LEARNING TO DETECT AND RECOGNISE ROAD SIGNS

CM3203
ONE SEMESTER INDIVIDUAL PROJECT
MODULE CREDITS – 40

AUTHOR: *TAPAN AGARWAL*
SUPERVISOR: *DR. YUKUN LAI*
MODERATOR: *MARTIN CAMINADA*

MAY 2022

Abstract

Road Signs Detection and Recognition is an essential part of autonomous driving systems as road signs provide route related cautions which helps in preventing road accidents. They keep the traffic flowing by assisting the travelers and guarantee the safety of travelers by informing them about the speed limits, turning points etc. The main aim of this whole project is to develop a prototype using which we can detect and recognize the presence of multiple road signs in different formats like images, videos and most importantly in real-time. For the development of this prototype, I implemented two different editions of YOLOv4 (You Only Look Once version 4) which is an object detection algorithm. One of them is the normal YOLOv4 and the second one is YOLOv4-tiny. I trained and tested both the editions on the GTSDB (German Traffic Sign Detection Benchmark) Dataset and achieved 100% mAP (Mean Average Precision) with 30 FPS (Frames Per Second) for the normal YOLOv4 and 99.66% mAP with 16 FPS for the YOL0v4-tiny.

Acknowledgements

I am grateful to have Dr Yukun Lai as my supervisor for this project and would like to express my sincere thanks to him for providing me with the opportunity to work on this project. This project would have not been possible without his esteemed guidance and supervision. He genuinely believed in my work and assisted me throughout this whole project. Lastly, I would like to thank my family and friends and every other person who supported me this whole time.

Table of Contents

1.0 Introduction

Every year we witness a huge number of road accidents that are caused due to inability of the public to follow the road signs. For example in 2019, The Texas Department of Transportation revealed that there were 42,000 stop-sign related road accidents out of which 13,422 were caused by the drivers ignoring the stop signs [1]. The US Department of Transportation conducted a survey which shows that more than 65% of the drivers exceed the speed limit by five to ten miles on that particular road and that leads to accidents, injuries and deaths as the reaction time for the driver to any unwanted dangerous situation decrease [2].

All of these problems related to road accidents are direct results of the driver missing the road signs or maybe ignoring them or not being attentive and focused while driving. Therefore a road sign detection and recognition system would prove to be very useful in these scenarios as they can be integrated with the cameras installed in the vehicles to detect and recognize various road signs and provide a warning or an alert message to the driver which the driver might have missed before due to some sort of distraction or inattentiveness. In this way, these detection and recognition system will not only prevent accidents but will also contribute to the success of self-driving cars as cars will also learn to adapt themselves in the situation of unwanted incidents by taking some major decisions based on the road signs like automatically reducing the speed and driving under the speed limit given in the road signs.

To develop these road sign detection and recognition systems, we need an object detection algorithm so that we can obtain details and elements from an image and interpret them. These object detection algorithms should not only be accurate but fast as well so that they can be used to detect road signs in the real-time. YOLO is one such algorithm belonging to the family of single-stage detection algorithms.

Even though there are advancements in the object detection algorithms like YOLO and deep neural networks like CNN, there are always some other factors which impacts the working of these algorithms and neural networks in detecting and recognizing road signs. For example, there could be a lot of distortions and noises in the captured images and videos of the road signs on days of extreme weather conditions like heavy rain, thunderstorms, hailstorms, fog etc. Different viewing angles and distances will also affect the working of these systems. Moreover, the background of the captured image may also confuse the system to differentiate between the road signs as there could be something present in the background which resembles the road sign. Then different lighting conditions in the surrounding may also influence the color composition of the captured road sign. Another possibility is the position of the road signs with respect to the camera such that the road signs might be hidden in trees or in other natural environments like snow etc. Therefore, these factors will somehow influence the accuracy of the system to detect and recognize the road signs [3].

In this project, I have used the GTSDB (German Traffic Sign Detection Benchmark) dataset to train as well as test my YOLOv4 and YOLOv4-tiny models. Initially, I meant to use the MSTD (Mapillary Traffic Sign Dataset) for this project but later I get to understand that the computing resources I was having were not enough to train such a huge dataset, therefore, I have to switch to the GTSDB dataset. In this report, I will also explain that what were the major issues coming in the training of our YOLO models using the Mapillary Traffic Sign Dataset.

2.0 Background

This part of the report focuses on providing an insightful overview on the technologies, datasets, algorithms and other theoretical as well as technical concepts used for building this prototype.

2.1 German Traffic Sign Detection Benchmark Dataset

We got this dataset from the INI Benchmark Website of INSTITUT FUR NEUROINFORMATIK. There were two types of datasets available on this website namely the German Traffic Sign Recognition Benchmark Dataset (GTSRB) and the German Traffic Sign Detection Benchmark Dataset (GTSDB) and both of them are very popular and widely used for computer vision and machine learning problems. GTSDB dataset is mainly used for the detection problems whereas GTSRB is used for Classification problems [4].

Our main task was to develop a prototype which can detect the presence of road signs in an image, video and in real-time, therefore, We choose the GTSDB dataset which was introduced at the IEEE International Joint Conference on Neural Networks 2013. This dataset contains 900 natural images of the road signs out of which 600 are the training images placed in one folder (2 GB) and 300 are the testing images placed in another folder (1 GB). The images are in PPM format and the size of road signs in the images differs between 16*16 to 128*128 pixels. The road signs given in the images are from different perspectives and in different lighting conditions [5].

Given below are some sample images from the GTSDB Dataset –

Figure – 1 Shows the images of GTSDB from [5]

When we choose the dataset, we ensure that the dataset is fully annotated because annotations provide us with the location of road signs or any other object in the image which we want to extract and work on. Annotations give us the coordinates of the ROI (Region of Interest) which simply means the part of the image containing the road sign. These coordinates will help us to train our model by telling the position of the road signs in the image. These coordinates will also help in drawing bound boxes around the road signs present in the image once they will be detected and identified.

In the GTSDB dataset, there is a text file given called ‘gt.txt’ which contains annotations for all the road signs present in all the 600 training images in the following format: The first column refers to the image name, the second, third, fourth and fifth column refers to the lower x, lower y, upper x and upper y coordinates of the road sign in that image respectively.

This is what ‘gt.txt’ looks like:-

```
00000.ppm;774;411;815;446;11
00001.ppm;983;388;1024;432;40
00001.ppm;386;494;442;552;38
00001.ppm;973;335;1031;390;13
00002.ppm;892;476;1006;592;39
00003.ppm;742;443;765;466;4
00003.ppm;742;466;764;489;9
00003.ppm;737;412;769;443;21
00004.ppm;898;342;967;409;21
00004.ppm;906;407;955;459;2
00005.ppm;1172;164;1284;278;9
00006.ppm;926;350;989;414;2
00007.ppm;825;406;864;445;9
00008.ppm;785;460;811;486;2
00008.ppm;779;424;816;461;12
00009.ppm;925;466;949;490;4
00010.ppm;1193;358;1269;436;12
```

Figure – 2 Screenshot of the sample of annotations from GTSDB

The last column in ‘gt.txt’ represents the class to which these road signs belong. After going through the GTSRB dataset, I get to understand that there are a total of 43 classes in which the road signs in the German Traffic Sign Dataset are distributed.

There is another file given with this dataset called ‘Readme.txt’ and in that file, these 43 classes are divided among 4 classes called ‘prohibitory’, ‘danger’, ‘mandatory’ and ‘other’. Therefore, we will also train our model to detect and classify the road signs in one of these 4 classes.

2.2 Mapillary Traffic Sign Dataset

The Mapillary Traffic Sign dataset is one of the biggest datasets present for doing the tasks related to detection as well as classification of road signs. It comprises diversified images of road signs taken all over the world under different weather and light conditions using different camera lenses and from different angles and viewpoints. MSTD contains around 100,000 images of which half are fully annotated and half are partially annotated [6].

For this project, I have downloaded and used the fully annotated part of MSTD and the structure of this fully annotated part is explained below:

1. It contains 36589 training images distributed over 3 folders where one folder contains 12197 images and the other two folders contain 12196 images each. (size of one folder = 9.66GB and all three combined = 29 GB)
2. Then it contains 5320 Validation images all in one folder. (size = 4.25 GB)
3. Then a folder containing a total of 41909 annotation files, 36589 (training) + 5320 (validation)
4. Then finally we have a folder containing 10544 test images (size = 8.34 GB)

Annotation files in MSTD are in JSON format and they encompass the width and height of the image along with the key, label and bounding box coordinates of the road signs present in that image. The structure of the annotation looks like this: -

```
{
  "width": 4128,
  "height": 3096,
  "ispano": false,
  "objects": [
    {
      "key": "vg4k8i7work3bhq9xhel1s",
      "label": "other-sign",
      "bbox": {
        "xmin": 1921.8984375,
        "ymin": 2244.90234375,
        "xmax": 2060.9765625,
        "ymax": 2313.685546875
      },
      "properties": {
        "barrier": false,
        "occluded": false,
        "out-of-frame": false,
        "exterior": false,
        "ambiguous": false,
        "included": false,
        "direction-or-information": true,
        "highway": false,
        "dummy": false
      }
    }
  ]
}
```


Figure – 3 Screenshot of the annotations from MSTD

2.3 Convolution Neural Network (CNN)

To recognize any image for example an image of a Koala, our brain first sees if the image contains features of the head of a Koala like eyes, nose and ears and if these features are present that means the head of a Koala is present in that image. In a similar way if the features of the body like hands and legs are present then the body of a Koala is present and continuing this if we have both head and body in the image then we can say that a Koala is present in the image. So to make computers recognize these tiny features, Convolution Neural Networks are used. CNN uses the concept of filters to locate the location of these tiny features in the image. There would be different filters for different features like eyes, nose, years etc. Now a convolution operation is performed using the original image and the filters so that we can get feature maps for eyes, nose, ears etc. and these feature maps gives us the location of those features like eyes, nose, eyers etc. in the image. So in a way filters are nothing but the feature detectors. After getting the feature map for the eyes, nose, ears, hands and legs, we can again perform the convolution operation to get the feature maps for the head and the body. These feature maps are in the form of 2D arrays so to feed them to the neural network, we flatten these 2D arrays into 1D arrays and then we join these 1D arrays. After we join these 1D arrays, we can make a fully connected dense neural network to classify if this is a Koala or not. In this whole process, the part where we used the convolution operation is called the Feature Extraction Part and the part where we are using the dense neural network is called the Classification part.

The figure below provides a diagrammatic explanation of the discussion above:-

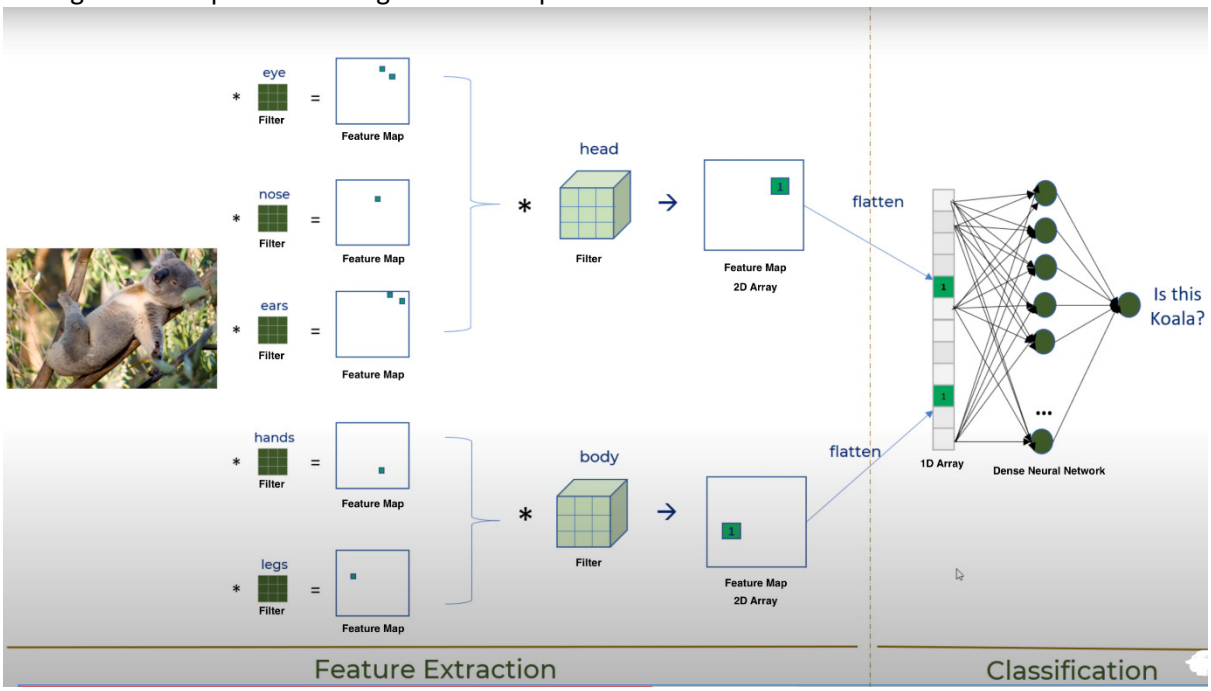


Figure – 4 Shows Feature Extraction and Classification part of the CNN from [10]

This is still not a complete CNN. There are two other components as well. The first component is ReLU which is an activation function used to bring non-linearity into our model so what it will do is it will replace all the negative values in our feature map with zeros and if the value is more than zero then it will keep it as it is. ReLU also speeds up the training process and it is faster to compute as we are doing only one check whether the number is greater than zero or not.

The figure below shows the feature map before and after applying the ReLU:-

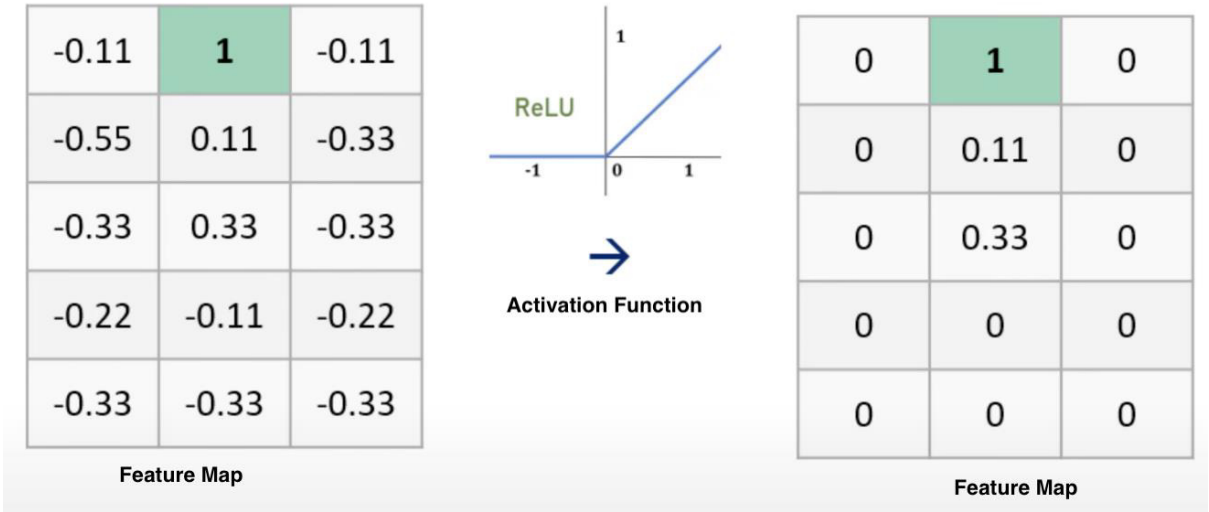


Figure – 5 Shows Feature Map before and after applying ReLU from [10]

Now if the image is large in size for example 1920*1080*3 (RGB channels) then there would be approx. 6 million neurons in the input layer and let's say that we have a hidden layer with

approx. 4 million neurons so we are talking about 6*4 = 24 million weights to be calculated just between input and the hidden layer and there could be several hidden layers so we might have to compute 500 million or even billion weights and that is too much computation. To address this problem of computation, the second component called the Pooling layer is used to reduce the size of the feature map. Thus, we can take a feature map and apply the Max Pooling operation on it to generate a new feature map which will be smaller in size than its previous version. The Max Pooling will choose the biggest number from all the 2 by 2 windows of the feature map to create a new feature map. So in this way, the Pooling layer helps in saving computation resources. Therefore Max Pooling along with convolution operations helps in position invariant feature detection doesn't matter where the eyes, nose, ears etc. are in the image, it will detect these features.

