Senior Design II Final Paper

Harrison Bounds, Brandon Donaldson, Lawan Hamidou

Project Overview:

Our design utilizes vision-based machine learning that was originally inspired by the open source API DonkeyCar. After experimenting with the DonkeyCar API, we decided to branch out and improve upon DonkeyCar's API by creating our own neural network. For the RC car to learn how to navigate on its own, two sets of data are required. (1) Images that are recorded using a webcam mounted to the RC car and (2) data sent from a wireless controller. The data sent from the controller (throttle and steering values) is assigned to each image gathered by the webcam mounted to the RC car. After completing several laps of a course (15-20 laps usually), the recorded data is used to optimize a Convolutional Neural Network (CNN). The purpose of the CNN is to identify patterns based on what the robot sees, and what controller command the robot receives. Once the CNN has been trained, it will predict the driving commands that need to be sent to the robot based on what is seen in the image the webcam records. After the CNN's accuracy level is at an acceptable level, an autopilot program is deployed to the robot, allowing it to navigate a course autonomously.

Hardware:

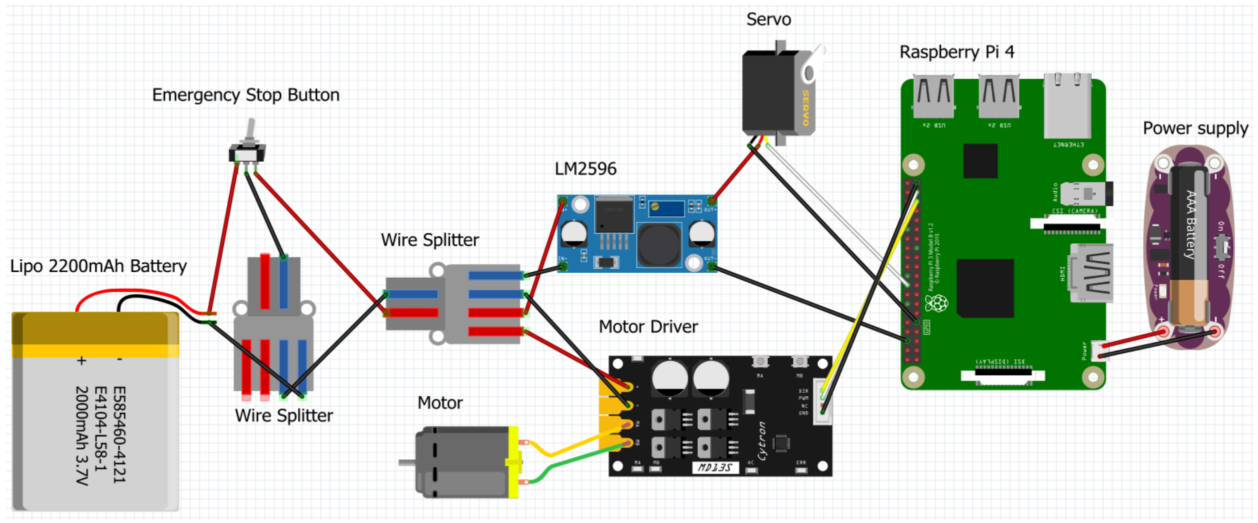
Our robot consisted of several pieces of hardware: A logitech webcam, Raspberry Pi 4, motor driver, RC car base (including a motor and steering servo), voltage regulator (regulated to 5.0v), powerbank to power the raspberry pi, a lithium battery (7.4v) to power the steering servo and motor, an emergency stop switch to interrupt the circuit between the lithium battery and motor, and a bluetooth controller/usb receiver.

Wiring Diagram:

When it comes to the way the different components of our hardware are wired up together, there are several parts that are most important, as in the Raspberry pi 4, acting as the brain, and the motor and servo. Those two are powered by a 7.4V Lithium-polymer battery through several connections. The battery first connects to an emergency stop switch that allows us to cut the power if needed without having to unplug any cables, and thanks to two wire splitters, we can conduct that power to the motor and the servo at the same time. After the positive end of the battery connects to one of the positive pins of the stop switch, the ground of it goes into one of the outputs for ground on the first splitter. The ground end of the switch connects to the input ground on the wire splitter. The second output for ground on that one goes to the input for ground on the second wire splitter, while the positive input gets the second positive pin on the emergency toggle switch plugged into it. This gives us the ability to power the servo and the motor since we now have two grounds and two positives on the output ends of the second wire splitter.

