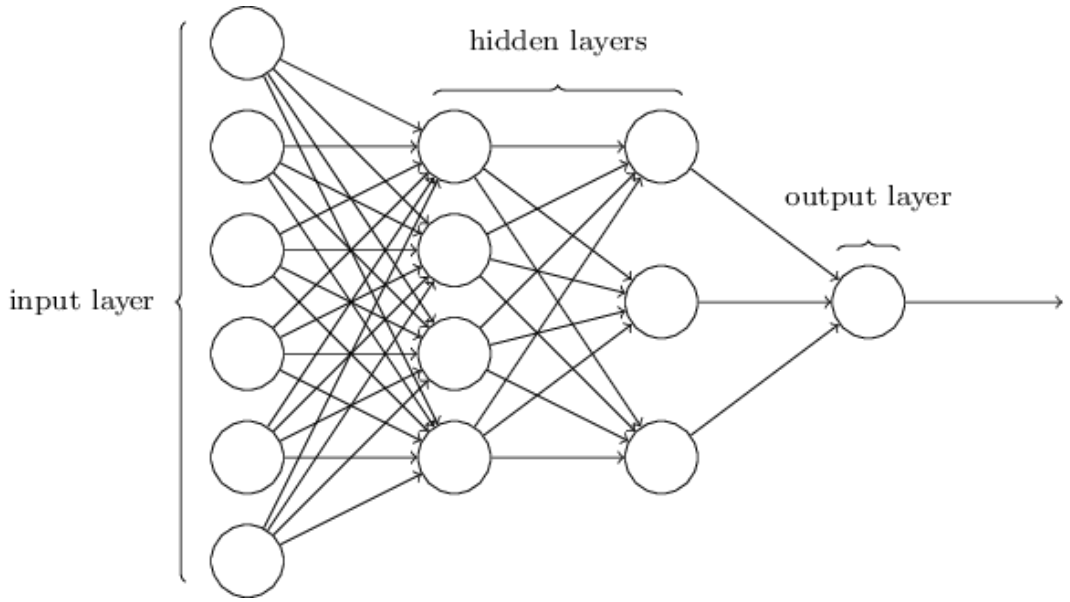
The Suitability of Behaviour Cloning in Autonomous Vehicles

The topic I'm researching is whether engineers should use convolutional neural networks to clone human behaviour to power self driving cars, or use a mixture of hard code and computer visions. The introduction video *Self-Driving Car Nanodegree Program Overview*(Udacity, n.d.) provided me with general description of how each method works. However, this source did not provide any details on how to build any of the systems, neither the advantages of them all.

Background Knowledge

As this project revolves around behavioral cloning, there are some background knowledge to understand before one could begin to understand this program. The normal way computer science works is you give a computer an algorithm an input, and it spits out an output. But as one could imagine, writing a program to mimic human behaviour given an image is not easy, so instead another process is used here called supervised machine learning. You give a computer a large number of inputs (called features) and a large of outputs (called labels) and the computer uses regression to find a path from the input to the output so that when given an unlabeled input, it could find the output.

A normal neural network works by connecting breaking down the input parameters into nodes, and connecting them to several hidden nodes organised into layers, then connecting the hidden layers to the output nodes, which gives the output.

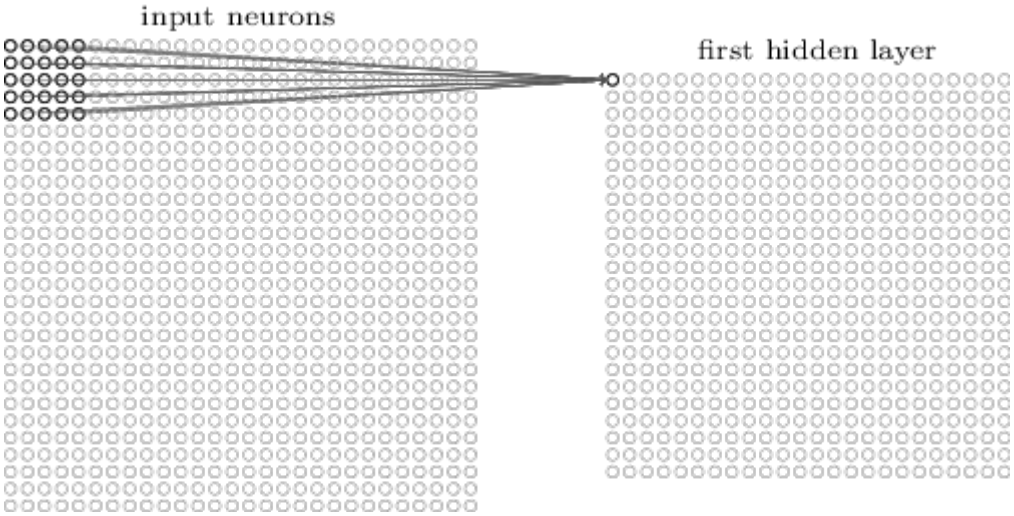


(diagram from Neural Network and Deep Learning)

As shown by the diagram above, every node is connected to every node in the next layer, and each connection is called a neuron. Each neuron could be expressed in the form of f(x)=mx+c, m is called the weight of the neuron, and c is called the bias. By adjusting the weight and bias of all the neurons in a neural network through a process called gradient descent, one could approach the labelled output. Then when the network encounters an input that it didn't see before, it could calculate the output with desired reasonable accuracy.

To mimic a drivers behaviour when driving, one needs to design a network that takes an input from cameras mounted on the car, and predict an output of steering angle. One could connect each pixel as an input node, however as an image has 3 dimensions (x,y,rgb), doing this will destroy some spacial information about the image as the input nodes is essentially a list of the pixels.

A better way of extracting output from an image is called a convolutional neural network, it uses similar concepts to a normal neural network, however it preseves the spacial dimention of an image.



(diagram from Neural Network and Deep Learning)

As shown in the diagram above, a convolutional neural network uses a local receltive field of specified square dimensions (e.g. 5*5) and to scan across an image and produces a new hidden layer. Every complete scan is called a filter. This process could be repeated for a specified amount of time and the values are all stored in a new 2 dimensional array in the next layer. So the dimension of one layer of nodes would be x*y*number of filters.