The figure below shows the feature map before and after applying the Max Pooling:-

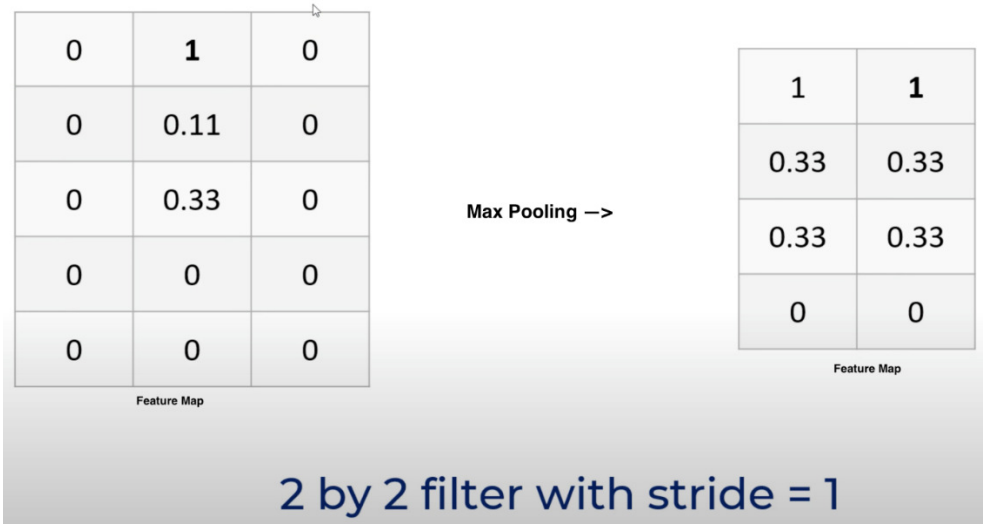


Figure – 6 Shows the Max Pooling operation on the Feature Map from [10]

Given below is the image of applying ReLU and Max Pooling on the Feature Map.

Figure – 7 Shows ReLU and Max Pooling operations on the Feature Map from [10]

So this is what the complete convolution neural network looks like:-

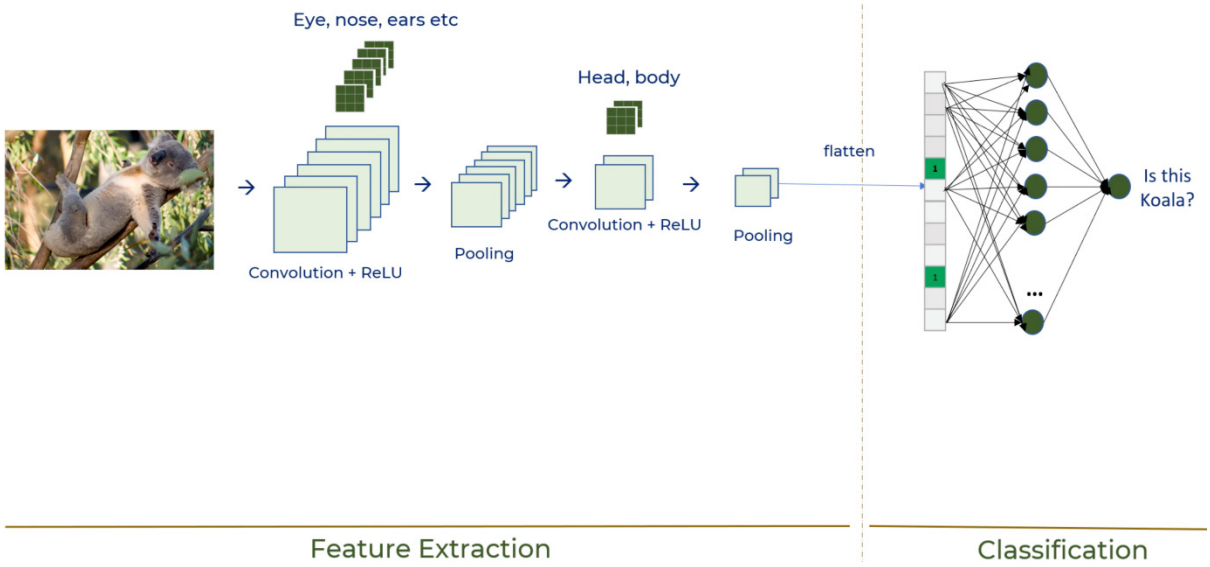


Figure – 8 Shows complete CNN from [10]

2.4 You Only Look Once (YOLO)

If we are working on the image classification problem where for example we want to decide if the image is of a dog or a person or both then the output of the neural network will be 1 for the dog class if the dog is present otherwise the output for the dog class will be 0 and same goes for the person class but when we talk about the object localization, we are not only telling which class the object belongs to but also the bounding box or the position of the object within the image. Therefore for developing detection and recognition prototypes, we want to train the neural network to not only classify the objects present in the image but also to detect the location or the bounding boxes for the objects in the image. This is where YOLO comes in. In YOLO, we train the model by feeding training images with the bounding box annotations so that our model can not only classify the objects present in the images but can also locate the position of the objects so that the bounding box can be drawn around the objects in the image.

To detect objects in the image, for example, if we have a dog object in the image then to detect this object YOLOv4 will divide the whole image into grid cells and will examine all the cells to find the center of the dog object so that it can predict the bounding box along with these four coordinates ‘ox’, ‘oy’, ‘ow’, ‘oh’ where ‘ox’ and ‘oy’ are the center ‘x’ and ‘y’ coordinates and ‘ow’ and ‘oh’ are the width and height of the bounding box with respect to the grid cell containing the center of the dog object [8].

The figure below gives the equations which can be used to calculate the center location, width and height of the bounding box with respect to the grid:-

c_x, c_y = location of the grid
b_x, b_y = location of the bounding box
p_w, p_h= anchor box prior got from clustering
b_w, b_h = dimension of the bounding box

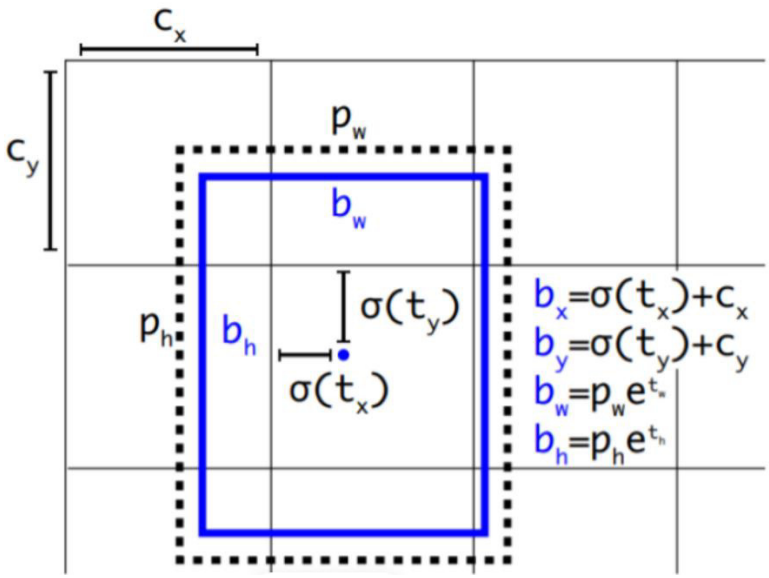


Figure - 9 Shows equations to calculate center location, width and height of the bounding box from [12]

For each bounding box, YOLO will also predict the probability (confidence score) of the class to which the object present in the bounding box belongs to. After that YOLO performs Non-Max Suppression to get the improved results by keeping the best bounding box and removing all other predicted bounding boxes having detection probability less than the given NMS threshold [11].

Figure – 10 Shows detected images before and after applying the Non-Max Suppression from [11]

Now let's have a brief look at the architecture of YOLOv4:-

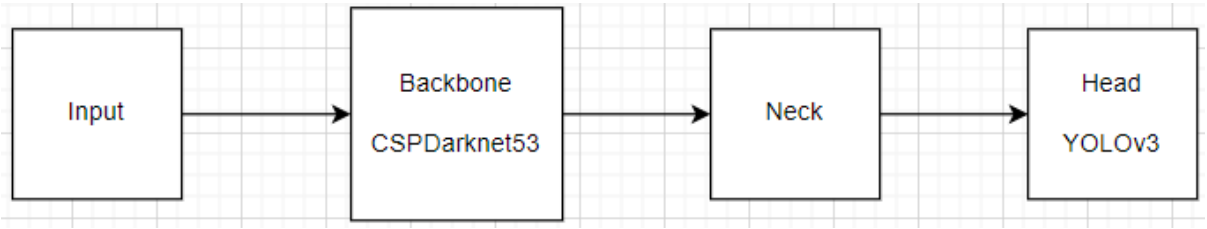


Figure – 11 Shows the architecture of YOLO

The images will pass through the Input block to the Backbone block. In Backbone Block, we use classification models. The task of this Backbone is to extract features from the images. So to extract the features, the CSPDarknet53 network is used in the Backbone which is a type of CNN. The next stage is the Neck block and the task of this block is to collect feature maps from different stages. In other words, the Neck block act as a feature aggregator. Then finally we have the Head block which can also be termed as the object detector. In this Head block using YOLOv3 head, we are performing object detection and plotting bounding boxes by dividing the whole image into grid cells as we discussed above and also shown in Figure-9. This architecture of YOLOv4 made it achieve 10% more Average Precision and 12% more Frames Per Second as compared to YOLOv3 [16].

Now we have two new features in YOLOv4 called BoF (Bag-of-Freebies) and BoS (Bag-of-Specials). Bag of Freebies simply refers to some techniques using which we can improve the accuracy of our model without adding to inference time in production [12]. These techniques could be Data Augmentation, Class Imbalance, Cost Function etc. We have different BoF for the Backbone and the Head. So the BoF used in the Backbone are CutMix, Mosaic data augmentation, DropBlock regularization and Class Label Smoothing whereas BoF used in the Head are CIoU-loss, CmBN, Mosaic, Self-Adversarial Training etc. Now Bag of Specials (BoF) also refers to some techniques which YOLOv4 uses as these techniques significantly boost the accuracy of the model and also increase the inference time [12]. Again Like BoF, we have different sets of BoS for the Backbone and the Head. Mish Activation and Cross-Stage Partial Connections are some of the BoS used in the Backbone whereas BoS used in the Head are Mish Activation, SPP-block, SAM-block, PAN path-aggregation block etc. [13]

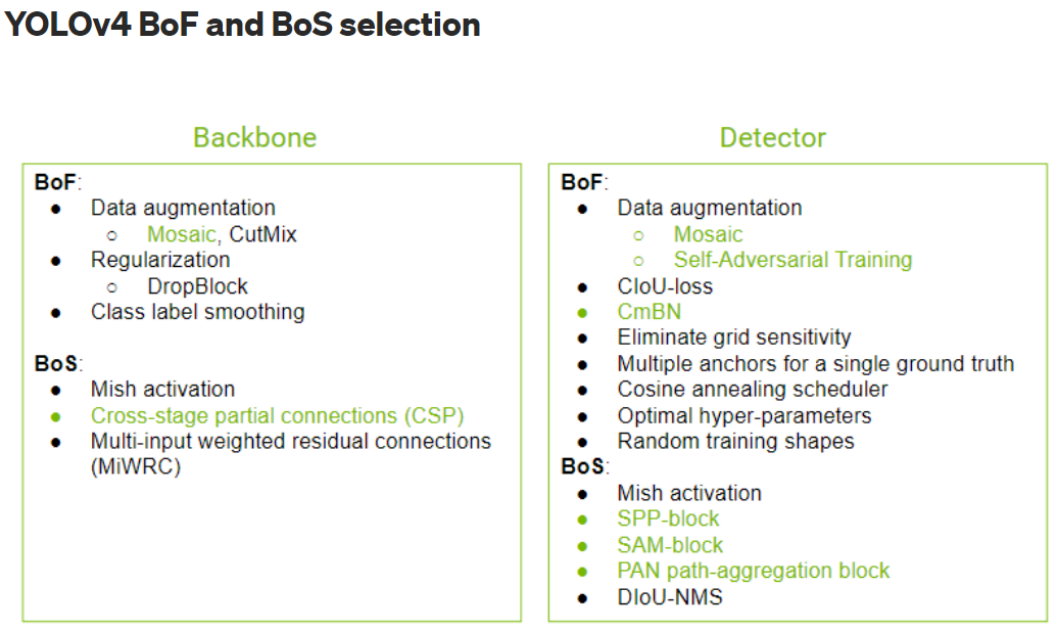


Figure – 12 Showing BoF and BoS used in Backbone and Detector from [13]

Now we have some evaluation metrics to determine if our model is performing well or not. Precision is one such metric for determining how accurate our predictions are. It determines how many of our model's guesses are actually right. Then we have another metric called Recall which assesses how well our model call find all the positives [18].

$$Precision = \frac{TP}{TP + FP}$$

$$Recall = \frac{TP}{TP + FN}$$

TP = True positive

TN = True negative

FP = False positive

FN = False negative

Figure – 13 Shows formula of Precision and Recall from [18]

The Intersection over Union (IoU) is another metric which tells us about the performance of the model by calculating the deviation of the predicted bounding box from the real bounding box [14] or in other words by determining how perfectly our predicted bounding box coincides with the real bounding box. Loss in IoU can be defined as the intersection of the area of the predicted and the real bounding boxes divided by the union of the area of the same predicted and the real bounding boxes [17].

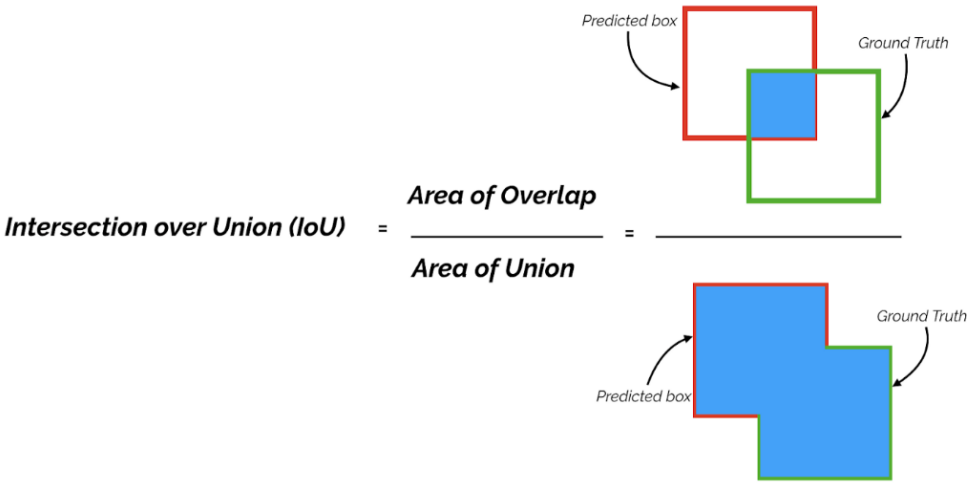


Figure – 14 Intersection Over Union from [18]

To decide if the prediction is a True Positive or a False positive, we take the help of IoU threshold. Therefore if IoU threshold is set to be for example 0.5 then any prediction in which IoU value is greater than this threshold will be a True Positive whereas a prediction in which this IoU value is less than the threshold will be a False Positive [18].

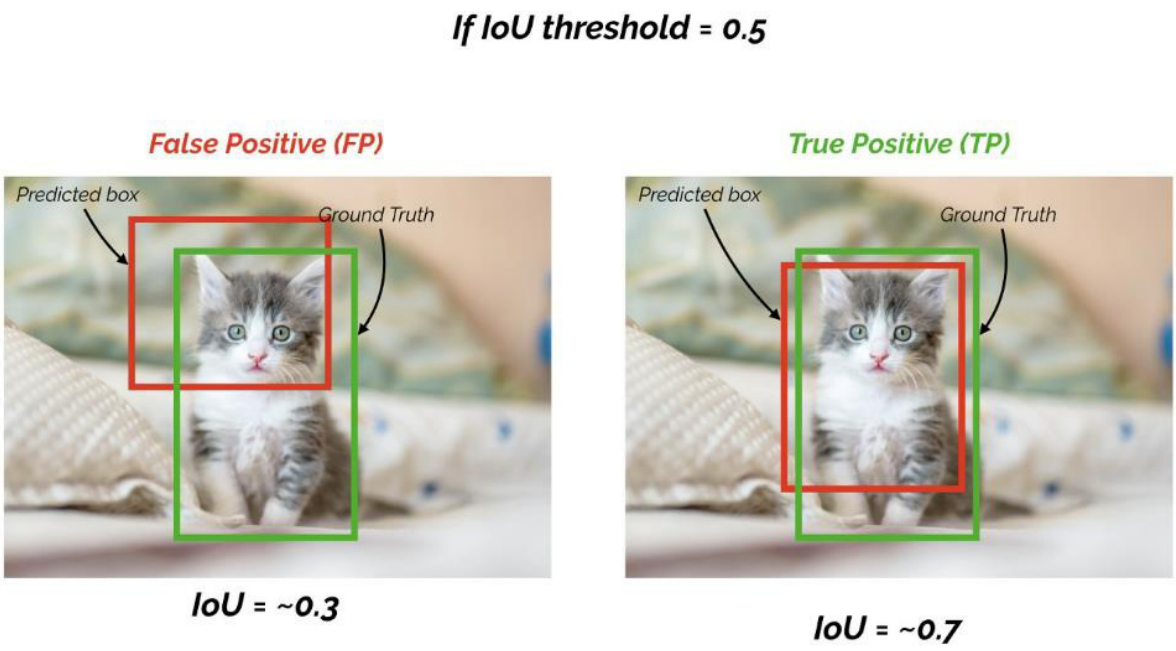


Figure – 15 Shows False Positive and True Positive from [18]

Lastly, AP (Average Precision) is the area under the Precision-Recall Curve. For each and every class of the objects, AP is calculated and the average of this AP is termed as mAP (Mean Average Precision) which is again a metric that states the performance of our model [18].

