Micah Dotzert

Student ID: 3398506

April 7, 2020

Source Code: https://github.com/md-hexdrive/Self-Driving-AI-Car-Project

Video (more may come later): https://youtu.be/rAWz5kUqRnQ

Self-Driving AI RC Car

Final Project for COMP 444

Summary:

For my final project in COMP 444, I decided to build a Self-Driving RC Car with AI. This car can drive autonomously along a track and automatically brake. It has two forward facing sensors: an ultrasonic distance sensor and a camera.

The car is controlled by a Raspberry Pi 3B. The car itself is a basic off-the-shelf RC Lamborghini. The programming for this project is in Python with the Google Colab based AI training occurring in a Jupyter Notebook. A combination of a Neural Network and a reactive control system drive the car autonomously. This project made use of many Python libraries, including GPIOZERO (for interacting

with the Pi's general-purpose, input-output [GPIO] pins), NumPy, OpenCV (for camera image processing), and TensorFlow/Keras (for the AI).

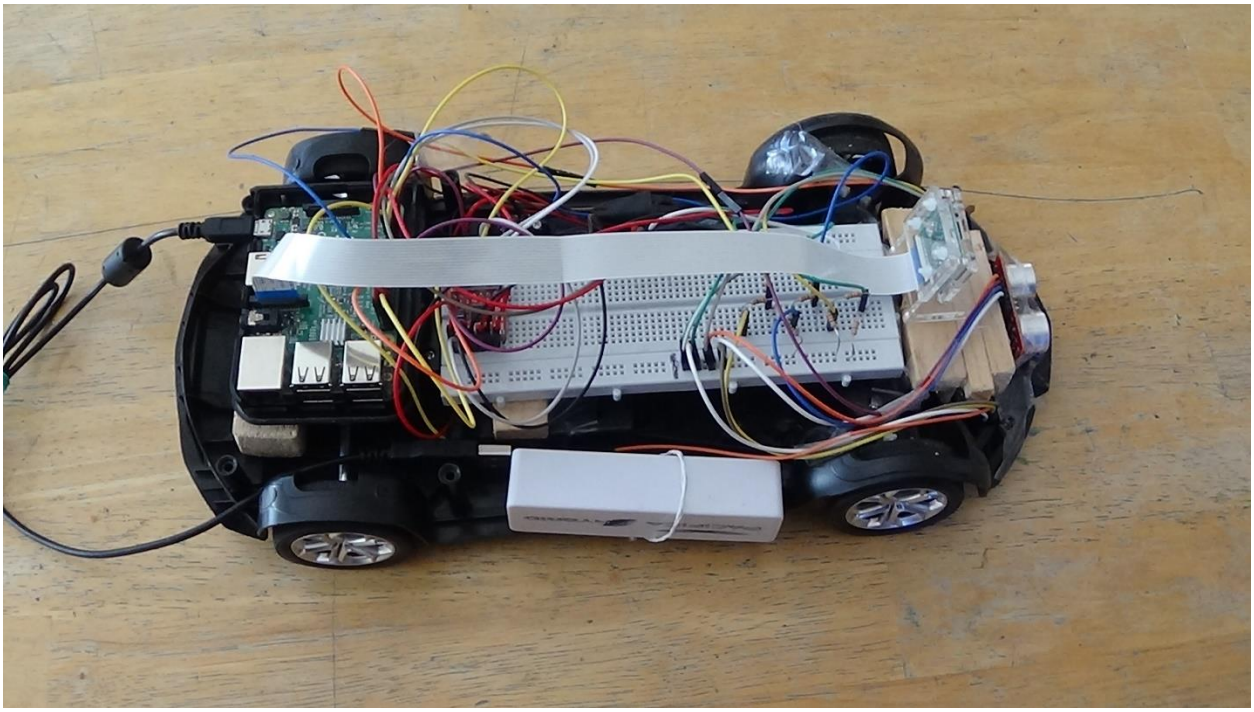

Figure 1 The Car. From left to right, the Raspberry Pi, the motor controller (not very visible), the breadboard, the Camera and stand, and the ultrasonic distance sensor, front. At the bottom you can see the USB battery that powers the Pi (the car motors use a different battery).

The car is "taught" how to drive around a track by a human driver who manually drives it around the track beforehand using **record_driving.py**. The car records video, pictures, and driving commands while under manual control. The pictures have the current driving command included in their file name. The pictures and the respective driving command are used to train a neural network. The training could theoretically be done on the Raspberry Pi itself, but backpropagation is resource intensive. (I tried training the AI on the Pi, but Python kept crashing after the 5th

epoch.) As a result, training of the neural network model was undertaken in Google Colab, using **train_ai_in_colab.ipynb**. The collected training data is randomly modified in Colab to generate a greater amount of training data for the network.

Once the trained neural network is saved to a file, that file can be downloaded onto the Raspberry Pi. The Python program **drive_ai_drive.py** drives the car autonomously with the aid of the network.

