

Lane Detection

Using YOLOv3



Performed on:
Google Colaboratory

Made by:
Mohit Kumar (2K18/EE/114)
Navneet Priyadarshi (2K18/EE/123)

Abstract

A model that can automatically learn the features of a lane in different scenarios and accurately detect lanes on the busy streets is proposed. The YOLO (You Only Look Once) is used in the construction of the model. The project was carried out on the Google Colaboratory and the results showed that the model can properly detect the lanes on busy streets.

Introduction

Self-driving vehicles are cars or trucks in which human drivers are never required to take control to safely operate the vehicle. Also known as autonomous or “driverless” cars, they combine sensors and software to control, navigate, and drive the vehicle.

Currently, there are no legally operating, fully-autonomous vehicles in India. There are, however, partially-autonomous vehicles—cars and trucks with varying amounts of self-automation, from conventional cars with brake and lane assistance to highly-independent, self-driving prototypes.

Though still in its infancy, self-driving technology is becoming increasingly common and could radically transform our transportation system (and by extension, our economy and society). Based on automaker and technology company estimates, level 4 self-driving cars could be for sale in the next several years.

Layers of autonomy

Different cars are capable of different levels of self-driving and are often described by researchers on a scale of 0-5.

Level 0: All major systems are controlled by humans

Level 1: Certain systems, such as cruise control or automatic braking, may be controlled by the car, one at a time

Level 2: The car offers at least two simultaneous automated functions, like acceleration and steering, but requires humans for safe operation

Level 3: The car can manage all safety-critical functions under certain conditions, but the driver is expected to take over when alerted

Level 4: The car is fully-autonomous in some driving scenarios, though not all

Level 5: The car is completely capable of self-driving in every situation

Impacts

The costs and benefits of self-driving cars are still largely hypothetical. More information is needed to fully assess how they'll impact drivers, the economy, equity, and environmental and public health.

Safety is an overarching concern. Many thousands of people die in motor vehicle crashes every year all over the world; self-driving vehicles could, hypothetically, reduce that number—software could prove to be less error-prone than humans—but cybersecurity is still a chief concern.

Equity is another major consideration. Self-driving technology could help mobilize individuals who are unable to drive themselves, such as the elderly or disabled. But the widespread adoption of autonomous vehicles could also displace millions of people employed as drivers, negatively impact public transportation funding, and perpetuate the current transportation system's injustices.

Environmental impacts are a serious concern and major uncertainty. Accessible, affordable, and convenient self-driving cars could increase the total number of miles driven each year. If those vehicles are powered by gasoline, then transportation-related climate emissions could skyrocket. If, however, the vehicles are electrified—and paired with a clean electricity grid—then transportation emissions could drop, perhaps significantly.

While ML is a crucial component of the centralized electronic control unit (ECU) in an autonomous car, efforts are being made to integrate ML even further in self-driving cars to shape them into state-of-the-art creations. One of the primary functions of ML algorithms in an autonomous car is continuous monitoring of the surrounding environment and accurately predicting the possible changes to that surrounding. This core task can be further segmented.

The Four Sub-Tasks:

- Object detection
- Object identification/recognition
- Object localization
- Movement prediction

Self-driving cars usually incorporate numerous sensors that help them make sense of their surroundings, including GPS, radar, lidar, sonar, odometry, and inertial measurement units. They also have advanced control systems that can interpret sensory information to identify obstacles and figure out suitable navigation paths.

The ML-based applications that run an autonomous car's infotainment system receive information from the sensor data fusion systems and make predictions accordingly. These algorithms can also integrate the driver's gesture, speech recognition, and language translation in the car's system.

At present, self-driving cars can perform the basic tasks of a human driver, such as controlling, navigating, and driving the vehicle, but of course, there are certain limitations to it as well. However, with further advancement of Machine Learning and improvement of self-driving car algorithms, we have a lot to look forward to from these autonomous cars.