3.0 Approach and Implementation

My approach in building this prototype would be to first download the dataset and analyze it, like the format of the images, and the format of the annotations given with the dataset. YOLO has its own format in which it accepts the images and the annotations therefore we first need to process the data in the format which YOLO accepts. Here, we have to remember that we are only processing the training dataset because it is the one which contains the annotations. The annotations or the location of the objects are not provided with the testing dataset. So while processing this training data into the YOLO format, I will divide it into 90:10 ratio where 90% of the training dataset will be used to train our YOLO model and 10% to validate our YOLO model. I will train my model using both the YOLOv4 and the YOLOv4-tiny which is an edition of YOLOv4 only. After training, I will evaluate the performance of both the models and will finally test them on different images from the testing dataset, on a pre-recorded video and lastly on the real-time images and video stream using our laptop's front camera.

3.1 Google Drive

We will be using Google Drive extensively in this project. Google Drive provides cloud services and we will store our dataset in google drive. Apart from the dataset, we will also store in google drive the files generated while training and testing our model for example the weight files which will be generated while training and other result files which we will get while testing and evaluating our model. Google Drive provides a range of cloud-native apps developed by google for example Docs, Sheets, Slides etc. Google Collaboratory is one such cloud-native app that we will use to write and run our code.

3.2 Google Colaboratory

Google Colaboratory is the cloud version of Jupyter Notebook that allows us to write and execute python code on Google's virtual servers. It is a very popular tool especially for doing the stuff related to Machine Learning. Most of the packages and machine learning libraries like NumPy, Pandas, Matplotlib, Keras, TensorFlow, PyTorch etc. are already installed in the Google Colab therefore we just need to import them. Google Colaboratory can easily be integrated with Google Drive and then we can access and manipulate the files present inside the Google Drive. Google Colab also has its own local runtime storage where we can extract zip files and work on them. Most importantly, Google Colab provides us with free GPU and TPU access and to train and test our YOLOv4 model, we do require a GPU. In Google Colab, the runtime session gets automatically recycled every 12 to 13 hours removing everything from the local storage and from the executed cells as well and this is the one downside of Google Colab which has impacted me a lot especially while training and testing the model. Moreover, Google Colab gives only limited access to GPU, after we exhaust all the GPU usage limits, we have to wait for a cool-off period only after which Colab again provides us with GPU access. Therefore, it will impact the training process if we have a large dataset to train or if the training needs to be continued for a longer duration. But still, Google Colab is the best tool available out there.

3.3 Dataset Processing

First we downloaded the dataset from [9]. We downloaded two zip files namely 'TrainIJCNN2013' and 'TestIJCNN2013'. These files were unzipped as folders with the same name. We then examined the structure of these two folders. 'TrainIJCNN2013' folder contains 600 training images along with some other folders and text files. 'gt.txt' is one of the text files which contains annotations for all the road signs present in the 600 training images. We have shown the structure of 'gt.txt' file in section 2.1 of the report and have also explained the format of the annotations in the same section. Then we have another text file called 'ReadMe.txt' which contains information on how to distribute 43 classes between 4 new classes called 'prohibitory', 'mandatory', 'danger' and 'other'. So in the 'TrainIJCNN2013' folder, those 600 training images, 'gt.txt' and 'ReadMe.txt' are the only files which are important to us and will be used to train our prototype and therefore apart from these three, we deleted all the other files and folders from 'TrainIJCNN2013' and moved 'gt.txt' and 'ReadMe.txt' out of this folder as the result, 'TrainIJCNN2013' folder now only contains 600 training images. 'TestIJCNN2013' folder contains 300 testing images without any annotations and we also don't require annotations for these testing images. This folder also contains a 'ReadMe.txt' file but this file is of no use to us and therefore we deleted this file so now we have only 300 testing images in the 'TestIJCNN2013' folder which will be used to test our model. Then we compressed the 'TrainIJCNN2013' and the 'TestIJCNN2013' folders back into the zip files so that they can be uploaded to the google drive and then extracted in the google colaboratory. Finally, we created a folder called 'YOLO' on our google drive and uploaded 'TrainIJCNN2013.zip', 'TestIJCNN2013.zip', 'gt.txt' and 'ReadMe.txt' files into this folder on our google drive. After uploading these files, we created two google colaboratory files called 'yolov4.ipynb' and 'yolov4tiny.ipynb' in the same folder which will contain our YOLOv4 and YOLOv4-tiny model respectively.

The next task is to set up the environment. We can do the data processing part without the GPU but to train our YOLOv4 model we do need a GPU, therefore, it is better to change the runtime to GPU otherwise we have to change it later and if we change it later then we will lose all the extracted and manipulated files present in our google colab's local runtime storage as the whole session will restart and then we have to run all the cells again from the beginning, therefore, we changed the hardware accelerator from None to GPU and set the runtime shape to High RAM in the starting only.

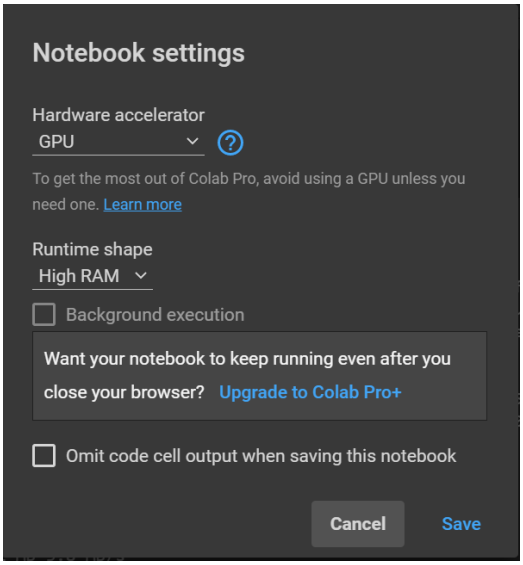


Figure – 16 is the screenshot showing the change in the runtime settings of the 'yolov4.ipynb' file.

One important thing to point out is that we upgraded to google colab pro because the basic version of google colab provides very limited access to the GPU services. It denies the access to GPU once we have trained our model for five to six hours until the cool-off period is over and this cool-off period ranges from 12 hours to even a couple of days depending on our GPU usage limit. Even after upgrading to colab pro, we will get disconnected while training our model but this time we will be able to connect back and access the GPU services instantly without waiting for the cool-off period.

Another important thing to mention here is that we are going to extract and work on the files in our local runtime storage rather than extracting and working on them in our google drive because if the files are present in our local runtime storage then it would be faster to read and write these files while processing the dataset and training the model as compared to accessing these files if they were present in our google drive.

We have given numbers to all the cells in 'yolov4.ipynb' and 'yolov4tiny.ipynb' so that any confusion can be avoided while indicating or talking about these cells as we move on.

Now in the first two cells, I am importing the libraries needed to build this prototype. As I kept writing code for this prototype, I came across various errors and thus I kept importing the libraries to resolve those errors and in some cases, I first installed the libraries using the pip command and then imported them. For this prototype, I first uninstalled OpenCV which was present by default in the google colab and then I installed the latest version of OpenCV which is 4.5.5 because few functions like cv2_imshow() were not working with the older versions of OpenCV.

```
# cell-1
!pip3 install bounding_box
!pip3 install cvlib
!pip3 uninstall opencv
!pip3 install --upgrade opencv-python
```

Figure – 17 screenshot of cell-1

```
# cell-2
!apt-get install p7zip-full
import os
import json
import PIL
from PIL import Image, ImageDraw, ImageColor, ImageFont
import numpy as np
import csv
import json
import pandas as pd
import shutil
import matplotlib.pyplot as plt
import time
import cv2
from google.colab.patches import cv2_imshow
import cvlib as cv
from bounding_box import bounding_box as bb
import matplotlib.gridspec as gridspec
import glob
%matplotlib inline
from IPython.display import display, Javascript, Image
from google.colab.output import eval_js
from base64 import b64decode, b64encode
import io
import html
from google.colab import files
cv2.__version__
```

Figure – 18 screenshot of cell-2

In cell-3, we are connecting our google drive with our 'yolov4.ipynb' colab file. This will allow us to read and write files and folders from and to the google drive using the colab file. By mounting google drive on google colab, we can access and manipulate the files present in our google drive. This will allow us to extract the files from google drive into our colab file's local runtime storage and then we can work on these files and can store the results back into our google drive.

```
# cell-3
from google.colab import drive
drive.mount('/content/drive')
```

Figure – 19 Screenshot of cell-3

Now we need to clone a repository called 'darknet' from Alexey Bochkovskiy's github. This repository has some configuration files which contains the structure of the neural networks. According to our dataset, we will make some changes in these configuration files and then use them to train, test and evaluate our model. Therefore, we cloned the darknet repository in our colab's local runtime storage and then we extracted the 'TrainIJCNN2013.zip' file which contains the training dataset into the 'darknet/data' folder so now the 'data' folder has the 'TraiIJCNN2013' folder which contains 600 training images. Cell-4 and cell-5 shows the code to clone the darknet repository and extract the training dataset respectively.

```
# cell-4
os.chdir('/content')
print(os.getcwd())
!git clone https://github.com/AlexeyAB/darknet
```

Figure – 20 Screenshot of cell-4

```
# cell-5
os.chdir('/content/darknet/data')
print(os.getcwd())
!unzip '/content/drive/MyDrive/YOLO/TrainIJCNN2013.zip'
```

Figure – 21 Screenshot of cell-5

Now the images in the training dataset which we extracted are in PPM format and YOLO does not support the 'ppm' format therefore we first need to change the format of the images. We changed the image format to PNG because in PNG the compression is lossless

[20] which means while processing these PNG images, we will not lose any of their details or characteristics. In cell-6, we are changing the format of the images from ‘ppm’ to ‘png’.

```
# cell-6
for fn in os.listdir('/content/darknet/data/TrainIJCNN2013'):
    if fn.endswith(".ppm"):
        prefix = fn.split(".ppm")[0]
        im = PIL.Image.open('/content/darknet/data/TrainIJCNN2013'+'/'+fn)
        name = prefix+'.png'
        rgbImage = im.convert('RGB')
        rgbImage.save('/content/darknet/data/TrainIJCNN2013'+'/'+name)
        #im.save('/content/darknet/data/TrainIJCNN2013'+'/'+prefix+'.png')
        os.remove('/content/darknet/data/TrainIJCNN2013'+'/'+fn)
    else:
        continue
```

Figure – 22 Screenshot of cell-6

Now we need to convert the format of the annotations given in ‘gt.txt’. We need to bring these annotations in the format which YOLO accepts. Let’s have a look at the current format of the annotations. The code written in cell-7 helps us to visualize the current format of the annotations for our training dataset.

```
# cell-7
data = pd.read_csv('/content/drive/MyDrive/YOLO/gt.txt', sep=';', header=None, names=["FileName", "xMin", "yMin", "xMax", "yMax", "Class"])
data.head()
```

	FileName	xMin	yMin	xMax	yMax	Class
0	00000.ppm	774	411	815	446	11
1	00001.ppm	983	388	1024	432	40
2	00001.ppm	386	494	442	552	38
3	00001.ppm	973	335	1031	390	13
4	00002.ppm	892	476	1006	592	39

Figure – 23 Screenshot of cell-7

The file names in the first column are ending with ‘ppm’ extension and we will change it to ‘png’ extension. Then the second and the third column of the data frame named ‘xMin’ and ‘yMin’ are the co-ordinates of the bottom-left corner of the bounding box containing our road sign in the image. Similarly, the fourth and the fifth column named ‘xMax’ and ‘yMax’ are the co-ordinates of the top-right corner of the bounding box. But in YOLO, we have a different way to represent a bounding box, it is represented by four variables which can be termed as ‘objectCenterX’, ‘objectCenterY’, ‘objectWidth’ and ‘objectHeight’ where ‘objectCenterX’ and ‘objectCenterY’ represent the co-ordinates of the center of the bounding box containing the object and ‘objectWidth’ and ‘objectHeight’ represents the width and height of the same bounding box. Here we have to remember that the value of these four variables describing a bounding box in YOLO should be normalized with respect to the image containing this bounding box. Therefore we first have to find the width and height of the image and then we will divide ‘objectCenterX’ and ‘objectWidth’ with the image width and finally we will divide ‘objectCenterY’ and ‘objectHeight’ with the image height. As ‘objectCenterX’ and ‘objectCenterY’ lies at the center of the line connecting bottom-left and top-right corners of the bounding box therefore to find the normalized value of ‘ObjectCenterX’, we will apply the midpoint formula and divide the outcome by the image width. As the result, objectCenterX = ((xMax + xMin)/2)/imageWidth and similarly we can find the ‘objectCenterY’ by applying the midpoint formula and dividing the outcome by

image height which gives us objectCenterX = ((yMax + yMin)/2)/imageHeight . We can find the normalized value of 'objectWidth' by subtracting 'xMin' from 'xMax' and then dividing the outcome by image width. Similarly the normalized value for the 'objectHeight' can be obtained by subtracting 'yMin' from 'yMax' and then dividing the outcome by image height. Therefore objectWidth = (xMax – xMin)/imageWidth and objectHeight = (yMax - yMin)/imageHeight . Another important thing which we want to do here is to distribute the 43 classes (0-42) mentioned in the sixth column of the data frame under the heading 'class'. Here we are going to distribute these 43 classes according to the instructions given in the 'ReadMe.txt' file which we uploaded to our google drive. So we distribute these 43 classes between 4 new classes called 'mandatory', 'prohibitory', 'other' and 'danger'. To summarize everything we discussed above, in cell-8 we first changed the filename extension in the data frame from 'ppm' to 'png' and while doing so, we also extracted the width and height of the image and added it to the data frame. The output of cell-8 shows how our updated data frame looks like.

```
# cell-8
base = "/content/darknet/data/TrainIJCNN2013/"
imageHeights = []
imageWidths = []
fileNames = data["FileName"]

for i in range(len(fileNames)):
  fn = fileNames[i]
  fn = fn[:-4] + ".png"
  fileNames[i] = fn
  img = PIL.Image.open(base + fn)
  width, height = img.size
  imageHeights.append(height)
  imageWidths.append(width)

data["imageWidth"] = imageWidths
data["imageHeight"] = imageHeights

print(data)
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:9: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  if __name__ == '__main__':
     FileName  xMin  yMin  xMax  yMax  Class  imageWidth  imageHeight
0    00000.png   774   411   815   446     11        1360          800
1    00001.png   983   388  1024   432     40        1360          800
2    00001.png   386   494   442   552     38        1360          800
3    00001.png   973   335  1031   390     13        1360          800
4    00002.png   892   476  1006   592     39        1360          800
..         ...   ...   ...   ...   ...    ...         ...          ...
847  00570.png   881   416   914   449      9        1360          800
848  00571.png  1287   361  1308   384     17        1360          800
849  00575.png   403   474   435   506     38        1360          800
850  00593.png   584   510   608   534     38        1360          800
851  00599.png   700   454   722   476      9        1360          800
```

Figure – 24 Screenshot of cell-8

In cell-9, we are distributing 43 classes between 4 new classes and then for each and every road sign present in our training dataset, we are calculating the value of variables that represents a bounding box in YOLO.

```
# cell-9
mandatory = [33,34,35,36,37,38,39,40]
prohibitory = [0,1,2,3,4,5,7,8,9,10,15,16]
other = [6,12,13,14,17,32,41,42]
danger = [11,18,19,20,21,22,23,24,25,26,27,28,29,30,31]

dics = []

for i, row in data.iterrows():
  cl = row["Class"]
  if cl in prohibitory:
    cl = 0
    ObjectClassLabel = 'prohibitory'
  elif cl in danger:
    cl = 1
    ObjectClassLabel = 'danger'
  elif cl in mandatory:
    cl = 2
    ObjectClassLabel = 'mandatory'
  else:
    cl = 3
    ObjectClassLabel = 'other'

  dics.append({
      'fileName': row["FileName"],
      'objectClassLabel': ObjectClassLabel,
      'objectclass': cl,
      'objectCenterX': ((row["xMin"] + row["xMax"]) / 2) / row["imageWidth"],
      'objectCenterY': ((row["yMin"] + row["yMax"]) / 2) / row["imageHeight"],
      'objectWidth': (row["xMax"] - row["xMin"]) / row["imageWidth"],
      'objectHeight': (row["yMax"] - row["yMin"]) / row["imageHeight"],
  })
```

