Deep Learning for Intelligent Transportation

Data Collection System, CNN Model & Simulation

A Major Qualifying Project

Submitted to the Faculty of

Worcester Polytechnic Institute

In partial fulfillment of the requirements for the

Degree in Bachelor of Science

In

Electrical and Computer Engineering

By

_____

Mengwen Li

_____

Javier Menchaca

Date: 03/24/2017

Project Advisor:

_____

Professor Xinming Huang

Abstract

The goal of this project is to advance WPI's intelligent transportation program through the creation of a data collection system, a Convolutional Neural Network (CNN) model for intelligent transportation, and a simulator to test the trained CNN model. The data collection system collects data from a vehicle- steering wheel angle, speed, and images of the road from three separate angles at the time of the data collection. A CNN model is then trained with the collected data. The trained CNN model is then tested on a simulator to evaluate its effectiveness.

Acknowledgements

This project could not have been completed without the help of certain individuals. These individuals provided us with needed guidance, expertise, and resources.

First, we would like to thank Professor Xinming Huang for advising our project, giving us guidance, providing us with resources, lab space, and the very kind use of his car for testing and research purposes. We would also like to thank Renato Gasoto, PHD student, for allowing us to utilize parts of his research in intelligent transportation simulations.

Last but not least, we would like to thank Worcester Polytechnic Institute for providing us with the opportunity, knowledge, and resources to pursue this project.

Executive Summary

The era of intelligent transportation is now. Intelligent transportation in the form of autonomous vehicles is as of this moment commercially available to consumers. Top car manufacturing companies such as Mercedes-Benz, General Motors, Nissan, Volvo, Audi, Ford, and notably Tesla, have successfully introduced autonomous vehicles into the market in some shape or form- whether it be in the form of lane keep, object detection, or full functioning autonomous driving. The field of intelligent transportation is currently one of the most prominent and popular fields in the industry, it is a field with lots of room to grow and develop.

The goal of this project was to advance WPI's intelligent transportation program through the creation of a data collection system, a Convolutional Neural Network (CNN) model for intelligent transportation, and a simulator to test the trained CNN model. By developing these tools, it was our aim to further enhance, advance, and aid WPI's intelligent transportation program.

This project was composed of three major core components: the data collection system, a Convolutional Neural Network (CNN) model, and a simulator. The data collection system was used to collect data from an active vehicle in real time, it collected the steering wheel angle, speed, and images of the road from three separate angles. The CNN model was then trained with the collected data from the vehicle. Then, having collected the data and trained a CNN model with it, the trained model was tested in a simulator to evaluate its effectiveness in a safe and controlled environment.

The result of this project was a modular collective system for intelligent transportation that can be implemented and tested in a variety of different vehicles to collect data, create

models based of the data, and then test the data in the controlled environment of a simulator. By using this system, the user can collect data, train a CNN model, and test his or her model in a simulator easily and efficiently. The data collection system works as expected- it captures images at a rate of 10 frames per second, collects the speed of the vehicle at a rate of 10 frames per second, and it collects the steering wheel angle of the vehicle at a rate of 30 frames per second. The Convolutional Neural Network model also works as expected, it is able to accurately and correctly produce the correct steering wheel angle for any given image or frame of a video that it is provided with. When provided with many images at a faster than expected rate, the model does struggle and often makes mistakes, however this is a problem that can be fixed with more training epochs as well as some refinement of the code. The simulator portion of the project was begun, with the image calibration portion of the real world 3D simulator completed; however due to time constraints the rest of the real world 3D simulator was not able to be completed. However, the team was still able to test the CNN through the use of the Udacity simulator. Overall, this project has produced a cohesive system of data collection, network model training, and network model testing that can be used to advance WPI's intelligent transportation program.

This project creates a wide variety of possible future directions. The first possible direction being the completion of the real world 3D simulator, at task that proved to be out of the scope of this project- however it is a task that would be well suited as the focus of a future project building on the progress completed here. Another obvious and compelling direction being the implementation of the trained CNN model not only on a simulator but also a real

active vehicle. Furthermore, a miniature car model could also be used to collect and test data from, barring access to a fully sized active vehicle

Table of Contents

List of Figures

List of Tables

1. Introduction

1.1 Motivation

Since the 1920's intelligent transportation in the form of autonomous vehicles has been the milestone and defining factor of people's idea of "*the future*". The concept of getting behind the wheel of a vehicle, sitting back, and letting the vehicle take control has long been conceived as an achievement or theoretical possibility that will happen sometime *in the future.* However, that future is here- and commercially available to consumers. Top car manufacturing companies such as Mercedes-Benz, General Motors, Nissan, Volvo, Audi, Ford, and notably Tesla, have been putting their best minds at work realizing this idea of the future today and have done so successfully with autonomous vehicles hitting the market as early as 2013. As of 2017, it is now legal for autonomous vehicles to operate in four US states, setting a precedent for many more states to follow. The field of intelligent transportation is currently one of the most prominent and popular fields in the industry, with huge advances being made on a regular basis.

The goal of this project was to advance WPI's intelligent transportation program through the creation of a data collection system, a Convolutional Neural Network (CNN) model for intelligent transportation, and a simulator to test the trained CNN model. By developing these tools, it was our aim to further enhance, advance, and aid WPI's intelligent transportation program.

1.2 Current State of the Field

Intelligent transportation has become a very prominent field in the automotive industry. This has led to a huge influx in funds allocated to researching and developing new and more efficient technologies for intelligent transportation. As a result, multiple research projects have been launched with the aim of developing data collection systems, Convolutional Neural Network models, and intelligent transportation simulators similar to this project.

NVIDIA's "DAVE-2 System" is one such research project. NVIDIA's "DAVE-2 System" project seeks to encapsulate end to end learning for self-driving cars. The project utilizes a convolutional neural network and a single front-facing camera to provide steering commands to a vehicle. NVIDIA's system observes a driver's input and behavior while on the road and uses pattern recognition to "teach" itself to navigate roads and unmarked paths. It collects data through three front facing cameras as well as the steering wheel angle of the vehicle. The outcome of this training is then evaluated in first a simulation, and then in a real world test. NVIDIA's simulation utilizes pre-recorded videos from a forward-facing camera on a human driven vehicle and then proceeds to transform the images into approximations of how their CNN model would react.

This project shares many similarities with NVIDIA's "DAVE-2 System". Both projects collect data from three front facing cameras and a vehicle's steering wheel angle; both project then proceed to feed this information to a neural network that utilizes pattern recognition to train itself. However, the two projects differ in terms of simulation. While NVIDIA utilizes pre-recorded video to approximate how its network model would operate, this project evaluates its progress through the use of a 3D real time simulation. In addition, this project is also capable of

training its neural network through the use of data collected in-simulation as well as real world data.

1.3 Proposed Design and Contributions

This project proposal was composed of three major core components: (1) the creation of data collection system, (2) a Convolutional Neural Network (CNN) model, and (3) a simulator.

(1) The data collection system proposed would be used to collect data from an active vehicle in real time, it would collect the steering wheel angle, speed, and images of the road from three separate angles.

(2) The CNN model proposed would then be trained with the collected data from the vehicle.

(3) The simulator would test/evaluate the effectiveness of the data collection system trained CNN model in a safe and controlled environment.

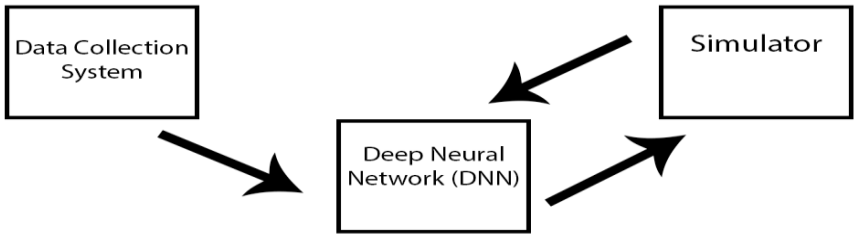


Figure 1 Proposed Design

The result of this project was proposed to be a modular collective system for intelligent transportation that could be implemented and tested in a variety of different vehicles to collect data, create models based of the data, and then test the data in the controlled environment of a simulator.

2. Project Logistics

2.1 Main Goal

The goal of this project was to advance WPI's intelligent transportation program through the creation of a data collection system, a Convolutional Neural Network (CNN) model for intelligent transportation, and a simulator to test the trained CNN model. By developing these tools, it was our aim to further enhance, advance, and aid WPI's intelligent transportation program.

This project was composed of three core components: the data collection system, a Convolutional Neural Network (CNN) model, and a simulator. The data collection system was used to collect data from an active vehicle in real time, it collected the steering wheel angle, speed, and images of the road from three separate angles. The CNN model was then trained with the collected data from the vehicle or with available datasets online. Then, having collected the data and trained a CNN model with it, the trained model was then tested in a simulator to evaluate its effectiveness in a safe and controlled environment.

