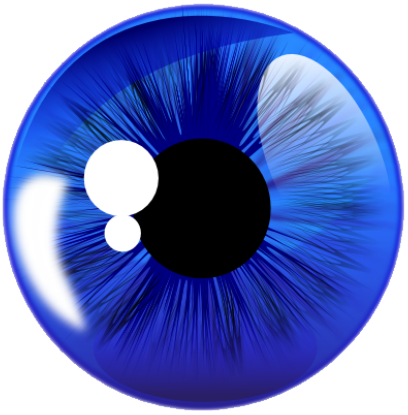


DeepEye

Deep Vision-Based Copilot
Advanced Driver-Assistance Systems

A Capstone Project
by

Thayer Alshaabi & Samuel Poquette

Submitted to the Faculty of the
Department of Software Technology
in partial fulfillment of the requirements for the degree of

BACHELOR OF SCIENCE
IN
COMPUTER SCIENCE And INNOVATION

Champlain College
2018

Abstract

This research project introduces a new framework for Advanced Driver-Assistance Systems (ADAS) called DeepEye. DeepEye is a computer vision-based copilot system powered by a deep convolutional neural network to perform real-time scene recognition. The system uses Tensorflow to classify relevant objects surrounding the driver. It also uses a set of image processing functions from the Open Computer Vision library to estimate whether the car is staying in the lane. It then alerts the driver if an object or a situation poses a potential threat through a custom-designed dashboard (warning interface). According to our testing results, we believe that our system would be a useful backup solution for automobiles equipped with ADAS sensory-based systems, particularly when sensors' feeds are not available. We evaluated the system performance on an hour's worth of driving videos running on a Titan XP GPU. We concluded that the system is up to 92% accurate in various weather conditions, times of day, and geospatial locations. This includes heavy thunderstorms, crowded areas, tunnels and highways, based on footage collected from both dash-cam feeds and synthetic video data from Grand Theft Auto V. We strongly believe the framework could be further improved by replacing our single-vision camera approach with multiple stereo-vision cameras to account for blind-spots and 3-dimensional depth perception. Furthermore, advanced sensors such as LIDARs and RADARs would also enhance our scene recognition, as they can accurately estimate trajectory and distance. We also faced several issues due to hardware limitations, which is understandable given the notion and complexity of image processing on a single GPU. With the right hardware onboard, DeepEye is an effective framework that contains most of the fundamental basis for real-life copilot and autopilot systems.

Acknowledgements

We gratefully acknowledge the support of NVIDIA Corporation with the donation of the Titan Xp GPU used for this research.

We would like to express our deep gratitude to Professor David Kopec, our capstone advisor, for his patient guidance, enthusiastic encouragement and useful critiques of this research work.

We would also like to express our very great appreciation to Dr. Joshua Auerbach for his valuable and constructive suggestions during the planning and development of this research work. His willingness to give his time so generously has been very much appreciated.

We would like to offer our special thanks to Professor Jonathan Ferguson for helping us to set up a workstation to use in our project.

Finally, we wish to thank our family, friends, and anyone who helped to make this vision become reality.

Thank you very much, everyone!

Thayer & Sam

Table of Contents

List of Figures

Chapter 1: Introduction

1.1 Problem Statement

Our project aims to successfully build, then evaluate a vision-based scene recognition model that classifies relevant objects surrounding a driving agent, and accurately estimates whether the driver is staying in lane or not. The system would analyze the scene, then alert the driver if any of the objects pose a potential threat through the dashboard.

1.2 Purpose Statement

This project will demonstrate how a vision-based system can be utilized for scene understanding when sensor-based systems are not available for autonomous driving.

1.3 Context

Ever since the concept of autonomous vehicles was introduced to the IT industry as an application of artificial intelligence, there have been tremendous improvements in computer vision and pattern recognition algorithms. However, recent progress in autonomous models has been extensively criticized by many experts in various disciplines for being too expensive, due to the time and the cost associated with human labor to collect massive training datasets, not to mention the complexity of such systems. Fortunately, video games are utilized to reduce the cost of collecting large training datasets, as well as providing a decent simulated environment to test models in their early stages. Surprisingly enough, some of the statistical reports show that supplementing a small percentage of real-world data with massive datasets acquired from video games would significantly improve an agent's performance.

In the last five years, manufacturers have demonstrated the potential of autonomous cars on highways. However, navigating noisy urban areas remains an unsolved problem. This is mainly due to the complexity of scenario-based scene recognition, and 3D traffic scene understanding. One of the solutions is to rely on GPS localization systems and advanced on-board sensors to create 3D maps of the environment. However, constructing accurate maps is nearly impossible in practice. Also, GPS signals are not always available, and the system would potentially fail in underground areas like tunnels, and in severe weather conditions. There are also many ethical and security issues that exist with current models. Hence, a combination of vision-based models in addition to sensory-based systems would certainly help to overcome many of the current challenges.

1.4 Significance of Project

Autonomous vehicle engineering appears to be a dominant industry in the future of information technology and artificial intelligence. Demonstrating how an autonomous agent can learn in an efficient, safe, and pragmatic virtual environment will have tremendous potential to revolutionize machine learning algorithms and computer-vision methodologies. Our project is mainly focused on real scene layout recognition through computer vision that would ultimately make use of advanced sensors (i.e. LIDAR) if the models were to be used for 'real-world' applications. However, our work will also train and test vision-based models for an autonomous copilot system without sensory inputs. We will provide several statistical evidences to demonstrate the performance of the system and its potential to generalize its learned model to real life driving environments.

Chapter 2: Related Work

2.1 Deep Convolutional Neural Networks

In 2014, [2] introduced a novel probabilistic generative model for multi-object traffic scene understanding. Their model does not rely on GPS, LIDAR or map knowledge. Instead, it takes advantage of a well-defined set of visual cues in the form of vanishing points, semantic scene labels, scene flow, and occupancy grids. They examined their model on 113 representations of intersections, and showed that their approach can recognize urban intersections reliably with an accuracy of up to 90%. For most of these intersections, traditional approaches based on lane markings and curbstones would have failed due to the absence of these feature cues. They ultimately proposed that utilizing Markov Chain Monte Carlo sampling to evaluate traffic scenes would improve it in terms of object detection and object orientation estimation in challenging and cluttered urban environments. In our project we will be considering a similar approach due to the lack of on-board sensory input to help us create an accurate 3D map of the scene. Instead, we will rely solely on visual cues to estimate the layout of urban intersections based on visual representation of the scene alone.

Later on, in 2015, [16] at Google proposed Inception, a 22-layer deep convolutional neural network architecture for classification and detection. While using 12 times fewer parameters than the previous state-of-the-art architecture of [7] (also known as AlexNet), the network was able to be significantly more accurate. The team adopted the network-in-network approach to amplify the representational power of neural networks. However, as noted by the development team there are two issues that come with massively DNNs architecture: the big challenge of overfitting, which is typically associated with networks that have a huge number of parameters, and the intensive use of more computational resources as the networks gets bigger and bigger. Interestingly, the team has managed to overcome both challenges by replacing the fully connected layer in the CNN with multiple, yet simpler spare layers, and spreading them across the network architecture.

