DriveBerry: Navigating Roads Through Lane and Signage Detection

1st Aditya Niraj Patkar
University of Maryland
College Park, MD
apatkar@umd.edu

2nd Saad Ur Rahman
University of Maryland
College Park, MD
saadr98@umd.edu

***Abstract*—This technical report details the "DriveBerry" project aimed at developing a miniature autonomous vehicle system using Raspberry Pi and Google Edge TPU. The project's core objective was to demonstrate two crucial functionalities for autonomous vehicles: precise lane following and accurate signage detection. To achieve this, a Raspberry Pi-based PiCar kit was used, coupled with Google Edge TPU for executing deep learning tasks.**

Our methodology involved a systematic approach, starting with the assembly of a functional prototype and installation of necessary software and hardware components. We utilized OpenCV for demonstrating real time lane navigation and capturing video data. A Convolutional Neural Network was trained to mimic the behavior of the OpenCV based system. The project encountered and overcame several challenges, including hardware limitations and software compatibility issues.

Key findings from this project include the successful implementation of lane navigation and traffic sign recognition capabilities in a scaled-down autonomous vehicle model. The modified prototype, equipped with a fisheye camera for enhanced field of view, demonstrated efficient lane tracking and sign detection under varied conditions using the OpenCV based network. The use of CNNs proved effective but less accurate compared to OpenCV in interpreting and responding lane lines. An object detection model was successfully able to detect stop signs with a real time performance of 30-40 frames per second boosted by the Google Edge TPU.

This project showcases the practical application of machine learning and image processing techniques in real-world driving scenarios. The successful integration of hardware and software components in this project offers valuable insights and a replicable model for similar initiatives in the field of robotics and automated transportation.

I. Introduction

Autonomous vehicles are slowly taking over the roads. Most of the newest cars have some kind of software based navigation systems implemented. Companies like Tesla and Rivian are putting advanced self-driving capabilities into their vehicles to navigate tricky situations on roads.

Two of the most fundamental aspects of self driving capabilities are lane detection and signage detection. Lane detection makes sure that the vehicle remains within its path, maintaining safety and order on the road. Meanwhile, sign detection provides the vehicle with vital cues on speed limits, warnings, and other regulations. Together, they form a basic yet essential foundation for an autonomous vehicle's decision-making process.

For this project, part of MSML642: Robotics class, we implement these two systems in a hardware based, deep learning focused solution powered by Raspberry Pi for general computing and Google Edge TPU for a blazing fast performance of the object detection model. The general goal of the "DriveBerry" project is to not only simulate these capabilities in a software environment but to also successfully implement them in a real-world hardware system. By doing so, the project aims to address the challenges posed by real-world conditions, which are often far from ideal, and to test the robustness of the proposed solutions in such environments.

The primary objectives include the assembly of a functional prototype using the PiCar kit and Google Edge TPU, and the utilization of OpenCV for capturing and processing real-world lane navigation data. A critical part of this project involves training a Convolutional Neural Network (CNN) to interpret this data for accurate lane and signage detection.

By successfully navigating the complexities of hardware assembly, software setup, and the implementation of machine learning models, "DriveBerry" serves as a beginner friendly and cost effective introduction to the modern world of machine learning based robotics. It signifies the challenges faced during software and hardware integration and talks about how to overcome them.[1]

II. Literature Review

Lane naviation and object detection have been important components of autonomous vehicle systems since the beginning of the domain. Over the years, researchers have explored various approaches to improve these tasks using computer vision techniques.

Few approaches have used fancier sensors like LiDAR and Radar for lane detection. For instance [1] proposed a sensor-fusion lane-detection system for autonomous vehicle navigation. The system integrated LiDAR and vision data to handle both structured and unstructured roads effectively. The authors demonstrated the reliability, effectiveness, and robustness of the system.

Another study used a combination of camera and GPS. [2] proposed a real-time computer vision and GPS-aided inertial

[1]The code repository for the DriveBerry can be found at https://github.com/adityapatkar/driveberry

navigation system. The system utilized OpenCV and sensor fusion techniques to achieve accurate lane-level navigation. The research findings demonstrate the effectiveness of the system in providing precise vehicle positioning and navigation information.