One thing to consider before that is that the voltage coming from the 7.4V battery is too high for the servo we're using. To avoid risks of malfunction, we are using an LM2596 voltage regulator set to 5V between the second splitter and the servo. The wiring for that is simple, we have a positive and negative input on the regulator, wired to one of the positives and one of the two negatives of the wire splitter respectively, and a + and – output connected to the servo. When it comes to the motor, we are using an MD13S motor driver to control it, which connects to the remaining positive and negative outputs of the wire splitter, and then branches into a connection to the motor.

Our Raspberry pi 4 is separately powered by a USB C connected power bank and has 6 wires connected to it. We have the first three joined to the motor controller in the following manner: the DIR pin connected to GPIO19, the PWM connection to the GPIO26 pin, and the ground on the motor controller to a ground on the Raspberry pi. The other three cables on the raspberry pi are connected as follows: the Pulse pin on the servo going to GPIO24, the ground on the voltage regulator to a ground on the Pi, and the ground output on the voltage regulator connected to a ground on the Raspberry pi. The final components are our webcam and controller which we connect to the Raspberry pi 4 through USB. Through this setup, we were able to build and wire our car in an efficient manner with everything we needed for the next steps.



Collecting Data:

Our data collection program is relatively simple (reference our collect_data.py program). The program starts by importing the necessary libraries: pygame (for the controller), opencv (for the camera), gpiozero (motor and servo controls via GPIO pins on the raspberry pi), and csv for recording throttle and steering values sent from the controller.

Once the libraries are imported, variables for the throttle and steering are initialized. The motor and servo are then assigned GPIO pins to receive signals from. The motor requires two pins, one for phase (GPIO19), and enabling the motor (GPIO26). In the same block of code that the GPIO pins are initialized for the servo and motor, steering trim is adjusted in the software with the *center* and *offset* variables (these values are typically unique to each servo motor and have to be tested manually before a final value is determined).

Continuing on to the *main* function, this will run through a loop that follows as: increase the increment of images from the webcam → reading the throttle and joystick position → checking throttle and joystick position to determine if the robot needs to move forward or backward & turn left or right → signal is sent to motor & steering servo for movement action → checking if webcam is recording → if it is recording, save the image as a 120x160 image, label it as a .jpg, open a file and record the steering and throttle value for the current frame count → check to see if the program has been interrupted manually → REPEAT. All of this data will be saved to the raspberry pi in the /data/ folder to be used for training.

Training the Data:

Once the data has been gathered (after collecting ~25k images), the data can now be trained. The data can be wirelessly transferred from the raspberry pi to a host machine using the "rsync" command in the command line for linux [`rsync -rv --progress --partial pi@<your_pi_ip_address>:~/mycar/data/  ~/mycar/data/`]. Once the data has been moved to the host machine, the train.py program can be started. Our training program includes some data augmentation functions that will randomly flip the image horizontally, rotate by up to 30 degrees, and inject noise to artificially create extra data that can be used to create a more robust model. This training program will call our neural network architecture that will pass the data through several convolutional layers and fully connected layers (the amount of layers depends on which architecture is chosen - either hblNet or DonkeyNet - more on this in the **Comparing Models** section below), the result after the data passes through the architecture are two values: predictions for steering and throttle values based on the image seen by the camera. The data will continue to be trained until the desired number of epochs has been achieved (can be edited in the code). We used 15 epochs for our training set. We came to the conclusion that 15 epochs was most efficient for our data training through trial and error to reach our desired loss values (~0.002 for our final loss values tended to show decent results on track). After the program finishes training a model, a file is created that shows a graph that compares the training and test losses to each other, and a .pth file that contains the autonomous model that can be loaded from the raspberry pi to navigate the course that the RC car was trained on. Once these files are created, the trained model (<filename.pth>) can be transferred back to the raspberry pi using the “rsync” command again.

Comparing Models:

