Lane Detection

Using edge detection and Machine Learning

Kabilan Manogaran
Ontario Tech University
Oshawa, Ontario
kabilan.manogaran@ontariotechu.net

Musabbir Ahmed Baki
Ontario Tech University
Oshawa, Ontario
musabbir.baki@ontariotechu.net

ABSTRACT

This paper contains two methods of detecting lanes. The first method used is using the Canny Edge detection technique to the lines produced by the lane. These edges are then used by Hough transform to detect the major features of the lane lines. To remove unnecessary image noise and objects, the image was masked to roughly contain the two lanes lines. This mask removed image artifacts like the sky, buildings, or anything that is outside of the rough triangular-shaped mask.

For the second attempt at detecting lanes, a simple Convolutional Neural Network was trained using the CULane dataset from The Chinese University of Hong Kong. Although detecting lanes using different Machine Learning techniques is worthwhile and has worked incredibly well. This simple CNN doesn't provide the correct result and requires a lot of additional modifications to be useful.

INTRODUCTION

As cars are becoming more abundant, car manufacturers are looking into ways to make them more comfortable and safe to drive. One of the safety features that has been implemented on many cars present on the road is Lane detection.

Lane detection is required for functionalities such as Lane Keep, Lane Departure Alerts, and Fully Autonomous driving. As such we have attempted to try out two ways of implementing a Lane detection program. Firstly, for Edge detection of the method, the images were filtered of their White and Yellow color. Then the edges were retrieved using the Canny Edge detection function. These Edges were then blurred and segmented before using Hough transform to create the lines that represented the lanes. Secondly, for the Machine Learning technique. A large image dataset was used to train a simple CNN model. This Model was fed raw images and then used to predict the angle and starting position of the lane lines.

1 Lane Detection with OpenCV

Our lane detection method with OpenCV focuses on filtering the image using multiple different procedures before detecting the lane lines using canny edge detection and hough transform.

The lane detection process we used starts by filtering the image to contain only white and yellow like pixels, then grayscaling and performing Gaussian blur on the result, followed up by canny edge detection, then segmenting the image to only have the part that contains the lane and finally performing hough transform to create the lane lines.

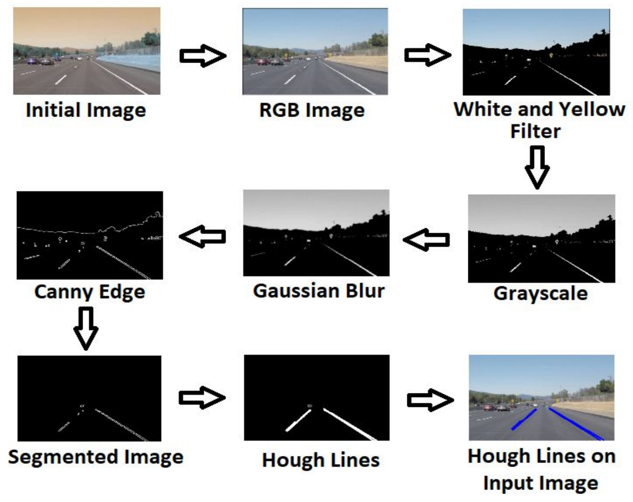


Figure 1: **The lane detection process.**

1.1 Data Set

For this method our data set consisted of images from the driver's perspective through a dashcam. We had 1 image and 1 dashcam video, both having different camera positions. The dashcam video can be considered as a set of many images, since each frame in the video is an image.

1.2 Lane Detection Process

1.2.1 White and Yellow Filter

The purpose of this step is to filter out the non-lane parts of the image by filtering based on pixel colour. Since lanes are generally either yellow or white, the idea is to only keep pixels that are of those colours and to disregard the rest.

Before we get started we convert the image read by OpenCV's imread() to RGB, since initially it will be in BGR form [1]. We do this by using OpenCV's cvtColor() function which performs the image type conversion on an image based on the arguments and returns the converted result [3]. In this case the type conversion argument is cv2.COLOR_BGR2RGB.

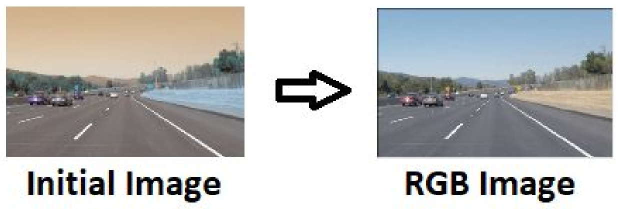


Figure 2: **Initial image converted from BGR to RGB form.**

We then define arrays to represent the start and end of the white and yellow colour's RGB ranges. This is used with OpenCV's inRange() function on the RGB input image, which returns a mask representing the values we care about [3].

