



University
of Windsor

Remote Sign Detection Driver Assistance System

Submitted in partial fulfillment of the requirements for the course

ELEC-4000: Capstone Design Project

Department of Electrical and Computer Engineering
University of Windsor

August 7, 2020

Remote Sign Detection Driver Assistance System

By

DE LUCA, Daniel (104530508)

MCCARTHY, Brendan (104610282)

VENUTO, Daniel (104620856)

Faculty Advisor

Dr. Roberto Muscedere




Department of Electrical and Computer Engineering

University of Windsor

August 7, 2020

Remote Sign Detection Driver Assistance System

“No action by any design team member contravened the provisions of the Code of Ethics and we hereby reaffirm that the work presented in this report is solely the effort of the team members and that any work of others that was used during the execution of the design project or is included in the report has been suitably acknowledged through the standard practice of citing references and stating appropriate acknowledgments”. The presence of the author's signatures on the signature page means that they are affirming this statement.

DE LUCA, Daniel	104530508		August 7, 2020
MCCARTHY, Brendan	104610282		August 7, 2020
VENUTO, Daniel	104620856		August 7, 2020

ABSTRACT

Object detection is a powerful technology that has expanded into the automotive industry in recent years with new advancements in processing technology and data availability. It has broadened the possibilities of driver assistance systems by connecting autonomous vehicles to the outside world. With street signs providing crucial instructions for autonomous vehicles to operate safely, the goal of this capstone was to create a remote sign detection system for use in autonomous vehicles. The developed system is capable of detecting traffic signs on a remote computer in real time. By utilizing a UDP (User Datagram Protocol) ethernet connection and an AdaBoost cascaded classifier for object detection, this system is capable of sending a simulated live feed from a Raspberry Pi to a remote PC where object detection of street signs is processed and displayed in real time. This was achieved with a robust communications network capable of sending frames at a rate of 112 Mbps while maintaining corruption and loss rates of 0.89% and 0.99% respectively. The reliability of the sign detection algorithm also contributed to the system's functionality with an AUC score of 0.95 and a processing time of 0.044 seconds per frame. With the combined speed of the communications network and sign detection algorithm, the end to end delay of the system was minimized to half a second. The fast speeds of the communications system and reliability of the sign detection algorithm facilitated the successful implementation of the remote sign detection system in real time. (Daniel Venuto)

TABLE OF CONTENTS

Section	Page
Introduction.....	1
Benchmarking.....	2
Design Criteria, Constraints, and Deliverables.....	2
Design Methodology.....	5
Physical Implementations.....	14
Experimental Methods/Model Validation.....	16
Design Specifications and Evaluation Matrix.....	18
Budget.....	23
Conclusions.....	24
COVID-19 Notes.....	24
Back Matter.....	25
a) References.....	25
b) Appendices.....	28

INTRODUCTION

Human error is the foremost cause of accidents and injury on public roads. This has generated a trend in the automotive industry to shift the decision making and control of vehicles away from humans and towards computers. From blind spot monitoring to adaptive cruise control, auto manufacturers are increasing the presence of computer-controlled aspects of daily driving on a yearly basis. Their goal is to develop a fully autonomous vehicle with vastly superior decision making and reaction times than a human could ever achieve. This technology has the potential to save countless lives and completely revolutionize human transportation. (Daniel Venuto)

The effectiveness of autonomous vehicle systems is directly correlated to the quality of information it can acquire about current driving conditions. Street signs are the most direct medium in which drivers are given information about these conditions. They dictate when drivers are required to stop and when it is safe to proceed while also regulating speed. Unlike humans, differentiating between these signs and signals is a complex task. By utilizing computer vision, the objective of this project is to provide an effective solution to the automated detection of street signs and traffic signals thereby establishing the connection between an autonomous car control system and the real world. (Daniel Venuto)

The scope of this project is centered around the development of a computer vision based traffic sign detection system for use in autonomous vehicles. The system is able to identify stop signs and lane endings signs in real time to alert an autonomous vehicle of current driving conditions. The system is only responsible for detection of these signs and does not possess the ability to make decisions or send control commands to the vehicle. For repeatability purposes, pre-recorded videos of a typical driving scenario are stored on an embedded device (Raspberry Pi) and a live feed of video frames is simulated rather than processing a live recording. This ensured accurate benchmarking of specific driving scenarios. (Brendan McCarthy)

The simulated live feed of video frames on the Raspberry Pi is wirelessly transmitted to a remote computer where the processing of the video feed takes place. On the remote device, a computer vision algorithm based on Haar features can detect the street signs in question and display its findings on the computer's screen. This real time data of road signs and signals provided by this system is crucial to the

safety of autonomous vehicles as it develops a connection between an autonomous car and real-world driving conditions. (Brendan McCarthy)

BENCHMARKING

The state-of-the-art technology in the street sign detection is present in Tesla's AutoAI technology for autonomous vehicles. A camera located in the front of the vehicle provides a live stream to the vehicle's onboard computer, called the FSDC (Full Self-Driving Computer). The FSDC uses serial communication to receive the incoming video, and is capable of processing 2.5 billion pixels per second [1]. The incoming video feed is processed by a convolution neural network (CNN) which is the current standard for autonomous vehicle object detection. According to a report [2], The chip is outfitted with two neuroprocessors for their deep neural networks, capable of 36 TOPS (36 trillion operations per second) at a 2 GHz clock speed. The FSDC is also equipped with 12 64-bit cores of ARM A72 architecture with a clock rate of 2.2 GHz. In comparison to this capstone project, processing power is significantly increased and a more robust neural network has been implemented to provide better and more reliable object recognition. (Daniel De Luca)

DESIGN CRITERIA, CONSTRAINTS, AND DELIVERABLES

Design Criteria

The remote sign detection system created in this project involves two different aspects that affect performance. Both the communications network and the sign detection performance affect the quality of the working system. A design criteria was established to encompass both aspects into a simple method to evaluate the final product. Deviation from real time performance and sign detection reliability were chosen as the two design criteria. With the objective of this project to develop a sign detection system for use in autonomous vehicles, real time performance and detection reliability are the fundamental qualities required to maintain passenger safety. (Daniel Venuto)

Constraints

The main constraints for the design criteria involved the processing capabilities of the remote PC (D.1). Although it was equipped with an AMD Ryzen 5 1600 clocked at 3.2GHz, processing the communications network and running the sign detection algorithm at the same time utilized all its capabilities. This led to many compromises to satisfy the design criteria. (Daniel Venuto)

All major design decisions for this system were made to accommodate the design criteria. The main concern pertaining to real time performance involved the speed of the communications network. These issues were foreseen early on and dictated the use of UDP as the communications protocol. UDP is the fastest communications protocol as no error checking is performed. Although this leaves the system vulnerable to corrupted and lost frames, this tradeoff was effective as video is recorded at a high framerate. A typical UDP loss rate of 2% would only result in a loss under 0.5 frames per second at 15 FPS (frames per second). When receiving images at such a high framerate this is negligible. The lack of error checking makes it possible to achieve real time performance and fitting with the design criteria. (Daniel Venuto)

The processing time of the machine learning algorithm was another concern for achieving real time performance as these algorithms are computationally expensive to run. In addition, the PC running the sign detection algorithm would also have to process the UDP packets for communications. This was not foreseen during the planning stages as a random forest classifier was originally intended to be implemented. During the development stages, the random forest architecture performed slowly and was not adequate for real time performance. This was attributed to its need to process around 100 decision trees per sample with no method for rejecting images early on during processing (early termination). A far better architecture was eventually implemented called a cascaded AdaBoost classifier. This machine learning algorithm is designed for efficiency as it can reject an image during the early processing stages and only requires the processing of around 10-15 decision trees. This architecture is explained further in the design methodology section. (Daniel Venuto)

The final compromise made with respect to real time performance involved the video quality. A lower video quality was chosen with a resolution of 720p (720 by 1280 pixels) and a frame rate of 15 FPS (frames per second). The effect of this was significant as it increased the speed of the communication channel with smaller file sizes and the speed of the sign detection algorithm as less

pixels needed to be processed. Corruption rates were much more manageable with 720p video and the algorithm processed in roughly half the time. (Daniel Venuto)

The second aspect of the design criteria was algorithm performance. This directly conflicted with real time performance as more complex and reliable algorithms came at a cost to processing speeds. This is why a convolution neural network was not chosen as the classifier architecture. A second compromise made with respect to algorithm performance was the choice to use 720p video. Although this improved processing speeds greatly, it came at a sacrifice to the sign detection algorithm's reliability. Signs could only be detected from closer distances in 720p compared to 1080p. (Daniel Venuto)

With the hardware constraints of the remote PC, this combination of communications protocol and algorithm design achieved the best results with respect to the design criteria. Deviation from real time performance was under half a second and the detection algorithm achieved an AUC score of 0.95 on the validation data. In addition, the corruption rate and loss rate in the UDP communications network is 0.89% and 0.99% respectively allowing the vast majority of frames to be processed. (Daniel Venuto)

