Building a Self-Driving RC Car Using Donkeycar

Overview:

For our final project, we decided to build and train an autonomous vehicle using the Donkey Car platform.

Donkey Car is a TensorFlow based, open source platform for autonomous RC car racing. It was initially developed by Will Roscoe and Adam Conway in late 2016[1] and currently has an active community of maintainers/users [2].

Here's how it works:

- Build the car: The essential components are an off-the-shelf RC car, a Raspberry Pi, and a camera. Recommended parts, instructions and even starter kits are available on the Donkey[3] website.

- Install the software: You'll need to install some software on the Pi and your computer. These include TensorFlow libraries plus the Donkey Car application[4]

- Build a track: To train the car, you'll want to construct a course. Laying down strips of tape to demarcate a lane (or multiple lanes) is the typical method. The idea is that the boundaries defined by the tape will hopefully be the primary visual queue used by the learning algorithm.

- Train your car: Next, you'll need to collect a bunch of data to use for training. From the donkey car interface, you'll start recording data as you drive the course. The Raspberry Pi will save images as well as labels recording the steering angle and throttle. After you've collected at least ~5k images, you are ready to begin training your model.

- Build a model: This is the step where you'll train a neural network on the training data that you've collected. Donkey Car software uses the Keras API to train a TensorFlow model.

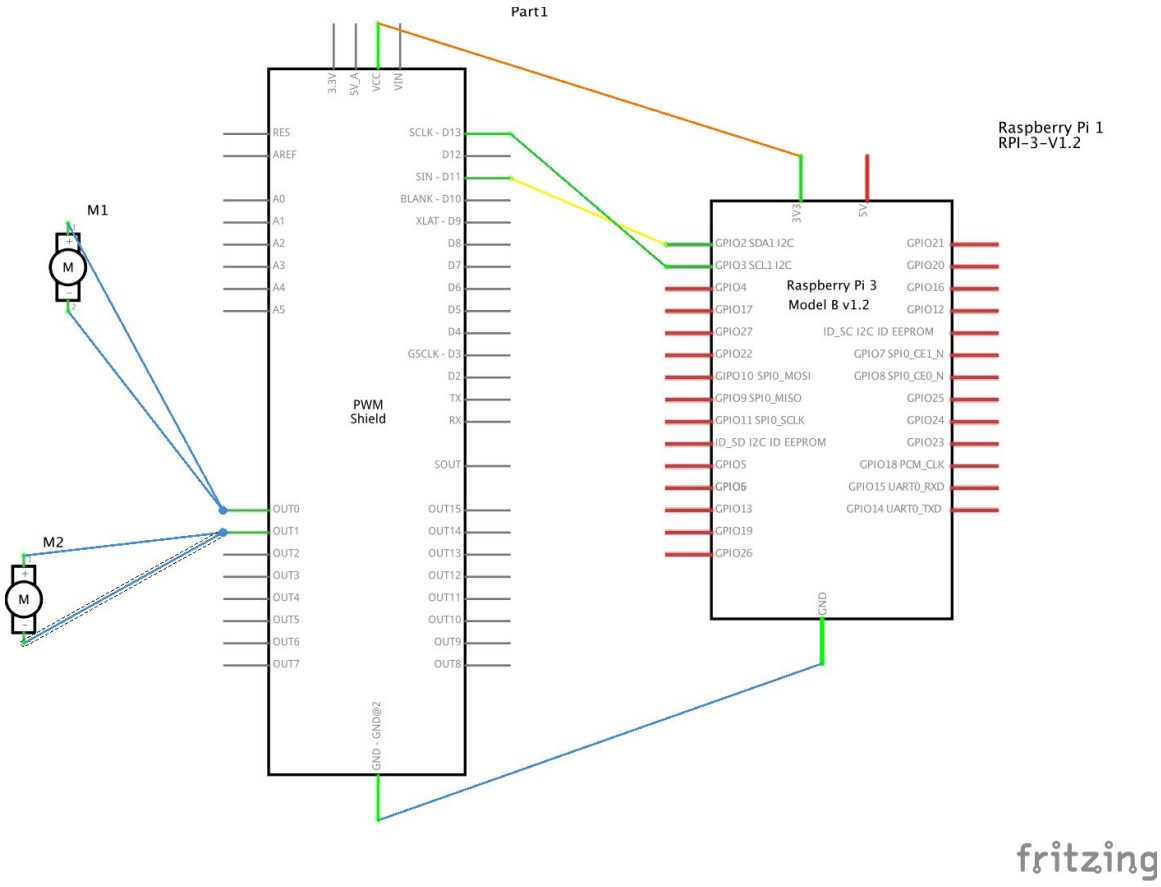
- Test the model: Upload the model and see how it performs. You can even race it against others at autonomous car racing events! [4]

The Build:

The key components of the car are:

1. The mechanical components, e.g., wheels, chassis, etc.
2. The electromechanical components, e.g., the motor and steering actuator.
3. The raspberry pi, this acts as the logical input for the whole system
4. A servo which acts as the electromechanical control for the system
5. A camera to collect input data
6. A 3d-printed roll cage for mounting the electronics and the camera
7. A battery to power the electromechanical components
8. A battery to power the raspberry pi

The circuit diagram is shown in the figure below.