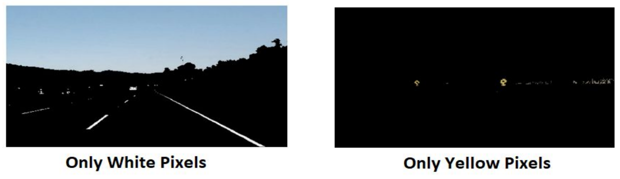


Figure 3: **The images filtered using the resulting masks.**

After combining the yellow and white masks obtained using the inRange function by using bitwise_or(), we finally filter the RGB image using bitwise_and() with our combined mask and the image.

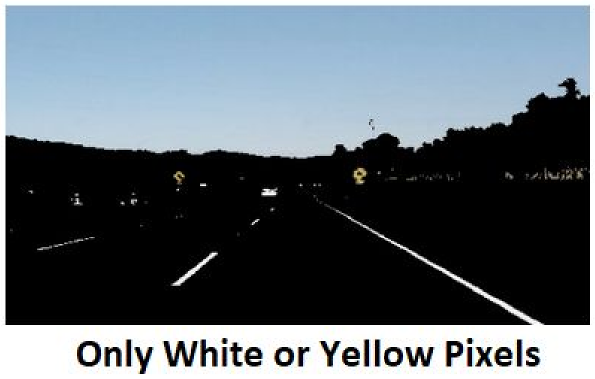


Figure 4: **The image filtered using the combined mask.**

1.2.2 Grayscale Image

In this step we convert the filtered image from the previous step to grayscale. This is done because the colours of the image are not relevant for the future steps, and by converting to grayscale we are going from 3 dimensions to 1 dimension. We are reducing the complexity of the image, and this in turn increases effectiveness for the functions used in the remainder of the process. To convert from RGB to gray, we again use OpenCV's cvtColor() function with the cv2.COLOR_RGB2GRAY argument.

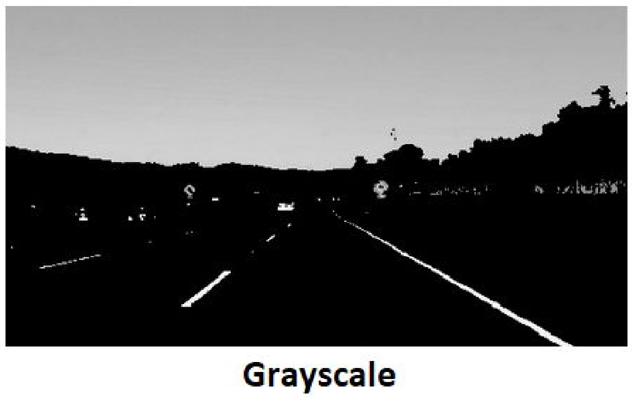


Figure 5: **The grayscale version of the filtered image.**

1.2.3 Gaussian Blur

In this step we use gaussian blur in order to smooth the image and reduce its noise. Doing this will help us reduce the chances of detecting non-lane features in the canny edge detection step.

We perform Gaussian blur using OpenCV's GaussianBlur() function [3]. For this function we pass the image, kernel size of (5, 5) and a value of 0 for sigmaX argument which represents the standard deviation in the x-direction. In this case the standard deviation for the y-direction, sigmaY argument, is given sigmaX's value since we did not provide one for it. When sigmaX and sigmaY are 0, their values are automatically calculated based on the kernel size [3].

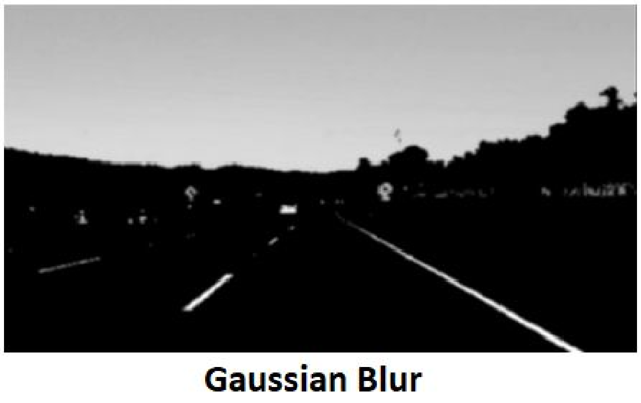


Figure 6: **The blurred version of the grayscale image.**

1.2.4 Canny Edge Detection

Now we filter the image further by performing edge detection, which assists us in detecting the lanes since the lanes are edges. We do this by using OpenCV's Canny function. This function also performs the hysteresis thresholding, we pass two numerical values into this function that represent the minimum and maximum value of the threshold [3].

