Report of IN2106 Masterpraktikum Simulation-Based Autonomous Driving in Crowded City (September 2023)

Danyang Long, Hao Yang

***Abstract*—In our practical course project, we did two imitation learning tasks in two different simulators. In task one, we re-implemented the end-to-end autonomous driving architecture from NVIDIA's paper [7]. We trained a convolution neural network Resnet50 to map raw pixels from three cameras (one front-facing and two side-facing) directly to steering commands. This end-to-end approach works surprisingly well at NVIDIA Udacity Simulator with only one lane. After training with data from manual operation mode, we use the trained model to process the images from the simulator when running autonomous driving mode and output the steering angle command to the simulator. The self-driving car performs well in turning, going through bridges, and "climbing" mountains. In task two, we trained not only a Resnet50 to output steering angle controls but also a Yolov5 (You Only Look Once Version 5) model to do object detection jobs and to output throttle and brake controls. The simulator used in this task is much more complicated than the one used in task one. We need to drive in a crowded city with traffic lights and vehicles around. After testing, the trained model can output ideal control and the car manages to do autonomous driving. It can stop between crosswalks when the traffic light is red. And it can also stop when there is a car in front of us. Sometimes it may even overtake the vehicle in front of it without causing traffic accidents.**

I. Introduction

CVOLUTIONAL Neural Networks [6] have revolutionized pattern recognition. With CNNs features can be learned automatically from training datasets. This kind of approach is especially powerful in image recognition tasks since the convolution operation can capture the 2D nature of images and it can scan an entire image by using the convolution kernels, which only learns relatively few parameters compared to the total number of operations. In the last few years, the adoption of CNN has exploded due to two important developments. On the one hand, many large labeled datasets such as Cityscapes [7] have become available for the use of training and validation. On the other hand, many generations of Graphics Processing Units (GPUs) have been developed and used to execute learning Algorithms, which tremendously accelerate tuning and inference.

In this paper, we first re-implemented the end-to-end autonomous driving work in NVIDIA's paper. Based on this work, we then tried to implement a new end-to-end autonomous driving program to let the car drive autonomously in a much more complicated city simulator. We train a CNN that works beyond pattern recognition. It learns the entire processing pipeline needed to steer a vehicle (or even give throttle and brake commands). The groundwork for this project was done years ago by NVIDIA [8] in which a car drives autonomously in trails without other traffic participants. We tried to add some object detection models and semantic segmentation models to process the input images so that our car can recognize its environment and react correctly to multiple traffic cases, e.g. stopping before a red traffic light or other still cars. The training data included images from three cameras coupled with left and right steering commands together with throttle and brake commands from a human operator. The Cityscape datasets are also used to train the object detection model.

This research paper is structured into eight distinct sections. Beginning with the initial segment, we provide a succinct introduction to the present study. Subsequently, we delve into a detailed exposition of the working environment and simulators employed within our project. The third section is dedicated to elucidating the data collection process. Following this, we expound upon the network architecture. In the fifth section, we clarify the procedure for aggregating and processing outputs generated by the YOLOv5 model and detection in order to derive the ultimate control command. The sixth section highlights the limitations encountered during the training and testing sessions. The seventh section presents the conclusive simulation results conducted on both images and the simulator. Finally, the concluding section of this scholarly work encompasses discussions and a comprehensive summary of our findings.

II. Overview of programming environment and simulators

The programming environment is Python because there are many available packages like Pytorch and Torchvision which can provide pre-trained CNN or RNN like Resnet50. With those pre-trained Networks, we can use them directly after changing their last output fc-layer. Besides, the package of Dataset can also help with the construction of datasets for training.

The two simulators used for the project are different.

One is provided by SudKul [9] and the map is shown in picture 1. When collecting images for training, the user just

This work was supervised by Zhou, Liguo, who is with the Department of Infomation, Technical University of Munich, Munich, 80333 Germany (email: liguo.zhou@tum.de).

Danyang Long is with the Department of Mathematics, Technical University of Munich, Munich, 80333 Germany (e-mail: danyang.long@tum.de).

Hao Yang is with the Department of Information, Technical University of Munich, Munich, 80333 Germany (e-mail: hao.yang@tum.de).

Fig. 1. a map of simulator 1 [9]

Fig. 2. a map of simulator 2 [10]

needs to press the record button and the picture will be saved in a file and the name of the pictures will be collected and saved in a text file.

The other one is given by a team of Liguo Zhou [10]. In this simulator, there are different kinds of traffic lanes and objects. Our car should not only obey the traffic rules but also avoid some other vehicles around it that do not obey the traffic rules. For example, a vehicle may hit our car from behind when our car stops before the red traffic light.

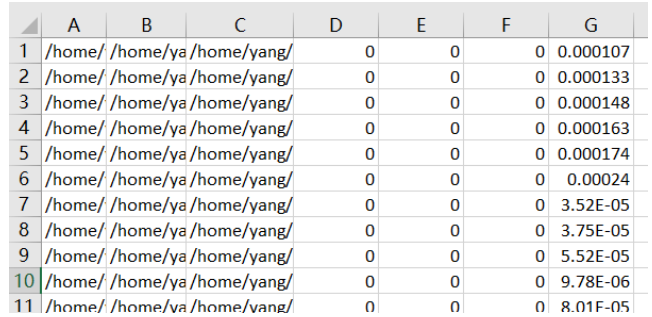
III. Data collection

In our project, we used three datasets for training different models (two CNNs and one Yolov5).

A. Datasets for CNN one

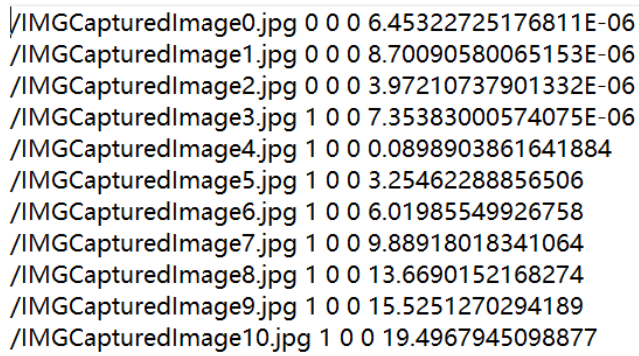
Before constructing new networks for the given simulator from Liguo Zhou, we tried to implement programs to check the result of paper [8]. The network used in this part is Resnet50 and the input size should be 224x224. The output of the simulator is three images per second that come from two side-facing and one front-facing camera. The resolution is 320x160 and there is one file named " $driving_log.csv$ " containing the saving paths of images and the information about the throttle, brake, steering angle, and the velocity of the car. Since the image size is different from the requirement of resnet model and the image is too simple, we did a resize process and took some augmentation methods to preprocess the images.

