SELF DRIVING CARS

INTRODUCTION AND HISTORY

- A self-driving car, also known as an autonomous vehicle, driverless car, or robo-car, is a vehicle that is capable of sensing its environment and moving safely with little or no human input.
- Experiments have been conducted on automated driving systems (ADS) since at least the 1920s, trials began in the 1950s. The first semi-automated car was developed in 1977, by Japan's Tsukuba Mechanical Engineering Laboratory, which required specially marked streets that were interpreted by two cameras on the vehicle and an analog computer. The vehicle reached speeds up to 30 kilometers per hour (19 mph) with the support of an elevated rail.

SELF DRIVING CAR

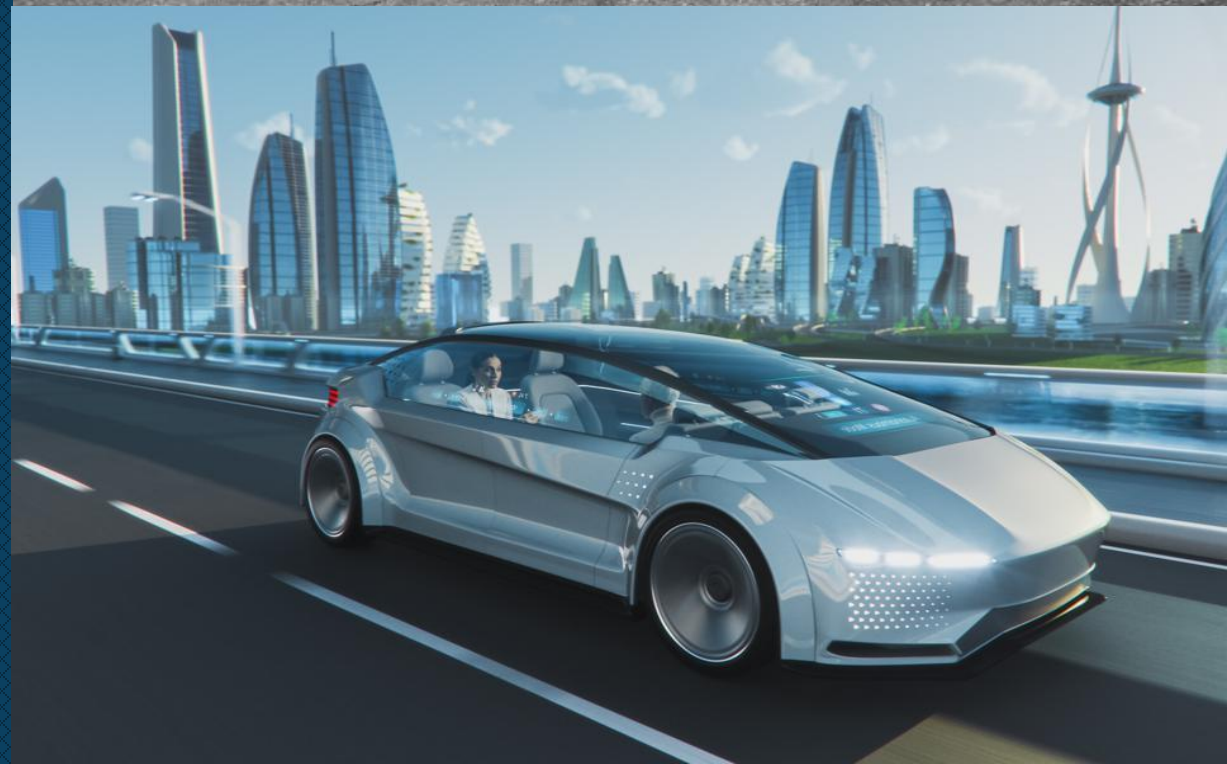
LANE DETECTION

Object Identification

Traffic signs classification

Tracking

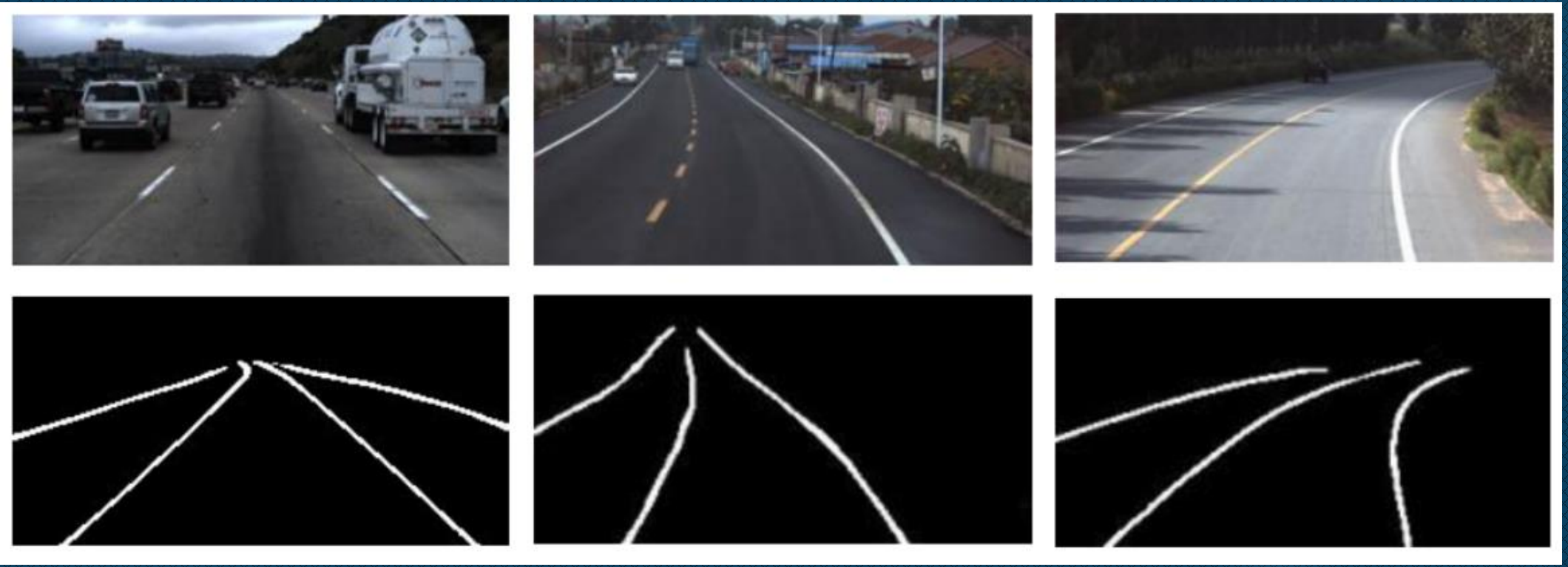
Predicting steering angles



LANE DETECTION