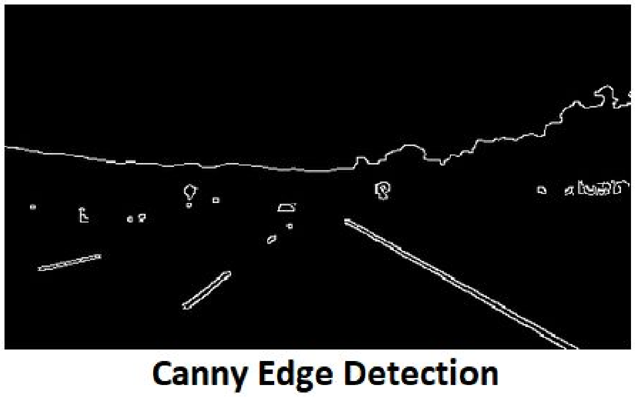


Figure 7: **Canny edge detection performed on the blurred image.**

1.2.5 Segmented Image

The image resulting from the canny edge detection has a lot of edges that we are not interested in, we want to filter them out. If the perspective does not change then the lane will always be in the same area of the image, we can use this information to our advantage. We can filter the image further by only considering the pixels that are in the general area where we believe the lane is.

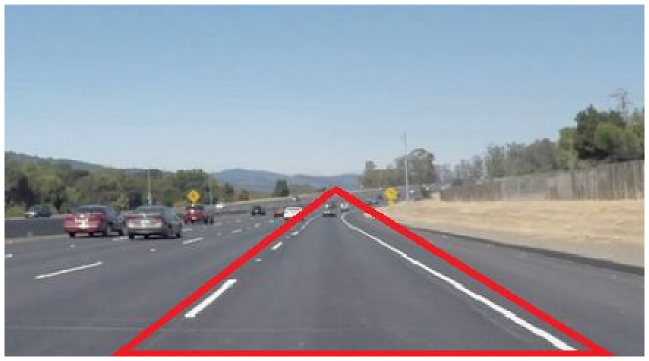


Figure 8: **The area where the lane is in this perspective.**

We need to create a mask that when used filters out all pixels except the ones in the triangle above. To do this we can use OpenCV's fillPoly() function which fills an area based on the vertices passed to it as arguments. In this case we can get the vertices of the red triangle above and use those. The ideal vertices were found manually through trial and error, in the future we would like to create a function that can find the ideal vertices dynamically. Once this mask has been created, we can use a bitwise_and()

with this and the image to filter the image.

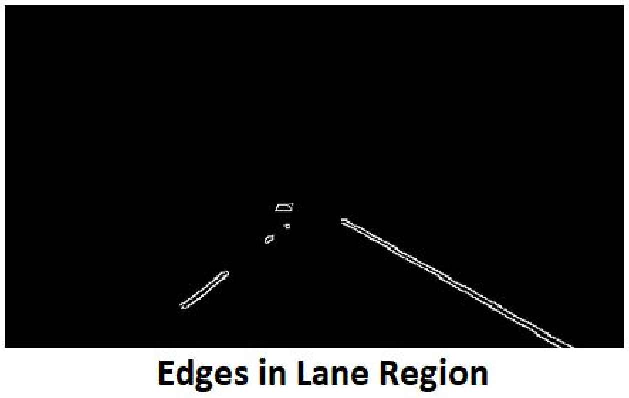


Figure 9: **Lane region of the edges image.**

1.2.6 Hough Transform

Now that we have the edges we need to get the lane line predictions. To do this we can use hough transform since it gives us lines that pass through the points on our image. It also has a voting feature where only lines with enough votes (the number of points it passes through) are returned, so this way we can filter out non-lane lines by providing a high enough vote threshold.

We perform hough transform using OpenCV's houghLinesP() function, its arguments are the values required to calculate the hough line as well as the vote threshold, minimum line length, and maximum line gap [3].