First of all, we used the letterbox function to resize the image. We first determine the longer dimension (width or height) of the image, then resize the longer dimension to the desired length. Next, we proportionally scale the other dimension. Finally, we place the resized image in the center of a blank frame of the target size and fill the symmetrical empty areas with black, resulting in an image that has been resized without altering its original ratio.



	A	B	C	D	E	F	G
1	/home/	/home/ya	/home/yang/	0	0	0	0.000107
2	/home/	/home/ya	/home/yang/	0	0	0	0.000133
3	/home/	/home/ya	/home/yang/	0	0	0	0.000148
4	/home/	/home/ya	/home/yang/	0	0	0	0.000163
5	/home/	/home/ya	/home/yang/	0	0	0	0.000174
6	/home/	/home/ya	/home/yang/	0	0	0	0.00024
7	/home/	/home/ya	/home/yang/	0	0	0	3.52E-05
8	/home/	/home/ya	/home/yang/	0	0	0	3.75E-05
9	/home/	/home/ya	/home/yang/	0	0	0	5.52E-05
10	/home/	/home/ya	/home/yang/	0	0	0	9.78E-06
11	/home/	/home/ya	/home/yang/	0	0	0	8.01E-05

Fig. 3. a screenshot of the CSV file



```
/IMGCapturedImage0.jpg 0 0 0 6.45322725176811E-06
/IMGCapturedImage1.jpg 0 0 0 8.70090580065153E-06
/IMGCapturedImage2.jpg 0 0 0 3.97210737901332E-06
/IMGCapturedImage3.jpg 1 0 0 7.35383000574075E-06
/IMGCapturedImage4.jpg 1 0 0 0.0898903861641884
/IMGCapturedImage5.jpg 1 0 0 3.25462288856506
/IMGCapturedImage6.jpg 1 0 0 6.01985549926758
/IMGCapturedImage7.jpg 1 0 0 9.88918018341064
/IMGCapturedImage8.jpg 1 0 0 13.6690152168274
/IMGCapturedImage9.jpg 1 0 0 15.5251270294189
/IMGCapturedImage10.jpg 1 0 0 19.4967945098877
```

Fig. 4. a screenshot of the txt file

Then we do image augmentation to make the images different and contain more information or noise. It includes random flips, random shadows, random brightness, and adding Gaussian noise. We also randomly chose one of the three images produced at the same time. This helps us get better generalization results.

B. Datasets for CNN two

The model to output the steering angle command in the program of simulator two is still Resnet50, but the environment of the simulator is much more complicated. On one hand, the road is no longer a one-way lane but consists of two-way lanes, and sometimes even multiple lanes with yellow and white lines. Cars need to ensure they stay within their lanes but also need to change lanes or brake when there's a car ahead and an empty lane beside them. On the other hand, the new map includes intersections, and cars must determine whether to make turns at these intersections. If they do, they need to enter the correct new lane rather than driving into opposing traffic.

To make the training results of the model more robust, it's necessary to collect data for various situations that cars might encounter while driving and the corresponding manual control data for handling these situations. For example, when a vehicle gets too close to the lane divider or a car in the adjacent lane, the simulator should output steering instructions in the opposite direction to keep the car safely in the center of the lane. During turning maneuvers, continuous small-angle steering instructions need to be output to maintain the turn without oversteering.

Group	Classes
flat	road · sidewalk · parking+ · rail track+
human	person* · rider*
vehicle	car* · truck* · bus* · on rails* · motorcycle* · bicycle* · caravan*+ · trailer*+
construction	building · wall · fence · guard rail+ · bridge+ · tunnel+
object	pole · pole group+ · traffic sign · traffic light
nature	vegetation · terrain
sky	sky
void	ground+ · dynamic+ · static+

Fig. 5. class definition of Cityscape Dataset

```
labels = [
    #       name                    id   trainId   category        catId     hasInstances   ignoreInEval
    Label(  'unlabeled'             ,  0 ,     255 , 'void'        , 0       , False        , True
    Label(  'ego vehicle'           ,  1 ,     255 , 'void'        , 0       , False        , True
    Label(  'rectification border'  ,  2 ,     255 , 'void'        , 0       , False        , True
    Label(  'out of roi'            ,  3 ,     255 , 'void'        , 0       , False        , True
    Label(  'static'                ,  4 ,     255 , 'void'        , 0       , False        , True
    Label(  'dynamic'               ,  5 ,     255 , 'void'        , 0       , False        , True
    Label(  'ground'                ,  6 ,     255 , 'void'        , 0       , False        , True
    Label(  'road'                  ,  7 ,       0 , 'flat'        , 1       , False        , False
    Label(  'sidewalk'              ,  8 ,       1 , 'flat'        , 1       , False        , False
    Label(  'parking'               ,  9 ,     255 , 'flat'        , 1       , False        , True
    Label(  'rail track'            , 10 ,     255 , 'flat'        , 1       , False        , True
    Label(  'building'              , 11 ,       2 , 'construction' , 2      , False        , False
    Label(  'wall'                  , 12 ,       3 , 'construction' , 2      , False        , False
    Label(  'fence'                 , 13 ,       4 , 'construction' , 2      , False        , False
    Label(  'guard rail'            , 14 ,     255 , 'construction' , 2      , False        , True
    Label(  'bridge'                , 15 ,     255 , 'construction' , 2      , False        , True
```

Fig. 6. parameter of defining classes of Cityscape Dataset

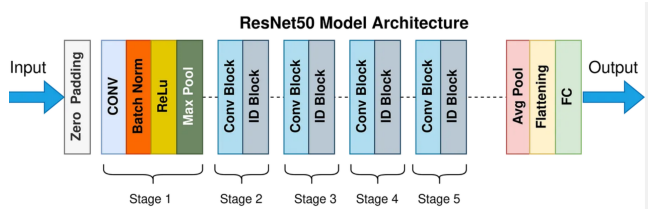


Fig. 7. Structure of Resnet50 [11]

Furthermore, due to the complexity of the simulator's map compared to NVIDIA's simulator, there is a significant delay in rendering images. This delay makes manual control of the car more challenging as the commands and scenes do not align. To address this issue, we reduced the number of cameras from three to just one front-facing camera while increasing the image resolution to 1280×480 to improve the accuracy of object detection on the images. These improvements greatly reduce the latency observed during data collection, but the performance of the autonomous driving testing mode did not see significant improvements.

In the end, this simulator outputs an image with a resolution of 1280×480 each time. Additionally, there is a text file containing information such as the path to the image, throttle, brake, steering angle, and car speed.