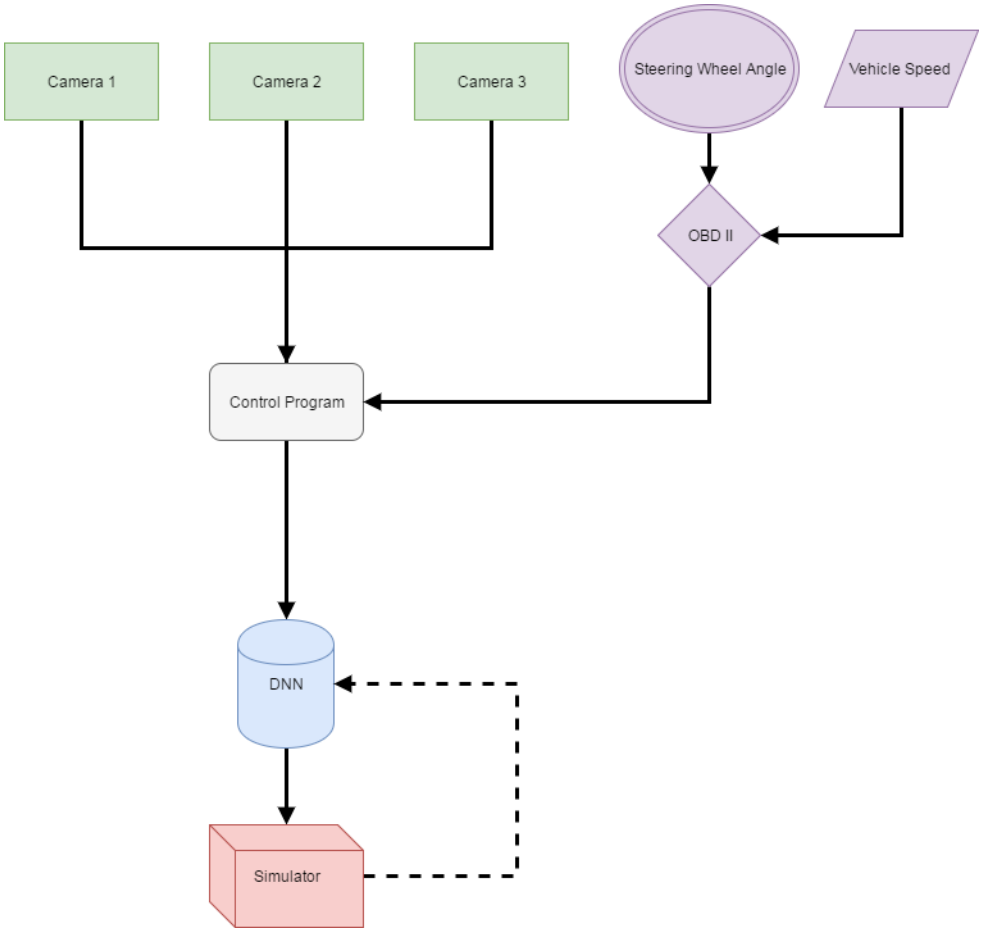


Figure 2 Main Goal

2.2 Project Objectives

The main objective of this project was to create a modular collective system for intelligent transportation that could be implemented and tested in a variety of different vehicles to collect data, create models based of the data, and then test the data in the controlled environment of a simulator. This objective was broken down into three sub-objectives.

2.2.1 Data Collection System

Create a data collection system modular in nature that can be placed on a test vehicle, used to collect data, then removed and taken back to the lab for either further

in-house testing, modifications, repairs, or safekeeping. Attaching and removing this data collection system should be quick and easy to accomplish.

2.2.2 Convolutional Neural Network Model

Develop a Convolutional Neural Network model that can be trained to operate a vehicle through real data gathered by the data collection system. The Convolutional Neural Network should also be able to be trained by simulated data created in a simulation. The CNN model should be capable of navigating in clearly marked roads with good lighting conditions.

2.2.3 Simulator

Develop a simulator to test a developed CNN model. The simulator should be capable of simulation a variety of driving conditions, obstacles, and road variations.

3. Data Collection System

3.1 Data Collection System Design

When designing the data collection system one of the first design aspects that needed to be decided upon was the location of the cameras and the amount of cameras needed. While having cameras located throughout the perimeter of the vehicle (front, side, and back) would be beneficial for a production vehicle, the scope of this project would not permit it. Instead, the team decided to focus on the front of the vehicle for the data collection system. In this data collection system, three cameras would be placed near the front of the vehicle, providing ample coverage of the road in front of the vehicle. The cameras would be spaced out evenly on the vehicle, with one two cameras near the outer edge of the vehicle and one located in the center. This positioning of the cameras would provide the data collection system with a very wide view of the road when all three camera images were stitched together.

Having decided on the camera count and position, the second logical design step was deciding on what camera model to use. The cameras would have to be compact in size, durable, and capable of taking pictures at a frequency of at least 10 frames per second. We decided to use a Chameleon3 camera- model number CM3-U3-13S2M-CS because of its open source nature software, competitive pricing, capture frequency and synchronous capabilities. Further information of the Chameleon3 camera may be found below in Table1.

Figure 3 Chameleon3 Camera

Chameleon3 Specifications	
Resolution	1288 x 964
Frame Rate	30 FPS
Megapixels	1.3 MP
Chroma	Mono
Machine Vision Standard	USB 3.1

Table 1 Chameleon3 Specifications

Once the camera model was decided, the third step was deciding on how the cameras were to be synchronized. One method for accomplishing this task was to utilize a frame grabber. A frame grabber would allow the team to utilize either USB or Ethernet to connect multiple cameras together and take synchronous shots. A second option would be to use a dedicated CPU and GPU for data collection system and write a program to synchronize the shots. Since the Chameleon3 that was to be used in the project had synchronous shot and open

source capabilities, it was decided that it would be more cost effective to write our own program to synchronize the camera shots.

Having determined the camera count, model, and method of synchronization, the fourth step in the data collection system design was determining a method of extracting and tabulating the vehicle's speed and steering wheel angle. The only viable solution for determining a vehicle's speed and steering wheel angle was determined to be through the OBDII port of the vehicle. Through the use of a CAN bus adapter and a custom computer program, the team would query the vehicle for its speed at a rate of 10 frames per second and query the steering wheel angle at a rate of 100 frame per second. This data would then be stored and tabulated with a corresponding time stamp.

The fifth and final step in the data collection system design was determining a way of synchronizing the images collected by the cameras with the speed and steering wheel angle collected through the OBDII port. In order to accomplish this, it was determined that the computer program for synchronizing the multiple cameras and the computer program for querying the vehicle for the speed and steering wheel angle would have to be merged together.

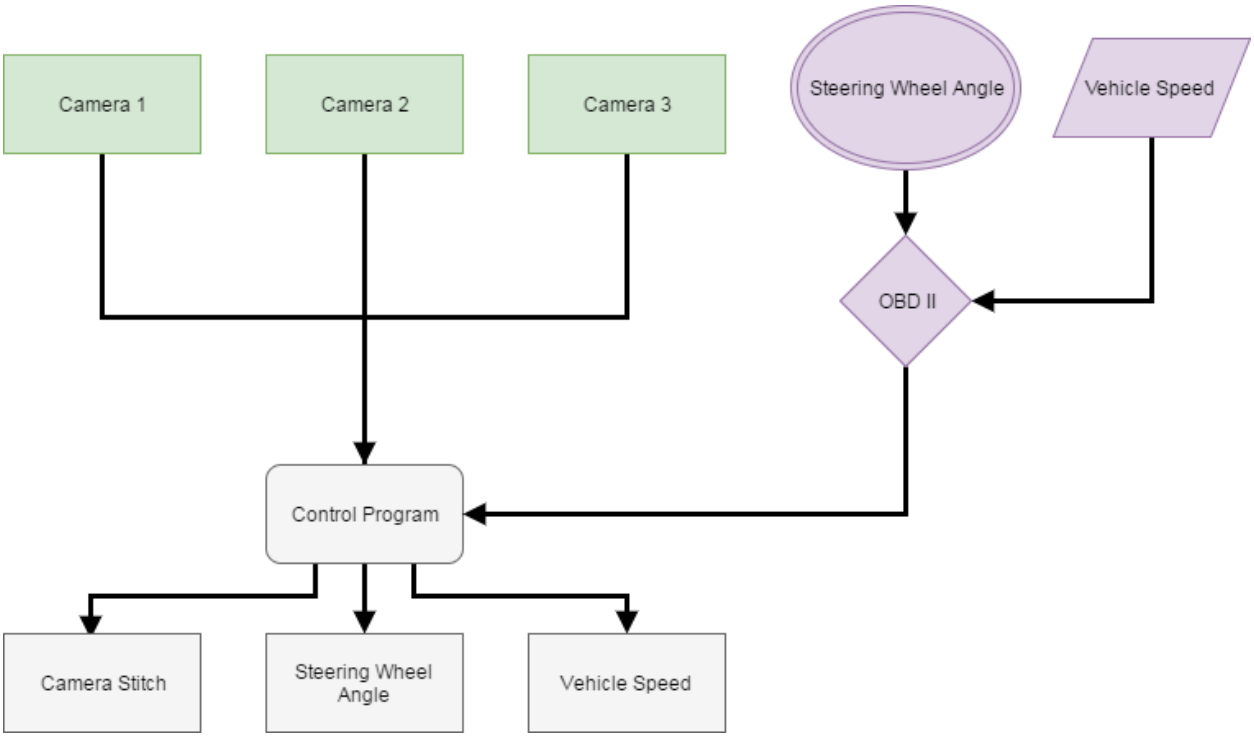


Figure 4 Data Collection System Overview

3.2 OBD II- On Board Diagnostics Background

On-board diagnostics- commonly referred to as OBD- is a feature available in all modern vehicles which allow them to perform self-diagnosis and provide vehicle reports to the user and/or manufacturer. OBD provides the user with detailed vehicle information from a variety of topics including but not restricted to: car speed, steering wheel angle, fuel and air detection, ignition, emissions control, transmission control, and vehicle state (drive, reverse, park, neutral). Because of these capabilities, OBD technology is often used by car owners and car manufacturers to provide simple diagnostic codes which allow the user to quickly identify problems with the vehicle.

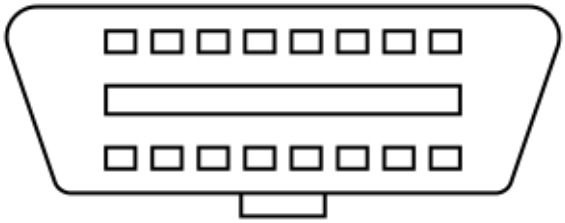


Figure 5 OBD-II Port

This project utilizes OBDII technology, an improved version of OBD which provides greater capability and standardization through different vehicles. OBDII are traditionally found and positioned in a location below the steering wheel and are usually hidden or covered by a removable compartment for ease of access. OBDII connectors are 16-Pin D-shaped connectors that transmit data over a CAN-bus protocol producing 4-digit hexadecimal PIDs (parameter IDs) for the user to read.

While the method of transmission is standard, manufacturers are not required to standardized the description of each individual PID value. Resulting in a wide range of vehicle-specific PID values that make it incredibly difficult to decipher the PID's description without direct information from the vehicle manufacturer.