LANE DETECTION - OpenCV

- Lane detection is a critical component of self-driving cars and autonomous vehicles. It is one of the most important research topics for understanding driving scene.



LANE DETECTION - OpenCV

- **CAPTURING AND DECODING VIDEO FILE:** We will capture the video using VideoCapture object and after the capturing has been initialized every video frame is decoded (i.e. converting into a sequence of images).
- **GRAYSCALE CONVERSION OF IMAGE**: The video frames are in RGB format, RGB is converted to grayscale because processing a single channel image is faster than processing a three-channel colored image.

```
cv2.cvtColor(image, cv2.COLOR_RGB2GRAY)
```

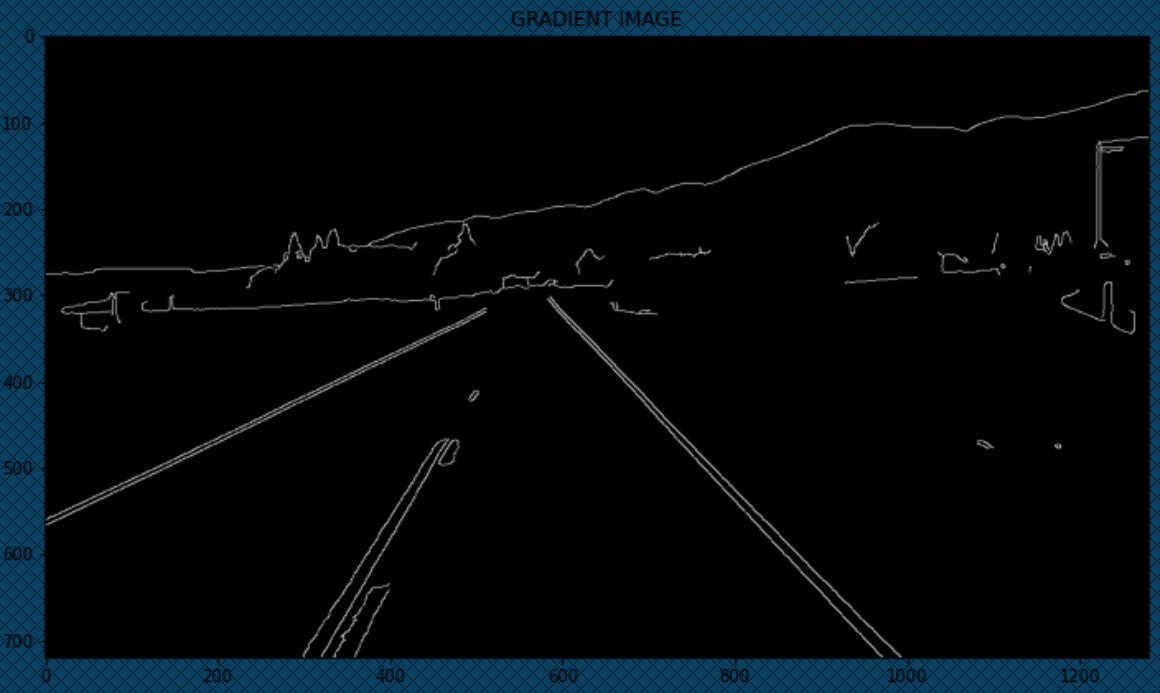
- **NOISE REDUCTION**: Noise can create false edges, therefore before going further, it's imperative to perform image smoothening. Gaussian filter is used to perform this process.

