CS 415 - Computer Vision Project - Object Detection for Assisted Driving and Self Driving Cars.

Team Helium

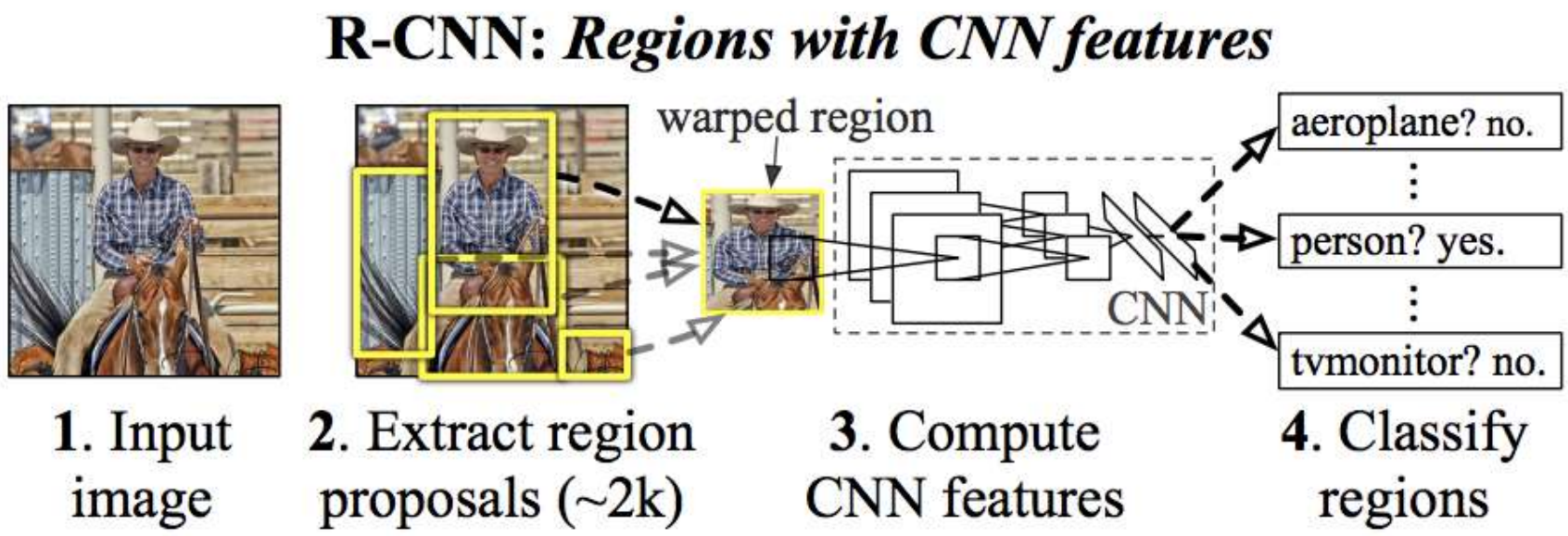
1. Neha Sriramulu
2. Eno Dhamo
3. Krzyzstof Para

1. Introduction

Object Detection is a term used to describe a collection of related computer vision tasks that involves detecting instances of semantic objects in digital photographs or videos. The ultimate purpose of object detection is to locate important items, draw rectangular bounding boxes around them and determine the class of each item discovered. Applications of object detection include detecting pedestrians for self-driving cars, monitoring agricultural crops, and even real-time ball tracking sports. An autonomous vehicle needs sensory input devices like cameras, radar and lasers to allow the car to perceive the world around it, creating a digital map. We'll be focusing on the imaging where cars perform object detection. Object detection is a two-part process, image classification and then image localization. Image classification is determining what the objects in the image are, like a car or a person, while image localization is providing the specific location of these objects. We train a convolutional neural network to recognize various objects, like traffic lights and pedestrians. A convolutional neural network performs convolution operations on images in order to classify them.

2. Related Work

- Some of the pre-existing algorithms for object detection are R-CNN, Fast R-CNN, RetinaNet, YOLO Framework.
- Several key enhancements over the years have boosted the speed of the algorithms, improving test time from the 0.02 frames per second (fps) of R-CNN to the impressive 155 fps of Fast YOLO.
- The R-CNN pipeline is the generation of regions in an image that could belong to a particular object. The authors of the paper use the selective search algorithm which works by generating sub-segmentations of the image that could belong to one object based on color, texture, size and shape and iteratively combining similar regions to form objects. This gives 'object proposals' of different scales.
- YOLO is a convolutional neural network (CNN) for object detection in real-time. The algorithm applies a single neural network to the full image, and then divides the image into regions and predicts bounding boxes and probabilities for each region. These bounding boxes are weighted by the predicted probabilities. It is popular because it achieves high accuracy while also being able to run in real-time. The algorithm "only looks once" at the image in the sense that it requires only one forward propagation pass through the neural network to make predictions.
- Only two teams can be compared to our implementation: Team Argon and Team Krypton. In Team Krypton's approach they used LeNet and Keras. However in Team Argon's approach they follow a similar model and use yolov3 with kitti and tensorflow, the kitti dataset may not be as robust as microsoft's COCO dataset.