Throughout the semester, our Team tested many different models to see the output of tuning hyper parameters, different layer sizes, and number of parameters. At first, the team started with the original model, DonkeyNet. This Convolutional Neural Network Architecture (see Figure 1.) consisted of five convolutional layers and three fully connected layers. Out of all the architectures tested, this was the most reliable. For this reason, we used this as a baseline to compare our other models to.
Next, we tried creating three other architectures, which used the same hyperparameters when training. The point of creating these architectures was to analyze how different layer sizes affect the accuracy of the network. The names of each of these architectures were simpleNet, moderateNet, and megaNet. simpleNet had only three convolutional layers and two fully connected layers. moderateNet was extremely similar to DonkeyNet, where the only difference was within the sizes of the convolutional layers. megaNet on the other hand, had eight convolutional layers and five fully connected layers. Comparing each of these to DonkeyNet, they all had major flaws. simpleNet was not reducing the image features enough, and with a smaller fully connected network, it was having difficulty decreasing the testing loss. moderateNet had a separate issue that was caused by the sizes of the convolutional layers. The last layers ended up being as big as 256, which is too many for a project of this stature. Similarly, megaNet was way too large and slow, by having the most amount of parameters. After each of the failures in these architectures, we decided to go back to the drawing board and analyze what makes DonkeyNet work so reliably. This is when hblNet (see Figure 2.) was created. This architecture consists of four convolutional layers, which reduces the image size down to 13x18 rather than DonkeyNet’s 8x13. The size of each layer was slightly larger than DonkeyNet’s, although it still ended at a size of 64 for the fourth layer. The fully connected layers were also totaled to three, but were cut in half in size from DonkeyNet. The purpose of this was to reduce the number of parameters in the network, making for a faster model (extremely useful for a real time autopilot). Within the architecture, we also tried tuning the hyperparameters of the model, with the best results using 20 epochs and a learning rate of 0.0001. (see Figure 3.) We also tried changing the loss function to cross entropy (from mse) and the optimizer to SGD (from Adam), but these could not compare to their original counterparts. In the end, DonkeyNet and hblNet both had successful test runs. For future work, I would take hblNet and experiment with larger sizes for the fully connected layers as well as a larger feature map selection to give the network ‘more to look at’. Using this method while attempting to keep the number of parameters at a reasonable amount, I believe would create a more robust and reliable model.

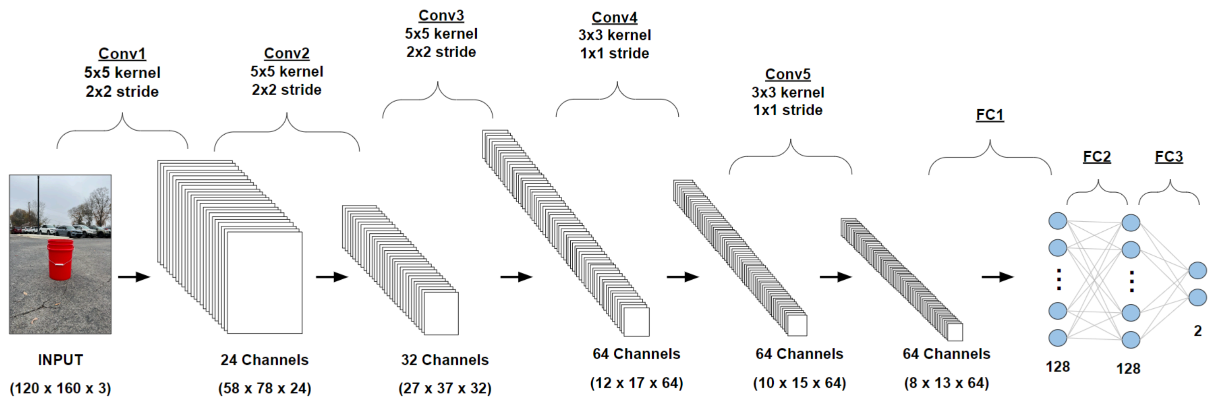


Figure 1.

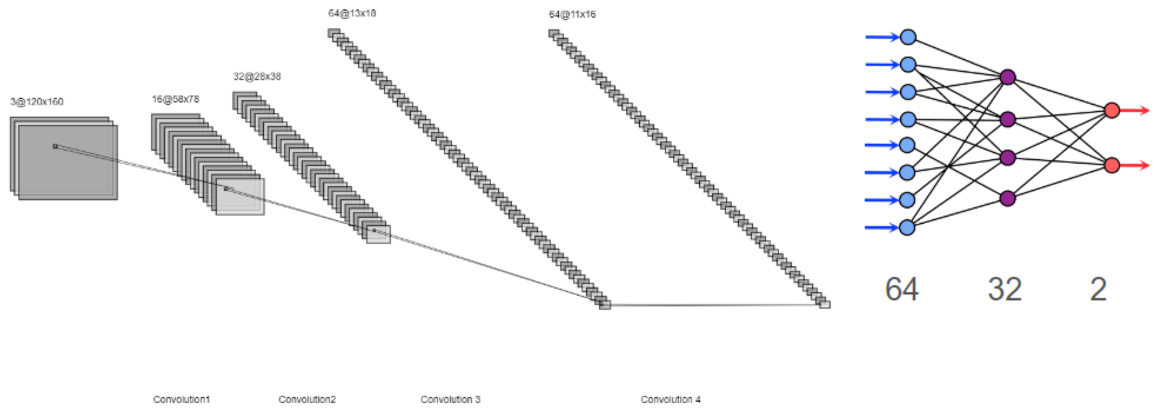


Figure 2.

