Capstone Project

Machine Learning Engineer Nanodegree

John A. Chen

June 21st, 2017

Teach a Car How to Drive Itself

Definition

Project Overview

Udacity launched the Self-Driving Car Engineer Nanodegree (SDCND) program to teach students how to build self-driving cars like Stanley: https://www.udacity.com/drive. In the third project in SDCND, the student's task is to architect and train an artificial deep neural network to steer a simulated car around a test track by feeding it track images and their associated steering angles. This approach to machine learning is called behavioral cloning and was based on the same concept of using "human reaction and decisions" that Stanley used in the Challenge. What is new is the use of artificial deep neural networks to steer the vehicle as described by NVIDIA: https://devblogs.nvidia.com/parallelforall/deep-learning-self-driving-cars/.

Figure 1: Stanley, the winning vehicle in the 2005 DARPA Grand Challenge

In this project, we created a trainer using Anaconda, Python 3.5, Keras, Tensorflow, NVIDIA CUDA and Titan X GPU, Udacity's Self-Driving Car Simulator to use a previously trained neural network to train a new neural network how to drive a simulated car on the same track as the one for project 3, but this time on its own using real-time image data from the simulator in autonomous mode and reinforcing corrective behavior. This project was inspired by DeepMind: https://deepmind.com/ and their human-level control through Deep Reinforcement Learning: https://deepmind.com/research/dqn/ and their AphaGo training method: https://deepmind.com/research/alphago/.

Problem Statement

For Project 3, Udacity has created a car simulator based on the Unity engine that uses game physics to create a close approximation to real driving, where the students' are to use Behavioral Cloning to train the simulated car to drive around a test track. The student's task are:

1. Collect left, center and right camera images and steering angle data by driving the simulated car
2. Design and create a CNN regression model architecture for learning the steering angle based on the center camera image
3. Train it to drive the simulated car
4. Test the trained model in the simulator in Autonomous mode.

5. Repeat 1-4 until the trained model can drive the simulated car repeatedly on the test track.

But, training in batch mode is tough! Times from step 1 to 3 could take hours, before trying on the simulator in Autonomous mode and find out:

1. Data collected was not enough
2. Data collected was not the right kind
3. Key areas in the track were not learned correctly by the model
4. Model was not designed or created correctly.

Clearly the data collection, model design, training and testing cycle needs to be faster. The goal of this project is to design an automated system to iterate the model design, data gathering and training in a tighter automated loop where successive improved models can be used to train even more advance models. To accomplish this goal, the tasks involved are the following:

1. Evaluate the current Udacity Self-Driving Car Simulator for the task
2. Design and create the automated training system and its training method
3. Select a trained DNN model as the initial generation (*g=0*)
4. Select target DNN models for training as the next generation (*g+1*)
5. Train using the automated training system
6. Test the target DNN model effectiveness
7. Discard target DNN model and repeat 4-6 if it does not meet metrics criteria
8. Repeat 4-7 successively using DNN model generation *g* to create generation *g+1*

Metrics

We will be using two metrics to score how well the target model is being trained. The first score will be mean square error of the steering angle using the previous generation as ground truth going one lap around the track:

$$Steering_{mse} = \sum_{n}^{1} \frac{(Steering_{g+1} - Steering_{g})^2}{n}$$

where *n* is the number of samples to go one lap around the test track, *g* is the generation of the trained model, *g+1* is the generation of the target model. This score will tell us how close the target model follows its training set (the trained model) and is applied by the Keras Adam optimizer as a loss function during model training. The second score is the accumulated *Cross Track Error* (CTE) going around the track:

$$Accumulated_{CTE} = \sum_{n}^{1} CTE$$

where *n* is the number of samples to go one lap around the test track. CTE can be easily calculated given a set of *waypoints* that the vehicle should follow around the test track by fitting a 3rd degree polynomial to the points and calculate the shortest distance from the vehicle to the curve. This score will tell us how close the target model follows its test set (the *waypoints*). We will consider any single CTE greater than 2 meters off from center (0) to be off track (the simulator test track is approximately 4 meters wide) and fail the model during testing. We will go in more details about *waypoints* and how they are used in calculating CTE in later sections.

Analysis

Data Exploration

The Udacity simulator for project 3, when in autonomous mode, will send the vehicles front camera images, throttle/brake combination, and steering angle. When the throttle is applied, the throttle value is positive number, while when the brake is applied a negative number is sent instead. For our training environment, we will set the throttle/brake such that it maintains 30mph speed via a PID controller. The number of samples that are needed to run the entire test track is variable depending on the speed of the vehicle. We have determined that around 900-1350 samples (time steps) are needed to go around the test track at 30mhp depending on the speed of the CPU and GPU. We used NVIDIA Dual 1080s in SLI and Titan X Pascal GPUs for training and testing. We had some difficulties obtaining the data from the Project 3 simulator to calculate the CTE, but will go into more details in the *Algorithms and Techniques*, and *Data Preprocessing* sections on how we address this problem. Regardless, we are obtaining the raw data at each sampling point from the simulator in the table to the right.

Features	Description	Type	Units
`image`	Vehicle center camera **(INPUT)**	320x160x3 JPEG compressed image - UINT8	RGB pixels
`speed`	Vehicle velocity ***PID controlled.***	float	Mile per Hour (mph)
`steering_angle`	Vehicle steering angle **(OUTPUT)** **MSE Calculation**	Float	Radians from -pi/4 to pi/4, where 0 is straight forward

The image feature is an JPEG encoded 320x160 pixel RGB image. This will be used by the trained model (g) to predict the ground truth steering angle and use by the target model (g+1) as training input. The speed feature will be used by the PID controller to maintain a 30mhp speed around the test track. This next table to the right shows the statistics on a data sample collected doing a little over a lap around the lake track. The sample collected includes speed, steering_angle.

	speed	steering_angle
count	945.000000	945.000000
mean	29.864284	-0.036600
std	3.789990	0.093276
min	0.000000	-0.442037
25%	29.999140	-0.081519
50%	30.018270	-0.030914
75%	30.050670	0.006358
max	36.455230	0.379061

NOTE: The `steering_angle` feature is not generated by the simulator. Instead the trained model will generate this feature to be used for training the target model.