Meanwhile, just using just a camera for lane navigation has shown promising results. [3] proposed end-to-end learning for self-driving cars, using a CNN to directly map pixels to steering commands. The system showed autonomous navigation in diverse scenarios without explicit feature training. Our CNN-based lane navigation shares similarities with the methodology and architecture of the end-to-end learning approach. Overall, the usage of OpenCV and CNNs combined with a camera has shown great potential in lane navigation systems.

Similarly, road signage detection systems have also relied on multiple sensors. [4] proposed a method for road detection using a fusion of LiDAR and camera data. The authors tested their approach on DAWN, KITTI, and MS-COCO datasets . The experimental results demonstrated the effectiveness of the proposed method, which outperformed state-of-the-art signage detection and tracking approaches, even under adverse weather conditions.

In conclusion, a lot of previous work has involved using multiple sensors for the tasks of lane navigation and road signage detection. With the monetory and resource considerations, having only a single camera, we decided to go forward with OpenCV and CNN based approaches for lane navigation. With the consideration of compute power available on the Raspberry Pi, using Google Edge TPU combined with a quantized object detection model made sense for stop sign detection, considering this was required to be real-time.

III. Methodology

The plan of action was designed to include every aspect of creating a miniature autonomous vehicle system for aforementioned tasks. This process included hardware assembly, software setup, and the development of both computer vision and machine learning components for autonomous navigation and object detection. [5]

A. Hardware Assembly

The hardware components of the project consist of several key tools, essential for building the DriveBerry autonomous vehicle. These include:

- **PiCar Kit [6]**: The base of our autonomous vehicle, consisting of 4 wheels, 2 motors, a chassis, and 3 servos (one for steering the front wheels, one for tilting the camera, and one for panning).
- **Raspberry Pi**: Serving as the central processing unit, this handles the input-output operations, inference, and communication tasks of the vehicle.
- **Google Edge TPU**: A critical component for processing the deep learning tasks which require parallel computation.

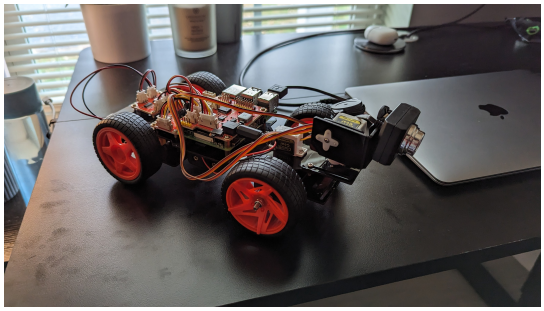

Fig. 1. DriveBerry after Assembly

- **Camera Sensor**: A USB camera attached to the Raspberry Pi, acting as the vehicle's eyes for capturing video data.
- **Power Supply**: Utilizing a rechargeable battery pack for the vehicle's power requirements.

The assembly process involved meticulous integration of these components. The chassis was assembled first, followed by mounting the wheels and attaching the servos. The Raspberry Pi was then mounted on top of the chassis. Careful attention was given to the wiring of the back wheel motors, USB camera, servos, HAT motor control PCB, and the Google Edge TPU, ensuring correct connections for optimal functionality. The completely assembled DriveBerry can be seen in figure 1.

B. Operating System Installation

The Raspberry Pi required a separate operating system installation. A Debian-based Linux OS was chosen for its compatibility and robustness. The OS was flashed onto a MicroSD card using the Raspberry Pi Imager tool. Additional configurations included enabling SSH for remote access and connecting the Pi to a Wi-Fi network for remote operation. This was done by following the official documentation [7].

C. Software Library Installation

The software setup involved the installation of various libraries, each serving a specific purpose in the project. The main installations include:

- **Conda**: To manage Python virtual environments, addressing incompatibilities with the latest Python versions available on Raspberry Pi.
- **OpenCV** and **Matplotlib**: For initial lane detection, video capturing, image processing, and graphical representations.
- **Samba** and **RealVNC**: Enabling remote file access (Samba) and remote desktop viewing (RealVNC) of the Raspberry Pi.
- **Cheese**: A simple camera viewer application used for testing the camera functionality.
- **PiCar Library**: A high-level library for controlling the vehicle, which required modification and manual installation due to maintenance issues.

