Report Outline

- Sections:
    - Abstract
    - Introduction
    - Material and Methods
    - Results
    - Discussion
    - Conclusions and Future Work
    - Acknowledgments
    - References
- IEEE conference format
- 4-6 pages long
- Worth 9 marks (out of 30)!!!

Project Rubric From Announcement:

Item	**Marks**
Functionality (complete tool/app, adherence to proposal)	10
Creativity	3
Documented (add comments) and styled source code. Put the code in a folder and zip it.	2
Report in IEEE conference format (4-6 pages long according to the template).	9
Installation guide (what is needed to run your program/execute code)	2
Video Demo (5 minutes max, show functionality and creativity)	2
Members' contribution (free form, including who did what)	2
Total	**30**

Abstract

Revolutionary advancements in autonomous robotics have been made possible by the rapid development of computer vision technologies. This project aims to create a computer vision program that can identify, track, and categorize objects in live video streams to assist a semi-autonomous robot with navigation and decision-making. A semi-autonomous robot with DC motors and a Raspberry Pi 5 was created to communicate with a Flask-based server to facilitate picture processing and control decisions. To enable accurate detection of roadway items like cars, pedestrians, and traffic signals, we used transfer learning to train bespoke models by extending OpenCV libraries and YOLO11 models. Various traffic signals were used in a simulation created on Unity to assess the trained model's accuracy and decision-making in real time.

Introduction

The features of autonomous systems have been radically altered by the combination of robotics and computer vision, opening the door for uses in industrial automation, self-driving cars, and other fields. By creating a semi-autonomous robot that uses computer vision to detect, track, and classify objects in real-time, this project expands on these developments. The objective is to develop a system that can navigate and make decisions in complex environments, particularly those that resemble roads.

The system's central component is a four-wheeled robot with DC motors and a Raspberry Pi 5 for controlling the motor and handling visual data. The robot records live video streams using USB webcams, which are then sent to a server running Flask for processing. The server uses YOLO11 models, which have been improved via transfer learning on the "Traffic and Road Signs Dataset," to identify important objects like pedestrians, cars, and road signs. The Raspberry Pi receives the control decisions that are informed by the processed data and use them to direct the robot's movements.

This report explores the design, implementation, and testing of the system, detailing the methodologies used for training the YOLO11 models and the challenges of real-time integration. Additionally, to assess the model's performance, a Unity-based simulation was developed, allowing us to test the trained models with various road signs under controlled conditions. By combining hardware, software, and simulation testing, this project aims to advance the practical applications of computer vision in robotics.

Material and Methods

Our robot is a 4-wheeled system designed for semi-autonomous navigation. The motor system is composed of 4 DC motors connected to a single L298N motor controller board with its own dedicated 12V external battery. A Raspberry Pi 5 was used in combination with USB webcams to collect visual data and manage motor control inputs.

To control the robot through the image stream, the Raspberry Pi remotely connects to a server, collects the current frame from the webcam using OpenCV, emits the captured data back to the server and awaits control inputs. These frames are then processed on the server and used as input to make predictions using our custom trained models. Based on the prediction results, the server makes a control decision and emits the control data back to the Raspberry Pi. To handle connections and data transfer, a Flask server utilizing Socketio was used to receive image data and return prediction data back to the Raspberry Pi to control the robot.

For object detection, we utilized transfer learning to create a custom model based on the Yolov11n and Yolov11s models, trained on the “Traffic and Road Signs Dataset”. The Yolov11n and Yolov11s models were chosen because of their superior inference speed as we planned to get predictions in real-time when compared to alternative models. The Yolov11n model was trained on the full dataset for 100 epochs, batch size 16, and 640 pixel resolution. We also trained both models with the same hyperparameters on a reduced dataset where we removed all but the 3 classes which we intended to use for our demo. Before processing the “Traffic and Road Signs Dataset” contained 10,000 images and after our reduction contained 2000, both maintaining a validation split of 0.4. As a result of our initial testing, we trained another model with images of a larger resolution in an attempt to increase accuracy at a distance. This round of training had slightly modified hyperparameters of batch size 8 with images of 1280 pixels. The smaller batch size was a result of limited GPU memory. All rounds of training included freezing all layers except the final classification layer. The computer specifications used for training the Yolo models were as follows: AMD Ryzen 5 5700X, 16GB RAM, Nvidia RTX 3060 Ti 8GB video memory, utilizing Nvidia CUDA cores.

