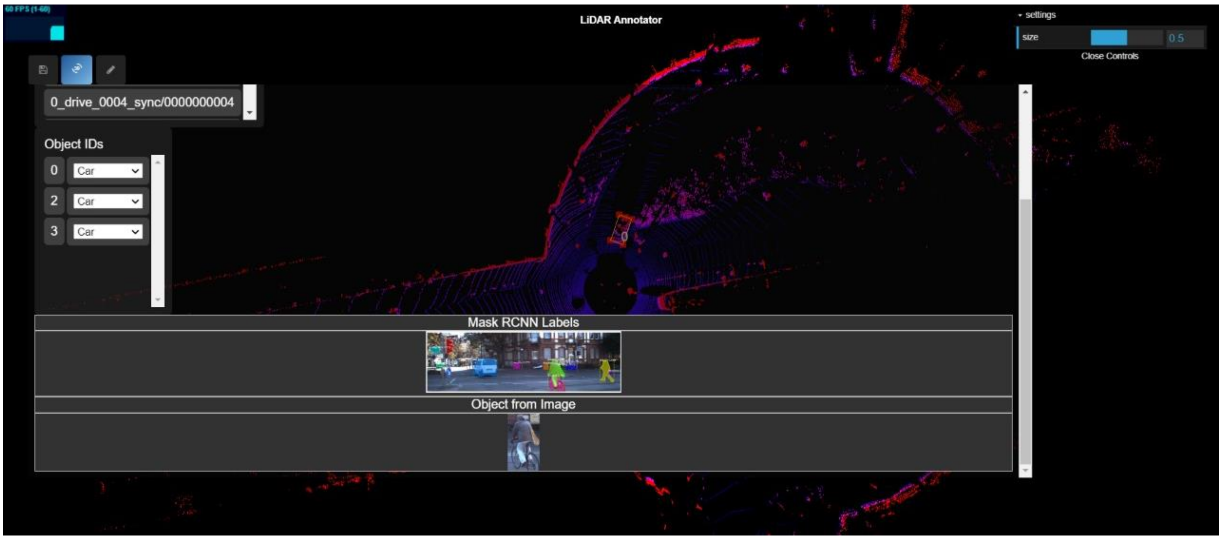


Figure 10: Mask RCNN labels button showing output of Mask RCNN for RGB image of loaded frame and object from image button showing a bike.

4.5 Object from image

The button on being clicked will display the RGB image object for the bounding box selected on point cloud.

5 CODE DESIGN

There are mainly 3 files that will be discussed in our project report.
App.py, mask_rcnn_demo.py, predictlabel.py.

5.1 App.py

This is the main web app file that is executed and it further calls all of our other python files for their respective functions.

Flask

Flask is a framework used for interfacing between python and web applications. Therefore we imported the flask requirements through:

From flask import Flask, render_template

To set up GUI the following code is executed:

```
app = Flask(__name__, static_url_path='/static')
DIR_PATH = os.path.dirname(os.path.realpath(__file__))

@app.route("/")
def root():
    return render_template("index.html")
```

Where 'index.html' contains the web app design and the static folder contains all the JavaScript (.js) files needed to interface our python code with the web application. The "@app.route("arg1" , arg2) " function carries out the function specified in its body at the URL: host 'arg1' and with method specified in 'arg2'. To set up web page we use render_template("index.html") (which returns our web page design) at the URL: host/

JSON

All the data being transferred to and from the web application is in JavaScript Object Notation (.JSON) type. Therefore we import the necessary libraries through:

Import Json
from flask import request, jsonify

There are two methods of requesting/transferring data from client to server.

- Get method: appends the parameters in URL (unsafe)
- POST method: carries request parameter within the body (secure)

To specify which method is to be used we write "methods = ['POST']" or "methods = ['GET']" as the second argument to "@app.route()" function.

To receive data from server we use the function: request.get_json(). We can further specify the exact sub-list of data we want to get e.g. request.get_json()['fname'] or request.get_json()['bounding boxes'] which gets the name of the file (.bin or .jpg) currently being viewed in our web app and the bounding box indices of that file respectively.

Mask-RCNN labels:

```
@app.route("/getMaskRCNNLabels", methods=['POST'])
def getMaskRCNNLabels():
    filename = request.get_json()['fname']
    get_mask_rcnn_labels(filename)
    return str(get_mask_rcnn_labels(filename))
```

This function is executed at the URL: host/getMaskRCNNLabels, with secure data transferring through POST method. It requests the filename of the loaded point cloud on which we have to perform Mask RCNN labeling : ‘fname’.
This is then sent as input parameter to our mask_rcnn.py file which contains the function ‘get_mask_rcnn_labels()’. We will further discuss this in our file “Mask_rcnn_demo.py”.
This function

- Generates masks on RGB image and places them in the statics/images folder as masked_image.png
- Calibrates corresponding masks from RGB onto the point cloud to give us clear clusters with masks which are easier to annotate.

The masks on point cloud are saved as ‘indices.bin’ in the app/output directory and are also displayed on the web app.

<u>Predict Labels:</u>

```
@app.route("/predictLabel", methods=['POST'])
def predictLabel():
    json_request = request.get_json()
    json_data = json.dumps(json_request)
    filename = json_request['filename'].split('.')[0]
    os.system("rd {}/*".format(os.path.join(DIR_PATH, "static/images")))
    predicted_label = predict_label(json_data, filename)
    in_fov = os.path.exists(os.path.join(DIR_PATH, "static/images/cropped_image.jpg"))
    return ",".join([str(predicted_label), str(in_fov)])
```

This functions is executed at the URL: host/predictlabel using the POST method. It requests all json data and dumps (converts it from python string to .json type) it into json_data. The filename (of the format 10000012.bin) is split at ‘ . ‘ to give us a new filename (of format 10000012).
The predict_label function takes both the json_data of the current file as well as the filename and returns the label to be uploaded on the web app corresponding to the selected object id.

Labels	Keywords					
Car	Car					
Van	van	minivan	Bus	Minibus		
Truck	Truck					
Pedestrian	Pedestrian	Person	Man	Woman	Walker	
cyclist	Motorcyclist	Bicyclist	Bicycle	motorcycle	bike	motorbike

Table 1: labels corresponding to object IDs.

5.2 Mask_RCNN_Demo.py

This the Mask RCNN interface file which is called from the get_mask_rcnn_labels() function in the app.py. It takes one input argument at runtime: the filename of current point cloud. This filename is of the format: drive/name for example: 4_drive_0005_sync/0000000140.
It can split the filename at ‘ / ’ to have the full path to image.
It starts off by naming directories:

ROOT_DIR	Current working directory i.e. directory of mask_rcnn_demo.py (app/Mask_RCNN)
MODEL_DIR	app/Mask_RCNN/logs
PARENT_DIR	app
DATA_DIR	app/test_datasets
COCO_MODEL_PATH	App/Mask_RCNN/mask_rcnn_coco.h
filename	App/test_dataset/drive/image/name.png

Table 2: Arguments used by mask_RCNN_demo.py

The model directory contains previous logs of our mask_r-cnn in case of re-training the mask_r-cnn on new dataset. During training the weights deep learned through Mask RCNN are kept in 'logs' and can then be retrieved at inference.
Since COCO has already supplied us with weights they trained during Mask RCNN training on their coco dataset, we can start by using these pre-trained weights to run inference on. These weights are stored in the directory COCO_MODEL_PATH as a ' .h ' file.
The data directory contains all the test dataset through which we find our current RGB image for point cloud.

There are two ways to use Mask R-CNN:

- Training: which will train the Mask R-CNN on a provided dataset and save the trained weights ' .h ' files in the logs category. These can then be retrieved by using model.last() function.
- Inference: which will load pre-trained weights from either the logs folder, retrieved through model.last() function, or from COCO_MODEL_PATH.

Training: This retrains the Mask R-CNN on a provided dataset. The dataset must have two parts:

- Training Dataset
- Validation Dataset

Both of these datasets must contain the RGB images and their coco instances (more on this mentioned in the Mask R-CNN retraining section)
The training can be initialized with 'coco' or 'last' where 'coco' starts training by using cooc_mask_rcnn.h file as a starting point for weights and the 'last' starts training by initializing the last log of our model.

Inference: We load the required configurations to our model.

```
class InferenceConfig(coco.CocoConfig):
    # Set batch size to 1 since we'll be running inference on
    # one image at a time. Batch size = GPU_COUNT * IMAGES_PER_GPU
    GPU_COUNT = 1
    IMAGES_PER_GPU = 1

config = InferenceConfig()
```

These were also taken from coco configurations (coco.CocoConfig) and we modified it by adding our GPU count and images to be processed per GPU.

```
model = modellib.MaskRCNN(mode="inference", model_dir=MODEL_DIR, config=config)
model.load_weights(COCO_MODEL_PATH, by_name=True)
```

We create a new model of Mask R-CNN in inference mode with our specified configurations and load the pre-trained weight from coco.
After getting the image (directory = filename) to perform inference on, we obtain the results of the inference through model.detect() function. These results are then visualized using the Mask R-CNN visualizer and the resulted image saved in the app/static/images folder as 'masked_image.jpg'.

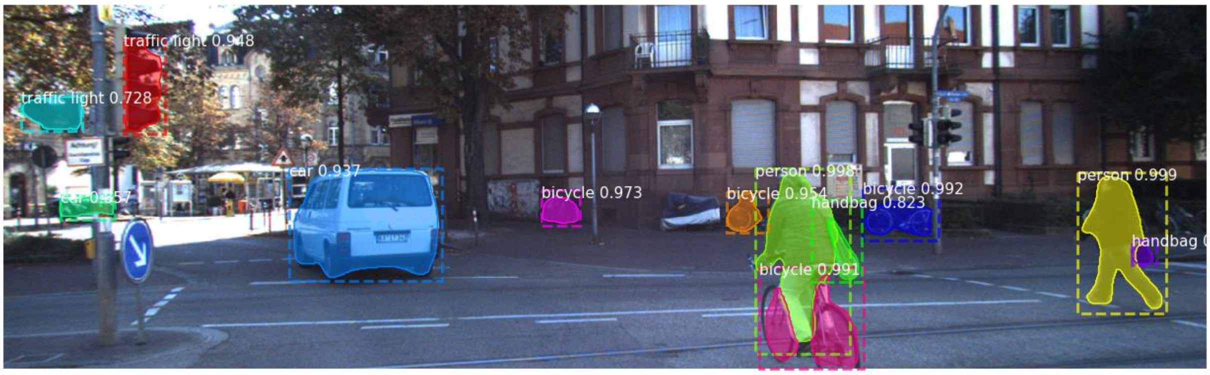


Figure 11: masked image visualized using Mask R-CNN visualizer.