- **TensorFlow, TensorFlow Lite, and PyCoral [8]**: These are the most important components from the deep learning perspective for training and inference of our CNN models. It was very tedious to get these to work. The TensorFlow Lite version required tensorflow to be greater than version 2.11 while PyCoral required it to be less than 2.7. This caused an issue. TFLite is very important for compiling TensorFlow models in smaller (Integer weights) format while PyCoral is required for running inference from these models on the Google Edge TPU. Turns out, the support for PyCoral has been stopped by Google. We got around this issue by using a wheel of PyCoral that was built by a random developer on Stackoverflow. We were able to successfully run a test image classification model on the Google Edge TPU that we attached to DriveBerry. We also were able to run a COCO object detection model. It ran with 1-2 FPS on Raspberry Pi CPU while it ran with 15-20 FPS on Google Edge TPU.

These software components were installed to ensure functionality, overcoming compatibility challenges and setting up a robust base for the subsequent phases of the project.

To test the integration of the software and hardware, we wrote 2 small scripts, one which turns the front wheels left and right and one which makes the car go forward and backward using the rear motors. The car was able to complete both tasks successfully.

D. Lane Navigation Using OpenCV

This phase was designed to enable the autonomous vehicle to recognize and follow road lanes, an essential functionality for any self-driving system. The process encompassed several key steps, each contributing to the overall effectiveness of lane detection.

1. Video Capturing: The first step in this process involved capturing video footage from a USB camera attached to the Raspberry Pi. This footage was crucial, as it served as the primary data source for detecting lane lines. The video stream was processed frame by frame, with each frame undergoing a series of transformations and analyses to accurately identify the road lanes.

2. Image Processing: n the image pre-processing stage, a critical transformation was the conversion of each frame from the BGR (Blue, Green, Red) color space to the HSV (Hue, Saturation, Value) color space. This conversion was essential to reduce the variability introduced by different lighting conditions and to isolate the color of the lane markings more effectively. For this project, blue tape was used for lane markings, and OpenCV's `inRange` function was utilized to create a mask that isolated this specific color range.

3. Edge Detection: Edge detection was a pivotal step in the lane detection process. The project utilized the Canny edge detection method [9], a well-established technique in computer vision, to identify the edges within each frame, focusing particularly on the areas with the isolated blue color. This approach helped in accurately detecting the boundaries of the lane lines. To further refine the focus and minimize distractions, a region of interest was selected, concentrating on the lower half of each frame where the lane lines actually decide the angle of the front wheels.

4. **Line Detection with Hough Transform**: After the preliminary steps of video capture, frame extraction, and edge detection, the detected edges representing potential lane lines were still in a pixel-form. The challenge was to translate these pixel points into meaningful line representations that could guide the autonomous vehicle. This is where the Hough Transformation played an important role.

The Hough Transformation works by converting each point in the image space into a sinusoidal curve in the Hough space. In this transformed space, each curve corresponds to a potential line in the image space. When multiple curves intersect at a point in Hough space, it signifies that a line exists in the image space at the coordinates represented by that intersection. This technique is particularly adept at detecting straight lines, which are common in road lane markings. In the context of the "DriveBerry" project, the Hough Transformation [10] was applied to the edges detected in the previous step. The OpenCV library provides a function, `HoughLinesP` to perform this transformation.

The result of this transformation was a set of lines, each defined by parameters like its position and angle. However, these lines were often numerous and fragmented. To create a coherent lane marker representation, the lines were processed further. This involved categorizing them into left and right lane markers based on their slope and then averaging the positions and angles of the lines within each category. This averaging process converted the multiple detected line segments into two distinct lines, representing the left and right boundaries of the lane. . The heading line, critical for determining the vehicle's trajectory, was calculated by averaging the farthest points of these lines. In scenarios where only one line was detected, the heading line was adjusted to match the slope of the single detected line - translating in a sharp turn.