Deliverables

The system developed in this capstone can display the output of a remote street sign detection algorithm on a remote PC in real time. A live feed of a pre-recorded driving video is simulated on the Raspberry Pi and sent to the remote PC via a UDP connection. The incoming video feed is processed by the sign detection algorithm and displayed on the device with bounding boxes denoting the location of the detected signs on the image. The system can identify both stop signs and lane ending signs from 720p video in real time. (Brendan McCarthy)

DESIGN METHODOLOGY

Haar Features

The underlying method that will be utilized for the sign detection algorithm is a computer vision method called Haar Cascade Classifiers. These classifiers utilize combinations of Haar features to identify specific objects in images that it has been trained to recognize. Haar features are rectangular kernels that detect different intensity boundaries on grayscale images that are based on Haar wavelets. These features are very similar to kernels used in convolutional neural networks (CNNs) [3]. An ideal Haar-like feature is a sharp boundary between a set of low intensity pixels and a set of high intensity pixels [4]. The three types of Haar-like features are shown in Figure 1 below. (Brendan McCarthy)

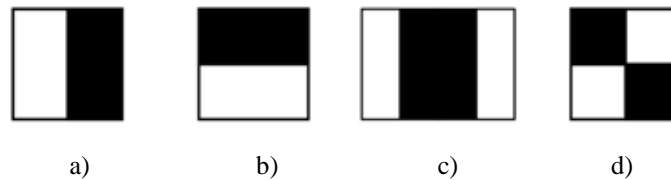


Figure 1) Haar-like Features: a) Edge, b) Edge, c) Line, d) Four-rectangle Features [4]

Haar features can be detected on any type of grayscale image. For instance, the pixels in the red region of a stop sign would have a lower pixel intensity when compared to the pixels on the sky region around the stop sign. This means a Haar feature could be created between the sky region and the red region inside the sign. This region inside the sign will have low intensity pixels with pixel values close to 0, while the sky region will have higher intensity pixels with pixel values close to 1. A rectangular feature can be created which is composed of these two regions forming an “edge” Haar feature as shown in Figure 2 below. (Brendan McCarthy)



Figure 2) Example of Haar feature on a stop sign

By identifying these specific features, the Haar cascade classifier will be able to classify the image as a stop sign. Haar cascades are developed to perform like this by training them on many

positive and negative images until the algorithm learns to extract the important features that characterize the object it is trying to detect. The python library OpenCV provides a platform to assist in the training of Haar cascades [5]. Using this method, different street signs such as stop signs and lane ending signs will be trained to be detected with accuracy. (Brendan McCarthy)

When extracting the Haar features from an image, there are many computations involved and larger images contain exponentially more Haar features. This makes it difficult to extract the features quickly. To overcome this, a function called integral image (denoted as “ii”) is used. The integral image function rapidly computes Haar features by creating a summed area table for the pixel values [6]. This function can be denoted by the equation below. (Brendan McCarthy)

$$I(x, y) = \sum_{\substack{x' \leq x \\ y' \leq y}} i(x', y')$$

Equation 1) Integral Image Function [6]

The equation represents the integral image function at a point x, y in the image. This equation takes the sum of the pixels above and to the left of its location to get the pixel value for the summed area table. By using four array references (A, B, C, D) to bound a region of interest, it is possible to take the sum of all pixel values in a rectangular region anywhere in the image. The figure below shows an example of calculating the integral image of a particular region. (Brendan McCarthy)

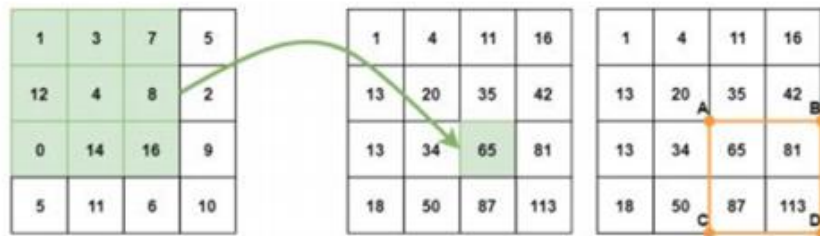


Figure 3) Example of calculating the sum of pixels in a rectangular region using ii function [7]

This is useful for detecting Haar features as the difference between the sum of the high intensity pixel region and low intensity pixel region must be calculated to detect a Haar feature.

Detection Window

The detection window is the region of interest on an image where Haar-like features are extracted from. The feature values are calculated using the integral image function and are used as

inputs to the sign detection algorithm. A detection window of 25x25 pixels was chosen as over 100 thousand possible Haar-like features exist in a region of this size. Using a larger window would exponentially increase the feature size posing computational concerns. With the use of a detection window, the location of each detected object on the image can be determined. (Daniel Venuto)

The image size that was processed by the sign detection algorithm was 720x1280 pixels (720p). This posed an issue as the maximum region of interest for the detection window was only 25x25 pixels. A sampling technique called a sliding window was implemented to overcome this issue. This works by iteratively applying the detection window at the majority of points in the x and y axis as shown in Figure 4 below.

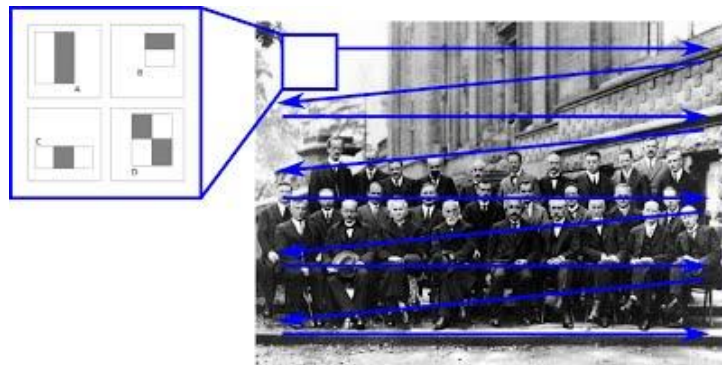


Figure 4) Sliding window example [8]

This allows the window to “slide” across the image and extract Haar features from all regions in the image. An added benefit to this technique is the location of each detected object is known as the coordinates of the detection window are recorded. (Daniel Venuto)

The fixed detection window size also posed another obstacle to object detection. If an object is larger than 25x25 pixels, it won’t be detected even with a sliding window. This was rectified with the use of image pyramids. Each image is downsampled multiple times to create an image pyramid [9]. The 25x25 sliding window is applied to each image scale in the pyramid allowing detection of a desired object that was larger than 25x25 pixels prior to downscaling. An example of an image pyramid and the effect of an image pyramid on an image of a stop sign is shown in Figure 5 below. (Daniel Venuto)

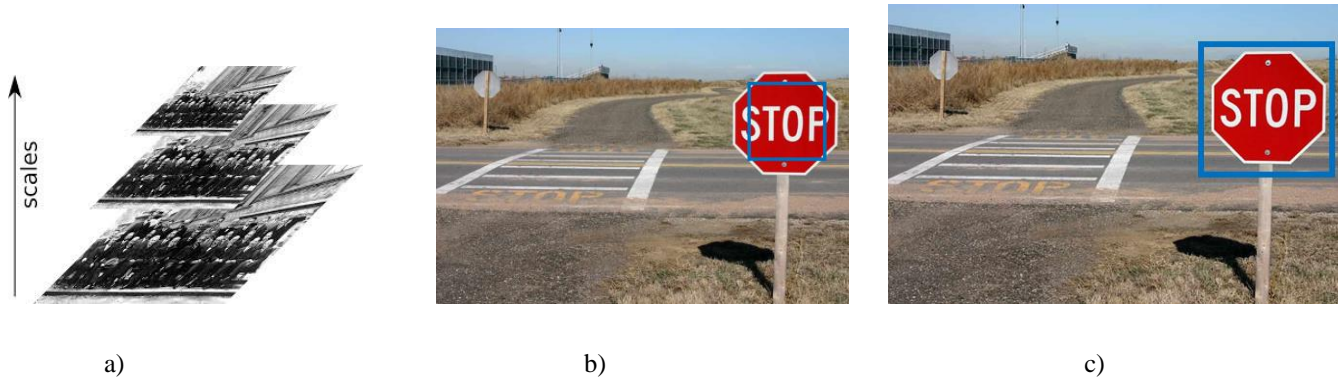


Figure 5) a) Image pyramid example [8] b) Image prior to applying image pyramid c) Image after applying image pyramid

The sliding window and image pyramids were implemented with the python library OpenCV in the display codes (A.4, B.5). Each frame received from the communications network is fed into the detectMultiScale function where sampling using a sliding window and image pyramid takes place. The sliding window and image pyramids were also implemented for the second detection method attempted using a random forest classifier. In this case, a series of for loops and resizing functions were created to apply a sliding window and image pyramid to each frame (C.1).