Our ‘index.html’ file, which is in charge of uploading static data onto the web application, then uploads this masked_image.jpg on the web application under Mask_RCNN labels through the following code:

```
<div id="flip">Mask RCNN Labels</div>
<div id="panel">
    <img src="/static/images/masked_image.jpg" alt = "mask" width="20%" height="auto">
</div>
```

The result is this image on the web app:

Figure 12: Masked image under Mask RCNN Labels button on web page

Calibration: The next step is to calibrate these masks onto the point cloud for masked clusters. For this we refer to the calib.py file in the ROOT_DIR (directory of mask_rcnn_demo.py) and pass it the app/classify/calib directory as argument.

The app/classify/calib directory contains the calibration data for transformations between co-ordinate frames. These transformations are from

- Velodyne to camera
- Camera to Camera
- IMU to Camera

We use calib.velo2img() function to transform veodyne point cloud co-ordinates to image co-ordinates called ‘im_coord’.

```
bin_name = os.path.join(DATA_DIR, drivename, "bin_data", fname) + ".bin"
scan = np.fromfile(
    os.path.join(bin_name),
    dtype=np.float32).reshape((-1, 4))
im_coord = calib.velo2img(scan[:, :3], 2).astype(np.int)
```

Where scan is our velodyne point cloud data, reshaped from flattened numpy array to 4 columns, x, y, z, intensity, and unknown no. of rows.

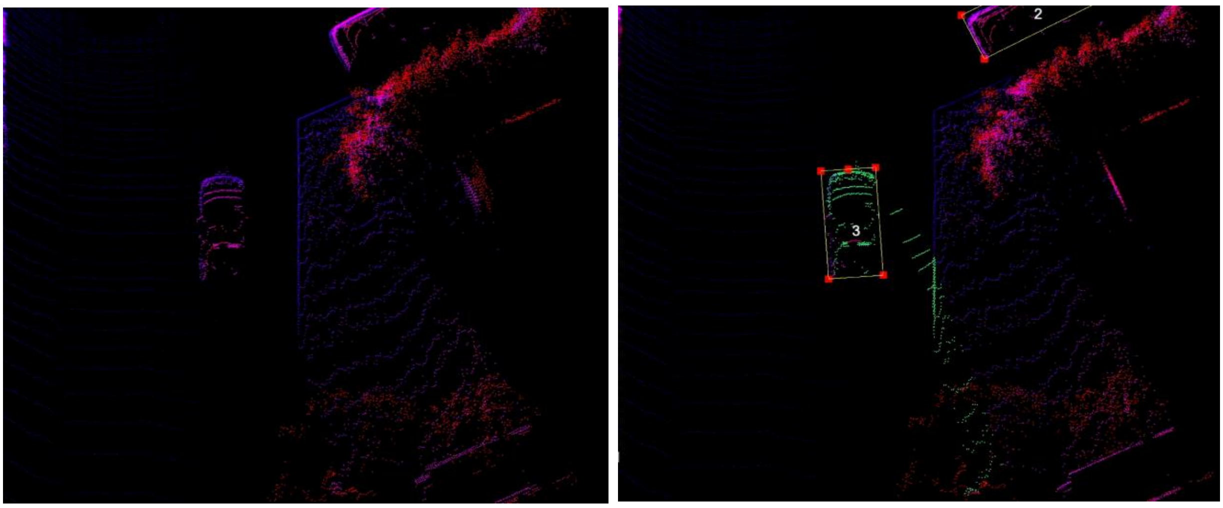
Some of these co-ordinates will be out of the range of actual corresponding image called ‘im’. So we get the width ‘w’ and height ‘h’ of our original image ‘im’ and delete any points in im_coord not contained within these dimensions.

```
scan2 = np.array([0],dtype = np.float32)
visible_indices = np.array([0],dtype = np.int)
for i in range(len(im_coord)):
    if im_coord[i][0] > 0 and im_coord[i][0] <= w and im_coord[i][1] > 0 and im_coord[i][1] <= h:
        if scan[i][0] >= 0:
            scan2 = np.append(scan2,scan[i])
            visible_indices = np.append(visible_indices, i)
scan2 = np.delete(scan2,0)
visible_indices = np.delete(visible_indices, 0)
scan2 = scan2.reshape((int(len(scan2)/4),4))
```

Where ‘scan2’ consists of only those velodyne points that have image co-ordinates contained within the width and height of original image.
Scan has all our velodyne points and ‘visible_indices’ contains the indices (numbering) of all those data points stored in ‘scan2’in the format of an array.

Then for all the mask indices we got from Mask R-CNN inference (use ‘contour’), we put the respective masks onto the velodyne point cloud, using the visible_indices.
In the end we have a an integer array of bounded_indices which is saved to app/output directory as indices.bin.

The get_mask_rcnn.py file responsible for executing the Mask_RCNN_demo.py then reads the bounded_indices from app/output/indice.bin and returns those to the app.py file which can then upload them onto the web-application.



(a) (b)

Figure 13: (a) Before calibration (b) After calibration.

5.3 Predict_label.py

This the python file which is responsible for correctly predicting the labels/object ID’s of our objects on point cloud and showing an RGB image of that object cropped out from the original image.

Directories	
CUR_DIR	app
DATA_DIR	app/test_dataset
bounding_box_filename	App/classify/bounding_boxes/fname.json
image_filename	App/test_dataset/drive/images/fname.png
Classifier2.py	
Function	babadook()
imports	Convert3d_2d.py

Convert3d_2d.py

Function	Generate_2d_lidar()
Imports	Get_coord.py
Attributes	*X: array of x co-ords of center of all B-Boxes*
	Y: array of y co-ords of center of all B-Boxes
	Width: array of width of all B-Boxes
	Length: array of length of all B-Boxes
	Angle: array of angle of all B-Boxes
	Image_output_path: path to save cropped-img.(classify/inception)
	Image_path: list of paths to all cropped images

Table 3: Arguments used by predict_label.py

It does the by first running ImageNet inference on the RGB image and cropping out objects and predicting their labels. It then translates these labels onto the point cloud and gives us our **pre-selected** object ID's on our web application as part of our Automated Annotation of LiDAR data.

It starts off by importing the necessary libraries and python files. One of these python files is the classifier.py file stored in app/classify directory.
There are two classifiers, the classifier.py and the classifier2.py. The classifier2.py is an updated version of the classifier more suited to working on our web application. Classifier.py had some errors which were removed in this new classifier2.py version.
So we import classifier2.py and its main function called 'babadook'.

The predict_abel.py takes json_data (all the data regarding the current point cloud in .json format) and the filename (format: drive/fname.bin) as input arguments.
And splits the filename at ' / ' and ' . ' to get 'drive' and 'fname'.

It creates a json file in app/classify/bounding_boxes/fname.json, which will then be passed the 'json_data' to hold. These are temporary and are then later on cleared in this python file.
It then calls the babadook() function and passes the 'image_filename' which has the complete path to our image
Aim: Get cropped images of bounding boxes in babadook() so classifier can run inference on them and predict labels for each image.
Steps carried out to predict labels:
Step 1: Get the center co-ordinates (x, y), width, height, angle of all the bounding boxes on our velodyne point cloud. This information is stored in bounding_box_filename. This step is done when babadook calls convert3d_2d.py. Convert3d_2d.py then further calls get_coord.py to retrieve this information from bounding_box_filename.

Step 2: Calibrate these co-ordinates of bounding boxes from velodyne to image co-ordinates using the calib.py file.

Step 3: After calibration crop out the object in bounding box from image and store the cropped image in the path defined by image_output_path (app/classify/inception) as a numbered file. Append the path to this image in image_path as app/classify/inception/1.jpg.

Step 4: Do this for all the bounding boxes of our current point cloud and keep appending the image_path so it is a list of paths to cropped images.

Step 5: This image_path is then returned to the babadook function which calls inference on all the cropped images whose path are stored in image_path.

Step 6: During inference of a cropped image, the image score and predicted label are stored in classify/write_data.txt which then get appended for all cropped images.

After successfully storing object scores and bounding box labels, our app.py file then uploads these labels onto the web application as object ID's of corresponding objects. When selecting a bounding box, the corresponding cropped image is uploaded onto the web app. This is done by exporting that particular image to app/static/images as cropped_image.png and index.html then shows it under object from image.

5.3.1 Results

One of the bounding boxes was around a cyclist. It retrieved the corresponding image within the bounding box and produced the 'cyclist' label

Figure 14: Cropped image

6 RE-TRAINING OF MASK R-CNN

6.1 Introduction

Mask R-CNN pre-trained on MS COCO dataset was used initially. Pre-trained weights for MS COCO are provided in MASK R-CNN Github Repository. It has training and evaluation code for MS COCO provided within.
But for Pakistani dataset with different and rather unique vehicles Mask R-CNN would need to be re-trained on custom dataset.
For that purpose we downloaded pictures of Pakistani traffic, used those for training and testing of the model. And also created synthetic dataset to have sufficient data for training.

6.2 Dataset Creation

We used subsequent frames and generated colored masks for objects manually using GIMP, as it's a free open source tool.

Figure 15: 640x480 pixels training image

Figure 16: Colored masked generated by GIMP

Definitions for these masks and some high level info is written in mask_definitions.json and dataset_info.json respectively.

6.2.1 Mask Definitions

Mask definitions are written in the following format:


```
"super_categories":
 {
    "vehicles": ["car" , "van" , "rickshaw" ],
    "trucks" : ["big_truck"],
    "bikes": ["motorbike"],
    "people" : ["pedestrian"]
 }


{
"masks":
        {
                "images/00000000.png":
                        {"mask": "masks/00000000.png",
                        "color_categories": {"(255, 0, 0)": {"category": "big_truck",
                        "super_category": "trucks"}, "(0, 255, 0)": {"category": "rickshaw",
                        "super_category": "vehicles"}
                        }
        }
```


Where a category and super category is defined for every colour in mask image.