cv2.GaussianBlur(gray, (5, 5), 0) Gaussian filter with a 5x5 kernel

LANE DETECTION - OpenCV

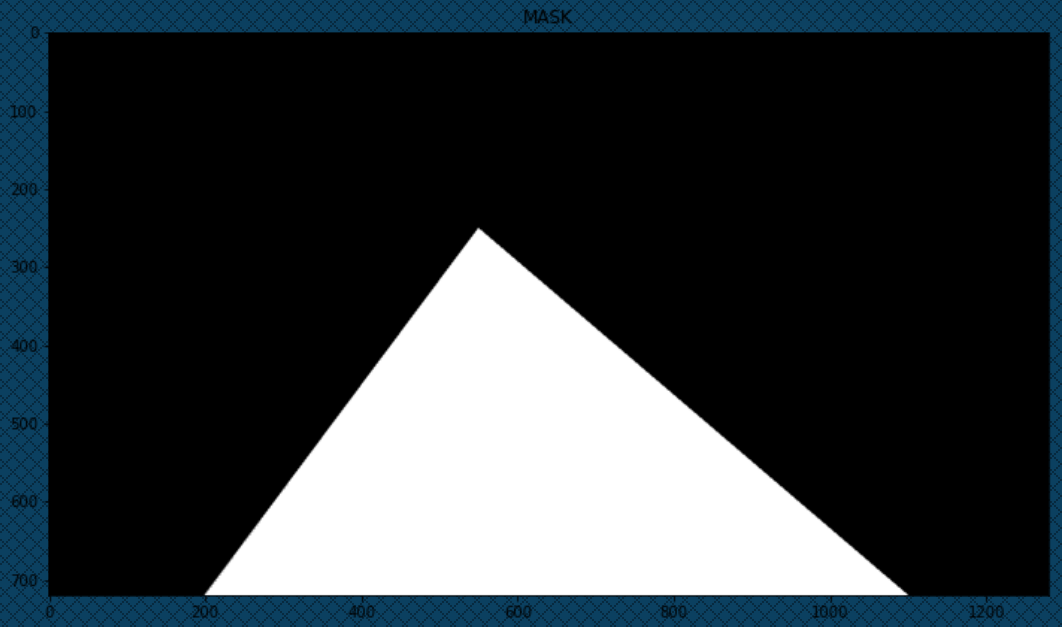
- **EDGE DTECTION - Canny Edge Detector**: It computes gradient in all directions of our blurred image and traces the edges with large changes in intensity. Outlines strongest gradients in image. Based on the smoothed image, derivatives in both the x (width) and y (height) direction are computed; these in turn are used to compute the gradient magnitude of the image.

cv2.Canny(blur, low_threshold, high_threshold)



LANE DETECTION - OpenCV

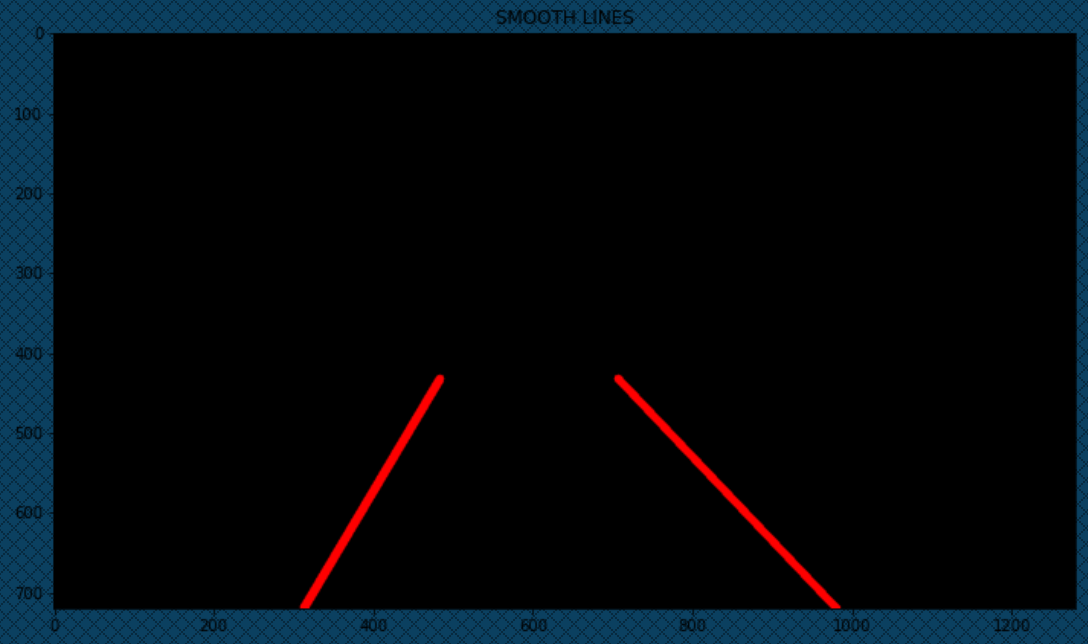
- **REGION OF INTEREST**: This step is to take into account only the region covered by the road lane. A mask is created here, which is of the same dimension as our road image. Furthermore, bitwise AND operation is performed between each pixel of our canny image and this mask. It ultimately masks the canny image and shows the region of interest traced by the polygonal contour of the mask.