Classifier Architecture

An Adaboost cascaded classifier architecture was implemented for the sign detection algorithm. The python library OpenCV was used to create and train this classifier (B.1-B.4). This architecture is composed of cascading stages with each stage containing a decision tree called a weak classifier. The decision trees accept specific Haar features from the detection window as inputs, with the amount and complexity of the inputs increasing at each stage. An image must pass all decision tree stages to be detected by the algorithm whereas a rejection at any stage results in a non-object classification and the remaining stages are not processed. By using a combination of weak classifiers, one strong classifier is formed. This technique is commonly referred to as ensemble learning [10]. A diagram of the AdaBoost cascaded classifier architecture is shown in Figure 6 on the next page. (Daniel Venuto)

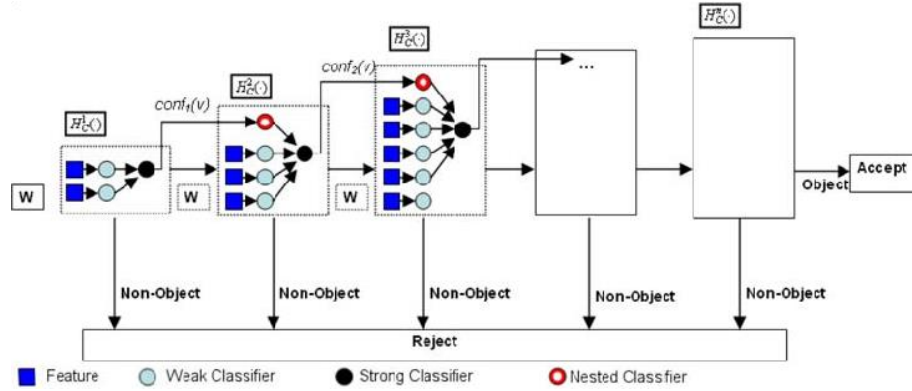


Figure 6) AdaBoost cascaded classifier architecture [11]

The process of rejection at any stage is called early termination and lowers the computational burden on the processor. Only images that actually contain the object in question or closely resemble it reach the higher order stages of the algorithm allowing it to focus the processor's resources on these candidates. Images that don't closely resemble the object are rejected quickly during the early stages saving valuable computational resources. This is important when sampling an image with the 25x25 pixel detection window described earlier as the vast majority of samples from the image do not contain the object, especially when detecting traffic signs. (Daniel Venuto)

A machine learning concept called Adaboost was implemented for the training of this algorithm. Each decision tree was trained on weighted samples meaning the incorrectly predicted samples from the previous stage are given a greater weight during the training of the next stage [12]. This technique allows the higher order stages to improve on the flaws of previous stages by identifying Haar features that the lower stages may have missed. The decrease in error at each following stage is maximized through this process. (Daniel Venuto)

Random Forest Classifier

A second method was attempted during the development of the sign detection model. Using the Sci-Kit Learn library for python, a classifier was developed using the random forest architecture (C.1). A random forest classifier uses a set of n decision trees, each accepting Haar features from a sliding detection window as inputs. The decision trees are separately trained to create variation among them [13]. Each tree fully processes the inputs and votes on if it detects a specific street sign or not. The majority vote from the n decision trees is calculated and dictates the final classification from the

algorithm. A diagram of the random forest classifier architecture is shown in Figure 7 below. (Daniel Venuto)

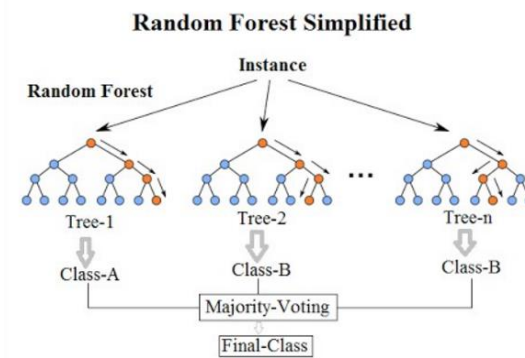


Figure 7) Random Forest architecture [13]

The use of multiple decision trees allows the algorithm to encompass multiple different training outcomes and helps the algorithm generalize between different variations of a specific street sign. Although a random forest can be effective for classification, it is much slower than the cascaded AdaBoost architecture and is not suitable for real time performance. This is attributed to its lack of early termination and large number of decision trees. Without early termination, the decision trees are more complex and all trees need to be processed for each sample. There is no rejection during the early stages like the cascaded AdaBoost architecture which reduces the computational burden. Also, 100 complex decision trees were needed for optimal results which is significantly more than the 12 weak ones used in adaboost. These factors made the random forest architecture unsuitable for the system's requirement of real time processing. (Daniel Venuto)

The random forest classifier was implemented using the Sci-Kit Learn library for python. Haar features were extracted from each image in the dataset using the integral image function and stored in an array. This data was split into training, testing, and validation sets that would be used to train and evaluate the classifier performance. A random forest classifier was defined with parameters including number of estimators, max depth, and max features. The classifier was trained on the dataset and validated using AUC score. To improve the processing speeds, a Sci-Kit Learn function was called to determine the most important Haar features and the algorithm was retrained on them to produce an improved classifier. The sliding detection window and image pyramid was implemented with a series of for loops and resizing functions. (Daniel Venuto)

To improve the models accuracy and efficiency, a grid search was implemented with Sci-Kit Learn. This trained the random forest on a set of predefined parameters for the number of estimators, max feature inputs per tree, and max tree depth. Every possibility from the predefined set of parameters was trained and validated to find the optimal parameter combination. The best result was achieved with a 100 estimators, 100 features per tree, and no max depth. Even with the optimized parameters, real time performance could not be achieved and the cascaded adaboost architecture was selected as the optimal model architecture. (Daniel Venuto)

Dataset

To implement the training of the Haar cascades, a dataset with positive and negative images was needed. To create the positive samples, street signs were overlaid onto background images that were a part of the initial dataset (B.1). To make each street sign unique, the overlaid signs were manipulated using an OpenCV function (B.3). These positive samples were manipulated by changing orientations in the x, y, and z axis, adding pixel noise, and changing the background colors. An example of the overlaid images can be seen in Figure 8 below. (Brendan McCarthy)



Figure 8) Example of positive image samples

The negative images were created by selecting images from the dataset that were very simple. It was important that these images did not contain instances of the street sign that was trying to be detected [14]. This would negatively affect the training and confuse the algorithm. However, images containing different street signs were used to avoid false positives in the classifier. This would make the training more accurate and ensure that the final stages of training would only detect the exact sign that was being tested. An example of the negative images is shown in Figure 9 below. (Brendan McCarthy)



Figure 9) Example of negative image samples

Classifier Validation

In order to evaluate the performance of the classifier, a code was created to run the classifier on a validation dataset (B.6). With the application of image pyramids and a sliding window to the images, each image was sampled multiple times and multiple detections could be present for any given image. This is especially prevalent in a poor quality classifier. To cope with this issue, multiple conditions were defined based on the true classification of the image and the number of detections made. These results were used in Sci-Kit Learn's performance metrics library to calculate the performance results. (Daniel Venuto)

Communications

The goal for the communications aspect of the project was to create a reliable data link between the Raspberry Pi and the remote PC. This link needed to be quick enough to minimize the delay between the transmission of data and the detection of important features within the data. To simulate a live feed coming from the Raspberry Pi, the video was processed into individual frames (A.1). This was done to simulate frames entering the directory from a live feed. To further facilitate faster transfer speeds, a UDP network was implemented to enable communications between the two computers. This was developed in python with a code developed for the transmission of packets on the Raspberry Pi (A.2) and a second code developed for receiving packets on the remote PC (A.3). UDP is a fast communications protocol compared to others. This is a result of the lack of error checking and the retransmission of lost packets, leaving the network prone to corruption and lost data. This is an important issue since there is now a risk for important information to be lost. While these frames are transmitting, they will be stored in a directory to be read by the image detection algorithm and displayed on the remote PC. (Daniel De Luca)

The User Datagram Protocol (UDP) is a low-latency, loss-tolerating connection which makes it beneficial for time sensitive communications [15]. UDP works by using the IP to get a datagram from one computer to another. UDP packets do not always take the same path, resulting in lost and out of order packets. A checksum capability is offered, but to keep transmission speeds high this option is not used. The link between the Pi and remote PC has a theoretical transmission rate of approximately 250 Mbps. Using this raw speed, however, resulted in large amounts of missing and corrupted data. This is due to limited processing power of the remote PC, as a significant processing power is needed to decompress and reconstruct the frames. The solution to this problem was to delay transmission after a specified amount of data was sent in the transmission code. Tests were completed by completing ten transmission rounds per configuration. The different configurations consisted of different delay times and sent data. After implementing this solution, the transmission speed was now limited to approximately 112 Mbps, but resulted in very little lost and corrupted data. (Daniel De Luca)