You only look once (YOLO) is a state-of-the-art, real-time object detection system. On a Pascal Titan X, it processes images at 30 FPS and has an mAP of 57.9% on COCO test-dev.

YOLOv3 is extremely fast and accurate. In mAP measured at .5, IOU YOLOv3 is on par with Focal Loss but about 4x faster. Moreover, you can easily trade off between speed and accuracy simply by changing the size of the model, no retraining required!

Prior detection systems repurpose classifiers or localizers to perform detection. They apply the model to an image at multiple locations and scales. High scoring regions of the image are considered detections.

It applies a single neural network to the full image. This network divides the image into regions and predicts bounding boxes and probabilities for each region. These bounding boxes are weighted by the predicted probabilities.

A single convolutional network simultaneously predicts multiple bounding boxes and class probabilities for those boxes. YOLO trains on full images and directly optimizes detection performance. This unified model has several benefits over traditional methods of object detection. First, YOLO is extremely fast. Since it frames detection as a regression problem it doesn't need a complex pipeline. It simply runs our neural network on a new image at test time to predict detections.

Second, YOLO reasons globally about the image when making predictions. Unlike sliding window and region proposal-based techniques, YOLO sees the entire image during training and test time so it implicitly encodes contextual information about classes as well as their appearance. Fast R-CNN, a top detection method, mistakes background patches in an image for objects because it can't see the larger context. YOLO makes less than half the number of background errors compared to Fast R-CNN.

Third, YOLO learns generalizable representations of objects. When trained on natural images and tested on the artwork, YOLO outperforms top detection methods like DPM and R-CNN by a wide margin. Since YOLO is highly generalizable it is less likely to break down when applied to new domains or unexpected inputs.

Approach

The steps (flow of work) in our approach are as follows:

- ❖ Finding a video suitable for the training and testing of the model. The video source can be from anywhere. It should be a dashcam video of a car on the street irrespective of the number of vehicles on the streets. The lanes should be visible from the camera angle.
- ❖ Retrieving of relevant frames from a video. A video is made of the number of the frames playing one after another. While retrieving the frames, irrelevant frames which don't contain any lane can be removed. Every other frame can be skipped because there is a minimal difference in 2-3 consecutive frames.
- ❖ To detect lanes, the thresholding of image using colorspace will be performed to binarize the white and yellow lines because lanes can be of either of these colours.
- ❖ Transforming the frame images to bird view. Parallel lines appear to converge on images from the front-facing camera due to perspective. To keep parallel lines parallel for photogrammetry, a bird's eye view transformation should be applied. This will be done using OpenCV.

- ❖ The annotation of transformed images to get labels. These labels will be used in the algorithm to train the model. A software named 'labellmg' from tzutalin is used to annotate the frames (marking all the boxes by us in every frame used for training). We used around 1000 images for this step.
- ❖ Training of model with the weights we got from the annotation in Google Colaboratory. The tuning of hyperparameters to get better accuracy was done in this step.
- ❖ Marking the lanes on the original frames with the help of a self-made algorithm.
- ❖ The evaluation of the model on a different dataset with images without annotation or labels.

Implementation

Training :

So first of all, mounting google drive in the Colaboratory and Importing of relevant libraries to be used in the code.

```
[ ] # Mounting Drive
from google.colab import drive
drive.mount('/content/gdrive', force_remount=True)
```

Mounted at /content/gdrive

```
[ ] import cv2
from google.colab.patches import cv2_imshow as ci #For using cv2.imshow
import numpy as np
import matplotlib.pyplot as plt
```

Extracting frames from a video

Then a video is captured from the google drive and its frames are stored in an array to use later. For this, a general dashcam video is used as our dataset. The video is of an Indian road having normal traffic.