6.2.2 Dataset info

High level information like contributor name, creation date and license etc. is provided in this JSON file as:


```
{
"info":
 {"description": "Road Training-dataset",
  "url": "http://RAMfyp.com/road/datasets/train",
  "version": "1",
  "contributor": "Rabeea Jawaid, Amal and Minhah Saleem",
  "year": 2020,
  "date_created": "07/21/2020"
 },

 "license":
  {
   "id": 0,
   "name": "Train License",
   "url": http://RAMfyp.com/licenses/train
  }
}
```


6.2.3 COCO Instances

coco_json_utils.py takes masks definitions and dataset info as input and as output provides coco_instances.json as:


```
{"info": {"description": "Road Training-dataset", "url": "http://RAMfyp.com/road/datasets/train",
"version": "1", "year": 2020, "contributor": "Rabeea Jawaid, Amal and Minhah Saleem",
"date_created": "07/21/2020"}, "licenses": [{"url": "http://RAMfyp.com/licenses/train", "id": 0, "name":
"Train License"}], "images": [{"license": 0, "file_name": "00000000.png", "width": 640, "height": 480,
"id": 0}, {"license": 0, "file_name": "00000001.png", "width": 640, "height": 480, "id": 1},….
…… {"segmentation": [[502.0, 392.5, 513.0, 391.5, 514.0, 382.5, 587.5, 382.0, 583.5, 365.0, 584.5, 336.0,
582.0, 294.5, 520.0, 293.5, 509.0, 292.5, 505.0, 288.5, 498.0, 295.5, 482.0, 297.5, 480.5, 288.0, 479.0, 286.5,
470.0, 286.5, 466.0, 280.5, 446.0, 280.5, 439.0, 287.5, 428.5, 290.0, 430.5, 315.0, 425.5, 324.0, 425.5, 348.0,
430.5, 353.0, 430.5, 376.0, 442.0, 380.5, 455.0, 380.5, 460.5, 379.0, 461.0, 372.5, 471.0, 371.5, 478.0, 388.5,
```

495.0, 392.5, 502.0, 392.5]], "iscrowd": 0, "image_id": 2005, "category_id": 4, "id": 4060, "bbox": [425.5, 280.5, 162.0, 112.0], "area": 14348.25}],…………
…….
"categories":
[{"supercategory": "vehicles", "id": 1, "name": "car"},
{"supercategory": "vehicles", "id": 2, "name": "van"},
{"supercategory": "vehicles", "id": 3, "name": "rickshaw"},
{"supercategory": "trucks", "id": 4, "name": "big_truck"},
{"supercategory": "bikes", "id": 5, "name": "motorbike"},
{"supercategory": "people", "id": 6, "name": "pedestrian"}]
}

Provides pixel values for masks and bounding boxes.
Now the dataset is in MS COCO format and can be used to train Mask R-CNN.

6.3 Synthetic Dataset

Synthetic images were created by putting different foregrounds on backgrounds with different variations like scaling and rotation etc.

(a) (b) (c)

Figure 17: (a) foreground cutout of a rickshaw, (b) background, (c) foreground placed on top of background with rotation.

These synthetic images don't make sense and seem a little absurd to human eye but serve pretty well as training images for our model.
Since we already know the pixels of foreground in this case, instead of making masks manually, we use python script to generate masks and coco instances JSON file.

6.3.1 Image composition script file

This script file takes foregrounds and backgrounds as input and returns images, their masks, mask_definitions.json and dataset_info.json files. These JSON files are then converted into coco_instances.json.
It consists of two classes:

6.3.1.1 Image Composition Class

It applies transformations on foregrounds and creates synthetic combined images. It also creates segmentation masks.
It applies following three transformations on images:

- Rotation
- Scaling
- Brightness change

6.3.1.2 Mask JSON Utils Class

This class is called by image composition class and is responsible for creating JSON definition file. It has following functions:

- Add category
- Add mask
- Get mask
- Get super-category
- Write masks to JSON

6.4 Training Mask R-CNN

We created around two thousand training images and two fifty validation images for detection of rickshaws and trucks.

6.4.1 Training Configurations

BACKBONE resnet50
BACKBONE_STRIDES [4, 8, 16, 32, 64]
BATCH_SIZE 1
BBOX_STD_DEV [0.1 0.1 0.2 0.2]
COMPUTE_BACKBONE_SHAPE None
DETECTION_MAX_INSTANCES 100
DETECTION_MIN_CONFIDENCE 0.7
DETECTION_NMS_THRESHOLD 0.3
FPN_CLASSIF_FC_LAYERS_SIZE 1024
GPU_COUNT 1
GRADIENT_CLIP_NORM 5.0
IMAGES_PER_GPU 1
IMAGE_CHANNEL_COUNT 3
IMAGE_MAX_DIM 640
IMAGE_META_SIZE 19
IMAGE_MIN_DIM 480
IMAGE_MIN_SCALE 0
IMAGE_RESIZE_MODE square
IMAGE_SHAPE [640 640 3]
LEARNING_MOMENTUM 0.9
LEARNING_RATE 0.001
LOSS_WEIGHTS {'rpn_class_loss': 1.0, 'rpn_bbox_loss': 1.0, 'mrcnn_class_loss': 1.0, 'mrcnn_bbox_loss': 1.0, 'mrcnn_mask_loss': 1.0}
MASK_POOL_SIZE 14
MASK_SHAPE [28, 28]
MAX_GT_INSTANCES 50
MEAN_PIXEL [123.7 116.8 103.9]
MINI_MASK_SHAPE (56, 56)
NAME cocosynth_dataset
NUM_CLASSES 7
POOL_SIZE 7
POST_NMS_ROIS_INFERENCE 500
POST_NMS_ROIS_TRAINING 1000
PRE_NMS_LIMIT 6000
ROI_POSITIVE_RATIO 0.33
RPN_ANCHOR_RATIOS [0.5, 1, 2]
RPN_ANCHOR_SCALES (8, 16, 32, 64, 128)
RPN_ANCHOR_STRIDE 1

RPN_BBOX_STD_DEV	[0.1 0.1 0.2 0.2]
RPN_NMS_THRESHOLD	0.7
RPN_TRAIN_ANCHORS_PER_IMAGE	256
STEPS_PER_EPOCH	1000
TOP_DOWN_PYRAMID_SIZE	256
TRAIN_BN	False
TRAIN_ROIS_PER_IMAGE	32
USE_MINI_MASK	True
USE_RPN_ROIS	True
VALIDATION_STEPS	5
WEIGHT_DECAY	0.0001

6.4.2 Inference Configurations

GPU_COUNT = 1
IMAGES_PER_GPU = 1
IMAGE_MIN_DIM = 720
IMAGE_MAX_DIM = 1280
DETECTION_MIN_CONFIDENCE = 0.85

6.4.3 Hardware

System used for re-training had:

- GEFORCE GTX 1050
- Driver version 451.77

6.4.4 Software

Following software combination worked for us:

- Cuda toolkit 10.1
- CuDNN 7.6.4
- Nvidia Standard driver for GEFORCE GTX 1050
- Tensorflow 2.2.0
- Tensorflow-estimator 2.2.0
- Tensorflow-gpu 2.2.0
- tensorflow-gpu-estimator 2.2.0

6.5 Results

Upon running inference in Train_mask_rcnn on Jupyter Notebook where the thereshold for detection confidence is set to 85%, training took 50.85 minutes and we got results similar to

Fig. 18.
Figure 18: predicted mask and bounding box for rickshaw with a score of 0.918.

Precision = 0.5714285714285714

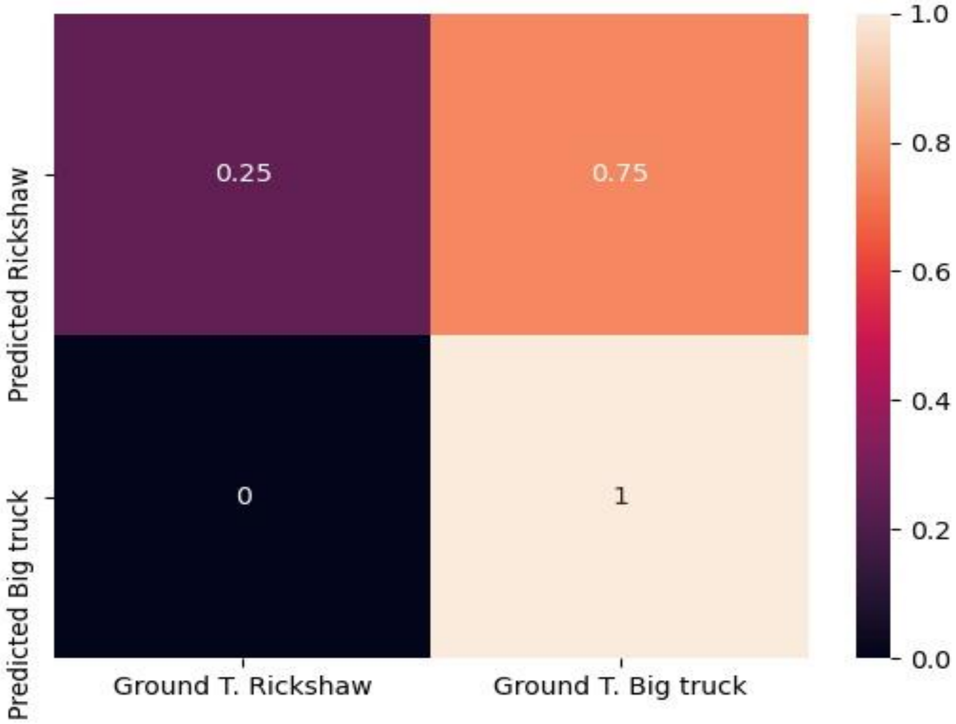


Figure 19: Confusion Matrix

6.5.1 Loss Function

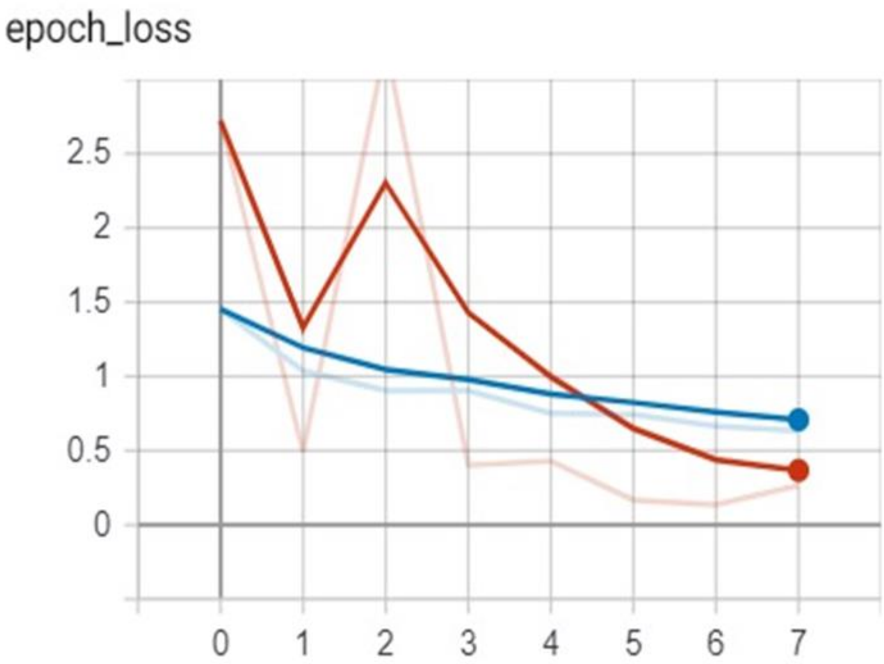


Figure 20: Loss function

Where red is for training data and blue is for validation.

6.6 Conclusion

The results are not as good as in the case of pre-trained Mask R-CNN because of lesser dataset used for re-training and because it is re-trained for only rickshaws and trucks and not any other vehicle type. So we observed it sometimes identified other vehicles as rickshaws as well and made mistakes which can be improved by further training. But for objects, it is trained for Mask R-CNN gave fairly good results.

7 Conclusion

The One-Click annotation tool was successfully made more efficient, giving us annotated images along with segmentation masks which were then used to calibrate LiDAR point cloud. We re-trained Mask R-CNN for two types of Pakistani vehicles and got satisfactory results given our dataset size. It can be trained for more data and we hope it will eventually work for Pakistani traffic as it works for foreign data.