C. Datasets for Yolov5

Yolov5 is a model used for object detection and semantic segmentation. In this project, its role is object detection, primarily focusing on detecting traffic lights, cars, and buses. To train this network, we require a large dataset containing images with objects belonging to the target detection classes, along with their corresponding label files. The Cityscape dataset is a publicly available dataset that meets our project requirements.

The Cityscape dataset presents a new large-scale dataset that contains a diverse set of stereo video sequences recorded in street scenes from 50 different cities, with high-quality pixel-level annotations of 5,000 frames in addition to a larger set of 20,000 weakly annotated frames. The dataset is thus an order of magnitude larger than similar previous attempts. There are a total of 30 classes in the dataset as shown in picture 5 but we only need three of them, i.e. traffic light, car, and bus. So, we need to adjust the dataset by retaining only the labels for the three categories we need and ignoring other class labels. This will reduce the complexity of model training and decrease training time.

To adjust the class definition, we should follow the link to the repository given on the website of Cityscape and then use the label.py file there. We change the "trainID" parameter to "255" and the "ignoreInEval" parameter to "True" if we don't want the class. Otherwise, we set "trainID" to the desired number representing the class and set "ignoreInEval" to "False". Then we execute the createTrainIdLabelImage.py file and the desired dataset with its label files will be output. After this step, we still need to execute one program to convert the format of cityscape dataset to the format that yolov5 can process. This file is named $2yolo_filter.py$ and the purpose of this program is to separate the originally mixed images and label images into different folders. It also generates three separate text files, each containing information about the images and their corresponding labels based on the image names. These text files are used to index the images and labels during training, validation, and testing.

IV. Network Architecture

There are two networks used in this project, which are Resnet50 and Yolov5. The former has a relatively simple structure and is a typical classification neural network model. The latter, on the other hand, has a more complex structure, incorporating various specialized model architectures, e.g. CSP structure and SPP module, and its parameter count is significantly larger than that of the former.

A. Resnet50

ResNet, short for Residual Networks, stands as one of the most widely acclaimed Convolutional Neural Network (CNN) architectures. Its groundbreaking introduction in 2015 by Microsoft Research, as detailed in the paper authored by He et al., shattered numerous records in the field of deep learning. As shown in picture 8, the ResNet-50 architecture can be broken down into 6 parts:

1) *Input Pre-processing*
   Before images are inputted into the model, we perform some data augmentation to allow our model to be more robust. A random flip, random rotation, random contrast, random shadow, random brightness, and some Gaussian noise are used to make the image set more varied.

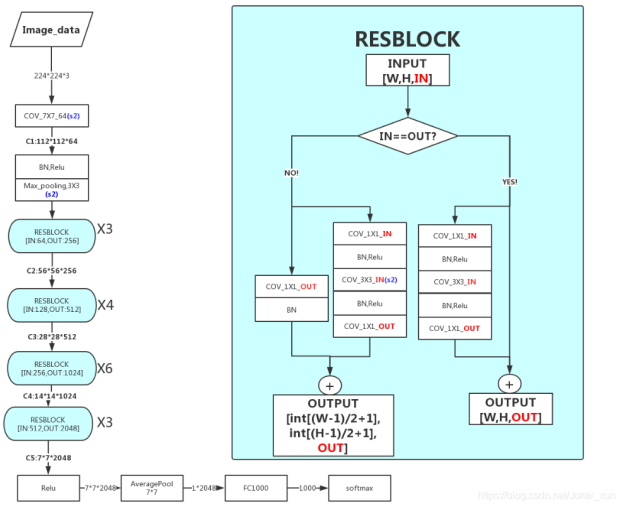


Fig. 8. Architecture of Resnet50 [13]

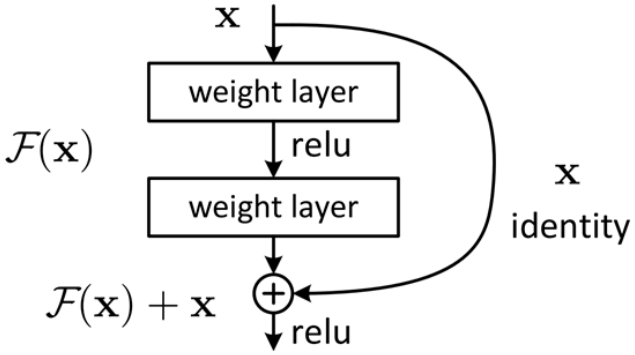


Fig. 9. Definition of residual block [12]

The parameters of the functions are probabilities, i.e. the chance that an image will undergo the selected transformation.

2) *Cfg[0] blocks*
This block comprises a convolutional layer followed by two Identity layers. To enhance numerical stability, a kernel constraint enforces weight normalization at regular intervals. Batch Normalization layers are inserted between each pair of consecutive layers. As shown in the picture 10, the identity layer means that the output is unchanged from the input of the function.
Input Shape : (56,56,64)
Output Shape : (56,56,256)

3) *Cfg[1] blocks*
This block consists of one Convolutional Layer and two Identity Layers, resembling the structure of the Cfg0 blocks. The primary distinction lies in the increased number of out channels for both the Convolutional and Identity layers.
Input Shape : (56,56,256)
Output Shape : (28,28,512)

4) *Cfg[2] blocks*
This block consists of 1 Convolutional layer followed by 5 Identity layers. It holds particular significance within the ResNet architecture, as variations of the model often diverge in this block.
Input Shape : (28,28,512)
Output Shape : (14,14,1024)

5) *Cfg[3] blocks*
In this network, the final set of Convolutional Layer blocks consists of 1 Convolutional Layer followed by 2 Identity Layers.
Input Shape : (14,14,1024)
Output Shape : (7,7,2048)

6) *Fully-Connected layer*
In this block, we employ an Average Pooling Layer, a Dropout Layer, and a Flatten Layer. Subsequently, the feature map undergoes flattening and is passed into a Fully Connected Layer, which serves as the foundation for producing predictive outcomes. A Softmax activation function is applied to yield logits and probabilities.
In the initial task of our project, the sole objective entails generating a steering angle output. Consequently, the output channel has been configured to a single channel. However, in the subsequent task, we necessitate the generation of three distinct signals: throttle, brake, and steering angle. To facilitate this, we have set the parameter "CLASS TYPE" (or output channel) to three, aligning with the multifaceted nature of the second task as outlined in our research paper.
Input Shape : (7,7,2048)
Output Shape : (1, CLASS TYPES)

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times23$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

Fig. 10. Architecture of Resnet series [12]

In the ResNet architecture, different versions utilize varying quantities of configuration blocks denoted as 'Cfg blocks' at different hierarchical levels, as illustrated in the accompanying figure 10. A comprehensive breakdown of these configurations is presented below to provide a thorough understanding of the network's architectural variations.