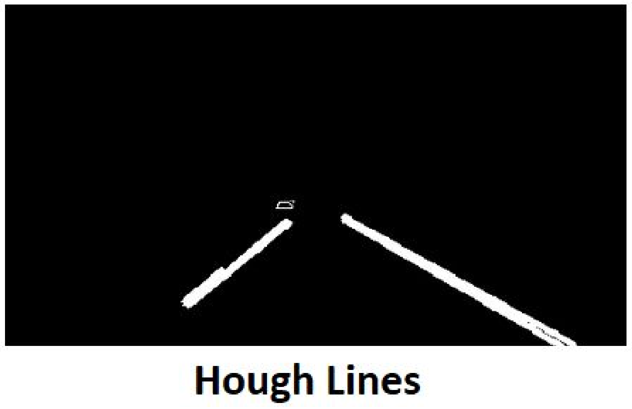


Figure 10: **Hough lines added to the edges image.**

The arguments for houghLinesP() were chosen through trial and error while checking which values had the best results. The ideal values will be different based on the quality and perspective of the images used. Finally we add the hough lines to the RGB image to show our lane.

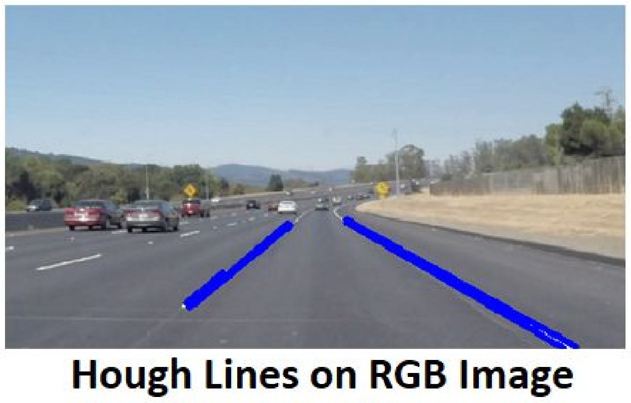


Figure 11: **Hough lines added to the RGB image.**

2 Convolutional Neural Network

Convolutional Neural Network is a powerful deep learning tool to perform generative and descriptive tasks [4]. Convolving through image Datasets can help create descriptive filters that commonly occur in the image sets. These filters can then be used to classify different images by how much each pattern fits the image. Using this general idea and an image dataset of roads from a dashcam, we attempted to train a simple CNN to see if it can help identify the lane lines.

2.1 Data Set

The dataset that has been used for this CNN model is the CULane Dataset. This data set is a large collection of around 133,235 images. The images were captured from six different drivers in Beijing from around 55 hours of recording.

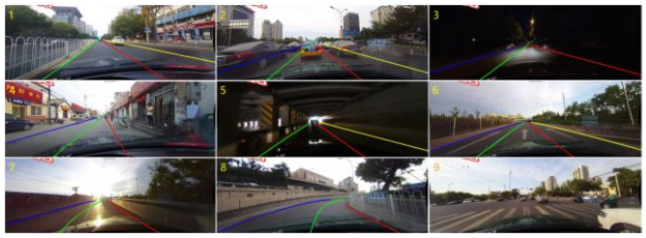


Figure 12: **9 partitions of images with labeling for lane lines. This image is taken from the Dataset site** [6]

Each image of a road is accompanied by up to 4 other label images. For instance the outer solid left lane line, the inner left lane line, right inner lane line and outer solid lane line.

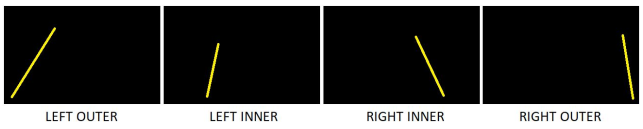


Figure 13: **4 examples of label images.**

2.2 Simple CNN Model

The Model consists of only 3 simple layers, a Conv2D layer with 32, 3x3 filters, a MaxPool2D layer of 2x2 pool size and 2 by 2 strides and lastly a Dense layer to reduce the dimensionality to 1 final integer value with a ReLu activation. The layers were then compiled with the Mean Squared Error as their Loss function.

```
model = Sequential([
   Input((590, 1640)),
   Reshape((590, 1640, 1)),
   Conv2D(32, kernel_size=(3, 3), padding='valid'),
   MaxPool2D(pool_size=(2,2), strides=(2,2)),
   Reshape((294 * 819 * 32,)),
   Dense(1, activation='relu')
])
model.compile(loss='mean_squared_error',
optimizer='adam', metrics=['acc'])
```

2.2.1 Image Preparation

Since the model is constructed to output an integer value. Each image dataset had to be prepared before training the model. Due to resource and time limitation the model was only trained using 60 randomly picked images and their label set. Only the 4 label images were read to identify the lane lines for each Road image *Figure 13. Lane Labels*. From these Lane images, the angles and the x-intercept were retrieved.