The current state-of-the-art deep neural network for object detection is noted in the Regions with Convolutional Neural Networks (R-CNN) architecture by [3] at UC Berkeley. The team developed a very interesting approach by breaking down the problem of object detection into two subcategories: utilizing very sophisticated region proposal algorithms to generate accurate bounding boxes, and optimizing CNN classifiers to identify object classes and categories. However, R-CNNs were initially computationally expensive due to the massive amount of resources spent on region proposals. Fortunately, the design was further improved and optimized at Microsoft by [13]. The team proposed a Region Proposal Network (RPN) architecture, which performs both object localization through bounding boxes and classification simultaneously at each position. The model was fully optimized to run in parallel on GPUs at near real-time frame rates. Their source code is available @RPN-Python.

There are many other types of deep CNNs that may, or may not, share some of the attributes of the models noted above. However, in our project we will be looking closely at these three architectures (AlexNet, Google's Inception, and RPNs) as some of the top-5 state-of-the-art object detectors to figure out the model that would be best suited for our application in the settings of vision-based autonomous driving. [4]

2.2 Deep Learning Optimization

In the summer of 2015, [8] revealed that Deep Neural Networks (DNNs) are easily fooled to mistakenly classify an unrecognizable image as some other recognizable object with high confidence prediction by distorting the image using evolutionary algorithms. For example, it is fairly easy to artificially generate images that are totally unrecognizable to humans (e.g. a random collection of black and white distorted lines), but DNNs would mistakenly classify such images as recognizable objects (e.g. labeling distorted black and white images as a penguin) with above 90% confidence. That leads to the bigger question in our project, which relates to the efficiency of utilizing video games for training and testing computer vision models for autonomous driving. Particularly, how much noise in an image is acceptable until a DNN trained with slightly noisy data generated from video games starts to mislabel new datasets. We will try to investigate this issue further in our project.

Remarkably, there has been a huge debate in the industry whether computer-generated (CG) imagery has enough resemblance to real-life images in order to improve the performance of computer vision models in practice. The paper by [15] presented several experiments to demonstrate that systems trained on synthetic RGB images may be used to improve the performance of CNNs on both image segmentation and depth estimation. They trained a ConvNet with over 60,000 frames that are extracted from Grand Theft Auto V (GTA-V), and showed that a CNN trained on synthetic data achieves a similar test result to a network that is trained on real-world data. They advocated that a mixed training dataset extracted from the real world and the state-of-the-art game engines gives the most improvement. We will be investigating a hybrid training strategy, by training our model on both datasets from the real world as well as images synthesized by video games and test its performance.

Interestingly, [14] at Intel Labs used Grand Theft Auto V to create a large training data set for computer vision applications. They explained that deep neural networks like Convolutional Neural Networks (CNNs), and Recurrent Neural Networks (RNNs) have been proven over and over to be the best solution in almost all computer vision problems. One of the key factors of constructing successful deep artificial neural networks (DNNs) is the abundance of massive datasets for both training and testing. However, collecting large datasets has been very expensive and time consuming. For example, each frame has to be traced and labeled, which took between 60 and 90 minutes per frame. Hence, tremendous efforts have been devoted recently to finding efficient and cost-effective ways to collect

sufficiently large datasets. The team created a software layer that sits between the game and the hardware that extracts frames from the game before they are rendered. The frames are automatically labeled for all the different objects in view using a hash function. This can then be fed to a machine-learning algorithm, which will identify the objects. The team gathered 25,000 images from the game, and labeled them all within 49 hours. Using a traditional approach would have taken at least 12 years. The team also found that models trained with $\frac{2}{3}$ game data and $\frac{1}{3}$ CamVid (dashcam footage) data outperformed models trained entirely with CamVid data alone. Their Sample code for reading the label maps and a split into training/validation/test set is available @IntelLabsGTAV.

Finally, [5] at the University of Michigan expanded on the work from the team at Intel Labs. They created a fully automated system that gathered and annotated training data for other vehicles, so that humans did not have to be in the loop during the annotation. They only used data gathered from synthetic environments, rather than $\frac{1}{3}$ real-world data. They also used open-source plugins to capture the data from the GPU buffer data. This data is analyzed to detect vehicles, and computations are made to draw tight bounding boxes around any detected vehicle without any human intervention. As previously stated, this program only detects other vehicles, so practical usage of this would require expansion for pedestrians, objects in the road, etc. However, this research shows that a training data set extracted solely from synthetic environments is viable for object detection. Code and dataset for reproducing the results is available @DrivingInTheMatrix.

2.3 Deep Learning Frameworks

As discussed before, there are many different CNN architectures. Our intention was to select the best-suited architecture (particularly the right speed/accuracy balance for our base model) that would serve efficiently given our target platform and limited hardware. As expected, we ended up experimenting with many different pre-trained models. During our experiment, we came across a very interesting CNN model called YOLO Net [11, 12]. The model provided there would be by far the best real-time object detection system achieving great speed/accuracy balance. However, there are two problems with using it: first, the model is built off of Darknet [10] and written in C. We would need a framework like Darkflow to act as a bridge to translate it to Tensorflow [1] and Python. Second, Darknet is only supported by a small community as opposed to Tensorflow, which has a huge community of developers at Google and around the world. Given that our intention is to make an open source platform for ADAS, we chose to use Tensorflow, which would guarantee better support in the long run.

Chapter 3: Methods

3.1 Design

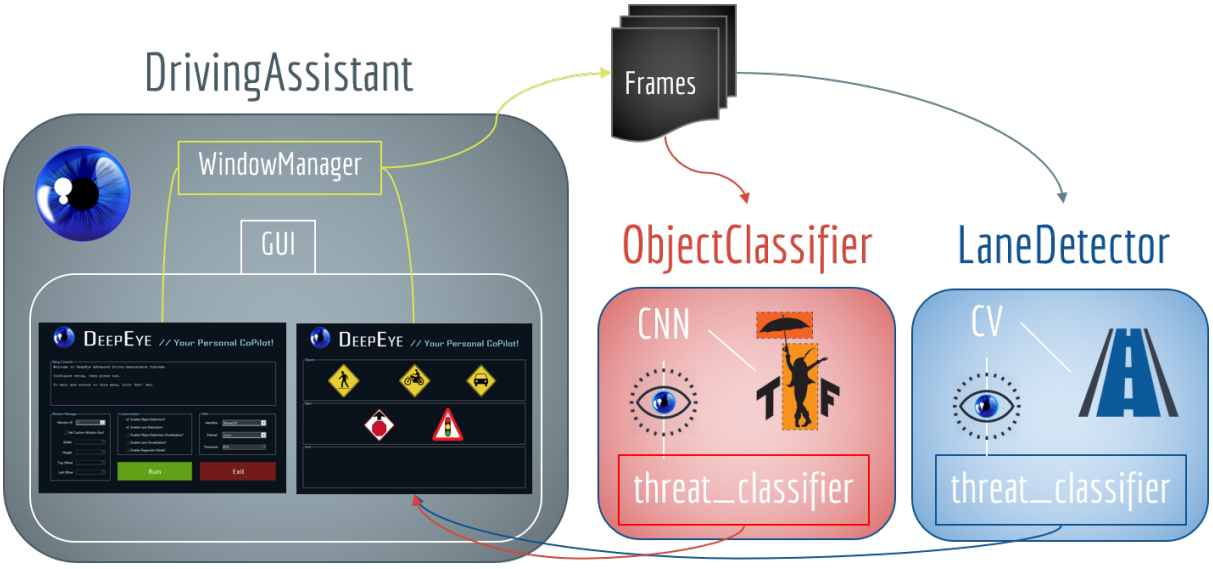


Figure 3.1: System Architecture

Summary Our system consists of three major layers. In the first layer, the *DrivingAssistant* initiates the system, and launches the *WindowManager* to start capturing frames from the screen and feeding them into our *ObjectClassifier* and *LaneDetector*. Once both classes are done with their scene recognition process, each class updates the warning interface accordingly to keep the driver safe.