Here is the first frame of the video.



```
[ ] vid = cv2.VideoCapture('/content/gdrive/My Drive/MLProject/lane2.mp4')

[ ] k = 0
    while k < 2100:
        ret, frame = vid.read()

        cv2.imwrite("frame%d.jpg" % k, frame)

        k = k + 1

    vid.release()

[ ] img = []

    for i in range(0,2100):
        img.append(cv2.imread("frame%d.jpg" % i))

    imgnp = np.array(img)
```

Thresholding and binarization of the frames

Then the image is thresholded for yellow and white colours (because lanes are most of these two colours only). Then the binarization and gradient of the images are done to get the contour of the lanes.

```
[ ] luv1 = [] #Array of frames in LUV colorspace
    luv1_L = [] #Array of L,U,V channels for the individual frames
    for i in range (0,2100):
        luv = cv2.cvtColor(imgnp[i], cv2.COLOR_BGR2LUV)
        luv1.append(luv)
        luv1_L.append(cv2.split(luv))
    luv1 = np.array(luv1)
    ci(luv1[0])
```

```
[ ] lab1 = [] #Array of frames in LAB colorspace
    lab1_B = [] #Array of L,A,B channels for the individual frames
    for i in range (0,2100):
        lab = cv2.cvtColor(imgnp[i], cv2.COLOR_BGR2LAB)
        lab1.append(lab)
        lab1_B.append(cv2.split(lab))
        # ci(hsv)
    lab1 = np.array(lab1)
    ci(lab1[0])
```

```
[ ] #Binary thresholding

    thr1_w = [] #Array of thresholded frames for white lines

    for i in range(0,len(imgu)):
        ret, thrw = cv2.threshold(luv1_L[i][0],150,255,cv2.THRESH_BINARY)
        thr1_w.append(thrw)

    thr1_y = [] #Array of thresholded frames for yellow lines

    for i in range(0,len(imgu)):
        ret, thry = cv2.threshold(lab1_B[i][0],200,255,cv2.THRESH_BINARY)
        thr1_y.append(thry)


    thr1_w = np.array(thr1_w)
    thr1_y = np.array(thr1_y)
```

```
[ ] #Threshold net output

    out = []

    for i in range(0,len(imgu)):
        out.append(np.bitwise_or(thr1_w[i],thr1_y[i]))

    out = np.array(out)
    ci(out[0])
```

```
 #Gradient
grad = []

for i in range(0, len(imgu)):
    g = cv2.cvtColor(imgu[i], cv2.COLOR_BGR2GRAY)

    sobel = cv2.Sobel(g, cv2.CV_64F, 1, 0)

    abs_sobel = np.absolute(sobel)
    scaled_sobel = np.uint8(255 * abs_sobel / np.max(abs_sobel))

    grad.append(np.bitwise_or(out[i], scaled_sobel))

ci(grad[0])
```

Here's the first frame after it goes all the things mentioned above:



Taking bird-view transformation

Now the bird-view of the frame is taken to remove the sky part from the video and focus on the lanes only. And by this transformation, the lanes get quite vertical which will increase the accuracy during marking them in 'labeling'.


```

#Taking bird view transform

img_ht = 350
img_wd = 650

init = np.float32([[0,img_ht], [650,img_ht], [0,0], [img_wd,0]])
fin = np.float32([[210,img_ht], [360, img_ht], [0,0], [img_wd+200, 0]])

#transformation matrices
tr_mat = cv2.getPerspectiveTransform(init, fin) #transform
tr_mat_inv = cv2.getPerspectiveTransform(fin, init) #inverse transform

#Taking out ROI and warping

warpl = []

for i in range(0, len(imgu)):
    img = grad[i]
    img = img[120:(img_ht+100), 0:img_wd] #cropping out ROI

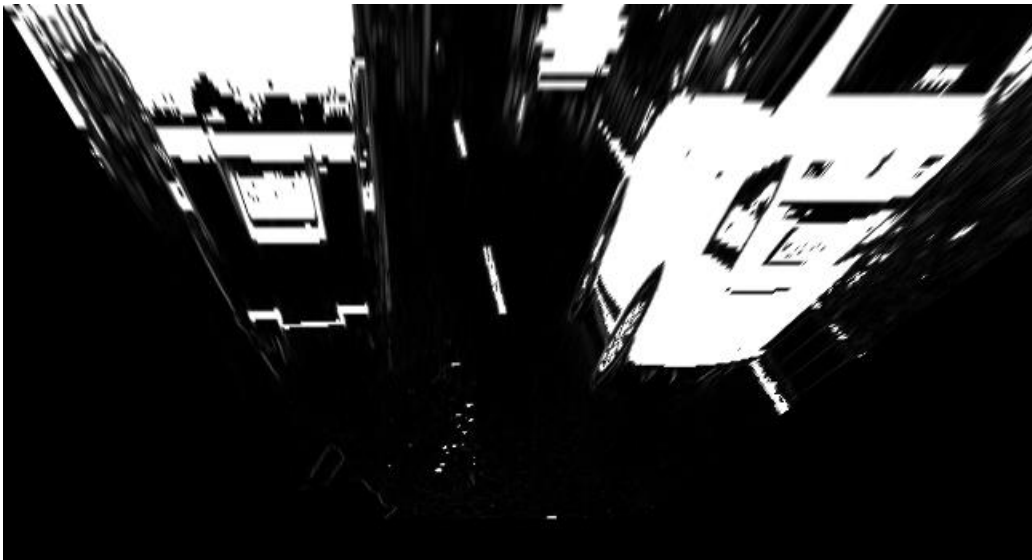
    warp_img = cv2.warpPerspective(img, tr_mat, (img_wd, img_ht)) #warping
    warpl.append(warp_img)

    # cv2.imwrite("/content/gdrive/My Drive/MLProject/transform-images2.0/threshold1-image-%d.jpg" % i, warp_img)

warpl = np.array(warpl)
ci(warpl[0])

```

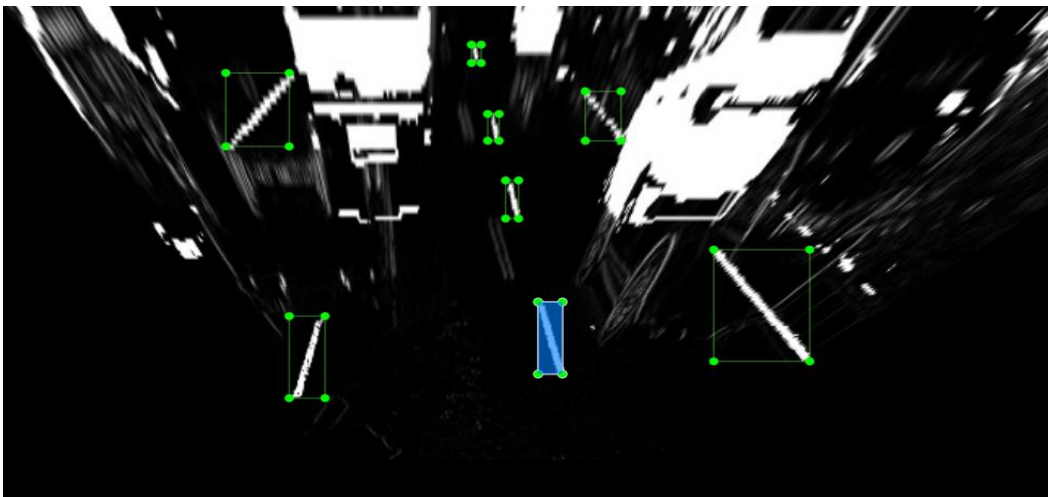
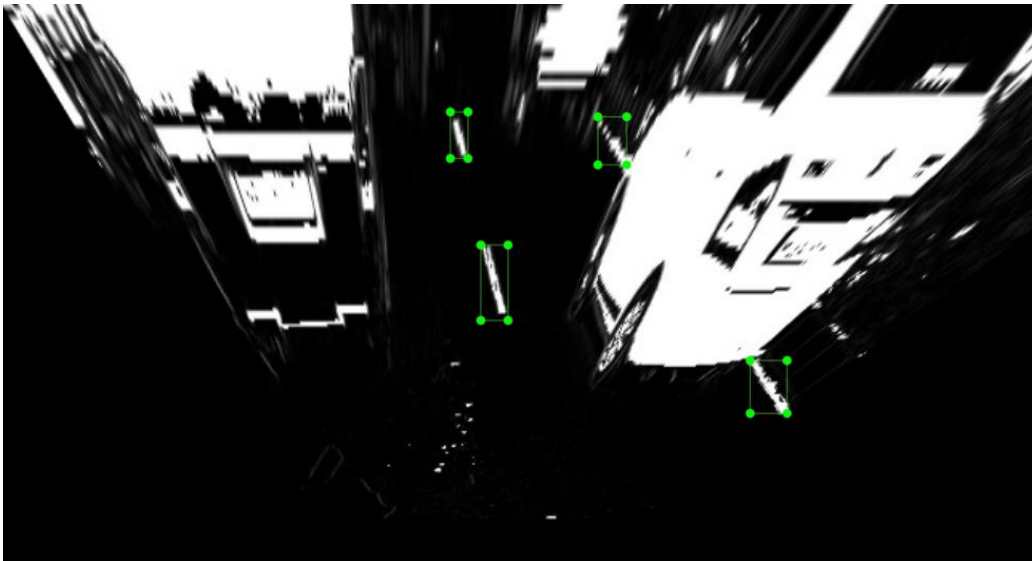
Here's the result of the first frame:



Labeling boxes in every frame manually

We then downloaded the transformed images and used 'tzutalin labeling' for labelling the boxes around the lanes for every image by ourselves. This process will return us the images and the coordinates of every box which will be our training dataset.

Here are some shots of labelling done by us manually:



We then cloned the darknet github repository and saved it to our google drive.

```
# clone darknet repo
import os

os.chdir("/content/gdrive/My Drive/MLProject/")

# !git clone https://github.com/AlexeyAB/darknet

# change makefile to have GPU and OPENCV enabled
import os
os.chdir("/content/gdrive/My Drive/MLProject/darknet/")
!sed -i 's/OPENCV=0/OPENCV=1/' Makefile
!sed -i 's/GPU=0/GPU=1/' Makefile
!sed -i 's/CUDNN=0/CUDNN=1/' Makefile
!sed -i 's/CUDNN_HALF=0/CUDNN_HALF=1/' Makefile
!sed -i 's/LIBSO=0/LIBSO=1/' Makefile

!make
```

After doing this we had to perform a few steps before we could start training our custom model. These steps were as follows –

- We had to create 5 files namely train.txt, test.txt, obj.names, obj.data, yolo-obj.cfg.
- train.txt contains in consecutive lines the paths to each of the images that we are using for training i.e the labelled images.
- test.txt is the same as train.txt except it contains paths to the images we are using for testing
- obj.names contains in consecutive lines the names of all the classes (i.e. the different types of objects we want to detect using YOLO)
- obj.data contains the no. of classes and the paths to train.txt, test.txt, obj.names and backup folder to store weights respectively.
- yolo-obj.cfg contains the contents of yolov3.cfg that was already available in the repository except a few changes were made for custom object detection. The changes were as follows –
 - i. The max_batches had to be set to no. of classes*2000 so in our case it was 2000.
 - ii. step had to be changed to 80% and 90% of the maxbatches i.e. step = 1600, 1800.
 - iii. There were 3 instances of yolo in the original file and 3 of convolutional layer that just preceded yolo so for each of them we had to make the following changes.