The data being transmitted is in the format of individual images or frames. The .JPEG image format was originally used, and through multiple rounds of transmission tests, it was discovered that the .PNG image format offered more reliable and consistent results, further lowering data corruption and loss. This is because PNG (Portable Network Graphics) was developed for the purpose of sending images through a network. The quality of the video was also sacrificed in order to ensure a reliable transmission. For example, a 1 min video that has a resolution of 1080p and frame rate of 60 fps would result in 3600 frames to be sent. The higher resolution would also result in greater file size further burdening our network. Different video qualities and frame rates were tested to determine the one best suited for the project. The same testing parameters of ten transmission rounds was used, and it was observed that a 480p resolution at a frame rate of 15 fps offered the best results for low corruption/loss. Though this resolution quality was too low and it was affecting the image detection algorithms accuracy, the next most viable option was to use 720p resolution at 15 fps as the corruption/loss rate was still below 1%. (Daniel De Luca)

Once transmission begins, all of the incoming frames are stored in a directory by the receiving code (A.3). This directory is constantly being scanned for frames by a third code running the sign detection algorithm (A.4). To avoid reading a file that is being written to by the receiver, a delay was used only during the reading of the first packet. This gives the remote PC time to finish decompressing and reconstructing the first frame. The stored frames identified in the directory are iterated in a for loop

and processed in the image detection algorithm frame by frame. Each frame is displayed on the remote PC and removed from the directory immediately after. Once all frames in the for loop are processed, the directory is rechecked for new frames and the process is repeated until frames stop entering the directory. (Daniel De Luca)

PHYSICAL IMPLEMENTATION

The physical implementation of the design incorporated the use of a Raspberry Pi to send pre-recorded video to a remote PC containing the detection algorithm. This design proved to be successful as the simulated live feed was able to detect stop signs with minimal delay. Shown below in Figure 10 is the physical setup for the project, displaying the ethernet connection between the raspberry pi and remote PC. (Brendan McCarthy)

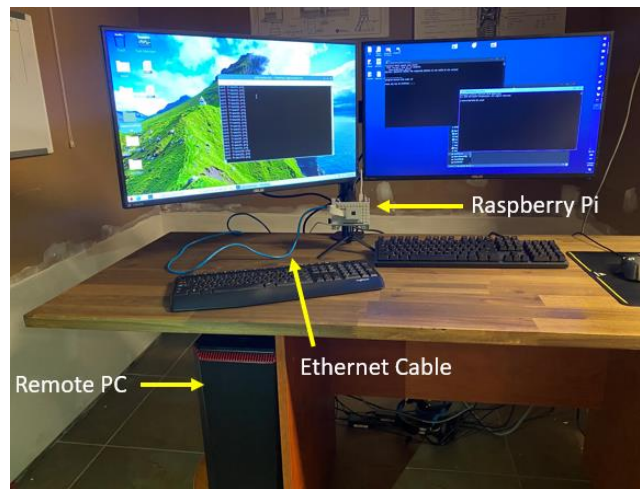


Figure 10) Project setup

The pre-recorded video included different instances of driving conditions while encountering the detectable street signs. These videos were recorded on a camera and saved to the Raspberry Pi. The videos were then pre-processed into individual frames using a pre-processing code (A.1). Utilizing a UDP connection, the video frames were sent via ethernet from the Raspberry Pi to the remote PC by a python code (A.2). A receiving version of this code was implemented on the remote PC to save the image frames into a directory individually (A.3). (Brendan McCarthy)

After receiving an image frame on the PC, a python code checks the directory for available frames and loads each frame into the AdaBoost cascaded classifier to evaluate the image and display the

live feed (A.4). Also, the algorithm simultaneously checks for corrupted frames and deletes any it discovers. After each frame is processed by the algorithm, it is then deleted from the directory to allow for the next frame to be processed. Shown below in Figure 11 is a flowchart to describe the direction of data flow in the system. (Brendan McCarthy)



Figure 11) System operation

The final output for the project shows dashboard camera footage of interaction with street signs. Using the functions and codes previously described, the display of the detection is performed accurately with minimum deviation from real-time footage. The image frames are successfully sent to the PC, processed, analyzed for detection, and assembled into a live feed. (Brendan McCarthy)

EXPERIMENTAL METHODS/MODEL VALIDATION

Sign Detection Algorithm

The design criteria discussed in section 4 formed the basis for the evaluation of the remote sign detection system. Deviation from real time and sign detection performance dictated the methods selected for validation of each aspect of the project. (Daniel Venuto)

The sign detection algorithm was validated using 4 different machine learning evaluation metrics. AUC score, precision, recall, and F1 score where each was plotted against the number of stages present in the algorithm architecture. AUC score compares true positive rate to the false positive rate at different threshold values for classification [16]. A high performance algorithm will maximize the area under this curve and achieve a value close to 1. The equations and diagram for this metric are shown below. (Daniel Venuto)

$$TPR = \frac{TP}{TP + FN}$$

Equation 2) True positive rate [16]

$$FPR = \frac{FP}{FP + TN}$$

Equation 3) False positive rate [16]

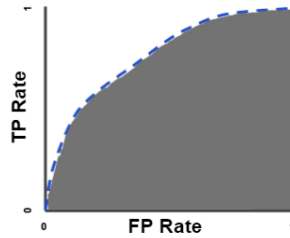


Figure 12) AUC of ROC curve example [16]

Precision is a measure of how precise a model is when predicting a positive image [17]. It compares the percentage of true positives relative to the total number of positives predicted. A high precision score denotes an algorithm that does not detect positives from a negative input often. The equation for precision is shown in Equation 4 below. (Daniel Venuto)

$$\text{Precision} = \frac{\text{True Positive}}{\text{True Positive} + \text{False Positive}}$$

Equation 4) Precision [17]

Recall is a measure of the percentage of positives detected from the dataset[17]. It compares the percentage of true positives relative to the total number of positives in the dataset. A high recall score

describes an algorithm that rarely misclassifies a positive input. The equation for recall is shown in Equation 5 below. (Daniel Venuto)

$$\text{Recall} = \frac{\text{True Positive}}{\text{True Positive} + \text{False Negative}}$$

Equation 5) Recall [17]

F1 score computes the harmonic mean of precision and recall [17]. An algorithm possessing a high F1 score will have both high recall and precision scores. This is a quality of a model that rarely generates false positives or false negatives meaning it rarely misclassifies positive or negative inputs. The equation for precision is shown in Equation 6 below. (Daniel Venuto)

$$\text{F1} = 2 \times \frac{\text{Precision} * \text{Recall}}{\text{Precision} + \text{Recall}}$$

Equation 6) F1 Score [17]

AUC score and Precision will be the main metrics for validation of the sign detection algorithm as they scrutinise algorithms with a high number of negative samples. This is the case for this system as even in the positive inputs, many negative instances are identified by the sliding window whereas only one positive instance exists. In addition, processing time of the model will also be measured to confirm real time performance. (Daniel Venuto)

Communications

To validate the network, many tests were carried out to measure the transmission rate and the average corruption and loss of data. Transmission rate was measured by sending a fixed file size and recording the amount time it took to send this file size. One round of testing consisted of ten different transmission rounds. Testing for average corruption and loss was done by utilizing different image formats, resolutions, frame rates, and transmission rates. The same testing methodology of ten transmission rounds per round of testing was done to obtain as accurate information as possible.

DESIGN SPECIFICATIONS AND EVALUATION MATRIX

Sign Detection Algorithm

The sign detection model was validated using a separate data from the training dataset. This ensured the machine learning model was tested on data it had not seen during the training stage. 728 samples were tested with an equal positive and negative data split. Four different machine learning metrics were recorded including AUC score, precision, recall, and F1 score. Each metric was plotted with reference to the number of stages in the model's architecture. A value close to 1 is ideal for each metric. Results are shown in Figures 13-16 below. (Daniel Venuto)