3. Problem Statement

Operating a car has become very important and a necessity for the modern-day world. People have to work together to follow the rules of the road. The main problem was that people with vision impairments like being color-blind, or have poor vision may have trouble seeing obstructions on the road or traffic signals. And this sometimes lead to people breaking rules and accidents. Many of these accidents are because of vision impairments that drivers may have. Assisted driving applications in cars can help prevent many of these accidents if they are equipped with a dashcam and a HUD. This is why my team and I wanted to focus on object detection to help assist drivers' who have trouble seeing or are color-blind. We decided to make use of the existing YOLOv3 architecture and implement it in PyTorch with pre-trained weights based on the COCO dataset to assist drivers by detecting objects like traffic signals, cars and make them more visible. This can help reduce the amount of accidents caused by vision impairments.

We will be evaluating different approaches for object detection that can be used in autonomous vehicles and focus on researching the most optimal implementation for real time object detection, classification and tracking with our modifications to the existing algorithms like YoloV3, semantic segmentation, R-CNNs etc. Object detection algorithms need to not only accurately classify and localize important objects, they also need to be incredibly fast at prediction time to meet the real-time demands of video processing. Our main end goal is to produce an optimized implementation of object detection pipeline to make it a viable option for use in assisted driving applications.

4. Our Approach

Our approach to deal with object detection for assisted driving is through a Pytorch implementation of the darknet model for the YOLO V3 architecture instead of directly using the darknet model which was originally written in CPP. 14 different classes which are considered hazards or stuff to be alert about will be detected, these include:

- person
- bicycle
- car
- motorbike
- bus
- train
- truck
- traffic light
- fire hydrant
- stop sign
- parking meter
- cat
- dog
- cow

The results of the detected objects will be pipelined by passing the bounding box coordinates, class and confidence to the next stage for further processing. Ideally, in the real world, an integrated application would perform all this simultaneously and an analysis on the runtime will be done at the end.

Resources:

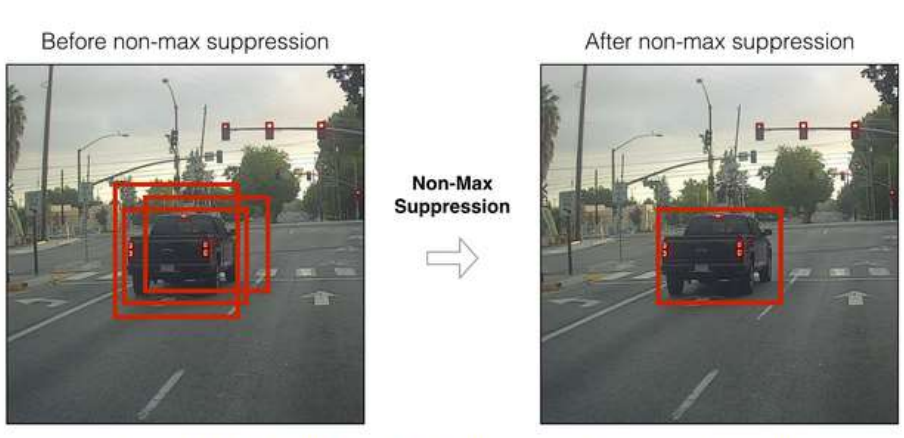
- YOLO V3 weights - Official YOLO V3 page - https://pjreddie.com/darknet/yolo/ (https://pjreddie.com/darknet/yolo/) (We have used pretrained weight adjustments for detection based on training with the COCO dataset, present in the folder.)
- YOLO V3 Configuration file - Official YOLO V3 page - https://pjreddie.com/darknet/yolo/ (https://pjreddie.com/darknet/yolo/) (This file describes the YOLOV3 model architecture completely, present in the folder.)
- CUDA GPU drivers - Official Nvidia page - https://developer.nvidia.com/cuda-downloads (https://developer.nvidia.com/cuda-downloads) (You need to have CUDA 10+ drivers installed to enable GPU, not included in the folder.)

