प्रगत संगणन विकास केंद्र, नोएडा
Centre for Development of Advanced Computing, Noida

A PROJECT REPORT

ON

Computer Vision Based Lane

Submitted In Partial Fulfilment of the Requirement for the Award of

Post Graduate Diploma in Artificial Intelligence (PG-DAI)

Under the Guidance of

MISS. SARUTI GUPTA

(Project Guide)

Submitted By

PUSHPENDER **PRN: 220320528022**

SAIF ALI **PRN: 220320528007**

NISAR SHAIKH **PRN: 220320528029**

ANIL KUMAR **PRN: 220320528054**

CONTENTS

INDEX	TITLE	PAGE NUMBER
I	Certificate	**2**
II	Acknowledgement	**3**
III	Abstract	**4**
IV	Introduction to the problem statement and the possible solution	**5**
V	Data Preprocessing	**6**
VI	Coding	**7-17**
VII	Results	**18**
VIII	Conclusion & Future Scope	**19-20**
IX	Reference & Bibliography	**21**

CDAC, B-30, Institutional Area, Sector-62

Noida (Uttar Pradesh)-201307

CERTIFICATE

CDAC, NOIDA

This is to certify that Report entitled **"A Computer Vision Based Lane Line Detection"** which is submitted by Pushpender, Saif Ali, Nisar Shaikh and Anil Kumar in partial fulfilment of the requirement for the award of "**Post Graduate Diploma in Artificial Intelligence"** (PG-DAI) to **CDAC, Noida** is a record of the candidates own work carried out by them under my supervision.

The documentation embodies results of original work, and studies are carried out by the student themselves and the contents of the report do not from the basis for the award of any other degree to the candidate or to anybody else from this or any other University/Institution.

SARUTI GUPTA MAM

(Project Guide)

ACKNOWLEDGEMENT

We would like to express our best sense of gratitude & endeavour with respect to **Miss. Saruti Gupta (Project Guide)** CDAC, Noida for suggesting the problems scholarly guidance and expert supervision during the course of this project. Special thanks to **Mr. Ravi Payal (Program Coordinator).**

We are very thankful to **Saruti Gupta Mam(Project Guide)** the project guide for constant simulated discussion, encouraging new ideas about this project.

PUSHPENDER	**SAIF ALI**	**NISAR SHAIKH**	**ANIL KUMAR**
PRN: 220320528022	**PRN: 220320528007**	**PRN: 220320528029**	**PRN: 220320528054**

ABSTRACT

Abstract: I Many technical improvements have recently been made in the field of road safety, as accidents have been increasing at an alarming rate, and one of the major causes of such accidents is a driver's lack of attention. To lower the incidence of accidents and keep safe, technological breakthroughs should be made. One method is to use Lane Detection Systems, which function by recognizing lane borders on the road and alerting the driver if he Switches to an incorrect lane marking. A lane detection system is an important part of many technologically advanced transportation systems. Although it is a difficult goal to fulfil because to the varying road conditions that a person encounters, particularly while driving at night or in daytime. A camera positioned on the front of the car catches the view of the road and detects lane boundaries. The method utilized in this research divides the video image into a series of sub-images and generates image-features for each of them, which are then used to recognize the lanes on the road. Several methods for detecting lane markings on the road have been presented.

Traffic safety is becoming increasingly crucial as urban traffic grows. People exiting lanes without respecting the laws cause the majority of accidents on the avenues. The majority of these are the outcome of the driver's Interrupted and sluggish behaviour. Lane discipline is essential for both drivers and pedestrians on the road. Computer vision is a form of technology that enables automobiles to comprehend their environment. It's an artificial intelligence branch that helps software to understand picture and video input. The system's goal is to find the lane markings. Its goal is to provide a safer environment and better traffic

conditions. The functionality of the proposed system can range from displaying road line positions to the bot on any outside display to more advanced applications like recognizing lane switching in the near future to reduce concussions caused on roadways.