The same process is repeated from the 1st hidden layer to the second and so on. Each time loosing 4 pixels from the x and 4 pixels from the y direction. To reduce the amount of

computation needed and extract bolder features from the image, sometimes the layers are downsampled by combining several pixels to one.

The whole process is then repeated until the x and y direction are both 1, so the dimension of this layer would be a 1 dimentional array of filters which is then connected to a normal neural network to give the output.

As one could see from the description above, the whole process has more parameters to tweek than comventional programs including but not limited: the dimension of the network itself, the learning rate of the network, and choosing the way the network evaluates success (the cost function).

This EPQ would be about how to choose these parameters as to make the steering angle output match what a human would decide to do, and analysing whether a similar system using end to end (straight from input image to output steering angle with no hard code algorithm in between) machine learning could realistically be used in autonomous driving and the benefits and drawbacks of this system vs one that uses hard code and sensors to decide the action of the vehicle.

History

Many see the second DARPA grand challenge as the breaking point for autonomous vehicles as it was the first time a vehicle successfullly navigated through a desert without human intervention. The winning team(The Stanford Racing team) used computer vision to map terrain and use path planning algorithms to guide the car in the desired direction according to the official Stanford team website(“Stanford Racing :: Home,” n.d.) and *Using CART to segment road images*(Davies and Lienhart, 2006).

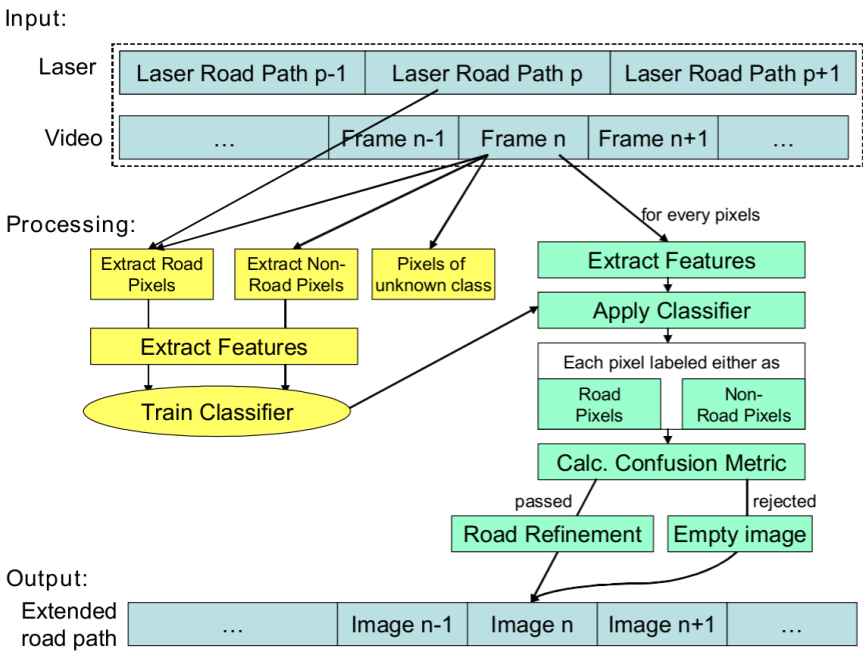


Diagram from paper mentioned above.

The paper provided an unbiased review of the technology used including both advantages and disadvantages, and even included links to the actual source code used. However, the method used is way too complicated for this project.

The winner of the DARPA urban challenge a few years later which focused on autonomous vehicles in an urban setting was from Carnegie Mellon University, the team used a preprogramed multi-level decision tree to respond to the terrain mapped by various sensors with little machine learning involved according to their website ("Tartan Racing @ Carnegie Mellon," n.d.) and *A multi-modal approach to the DARPA Urban Challenge* (Urmson et al., 2007).

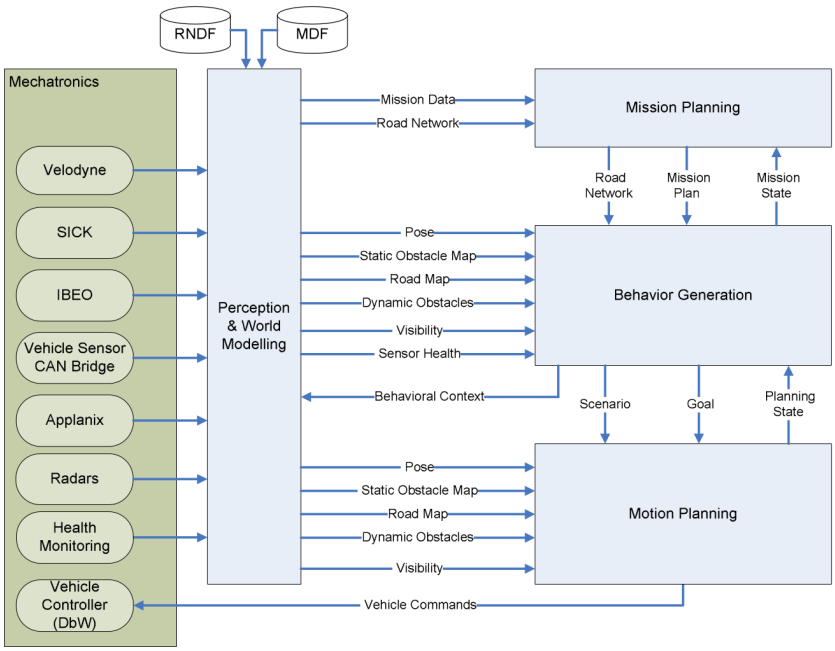


Diagram from paper mentioned above.

The source again was too advanced to be used in this project.

In 2010, Vislab from the University of Parma attempted a 13000 km drive from italy to China . According to this article at spectrum.ieee.org(Guizzo, 2010), the team used one van to navigate and map the landscape with human intervention, and another van that followed the first. The paper *The VisLab Intercontinental Autonomous Challenge: an extensive test for a platoon of intelligent vehicles* (Bertozzi et al., n.d.) and *The VisLab Intercontinental Autonomous Challenge: 13,000 km, 3 months,… no driver (Broggi et al., 2011)* confirmed the idea.

