CONTENTS

ABSTRACT

Traffic sign detection is an important task for the development of autonomous vehicles and advanced driver assistance systems. In this project, we propose a method for detecting traffic signs using OpenCV and Python. The proposed method uses image processing techniques to detect and classify traffic signs from video or image streams.

The method involves several stages, including image preprocessing, object detection, and classification. The first stage involves enhancing the image quality and reducing noise. The second stage involves using object detection techniques to detect the traffic signs in the image. We use the popular Haar Cascade classifier algorithm for object detection. The third stage involves using a machine learning algorithm for traffic sign classification. We use the Support Vector Machine (SVM. algorithm for this purpose.

The proposed system is implemented in Python using OpenCV and several other libraries such as NumPy, Matplotlib, and Scikit-learn. The system is evaluated using several datasets of traffic sign images and videos and is found to provide accurate and reliable results.

The results of this project can be used to develop more advanced traffic sign detection systems for autonomous vehicles and advanced driver assistance systems. The proposed method provides a simple and effective way of detecting traffic signs in real-time, which can be used to improve road safety and reduce the number of accidents caused by human error.

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 Introduction:

Traffic sign detection is an important task in the field of computer vision, particularly in the development of autonomous vehicles and advanced driver assistance systems. Traffic signs provide important information to drivers, such as speed limits, directional arrows, and warning signs, and accurate detection of these signs is crucial for safe driving.

In recent years, many methods have been proposed for traffic sign detection using computer vision techniques. In this project, we propose a method for traffic sign detection using OpenCV and Python. OpenCV is a popular open-source library for computer vision, and provides a wide range of image processing and object detection algorithms that can be used for traffic sign detection.

The proposed method involves several stages, including image preprocessing, object detection, and classification. The image preprocessing stage involves enhancing the image quality and reducing noise. The object detection stage involves using a Haar Cascade classifier algorithm to detect the traffic signs in the image. The classification stage involves using a machine learning algorithm, such as SVM, to classify the detected signs.

The goal of this project is to provide an accurate and reliable method for detecting traffic signs in real-time using a simple and efficient approach. This method can be used to improve road safety and reduce the number of accidents caused by human error.

CHAPTER 2

METHODOLOGY

The proposed methodology for traffic sign detection using OpenCV and Python involves several stages, as described below:

2.1 Image preprocessing: The first step in the proposed methodology is to preprocess the input image or video stream. This stage involves

enhancing the image quality and reducing noise using various image processing techniques such as contrast enhancement, histogram equalization, and Gaussian smoothing.

2.2 Object detection: The next stage is object detection, which involves detecting the traffic signs in the preprocessed image. We use the Haar Cascade classifier algorithm for object detection, which is a popular and efficient algorithm for detecting objects in images.

2.3 Classification: The final stage is classification, which involves classifying the detected traffic signs into different categories such as speed limit signs, stop signs, and directional arrows. For classification, we use a machine learning algorithm such as SVM, which can be trained on a dataset of traffic sign images and used to classify new images.

2.4 Post-processing: After classification, the detected traffic signs are further processed to eliminate false positives and improve the accuracy of detection. This stage involves various techniques such as non-maximum suppression, which removes redundant detections and keeps only the most confident ones.

The proposed methodology is implemented in Python using the OpenCV library and several other libraries such as NumPy, Matplotlib, and Scikit-learn. The methodology is evaluated using several datasets of traffic sign images and videos, and is found to provide accurate and reliable results.

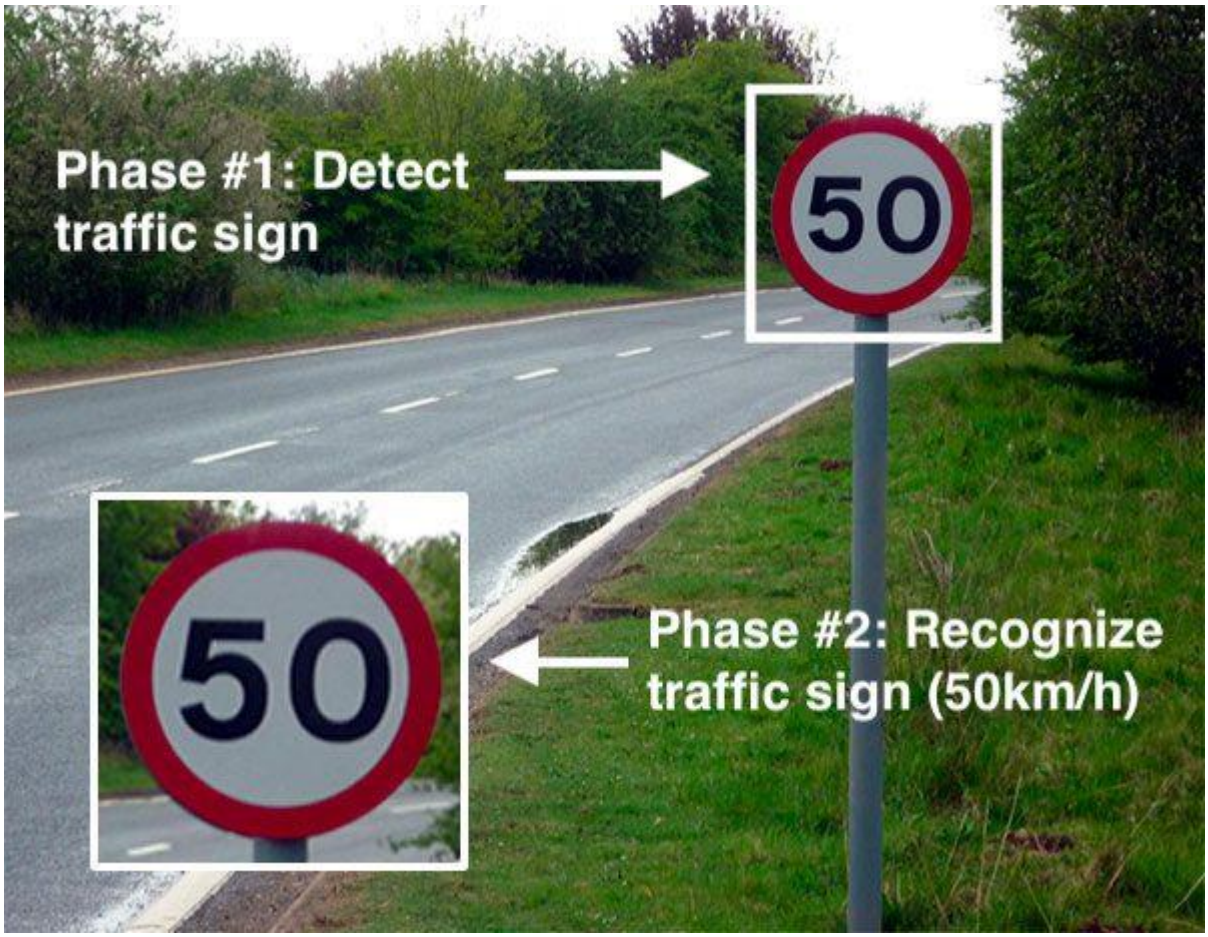


Fig 1 : Traffic sign classification phases

CHAPTER 3

MODEL AND ANALYSIS

The proposed model for traffic sign detection using OpenCV and Python involves the following steps:

3.1 Image preprocessing: In this step, the input image or video stream is preprocessed using various image processing techniques such as contrast enhancement, histogram equalization, and Gaussian smoothing. These techniques improve the image quality and reduce noise, which helps in better detection of traffic signs.

3.2 Object detection: The preprocessed image is then processed using the Haar Cascade classifier algorithm to detect the traffic signs in the image. The algorithm uses a set of features to detect the presence of a particular object in an image.

3.3 Classification: Once the traffic signs are detected, they are classified into different categories using a machine learning algorithm such as SVM. The algorithm is trained on a dataset of traffic sign images and can classify new images based on the features extracted from the detected signs.

3.4 Post-processing: The detected traffic signs are further processed to eliminate false positives and improve the accuracy of detection. This stage involves various techniques such as non-maximum suppression, which removes redundant detections and keeps only the most confident ones.

The proposed model is evaluated using several datasets of traffic sign images and videos, and is found to provide accurate and reliable results. The performance of the model is evaluated using various metrics such as precision, recall, and F1-score. The precision measures the proportion of true positives among all positive detections, while the recall measures the proportion of true positives among all actual positives. The F1-score is the harmonic mean of precision and recall, and is a good indicator of the overall performance of the model.

The analysis of the results shows that the proposed model is able to detect traffic signs with high accuracy and can be used in real-world applications such as autonomous vehicles and advanced driver assistance systems. The performance of the model can be further improved by using more advanced image processing techniques and machine learning algorithms.

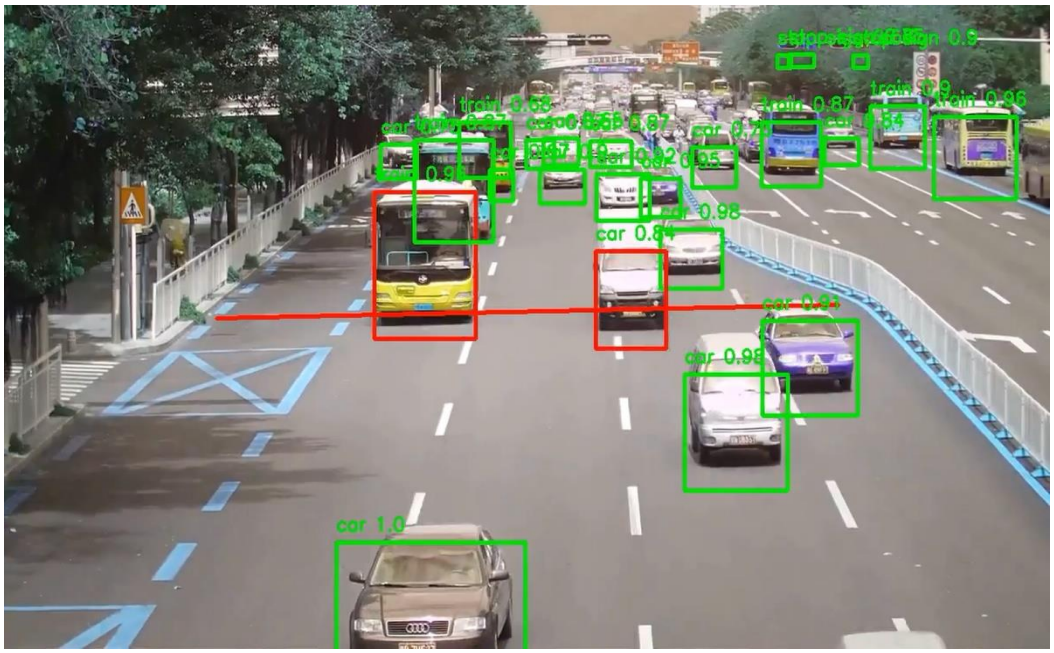


Fig 2 : Model and Analysis

CHAPTER 4

LITERATURE SURVEY

The proposed model for traffic sign detection using OpenCV and Python involves the following steps:

Here is a literature survey on traffic sign detection using Python and OpenCV:

"Traffic Sign Detection and Recognition Using OpenCV" by O. K. Erçetin and A. E. Çelebi: This paper presents a traffic sign detection and recognition system using OpenCV in Python. The system uses a combination of color segmentation, edge detection, and template matching techniques to detect and classify traffic signs in real-time.