B. Yolov5

YOLO (You Only Look Once) is a cutting-edge real-time object detection algorithm introduced by Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi in 2015. This revolutionary algorithm was initially pre-trained on the COCO dataset and stands out by utilizing a single neural network to process entire images. The core idea involves dividing the image into distinct regions, followed by predicting probabilities and bounding boxes for each of these regions.

One of the hallmark features of YOLO is its exceptional blend of speed and accuracy, making it a formidable choice

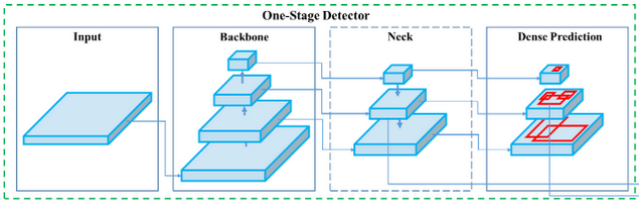


Fig. 11. Single-Stage object detector [14]

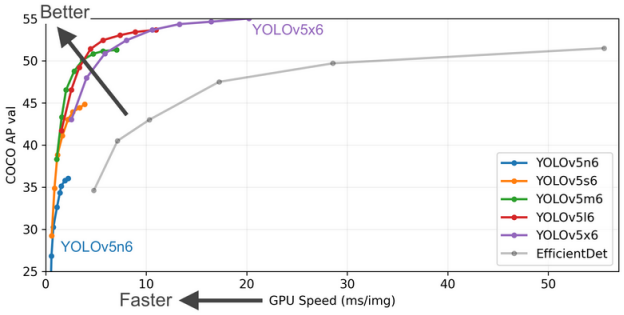


Fig. 12. Performance of YOLOv5 different sizes models [5]

for a wide range of applications, including healthcare, security surveillance, and autonomous driving. Over the years, the Ultralytics team has been dedicated to enhancing this model, leading to the release of several improved versions.

In this paper, we delve into the fifth iteration of the YOLO algorithm, YOLOv5. This version represents the latest evolution of the YOLO series, promising even greater performance and capabilities. We use the algorithm in our project to do real-time object detection for the autonomous driving simulator.

1) Object detector: There are two types of object detection models: two-stage object detectors and single-stage object detectors.

A two-stage object detector follows a distinctive two-step methodology for the detection and localization of objects in images. In the initial stage, the detector generates a collection of region proposals within the input image, which serve as potential bounding boxes encompassing objects of interest. While Selective Search remains a conventional method for region proposal, contemporary two-stage detectors often employ more sophisticated techniques like Region Proposal Networks (RPNs).

In the subsequent stage, the region proposals undergo refinement and classification into specific object categories. This process involves feature extraction from each proposal, followed by classification using a dedicated network to assign precise class labels (e.g., ”cat,” ”dog”) to each proposal. Additionally, a regression network fine-tunes the bounding box coordinates to achieve accurate localization of the objects within the proposals.

Two-stage object detectors find their niche in scenarios where achieving high detection accuracy is of paramount importance. This includes applications like autonomous driving, medical imaging, and advanced surveillance systems, where precision plays a critical role. However, it’s worth noting that their computational complexity may render them less suitable for real-time applications.

A single-stage object detector is a type of computer vision model designed for the rapid and efficient detection of objects within images. Unlike two-stage detectors, which involve a multi-step process, single-stage detectors directly predict bounding boxes and object class labels in a single pass through the neural network. These detectors are known for their speed and simplicity, making them well-suited for real-time applications, such as object detection in video streams or on embedded devices.

One of the most popular architectures for single-stage object detection is the You Only Look Once (YOLO) model, which divides the input image into a grid and assigns bounding boxes and class probabilities to each grid cell. This approach allows for simultaneous object localization and classification, making it efficient and effective for a wide range of applications, including autonomous vehicles, surveillance, and robotics.

A picture of a single-stage object detector is shown in 11. There are four components of this architecture: Input, Backbone, Neck, and a Head to make Dense Prediction:

∗ Backbone

The backbone serves as a pre-trained network tasked with extracting intricate feature representations from images. Its primary purpose is to decrease the spatial resolution of the image while simultaneously augmenting its feature (channel) resolution.

∗ Neck

The function of the model neck is feature pyramids extraction. This helps the model with good generalization to objects of different sizes and scales.

∗ Neck

The model head serves as the final stage for performing crucial operations. It applies anchor boxes to feature maps and produces the ultimate output, including class predictions, objectness scores, and bounding box coordinates.

2) YOLOv5 Architecture: The Yolov5 model was released with five different versions and version l is used in this paper:

- n for extra small (nano) size model
- s for small size model
- m for medium size model
- l for large-size model
- x for extra large size model

The models exhibit no operational differences, except for variations in the number of layers and parameters, as illustrated in the picture 13. They all consist of the same 3 components: CSP-Darknet53 as a backbone, SPP and PANet in the model neck, and the head used in YOLOv4. The detailed information about the network architecture is shown in picture 14.

- CSP-Darknet53

YOLOv5 employs CSP-Darknet53 as its backbone architecture, which is essentially the convolutional network Darknet53 originally used as the backbone for YOLOv3. In CSP-Darknet53, the authors have implemented the Cross Stage Partial (CSP) network strategy. CSPNet, an abbreviation for Cross Stage Partial Networks, addresses the issue of redundant gradient information present in other well-known convolutional neural network backbone frameworks. It achieves

Model	size (pixels)	mAP^{val} 0.5:0.95	mAP^{val} 0.5	Speed CPU b1 (ms)	Speed V100 b1 (ms)	Speed V100 b32 (ms)	params (M)	FLOPs @640 (B)
YOLOv5n	640	28.0	45.7	**45**	**6.3**	**0.6**	**1.9**	**4.5**
YOLOv5s	640	37.4	56.8	98	6.4	0.9	7.2	16.5
YOLOv5m	640	45.4	64.1	224	8.2	1.7	21.2	49.0
YOLOv5l	640	49.0	67.3	430	10.1	2.7	46.5	109.1
YOLOv5x	640	50.7	68.9	766	12.1	4.8	86.7	205.7
YOLOv5n6	1280	36.0	54.4	153	8.1	2.1	3.2	4.6
YOLOv5s6	1280	44.8	63.7	385	8.2	3.6	12.6	16.8
YOLOv5m6	1280	51.3	69.3	887	11.1	6.8	35.7	50.0
YOLOv5l6	1280	53.7	71.3	1784	15.8	10.5	76.8	111.4
YOLOv5x6	1280	55.0	72.7	3136	26.2	19.4	140.7	209.8
+ TTA	1536	**55.8**	**72.7**	-	-	-	-	-

Fig. 13. Detailed information of YOLOv5 different sizes models [14]