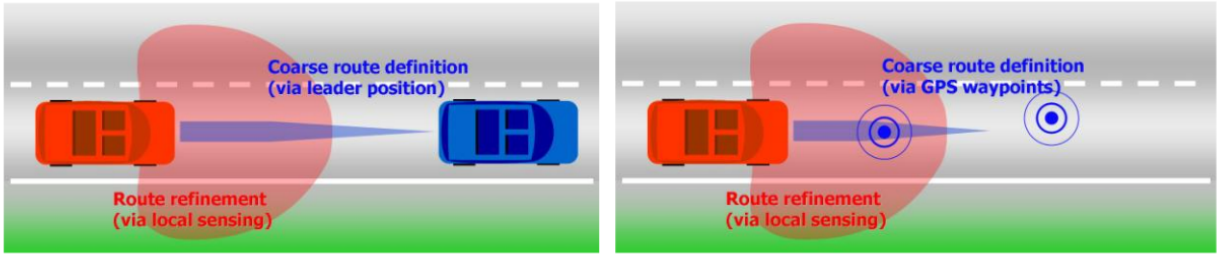


Figure 6: Left: leader follower approach when the leader is in line of sight; Right: leader follower approach when the leader is not visible by the follower

Diagram from paper mentioned above.

Although method used is indeed suitable for unmapped roads, it did not fit with the goal of this project.

Stanford's dynamic design labs has been participating in the development of autonomous vihecles, and in 2012, they have developed a car is designed to run at speeds of 100+ miles per hour. The specifics of the technology was not able to be found with reasonable effort, however, judging by the research done at stanford including *(*"Safe driving envelopes for path tracking in autonomous vehicles | Dynamic Design Lab," n.d.) and ("Motion Planning for Autonomous Vehicles | Dynamic Design Lab," n.d.), the team uses sensor fusion path tracking to plan the path of the car algorithmically.

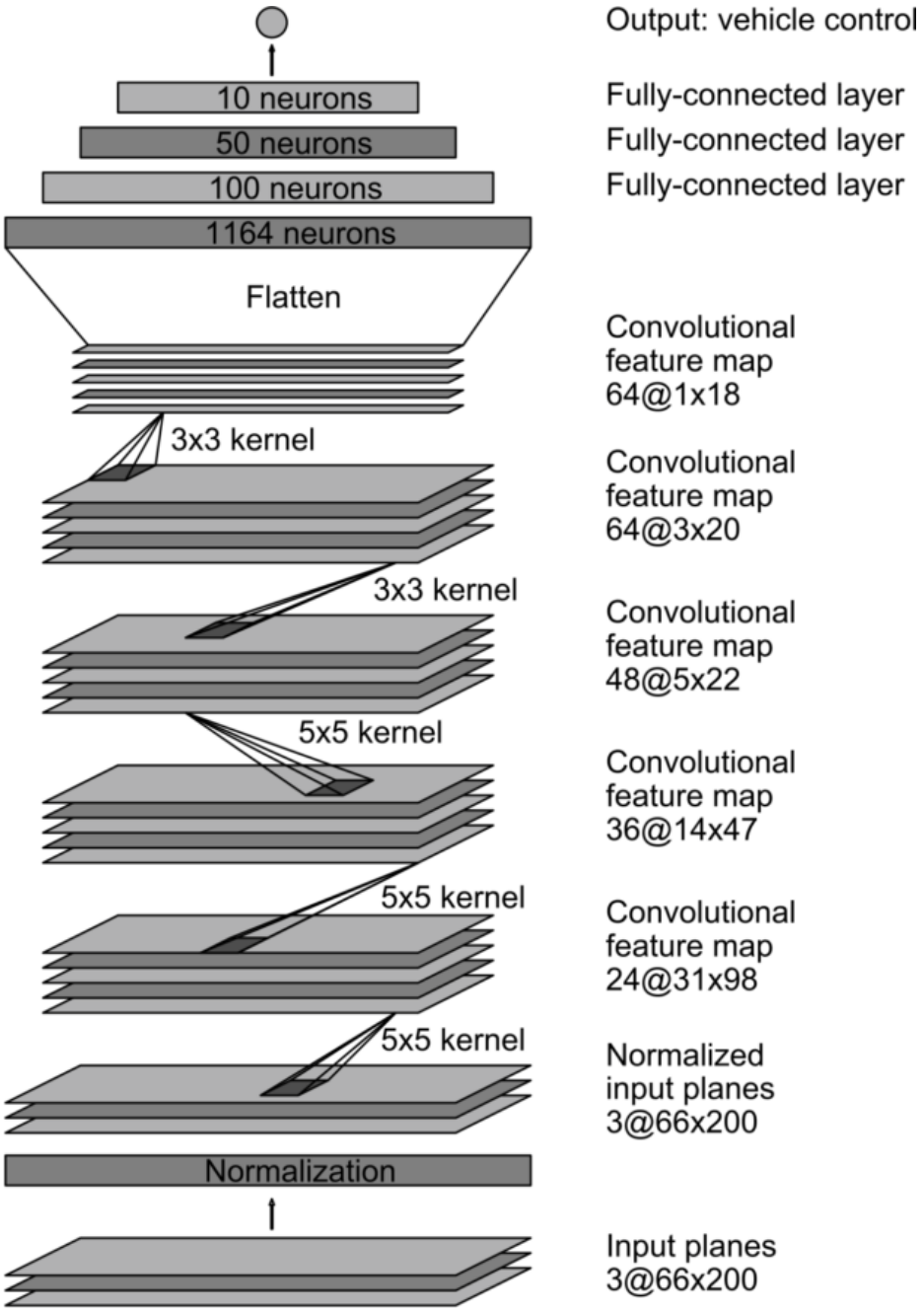
Carnecie Mellon is also an active participant in the field with a fleet of testing vehicles running software that does object recognition and trajectory planning modified from the source used in the DARPA challenges. This system does actually try to mimic human behaviour when driving according to *Towards Fully Autonomous Driving: Systems and Algorithms (Levinson et al., 2011)* , however it differs from the method investigated in this project in which the machine learning process happens end to end.