REFERENCES

[1] B. Wu, A. Wan, X. Yue, and K. Keutzer, "Squeezeseg: Convolutional neural nets with recurrent crf for real-time road-object segmentation from 3d lidar point cloud," in 2018 IEEE International Conference onRobotics and Automation (ICRA). IEEE, 2018, pp. 188.

[2] B. Li, T. Zhang, and T. Xia, "Vehicle detection from 3d lidar using fully convolutional network," arXiv preprint arXiv:1608.07916, 2016.

[3] S. Lambert and . E. Granath, "LiDAR systems: costs, integration, and major manufacturers," 05 03 2020. [Online]. Available: https://www.mes-insights.com/lidar-systems-costs-integration-and-major-manufacturers-a-908358/?cmp=go-aw-art-trf-MES_DSA-20200217&gclid=EAIaIQobChMImJuz-cjh6gIV1ojVCh0-igxtEAAYASAAEgLbRvD_BwE. [Accessed 23 07 2020].

[4] N. Wijeyasinghe and K. Ghaffarzadeh, "Lidar 2020-2030: Technologies, Players, Markets & Forecasts," 14 08 2019. [Online]. Available: https://www.idtechex.com/en/research-report/lidar-2020-2030-technologies-players-markets-and-forecasts/694. [Accessed 23 07 2020].

[5] K. He, G. Gkioxari, P. Dollár and R. Girshick, "Mask r-cnn," *Proceedings of the IEEE international conference on computer vision,* pp. 2961-2969, 2017.

[6] J. Redmon, S. Divvala, . R. Girshick and A. Farhadi, "You only look once: Unified, real-time object detection," *Proceedings of the IEEE conference on computer vision and pattern recognition,* pp. 779-788, 2016.

[7] M. Himmelsbach, A. Mueller, T. Luttel, and H.-J. Wunsche, "Lidar-based 3d object perception," in Proceedings of 1st international workshop on cognition for technical systems, vol. 1, 2008.

[8] Zhou, Yin, and Oncel Tuzel. "Voxelnet: End-to-end learning for point cloud based 3d object detection." Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. 2018.

[9] Charles Ruizhongtai Qi, Hao Su, Kaichun Mo, and Leonidas J. Guibas. "PointNet: Deep Learning on Point Sets for 3D Classification and Segmentation". In: CoRR abs/1612.00593 (2016).

[10] A. Geiger, P. Lenz, and R. Urtasun, "Are we ready for autonomous driving? the kitti vision benchmark suite," in Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on. IEEE, 2012, pp. 3354–3361.

[11] X. Huang, X. Cheng, Q. Geng, B. Cao, D. Zhou, P. Wang, Y. Lin, and R. Yang, "The apolloscape dataset for autonomous driving," in Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops, 2018, pp. 954–960.

[12] L. Castrejon, K. Kundu, R. Urtasun, and S. Fidler, "Annotating object instances with a polygon-rnn," in Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2017, pp. 5230–5238.

[13] D. Acuna, H. Ling, A. Kar, and S. Fidler, "Efficient interactive annotation of segmentation datasets with polygon-rnn++," 2018.

[14] C. Vondrick, D. Patterson, and D. Ramanan, "Efficiently scaling up crowdsourced video annotation," International Journal of Computer Vision, pp. 1–21, 10.1007/s11263-012-0564-1. [Online]. Available: http://dx.doi.org/10.1007/s11263-012-0564-1.

[15] B. Wang, V. Wu, B. Wu and K. Keutzer, "LATTE: Accelerating LiDAR Point Cloud Annotation via Sensor Fusion, One-Click Annotation, and Tracking," *2019 IEEE Intelligent Transportation Systems Conference (ITSC)* , pp. 265-272, 2019.

[16] F. Ma, G. V. Cavalheiro and S. Karaman, "Self-Supervised Sparse-to-Dense: Self-Supervised Depth Completion from LiDAR and Monocular Camera," *2019 International Conference on Robotics and Automation (ICRA)*, pp. 3288-3295, 2019.

PACKAGE REQUIREMENTS

1. absl-py==0.7.1
2. astor==0.7.1
3. Click==7.0
4. cycler==0.10.0
5. decorator==4.4.0
6. Flask==1.0.2
7. gast==0.2.2
8. grpcio==1.19.0
9. h5py==2.9.0
10. imageio==2.5.0
11. itsdangerous==1.1.0
12. Jinja2==2.10.1
13. Keras==2.2.4
14. Keras-Applications==1.0.7
15. Keras-Preprocessing==1.0.9
16. kiwisolver==1.0.1
17. Markdown==3.1
18. MarkupSafe==1.1.1
19. matplotlib==3.0.3
20. mock==2.0.0
21. networkx==2.2
22. numpy==1.16.2
23. pbr==5.1.3
24. Pillow==6.0.0
25. protobuf==3.7.1
26. pyparsing==2.4.0
27. python-dateutil==2.8.0
28. PyWavelets==1.0.2
29. PyYAML==5.1
30. scikit-image==0.15.0
31. scipy==1.2.1
32. six==1.12.0
33. tensorboard==1.13.1
34. tensorflow==1.13.1 **(Latte app) , tensorflow==2.2 (Re-training of Mask R-CNN)**
35. tensorflow-estimator==1.13.0 (Latte app), tensorflow-estimator==2.2 (Re-training of Mask R-CNN)
36. **tensorflow-gpu==2.2** (For re-training Mask R-CNN only)
37. **tensorflow-gpu-estimator==2.2** (For re-training Mask R-CNN only)
38. termcolor==1.1.0
39. Werkzeug==0.15.2

CODE

App.py

```
from flask import Flask, render_template, request, jsonify
from models import BoundingBox
from pointcloud import PointCloud
from predict_label import predict_label
from mask_rcnn import get_mask_rcnn_labels
from frame_handler import FrameHandler
from bounding_box_predictor import BoundingBoxPredictor
import numpy as np
import json
import os
from tracker import Tracker
from pathlib import Path

app = Flask(__name__, static_url_path='/static')
DIR_PATH = os.path.dirname(os.path.realpath(__file__))

@app.route("/")
def root():
   return render_template("index.html")

@app.route("/initTracker", methods=["POST"])
def init_tracker():
   json_request = request.get_json()
   pointcloud = PointCloud.parse_json(json_request["pointcloud"])
   tracker = Tracker(pointcloud)
   return "success"

@app.route("/trackBoundingBoxes", methods=['POST'])
def trackBoundingBox():
   json_request = request.get_json()
   pointcloud = PointCloud.parse_json(json_request["pointcloud"],
json_request["intensities"])
   filtered_indices = tracker.filter_pointcloud(pointcloud)
   next_bounding_boxes = tracker.predict_bounding_boxes(pointcloud)
   print(next_bounding_boxes)
   return str([filtered_indices, next_bounding_boxes])


@app.route("/updateBoundingBoxes", methods=['POST'])
def updateBoundingBoxes():
   json_request = request.get_json()

   bounding_boxes =
BoundingBox.parse_json(json_request["bounding_boxes"])
   tracker.set_bounding_boxes(bounding_boxes)
   return str(bounding_boxes)
```

```
@app.route("/predictLabel", methods=['POST'])
def predictLabel():
   json_request = request.get_json()
   json_data = json.dumps(json_request)
   filename = json_request['filename'].split('.')[0]
   os.system("rd {}/*".format(os.path.join(DIR_PATH,
"static/images")))
   predicted_label = predict_label(json_data, filename)
   in_fov = os.path.exists(os.path.join(DIR_PATH,
"static/images/cropped_image.jpg"))
   return ",".join([str(predicted_label), str(in_fov)])


@app.route("/getMaskRCNNLabels", methods=['POST'])
def getMaskRCNNLabels():
   filename = request.get_json()['fname']
   get_mask_rcnn_labels(filename)
   return str(get_mask_rcnn_labels(filename))
   #return 0

@app.route("/writeOutput", methods=['POST'])
def writeOutput():
   frame = request.get_json()['output']
   f_name = frame['filename']
   drivename, fname = f_name.split('/')
   fh.save_annotation(drivename, fname, frame["file"])

   json_data = frame["file"]
   os.system("rd {}/*".format(os.path.join(DIR_PATH,
"static/images")))
   return str("hi")


@app.route("/loadFrameNames", methods=['POST'])
def loadFrameNames():
   return fh.get_frame_names()

@app.route("/getFramePointCloud", methods=['POST'])
def getFramePointCloud():
   json_request = request.get_json()
   fname = json_request["fname"]
   drivename, fname = fname.split("/")
   data_str = fh.get_pointcloud(drivename, fname, dtype=str)
   annotation_str = str(fh.load_annotation(drivename, fname,
dtype='json'))

   return '?'.join([data_str, annotation_str])

@app.route("/predictBoundingBox", methods=['POST'])
def predictBoundingBox():
   json_request = request.get_json()
```

```
   filename = json_request["fname"]
   drivename, fname = filename.split("/")
   point = json_request["point"]
   point = np.array([point['z'], point['x'], point['y']])
   frame = fh.get_pointcloud(drivename, fname, dtype=float,
ground_removed=True)
   return str(bp.predict_bounding_box(point, frame))

@app.route("/predictNextFrameBoundingBoxes", methods=['POST'])
def predictNextFrameBoundingBoxes():
   json_request = request.get_json()

   fname = json_request["fname"]
   drivename, fname = fname.split("/")
   frame = fh.load_annotation(drivename, fname)
   res = bp.predict_next_frame_bounding_boxes(frame)
   keys = res.keys()
   for key in keys:
      res[str(key)] = res.pop(key)
   print(res)

   return str(res)

@app.route("/loadAnnotation", methods=['POST'])
def loadAnnotation():
   json_request = request.get_json()
   fname = json_request["fname"]
   frame = fh.load_annotation(fname)
   return str(frame.bounding_boxes)


if __name__ == "__main__":
   fh = FrameHandler()
   bp = BoundingBoxPredictor(fh)
   os.system("rd {}/*".format(os.path.join(DIR_PATH,
"static/images")))
   app.run()
```