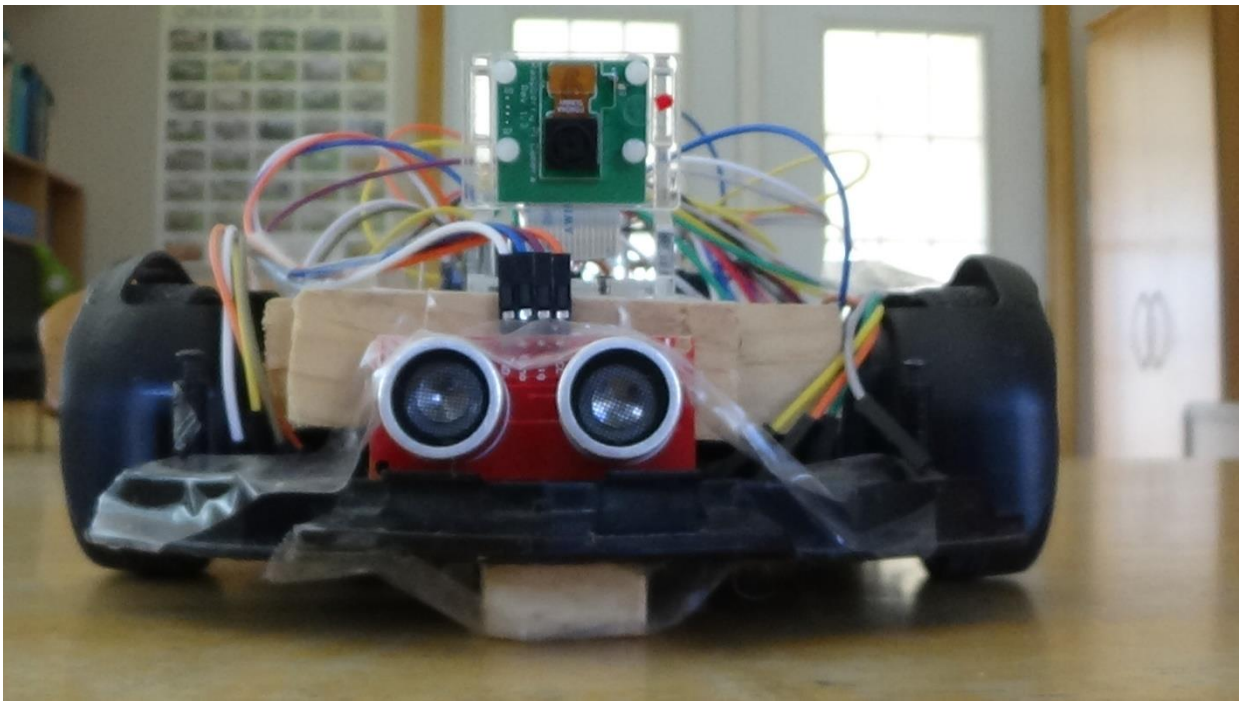

Figure 2, Front of Car, Camera and Ultrasonic Distance Sensor Shown with mounting hardware, the small wooden front suspension support is located underneath the car.

The control scheme when driving under AI control is as follows. The Raspberry Pi captures an image from its camera. That image is passed to the neural network which processes it to determine the most likely output. The user can stop the car at any time with the **space bar.** The ultrasonic distance sensor will also stop the car if it detects an obstacle less the 45cm away. One of the neural network's possible outputs is stop, which stops the car dead in its tracks. If none of the stop conditions are satisfied, one of the network's possible output commands are executed: drive forward and turn left, drive forward and go straight, and drive forward and turn right.

There is no explicit programming that tells this car how to drive around a track. Instead, a user teaches it how to navigate the track, and the car then learns how to drive from them (this same technique is used to train real self-driving cars, i.e., Tesla).

driving.py is where the code for driving the car itself is held along with code that determines the current driving status. **distance_monitoring.py** handles the ultrasonic distance sensor code. **interpret_frame.py** renders the car's current driving status on the video feed. **test_autopilot.py** tests the AI on a previous video recording.

Credits:

For this project, I collaborated with my brother to build the car, and we discussed how to solve several problems together. My dad and him also made some wooden supports for the car to help secure and level hardware. I am very grateful for their help.

I was aided in coding the car by consulting a number of online sources, particularly the Raspberry Pi, TensorFlow, and OpenCV documentation/tutorials; the DeepPiCar tutorials and code; and this tutorial, which showed me how to interface the Raspberry Pi with the TB6612FNG motor controller. The code for training the AI is largely based on end_to_end_lane_navigation.ipynb from the DeepPiCar tutorials and some of the DeepPiCar programming helped me out in other places in my project, but I wrote most of the code myself.

This is a cool project and I would like to thank Richard Huntrods and Athabasca University for the opportunity and the know-how to create it!

I created a discussion post/help request when the car wasn't turning properly, and Charles Sedgewick answered it and confirmed our suspicions of what we thought were the main causes of it. So, Thank You Charles for your help!

Parts:

- An RC Lamborghini car (with significant modifications), but any RC car could, in theory work, if it is big enough. Also, needed are the car's original battery (or a viable replacement) and the charger for that battery.
- Raspberry Pi Model 3B
- Raspberry Pi unofficial Camera Module (with 30cm connector and acrylic case/stand)
- 6 ultrasonic distance sensors (including the one included in the SIK) - two supports came with the other 5 sensors and were used to support two of the sensors, then all the sensors, except for one, were removed.
- The TB6612FNG motor controller (from the SIK)
- A breadboard (twice the length of the SIK's)
- Male-to-male, male-to-female, and female-to-female jumper wires.
- A couple of wooden supports to help level and secure the PI and Breadboard inside the car (my dad and brother made those, so I am Thankful for their help).
- A case for the Pi, which has slots in the bottom that can slide over screw heads, which is how it is secured to its support inside the car (there is a screw drilled into one of the wooden supports).

- 330Ω resistors, to limit the return voltage from the ultrasonic sensor to the Raspberry Pi (because the Pi's pins shouldn't directly receive the 5V returned from those sensors)
- Electrical and Scotch ™ tape

Helpful Resources:

Here are some resources that I found helpful over the course of the project:

- The very first place I saw this type of project done (a few years ago!).
- Another example of this project, originally found here.
- The DeepPiCar series of tutorials were very instructional and I used the training code in part-5 to train my AI in Google Colab.
- Google Colab itself is very useful and where I trained the AI.
- https://techwithsach.com/build-a-self-driving-rc-car-using-raspberry-pi-and-machine-learning-using-google-colab/
- https://www.instructables.com/id/Raspberry-Pi-Remote-Controlled-Car-1/
- SparkFun documentation, including the TB6612FNG Hookup Guide.
- A project where the Raspberry Pi was used with the TB6612FNG motor controller https://www.bluetin.io/dc-motors/motor-driver-raspberry-pi-tb6612fng/.

- The Raspberry Pi Foundation's tutorials, including their *Camera Module Essentials* and *GPIOZERO Essentials* books produced by the MagPi, available for free download as PDF files. Also, the Physical Computing with Python tutorials on their website were useful in learning how to use the gpiozero python library.
- The Raspberry Pi GPIO pinout guide https://pinout.xyz/ (this is taped to the wall beside my computer for reference - the Raspberry Pi's pins are not labeled like the RedBoard's pins, so this is essential).
- The OpenCV and TensorFlow official websites and tutorials.
- More than a few Stack Exchange articles, and Google searches were helpful in finding solutions to problems.
    - https://raspberrypi.stackexchange.com/questions/17017/how-do-i-run-a-command-line-command-in-a-python-script (this page and the one below helped me get drive the car properly
    - https://raspberrypi.stackexchange.com/questions/99913/permanently-set-keyboard-repeat-rate-on-raspberry-pi
    - https://stackoverflow.com/questions/13207678/whats-the-simplest-way-of-detecting-keyboard-input-in-python-from-the-terminal helped with the original creation of the keyboard control program for the car (scraped it for an OpenCV variant).

- The Python documentation / tutorials were very helpful in learning Python and about its standard library.
- The free online interactive tutorials created by the University of Waterloo where I first learned Python: https://cscircles.cemc.uwaterloo.ca/.
- Here is a Udemy course that teaches you about making AI (one of the AIs you build runs a virtual self-driving car): https://www.udemy.com/course/artificial-intelligence-az/.

Hacking the Car

We started working on the project the day after I requested instructor approval for this project.

To start, the car was disassembled to reveal its internals, make room for the electronic hardware, and prepare to interface the hardware with the Raspberry Pi. The car's fake interior and windshield were completely removed (everything being held together with star screws).