Current Technology

It is also worth noting that companies such as Tesla and Uber are actively trying to develop fully autonomous vehicles, with the most advanced by far being Google's Waymo(formerly the Google Self-drivng Car Project). In May 2018, Waymo began an a public trial of it's driverless car sharing program which is fully autonomous("Early Rider Program," n.d.) having driven 13,000,000 km already. It is hard to find information on the exact source code used by companies such as Waymo, however according to videos released by waymo(Waymo, n.d.), a self an self-driving car engineer on Quora("What are some of the technologies used in the Google self-driving car? - Quora," n.d.) and a TED talk by one of the Google Self-driving car members(Urmson et al., 2015), they use similar techniques such as object recognition and

path planning mentioned above, the difference is that Google has a more servers optimized for machine learning, therefore can process data faster and produce a better result.

The computer hardware manufacturer Nvidia is also interested in autonomous driving technology and is a supplier for the uber team. They have investigated the possibility of using a convolutional neural network to read images from a camera and learn from a human the reaction to take with reasonable success, a research blog by Nvidia details the technology used and their development process (“End-to-End Deep Learning for Self-Driving Cars,” 2016), as the article goes in depth into the how the network comes together and the logic in each step, and that the method described is exactly the method this project aims to evaluate, this source is very valuable. The exact network structure used is show below:



Practical Information

The open source code provided by Udacity's nanodegree(*CarND-Behavioral-Cloning-P3,* 2018) is very valuable to the development of this project as it provided an easy way to get training data and test algorithms without resorting to using unrealistic games like Grand Theft Auto, or building a physical car and environment before knowing the viability of it all; however I did not enroll in the actual program("Self Driving Car Engineer Nanodegree | Udacity," 2018) as it was considered costly both in terms of time, and finance.

Stanford's online course cs231n ("CS231n," 2018) is another vital resource for this project as it helped familiarise me with basic python commands, and explained in detail how a convolutional neural network works at a level that is easily understood, and pointed me in the right direction for about similar topics.

Google's machine learning crash course("Machine Learning Crash Course | Google Developers," 2018) was the resource I settled on after bouncing between courses on Udacity and several other sites. I find this source particularlly useful as it is very beginner friendly, and explains machine learning concepts that are hard to understand with careful descriptions and interactive diagrams. It also assumes zero prior knowledge of the topic so iteractive sessions actually help me familiarise with python's datastructure, which is necessary knowledge for building a machine learning project as to maximise the efficiency of the machine.

The guide from Tensorflow on building an iris classification program("Premade Estimators," n.d.) was helpful as it provided a clear run down of the steps used to build a machine learning model using Tensorflow. It also cleared my confusion of several terminology such as "label" and "feature" which is necessary for understanding some of the sources listed above. The list at the Machine Learning Glossary("Machine Learning Glossary | Google Developers," 2018) is helpful in the same way as it cleared my confusion on terms such as "supervised" and "unsupervised".

The book *Neural Networks and Deep Learning(Nielsen, 2015)* provided a great source of knowledge for understanding the underlying mechanisms of machine learning, as well as explaining what the parameters in the code means. It is also more theoretical than the other sources listed therefore more general and versatile. This gave me the ability to understand the working of the wole system better, to judge what settings to use in my own implementation of the program, and gave me the ability to tweak theses same parameters later if my code does not work the first time it compiles. For example, it explained the concept of backpropagation better than the courses above could.

The blog post *An augmentation based deep neural network approach to learn human driving behavior (Yadav, 2016)* provides a detailed description of how to work with the specific task of teaching a car to lane-keep. It provides diagrams and suggestions on the tecniques used to

generate training images including shifting, reflecting, and randomising the images as well as changing the brightness of the image, and gave a model of what the author's best attempt at such an neural network. It also suggest future investigations to be done after this task.

The github repository of user subodh-malgonde(Malgonde, 2018) provides an example of developers attempting to do a similar project as this one. The author documented some of the techniques used, many of which is similar to the ones mentioned in the previous blog post, and an example of finished code which may come in useful when this project comes to the testing phase for evaluation. However, the code itself is less well commented and documented therefore harder to understand.

The github repository of user Naokishibuya(Shibuya, 2018) also shows another developer attempting to use a convolutional neural network to train a car to drive around a track. This source is better than the previous ones in my opinion as it not only provides detailed description of the techniques used to generate additional training images, but a detailed explanation of the model used, as well as how each parameter is tweaked and why he tweaked them. These information may come in useful when doing the investigation later as it is a great source of knowledge for how to debug my own code. This source also provides an example of a trained model so it could potentially be used to evalute my own code later in the process. The exact network used by Naokishibuya is shown below:

Layer (type)	Output Shape	Params	Connected to
lambda_1 (Lambda)	(None, 66, 200, 3)	0	lambda_input_1
convolution2d_1 (Convolution2D)	(None, 31, 98, 24)	1824	lambda_1
convolution2d_2 (Convolution2D)	(None, 14, 47, 36)	21636	convolution2d_1
convolution2d_3 (Convolution2D)	(None, 5, 22, 48)	43248	convolution2d_2
convolution2d_4 (Convolution2D)	(None, 3, 20, 64)	27712	convolution2d_3
convolution2d_5 (Convolution2D)	(None, 1, 18, 64)	36928	convolution2d_4
dropout_1 (Dropout)	(None, 1, 18, 64)	0	convolution2d_5
flatten_1 (Flatten)	(None, 1152)	0	dropout_1
dense_1 (Dense)	(None, 100)	115300	flatten_1
dense_2 (Dense)	(None, 50)	5050	dense_1

Layer (type)	Output Shape	Params	Connected to
dense_3 (Dense)	(None, 10)	510	dense_2
dense_4 (Dense)	(None, 1)	11	dense_3
	Total params	252219	

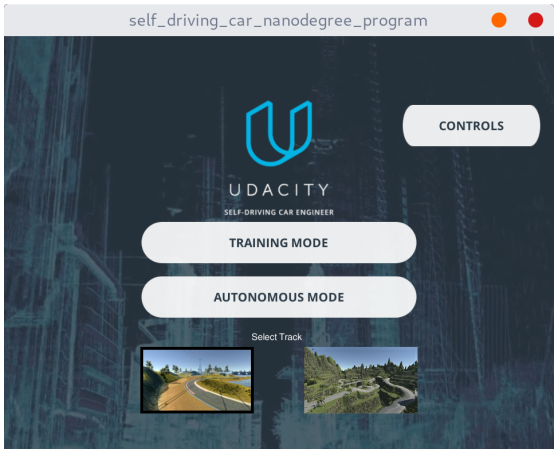
Table from source mentioned above.