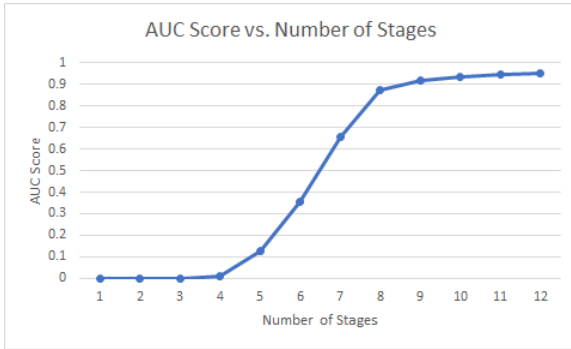


Figure 13) AUC Score vs. Number of Stages Figure

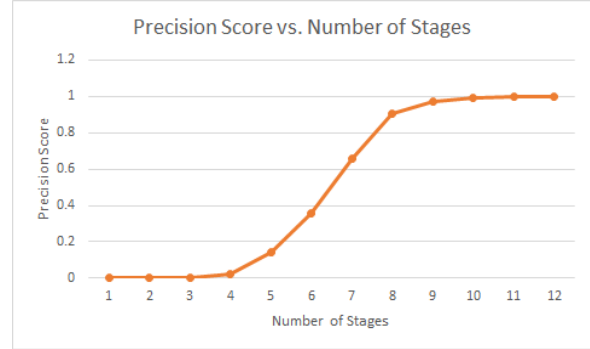


Figure 14) Precision Score vs. Number of Stages

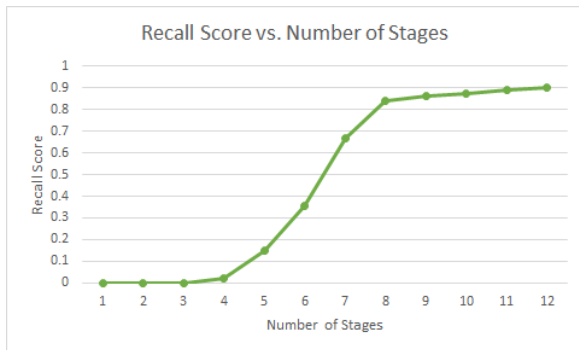


Figure 15) Recall Score vs. Number of Stages Figure

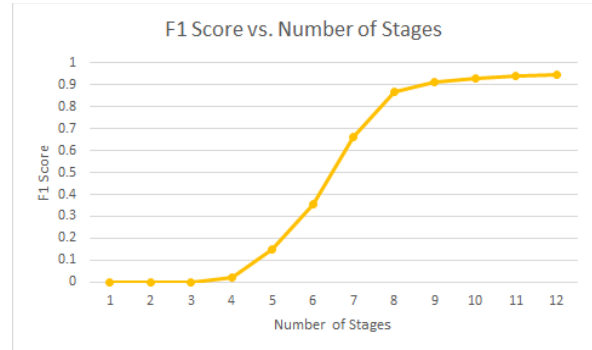


Figure 16) F1 Score vs. Number of Stages

From the figures above it can be seen that the classifier performs its best when using 12 stages. With this condition scores of 0.9505, 1, 0.9011, and 0.948 were achieved for AUC, precision, recall, and F1 Respectively. As values as close to 1 as possible are ideal, the results achieved by the classifier are excellent. (Daniel Venuto)

The most important metrics for the classifier are AUC and precision scores as they scrutinize classifiers with a high number of negative samples. This is relevant in street sign detection as stop signs make up a small percentage of the frames during a typical driving experience. In addition, stop signs make up a small percentage of the samples in a positive frame as a sliding window implementation is

used. With values of AUC score and precision at 0.9505 and 1 respectively, the algorithm performs as desired. The precision value of 1 means the classifier never produced a false positive in the validation data which is difficult given the high number of negative samples compared to positive ones. This does come at a sacrifice to recall though as a value of only 0.9011 was achieved. This means the algorithm produces more false negatives than expected. (Daniel Venuto)

The processing time of the model was also measured at resolutions of 720p and 1080p. Each frame was processed in approximately 0.095 seconds in 1080p and 0.044 seconds in 720p. To maintain a framerate of 15 FPS, the processing time per frame must be below 0.067 seconds. Given the processor restrictions, this was only achieved with 720p resolution. Although real time performance was achieved it came with a compromise to detection distance. Street Signs could only be detected at 30m in 720p as compared to 50m in 1080p. (Daniel Venuto)

Communications

The overall design specifications for the communications portion needed to satisfy two major goals of speed and reliability. Reliability is to be defined as minimum corrupted and lost data. These two goals are important for ensuring that the image detection algorithm is reading the frames as close to real time as possible to simulate a live feed. Originally, the .JPEG image format was used for the frame data. However, corruption and data loss was frequent, and testing began to discover a more suitable image format. The .PNG format was chosen and testing was performed to observe its performance. The testing was done through ten transmission rounds and recording the average frame corruption, loss, and compromised files. The compromise measure was done by adding the average corruption and loss. This was done to achieve a better measurement of comparison for the two formats. A comparison of the test results for .PNG and .JPEG are shown in Figure 17 on the next page. (Daniel De Luca)

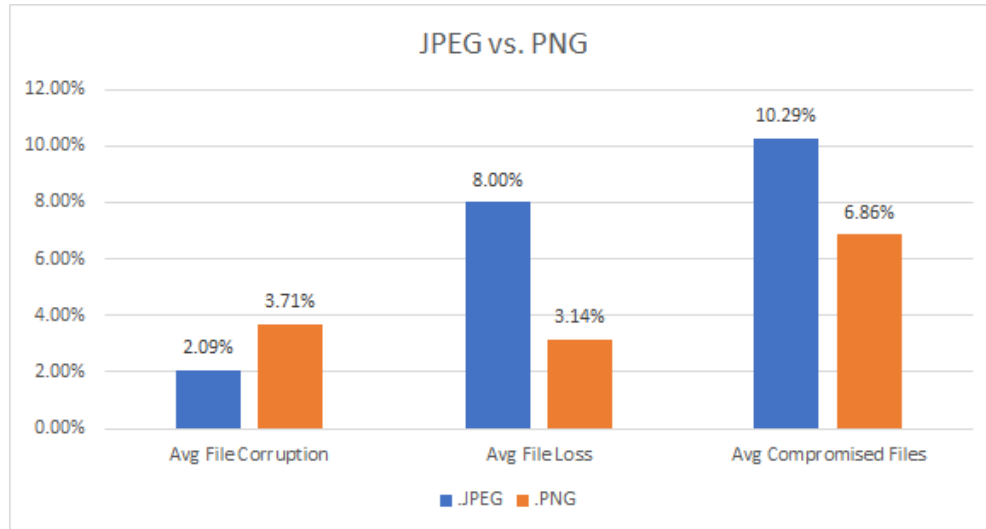


Figure 17) Comparison between image formats

It is observed from these testing rounds that the .PNG image file format offers better results. Though average file corruption was found more in .PNG, it performed exceptionally better in file loss. With these results, the speed of the network was now to be improved. A buffer size of 4 kB was chosen for sending and receiving. Larger buffer sizes were tested, but these led to an increase in compromised data. With this chosen buffer size, the same testing methodology of ten transmission rounds were performed to determine the average transmission rate. A total file size of 490 MB was used to achieve adequate results as shown in Tables 1-2 below. (Daniel De Luca)

Table 1) Testing Round 1

Round	Time (s)	Rate (Mbps)
1	22.47	175.5
2	17.8	221.5
3	15.25	258.6
4	14.93	264.1
5	14.91	264.5
6	14.57	270.7

Table 2) Testing Round 2

Round	Time (s)	Rate (Mbps)
1	31.88	123.7
2	20.77	189.9
3	14.55	271.0
4	14.75	267.4
5	14.8	266.5
6	14.82	266.1

7	14.83	265.9
8	14.49	272.2
9	14.88	265.0
10	14.85	265.6
Average		252.4

7	14.64	269.4
8	14.62	269.7
9	14.78	266.8
10	14.59	270.3
Average		246.1

An average transmission rate of approximately 250 Mbps was achieved. It has also been observed that an apparent network speed up occurs. This was hard to test for as the speed up seemed to occur when the Raspberry Pi was initially powered on. The above tests were performed on different days resulting in this observation. The exact reason for this has not been determined. The results from the speedup test are shown in Figure 18 below. (Daniel De Luca)

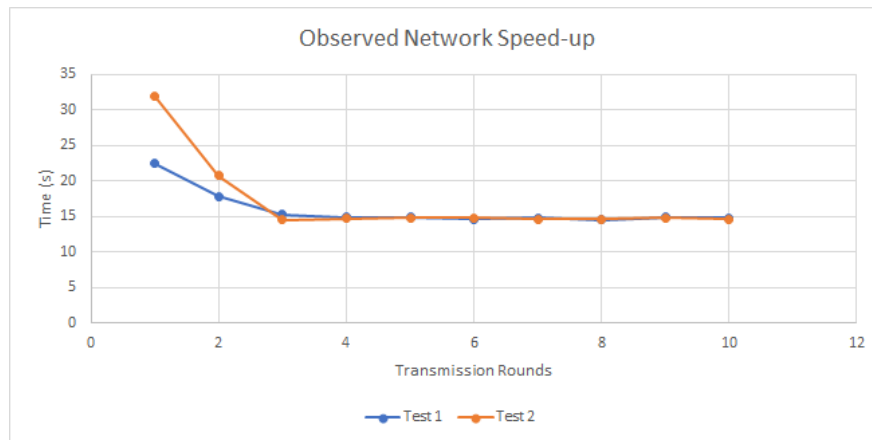


Figure 18) Observed network speed-up

Utilizing the raw speed for the network caused an increase in corruption and data loss. This is due to the limited processing power of the remote PC. To combat this increase, a solution to throttle the speeds was implemented. The first solution was to apply a form of flow control. Since UDP has no flow control, there is too much data arriving before the remote PC can handle it, leading to lost data. The flow control solution caps how much data is sent before stalling the program. In this case, after 28 kB of data is sent the program is stalled by 1 ms, and after 50 frames have been completely sent the program is stalled by 2 ms. These times were chosen as they provided flow control without slowing the process too

much. This solution slows the transmission rate to approximately 112 Mbps. Since the rate has now been more than halved, a method to keep the network unburdened needed to be created. This solution was to determine the quality of images being passed through the network, since higher resolution would require more bandwidth. The resolutions tested were 720p and 480p at frame rates of 30 fps and 15 fps. 1080p had not been thoroughly tested as it had already been determined this resolution would burden the network too much. Figure 19 shows the results of these tests below.