After receiving the parts, we started assembly. Immediately after starting assembly, we realized that we had some part incompatibilities that would require a few trips to the hardware store.

The first one was the screws, we needed screws of a specific size to fit in the roll cage for the car. This need was solved by ordering the correct screws from Amazon.

The more critical incompatibility was the battery connector between the lithium-polymer (LiPo) battery and the car's motor. We were not able to find an adapter between these two connectors at any hobby stores in Austin or online.

Wanting to use the LiPo battery to increase the amount of training data collection we could do on a single charge. We decided we would solder an adapter together from individual components. With the history of LiPo batteries fresh in our mind, we decided to recruit a friend more familiar with soldering.

Software

With the hardware assembled, it was time to turn on the Raspberry Pi and install the necessary software. This turned out to be more of a process than we would have liked.

The first issue we ran into was the fact that we did not have a spare keyboard and mouse which could be plugged into the Raspberry Pi. Without input devices, we had to configure the Raspberry Pi to connect to our home WiFi at boot time so that we could then SSH into it. Unluckily for us, Debian recently changed the format of the file which controls how the operating system connects to WiFi and all the resources we found online were for the old file format. We eventually resolved this issue which we can dig up the solution for if you email us at blog@howinator.io.

After connecting to the Raspberry Pi, we got the correct version of Python and tensorflow installed on the Raspberry Pi which was not seamless, but is well documented on the internet.

Calibration

In order to maintain consistency between runs, the car is calibrated of throttle and steering. This was a fairly straightforward process of iteratively modifying the maximum value for steering and throttle until the car is at its maximum left and right steering position, and the car is at a comfortable maximum throttle.

The next step in the process
Training the Autopilot:

Tensorflow requires a lot of data in order to build a good model. For a simple track, you may be able to build a reasonable autopilot with a single training run of around 10 laps, but the model will be very fragile. Different lighting conditions will impact the model performance and the model will not be able to generalize to other tracks. Also, if you let the autopilot attempt the same course in the reverse direction, it will likely not work at all.

A big consideration when collecting data is managing generalization error. The obvious example is if you collect all of your training data on a single course, there will likely be many visual cues along the course that will be picked up on by the neural network in addition to the lane markings. If you aren't collecting a lot of data, then you may be inadvertently training the autopilot to turn left whenever it sees a tree or a chair.

We performed many training runs on different courses both inside and outside

Outdoor training runs: [add some info on the outdoor training?]

Indoor training runs were done in a lighted 320 square foot building with windows allowing a variable amount of additional light in. The autopilot performed fairly well after training on the first course at night:

The same autopilot did not do so well during daytime when lighting was different. Additional training runs were done various daylight conditions, and also with several different course setups, including a couple of courses that were designed to

present the autopilot nearly identical backdrops where the only visual cue would be the course markings:

Figure 1: Course modified to present the AI with identical backdrops in order to push it to make a decision based on lane markings alone