Stage 1: Copilot (Driving Assistant) Our system works in a cycle. Our master class where the cycle starts is the *DrivingAssistant*, containing the main interface and warning interface. The main interface controls any 'pre-launch' system customization that will be covered in detail later on in the user interface section. The *DrivingAssistant* also contains the *Copilot-Dashboard (warning interface)*, which issues the driver warnings through artificial audible and/or visual cues. More importantly, the *DrivingAssistant* initiates the *WindowManager* that captures frames from a given monitor, and sends a copy of each frame to the next layer in our system: *ObjectClassifier* and *LaneDetector* respectfully in that order. In a real-world application, the *WindowManager* would capture frames using a camera mounted in the center of the car. But, for the purpose of simplicity and ease of testing, we capture the frames directly from a monitor playing a video of driving footage.

Stage 2: Object Detection This class is basically a deep convolutional neural network mainly used for object detection and image classification. The network takes in a raw pixel image, and then classifies it as one of the labels defined in the dataset. In practice, our CNN would take in a *numpy* array of pixel data as input streams, and draw bounding boxes around the objects that were detected by the network bundled with its confidence in that prediction. Then, we will track the bounding boxes that were generated by our object detector over a sequence of frames to check if an object or a situation may pose a potential threat for the driving agent. Lastly, the results of our scene analysis system will then be illustrated in the warning interface that notifies the driver of any upcoming threats.

Stage 3: Lane Detection This class uses a set of functions in the Open Computer-Vision library [9] to detect the current lane that the car is driving in, then highlights both the road markers, as well as, the area enclosed by your lane onto the given frame. Thereafter, the class relies on the fact that the camera capturing those frames is mounted relatively in the center of the car, and it measures how well the driver is staying centered in the lane. Then, it updates the warning interface to alert the driver when the car is slightly off lane or completely departing the current lane.

3.2 Frameworks

Python: Python, specifically 3.6, will be our primary language for coding. We do not anticipate using any other programming language, as Python suits our needs. Python is a great language for Machine Learning programs such as ours. Python is well designed, robust, scalable, and fast - some of the important factors for applications in artificial intelligence. Being open source is also helpful, as there is a lot of support online amongst its users.

CUDA: CUDA is a parallel computation API created by Nvidia that optimizes code by processing it through the GPU(s), in order to harness their computational power (Titan XP and GTX 1080, respectively). Using GPU's greatly improves the performance of Machine Learning over standard CPU usage, which is confined to sequential processing.

TensorFlow: TensorFlow is an open source math library developed and used by Google. Amongst other things, TensorFlow provides a framework for neural networks, making it very popular in Machine Learning applications, and essential to our project. We are using a version of TensorFlow that works alongside CUDA in order to utilize our GPUs efficiently [1].

Tensorflow Object Detection API: We used the newly released detection platform in Tensorflow [4]. This platform provides pre-trained models, training and hyperparameter tuning pipelines for (Faster R-CNN, SSD, and R-FCN) network meta-architectures

coupled with various feature extractors like (Resnet-101, VGG-16, Inception v2-v3, and some others). That said, some of the implementations of the neural networks mentioned above are slightly modified to optimize their performance. Notably, all models were trained and tested on a TITAN X GPU, which is very convenient, as it would minimize any potential factors to affect our results due to hardware differences. Ultimately, our implementation uses the (faster-rcnn-resnet101 — mAP=32) model. It's a fair compromise to achieve a relatively high mean average precision (mAP) without slowing down the model's performance too much. In other words, it provides the right balance of speed and accuracy for our base model that would serve efficiently given our target platform and limited hardware.

OpenCV: OpenCV, as the name suggests, is an open source computer vision software library, originally developed by Intel. It is mainly used in tandem with Machine/Deep Learning frameworks, such as TensorFlow. This library will be used to aid the development of our object recognition algorithms by providing effective and easy-to-use tools for capturing screen frames and modifying them as needed.

TKinter: Tkinter is a Python library for building GUI's. It is the default method for doing so, as it comes installed with Python. There are other more advanced GUI libraries for Python, but we used Tkinter because it was easy to learn; neither of us had much experience creating GUI's. It also allowed us to use Page, a drag-and-drop GUI builder, which generated Python code using Tkinter. This saved us time by generating some boilerplate code for widget placement and positioning. Most of the functionality still needed to be coded, and eventually most of the original code generated by Page was refactored and expanded.

3.3 Algorithms

3.3.1 Deep Convolutional Neural Networks

CNNs are customized and fully optimized feed-forward artificial neural networks that have outperformed the state-of-the-art techniques for detecting visual imagery and classifying them with very high level of accuracy. There are four fundamental types of layers that a CNN may consist of: Convolutional, Rectifier, Pooling, and Fully Connected layers, where the output of each one of these layers is fed into the next layer in a network architecture accordingly [6].

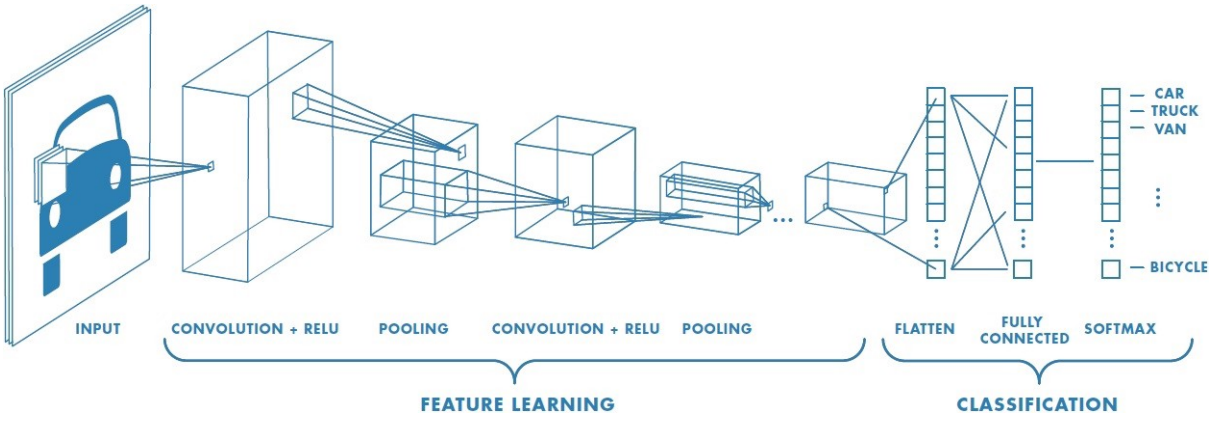


Figure 3.2: Deep Convolutional Neural Networks Architecture

Convolutional Layer (CONV) computes the corresponding output of neurons that are connected to a set of positions in the input matrix.

Normalization Layer also known as the Rectifier Layer (RELU) works as a filter by applying a threshold that would keep the volume of the image unchanged but implicitly performing an element-wise activation function to format the result statistically as a value ranging between $0 \rightarrow 1$.

Pooling Layer (POOL) reduces the complexity of image by downgrading the spatial dimensions, specifically height and width.

Fully Connected Layer (FC) classifies the given object against a set of predetermined classes by estimating the accuracy scores for each class, then choosing the one with the highest probability.