Keywords: Lane Detection, Artificial Intelligence, Computer Vision, Traffic Safety

INTRODUCTION TO THE PROBLEM STATEMENT AND THE POSSIBLE SOLUTION

The problem of Road Lane Detection and signal detection is to find out the lane and traffic signs automatically for self-driving cars. It is all due to the advancement in computer vision and deep learning that it become possible to detect road track from video's frames and to detect traffic signs during the process of the self-driving. The proposed methods can be used for object detection, object position detection (left, front, or right), steering suggestion, and road lane's guidance. But there are various challenges involved like variability in vehicle shape, safety of the car, over lighting, sharp-turned roads, and collision warning, and different road environments. The detection of an Object, identification of an Object in real time is a major concern. A prominent example of a safety failure is the 2016 Tesla auto-pilot accident, where the sensors of the vehicle were blended by the sun and the system failed to recognize the truck coming from the right, leading to the crash.

The task that we wish to perform is that of real-time lane detection in a video. There are multiple ways we can perform lane detection. We can use the learning-based approaches, such as training a deep learning model on an annotated video dataset, or use a pre-trained model. So, before solving the lane detection problem, we have to find a way to ignore the unwanted objects from the driving scene. One thing we can do right away is to narrow down the area of interest. Instead of working with the entire frame, we can work with only a part of the frame.

Data Pre-processing

The image files were collected from Kaggle. The dataset is divided into two parts training and validation data.

For the pre-trained model, ImageDataGenerator is used in which the image is rescaled, its height and width are shifted, creates a mirror image, zoomed images, etc. The process works with both training images. In validation images, the rescaling of images occurred.

In train generator, make one single image into many images and converts the image's color mode into grayscale, the batch size divides images into batches and stores them in categorical mode, and in the last shuffle the order of images that are being yielded.

For testing or prediction, a video of a road lanes is taken whose frontal side is fullyvisible and make a "data" directory in which we extract the images from the videos store them in the "data" directory if the directory already exists then it stores in an already existing directory and stores the extracted images in the "data" if an error comes when there is an issue.

Images are written frame by frame, after that, it will capture images and store them in .jpg image format, it will create the images and store them in form of consecutive numerical names.

It will generate images till the video ends, after that it will stop or break, and the cam will release and it will destroy all the images which are still working in the backend.

Data Pre-processing

The image files were collected from Kaggle. The dataset is divided into two parts training and validation data.

For the pre-trained model, ImageDataGenerator is used in which the image is rescaled, its height and width are shifted, creates a mirror image, zoomed images, etc. The process works with both training images. In validation images, the rescaling of images occurred.

In train generator, make one single image into many images and converts the image's color mode into grayscale, the batch size divides images into batches and stores them in categorical mode, and in the last shuffle the order of images that are being yielded.

For testing or prediction, a video of a person is taken whose frontal face is fully visible and make a "data" directory in which we extract the images from the videos store them in the "data" directory if the directory already exists then it stores in an already existing directory and stores the extracted images in the "data" if an error comes when there is an issue.

Images are written frame by frame, after that, it will capture images and store them in .jpg image format, it will create the images and store them in form of consecutive numerical names.

It will generate images till the video ends, after that it will stop or break, and the cam will release and it will destroy all the images which are still working in the backend.

CODING

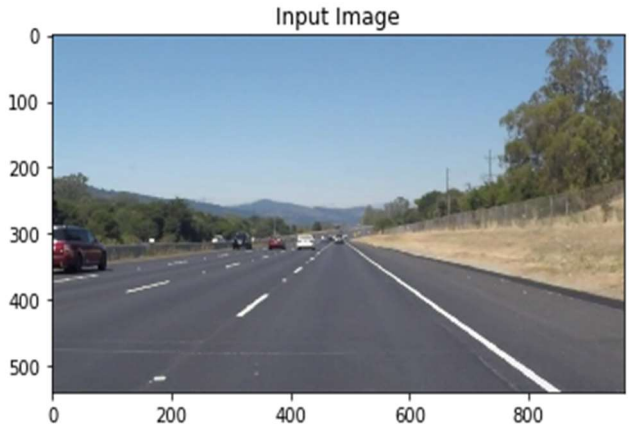
```
pip install opencv-python

import numpy as np # linear algebra

import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)

import os


import matplotlib.pyplot as plt

import matplotlib.image as mpimg

import numpy as np

import cv2

%matplotlib inline



!git clone https://github.com/Pushpsorout/Lane_Line_Detection



from distutils.dir_util import copy_tree

import shutil

copy_tree("./Lane_Line_Detection/test_images", "./test_images")

copy_tree("./Lane_Line_Detection/test_videos", "./test_videos")

shutil.rmtree('./Lane_Line_Detection', ignore_errors=False, onerror=None)
```