I chose to use Tensorflow as the library to provide the codebase necessary for machine learning as it is well documented, open source, and has a community of developers working on similar projects. *Build a Convolutional Neural Network using Estimators* ("Build a Convolutional Neural Network using Estimators | TensorFlow," 2018) is a great source of well explained and well documented code on how to use Tensorflow for Convolutional neural networks such as this one. The code in the examples are also adaptable so may come in handy when I build my own network. However, this example is a classification problem so it may be different from when this project needs to solve which is a regression problem.

Data collection methods

Typically in the field of self-driving cars, early models are either tested with proprietary simulations or a game such as Grand Theft Auto. Both of which are unsuitable for this project as proprietary software is expensive and Grand Theft Auto is not an accurate representation of the driving situation my vehicle is designed for. So instead I used Udacity's self-driving simulator mentioned above.

The software contains two modes, recording and driving, and two tracks to test the cars on,

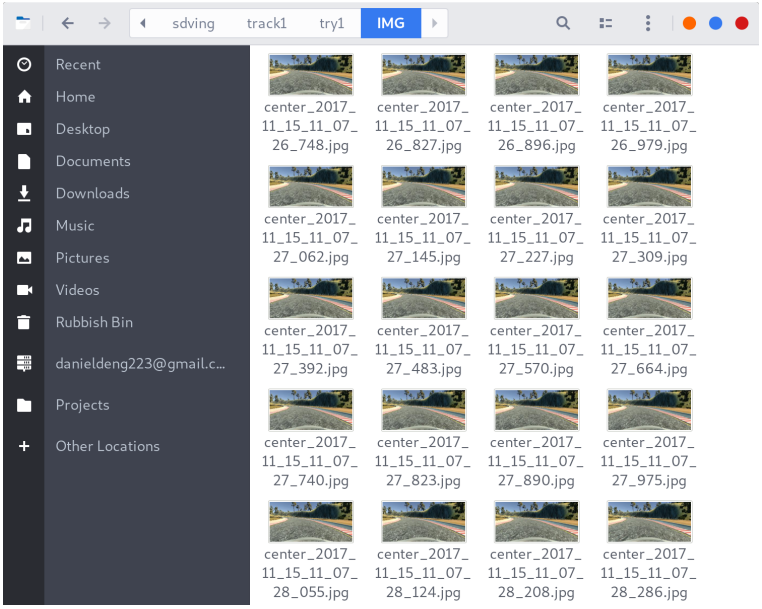


the lake track which is relatively easy, and a harder jungle track that contains hard turns that contains hard left and right turns to confuse the driving software.

Training mode allows the collection of data from a human driver. In this case, I recorded myself driving both tracks 2 cycles each using arrow keys as controls, and repeated the process 5 times. This allowed me to collect in total 20 cycles around the two track.

The program automatically recorded some driving data from the user, stored in a .csv file the user could access. The program records 8 rows per second, each row contains from left to right, the storage locations of three images from the simulated cameras located at the left, center and right side of the car, the steering, the throttle, the break, and the current speed.