3.3.2 Color Thresholding & Edge Detection

Additionally, We used a combination of color and gradient thresholds along with an edge detection algorithm to create a bitmap of zeros and ones that locates the exact pixels of the road markers.

Camera Calibration and Frame Undistortion: 3D objects are normally transformed into 2D projections of the objects in a 2D-space, which causes various levels of distortion depending on the camera used to take the picture. We start by preparing (object points), which will be the (x, y, z) coordinates of the chessboard corners in the original image. We then used the output *object_points* and *image_points* to compute the camera calibration and distortion coefficients using the **cv2.calibrateCamera()** function. We applied this distortion correction to the test image chessboards using the **cv2.undistort()** function and obtained this result:

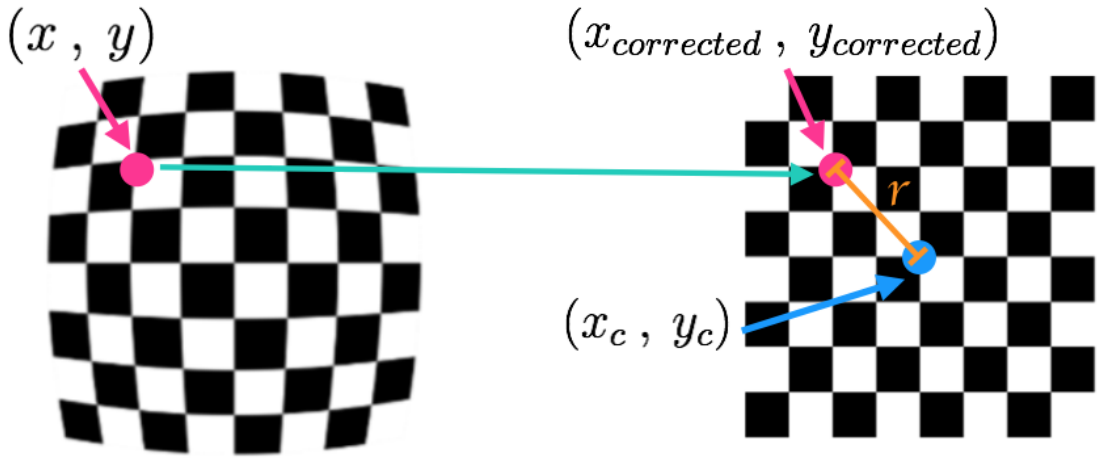
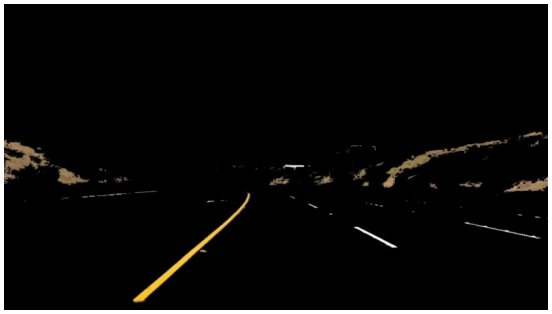


Figure 3.3: Camera Calibration and Frame Undistortion

HSV Mask: First, we applied an HSV mask to the undistorted image, which helps to extract the yellow lines by focusing on the hue and saturation of the pixel and not so much on how dark the pixel might be.

Canny Edge Detection: It is a technique widely used in computer vision systems to essentially extract structural information from an image and dramatically reduce the amount of data that needs to be be processed. We applied this operator only to the bottom half of the image assuming that our region of interest is only the lower section of the input image where lane markings are expected to be.

Gaussian Noise: Then, we take the resulting image from our edge detection algorithm and apply **cv2.equalizeHist**, followed by **cv2.GaussianBlur**, which improves the contrast of the image to easily detect any white road marks and remove any noise in the image.



(a) HSV Mask

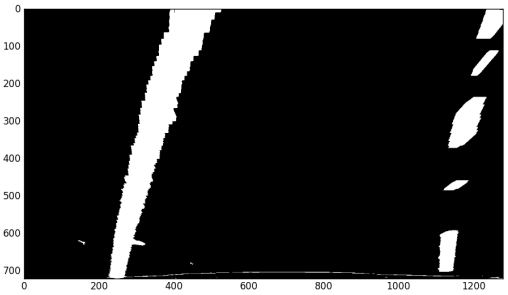
(b) Canny Edge Detection

(c) After Removing Gaussian Noise

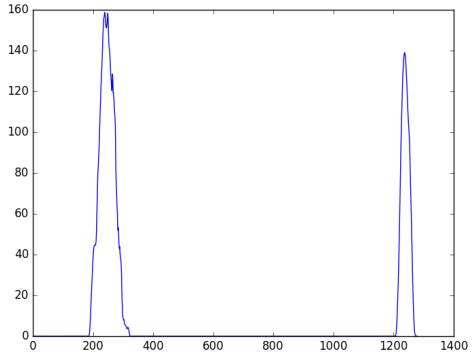
Figure 3.4: Color Thresholding & Edge Detection

Perspective Transform: This is a very basic, yet crucial step to figure out the lane curvature. The idea is to map the pixels in a given image to a bird's-eye view, in order to view a lane from above. This requires **cv2.getPerspectiveTransform** and **cv.warpPerspective**.

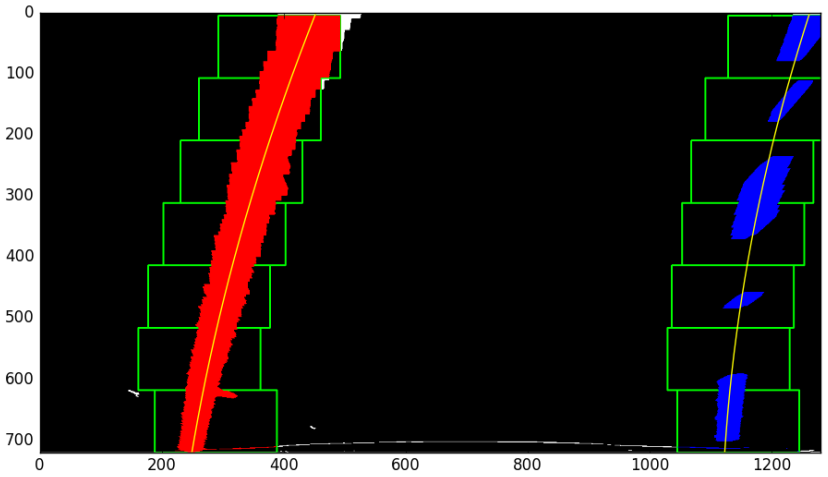
Sliding Window Search: For this step, we apply an intensive search to identify the exact pixels corresponding to the road marks. Starting from the very bottom of the image, we insert two windows, one for each peak point of the histogram of the bitmap. We then adjust the window's position based on the average density of pixels within the given window and slide the windows upwards until we reach the end of the image. Lastly, we draw a line through the center points of the windows to represent the lane boundary.



(a) Bird's-eye view (bitmap)



(b) Bitmap Histogram



(c) Bird's-eye view (lane detected)

Figure 3.5: Sliding Window Search

Sliding Window Search Optimization: Initially, the lane detector was hard-coded to detect lane-pixels by focusing on the bottom-half of the frame. Then, we updated our *LaneDetector* class to adopt various frame sizes. It now captures one-third of the frame vertically, and two-thirds of the frame horizontally starting from the center (illustrated in the graph below).