3.3 Data Collection System Implementation

This section will detail the construction of the data collection system. The data collection system is comprised of 3 main components. (1) The image collection system comprised of 3 cameras linked in synchronous fashion, (2) the custom computer program that instructs the OBDII data logger to query and log the vehicle's speed and steering wheel angle while simultaneously instructing the image collection system to capture images and save them

in relation to the OBDII coordinates, and (3) the attachment mechanism used to attach the data collection system to the vehicle.

3.3.1 Image Collection System Set-Up

The image collection system built was composed of three CM3-U3-13S2C-CS cameras placed on a vehicle, facing the front of the car. While the general position and location of the cameras was pre-determined very early on in the project, the exact method of mounting was not. In order to mount the cameras, several approaches were considered and tested. The first approach was using T-slotted 80/20 aluminum bars to mount the cameras on top of the vehicle. The three cameras were first attached to an 80/20 aluminum bar using screws, then, the 80/20 aluminum bar was attached to the ski racks located on top of the test vehicle. A figure showing the 80/20 aluminum bar mounting mechanism design is shown below:

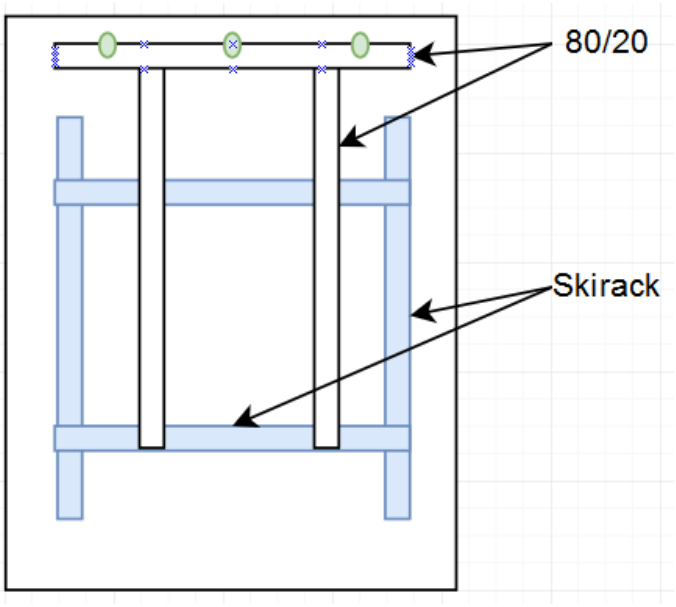


Figure 6 80/20 Aluminum Design

The images below display the actual implementation of the 80/20 aluminum bar mounting mechanism:

Figure 7 80/20 Aluminum Design Example 1

Figure 8 80/20 Aluminum Design Example 2

After implementing the 80/20 aluminum bar design, three problems became apparent. The first problem was that the ski-rack on the vehicle was not attached tightly enough to the vehicle; when the vehicle was driven, the ski-rack would shift laterally back and forth in small increments, causing the cameras to shift positions constantly.

After testing various other vehicles, it soon became apparent that this problem was not isolated to the test vehicle. Furthermore, the ski-racks could not be tightened to eliminate this problem. The second problem was that the cameras were located too far back along the roof of the vehicle. According to the team's measurements, objects within 2 meters of the front of the vehicle could not be captured by the cameras due to their extended distance from the front of the vehicle. The third problem in this configuration was that the cameras were unprotected and exposed to the elements- because of this, data collection would be limited to clear weather conditions, since any rain or snow would damage the cameras.

In order to resolve these drawbacks, the team decided to place the cameras on the dashboard of the vehicle. One camera in the center of the dashboard, flanked by the other two cameras on the sides. By doing this, the cameras would experience approximately the same range of view as a human driver. The cameras were attached to the dashboard through the use of camera mounts and adhesive. The following figure shows the configuration that was used:

Figure 9 Camera Layout

The image collection system posed a unique challenge in determining a way of powering the three cameras, storing the information from the cameras, storing the information from the OBD II port, and running the multithreaded program- all while inside a moving vehicle. Since this data was to be collected on in real time in a moving vehicle, a dedicated desktop with a dedicated GPU was impossible. Instead, it was decided that a high-end laptop with a dedicated GPU would be utilized in conjunction with an independently powered USB hub to power the three cameras. To solve this

challenge, the team acquired a Dell XPS 15 with an NVIDIA GTX 960M to meet the GPU needs. Then, the three cameras were powered by a self-contained USB hub located inside the vehicle while the cameras were connected to the mobile GPU through the use of three individual USB3.1 cables. The Dell XPS 15 laptop was used to house both the program instructing the cameras to capture images as well as store the images acquired.

3.3.2 Multi-Thread Image & Data Logger Program Overview

In order to instruct the three cameras in the image collection system to capture images synchronously, the construction of a computer program was necessary. The program operated as such: After a trigger was sent to camera, the image would first be stored in the camera buffer. Then, a retrieve buffer function would be used to retrieve the image into memory, after which, the save image function would be used to save the image to the disk. In the program, a total of three threads are used: one to trigger the cameras simultaneously and two threads to collect the steering wheel angle and speed data.

In the program, a total of three threads were started: the first thread was used for controlling the cameras, a second thread was used for querying the OBD-II port for speed and data, and a third one was used to monitor the steering wheel angle data. Once the camera thread was started, it would trigger the cameras every 100 ms to achieve a frame rate of 10 FPS. After a camera was triggered, the image would first be stored in the camera buffer. Then, a retrieve buffer function would be used to retrieve

the image into memory, after which, the save image function would be executed in a separate thread to save the image to the disk.

In order to retrieve steering wheel angle data and speed data, AT commands for ELM327 were used. For steering wheel angle data, 'ATCRA 025 \r' was first sent. This command would instruct ELM327 to only send back frames with id 025, which represents steering wheel angle data frame. Then, 'ATMA \r' was sent so that the chip would start receiving data. To get the speed data, command '010D \r' was sent every time the cameras were triggered. On receiving this command, the chip would query OBD-II port for current vehicle speed and send it back to the program. A Matlab program was written to decode the packets received into the steering wheel angle and the speed data, this data was then saved in a .mat file.

During experimentation, the team discovered that triggering the camera, retrieving the buffer, and saving an image on the same thread, would take around 300ms per image since writing an image to disk takes a long time. This 300ms time period did not meet the 10 frames per second requirement. To fix this problem, the cameras are triggered in one thread and the images are saved to disk in another thread. The images that were not yet saved were pushed into a queue until they could be saved.

During this process, it was also discovered that triggering the camera in low light conditions significantly lengthened the time that it took to capture an image. This was because when auto shuttering was enabled, the shutter time was much longer than when light condition were adequate. An upper bound for the shutter time of the camera

was subsequently set to make sure that even when the light conditions were not adequate, the desired frame rate would be maintained.

3.3.3 Multi-Thread Image & Data Logger Program Logistics

This section will describe the Multi-Thread Image & Data Logger Program in further detail:

This program used the FlyCapture2 library for basic camera control. At the start of the program, three threads are created, one for getting steering wheel angle data, one for getting speed data and one for controlling the cameras. The code for construction of threads and initialization of semaphores are shown in the following figure:

```
798 int main() {
799     clockFiles[0].open("./img/clock_0.txt");
800     clockFiles[1].open("./img/clock_1.txt");
801     clockFiles[2].open("./img/clock_2.txt");
802     max_gap.open("./img/gap.txt");
803     pftr.open("./img/pftr.txt");
804     for(int i = 0; i < 3; i ++) {
805         sem_init(&(save_sems[i]), 0, 0);
806     }
807     for(int i = 0; i < 4; i ++) {
808         sem_init(&trigger_sem[i], 0, 0);
809     }
810     thread t[3];
811     for(int i = 0; i < 3; i ++) {
812         t[i] = thread(thread_func, i);
813     }
814     for(int i = 0; i < 3; i ++) {
815         t[i].join();
816     }
817     for(int i = 0; i < 3; i ++) {
818         clockFiles[i].close();
819     }
820     return 0;
821 }
```

Figure 10 Code used to start threads and initialize semaphores.

Three clock files, a gap file and a pftr file were first opened to record the timestamps used for debugging purpose. Then, three semaphores were initialized to control the saving image threads corresponding to three cameras. Another four semaphores were initialized in order to synchronize the triggering threads for three cameras and the thread used for querying speed information. Finally, three threads are created. One running the function used to get steering wheel angle data, one running the function used to get the speed data and one controlling the cameras.

The following code snippet showed the function used to retrieve speed data:

```
112  void retrieveSpeedData() {
113      string portName;
114      portName = SPEED_PORT;
115      int fd = open((char*)portName.c_str(), O_RDWR | O_NOCTTY | O_SYNC);
116      if(fd < 0) {
117          cout << "Open failed" << endl;
118          return;
119      }
120      ofstream fileStream;
121      fileStream.open(speedFile.c_str());
122      set_interface_attribs(fd, B115200, 0);
123      set_blocking(fd, 0);
124      string msg = "010D \r";
125      vector<char> val;
126      while(1) {
127          sem_wait(&trigger_sem[3]);
128          write(fd, (char*)msg.c_str(), msg.length());
129          usleep(20000);
130          char buff[2];
131          int rtnCnt = 0;
132          while(read(fd, buff, sizeof(buff) - 1) > 0) {
133              if((int)buff[0] == '\n') {
134                  rtnCnt ++;
135                  auto ms = std::chrono::duration_cast<std::chrono::milliseconds>(std::chrono::system_clock::now().time_since_epoch());
136                  fileStream << " Frame_"<< gCnt << "_" << ms.count() << endl;
137                  val.clear();
138                  if(rtnCnt == 3) {
139                      break;
140                  }
141              } else {
142                  fileStream << buff[0];
143                  val.push_back(buff[0]);
144              }
145          }
146          sem_post(&main_sem);
147          if(FLAG == 0) {
148              fileStream.close();
149              break;
150          }
151      }
152  }
```

Figure 11 Code used to get speed data from OBD-II port.

At the start of this function, communication between computer and the ELM327 chip used to get speed data was established. The baud rate of the serial port was set to 115200 and the port was set to non-blocking. The file used to store the speed information was then opened. Then, after the semaphore controlling the speed collection thread was posted, '010D \r' was sent through the serial port. The returned result was written to the file along with a timestamp and the current frame count. Finally, the function posted to main semaphore to indicate that it was finished.