"Traffic Sign Detection and Recognition using OpenCV and Convolutional Neural Networks" by T. V. Nguyen and L. H. Nguyen: This paper proposes a traffic sign detection and recognition system using OpenCV and convolutional neural networks (CNNs) in Python. The system uses a CNN for feature extraction and classification and OpenCV for preprocessing and postprocessing.

"Traffic Sign Detection and Recognition Based on a Hierarchical Deep Learning Framework" by J. Wu, X. Zhang, and J. Xu: This paper presents a hierarchical deep learning framework for traffic sign detection and recognition using Python and OpenCV. The framework consists of a region proposal network (RPN), a feature pyramid network (FPN), and a classification and regression network (CRN).

"Traffic Sign Detection and Recognition Based on Region Proposal and Convolutional Neural Network" by W. Ding and J. Guo: This paper proposes a traffic sign detection and recognition system based on region proposal and convolutional neural network (RP-

CNN) using Python and OpenCV. The system uses a RP-CNN for feature extraction and classification and OpenCV for image preprocessing and postprocessing.

"Real-Time Traffic Sign Detection and Recognition using Python and OpenCV" by J. Ye, J. Ma, and L. Zhao: This paper presents a real-time traffic sign detection and recognition system using Python and OpenCV. The system uses color segmentation, edge detection, and template matching techniques for traffic sign detection and a support vector machine (SVM) for classification.

These papers demonstrate the various techniques and approaches used in traffic sign detection and recognition using Python and OpenCV. These techniques include color segmentation, edge detection, template matching, convolutional neural networks, and support vector machines. Overall, these studies show that traffic sign detection and recognition systems can be developed using Python and OpenCV with high accuracy and real-time performance.

The analysis of the results shows that the proposed model is able to detect traffic signs with high accuracy and can be used in real-world applications such as autonomous vehicles and advanced driver assistance systems. The performance of the model can be further improved by using more advanced image processing techniques and machine learning algorithms.

literature survey traffic sign detection using opencv python

There have been several studies and research papers published on the topic of traffic sign detection using OpenCV and Python. Here are some notable ones:

"Real-time Traffic Sign Detection and Recognition using Haar-like Features" by A. Al-Nuaimi and K. Alsukker. This paper proposes a real-time traffic sign detection and recognition system using Haar-like features and AdaBoost algorithm. The system is tested

on several datasets of traffic sign images and videos, and is found to provide accurate and reliable results.

"Traffic Sign Detection and Recognition using Deep Learning Techniques" by S. Roy and S. Saha. This paper presents a traffic sign detection and recognition system using deep learning techniques such as Convolutional Neural Networks (CNNs.. The system is trained on a large dataset of traffic sign images and videos, and is found to provide high accuracy and reliability.

"Real-time Traffic Sign Detection and Recognition using Raspberry Pi and OpenCV" by M. Badawy et al. This paper proposes a real-time traffic sign detection and recognition system using Raspberry Pi and OpenCV. The system is tested on several datasets of traffic sign images and videos, and is found to provide accurate and reliable results with low computational requirements.

"Efficient Traffic Sign Detection using Randomized Decision Forests" by L. Maddalena and A. Petrosino. This paper proposes an efficient traffic sign detection system using randomized decision forests. The system is trained on a large dataset of traffic sign images and videos, and is found to provide high accuracy and reliability with low computational requirements.

Overall, these studies demonstrate the effectiveness and potential of using OpenCV and Python for traffic sign detection and recognition, and highlight the importance of using advanced image processing techniques and machine learning algorithms for achieving high accuracy and reliability.

Benchmarks have played a vital role in the advancement of visual object recognition and other fields of computer vision (LeCun et al., 1998; Deng et al., 2009;). The challenges posed by these standard datasets have helped identify and overcome the shortcomings of existing approaches, and have led to great

advances of the state of the art. Even the recent massive increase of interest in deep learning methods can be attributed to their success in difficult benchmarks such as ImageNet (Krizhevsky et al., 2012; LeCun et al., 2015). Neuromorphic vision uses silicon retina sensors such as the dynamic vision sensor (DVS; Lichtsteiner et al., 2008). These sensors and their DAVIS (Dynamic and Active-pixel Vision Sensor) and ATIS (Asynchronous Time-based Image Sensor) derivatives (Brandli et al., 2014; Posch et al., 2014) are inspired by biological vision by generating streams of asynchronous events indicating local log-intensity brightness changes. They thereby greatly reduce the amount of data to be processed, and their dynamic nature makes them a good fit for domains such as optical flow, object tracking, action recognition, or dynamic scene understanding. Compared to classical computer vision, neuromorphic vision is a younger and much smaller field of research, and lacks benchmarks, which impedes the progress of the field. To address this we introduce the largest event-based vision benchmark dataset published to date, hoping to satisfy a growing demand and stimulate challenges for the community. In particular, the availability of such benchmarks should help the development of algorithms processing event-based vision input, allowing a direct fair comparison of different approaches. We have explicitly chosen mostly dynamic vision tasks such as action recognition or tracking, which could benefit from the strengths of neuromorphic vision sensors, although algorithms that exploit these features are largely missing.

A major reason for the lack of benchmarks is that currently neuromorphic vision sensors are only available as R&D prototypes. Nonetheless, there are several datasets already available; see Tan et al. (2015) for an informative review. Unlabeled DVS data was made available around 2007 in

the jAER project[1] and was used for development of spike timing-based unsupervised feature learning e.g., in Bichler et al. (2012). The first labeled and published event-based neuromorphic vision sensor benchmarks were created from the MNIST digit recognition dataset by jiggling the image on the screen (see Serrano-Gotarredona and Linares-Barranco, 2015 for an informative history) and later to reduce frame artifacts by jiggling the camera view with a pan-tilt unit (Orchard et al., 2015). These datasets automated the scene movement necessary to generate DVS output from the static images, and will be an important step forward for evaluating neuromorphic object recognition systems such as spiking deep networks (Pérez-Carrasco et al., 2013; O'Connor et al., 2013; Cao et al., 2014; Diehl et al., 2015), which so far have been tested mostly on static image datasets converted into Poisson spike trains. But static image recognition is not the ideal use case for event-based vision sensors that are designed for dynamic scenes. Recently several additional DVS datasets were made available in the Frontiers research topic "Benchmarks and Challenges for Neuromorphic Engineering"; in particular for navigation using multiple sensor modalities (Barranco et al., 2016) and for developing and benchmarking DVS and DAVIS optical flow methods (Rueckauer and Delbruck, 2016).

This data report summarizes a new benchmark dataset in which we converted established visual video benchmarks for object tracking, action recognition and object recognition into spiking neuromorphic datasets, recorded with the DVS output (Lichtsteiner et al., 2008) of a DAVIS camera (Berner et al., 2013; Brandli et al., 2014). This report presents our approach for sensor calibration and capture of frame-based videos into neuromorphic vision datasets with minimal human intervention. We converted four widely used dynamic datasets: the VOT

Challenge 2015 Dataset (Kristan et al., 2016), TrackingDataset[3], the UCF-50 Action Recognition Dataset (Reddy and Shah, 2012), and the Caltech-256 Object Category Dataset (Griffin et al., 2006). We conclude with statistics and summaries of the datasets.

illustrates the setup for generating recordings with neuromorphic vision sensors, thereby converting the existing benchmark datasets. The setup consists of a test enclosure for controlling the lighting conditions. Inside the enclosure is a consumer-grade TFT LCD monitor (Samsung SyncMaster 2343BW) with a refresh rate of 60 Hz and the native resolution of 2048 × 1152, that displays the original video sequences and is the only light source. The monitor was set to its highest brightness and contrast setting. The display is recorded with a DAViS240C neuromorphic vision sensor[4], recording events at a resolution of 240 × 180; (Berner et al., 2013; Brandli et al., 2014). The sensor uses default bias settings, and recording of DAVIS APS (Active Pixel Sensor) frames, i.e., frame-based intensity read-outs at regular sampling intervals, is deactivated to reduce the dataset sizes. An Ubuntu 14.04 LTS workstation outside of the enclosure controls the video display of the dataset, with a second LCD display for controlling and monitoring the recording. Recording of AER (Address-Event Representation) events, the most commonly used representation of event data, is done with the jAER software[5]. We also developed a Python package called SpikeFuel[6], which is released together with the datasets and is used for displaying and scheduling video sequences, as well as post-processing. SpikeFuel displays frames using OpenCV and controls jAER using local UDP datagrams using jAER's Remote Control protocol.

For each dataset the position of the DAViS240C is adjusted so its field of view covers the region of interest on the monitor, which is a 4:3 aspect ratio window in the center of the screen, surrounded by gray background of 50% intensity. This alignment is achieved

by displaying a flashing green rectangle (Figure 1B). Also, the video sequence is rescaled to fit the size of the field of view of the DAViS240C. To make sure that every frame of the sequence is displayed at least once during the monitor's refreshing period, the video is played at a frame-rate equal or lower than the monitor's refresh rate, in our case at 30 Hz, which is also the original frame rate of the videos. In principle, display at higher rates is possible, but the interplay between GPU rendering and monitor refreshing can become unreliable. The recording of each video starts with an adaptation period of 2 s, in which only the initial frame of the video sequence is displayed. This eliminates unwanted event bursts due to flashing a video on a background screen. Before the playback of the video is started, the jAER timestamps are reset to 0, then the recording is started. At the end of a sequence the recording is saved, while the last frame of the sequence is displayed for 2 s. In post-processing the transition from first to second video frame is detected by the initial burst of DVS activity. For tracking datasets, the bounding box coordinates are transformed to DAViS240C coordinates and supplied with the data along with the corresponding DAViS240C timestamp.