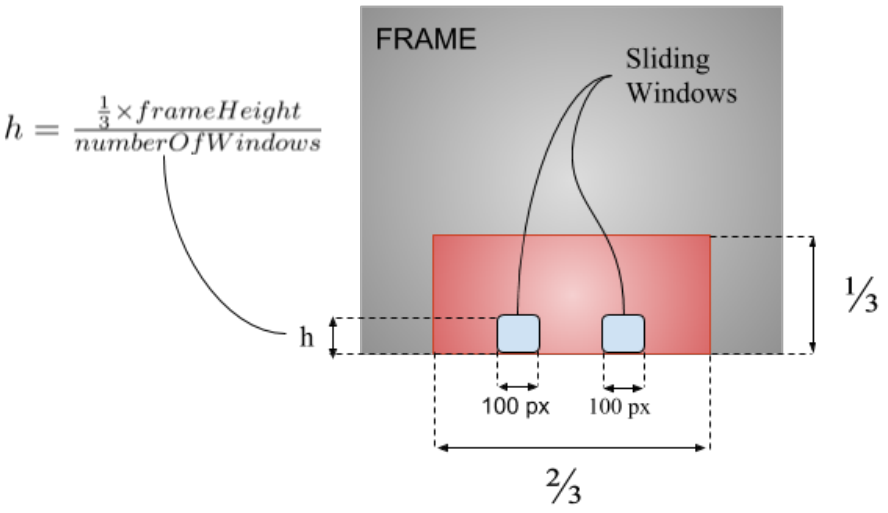


Figure 3.6: Sliding Window Search Optimization

Collision Detection We limited pedestrian warnings to people who are crossing in front of the driver only. Furthermore, we limited vehicle and bike warnings by excluding any objects that are detected outside of a predefined area in the given frame (our main region of interest (ROI) highlighted in yellow in the graph below). This is so warnings are not given for distant objects. In addition, we created a collision warning, which alerts the driver if there's a potential collision with an object very close to the car. This uses an even smaller box (the collision ROI) at the bottom of a given frame, and checks if the bottom of an object was detected within the **COLLISION ROI** highlighted in red in the graph below.

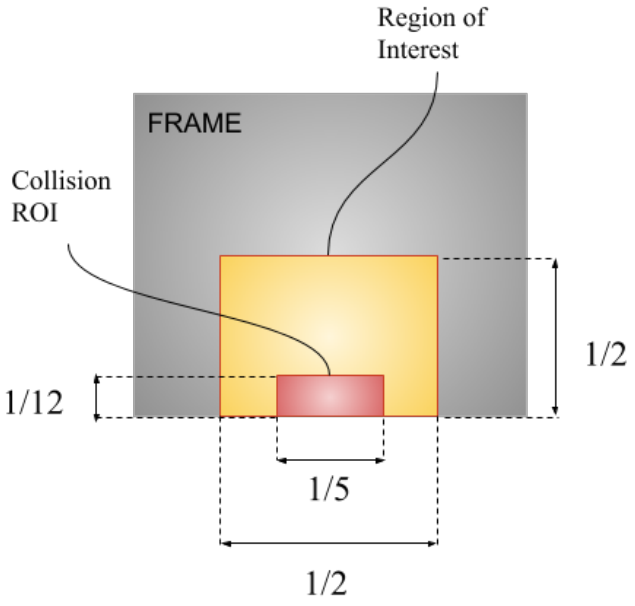
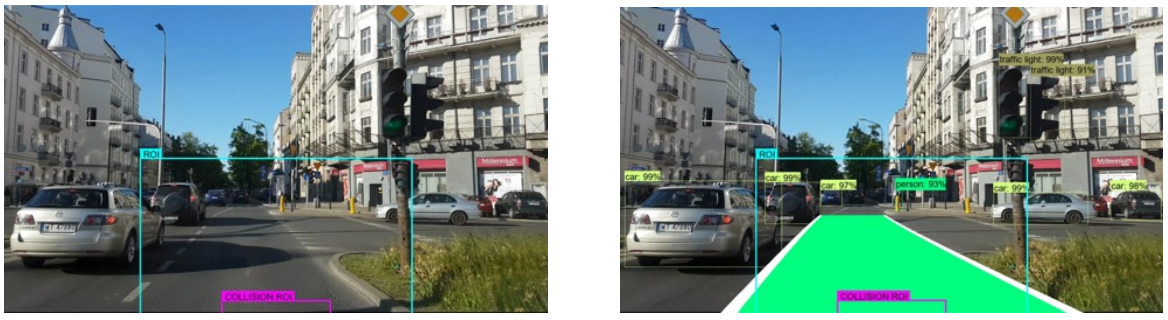


Figure 3.7: Detection — Region of Interest (ROI)

3.4 Analytical Methods

Benchmark: We added a diagnostic mode option that allows us to run some testing scripts to evaluate the system performance in various situations. When you run the system in diagnostic mode, the system will take 10-15 screenshots every minute (varies based on the frame rate). Each screenshot demonstrates a given frame captured by our system prior to our sense analysis and after our detection and classification.



(a) Pre — Scene Recognition

(b) Post — Scene Recognition

Figure 3.8: Diagnostic Mode

Then, we made a **pandas** data frame that stores the ID of the captured frame along with a set of attributes that were detected in the given frame. Each screenshot will be associated with a data entry that gets generated automatically to describe the given frame, and will eventually save to a CSV file with a timestamp to indicate when it was done. Finally, each data entry in our log files will get an "accuracy" score based on the true/false positive predictions, then we will calculate an average score to express how well the system is doing.

ID	PEDESTRIAN		VEHICLES		BIKES		STOP_SIGN		TRAFFIC_LIGHT		OFF_LANE		COLLISION		ACCURACY
	T	P	T	P	T	P	T	P	T	P	T	P	T	P	SCORE
1	1	1	1	1	1	0	0	0	0	1	0	0	0	0	0.857143
2	0	1	1	1	0	0	0	0	0	1	1	0	0	0	0.571429
3	1	1	1	1	0	0	0	0	0	0	1	1	0	0	1.000000
4	0	1	1	1	0	0	0	0	0	1	0	0	0	0	0.714286

Figure 3.9: Data Log

Practical/Visual Testing: Additionally, we will test the program ourselves and compare the output with what we would have estimated if we were driving the car.

Hands-On Experience: Lastly, we evaluated our system in the settings of Grand Theft Auto V, as it was gratifying to watch the system in-action with somebody driving a vehicle in the game with DeepEye operating behind the scenes as a copilot assisting the driver. Being a virtual environment, this also allowed us to easily test in many different driving environments, weather conditions, and times of day.

3.5 Features

Object/Lane Detection & Scene Recognition: This will be our most fundamental feature, which is required for all else to function. TensorFlow and OpenCV will be used to do this, using a Convolutional Neural Network. This system will create tight bounding boxes around objects, and classify them as other vehicles, pedestrians, bicycles, lanes, street signs, etc.