Exploratory Visualization

As explaining in the previous section, speed will be controlled by a proportional-integral-derivative (PID) controller. It essentially set the vehicle into cruse control, but needs about 36 time steps to go from zero to 30mph and another 100 more times steps to settle to that speed as can be seen in figure 2.

Steering angles are a little more interesting. They are what the model will try to predict given the front camera image. This is assumed to be a regression problem because steering angle values are continuous; however, on closer examination, they can be grouped into bins and plotted in a histogram to show their real distribution. Looking at the steering angle statistics, we can see that the lowest steering angle is at -0.442037 and the maximum is at 0.379061 radians along the test track, which is less than the -pi/4 (-0.7854) to pi/4 (0.7854) range.

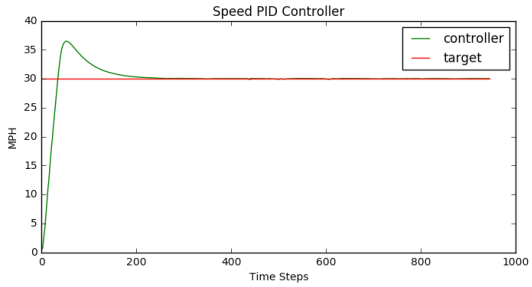

Figure 2: Speed PID Controller Behavior

We create 9 bins from -0.4 to 0.4 with 0.1 increments and the steering angle distribution is shown in Figure 3. The majority of the steering angles are straight, with the distribution skewing to the left. This makes sense because the vehicle is going around the test track counter-clockwise with majority of the time making left turns.

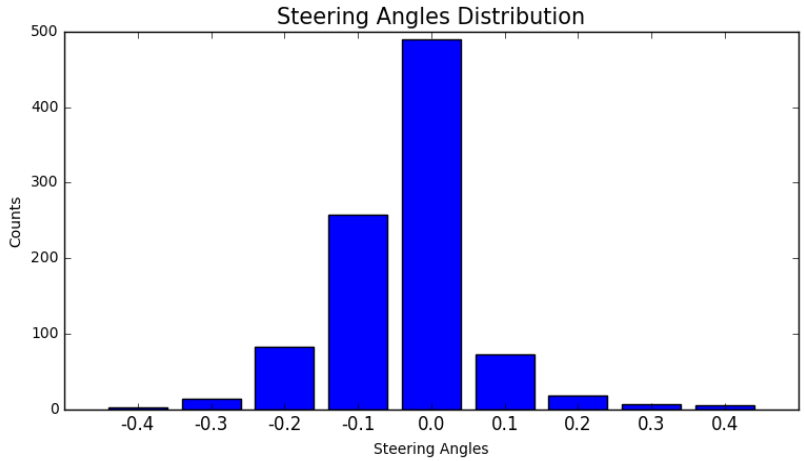

Figure 3: Steering angle distribution for single lap around test track

We can visually inspect this by randomly selecting a front camera image associated with the steering angle as in figure 4.

From the figure, it appears that the -0.4 steering angles are associated with the start of the lap, since the vehicle is placed to the right of center, and 0.4 steering angles are associated with 5 time steps at a particular curve at the test track. The majority of the steering angle is in the -0.2 to 0.2 range, with majority of the time the vehicle is steering straight.

We tested the theory that the vehicle could be controlled by a classifier by limiting the values of the returning steering angles to the 9 values from -0.4 to 0.4, and the vehicle was able to stay on the test track with surprisingly good accuracy. We will use this insight when deciding if the predicted steering angle from the target needs to be corrected by the ground truth steering angle if their difference are more than 0.05 apart.

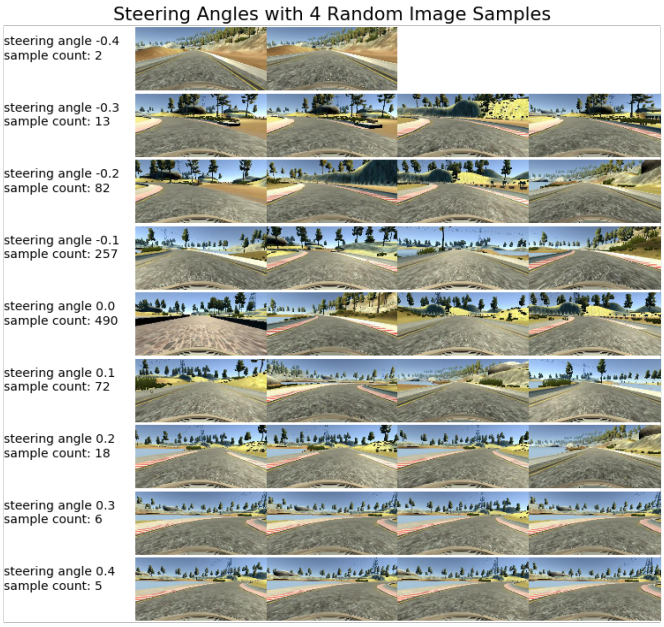

Figure 4: Image samples associated with steering angles

Algorithms and Techniques

As stated before, the goal of this project is to design an automated system to train target convoluted neural networks (CNN) steering models using previously trained CNN models. Below is a diagram of the architecture of the automated system and how the trains the target CNN. It is similar as the one used for our project 3 Agile Trainer (https://github/diyjac/AgileTrainer) where a continuous stream of images is stored into a randomized replay buffer for training batches, but now instead of manual intervention by an human, we use a pre-trained model to handle corrective steering actions. Similar to reinforcement learning, we will have a training period where the majority of the time the target model will just receive training at higher learning rates; however, as time elapse, we will reduce the training and start having the target predict the steering and any mistakes will be corrected by the trained model and added to the target model for training refinement.