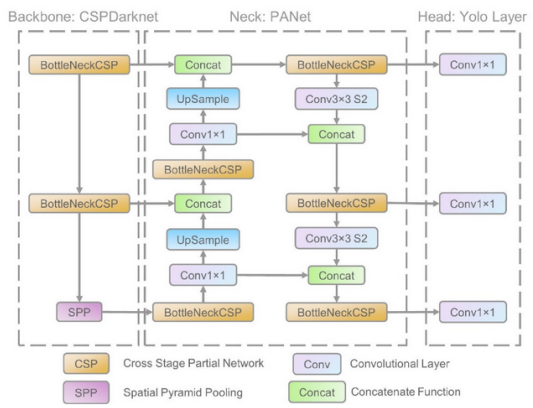


Fig. 14. Network Architecture for YOLO v5 [15]

this by seamlessly integrating gradient variations across the feature maps, thereby reducing both model parameters and floating-point operations per second (FLOPS). This approach strikes a balance between inference speed and accuracy while minimizing the model's size.

In its essence, CSPNet draws inspiration from the DenseNet architecture. It duplicates feature maps from the base layers and directs these duplicates through dense blocks to the subsequent stages, effectively isolating feature maps from the base layers. This strategy effectively mitigates the vanishing gradient problem, a challenge encountered in very deep networks where the propagation of lost signals becomes increasingly difficult. It promotes feature propagation and encourages the network to reuse features, ultimately leading to a reduction in the total number of network parameters. The architecture is shown in picture 15.

• FOCUS Module

The Focus module in yolov5 performs image slicing before entering the backbone network. Specifically, it extracts values from an input image at every other pixel, akin to nearest-neighbor downsampling. This operation yields four complementary images that are roughly the same in dimensions, preserving all the original information. Consequently, the spatial information in terms of width (W) and height (H)

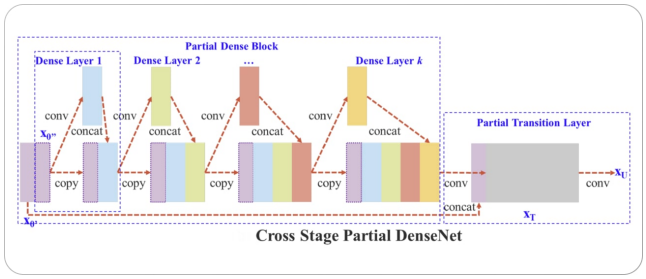


Fig. 15. Architecture of Cross Stage Partial DenseNet [16]

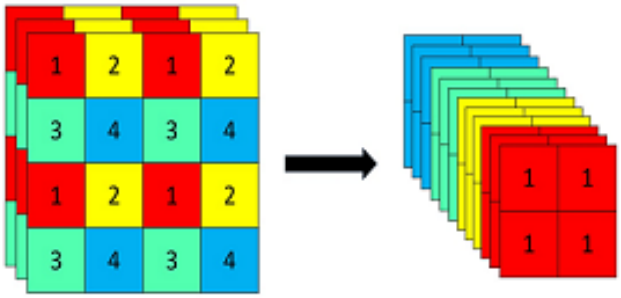


Fig. 16. process of focus module [17]

is concentrated within the channel space. The input channels are expanded fourfold, resulting in a concatenated image with 12 channels, as opposed to the original RGB three-channel format. Subsequently, the obtained new image is subjected to a convolutional operation, ultimately producing a downsampled feature map with twice the spatial resolution without any loss of information, as per the paper's methodology.

Focus module is used to reduce FLOPS, it has nothing to do with mAP. One Focus layer can replace 3 yolov3/4 layers, not just one layer.

• FPN + PAN Structure As shown in the picture 17, FPN (Feature Pyramidal Network) operates as a top-down model, transmitting high-level features downwards, while lower-level features do not influence the upper-level features. In this FPN approach, the top-down information flow occurs layer by layer, incurring greater computational complexity. PAN addresses this issue.

PANet (Path Aggregation Network), a feature pyramid network, was employed in prior iterations of YOLO, such as YOLOv4, to enhance information propagation and aid in precise pixel localization for mask prediction. In YOLOv5, this network has undergone a modification involving the application of the CSPNet strategy, as illustrated in the network architecture diagram.

In short, the FPN architecture effectively captures high-level semantic features from the top layers down to the bottom, while the PAN architecture excels at capturing precise positional features from the bottom layers up.

• SPP Module

The SPP (Spatial Pyramid Pooling) block, shown in picture 18, is designed to aggregate input information and produce a fixed-length output. It offers the advantage of significantly expanding the receptive field while segregating the most rele-

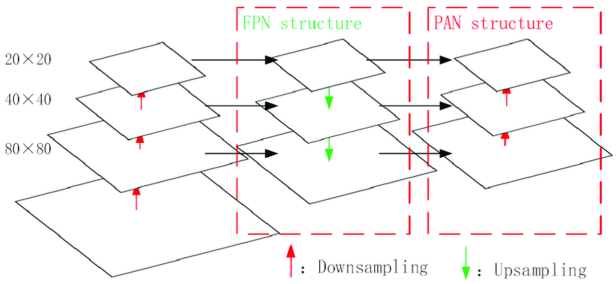


Fig. 17. Process of FPA+PAN [17]

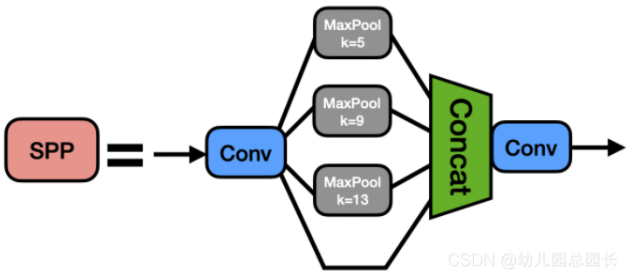


Fig. 18. Process of SPP [17]

vant contextual features without compromising network speed. This block has found application in prior iterations of YOLO, such as YOLOv3 and YOLOv4, where it was employed to separate crucial features from the backbone.

However, in the context of YOLOv5 (versions 6.0 and 6.1), a modified variant known as SPPF (Spatial Pyramid Pooling Fusion) has been introduced. SPPF is specifically designed to enhance the network's speed, while still harnessing the benefits of spatial pyramid pooling for feature extraction.

3) Activation Function: The selection of an activation function plays a pivotal role in the architecture of any deep learning model. In the case of YOLOv5, the authors opted for the SiLU (Sigmoid Linear Unit) activation function, commonly known as the swish activation function, for use in convolutional operations within the hidden layers. Conversely, the Sigmoid activation function was employed in convolutional operations within the output layer.