To record the steering wheel angle, the following code was used:

```
void sensor_func(int flg) {
    string portName;
    portName = SW_ANGLE_PORT;
    int fd = open((char*)portName.c_str(), O_RDWR | O_NOCTTY | O_SYNC);
    if(fd < 0) {
        cout << "Open failed" << endl;
        return;
    }
    ofstream fileStream;
    fileStream.open(swFile.c_str());
    set_interface_attribs(fd, B115200, 0);
    set_blocking(fd, 0);
    string msg = "ATZ \r";
    write(fd, (char*)msg.c_str(), msg.length());
    sleep(1);
    msg = "ATCRA 025 \r";
    write(fd, (char*)msg.c_str(), msg.length());
    sleep(1);
    msg = "ATMA \r";
    write(fd, (char*)msg.c_str(), msg.length());
    sleep(1);
    // Read data.
    char buff[2];
    vector<char> val;
    while(read(fd, buff, sizeof(buff) - 1) > 0) {
        if((int)buff[0] == '\n') { // \n 10; \r 13
            auto ms = std::chrono::duration_cast<std::chrono::milliseconds>(std::chrono::system_clock::now().time_since_epoch());
            fileStream << " Frame_"<< gCnt << "_" << ms.count() << endl;
            continue;
        } else {
            fileStream << buff[0];
        }
    }
    fileStream.close();
}
```

Figure 12 Code used to get steering wheel angle data from OBD-II port.

Initially, communication between computer and the ELM327 chip used to collect steering wheel angle data was established and the file used to store the data was opened. ‘ATZ \r’ was then sent to initialize ELM327. Next, ‘ATCRA 025 \r’ was sent to configure the chip so that it only get the frame with id number 025, which corresponds to the steering wheel angle. Finally, ‘ATMA \r’ was sent so the chip started to receive data. The frames was then saved to the file along with a timestamp and the current frame number. Because the frame rate for the steering wheel angle data was 100 fps

(10 ms per frame), 10 steering wheel angle data was recorded for one frame. The average was calculated and used as the label of the corresponding frame.

In order to control the cameras, the ‘camera_func’ was written. The cameras were set to using software trigger, with 4 buffers, 5 second grab timeout, ‘BUFFER_FRAMES’ as grab mode and using high performance retrieve buffer with a packet size of 4689 bytes. An upper bound of 8.69ms was set for shutter time in order to maintain a constant 10 frames per second. The image taken was in 1288 * 964, which is the maximum resolution supported by the camera. The retrieved images were also set to contain a timestamp information.

To trigger the three cameras at the same time and making sure that each image is taken within 100 ms, the following code was used:

```
606         while(1) {
607             gCnt++;
608             // Poll for trigger ready.
609             auto t0 = std::chrono::high_resolution_clock::now();
610             for(int i = 0; i < 4; i ++) {
611                 sem_post(&trigger_sem[i]);
612             }
613             for(int i = 0; i < 4; i ++) {
614                 sem_wait(&main_sem);
615             }
616             auto t1 = std::chrono::high_resolution_clock::now();
617             std::chrono::duration<float> dur = t1 - t0;
618             auto dur_ms = std::chrono::duration_cast<std::chrono::milliseconds>(dur);
619             max_gap << dur_ms.count() << endl;
620             if(dur_ms.count() * 1000 < sleepTime) {
621                 usleep(sleepTime - dur_ms.count() * 1000);
622             } else {
623                 pftr << "dropped frame ..." << endl;
624             }
625             if(FLAG == 0) {
626                 break;
627             }
628         }
```

Figure 13 Camera Sync Code

The main thread first post to the four trigger semaphores which control three camera threads and the speed collection thread. Then, it waits for all four threads to finish. The difference between 100 ms and the longest time used by the four threads was then calculated, and the main thread sleep for this period of time before triggering these four threads again. So, a constant 10 frames per second can be achieved and also all the cameras are triggered at the same time.

The function used to trigger the cameras was shown below:

```
672  void triggerAndRetrieveBuffer(Camera* camera, int cameraCnt) {
673      Error error;
674      while(1) {
675          PollForTriggerReady(camera);
676          sem_wait(&trigger_sem[cameraCnt]);
677          auto t0 = std::chrono::high_resolution_clock::now();
678      #ifdef SOFTWARE_TRIGGER_CAMERA
679          bool retVal = FireSoftwareTrigger(camera);
680          if(!retVal) {
681              cout << "Error firing software trigger" << endl;
682              return;
683          }
684      #endif
685          Image tmpImg;
686          error = camera -> RetrieveBuffer(&tmpImg);
687          if(error != PGRERROR_OK) {
688              PrintError(error);
689              return;
690          }
691          auto t1 = std::chrono::high_resolution_clock::now();
692          std::chrono::duration<float> fs = t1 - t0;
693          std::chrono::milliseconds fs_ms = std::chrono::duration_cast<std::chrono::milliseconds>(fs);
694          int ms_cnt = fs_ms.count();
695          pftr << ms_cnt << endl;
696          pair<int,Image> tmpPair = make_pair(gCnt,Image());
697          tmpPair.second.DeepCopy(&tmpImg);
698          cam_images.at(cameraCnt).push(tmpPair);
699          sem_post(&(save_sems[cameraCnt]));
700          t1 = std::chrono::high_resolution_clock::now();
701          fs = t1 - t0;
702          fs_ms = std::chrono::duration_cast<std::chrono::milliseconds>(fs);
703          ms_cnt = fs_ms.count();
704          clockFiles[cameraCnt] << ms_cnt << endl;
705          if(FLAG == 0) {
706              cout << "return here" << endl;
707              sem_post(&main_sem);
708              return;
709          }
710          sem_post(&main_sem);
711      }
712  }
```

Figure 14 Camera Trigger Code

This function first calls 'PollForTriggerReady' to make sure that the camera is currently ready to be triggered. Then, it waits for the 'trigger_sem' to be increased by the main semaphore. After it is signaled, a software trigger is fired and the buffer is retrieved to get the image. Then, the retrieved image is pushed to a queue for the save image thread to save it and the save image thread is signaled.

The save image thread was shown in the following:

```
714  void saveImage(unsigned int cam_num) {
715      Error error;
716      //int cnt = 0;
717      while(1) {
718          int sv;
719          sem_getvalue(&save_sems[cam_num], &sv);
720          cout << sv << endl;
721          if(sv == 0 && FLAG == 0) {
722              cout << "Before save image return" << endl;
723              isSaving[cam_num] = 0;
724              return;
725          }
726          isSaving[cam_num] = 1;
727          sem_wait(&(save_sems[cam_num]));
728          int tmpId = cam_images.at(cam_num).front().first;
729          Image img = cam_images.at(cam_num).front().second;
730          cam_images.at(cam_num).pop();
731          Image convertedImage;
732          error = img.Convert(PIXEL_FORMAT_RGB8, &convertedImage);
733          if(error != PGRERROR_OK) {
734              PrintError(error);
735              isSaving[cam_num] = 0;
736              return;
737          }
738          // Get time stamp:
739          TimeStamp timestamp = img.GetTimeStamp();
740          long long tp = timestamp.seconds*1e6 + 1000 * (timestamp.microSeconds / 1000);
741          ostringstream filename;
742          filename << "./img/cam"<<cam_num<<"/image_" << cam_num << "_"<<tmpId<< ".jpg";
743          cout << "image_" << cam_num << "_"<<tmpId << "_" << tp << " saved" << endl;
744          // usleep(70000);
745          // error = convertedImage.Save(filename.str().c_str());
746          error = convertedImage.Save(filename.str().c_str());
747          if(error != PGRERROR_OK) {
748              PrintError(error);
749              isSaving[cam_num] = 0;
750              return;
751          }
752          isSaving[cam_num] = 0;
753      }
754  }
```

Figure 15 Camera Image Thread Code

When this thread is triggered, it will get an image from the global queue and save it to the corresponding directory with the camera number, frame number and timestamp as the name of the image.

3.3 Data Collection System Results

Using this data collection program, the images can be taken at a rate of 10 frames per second and the steering wheel angle and speed corresponding to the images were recorded. These data can then be used to construct a real-world driving simulator and can also be used as the training data for our neural network.

4. Convolutional Neural Network Model for Lane Keeping

4.1 Introduction to End-to-End Learning for Lane Keeping

An end-to-end learning approach was implemented to obtain a steering wheel angle output based on an input frame. The Convolutional Neural Network model the team used was based on the network used by comma.ai. The model was trained on a Udacity dataset and was evaluated by using the output to control the car in autonomous mode- this was done on the Udacity simulator at a constant speed.

A traditional approach to lane keeping uses computer vision techniques such as Hough transformation and edge detection to try to identify lanes on a given frame. Control logics are then applied to adjust the steering wheel angle based on the detected lanes. The problem for this approach is that performance relies heavily on how clearly the lanes can be extracted. Because road conditions are very complicated, manually defined features are not able to identify lanes perfectly in complex situations. However, for the end-to-end learning approach using Convolutional Neural Network, the steering wheel angle output is based on the inputted raw image. The parameters of the neural network is optimized automatically using back-propagation based on the mean squared error between the predicted angle and the labeled angle for the training image. Because the neural network is self-optimized, labeling the lanes on the image and extracting features manually is not needed, which makes pre-processing the images much easier. If training data for an abundant collection of road conditions are available, the prediction given by the neural network will be very accurate.

4.2 Convolutional Neural Network Overview

In this portion of the project, the team set about to develop a Convolutional Neural Network model that could be trained to operate a vehicle through real data gathered by the data collection system and/or through pre-existing datasets. The goal of this portion of the project was to create a CNN model capable of navigating clearly marked roads or paths with good lighting conditions.

The section of the project is broken up into two main sections. The first part details the creation of the CNN model as well as the training of the model. The second part consists of testing the CNN model created.