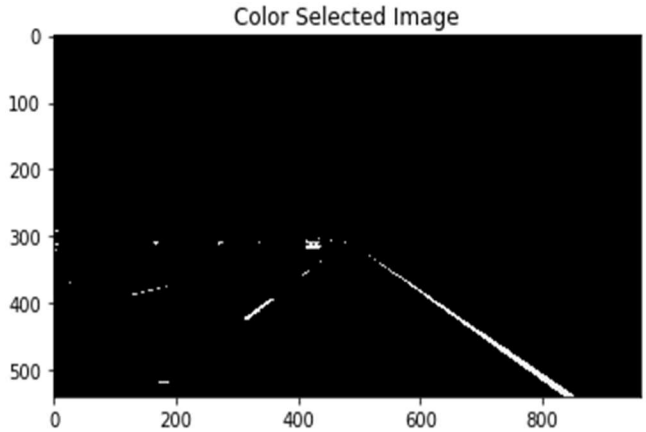
1. Color Selection

First let us select some colors. For Instance: Lane Lines are usually White in color and we know the RGB value of White is (255,255,255). Here we will define a color threshold in the variables red_threshold, green_threshold, and blue_threshold and populate rgb_threshold with these values. This vector contains the minimum values for red, green, and blue (R,G,B) that I will allow in my selection.

```
import matplotlib.pyplot as plt

import matplotlib.image as mpimg

import numpy as np

image = mpimg.imread('test_images/solidWhiteRight.jpg')


ysize = image.shape[0]

xsize = image.shape[1]

color_select = np.copy(image)


# Define color selection criteria

###### MODIFY THESE VARIABLES TO MAKE YOUR COLOR SELECTION

red_threshold = 200

green_threshold = 200

blue_threshold = 200

######


rgb_threshold = [red_threshold, green_threshold, blue_threshold]


thresholds = (image[:,:,0] < rgb_threshold[0]) \

        | (image[:,:,1] < rgb_threshold[1]) \
```

```
        | (image[:,:,2] < rgb_threshold[2])

color_select[thresholds] = [0,0,0]


plt.imshow(image)

plt.title("Input Image")

plt.show()

plt.imshow(color_select)

plt.title("Color Selected Image")

plt.show()
```