Copilot – Advanced Driver-Assistance System (ADAS): This feature determines if the objects detected pose a potential threat to the driver. To do this, it takes the type of object and its trajectory into account. Context is important; oncoming traffic in the other lane will get close to the driver, but does not normally pose a threat. In this case, trajectory is most important. As long as the other car does not intersect with the driver's trajectory, there is no threat posed. If an potential collision is detected, an audible warning will alert the driver. The following is a list of warnings given:

- Pedestrians nearby
- Vehicles nearby
- Bikes/Motorcycles nearby
- Stop sign ahead
- Traffic light ahead
- Forward collision warning
- Lane departure warning

3.6 Test Plan

An important part of the project will be analyzing the accuracy of our scene recognition. The system would be easily tested on videos that are publicly available online (collected from a camera mounted on the front windshield of a vehicle). Our plan is to test our system on a set of videos; we will then provide some statistical charts for the accuracy of our object/lane detectors. Once the system passes the object detection test, we will evaluate the system's performance ourselves visually by comparing the output of the warning interface to what would be expected if we were in the same situation. Through extensive visual testing, we would be able to highlight any unnecessary warnings and/or outliers in the system. At the final testing stage, we demonstrated our system in Grand Theft Auto V, which allowed us to control the environment surrounding our agent, and test our system on various weather/lighting conditions.

3.7 Criteria and Constraints

The biggest constraint is our limited access to powerful hardware resources. Notably, most machine learning applications, particularly the ones that rely on computer vision, require advanced and super optimized GPUs to process the frames that are imported into the system accurately and efficiently. Unfortunately, we only had a single Titan Xp, while a real-time ADAS would probably use a set of Tesla/Quadro GPUs, which are more powerful and highly optimized for deep-learning and computer-vision tasks.

Chapter 4: Results

4.1 Final User Interface

As previously mentioned, we used Tkinter for creating the interface, with some code generation from Page, a drag-and-drop Python GUI program. Our original interface consisted simply of a setup window for choosing parameters to send to the Driving Assistant. This was because we were constantly changing the parameters during development, and since they were hard coded, we would have to go back and change the code each time. The following are parameters changeable in the interface:

Monitor ID: This selects which monitor to feed frames. The default values are $0, 1, and\ 2$. For the typical dual-monitor setup, 0 is both, whereas 1 and 2 are the other individual monitors.

Custom Window Size: If the Set Custom Window Size button is not checked, then the frame size will default to the size of the entire monitor that's feeding the visual stream. If it's checked, this allows the user to enter a custom size (a smaller window size results in better performance).

Top/Left Offset: By default, the visual stream will be feeding from the very top left corner on the monitor, even with a custom smaller window size. Adding an offset allows the user to change what part of the screen is being fed to the visual feed.

Enable Object Detection: Toggles whether to run object detection.

Enable Lane Detection: Toggles whether lane detection runs – will significantly decrease FPS.

Enable Object Visualization: Toggles whether to visualize objects – will decrease FPS.

Enable Lane Visualization: Toggles whether to visualize lanes – will decrease FPS.

Diagnostic Mode: Toggles whether to run in diagnostic mode. Every time the run method is called, a frame is exported before and after the object/lane detection. The values of the threat dictionary are added to a CSV that lists the threat dictionary results from each frame for testing purposes.

CNN: There are three options for changing the feature identifier model. The default is Resnet101, which gives us the best hardware performance. NAS is a more accurate model, however its performance is slower. Inception-Resnet is a compromise between the two, however we did not find a measurable performance benefit.

Dataset: There are two options: Coco and Kitti. Coco is an American trained dataset, with more classes, so this is the default option. Kitti has fewer classes, and was trained in Europe.

Threshold: This controls the confidence threshold cutoff for detected objects. Any object detected with a confidence below the set threshold will not be displayed onscreen. The default value is 85%.

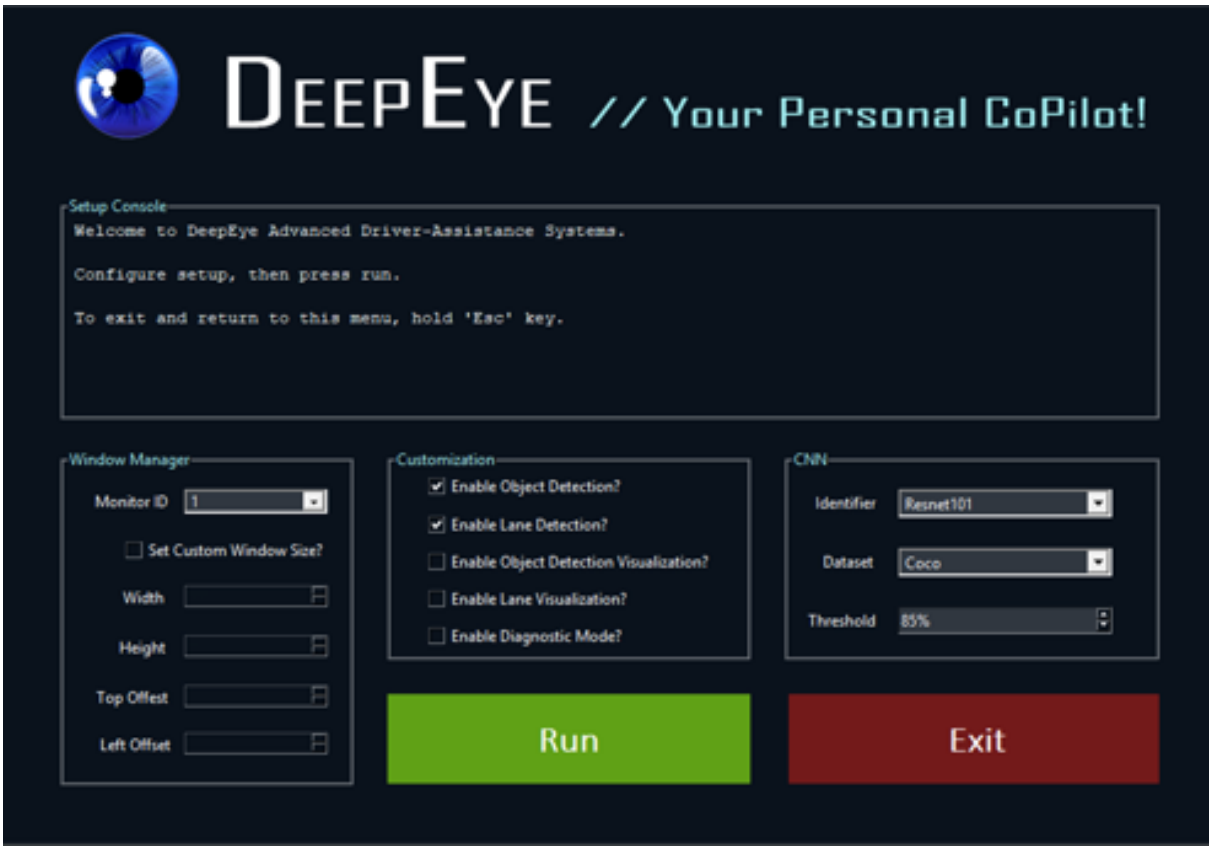


Figure 4.1: Main Interface

After the first iteration, we added a text box to display text normally output in a separate terminal window, to make it cleaner and let the user know it's running, as it can take a while to load. After setting the parameters, the user can press the "Run" button to start the program. Once it starts, the frames containing the setup widgets are hidden behind the new runtime interface. While not ideal, this was done to bypass limitations related to Tkinter, which will be further discussed below.