4.1 Implementation of YOLO V3 using Pytorch

- Step - 1) Building the model from the configuration file: The yolov3.cfg file is used to build the model in the construct_cfg and build_network functions in the DNmodel.py file. The network is built by inheriting the nn.Module. A sample of the network structure from the cfg file parsed that is stored in a list of dictionaries is shown below

```
In [1]: runfile('C:/Users/NehaS/Desktop/CS415 PROJECT/YoloV3Pytorch/DNModel.py',
wdir='C:/Users/NehaS/Desktop/CS415 PROJECT/YoloV3Pytorch')
[{'type': 'net', 'batch': '1', 'subdivisions': '1', 'width': '320', 'height': '320',
'channels': '3', 'momentum': '0.9', 'decay': '0.0005', 'angle': '0', 'saturation':
'1.5', 'exposure': '1.5', 'hue': '.1', 'learning_rate': '0.001', 'burn_in': '1000',
'max_batches': '500200', 'policy': 'steps', 'steps': '400000,450000', 'scales': '.
1,.1'}, {'type': 'convolutional', 'batch_normalize': '1', 'filters': '32', 'size': '3'
'stride': '1', 'pad': '1', 'activation': 'leaky'}, {'type': 'convolutional',
'batch_normalize': '1', 'filters': '64', 'size': '3', 'stride': '2', 'pad': '1',
'activation': 'leaky'}, {'type': 'convolutional', 'batch_normalize': '1', 'filters':
'32', 'size': '1', 'stride': '1', 'pad': '1', 'activation': 'leaky'}, {'type':
'convolutional', 'batch_normalize': '1', 'filters': '64', 'size': '3', 'stride': '1',
'pad': '1', 'activation': 'leaky'}, {'type': 'shortcut', 'from': '-3', 'activation':
'linear'}, {'type': 'convolutional', 'batch_normalize': '1', 'filters': '128', 'size':
'3', 'stride': '2', 'pad': '1', 'activation': 'leaky'}, {'type': 'convolutional',
'batch_normalize': '1', 'filters': '64', 'size': '1', 'stride': '1', 'pad': '1',
'activation': 'leaky'}, {'type': 'convolutional', 'batch_normalize': '1', 'filters':
'128', 'size': '3', 'stride': '1', 'pad': '1', 'activation': 'leaky'}, {'type':
'shortcut', 'from': '-3', 'activation': 'linear'}, {'type': 'convolutional',
```

- Step - 2) The load_weights function loads the weights from the yolov3.weights file. Edit the coco.names file to edit the classes you want to detect.
- Step - 3) The transformOutput function in util.py performs the operations to make predictions on the bounding box. 0 is returned if no class is detected, else the class id integer. The returned value are the batch x grids x bounding box attributes which are going to be pipelined into our next process.
- Step - 4) Non Max Suppression is implemented for each class of objects to ensure the same object has not been detected more than once. NMS is implemented in the write_results function where all the confidence levels are sorted in decreasing order and the IOUs are retrieved by the bbox_iou function in util.py. The results below the mentioned threshold are filtered out to ensure there is only 1 detection of each object.



In this example, the model has predicted 3 cars, but it's actually 3 predictions of the same car. Running non-max suppression (NMS) will select only the most accurate (highest probabiliy) one of the 3 boxes.

