Traffic Sign Detection using Deep Learning

LY Project

Project Incharge: Prof. Maruti Zalte

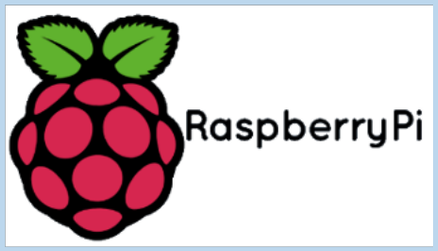
Presenters:
Aditya Shenoy- 1813057
Fenny Keniya- 1813051
Navneet Parab-1813035
Shikta Das- 1813011

Introduction

- Traffic signs contain necessary messages about vehicle safety and they show the latest traffic conditions, define road rights, forbid and allow some behaviours and driving routes, cue dangerous messages and so on.

- Being able to identify traffic signs accurately and effectively can improve the driving safety.

- This project aims to bring forward a traffic sign recognition technique on the strength of deep learning which includes the detection and classification of signs.



Objectives

- Construct an object detection code for reading traffic signs.
- Analyse and classify various types of traffic signs like stop, speed limit etc.
- Design a system that alerts the drive in real time about the various traffic signs on the road.
- Increase safety by preventing accidents.

Literature Survey 1/4

Traffic Sign Detection and Recognition Based on Convolutional Neural Network

Summary

- In this paper a traffic sign recognition method on account of deep learning is proposed, which mainly aims at circular traffic signs.
- By using image preprocessing, traffic sign detection, recognition and classification, this method can effectively detect and identify traffic signs.

Technology used

TensorFlow, CNN.

Future scope

- CNN shows great efficiency but cannot work real time.
- This is because it first finds area of interest and then object detection occurs.

Literature Survey 2/4

Comparative study of various image classification deep learning algorithms

Summary

- This research paper focused on comparing the pros and cons of various existing advanced deep learning techniques for object detection and classification like FRCNN (Faster Region based Convolutional Neural Network), YOLOv3 (You Only Look Once) and SSD (Single Shot multibox Detector) based on accuracy, time complexity, use cases and GPU requirements.

Technology used

Faster Region based Convolutional Neural Network, You Only Look Once (YOLOv3), Single Shot multibox Detector

Future scope

- The authors of the paper are optimistic about newer, more complex technologies that will make object detection easier and more accurate, each with its own niche use case.

Literature Survey 3/4

Real-Time Object Detection with Yolo

Summary

- This research paper discussed the overview of YOLO algorithm (You Only Look Once) and its working in detail for real-time object detection. by explaining the difference between CNN and YOLO algorithm which differ by their use of Classification and Regression.
- For the training of the entire model, loss functions such as Classification, Localization and Confidence were described in detail.

Technology used

You Only Look Once, Anchor Boxes

Future scope

- The authors of the paper are confident about the accuracy and speed of the YOLO algorithm but also mention localization errors as a field for improvement.

Literature Survey 4/4

Detection of Malaysian Traffic Signs via Modified YOLOv3 Algorithm

Summary

- This paper presents an implementation of deep learning framework via YOLOv3 with an inclusion of SPP prior to its final convolution layer on the Malaysian Traffic Sign images.

- Results have shown to be promising, in which a mAP value of 0.825 is obtained that indicates that the proposed CNN framework has correctly detected the traffic sign at 82.5% accuracy.