The car's electronics were relatively simple. It had an RF antenna, two headlights, a 6V battery, an on/off switch, the main control board, two DC motors, and the wires to connect everything together. One motor is controlling forward and reverse motion and the other moves the front wheels left and right. I had expected the front

motor to be a servo motor, but I was surprised to find that it was a DC motor instead. It needs to be controlled through a motor controller, otherwise it would only be able to turn in one direction.

After a bit of deliberation, the car's control board was deemed irrelevant and was completely removed. (If we left it, in how would we control the car? We would likely have had to interface the Pi with the antenna, which would require knowing the signals used to tell the car what to do.) Instead, the SIK's motor controller will run the motors. The antenna and front lights were also removed (at least for now).

The car's remaining components (battery and motors) needed to become breadboard-compatible and attaching them to jumper wires seemed to be the solution. At first soldering the wires seemed to be appropriate, but then we decided to try connecting them first in a less-permanent (and involved) way: stripping the ends of the car's wires (with electrical wire stripers) and inserting the exposed ends into the female ends of male-to-female jumper wires. The stripped wires were not going to stay inserted on their own, so their exposed ends were folded in half and twisted to become big enough to stay. The car-breadboard wire connections were then wrapped in electrical tape to stay together.

A breadboard (twice the length of the SIK's) was then installed inside the car body with the motor controller and the car's battery and motors attached to it. Later, we taped a small wood support underneath one end to level it.

Getting the Pi ready – Part 1

The first steps in getting ready to use my Raspberry Pi (model 3B) ready for this project are:

1. Setting up a VNC connection between the PI and my Windows 10 desktop (so that I can work on both at once).

2. Learning how to use the Raspberry Pi's GPIO pins to interact with physical circuits. Before this Christmas, I didn't have any male-to-female jumper wires; therefore, I couldn't use the PI for physical computing before now.

For this learning experience, I am following some of the Raspberry Pi Foundation's official tutorials. Recreating the LED blink example was easy enough, but when I tried to do a traffic light simulation with LED's, two of the lights wouldn't work. The lights and resistors both work on their own, ~~but apparently, some of the new jumper wires are defective.~~ EDIT – The jumper wires were not the problem here. It looks the breadboard's power strips, running the

length of the board are discontinuous in the middle; therefore, the ground connection doesn't run the length of the board and two of the LEDs were isolated from the ground connection.

Learning How to Interact with the Motor Controller

I found a website where the author used the SIK's motor controller with a Raspberry Pi and I will base my car's driving control code on code hosted on it: https://www.bluetin.io/dc-motors/motor-driver-raspberry-pi-tb6612fng/.

Both motors are controlled with the motor controller. Motor control channel A drives the car forwards and backwards, while channel B turns the car left and right.

I am not sure how to control the car motors' speed from the GPIO zero library. The car moves extremely fast. One and a half seconds of the drive motor running continuously on high would probably take the car the entire way across the room I am testing it in. (**EDIT:** 2020-03-25 The motor speed can be controlled by setting the PWMOutputDevice's value member to the appropriate value in the range from 0 to 1. Right now, I have the driving motor speed set to .5, so that the car moves slow enough to be safe [and not crash into anything at high speeds, which is what it used to do].)

The gpiozero library has a built-in motor control class, but that class uses two pins to control each motor, while the motor controller uses three connections per motor. Therefore, PWMOutputDevice and DigitalOutputDevice objects are used to control motor outputs.

Remote Controlling the Car over WIFI

With VNC enabled on the PI, remote manual control of the car over WIFI is possible. When the Camera Module is installed, we should be able to both stream live video from the car while it is driving and control it from another room.

I was able to create a program that can detect when a key is pressed and tells the car to move forward, backwards, left, straight ahead, and/or right. The keyboard arrow keys can be read from a Python Pygame program running on the Raspberry Pi and whichever key is pressed determines what the car does.

The programming for this was based on stack overflow post: https://stackoverflow.com/questions/13207678/whats-the-simplest-way-of-detecting-keyboard-input-in-python-from-the-terminal and code from the PiBuster example program from the *Raspberry Pi For Dummies* book by Sean McManus and Mike Cook.

Power Supply Issues

The car itself has two power sources: the car's original battery pack, which runs the motors and a rechargeable USB battery pack which runs the Raspberry Pi and everything else through it. When stationary, the PI is powered by a wall plug cellphone charger.

When we first got the PI, we used a cellphone charger that had a very thin wire and didn't provide enough voltage for the PI, which resulted in the lightning symbol constantly appearing in the top-right corner. It was a while before we moved to a more secure power supply, but now the wall power source produces adequate voltage.

These newer USB battery packs have been bringing back the old lightning bolt, which could be a problem as the CPU automatically runs at half-speed when it is shown, and the lightning bolt indicates a risk of corrupting the SD card's contents. Based what I have been reading online, the under-voltage reading is more likely the micro-USB cable's fault rather than the battery pack itself. Moving to a better cable certainly reduces the frequency of the low voltage warnings, but doesn't eliminate them, especially if the cable is messed with too much.

Ultrasonic Sensor:

In addition to the future camera that will be used on the PI, we are also going to use ultrasonic sensors to detect obstacles. The plan is to use up to six sensors, three pointing forwards and three backwards.

Until they arrive, we are testing with the SIK's own ultrasonic sensor for collision detection. While getting ready to use it with the car, we got to see just how accurate it can be at determining distances.

During testing with this sensor, we also became quite familiar with the concept of voltage dividers. It turns out that it is not a very good idea to return 5V, which the sensor uses, directly to the Pi's GPIO pins, or you run the risk of permanently damaging that pin (which can only take 3.3v (3V3)).

When first using the distance sensor with the PI, all the examples of using distance sensors that we saw, used voltage dividers between the sensor's ground and echo pins and the pin on the PI where the signal is received. Since we didn't have the same resistors as in the examples, we used a 330Ω and a 10KΩ resistor in the divider, because we were ignorant of why the divider was there (to protect the Pi from higher voltages than what it could handle) and I am not entirely sure that I knew it was supposed to be a voltage divider. That setup was giving the Pi

(10000Ω / (330Ω + 10000Ω) * 5V) = ~4.84 volts – well above what it should receive.

Our Pi was unharmed by the experience (we are still using the same pin to receive sonic echoes), and the only indication that we could have had a problem was given when I was reading more about using the sensor the next day. Now we are using a 330Ω resistor with a 510Ω resistor (from another kit) for the voltage divider for that sonic sensor. This setup gives the Pi 3.03V ((510Ω / (330 Ω + 510 Ω) × 5V) = 3.03V).

Of course, we only have one 510 Ω resistor; therefore, we needed to figure out how to get the remaining sensors working safely. One approach would be to buy more 510 Ω resistors and use them in conjunction with the 330 Ω resistors we already have. Another approach would be to buy logic level converters to convert the 5V signals to 3V3. The approach we are using, for the remaining sensors, is to create voltage dividers with a pair of 330Ω resistors, which will convert the 5V returned from echo to 2.5V signals for safe use with the Pi.