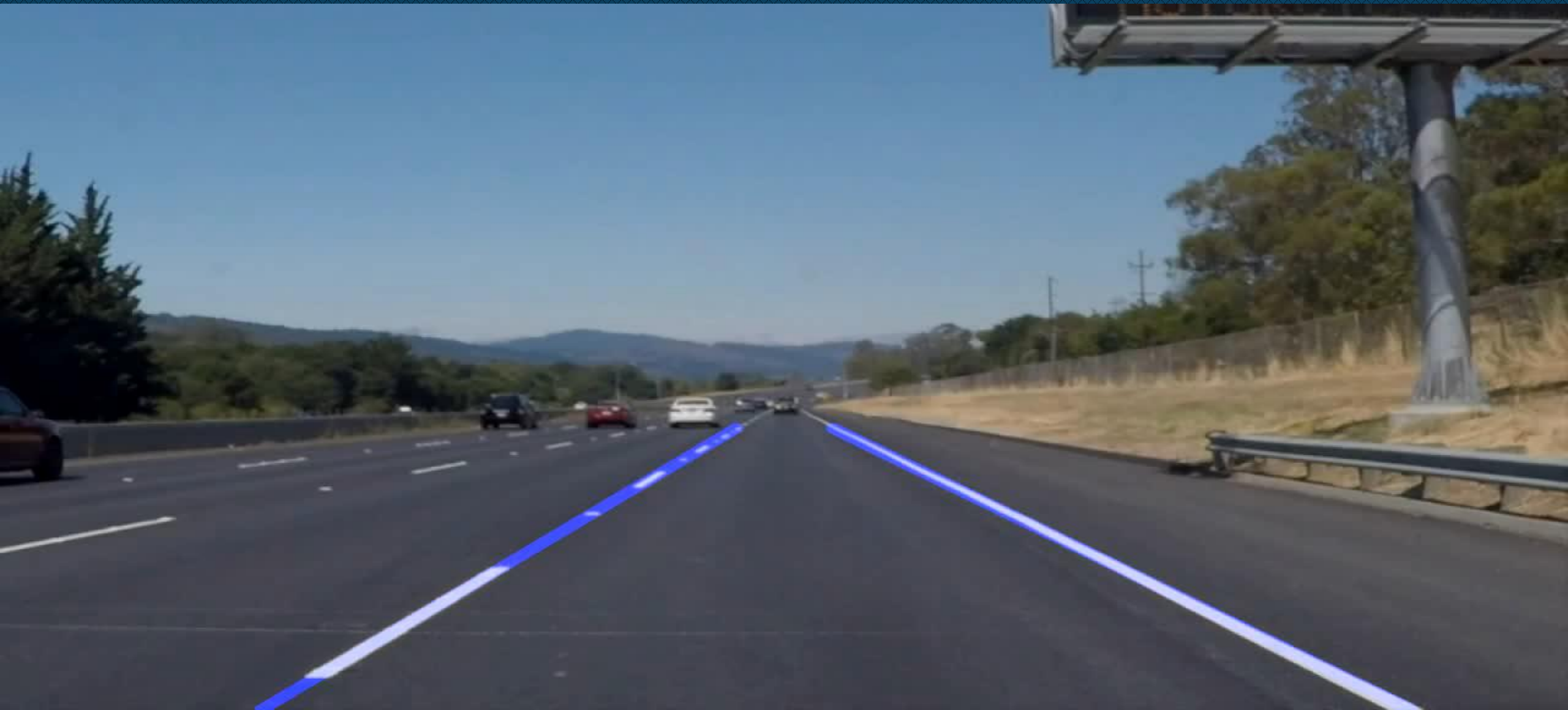
LANE DETECTION - OpenCV

- **HOUGH LINE TRANSFORM**: The Hough Line Transform is a transform used to detect straight lines. The Probabilistic Hough Line Transform is used here, which gives output as the extremes of the detected lines.

```
cv2.HoughLinesP(masked_image, rho, theta,
        threshold, minLineLength, maxLineGap)
```