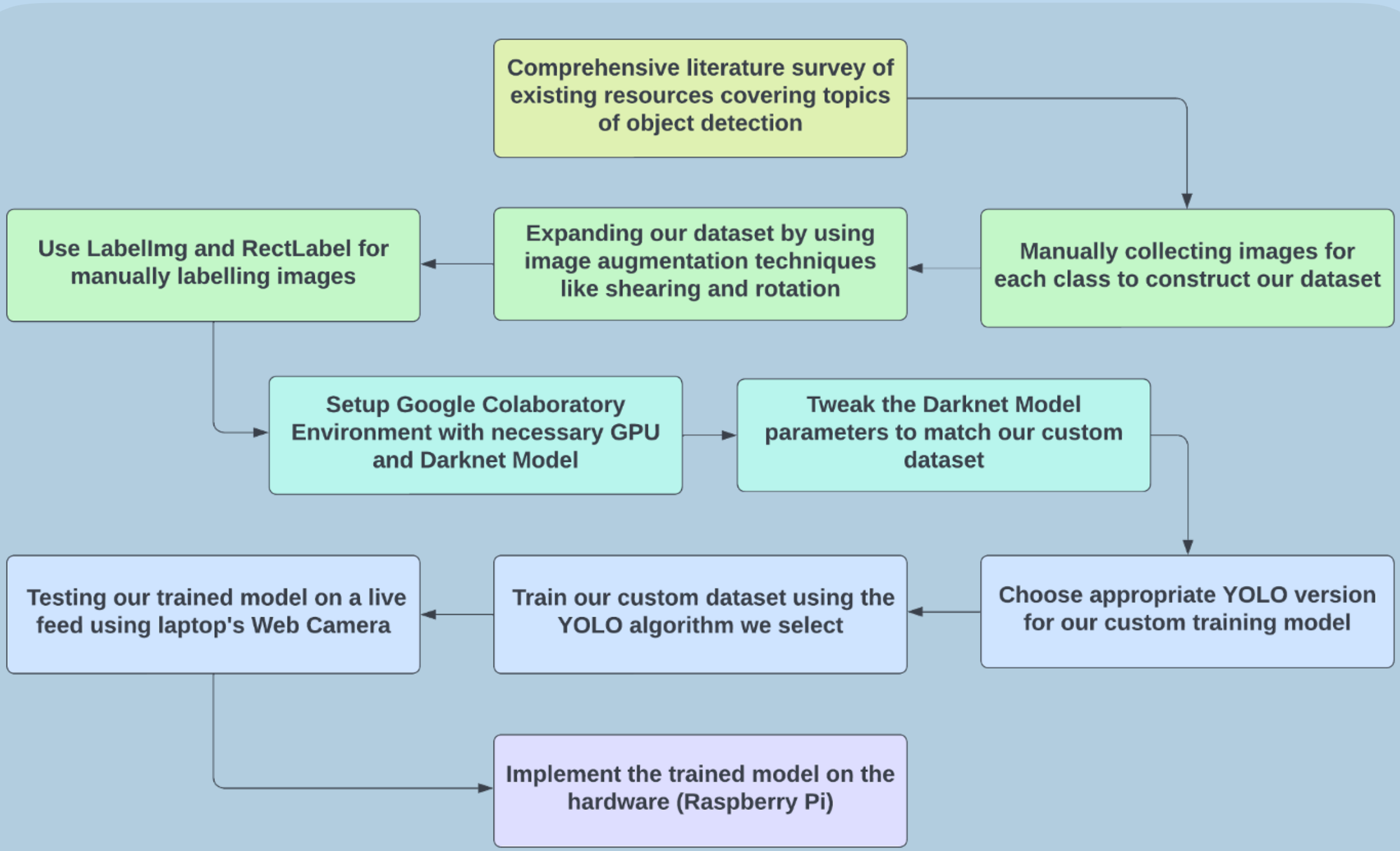
Technology used

You Only Look Once version 3 (YOLOv3) algorithm with an inclusion of SPP prior to its final convolution layer

Future scope

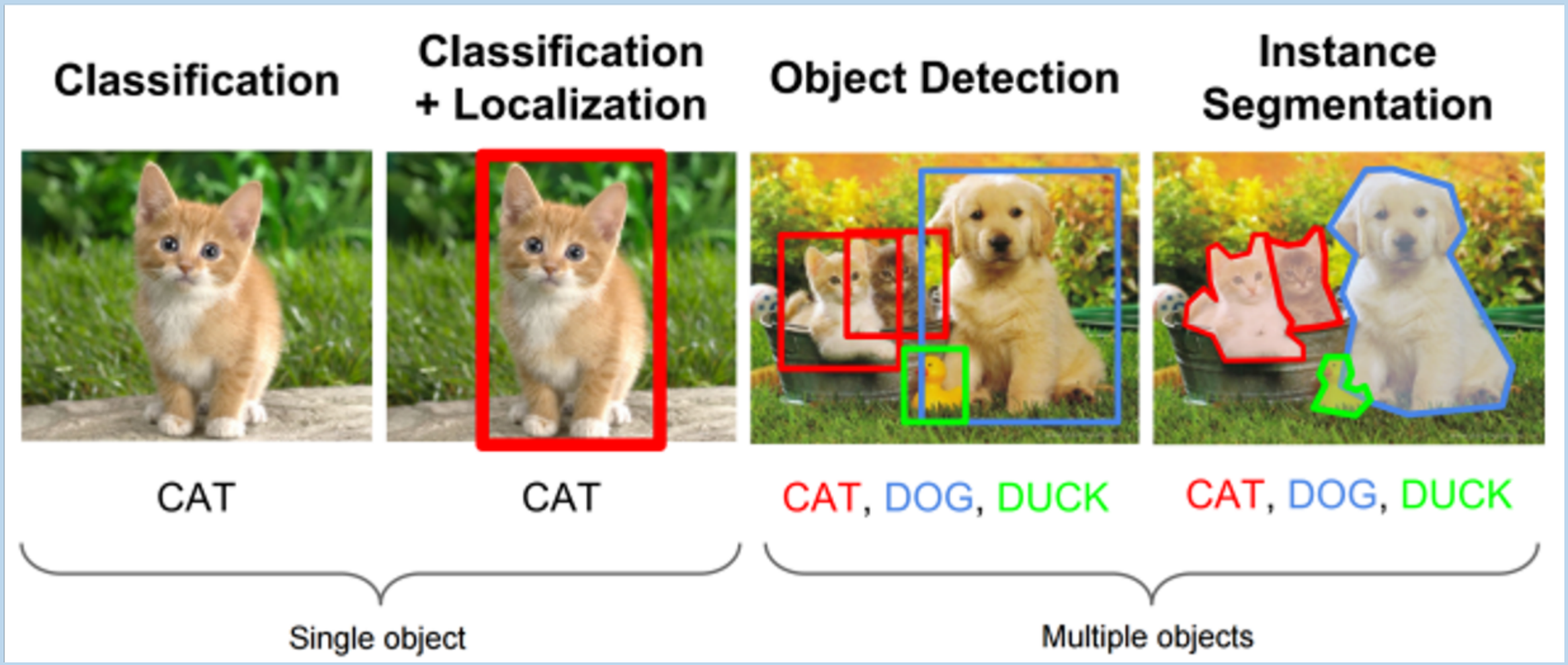
- To impove the mAP value by further fine tuning the SPP algorithm.

Project Design (Block Diagram)



Object Detection

Object detection is a computer vision technique that works to identify and locate objects within an image or video.



YOLO

- YOLO is a Convolutional Neural Network (CNN) for performing object detection in real-time. CNNs are classifier-based systems that can process input images as structured arrays of data and identify patterns between them.
- YOLO and other convolutional neural network algorithms “score” regions based on their similarities to predefined classes. High-scoring regions are noted as positive detections of whatever class they most closely identify with.

YOLO Versions Comparison

V1	V2	V3	V4
Simple & Fast	Remove the fully-connected layer at the end	Good performance over a wide range of input resolutions	Less accuracy but much higher FPS
Single monolithic network needs to take care of feature extraction, box regression and classification	Truly resolution-independent architecture	Several checkpoints: each for a different input resolutions, but in fact the network parameters stored are identical	Object detector that trains on a single GPU with a decreased mini-batch size
Divides the subject into NxN grids. Each grid represents a classifier,	Multiscale training	Score is identical to Faster-RCNN-ResNet50 but 17 times faster	Data augmentation that transforms four images of the training dataset into one
darknet framework, which is modified with 24 convolution and 2 fully connected layers	darknet 19 which has 19 convolution layers and 5 max pulling layers, with the last layer being a softmax layer for classification	darknet-53 (it has 52 convolutions) contains skip connections and 3 prediction heads	Self-Adversarial Training: the neural network changes the original image instead of the weights

Project Implementation

Obtaining the Dataset

The four traffic signs selected are as follows:

- **Speed Limit**
- **Go Slow**
- **One Way**
- **No Stop**

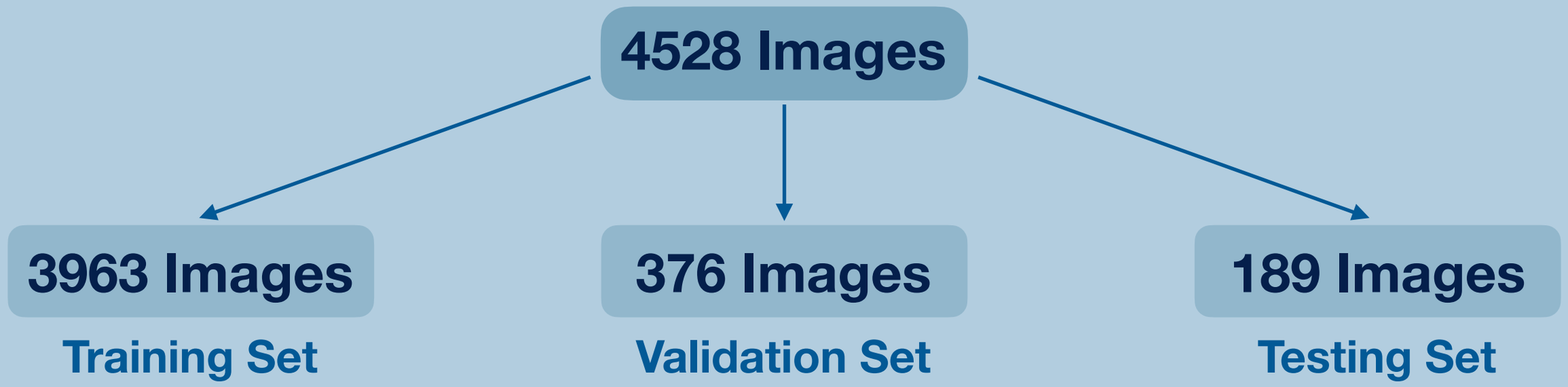
To expand our dataset further, we applied image augmentation techniques:

- **Rotation (-15° & +15°)**
- **Shearing (±15° horizontal & vertical)**

Finally, our dataset consists of **4528 images** across four classes of traffic signs.

Project Implementation

Obtaining the Dataset



All the photos in the dataset are manually clicked by us using four different cameras. The specifications of the four cameras are as follows:

1. Samsung A52, 4 cameras- 64MP, 12 MP, 5 MP, 5 MP
2. iPhone 12, 2 cameras-12 MP, 12MP
3. iPhone 13, 2 cameras-12 MP, 12MP
4. iPhone 13, 2 cameras- 12 MP, 12 MP

Project Implementation

Variability in the Dataset

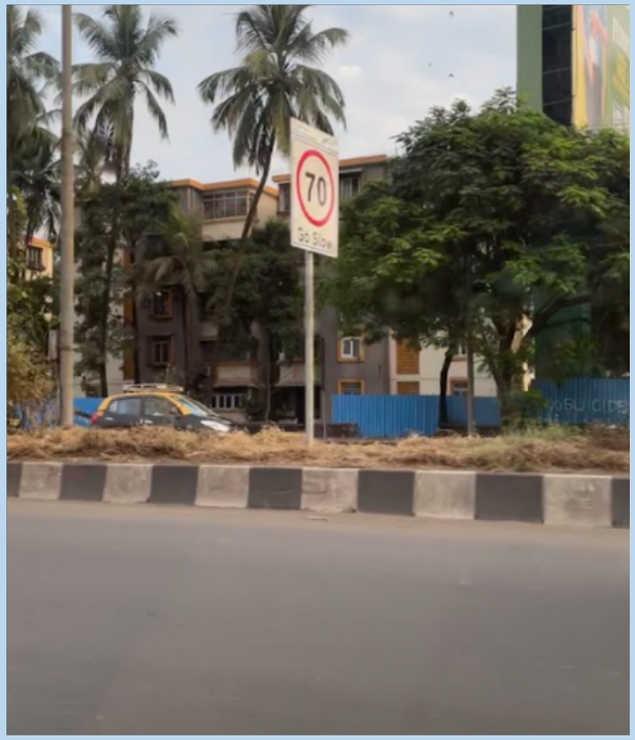
- Taking pictures with **different lighting** (day and night).
- Taking pictures from **different distances and angles**.
- Taking pictures of the traffic sign with **multiple signs in the frame**.
- Including **different types of traffic signs** of each class.
- Taking pictures from **moving vehicles as well as stationary**.

Project Implementation

Variability in the Dataset: One Way Sign

Project Implementation

Variability in the Dataset: Speed Limit Sign



Project Implementation

Variability in the Dataset: Go Slow Sign

Project Implementation

Variability in the Dataset: No Stop Sign

Project Implementation

Labelling of the Dataset

The following parameters were included in the .txt label files:

- object-class
- x - coordinate
- y - coordinate
- width
- height

```
<object-class_1> <x_1> <y_1> <width_1> <height_1>
<object-class_2> <x_2> <y_2> <width_2> <height_2>
                       ...
<object-class_n> <x_n> <y_n> <width_n> <height_n>
```

For eg. 3 0.565833 0.446250 0.595000 0.787500

No Stop | Center of the Bounding Box | Dimensions of the Bounding Box

Project Implementation

Setting up Google Colab environment with necessary GPU, mounting the drive and darknet to run our model

1- Mount the google drive

- On the google drive, create a main folder for YOLO custom detection
- Within that folder, create subfolders for backup and custom weight
- Upload the dataset zip file within the main folder

```
[ ]  #import google drive
     from google.colab import drive
```

```
#mount google drive
drive.mount('/content/drive')
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount=True).
```

```
[ ]  #check if the dataset exisit
     !ls '/content/drive/My Drive/yolo_custom_model_Training'
```

```
images.zip
```

2- Clone the Darknet

```
[ ]  !git clone 'https://github.com/AlexeyAB/darknet' '/content/drive/My Drive/yolo_custom_model_Training/darknet'
```

```
Cloning into '/content/drive/My Drive/yolo_custom_model_Training/darknet'...
remote: Enumerating objects: 15412, done.
remote: Total 15412 (delta 0), reused 0 (delta 0), pack-reused 15412
Receiving objects: 100% (15412/15412), 14.02 MiB | 5.29 MiB/s, done.
Resolving deltas: 100% (10356/10356), done.
Checking out files: 100% (2050/2050), done.
```