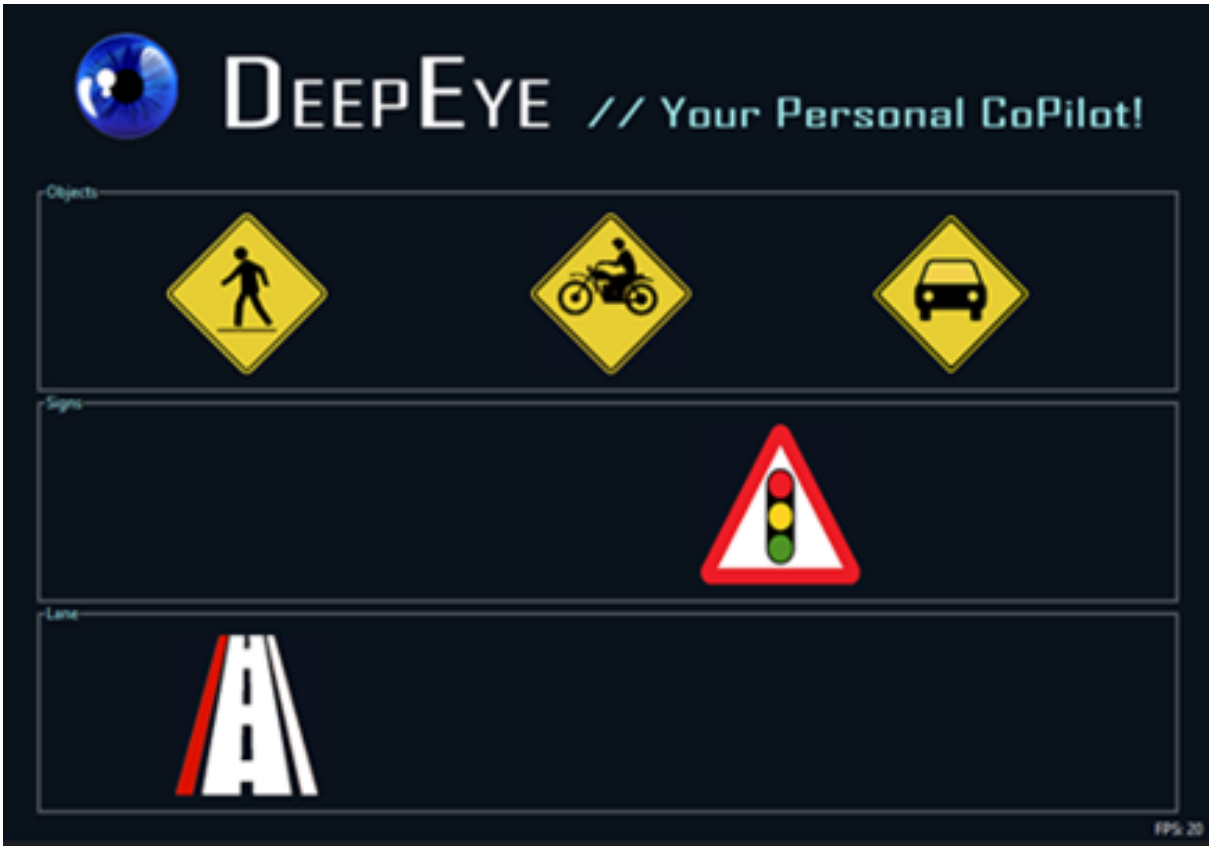


Figure 4.2: Dashboard (Warning Interface)

In the runtime interface, there are three frames: One for objects (vehicles, bikes, and pedestrians), one for signs (stop signs and traffic lights), and one for lane detection. In each frame, an icon will show up for each detected object. In the lane detection frame, an icon will display, informing the driver the position of the lane they are in. If no lane is detected, a separate icon displays. When a collision threat is detected, a warning icon is displayed while all other icons are cleared from the interface in order to bring greater attention to this. An audible beep is also played.

While the program is running, the user can press "Escape" to stop the program and return to the setup menu. There they can either edit parameters and run again, or press "Exit" to quit the program altogether.

4.2 Analytical/Testing Results

Summary: While running in diagnostic mode, two screenshots are taken every several frames; one before the object and lane detection runs, and one after, which shows the objects and lane detected. When the screenshots are taken, the values of the threat dictionary are added to a CSV with the corresponding frame. A value of (0) means the threat is not present, while a (1) means it was detected. We made a copy of this data frame, removed the generated values, and then manually added them in by going through each frame and determining if the threat was present to represent the ground-truth. We then compared the original data frame to the one we filled out.

Overall Performance: We ran the diagnostic mode on roughly an hour's worth of driving videos. Particularly, (26) minutes of dash-cam streams captured using a dash-camera mounted on the front windshield of the car, and (30) minutes of first-person view of us driving in GTA-V. Then, we ran our testing script to figure out our ratio for true positive/negative predictions (when the system was correct) compared to false predictions. Lastly, we designed a rating system on a (0-100 scale) to describe the accuracy of our system – meaning how the system is doing in terms of making the right predictions. Comparing the data frame that was automatically generated by our system to the ground-truth data frame showed that the our system is 92% accurate in various situations such as (crowded areas, tunnels, and highways) on the dash-cam streams, while being 87% accurate in various weather/lighting conditions controlled within a virtual environment (GTA-V) as shown in the tables below:

Dash-Cam

Overall Performance ≅ 26 (minutes)	
count	274 frames
mean	0.915537
std	0.114775
min	0.428571
25%	0.857143
50%	1.000000
75%	1.000000
max	1.000000

(a)

GTA-V

Overall Performance ≅ 30 (minutes)	
count	422 frames
mean	0.874596
std	0.114521
min	0.428571
25%	0.857143
50%	0.857143
75%	1.000000
max	1.000000

(b)

Figure 4.3: Overall Performance

Performance By Individual Target: To further investigate our results, we modified our testing script to include a rating score for each feature in the system. As expected, most scores indicated that our system is performing fairly well, with the exception of off-lane detection and collision detection. Notably, research in the last few years has showed that vision-based solutions for both off-lane detection and collision detection are not ideal. There are several approaches that would achieve better results such as sensory-based system (LIDAR, RADAR, etc). The resulting scores for each feature are stated in the tables below:

Dash-Cam

Individual Performance ≅ 26 (minutes)	
Frame Count	274
Vehicles	93%
Pedestrians	92%
Bikes	96%
Stop Signs	100%
Traffic Lights	89%
Off Lane	74%
Collision Threats	97%

(a)

GTA-V

Individual Performance ≅ 30 (minutes)	
Frame Count	422
Vehicles	70%
Pedestrians	83%
Bikes	98%
Stop Signs	98%
Traffic Lights	66%
Off Lane	N/A
Collision Threats	97%

(b)

Figure 4.4: Individual Performance

Chapter 5: Conclusion

5.1 Challenges and Solutions

We had several challenges throughout this project that took some additional time to work through.

TKinter: The development of the interface proved to be more challenging than expected due to the limitations of *TKinter*. Originally, the setup interface and the runtime interface would be separate. However, we learned that only one instance of *Tkinter* can run at a time. So, we decided to hide the widgets in the setup interface behind those of the runtime interface when the program is running. Essentially, everything is in one window. In general, this is not an ideal solution, but for the scope and purpose of this project it will suffice. We also had communication problems between the interface and the Driving Assistant class. Originally, when the program was running (before the runtime interface was created), the setup interface would become unresponsive. We knew eventually this would be a problem, since our future runtime interface would suffer the same issue. We discovered the problem was a common threading issue, and the solution was simply to call the *mainLoop* function (which creates the Driving Assistant and runs the program) in a new thread.

Inter-process Communication: The development of the interface also suffered a communication issue. When a frame is analyzed, a dictionary is updated with the values of each possible threat (present or not present). The **updateState** function reads this dictionary and decides which icons to display based on the values of the threat dictionary. However, **updateState** did not have access to the dictionary because of the code structure. To fix this, we rearranged the code structure to it is present state, which gave **updateState** access to the dictionary, and structurally more sense overall.