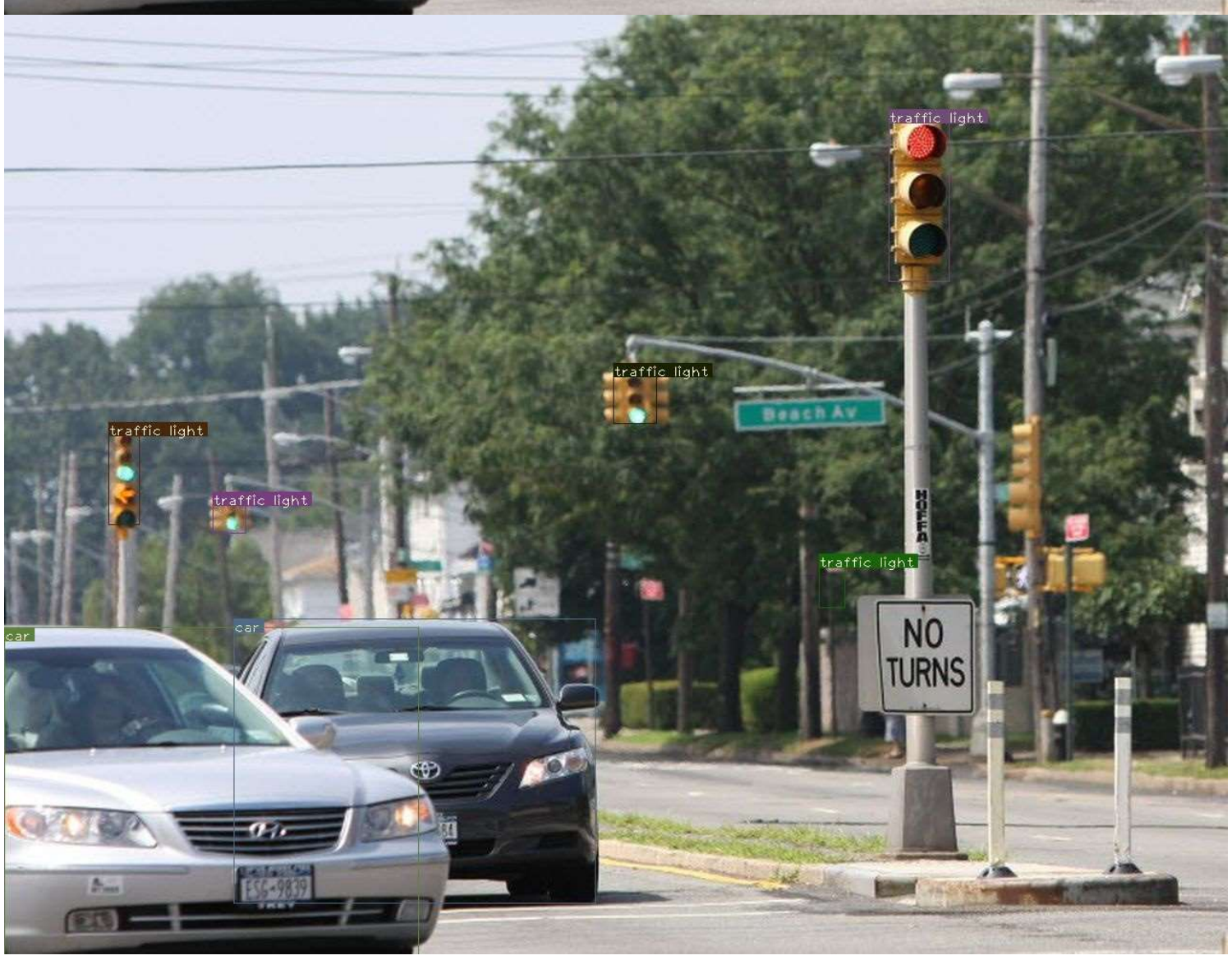
4.2 Demo of Pytorch YOLOV3

The demo can be tested on any image or mp4 file(low fps). Add the image to the folder and type --video <nameofvid.mp4> to the parameters of the detect_video.py function or detect.py -- image <image.jpg> to only process images. The result of images will be in the result folder.

In [5]:
```
!python detect.py --images ./images/test1.jpg
```

```
Network loaded
Size before transform =>  torch.Size([1, 255, 8, 8])
Size after transform =>  torch.Size([1, 192, 85])
Size before transform =>  torch.Size([1, 255, 16, 16])
Size after transform =>  torch.Size([1, 768, 85])
Size before transform =>  torch.Size([1, 255, 32, 32])
Size after transform =>  torch.Size([1, 3072, 85])
torch.Size([1, 4032, 85])
Network loaded
Size before transform =>  torch.Size([1, 255, 8, 8])
Size after transform =>  torch.Size([1, 192, 85])
Size before transform =>  torch.Size([1, 255, 16, 16])
Size after transform =>  torch.Size([1, 768, 85])
Size before transform =>  torch.Size([1, 255, 32, 32])
Size after transform =>  torch.Size([1, 3072, 85])
Objects Detected:    car car car car car car car car car truck traffic light traffic light traffic light traffic ligh
t traffic light traffic light traffic light
----------------------------------------------------------
```

Detections on images.



Detections on videos.

Note: You need to have an NVIDIA GPU and correct CUDA drivers enabled to get decent FPS(Max 44, averages around 20-35 in our implementation.)

In [ ]:
```
!python detect_video.py --video vid1.mp4
```

4.3 Traffic Light Detection

A Masking R-CNN is used to load a pretrained .h5 file and visualize the results of the neural network one image at a time. Modifications to the original function were made in order to handle custom parameters like saving regions of interest to a folder. The generated bounding-boxes allowed to filter the regions of interest from all class objects to just the very specific one needed for this application which was the "Traffic Light" object. Once the bounding box coordinates were found, TensorFlow's crop was used for bounding boxes to save the images in a separate directory because this was useful for plotting the results later. The original implementation already did a splendid job at categorizing the different objects with different separate colors and was left unchanged. Next, the images needed a standardize function that would make sure there was consistency in the input images for later analysis, this was used from the link given in the bibliography under github repo(8). Now that the images were standardized, a function that would detect the colors using the sum of total saturation on the image and averaging out the range of values and then applying a bitwise and operation to these color channel layers, the link used for this is also given in the bibliography under github repo(7). Finally, some plots have been added in between each function for extra visualization.