Figure – 25 Screenshot of cell-9

Now we have converted the annotations into the format which YOLO accepts. In cell-10, we are visualizing the data frame which contains the transformed annotations. We can see in the output of cell-10 that there are three road signs in the image '00001.png' as we have 3 rows for this image and these three rows contains the information to draw the bounding box around these three road signs.

```
# cell-10
anno_data = pd.DataFrame(dics)
print(anno_data)
print()
for dic in dics:
  if dic["fileName"] == "00001.png":
    print(dic)
```

```
     fileName objectClassLabel  objectclass  objectCenterX  objectCenterY  \
0    00000.png           danger            1       0.584191       0.535625
1    00001.png        mandatory            2       0.737868       0.512500
2    00001.png        mandatory            2       0.304412       0.653750
3    00001.png            other            3       0.736765       0.453125
4    00002.png        mandatory            2       0.697794       0.667500
..         ...              ...          ...            ...            ...
847  00570.png      prohibitory            0       0.659926       0.540625
848  00571.png            other            3       0.954044       0.465625
849  00575.png        mandatory            2       0.308088       0.612500
850  00593.png        mandatory            2       0.438235       0.652500
851  00599.png      prohibitory            0       0.522794       0.581250

     objectWidth  objectHeight
0       0.030147       0.04375
1       0.030147       0.05500
2       0.041176       0.07250
3       0.042647       0.06875
4       0.083824       0.14500
..           ...           ...
847     0.024265       0.04125
848     0.015441       0.02875
849     0.023529       0.04000
850     0.017647       0.03000
851     0.016176       0.02750

[852 rows x 7 columns]

{'fileName': '00001.png', 'objectClassLabel': 'mandatory', 'objectclass': 2, 'objectCenterX': 0.7378676470588236, 'objectCenterY': 0.5125, 'objectWidth': 0.030147058823529412, 'objectHeight': 0.055}
{'fileName': '00001.png', 'objectClassLabel': 'mandatory', 'objectclass': 2, 'objectCenterX': 0.3044117647058823, 'objectCenterY': 0.65375, 'objectWidth': 0.041176470588235294, 'objectHeight': 0.0725}
{'fileName': '00001.png', 'objectClassLabel': 'other', 'objectclass': 3, 'objectCenterX': 0.736764705882353, 'objectCenterY': 0.453125, 'objectWidth': 0.04264705882352941, 'objectHeight': 0.06875}
```

Figure – 26 Screenshot of cell-10

Now we have the annotations in the data frame but to feed it to the YOLO, we need to put these annotations in the text file. The name of the image containing the road signs and the name of the text file containing bounding box annotations for those road signs should be exactly same. So for example as we have seen above that we have three road signs in

'00001.png' therefore we have to create a text file whose name will be '00001.txt' and this text file will contain three rows as we have three road signs in '00001.png'. So each row in '00001.txt' corresponds to one road sign in '00001.png'. Now the first value of the row represents the class to which the object or the road sign enclosed in the bounding box belongs to. The name of class will appear on the top of the bounding box with the probability score. Then second, third, fourth and fifth values of the row represents objectCenterX, objectCenterY, objectWidth and objectHeight respectively. In this way for each and every image, there should be a text file with the same name as the image and the text file should contain the bounding box annotations for all the road signs present in the image.

```
# cell-11
train_directory_path = '/content/darknet/data/TrainIJCNN2013'
os.chdir(train_directory_path)

for cd, d, fls in os.walk('.'):

  for fl in fls:

    if fl.endswith('.png'):

      image_title = fl[:-4]

      yolo_file = anno_data.loc[anno_data['fileName']==fl]

      df = yolo_file.loc[:,['objectclass','objectCenterX','objectCenterY','objectWidth','objectHeight']].copy()

      save_path = train_directory_path + '/' + image_title + '.txt'

      df.to_csv(save_path, header=False, index=False, sep=' ')
```

Figure – 27 Screenshot of cell-11

In cell-11, we have written the code to create the text files for all of our training images. We are walking through all the training images in 'TrainIJCNN2013' folder and then we are extracting the name of the images so that we can match these names with the names given in the 'fileName' column of our data frame and then extract the rows where image name matches with the file names. After extracting these rows, we are creating a text file whose name will be same as the image name and finally, we are putting these extracted rows into the newly created text files. Figure – 28 shows how 'TrainIJCNN2013' folder looks before executing cell-11 and Figure – 29 shows the 'TrainIJCNN2013' folder after executing cell-11. Figure – 30 shows how one of the newly created text files '00001.txt' looks.

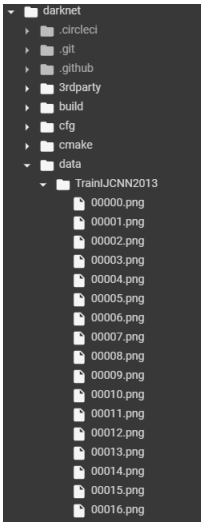


Figure – 28 Screenshot of 'TrainIJCNN2013' before executing cell-11

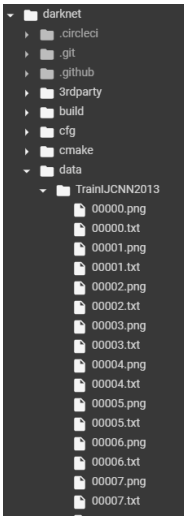


Figure – 29 Screenshot of 'TrainIJCNN2013' after executing cell-11

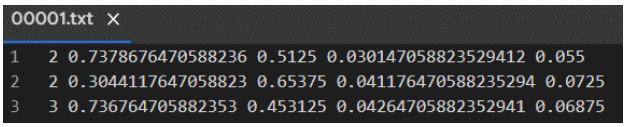


Figure – 30 Screenshot of '00001.txt'

Now we will create four new files namely 'classNames.names', 'train.txt', 'test.txt' and 'image_data.data'. These four files will be required to initiate and complete the training process. We will discuss these four files one by one. The first file is 'classNames.names'. This file should contain the names of the classes in which we have distributed our road signs. So in our case, we have four classes which are 'prohibitory', 'danger', 'mandatory' and 'other'. The order in which these class names should be written in 'classNames.names' must be in accordance with the numbering given in our data frame so for example, in the first row of the data frame, we have 'objectClass = 1' and 'objectClassLabel = danger' for the 'fileName = 00001.png' that means 'danger' should be written at the second position in the 'classNames.names' file as we are counting the index from zero. Similarly, 'prohibitory', 'mandatory' and 'other' should be written at the first, third and the fourth position respectively. The next step would be to divide the training images in 90:10 ratio where 90% of the training images will be used to train our model and the remaining 10% to validate our model. In 'train.txt' file, we will provide the complete path to 90% of the training images and in 'test.txt' file, we will provide the complete path to the remaining 10% of the training images. Now let's discuss our fourth and the final file which is 'image_data.data'. In 'image_data.data', we have to provide values to five variables. The first variable is 'classes' and the value of this variable is the total number of classes we have and in our case, we have four classes therefore value of this variable should be '4'. Then the second variable is called 'train' and the value of this variable is the path to the 'train.txt' file which we just created. The third and the fourth variables are called 'valid' and 'names' respectively where the value of 'valid' variable is the path to 'test.txt' file and the value of 'names' variable is the path to 'classNames.names' file. Finally, the fifth variable is called 'backup' and the value of this variable is the path where we want to store our weights file which will be generated while training our model. This weights file is very important to us because this weights file contains the whole training process. After every 100 iterations, the value of all the parameters like weights, biases, avg loss, mAP etc. are stored in this weights file. In other words, this weights file contains our whole YOLOv4 model. The aim of the whole training process is to obtain this weights file. This weights file will be required to test and use our model to detect and recognize road signs in various formats like images, videos and in real-

time. Coming back to the 'backup' variable, for its value, we are giving the path to the 'YOLO' folder which is present in our google drive. This means that we will be saving the weights file in our google drive and not in our local runtime storage because if the weights file is saved in the local runtime storage and the training gets interrupted in between due to the session timeout or due to the usage limit then we will lose the weights file along with the training progress saved in it and we then have to start the whole training process again from the beginning. Therefore, we will save our weights file in the google drive so that using this weights file we can resume the training process from the last saved point otherwise we have to start the whole training process from the beginning every time the session gets disconnected.

Now let's see how we implemented everything that we discussed above. In cell-12, we are making a file called 'objectLabelNames.txt' so that we can use this file later to create the 'classNames.names' file. In 'objectLabelNames.txt', we are putting the names of our four classes in the order we discussed above and later we will iterate through this file and copy its contents to create 'classNames.names' file. In cell-13, we are simply seeing the contents of 'objectLabelNames.txt' so that we can be sure that the order is correct. Sometimes cell-13 do not provide the outcome when we ran it for the first time therefore we might need to run this cell twice.

```
# cell-12
objectLabelDict = {0: 'prohibitory', 1: 'danger', 2: 'mandatory', 3: 'other'}
objectLabelNames = open('/content/darknet/data/objectLabelNames.txt', 'w')
for x in objectLabelDict:
  objectLabelNames.write(objectLabelDict[x]+'\n')

# cell-13
objectLabelNames = open('/content/darknet/data/objectLabelNames.txt', "r")
print(objectLabelNames.read())
```

```
prohibitory
danger
mandatory
other
```

Figure – 31 Screenshot of cell-12 and cell-13

Now from cell-14, we are starting the process to create 'train.txt' and 'test.txt' files. In cell-14, we are going to the 'TrainIJCNN2013' folder which contains our training dataset and then in cell-15, we are creating a list called 'path_list' in which we are storing the path to all the images present in the 'TrainIJCNN2013' folder. After that in cell-16, we are splitting the contents of 'path_list' in 90:10 ratio where 'path_list' now contains the path to 90% of the training images and 'path_list_test' contains the path to 10% of the training images.

```
# cell-14
pathToImages = '/content/darknet/data/TrainIJCNN2013'
os.chdir(pathToImages)
print(os.getcwd())
```

```
/content/darknet/data/TrainIJCNN2013
```

```
# cell-15

path_list = []

for cd, d, fls in os.walk('.'):

  for fl in fls:

    if fl.endswith('.png'):

      file_loc = pathToImages + '/' + fl

      path_list.append(file_loc+'\n')
```

```
# cell-16

path_list_test = path_list[:int(len(path_list)*0.10)]

path_list = path_list[int(len(path_list)*0.10):]
```

Figure – 32 Screenshot of cell-14, cell-15 and cell-16

In cell-17, we are using 'path_list' to create 'train.txt' file by copying the contents of 'path_list' to 'train.txt'. Similarly, we are creating the 'test.txt' file and copying the contents from 'path_list_test' to 'test.txt'. So after executing cell-17, we would have made 'train.txt' and 'test.txt'. Figure 34 and Figure 35 shows the sample of 'train.txt' and 'test.txt'.

```
# cell-17

with open('/content/darknet/data/train.txt', 'w') as train:
  for i in path_list:
    train.write(i)


with open('/content/darknet/data/test.txt', 'w') as test:
  for i in path_list_test:
    test.write(i)
```

Figure – 33 Screenshot of cell-17

train.txt × test.txt

```
1  /content/darknet/data/TrainIJCNN2013/00027.png
2  /content/darknet/data/TrainIJCNN2013/00496.png
3  /content/darknet/data/TrainIJCNN2013/00404.png
4  /content/darknet/data/TrainIJCNN2013/00334.png
5  /content/darknet/data/TrainIJCNN2013/00047.png
6  /content/darknet/data/TrainIJCNN2013/00510.png
7  /content/darknet/data/TrainIJCNN2013/00424.png
8  /content/darknet/data/TrainIJCNN2013/00018.png
9  /content/darknet/data/TrainIJCNN2013/00337.png
10 /content/darknet/data/TrainIJCNN2013/00065.png
```

Figure – 34 Sample of 'train.txt'

train.txt test.txt ×

```
1  /content/darknet/data/TrainIJCNN2013/00588.png
2  /content/darknet/data/TrainIJCNN2013/00565.png
3  /content/darknet/data/TrainIJCNN2013/00218.png
4  /content/darknet/data/TrainIJCNN2013/00149.png
5  /content/darknet/data/TrainIJCNN2013/00520.png
6  /content/darknet/data/TrainIJCNN2013/00028.png
7  /content/darknet/data/TrainIJCNN2013/00123.png
8  /content/darknet/data/TrainIJCNN2013/00341.png
9  /content/darknet/data/TrainIJCNN2013/00579.png
10 /content/darknet/data/TrainIJCNN2013/00244.png
```

Figure – 35 Sample of 'test.txt'

Then in cell-18, we are creating our ‘classNames.names’ file and copying all the contents from ‘objectLabelNames.txt’ into it. Figure-37 shows how our ‘classNames.names’ file looks.

```
# cell-18

# initialise the counter
i = 0

pathToDataFolder = '/content/darknet/data'

with open(pathToDataFolder+'/'+'classNames.names','w') as cls, open(pathToDataFolder+'/'+'objectLabelNames.txt','r') as text:

  for l in text:
    cls.write(l)

    i += 1
```

Figure – 36 Screenshot of cell-18

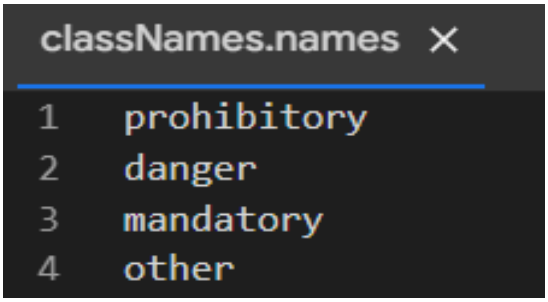


Figure – 37 Screenshot of ‘classNames.names’

Finally in cell-19, we are creating our ‘image_data.data’ file and in it, we are creating five variables and giving value to them in the way we discussed above. Cell-20 shows us the contents of ‘image_data.data’ file and Figure-39 shows what our newly created ‘image_data.data’ file actually looks. With the execution of cell-20, we have completed the data processing part to build our YOLOv4 model. The data processing part to build YOLOv4-tiny would be similar so all the cells from 1-20 in ‘yolov4.ipynb’ and in ‘yolov4tiny.ipynb’ are exactly same.

```
# cell-19

with open(pathToDataFolder+'/'+'image_data.data', 'w') as RequiredData:

  RequiredData.write('classes = ' + str(i) + '\n')


  RequiredData.write('train = ' + pathToDataFolder + '/' + 'train.txt' + '\n')


  RequiredData.write('valid = ' + pathToDataFolder + '/' + 'test.txt' + '\n')


  RequiredData.write('names = ' + pathToDataFolder + '/' + 'classNames.names' + '\n')


  RequiredData.write('backup = /content/drive/MyDrive/YOLO/')
```

```
# cell-20

imageData = open('/content/darknet/data/image_data.data', "r")
print(imageData.read())
```

```
classes = 4
train = /content/darknet/data/train.txt
valid = /content/darknet/data/test.txt
names = /content/darknet/data/classNames.names
backup = /content/drive/MyDrive/YOLO/
```

Figure – 38 Screenshot of cell-19 and cell-20

image_data.data ×

```
1  classes = 4
2  train = /content/darknet/data/train.txt
3  valid = /content/darknet/data/test.txt
4  names = /content/darknet/data/classNames.names
5  backup = /content/drive/MyDrive/YOLO/
```

Figure – 39 Screenshot of 'image_data.data'

3.4 Training

Now we are preparing to start the training process. For different editions of YOLO for example YOLOv4 and YOLOv4-tiny, we have different configurations or .cfg files. So for YOLOv4, we have 'yolov4-custom.cfg' and for YOLOv4-tiny, we have 'yolov4-tiny-custom.cfg'. These cfg files contain parameters and convolution layers which determines how the whole training process will be carried out.