Swish activation function:

$$y_{mish} = x \times \tanh(\ln(1 + e^x))$$

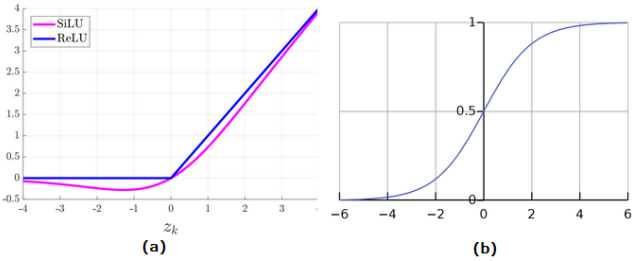


Fig. 19. Graph of the activation functions used in YOLOv5. (a) SiLU function graph. (b) Sigmoid function graph [14]

Fig. 20. function of NMS [18]

4) NMS Method: Non-maximum suppression (NMS) is a crucial technique in computer vision and object detection tasks. Its primary purpose is to eliminate redundant or overlapping detections to retain only the most relevant and distinct ones. NMS operates by comparing the confidence scores or probabilities associated with each detection and considering their spatial overlap. The basic idea behind NMS can be summarized as follows:

- Given a set of detected objects (e.g., bounding boxes) along with their associated confidence scores, NMS first sorts these detections based on their confidence scores in descending order.
- It then starts with the highest-scoring detection and examines its overlap with other detections.
- If the overlap (often measured using Intersection over Union, IoU) between the current detection and another detection exceeds a predefined threshold, the lower-scoring detection is suppressed or removed.
- This process continues iteratively, moving to the next highest-scoring detection until all detections have been examined. As a result, only the most confident and non-overlapping detections are retained.

V. Object Detection

Our detection is highly dependent on the outcomes of predictions generated by YOLOv5. This convolutional neural network framework offers numerous advantages that facilitate the realization of autonomous driving simulation, with a particular emphasis on the imperative requirement for real-time performance within the context of our project. YOLOv5 demonstrates a commendable combination of high inference speed, a relatively robust detection accuracy, and a lightweight architecture characterized by a reduced parameter count. Furthermore, it possesses the capability to detect and classify multiple object classes, such as vehicles and traffic lights, which renders it exceptionally suitable for real-time applications. Consequently, we have adopted YOLOv5 as our primary detection model.

In its initial configuration, version 0.25 of the simulator renders the original image at a resolution of 1080 by 160 pixels. Owing to the low image quality, the YOLOv5 detection exhibits constraints, as it can only successfully identify a single traffic light, which, notably, is not the intended target traffic signal in the illustrative example depicted in Figure 21. Consequently, it becomes apparent that the present configuration falls short of meeting the requirements for the task of object detection, prompting our team to implement a revised iteration, denoted as version 0.3 of the simulator. Figure 22 depicts a

Fig. 21. Source Image from the Simulator v0.25[4] vs Image after YOLOv5[5] Object Detection

discernible positive outcome achieved through the utilization of the YOLOv5 object detection.

Fig. 22. Image after YOLOv5[5] Object Detection by Source Image from Simulator v0.3[4]

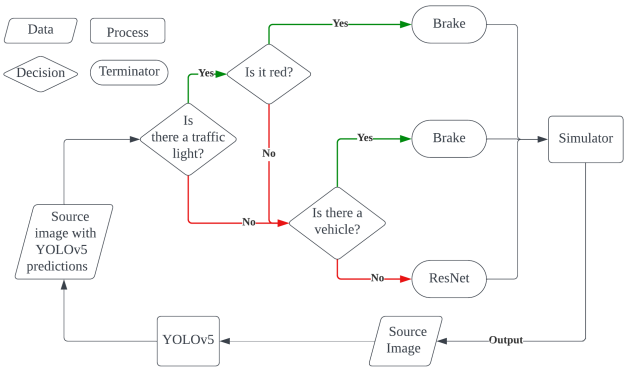


Fig. 23. Detection Flowchart

With the parameters of predictions in object detection from YOLOv5, our primary focus resided in the domains of traffic light detection and collision detection. Figure 23 is a flowchart elucidating the logical framework of the detection process. In the initial phase, the process commences with the identification of a red traffic light, subsequently proceeding to the second stage, which entails ascertaining the existence of a vehicle ahead of our own. Following the acquisition of predictive results, we proceed to transmit this information to the simulator, subsequently generating a source image. The procedure is then followed by engagement in the YOLOv5 evaluation, forming an iterative procedure.

Figure 24 delineates the disparity between the source image and the image after detection. Our initial step entails the exposition of the traffic light detection methodology.

Fig. 24. Source Image from the Simulator v0.3[4] vs Image after Detection

A. *Traffic Light Detection*

The first stage in the process of traffic light detection involves the establishment of a predetermined range for the determination of traffic lights within the field of observation and determines the distance between our vehicle and the traffic signal. Optimally, the desired outcome is for the vehicle to achieve a full cessation of motion before approaching the pedestrian crosswalk when faced with a red light scenario. Hence, based on empirical observations and several tests, we can deduce that the thin red rectangular region depicted in the lower section of Figure 24 serves as the qualifying rectangle for the targeted traffic light. Therefore, the first stage of our detection methodology entails the determination of the presence of a traffic signal within the designated rectangle area, as illustrated in the accompanying flowchart.

If a traffic signal is present within the predefined rectangle region, the ensuing course of action continues to the initiation procedures for detecting the presence of a red signal. In the current phase of the methodology, data preprocessing is undertaken. This involves the extraction of the traffic light component from the original image, followed by the standardization of the extracted image to dimensions of 32 by 32 pixels. This meticulous procedure is anticipated to enhance overall accuracy, as it capitalizes on the premise that images with resemblant images tend to perform analogous features.

With inspiration from the works of Leonng, 2020 ([1]) and CSDN, 2023 ([3]), we have developed a red detection methodology that integrates aspects from both sources. The first approach initially employed red, yellow, and green masks, which were subsequently partitioned into three distinct segments: upper, middle, and lower sections. The identification of the color was accomplished by conducting a comparative analysis of the brightness levels within each of these segments. The second approach entails the systematic computation of red, yellow, and green color values on a pixel-by-pixel basis, subsequently generating the average color level. The identification of the color was achieved through the utilization of this calculated average. In the event that the traffic signal displays a red

indication while the reality registers a green signal, we proceed to apply the red mask onto the entire image and subsequently analyze its constituent red components. The following tabular explains the performance of this methodology on the dataset obtained from the MIT self-driving course ([2]). This dataset comprises 181, 9, and 107 testing images corresponding to red, yellow, and green traffic signals, respectively.