5. Angle Calculation: The angle of steering is calculated based on the following formula:

$$\text{angle\_to\_mid\_radian} = \arctan\left(\frac{x\_offset}{y\_offset}\right) \quad (1)$$

where y-offset is set to half the height of the frame. This is based on the assumption that the camera is centered on the vehicle. x-offset is set to the midpoint of the two lines detected. The angle is then converted into degrees using the formula below.

$$\text{angle\_to\_mid\_deg} = \text{angle\_to\_mid\_radian} \times \frac{180}{\pi} \quad (2)$$

Finally, the steering angle is calculated using:

$$\text{steering\_angle} = \text{angle\_to\_mid\_deg} + 90 \quad (3)$$

This is to compensate for the fact that in PiCar library, going straight equals to 90 degrees.

To ensure smooth navigation and avoid abrupt directional changes, the project implemented angle thresholding. This

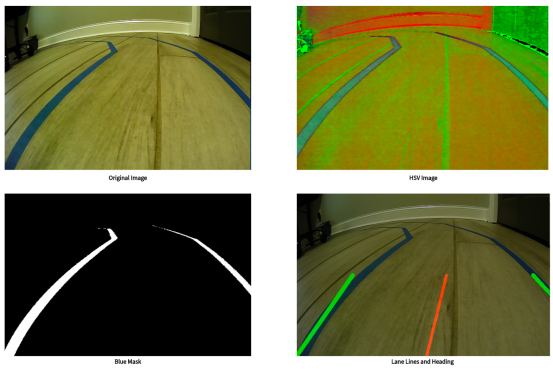

Fig. 2. Original, HSV, Heading and Blue Mask Images (Clockwise)

involved setting a limit on how much the steering angle could vary from one frame to the next, providing a more consistent and stable lane-following behavior. It essentially does what a PID controller would normally do.

6. Kinematics: For each frame coming from the camera of DriveBerry, the above steps are performed. The car initially starts with a 90 degree angle and a speed of 35. Per second, around 20-25 frames are processed, and the front wheels are turned based on the calculated angle, thus making sure that DriveBerry seamlessly navigates the lanes.

The car was driven on different tracks and the video was saved for each such navigation attempt. This video would further be used to train a convolutional neural network which will predict the steering angle given a frame. A small script was written to run the above steps on the saved videos instead of actual camera feed. The script converted each frame to an image and saved the image, the filename of which included the steering angle. These images would act as the training data for the convolutional neural network.

E. Lane Navigation using Convolutional Neural Network

Using the prepared images, a convolutional neural network was trained to predict the steering angle. We will talk about the data exploration, data processing, CNN architecture, model training, evaluation and integration.

1. Data Exploration: The prepared dataset consisted of 692 distinct frames captured by the raspberry pi camera. It was observed through a histogram that the steering angle was densely saturated around 80 to 90 degrees with less samples being below 60 degrees or above 120 degrees. The dataset was split into training and validation datasets, the histograms of both looked similar. This can be observed in figure 3.

2. Data Pre-processing: Image augmentation played a crucial role in enhancing the performance of the CNN used for steering control. A random selection of images went through a one or more of a slew of transformations which included zooming, panning, adjusting brightness, blurring, and flipping. Figure 4 shows one such image that was flipped. The primary purpose of image augmentation in this context was to ensure that the CNN could effectively handle a wide variety of driving conditions and scenarios. Essentially, this would ensure that

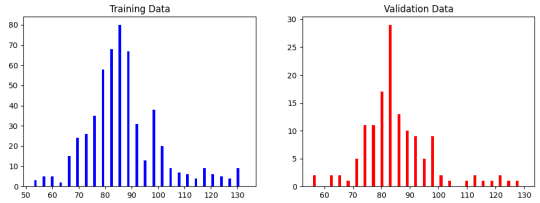

Fig. 3. Streering Angles of Training and Validation Datasets

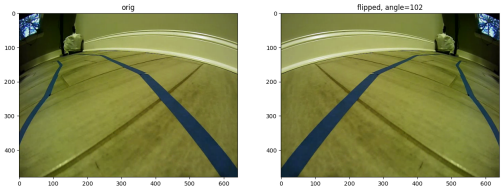

Fig. 4. Flipped Image

the model does not over-fit to the training data. The images were further pre-processed to match the input dimensions and colours as required by the model.