In the above output we can clearly see the lane lines

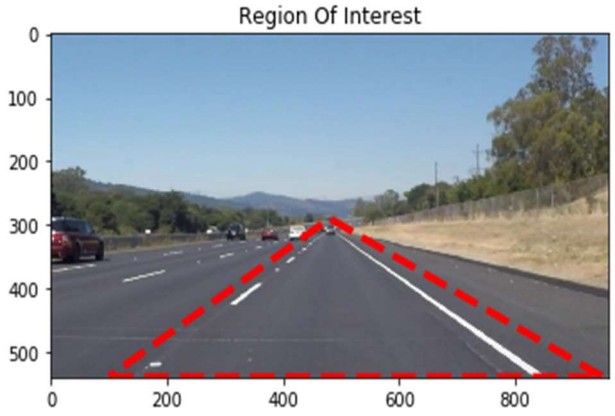
2. Region Masking

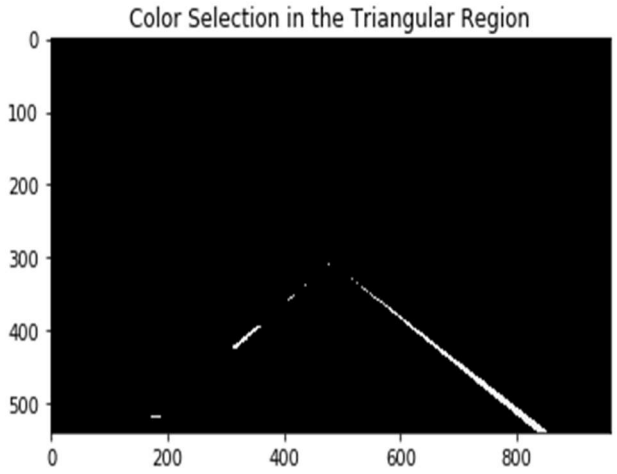
I'll assume that the front facing camera that took the image is mounted in a fixed position on the car, such that the lane lines will always appear in the same general region of the image. Next, I'll take advantage of this by adding a criterion to only consider pixels for color selection in the region where we expect to find the lane lines.

```
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
import numpy as np

image = mpimg.imread('test_images/solidWhiteRight.jpg')

ysize = image.shape[0]
xsize = image.shape[1]
color_select = np.copy(image)
line_image = np.copy(image)

# MODIFY THESE VARIABLES TO MAKE YOUR COLOR SELECTION
red_threshold = 200
green_threshold = 200
blue_threshold = 200

rgb_threshold = [red_threshold, green_threshold, blue_threshold]

left_bottom = [100, 539]
right_bottom = [950, 539]
```

```
apex = [480, 290]

fit_left = np.polyfit((left_bottom[0], apex[0]), (left_bottom[1], apex[1]), 1)
fit_right = np.polyfit((right_bottom[0], apex[0]), (right_bottom[1], apex[1]), 1)
fit_bottom = np.polyfit((left_bottom[0], right_bottom[0]), (left_bottom[1], right_bottom[1]), 1)

color_thresholds = (image[:,:,0] < rgb_threshold[0]) | \
                   (image[:,:,1] < rgb_threshold[1]) | \
                   (image[:,:,2] < rgb_threshold[2])

XX, YY = np.meshgrid(np.arange(0, xsize), np.arange(0, ysize))
region_thresholds = (YY > (XX*fit_left[0] + fit_left[1])) & \
                    (YY > (XX*fit_right[0] + fit_right[1])) & \
                    (YY < (XX*fit_bottom[0] + fit_bottom[1]))

color_select[color_thresholds | ~region_thresholds] = [0, 0, 0]
line_image[~color_thresholds & region_thresholds] = [9, 255, 0]

plt.imshow(image)
x = [left_bottom[0], right_bottom[0], apex[0], left_bottom[0]]
y = [left_bottom[1], right_bottom[1], apex[1], left_bottom[1]]
plt.plot(x, y, 'r--', lw=4)
plt.title("Region Of Interest")
plt.show()
plt.imshow(color_select)
```

```
plt.title("Color Selection in the Triangular Region")

plt.show()

plt.imshow(line_image)

plt.title("Region Masked Image [Lane Lines in Green]")

plt.show()
```