Project Implementation

3- Compile Darknet after enabling GPU, CUDA and OpenCV

```
[ ]  # change makefile to have GPU and OPENCV enabled
     %cd '/content/drive/My Drive/yolo_custom_model_Training/darknet'
     !sed -i 's/OPENCV=0/OPENCV=1/' Makefile
     !sed -i 's/GPU=0/GPU=1/' Makefile
     !sed -i 's/CUDNN=0/CUDNN=1/' Makefile
     !make
```

```
[ ]  %cd '/content/drive/My Drive/yolo_custom_model_Training/darknet'

     /content/drive/My Drive/yolo_custom_model_Training/darknet
```

```
[ ]  !ls

     3rdparty        DarknetConfig.cmake.in  json_mjpeg_streams.sh  scripts
     backup          darknet_images.py       LICENSE                src
     build           darknet.py              Makefile               vcpkg.json
     build.ps1       darknet_video.py        net_cam_v3.sh          video_yolov3.sh
     cfg             data                    net_cam_v4.sh          video_yolov4.sh
     cmake           image_yolov3.sh         obj
     CMakeLists.txt  image_yolov4.sh         README.md
     darknet         include                 results
```

4- Create a copy of yolov4 (or any other version of yolo you want to train)

- Need to run this once

```
[ ]  !cp cfg/yolov3.cfg cfg/yolov3_training.cfg
```

5- Create a copy of yolov4 (or any other version of yolo you want to train)

- Need to run this once

```
     #Modify YOLO archicture
     !sed -i 's/batch=1/batch=64/' cfg/yolov3_training.cfg
     !sed -i 's/subdivisions=1/subdivisions=16/' cfg/yolov3_training.cfg
     !sed -i 's/max_batches = 500200/max_batches = 2000/' cfg/yolov3_training.cfg
     !sed -i '610 s@classes=80@classes=1@' cfg/yolov3_training.cfg
     !sed -i '696 s@classes=80@classes=1@' cfg/yolov3_training.cfg
     !sed -i '783 s@classes=80@classes=1@' cfg/yolov3_training.cfg
     !sed -i '603 s@filters=255@filters=18@' cfg/yolov3_training.cfg
```

Project Implementation

DarkNet-53

- It is an open source convolutional neural network written in C and CUDA
- It is an improvement of the previous DarkNet-19 framework
- It is used as the foundation upon which all YOLOv3 algorithms are trained

	Type	Filters	Size	Output
	Convolutional	32	3 × 3	256 × 256
	Convolutional	64	3 × 3 / 2	128 × 128
1×	Convolutional	32	1 × 1	
	Convolutional	64	3 × 3	
	Residual			128 × 128
	Convolutional	128	3 × 3 / 2	64 × 64
2×	Convolutional	64	1 × 1	
	Convolutional	128	3 × 3	
	Residual			64 × 64
	Convolutional	256	3 × 3 / 2	32 × 32
8×	Convolutional	128	1 × 1	
	Convolutional	256	3 × 3	
	Residual			32 × 32
	Convolutional	512	3 × 3 / 2	16 × 16
8×	Convolutional	256	1 × 1	
	Convolutional	512	3 × 3	
	Residual			16 × 16
	Convolutional	1024	3 × 3 / 2	8 × 8
4×	Convolutional	512	1 × 1	
	Convolutional	1024	3 × 3	
	Residual			8 × 8
	Avgpool		Global	
	Connected		1000	
	Softmax			

Project Implementation

Version Testing

V1

```
 100 conv    256       3 x 3/ 1     52 x  52 x 128 ->   52 x  52 x
 101 conv    128       1 x 1/ 1     52 x  52 x 256 ->   52 x  52 x
 102 conv    256       3 x 3/ 1     52 x  52 x 128 ->   52 x  52 x
 103 conv    128       1 x 1/ 1     52 x  52 x 256 ->   52 x  52 x
 104 conv    256       3 x 3/ 1     52 x  52 x 128 ->   52 x  52 x
 105 conv     18       1 x 1/ 1     52 x  52 x 256 ->   52 x  52 x
 106 yolo
[yolo] params: iou loss: mse (2), iou_norm: 0.59, obj_norm: 1.00,
1.00
Time elapsed = 7.69420 hours
Total BFLOPS 65.304
avg_outputs = 516723
Allocate additional workspace_size = 52.43 MB
Done! Loaded 75 layers from weights-file
Learning Rate: 0.001, Momentum: 0.9, Decay: 0.0005
```

V2

```
   CUDA-version: 11010 (11020), cuDNN: 7.6.5, GPU count: 1
 OpenCV version: 3.2.0
Yolov2.3.2_training
 0 : compute_capability = 750, cudnn_half = 0, GPU: Tesla T4
net.optimized_memory = 0
mini_batch = 4, batch = 128, time_steps = 1, train = 1 12.3809 hours left
   layer   filters  size/strd(dil)      input                output
   0 Create CUDA-stream - 0
 Create cudnn-handle 0
conv     32       3 x 3/ 1    832 x 832 x   3 ->  832 x 832 x  32 0.489 BF
   1 conv     64       3 x 3/ 2    416 x 416 x  32 ->  208 x 208 x  64 1.595 BF
   2 conv     32       1 x 1/ 1    208 x 208 x  64 ->  208 x 208 x  32 0.177 BF
   3 conv     64       3 x 3/ 1    208 x 208 x  32 ->  208 x 208 x  64 1.595 BF
   4 Shortcut Layer: 1,  wt = 0, wn = 0, outputs: 416 x 416 x  64 0.008 BF
   5 conv    128       3 x 3/ 2    208 x 208 x  64 ->  104 x 104 x 128 1.595 BF
   6 conv     64       1 x 1/ 1    104 x 104 x 128 ->  104 x 104 x  64 0.177 BF
   7 conv    128       3 x 3/ 1    104 x 104 x  64 ->  104 x 104 x 128 1.595 BF
   8 Shortcut Layer: 5,  wt = 0, wn = 0, outputs: 208 x 208 x 128 0.001 BF
   9 conv     64       1 x 1/ 1    104 x 104 x 128 ->  104 x 104 x  64 0.177 BF
  10 conv    128       3 x 3/ 1    104 x 104 x  64 ->  104 x 104 x 128 1.595 BF
  11 Shortcut Layer: 8,  wt = 0, wn = 0, outputs: 52 x 52 x 128 0.001 BF
  12 conv    256       3 x 3/ 2    104 x 104 x 128 ->   52 x  52 x 256 1.595 BF
  13 conv    128       1 x 1/ 1     52 x  52 x 256 ->   52 x  52 x 128 0.177 BF
```