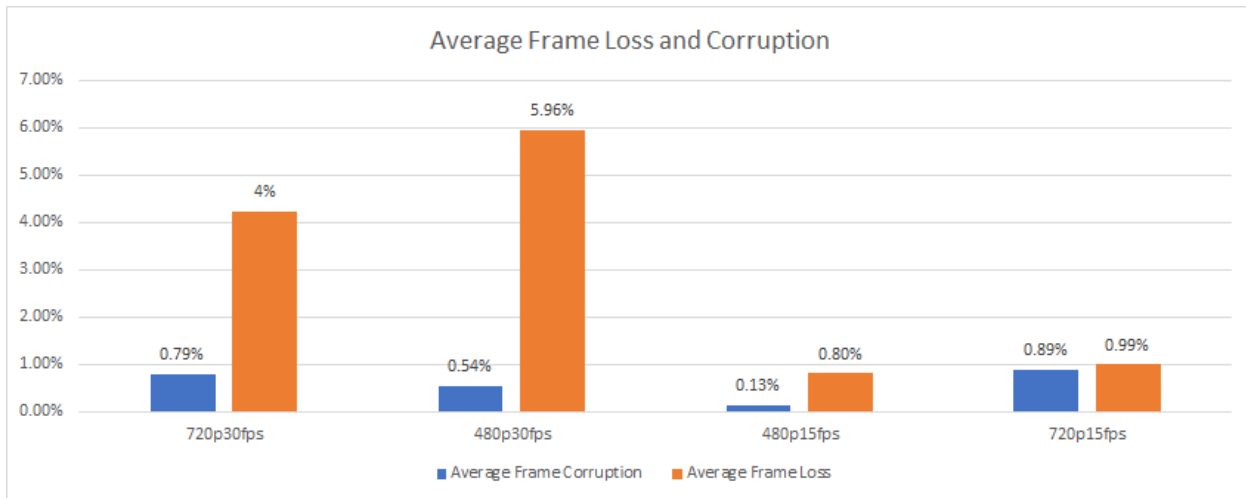


Figure 19) Average corruption and loss rates across different video qualities

The resolution of 480p at 15 fps offered the best results, but at the cost of the algorithms detecting abilities. Since 720p at 15 fps still offered good results, it was decided to use this configuration for the final implementation. (Daniel De Luca)

BUDGET

After fully implementing the design, the total cost did not exceed the initial budget set. The Raspberry Pi was supplied by the supervisor which came with a camera module, the remote PC used for receiving and running the classifier was a desktop computer owned by one of the group members, and all the libraries used for programming were open sourced. A linux server was rented to train the classifier as there wasn't a sufficient amount of RAM. However, a \$100 free credit was given making the process free to use. The cost of supplies is shown in Table 3 below. (Daniel De Luca)

Table 3) List of Supplies

Item	Price
Raspberry Pi Model 3B+	Free
Raspberry Pi Camera v2	Free
Desktop Computer	Free
Libraries	Open source
Server Space	Free

Overall cost of supplies: \$0

Budget: \$500

Although the project was able to be completed without spending any money, to improve overall performance would cause a significant increase in costs. For example, the remote PC is running with an AMD Ryzen 5 1600 processor with 16 GB of RAM. It had been observed that processing power was a bottleneck in this project, and improvements to this factor could cost anywhere between \$500 and \$1000 depending on the processor chosen. This has the potential to double the original budget for an increase in performance. (Daniel De Luca)

CONCLUSIONS

The presented approach for remote street sign detection using Haar features was proven to be successful as the detection algorithm accurately identified street signs while maintaining minimal data loss and real time performance. The desired results for the detection algorithm performance were to achieve ROC and precision scores of 1. The measured results for ROC and precision were calculated to be 0.9505 and 1 respectively for the classifier. For the communication aspect of the project, the desired results were to maintain zero image frame corruption and loss. After numerous tests, the measured results were found to be 0.89% and 0.99% for frame corruption and loss respectively. In addition, the deviation from real time in the simulated live feed was less than half a second, fitting with the design goals. By developing a reliable sign detection algorithm operating over a robust communication network in speeds very close to real time, the design criteria established in the early stages of the project were fulfilled. (Brendan McCarthy)

The use of a remotely operated sign detection stood as the innovative design aspect for the final product. The capabilities of remote detection allow it to collect video, pre-process the video to frames, and seamlessly transfer the frames to another device with limited losses. This design is beneficial for vehicles as the embedded device is compactful enough to limit space and handle the pre-processing power. Also, this design could be used for other variations of remote detection such as wireless implementation. A wireless communications design for the project would be a more commercial model and may be used for future project interpretations. With the remote connection of the sign detection algorithm, a connection between an autonomous car control algorithm and the outside world could be established from any location. (Brendan McCarthy)

COVID-19 NOTES

With the outbreak of COVID-19 during this project, there were minimal changes to the final design. The outbreak prohibited the group from meeting to work on similar aspects of the project. However, the group was still able to communicate virtually and achieve all the desirable milestones.

REFERENCES

- [1] TopSpeed. "Tesla Autonomy Day 2019 - Full Self-Driving Autopilot - Complete Investor Conference Event," *YouTube*, April. 23, 2019. [Video file]. Available: <https://www.youtube.com/watch?v=-b041NXGPZ8&t=445s>. [Accessed: July 30, 2020].
- [2] "Tesla FSDC (Full Self-Driving Computer).," *TAdviser.ru*. [Online]. Available: [http://tadviser.com/index.php/Product:Tesla_FSDC_\(Full_Self-Driving_Computer\)](http://tadviser.com/index.php/Product:Tesla_FSDC_(Full_Self-Driving_Computer)). [Accessed: 07-Aug-2020].
- [3] C.-F. Wang, "What's the Difference Between Haar-Feature Classifiers and Convolutional Neural Networks?," 04-Aug-2018. [Online]. Available: <https://towardsdatascience.com/whats-the-difference-between-haar-feature-classifiers-and-convolutional-neural-networks-ce6828343aeb#:~:text=A%20Haar%2DFeature%20is%20just,%E2%80%9D%2C%20used%20to%20detect%20edges.&text=Haar%2DFeatures%20are%20good%20at%20detecting%20edges%20and%20line>. [Accessed: 28-Jul-2020].
- [4] A. Mordvintsev, "Face Detection using Haar Cascades¶," *OpenCV*, 00-Jan-2013. [Online]. Available: https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_objdetect/py_face_detection/py_face_detection.html. [Accessed: 21-Jul-2020].
- [5] OpenCV, "Cascade Classifier Training," *OpenCV*. [Online]. Available: https://docs.opencv.org/master/d8/tutorials_traincascade.html. [Accessed: 25-May-2020].
- [6] P. Viola, M. Jones, "Rapid Object Detection using a Boosted Cascade of Simple Features," [Online]. Available: <https://www.cs.cmu.edu/~efros/courses/LBMV07/Papers/viola-cvpr-01.pdf>. [Accessed: 19-July-2020]

- [7] G. Bhadani, "Haar Cascade Classifier vs Histogram of Oriented Gradients(HOG)," 13-May-2020. [Online]. Available: <https://medium.com/@goutam0157/haar-cascade-classifier-vs-histogram-of-oriented-gradients-hog-6f4373ca239b>. [Accessed: 21-Jul-2020].
- [8] Z. Ye, "Viola-Jones Face Detection - 5KK73 GPU Assignment 2012," 5KK73 GPU 2012, 2012. [Online]. Available: <https://sites.google.com/site/5kk73gpu2012/assignment/viola-jones-face-detection>. [Accessed: 20-Jul-2020].
- [9] "Pyramid (image processing)," Wikipedia, 16-Apr-2020. [Online]. Available: [https://en.wikipedia.org/wiki/Pyramid_\(image_processing\)](https://en.wikipedia.org/wiki/Pyramid_(image_processing)). [Accessed: 02-Aug-2020].
- [10] "Ensemble learning," Wikipedia, 13-Jul-2020. [Online]. Available: https://en.wikipedia.org/wiki/Ensemble_learning. [Accessed: 02-Aug-2020].
- [11] J. Ruiz-del-Solar and R. Verschae, "Object detection using cascades of boosted classifiers - ppt download," 15-Dec-2006. [Online]. Available: <https://slideplayer.com/slide/5754730/>. [Accessed: 25-Jul-2020].
- [12] "AdaBoost," Wikipedia, 02-Aug-2020. [Online]. Available: <https://en.wikipedia.org/wiki/AdaBoost>. [Accessed: 05-Aug-2020].
- [13] "Random forest," 28-Jul-2020. [Online]. Available: https://en.wikipedia.org/wiki/Random_forest. [Accessed: 02-Aug-2020].
- [14] J. Allen, "OpenCV-object-detection-tutorial," *OpenCV-object-detection-tutorial by JohnAllen*. [Online]. Available: <https://johnallen.github.io/opencv-object-detection-tutorial/>. [Accessed: 24-Jul-2020].
- [15] M. Rouse, "What is UDP (User Datagram Protocol)?," 16-Apr-2020. [Online]. Available: <https://searchnetworking.techtarget.com/definition/UDP-User-Datagram-Protocol>. [Accessed: 02-Aug-2020].
- [16] "Classification: ROC Curve and AUC | Machine Learning Crash Course," Google. [Online]. Available: <https://developers.google.com/machine-learning/crash-course/classification/roc-and-auc>. [Accessed: 03-Aug-2020].