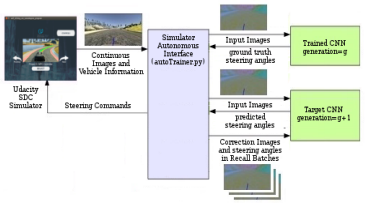

Figure 5: Trainer in Training Mode

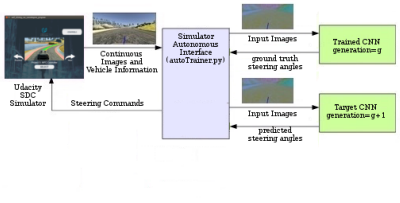

Figure 6: Trainer in Testing Mode

We will start off by using our trained NVIDIA model as generation 0 (*g=0)*, and a modified NVIDIA target models in our test as generation 1-3 to see if this successive models can perform well on driving the car around the test lake track. Below are the models used in this evaluation and their generations:

Model	Description	g
NVIDIA	Pre-Trained NVIDIA Steering model.	0
NVIDIA	64x64 model – no weights	1
NVIDIA	64x64 model – no weights	2
NVIDIA	64x64 model – no weights	3

The initial selected model architecture for all generations is NVIDIA's Steering model: https://devblogs.nvidia.com/parallelforall/deep-learning-self-driving-cars/ (*Figure 7*). Each model will be given 20 training and test sessions and the test session with the best resulting weights will be selected as the trainer for the next generation. We wanted to test concept before venturing further with custom models.

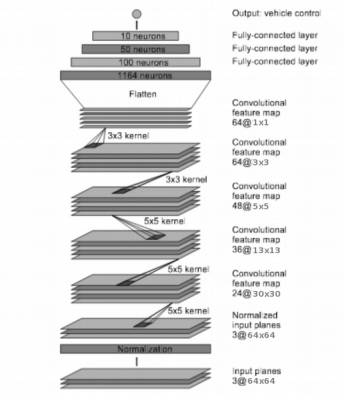

Figure 7: NVIDIA Steering Model with 64x64x3 input

Benchmark

We will be using Accumulated CTE (ACTE) to test the end result of the target models and see how well they perform against the trained model after each training session. Our goal is to train at least one model with same or better ACTE at each generation after successfully completing an autonomous lap.

Methodology

Data Preprocessing

For data preprocessing, our system needs to be able to:

1. Crop and resize the image according to each model (generation)'s needs.
2. Obtain the CTE based on the current location of the vehicle in the simulator.

Our original image from the simulator is 320x160 pixels (samples are shown in *Figure 4*). Each model (generation) is encapsulated in a python **`Model`** class and have an unique **`preprocessing`** class function that will crop and resize this image to a pixel resolution according to its needs. We use OpenCV **`resize`** function to do the image resizing.

To calculate for CTE we need waypoints. We will go into details on how we were able to get the track *waypoints* and the vehicles current location from the simulator in the next section, *Implementation*. However, the waypoints and current location information are provided in a global X, Y coordinates and we need to calculate their coordinates based on the vehicle's position as the origin (0,0) to be able to calculate the CTE, which is defined as the shortest distance from the vehicle and the curve form from connecting the *waypoints* on the track. So, the first task is to perform a map rotation around the vehicle to where its heading is θ (psi). For a rotation through an angle θ with the vehicle heading in the positive x-axis direction, the operation looks like this using matrix multiplication:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

Figure 8 shows an example of what the map rotation visually looks like:

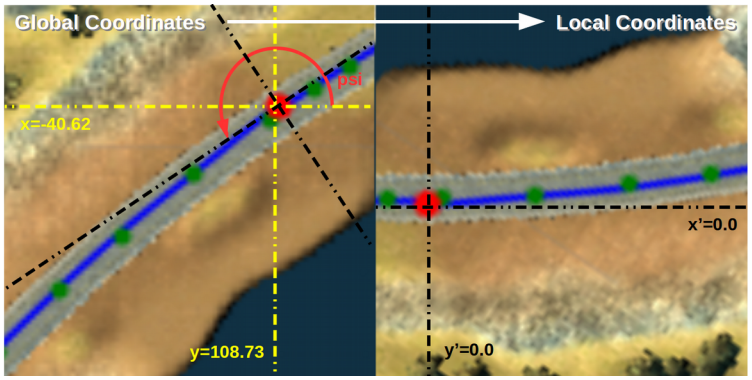

Figure 8: Map Rotation Around a vehicle based on its heading

We will show you how we obtain the map of test track in the *Implementation* section.
Using the newly calculated *waypoints* in local vehicle coordinates, we use **`numpy's polyfit`** function to fit a 3rd degree polynomial to the *waypoints* and then set **`x'`** to 0 to get curve's value in **`y'`**, the CTE.

```
# calculate cross track error (cte)
poly = np.polyfit(np.array(vptsx), np.array(vptsy), 3)
polynomial = np.poly1d(poly)
self.cte = polynomial([0.])[0]
self.acte += self.cte
```

Implementation

As discussed before, the Udacity simulator for project 3, when in autonomous mode, will send the vehicles front camera images, throttle/brake combination, and steering angle, but lacks the *waypoints*, as well as vehicle orientation, for doing the CTE calculations, but we found that for term 2 project 5, Model Predictive Control (MPC), the MPC simulator did have these features, but lack the front camera images that we needed for training the models. But, since the Udacity simulator is open sourced, we can fork a copy and make the modification necessary to include the front camera image. This modification is detailed here: https://github.com/diyjac/self-driving-car-sim/commit/6d047cace632f4ae22681830e9f6163699f8a095, and is available in the github: https://github.com/diyjac/self-driving-car-sim. I have include the binary executables at the following google drive shared link for your evaluation enjoyment:

1. Windows: https://drive.google.com/file/d/0B8uuDpajc_uGVVRYczFyQnE3Rkk/view?usp=sharing
2. MacOS: https://drive.google.com/file/d/0B8uuDpajc_uGTnJvV0RXbjRMcDA/view?usp=sharing
3. Linux: https://drive.google.com/file/d/0B8uuDpajc_uGTld2LXl3VFYzZHc/view?usp=sharing