Let's first discuss the training process for our YOLOv4 model. As mentioned above, we will be using 'yolov4-custom.cfg' file to train our YOLOv4 model. We will make two copies of 'yolov4-custom.cfg', one for training our model and another for testing our model so that we can avoid making alterations in 'yolov4-custom.cfg' again and again. This will save our time and effort and will also prevent us from making mistakes. So we have named the first copy as 'yolov4-custom_train.cfg' which will be used to train our model. The second copy is called 'yolov4-custom_test.cfg' and we will use it to test our model. Now there are lots of parameters present in these configuration files and we need to change the values of some of the parameters. We will first make changes in 'yolov4-custom_train.cfg'. In 'yolov4-custom_train.cfg', we will keep 'batch=64' but will change the value of 'subdivisions' from 16 to 32. If we have 'batch=64' and 'subdivision=32' that means in one iteration we are passing 64 images and in every subdivision we have 2 images. We have kept the value of 'subdivisions' equal to 32, we could have kept it 16 but this will utilize more GPU memory and we have limited GPU resources that's why we kept it to 32 but if we keep a higher value for 'subdivisions' like 32 or 64 then it would take longer to complete the training. The next parameter is 'max_batches' and it refers to the total number of iterations that will be done during the whole training process. Now to calculate the value of this parameter we use a formula which is 'max_batches = Total number of classes * 2000' and we have a total of 4 classes therefore for our model 'max_batches' is equal to 8000. Here we need to remember that the minimum value of 'max_batches' should always be equal to 6000 so for example if we had 1,2 or 3 classes then the value of 'max_batches' would be 6000. The next parameter is called 'steps' and the value of 'steps' is always equals to 90% and 80% of the 'max_batches' therefore in our case 'steps = 7200,6400'. This 'steps' parameter determines that if 'policy = steps' then the 'learning_rate' will be adjusted after 6400 and 7200 batches or iterations.

Now in 'yolov4-custom_train.cfg', we have three [yolo] layers in which we have a parameter called 'classes'. The value of this parameter should be equal to the total number of classes we have therefore in our case, the value of this 'classes' parameter should be equal to 4. We have three [yolo] layers in 'yolov4-custom_train.cfg' and we will change the value of 'classes' to 4 in all the three [yolo] layers. Then we will make changes in [convolution] layers but only in those [convolution] layers which are present just above the [yolo] layers. Therefore, we have three such [convolution] layers. In these [convolution] layers, we will change the value of 'filters' parameter from 255 to 27. There is a formula to calculate the value of this 'filters' parameter which is 'filters = (Total number of classes + 5) * 3' and in our case we have total 4 classes therefore 'filters = 27' and this 'filters' parameter represents the number of convolution kernels.

We have some other parameters like 'width' and 'height' which represents the network size. The values of 'width' and 'height' parameters should be a multiple of 32. If we keep the value of 'width' and 'height' on the higher side for example 608 then this will increase the mAP of our model but at the same time this will also increase the training time and decrease the FPS of our model. Whereas if we keep lower values for 'width' and 'height' for example

416 then it will decrease the mAP for our model but will also decrease the training time and increase the FPS of the model. We are using the default values of 'width' and 'height' parameters which are 608 for our YOLOv4 model and 416 for our YOLOv4-tiny model.

Then we have parameters for Data Augmentation like 'angle', 'saturation', 'exposure', 'hue', 'cutmix'. 'mosaic' etc. and for Optimization like 'momentum', 'decay', 'learning_rate' etc. but we are not making any alterations to these parameters. Now in 'yolov4-custom_test.cfg', we have set the values of both the 'batch' and the 'subdivisions' to 1. The value of other parameters in 'yolov4-custom_test.cfg' like 'max_batches', 'steps', 'filters' and 'classes' will be changed in a similar way as we did in 'yolov4-custom_train.cfg'.

Figure – 40 shows the screenshot of cell-21 of the 'yolov4.ipynb' where we are first making copies of 'yolov4-custom.cfg' and changing the values of the parameters in these copies as we discussed above.

```
# cell-21

print(os.getcwd())
os.chdir('/content/darknet')

!cp cfg/yolov4-custom.cfg cfg/yolov4-custom_train.cfg
!cp cfg/yolov4-custom.cfg cfg/yolov4-custom_test.cfg

%cd cfg
#!sed -i 's/batch=64/batch=32/' yolov4-custom_train.cfg
!sed -i 's/subdivisions=16/subdivisions=32/' yolov4-custom_train.cfg
!sed -i 's/max_batches = 500500/max_batches = 8000/' yolov4-custom_train.cfg
!sed -i 's/steps=400000,450000/steps=7200,6400/' yolov4-custom_train.cfg
!sed -i 's/filters=255/filters=27/' yolov4-custom_train.cfg
!sed -i 's/classes=80/classes=4/' yolov4-custom_train.cfg

!sed -i 's/batch=64/batch=1/' yolov4-custom_test.cfg
!sed -i 's/subdivisions=16/subdivisions=1/' yolov4-custom_test.cfg
!sed -i 's/max_batches = 500500/max_batches = 8000/' yolov4-custom_test.cfg
!sed -i 's/steps=400000,450000/steps=7200,6400/' yolov4-custom_test.cfg
!sed -i 's/filters=255/filters=27/' yolov4-custom_test.cfg
!sed -i 's/classes=80/classes=4/' yolov4-custom_test.cfg
%cd ..
```

Figure – 40 Screenshot of cell-21 from 'yolov4.ipynb'

Now in YOLO, we can start the training process from the very scratch or we can use YOLOv4 pre-trained weights for the convolution layers present inside our cfg file to speed up the training process. Different editions of YOLO have different pre-trained weights file which can be used with them as they are the most compatible ones and are already trained with them on the coco dataset. So for example, 'yolov4-custom.cfg' has its own pretrained weights file called 'yolov4.conv.137' which is trained up to 137 convolution layers and we are going to use this pre-trained weights file to train our YOLOv4 model. In cell-22 of 'yolov4.ipynb', we are downloading the 'yolov4.conv.137' pre-trained weights file from the AlexeyAB's github repository.

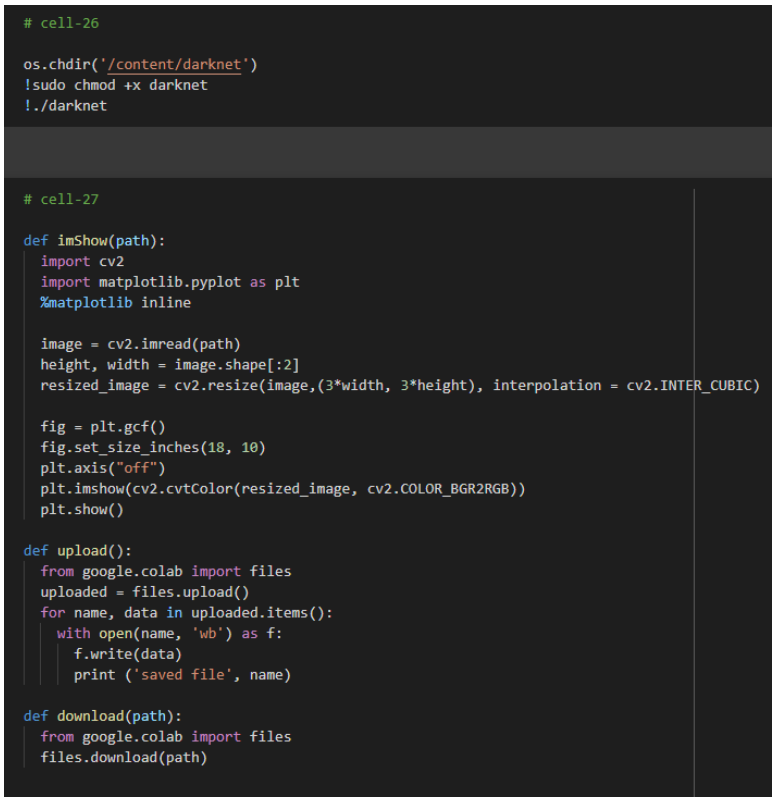
```
# cell-22

os.chdir('/content/darknet')
print(os.getcwd())
!wget https://github.com/AlexeyAB/darknet/releases/download/darknet_yolo_v4_pre/yolov4.conv.137
```

Figure – 41 Screenshot of cell-22 from 'yolov4.ipynb'

Now in cell-23, we are checking that if we actually have the access to the GPU services provided by google colab. Then in cell-24 and cell-25, we are going into the darknet folder where we have our 'Makefile'. We are making some changes in this 'Makefile' like we are enabling 'OPENCV', 'GPU' etc. so that we can use these services. The code written in cell-25 has been taken from [19].

Figure – 42 Screenshot of cell-23, 24 and 25 from 'yolov4.ipynb'

In cell-26, we are providing the permissions to our darknet folder. Then in cell-27, we have defined some functions for example 'imShow(path)' which we will use later to display the images. We have two more functions 'upload' and 'download(path)' which can be used to upload and download the files. The code written in cell-27 has been taken from [21].



```
# cell-26

os.chdir('/content/darknet')
!sudo chmod +x darknet
!./darknet
```

```
# cell-27

def imShow(path):
  import cv2
  import matplotlib.pyplot as plt
  %matplotlib inline

  image = cv2.imread(path)
  height, width = image.shape[:2]
  resized_image = cv2.resize(image,(3*width, 3*height), interpolation = cv2.INTER_CUBIC)

  fig = plt.gcf()
  fig.set_size_inches(18, 10)
  plt.axis("off")
  plt.imshow(cv2.cvtColor(resized_image, cv2.COLOR_BGR2RGB))
  plt.show()

def upload():
  from google.colab import files
  uploaded = files.upload()
  for name, data in uploaded.items():
    with open(name, 'wb') as f:
      f.write(data)
      print ('saved file', name)

def download(path):
  from google.colab import files
  files.download(path)
```

Figure – 43 Screenshot of cell-26 and 27 from 'yolov4.ipynb'

Finally, in cell-28, we are providing the command to start the training of our YOLOv4 model. Here to initiate the training, we required three files which we have prepared till now. These three files are 'image_data.data', 'yolov4-custom_train.cfg' and 'yolov4.conv.137'. The training will start after executing cell-28 and with the time we will see that as the training continues, avg loss will keep on decreasing and mAP will keep on increasing as more and more iterations will be carried out. During the training process, YOLO will create some weights file in which it will keep on saving the training progress of our model. 'yolov4-custom_train_last.weigths' is one such weights file that YOLO will first create after we start the training. The progress after every 100 iterations will be saved in this file. YOLO will also create 'yolov4-custom_train_best.weights' file in which it will save the weights where accuracy or the mAP was the highest. YOLO will keep on updating these weights file as the training continues. YOLO will also create separate weights file after every 1000 iterations and will save it with the name 'yolo-custom_train_1000.weights', 'yolo-custom_train_2000.weights' till the 'yolo-custom_train_8000.weights'. Our YOLO model will perform 8000 iterations because in 'yolov4-custom_train.cfg', we have set the value of 'max_batches' equal to 8000. After 8000 iterations are completed, YOLO will create one last weights file called 'yolov4-custom_train_final.weights' in which it will save the final weights. The execution of our cell-28 stopped after 7 to 8 hours because google colab even with the pro version provides limited access to the GPU services at a time and therefore in cell-29, we have provided the command to resume the training from the last saved point. Here in cell-29, instead of 'yolov4.conv.137', we are using 'yolov4-custom_train_last.weights' to resume the training as YOLO saved the training weights of the last 100^{th} iterations in this file. The execution of cell-29 also stops after the same time period as cell-28 so we will run cell-29 for the second time. In total it took 20 to 24 hours to completely train our YOLOv4 model. All the weights file which will be created during the execution of cell-28 and cell-29 will be saved in the 'YOLO' folder which is present in our google drive. This is because for the value of the 'backup' variable in the 'image_data.data' file, we provided the path to the 'YOLO' folder present in our google drive and this was important otherwise we would have loosed all the weights file when the session gets disconnected.

```
# cell-28

!./darknet detector train data/image_data.data cfg/yolov4-custom_train.cfg yolov4.conv.137 -map -dont_show
```

Figure – 44 Screenshot of cell-28 from 'yolov4.ipynb'

```
# cell-29

!./darknet detector train data/image_data.data cfg/yolov4-custom_train.cfg /content/drive/MyDrive/YOLO/yolov4-custom_train_last.weights -map -dont_show
```

Figure – 45 Screenshot of cell-29 from 'yolov4.ipynb'

The training process for our YOLOv4-tiny model is exactly similar to the YOLOv4 model. This time in cell-21 of 'yolov4tiny.ipynb', we are making two copies of 'yolov4-tiny-custom.cfg' and then changing parameters in these two copies in a similar way we discussed above. Here in 'yolov4-tiny-custom.cfg', we have only two [yolo] layers unlike in 'yolov4-custom.cfg' where we had three [yolo] layers. Therefore values of 'filters' and 'classes' parameters would only be changed two times each in the copies of 'yolov4-tiny-custom.cfg' as compared to the copies of 'yolov4-custom.cfg' where values of both the parameters have been changes three times each. Then in cell-22, for our YOLOv4-tiny model, we are using 'yolov4-tiny.conv.29' pre-trained weights file which is trained up to 29 convolution layers. Cells from 23 to 27 of 'yolov4tiny.ipynb' are exactly same as the cells from 23 to 27 in 'yolov4.ipynb'.

```
# cell-21

print(os.getcwd())
os.chdir('/content/darknet')

!cp cfg/yolov4-tiny-custom.cfg cfg/yolov4-tiny-custom_train.cfg
!cp cfg/yolov4-tiny-custom.cfg cfg/yolov4-tiny-custom_test.cfg

%cd cfg
#!sed -i 's/batch=64/batch=32/' yolov4-tiny-custom_train.cfg
!sed -i 's/subdivisions=1/subdivisions=32/' yolov4-tiny-custom_train.cfg
!sed -i 's/max_batches = 500200/max_batches = 8000/' yolov4-tiny-custom_train.cfg
!sed -i 's/steps=400000,450000/steps=7200,6400/' yolov4-tiny-custom_train.cfg
!sed -i 's/filters=255/filters=27/' yolov4-tiny-custom_train.cfg
!sed -i 's/classes=80/classes=4/' yolov4-tiny-custom_train.cfg

!sed -i 's/batch=64/batch=1/' yolov4-tiny-custom_test.cfg
#!sed -i 's/subdivisions=1/subdivisions=1/' yolov4-tiny-custom_test.cfg
!sed -i 's/max_batches = 500200/max_batches = 8000/' yolov4-tiny-custom_test.cfg
!sed -i 's/steps=400000,450000/steps=7200,6400/' yolov4-tiny-custom_test.cfg
!sed -i 's/filters=255/filters=27/' yolov4-tiny-custom_test.cfg
!sed -i 's/classes=80/classes=4/' yolov4-tiny-custom_test.cfg
%cd ..
```

Figure – 46 Screenshot of cell-21 from 'yolov4tiny.ipynb'

```
# cell-22

os.chdir('/content/darknet')
print(os.getcwd())
!wget https://github.com/AlexeyAB/darknet/releases/download/darknet_yolo_v4_pre/yolov4-tiny.conv.29
```

Figure – 46 Screenshot of cell-22 from 'yolov4tiny.ipynb'

Then in cell-28 of 'yolov4tiny.ipynb', we are again giving the command to start the training using the files 'image_data.data', 'yolov4-tiny-custom_train.cfg' and 'yolov4-tiny.conv.29'. Cell-29 contains the command to resume the training. During the training process of our YOLOv4-tiny model, the weights file will be created in a similar way as they were created while training our YOLOv4 model. The weights file will get stored in the 'YOLO' folder

present in our google drive. The whole training of our YOLOv4-tiny model is completed within 7 to 8 hours and therefore we didn't need to execute cell-29. One of the reasons that the training process was fast for our YOLOv4-tiny model is that the 'width' and 'height' parameters in the 'yolov4-tiny-custom_train.cfg' has the value set to 416 as compared to 608 in 'yolov4-custom_train.cfg'. Another reason could be that in 'yolov4-tiny-custom_trian.cfg', we have less number of [yolo] and [convolution] layers as compared to the 'yolov4-custom_train.cfg'.

```
# cell-28

!./darknet detector train data/image_data.data cfg/yolov4-tiny-custom_train.cfg yolov4-tiny.conv.29 -map -dont_show
```

Figure – 47 Screenshot of cell-28 from 'yolov4tiny.ipynb'

```
# cell-29

!./darknet detector train data/image_data.data cfg/yolov4-tiny-custom_train.cfg /content/drive/MyDrive/YOLO/yolov4-tiny-custom_train_last.weights -map -dont_show
```

Figure – 48 Screenshot of cell-29 from 'yolov4tiny.ipynb'

Now here I would like to state that in the starting I was trying to train my YOLOv4 model using the Mapillary Traffic Sign Dataset. To train the YOLOv4 model on this dataset, I moved all the training and the validation images in one folder which I called 'train_valid'. This 'train_valid' folder contains a total of 41909 images and was around 33 GB in size. I followed the same steps in the same way as I have described above to train the YOLOv4 model using the Mapillary Traffic Sign Dataset. I created a separate folder for it on my google drive with the name 'YOLO-MTSD' and then I upload the 'train_valid' and the annotations folders in the 'YOLO-MTSD' folder on my google drive. I then created the annotation files in the YOLO format. I also created the other necessary files needed to start the training. I then started the training process and after training it for around 20 hours I realized that it is not a good idea to train our model on the Mapillary Traffic Sign Dataset because even after 20 hours of training the model has completed only 3581 iterations and it was showing that the time remaining to complete the training was around 3455 hours which is not feasible to train for. Another important thing was that in Mapillary Traffic Sign Dataset, we had 401 unique classes and we saw above that if we have 4 unique classes then it took us 20 to 24 hours to train our model therefore imagine the time period it will take to train our model using Mapillary Traffic Sign Dataset with the type of computing resources which we have. After completing 3581 iterations, the avg loss which we were getting was around 5.06 which was very high and the mAP was around 0.05% with which it is not possible to detect and recognize anything. Because of all these issues majorly because of the size of Mapillary Traffic sign Dataset and limited availability of the computing resources, I discussed these problems with my supervisor and finally decided to use the GTSDB Dataset for this project. Figure-49 shows the screenshot of the last iteration where we had to stop the training process for our YOLOv4 model on the Mapillary traffic Sign Dataset.