- [17] K. P. Shung, “Accuracy, Precision, Recall or F1?,” Medium, 10-Apr-2020. [Online]. Available: <https://towardsdatascience.com/accuracy-precision-recall-or-f1-331fb37c5cb9>. [Accessed: 01-Aug-2020].
- [18] Hamoudaq, “sending/receiving file UDP in python,” 21-Dec-2012. [Online]. Available: <https://stackoverflow.com/questions/13993514/sending-receiving-file-udp-in-python>. [Accessed: 29-Apr-2020].
- [19] “Face classification using Haar-like feature descriptor,” Face classification using Haar-like feature descriptor - skimage v0.18.dev0 docs. [Online]. Available: https://scikit-image.org/docs/dev/auto_examples/applications/plot_haar_extraction_selection_classification.html. [Accessed: 28-Apr-2020].

APPENDICES

Appendix A: Communications Code

pre_processing.py (A.1)

```
import cv2
import os.path

# put path where video is located
video = cv2.VideoCapture("\\Cap_Stone\\Test_video.mp4")

# Use n as a counter for frames
n = 0
i = 1

while video.isOpened():
    good_frame, image = video.read()
    # check if the frame has been read, if not then break out of the loop
    if good_frame == False or n == 500:
        break
    # set modulo to 1 to collect every frame
    elif n % 1 == 0:
        image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
        # Save the frames into a specified folder
        im_dir = "\\Dest\\frame%s.jpg" % i
        cv2.imwrite(im_dir, image)
        i = i + 1
    # increment the frame count
    n = n + 1
```

sender.py (A.2) [18]

```
from socket import *
import time
import sys
import os
import fnmatch
from natsort import natsorted

# Establishing UDP connection
sock = socket(AF_INET, SOCK_DGRAM) # Create UDP socket
host = "169.254.127.187"           # Change to receiver IP
port = 12345
buf = 4096                         # buffer size of 4 kB
addr = (host, port)

# Sending images over UDP connection
path = fnmatch.filter(os.listdir("/home/pi/Desktop/CapstoneSource"), '*.png')
path = natsorted(path)
source = len(path)

n = 0
i = 0

for x in range(0, source):

    sock.sendto(path[x], addr)
    f = open(path[x], "rb")
    data = f.read(buf)

    while(x < source):

        if(sock.sendto(data, addr)):
            data = f.read(buf)
            i = i + 1

        #Delay 1ms after sending 28kB of data
        if(i > 7):
```

```

        time.sleep(0.001)
        i = 0
    else:
        break

    f.close()
    print("sent frame%s" % (x) + ".png")
    n = n + 1

    #Delay 2ms after sending 50 frames
    if (n == 50):
        time.sleep(0.002)
        n = 0

print "Closing socket"
sock.close()

```

receiver.py (A.3) [18]

```

from socket import *
import sys
import select
import os
import time

# Establishing UDP connection
host = "169.254.127.187"          # Change to receiver IP
port = 12345
sock = socket(AF_INET, SOCK_DGRAM) # create UDP socket
sock.bind((host, port))
addr = (host,port)
print("Starting up on %s; Port #s;" % (addr))
buf = 4096                      # buffer size of 4 kB

while True:
    print("Waiting to receive data")
    data,addr = sock.recvfrom(buf)
    print ("Received File:", data.strip())

```

```

f = open(data.strip(), 'wb')
data,addr = sock.recvfrom(buf)

while(data):
    f.write(data)
    data,addr = sock.recvfrom(buf)
f.close()

```

image_reader.py (A.4)

```

import sys
import os
import time
import glob
from natsort import natsorted
import cv2
from skimage import io

# set the path of the .xml file
stopsign_cascade =
cv2.CascadeClassifier('C:/Users/Public/Project/data/cascade.xml')
i = 0
clean = 0    # keeps track of non-corrupted frames
corr = 0     # keeps track of corrupted frames
empty = 0    # keeps track of empty frames
total = 0    # total frames sent
lost = 0     # determines how many frames were lost

f = 1820     # for testing purposes, set this to the number of frames being sent
# Function to determine if frame is corrupted or not
def c_corrupt(img_file):
    try:
        check = io.imread(img_file)
    except:
        return False
    return True
print('Waiting for Frames...')
while True:

```

```

try:
    files = natsorted(glob.glob('C:/Users/Public/Project/udp/*.png'))
    if (i == 0):
        time.sleep(0.1)
    for image in files:
        check = c_corrupt(image)
        if check == True:
            frame = cv2.imread(image, 0)
            if frame is not None:
                signDetection =
stopsign_cascade.detectMultiScale(frame, 1.3, 10)
                for (x, y, w, h) in signDetection:
                    cv2.rectangle(frame, (x, y), (x+w, y+h), (255,
0, 0), 2)

                    cv2.imshow('img', frame)
                    k = cv2.waitKey(35) & 0xff
                    if k == 27:
                        break
                    os.remove(image) # deletes clean frames
                    Clean = clean + 1
                    i = i + 1
                else: # deletes empty frames
                    os.remove(image)
                    empt = empt + 1
                    i = i + 1
            else: # deletes corrupted files
                os.remove(image)
                corr = corr + 1
                i = i + 1

    # Ctrl+C ends the stream and prints clean, corrupted, empty, and lost frames
except KeyboardInterrupt:
    total = clean + corr + empt
    lost = f - total
    print('\nStream Terminated\n' + 'Clean frames = ' + str(clean) +
'\nCorrupted frames = ' + str(corr) + '\nEmpty frames = ' + str(empt))
    print('\nTotal Frames Received = ' + str(total) + '\nLost Frames = ' +
str(lost))

```

Break

Appendix B: AdaBoost Cascaded Classifier Implementation

background.py (B.1)

```
import os
from natsort import natsorted

i = 0
dir = os.listdir('img_dir')
dir = natsorted(dir)

# This function runs through the negative image folder and creates a .txt file
containing all the file names
for name in ['img_dir']:
    for image in dir: # return the a list of the names in the directory path
        if i < 1300: # Check if i reaches a value to manipulate the dataset
            with open('bg_neg.txt', 'a') as file:
                file.write(name+'/'+image+'\n') # write a name to the text file
            i = i + 1
```

pos_resize.py (B.2)

```
import cv2

img = cv2.imread('stop_sign.jpg', 0) # Read the image in greyscale
# Resize the image to 50 by 50
image = cv2.resize(img, (50, 50), interpolation=cv2.INTER_AREA)
# Write the new image to the directory with a specified name
cv2.imwrite('trainingstop.jpg', image)
```

opencv_createsamples.py (B.3)

```
# creating the overlaid images and saving in info file
opencv_createsamples -img trainingstop.jpg -bg bg_pos.txt -bgcolor 0 -bgthresh 10 -
info info/info.lst -pngoutput info -maxxangle 0.5 -maxyangle -0.5 -maxzangle 0.5 -
num 2400

# creating positive samples and saving in pos_samples.vec
opencv_createsamples -info info/info.lst -num 2400 -w 25 -h 25 -vec pos_samples.vec
```

opencv_traincascade.py (B.4)

```
# training the cascade classifier to 12 stages
opencv_traincascade -data data -vec pos_samples.vec -bg bg_neg.txt -numPos 2200 -
numNeg 1100 -numStages 12 -w 25 -h 25
```