In each of the instances of yolo the value of “classes” were changed to the no. of classes in our data.

In the instances of convolutional layer just preceding yolo, the value of filters were changed to (classes + 5)*3

obj.data -

```
1 classes = 1                #No of classes in we're trying to detect
2 training = data/train.txt   #Path to train.txt file
3 valid = data/test.txt      #Path to test.txt file
4 names = data/obj.names     #Path to obj.names file
5 backup = backup/           #Backup folder where weights are to be stored
```

obj.names -

```
obj.names X
1 line
```

After making the changes as mentioned above we then trained our model using the code in snippets below -

```
!chmod 755 /content/gdrive/My\ Drive/MLProject/darknet/darknet
%cd /content/gdrive/My\ Drive/MLProject/darknet

!./darknet detector train data/obj.data cfg/yolo-obj.cfg darknet53.conv.74 -dont_show

[ ] !chmod 755 /content/gdrive/My\ Drive/MLProject/darknet/darknet
%cd /content/gdrive/My\ Drive/MLProject/darknet

!./darknet detector map data/obj.data cfg/yolo-obj.cfg backup/yolo-obj_final.weights -dont_show
```

Testing :

For the testing, we imported the relevant libraries and mounted the google drive.

Separate functions defined for thresholding and transformation

Then we made functions for thresholding and transformation of an image when called. These both functions work like in the training.

```
def threshold(inp):
    LUV = cv2.cvtColor(inp, cv2.COLOR_BGR2Luv)
    L,U,V=cv2.split(LUV)
    ret,imageL= cv2.threshold(L, 150,255,cv2.THRESH_BINARY)
    #cv2_imshow(imageL)

    LAB=cv2.cvtColor(inp, cv2.COLOR_BGR2LAB)
    L,A,B=cv2.split(LAB)
    ret,imageB= cv2.threshold(B,200,255,cv2.THRESH_BINARY)
    #cv2_imshow(imageB)

    gray=cv2.cvtColor(inp,cv2.COLOR_BGR2GRAY)
    sobelx = cv2.Sobel(gray, cv2.CV_64F, 1, 0)
    abs_sobelx= np.absolute(sobelx)
    scaled_sobel = np.uint8(255 * abs_sobelx / np.max(abs_sobelx))

    output = np.zeros(shape = imageL.shape)
    output = np.bitwise_or(imageL,imageB)

    return output
```

```
[ ] #Taking bird view transform
def transform (image):
    img_ht = 350
    img_wd = 650

    init = np.float32([[0,img_ht], [650,img_ht], [0,0], [img_wd,0]])
    fin = np.float32([[210,img_ht], [360, img_ht], [0,0], [img_wd+200, 0]])

    #transformation matrices
    tr_mat = cv2.getPerspectiveTransform(init, fin) #tranform
    tr_mat_inv = np.linalg.inv(tr_mat) #inverse transform

    #Taking out ROI and warping

    # image = image[120:(img_ht+100), 0:img_wd] #Cropping out ROI
    warp_img = cv2.warpPerspective(image, tr_mat, (650, 350), cv2.INTER_LINEAR) #warping

    return warp_img,tr_mat,tr_mat_inv
```

Addition of darknet.py and making some changes in its code

After that, the code of 'darknet.py' is added, which is a file in darknet which is mostly used for Yolo predictions, given the weight and the image.

There is a function called 'performDetect' in 'darknet.py', which we called to return the coordinates of the boxes predicted for lanes on the test transformed image.

Then, the main code for testing is there, which reads the test video from the google drive and a loop is there which runs until the video ends.

```
video= cv2.VideoCapture('/content/gdrive/MyDrive/MLProject/lane2.mp4')
success,vid_frame = video.read()

i = 0
iterations = 10          # change to number of frames you want
while (success and i < iterations):
```

Marking the lanes using the predicted coordinates of boxes

As the loop is for every frame in the video, the following steps will be applied to the whole video frame-by-frame.