Hardware Limitations: There were also problems caused by hardware limitations. We ran our program on two different systems, with an nVidia Titan Xp and 1080GTX respectively, yet still they struggled to achieve decent frame rates. A slow frame rate results in a delay between what the driver is actually seeing, and what has been processed and is reflected in the interface. For general object detection, it is less of a concern. However, it renders the collision threat detection fairly useless. When testing intentional collisions in GTA, the actual collisions almost always occurred before the program was able to give a warning. The slow frame rate also results in many missing frames, which further reduces its effectiveness - oftentimes in the event of a collision, no frames passed through the program that would have indicated an imminent collision. The only solution to this problem is better hardware. Other researchers implementing similar systems used four Titan Xp's bridged together. We did not have the budget for this, so we do not know how it would have performed with that level of processing power.

5.2 Future Work

There were many features that we wanted to add to our project that did not end up in the first release due to hardware limitations and the fact that it would be too time consuming. For example, currently our system generically detects traffic lights, however it does not specifically detect whether that is a green, yellow or red light. In the future, it would be really useful to add the distinction to get a more meaningful warning rather than just detecting a traffic light ahead. Furthermore, the only traffic signs our system can detect are traffic lights and stop signs. It would be helpful and more insightful to detect all traffic signs such as speed limits, warning signs, temporary traffic control signs, regulatory, and guide signs.

Further, our lane detection algorithm is certainly not perfect. It relies on a combination of gradient thresholds along with an edge detection algorithm to locate the exact pixels that represent the road markers. That could be improved by retraining our CNN to detect the current lane as well as other objects. However, both techniques will still perform poorly in severe weather conditions, or even bad lighting situations where the road markers are not necessarily clear. One possible solution would be to repaint all road markers with a special type of paint that could be detected with an on-board sensor. Additionally, it would be even more useful to add lane curvature detection, which is crucial if the system were to be deployed in a fully autonomous driving mode.

Finally, our current implementation only targets single-camera forward detections. In other words, it detects objects in front of the driving agent. Ideally, the system would be deployed with cameras that cover an entire 360degree view around the car, which would allow to detect objects in the blind spots of the driving agent. Ultimately, we recommend replacing the single-camera approach with a stereovision approach. They may be more expensive, but they are able to transform the 2-D spacial view into a 3-dimensional structure by deriving visual cues from two eyes instead of one. This is how humans imply the perception of depth to the objects surrounding us.

5.3 Project Importance

We have learned much from this project about the impact Artificial Intelligence will have on the automotive industry. We have several important conclusions based on the results of this project.

Object Detection: We believe computer-vision based object detection is a useful system that can be implemented in future vehicles. While observant drivers may not find our implementation particularly helpful since it only takes in a front facing camera view, a 360degree camera view would be very useful, as it could detect important objects in blind spots. This could substantially reduce motorcycle crashes, as they often happen because other drivers do not see them in their blind spots. More specific audible warnings could easily be added in such a system: "Motorcycle on the left side, use caution." We believe

the frame rates we achieved with our hardware are suitable enough for this general object detection purpose.

Lane Detection: Over the last decade, it has been noted that using computer-vision for lane detection is not an ideal solution due to multiple issues. First, a deep learning approach powered by neural networks may perform better than the color filters and thresholding approach. Conceptually, using neural nets for lane detection would be more resource intensive, and the complexity of the system will increase exponentially as the underlying architecture grows bigger. However, using either approach to detect road markers becomes more complicated in common scenarios such as severe weather conditions where the markers are not visible on the road.

Collision Detection: Based on our results, we cannot recommend computer vision based collision detection systems. There are several major problems with our collision detection system. The first is the hardware - because we cannot get a high frame rate, collisions usually happen before the system can predict one. More advanced (and expensive) hardware may help. The size and position of the collision box is rather arbitrary, and has to be changed with different angles, width of view, etc. A consistent camera angle would be needed, as well as a custom shaped collision area, rather than a rectangular box. The last major issue is the fact that the object classifier cannot classify objects that are right in front of the camera, because it cannot see the entirety of the object. Buildings and vehicles close to the camera are examples of this. If the driver is about to collide with an object that has not been detected, the system cannot warn the driver. There is no feasible way to solve this problem. Radar/LIDAR based systems are much more practical, and do not suffer any of the problems our computer-vision based system does.

References

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. J. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Józefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. G. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. A. Tucker, V. Vanhoucke, V. Vasudevan, F. B. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," *CoRR*, vol. abs/1603.04467, 2016. [Online]. Available: https://www.tensorflow.org/api_docs/python/

[2] A. Geiger, M. Lauer, C. Wojek, C. Stiller, and R. Urtasun, "3d traffic scene understanding from movable platforms," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 36, no. 5, pp. 1012–1025, May 2014.

[3] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," in *Proceedings of the 2014 IEEE Conference on Computer Vision and Pattern Recognition*, ser. CVPR '14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 580–587.

[4] J. Huang, V. Rathod, C. Sun, M. Zhu, A. Korattikara, A. Fathi, I. Fischer, Z. Wojna, Y. Song, S. Guadarrama, and K. Murphy, "Speed/accuracy trade-offs for modern convolutional object detectors," *CoRR*, vol. abs/1611.10012, 2016. [Online]. Available: http://arxiv.org/abs/1611.10012

[5] M. Johnson-Roberson, C. Barto, R. Mehta, S. N. Sridhar, K. Rosaen, and R. Vasudevan, "Driving in the matrix: Can virtual worlds replace human-generated annotations for real world tasks?" in *2017 IEEE International Conference on Robotics and Automation (ICRA)*, May 2017, pp. 746–753.

[6] A. Karpathy and J. Johnson, "Cs231n: Convolutional neural networks for visual recognition," http://cs231n.github.io/convolutional-networks/, Stanford.

[7] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2012, pp. 1097–1105.

[8] A. Nguyen, J. Yosinski, and J. Clune, "Deep neural networks are easily fooled: High confidence predictions for unrecognizable images," in *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2015, pp. 427–436.

[9] *OpenCV API Reference*, OpenCV. [Online]. Available: https://docs.opencv.org/3.0-beta/index.html

[10] J. Redmon, "Darknet: Open source neural networks in c," http://pjreddie.com/darknet/, 2013–2016.

[11] J. Redmon, S. K. Divvala, R. B. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," *CoRR*, vol. abs/1506.02640, 2015. [Online]. Available: http://arxiv.org/abs/1506.02640

[12] J. Redmon and A. Farhadi, "YOLO9000: better, faster, stronger," *CoRR*, vol. abs/1612.08242, 2016. [Online]. Available: http://arxiv.org/abs/1612.08242

[13] S. Ren, K. He, R. Girshick, and J. Sun, "Faster r-cnn: Towards real-time object detection with region proposal networks," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 39, no. 6, pp. 1137–1149, June 2017.

[14] S. R. Richter, V. Vineet, S. Roth, and V. Koltun, *Playing for Data: Ground Truth from Computer Games*. Cham: Springer International Publishing, 2016, pp. 102–118.

[15] A. Shafaei, J. Little, and M. Schmidt, "Play and learn: Using video games to train computer vision models," *CoRR*, vol. abs/1608.01745, 2016.

[16] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Computer Vision and Pattern Recognition (CVPR)*, 2015.