4.3 Implementation of the Convolutional Neural Network

To implement this convolutional neural network, Keras, a high-level neural networks API, written in Python and running on top of Tensorflow, was used. The neural network structure was shown in the following image:

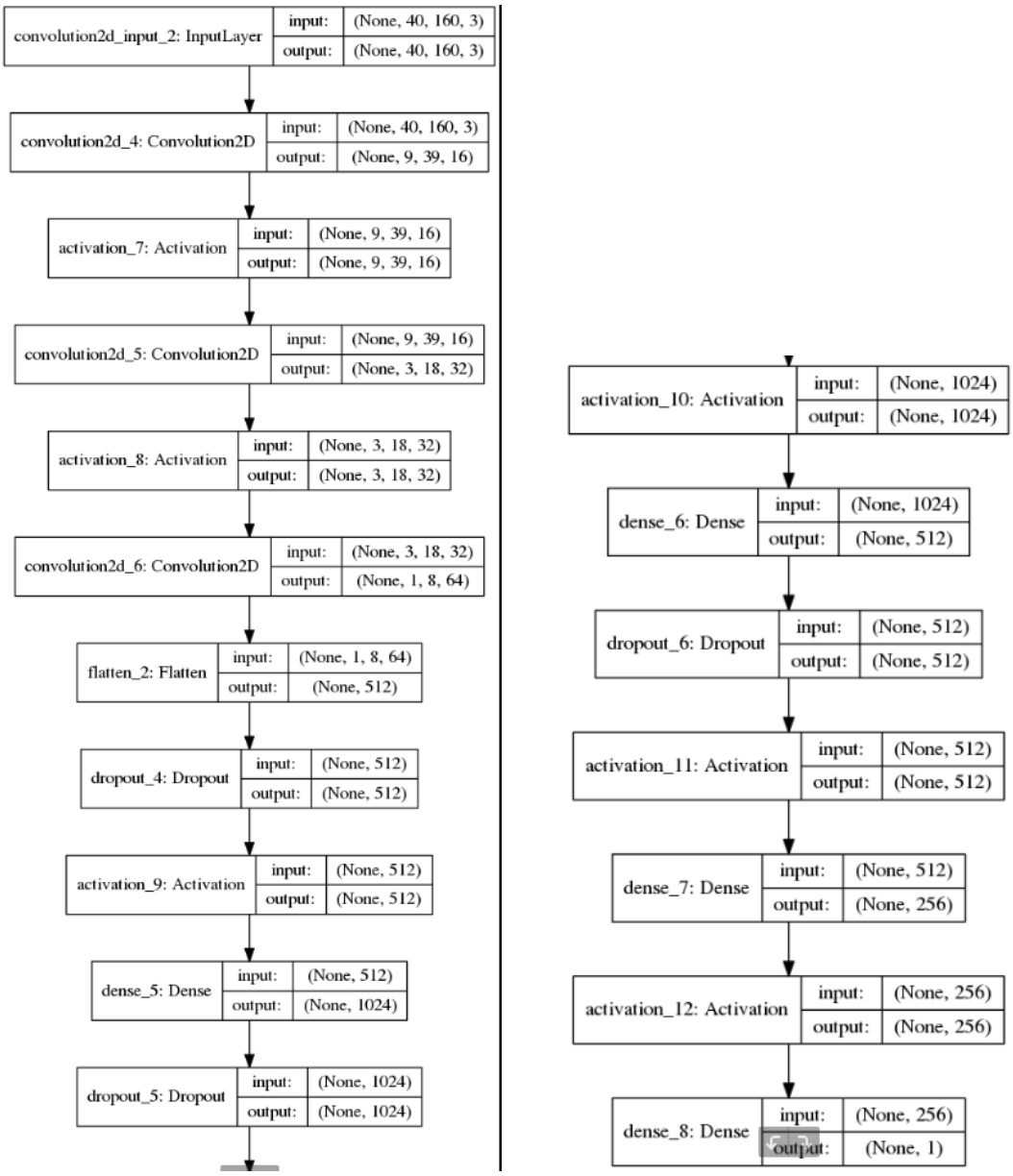


Figure 16 CNN Model

For this implementation, the input images were pre-processed before being fed into the neural network. First, in pre-processing, the upper half of the images were removed because the removed portion did not affect the result of lane keeping. Then the remaining images were shrunk by 0.5 to reduce memory usage and training time. Next, the images were converted from RGB to YUV. Finally, the images were adjusted to zero mean and unit variance to ensure

that convergence could be reached quickly. The code below shows the process of pre-processing images:

```
13    def processImg(img):
14        yuv = cv2.cvtColor(img, cv2.COLOR_RGB2YUV)
15        img = img[80:160, :, :]
16        img = ndimage.interpolation.zoom(img, [0.5, 0.5, 1])
17        mean = np.mean(img)
18        img = img - mean
19        s = np.std(img)
20        img = img / s
21        return img
```

Figure 17 Image Pre-Processing

Since three images were captured by left, center and right cameras at each frame, the steering wheel angle corresponding to the left image was decreased by 0.25 and the angle corresponding to the right image was increased by 0.25. By doing this, the captured images were able to be fully utilized.

The code used to implement the convolutional neural network is shown in the following image:

```
21    def cnn(inputshape):
22        global conv1
23        model = Sequential()
24        # Batch size is none so the model can process any batch size.
25        conv1 = Convolution2D(16, 8, 8, border_mode='valid', input_shape=inputshape, subsample=(4, 4), init='glorot_uniform', bias=True)
26        model.add(conv1)
27        model.add(Activation('relu'))
28        model.add(Convolution2D(32, 5, 5, border_mode='valid', subsample=(2, 2), init='glorot_uniform', bias=True))
29        model.add(Activation('relu'))
30        model.add(Convolution2D(64, 3, 3, border_mode='valid', subsample=(2, 2), init='glorot_uniform', bias=True))
31        model.add(Flatten())
32        model.add(Dropout(0.2))
33        model.add(Activation('relu'))
34        model.add(Dense(1024, init='glorot_uniform', bias=True))
35        model.add(Dropout(0.5))
36        model.add(Activation('relu'))
37        model.add(Dense(512, init='glorot_uniform', bias=True))
38        model.add(Dropout(0.5))
39        model.add(Activation('relu'))
40        model.add(Dense(256, init='glorot_uniform', bias=True))
41        model.add(Activation('relu'))
42        model.add(Dense(1, init='glorot_uniform', bias=True))
43        # Print model summary here.
44        model.summary()
45        # model.compile(optimizer='adam', loss='mse')
46        return model
```

Figure 18 CNN Implementation Code

The input image to the model has a size of 40 * 160. The first convolutional layer has a patch size of 8 * 8 with 16 output channels and a stride size of 4. Then, the outputs are passed through a rectified linear unit (ReLU) layer. The second convolutional layer has a patch size of 5 * 5, 32 output channels and a stride size of 2. This layer is also followed by a ReLU layer. The third convolutional layer has a patch size of 3 * 3, 64 output channels and a stride size of 2. Then, after flattening, a dropout rate of 20% was applied and another ReLU layer was added. Finally, four fully connected layers with 1024, 512, 256 and 1 neurons were added to output the steering wheel angle. Xavier initialization was used to initialize the weight for all the layers. Using xavier initialization ensures that the scale of initialization is based on the input and output neurons.

To compile and train the neural network, the following code was used:

```
input_size = (40, 160, 3)
model = cnn(input_size)
plot(model, to_file='model.png', show_shapes=True)
adam = Adam(lr=1e-4, beta_1=0.9, beta_2=0.999, epsilon=1e-08, decay=0.0)
model.compile(optimizer=adam, loss='mse')
try:
    model.load_weights('model.h5')
except Exception as e:
    print(e)
for i in range(len(files)):
    img_train, img_valid, label_train, label_valid = loadcsv('/home/mengwen/Desktop/gasoto_data/IMG/', files[i], discard[i])
    history = model.fit_generator(getData(img_train, label_train, input_size, batch_size[i], flip_img[i]), \
                                  nb_epoch=epochs[i], verbose=1, samples_per_epoch=len(img_train), \
                                  validation_data=getData(img_valid, label_valid, input_size, batch_size[i], flip_img[i]), \
                                  nb_val_samples=len(img_valid))

with open('model.json', 'w') as f:
    f.write(model.to_json())
model.save_weights('model.h5')
```

Figure 19 CNN Compilation

The Adam optimizer, which is a method for stochastic optimization, provided by Keras, was used to optimize the neural network. And mean squared error between the predicted and labeled angle was used as a loss function. In order to train the network, training data captured

from Udacity simulator was used. A figure showing the steering wheel angle distribution of the training data was shown below:

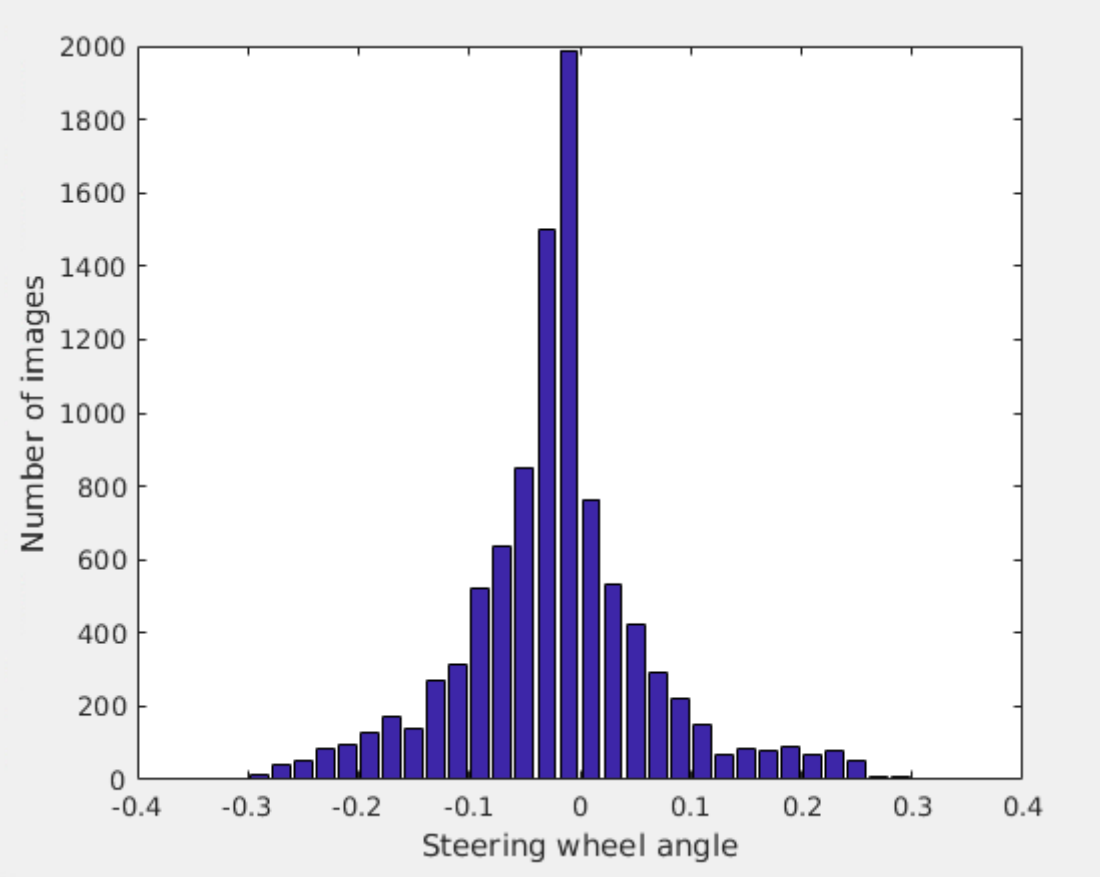


Figure 20 Steering Wheel Angle Distribution

From the distribution, we can tell that in most frames, the car is driving nearly straight, which means that the steering wheel angle is 0. Since there were not enough training data on sharp turns, the car did not perform very well when encountering a very sharp turn.