V3

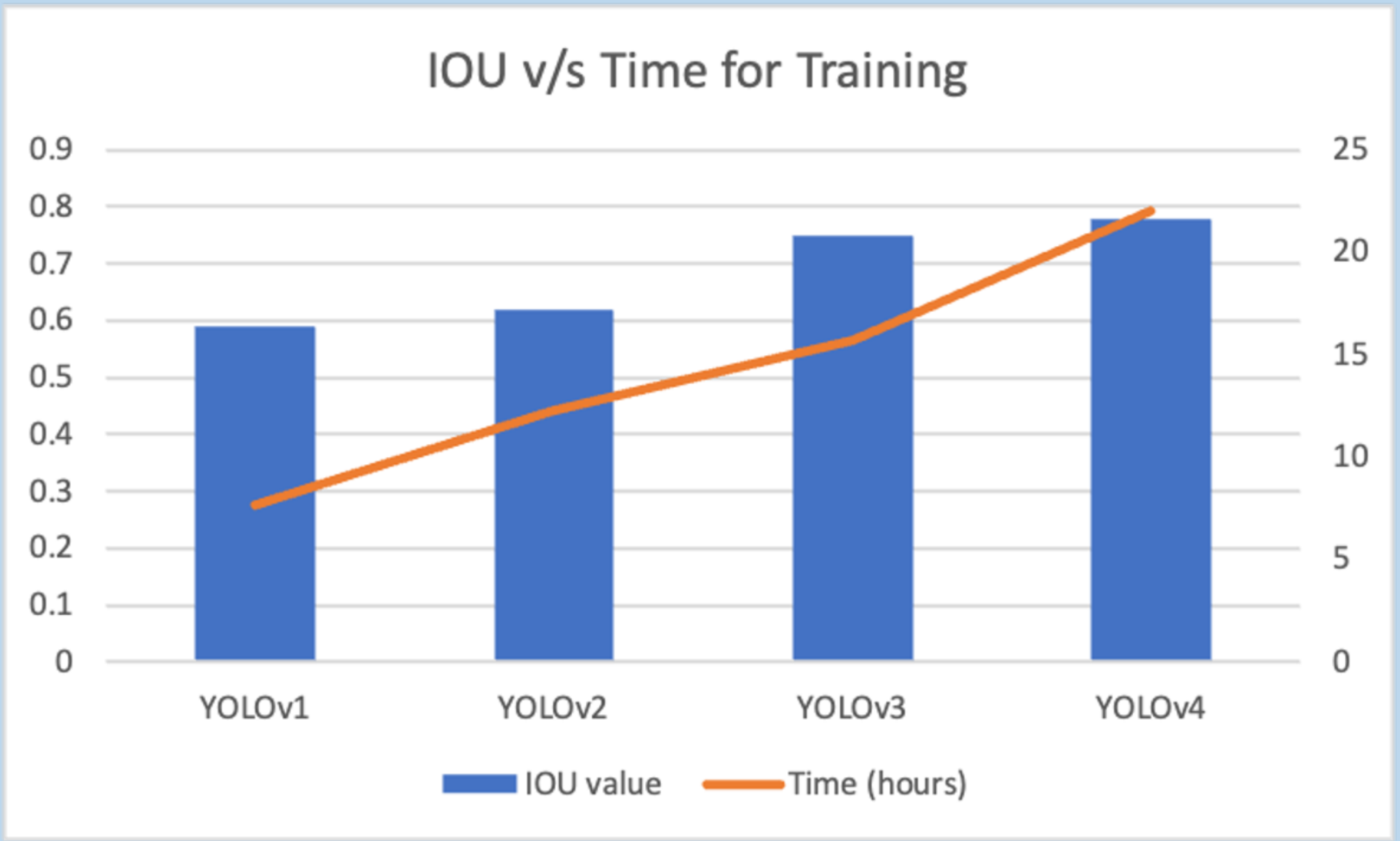
```
CUDA-version: 11010 (11020), cuDNN: 7.6.5, GPU count: 1
 OpenCV version: 3.2.0
yolov3_training
 0 : compute_capability = 750, cudnn_half = 0, GPU: Tesla T4
net.optimized_memory = 0
mini_batch = 4, batch = 64, time_steps = 1, train = 1 15.72360 hours left
   layer   filters  size/strd(dil)      input                output
   0 Create CUDA-stream - 0
 Create cudnn-handle 0
conv     32       3 x 3/ 1    416 x 416 x   3 ->  416 x 416 x  32 0.299 BF
   1 conv     64       3 x 3/ 2    416 x 416 x  32 ->  208 x 208 x  64 1.595 BF
   2 conv     32       1 x 1/ 1    208 x 208 x  64 ->  208 x 208 x  32 0.177 BF
   3 conv     64       3 x 3/ 1    208 x 208 x  32 ->  208 x 208 x  64 1.595 BF
   4 Shortcut Layer: 1,  wt = 0, wn = 0, outputs: 208 x 208 x  64 0.003 BF
   5 conv    128       3 x 3/ 2    208 x 208 x  64 ->  104 x 104 x 128 1.595 BF
   6 conv     64       1 x 1/ 1    104 x 104 x 128 ->  104 x 104 x  64 0.177 BF
   7 conv    128       3 x 3/ 1    104 x 104 x  64 ->  104 x 104 x 128 1.595 BF
   8 Shortcut Layer: 5,  wt = 0, wn = 0, outputs: 104 x 104 x 128 0.001 BF
   9 conv     64       1 x 1/ 1    104 x 104 x 128 ->  104 x 104 x  64 0.177 BF
  10 conv    128       3 x 3/ 1    104 x 104 x  64 ->  104 x 104 x 128 1.595 BF
  11 Shortcut Layer: 8,  wt = 0, wn = 0, outputs: 104 x 104 x 128 0.001 BF
  12 conv    256       3 x 3/ 2    104 x 104 x 128 ->   52 x  52 x 256 1.595 BF
  13 conv    128       1 x 1/ 1     52 x  52 x 256 ->   52 x  52 x 128 0.177 BF
  14 conv    256       3 x 3/ 1     52 x  52 x 128 ->   52 x  52 x 256 1.595 BF
  15 Shortcut Layer: 12,  wt = 0, wn = 0, outputs:  52 x  52 x 256 0.001 BF
  16 conv    128       1 x 1/ 1     52 x  52 x 256 ->   52 x  52 x 128 0.177 BF
  17 conv    256       3 x 3/ 1     52 x  52 x 128 ->   52 x  52 x 256 1.595 BF
  18 Shortcut Layer: 15,  wt = 0, wn = 0, outputs:  52 x  52 x 256 0.001 BF
```