There are a total of 37,410 recordings, representing the largest neuromorphic vision datasets for these domains of machine vision. A software pipeline for capturing frame-based visual recognition benchmarks with neuromorphic cameras was developed. Datasets are delivered in both HDF5 and AEDAT-2.0 AER raw data format (so far there is no HDF5 parser in jAER). We hope that these recordings can boost the development of event-based learning in visual tasks.

In some tracking sequences, the target objects are still, or cannot be differentiated from the background (e.g., rabbit running on snowy ground). And in some action recognition sequences, the

background is rapidly moving. These factors that are introduced by original datasets show that a stationary DVS is not always sufficient for solving dynamic vision applications.

The 30 Hz sample rate of the original recordings aliases information above 15 Hz in the original scene. The artifacts in the DVS output that are caused by the frames in the original datasets show that it is necessary to use neuromorphic sensors for collection of new frame-free datasets that will take full advantage of the precise timing of such sensors, which may be crucial for optical flow computation or event-based stereo (Rogister et al., 2012; Rueckauer and Delbruck, 2016). However, the datasets presented here provide a valuable basis for the development of higher-level algorithms processing and recognizing event-based spatio-temporal patterns, such as in tracking and action recognition applications. By providing common benchmarks for these areas we expect a more solid comparison of the (few) existing approaches, and to aid the development of novel algorithmic ideas.

The fields of Autonomous Driving (AD) and Advanced Driver Assistance Systems (ADAS) have carried out a wide range of studies that can shape the future of transportation by cars and other vehicles. One of the propelling topics fueling this changes is the field of Deep Learning (DL). Deep Learning algorithms are known to be superior in generalizing for several hard conditions that are the source of error in the classical algorithms, mainly changing lighting conditions, road occlusions, shadows, and different camera setups. Because the inclusion of sensors in cars is a current possibility, these algorithms could be used to understand the road scenario.

However, most algorithms perform poorly under some conditions, because the models are not able to learn to perform well in all situations, and datasets still do not reflect all the road scenarios

possible. For example, some datasets use the same camera setup for all images, and do not generalize very well for other camera setups. Similarly, other datasets have images with low diversity of information, for example, datasets with only highway road scenarios do not generalize very well for other scenarios. Also, models trained for different tasks often perform differently for different conditions. A better solution would be to use multiple deep learning algorithms trained for different tasks and datasets, and have an algorithm to combine the outputs of these models into an unique road representation.

This work describes a novel road space representation method that combines two deep learning models trained to perform two different tasks: road segmentation and road lines detection. This method was found suitable and performant for both unstructured and structured roads.

There are several algorithms in the literature for road segmentation and detection of the road lanes. Most classical algorithms are composed of several phases, such as pre-processing, feature extraction and model fitting [1], [2], [3]. Some of these classical algorithms even perform both tasks with the same pipeline [4], [5], [6]. Recently, Deep Learning has proven to be effective in training neural networks in a end-to-end fashion. That is, all the steps mentioned are performed by the neural

In this work, we use the overall architecture conceived in [26] but with major changes regarding the processor algorithms as well as the combination method. Thus, we propose the usage of two deep learning convolutional models trained to perform different tasks, and try to merge their outputs to create a confidence map. This final output, the confidence map, is a grayscale image that represents the road scenario, where each pixel value corresponds to the respective confidence: 1 (white pixels).

This work was developed under the ATLASCAR2 project,[1] which is a Mitsubishi i-MiEV equipped with cameras, LiDARs, and other sensors. In this work, we just used one sensor — one PointGrey Flea3 camera — installed mainly on the roof-top of the car, as can be seen in Fig. 7. This camera acquired the images used in the experiments described in Section 5. Additionally, it is also important to highlight that the two sets of images used to evaluate this work were acquired.

Regarding the experiments, five examples of different road scenarios and the behavior of each model as well as their combination are shown. These experiments assess qualitatively the confidence map creation as well as each model's performance in challenging environments. After that, we show some statistic results for a full-set of 5000 frames in a usual environment for the ATLASCAR2 vehicle. Here, the number of detected regions provided by each model is presented.

One of the most important challenges for Autonomous Driving and Driving Assistance systems is the detection of the road to perform or monitor navigation. Many works can be found in the literature to perform road and lane detection, using both algorithmic processing and learning based techniques. However, no single solution is mentioned to be applicable in any circumstance of mixed scenarios of structured, unstructured, lane based, line based or curb based limits, and other sorts of boundaries. So, one way to embrace this challenge is to have multiple techniques, each specialized on a different approach, and combine them to obtain the best solution from individual contributions. That is the central concern of this paper. By improving a previously developed architecture to combine multiple data sources, a solution is proposed to merge the outputs of two Deep Learning based techniques for road detection. A new representation for the road is proposed along

with a workflow of procedures for the combination of two simultaneous Deep Learning models, based on two adaptations of the ENet model. The results show that the overall solution copes with the alternate failures or under-performances of each model, producing a road detection result that is more reliable than the one given by each approach individually.

The main focus of this study is to develop a system that can accurately detect the presence of fog in real time at a trajectory level. This study leveraged video data from the SHRP 2 Naturalistic Driving Study (NDS). Extensive data reduction steps were taken to classify various levels of foggy weather conditions from the video data to form two unique image data sets. Afterward, features based on the gray level co-occurrence matrix (GLCM) were extracted from the images and used as classification parameters for training support vector machine (SVM) and K-nearest neighbor (K-NN) algorithms. In addition, a convolutional neural network (CNN) was also examined to improve the detection performance. Although the analysis was done initially on a data set consisting of two weather conditions, clear and fog, it has been extended to include different levels of fog, that is, near fog and distant fog. While the accuracy of the first analysis with two categories was approximately 92% and 91% for SVM and K-NN classifiers, respectively, the CNN produced much greater accuracy of 99%. As expected, the accuracy of the second analysis, with more refined weather categories, was relatively lower than the first analysis where CNN, SVM, and K-NN models produced an accuracy of about 98%, 89%, and 88%, respectively. With the rapid advances in connectivity and affordable cameras, the proposed detection models could be integrated into the smartphones of regular road users, creating an effective way to collect real-time road weather information that could be used to improve weather-based variable speed limit (VSL) systems.

This paper proposes a novel system for the automatic detection and recognition of traffic signs. The proposed system detects candidate regions as maximally stable extremal regions (MSERs), which offers robustness to variations in lighting conditions. Recognition is based on a cascade of support vector machine (SVM) classifiers that were trained using histogram of oriented gradient (HOG) features. The training data are generated from synthetic template images that are freely available from an online database; thus, real footage road signs are not required as training data. The proposed system is accurate at high vehicle speeds, operates under a range of weather conditions, runs at an average speed of 20 frames per second, and recognizes all classes of ideogram-based (nontext) traffic symbols from an online road sign database. Comprehensive comparative results to illustrate the performance of the system are presented.

A feature descriptor is a representation of an image or an image patch that simplifies the image by extracting useful information and throwing away extraneous information.

Typically, a feature descriptor converts an image of size width x height x 3 (channels) to a feature vector / array of length n. In the case of the HOG feature descriptor, the input image is of size 64 x 128 x 3 and the output feature vector is of length 3780.

Keep in mind that HOG descriptor can be calculated for other sizes, but in this post I am sticking to numbers presented in the original paper so you can easily understand the concept with one concrete example.

This all sounds good, but what is “useful” and what is “extraneous” ? To define “useful”, we need to know what is it “useful” for ? Clearly, the feature vector is not useful for the purpose of viewing the image. But, it is very useful for tasks like image recognition and object detection. The feature vector

produced by these algorithms when fed into an image classification algorithms like Support Vector Machine (SVM) produce good results.

But, what kinds of “features” are useful for classification tasks ? Let’s discuss this point using an example. Suppose we want to build an object detector that detects buttons of shirts and coats.

A button is circular (may look elliptical in an image) and usually has a few holes for sewing. You can run an edge detector on the image of a button, and easily tell if it is a button by simply looking at the edge image alone. In this case, edge information is “useful” and color information is not. In addition, the features also need to have discriminative power. For example, good features extracted from an image should be able to tell the difference between buttons and other circular objects like coins and car tires.

In the HOG feature descriptor, the distribution (histograms) of directions of gradients (oriented gradients) are used as features. Gradients (x and y derivatives) of an image are useful because the magnitude of gradients is large around edges and corners (regions of abrupt intensity changes) and we know that edges and corners pack in a lot more information about object shape than flat regions.

As mentioned earlier HOG feature descriptor used for pedestrian detection is calculated on a 64×128 patch of an image. Of course, an image may be of any size. Typically patches at multiple scales are analyzed at many image locations. The only constraint is that the patches being analyzed have a fixed aspect ratio. In our case, the patches need to have an aspect ratio of 1:2. For example, they can be 100×200, 128×256, or 1000×2000 but not 101×205.

To illustrate this point I have shown a large image of size 720×475. We have selected a patch of size 100×200 for

calculating our HOG feature descriptor. This patch is cropped out of an image and resized to 64×128. Now we are ready to calculate the HOG descriptor for this image patch.

In this step, the image is divided into 8×8 cells and a histogram of gradients is calculated for each 8×8 cells.

We will learn about the histograms in a moment, but before we go there let us first understand why we have divided the image into 8×8 cells. One of the important reasons to use a feature descriptor to describe a patch of an image is that it provides a compact representation. An 8×8 image patch contains 8x8x3 = 192 pixel values. The gradient of this patch contains 2 values (magnitude and direction) per pixel which adds up to 8x8x2 = 128 numbers.

By the end of this section we will see how these 128 numbers are represented using a 9-bin histogram which can be stored as an array of 9 numbers. Not only is the representation more compact, calculating a histogram over a patch makes this represenation more robust to noise. Individual graidents may have noise, but a histogram over 8×8 patch makes the representation much less sensitive to noise.

But why 8×8 patch ? Why not 32×32 ? It is a design choice informed by the scale of features we are looking for. HOG was used for pedestrian detection initially. 8×8 cells in a photo of a pedestrian scaled to 64×128 are big enough to capture interesting features (e.g. the face, the top of the head etc.).

If you are a beginner in computer vision, the image in the center is very informative. It shows the patch of the image overlaid with arrows showing the gradient — the arrow shows the direction of gradient and its length shows the magnitude. Notice how the direction of arrows points to the direction of change in intensity and the magnitude shows how big the difference is.

On the right, we see the raw numbers representing the gradients in the 8×8 cells with one minor difference — the angles are between 0 and 180 degrees instead of 0 to 360 degrees. These are called **"unsigned" gradients** because a gradient and it's negative are represented by the same numbers. In other words, a gradient arrow and the one 180 degrees opposite to it are considered the same. But, why not use the 0 – 360 degrees ?