Predict_label.py

```
import os
import json
import glob
from classify.classifier2 import babadook

CUR_DIR = os.path.dirname(os.path.realpath(__file__))
DATA_DIR = os.path.join(CUR_DIR, "test_dataset")
IMAGE_DIR = os.path.join(DATA_DIR)

def predict_label(json_data, filename):
  drivename, fname = filename.split("/")
  df = drivename + '_' + fname
```

```
  fname = fname.split(".")[0]
  bounding_box_path = os.path.join("classify/bounding_boxes",(df +'.json'))
  bounding_box_filename = os.path.join(CUR_DIR, bounding_box_path)
  output_path = os.path.join(CUR_DIR, "classify/write_data.txt")
  image_filename = os.path.join(DATA_DIR, drivename, "image", fname+'.png')

  images_to_delete = os.path.join(CUR_DIR, "classify/inception/*jpg")
  all_files = glob.glob(images_to_delete)
  for f in all_files:
    os.remove(f)

  try:
    open(bounding_box_filename, 'w').close()
  except Exception as e:
    pass
  with open(bounding_box_filename,'a') as f:
    f.write(json_data)
  babadook(df)
  folder_to_empty = os.path.join(CUR_DIR, "classify/bounding_boxes/*json")
  files = glob.glob(folder_to_empty)
  for f in files:
    os.remove(f)
  f = open(output_path, "r")
  data = f.readlines()

  os.system("rd classify/bounding_boxes/*.json")
  hi = ''
  l = open(os.path.join(CUR_DIR, 'keywords.txt'), "w")

  for x in data:
    hi = get_keyword(x)
    l.write(str(hi) + '\n')
  f.close()
  l.close()
  return hi

def get_keyword(data):
  pedestrian_keywords = {'person', 'man', 'woman', 'walker', 'pedestrian'}
  car_keywords = {'car'}
  van_keywords = {'van', 'minivan', 'bus', 'minibus'}
  truck_keywords = {'truck'}
  cyclist_keywords = {'cyclist', 'motorcyclist', 'unicyclist', 'bicycle', 'motocycle',
            'bike', 'motorbike', 'unicycle', 'monocycle', 'rickshaw'}

  words = []
  for w in data.split(','):
    words.extend(w.split(' '))
  words = set(words)
  if words.intersection(car_keywords):
    return 'car'
  if words.intersection(van_keywords):
    return 'van'
```

```
  if words.intersection(truck_keywords):
    return 'truck'
  if words.intersection(pedestrian_keywords):
    return 'pedestrian'
  if words.intersection(cyclist_keywords):
    return 'cyclist'
  return -1
```

Mask_rcnn_demo.py

```
import matplotlib
matplotlib.use('TkAgg')
import os
import sys
import random
import math
import numpy as np
import skimage.io
import matplotlib
import matplotlib.pyplot as plt

from . import coco
from . import utils
from . import model as modellib
from . import visualize

import PIL
from PIL import Image
from .calib import Calib
from time import time

from . import chooseFile
import argparse
import matplotlib.path as mpltPath
def vot(data_filename):
    ROOT_DIR = os.path.dirname(os.path.realpath(__file__))
    print(ROOT_DIR)
    MODEL_DIR = os.path.join(ROOT_DIR, "logs")
    PARENT_DIR = os.path.abspath(os.path.join(ROOT_DIR, os.pardir))
    DATA_DIR = os.path.join(PARENT_DIR, "test_dataset")
    COCO_MODEL_PATH =  os.path.join(ROOT_DIR, "mask_rcnn_coco.h5")

    class InferenceConfig(coco.CocoConfig):
        GPU_COUNT = 1
        IMAGES_PER_GPU = 1

    config = InferenceConfig()
    model = modellib.MaskRCNN(mode="inference", model_dir=MODEL_DIR,
config=config)
    model.load_weights(COCO_MODEL_PATH, by_name=True)
    class_names = ['BG', 'person', 'bicycle', 'car', 'motorcycle', 'airplane',
```

```
               'bus', 'train', 'truck', 'boat', 'traffic light',
               'fire hydrant', 'stop sign', 'parking meter', 'bench', 'bird',
               'cat', 'dog', 'horse', 'sheep', 'cow', 'elephant', 'bear',
               'zebra', 'giraffe', 'backpack', 'umbrella', 'handbag', 'tie',
               'suitcase', 'frisbee', 'skis', 'snowboard', 'sports ball',
               'kite', 'baseball bat', 'baseball glove', 'skateboard',
               'surfboard', 'tennis racket', 'bottle', 'wine glass', 'cup',
               'fork', 'knife', 'spoon', 'bowl', 'banana', 'apple',
               'sandwich', 'orange', 'broccoli', 'carrot', 'hot dog', 'pizza',
               'donut', 'cake', 'chair', 'couch', 'potted plant', 'bed',
               'dining table', 'toilet', 'tv', 'laptop', 'mouse', 'remote',
               'keyboard', 'cell phone', 'microwave', 'oven', 'toaster',
               'sink', 'refrigerator', 'book', 'clock', 'vase', 'scissors',
               'teddy bear', 'hair drier', 'toothbrush']

   drivename, fname = data_filename.split("/")
   filename = os.path.join(DATA_DIR, drivename, "image", fname) + ".png"
   image = Image.open(filename)
   start = time()
   img = skimage.io.imread(filename)
   results = model.detect([img], verbose=1)
   print(type(img))
   r = results[0]
   print("identify image time: " + str(time()-start))
   contour = visualize.display_instances(img, r['rois'], r['masks'], r['class_ids'], class_names,
r['scores'])
   print(r['class_ids'])
   calib_dir = os.path.join(PARENT_DIR, "classify/calib")
   calib = Calib(calib_dir)
   #calib = Calib('C:/Rabeea/UNI/STUDY/fyp/latte/app/classify/calib')
   im = PIL.Image.open(os.path.join(filename))
   w, h = im.size
   bin_name = os.path.join(DATA_DIR, drivename, "bin_data", fname) + ".bin"
   scan = np.fromfile(
      os.path.join(bin_name),
      dtype=np.float32).reshape((-1, 4))
   im_coord = calib.velo2img(scan[:, :3], 2).astype(np.int)
   im_coord2 = [im_coord[i] for i in range(len(im_coord)) if im_coord[i][0] > 0 and
im_coord[i][0] <= w and im_coord[i][1] > 0 and im_coord[i][1] <= h and scan[i][0]>=0]

   scan2 = np.array([0],dtype = np.float32)
   visible_indices = np.array([0],dtype = np.int)
   for i in range(len(im_coord)):
      if im_coord[i][0] > 0 and im_coord[i][0] <= w and im_coord[i][1] > 0 and
im_coord[i][1] <= h:
         if scan[i][0] >= 0:
            scan2 = np.append(scan2,scan[i])
            visible_indices = np.append(visible_indices, i)
   scan2 = np.delete(scan2,0)
   visible_indices = np.delete(visible_indices, 0)
   scan2 = scan2.reshape((int(len(scan2)/4),4))
   start_time = time()
```

```
    print(len(contour))
    bounded_indices = np.array([0], dtype=np.int)
    for i in range(len(contour)):
        for q in contour[i][0]:
            temp = q[0]
            q[0] = q[1]
            q[1] = temp
        polygon = [q for q in contour[i][0]]
        path = mpltPath.Path(polygon)
        inside2 = path.contains_points(im_coord2)
        scan3 = np.array([0],dtype = np.float32)

        for j in range(len(inside2)):
            if inside2[j] ==True:
                scan3 = np.append(scan3,scan2[j])
                bounded_indices = np.append(bounded_indices, visible_indices[j])
        scan3 = np.delete(scan3,0)
        scan3 = scan3.reshape((int(len(scan3)/4),4))
        class_id = r['class_ids'][i]
        label = class_names[class_id]

        if len(scan3) != 0:
            pass
        class_id = r['class_ids'][i]
        label = class_names[class_id]

    bounded_indices = np.delete(bounded_indices, 0)
    bounded_indices.tofile(os.path.join(PARENT_DIR, "output/indices.bin"))
    print("Matplotlib contains_points Elapsed time: " + str(time()-start_time))
```

Re-training of Mask R-CNN

```
import os
import sys
import json
import numpy as np
import time
from PIL import Image, ImageDraw
from pathlib import Path


ROOT_DIR = '../../Mask_RCNN-master/'
assert os.path.exists(ROOT_DIR), 'ROOT_DIR does not exist. Did you forget to read the
instructions above? ;)'

sys.path.append(ROOT_DIR)
from mrcnn.config import Config
import mrcnn.utils as utils
from mrcnn import visualize
import mrcnn.model as modellib
```

```
MODEL_DIR = os.path.join(ROOT_DIR, "logs")
COCO_MODEL_PATH = os.path.join(ROOT_DIR, "mask_rcnn_coco.h5")

if not os.path.exists(COCO_MODEL_PATH):
    utils.download_trained_weights(COCO_MODEL_PATH)

class CocoSynthConfig(Config):
    NAME = "cocosynth_dataset"
    GPU_COUNT = 1
    IMAGES_PER_GPU = 1
    NUM_CLASSES = 1 + 6  # background + 6 road objects types
    IMAGE_MIN_DIM = 480
    IMAGE_MAX_DIM = 640
    STEPS_PER_EPOCH = 1000
    VALIDATION_STEPS = 5
    BACKBONE = 'resnet50'
    RPN_ANCHOR_SCALES = (8, 16, 32, 64, 128)
    TRAIN_ROIS_PER_IMAGE = 32
    MAX_GT_INSTANCES = 50
    POST_NMS_ROIS_INFERENCE = 500
    POST_NMS_ROIS_TRAINING = 1000

config = CocoSynthConfig()
config.display()


class CocoLikeDataset(utils.Dataset):
    def load_data(self, annotation_json, images_dir):
        json_file = open(annotation_json)
        coco_json = json.load(json_file)
        json_file.close()
        source_name = "coco_like"
        for category in coco_json['categories']:
            class_id = category['id']
            class_name = category['name']
            if class_id < 1:
                print('Error: Class id for "{}" cannot be less than one. (0 is reserved for the
background)'.format(class_name))
                return

            self.add_class(source_name, class_id, class_name)

        annotations = {}
        for annotation in coco_json['annotations']:
            image_id = annotation['image_id']
            if image_id not in annotations:
                annotations[image_id] = []
            annotations[image_id].append(annotation)

        seen_images = {}
        for image in coco_json['images']:
```

```
      image_id = image['id']
      if image_id in seen_images:
        print("Warning: Skipping duplicate image id: {}".format(image))
      else:
        seen_images[image_id] = image
        try:
          image_file_name = image['file_name']
          image_width = image['width']
          image_height = image['height']
        except KeyError as key:
          print("Warning: Skipping image (id: {}) with missing key:
{}".format(image_id, key))

        image_path = os.path.abspath(os.path.join(images_dir, image_file_name))
        image_annotations = annotations[image_id]
        self.add_image(
          source=source_name,
          image_id=image_id,
          path=image_path,
          width=image_width,
          height=image_height,
          annotations=image_annotations
        )

  def load_mask(self, image_id):
    image_info = self.image_info[image_id]
    annotations = image_info['annotations']
    instance_masks = []
    class_ids = []

    for annotation in annotations:
      class_id = annotation['category_id']
      mask = Image.new('1', (image_info['width'], image_info['height']))
      mask_draw = ImageDraw.ImageDraw(mask, '1')
      for segmentation in annotation['segmentation']:
        mask_draw.polygon(segmentation, fill=1)
        bool_array = np.array(mask) > 0
        instance_masks.append(bool_array)
        class_ids.append(class_id)

    mask = np.dstack(instance_masks)
    class_ids = np.array(class_ids, dtype=np.int32)

    return mask, class_ids
dataset_train = CocoLikeDataset()
dataset_train.load_data('../datasets/road_datasets/training/coco_instances.json',
            '../datasets/road_datasets/training/images')
dataset_train.prepare()

dataset_val = CocoLikeDataset()
dataset_val.load_data('../datasets/road_datasets/validation/coco_instances.json',
           '../datasets/road_datasets/validation/images')
```

```
dataset_val.prepare()
for name, dataset in [('training', dataset_train), ('validation', dataset_val)]:
    print(f'Displaying examples from {name} dataset:')

    image_ids = np.random.choice(dataset.image_ids, 3)
    for image_id in image_ids:
        image = dataset.load_image(image_id)
        mask, class_ids = dataset.load_mask(image_id)
        visualize.display_top_masks(image, mask, class_ids, dataset.class_names)
model = modellib.MaskRCNN(mode="training", config=config,
                          model_dir=MODEL_DIR)
init_with = "coco"  # imagenet, coco, or last

if init_with == "imagenet":
    model.load_weights(model.get_imagenet_weights(), by_name=True)
elif init_with == "coco":
    model.load_weights(COCO_MODEL_PATH, by_name=True,
                       exclude=["mrcnn_class_logits", "mrcnn_bbox_fc",
                                "mrcnn_bbox", "mrcnn_mask"])
elif init_with == "last":
    model.load_weights(model.find_last(), by_name=True)
start_train = time.time()
model.train(dataset_train, dataset_val,
            learning_rate=config.LEARNING_RATE,
            epochs=4,
            layers='heads')
end_train = time.time()
minutes = round((end_train - start_train) / 60, 2)
print(f'Training took {minutes} minutes')
start_train = time.time()
model.train(dataset_train, dataset_val,
            learning_rate=config.LEARNING_RATE / 10,
            epochs=8,
            layers="all")
end_train = time.time()
minutes = round((end_train - start_train) / 60, 2)
print(f'Training took {minutes} minutes')

class InferenceConfig(CocoSynthConfig):
    GPU_COUNT = 1
    IMAGES_PER_GPU = 1
    IMAGE_MIN_DIM = 720
    IMAGE_MAX_DIM = 1280
    DETECTION_MIN_CONFIDENCE = 0.85


inference_config = InferenceConfig()
model = modellib.MaskRCNN(mode="inference",
                          config=inference_config,
                          model_dir=MODEL_DIR)
model_path = str(Path(ROOT_DIR) / "logs" /
"box_synthetic20190328T2255/mask_rcnn_box_synthetic_0016.h5" )
```

```
model_path = model.find_last()
assert model_path != "", "Provide path to trained weights"
print("Loading weights from ", model_path)
model.load_weights(model_path, by_name=True)
import skimage
real_test_dir = '../datasets/road_datasets/testing/images'
image_paths = []
for filename in os.listdir(real_test_dir):
    if os.path.splitext(filename)[1].lower() in ['.png', '.jpg', '.jpeg']:
        image_paths.append(os.path.join(real_test_dir, filename))

for image_path in image_paths:
    img = skimage.io.imread(image_path)
    img_arr = np.array(img)
    results = model.detect([img_arr], verbose=1)
    r = results[0]
    visualize.display_instances(img, r['rois'], r['masks'], r['class_ids'],
                                dataset_train.class_names, r['scores'], figsize=(8,8))
```