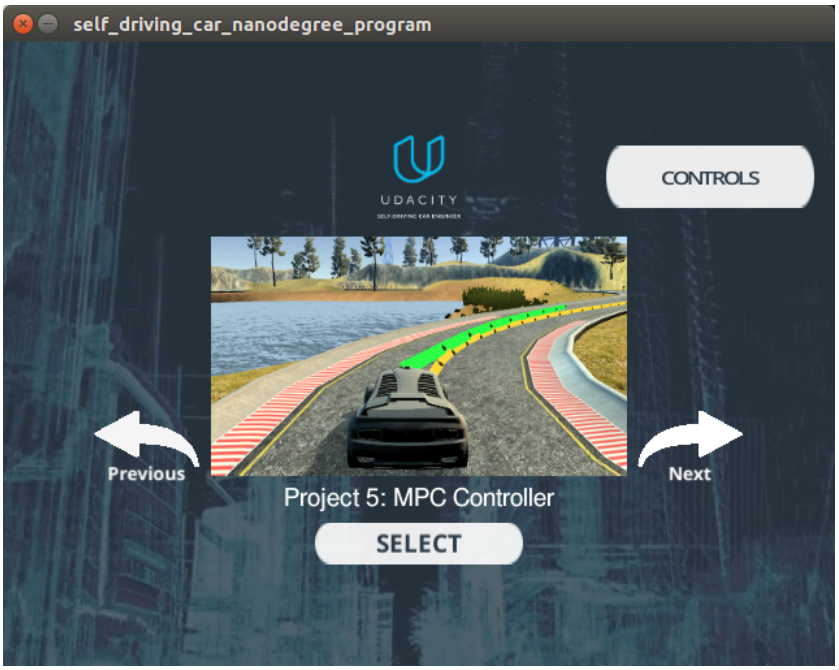

Figure 9: Enchanced MPC Controller Entry Screen

So, with the updated Udacity term 2 simulator, we can now obtain the following additional raw data at each sampling point:

Features	Description	Type	Units	Data Sample
`x, y`	Vehicle global position **CTE Calculation**	Float, Float	X,Y Cartesian coordinates ranging from -200 to 200 in meters.	`-40.62, 108.73` ***(starting position)***
`θ (psi)`	Vehicle global yaw heading **CTE Calculation**	Float	Radians from 0 to 2*pi, where 0 points to the positive x axis.	`3.733651`

Features	Description	Type	Units	Data Sample

`ptsx, ptsy`	5 nearest *waypoints* in global coordinates ahead of the vehicle
	
	CTE Calculation
		Array of 6 Floats, Array of 6 Floats	2 Arrays of X, Y Cartesian coordinates ranging from -200 to 200 in meters.	`Ptsx = [-32.16173, -43.49173, -61.09, -78.29172, -93.05002, -107.7717]`
				
				`ptsy = [113.361, 105.941, 92.88499, 78.73102, 65.34102, 50.57938]`

The MPC project also includes a CSV file **`lake_track_waypoints.csv`** that has all of the *waypoints* in the environment, so we can plot and visually see where the *waypoints* have been placed on the test track. According to the CSV file, there are around 70 waypoints, and their plot is shown in *Figure 10*:

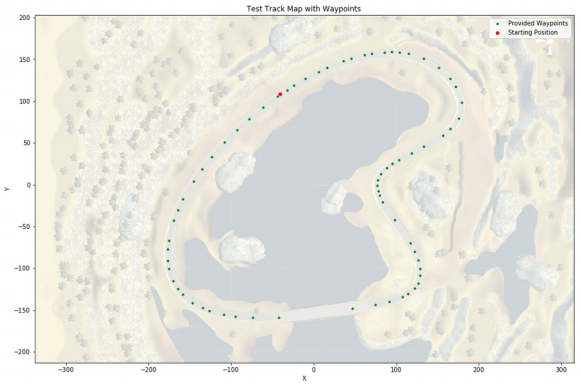

Figure 10: Waypoint locations on test track

We obtain the map by rotating the camera in the Unity Editor in the simulator scene to point straight down to the ground and then taking a snapshot of the image which is something like a satellite view in *Google Earth*.

The **red dot** in Figure 9 shows where the vehicle is located at the start of each training and test sessions. When we ran multiple lap tests around the test track in the simulator using the generation 0 model and collected the X, Y coordinates at each time step, we were able to plot them out and produce what you see in *Figure 11*. As can be seen, the generation 0 model pretty much hits all of the *waypoints* within approximately half a meter from center. The accumulated CTE for the generation 0 model was 458.3522177 meters after 1235 timesteps that completes a lap around the track. It appears the pre-trained model favors being to the left of center.

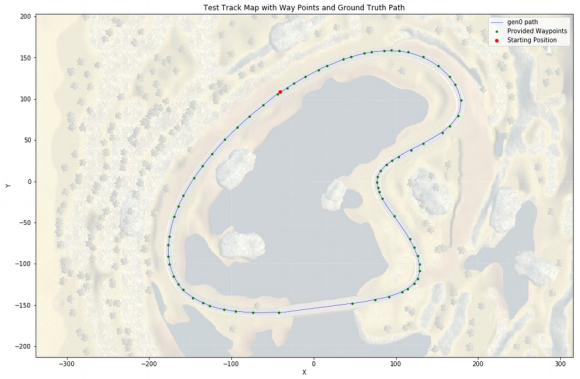

Figure 11: Generation 0 Ground Truth Path

As discussed earlier, the test track in the simulator is approximately 4 meters wide, so any single CTE value greater than 2 meters off center will be considered off track and a failure by the trainer during testing sessions.

The trainer itself is implemented in **`Python 3.6, Keras 2.0.5 and Tensorflow 1.2`** as a command line interface software pipeline embedded in an event loop. Raw data from the simulator is obtain via a websocket interface and sent through the pipeline for processing depending on its current mode. The overall interface has already discussed in the *Algorithms and Technique* section and you may review their architecture in *Figure 5* and *6*. The goal of the design of the architecture is to makes it easier and faster to train next generation models using already trained current generation models. The internal architecture of the system is shown in *Figure 12* and explained briefly below

1. **Simenv**: Python class that handles websocket events from the Udacity simulator, marshals requests to the pipeline, and sends steering, throttle and reset commands back to the simulator.
2. **Trainer**: Python class that handles loading and prediction requests to the trainer model.
3. **Agent**: Python class that handles creating, training and prediction requests to the target model. The Agent class also creates the Recall class for storing the training set.