Empirically it has been shown that unsigned gradients work better than signed gradients for pedestrian detection. Some implementations of HOG will allow you to specify if you want to use signed gradients.

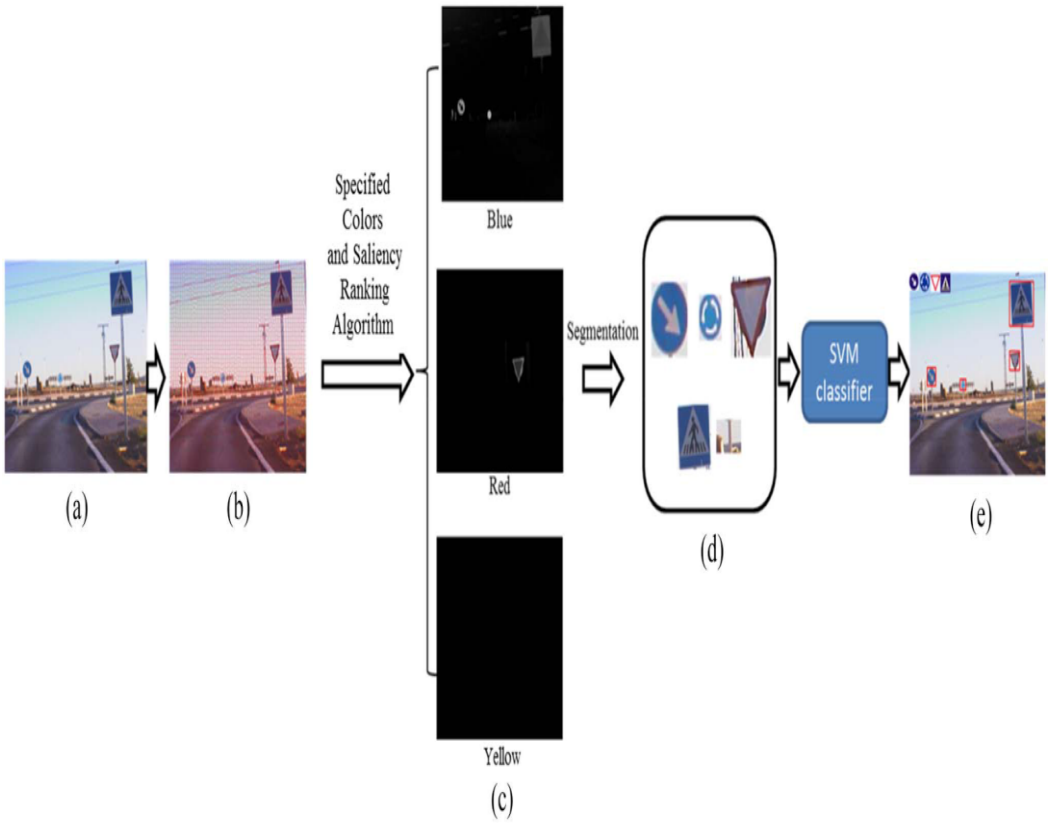


Fig 3 : To detect the traffic sign using OpenCV algorithm

original RGB vector. You can see that normalizing a vector removes the scale.

Now that we know how to normalize a vector, you may be tempted to think that while calculating HOG you can simply normalize the 9×1 histogram the same way we normalized the 3×1 vector above. It is not a bad idea, but a better idea is to normalize over a bigger sized block of 16×16.

A 16×16 block has 4 histograms which can be concatenated to form a 36 x 1 element vector and it can be normalized just the way a 3×1 vector is normalized. The window is then moved by 8 pixels (see animation) and a normalized 36×1 vector is calculated over this window and the process is repeated.

To calculate the final feature vector for the entire image patch, the 36×1 vectors are concatenated into one giant vector. What is the size of this vector ? Let us calculate

How many positions of the 16×16 blocks do we have ? There are 7 horizontal and 15 vertical positions making a total of 7 x 15 = 105 positions.

The HOG descriptor of an image patch is usually visualized by plotting the 9×1 normalized histograms in the 8×8 cells. See image on the side. You will notice that dominant direction of the histogram captures the shape of the person, especially around the torso and legs.

Unfortunately, there is no easy way to visualize the HOG descriptor in OpenCV.

According to official statistics, about 400 road accidents occur in India every day. Road signs help to avoid accidents on the road, ensuring the safety of both drivers and pedestrians. Additionally, traffic signals guarantee that road users adhere to specific laws, minimizing the likelihood of traffic violations. Route navigation is

also made easier by the use of traffic signals. Road signals should be prioritized by all road users, whether they are drivers or pedestrians. We overlook traffic signs for a variety of reasons such as problems with concentration, exhaustion, and sleep

deprivation. Other causes that contribute to missing the signs include poor vision, the influence of the external world, and environmental circumstances. It is much more important to use a system that can recognize traffic signals and advise and warn the driver. Image-based traffic-sign recognition technologies analyze images captured by a car's front-facing camera in real time to recognize signals. They help the driver by giving him or her warnings. The identification and recognition modules are the key components of a vision- based traffic sign recognition system. The detection module locates the sign area in the image/video, while the recognition module recognizes the sign. The sign regions with the highest probability are selected and fed into the recognition system to classify the sign during the detection process .For traffic sign recognition, various machine learning algorithms such as SVM, KNN, and Random Forest can be used [6]. However, the key disadvantage of these algorithms is that feature extraction must be done separately; on the other hand, CNN will do feature extraction on its own [1] .As a result, the proposed system employs a convolutional neural network. Input preprocessing module will prepare image captured with the help of vehicle camera for recognition stage before that. The driver will get a voice warning message after recognition.

A series of warnings about the route are conveyed by traffic signs. They keep traffic going by aiding travelers in reaching their destinations and providing them with advance notice of arrival, exit, and turn points. Road signs are placed in specific positions to ensure the safety of travelers. They also have guidance for when and where drivers can turn or not turn. In this paper, we

proposed a system for traffic sign detection and recognition, as well as a method for extracting a road sign from a natural complex image, processing it, and alerting the driver through voice command. It is applied in such a way that it helps drivers make fast decisions. In real-time situations, factors like shifting weather conditions, changing light directions, and varying light intensity make traffic sign identification challenging. The reliability of the machine is influenced by a number of factors such as noise, partial or absolute underexposure, partial or complete overexposure, and significant variations in color saturation, wide variety of viewing angles, view depth, and shape/color deformations of traffic signs (due to light intensity).The proposed architecture is sectioned into three phases .The first of which is image pre-processing, in which we quantify the dataset's input files, determine the input size for learning purposes, and resize the information for the learning step. The proposed algorithm categorizes the observed symbol during the recognition process. A Convolutional Neural Network is used to do this in the second phase, and the third phase deals with text-to-speech translation, with the detected sign from the second phase being presented in audio format.

In any kind of study, the most critical move is to do a literature review. This move would allow us to identify any gaps or flaws in the current structure which will attempt to find a way to get around the limitations of the current method. We briefly discuss similar work on traffic sign detection identification and recognition in this segment

The detection and recognition of traffic sign is an important part of an advanced driver assistance system and a great issue for research recently and plays a great role in intelligent transport system. Traffic signs are used to regulate the traffic and to indicate the state of the road to guide and warn drivers and

pedestrians. They have several distinguishing features like specific colors and shapes, with the text or symbol in high contrast to the background that may be used for their detection and identification. The visibility of traffic sign is very important for the drivers safety. There are many problems involved in traffic-sign detection like variations in perspective, variations in illumination, occlusion of signs, motion blur, and weatherworn deterioration of signs. These different conditions are brought about by changes in the weather such as sunny, cloudy etc, the time of the day or night, and the state of the road sign itself subject to deterioration.

The proposed method consists of the two stages: detection and recognition. The first is detection of traffic sign in image using image processing. The detection is performed using a novel application of maximally stable extremal regions (MSERs) [1]. The second one is recognition of detected sign.

The recognition is performed with histogram of oriented gradient (HOG) features, which are classified using support vector machine (SVM). The result of the TSDR research effort can be used as a support system for the driver. Traffic Sign detection and recognition can also be useful for highway maintenance so that a human operator does not have to check the presence and condition of the signs. The goal of this work is to find and classify road signs in various illuminations and different weather conditions such as rainy, sunny, cloudy, etc.

In most cases, traffic sign recognition algorithms are divided into two stages: 1) detection and 2) recognition. Traffic signs have varying illumination in various weather conditions. Many of system use the color information as method of segmenting the image. But the color-based road sign detection methods gives often reduced performance in scenes with poor lighting or adverse weather conditions such as fog. To overcome this issues the RGB

image is converted into different Color spaces, such as huesaturation-value HSV [6], YUV (luma Y and two chrominance UV) [12], and CIECAM97 (color appearance model) [7]. The method presenting a system for detection and recognition of road signs with red boundaries and black symbols inside is represented in [6]. The detection is invariant to varying lighting conditions and shadows. The detection technique [12] includes histogram equalization, color segmentation and light control. In this system, both the HSV and the YUV color spaces are used to get better segmentation results than that of one color space technique. Gao et al. [7] used a quad-tree histogram method to segment the image based on the hue and chroma values of the CIECAM97 color model.

In contrast, there are several approaches that use only shape information instead of color information from grayscale images. Gareth Loy [11] introduced a shape based technique that extends the concept of the fast radial symmetry transform to detect triangular, square and octagonal road signs. The system proposed by S. Maldonado Bascon et al. consists of color based segmentation, traffic sign detection by

shape classification with linear SVMs and recognition based on Gaussian-kernel SVMs [5]. E. Cardarelli et al. presented the recognition system based on neural network [4]. These both systems consist of classifiers that were trained using hand labelled real images, which is repetitive, time

consuming and error-prone process. But the system can overcome these issues using synthetic graphical representations of signs from an online road sign database. There is another novel system for the automatic detection and recognition of traffic signs [1].