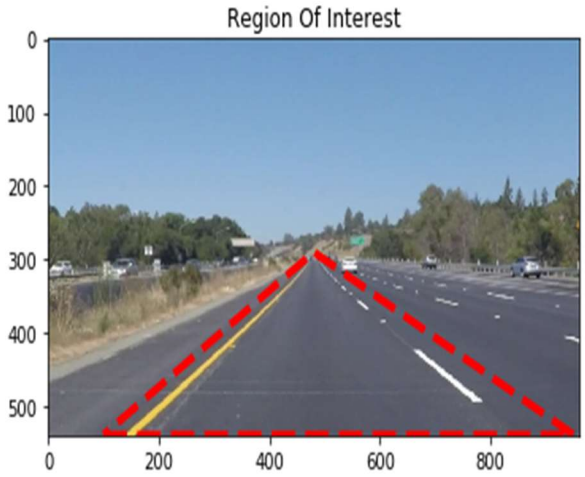


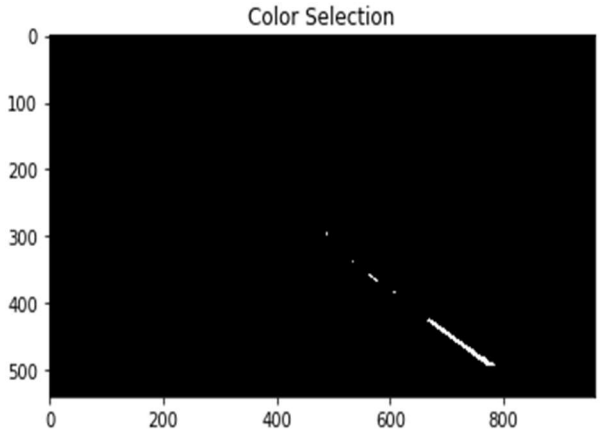


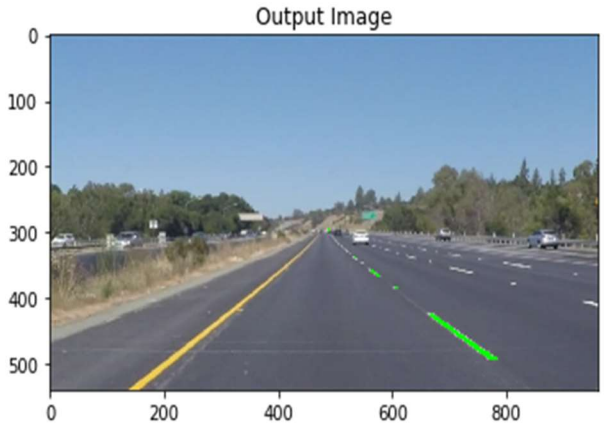
Check the below test.

```
import matplotlib.pyplot as plt

import matplotlib.image as mpimg

import numpy as np


image = mpimg.imread('test_images/solidYellowLeft.jpg')


ysize = image.shape[0]

xsize = image.shape[1]

color_select = np.copy(image)

line_image = np.copy(image)


# MODIFY THESE VARIABLES TO MAKE YOUR COLOR SELECTION

red_threshold = 200

green_threshold = 200

blue_threshold = 200


rgb_threshold = [red_threshold, green_threshold, blue_threshold]


left_bottom = [100, 539]

right_bottom = [950, 539]

apex = [480, 290]


fit_left = np.polyfit((left_bottom[0], apex[0]), (left_bottom[1], apex[1]), 1)

fit_right = np.polyfit((right_bottom[0], apex[0]), (right_bottom[1], apex[1]), 1)
```

```
fit_bottom = np.polyfit((left_bottom[0], right_bottom[0]), (left_bottom[1], right_bottom[1]), 1)


color_thresholds = (image[:,:,0] < rgb_threshold[0]) | \
                   (image[:,:,1] < rgb_threshold[1]) | \
                   (image[:,:,2] < rgb_threshold[2])


XX, YY = np.meshgrid(np.arange(0, xsize), np.arange(0, ysize))
region_thresholds = (YY > (XX*fit_left[0] + fit_left[1])) & \
                    (YY > (XX*fit_right[0] + fit_right[1])) & \
                    (YY < (XX*fit_bottom[0] + fit_bottom[1]))


color_select[color_thresholds | ~region_thresholds] = [0, 0, 0]
line_image[~color_thresholds & region_thresholds] = [9, 255, 0]


x = [left_bottom[0], right_bottom[0], apex[0], left_bottom[0]]
y = [left_bottom[1], right_bottom[1], apex[1], left_bottom[1]]
plt.plot(x, y, 'r--', lw=4)
plt.title("Region Of Interest")
plt.show()
plt.imshow(color_select)
plt.title("Color Selection")
plt.show()
plt.imshow(line_image)
plt.title("Output Image")
plt.show()
```