Image_composition.py

```
#!/usr/bin/env python3

import json
import warnings
import random
import numpy as np
from datetime import datetime
from pathlib import Path
from tqdm import tqdm
from PIL import Image, ImageEnhance

class MaskJsonUtils():
    """ Creates a JSON definition file for image masks.
    """

    def __init__(self, output_dir):
        """ Initializes the class.
        Args:
            output_dir: the directory where the definition file will be saved
        """
        self.output_dir = output_dir
        self.masks = dict()
        self.super_categories = dict()

    def add_category(self, category, super_category):
        """ Adds a new category to the set of the corresponding super_category
        Args:
            category: e.g. 'eagle'
            super_category: e.g. 'bird'
        Returns:
            True if successful, False if the category was already in the dictionary
        """
```

```
        if not self.super_categories.get(super_category):
            # Super category doesn't exist yet, create a new set
            self.super_categories[super_category] = {category}
        elif category in self.super_categories[super_category]:
            # Category is already accounted for
            return False
        else:
            # Add the category to the existing super category set
            self.super_categories[super_category].add(category)

        return True # Addition was successful

    def add_mask(self, image_path, mask_path, color_categories):
        """ Takes an image path, its corresponding mask path, and its color categories,
            and adds it to the appropriate dictionaries
        Args:
            image_path: the relative path to the image, e.g. './images/00000001.png'
            mask_path: the relative path to the mask image, e.g. './masks/00000001.png'
            color_categories: the legend of color categories, for this particular mask,
                represented as an rgb-color keyed dictionary of category names and their super
categories.
                (the color category associations are not assumed to be consistent across images)
        Returns:
            True if successful, False if the image was already in the dictionary
        """
        if self.masks.get(image_path):
            return False # image/mask is already in the dictionary

        # Create the mask definition
        mask = {
            'mask': mask_path,
            'color_categories': color_categories
        }

        # Add the mask definition to the dictionary of masks
        self.masks[image_path] = mask

        # Regardless of color, we need to store each new category under its supercategory
        for _, item in color_categories.items():
            self.add_category(item['category'], item['super_category'])

        return True # Addition was successful

    def get_masks(self):
        """ Gets all masks that have been added
        """
        return self.masks

    def get_super_categories(self):
        """ Gets the dictionary of super categories for each category in a JSON
            serializable format
        Returns:
```

```
            A dictionary of lists of categories keyed on super_category
        """
        serializable_super_cats = dict()
        for super_cat, categories_set in self.super_categories.items():
            # Sets are not json serializable, so convert to list
            serializable_super_cats[super_cat] = list(categories_set)
        return serializable_super_cats

    def write_masks_to_json(self):
        """ Writes all masks and color categories to the output file path as JSON
        """
        # Serialize the masks and super categories dictionaries
        serializable_masks = self.get_masks()
        serializable_super_cats = self.get_super_categories()
        masks_obj = {
            'masks': serializable_masks,
            'super_categories': serializable_super_cats
        }

        # Write the JSON output file
        output_file_path = Path(self.output_dir) / 'mask_definitions.json'
        with open(output_file_path, 'w+') as json_file:
            json_file.write(json.dumps(masks_obj))

class ImageComposition():
    """ Composes images together in random ways, applying transformations to the foreground
to create a synthetic
        combined image.
    """

    def __init__(self):
        self.allowed_output_types = ['.png', '.jpg', '.jpeg']
        self.allowed_background_types = ['.png', '.jpg', '.jpeg']
        self.zero_padding = 8 # 00000027.png, supports up to 100 million images
        self.max_foregrounds = 3
        self.mask_colors = [(255, 0, 0), (0, 255, 0), (0, 0, 255)]
        assert len(self.mask_colors) >= self.max_foregrounds, 'length of mask_colors should
be >= max_foregrounds'

    def _validate_and_process_args(self, args):
        # Validates input arguments and sets up class variables
        # Args:
        #     args: the ArgumentParser command line arguments

        self.silent = args.silent

        # Validate the count
        assert args.count > 0, 'count must be greater than 0'
        self.count = args.count

        # Validate the width and height
        assert args.width >= 64, 'width must be greater than 64'
```

```
        self.width = args.width
        assert args.height >= 64, 'height must be greater than 64'
        self.height = args.height

        # Validate and process the output type
        if args.output_type is None:
            self.output_type = '.jpg' # default
        else:
            if args.output_type[0] != '.':
                self.output_type = f'.{args.output_type}'
            assert self.output_type in self.allowed_output_types, f'output_type is not supported:
{self.output_type}'

        # Validate and process output and input directories
        self._validate_and_process_output_directory()
        self._validate_and_process_input_directory()

    def _validate_and_process_output_directory(self):
        self.output_dir = Path(args.output_dir)
        self.images_output_dir = self.output_dir / 'images'
        self.masks_output_dir = self.output_dir / 'masks'

        # Create directories
        self.output_dir.mkdir(exist_ok=True)
        self.images_output_dir.mkdir(exist_ok=True)
        self.masks_output_dir.mkdir(exist_ok=True)

        if not self.silent:
            # Check for existing contents in the images directory
            for _ in self.images_output_dir.iterdir():
                # We found something, check if the user wants to overwrite files or quit
                should_continue = input('output_dir is not empty, files may be
overwritten.\nContinue (y/n)? ').lower()
                if should_continue != 'y' and should_continue != 'yes':
                    quit()
                break

    def _validate_and_process_input_directory(self):
        self.input_dir = Path(args.input_dir)
        assert self.input_dir.exists(), f'input_dir does not exist: {args.input_dir}'

        for x in self.input_dir.iterdir():
            if x.name == 'foregrounds':
                self.foregrounds_dir = x
            elif x.name == 'backgrounds':
                self.backgrounds_dir = x

        assert self.foregrounds_dir is not None, 'foregrounds sub-directory was not found in
the input_dir'
        assert self.backgrounds_dir is not None, 'backgrounds sub-directory was not found in
the input_dir'
```

```
        self._validate_and_process_foregrounds()
        self._validate_and_process_backgrounds()

    def _validate_and_process_foregrounds(self):
        # Validates input foregrounds and processes them into a foregrounds dictionary.
        # Expected directory structure:
        # + foregrounds_dir
        #     + super_category_dir
        #         + category_dir
        #             + foreground_image.png

        self.foregrounds_dict = dict()

        for super_category_dir in self.foregrounds_dir.iterdir():
            if not super_category_dir.is_dir():
                warnings.warn(f'file found in foregrounds directory (expected super-category
directories), ignoring: {super_category_dir}')
                continue

            # This is a super category directory
            for category_dir in super_category_dir.iterdir():
                if not category_dir.is_dir():
                    warnings.warn(f'file found in super category directory (expected category
directories), ignoring: {category_dir}')
                    continue

                # This is a category directory
                for image_file in category_dir.iterdir():
                    if not image_file.is_file():
                        warnings.warn(f'a directory was found inside a category directory,
ignoring: {str(image_file)}')
                        continue
                    if image_file.suffix != '.png':
                        warnings.warn(f'foreground must be a .png file, skipping: {str(image_file)}')
                        continue

                    # Valid foreground image, add to foregrounds_dict
                    super_category = super_category_dir.name
                    category = category_dir.name

                    if super_category not in self.foregrounds_dict:
                        self.foregrounds_dict[super_category] = dict()

                    if category not in self.foregrounds_dict[super_category]:
                        self.foregrounds_dict[super_category][category] = []

                    self.foregrounds_dict[super_category][category].append(image_file)

        assert len(self.foregrounds_dict) > 0, 'no valid foregrounds were found'

    def _validate_and_process_backgrounds(self):
        self.backgrounds = []
```

```
        for image_file in self.backgrounds_dir.iterdir():
            if not image_file.is_file():
                warnings.warn(f'a directory was found inside the backgrounds directory, 
ignoring: {image_file}')
                continue

            if image_file.suffix not in self.allowed_background_types:
                warnings.warn(f'background must match an accepted type 
{str(self.allowed_background_types)}, ignoring: {image_file}')
                continue

            # Valid file, add to backgrounds list
            self.backgrounds.append(image_file)

        assert len(self.backgrounds) > 0, 'no valid backgrounds were found'

    def _generate_images(self):
        # Generates a number of images and creates segmentation masks, then
        # saves a mask_definitions.json file that describes the dataset.

        print(f'Generating {self.count} images with masks...')

        mju = MaskJsonUtils(self.output_dir)

        # Create all images/masks (with tqdm to have a progress bar)
        for i in tqdm(range(self.count)):
            # Randomly choose a background
            background_path = random.choice(self.backgrounds)