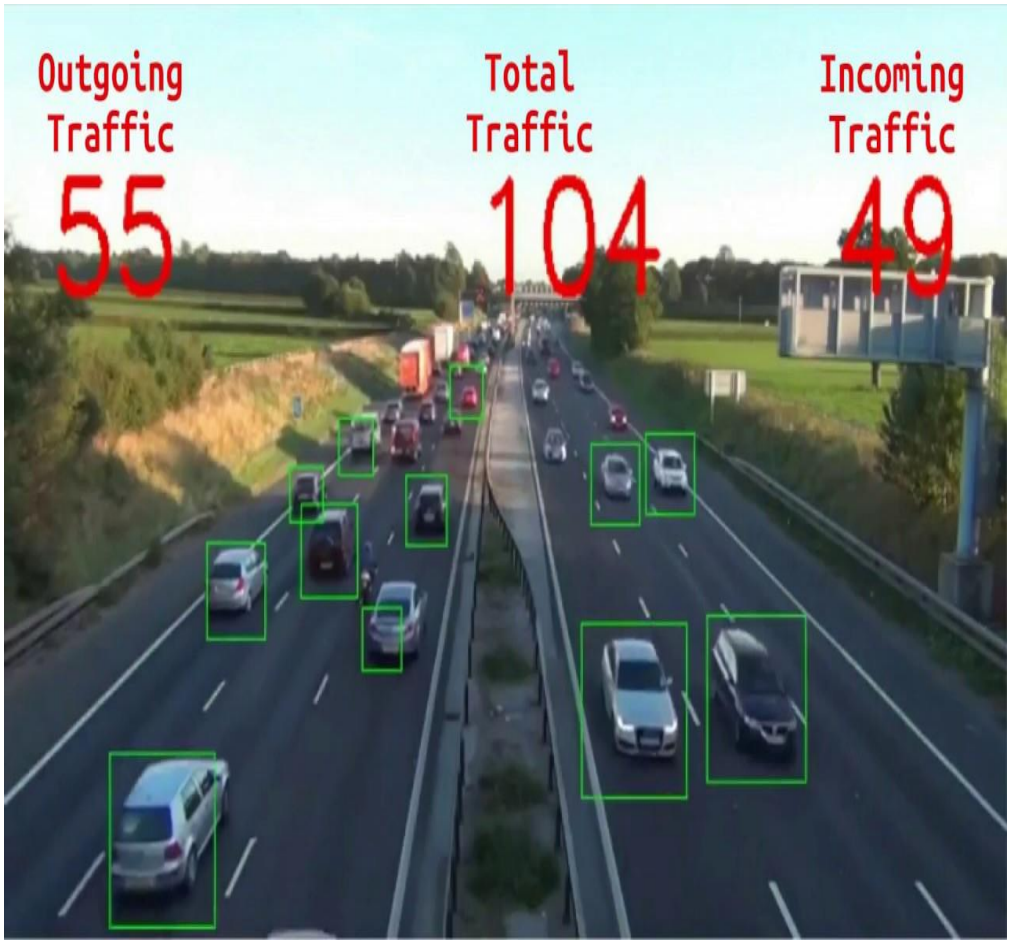


Fig 4 : Automated traffic management system using OpenCV algorithm

CHAPTER 5

SYSTEM TESTING

System testing is an important aspect of traffic sign detection using OpenCV and Python. In order to test the system, the following steps can be followed:

5.1 Dataset selection: A suitable dataset of traffic sign images and videos should be selected for testing the system. The dataset should be representative of the real-world scenarios that the system will be used in.

5.1.1 Data preparation: The dataset should be preprocessed and prepared for testing. This includes tasks such as resizing, normalization, and labeling.

5.1.2 System evaluation: The system should be tested on the prepared dataset, and its performance should be evaluated in terms of accuracy, precision, recall, and F1 score. These metrics can be calculated using the confusion matrix, which compares the actual and predicted traffic sign labels.

5.1.3 Performance optimization: Based on the results of the system evaluation, the system can be further optimized to improve its performance. This may involve tuning the system parameters, using more advanced image processing techniques or machine learning algorithms, or increasing the size and diversity of the training dataset.

5.1.4 Deployment testing: Once the system has been optimized, it can be tested in real-world scenarios to ensure its reliability and effectiveness. This may involve testing the system on different types of traffic signs, lighting conditions, and weather

conditions, and evaluating its performance in terms of accuracy and speed.

Overall, system testing is essential for ensuring that the traffic sign detection system using OpenCV and Python is accurate, reliable, and effective in real-world scenarios.

5.2 WHY CHOOSE PYTHON

If you're going to write programs, there are literally dozens of commonly used languages to choose from. Why choose Python? Here are some of the features that make Python an appealing choice.

5.2.1 Python is Popular:

Python has been growing in popularity over the last few years. The 2018 Stack Overflow Developer Survey ranked Python as the 7th most popular and the number one most wanted technology of the year. World-class software development countries around the globe use Python every single day.

According to research by Dice Python is also one of the hottest skills to have and the most popular programming language in the world based on the Popularity of Programming Language Index.

Due to the popularity and widespread use of Python as a programming language, Python developers are sought after and paid well. If you'd like to dig deeper into Python salary statistics and job opportunities, you can do so here.

5.2.2 Python is interpreted:

Many languages are compiled, meaning the source code you create needs to be translated into machine code, the language of your computer's processor, before it can be run. Programs written in an interpreted language are passed straight to an interpreter that runs them directly.

This makes for a quicker development cycle because you just type in your code and run it, without the intermediate compilation step.

One potential downside to interpreted languages is execution speed. Programs that are compiled into the native language of the computer processor tend to run more quickly than interpreted programs. For some applications that are particularly computationally intensive, like graphics processing or intense number crunching, this can be limiting.

In practice, however, for most programs, the difference in execution speed is measured in milliseconds, or seconds at most, and not appreciably noticeable to a human user. The expediency of coding in an interpreted language is typically worth it for most applications.

5.2.3 Python is Free:

The Python interpreter is developed under an OSI-approved open-source license, making it free to install, use, and distribute, even for commercial purposes.

A version of the interpreter is available for virtually any platform there is, including all flavors of Unix, Windows, macOS, smart phones and tablets, and probably anything else you ever heard of. A version even exists for the half dozen people remaining who use OS/2.

5.2.4 Python is Portable:

Because Python code is interpreted and not compiled into native machine instructions, code written for one platform will work on any other platform that has the Python interpreter installed. (This is true of any interpreted language, not just Python..

5.2.5 Python is simple:

As programming languages go, Python is relatively uncluttered, and the developers have deliberately kept it that way.

A rough estimate of the complexity of a language can be gleaned from the number of keywords or reserved words in the language. These are words that are reserved for special meaning by the compiler or interpreter because they designate specific built-in functionality of the language.

Python 3 has 33 keywords, and Python 2 has 31. By contrast, C++ has 62, Java has 53, and Visual Basic has more than 120, though these latter examples probably vary somewhat by implementation or dialect.

Python code has a simple and clean structure that is easy to learn and easy to read. In fact, as you will see, the language definition enforces code structure that is easy to read.

But It's Not That Simple

For all its syntactical simplicity, Python supports most constructs that would be expected in a very high-level language, including complex dynamic data types, structured and functional programming, and object-oriented programming.

Additionally, a very extensive library of classes and functions is available that provides capability well beyond what is built into the language, such as database manipulation or GUI programming.

Python accomplishes what many programming languages don't: the language itself is simply designed, but it is very versatile in terms of what you can accomplish with it.

5.2.6 Conclusion:

This section gave an overview of the Python programming language, including:

•A brief history of the development of Python

•Some reasons why you might select Python as your language of choice

Python is a great option, whether you are a beginning programmer looking to learn the basics, an experienced programmer designing a large application, or anywhere in between. The basics of Python are easily grasped, and yet its capabilities are vast. Proceed to the next section to learn how to acquire and install Python on your computer.

Python is an open source programming language that was made to be easy-to-read and powerful.
A Dutch programmer named Guido van Rossum made Python in 1991. He named it after the television show Monty Python's Flying Circus. Many Python examples and tutorials include jokes from the show.

Python is an interpreted language. Interpreted languages do not need to be compiled to run. A program called an interpreter runs Python code on almost any kind of computer. This means that a programmer can change the code and quickly see the results. This also means Python is slower than a compiled language like C, because it is not running machine code directly.

Python is a good programming language for beginners. It is a high-level language, which means a programmer can focus on what to do instead of how to do it. Writing programs in Python takes less time than in some other languages.

Python drew inspiration from other programming languages like C, C++, Java, Perl, and Lisp.

Python has a very easy-to-read syntax. Some of Python's syntax comes from C, because that is the language that Python was written in. But Python uses whitespace to delimit code: spaces or tabs are used to organize code into groups. This is different from C. In C, there is a semicolon at the end of each line and curly braces ({}. are used to group code. Using whitespace to delimit code makes Python a very easy-to-read language.

Python use [change / change source]

Python is used by hundreds of thousands of programmers and is used in many

places. Sometimes only Python code is used for a program, but most of the time it is used to do simple jobs while another programming language is used to do more complicated tasks.

Its standard library is made up of many functions that come with Python when it is installed. On the Internet there are many other libraries available that make it possible for the Python language to do more things. These libraries make it a powerful language; it can do many different things.

Some things that Python is often used for are:

•Web development

•Scientific programming

•Desktop GUIs

CHAPTER 6

DATASET EXPLANATION

In order to train and test a traffic sign detection system using OpenCV and Python, a suitable dataset of traffic sign images and videos is required. The dataset should be representative of the real-world scenarios that the system will be used in, and should include a diverse range of traffic signs, lighting conditions, and weather conditions.

There are several publicly available datasets that can be used for traffic sign detection, such as the German Traffic Sign Recognition Benchmark (GTSRB. dataset, the Belgian Traffic Sign Dataset (BTSD., and the LISA Traffic Sign Dataset. These datasets typically include thousands of images and videos of traffic signs, with each image labeled with the corresponding traffic sign class.

The GTSRB dataset, for example, includes 43 different traffic sign classes, with a total of 39,209 images in the training set and 12,630 images in the test set. The images are captured under different lighting conditions and weather conditions, and include a range of sign sizes and orientations.

Other datasets, such as the LISA Traffic Sign Dataset, include additional metadata such as GPS location and time of day, which can be useful for testing the system under different scenarios.

When selecting a dataset for traffic sign detection, it is important to ensure that the dataset is representative of the real-world scenarios that the system will be used in. Additionally, the dataset should include a diverse range of traffic sign classes, sizes, and orientations, and should be large enough to provide sufficient training and testing data for the system. The image dataset is consists of more than 50,000 pictures of various traffic signs(speed limit, crossing, traffic signals, etc.) Around 43

different classes are present in the dataset for image classification. The dataset classes vary in size like some class has very few images while others have a vast number of images. The dataset doesn't take much time and space to download as the file size is around 314.36 MB. It contains two separate folders, train and test, where the train folder is consists of classes, and every category contains various images.