/home/daniel/Projects/sdving/track1/try1/IMG/left_2017_11_15_11_07_26_74	/home/daniel/Projects/sdving/track1/try1/IMG/right_2017_11_15_11_07_26_74	0	0	0 8.45582E-06
/home/daniel/Projects/sdving/track1/try1/IMG/left_2017_11_15_11_07_26_82	/home/daniel/Projects/sdving/track1/try1/IMG/right_2017_11_15_11_07_26_82	0	0	0 1.03804E-05
/home/daniel/Projects/sdving/track1/try1/IMG/left_2017_11_15_11_07_26_89	/home/daniel/Projects/sdving/track1/try1/IMG/right_2017_11_15_11_07_26_89	0	0	0 2.30887E-06
/home/daniel/Projects/sdving/track1/try1/IMG/left_2017_11_15_11_07_26_97	/home/daniel/Projects/sdving/track1/try1/IMG/right_2017_11_15_11_07_26_97	0	0	0 1.09118E-05
/home/daniel/Projects/sdving/track1/try1/IMG/left_2017_11_15_11_07_27_06	/home/daniel/Projects/sdving/track1/try1/IMG/right_2017_11_15_11_07_27_06	0	0	0 2.14906E-05
/home/daniel/Projects/sdving/track1/try1/IMG/left_2017_11_15_11_07_27_14	/home/daniel/Projects/sdving/track1/try1/IMG/right_2017_11_15_11_07_27_14	0	0	0 3.28098E-06
/home/daniel/Projects/sdving/track1/try1/IMG/left_2017_11_15_11_07_27_22	/home/daniel/Projects/sdving/track1/try1/IMG/right_2017_11_15_11_07_27_22	0	0	0 1.85604E-05
/home/daniel/Projects/sdving/track1/try1/IMG/left_2017_11_15_11_07_27_30	/home/daniel/Projects/sdving/track1/try1/IMG/right_2017_11_15_11_07_27_30	0	0	0 1.9098E-05
/home/daniel/Projects/sdving/track1/try1/IMG/left_2017_11_15_11_07_27_39	/home/daniel/Projects/sdving/track1/try1/IMG/right_2017_11_15_11_07_27_39	0	0.2477452	0 0.2226586
/home/daniel/Projects/sdving/track1/try1/IMG/left_2017_11_15_11_07_27_48	/home/daniel/Projects/sdving/track1/try1/IMG/right_2017_11_15_11_07_27_48	0	0.4976168	0 0.7050151
/home/daniel/Projects/sdving/track1/try1/IMG/left_2017_11_15_11_07_27_57	/home/daniel/Projects/sdving/track1/try1/IMG/right_2017_11_15_11_07_27_57	0	0.7494385	0 1.409544
/home/daniel/Projects/sdving/track1/try1/IMG/left_2017_11_15_11_07_27_66	/home/daniel/Projects/sdving/track1/try1/IMG/right_2017_11_15_11_07_27_66	0	1	0 2.60636
/home/daniel/Projects/sdving/track1/try1/IMG/left_2017_11_15_11_07_27_74	/home/daniel/Projects/sdving/track1/try1/IMG/right_2017_11_15_11_07_27_74	0	1	0 3.542855
/home/daniel/Projects/sdving/track1/try1/IMG/left_2017_11_15_11_07_27_82	/home/daniel/Projects/sdving/track1/try1/IMG/right_2017_11_15_11_07_27_82	0	1	0 4.481984
/home/daniel/Projects/sdving/track1/try1/IMG/left_2017_11_15_11_07_27_89	/home/daniel/Projects/sdving/track1/try1/IMG/right_2017_11_15_11_07_27_89	0	1	0 5.411661
/home/daniel/Projects/sdving/track1/try1/IMG/left_2017_11_15_11_07_27_97	/home/daniel/Projects/sdving/track1/try1/IMG/right_2017_11_15_11_07_27_97	0	1	0 6.334213
/home/daniel/Projects/sdving/track1/try1/IMG/left_2017_11_15_11_07_28_05	/home/daniel/Projects/sdving/track1/try1/IMG/right_2017_11_15_11_07_28_05	-0.2	1	0 7.240939
/home/daniel/Projects/sdving/track1/try1/IMG/left_2017_11_15_11_07_28_12	/home/daniel/Projects/sdving/track1/try1/IMG/right_2017_11_15_11_07_28_12	-0.35	1	0 7.902247
/home/daniel/Projects/sdving/track1/try1/IMG/left_2017_11_15_11_07_28_20	/home/daniel/Projects/sdving/track1/try1/IMG/right_2017_11_15_11_07_28_20	0	1	0 8.793874
/home/daniel/Projects/sdving/track1/try1/IMG/left_2017_11_15_11_07_28_28	/home/daniel/Projects/sdving/track1/try1/IMG/right_2017_11_15_11_07_28_28	0	1	0 9.689331
/home/daniel/Projects/sdving/track1/try1/IMG/left_2017_11_15_11_07_28_35	/home/daniel/Projects/sdving/track1/try1/IMG/right_2017_11_15_11_07_28_35	-0.2	1	0 10.56859
/home/daniel/Projects/sdving/track1/try1/IMG/left_2017_11_15_11_07_28_44	/home/daniel/Projects/sdving/track1/try1/IMG/right_2017_11_15_11_07_28_44	-0.4	1	0 11.409
/home/daniel/Projects/sdving/track1/try1/IMG/left_2017_11_15_11_07_28_51	/home/daniel/Projects/sdving/track1/try1/IMG/right_2017_11_15_11_07_28_51	0	1	0 12.08296
/home/daniel/Projects/sdving/track1/try1/IMG/left_2017_11_15_11_07_28_60	/home/daniel/Projects/sdving/track1/try1/IMG/right_2017_11_15_11_07_28_60	0	1	0 13.12331
/home/daniel/Projects/sdving/track1/try1/IMG/left_2017_11_15_11_07_28_67	/home/daniel/Projects/sdving/track1/try1/IMG/right_2017_11_15_11_07_28_67	-0.15	1	0 13.85395
/home/daniel/Projects/sdving/track1/try1/IMG/left_2017_11_15_11_07_28_74	/home/daniel/Projects/sdving/track1/try1/IMG/right_2017_11_15_11_07_28_74	0	1	0 14.71409
/home/daniel/Projects/sdving/track1/try1/IMG/left_2017_11_15_11_07_28_82	/home/daniel/Projects/sdving/track1/try1/IMG/right_2017_11_15_11_07_28_82	0	1	0 15.59788
/home/daniel/Projects/sdving/track1/try1/IMG/left_2017_11_15_11_07_28_89	/home/daniel/Projects/sdving/track1/try1/IMG/right_2017_11_15_11_07_28_89	-0.2	1	0 16.41608
/home/daniel/Projects/sdving/track1/try1/IMG/left_2017_11_15_11_07_28_97	/home/daniel/Projects/sdving/track1/try1/IMG/right_2017_11_15_11_07_28_97	-0.4	1	0 17.18055
/home/daniel/Projects/sdving/track1/try1/IMG/left_2017_11_15_11_07_29_04	/home/daniel/Projects/sdving/track1/try1/IMG/right_2017_11_15_11_07_29_04	0	1	0 17.59222
/home/daniel/Projects/sdving/track1/try1/IMG/left_2017_11_15_11_07_29_13	/home/daniel/Projects/sdving/track1/try1/IMG/right_2017_11_15_11_07_29_13	0	1	0 18.79978
/home/daniel/Projects/sdving/track1/try1/IMG/left_2017_11_15_11_07_29_28	/home/daniel/Projects/sdving/track1/try1/IMG/right_2017_11_15_11_07_29_28	-0.35	1	0 20.04031



Using the data recorded, a user could then write and train their models to have a go at the autonomous mode, which allows other programs to communicate with the simulation acting as a localhost “server”.

To attempt the challenge, I wrote two python scripts, drive.py and model.py. Model.py is in charge of loading and training the model using recorded data, and drive.py is in charge of driving the car using presumably an already trained model.

As one could imagine, the hard work is to finish model.py so here is my attempt. Looking at the types of data given, images, steering, throttle, and speed, one could simplify the workload quite a bit. As there is no obstacle in the course, it could be presumed that the only factors affecting the ideal throttle of the car is the speed of the car and the steering angle of the car, if one could find a simple relationship between the variables, a machine learning model is not needed to predict the throttle. As the speed of the car is already given by the program, the aim of the program should be to predict the steering angle of the car given an input image, and to do this one could use a regression model using convolutional neural networks.