Overall, 30 epochs of training were performed on all the training data with a batch size of 64 and 16 epochs of training were performed on the images with steering wheel angle not equal to 0.

4.4 Results of the Convolutional Neural Network

In order to evaluate the convolutional neural network, a variety of test were conducted. The first test was done through the use of an online framework called 'DeepTesla' was used. DeepTesla is a online platform for testing end-to-end steering models. The code used in DeepTesla is shown in the following figure:

```
    "network" : [
        { "type" : "input", "out_sx" : 200, "out_sy" : 66, "out_depth" : 3 },
        { "type" : "conv", "sx" : 5, "filters" : 16, "stride" : 2, "pad" : 2, "activation" : "relu" },
        { "type" : "pool", "sx" : 3, "stride" : 2},
        { "type" : "conv", "sx" : 3, "filters" : 32, "stride" : 2, "pad" : 2, "activation" : "relu" },
        { "type" : "conv", "sx" : 3, "filters" : 64, "stride" : 2, "pad" : 2, "drop_prob" : 0.2,
            "activation" : "relu" },
        { "type" : "fc", "num_neurons": 1024, "drop_prob": 0.5, "activation" : "relu"},
        { "type" : "fc", "num_neurons": 512, "drop_prob": 0.5, "activation" : "relu"},
        { "type" : "fc", "num_neurons": 256, "activation" : "relu"},
        { "type" : "regression", "num_neurons" : 1 }
    ],
    "trainer" : { "method" : "adadelta", "batch_size" : 32, "l2_decay" : 0.0001 }
```

Figure 21 CNN Test Code

Since the input images from DeepTesla had different sizes than the training images the team used in the project, the patch size of the first convolutional layer was changed to 5 and a pooling layer with patch size 3 and stride 2 was added.

The following plot shows the change of mean squared error as the number of training images fed to the neural network increased:

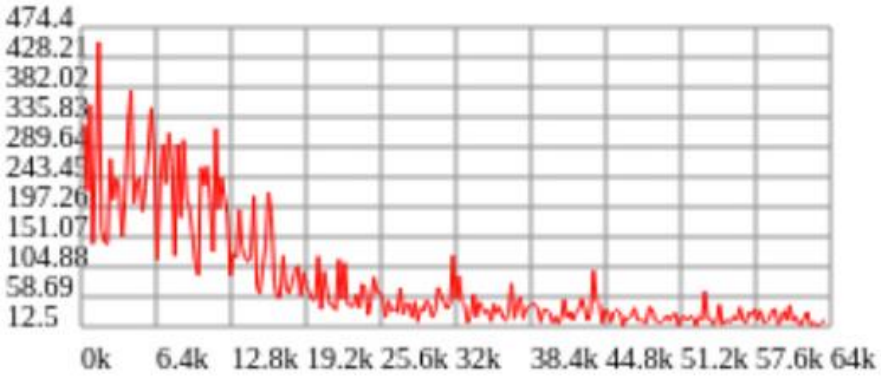


Figure 22 Mean Squared Errors

As the plot above shows, the mean squared error decayed as the network was fed by more images. After about 57k images were used for training, the error reached a stable value of less than 10.

The following figures show visualizations of the images generated by the first and second convolutional layers along with the corresponding training image:

Figure 23 Training image, actual angle and predicted angle

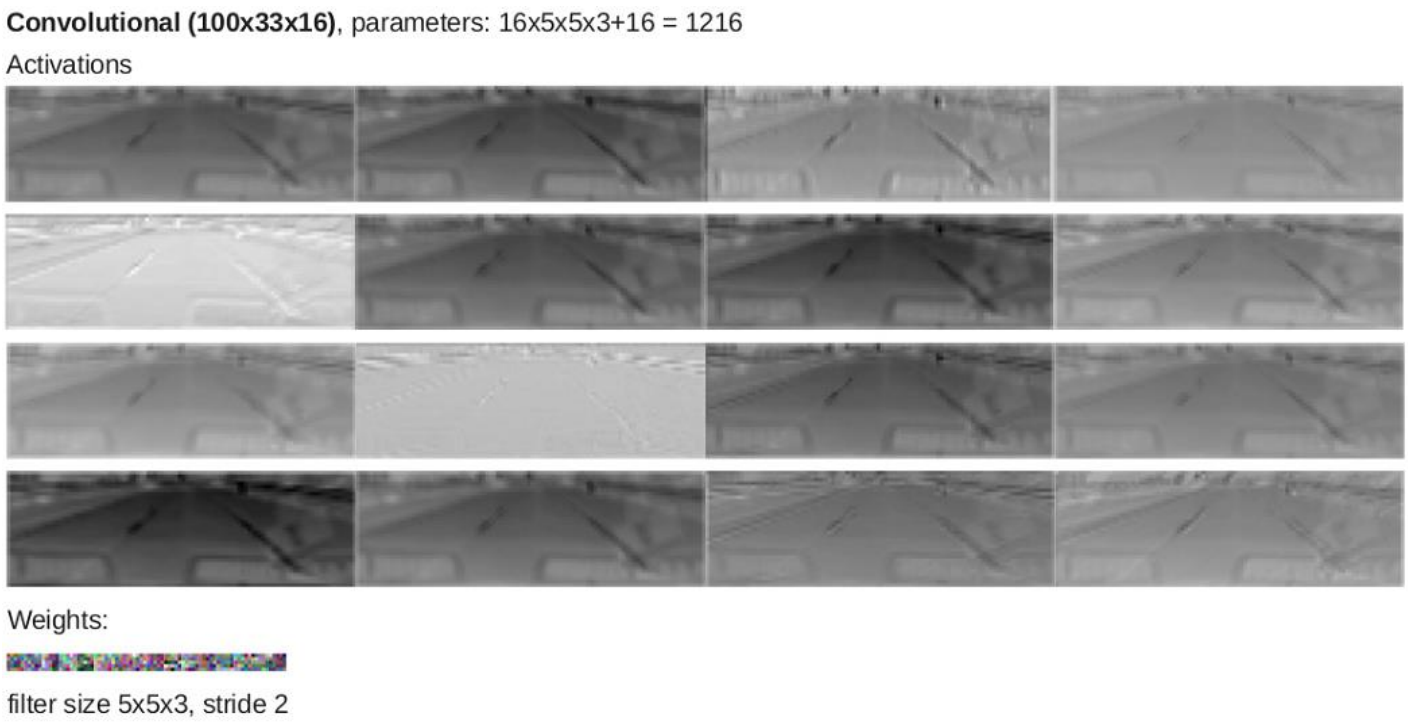
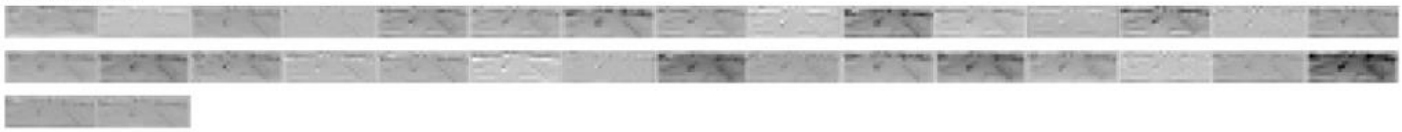


Figure 24 Visualization of the images generated by the first convolutional layer

Convolutional (26x9x32), parameters: 32x3x3x16+32 = 4640

Activations



Weights hidden, too small

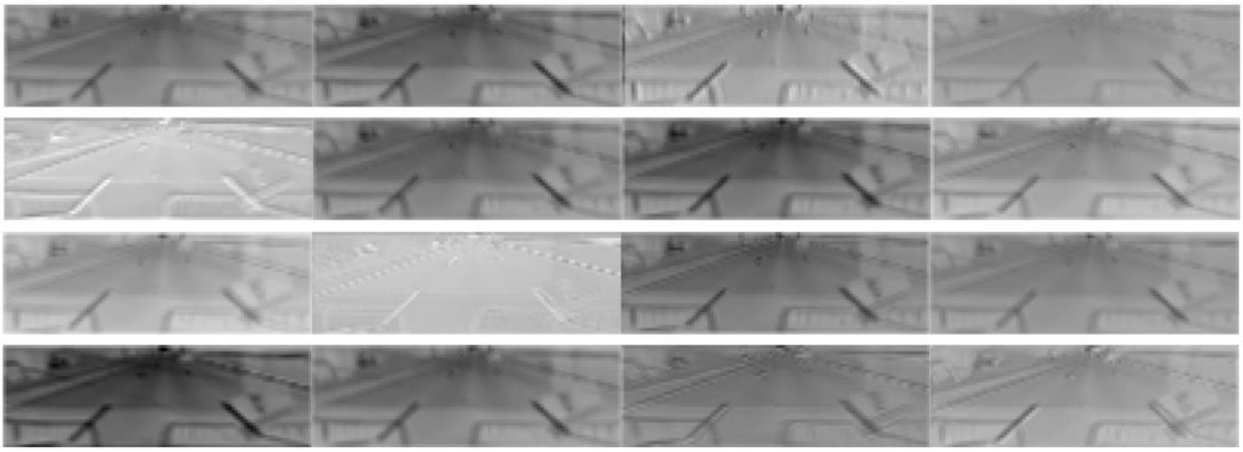
filter size 3x3x16, stride 2

Input (200x66x3)