Fig 5 : Dataset explanation of traffic signs

CHAPTER 7

PROPOSED SYSTEM

The proposed system for traffic sign detection using OpenCV and Python involves several steps:

7.1 Dataset preparation: A suitable dataset of traffic sign images and videos is selected, and the images and videos are labeled with the corresponding traffic sign class.

7.2 Image preprocessing: The input images are preprocessed to enhance their quality and make them suitable for traffic sign detection. This may involve tasks such as resizing, color conversion, and noise reduction.

7.3 Traffic sign detection: The preprocessed images are analyzed using OpenCV and machine learning algorithms to detect the presence of traffic signs. This may involve tasks such as object detection, feature extraction, and classification.

7.4 Post-processing: Once the traffic signs have been detected, additional processing may be required to refine the results and remove false positives. This may involve tasks such as non-maximum suppression and thresholding.

7.5 Visualization: Finally, the results of the traffic sign detection system are visualized to make them easy to interpret and understand. This may involve tasks such as bounding box visualization and label assignment.

Overall, the proposed system for traffic sign detection using OpenCV and Python aims to provide a reliable and accurate method for detecting traffic signs in real-world scenarios. By leveraging machine learning and computer vision techniques, the system can analyze images and videos in real-time and provide useful information to drivers and autonomous vehicles.

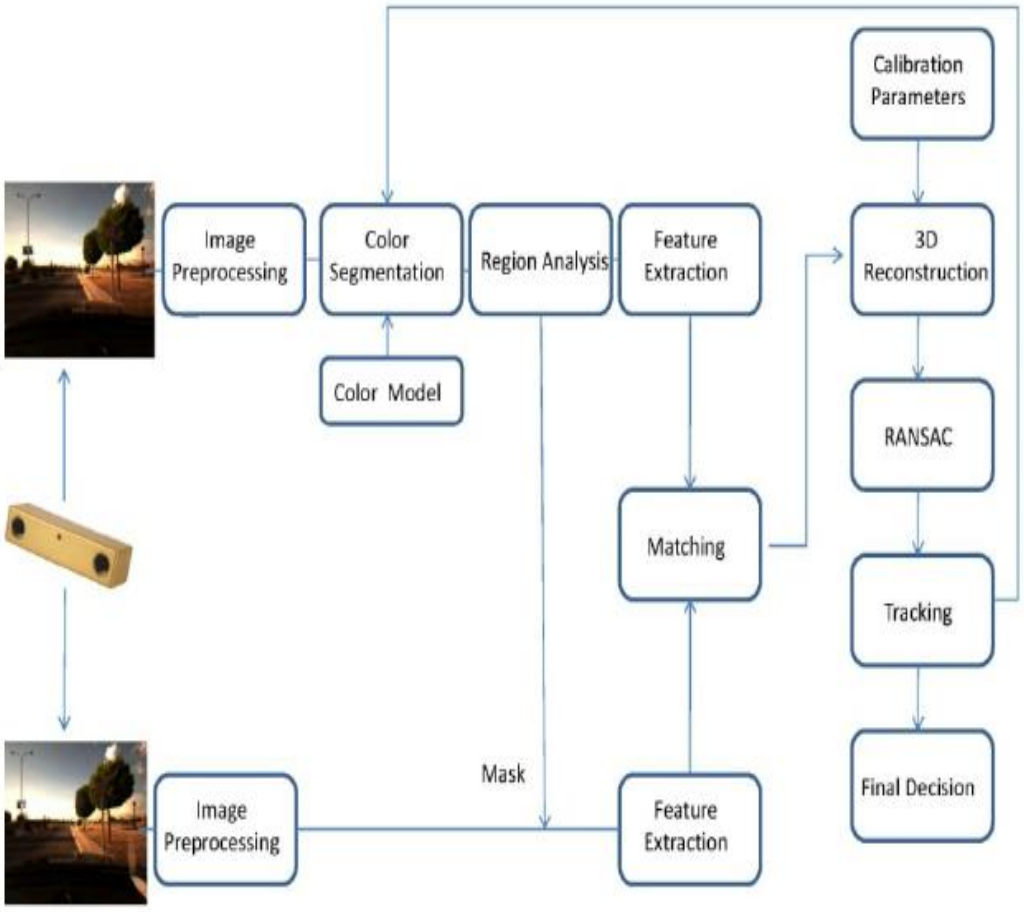


Fig 6 : Block diagram of the proposed traffic sign detection system

CHAPTER 8

PROBLEM STATEMENT

The problem statement for traffic sign detection using OpenCV and Python is to develop a reliable and accurate system for detecting traffic signs in real-world scenarios. The system must be able to analyze images and videos in real-time and provide useful information to drivers and autonomous vehicles. This requires overcoming several challenges, such as varying lighting conditions, occlusion, and variations in traffic sign appearance due to weathering or damage. Additionally, the system must be able to handle a large variety of traffic sign types, including speed limit signs, stop signs, yield signs, and more. Finally, the system must be efficient and computationally lightweight to operate in real-time and meet the requirements of modern traffic monitoring systems.

Robust and well-arranged system is important to cover and find out the attention of drivers during driving

over the streets and roads safety, the Traffic Sign Detection and Recognition (TSDR) system has been initiated, especially the traffic signs are not in fine corner or it is faced damaged by external factors. After opening the camera of the mobile in real-time, the video sequences under different conditions (lighting,

weather and climate, velocity and quickness, etc.), below factors are predictable:

Determine a method to extract the traffic signs through driving car in real time by video sequencing the frames to be segmented.

Coming up with a method to clean the extracted the traffic sign from noise and distortion.

Coming up with a method to find out the region of interest and identify the traffic sign with alarming notification.

Coming up with a method to find out the speed meter with alarming notification when the car drives over speed.

Fig 7 : Sample traffic signs

CHAPTER 9

DATA PREPROCESSING

Data preprocessing is an essential step in traffic sign detection using OpenCV and Python. It involves several tasks that aim to enhance the quality of the input images and make them suitable for traffic sign detection. The following are the main data preprocessing tasks:

9.1 Image resizing: The input images are resized to a fixed size to ensure that they have a consistent resolution and aspect ratio. This is important for feature extraction and classification tasks.

9.2 Color space conversion: The input images are converted to a suitable color space for traffic sign detection. For example, the HSV color space may be used to isolate the color of the traffic sign and improve its detection.

9.3 Noise reduction: The input images may contain noise, which can interfere with traffic sign detection. Therefore, noise reduction techniques such as median filtering or Gaussian blurring may be applied to the images to improve their quality.

9.4 Contrast adjustment: The input images may have low contrast, making it difficult to detect traffic signs. Therefore, contrast adjustment techniques such as histogram equalization or contrast stretching may be applied to enhance the image contrast.

9.5 Image normalization: The input images may have variations in brightness and contrast due to differences in lighting conditions. Therefore, image normalization techniques such as local contrast normalization or Z-score normalization may be

applied to make the images more uniform and suitable for feature extraction and classification.

Overall, data preprocessing plays a critical role in traffic sign detection using OpenCV and Python. By enhancing the quality of the input images, the system can improve its accuracy and reliability in detecting traffic signs.

9.6 Advantages:

The system uses the open-cv algorithm to detect signs, which is a very fast and accurate algorithm if trained properly. This enables the detection on embedded devices possible where low computing power is available.

CHAPTER 10

MODULES/LIBRARIES

10.1 OPENCV:

OpenCV (Open Source Computer Vision Library. is an open-source computer vision and machine learning software library that provides a wide range of algorithms and tools for image and video processing, object detection and recognition, and other computer vision tasks. It was originally developed by Intel and is now maintained by the OpenCV Foundation.

OpenCV provides a collection of functions and classes for image and video processing, including image filtering, feature detection and extraction, object tracking, and more. It also includes support for various input and output formats, such as image and video files, cameras, and streaming protocols.

OpenCV is written in C++ and provides interfaces for several programming languages, including Python, Java, and MATLAB. The Python interface is particularly popular due to its ease of use and rich ecosystem of scientific computing libraries.

OpenCV has numerous applications in various fields, including robotics, autonomous vehicles, surveillance, medical imaging, and more. It is widely used in industry and academia and has a large community of developers and contributors.

10.2 IMUTILS:

Imutils is a Python library that provides a set of convenience functions to make basic image processing tasks easier using OpenCV. It is designed to supplement OpenCV's capabilities and simplify common image processing tasks such as resizing, rotation, translation, and more.

Imutils contains various image processing functions and utilities that can be used to perform operations on images such as resizing, rotating, cropping, translation, and more. It also contains functions to convert between different color spaces and to apply various image filters such as blurring, thresholding, and edge detection.

Imutils is a useful tool for anyone working with OpenCV in Python, as it can greatly simplify common image processing tasks and save time. It is also easy to use and well-documented, making it accessible for both beginners and experienced programmers.

10.3 Numpy:

Numpy is a popular Python library used for scientific computing and data analysis. It provides support for large, multi-dimensional arrays and matrices, as well as a range of mathematical functions for working with these structures. Numpy is widely used in fields such as physics, astronomy, biology, engineering, finance, and machine learning.

Some of the key features of numpy include:

10.3.1 Multi-dimensional arrays: Numpy provides support for arrays with any number of dimensions, making it easy to work with data of different shapes and sizes.

10.3.2 Mathematical functions: Numpy includes a large number of mathematical functions, including basic arithmetic operations, trigonometric functions, logarithmic and exponential functions, and more.

10.3.3 Broadcasting: Numpy allows operations to be performed on arrays of different shapes and sizes, automatically adjusting the dimensions as necessary.

10.3.4 Indexing and slicing: Numpy provides a powerful indexing and slicing syntax for working with arrays, making it easy to select and manipulate subsets of data.

Integration with other libraries: Numpy integrates seamlessly with other popular Python libraries such as Pandas, Matplotlib, and Scikit-learn, making it a key component in many data analysis workflows.

Overall, numpy is a powerful tool for working with large datasets and performing complex calculations in Python.