Region Of Interest
0
100
200
300
400
500
0
200
400
600
800

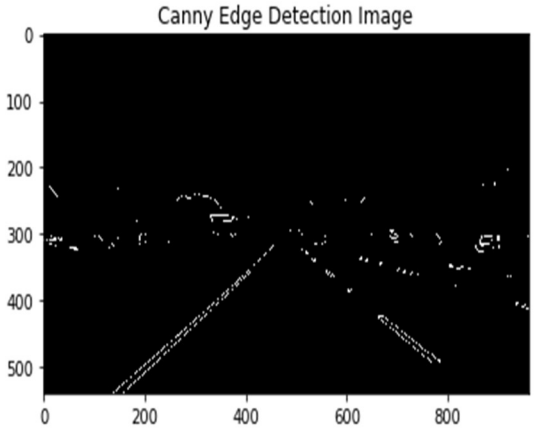

Color Selection
0
100
200
300
400
500
0
200
400
600
800


Output Image
0
100
200
300
400
500
0
200
400
600
800

3. Canny Edge Detection

Now we are applying Canny to the gray-scaled image and our output will be another image called edges. low_threshold and high_threshold are your thresholds for edge detection.

```
import matplotlib.pyplot as plt

import matplotlib.image as mpimg

import numpy as np

import cv2

image = mpimg.imread('test_images/solidYellowLeft.jpg')

gray = cv2.cvtColor(image,cv2.COLOR_RGB2GRAY)


kernel_size = 5 # Must be an odd number (3, 5, 7...)

blur_gray = cv2.GaussianBlur(gray,(kernel_size, kernel_size),0)


low_threshold = 180

high_threshold = 240

edges = cv2.Canny(blur_gray, low_threshold, high_threshold)


plt.imshow(edges, cmap='Greys_r')

plt.title("Canny Edge Detection Image")

plt.show()
```



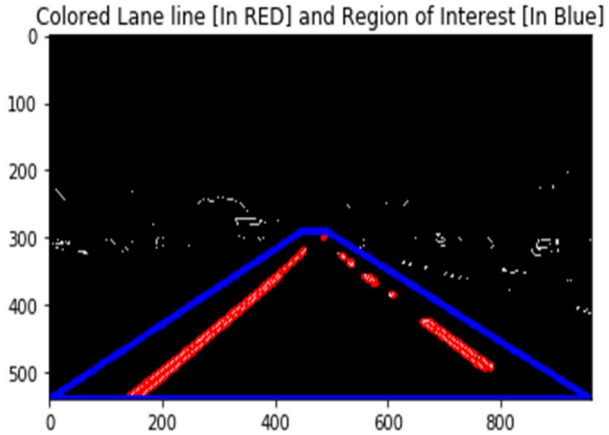
4. Hough Transform and detecting Lane Lines

In image space, a line is plotted as x vs. y, but in 1962, Paul Hough devised a method for representing lines in parameter space, which we will call “Hough space” in his honor.

```
image = mpimg.imread('test_images/solidYellowLeft.jpg')

gray = cv2.cvtColor(image,cv2.COLOR_RGB2GRAY)


kernel_size = 5

blur_gray = cv2.GaussianBlur(gray,(kernel_size, kernel_size),0)


low_threshold = 180

high_threshold = 240

edges = cv2.Canny(blur_gray, low_threshold, high_threshold)


mask = np.zeros_like(edges)

ignore_mask_color = 255
```

```
    imshape = image.shape
    vertices = np.array([[(0,imshape[0]),(450, 290), (490, 290), (imshape[1],imshape[0])]],
dtype=np.int32)
    cv2.fillPoly(mask, vertices, ignore_mask_color)
    masked_edges = cv2.bitwise_and(edges, mask)

    rho = 1
    theta = np.pi/180
    threshold = 2
    min_line_length = 4
    max_line_gap = 5
    line_image = np.copy(image)*0

    lines = cv2.HoughLinesP(masked_edges, rho, theta, threshold, np.array([]),
                            min_line_length, max_line_gap)

    for line in lines:
       for x1,y1,x2,y2 in line:
          cv2.line(line_image,(x1,y1),(x2,y2),(255,0,0),10)

    color_edges = np.dstack((edges, edges, edges))

    lines_edges = cv2.addWeighted(color_edges, 0.8, line_image, 1, 0)
    lines_edges = cv2.polylines(lines_edges,vertices, True, (0,0,255), 10)
```

```
plt.imshow(image)

plt.title("Input Image")

plt.show()

plt.imshow(lines_edges)

plt.title("Colored Lane line [In RED] and Region of Interest [In Blue]")


plt.show()
```