Activations (actual angle: 1.5, predicted angle: 1.7)

Convolutional (100x33x16), parameters: 16x5x5x3+16 = 1216

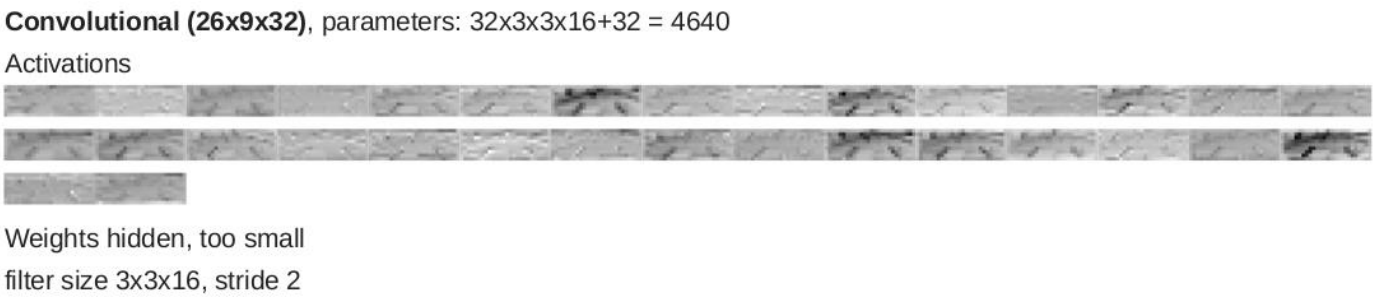
Activations



Weights:

filter size 5x5x3, stride 2

Figure 25 Visualization of the images generated by the second convolutional layer



From the figures above, one can tell that lane markings were automatically extracted as features by the convolutional layers. And the predicted angle was almost the same as the labeled angle.

After evaluating the convolutional neural network on DeepTesla online platform, the team proceeded to do a second test, this time using the Udacity driving simulator. Udacity is a machine learning specific simulation tool with a simple design and environment; Udacity allows users to both test and train CNN models. For this project, Udacity driving simulator was used to determine whether the neural network can successfully keep the car in lane. The overall performance of the CNN model was good. The car was able to drive a whole loop around the simulated track without human intervention. The only problem experienced in the simulation occurred when the vehicle encountered sharp turns; in this case the steering wheel angle predicted by the neural network was sometimes smaller than the actual angle need so the car would drive out of lane for a very brief period time before automatically adjusting back. A video showing the convolutional neural network automatically keeping the car in lane is shown in the following link:

https://drive.google.com/file/d/0Bx0yLSU7DR9qRXlKUUdMOWpPVXM/view?usp=sharing

5. Camera Calibration [Simulator]

6.1 Introduction

To build a simulator using real world data, we need to be able to transform a 3D world coordinate to 3D camera coordinate, and also transform a 2D camera coordinate to pixel coordinates in the image frame. As a result, it is necessary for us to find the extrinsic matrix and intrinsic matrix for the cameras we used.

6.2 Intrinsic Matrix

In order to map the camera coordinates to the pixel coordinates in the image frame, the intrinsic matrix need to be found. The intrinsic matrix can be expressed as the following:

$$\begin{bmatrix} f_x & 0 & 0 \\ s & f_y & 0 \\ c_x & c_y & 1 \end{bmatrix}$$

In this matrix, fx, fy represents the focal length in pixels. S represents the skew coefficient between x and y axis and cx, cy represents the offsets. To transform a camera based 2D coordinate to 2D point in the image plane, the following formula was used:

$$\boldsymbol{p}_i = K \cdot \boldsymbol{p}_c$$

where pi represents the homogenous pixels on the image plane, pc represents the camera-based 3D coordinate and K represents the intrinsic matrix.

6.3 Extrinsic Matrix

Extrinsic matrix was used to transform a 3D world coordinate to a 3D camera coordinate. The extrinsic matrix can be represented as the following:

$$R_{w,c}\boldsymbol{t}_{w,c}$$

Rw,c in the formula represents the rotation matrix of the camera system, and tw, c in the formula represents the translation of the optical center from the origin of the world coordinate. In order to transform a point from world coordinate to camera coordinate, the following formula was used:

$$\boldsymbol{p}_c = \left(R_{w,c}\boldsymbol{t}_{w,c}\right)\boldsymbol{p}_w$$

where pc represents the 3D camera coordinate, pc represents the camera-based 3D coordinate, Rw, c represents the rotation matrix and tw, c represents the transformation matrix. Putting the intrinsic matrix and extrinsic matrix together, a 3D world coordinate can be transformed to pixel coordinate on image plane. The following formula was used for this operation:

$$\boldsymbol{p}_i = K\cdot\left(R_{w,c}\boldsymbol{t}_{w,c}\right)\boldsymbol{p}_w$$

In this formula, Kis a 3 by 3 matrix representing the intrinsic matrix for the camera, (Rw, ctw, c)is a 3 by 4 matrix represe the extrinsic matrix for the camera. Camera calibration was done to

find the intrinsic matrix and the extrinsic matrix using the measured image plane coordinates and the world coordinates.

6.4 Implementation for finding camera parameters

Initially, 'Camera Calibrator' application from computer vision toolbox in Matlab was used to find the camera parameters for individual cameras. A problem for this approach is that since there are small errors when calculating intrinsic and extrinsic matrices, when the camera' locations were calculated based on camera parameters we got, it didn't correspond to the actual location of the cameras. Since the relationship between cameras are already known, these three cameras are treated as two sets of stereo camera, and the 'Stereo Camera Calibrator' application was used. This application takes at least 10 images of checkerboard from a pair of cameras as well as the size of the checkerboard. It will automatically detect the cross points of checkerboard and calculate the camera parameters. Also, the parameters needed to correct radial distortion will also be given. Because the camera locations were now found in groups, the result was more accurate.

The application with an image from left camera and one from center camera was shown in the following:

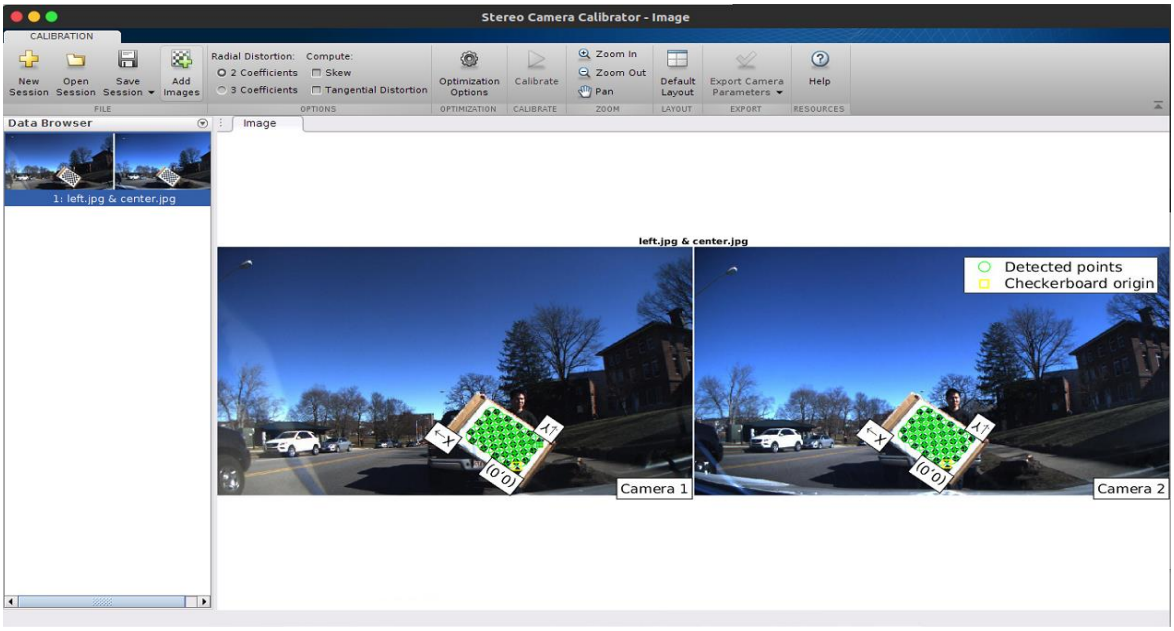


Figure 26 Calibration Example 1

The application with an image from center camera and one from right camera was shown in the following:

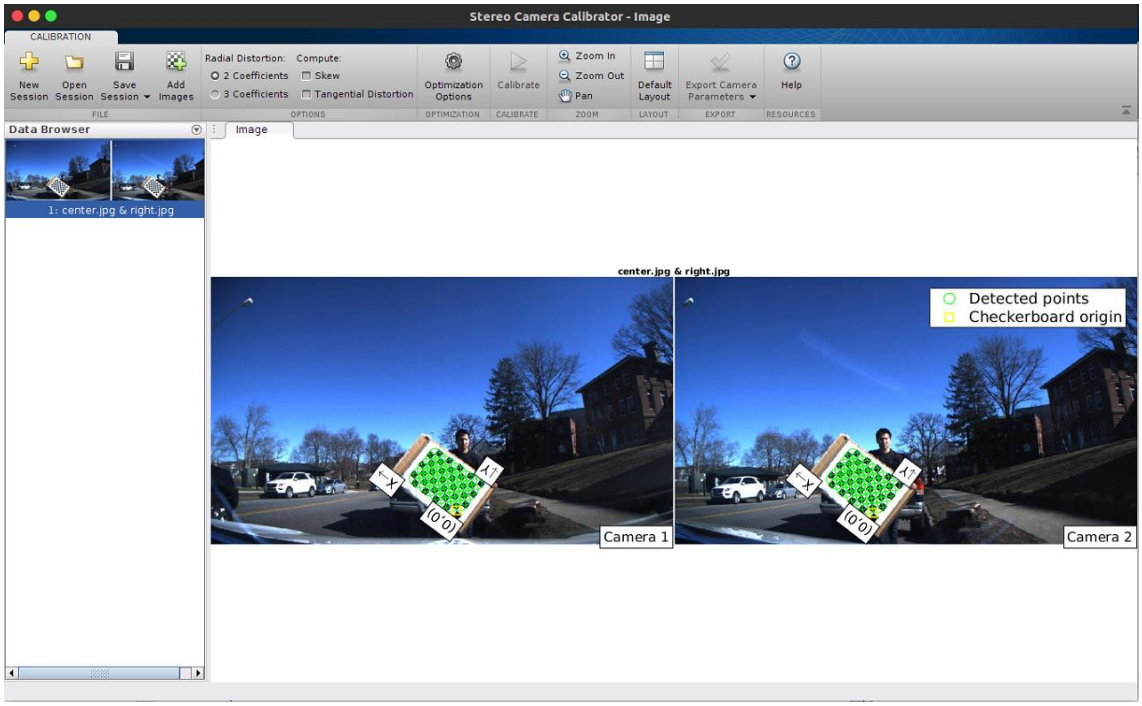


Figure 27 Calibration Example 2

Also, the translation matrices in world units and the rotation matrices of left camera relative to center camera and of right camera relative to center camera were found.