3. CNN Architecture:

The architecture of the model is very similar to what the researchers used in [3]. The model, at the heart of it includes a simple convolutional neural network, which is known for its ability to extract visual features from images. Our model includes 5 convolutional layers with Elu activation. The input to the model is of size 66*200*3. There is a dropout layer between the fourth and fifth convolutional layer which helps the model generalize better. The convolutional layers are followed by 4 dense layers, the last of which outputs a single number which would be the steering angle. We used Keras for building and training the model. Figure 5 shows the architecture of the CNN model.

4. Model Training and Evaluation: As the steering angle is a continuous variable and can be anything between 0 to 180, this is essentially a regression problem. The model was trained

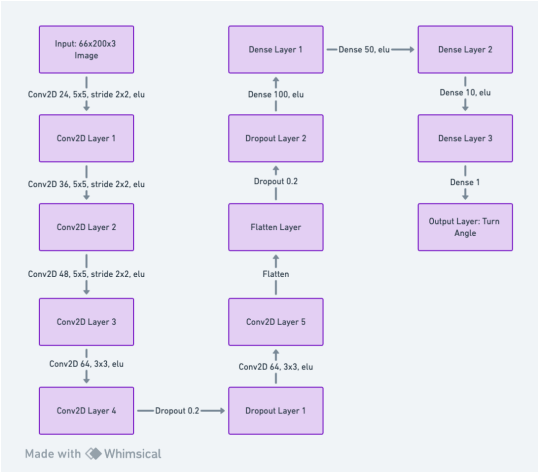

Fig. 5. CNN Architecture

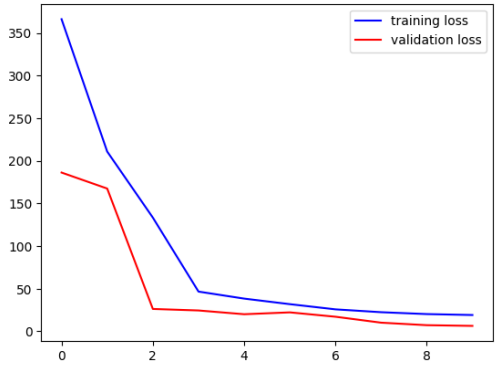

Fig. 6. Training and Validation Loss

with a learning rate of `1e-3` with the Adam optimizer and Mean Squared Error (MSE) as the loss function. The training process involved `10` epochs with a batch size of `100`. The model performed fairly well on the test set, with an MSE of 8.8 and R squared of 94.03%. The trained model was saved for further use inside the DriveBerry codebase. Figure 6 shows the training and validation loss of of the model during training with respect to epochs.

5. Integration and Testing:

The trained CNN model, once optimized, was integrated into the DriveBerry vehicle's control system. The frame coming from the car's camera is pre-processed in the same way as the images were pre-processed during training. The steering angle is computed by calling the `predict` method of the saved model. The front wheels are then turned with the predicted angle.

While able to navigate the lanes, using CNN model did not outperform the reliable and very accurate OpenCV based navigation. It was observed that while performing CNN based lane navigation, the car would often go out of the lane during the sharper turns. This can be attributed to the distribution of the training data not being uniform. We thing that with more uniform data, the model can be improved to perform better.

F. Object Detection

In the "DriveBerry" project, alongside the CNN for lane navigation, an object detection system was implemented to recognize stop signs. This system used a pre-trained TensorFlow Lite (TFLite) model, optimized for efficiency using quantization and deployed using the PyCoral library, which is specifically designed for edge devices like the Google Edge TPU. Quantization involves reducing the precision of the numbers in the model (e.g., converting from floating-point to 8-bit integers), which decreases the model size and speeds up inference with minimal loss in accuracy. This is particularly beneficial for deployment on resource-constrained devices like Edge TPU.