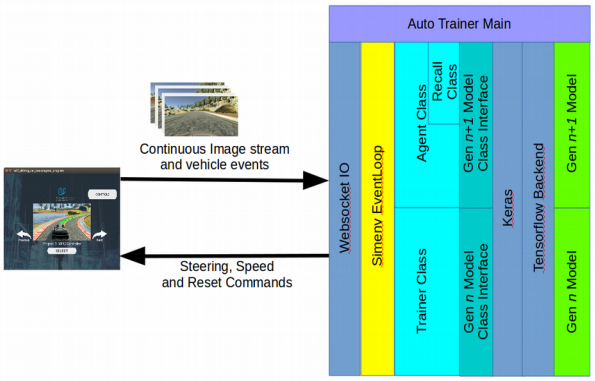

Figure 12: Auto Trainer Software Architecture

4. **Recall**: Python class that handles storing 500 images and baseline steering angles for training of the target model. The recall memory is a FIFO (first in first out) and has a half life of a training session.
5. **Model**: Generic Python class interface that encapsulates both the trainer and the target models so they can be used interchangeably. It also implements a logging function that logs the progress of the target model into a CSV file that can be used by the `training_visualization.ipynb` Jupyter notebook to visualize in real-time during training.
6. **Auto Trainer Main**: Main Python CLI component that handles trainer/target model selection, speed setting and recall memory size.
7. **Websockets**, **Keras** and **Tensorflow**: 3rd party Python modules used by the trainer.

Refinement

When we ran through the initial implementation, the initial results were not very positive. Our first generation results were good, but acceptable third generation candidates were getting scarce. It seems that as the models were learning to better steer in the simulator, it was getting harder and harder to train next generation models at the same or lower ACTE. In some cases, we were not able train a viable target model for the next generation. A lot of the issues seems to be that the target models were not given enough training session to learn to get through the change in texture of the bridge or around some of the sharper bends in the test track as can be seen in *Figure 13* for g=2. On top of this, it was taking around an hour just to go through the automated training per generation with a GPU! To resolve these issue, we increase the number of training and testing sessions to 40 and got a boost in successful results. In some cases, we still have a

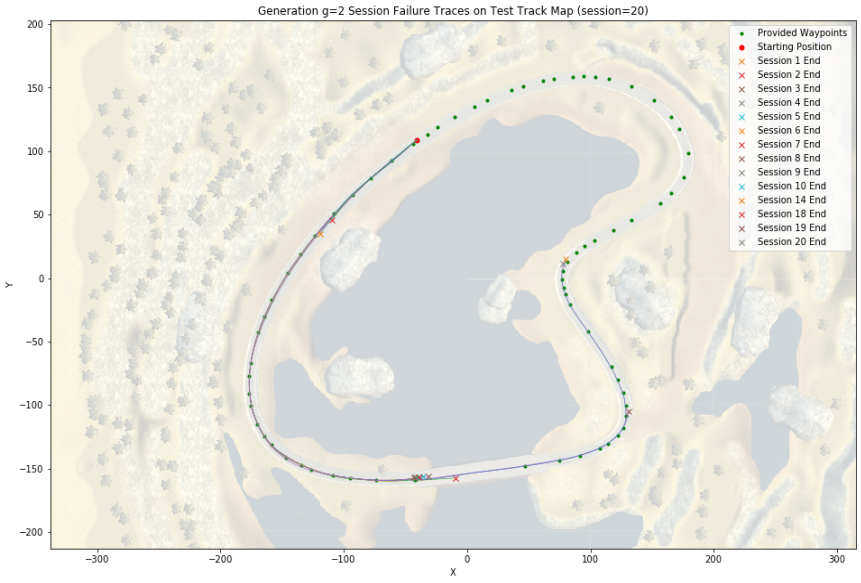

Figure 13: Target Model Fails to Stay on the Test Track at Session=20

large amount of failures, but the increase in successes made the technique viable. Looking at *Figure 14*, show that for the second generation we added an additional 16 viable candidates by increasing the training and testing session from 20 to 40. In addition, in our testing, we were able to get at least 8 model instances per generation trained by this technique to successfully drive autonomously around the test track during the test session. Some of them actually scored better ACTE than the initial generation 0 model! On top of that, we were able to choose a better candidate for the next generation to train even better models. We will go through the results in detail in the *Model Evalution and Validation* section. We even created a **`testdrive.py`** python test script to independently verify these autonomous batch training and testing sessions. After this, we decided to add two additional models to test if the technique will work with non-NVIDIA models.

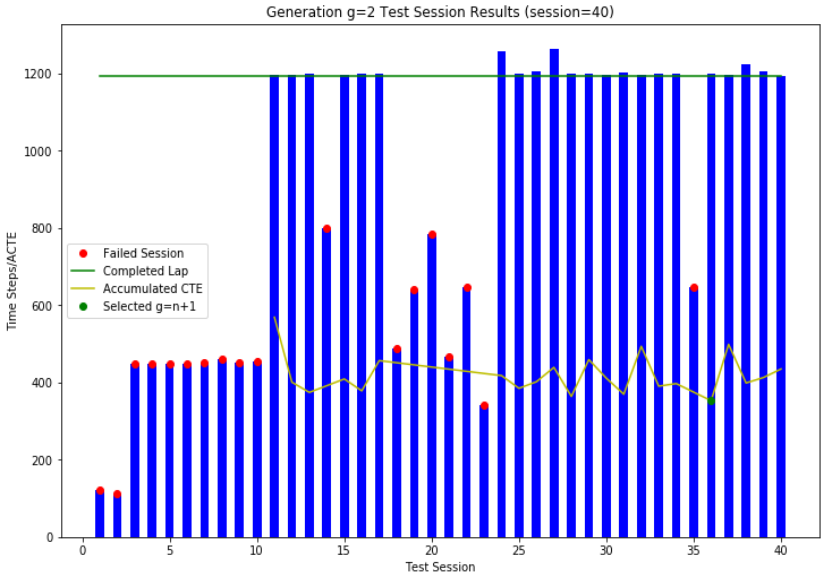

Figure 14: Number of Successes Increased when session was set to 40

Model	Description	g
Custom 1	Custom 32x32 model – no weights	4
Custom 2	Custom 16x16 model – no weights	5