Then, only the extrinsic matrices for the center camera need to be found to determine the extrinsic matrices for all three cameras. Initially, we tried to put a checkerboard on the ground and use the application to calculate the extrinsic matrix, but as the figure shown in the following, the checkerboard was too small to be clearly detected.

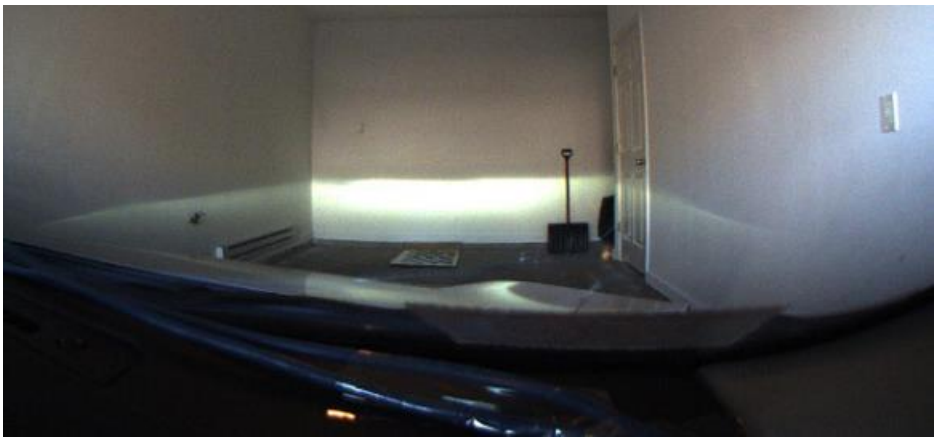


Figure 28 Calibration Example 3

We then decided to use the endpoints and cross points of lane markings on a parking lot as coordinates in world coordinate. The following image was finally used to calculate the extrinsic matrix of the center camera:

Figure 29 Extrinsic matrix center of camera

The coordinates of the endpoints and cross points were first calculated according to the distances measured. Then, radial distortion was corrected using the camera parameters calculated by the application. The resulting image was shown in the following:

Figure 30 Image correction

The following code was then used to calculate the extrinsic matrices for the cameras:

```
[rotationMatrix,translationVector] = extrinsics(img211(:,:),worldPoints(:,:),cameraParams{2});
cameraEx{2} = [rotationMatrix;translationVector];

cameraEx{1} = cameraEx{2}*r1to0;
cameraEx{1}(4,:) = cameraEx{1}(4,:) + t1to0;

cameraEx{3} = cameraEx{2}*r1to2;
cameraEx{3}(4,:) = cameraEx{3}(4,:) + t1to2;
```

Figure 31 Calculating Extrinsic Matrices

The function ‘extrinsics’ from computer vision toolbox in Matlab was used to get the rotation matrix and translation vector for the center camera. This function takes in the coordinates of the points on the image without lense distortion, the world points we calculated and the camera parameters we got from the application. Then, based on the relationship between left, center and right cameras, all three extrinsic matrices were calculated.

The camera positions in the world coordinate were then plotted to make sure that the extrinsic matrices were accurate. The resulting plot was shown in the following:

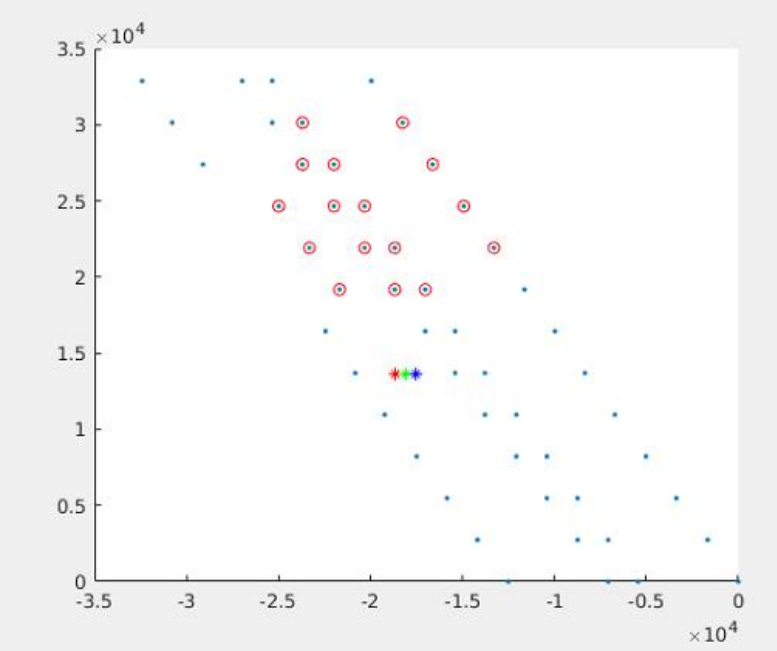


Figure 32 Calculated Camera Positions

In this figure, the red circles represent the points we used, the red, green and blue stars represent the left, center and right cameras. The locations of the cameras shown in the figure was very close to the actual locations of the cameras in the chosen image.

7. Project Results

The main objective of this project was to create a modular collective system for intelligent transportation that could be implemented and tested in a variety of different vehicles to collect data, create models based of the data, and then test the data in the

controlled environment of a simulator. This objective was broken down into three sub-objectives: create a data collection system, develop a Convolutional Neural Network model that could be trained to operate a vehicle, and develop a simulator to both test and train network models. In this project, all of these objectives were met, and thus the main goal was met too.

The data collection system works as expected- it captures images at a rate of 10 frames per second, collects the speed of the vehicle at a rate of 10 frames per second, and it collects the steering wheel angle of the vehicle at a rate of 30 frames per second. This data can then be used to construct a real-world driving simulator and can also be used as the training data for our neural network.

The Convolutional Neural Network model also works as expected, it is able to accurately and correctly produce the correct steering wheel angle for any given image or frame of a video that it is provided with. When provided with many images at a faster than expected rate, the model does struggle and often makes mistakes, however this is a problem that can be fixed with more training epochs as well as some refinement of the code.

The simulator portion of the project was begun, with the image calibration portion of the real world 3D simulator completed; however due to time constraints the rest of the real world 3D simulator was not able to be completed. However, the team was still able to test the CNN through the use of the Udacity simulator.

Overall, this project has produced a cohesive system of data collection, network model training, and network model testing that can be used to advance WPI's intelligent transportation program.

8. Conclusion & Future Work

In conclusion, the team considers this project to be a success. The team succeed in creating a data collection system and a Convolutional Neural Network (CNN) model for intelligent transportation. The simulator portion proved to be beyond the scope of this project, however substantial advances in the simulator were made in the form of the camera calibration- progress that can be built upon by future projects. The data collection system produced excellent results, logging the speed, steering wheel angle, and stitching three different camera angles together. The Convolutional Neural Network model is able to produce the correct steering wheel angle for any given image it is provided with. The simulator portion of the project was begun, with the image calibration portion of the real world 3D simulator completed; however, the scope of the real world 3D simulator proved to be too large and was not able to be completed due to time constraints the rest of the real world 3D simulator was not able to be completed. By developing these tools, the team was able to enhance and advance Worcester Polytechnic Institute's intelligent transportation program.

This project created a wide variety of possible future directions. A possible option being the completion of the simulator, a project that would build upon the image calibration implanted in this project and proceed to build a complete real world 3D simulator. Another obvious and compelling direction being the implementation of the trained CNN model not only on a simulator but also a real active vehicle. Furthermore, a miniature car model could also be used to collect and test data from, barring access to a fully sized active vehicle. This project was

limited largely by the time constraints presented in the 14 weeks available. Plans had and have been made for both a miniature and full sized vehicle to test aspects of our intelligent transportation project, however both of these will not arrive by the time this project is completed; future projects will be able to take advantages of these resources to implement some or all of the possible future directions listed above. This would result in more efficient and robust data collection, CNN models, and true to life tests.

Bibliography

Bojarski, Mariusz, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, Xin Zhang, Jake Zhao, and Karol Zieba. "End to End Learning for Self-Driving Cars."NVIDIA. NVIDIA Corporation, 25 Apr. 2016. Web. 20 Sept. 2016. <https://images.nvidia.com/content/tegra/automotive/images/2016/solutions/pdf/end-to-end-dl-using-px.pdf>.

"DeepTesla - Deep Learning for Self-Driving Cars." http://selfdrivingcars.mit.edu/deepteslajs/. Accessed 21 Mar. 2017.

Brown, A. (2017, February 22). Udacity/self-driving-car-sim. Retrieved March 22, 2017, from https://github.com/udacity/self-driving-car-sim

FLIR Integrated Imaging Solutions. (2017). FlyCapture SDK. Retrieved March 22, 2017, from https://www.ptgrey.com/flycapture-sdk

Appendix