Initially, we had plans to train our own model for detecting objects on the path by using recorded videos of the car running on various tracks with different signs including stop sign, red light, speed limit signs and toy pedestrians. This was unviable as the support for PyCoral library has been terminated by Google. While we tried multiple approaches [11] [12], including using different version combination of libraries and going through various issue solving threads [13] [14].

The PyCoral library was used to interface with the Google Edge TPU. This enabled the execution of the quantized model at high speeds, taking advantage of the TPU's capability for fast parallel processing. First, the input frame was resized to match the input size of the model, which is a `SSD MobileNet V2` [15] model from [16]. This model was pre-trained to detect stop signs. A score threshold of 0.65 was established for object detection to ensure a balance between precision and recall, minimizing false positives while maintaining sensitivity to actual stop signs.

The labels file corresponding to the pre-trained model was loaded to interpret the output of the detection process. The objects in the frame were detected using PyCoral's `get_objects` method, which returns the ids of the detected objects along with the bounding box coordinates. To determine the proximity of detected objects, a ratio calculation was employed, comparing the height of the detected object to the height of the frame. An object was deemed close if this ratio exceeded a predefined threshold, indicating its immediate relevance to the vehicle's path.

When a stop sign was identified within close proximity, the vehicle's control system was triggered to respond appropriately. This involved halting the car for a duration of three seconds, achieved by setting the speed of the back wheels to zero. After this brief stop, the car resumed its motion, continuing its navigation path. This mechanism ensured the vehicle adhered to the traffic rule corresponding to the stop sign.

IV. Results

Our implementation of autonomous driving functionalities, including lane navigation using OpenCV, a Convolutional Neural Network (CNN) and stop sign recognition using pre-trained quantized model, yielded significant results. This section presents an analysis of these outcomes.

A. Lane Navigation Results

- The OpenCV based approach for lane navigation approached proved to be significantly better than the CNN based approach. This could be due to the fact that the CNN was trained on the same data that was produced by OpenCV based approach and the fact that the data was not uniformly distributed.
- The CNN model for lane navigation was evaluated on a test set based on its accuracy in predicting steering angles. The model achieved a Mean Squared Error (MSE) of 8.8 t, indicating a decent level of precision in its predictions.
- During real world testing, the OpenCV based approach adhered to the lane 100% of the time on 2 different tracks. The CNN based approach adhered to the lane for 83% of the time on the same tracks.

- Both approaches were tested under different lighting conditions. Both approaches suffered marginal performance degradation in low lighting conditions. CNN based model was impacted more than the OpenCV based approach. The overall performance degradation can be attributed to the low resolution and quality of the camera which would not capture accurate colors in low lighting conditions. The better performance of OpenCV based approach can be explained by the fact that the upper and lower limits of blue colors were set after heavy testing and tuning.

B. Object Detection Results

- The object detection system, using the quantized SSD MobileNet V2 model on the Google Edge TPU, identified stop signs with a detection accuracy of 100%, based on the number of correctly identified stop signs over total stop signs encountered.
- The system maintained a high detection rate even at different times of the day
- The model ran with consistent framerate of 30+ FPS even after 10+ minutes of continuous operation.

Overall, the system is reliable and fast in an indoor small scale implementation. Combining the OpenCV based approach with an edge device based object detection would prove to give a good real world performance. The use of CNN for lane navigation needs to be further explored with more data and/or different architectures. Usage of more sensors like LiDAR, Proximity sensor can be explored to further aid the vehicle.

V. Conclusion

In conclusion, the "DriveBerry" project successfully demonstrates the integration and application of machine learning and computer vision techniques in a miniature autonomous vehicle system. Utilizing a combination of Raspberry Pi, Google Edge TPU, and a few software tools, this project has successfully implemented and compared two distinct methods for lane navigation - one based on OpenCV and the other on Convolutional Neural Networks (CNNs), along with an efficient object detection system for recognizing stop signs.