Additional Distance Sensors:

Later, we began adding more distance sensors, the front ones at first, to the car. The gpiozero library allows all three sensors to run at once but testing their effectiveness while driving the car's systems has so far been sketchy. I think the

sensors need to run in a different thread from the thread that runs the motors. Also, giving the motors commands seems to take a bit of time for a response to occur, maybe I am making some mistakes with they run, or maybe there really is a time lag?

The left and right sensors at the front are attached with the aid of screws and a pair of mounts included with the new sensors.

The newer sensors are slightly bigger and appear more professional quality than the SIK sensor, which is mounted pointing straight ahead.

Edit:

We ended up removing all but the front middle ultrasonic distance sensor to help reduce the weight on the front of the car.

The Raspberry Pi Camera Module

We bought an unofficial camera module off Amazon from here. One of the biggest surprises was that it can see light from my tv remote. TV remotes like mine typically use IR light to transmit commands to TVs. This remote's light can't be seen with the naked eye, but the camera module can still see it. Camera modules that can see Infra-Red light do exist, but this wasn't supposed to be one. When placed in the dark, light from the remote control isn't enough to light my face up

from the camera's perspective, so the camera probably isn't capable of night vision.

This camera module came with a clear acrylic case/stand that looks nice. It also comes with two connector cable lengths. We choose to use the longer cable.

The acrylic case doesn't have instructions included; therefore, you must put it together based on the pictures given on the amazon page. You need to either be handy and/or have an extra pair of hands to put it together. Initially we constructed the case with the camera slopped upwards, but we need it slopped downwards for it to see the ground and read ground markings. We were going to use a wedge to prop it up on an angle that would allow it to see the ground better, but then we realized we could take the stand apart and flip the camera so that it slopes downwards.

Video quality from the camera is remarkably good but trying to run the camera at higher framerates results in junky video feed. I guess we won't be using it for slow-motion videos, but we won't need that for this project.

At first the video feed from the camera had a noticeable delay, but then we realized that the camera feed was being sent from the pi itself over the network with VNC to my Windows 10 desktop. No wonder there was some lag 😊. Switching my

monitor's input to the Raspberry Pi allowed us to see the video in real time. The same kind of lag can occur with ultrasonic sensor readings.

Getting the Pi ready – Part 2

Here comes the "fun" part: why it is always a good idea to back up your computer's stuff (especially on the Raspberry Pi). To get this project completed, I must install some prerequisite software on the Raspberry Pi, i.e., OpenCV, TensorFlow, and their supporting software. During my attempt to do this, the Raspberry Pi suddenly began expressing some bugs, first the dedicated Python IDEs preinstalled on the Pi stopped working, then suddenly the title bars for all the applications disappeared.

The problem was clear. For the umpteenth time since getting it, my Raspberry Pi's operating system, stored on a relatively volatile Micro SD card's flash memory, was corrupted (probably due to multiple low voltage events and sudden power loss, from forced shutdowns). Instead of waiting for the operating system to completely crash, I decided to backup what I could, and overwrite the OS. Since the OS was installed from NOOBS, the first thing I tried was to reinstall the OS from the NOOBS installation on the card. That failed part-way through, indicating that NOOBS was also corrupted. The only solution was to reformat and repartition the

SD card in Windows and download a new version of NOOBS with the latest Raspbian distro. The download itself took a couple hours, after which the files were copied to the OS and the regular installation process was carried out.

This all worked out fine, because I didn't have the latest Raspbian distro installed on the Pi and kind of wanted an upgrade. The new installation looks nicer.

After all this, I installed OpenCV and its dependencies. TensorFlow and Keras were a different story. Importing TensorFlow raises some exception, and there might have been an issue with installing it or a dependency. To install Keras requires scipy to be built on the Pi, and there is some problem that occurs while that happens. In the end, I said, "forget it for now with TensorFlow and Keras" and started working with OpenCV for the time being. I will install the other two later.

OpenCV:

I started practicing with a handful of OpenCV tutorials on my desktop computer, before I used OpenCV on the Raspberry Pi with its camera module. I am following the tutorial on https://towardsdatascience.com/deeppicar-part-4-lane-following-via-opencv-737dd9e47c96 to setup basic lane following with the car. This step in that tutorial doesn't use AI for driving, but it instead hard codes the driving data processing with OpenCV.

At present, I haven't gotten to the driving part of the programming, but the lane detecting portion isn't working flawlessly yet. Running it over real-time video from the Pi Camera produces jittery lines. One reason could be that I'm using blue construction paper while he used painters' tape, but the colors are similar enough that it shouldn't be a problem. I think my camera is at a different enough angle that masking half the screen isn't covering enough noise to get proper line readings.

I am going to switch strategy and get the AI to figure out the correct turning angle on its own.

Line Following System:

The current plan is to drive the car remotely along a track while it is recording the steering angle and video of the drive, then use the recorded data to teach a neural network how to drive the car around the track on its own. This programming will be based somewhat on code from the DeepPiCar series of tutorials.

TensorFlow and Keras are finally running on the Pi. https://ai-pool.com/d/how-to-install-keras-on-raspberry-pi- was extremely helpful in getting it going.

Manually Driving the Car:

It took a bit of deliberation to figure out how to control the car while showing it how to drive. My keyboard control program only turns the front wheels to three

positions, full-left, straight, and full-right. Therefore, since the car in the DeepPiCar tutorial can move its wheels to specific angles, I figured my car wheels should also be controlled so that they can be set to any specific position. I figured I would need to use a different form of control to show the car how to drive.

My first thought would have been to use the car's original remote controller to control the car. That wouldn't have worked. The controller and the car were originally designed to move the motors at two speeds: full-power or off, moving at slower speeds or smaller angles would not work. Also, I am not sure how the car would receive signals from the remote.

My younger brother has built his own RC Airplane. He has three extra receivers for his remote. Online, there are examples of using RC receivers with Arduinos and even one instance with the Raspberry Pi. So, I did consider using it. However, the Raspberry Pi doesn't use a RTOS, and the signals sent from the receiver are too fast to reliably receive. My brother also did not like the idea of using the controller (and he is right - as usual). The one example of someone using an RC receiver online originally used servo output signals to control the Pi but ended up using serial communication between the Pi and the receiver (our receivers don't have that).

I also considered using an XBOX 360 controller for this purpose but dropped it soon afterwards.

After we discussed the options, we agreed that just using the computer keyboard to control the car was the best choice possible. This will require some changes to the neural network from what was used in the DeepPiCar tutorials, which is probably a good idea anyways (I don't want to copy everything exactly).