4.4 Implementation of Traffic Light Detection and plots

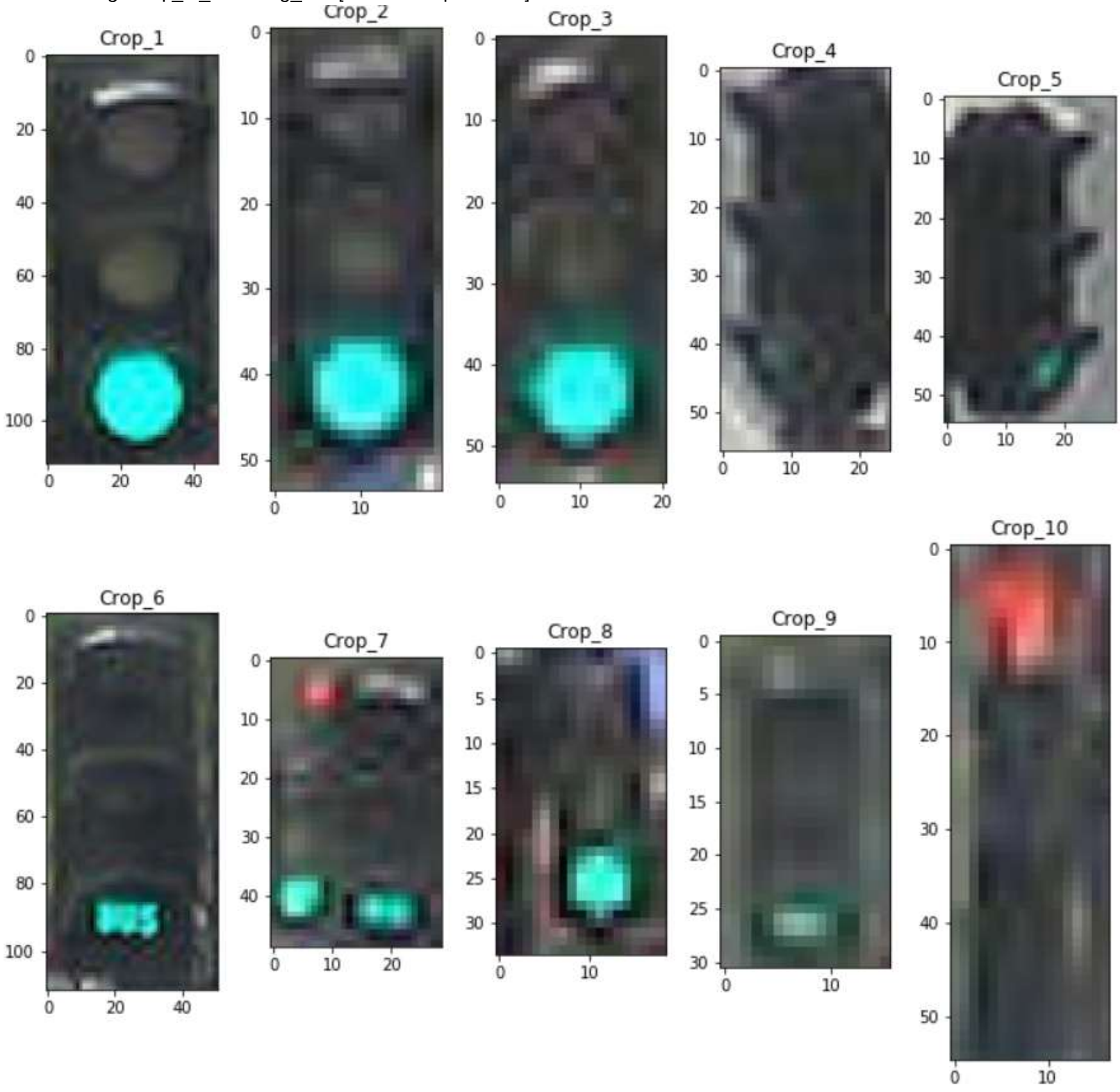
- Step 1: Load the label settings for each object
- Step 2: Initialize the prediction layer
- Step 3: Run the normal implementation with all the layers

```
Total time of function execution:  10.428970336914062
```