The OpenCV-based lane navigation system exhibited superior performance in comparison to the CNN approach, demonstrating 100% lane adherence in our tests. This can be attributed to the meticulous calibration of color thresholds and the robustness of traditional computer vision techniques in structured environments. However, the CNN-based method, despite its slightly lower performance, represents an important way of using advanced, data-driven autonomous navigation systems. This disparity in performance also highlights the importance of diverse and balanced training data for machine learning models.

The project's object detection system effectively recognized stop signs in real-time, leveraging the power of the Google Edge TPU. This capability is crucial for ensuring compliance with traffic rules and enhancing the safety aspects of autonomous vehicles. The speed at which the Edge TPU does inference is highly important to the real time performance of the vehicle.

Key lessons from the "DriveBerry" project include the importance of syncing hardware and software components, the challenges and potentials of integrating machine learning in robotics, and the need for extensive testing under various conditions to refine autonomous systems. Future improvements could involve enriching the dataset for the CNN, integrating additional sensors for enhanced environmental perception, and exploring alternative neural network architectures.

"DriveBerry" serves as a valuable lesson for us, providing insights into the practical challenges and solutions in the field of autonomous robotics. It lays the groundwork for further exploration in this already well established yet rapidly evolving domain.

References

[1] Q. Li, L. Chen, M. Li, S.-L. Shaw, and A. Nüchter, "A sensor-fusion drivable-region and lane-detection system for autonomous vehicle navigation in challenging road scenarios," *IEEE Transactions on Vehicular Technology*, vol. 63, no. 2, pp. 540–555, 2014.

[2] A. Vu, A. Ramanandan, A. Chen, J. A. Farrell, and M. Barth, "Real-time computer vision/dgps-aided inertial navigation system for lane-level vehicle navigation," *IEEE Transactions on Intelligent Transportation Systems*, vol. 13, no. 2, pp. 899–913, 2012.

[3] M. Bojarski, D. D. Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, X. Zhang, J. Zhao, and K. Zieba, "End to end learning for self-driving cars," 2016.

[4] L. Caltagirone, M. Bellone, L. Svensson, and M. Wahde, "Lidar–camera fusion for road detection using fully convolutional neural networks," *Robotics and Autonomous Systems*, vol. 111, pp. 125–131, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0921889018300496

[5] Raja961, "Autonomous Lane-Keeping Car Using Raspberry Pi and OpenCV — instructables.com," https://www.instructables.com/Autonomous-Lane-Keeping-Car-Using-Raspberry-Pi-and/, [Accessed 04-12-2023].

[6] "Sunfounder picar v kit," https://docs.sunfounder.com/projects/picar-v/en/latest/, [Accessed 04-12-2023].

[7] "Raspberry Pi Documentation - Getting started — raspberrypi.com," https://www.raspberrypi.com/documentation/computers/getting-started.html, [Accessed 04-12-2023].

[8] K. Dmitriykovalev, "GitHub - google-coral/pycoral: Python API for ML inferencing and transfer-learning on Coral devices — github.com," https://github.com/google-coral/pycoral, [Accessed 04-12-2023].

[9] J. Canny, "A computational approach to edge detection," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. PAMI-8, no. 6, pp. 679–698, 1986.

[10] L. Chandrasekar and G. Durga, "Implementation of hough transform for image processing applications," in *2014 International Conference on Communication and Signal Processing*, 2014, pp. 843–847.

[11] "Object Detection with TensorFlow Lite Model Maker," https://www.tensorflow.org/lite/models/modify/model_maker/object_detection, [Accessed 03-12-2023].

[12] "Reddit - Dive into anything — reddit.com," https://www.reddit.com/r/ShinobiCCTV/comments/1761b8n/cant_seem_to_get_coral_working/, [Accessed 03-12-2023].

[13] "Python 3.10 and 3.11 support?" https://github.com/google-coral/pycoral/issues/85, [Accessed 03-12-2023].

[14] "edgetpu make interpreter fails without printing error," https://github.com/google-coral/pycoral/issues/57#issuecomment-949997068, [Accessed 03-12-2023].

[15] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "Mobilenetv2: Inverted residuals and linear bottlenecks," 2019.

[16] "Models - Object Detection — Coral — coral.ai," https://coral.ai/models/object-detection/, [Accessed 03-12-2023].