Training Data Storage:

Training data is recorded as images with the current driving command included in the filename. Since there are four driving conditions: drive-forward-and-turn-left is stored as 0, drive-straight-ahead is 1, drive-forward-and-turn-right is 2, and stop is 3. The driving command is stored at the end of the filename, right before the filetype extension. The rest of the filename includes a unique date-time stamp created at the point in time when this batch of training data began creation and the frame number of this image.

Recorded Data:

There were several adjustments made to the recording procedures throughout the project, and there are a few quirks to the training itself.

At first, training instances were distinguished by the Pi's system time in milliseconds when the program started. Later, I noticed a better way to identify

recordings that was indicated in the DeepPiCar's training code, using the **datetime** library. This also allowed a more concise way to tell recordings apart by inserting underscores into the middle of the date, so that instead of reading a long string of unnecessary numbers, you can just identify the ones related to hour/minute/second.

Figure 3: image_642_2002_28_175053_1.jpg

Here is an example of a training picture and its name: image_642_2002_28_175053_1.jpg. The first number is which frame number of the video this picture is, in this case, #642. This next numbers are the formatted date as "%y%m_%d_%H%M%S". The final number is the current driving command, in this case, #1 = go straight.

The Raspberry Pi only records training pictures when it is moving forward or when the space key is being pressed. Otherwise, a video is still being recording (and the frame number is being incremented with every video frame). Therefore, significant differences between one frame and the next indicate that the car was stopped between those frames. Stopping the car right before or during a turn is a bad recipe

for this car to miss making the turn, so this can help determine the point in the data where the driving went wrong and that you should not train the model with data after this point. Later, when I added the ability for the car's driving to be visualized, the activity visualization was added to the training data when I had not intentionally intended for it to be included. I tried at first to prevent the visualization from being added to the training data. Afterwards, I figured out that it was beneficial, because the activity visualization displayed on a frame can be an easier indication of where driving mistakes occurred.

As shown the image above, the driving command recorded in the filename doesn't agree with the what is visualized at the bottom of the image (drive straight ahead vs. stop). If I see something like this in the middle of a training set, I obviously stopped around there, maybe not at that specific frame but soon before or after. The fact that the visualization doesn't line up with the filename description is a quirk, but not a deal breaker – I crop the visualization out of the picture before it gets to the network.

Restarting

At first, all the data from a recording session were placed in a common directory, but that made it hard to sort out good and bad data, and if you wanted to make a new recording, you had to restart the program manually. The solution was to allow

the user restart recording in a new directory from within the program. Mistakes made in recording data are now likely located towards the end of that data because I would start a new recording after the mistake.

Later, I realized it would be nice to have all the recordings created during a session to be grouped together, not intermixed with all other folders of recordings where you had to sort through and figure it out manually. The solution is to have a master directory for each session and store the restarted training data in subdirectories for each batch of data.

Controlling the car:

One way to drive the car is from a Pygame window, another is using OpenCV's waitkey() method. Although I already have a Pygame based control program, using OpenCV to control it seems more relevant. Both approaches have an issue with not detecting that a key is being continuously held down. The Raspberry Pi receives intermittent key pressed / released signals which is likely related to controlling the Pi over a VNC connection. The mostly likely solution is to track previous key press signals and continue driving the same direction if a particular key press command was received in the last few key events. The best data structure to store a record of past key presses seems to be the python standard library deque data-

structure. It can be set to a maximum length (which it will never exceed) and new objects added at one end of the deque when it is at maxlen will cause objects at the other end to be automatically dumped. It also allows for random access (so that we can check if a command is already in the deque).

NEW PLAN:

Scrap wireless control of the car, plug a keyboard directly into it, and walk it around the track. This will avoid having to deal with inconsistent keyboard command signals, which have been bugging me for a little bit now.

Since Pygame seemed more consistent beforehand, I tried to use it to control the car while using OpenCV to record driving data. I was never able to get this work, as both programs must run simultaneously in different threads. After discussing the options with my brother, we decided to use OpenCV to drive the car.

There were a couple problems I had with using OpenCV for control, so they and their solutions are listed below:

OpenCV's waitkey() method doesn't detect key releases, so the program determines that no keys are pressed (stop the car) when none of the control keys are pressed.

waitkey() also can't be used to detect keyboard combinations. In my Pygame implementation of remote control, the left and right arrow keys turn the front wheels while the up and down arrow keys move the car forwards and in reverse. In Pygame, both the forward and left arrow keys need to be held simultaneously to turn the car. Unfortunately, this can't be done in OpenCV. Instead, I used the W, A, S, and D keys to control the car and had the W key move the car straight ahead, the A key moves the car ahead and to the left, the D key moves ahead to the right, and the S key backs the car straight up. This is somewhat consistent to many video games where the W, A, S, D keys are used for control as opposed to using the arrow keys.

New Challenge:

The track the car follows is made of white paper, on top of which the car will drive. When trying the car out, it became apparent that the car can't turn easily enough to follow the bend in the track. We noticed this earlier but had not dealt with it yet. When the car's front tires are resting on the ground, they can't turn as far as they should. When the front tires are held up in the air, they can turn all the way to their limits.

The front tires and the parts that attach them to the car can move up and down slightly. The weight of the hardware at the front of the vehicle may be more than

the car is designed to operate with. We may have to lighten the load, modify the front axle system, and/or add something underneath the car that slightly raises it up so that the wheels can turn as far as possible. (**EDIT:** In the end we did all three.)

If we don't fix this, the car won't be able to turn as tightly as we want, and it may not be able to follow our track. We could make a new track.

Challenge Solved:

The car can now follow its track well if the driver is good enough (we must drive it correctly around the track to get training data for teaching it how to drive on its own).

Part of the above problem stems from running on low battery power for the car's motors. When the car's driving battery starts getting low, it doesn't turn as far as it should. Eventually, we bought a new and better battery off the internet to help power the car for longer. The original batteries don't seem to last very long.

A small square of wood is now taped to the bottom of the car near the front axle. It is supposed to support the front wheels and keep them from sagging too much. It is also important that this support is properly positioned, otherwise, the car will ride too high and not turn at all or the car will not be supported enough.

We also opened the front of the car to look at the steering mechanism. The two steering stoppers that help limit the car's steering range were removed before reassembly. The camera's mount has been upgraded and it now has a better and more secure/stable platform as a result.

Later, we removed all ultrasonic distance sensors but the front facing one to reduce weight on the front.

We also got a better battery for the motors, which lasts longer and is very reliable.

We can now start collecting data to train the car. While driving it around the track, we noticed that the car's video feed was too blurry because the camera's image sensor wasn't rigidly attached to its circuit board and was vibrating independently from the car while the car was driving. It had a peel off adhesive backing that we used to secure it to the circuit board. It now produces much better video while the car is driving.