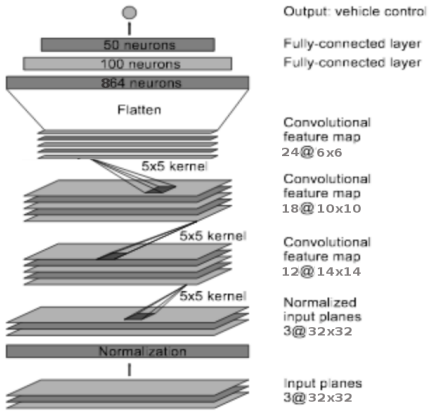

Figure 15: Custom 1 Model

In addition, we folded steps 5 (training) and 6 (testing) from the *Problem Statement* section into the automated training system to make it easier to evaluate new models for the task and reduce the wait time to get the resuls. This is important to free model designers from the drudgeries of manual testing to focus more on new model designs.

We also found that speed of the training machine does have an impact to the resulting model. If anyone wants to use the models that is generated by our system on a non-GPU CUDA enabled system, they should use the `--speed` option to slow down the vehicle to 15 Mph so that the CPU only tensorflow can predict the steering angle fast enough to keep the vehicle on the test track using the **`testdrive.py`** python test script.

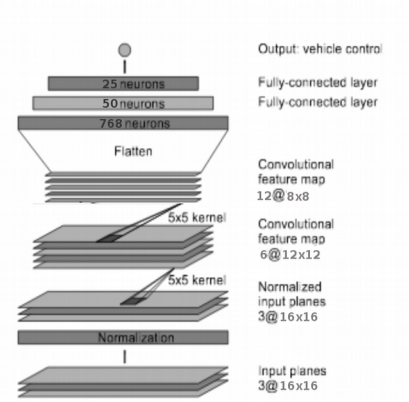

Figure 16: Custom 2 Model

Results

Model Evaluation and Validation

Using this technique we were able to automate the training of 5 target models successively, two of which were of different architectures as mentioned in the *Refinement* section. Our final evaluation of this technique was to be able to use the trained models and drive them autonomously on the test track, and we were able to verify that using the **`testdrive.py`** python test script. We provided an example training session on YouTube that you may access using this URL: https://www.youtube.com/watch?v=4f21TkZquqo&t=882s. While the original goal was to run only 20 training and testing iterations, we found that 40 iterations were better for our purpose, since it gives use more candidates from which to chose our next trainer as can be seen in *Figure 17*. Most trained models have steering MSE of 0.001 or less as the result of the training as can be seen in *Figure 18*.

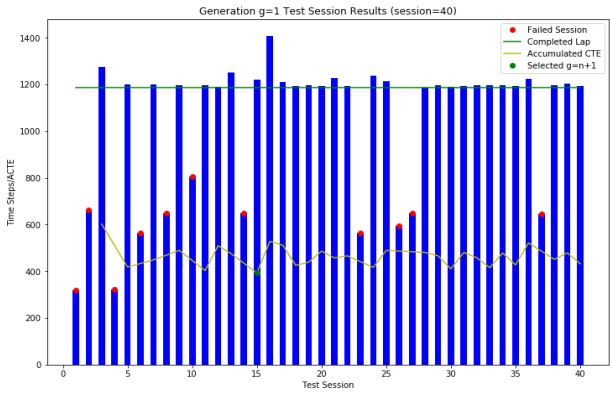

Figure 17: Generation 1 Test Session Results

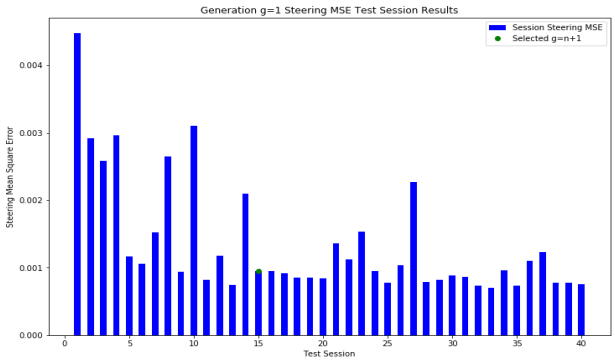

Figure 18: Generation 1 Mean Square Error

We also tried to change the cruse control speed on the car from 30Mph to 40Mph see how well the models performed, and surprisingly it worked! Especially since the generation 0 model was trained against Udacity Project 3 simulator that was limited to just 30Mph! After a few tries, we were able to successfully train the target model to drive at 40Mph. You may watch the speed test training session using this URL: https://www.youtube.com/watch?v=ROu111VSG_c.