LANE DETECTION - OpenCV



OBJECT IDENTIFICATION

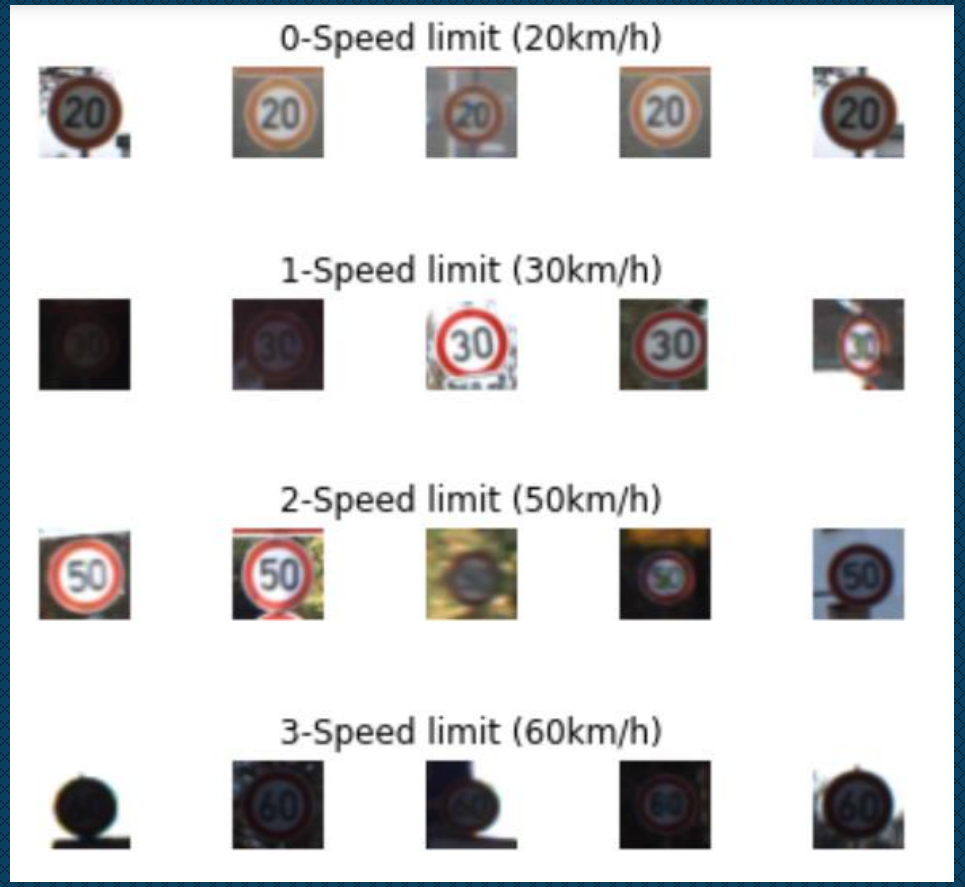
Object Identification - CNN

EDA

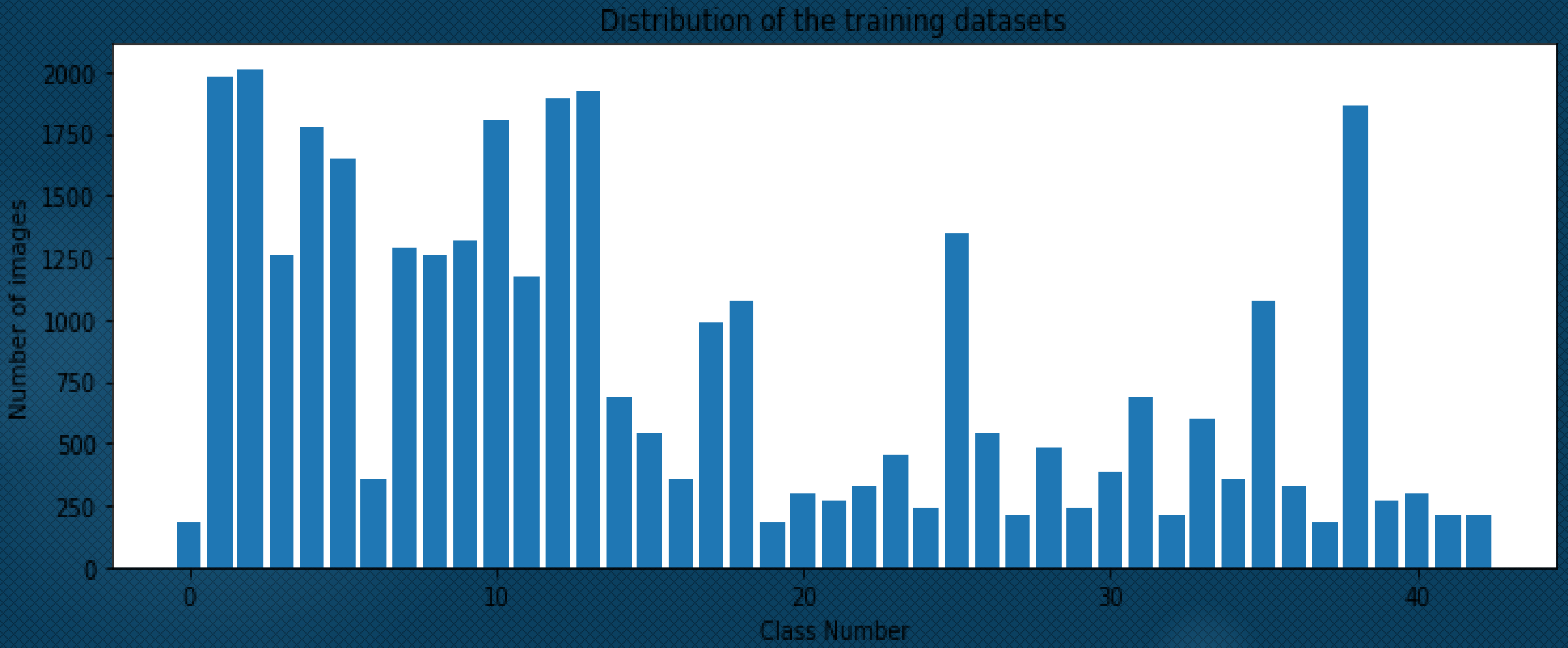
- For Object Identification, GERMAN TRAFFIC SIGN dataset is used. Pickled Dataset is cloned from https://bitbucket.org/jadslim/german-traffic-signs.
- Dataset contains train.p - Training set, test.p - Test set, valid.p - Validation set, Signnames.csv - Traffic sign Names.
- The pickled data is a dictionary with 4 key/value pairs:
- 'features' is a 4D array containing raw pixel data of the traffic sign images, (num examples, width, height, channels).