TensorFlow:

While a lot of the previous work was happening, I have also been acquainting myself with the TensorFlow machine-learning library. TensorFlow is produced by Google and is one of the most popular libraries for creating AI.

My first encounters with it started as a little experimentation by following official tutorials a few years ago, before I had really learned Python. Since then I have taken COMP 456, Athabasca University's AI course (which didn't deal very much with neural networks) and worked on a Udemy AI course, which uses PyTorch instead of TensorFlow.

However, I have learned more about TensorFlow now than I ever previously did.

I am following code from the DeepPiCar series of online tutorials as my basis for training my AI. The training will have to happen inside Google Colab because training such a big neural network takes time and computing power that my computers don't have on their own and the Raspberry Pi certainly doesn't possess. (**EDIT:** I eventually tried training the network on the Raspberry Pi. The training did seem to progress at a similar rate to training in Colab, training the network for 30 randomly augmented images and testing for 20 un-augmented images per epoch. But after the 5th epoch, Python would crash on the Pi, which is not so great because my training regiment runs for 25 epochs [I did try running the training for 5 epochs, stopping and restarting Python then loading the model from the disk and continuing to train it. This would have to be done five times in a row, in theory, to match one execution online, not very convenient.]. Also, while training, the Raspberry Pi's CPU usage is at 95%+.)

Starting from the DeepPiCar code, I had to make multiple modifications to train the AI. The DeepPiCar was able to steer to a specific angle with a high degree of precision. Its neural network was trained with driving data from a hand coded lane following program that manually identified lane lines using OpenCV, the neural network is supposed to determine the correct angle to steer the front wheels to using only a video feed as input.

My car can't use steering angles, so instead my network outputs four commands: go forward to the left, go forward straight, go forward to the right, and stop, represented in an array as [0, 1, 2, 3]. The neural network outputs the probability that any of these options is the correct one for a given video input. The output with the highest probability of being true is the action that is taken by the car.

So, if my network determines that action 3 (stop) is the best decision to take based on the input image, then the car will stop.

Training Procedure:

The procedure for getting the car to drive by itself is as follows.

The first step is for the user to record training data (both pictures and driving commands) by driving the car around the selected track (run **record_driving.py** to

do this). The greater the range of situations the car is shown how to navigate, the better the car's chances of maneuvering around the track successfully. The user must also be able to drive the car properly around the track. If they can't, then the car certainly won't be able drive properly.

Once adequate training data has been collected, the user can then transfer that data off to Google Drive. In my case, I first transferred the data to my Windows 10 desktop, then on to the internet. The user may have to filter through the training data to remove data that is incorrect or where the driver made mistakes.

Once the files are uploaded, the training can begin in Google Colab (the user will have to allow Colab access to their Google Drive for the program to access the training data and save the TensorFlow neural network model). Colab is a free to use cloud based Jupyter Notebook Python environment often used for AI development. Basically, Google has given everyone access to some free processing power for machine learning and other applications.

The training program is mainly based on code from the DeepPiCar GitHub repo. The training program starts by mounting Google Drive (and asking the user for authorization to do so) then creating the model output directory (if it doesn't exist). The program then imports the necessary libraries TensorFlow, OpenCV, Matplotlib, NumPy, etc.

Then the training data is loaded. Originally, those files were hosted on the DeepPiCar creator's website, but I put mine in Google Drive. The image file paths and control data for each image are then recorded in arrays.

The program plots histograms of the distribution of driving commands held in the data. The program has augmentation functions that randomly adjust a property of the training images, brightness, blur, zoom, random translation, and, originally, randomly flipping images (I removed that one because the car was turning the exact opposite direction it was supposed to). The program preprocesses the training data before they are passed to the network (same thing happens when the AI is driving).

The neural network is built, and training commences for 25 epochs. Once complete, the network is tested and the error over 100 tests is computed. Also, the training and validation loss and accuracy after every epoch are recorded and plotted on graphs.

Once the training is complete, the network, which is saved in Google Drive, can be downloaded onto the Pi and used to run the car.

Testing tracks:

There is the question of what type of track to use. In theory, any driving track could be used with the car so long as it is possible for the car to drive along it and the car can be trained to drive along it.

We started with driving along a track made from a large roll of paper that we had for quite some time. There were two straight sections and a smooth curve. Unfortunately, between the fact that my car doesn't always turn properly (when its battery is low) and the small area available to make the turn, I only made one successful run around the track's curve (going along the straight parts is easy).

After many tries to get successful training data for the car when driving it along the track, I switched to another type of track where I tried to make a track out of blue construction paper (much harder to keep in place). I tried training the car using data obtained from driving it around the track, but for some reason, its resulting behavior was unpredictable, sometimes it would go the exact opposite direction of what it should.

I then scraped the blue construction paper track idea and decided to try driving the car straight along the original white track, while just getting the AI car to determine when to stop and when to drive straight ahead. Using this approach, it becomes clear that there is noticeable time lag between the car driving off the end

of the track and the car stopping, initially a good distance away. Slowing the car's speed down decreases the lag time, but there must be something slowing the processing of things down. I also cut the size of recorded videos in half from 640x480 to 320x240 and am attempting to shrink the size of the neural net. Disabling ultrasonic sensor readings doesn't have a noticeable effect on the lag.

So now I am wondering: what else can be done increase the speed the AI driving program runs at? Do I have to disable VNC? Does the Raspberry Pi need to go headless? I am still seeing the lightning bolt in the top-right of the screen, which is warning me of lower than sufficient voltage reaching the Pi and that the Pi's processing power is cut in half as a result, so do I need a new power source/cable to run the Pi? The answer to most of these questions ends up being no, and the car doesn't seem to have a problem with processing speed.

Regardless, the car originally stopped well beyond the end of the track (a few feet in fact – not good), but now stops around half of its length from the end of the track, which is a much better result.

The size of the training images is reduced even further to 85x64. For one, they don't need to be that huge. In my Udemy AI course, an AI was trained to play Doom with only 64x64 input images as input. The TensorFlow 2.0 initial tutorial uses 28x28 input images to classify types of clothing.

For another, the first time I tried uploading images to Google Drive for use in Google Colab, I quickly realized just how big a task it is. With the number of pictures needed for proper training of the AI, it could take hours to upload 1000+ (I later on end up using training sets that size) on my internet connection, not to mention the amount of storage those pictures need wherever they are located.

Success!

Last night (March 12, 2020), the car finally drove around a section of track that was outlined with blue tape with good accuracy. It wasn't perfect, but now I made a track that loops around a pool table that I am going to teach the car how to drive around.

A track made from blue tape is better than the previous tracks. It is also what the DeepPiCar creator used. This type of track is flexible and easy to reconfigure (not possible with a big roll of paper) while also staying firmly rooted on the floor (not easily possible with individual pieces of paper that are knocked out of place whenever the car [or anything else] runs [or walks] into them).