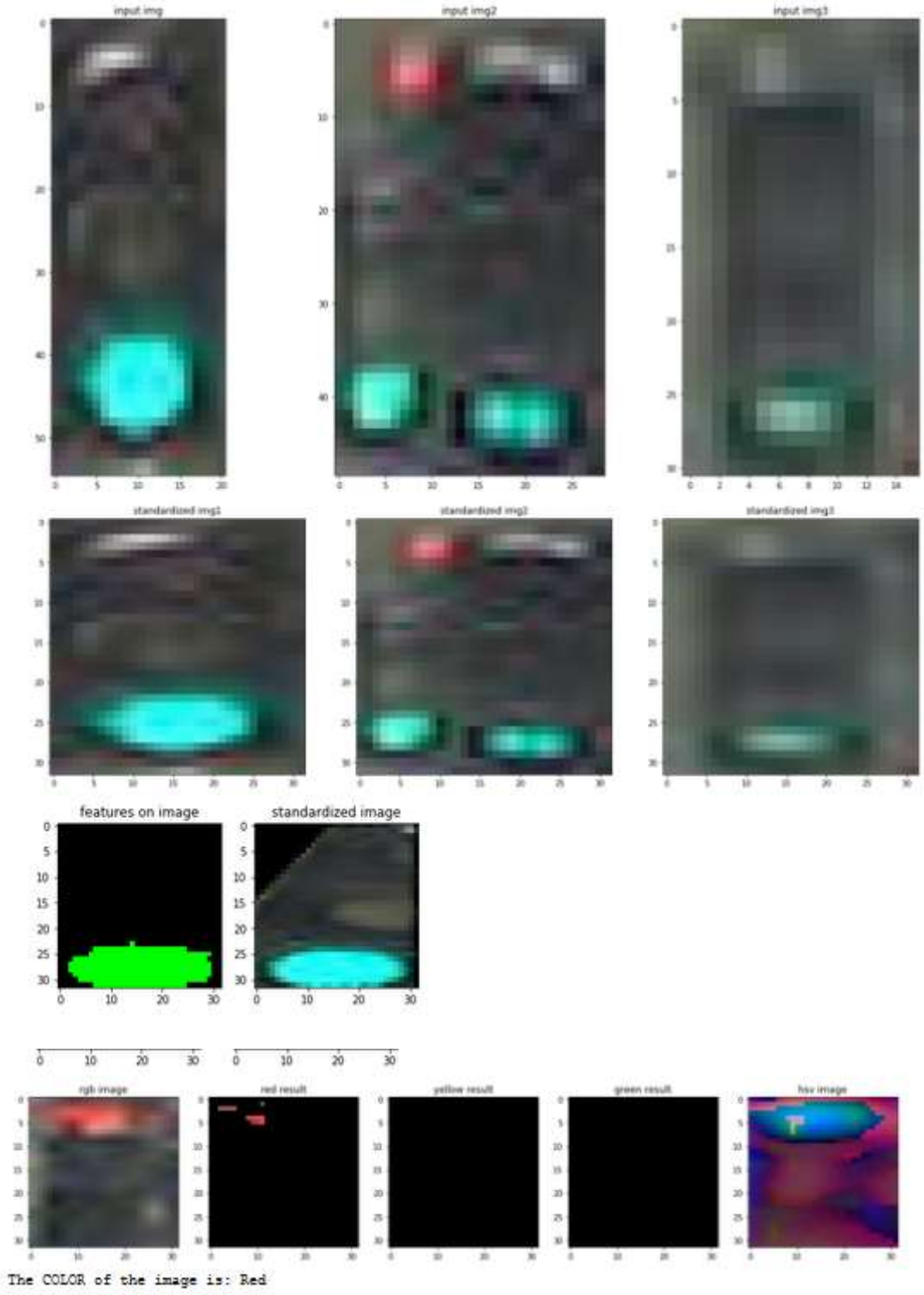
- Step 4: Modify above to filter for one object

Cropping techniques attempted for this portion

- numpy index slicing [Bugs with our configuration]
- numpy masking [Bugs with our configuration]
- Image.crop from PIL [The cropped image was in the incorrect area/axis]
- skimage.util.crop function [Bugs with our configuration]
- tensorflow.image.crop_to_bounding_box [Desired output found]