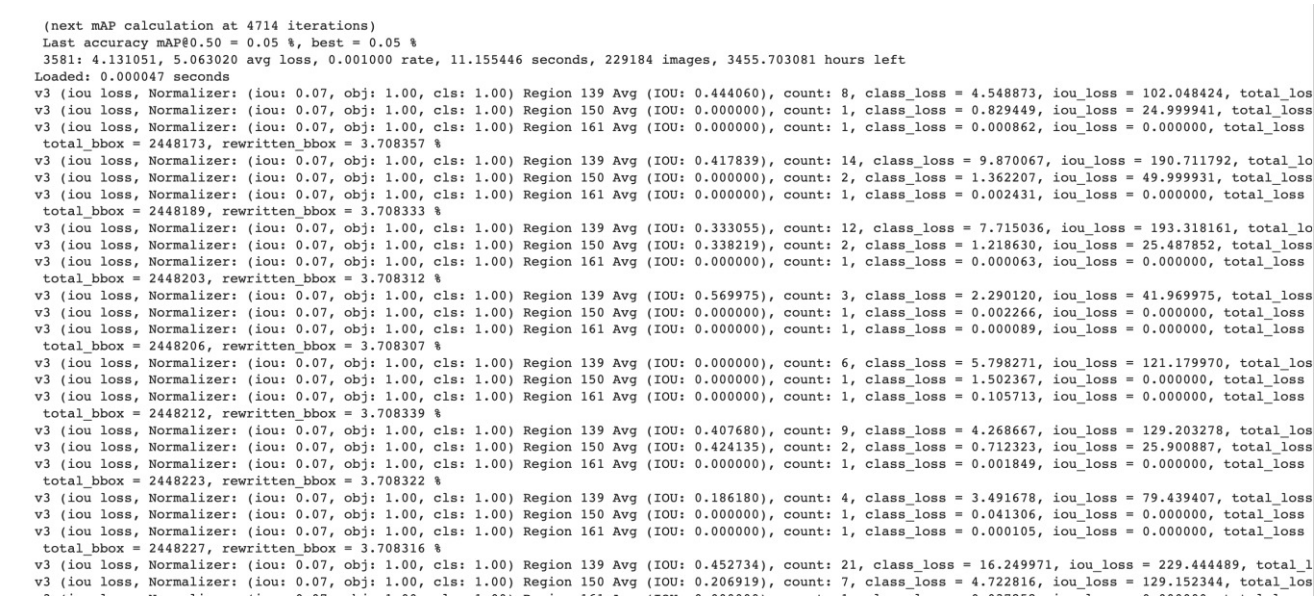


Figure – 49 Screenshot of the last iteration from the training process of our YOLOv4 model using MTSD

3.5 Results and Testing

Now during the training process, YOLO generates a graph to show the performance of our weights. This graph displays the avg loss and the mAP of our model. In the graph, we were able to see that as the iterations are carried on, the avg loss keeps on decreasing and the mAP keeps on increasing. Unfortunately, we don't have that graph because that graph is only available until the training process keeps on running. If due to any reason the training gets interrupted in between then we will lose the graph and will also lose the progress shown in the graph. When we resume the training, this graph will be available again but it will show the progress from the current iteration and the graph for the previous iterations will be lost forever as the lines of avg loss and mAP in the graph are updated with the iterations in the real-time until the training continues and our training process got interrupted because of the limited access to the GPU runtime. To bypass this issue, we have a command using which we can check the accuracy or the performance of our model. This command gives us the mAP along with IoU, precision, recall and F1-score for the weights file generated during the training process. In total, we have 11 such weights file for both the models which we have already discussed above. For our YOLOv4 model, we have one weights file for every 1000 iterations from 1000 to 8000 having their names in the format 'yolov4-custum_train_xxxx.weights' where xxxx represents values from 1000, 2000 … to 8000 and the remaining three files have last, final and best in the place of xxxx. The same thing goes for our YOLOv4-tiny model but the format of the weights filename is 'yolov4-tiny-custom_train_xxxx.weights'.

Now in cells 30 to 40 of both the files namely 'yolov4.ipynb' and 'yolov4tiny.ipynb', we are using this command to check the performance of all the weights file. To show some of the samples of this command from both the files, Figure-50 and Figure-51 shows the command to find the performance of 'yolov4-custom_train_8000.weights' and 'yolov4-custom_train_best.weights' respectively which were created during the training process of the YOLOv4 model. Similarly, Figure-52 and Figure-53 shows the command to find the performance of 'yolov4-tiny-custom_train_8000.weights' and 'yolov4-tiny-custom_train_best.weights' respectively which were created during the training process of the YOLOv4-tiny model. Before cell-30, we were using 'yolov4-custom_train.cfg' and 'yolov4-tiny-custom_train.cfg' to train our YOLOv4 and our YOLOv4-tiny model respectively but from cell-30 we will be using 'yolov4-custom_test.cfg' and 'yolov4-tiny-cutom_test.cfg' files to check the performance and to do the testing of our YOLOv4 and YOLOv4-tiny model respectively.

```
# cell-37

!./darknet detector map data/image_data.data cfg/yolov4-custom_test.cfg /content/drive/MyDrive/YOLO/yolov4-custom_train_8000.weights -points 0
```

Figure – 49 Screenshot of cell-37 from 'yolov4.ipynb'

```
# cell-40

!./darknet detector map data/image_data.data cfg/yolov4-custom_test.cfg /content/drive/MyDrive/YOLO/yolov4-custom_train_best.weights -points
```

Figure – 50 Screenshot of cell-40 from 'yolov4.ipynb'

```
# cell-37

!./darknet detector map data/image_data.data cfg/yolov4-tiny-custom_test.cfg /content/drive/MyDrive/YOLO/yolov4-tiny-custom_train_8000.weights -points 0
```

Figure – 51 Screenshot of cell-37 from 'yolov4tiny.ipynb'

Figure – 52 Screenshot of cell-40 from 'yolov4tiny.ipynb'

Now to summarize the output given by the cells from 30 to 40 in both the files, we have created two tables, one for the results of the cells from 30 to 40 in the 'yolov4.ipynb' file and another for the results given by the cells 30 to 40 in the 'yolov4.ipynb' file.

Parameters	1000	2000	3000	4000	5000	6000	7000	8000	last	final	Best
mAP	94.09%	100 %	100 %	100 %	100 %	100 %	100 %	100 %	100 %	100 %	100 %
IoU	76.27%	89.82%	90.75%	91.41%	91.36%	89.80%	92.69%	93.05%	93.05%	93.05%	92.36%
precision	0.90	1.0	1.0	1.0	1.0	1.0	1.0	0.99	0.99	0.99	1.0
recall	0.99	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
F1-score	0.94	1.0	1.0	1.0	1.0	1.0	1.0	0.99	0.99	0.99	1.0

Figure – 53 Summary of the results given by the execution of cells 30 to 40 in 'yolov4.ipynb'

Parameters	1000	2000	3000	4000	5000	6000	7000	8000	last	final	best
mAP	70.58%	92.06%	95.69%	97.15%	97.03%	97.32%	95.39%	97.23%	97.23%	97.23%	99.66%
IoU	42.07%	64.53%	74.73%	77.51%	80.45%	80.29%	78.78%	81.34%	81.34%	81.34%	80.56%
precision	0.60	0.86	0.92	0.95	0.96	0.96	0.95	0.92	0.92	0.92	0.96
recall	0.80	0.95	0.97	0.99	0.97	0.99	0.97	0.99	0.99	0.99	0.99
F1-score	0.68	0.90	0.95	0.97	0.97	0.97	0.96	0.95	0.95	0.95	0.97

Figure – 54 Summary of the results given by the execution of cells 30 to 40 in 'yolov4tiny.ipynb'

Now to better visualize and compare the performance of both the models, we are plotting graphs of mAP, IoU, precision, recall and F1-score using the values given in the tables above to evaluate the accomplishment of both the models. We are plotting the graphs for all 8000 iterations and these graphs especially the mAP one will give us an accurate idea about the performance of the weights which were generated during the training process.

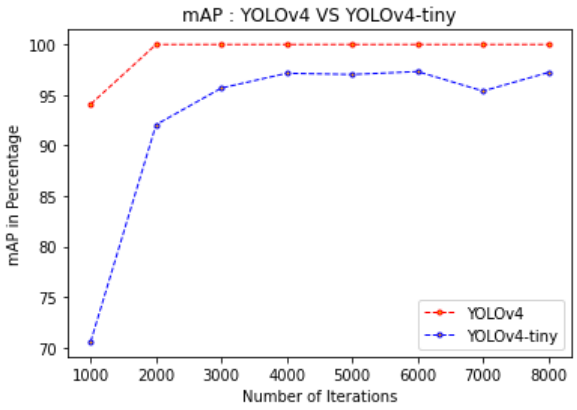


Figure-55 mAP Graph for YOLOv4 vs YOLOv4-tiny

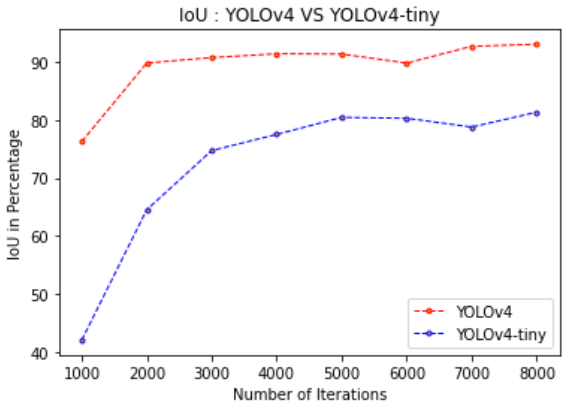


Figure-56 IoU Graph for YOLOv4 vs YOLOv4-tiny

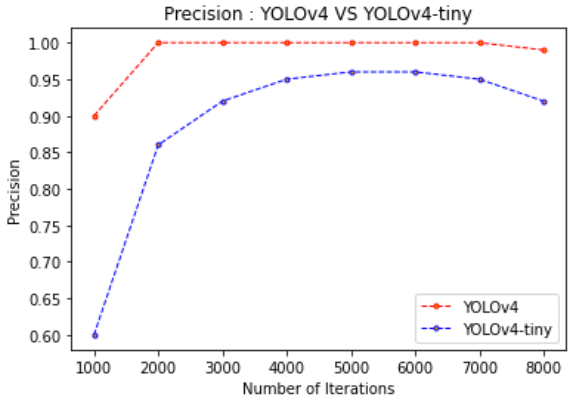


Figure-57 precision : YOLOv4 vs YOLOv4-tiny

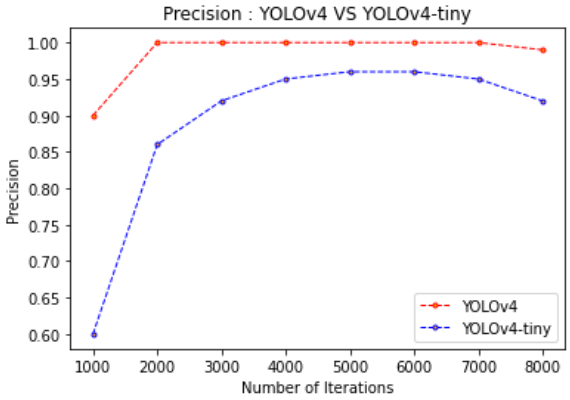


Figure-58 recall: YOLOv4 vs YOLOv4-tiny

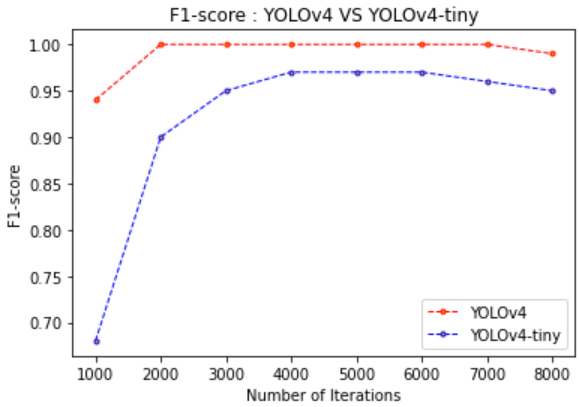


Figure-59 F1-score : YOLOv4 vs YOLOv4-tiny

From the figures above, we can see that the 'YOLOv4' model is performing slightly better than the 'YOLOv4-tiny' model. Although there is not much difference in the performance of both the models. If we talk about the mAP then initially in the starting for the first 1000 iterations, there is a significant gap in the performance of both the models as for the first 1000 iterations we are getting 94.09% mAP for the YOLOv4 model as compared to just 70.58% mAP for the YOLOv4-tiny model. But we can see a substantial dip in the gap between the performance of both the models as after the completion of 8000 iterations the mAP for the YOLOv4 model is 100% and the mAP for

our YOLOv4-tiny model is 97.23%. From the tables given in Figure – 53 and Figure – 54, we can also conclude that the last and the final weights file are the exact replicas of the 8000th weights file in their respective YOLO models as in both the tables the values given in the column of '8000', 'last' and 'final' are exactly similar. Now when it comes to finally determining the accuracy of our YOLO models, then we take the best weights into account. The best weights for our YOLOv4 model gives us 100% mAP and the best weights for our YOLOv4-tiny model gives us 99.66% mAP and this shows that both the models perform equally well. We will be using these best weights for the purpose of testing as well therefore to test our YOLOv4 model, we will be using the 'yolov4-custom_train_best.weights' file and to test our YOLOv4-tiny model, we will use 'yolov4-tiny-custom_train_best.weights' file.