The car's neural network, in the end, is largely based on what was used in the DeepPiCar tutorial for lane navigation as is the code for training it. The changes made to the network include changing the size of the input to match the input image size of 85 X 64 pixels and changing the output layer to use the SoftMax

activation function while increasing the number of outputs from one to four (because this is now a categorization problem).

My neural network is designed to produce a range of four output values that correspond to 4 driving commands. However, the DeepPiCar's neural network was designed to output a range of steering angles from 45 to 135 degrees.

I tried making smaller networks with fewer layers because I figured that the DeepPiCar's neural network was overkill for the task at hand, having multiple levels of Convolutional and Dense layers for a network that did not need to produce a huge number of potential outputs. The DeepPiCar network is based on a network Nvidia used to drive a full-scale car in the real world.

Among TensorFlow's official tutorials is an example where a neural network only a few layers deep can categorize 28x28 images as various types of clothing with over 90% accuracy. Given that I was just trying to categorize input images to produce 4 possible output actions, I figured that I could do something similar. At the same time, I hoped to speed up image processing on the Pi. Interestingly, fewer layers doesn't = fewer parameters to train in the network or faster training. On the contrary, almost all the alternative networks I tried had far more parameters to train than the Nvidia Model. In fact, the Nvidia model seemed to be faster to train than

the other models while requiring a fraction of the training needed in the DeepPiCar tutorials.

I was able to get the car to drive along the straight stretch of white paper and stop automatically when needed using a scaled down model that had one Convolutional layer and one Dense output layer with a flatten and dropout layers in-between. The same network also was good at driving around that short section of blue tape track → well not perfect, but that might be due to driving mistakes while collecting training data.

I was forced to go back to the Nvidia neural network model after trying various alterations of the smaller network with training data I collected from driving the car around the larger loop. The smaller networks were not getting the training accuracy I wanted, and maybe I should have taken the hint that something was wrong. They only got to ~88% accuracy on training data and 90%+ on validation data while the Nvidia model got 94 and 95%, respectively in Google Colab.

The trouble is that the car didn't drive around the track properly. At one end of the track the car consistently missed the turn. When driving manually, it was also hard to make that turn; therefore, I widened that end of the track to allow the car to navigate it better.

Drive Using AI Program:

The program the drives the car using AI is structured works as follows.

First, the program takes in the live camera feed and sends it to the trained neural network. The neural network processes the image and passes the most likely action that the program should take to the program. The program will then try to perform that action.

Meanwhile, the car's program is polling two other sources for commands: user keyboard activity and the forward ultrasonic sensor. Anything object that the ultrasonic distance sensor detects is closer than 45cm will cause the car to stop. If the user hits 'q' on a keyboard, the program will end. The space key functions as a pause/play command for the user, when hit, the car will toggle between stopped and driving, so long as neither the network nor the ultrasonic sensor determines that the car should remain stopped.

The neural network can be trained with images that are designated as situations when it should stop. If it determines the input image matches a stop condition, the car will stop based only on what it sees, i.e., "Am I still driving on the track?"

Of course, having a good neural network is one thing, as is the program that makes use of it. But in this case, properly training the network and having a vehicle fully capable of doing what you want it to do is also very important.

Driving Around a Full-Loop Track:

Okay, after getting the car to drive along a short segment of track, the next goal was to drive the car along a longer stretch of track, a full loop. This took a lot of attempts to get right, unfortunately.

Most of the trouble probably came from driver error/bad training data, battery supply issues, lighting conditions, and/or choice of neural network. Most of the programming is probably correct.

For instance, I had a couple of revelations about driving the car in the past little while. For one, when you are making the turn, don't stop midway. This seems like common sense to someone driving a big car, but I somehow forgot it. The car needs time when driving around a turn to fully move its wheels to the right. When I was collecting training data by driving around a turn, I frequently stopped midway, which caused the tires to move away from being fully turned, resulting in them needing more time to turn back to full right in order complete the turn, if they even returned to that position. ~~I think this is a problem when the car is low on juice and cannot turn as far.~~

Also, you need to be driving straight before you start turning, otherwise the car's wheels won't turn as far as they should. Don't start turning before you are moving → my code triggers the car's wheels to turn before the car starts moving. I might

need to change this. Funny thing, I have done the same thing before with the family van.

As a result of difficulty driving the car around the track, I had to re-write some of the autonomous driving and recording of driving data programs to incorporate functionality to display real-time visualization of what driving procedure is currently underway. My code for recording and viewing the current status of the car (and keyboard interaction with it) while it is driving around a track has been moved to a subclass of RecordDriving, the class that is used to record training data. As a result, the code for driving the car autonomously is a lot clearer.

At least some of the problem with driving around the track has to do with bad training data and trying to do to many things at once. I had to remove to code that randomly flipped training data from the training code because the car kept turning the wrong direction, turning left when it should turn right. Also, the data I used to train the car how to stop caused it to stop in the middle of the track.

After getting new training data, I tried to get the car to go around the track again. Most of its activity was still random, but there was at least one loop where it seemed to follow the track well if you count not staying within the lines. However, it was hard to get it to do that properly. I probably still need to get better training

data to get it to follow the track properly and make sure I train and test in consistent lighting conditions (get the training data and then come back quick and test it).

Kitchen Table

I was discussing the current problems with my family and the suggestion came from my mom and brother: “Does the car need to run around a track?” “Why don’t you drive it around the basement?” “No wait, there is too much dog hair down there.” “Well there is dog hair everywhere, does it really matter?” “Why not drive it around the Kitchen Table?”

At that point I was like “Okay let’s try that.”

I don’t remember the exact conversation, but it went something like that.

So, we did this. First, I drove around the table for a few laps. Then I took the training data and trained the neural network model in Google Colab. Then I had some record video of the car running around the table autonomously and

SUCCESSFULLY! YES! FINALLY!

Okay, it wasn’t perfect, and I made a few mistakes while collecting training data that I can directly correlate to places where the car goofed up while on autopilot. Regardless, it performed well above what it was doing on my custom track. It

might help that this occurred at night when all the curtains where closed and the lights were all on, giving this room much better and more consistent lighting than where I was testing before. The upstairs room where I put my track has a lot of windows open during the daytime and bad lighting at night. Time of day, weather conditions outside, and glare can all play havoc on what the car is seeing. Therefore, it is greatly preferable to drive the car in my family's kitchen or the basement than upstairs.

This whole period took less than an hour and is quite illuminating on how fast a turn-around time this car can have if you make perfect training runs and/or don't care if you made some mistakes. In my case, in 3 out of 4 training laps, I drove the car into a mat at one end of the track; therefore, it shouldn't come a surprise that the car made mistakes at that corner. Sometimes it would run into the mat. Other times it turned too early and ran into a chair.

Round 2:

Taking new training data adapted to daylight conditions produced better results. There were some laps where the car did not run into anything.