With that in mind, here is how I attempted to model the problem in the order of the code being run:

```
    for track_n in range(1, 2):
        for try_n in range(1, 6):
            #Read the images and load the data.
            file_dir = "../sdving/track" + str(track_n) + "/try" + str(try_n) + "/driving_log.csv"
            driving_sample = pd.read_csv(file_dir, sep="," , names = ['Center', 'Left', 'Right',
'Steering', 'Throttle', 'Break', 'Speed'])

            for index, row in driving_sample.iterrows():
                image_data.append(cv2.resize(cv2.cvtColor(cv2.imread(row["Center"])[60:-25, :, :],
cv2.COLOR_RGB2YUV),(200, 66),cv2.INTER_AREA))
                steering_data.append(row["Steering"])
    steering = np.reshape(steering_data, (-1, 1))
    train_images, test_images, train_labels, test_labels =
train_test_split(np.array(image_data),np.array(steering, dtype=np.float32), test_size=0.2,
random_state=0)
```

This contains two embedded for loops that cycles through the required directories and finds the location of the required .csv. After finding the .csv, the program then reads the file into an numpy array, which the rest of the program could understand.

From the numpy array, the program then reads the location of the recorded images and loads them into a separate array after cropping to reduce computing time. The images are converted from its RGB colours to YUV colours as my research shows that this yields better results. I chose only to use the middle images as left and right images provided little to training the model and ignoring them reduced the training time required. Attempts to use the images as artificially expanded datasets by shifting left and right as suggested by Naokishibuya is unsuccessful and did not yield a significant result. This array of images acts as the features(x) of the model.

The recorded steering angle is then copied into an separate array and transposed from a row to a column in order to compare with the predicted value by the model, acting as the label(y). The data is then split into training and test sets for the model, with 80% of the data used for training, and 20% used for evaluating.

```
behaviour_regressor = tf.estimator.Estimator(
    model_fn=cnn_model_fn, model_dir="./Behaviour-cloning-model")

train_input_fn = tf.estimator.inputs.numpy_input_fn(
    x={"x": train_images},
    y=train_labels,
    batch_size=40,
    num_epochs=50,
    shuffle=True)
behaviour_regressor.train(
    input_fn=train_input_fn,
    steps=200000
    )
```

After getting the data ready, one could start writing the code to input the data into the convolutional network(that is not written yet). Tensorflow requires all input data to be in the form of a dictionary, so the training images is reshaped accordingly. The training images dictionary is used as x, and training labels as y. The model's job is to reduce the difference between the given y and the calculated y by the network, when given the same x. In this case, one wants the calculated steering angle by the network to be as close as possible to the one recorded by myself.

The code above tells the CNN model x and y, then tells it to chooses a batch of 40 images. This means it will calculate the cost function and adjust the weights and biases every 40 images. As the number of epochs is 50, it then repeats this process 50 times on the same images, before shuffling the images again and selecting 40 more. This process makes sure that all parts of the given data produced is used in training, not just the beginning. I chose a batch size of 40 because that is the largest my laptop can handle as a larger batch size usually means less biased training. The above process is summarised as a global step, and in the code above I requested 200000 global steps every time the program is ran. In reality, I usually stop the program manually before the training session ends.

```
def cnn_model_fn(features, labels, mode):
    input_layer = tf.reshape(tf.cast(features["x"], tf.float32)/127.5-1.0, [-1, 66, 200, 3])
    # Convolutional Layer #1 & Pooling Layer #1
    conv1 = tf.layers.conv2d(
        inputs=input_layer,
        filters=24,
        kernel_size=[5, 5],
        padding="valid",
        activation=tf.nn.elu)
    pool1 = tf.layers.max_pooling2d(inputs=conv1, pool_size=[2, 2], strides=2)
    # Convolutional Layer #2 & Pooling Layer #2
    conv2 = tf.layers.conv2d(
        inputs=pool1,
        filters=36,
        kernel_size=[5, 5],
        padding="valid",
        activation=tf.nn.elu)
    pool2 = tf.layers.max_pooling2d(inputs=conv2, pool_size=[2, 2], strides=2)
    # Convolutional Layer #3 & Pooling Layer #3
    conv3 = tf.layers.conv2d(
        inputs=pool2,
        filters=48,
        kernel_size=[5, 5],
        padding="valid",
        activation=tf.nn.elu)
    pool3 = tf.layers.max_pooling2d(inputs=conv3, pool_size=[2, 2], strides=2)

    conv4 = tf.layers.conv2d(inputs=pool3,filters=64,kernel_size=[3, 3],padding="valid",
activation=tf.nn.elu)
    conv5 = tf.layers.conv2d(inputs=conv4,filters=64,kernel_size=[3, 3],padding="same",
activation=tf.nn.elu)

    dropout = tf.layers.dropout(inputs=conv5, rate=0.5, training=mode ==
tf.estimator.ModeKeys.TRAIN)
    dropout_flat = tf.contrib.layers.flatten(dropout)
    dense1 = tf.layers.dense(inputs=dropout_flat, units=500, activation=tf.nn.elu)
    dense2 = tf.layers.dense(inputs=dense1, units=50, activation=tf.nn.elu)
    dense3 = tf.layers.dense(inputs=dense2, units=10, activation=tf.nn.elu)
    dense4 = tf.layers.dense(inputs=dense3, units=1)
    result = tf.identity(dense4, name="result")
```

The above code then defines the exact shape and size of the model used. As shown, the model contains 5 convolutional layers and 4 fully connected layers. The network takes a cropped 66*200 image as an input and scales the 255 bit colour values down to -1 to 1 by dividing by 127.5 and subtracting 1. This is after being converted into floating point numbers as if the values stayed as integers, the calculation will produce an all black or all white image which is what I got for a long time. The scaling is to ensure that the cost function does not diverge: as root mean squared error is used, a large initial value may cause the cost function to increase until infinity, so by scaling the values down, this means that the cost function will always converge to zero if a low learning rate is used. An example input layer is shown below:

The first three convolutional layers contain a kernel size of 5 by 5, this means that it uses 5 by 5 grids to scan across the image, and produces the new layer, with in order 24, 36, and 48 filters. The padding is valid so that the scanning process ignores the edge pixels, and each of the 3 layers has a 2 by 2 pooling layer after it, these work by averaging 4 pixels into 1. These methods reduce the workload needed and extract the most defining features out of the image in order to yield a more consistent result. This outputs an image similar to the following, where all the features from the original image is hopefully extracted onto:

After these three layers are 2 more convolutional layers in an effort to extract more features followed by a dropout layer. A dropout layer is a layer that only allows a random half of the layer to be trained at once, the idea of this layer is to prevent over-fitting, the idea that the trained model will only be good with dealing with the training data and cannot successfully generalise beyond it. Randomising the training process prevents excessively large weights and biases from existing, therefore limiting the chance of over-fitting.