V4

```
CUDA-version: 11010 (11020), cuDNN: 7.6.5, GPU count: 1
 OpenCV version: 3.2.0
yolov4_training
 0 : compute_capability = 370, cudnn_half = 0, GPU: Tesla K80
net.optimized_memory = 0
mini_batch = 8, batch = 64, time_steps = 1, train = 1 22.00416 hours left
   layer   filters  size/strd(dil)      input                output
   0 Create CUDA-stream - 0
 Create cudnn-handle 0
conv     32       3 x 3/ 1    608 x 608 x   3 ->  608 x 608 x  32 0.639 BF
   1 conv     64       3 x 3/ 2    608 x 608 x  32 ->  304 x 304 x  64 3.407 BF
   2 conv     64       1 x 1/ 1    304 x 304 x  64 ->  304 x 304 x  64 0.757 BF
   3 route  1                                     ->  304 x 304 x  64
   4 conv     64       1 x 1/ 1    304 x 304 x  64 ->  304 x 304 x  64 0.757 BF
   5 conv     32       1 x 1/ 1    304 x 304 x  64 ->  304 x 304 x  32 0.379 BF
   6 conv     64       3 x 3/ 1    304 x 304 x  32 ->  304 x 304 x  64 3.407 BF
   7 Shortcut Layer: 4,  wt = 0, wn = 0, outputs: 304 x 304 x  64 0.006 BF
   8 conv     64       1 x 1/ 1    304 x 304 x  64 ->  304 x 304 x  64 0.757 BF
   9 route  8 2                               ->  304 x 304 x 128
  10 conv     64       1 x 1/ 1    304 x 304 x 128 ->  304 x 304 x  64 1.514 BF
  11 conv    128       3 x 3/ 2    304 x 304 x  64 ->  152 x 152 x 128 3.407 BF
  12 conv     64       1 x 1/ 1    152 x 152 x 128 ->  152 x 152 x  64 0.379 BF
  13 route  11                                    ->  152 x 152 x 128
  14 conv     64       1 x 1/ 1    152 x 152 x 128 ->  152 x 152 x  64 0.379 BF
```