  The dimensions of the images are 32x32x3.
- 'labels' is a 1D array containing the label/class id of the traffic sign. The file signnames.csv contains id -> name mappings for each id.

  number of labels = 43



Object Identification - CNN



Object Identification - CNN

DATA PREPROCESSING

As a part of data preprocessing

Grayscale Conversion, Histogram Equalization,

Normalization, Reshaping

MODELING: LeNet-4

The six layers of LeNet-4 were as follows:

Layer C1: Convolution Layer (num_kernels=30, kernel_size=5×5)

Layer S2: Average Pooling Layer (kernel_size=2×2)

Layer C3: Convolution Layer (num_kernels=15, kernel_size=3x3)

Layer S4: Average Pooling Layer (kernel_size=2×2)

Layer F5: Fully Connected Layer (out_features=512)

Layer F6: Fully Connected Layer (out_features=43)

```
def leNet_model(filters_l1, filters_l2, units_fc, lr):
    model = Sequential()

    # First Convolutional and Pooling Layer
    model.add(Conv2D(filters_l1, (5, 5), input_shape = (32, 32, 1), activation = 'relu'))
    model.add(MaxPooling2D(pool_size = (2, 2)))

    # Second Convolution and Pooling Layer
    model.add(Conv2D(filters_l2, (3, 3), activation = 'relu'))
    model.add(MaxPooling2D(pool_size = (2, 2)))

    # Flatten Images
    model.add(Flatten())

    # Fully Connected Layer
    model.add(Dense(units_fc, activation = 'relu'))

    # Dropout Layer
    model.add(Dropout(0.5))

    # Output Layer
    model.add(Dense(num_classes, activation = 'softmax'))

    # Compile Model
    model.compile(Adam(learning_rate = lr), loss = 'categorical_crossentropy', metrics = ['accuracy'])

    return model
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 28, 28, 30)        780
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 14, 14, 30)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 12, 12, 15)        4065
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 6, 6, 15)          0
_________________________________________________________________
flatten (Flatten)            (None, 540)               0
_________________________________________________________________
dense (Dense)                (None, 512)               276992
_________________________________________________________________
dropout (Dropout)            (None, 512)               0
_________________________________________________________________
dense_1 (Dense)              (None, 43)                22059
=================================================================
Total params: 303,896
Trainable params: 303,896
Non-trainable params: 0
_________________________________________________________________
None
```

Object Identification - CNN

MODEL PERFORMANCE

- LeNet-4 Model Test Accuracy: 0.8908947110176086

- To improve model performance the model is fine tuned by adding more layers.

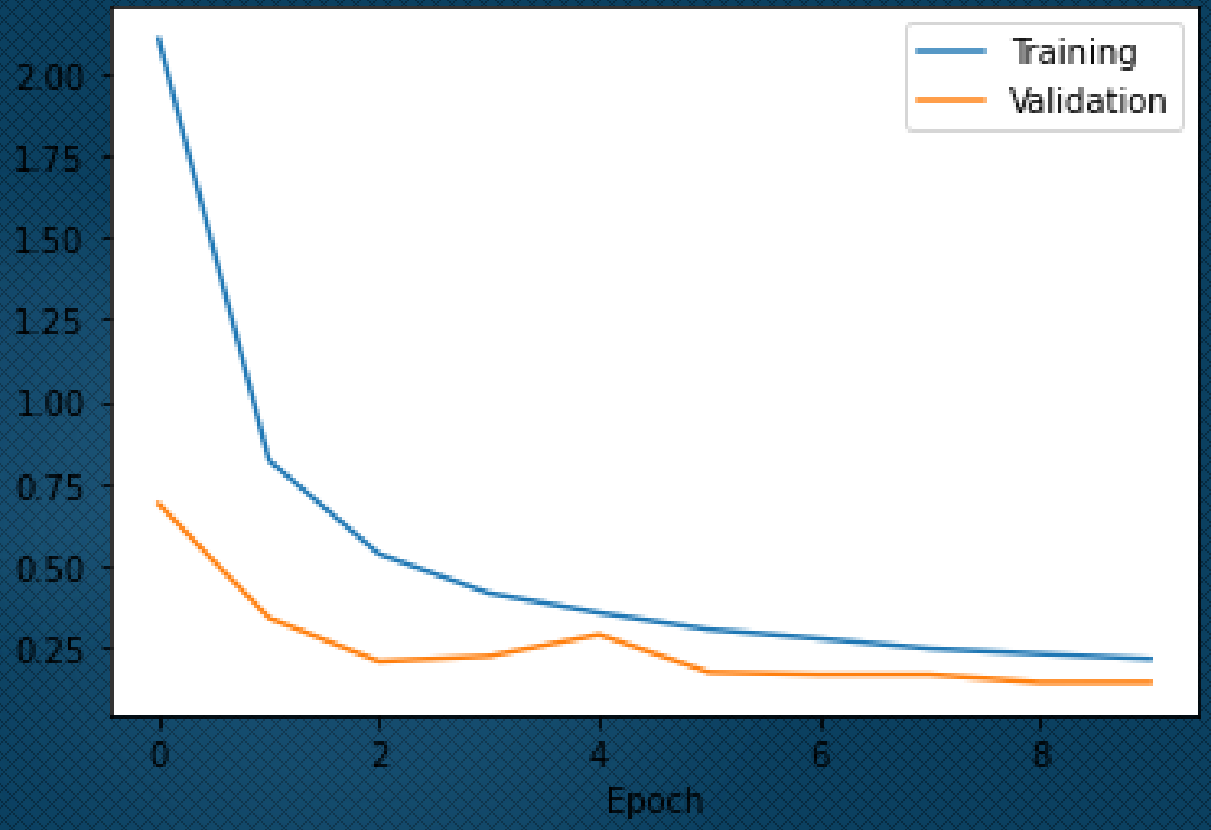
Test Accuracy of fine tuned model: 0.9538400769233704