Methodology	Accuracy (of 297 images)
Method 1	0.92929
Method 2	0.97643
Method	0.98316

In order to enhance the reliability of our findings in the realm of red signal detection, we meticulously assembled a dataset comprising 135, 60, and 132 images depicting exclusively traffic lights extracted from the source imagery of the simulator, corresponding to red, yellow, and green traffic signals, respectively. in Figure 25, we present the performance results of our methodology for red signal detection when applied to the dataset.

```
✓ 1.7s
Accuracy: 0.9847094801223242
Number of misclassified images = 5 out of 327
```

Fig. 25. Accuracy Assessment on a Self-Constructed Dataset of Traffic Signals

In the event of a red traffic light being detected within the predefined rectangle region, we engage in the manipulation of the output control parameters originating from ResNet, namely the brake, throttle, and steering angle. Subsequently, we transmit a notification denoting the red status of the traffic signal and set the control variables as follows: brake = 1, throttle = 0, and steering angle = 0. The following figure is an exemplary case of the red signal.

If no red traffic lights are present within the predefined rectangle region, the subsequent course of action aligns with the collision detection procedures outlined in the following subsection.

B. Collision Detection

In a manner akin to the approach employed during the traffic light detection procedure, we set a detection polygon area denoted as a hazardous zone. When a vehicle is in proximity to the predefined polygon detection zone, we create an additional tangent ellipse region around the approaching vehicle and inside the YOLOv5 detection rectangle. Subsequently, we ascertain the existence of an intersection area between the polygon region and the ellipse.

In the course of developing our methodology, we encountered a significant challenge: the direct utilization of predictions from the YOLOv5 detection model proved to be infeasible. The bounding boxes generated from the YOLOv5 model

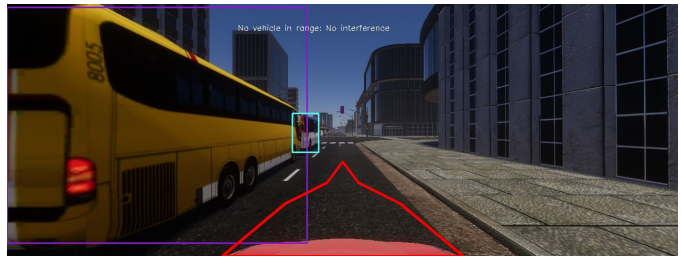


Fig. 26. Predictions from YOLOv5 with the Polygon Detection Region

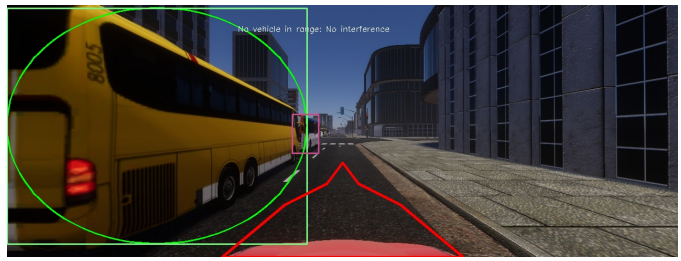


Fig. 27. Extreme Case 1

contain extraneous and inefficacious information, thereby introducing ambiguity in collision detection pertaining to both the preceding and adjacent vehicles. As an illustrated example, the bounding box of the yellow bus depicted in Figure 26 generated by YOLOv5 does not precisely conform to the actual geometric contours of the bus. Consequently, it exhibits an intersection with our designated detection polygon. Nevertheless, it is important to underscore that under such situations, the detection will set the brake to 1 when utilizing the YOLOv5 predictions directly. However, the ideal case is that the detection result does not interfere with the ResNet control outputs.

Fig. 28. Extreme Case 2

The primary objective is to minimize extraneous data within the bounding box. Initially, we explore the feasibility of employing affine and perspective transformations. However, this approach proves impractical due to the associated camera parameter loss and excessive time consumption. Consequently, we contemplate the utilization of a semantic segmentation model, with the intention of pursuing further investigation provided that sufficient time is available for dataset collection and model training. Ultimately, the most efficient and expeditious technique, ideal for our purposes, involves delineating an elliptical region within the bounding box. Several images have been subjected to rigorous testing to assess the viability of our collision detection methodology. Presented

herein are two exemplary extreme scenarios, which illustrate the robust performance of our approach in the majority of cases. Nonetheless, it is worth noting a specific limitation that warrants consideration. In the context of Figure 28, when situated at the position of the blue small sedan, our collision detection system may produce false positives due to the presence of an oncoming red vehicle traveling in the opposing direction.

In accordance with the procedural representation illustrated in Figure 23, when positive collision predictions are presented, a notification signaling the 'A vehicle is in front of us' status of the traffic signal is transmitted, concomitant with the establishment of the following control variables: brake = 1, throttle = 0, and steering angle = 0. In the absence of any detection, the system continues to adhere to the control protocol outlined by the ResNet framework.

VI. Limitations

The principal limitations inherent to our project primarily revolve around the simulation tool we utilized, which demonstrates its performance during both the training and testing sessions.

A. Training Session

1) The duration of the traffic light is excessively long, resulting in the need for a larger dataset and less effective training data.
2) Segments within a dataset exhibit discontinuity as illustrated in Figure 29. This issue will result in an inadequate training dataset, non-continuous weight values, and a prolonged duration required for the examination of the training dataset.

Fig. 29. Discontinuous Dataset

3) Exclusive utilization of integer inputs is mandated for keyboard users, yet occasional issues arise with regard to brake functionality for users employing controllers. In the context illustrated in Figure 30, the number of control variables was deliberately randomized subsequent to the brake action of a controller.

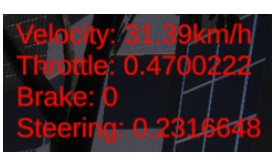


Fig. 30. Fail to Brake with a Controller

4) Other road participants exhibit non-compliance with established traffic regulations. Presented herein is a video (link) that elucidates a specific instance. Within the video, the white van is observed executing a right-hand turn while occupying the left-turn lane. The black sedan demonstrates disregard for the red traffic signal, proceeding to make a left-hand turn.

B. Testing Session

The principal concern during the testing session pertains to a pronounced latency of approximately 350 seconds within the simulator. This issue is observed from several perspectives, which will now be elucidated, encompassing aspects related to video and image.

1) Due to the substantial delay observed in our simulator, the initial detection zone proves inadequate for effective identification. Consequently, it becomes imperative to expand the surveillance area, encompassing both traffic signals and vehicles, in preparation for prospective responses by our autonomous vehicles.
2) Our vehicle initially managed to come to a halt but subsequently experienced difficulties in halting upon receiving the regulatory signal, primarily attributable to the unstable and discrete nature of the brake command. In the end, the traffic light fell outside the detection range, leading to the vehicle's failure to stop at the red light and an illegal pass.