5. Let's Make a Lane Detection Pipeline

1. Gray Scale
2. Gaussian Smoothing
3. Canny Edge Detection
4. Region Masking
5. Hough Transform
6. Draw Lines [Mark Lane Lines with different Color]

```
import math


def grayscale(img):
    return cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)


def canny(img, low_threshold, high_threshold):
    return cv2.Canny(img, low_threshold, high_threshold)


def gaussian_blur(img, kernel_size):
    return cv2.GaussianBlur(img, (kernel_size, kernel_size), 0)


def region_of_interest(img, vertices):
    mask = np.zeros_like(img)

    if len(img.shape) > 2:
        channel_count = img.shape[2]  # i.e. 3 or 4 depending on your image
        ignore_mask_color = (255,) * channel_count
```

```
    else:
        ignore_mask_color = 255

    cv2.fillPoly(mask, vertices, ignore_mask_color)

    masked_image = cv2.bitwise_and(img, mask)
    return masked_image


def draw_lines(img, lines, color=[255, 0, 0], thickness=10):
    for line in lines:
        for x1,y1,x2,y2 in line:
            cv2.line(img, (x1, y1), (x2, y2), color, thickness)

def slope_lines(image,lines):

    img = image.copy()
    poly_vertices = []
    order = [0,1,3,2]

    left_lines = [] # Like /
    right_lines = [] # Like \
    for line in lines:
        for x1,y1,x2,y2 in line:
```

```
        if x1 == x2:
            pass #Vertical Lines
        else:
            m = (y2 - y1) / (x2 - x1)
            c = y1 - m * x1

            if m < 0:
                left_lines.append((m,c))
            elif m >= 0:
                right_lines.append((m,c))

left_line = np.mean(left_lines, axis=0)
right_line = np.mean(right_lines, axis=0)
for slope, intercept in [left_line, right_line]:
    rows, cols = image.shape[:2]
    y1= int(rows) #image.shape[0]
    y2= int(rows*0.6) #int(0.6*y1)
    x1=int((y1-intercept)/slope)
    x2=int((y2-intercept)/slope)
    poly_vertices.append((x1, y1))
    poly_vertices.append((x2, y2))
    draw_lines(img, np.array([[[x1,y1,x2,y2]]]))

poly_vertices = [poly_vertices[i] for i in order]
cv2.fillPoly(img, pts = np.array([poly_vertices],'int32'), color = (0,255,0))
```

```
    return cv2.addWeighted(image,0.7,img,0.4,0.)


def hough_lines(img, rho, theta, threshold, min_line_len, max_line_gap):
    lines = cv2.HoughLinesP(img, rho, theta, threshold, np.array([]), minLineLength=min_line_len, maxLineGap=max_line_gap)
    line_img = np.zeros((img.shape[0], img.shape[1], 3), dtype=np.uint8)
    line_img = slope_lines(line_img,lines)
    return line_img


def weighted_img(img, initial_img, α=0.1, β=1., γ=0.):
    lines_edges = cv2.addWeighted(initial_img, α, img, β, γ)
    return lines_edges
def get_vertices(image):
    rows, cols = image.shape[:2]
    bottom_left  = [cols*0.15, rows]
    top_left     = [cols*0.45, rows*0.6]
    bottom_right = [cols*0.95, rows]
    top_right    = [cols*0.55, rows*0.6]

    ver = np.array([[bottom_left, top_left, top_right, bottom_right]], dtype=np.int32)
    return ver


def lane_finding_pipeline(image):
```

```
    gray_img = grayscale(image)
    smoothed_img = gaussian_blur(img = gray_img, kernel_size = 5)
    canny_img = canny(img = smoothed_img, low_threshold = 180, high_threshold = 240)
    masked_img = region_of_interest(img = canny_img, vertices = get_vertices(image))
    houghed_lines = hough_lines(img = masked_img, rho = 1, theta = np.pi/180, threshold = 20, min_line_len = 20, max_line_gap = 180)
        output = weighted_img(img = houghed_lines, initial_img = image, α=0.8, β=1., γ=0.)

    return output
```

- **Test our Algorithm Pipeline with different Images**

```
    for image_path in list(os.listdir('./test_images')):
    fig = plt.figure(figsize=(20, 10))
    image = mpimg.imread(f'./test_images/{image_path}')
    ax = fig.add_subplot(1, 2, 1,xticks=[], yticks=[])
    plt.imshow(image)
    ax.set_title("Input Image")
    ax = fig.add_subplot(1, 2, 2,xticks=[], yticks=[])
    plt.imshow(lane_finding_pipeline(image))
    ax.set_title("Output Image [Lane Line Detected]")
    plt.show()
```