Self-driving/Dave2 model Section:

We have built a CNN model and trained it from scratch to be able to use that on the robot. The model is based on the Nvidia Dave2 model mentioned in [1], and the idea is that we put 3 cameras in front of some cars, and then we drive them around for hours in different conditions and record the images along with the steering wheel angle at the time of each 3 images taken, and then feed them to the model. The model has proven to work surprisingly well with such limited data, and it is able to now predict accurate steering angles and drive the car without much human supervision [1]. Although we have not done any manual lane detection, the CNN model after enough training, is able to accurately identify the lanes of many roads and predict a steering angle.

We have used an open-sourced Car Simulation app from Udacity built with Unity, to both generate data for the model, and also to test the model. The simulation also has 3 cameras on top of the car just like mentioned in [1]. We drove the car on one track for 7 rounds, and then on another track for only 1 round, after training, the car was able to drive on the first track without crashing, or crashing at a maximum of once per round, and on the second track with a very low crash rate. Even if we do not use any data from the second track, it is still able to generalize well on unseen tracks.

After gathering the data, we used downsampling to reduce the number of records in our dataset where steering angles were zero. This is because most of the time we are driving straight, and if we do not do this reduction, the model gravitates towards predicting zero mostly for the steering angle which will perform very poorly on turns.

For splitting the data into training and validation sets, we used 30% for validation and stratified the steering angles for a balanced distribution.

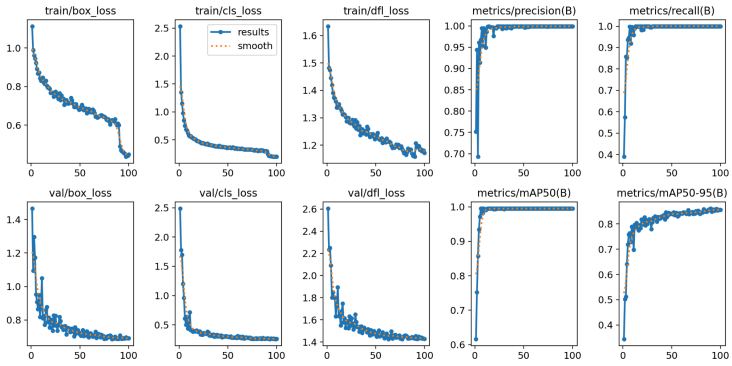
For image processing, we cropped the images to remove the sky and bottom of the car and resized and converted the colour space to YUV to match the Nvidia model exactly. For augmentation, we translated vertically and horizontally and rotated the images by a slight degree, and cast a random shadow on the images to make the model generalize well in unseen conditions. We also randomly changed the brightness of the images to simulate being in different weather conditions as it was difficult for us to add new tracks with Unity.

Our CNN model with 252,219 parameters, processes input images of shape [66, 200, 3] through a total of 13 layers, first layer is a normalization layer that scales pixel values to the range [-1, 1], followed by five convolutional layers (24, 36, 48, 64, and 64 filters respectively, with kernel sizes of (5,5) and (3,3), ELU activation, and strides of 2x2 for the first 3 convolutional layers). Dropout with a rate of 0.5 is applied after to prevent overfitting. The model then flattens the data and passes it through dense layers of sizes 100, 50, 10, and a final output layer with one unit which is the steering wheel value. We used the Adam optimizer with a learning rate of 0.00001, a batch size of 40, and Mean Squared Error loss over 10 epochs, and for some rounds, saved only the best model based on validation loss.