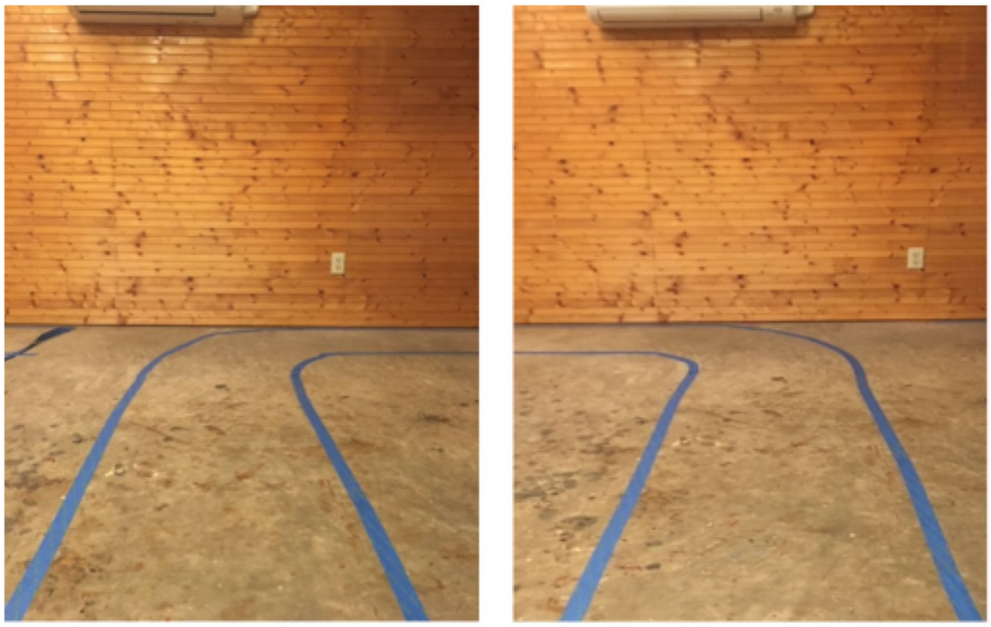


Figure 2: Course modified to present the AI with identical backdrops in order to push it to make a decision based on lane markings alone

The final training indoor course was relatively more complex, but after training with all the previous courses, the autopilot performed very well.

Building a model:

Models are trained with Tensorflow using the annotated images collected during training runs. The autopilot training method included in the standard Donkeycar distribution[1] works fairly well, but we did find some other options available. One popular distribution[6] includes several additional methods for autopilot training.

One of these included methods is a recurrent neural network (RNN) option that seemed to work very well. RNNs have a temporal component, so the output of the model not only depends on the current image being evaluated, but also on the output of previous images. The specific architecture we used is two-layer LSTM RNN with the exact architecture shown below.

```
x = Sequential()
x.add(TD(Cropping2D(cropping=((60,0), (0,0))), input_shape=img_seq_shape )) #trim 60 pixels off top
x.add(TD(Convolution2D(24, (5,5), strides=(2,2), activation='relu')))
x.add(TD(Convolution2D(32, (5,5), strides=(2,2), activation='relu')))
x.add(TD(Convolution2D(32, (3,3), strides=(2,2), activation='relu')))
x.add(TD(Convolution2D(32, (3,3), strides=(1,1), activation='relu')))
#x.add(TD(Convolution2D(32, (3,3), strides=(1,1), activation='relu')))
x.add(TD(MaxPooling2D(pool_size=(2, 2))))
x.add(TD(Flatten(name='flattened')))
x.add(TD(Dense(100, activation='relu')))
x.add(TD(Dropout(.1)))

x.add(LSTM(128, return_sequences=True, name="LSTM_seq"))
x.add(Dropout(.1))
x.add(LSTM(128, return_sequences=False, name="LSTM_out"))
x.add(Dropout(.1))
x.add(Dense(128, activation='relu'))
x.add(Dropout(.1))
x.add(Dense(64, activation='relu'))
x.add(Dense(10, activation='relu'))
x.add(Dense(num_outputs, activation='linear', name='model_outputs'))

x.compile(optimizer=optimizer, loss='mse')
```

Model Performance:

In addition to finding that RNNs performed better than many other model types, we found there were significant differences in the choice of the optimizer. Comparing between RMSProp and Adam, we found the time-per-epoch was approximately the same between the two, but the training error was a full 2% less for Adam.

Model Performance:

We had trouble comparing the testing error of each model since there is no “right” answer for what the model should do at each position on the track. We considered quantifying testing error by counting the number of times that the model deviated outside the lane. This is an imperfect measure though because some models may be more tolerant to returning to the track after deviating outside the lane. In the end, we decided to go with a qualitative measure for model performance.

These results are shown in the table below.

Type	Batch Size	Seq. Length	Optimizer	val_loss	Steps/Epoch	Seconds/Epoch	Test Perf. (1-5)	Notes
Linear	128	N/A	Adam	0.07	404	39	2	
RNN	128	3	RMSProp	0.038	404	121	3	
RNN	64	3	RMSProp	0.1357	967	116	2	
RNN	**128**	**3**	**Adam**	**0.0351**	**407**	**97**	**4**	**k=.9**
RNN	128	30	Adam	0.0225	413	602	1	k=.9
RNN	128	80	Adam	0.0357	407	1582	0	k=.9
RNN	128	7	Adam	0.0328	415	146		k=.9

References:

[1]https://makezine.com/projects/build-autonomous-rc-car-raspberry-pi/
[2]https://donkeycar.slack.com
[3]donkeycar.com
[4]https://github.com/autorope/donkeycar
[5]https://diyrobocars.com
[6]https://github.com/tawnkramer/donkeycar