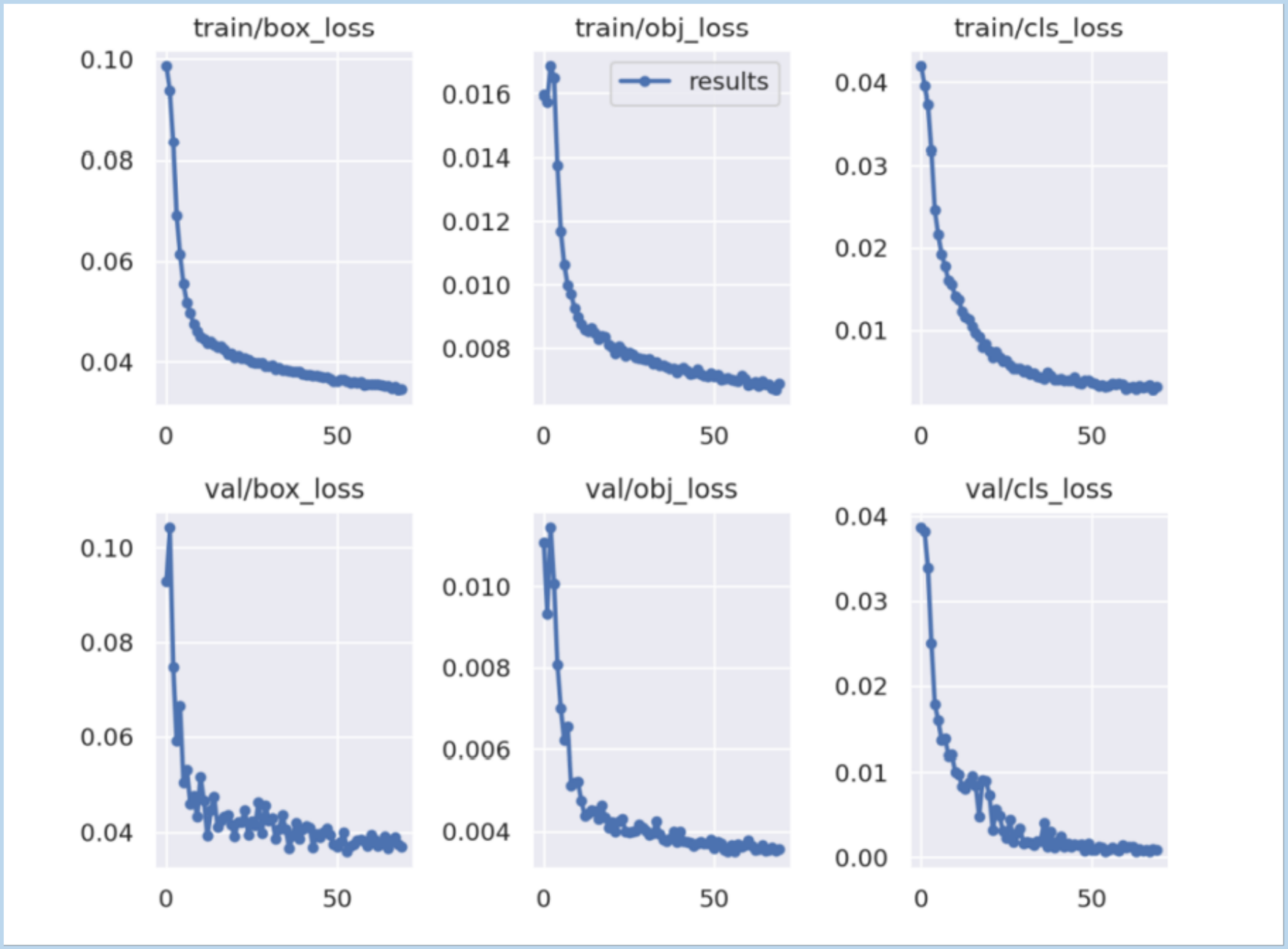
Project Implementation

Version Testing



Project Implementation

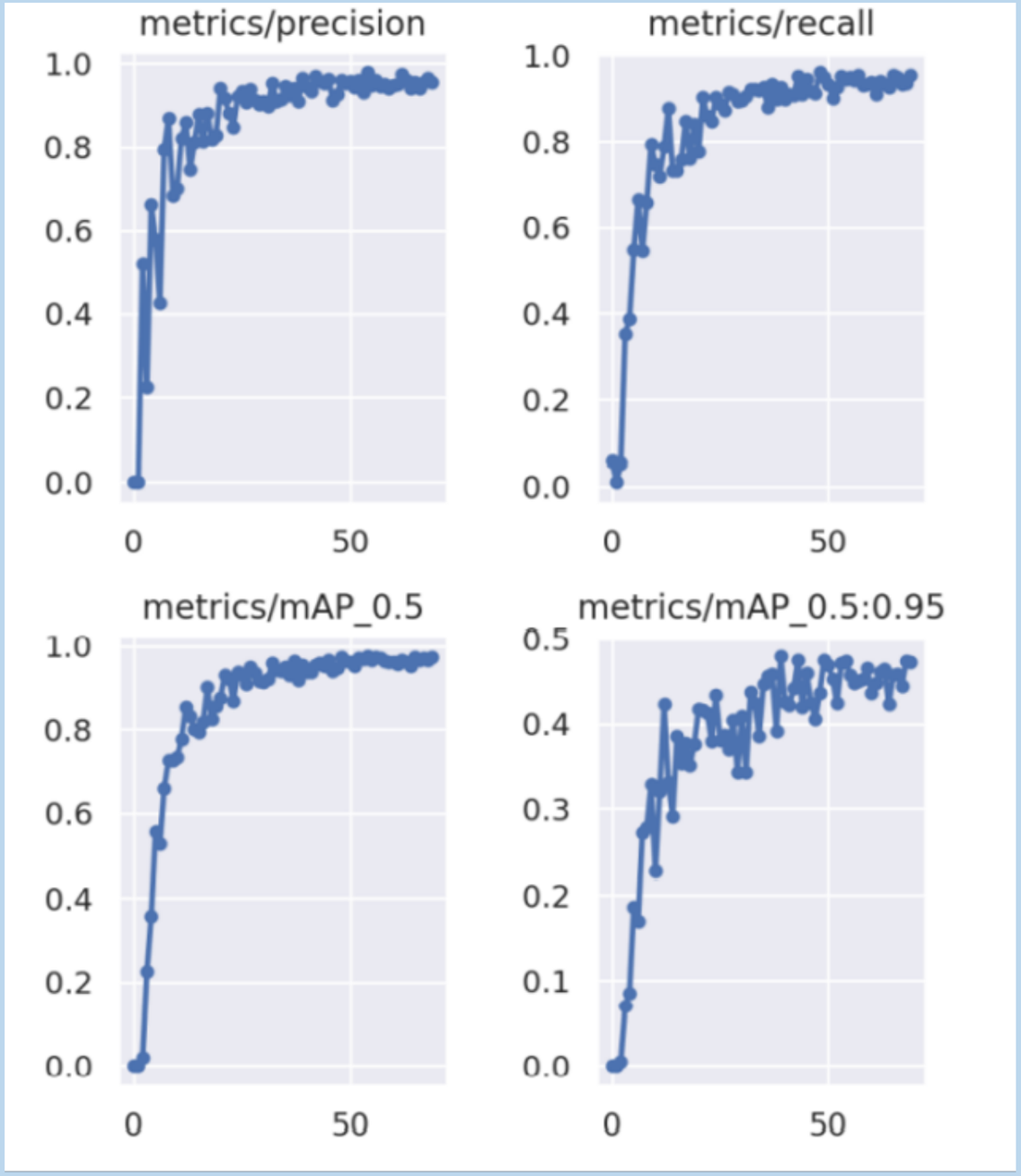
Training Metrics for V3



Project Implementation

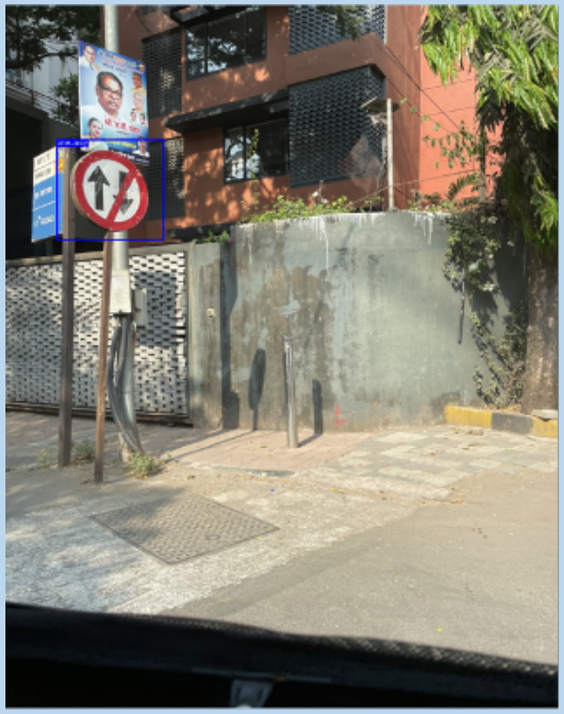
Testing Metrics for V3

Metrics	Values
mAP	94.3%
Precision	95.9%
Recall	92.6%



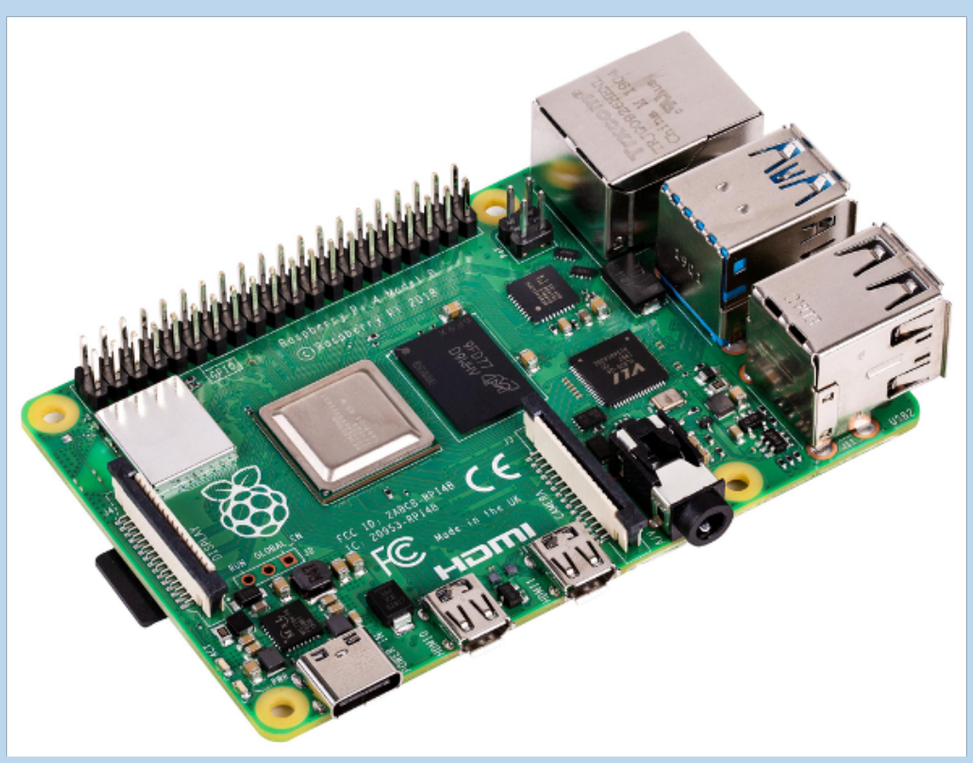
Project Implementation

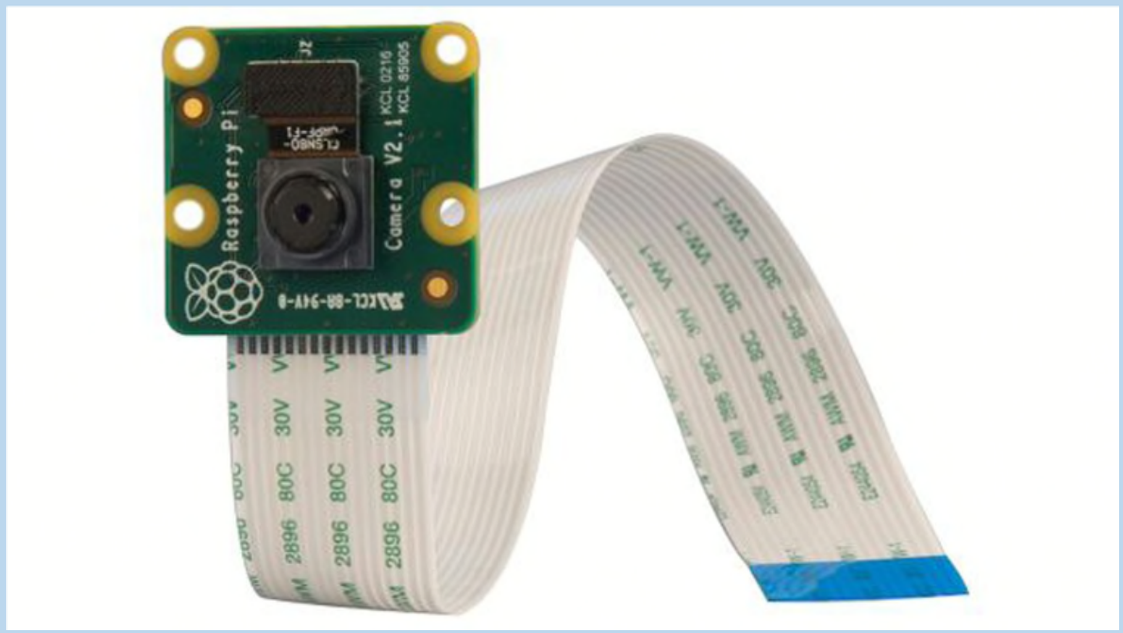
Testing Phase Output



Hardware Implementation

Raspberry Pi	Pi Camera
Version 4	Version 2
4 GB RAM	8 MP Optical Sensor
ARM architecture	1080p 30 fps 720p 60 fps





Hardware Implementation

- Recognise our respective traffic signs with a live feed
- Add a hardware interface to our model
- Raspberry Pi and a USB Camera/Pi Camera
- OS Installation and Camera compatibility with Pi version
- Using trained model ‘.h5’ file and OpenCV

Read the live feed

Convert to a readable format

Run through object detection model

Display bounding boxes

Conclusion

- We were able to conduct an inclusive study on the various kinds of Object Detection algorithms and the technologies around them for their real-time applications.
- YOLO version 3 performs the learning part of the deep learning model at a faster rate with lesser computational resources as compared to its other counterparts.
- For model training, we successfully created an original custom dataset consisting of various images of traffic signs around Mumbai and used it in our model.
- Finally, we implemented our code and model on Raspberry Pi for hardware implementation using a Pi Camera.
- Our model during the test phase gave a precision of 95.9%, recall of 92.6%, and a mean average precision value of about 94.3%

Future Work

- Our model is accurately detecting four classes of traffic signs.
- Our first scope of further work can be an inclusion of more traffic sign classes that exist on the roads of Mumbai.
- We can improve the accuracy and robustness of our model by further increasing the size of our dataset. This will help us give more precise weight values for the neural network in our model.
- We can improve the mAP values by fine-tuning our biases and weight values in this algorithm.
- We can make an alert system that can notify the driver about the upcoming traffic sign which it can recognize through our model.
- A traffic sign detection and recognition system can be deployed in self-driving cars