- To further improve the model performance, Images are augmented using ImageDataGenerator

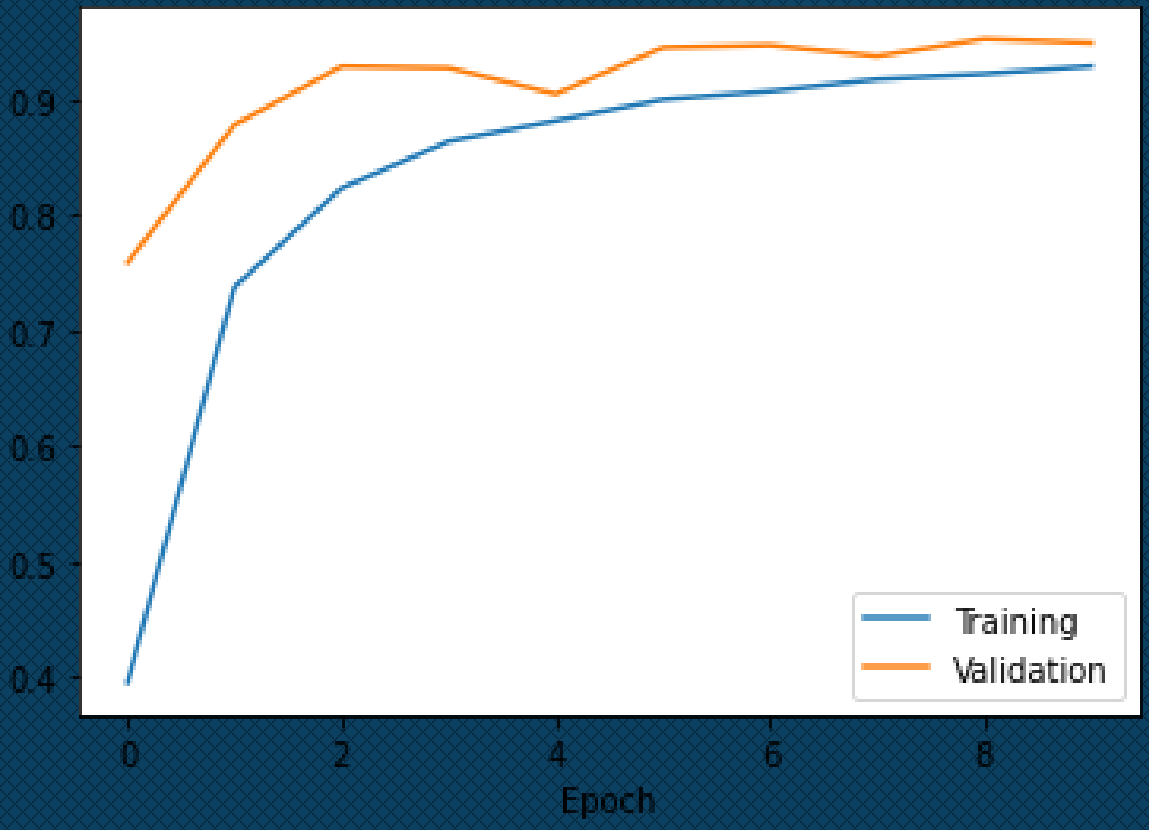
```
from keras.preprocessing.image import ImageDataGenerator

datagen = ImageDataGenerator(width_shift_range = 0.1,
                             height_shift_range = 0.1,
                             zoom_range = 0.2,
                             shear_range = 0.1,
                             rotation_range = 10,
                             horizontal_flip = True)

datagen.fit(X_train)
```

Object Identification - CNN

LOSS



ACCURACY



Object Identification - CNN

PREDICTION RESULT

```
# Test Image
pred = np.argmax(final_model.predict(img), axis = 1)
print("Predicted sign: ", pred)
print(data['SignName'][pred])
```

```
Predicted sign:  [29]
29    Bicycles crossing
Name: SignName, dtype: object
```


```
# Test Image
pred = np.argmax(final_model.predict(img), axis = 1)
print("Predicted sign: ", pred)
print(data['SignName'][pred])
```

```
Predicted sign:  [34]
34    Turn left ahead
Name: SignName, dtype: object
```

TRACKING Behavioral Cloning

TRACKING – Udacity Simulator

BEHAVIORAL CLONING

Behavioral Cloning is literally **cloning** the **behavior** of the **driver**. The idea is to train Convolution Neural Network(CNN) to mimic the **driver** based on training data from **driver's driving**.