            num_foregrounds = random.randint(1, self.max_foregrounds)
            foregrounds = []
            for fg_i in range(num_foregrounds):
                # Randomly choose a foreground
                super_category = random.choice(list(self.foregrounds_dict.keys()))
                category = random.choice(list(self.foregrounds_dict[super_category].keys()))
                foreground_path = random.choice(self.foregrounds_dict[super_category][category])

                # Get the color
                mask_rgb_color = self.mask_colors[fg_i]

                foregrounds.append({
                    'super_category':super_category,
                    'category':category,
                    'foreground_path':foreground_path,
                    'mask_rgb_color':mask_rgb_color
                })

            # Compose foregrounds and background
            composite, mask = self._compose_images(foregrounds, background_path)

            # Create the file name (used for both composite and mask)
            save_filename = f'{i:0{self.zero_padding}}' # e.g. 00000023.jpg
```

```
            # Save composite image to the images sub-directory
            composite_filename = f'{save_filename}{self.output_type}' # e.g. 00000023.jpg
            composite_path = self.output_dir / 'images' / composite_filename # e.g. my_output_dir/images/00000023.jpg
            composite = composite.convert('RGB') # remove alpha
            composite.save(composite_path)

            # Save the mask image to the masks sub-directory
            mask_filename = f'{save_filename}.png' # masks are always png to avoid lossy compression
            mask_path = self.output_dir / 'masks' / mask_filename # e.g. my_output_dir/masks/00000023.png
            mask.save(mask_path)

            color_categories = dict()
            for fg in foregrounds:
                # Add category and color info
                mju.add_category(fg['category'], fg['super_category'])
                color_categories[str(fg['mask_rgb_color'])] = \
                    {
                        'category':fg['category'],
                        'super_category':fg['super_category']
                    }

            # Add the mask to MaskJsonUtils
            mju.add_mask(
                composite_path.relative_to(self.output_dir).as_posix(),
                mask_path.relative_to(self.output_dir).as_posix(),
                color_categories
            )

        #Write masks to json
        mju.write_masks_to_json()

    def _compose_images(self, foregrounds, background_path):
        # Composes a foreground image and a background image and creates a segmentation mask
        # using the specified color. Validation should already be done by now.
        # Args:
        #     foregrounds: a list of dicts with format:
        #       [{
        #           'super_category':super_category,
        #           'category':category,
        #           'foreground_path':foreground_path,
        #           'mask_rgb_color':mask_rgb_color
        #       },...]
        #     background_path: the path to a valid background image
        # Returns:
        #     composite: the composed image
        #     mask: the mask image
```

```
        # Open background and convert to RGBA
        background = Image.open(background_path)
        background = background.convert('RGBA')

        # Crop background to desired size (self.width x self.height), randomly positioned
        bg_width, bg_height = background.size
        max_crop_x_pos = bg_width - self.width
        max_crop_y_pos = bg_height - self.height
        assert max_crop_x_pos >= 0, f'desired width, {self.width}, is greater than
background width, {bg_width}, for {str(background_path)}'
        assert max_crop_y_pos >= 0, f'desired height, {self.height}, is greater than
background height, {bg_height}, for {str(background_path)}'
        crop_x_pos = random.randint(0, max_crop_x_pos)
        crop_y_pos = random.randint(0, max_crop_y_pos)
        composite = background.crop((crop_x_pos, crop_y_pos, crop_x_pos + self.width,
crop_y_pos + self.height))
        composite_mask = Image.new('RGB', composite.size, 0)

        for fg in foregrounds:
            fg_path = fg['foreground_path']

            # Perform transformations
            fg_image = self._transform_foreground(fg, fg_path)

            # Choose a random x,y position for the foreground
            max_x_position = composite.size[0] - fg_image.size[0]
            max_y_position = composite.size[1] - fg_image.size[1]
            assert max_x_position >= 0 and max_y_position >= 0, \
            f'foreground {fg_path} is too big ({fg_image.size[0]}x{fg_image.size[1]}) for the
requested output size ({self.width}x{self.height}), check your input parameters'
            paste_position = (random.randint(0, max_x_position), random.randint(0,
max_y_position))

            # Create a new foreground image as large as the composite and paste it on top
            new_fg_image = Image.new('RGBA', composite.size, color = (0, 0, 0, 0))
            new_fg_image.paste(fg_image, paste_position)

            # Extract the alpha channel from the foreground and paste it into a new image the size
of the composite
            alpha_mask = fg_image.getchannel(3)
            new_alpha_mask = Image.new('L', composite.size, color = 0)
            new_alpha_mask.paste(alpha_mask, paste_position)
            composite = Image.composite(new_fg_image, composite, new_alpha_mask)

            # Grab the alpha pixels above a specified threshold
            alpha_threshold = 200
            mask_arr = np.array(np.greater(np.array(new_alpha_mask), alpha_threshold),
dtype=np.uint8)
            uint8_mask = np.uint8(mask_arr) # This is composed of 1s and 0s

            # Multiply the mask value (1 or 0) by the color in each RGB channel and combine to
get the mask
```

```
        mask_rgb_color = fg['mask_rgb_color']
        red_channel = uint8_mask * mask_rgb_color[0]
        green_channel = uint8_mask * mask_rgb_color[1]
        blue_channel = uint8_mask * mask_rgb_color[2]
        rgb_mask_arr = np.dstack((red_channel, green_channel, blue_channel))
        isolated_mask = Image.fromarray(rgb_mask_arr, 'RGB')
        isolated_alpha = Image.fromarray(uint8_mask * 255, 'L')

        composite_mask = Image.composite(isolated_mask, composite_mask, isolated_alpha)

    return composite, composite_mask

def _transform_foreground(self, fg, fg_path):
    # Open foreground and get the alpha channel
    fg_image = Image.open(fg_path)
    fg_alpha = np.array(fg_image.getchannel(3))
    assert np.any(fg_alpha == 0), f'foreground needs to have some transparency:
{str(fg_path)}'

    # ** Apply Transformations **
    # Rotate the foreground
    angle_degrees = random.randint(0, 359)
    fg_image = fg_image.rotate(angle_degrees, resample=Image.BICUBIC, expand=True)

    # Scale the foreground
    scale = random.random() * .5 + .5 # Pick something between .5 and 1
    new_size = (int(fg_image.size[0] * scale), int(fg_image.size[1] * scale))
    fg_image = fg_image.resize(new_size, resample=Image.BICUBIC)

    # Adjust foreground brightness
    brightness_factor = random.random() * .4 + .7 # Pick something between .7 and 1.1
    enhancer = ImageEnhance.Brightness(fg_image)
    fg_image = enhancer.enhance(brightness_factor)

    # Add any other transformations here...

    return fg_image

def _create_info(self):
    # A convenience wizard for automatically creating dataset info
    # The user can always modify the resulting .json manually if needed

    if self.silent:
        # No user wizard in silent mode
        return

    should_continue = input('Would you like to create dataset info json? (y/n) ').lower()
    if should_continue != 'y' and should_continue != 'yes':
        print('No problem. You can always create the json manually.')
        quit()

    print('Note: you can always modify the json manually if you need to update this.')
```

```
        info = dict()
        info['description'] = input('Description: ')
        info['url'] = input('URL: ')
        info['version'] = input('Version: ')
        info['contributor'] = input('Contributor: ')
        now = datetime.now()
        info['year'] = now.year
        info['date_created'] = f'{now.month:0{2}}/{now.day:0{2}}/{now.year}'

        image_license = dict()
        image_license['id'] = 0

        should_add_license = input('Add an image license? (y/n) ').lower()
        if should_add_license != 'y' and should_add_license != 'yes':
            image_license['url'] = ''
            image_license['name'] = 'None'
        else:
            image_license['name'] = input('License name: ')
            image_license['url'] = input('License URL: ')

        dataset_info = dict()
        dataset_info['info'] = info
        dataset_info['license'] = image_license

        # Write the JSON output file
        output_file_path = Path(self.output_dir) / 'dataset_info.json'
        with open(output_file_path, 'w+') as json_file:
            json_file.write(json.dumps(dataset_info))

        print('Successfully created {output_file_path}')


    # Start here
    def main(self, args):
        self._validate_and_process_args(args)
        self._generate_images()
        self._create_info()
        print('Image composition completed.')



if __name__ == "__main__":
    import argparse

    parser = argparse.ArgumentParser(description="Image Composition")
    parser.add_argument("--input_dir", type=str, dest="input_dir", required=True,
help="The input directory. \
                        This contains a 'backgrounds' directory of pngs or jpgs, and a
'foregrounds' directory which \
                        contains supercategory directories (e.g. 'animal', 'vehicle'), each of which
contain category \
                        directories (e.g. 'horse', 'bear'). Each category directory contains png
```

```
images of that item on a \
                transparent background (e.g. a grizzly bear on a transparent
background).")
    parser.add_argument("--output_dir", type=str, dest="output_dir", required=True,
help="The directory where images, masks, \
                and json files will be placed")
    parser.add_argument("--count", type=int, dest="count", required=True, help="number
of composed images to create")
    parser.add_argument("--width", type=int, dest="width", required=True, help="output
image pixel width")
    parser.add_argument("--height", type=int, dest="height", required=True, help="output
image pixel height")
    parser.add_argument("--output_type", type=str, dest="output_type", help="png or jpg
(default)")
    parser.add_argument("--silent", action='store_true', help="silent mode; doesn't prompt
the user for input, \
                automatically overwrites files")

    args = parser.parse_args()

    image_comp = ImageComposition()
    image_comp.main(args)
```