The x-intercept was taken from the last row of the image where the line starts from. The angle of the line was retrieved using the inverse tangent of the derivatives.

```
H_x = np.array([[1,1, 1,-1,-1,-1]])
H_y = np.array([[1,1,1],[-1,-1,-1]])
```

Figure 14: **Firstly, to find the derivative the Label images were Convolved using 2 filters.**

```
np.arctan2(derv_x, derv_y)
```

Figure 15: **Using the X and Y derivative of the Lebel images, the angle can be found using the Arctan formula.**

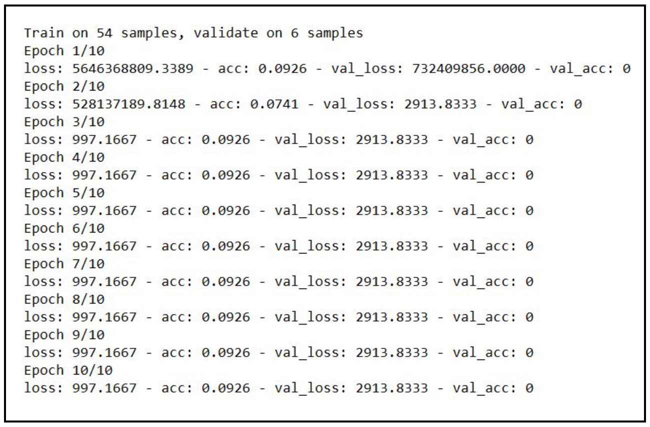
2.2.2 Model Training

The idea behind training is that the model is going to train 2 models for each Lane line, in total 8 times. Each training corresponds to 1 integer value. For instance for the Left Outer lane the Model trains twice, once for the Angle of the Lane and the second time for the Positioning of the Lane.

For each of the training, the model used 60 randomly picked images. The fit function was run with 10 epochs and 0.1 for validation set *(6 images in this case)*.

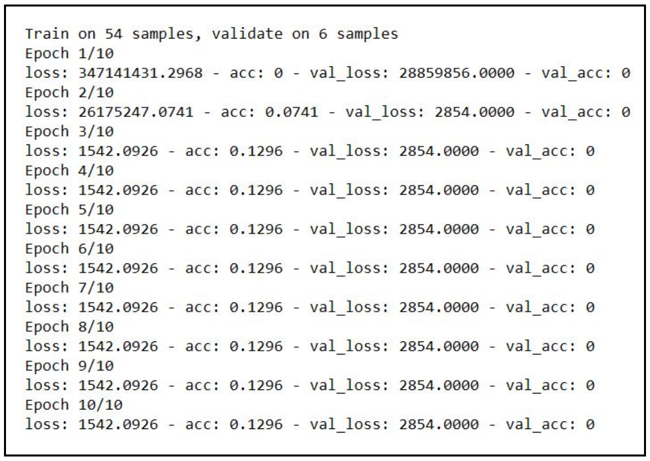
```
model.fit(images, leftLineAngle, validation_split = 0.1,
epochs=10)
```

Each training was attempted twice, one with the raw image and one after applying a rectangular mask to roughly isolate the road and lanes.



```
Train on 54 samples, validate on 6 samples
Epoch 1/10
loss: 5646368809.3389 - acc: 0.0926 - val_loss: 732409856.0000 - val_acc: 0
Epoch 2/10
loss: 528137189.8148 - acc: 0.0741 - val_loss: 2913.8333 - val_acc: 0
Epoch 3/10
loss: 997.1667 - acc: 0.0926 - val_loss: 2913.8333 - val_acc: 0
Epoch 4/10
loss: 997.1667 - acc: 0.0926 - val_loss: 2913.8333 - val_acc: 0
Epoch 5/10
loss: 997.1667 - acc: 0.0926 - val_loss: 2913.8333 - val_acc: 0
Epoch 6/10
loss: 997.1667 - acc: 0.0926 - val_loss: 2913.8333 - val_acc: 0
Epoch 7/10
loss: 997.1667 - acc: 0.0926 - val_loss: 2913.8333 - val_acc: 0
Epoch 8/10
loss: 997.1667 - acc: 0.0926 - val_loss: 2913.8333 - val_acc: 0
Epoch 9/10
loss: 997.1667 - acc: 0.0926 - val_loss: 2913.8333 - val_acc: 0
Epoch 10/10
loss: 997.1667 - acc: 0.0926 - val_loss: 2913.8333 - val_acc: 0
```