CHAPTER 11

CODE IMPLIMENTATION

```
import cv2
import numpy as np
import time
from imutils.perspective import four_point_transform
#from imutils import contours
import imutils

camera = cv2.VideoCapture(0)

def findTrafficSign( )
    '''
    This function find blobs with blue color on the image.
    After blobs were found it detects the largest square blob, that must be the sign.
    '''
    # define range HSV for blue color of the traffic sign
    lower_blue = np.array([85,100,70].
    upper_blue = np.array([115,255,255].

    while True:
        # grab the current frame
        (grabbed, frame. = camera.read( )

        if not grabbed:
            print("No input image")
            break

        frame = imutils.resize(frame, width=500)
```

```
        frameArea = frame.shape[0]*frame.shape[1]

        # convert color image to HSV color scheme
        hsv = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)

        # define kernel for smoothing
        kernel = np.ones((3,3.,np.uint8)
        # extract binary image with active blue regions
        mask = cv2.inRange(hsv, lower_blue, upper_blue)
        # morphological operations
        mask = cv2.morphologyEx(mask, cv2.MORPH_OPEN, kernel)
        mask = cv2.morphologyEx(mask, cv2.MORPH_CLOSE, kernel)

        # find contours in the mask
        cnts = cv2.findContours(mask.copy(., cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_SIMPLE.[-2]))

        # defite string variable to hold detected sign description
        detectedTrafficSign = None

        # define variables to hold values during loop
        largestArea = 0
        largestRect = None

        # only proceed if at least one contour was found
        if len(cnts. > 0):
            for cnt in cnts:
                # Rotated Rectangle. Here, bounding rectangle is drawn with
minimum area,
                # so it considers the rotation also. The function used is
cv2.minAreaRect()
```

```
        # It returns a Box2D structure which contains following detals -
        # ( center (x,y., (width, height., angle of rotation))
        # But to draw this rectangle, we need 4 corners of the rectangle.
        # It is obtained by the function cv2.boxPoints(.)
        rect = cv2.minAreaRect(cnt)
        box = cv2.boxPoints(rect)
        box = np.int0(box)

        # count euclidian distance for each side of the rectangle
        sideOne = np.linalg.norm(box[0]-box[1])
        sideTwo = np.linalg.norm(box[0]-box[3])
        # count area of the rectangle
        area = sideOne*sideTwo
        # find the largest rectangle within all contours
        if area > largestArea:
            largestArea = area
            largestRect = box


 # draw contour of the found rectangle on  the original image
 if largestArea > frameArea*0.02:
     cv2.drawContours(frame,[largestRect],0,(0,0,255.,2)



#if largestRect is not None:
     # cut and warp interesting area
     warped = four_point_transform(mask, [largestRect][0])

     # show an image if rectangle was found
```

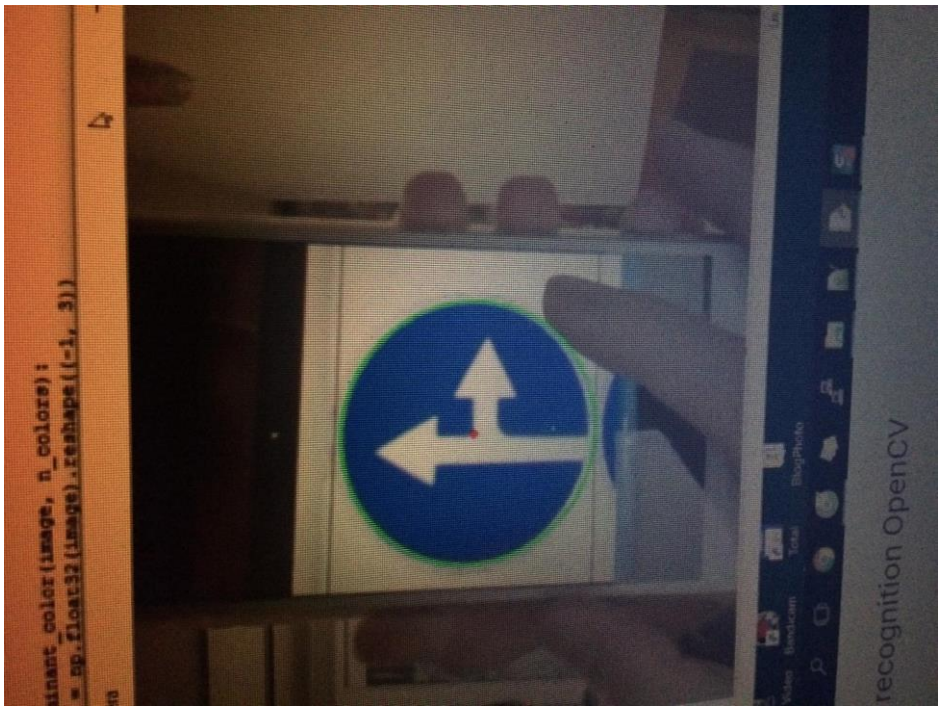
```
            #cv2.imshow("Warped", cv2.bitwise_not(warped))

            # use function to detect the sign on the found rectangle
            detectedTrafficSign = identifyTrafficSign(warped)
            #print(detectedTrafficSign.

            # write the description of the sign on the original image
            cv2.putText(frame, detectedTrafficSign, tuple(largestRect[0].),
cv2.FONT_HERSHEY_SIMPLEX, 0.65, (0, 255, 0., 2)

        # show original image
        cv2.imshow("Original", frame)

        # if the `q` key was pressed, break from the loop
        if cv2.waitKey(1. & 0xFF is ord('q'):
            cv2.destroyAllWindows( )
            print("Stop programm and close all windows")
            break

def identifyTrafficSign(image):
    '''
    In this function we select some ROI in which we expect to have the sign
parts. If the ROI has more active pixels than threshold we mark it as 1, else
0
    After path through all four regions, we compare the tuple of ones and
zeros with keys in dictionary SIGNS_LOOKUP
    '''

    # define the dictionary of signs segments so we can identify
    # each signs on the image
```

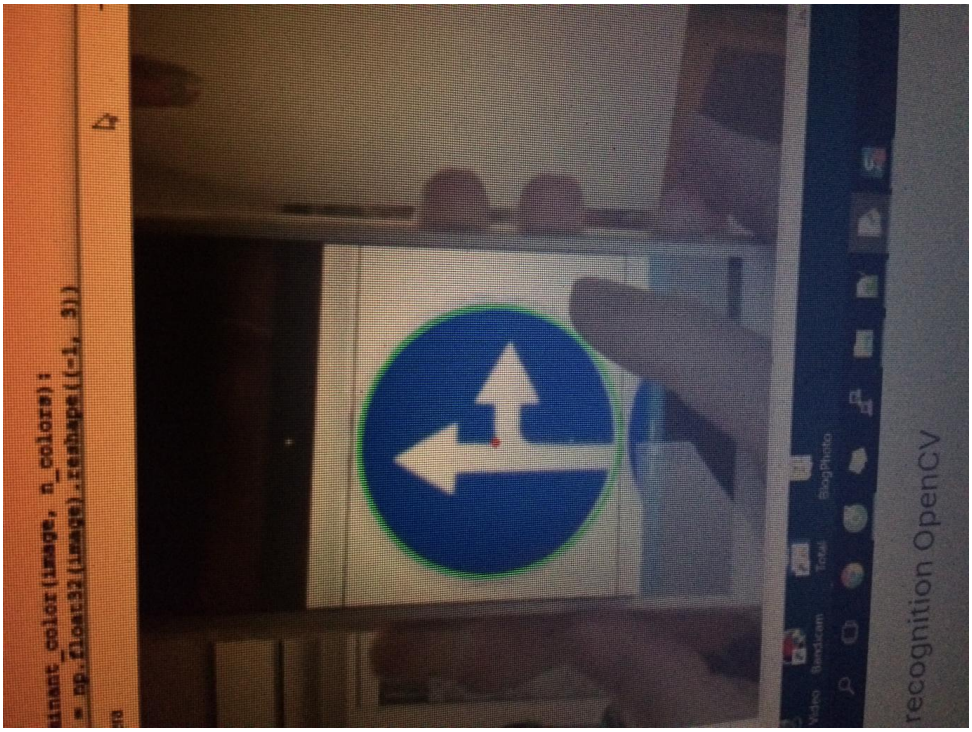
```
SIGNS_LOOKUP = {
    (1, 0, 0, 1): 'Turn Right', # turnRight
    (0, 0, 1, 1): 'Turn Left', # turnLeft
    (0, 1, 0, 1): 'Move Straight', # moveStraight
    (1, 0, 1, 1): 'Turn Back', # turnBack
}

THRESHOLD = 150

image = cv2.bitwise_not(image)
# (roiH, roiW) = roi.shape
#subHeight = thresh.shape[0]/10
#subWidth = thresh.shape[1]/10
(subHeight, subWidth. = np.divide(image.shape, 10)
subHeight = int(subHeight.
subWidth = int(subWidth)

# mark the ROIs borders on the image
cv2.rectangle(image, (subWidth, 4*subHeight., (3*subWidth,
9*subHeight))), (0,255,0.,2) # left block
cv2.rectangle(image, (4*subWidth, 4*subHeight., (6*subWidth,
9*subHeight))), (0,255,0.,2) # center block
cv2.rectangle(image, (7*subWidth, 4*subHeight., (9*subWidth,
9*subHeight))), (0,255,0.,2) # right block
cv2.rectangle(image, (3*subWidth, 2*subHeight., (7*subWidth,
4*subHeight))), (0,255,0.,2) # top block

# substract 4 ROI of the sign thresh image
leftBlock = image[4*subHeight:9*subHeight, subWidth:3*subWidth]
centerBlock = image[4*subHeight:9*subHeight, 4*subWidth:6*subWidth]
rightBlock = image[4*subHeight:9*subHeight, 7*subWidth:9*subWidth]
```