Round 3:

This time results were not as nice. I guess there was some bad training data. Interestingly, removing a chair from one side of the table caused the car to turn

right where the chair originally was and drive under the table, but putting the chair back fixed this.

As part of the DeepPiCar tutorial code that I use for training the car, the training and validation accuracy and loss results are recorded at the end of every epoch (they are also recorded in a history.pickle file). Once all the epochs are complete, this data is displayed on graphs with matplotlib. This is a way of visualizing the results of training. If the original data was inconsistent, like it was for this training set, the lines will be zigzagging wildly up and down.

Round 4:

For this test, I used a modification of the previous training set, where I removed some questionable data. I also trained a second model with even more training data removed, but I did not need it.

This time, the car performed nearly flawlessly: picture-perfect. Only three test-laps had problems: the first one, the last one, and one where I stopped it then let run again – into the couch. Several other laps went awesome! The car autonomously steers away from chairs and stays on track.

Now we can go back to the blue tape track and ***NAIL*** navigating around it.

How did I miss that this entire time?!

Okay, so to get the results above took several attempts at recording data, training, then testing the results to get the desired effect: the car running properly along the track – and even then, it wasn't perfect. I had to also get training data that taught it how to avoid obstacles because it would not do it on its own.

But this whole time, there was a major underlying problem with the system that went unaddressed: I was manually driving the car teaching it how to drive while using significantly different driving behavior from what the car exhibited on autopilot. Fixing this is like night and day from what I had before. To be sure, nothing was fundamentally wrong with the Autonomous driving code, the training code, or the manual driving code. The real problem lay inside the keyboard repeat time settings on the Raspberry Pi. The Pi was set so that the when a key was held down, it would wait for about 500 milliseconds before repeating the key held signal after which the key down signal would be sent repeatedly every 30 milliseconds. The key down signal was received by OpenCV's waitkey() method, if no key was received, the car would stop and straighten its wheels. Trouble is between the time when the key down signal was first sent and the time when it started to be repeatedly sent, the car received no key down signal and interpreted it like no key was being pressed. Therefore, the car received the stopDriving() command multiple time both when switching keys – when you want it to say turn

right rather than go straight or when it started moving after being stopped. This in turn affected the driving behavior of the car significantly.

The Solution:

The first solution would be to change the keyboard repeat message times – I played around with them earlier in the project and then forget about them. Trouble is, the smallest time you can set before the keyboard repeats the signal is 100 milliseconds, which is enough to sneak in at least on stopDriving() signal to the car. The other problem is that setting these times can mess with typing on the Pi when the program isn't running.

The real solution is to set keyboard repeat times inside the record_driving.py code. This is done by calling a Linux shell command, "**xset r rate 30 30**" (from this post) with a Python function that executes shell commands, "**os.system()**" (from this post). I set the time between the first key press to the first key repeat to 30, and the time between repeats to 30 at the start of the program. Later, at the end of a recording, the keyboard repeat times are set back to normal.

The Solution, Part 2:

I have tested this new modification three times. It appears that more is going on besides keyboard repeat times that are affecting the car's performance. For one, I believe that training the AI and using the AI still don't occur under the same

circumstances. Because I manually drive the car with a corded keyboard connected, then detach the keyboard to drive manually, the car has different driving characteristics because of the cord's weight, drag, and/or being held back by the user. I believe I need to hold the cord up off the ground while training and make sure that I am not pulling on the car and inducing drag for it.

Another problem could be lighting conditions.

I do know that more than once the car produced decent driving around the blue tape track, but both times the keyboard was connected. The second time, the keyboard was disconnected midway through, but the car immediately started misbehaving until its batteries became depleted. After that run, I finally understood this problem. The point is the car can now run autonomously around the blue tape track with the keyboard connected to it, but when the keyboard is disconnected, the car can't properly run anymore.

I would like to get the car running keyboard-less, but my final exam is just over two weeks away and I need to submit this project and prepare for it(also, we are going into busy season on the farm and I might not have as much free time before the exam). Therefore, because I already have a video on YouTube demonstrating its ability to drive autonomously, I will have to submit the project as is. I do intend

to get the car working better later and will post a link to the video on the Landing (or even in the project comments if I can do so before this project is marked).

Things I would do differently next to:

I would invest in a better car chassis with stronger batteries and a good suspension. A chassis that doesn't use a regular dc motor to control steering would be preferred (we already had this car at our house, so it was cheaper to use what we already had).

The publicly available Donkey Car self-driving RC car system might be a better and more advanced alternative to my setup.

Next time around, I would try to make the whole system easier to use. Maybe I would figure out how to train the AI directly on the Raspberry Pi, or at least on a local computer as opposed to the cloud.

Double-checking and editing training data should be easier than going over it in a file manager.

Controlling the car manually with a keyboard is not the most effective or intuitive control method possible but works fine for us. A wireless control method, even a wireless keyboard (we don't have one) would be more effective.

Reflections:

I learned a lot over the course of this project.

For starters, I enjoyed working with my brother: he is good at working with his hands and building things while I am good a programming and software. We make a great team, and this will only be the first big project we worked together on. Besides, running ideas past other people is a good practice and helps make things as whole better.

I started ~~using~~ posting code to GitHub partway through the course, and really got into it after I started the project. Surprise, surprise, I really like GitHub.

I proposed this project partly because we already had much of the hardware needed to complete it; therefore, it wouldn't take much of a budget to complete the project. We only bought the camera, a new battery pack for the car's motors, and a pack of 5 ultrasonic distance sensors with some accessories (originally, two of the sensors were mounted with brackets that came with them to the front of the car, but they were unnecessary and we ditched them to help save weight). I am also a fan of Tesla and autonomous self-driving vehicles, and this project is a scaled down version of one (it's even ***ELECTRIC***).

It seems that getting good training data is a bigger problem than using that data. If you have a good program/neural-network design, but bad training data for that

network, the network is going to be useless. You need to make sure the training environment is close enough to the testing environment for this to work (or that you have a wide enough range of training data that covers your testing environment conditions).

If we were hand coding lane following (like they did at first in the DeepPiCar tutorials [part 4]), it would take a lot of custom code and time to make the lane follower, which would be inflexible and useless in a different environment. With the neural network, the program just needs new training to transition to a new environment.

I did spend a bunch of time on this project, a few months at least. I am glad it is finally working properly.

As noted, I really didn't find all the bugs in the project until towards the end of it; therefore, I spend a lot of time banging my head against the wall trying to solve these problems (and I didn't even suspect the root causes!). I must have run dozens of tests with the car. My Google drive already has 26 separate directories that each hold a trained model (I renamed the model and the data folder for each dataset). There were five different test tracks: the white paper track, the blue construction paper track, the short blue tape section, the full tape loop, and the kitchen table.