Figure 16: **Training epochs contain accuracy and loss values. The Images used for this training phase were not Masked.**



```
Train on 54 samples, validate on 6 samples
Epoch 1/10
loss: 347141431.2968 - acc: 0 - val_loss: 28859856.0000 - val_acc: 0
Epoch 2/10
loss: 26175247.0741 - acc: 0.0741 - val_loss: 2854.0000 - val_acc: 0
Epoch 3/10
loss: 1542.0926 - acc: 0.1296 - val_loss: 2854.0000 - val_acc: 0
Epoch 4/10
loss: 1542.0926 - acc: 0.1296 - val_loss: 2854.0000 - val_acc: 0
Epoch 5/10
loss: 1542.0926 - acc: 0.1296 - val_loss: 2854.0000 - val_acc: 0
Epoch 6/10
loss: 1542.0926 - acc: 0.1296 - val_loss: 2854.0000 - val_acc: 0
Epoch 7/10
loss: 1542.0926 - acc: 0.1296 - val_loss: 2854.0000 - val_acc: 0
Epoch 8/10
loss: 1542.0926 - acc: 0.1296 - val_loss: 2854.0000 - val_acc: 0
Epoch 9/10
loss: 1542.0926 - acc: 0.1296 - val_loss: 2854.0000 - val_acc: 0
Epoch 10/10
loss: 1542.0926 - acc: 0.1296 - val_loss: 2854.0000 - val_acc: 0
```

Figure 17: **Training epochs contain accuracy and loss values. The Images used for this training phase were Masked.**

Both the training shows that after 2 or 3 epochs the model stops learning. With a really high Loss value and 0 validation accuracy the model shows that it hasn't effectively learned much from the images themselves.

This is a problem, it shows that the simple convolution model isn't effective in learning enough significant patterns in the images to identify the lanes. There are many factors of why this model doesn't work. Starting off with the dataset itself, the images are quite low resolution and in most of the images the lanes are blurry and aren't the most prominent feature of the images themselves. Additionally, due to the limitations of the system, whenever the training set was more than 60 images the machine would run out of memory. This led to a really small learning set and a tiny validation set of only 6 images. Lastly, a simple 3 layered CNN will not be able to capture the right amount of pattern needed to identify the lanes from noisy images such as these which include many artifacts like crowds, cars, and buildings.

3 Improvements and Conclusion

The Lane detection problem has been evolving. There are multiple accurate implementations that work really well. However, with these two implementations there are a lot of improvements that can be made.

For the OpenCV implementation, an extra step could be added to create lanes using the hough lines generated from the last step for a better visual interpretation of the lane. Also as mentioned before, a way to determine the general area the lane is in based on perspective would be ideal so that we do not have to manually set values for segmentation for different data sets.

With regards to the improvements for the CNN implementation. Firstly, A more sophisticated model is required to detect the lanes that are not the most prominent feature of each picture. Secondly, using a larger set of images to train the model will wield better descriptors and identify the patterns better. Lastly, images may require a lot of preprocessing before they are used to train the model. This can include image sharpening, better image masking and noise reduction.

ACKNOWLEDGMENTS

Special thanks to our friends and family members who have been taking care of us during these tough times.

REFERENCES

[1] doxygen (Ed.). (2020, April 10). Image file reading and writing. Retrieved April 10, 2020, from https://docs.opencv.org/master/d4/da8/group__imgcodecs.html#ga288b8b3da0892bd651fce07b3bbd3a56
[2] Lin, C.-en. (2018, December 17). Retrieved from https://towardsdatascience.com/tutorial-build-a-lane-detector-679fd8953132
[3] opencv dev team (Ed.). (2019, December 31). Opencv Documentation! Retrieved April 10, 2020, from https://docs.opencv.org/2.4/
[4] Rouse, M. (2018, April 26). What is convolutional neural network? - Definition from WhatIs.com. Retrieved April 10, 2020, from https://searchenterpriseai.techtarget.com/definition/convolutional-neural-network
[5] Sqalli, M. (2016, November 6). Retrieved from https://medium.com/@MSqalli/lane-detection-446986c44021
[6] Zhan, X., Li, J., Cao, X., Shi, J., Luo, P., & Tang, X. (2018, February). Spatial As Deep: Spatial CNN for Traffic Scene Understanding.