COCO_json_utils.py

```
#!/usr/bin/python

import numpy as np
import json
from pathlib import Path
from tqdm import tqdm
from skimage import measure, io
from shapely.geometry import Polygon, MultiPolygon
from PIL import Image

class InfoJsonUtils():
    """ Creates an info object to describe a COCO dataset
    """
    def create_coco_info(self, description, url, version, year, contributor, date_created):
        """ Creates the "info" portion of COCO json
        """
        info = dict()
        info['description'] = description
        info['url'] = url
        info['version'] = version
        info['year'] = year
        info['contributor'] = contributor
        info['date_created'] = date_created

        return info

class LicenseJsonUtils():
    """ Creates a license object to describe a COCO dataset
```

```
    """
    def create_coco_license(self, url, license_id, name):
        """ Creates the "licenses" portion of COCO json
        """
        lic = dict()
        lic['url'] = url
        lic['id'] = license_id
        lic['name'] = name

        return lic

class CategoryJsonUtils():
    """ Creates a category object to describe a COCO dataset
    """
    def create_coco_category(self, supercategory, category_id, name):
        category = dict()
        category['supercategory'] = supercategory
        category['id'] = category_id
        category['name'] = name

        return category

class ImageJsonUtils():
    """ Creates an image object to describe a COCO dataset
    """
    def create_coco_image(self, image_path, image_id, image_license):
        """ Creates the "image" portion of COCO json
        """
        # Open the image and get the size
        image_file = Image.open(image_path)
        width, height = image_file.size

        image = dict()
        image['license'] = image_license
        image['file_name'] = image_path.name
        image['width'] = width
        image['height'] = height
        image['id'] = image_id

        return image

class AnnotationJsonUtils():
    """ Creates an annotation object to describe a COCO dataset
    """
    def __init__(self):
        self.annotation_id_index = 0

    def create_coco_annotations(self, image_mask_path, image_id, category_ids):
        """ Takes a pixel-based RGB image mask and creates COCO annotations.
        Args:
            image_mask_path: a pathlib.Path to the image mask
            image_id: the integer image id
```

```
        category_ids: a dictionary of integer category ids keyed by RGB color (a tuple
converted to a string)
            e.g. {'(255, 0, 0)': {'category': 'owl', 'super_category': 'bird'} }
    Returns:
        annotations: a list of COCO annotation dictionaries that can
        be converted to json. e.g.:
        {
            "segmentation": [[101.79,307.32,69.75,281.11,...,100.05,309.66]],
            "area": 51241.3617,
            "iscrowd": 0,
            "image_id": 284725,
            "bbox": [68.01,134.89,433.41,174.77],
            "category_id": 6,
            "id": 165690
        }
    """
    # Set class variables
    self.image_id = image_id
    self.category_ids = category_ids

    # Make sure keys in category_ids are strings
    for key in self.category_ids.keys():
        if type(key) is not str:
            raise TypeError('category_ids keys must be strings (e.g. "(0, 0, 255)")')
        break

    # Open and process image
    self.mask_image = Image.open(image_mask_path)
    self.mask_image = self.mask_image.convert('RGB')
    self.width, self.height = self.mask_image.size

    # Split up the multi-colored masks into multiple 0/1 bit masks
    self._isolate_masks()

    # Create annotations from the masks
    self._create_annotations()

    return self.annotations

def _isolate_masks(self):
    # Breaks mask up into isolated masks based on color

    self.isolated_masks = dict()
    for x in range(self.width):
        for y in range(self.height):
            pixel_rgb = self.mask_image.getpixel((x,y))
            pixel_rgb_str = str(pixel_rgb)

            # If the pixel is any color other than black, add it to a respective isolated image
mask
            if not pixel_rgb == (0, 0, 0):
                if self.isolated_masks.get(pixel_rgb_str) is None:
```

```
                        # Isolated mask doesn't have its own image yet, create one
                        # with 1-bit pixels, default black. Make room for 1 pixel of
                        # padding on each edge to allow the contours algorithm to work
                        # when shapes bleed up to the edge
                        self.isolated_masks[pixel_rgb_str] = Image.new('1', (self.width + 2, self.height + 2))

                    # Add the pixel to the mask image, shifting by 1 pixel to account for padding
                    self.isolated_masks[pixel_rgb_str].putpixel((x + 1, y + 1), 1)

    def _create_annotations(self):
        # Creates annotations for each isolated mask

        # Each image may have multiple annotations, so create an array
        self.annotations = []
        for key, mask in self.isolated_masks.items():
            annotation = dict()
            annotation['segmentation'] = []
            annotation['iscrowd'] = 0
            annotation['image_id'] = self.image_id
            if not self.category_ids.get(key):
                print(f'category color not found: {key}; check for missing category or antialiasing')
                continue
            annotation['category_id'] = self.category_ids[key]
            annotation['id'] = self._next_annotation_id()

            # Find contours in the isolated mask
            mask = np.asarray(mask, dtype=np.float32)
            contours = measure.find_contours(mask, 0.5, positive_orientation='low')

            polygons = []
            for contour in contours:
                # Flip from (row, col) representation to (x, y)
                # and subtract the padding pixel
                for i in range(len(contour)):
                    row, col = contour[i]
                    contour[i] = (col - 1, row - 1)

                # Make a polygon and simplify it
                poly = Polygon(contour)
                poly = poly.simplify(1.0, preserve_topology=False)

                if (poly.area > 16): # Ignore tiny polygons
                    if (poly.geom_type == 'MultiPolygon'):
                        # if MultiPolygon, take the smallest convex Polygon containing all the points in the object
                        poly = poly.convex_hull

                    if (poly.geom_type == 'Polygon'): # Ignore if still not a Polygon (could be a line or point)
                        polygons.append(poly)
```

```
                segmentation = np.array(poly.exterior.coords).ravel().tolist()
                annotation['segmentation'].append(segmentation)

            if len(polygons) == 0:
                # This item doesn't have any visible polygons, ignore it
                # (This can happen if a randomly placed foreground is covered up
                #  by other foregrounds)
                continue

            # Combine the polygons to calculate the bounding box and area
            multi_poly = MultiPolygon(polygons)
            x, y, max_x, max_y = multi_poly.bounds
            self.width = max_x - x
            self.height = max_y - y
            annotation['bbox'] = (x, y, self.width, self.height)
            annotation['area'] = multi_poly.area

            # Finally, add this annotation to the list
            self.annotations.append(annotation)

    def _next_annotation_id(self):
        # Gets the next annotation id
        # Note: This is not a unique id. It simply starts at 0 and increments each time it is called

        a_id = self.annotation_id_index
        self.annotation_id_index += 1
        return a_id

class CocoJsonCreator():
    def validate_and_process_args(self, args):
        """ Validates the arguments coming in from the command line and performs
            initial processing
        Args:
            args: ArgumentParser arguments
        """
        # Validate the mask definition file exists
        mask_definition_file = Path(args.mask_definition)
        if not (mask_definition_file.exists and mask_definition_file.is_file()):
            raise FileNotFoundError(f'mask definition file was not found:
{mask_definition_file}')

        # Load the mask definition json
        with open(mask_definition_file) as json_file:
            self.mask_definitions = json.load(json_file)

        self.dataset_dir = mask_definition_file.parent

        # Validate the dataset info file exists
        dataset_info_file = Path(args.dataset_info)
        if not (dataset_info_file.exists() and dataset_info_file.is_file()):
            raise FileNotFoundError(f'dataset info file was not found: {dataset_info_file}')
```

```
        # Load the dataset info json
        with open(dataset_info_file) as json_file:
            self.dataset_info = json.load(json_file)

        assert 'info' in self.dataset_info, 'dataset_info JSON was missing "info"'
        assert 'license' in self.dataset_info, 'dataset_info JSON was missing "license"'

    def create_info(self):
        """ Creates the "info" piece of the COCO json
        """
        info_json = self.dataset_info['info']
        iju = InfoJsonUtils()
        return iju.create_coco_info(
            description = info_json['description'],
            version = info_json['version'],
            url = info_json['url'],
            year = info_json['year'],
            contributor = info_json['contributor'],
            date_created = info_json['date_created']
        )

    def create_licenses(self):
        """ Creates the "license" portion of the COCO json
        """
        license_json = self.dataset_info['license']
        lju = LicenseJsonUtils()
        lic = lju.create_coco_license(
            url = license_json['url'],
            license_id = license_json['id'],
            name = license_json['name']
        )
        return [lic]

    def create_categories(self):
        """ Creates the "categories" portion of the COCO json
        Returns:
            categories: category objects that become part of the final json
            category_ids_by_name: a lookup dictionary for category ids based
                on the name of the category
        """
        cju = CategoryJsonUtils()
        categories = []
        category_ids_by_name = dict()
        category_id = 1 # 0 is reserved for the background

        super_categories = self.mask_definitions['super_categories']
        for super_category, _categories in super_categories.items():
            for category_name in _categories:
                categories.append(cju.create_coco_category(super_category, category_id,
category_name))
                category_ids_by_name[category_name] = category_id
                category_id += 1
```

```
        return categories, category_ids_by_name

    def create_images_and_annotations(self, category_ids_by_name):
        """ Creates the list of images (in json) and the annotations for each
            image for the "image" and "annotations" portions of the COCO json
        """
        iju = ImageJsonUtils()
        aju = AnnotationJsonUtils()

        image_objs = []
        annotation_objs = []
        image_license = self.dataset_info['license']['id']
        image_id = 0

        mask_count = len(self.mask_definitions['masks'])
        print(f'Processing {mask_count} mask definitions...')

        # For each mask definition, create image and annotations
        for file_name, mask_def in tqdm(self.mask_definitions['masks'].items()):
            # Create a coco image json item
            image_path = Path(self.dataset_dir) / file_name
            image_obj = iju.create_coco_image(
                image_path,
                image_id,
                image_license)
            image_objs.append(image_obj)

            mask_path = Path(self.dataset_dir) / mask_def['mask']

            # Create a dict of category ids keyed by rgb_color
            category_ids_by_rgb = dict()
            for rgb_color, category in mask_def['color_categories'].items():
                category_ids_by_rgb[rgb_color] = category_ids_by_name[category['category']]
            annotation_obj = aju.create_coco_annotations(mask_path, image_id,
category_ids_by_rgb)
            annotation_objs += annotation_obj # Add the new annotations to the existing list
            image_id += 1

        return image_objs, annotation_objs

    def main(self, args):
        self.validate_and_process_args(args)

        info = self.create_info()
        licenses = self.create_licenses()
        categories, category_ids_by_name = self.create_categories()
        images, annotations = self.create_images_and_annotations(category_ids_by_name)

        master_obj = {
            'info': info,
            'licenses': licenses,
```

```
        'images': images,
        'annotations': annotations,
        'categories': categories
    }

    # Write the json to a file
    output_path = Path(self.dataset_dir) / 'coco_instances.json'
    with open(output_path, 'w+') as output_file:
        json.dump(master_obj, output_file)

    print(f'Annotations successfully written to file:\n{output_path}')

if __name__ == "__main__":
    import argparse

    parser = argparse.ArgumentParser(description="Generate COCO JSON")

    parser.add_argument("-md", "--mask_definition", dest="mask_definition",
        help="path to a mask definition JSON file, generated by MaskJsonUtils module")
    parser.add_argument("-di", "--dataset_info", dest="dataset_info",
        help="path to a dataset info JSON file")

    args = parser.parse_args()

    cjc = CocoJsonCreator()
    cjc.main(args)
```