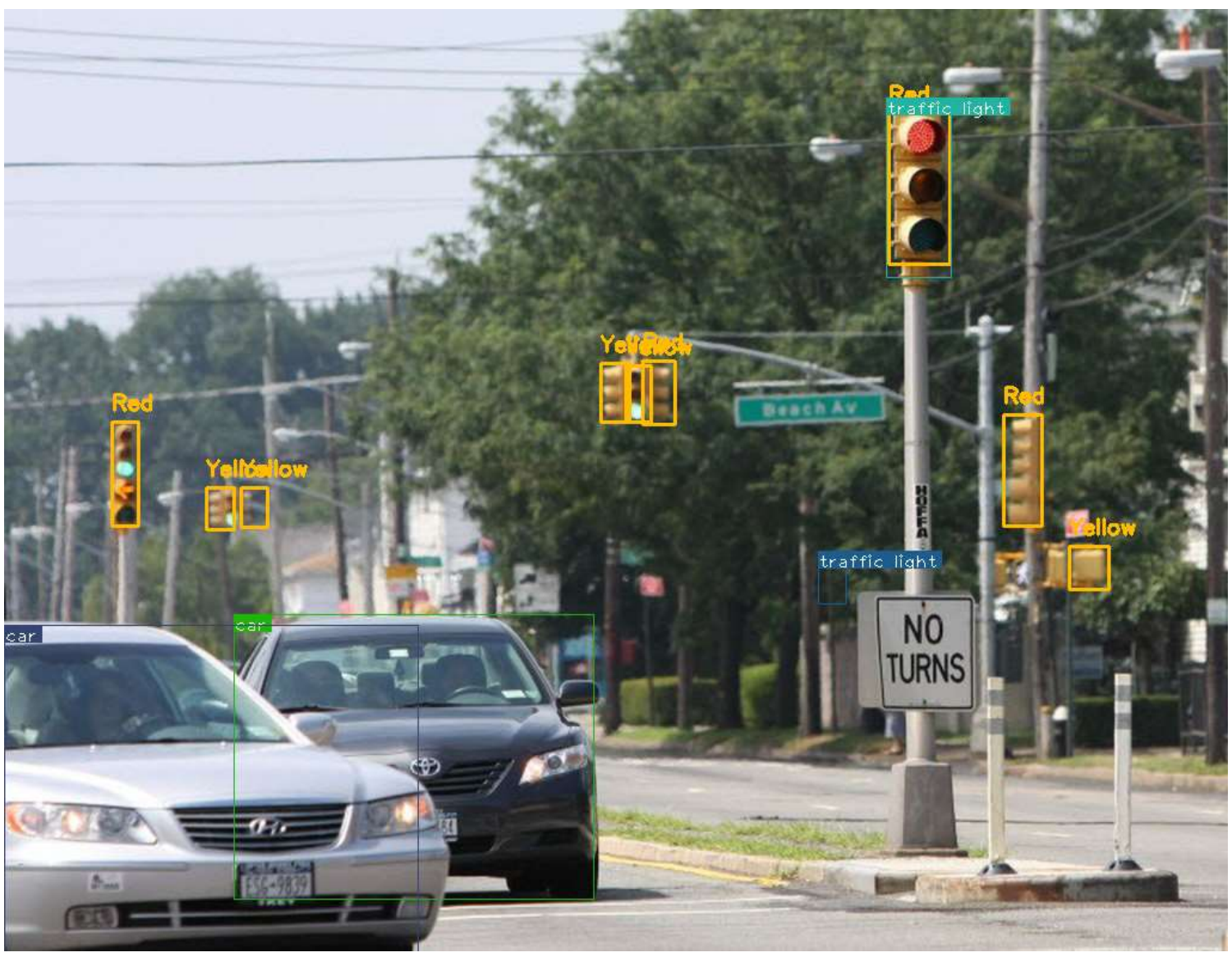
- Step 5: Standardize the images for analysis
- Green light detection.



- Red light detection.

5. Results

The final results after pipelining the object detection and traffic light detection would result in the following images.



6. Conclusion

Although the model does detect objects like traffic lights that are hardly visible from a distance which almost solves our purpose, you can see the results are not perfect, there is a small classification error in object detection and signal color which doesn't allow it to be ready for real world applications. The future work includes:

- Retraining the model with more data to differentiate traffic lights from stop signs as observed in the video, it was a major source of error.
- Implementing a version of traffic light detection using CNNs which run fast and meet standards of real time detection.
- Making the pipeline faster through either parallel processing or retraining the model with a custom annotated datatset to detect different configurations of traffic lights.(Currently runs at 20(+/-)5 fps).
- Extending the implementation of traffic light detection to videos.

7. Bibliography

Papers

1. R. Girshick, Fast R-CNN (2015), IEEE International Conference on Computer Vision (ICCV).
2. J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, You only look once: Unified, real-time object detection (2015)
3. Jonathan Long, Evan Shelhamer and Trevor Darrell, Fully Convolutional Networks for Semantic Segmentation (2015)
4. J. Redmon and A. Farhadi, Yolov3: An incremental improvement
5. Object Detection for Autonomous Vehicles by Gene Lewis, Stanford University

6. A. Neubeck and L. Van Gool, "Efficient Non-Maximum Suppression," 18th International Conference on Pattern Recognition (ICPR'06), Hong Kong, 2006