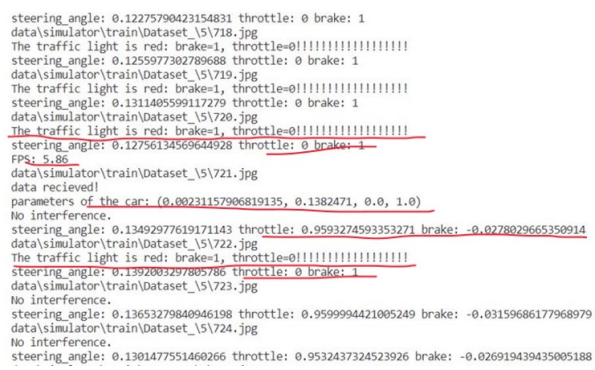


Fig. 31. Command Window of Failure Stop

3) The video (link) demonstrates that the vehicle effectively came to a halt prior to the red traffic signal; however, it subsequently collided and was propelled forward by the following car.

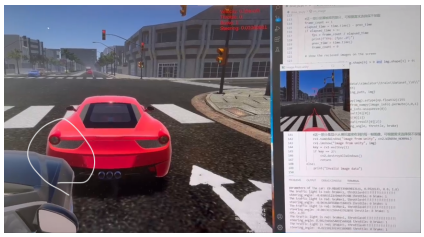


Fig. 32. Crash

While acknowledging the significant developmental potential inherent in our simulator, it is prudent to consider self-driven modifications to enhance its functionality. Additionally,

should sufficient time permit, we intend to explore optimizations of the simulator such as reducing the frame rate and image quality within the simulator.

VII. Simulation

The initial segment of this section entails the exposition of our detection methodology, wherein we elucidate our detection predictions across various experimental scenarios. The subsequent subsection encompasses both a screenshot and a video showcasing the conclusive outcome of our work, while the concluding segment of this section delves into an examination of the enhancements achieved through the course of this project.

A. Detection Result Demonstration

We present three illustrative cases. For a more comprehensive and dynamic exploration, we kindly direct you to the video located within the subsequent subsection.

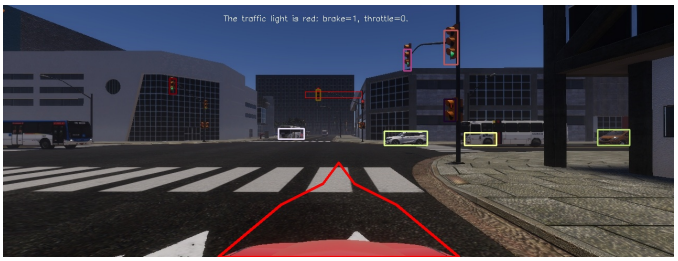


Fig. 33. Case: brake = 0 (Red Traffic Light)

Fig. 34. Case: No Interference

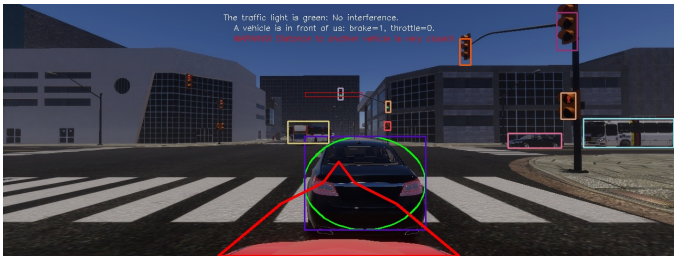


Fig. 35. Case: brake = 0 (Vehicle Detection)

B. Simulator Result Demonstration

We conducted our simulation experiments within an environment featuring a computational GPU configuration of the RTX 3080 Ti. Despite obtaining results of modest satisfaction, we successfully attained control over the vehicle's braking system, throttle, and steering angle. A visual representation of our ultimate outcomes is available in the accompanying video presentation (link).

C. Improvements

For the task of traffic light detection, it is advisable to define multiple distinct regions of interest within the image, each dedicated to detecting traffic lights independently. By generating multiple predictions from these defined regions and subsequently comparing their outcomes, the overall accuracy of traffic light detection can be significantly enhanced. As an illustration, one approach involves the simultaneous consideration of multiple traffic lights where two traffic lights are positioned directly ahead and an additional four traffic lights are distributed to the sides of the observer. subsequently, the weights are assigned to each of these traffic lights, and a threshold value is established for decision-making.

For the task of collision detection, it is worth considering that employing semantic segmentation as a methodology may yield superior outcomes due to its capacity to delineate predicted regions that closely approximate the object's dimensions, potentially resulting in enhanced performance and accuracy.

From the aspect of control, it is also possible to create an additional region wherein the brake parameter is set to 0.5 or even a linear function, designating it as a buffer area. This configuration facilitates a gradual deceleration as opposed to an abrupt stop.

VIII. Conclusion

In this project, we acquired a comprehensive understanding of the ResNet and YOLOv5 algorithms and applied them within a simulation environment. Despite encountering certain constraints, we were unable to attain our desired results with absolute precision; nonetheless, we managed to achieve a degree of functionality.

References

[1] Leongg, Ian. "Traffic Light Classifier," 2020, URL: Github repository
[2] MIT self-driving car course
[3] CSDN - OpenCV Traffic Light Classifier
[4] TUM Simulator
[5] Jocher, Glenn *et al.*, "Ultralytics/yolov5: v3.1 - Bug Fixes and Performance Improvements," *Zenodo.*, to be published. DOI: 10.5281/zenodo.4154370
[6] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. Backpropagation applied to handwritten zip code recognition. Neural Computation, 1(4):541–551, Winter 1989. .
[7] The Cityscapes Dataset: Semantic, instance-wise, dense pixel annotations of 30 classes. https://www.cityscapes-dataset.com/
[8] End-to-end learning for self-driving cars. Karol Zieba, NVIDIA Corporation, Holmdel, NJ 07735 Website.
[9] Git-lab link
[10] Git-lab link
[11] The Annotated ResNet-50
[12] K. He, X. Zhang, S. Ren and J. Sun, "Deep Residual Learning for Image Recognition," 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), Las Vegas, NV, USA, 2016, pp. 770-778, doi: 10.1109/CVPR.2016.90.
[13] Website: structure of Resnet.
[14] Website: YOLO v5 model architecture.
[15] XU, Renjie, LIN, Haifeng, LU, Kangjie, et al. A forest fire detection system based on ensemble learning. Forests, 2021, vol. 12, no 2, p. 217.

[16] YOLO: Algorithm for Object Detection Explained.
[17] https://www.cnblogs.com/boligongzhu/p/15508249.html
[18] Non Maximum Suppression: Theory and Implementation in PyTorch.