To Track steering angles of a car by behavioral cloning the following steps are followed

1. Data Recording
2. Data Processing
3. Model Training
4. Model Testing

DATA RECORDING

The simulator has two modes - Training mode and Autonomous mode. Training mode is used to collect training data by driving through the tracks and recording the driving data in a folder. The Autonomous mode is used to test a trained model.

TRACKING – Udacity Simulator

DATA PROCESSING

Data processing is done to allow our model to be able to easily work with raw data for training. In this project, the data processing is built into a generator (keras fit_generator) to allow for real-time processing of the data. The advantage here is that, in the case that we are working with a very large amount of data, the whole dataset is not loaded into memory, we can therefore work with a manageable batch of data at a time. Hence the generator is run in parallel to the model, for efficiency.

The following are the processing steps carried out on the data:

- **Randomly choose from center, left and right camera images:** The simulator provides three camera views namely; center, left and right views. Since we are required to use only one camera view, we choose randomly from the three views. While using the left and right images, we add and subtract 0.25 to the steering angles respectively to make up for the camera offsets 1.
- **Translate image (Jitter) and compensate for steering angles:** Since the original image size is 160x320 pixels, we randomly translate image to the left or right and compensate for the translation in the steering angles with 0.008 per pixel of translation. We then crop a region of interest of 120x220 pixel from the image. Note that I only translated in the horizontal direction for my solution.

TRACKING – Udacity Simulator

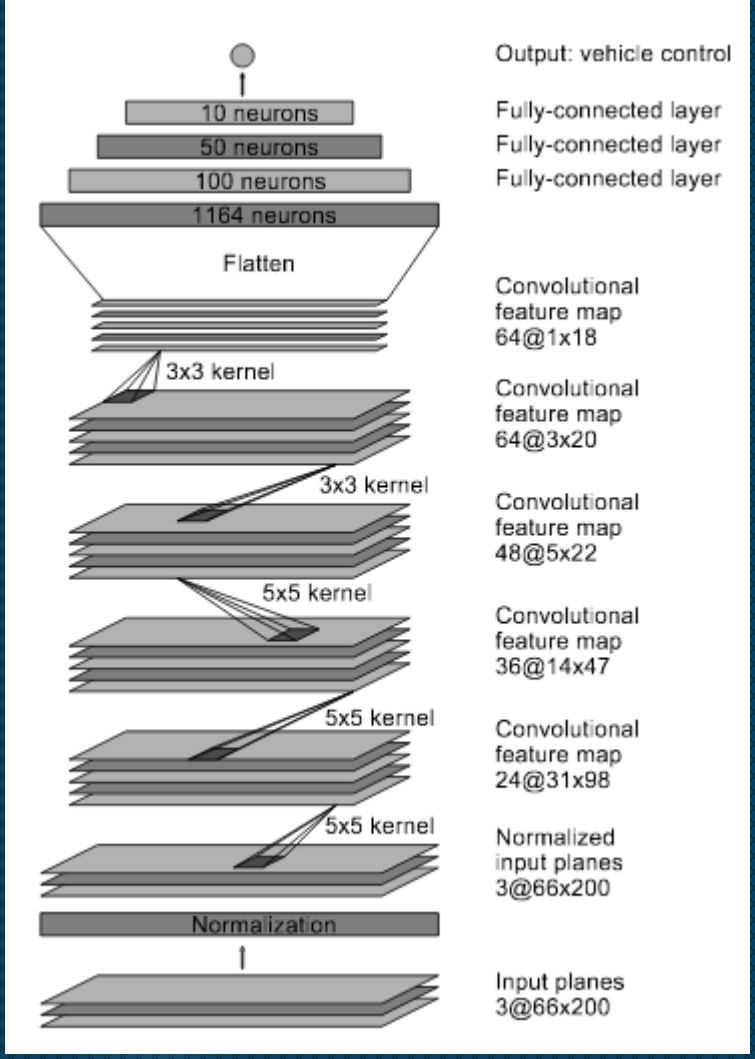
DATA PROCESSING

- **Randomly flip image:** In other to balance left and right images, we randomly flip images and change sign on the steering angles. The following figure shows the view from the left, right and center cameras after been jittered, cropped, and angles corrected. The right camera view has been flipped so it looks like a left camera image.
- **Brightness Augmentation** We simulate different brightness occasions by converting image to HSV channel and randomly scaling

TRACKING – Udacity Simulator

MODELING

The Nvidia model was adopted for training, because it gave better result after experimenting with other kinds of model (e.g. comma.ai). The network consists of 9 layers, including a normalization layer, 5 convolutional layers and 3 fully connected layers. Converse to the Nvidia model, input image was split to HSV planes before been passed to the network.



MODEL TESTING

The trained model is tested on the first track. This gave a good result, as the car could drive on the track smoothly.

https://drive.google.com/file/d/13HRePDByvP59PMka2FXCXAgRc7l3z4oh/view

TRACKING – Udacity Simulator

Conclusion

- A deep learning model to drive autonomously on a simulated track was trained by using a human behavior-sampled data. The model, which includes 5 layers of convolution and 3 more fully connected layers, was able to learn by cloning human behavior, and was able to generalize response to a new test track. Though the model only controls the steering angle, it can also be extended to control throttle and brake.

Thank You!