We first trained on a Mac Studio with M2 Max chip, and it took about 3 hours to finish 10 epochs with CPU utilization of around 300% out of 1200% (100% for each core), and GPU utilization of only 20% out of 100%. We then moved to AWS with a g5.xlarge instance that has one Nvidia A10G GPU. Although the Nvidia GPU is more powerful, we still saw the same 3 hours of training because only 5% of the GPU was being utilized.

Results

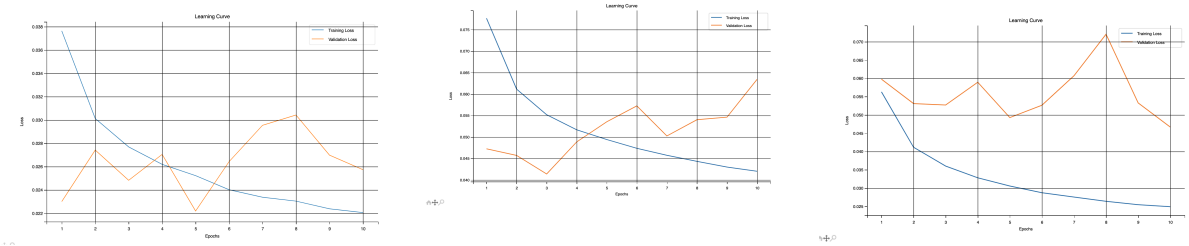
Of the four Yolo models trained, the best performing model was the Yolov11n on the reduced dataset using image size 640. As shown in **figure 1** the model achieved a relatively high accuracy and minimized the loss functions significantly over the 100 epochs. The models trained on the entire dataset proved to perform worse overall and contained a significant amount of misclassifications. The reduced dataset solved these issues although limited its versatility. Additionally, the Yolov11s models proved to have significantly greater latency when compared to the Yollov11n model.



(Figure 1 - Yolov11n train results [reduced dataset])

While testing in a controlled environment the data is promising. However, testing in our real-time environment proved to have its own difficulties. The model had difficulties picking up features from objects too far in the distance. Additionally, there were still a few rare misclassifications of the background although of low confidence.

Although our CNN model managed to drive the car even on unseen tracks, we did not succeed in getting a good learning curve and reducing overfitting. We expect to get a better result if we gather more data (we did about 1 hour of driving and Nvidia about 72 hours). Here are some of the learning curves that we got after trying different hyperparameters.



We got the best result from round 5 of our training (model can be found in source code), which is the image on the right.

Discussion

- Interpretation of results related to our main objectives
-

Conclusion

Future Work

To improve our Dave2 CNN model performance, we can add new tracks to our simulation with Unity with different weather conditions. For training the Dave2 model, they used data from 72 hours of driving in many different conditions [1] while we only used 1 hour of driving on the same track. Although we used augmentation to synthetically generate new images, like adding random shadows and changing the brightness to simulate different weather conditions, we can still benefit from having completely new tracks and collecting data for more hours to improve our learning curve.

Additionally, training our CNN model takes about 3 hours to complete, but our CPU usage is only around 20% and GPU usage is only 5% on an Nvidia A10G GPU. The model is being trained with Cuda on the GPU, but there is a bottleneck in the code that if we fix we should be able to significantly reduce our training time. If we do all the image processing using optimized C++ code, or using Tensorflow, we may be able to utilize our hardware more.

Our CNN model also does not take throttle and reversing into account, it only deals with camera images and steering wheel value, so if the car stops at a point, it will not be able to back up and fix itself. But if we feed the throttle, reverse and speed values to the model as well, it should be able to operate in more conditions.

Acknowledgements

For the robot car, we used the Self Driving Car Using Raspberry Pi for inspiration regarding a majority of the robot- Pi interaction modules. For the Yolo integrations, we largely followed the Ultralytics documentation and GitHub pages. The dataset we used was from Roboflow and was freely available for use for research purposes.

For the simulation, we used the Udacity Car Simulation. And for the implementation of the model, we have been heavily inspired by the Car Behavioral Cloning MIT-licensed open-source project.

References

[1] M. Bojarski, D. Del 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,” NVIDIA Corporation, Holmdel, NJ, Apr. 2016. [Online]. Available: https://arxiv.org/abs/1604.07316.