Online Articles

1. https://blog.paperspace.com/how-to-implement-a-yolo-object-detector-in-pytorch/ (https://blog.paperspace.com/how-to-implement-a-yolo-object-detector-in-pytorch/)
2. https://medium.com/the-advantages-of-densenet/yolov3-pytorch-video-image-model-488721214470 (https://medium.com/the-advantages-of-densenet/yolov3-pytorch-video-image-model-488721214470)
3. https://towardsdatascience.com/object-detection-and-tracking-in-pytorch-b3cf1a696a98 (https://towardsdatascience.com/object-detection-and-tracking-in-pytorch-b3cf1a696a98)
4. https://machinelearningmastery.com/how-to-perform-object-detection-with-yolov3-in-keras/ (https://machinelearningmastery.com/how-to-perform-object-detection-with-yolov3-in-keras/)
5. https://towardsdatascience.com/color-identification-in-images-machine-learning-application-b26e770c4c71 (https://towardsdatascience.com/color-identification-in-images-machine-learning-application-b26e770c4c71)
6. https://docs.scipy.org/doc/scipy/reference/cluster.vq.html#module-scipy.cluster.vq (https://docs.scipy.org/doc/scipy/reference/cluster.vq.html#module-scipy.cluster.vq)
7. https://www.dataquest.io/blog/tutorial-colors-image-clustering-python/ (https://www.dataquest.io/blog/tutorial-colors-image-clustering-python/)
8. https://www.geeksforgeeks.org/python-opencv-cv2-imwrite-method/ (https://www.geeksforgeeks.org/python-opencv-cv2-imwrite-method/)

Youtube Tutorials

1. https://www.youtube.com/watch?v=NM6lrxy0bxs (https://www.youtube.com/watch?v=NM6lrxy0bxs)
2. https://www.youtube.com/watch?v=9s_FpMpdYW8 (https://www.youtube.com/watch?v=9s_FpMpdYW8)
3. https://www.youtube.com/watch?v=chVamXQp9so (https://www.youtube.com/watch?v=chVamXQp9so)
4. https://www.youtube.com/watch?v=chVamXQp9so (https://www.youtube.com/watch?v=chVamXQp9so)
5. https://www.youtube.com/watch?v=s8Ui_kV9dhw (https://www.youtube.com/watch?v=s8Ui_kV9dhw)

Github Repositories

1. https://github.com/ayooshkathuria/pytorch-yolo-v3 (https://github.com/ayooshkathuria/pytorch-yolo-v3)
2. https://github.com/experiencor/keras-yolo3 (https://github.com/experiencor/keras-yolo3)
3. https://github.com/matterport/Mask_RCNN (https://github.com/matterport/Mask_RCNN)
4. https://github.com/andylucny/COCO_SemanticSegmentation_MaskRCNN_keras_tf2_Example (https://github.com/andylucny/COCO_SemanticSegmentation_MaskRCNN_keras_tf2_Example)
5. https://www.geeksforgeeks.org/python-draw-rectangular-shape-and-extract-objects-using-opencv/ (https://www.geeksforgeeks.org/python-draw-rectangular-shape-and-extract-objects-using-opencv/)
6. https://github.com/AbdullahWali/ImageNet-Bounding-Box-Crop-Tool (https://github.com/AbdullahWali/ImageNet-Bounding-Box-Crop-Tool)
7. https://github.com/designeryuan/traffic_light-classified/blob/master/traffic_light_classified-master/Traffic-Light-Classfy.ipynb (https://github.com/designeryuan/traffic_light-classified/blob/master/traffic_light_classified-master/Traffic-Light-Classfy.ipynb)
8. https://github.com/kobbled/ITSDC-Udacity-Traffic-Light-Classifier/blob/master/Traffic_Light_Classifier.py (https://github.com/kobbled/ITSDC-Udacity-Traffic-Light-Classifier/blob/master/Traffic_Light_Classifier.py)

8. Contribution

1. **Neha Sriramulu** - Implementation of the YOLOV3 architecture using Pytorch. Experimentation on other implementation using Keras, DarkNet too. Keras was too slow fps wise and using the original DarkNet didn't allow customization of the bounding box function which was crucial for further stages. Worked on the introduction, related work, part of the problem statement, our approach(4-4.2), results, conclusion and bibliography for the report.

2. **Eno Dhamo** - Color classification of traffic light without using CNNs, adding functions needed for plotting observations. Implementation with keras and tensorflow. Experimentation on using image slicing. A hindrance in cropping early on, however tensorflow's crop to box function helped with our customized bounding boxes. Expanded on an implementation using numpy to standardize the images. Worked on Our approach(4.3,4.4) for the report.

3. **Kryzstof Para** - The main part of the work done was research based on RCNNs. Helped with an implementation of traffic light detection for the project, but we ended up using Eno's implementation. Worked on describing the problem statement and analyzing the other team's work for the report.