Input Image

Output Image [Lane Line Detected]

Input Image

Output Image [Lane Line Detected]

Input Image

Output Image [Lane Line Detected]

Input Image

Output Image [Lane Line Detected]

Input Image

Output Image [Lane Line Detected]

Input Image

Output Image [Lane Line Detected]

- **Try with Video Stream**

```
!pip install moviepy
!pip3 install imageio==2.4.1
!pip install --upgrade imageio-ffmpeg
!pip install ffmpeg-python
 conda install ffmpeg -c conda-forge
```

```
from moviepy.editor import VideoFileClip
from IPython.display import HTML
```

Test with Video Clip 1 [Solid White Lane Lines]

```
white_output = './solidWhiteRight.mp4'
clip1 = VideoFileClip("test_videos/solidWhiteRight.mp4")
white_clip = clip1.fl_image(lane_finding_pipeline) #NOTE: this function expects color images!!
%time white_clip.write_videofile(white_output, audio=False)
```

Output Video

```
HTML("""
<video width="960" height="500" controls>
  <source src="{0}">
</video>
""".format(white_output))
```

Test with Video Clip 2 [With Yellow Lane Lines]

```
yellow_output = './solidYellowLeft.mp4'

clip2 = VideoFileClip('test_videos/solidYellowLeft.mp4')

yellow_clip = clip2.fl_image(lane_finding_pipeline)

%time yellow_clip.write_videofile(yellow_output, audio=False)
```

Output

```
HTML("""

<video width="960" height="500" controls>

  <source src="{0}">

</video>

""".format(yellow_output))
```

RESULTS

1. White Line Detection:

   Screenshot of output Video

2. Yellow Line Detection:

   Screenshot of output Video

CONCLUSION AND FUTURE SCOPE

Because we use modular implementation, updating algorithms is simple, and work on the model can be continued in the future. We insert the model's pickle file into the relevant locations, which can then be simply transferred to goods. As a result, compiling the full big code might be avoided easily. We can also enhance the idea by creating a new future in which the road can be identified in the dark, or at night. In daylight, the color recognition and selection process are highly effective. Adding shadows will make things a little noisier, but it won't be as rigorous as driving at night or in low light (e.g., heavy fog). And this research can only recognize lanes on bitumen roads, not on the loamy soil roads that are ubiquitous in Indian villages. As a result, this project can be improved to detect and avoid accidents on loamy soil roads found in communities.

The more the number of training samples, the more accurate will be the classifier prediction. The pre-trained model is created from the training and validation data. The videos captured are of more than thousand frames, out of which each frames were considered here for prediction purpose. And by calculating the sad, happy, and total emotions in frames and predict the depression level. This process can be done for the entire video, by finding out the key frames of the video, by using a key frame extraction technique in the future work. However, for more accurate depression detection, the history of the person should also to be taken into consideration. Therefore, in the future work, more videos of the same person, taken at different time duration can be considered. This may help to analyze and compare the past and the present mental state of the person and provide more information to the process of depression level identification.

REFERENCES & BIBLIOGRAPHY

- Shopa, P., N. Sumetha and P.S.K Pathra. “Traffic sign detection and recognition using OpenCV", International Conference on Information Communication and Embedded Systems (ICICES2014), 2014

- “Segnet: A deep convolutional encoder-decoder architecture for robust semantic pixelwise labelling,” arXiv preprint arXiv:1505.07293, 2015.

- J. Long, E. Shelhamer, and T. Darrell, “LANE DETECTION TECHNIQUES” – A Review.” in Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2015, pp. 3431–3440.

- M.Cordts,M. Omran,S.Ramos, T.Rehfeld,M. Enzweiler, R.Benenson,U. Franke,S.Roth, and B.Schiele, “The cityscapes dataset for semantic urban scene understanding,” in Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2016.

Articles:

- https://www.ncbi.nlm.nih.gov/pmc/articles/PMC6679325/
- https://iopscience.iop.org/article/10.1088/1755-1315/440/3/032126/pdf
- https://ijirt.org/Article?manuscript=151905
- https://www.extrica.com/article/22023