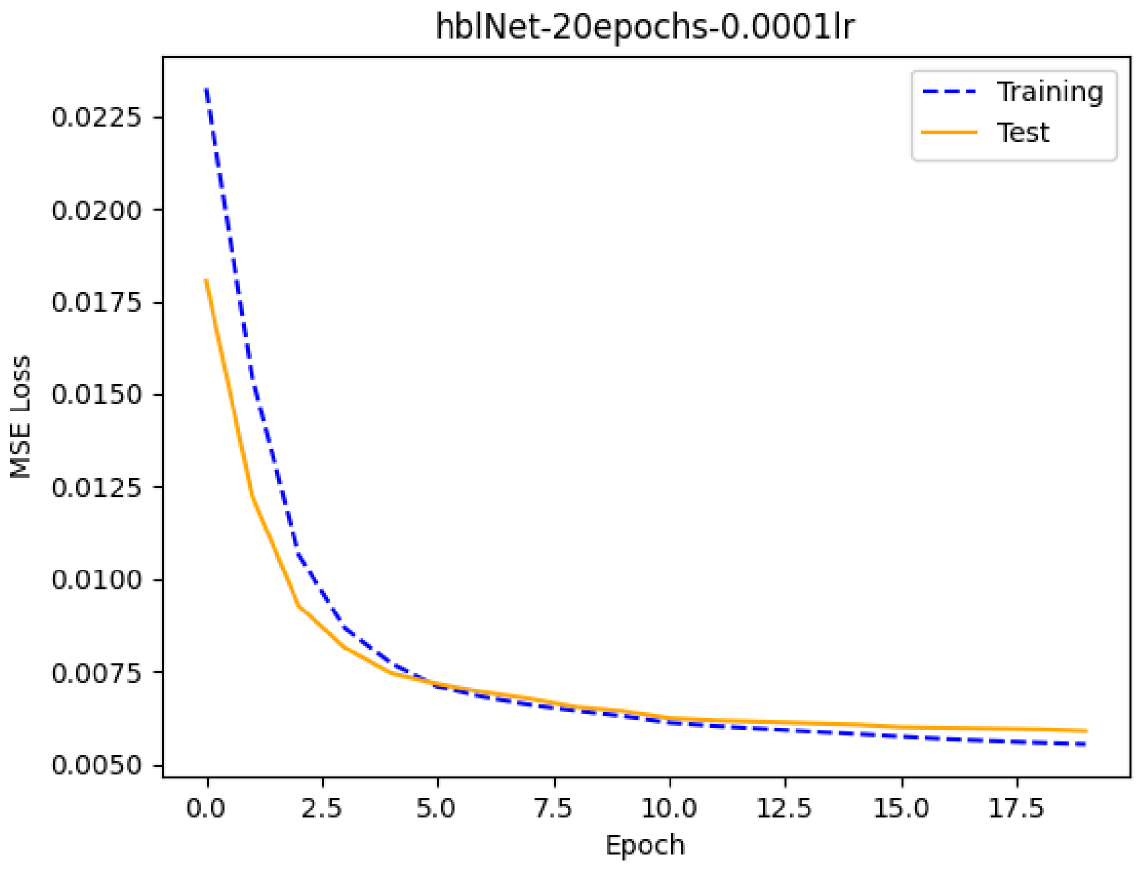


Figure 3.

Conclusion:

After assembling the hardware, collecting data, training a model, we can finally deploy an autopilot. As mentioned in the previous section, the two most successful models during this project were the original DonkeyNet, and our own hblNet. One issue that our team encountered while attempting to train successful models, was the brightest of the training area. The sunlight has a great impact on the performance of the model, due to the brightness varying at different times. Another minor issue we encountered was the time it took for the whole process of collecting data (which was roughly 25 minutes). For future work, implementing a noise injection algorithm to force the model to be exposed to varying pixel brightnesses could potentially mitigate the issue with sunlight. Additionally, data augmentation would help solve the problem with prolonged time to collect data by making the training set more robust (allowing for less original data). Finally, experimenting with new architectures and CNN techniques like dropout layers and batch normalization could lead to a lower training and testing loss in the network.