- The input frame is resized to a size, which is suitable for our algorithm.
- The resized frame is thresholded and transformed using functions we defined earlier.

```
resize_img = cv2.resize(vid_frame, dsize=(650, 350), interpolation=cv2.INTER_CUBIC)

threshold_img = threshold (resize_img)

transformed_img,tr_matrix,tr_mat_inv = transform (threshold_img)

cv2.imwrite ("/content/gdrive/MyDrive/MLProject/MLimages/frame%d.jpg" %i, transformed_img)
```

- The image is sent to the 'performDetect' function to get the coordinates of the boxes around the lanes in that image.
- For every box, a pointer is traversed all through it and gives us the coordinates of the pixels that are on the lanes (white pixels are of the lanes in all the boxes).
- Using these coordinates of pixels in transformed image and the inverse matrix we got from the transform matrix, we map these pixels onto the non-bird view image using the equation given below. (x,y) represents the mapped pixels on non-bird view image and [a11,a10.....] are the values of the inverse matrix (u,v represents the pixels which are to be mapped).

$$\begin{bmatrix} x' \\ y' \\ w' \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \begin{bmatrix} u \\ v \\ w \end{bmatrix} \text{ where } x = \frac{x'}{w'} \text{ and } y = \frac{y'}{w'}.$$

```
#marking lanes in original frame

height, width = okimg.shape[0], okimg.shape[1]
for k in range(count):
    #count contains the number of boxes in the frame
    for a in range(max(0, int(D['detections'][k][2][1] - (D['detections'][k][2][3]) / 2)), min(height, int(D['detections'][k][2][1] + (D['detections'][k][2][3]) / 2))):
        for b in range(max(0, int(D['detections'][k][2][0] - (D['detections'][k][2][2]) / 2)), min(width, int(D['detections'][k][2][0] + (D['detections'][k][2][2]) / 2))):
            if okimg[a, b] == 255 :
                n = tr_mat_inv.dot(np.array([[b], [a], [1]]));
                p = int(n[0][0] / n[2][0])
                q = int(n[1][0] / n[2][0])
                # target_image[q, p] = (0,0,255)
                cv2.circle(target_image, (p,q), 1, (0,255,64), 2)
```

- Then we saved the marked frames in the google drive.

Here are the results of the predictions of our model:



After writing all the frames in the drive, we made a video of all those frames, and in that video, the lanes are marked.

Conclusion

In this project, a two-stage network is designed to learn the lane features in different scenarios automatically, and a model that can detect a lane in complex scenarios is obtained. In the first stage, the making of database and training is done. And in the second stage, the model was tested with unlabelled images.

The results were good and the accuracy can be improved further with training for even more frames.

References

1. For Video capture and splitting –

<https://www.geeksforgeeks.org/python-program-extract-frames-using-opencv/>

2. For Image thresholding, Gradient, Colorspaces –

https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_imgproc/py_table_of_contents_imgproc/py_table_of_contents_imgproc.html

https://docs.opencv.org/master/d4/d86/group_imgproc_filter.html#gacea54f142e81b6758cb6f375ce7_82c8d

https://docs.opencv.org/master/d7/d1b/group_imgproc_misc.html#ggaa9e58d2860d4afa658ef70a9b1115576a147222a96556ebc1d948b372bcd7ac59

3. For annotation of frames and YOLO –

<https://bleedai.com/training-a-custom-object-detector-with-tensorflow-and-using-it-with-opencv-dnn-module/>

<https://github.com/tzutalin/labelImg>

<https://github.com/AlexeyAB/darknet>

https://colab.research.google.com/drive/1ITGZsfMaGUpBG4inDIQwIJVW476ibXk_#scrollTo=9x9BFQOfNowN

4. For training -

<https://github.com/AlexeyAB/darknet>