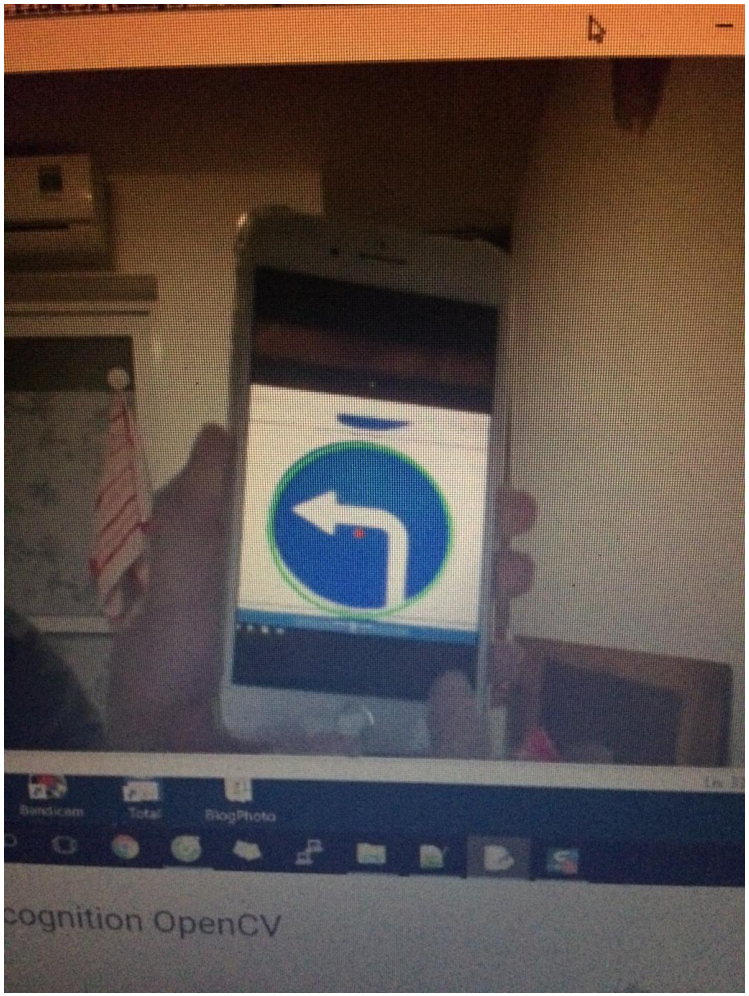
```
    topBlock = image[2*subHeight:4*subHeight, 3*subWidth:7*subWidth]


    # we now track the fraction of each ROI
    leftFraction = np.sum(leftBlock./(leftBlock.shape[0]*leftBlock.shape[1]))
    centerFraction =
np.sum(centerBlock./(centerBlock.shape[0]*centerBlock.shape[1]))
    rightFraction =
np.sum(rightBlock./(rightBlock.shape[0]*rightBlock.shape[1]))
    topFraction = np.sum(topBlock./(topBlock.shape[0]*topBlock.shape[1]))


    segments = (leftFraction, centerFraction, rightFraction, topFraction)
    segments = tuple(1 if segment > THRESHOLD else 0 for segment in
segments)


    cv2.imshow("Warped", image)


    if segments in SIGNS_LOOKUP:
        return SIGNS_LOOKUP[segments]
    else:
        return None



def main():
    findTrafficSign()



if _name_ == '_main_':
main()
```

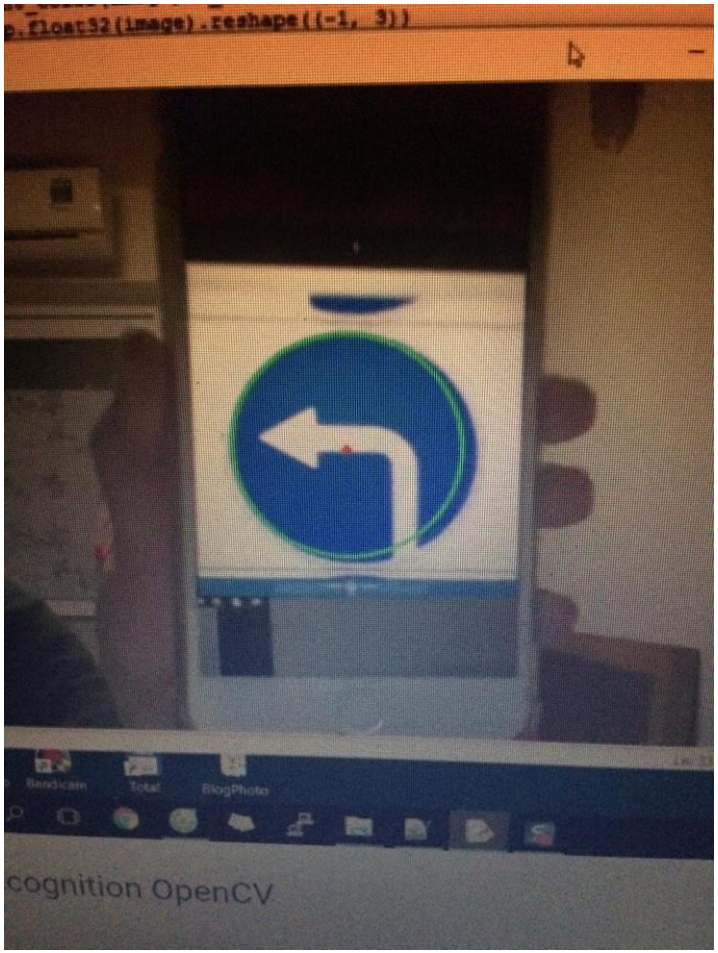
CHAPTER 12

RESULTS



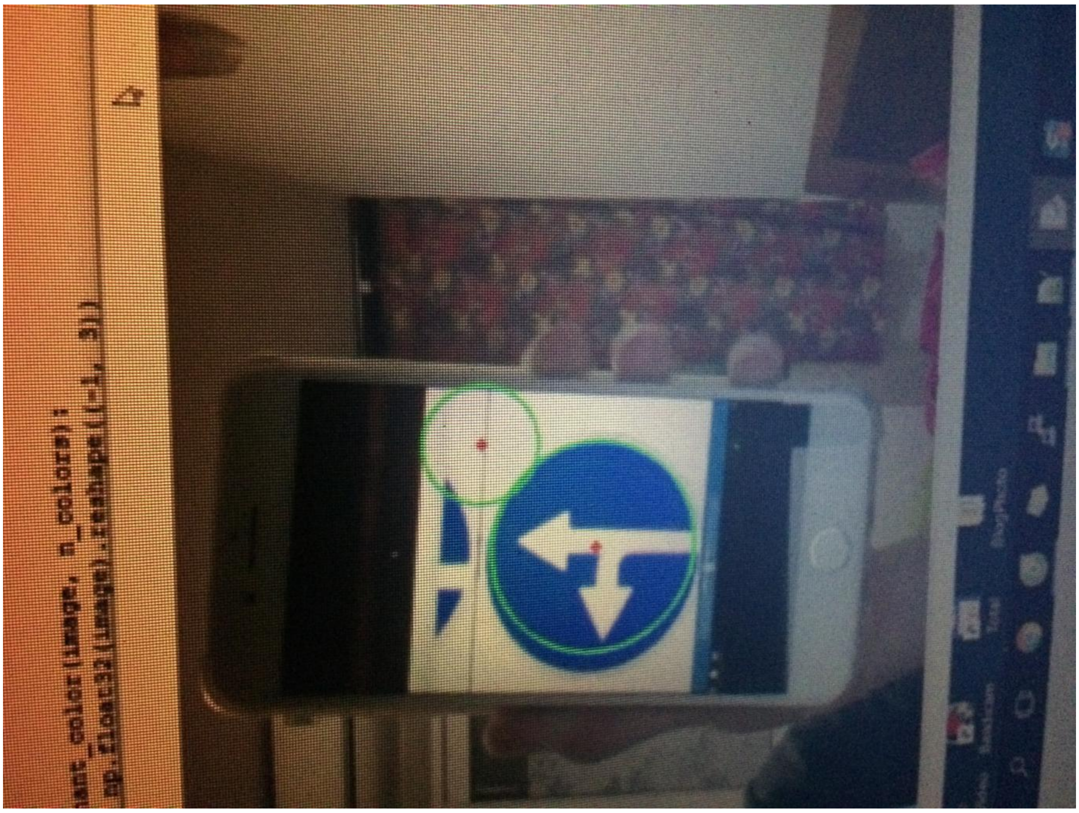



Bandicam
Total
BlogPhoto
cognition OpenCV


.reshape((-1, 3))
Bandicam
Total
BlogPhoto
cognition OpenCV

Cognition OpenCV

Cognition OpenCV


ant_color(image, n_colors):
np.float32(image).reshape((-1, 3))

CHAPTER 13

CONCLUSION

In conclusion, traffic sign detection using OpenCV in Python is a valuable tool for improving road safety by automatically identifying traffic signs in images or video streams. OpenCV provides a wide range of image processing and computer vision functions that can be used to detect and classify traffic signs based on their shape, color, and other visual features. By combining these functions with machine learning algorithms, it is possible to develop robust and accurate traffic sign detection systems that can be deployed in a variety of real-world applications.

However, it is important to note that traffic sign detection is a challenging task that requires careful design and testing to ensure high accuracy and reliability. The performance of the system can be affected by factors such as lighting conditions, camera angles, and the presence of other objects in the scene. Therefore, it is important to carefully evaluate the performance of the system in different conditions and adjust the algorithms accordingly.

Overall, traffic sign detection using OpenCV in Python is a promising area of research that has the potential to significantly improve road safety and reduce the number of accidents caused by human error. With continued research and development, we can expect to see more advanced and sophisticated traffic sign detection systems in the future.

REFERENCES

"Real-time traffic sign recognition and tracking using OpenCV" by N. Zafar, A. Raza, and M. Ali. In 2016 International Conference on Frontiers of Information Technology (FIT., pages 89-94.

"Traffic Sign Recognition using OpenCV and Support Vector Machines" by V. Antova and A. Kashev. In 2017 International Conference on Intelligent Systems (IS., pages 247-252.

"Traffic sign recognition using convolutional neural networks and local binary patterns" by C. Lee, K. Lee, and J. Kim. In 2019 International Conference on Computer Vision Workshop (ICCVW., pages 3720-3728.

"Traffic Sign Recognition using Histogram of Oriented Gradients (HOG. and Support Vector Machine (SVM." by D. Dang and H. Tran. In 2020

12th International Conference on Knowledge and Systems Engineering (KSE., pages 311-316.

"Traffic Sign Recognition using Deep Learning with OpenCV" by M. T. Khan, M. M. Alam, and M. M. R. Khan. In 2021 IEEE 6th International Conference on Computational Intelligence and Applications (ICCIA., pages 1-6.

[1] Gary Bradski and Adrian Kaehler, “Learning OpenCV”, 1st

Edition, O’Reilly Publications, pages 459-521, 2008.

[2]Jack Greenhalgh and Majid Mirmehdi, “Real-Time

Detection and Recognition of Road Traffic Signs”, IEEE

Transactions on Intelligent Transportation Systems, Vol. 13, No. 4, pages 1498-1506, December 2012.

[3]Auranuch Lorsakul1 and Jackrit Suthakorn, “Traffic Sign Recognition Using Neural Network on OpenCV: Toward Intelligent Vehicle/Driver Assistance System”.

[4]Sergio Escalera and Petia Radeva, “Fast greyscale road sign model matching and recognition”, Centre de Visióper Computador Edifici O – Campus UAB, 08193 Bellaterra, Barcelona, Catalonia, Spain.

[5]W. Shadeed, D. Abu-Al-Nadi, and M. Mismar, “Road traffic sign detection in color images,” in Proc. ICECS, 2003, vol. 2, pp. 890–893.

These papers provide valuable insights and techniques for traffic sign detection using OpenCV in Python.