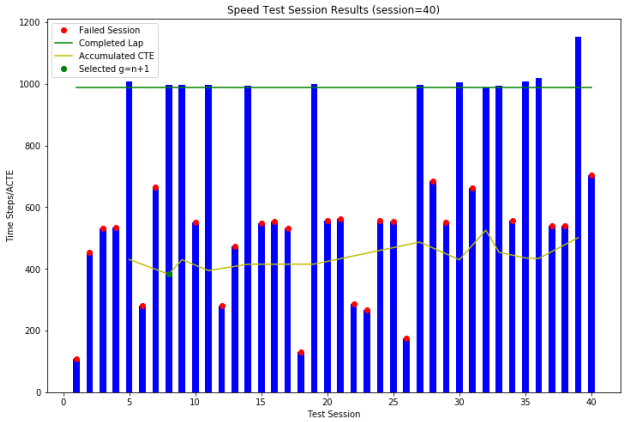

Figure 20: Speed Training Sessions Results

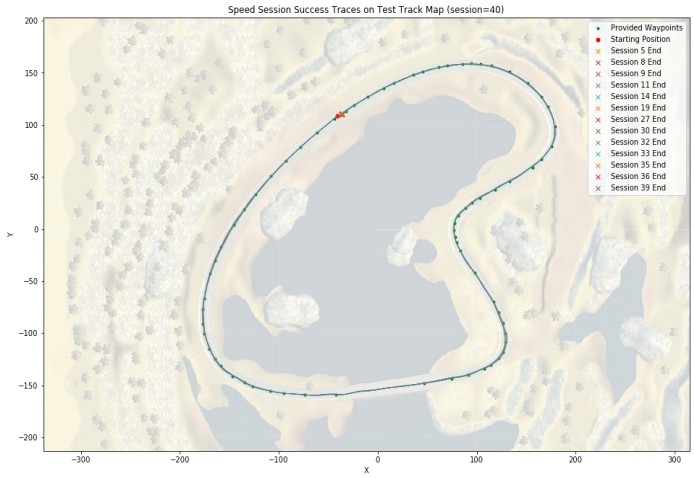

Figure 19: Target at 40Mph Speed

Justification

After countless hours of training and testing sessions, the table below shows the resulting ACTE statistics achieved with this CNN training method for test sessions that were able to drive around the test track successfully and autonomously. Since the accumulated cross track error (ACTE) accounts for how far the vehicle is deviating from the center of the road over time, a smaller value is more desirable to replicate than larger values. This was the reasoning to select the session that was able to achieve the minimum ACTE to train the next model generation; and the effect can be seen in the table as the mean and the max both falls from generation to generation, even if the model architecture are dissimilar as in the case of generation 4 and 5 as discussed in the *Refinement* section. We calculate the improvement from the ACTE Mean of the generation as compared to its previous generation using this formula:

$$Improvement_{ACTE} = \frac{(ACTE_{g-1} - ACTE_{g})}{ACTE_{g-1}}$$

	g=0	*g=1*	*g=2*	*g=3*	*g=4*	*g=5*	*speed*
Successful Sessions	N/A	29	22	10	11	10	13
ACTE Mean	458.35	464.03	418.87	390.23	388.36	369.07	441.14
ACTE STD	N/A	45.15	51.33	38.59	14.86	28.70	41.22
ACTE Max	458.35	601.88	568.95	489.32	411.58	406.13	525.65
ACTE Min	458.35	392.79	352.83	359.89	358.35	329.07	382.77
Improvement	N/A	-1.24%	9.73%	6.84%	0.48%	4.97%	3.75%

Since the mean ACTE either stayed the same or dropped, except for g=1, we believe that we were able to achieve the results of the goal especially since the real proof is if the model was able to drive the vehicle on the test track in the simulator and we believe we proofed that as well in the two YouTube video links provided in the *Model Evaluation and Validation* section.

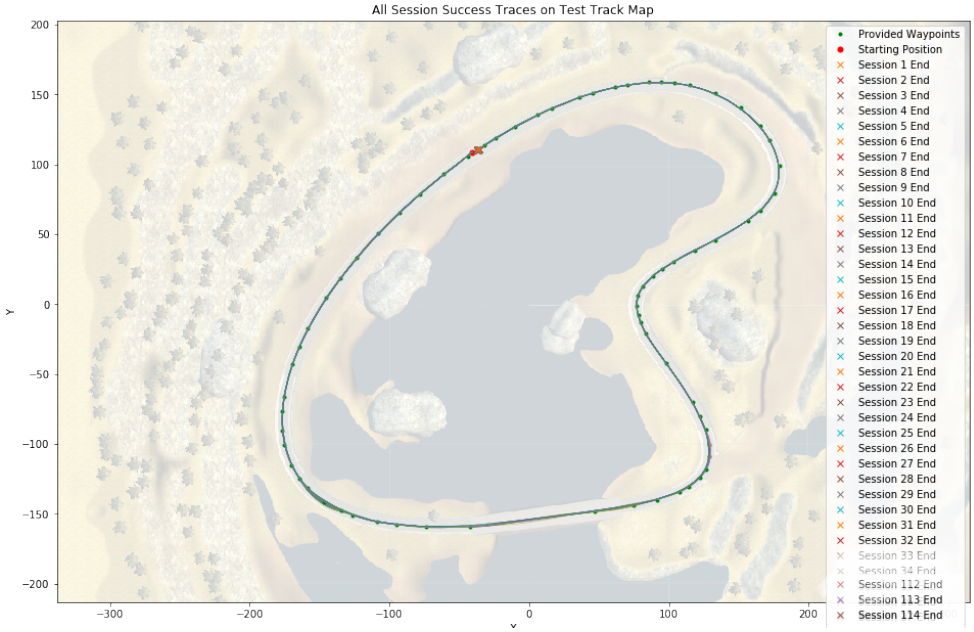

Figure 21: All Successful Sessions Traces on Test Track Map

Conclusion

Free-Form Visualization

One aspect of this project that we thoroughly enjoyed was watching the target model training progress as it goes off track from one location, learns to go further and then goes off track again in a new location. Since the target model will log its progress in real-time in a CSV file, we created a Jupyter notebook that can read this CSV and construct the target model current learning progress. It was the first time we were able to see a neural network learning in progress.

For instance, in *Figure 22* the target model is showing its learning in progress as it discovers and learn how to steer to stay on course during an earlier speed test. All of the colorful x marks locations where it failed in a test session, but it progress repeatedly and goes further and further into the lap around the test track in the simulator. The first couple of sessions, it could not drive more than a few waypoints away. Then it was able to get passed that and get stuck repeatedly at the beginning of the bridge. The next major clustering of failures happens at the first sharp curve after the bridge and then again when it had to learn to make its first right turn. When the speed was normal, the target model usually do not have any difficulties after this obstacle; however, with the added speed and latency associated with it, it was hard for it to make it completely around the next sharp left turn. Finally it was able to find a solution in session 40 as shown in *Figure 23*.