Now during the dataset processing, we have prepared a file called 'test.txt' in which we provided the path to 10% of the training images. Figure-62 displays the sample of the 'test.txt'. In cell-41 of 'yolov4.ipynb' and 'yolov4tiny.ipynb', we are using both the YOLO models to check their performance on the images whose paths are mentioned in the 'test.txt' file. The outcome of cell-41 from both the colab files will be transferred to a json file which we can download to our computer's local storage. Figure-63 and Figure-64 shows the sample of json file which contains the result of cell-41 for our YOLOv4 and YOLOv4-tiny model respectively.

```
# cell-41

os.chdir('/content/darknet')
!./darknet detector test data/image_data.data cfg/yolov4-custom_test.cfg /content/drive/MyDrive/YOLO/yolov4-custom_train_best.weights -ext_output -dont_show -out resu
download('result.json')
```

Figure – 60 Screenshot of cell-41 from 'yolov4.ipynb'

```
# cell-41

os.chdir('/content/darknet')
!./darknet detector test data/image_data.data cfg/yolov4-tiny-custom_test.cfg /content/drive/MyDrive/YOLO/yolov4-tiny-custom_train_best.weights -ext_output -dont_show
download('result.json')
```

Figure – 61 Screenshot of cell-41 from 'yolov4tiny.ipynb'

train.txt test.txt

```
1  /content/darknet/data/TrainIJCNN2013/00588.png
2  /content/darknet/data/TrainIJCNN2013/00565.png
3  /content/darknet/data/TrainIJCNN2013/00218.png
4  /content/darknet/data/TrainIJCNN2013/00149.png
5  /content/darknet/data/TrainIJCNN2013/00520.png
6  /content/darknet/data/TrainIJCNN2013/00028.png
7  /content/darknet/data/TrainIJCNN2013/00123.png
8  /content/darknet/data/TrainIJCNN2013/00341.png
9  /content/darknet/data/TrainIJCNN2013/00579.png
10 /content/darknet/data/TrainIJCNN2013/00244.png
```

Figure – 62 Sample of 'test.txt'

```
[
{
 "frame_id":1,
 "filename":"/content/darknet/data/TrainIJCNN2013/00588.png",
 "objects": [

 ]
},
{
 "frame_id":2,
 "filename":"/content/darknet/data/TrainIJCNN2013/00565.png",
 "objects": [
  {"class_id":3, "name":"other", "relative_coordinates":
{"center_x":0.549779, "center_y":0.441587, "width":0.023357,
"height":0.043676}, "confidence":0.992254}
 ]
},
{
 "frame_id":3,
 "filename":"/content/darknet/data/TrainIJCNN2013/00218.png",
 "objects": [
  {"class_id":2, "name":"mandatory", "relative_coordinates":
{"center_x":0.600745, "center_y":0.557893, "width":0.013928,
"height":0.022587}, "confidence":0.986618}
 ]
},
```


Figure – 63 Sample of the result of cell-41 from 'yolov4.ipynb'


```
[
{
 "frame_id":1,
 "filename":"/content/darknet/data/TrainIJCNN2013/00588.png",
 "objects": [

 ]
},
{
 "frame_id":2,
 "filename":"/content/darknet/data/TrainIJCNN2013/00565.png",
 "objects": [
  {"class_id":3, "name":"other", "relative_coordinates":
{"center_x":0.550362, "center_y":0.439647, "width":0.024296,
"height":0.042300}, "confidence":0.996997}
 ]
},
{
 "frame_id":3,
 "filename":"/content/darknet/data/TrainIJCNN2013/00218.png",
 "objects": [
  {"class_id":2, "name":"mandatory", "relative_coordinates":
{"center_x":0.601640, "center_y":0.556118, "width":0.014465,
"height":0.023790}, "confidence":0.997815}
 ]
},
```


Figure – 64 Sample of the result of cell-41 from 'yolov4tiny.ipynb'


```
3 0.5496323529411765 0.44125 0.02426470588235294 0.0425
```


Figure – 65 Screenshot of '00565.txt'


```
2 0.6011029411764706 0.558125 0.013970588235294118 0.02375
```


Figure – 66 Screenshot of '00218.txt'

As we can see in Figure-63 and in Figure-64 that both the YOLO models have not detected any road sign in the image '00588.png' because of the fact that we don't actually have any road signs in the '00588.png'. This can be verified by looking at the '00588.txt' file which is empty as there are no annotations for '00588.png'. Then in image '00565.png', both our YOLO models are detecting and recognizing the road sign as 'other' where YOLOv4 is giving us 99.22% confidence or the probability score and YOLOv4-tiny is giving 99.69% probability score. Apart from this high probability score, both the models are perfectly identifying the location of the road sign in the image. All of this can be verified from Figure-65 showing the screenshot of '00565.txt' which contains bounding box annotations for the road signs present in '00565.png'. The bounding box annotations which our YOLOv4 and YOLOv4-tiny models have found for '00565.png' are very very accurate and close to the values of annotations given in '00565.txt'. In '00218.png', both our YOLO models have detected and recognised the road sign as 'mandatory' where the YOLOv4 is giving the probability score of 98.66% and the probability score for YOLOv4-tiny is 99.78%. Again the bounding box annotations given by both the models are accurate and this can be verified from Figure-66 showing screenshot of '00218.txt' file which contains the bounding box annotation and the class of the road signs present in '00218.png'.

Now we will test both our YOLO models on the test images of the GTSDB Dataset. These are the test images therefore we don't have any annotations for them. These test images are present in 'TestIJCNN2013' folder which we uploaded on our google drive. In cell-42, we are first extracting 'TestIJCNN2013' folder in our local runtime storage and then in cell-43, we are converting the format of these images by changing their name extension from ppm to png. Then finally in cell-44, we are going into the

darknet folder and providing permissions to it as now we will be testing our YOLO models on the random images from the 'TestIJCNN2013' folder. The cells from 42 to 44 in both the colab files are exactly same.

```
# cell-42

os.chdir('/content')
!unzip '/content/drive/MyDrive/YOLO/TestIJCNN2013.zip'
```

```
# cell-43

for dir, subdirs, fls in os.walk("/content/TestIJCNN2013Download"):
    for fl in fls:
        if fl.endswith('.ppm'):
            newFileName = fl[:-4] + '.png'
            os.rename(os.path.join("/content/TestIJCNN2013Download", fl), os.path.join("/content/TestIJCNN2013Download", newFileName))
```

```
# cell-44

os.chdir('/content/darknet')
!sudo chmod +x darknet
!./darknet
```

Figure – 67 Screenshot of cell-42, 43 and 44 from 'yolov4.ipynb'

Now in cells 45 to 49 in both the colab files, we are accessing a random image from the 'TrainIJCNN2013' folder and then we are testing both the YOLO models on that image using the test configuration file and the best weights. To discuss two of the cells from 45 to 49, in cell-45, we are testing both the models on '00155.png'. Both YOLOv4 and YOLOv4-tiny are able to detect and recognize three road signs in '00155.png' which are 'mandatory', 'prohibitory' and 'other' with the probability of 99% , 99% and 100% for the YOLOv4 as seen in Figure-70 and the probability of 99%, 100% and 100% for the YOLOv4t-tiny as seen in the Figure-72. Along with probability of the class, our YOLO models are also giving us the location where the road signs are detected in the image. Then in cell-47 of both the colab files, we are testing our YOLO models on image '00269.png'. YOLOv4 and YOLOv4-tiny, both the models have detected and recognized the road signs as 'danger' with the 100% probability. The results for YOLOv4 are shown in Figure-75 and the results for YOLOv4-tiny are displayed in Figure-77.

Figure – 68 Screenshot of '00155.png'

```
# cell-45

os.chdir('/content/darknet')
img_path = "/content/TestIJCNN2013Download/00155.png"
img_initial = cv2.imread(img_path, cv2.IMREAD_UNCHANGED)
cv2_imshow(img_initial)
!./darknet detector test data/image_data.data cfg/yolov4-custom_test.cfg /content/drive/MyDrive/YOLO/yolov4-custom_train_best.weights {img_path} -dont-show -ext_outpu
img = cv2.imread("/content/darknet/predictions.jpg", cv2.IMREAD_UNCHANGED)
cv2_imshow(img)
```

Figure – 69 Screenshot of cell-45 from 'yolov4.ipynb'

Figure – 70 Result of cell-45 from 'yolov4.ipynb'

```
# cell-45

os.chdir('/content/darknet')
img_path = "/content/TestIJCNN2013Download/00155.png"
img_initial = cv2.imread(img_path, cv2.IMREAD_UNCHANGED)
cv2_imshow(img_initial)
!./darknet detector test data/image_data.data cfg/yolov4-tiny-custom_test.cfg /content/drive/MyDrive/YOLO/yolov4-tiny-custom_train_best.weights {img_path} -dont-show
img = cv2.imread("/content/darknet/predictions.jpg", cv2.IMREAD_UNCHANGED)
cv2_imshow(img)
```

Figure – 71 Screenshot of cell-45 from 'yolov4tiny.ipynb'


```
mandatory: 99%  (left_x:  564   top_y:  599   width:   31   height:   26)
prohibitory: 100%       (left_x:  853   top_y:  517   width:   29   height:   31)
other: 100%     (left_x: 1091   top_y:  319   width:  115   height:   81)
```

Figure – 72 Result of cell-45 from 'yolov4tiny.ipynb'

Figure – 73 Screenshot of '00269.png'

```
# cell-47

img_path = "/content/TestIJCNN2013Download/00269.png"
img_initial = cv2.imread(img_path, cv2.IMREAD_UNCHANGED)
cv2_imshow(img_initial)
!./darknet detector test data/image_data.data cfg/yolov4-custom_test.cfg /content/drive/MyDrive/YOLO/yolov4-custom_train_best.weights {img_path} -dont-show -ext_outpu
img = cv2.imread("/content/darknet/predictions.jpg", cv2.IMREAD_UNCHANGED)
cv2_imshow(img)
```

Figure – 74 Screenshot of cell-47 from 'yolov4.ipynb'

Figure – 75 Result of cell-47 from 'yolov4.ipynb'

```
# cell-47

img_path = "/content/TestIJCNN2013Download/00269.png"
img_initial = cv2.imread(img_path, cv2.IMREAD_UNCHANGED)
cv2_imshow(img_initial)
!./darknet detector test data/image_data.data cfg/yolov4-tiny-custom_test.cfg /content/drive/MyDrive/YOLO/yolov4-tiny-custom_train_best.weights {img_path} -dont-show
img = cv2.imread("/content/darknet/predictions.jpg", cv2.IMREAD_UNCHANGED)
cv2_imshow(img)
```

Figure – 76 Screenshot of cell-47 from 'yolov4tiny.ipynb'

Figure – 77 Result of cell-47 from 'yolov4tiny.ipynb'

With this, we got an idea regarding the accuracy of our YOLO models but we also have to test our models to get an idea regarding the avg FPS (Frames Per Second) so that we can determine which model processes the video and generates the desired output faster. So for that in cell-50, we are downloading a video file in our colab's local runtime storage. Then in cell-51 of both the colab files, we are testing both the YOLO models on this video file. Surprisingly, we are getting more FPS with our YOLOv4 model. With YOLOv4, we are getting on average 30 FPS as shown in Figure-x and with YOLOv4-tiny we are getting 16 FPS on average as shown in figure-y. Therefore in terms of FPS, the performance of YOLOv4 model is better as compared to the performance of YOLOv4-tiny model. I will upload the original video file along with the video files which we got after testing the YOLO models on it. The video files which we get after executing cell-51 shows continuous detection of road signs.

Figure – 78 Screenshot of cell-50 from 'yolov4.ipynb' and 'yolov4tiny.ipynb'

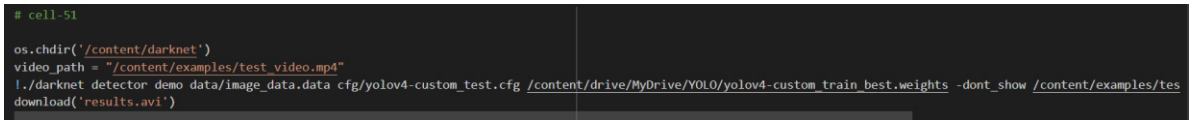


Figure – 79 Screenshot of cell-51 from 'yolov4.ipynb'

Figure – 80 Screenshot of cell-51 from 'yolov4tiny.ipynb'

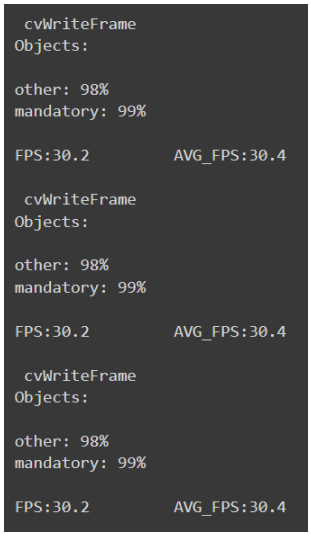


Figure – 81 Sample of the result of cell-51 from 'yolov4.ipynb'

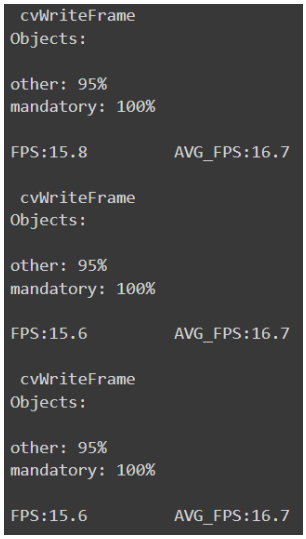


Figure – 82 Sample of the result of cell-51 from 'yolov4tiny.ipynb'

Now I request to please clean your laptop's front camera because in cell-52 of both the colab files, we have written the code that will turn on the front camera of our laptop and we can bring anything like phone or paper having any road sign on it and then we can capture the image by clicking on the capture button. The code written in cell-52 will process this captured image to detect and recognize road signs present in that image and will then show the captured image with the detected road signs along with their class probabilities and bounding box annotations. The code written in cell-52 is a bit longer to be displayed in this report. Figure-83 shows the result of cell-52 for our YOLOv4 model and Figure-84 shows the result of cell-52 for our YOLOv4-tiny model. The results were quite impressive. The image which we captured had 12 prohibitory road signs and both of our models were able to detect and recognize all the 12 road signs correctly with high probabilities and exact locations as shown in Figure-83 and Figure-84. We took the whole code written in cell-52 from [22] and then we made some minute changes to it.

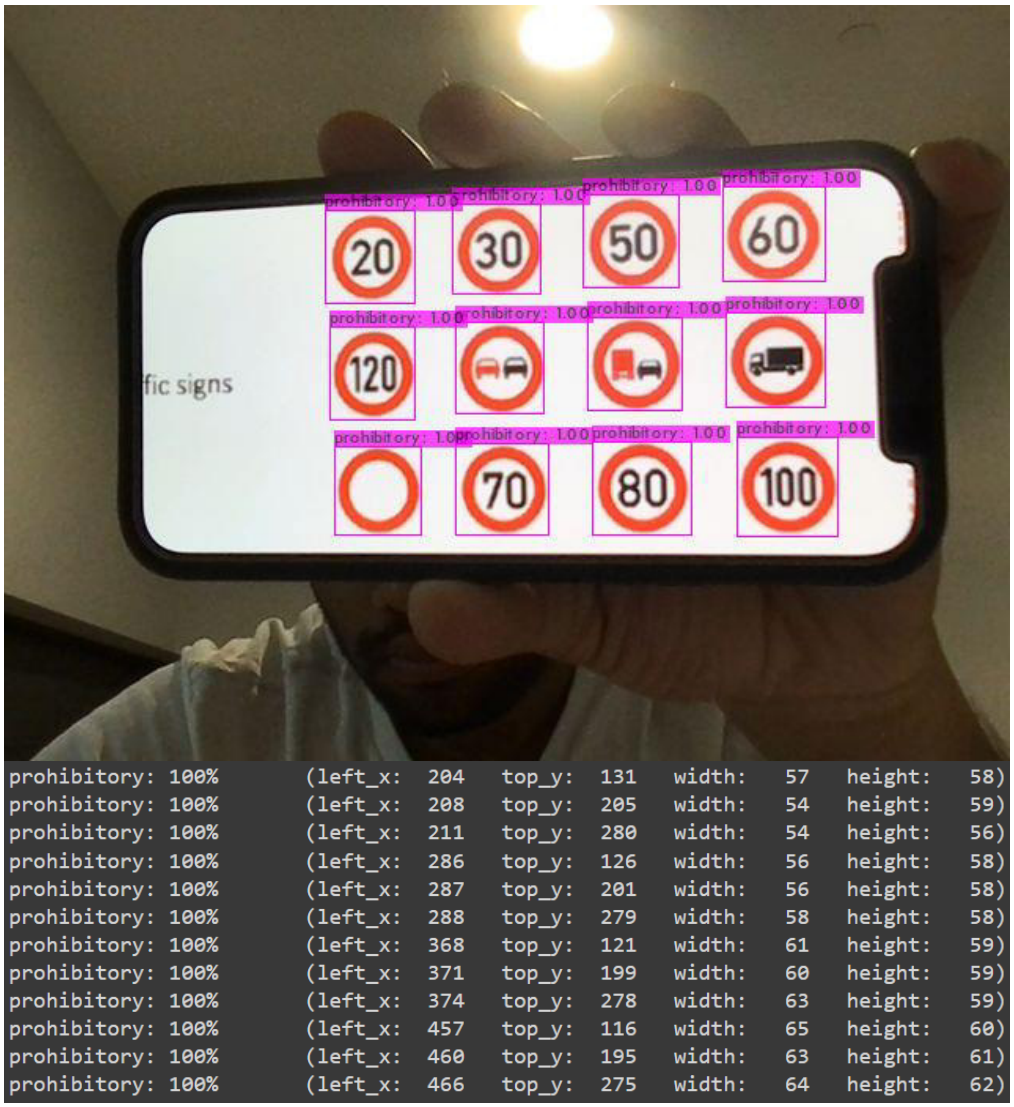
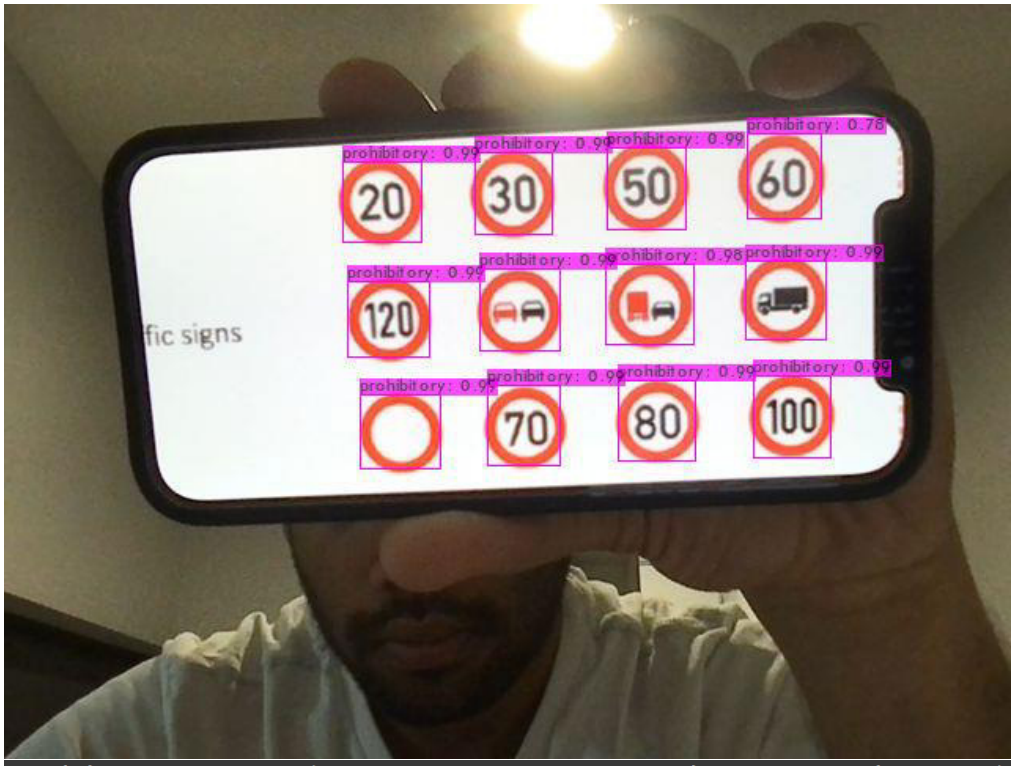