Display.py (B.5)

```
import cv2

stopsign_cascade = cv2.CascadeClassifier('data/cascade.xml')
vid = cv2.VideoCapture('test_video.mp4')

while True:
    ret, frame = vid.read() # Read the video
    frame_gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY) # Convert to grayscale
    # Detection function
    signDetection = stopsign_cascade.detectMultiScale(frame_gray, 1.3, 10)
    for (x, y, w, h) in signDetection:
        print(x, y, w, h) # Print the location of the sign, based on pixels
        # Draw rectangle upon detection
        cv2.rectangle(img, (x, y), (x+w, y+h), (255, 0, 0), 2)
    cv2.imshow('img', img) # Create a window to show output
    k = cv2.waitKey(18) & 0xff
    if k == 27: # if escape key is pressed then break
        break

cap.release()
cv2.destroyAllWindows()
```

Validation.py (B.6)

```
import cv2
import sklearn.metrics
import os
import numpy as np
from natsort import natsorted
```

```

cascade = cv2.CascadeClassifier('Cascades/Train 4/cascade.xml')

img_dir = 'Validation Data/'
images_raw = os.listdir(img_dir)
images_raw = natsorted(images_raw)
images = []
for img in images_raw:
    img = cv2.imread('Validation Data/' + img, 0)
    images.append(img)

images = np.array(images)
print(images.shape)

i = 0
y_pred = []
y_true = np.array([1] * 364 + [0] * 364)
for test_img in range(728): #images:
    signs = cascade.detectMultiScale(images[test_img], 1.3, 1)
    for (x, y, w, h) in signs:
        cv2.rectangle(images[test_img], (x, y), (x+w, y+h), (255, 0, 0), 1)
    if y_true[i] == 1:
        if len(signs) == 1:
            y_pred.append(1)
            print('Image:' + str(i), 'True')
        elif len(signs) == 0:
            y_pred.append(0)
            print('Image:' + str(i), 'False')
        elif len(signs) > 1:
            y_pred.append(0)
            print('Image:' + str(i), 'False to many')
    elif y_true[i] == 0:
        if len(signs) != 0:
            y_pred.append(1)
            print('Image:' + str(i), 'True')
        elif len(signs) == 0:
            y_pred.append(0)
            print('Image:' + str(i), 'False')

```



```
i = i + 1
```

```
auc = sklearn.metrics.roc_auc_score(y_true, y_pred)
print("AUC Score:", "{:.4f}".format(auc))
pre = sklearn.metrics.precision_score(y_true, y_pred)
print("Precision Score:", "{:.4f}".format(pre))
recall = sklearn.metrics.recall_score(y_true, y_pred)
print("Recall Score:", "{:.4f}".format(recall))
f1 = sklearn.metrics.f1_score(y_true, y_pred)
print("F1 Score:", "{:.4f}".format(f1))
```

Appendix C: Random Forest Classifier Code

random_forest.py (C.1) [19]

```
import os
import sys
import numpy as np
import joblib
import sklearn
import cv2
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import roc_auc_score
from skimage.transform import integral_image
from skimage.feature import haar_like_feature
from skimage.feature import haar_like_feature_coord
from sklearn.metrics import precision_score
from sklearn.model_selection import GridSearchCV

# Integral Image Function
def extract_feature_image(img, feature_type, feature_coord=None):
    ii = integral_image(img)
    return haar_like_feature(ii, 0, 0, ii.shape[0], ii.shape[1],
feature_type=feature_type, feature_coord=feature_coord)

# Load dataset
images = np.load('dataset_25x25.npy')
print("Dataset Size:", images.shape)
feature_types = ['type-2-x', 'type-2-y']

# Extract features from each image in dataset
X=list()
i = 1
for img in images:
    x=np.array(extract_feature_image(img, feature_types))
    X.append(x)
    if(i%100 == 0):
        print("done" + str(i))
        np.save('X', X)
```

```

i = i + 1

# Define y true values
y = np.array([1] * 1200 + [0] * 1200)
# Dataset train, test, val split
X_train, X_test, y_train, y_test = train_test_split(X, y, train_size=2040,
random_state=0)
feature_coord, feature_type = \
    haar_like_feature_coord(width=images.shape[2], height=images.shape[1],
                           feature_type=feature_types)

# Grid Search (Commented out because it only needs to be done once)
#parameters = {'max_features': [10, 100, 1000], 'n_estimators': [5, 10, 100],
'max_depth': [3, 5, 20, None]}
#clf = GridSearchCV(estimator=RandomForestClassifier(), param_grid=parameters,
scoring='precision', verbose=2)
#clf.fit(X_train, y_train)
#print(clf.cv_results_)
#print(clf.best_params_)
#print(auc_full_features)

# Define model
model = RandomForestClassifier(n_estimators=100, max_depth=None, max_features=100,
n_jobs=-1, random_state=0)
# Train model
model.fit(X_train, y_train)
#Print AUC score
auc_full_features = roc_auc_score(y_test, model.predict_proba(X_test)[: , 1])
print(auc_full_features)

# Save model
joblib.dump(model, 'model_1.pkl')

# Extract most important features
idx_sorted = np.argsort(model.feature_importances_)[:-1]
cdf_feature_importances = np.cumsum(model.feature_importances_[idx_sorted])
cdf_feature_importances /= cdf_feature_importances[-1] # divide by max value

```

```

sig_feature_count = np.count_nonzero(cdf_feature_importances < 0.80)
sig_feature_percent = round(sig_feature_count /
                             len(cdf_feature_importances) * 100, 1)
print('{} features, or {}%, account for 80% of branch points in the '
      'random forest.'.format(sig_feature_count, sig_feature_percent))
feature_coord_sel = feature_coord[idx_sorted[:sig_feature_count]]
feature_type_sel = feature_type[idx_sorted[:sig_feature_count]]

# Extract important Haar features
X=list()
i = 1
for img in images:
    x=np.array(extract_feature_image(img, feature_type_sel, feature_coord_sel))
    X.append(x)
    if(i%100 == 0):
        print("done" + str(i))
        np.save('X2', X)
    i = i + 1

# New train, val, test split
y = np.array([1] * 1200 + [0] * 1200)
X_train, X_test, y_train, y_test = train_test_split(X, y, train_size=2040,
random_state=0)

# Retrain model
model = RandomForestClassifier(n_estimators=100, max_depth=None, max_features=100,
n_jobs=-1, random_state=0)
model.fit(X_train, y_train)
auc_full_features = roc_auc_score(y_test, clf.predict_proba(X_test)[: , 1])
print(auc_full_features)

# Define sliding window parameters
x_cord = []
y_cord = []
step_ratio = 25

# Load Image

```

```

img = cv2.imread("1.jpg", 0)
image_size_y, image_size_x = img.shape
print(image_size_y, image_size_x)

# Apply sliding window and image pyramid
for dim in (image_size_x*0.10, image_size_x*0.15, image_size_x*0.20):
    for y in range(0, int(image_size_y) - int(dim), step_ratio):
        for x in range(0, int(image_size_x) - int(dim), step_ratio):
            image_crop = img[y:y + int(dim) , x:x + int(dim)]
            image_resized = cv2.resize(image_crop, (25, 25),
interpolation=cv2.INTER_AREA)
            input = extract_feature_image(image_resized, feature_types)
            input = input.reshape(1, -1)
            result = model.predict(input)
            print(result, x, y)
            if result == 1:
                x_cord.append(x)
                y_cord.append(y)

# Draw rectangles
for x, y in zip(x_cord, y_cord):
    cv2.rectangle(img, (x, y), (x + dim, y + dim), (255, 0, 0), 2)

# Save image
cv2.imwrite('image_test.jpg', img)

```

Appendix D: Device Specifications and Project Schedule

Device Specifications (D.1)

Device	CPU	RAM
Raspberry Pi Model 3B+	<ul style="list-style-type: none">- ARM Cortex-A53- Quad core- 64-bit- 1.4GHz	1 GB LPDDR2 SRAM
Remote PC	<ul style="list-style-type: none">- AMD Ryzen 5 1600- 6 cores, 12 threads- 64-bit- 3.2GHz	16 GB DDR4 SDRAM

Project Schedule (D.2)

Project Milestones	Date Achieved
<ul style="list-style-type: none">- Send an image from the embedded system to the remote device	May 19th, 2020
<ul style="list-style-type: none">- Pre-process each simulated video frame into a file that fits size constraints	May 24th, 2020
<ul style="list-style-type: none">- Remotely send a continuous series of images from the embedded system to the remote device	June 7th, 2020
<ul style="list-style-type: none">- Develop an image detection algorithm to detect single images already stored on the remote device	July 18th, 2020
<ul style="list-style-type: none">- Create software to accept remotely sent images and run image detection algorithm	July 20th, 2020
<ul style="list-style-type: none">- Benchmark classification accuracy and improve any flaws	July 22nd, 2020
<ul style="list-style-type: none">- Simulate a live video of a dash cam on the embedded system	July 24th, 2020

- Benchmark and optimize the image detection algorithm to detect the live video feed	July 26th, 2020
- Image detection for stop sign	July 18th, 2020
- Image detection form yield sign	Not completed
- Image detection for lane-ending sign	July 20th, 2020