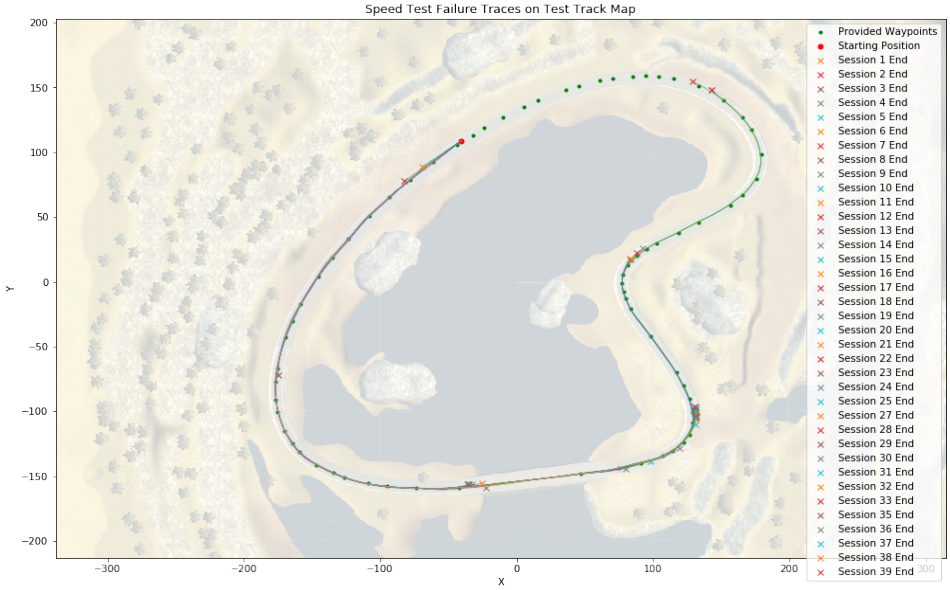

Figure 22: Earlier Test Run with Target Model Learning in Progress.

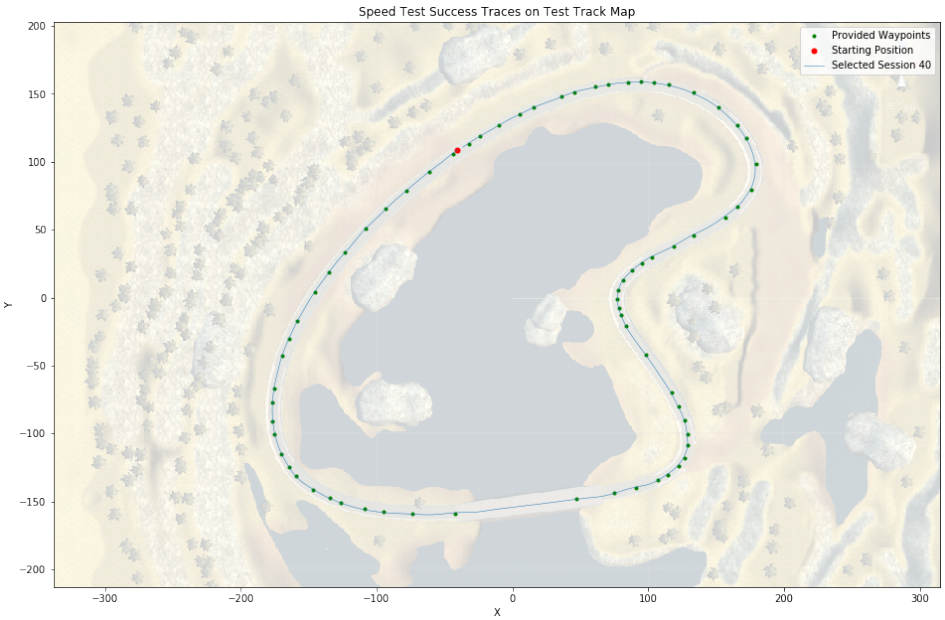

Figure 23: Earlier Test Run with Target Model Finding Solution

Reflection

The goal of this project was to design an automated system to iterate the model design, data gathering and training in a tighter automated loop where successive improved models can be used to train even more advance models. We designed and built a system to perform the automated training and testing loop by using python 3, Keras, Tensorflow with NVIDIA CUDA GPU acceleration. But, even with the automated system and CUDA acceleration, the amount of time to train the network model does not disappear. It still takes around 2 hours for the trainer to go through 40 sessions (~90 seconds per lap around the test track) of training and testing each generation, so may not be a complete solution for automation. If you do not have GPU, this solution is not for you. The simulator needs real-time actuation and if steering and throttle commands are not given at the correct time interval, the simulated vehicle will go off track and crash into the lake or go off into the woods. Also, as the speed tests proves, using this technique to train a target model that the trainer model was never trained for could be done if the trainer model is robust enough and generalize well.

Improvement

There are many topics to study further in improving this automated system that we do not have time for now that the Capstone project is due. A few are listed below.

Increase Speed Test to 50Mph: Test 50Mph cruse control speed using the 40Mph target models as trainers. This will test if the 30Mph to 40Mph improvement was limited to just the generation 0 trainer.

Implement the solution as an AWS cloud service: David Silver and I spoke recently about this for training a new set of autonomous vehicle agents in a simulated multi-vehicle simulation. This will allow more people to access this new technique and build autonomous vehicle agents more easily.

Additional model architecture should be tested: We only added two custom models into the automated system, additional models can test how well the trainer trains and show any deficiencies.

Reenforcement Learning: Use this as starting point to build a Reenforcement learning agent using a DeepQ Learner: https://www.cs.toronto.edu/~vmnih/docs/dqn.pdf. The current project is more an automated behavior cloning with some master/apprenticeship aspect; where as, Deep Q Reenforcement learning would just feed the a model with the simulation images and a reward function and let the model learn how to drive on its own.

Implement in a RoboCar: RoboCars are autonomous vehicles the size of a remote control cars. Usually powered by Raspberry Pi and camera, they can run tensorflow as inference engines to predict steering angles given a pre-trained model. Why not try to use telemetry like the simulator and have the trainer train the model for the RoboCar in real-time? https://medium.com/self-driving-cars/diy-robocar-meetup-8112f5f8b23c.