Figure – 83 Result of cell-52 from 'yolov4.ipynb'




```
prohibitory: 99%        (left_x:  215   top_y:  100   width:   50   height:   51)
prohibitory: 99%        (left_x:  219   top_y:  177   width:   52   height:   48)
prohibitory: 99%        (left_x:  226   top_y:  249   width:   52   height:   46)
prohibitory: 99%        (left_x:  299   top_y:   94   width:   49   height:   52)
prohibitory: 99%        (left_x:  303   top_y:  169   width:   51   height:   52)
prohibitory: 99%        (left_x:  307   top_y:  243   width:   47   height:   51)
prohibitory: 98%        (left_x:  383   top_y:  166   width:   56   height:   51)
prohibitory: 99%        (left_x:  383   top_y:   91   width:   50   height:   52)
prohibitory: 99%        (left_x:  391   top_y:  240   width:   49   height:   51)
prohibitory: 99%        (left_x:  471   top_y:  163   width:   51   height:   50)
prohibitory: 78%        (left_x:  472   top_y:   83   width:   47   height:   54)
prohibitory: 99%        (left_x:  477   top_y:  236   width:   48   height:   52)
```

Figure – 84 Result of cell-52 from 'yolov4tiny.ipynb'

Now the code written in cell-53 and in cell-54 of both the colab files will allow us to detect and recognize the road signs in the real-time by keeping the video stream running with the help of laptop's front camera. After executing cell-54, if we bring any road sign in front of the laptop's camera then we will be able to see bounding boxes around the road signs along with the probability scores until we stop the live video stream. Again the code written in cell-53 and in cell-54 is very large to be displayed in this report. The code present in cell-53 has been taken from the darknet.py file which is present inside the darknet folder in our google colab local runtime storage and the code written in cell-54 has been completely taken from [23].

Now initially when I was trying to run the code written in cell-53 and in cell-54 for the first time then I was getting some errors related to the changes which we made in the Makefile in cell-25. Therefore to resolve those errors, I opened the terminal of our google colab's file and in that terminal as shown in the Figure-85, I executed the following command : sed -i 's/LIBSO=0/LIBSO=1/' Makefile

After running the following command, I never again got any problem or any error while running the code written in cell-53 and in cell-54. Therefore, if anyone will be running these colab files in different google account or in some other place and if they got any errors while executing cell-53 then consider doing the same thing as we have done in Figure-85.

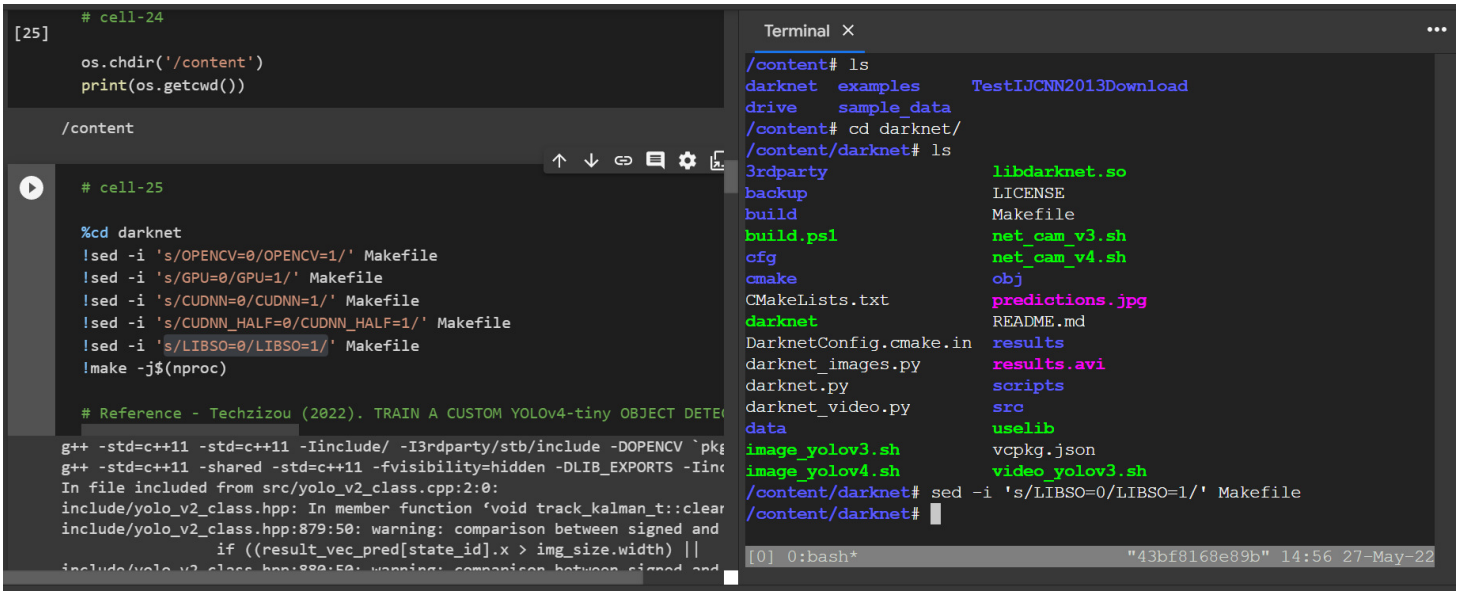


Figure-85 Showing the way to resolve errors if any while executing cell-53

The cells in 'yolov4.ipynb' and in 'yolov4tiny.ipynb' must be executed in the same order in which they are made. The code in the cells of both the colab files will not work if we change the location of the colab files or the dataset or the annotations or any other folder and file whose path is used in the code. Therefore below this paragraph, I have provided the login credentials of the google drive in which we made this prototype.

Email – hartfortandrew@gmail.com
Password – Work@12345

The google drive contains two folders, one of them is 'YOLO' and we are primarily concerned with this folder because it contains 'yolov4.ipynb', 'yolov4tiny.ipynb', GTSDB Dataset, weights which were generated while training, results etc. Therefore in 'YOLO' folder, we have stored our YOLOv4 and YOLOv4-tiny model which is trained on the GTSDB dataset. The other folder is 'YOLO-MTSD' which contains the YOLOv4 prototype which we were trying to train on the Mapillary Traffic Sign Dataset. This 'YOLO-MTSD' folder also contains the MTSD dataset.

4.0 Conclusion

In this project, we have successfully created a prototype using which we are able to detect and recognize the road signs in various formats like images, videos and in the real-time as well. Both of our YOLO models have performed exceptionally well. There is almost no difference in the performance of both the models when it comes to the accuracy as with the best weights of YOLOv4, we are achieving 100% mAP and with the best weights of our YOLOv4-tiny model we are achieving 99.66% mAP but when it comes to FPS then, in that case, our YOLOv4 model is performing better compared to our YOLOv4-tiny model as with YOLOv4, we are getting around 30 FPS and with YOLOv4 tiny, we are getting around 16 FPS. Therefore if asked to choose one model between YOLOv4 and YOLOv4-tiny then I will choose YOLOv4 because the accuracy wise performances of both the models are same but with YOLOv4 we are somehow getting more FPS.

4.1 Future Work

For future work, we can use large and more complex datasets to train our YOLO models. We can combine different datasets of road signs together and train our YOLO model on this combined dataset to expand the reach of our YOLO models. For example, we can combine the German Traffic Sign Dataset with the Belgium Traffic Sin Dataset and while doing so we can work with different editions of YOLOv4 for example YOLOv4-p5, YOLOv4-p6 etc. as these editions have more [convolution] and [yolo] layers and they also use different activation functions. Then apart from the parameters which we changed in our configurations files, we can tweak other parameters for example parameters for Data Augmentation like angle, saturation, exposure, hue etc. and parameters for Optimization like momentum, decay, learning rate etc. to see how these changes affects the performance of the YOLO models. In future, we can also implement the trained YOLO models using Keras and TensorFlow to see what other functionalities do they provide to our YOLO models.

5.0 Reflection

This project has given me some experiences which I can never forget in my life. In the starting, when is saw this project, I was confident that I will be doing this project because I wanted to intensify my knowledge of machine learning which we haven't been taught in our bachelor's degree.

In the beginning, I was not entirely sure that what I actually have to make. I thought that I need to make only a machine learning model which can recognize the class of the traffic sign so I was thinking that I only have to make a classification model and therefore in the starting phase of this project, I made a CNN model using Keras and trained it on GTSRB(German Traffic Sign Recognition Benchmark) Dataset. Though it was not something which I meant to do but this helped me immensely to build a strong base so that I can do what I actually have to.

In the initial weeks of the project, I watched a large number of tutorials on the YouTube to develop a base in machine learning and understood how neural network works, how we can train different neural networks for example CNN in Keras, then what are object detection algorithms and how they work. I understood different types of object detection algorithms present out there for example YOLO and Region Proposal Networks and then I researched them deeply that which object detection algorithm should I choose, which object detection algorithm would be the best to implement looking at the nature of the project.

I also struggled a lot when I was not able to train my prototype on the Mapillary Traffic Sign Dataset because of the lack of computing resources but somehow everything gets sorted. This project made me realize the importance of time management as in the beginning, I was taking this report writing part very lightly but later understood that it's a time-consuming process and should have worked on it parallel to the coding. Because of this project, I gained a considerable amount of knowledge on machine learning and using google colaboratory for training the machine learning models which will help me with my future goals in the field of Business Analytics.

During this dissertation, I was also applying for masters which also took a considerable amount of time from my schedule which was dedicated to this dissertation project but everything was worth it. In the end I would like to say that because of this project I acquired some valuable skills like researching the content and drawing insights from it. Then I also gained documentation skills as it requires some practice to translate your understanding into words so that the reader can actually understand what you want to convey.

6.0 Table of Abbreviations

YOLO	You Only Look Once
YOLOv4	You Only Look Once Version 4
mAP	Mean Average Precision
AP	Average Precission
CNN	Convolution Neural Network
FPS	Frames Per Second
GTSDB	German Traffic Sign Detection Benchmark
R-CNN	Region-Based Convolution Neural Network
MSTD	Mapillary Traffic Sign Dataset
ANN	Artificial Neural Network
ROI	Region of Interest
BoF	Bag of Freebies
BoS	Bag of Specials
IoU	Intersection over Union

7.0 References

[1] Lovell, Lovell, Isern & Farabough, LLP. (2021). *Stop Sign Accidents Statistics | Amarillo Personal Injury Attorneys*. [online] Available at: https://www.lovell-law.net/blog/personal-injury/stop-sign-accidents-statistics/.

[2] Watts Guerra LLP. (2021). *Causes of Major Road Accidents and the Most Common Types*. [online] Available at: https://wattsguerra.com/causes-of-major-road-accidents-and-the-most-common-types/

[3] Wan, H., Gao, L., Su, M., You, Q., Qu, H. and Sun, Q. (2021). A Novel Neural Network Model for Traffic Sign Detection and Recognition under Extreme Conditions. *Journal of Sensors*, [online] 2021, p.e9984787. Available at: https://www.hindawi.com/journals/js/2021/9984787/

[4] benchmark.ini.rub.de. (n.d.). *German Traffic Sign Benchmarks*. [online] Available at: https://benchmark.ini.rub.de/about.html

[5] benchmark.ini.rub.de. (n.d.). *German Traffic Sign Benchmarks*. [online] Available at: https://benchmark.ini.rub.de/gtsdb_dataset.html

[6] www.mapillary.com. (n.d.). *Mapillary*. [online] Available at: https://www.mapillary.com/dataset/trafficsign.

[7] Bochkovskiy, Alexey, Chien-Yao Wang, and Hong-Yuan Mark Liao. "YOLOv4: Optimal Speed and Accuracy of Object Detection." *arXiv preprint arXiv:2004.10934* (2020).

[8] J. Redmon and A. Farhadi, "YOLOv3: An Incremental Improvement", arXiv.org, 2021. Available: https://arxiv.org/abs/1804.02767.

[9] sid.erda.dk. (n.d.). *Public Archive: ff17dc924eba88d5d01a807357d6614c*. [online] Available at: https://sid.erda.dk/public/archives/ff17dc924eba88d5d01a807357d6614c/published-archive.html

[10] www.youtube.com. (n.d.). *Simple explanation of convolutional neural network | Deep Learning Tutorial 23 (Tensorflow & Python)*. [online] Available at: https://www.youtube.com/watch?v=zfiSAzpy9NM&t=493s

[11] S, V.B. (2020). *Using YOLOv3 for detection of traffic signs.* [online] Analytics Vidhya. Available at: https://medium.com/analytics-vidhya/using-yolov3-for-detection-of-traffic-signs-57f30b3c561c

[12] Jun 4, J.S. and Read □ ⍰ 〈 ◊2020 10 M. (2020). *Breaking Down YOLOv4*. [online] Roboflow Blog. Available at: https://blog.roboflow.com/a-thorough-breakdown-of-yolov4/.

[13] Rajput, V. (2022). *YOLO v4 explained in full detail*. [online] AIGuys. Available at: https://medium.com/aiguys/yolo-v4-explained-in-full-detail-5200b77aa825.

[14] RUGERY, P. (2020). *Explaining YoloV4 a one stage detector*. [online] Medium. Available at: https://becominghuman.ai/explaining-yolov4-a-one-stage-detector-cdac0826cbd7.

[15] OpenGenus IQ: Computing Expertise & Legacy. (2022). *YOLOv4 model architecture*. [online] Available at: https://iq.opengenus.org/yolov4-model-architecture/.

[16] techzizou007 (2021). *YOLOv4 VS YOLOv4-tiny*. [online] TECHZIZOU. Available at: https://techzizou.com/yolov4-vs-yolov4-tiny-custom/

[17] Aggarwal, A. (2021). *YOLO Explained*. [online] Analytics Vidhya. Available at: https://medium.com/analytics-vidhya/yolo-explained-5b6f4564f31

[18] Yohanandan, S. (2020). *mAP (mean Average Precision) might confuse you!* [online] Medium. Available at: https://towardsdatascience.com/map-mean-average-precision-might-confuse-you-5956f1bfa9e2

[19] Techzizou (2022). *TRAIN A CUSTOM YOLOv4-tiny OBJECT DETECTOR (Using Google Colab)*. [online] Analytics Vidhya. Available at: https://medium.com/analytics-vidhya/train-a-custom-yolov4-tiny-object-detector-using-google-colab-b58be08c9593

[20] Digital Trends. (2020). *JPEG vs. PNG: Which Compressed Image Format Is Better?* [online] Available at: https://www.digitaltrends.com/computing/jpeg-vs-png/.

[21] colab.research.google.com. (n.d.). *Google Colaboratory*. [online] Available at: https://colab.research.google.com/drive/1_GdoqCJWXsChrOiY8sZMr_zbr_fH-0Fg?usp=sharing#scrollTo=A9mYUoKOWWlR

[22] 262588213843476 (n.d.). *yolov4-tiny_webcam_images.py*. [online] Gist. Available at: https://gist.github.com/techzizou/6b55042ba33edbec3b53c78607fb0bb0#file-yolov4-tiny_webcam_images-py

[23] 262588213843476 (n.d.). *yolov4-tiny_live_webcam.py*. [online] Gist. Available at: https://gist.github.com/techzizou/6433490f17d5b7656976b453df63c406#file-yolov4-tiny_live_webcam-py