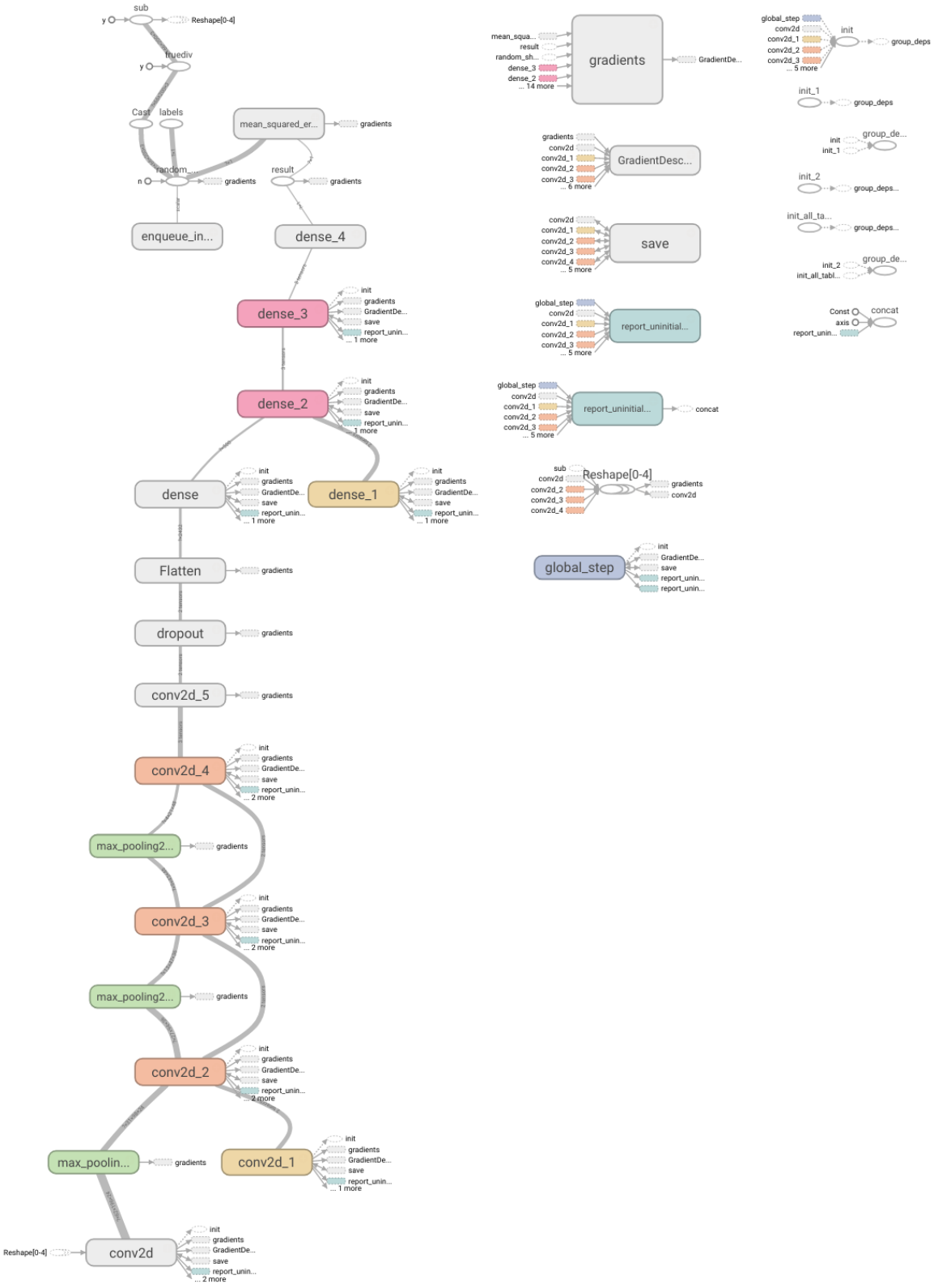
The network is followed by 4 fully connected layers, with in order 500, 100, 10, and 1 neurons to collect the information given by the convolutional layers into a single value of steering angle.

```
if mode == tf.estimator.ModeKeys.PREDICT:
    return tf.estimator.EstimatorSpec(mode=mode, predictions=result)
loss = tf.losses.mean_squared_error(labels=labels,predictions=result)

if mode == tf.estimator.ModeKeys.TRAIN:
    optimizer = tf.train.GradientDescentOptimizer(learning_rate=0.00001)
    train_op = optimizer.minimize(
        loss=loss,
        global_step=tf.train.get_global_step())
    return tf.estimator.EstimatorSpec(mode=mode, loss=loss, train_op=train_op)
```

If the network is in prediction mode, the program then returns the value of the last node as it's prediction. If not then it calculates the loss function using mean squared error with the labels being the steering angle produced by humans and the prediction being the value of the last node. If the model is in training mode, it then tries to reduce the cost function by a process called gradient descent.

I tried 1, 0.1, 0.001, and 0.0001 as the learning rate for gradient descent with varying result. Using too high of an learning rate may cause the cost function to never converge but using too low a learning rate could increase the computing time significantly. Through trial and error, I found that 1 and 0.1 are both too high and caused the cost function to diverge, 0.001 is a good learning rate to start with but also does not converge on an low enough cost function. In the end, a learning rate of 0.001 is used at the beginning to reduce the computing time, then I lowered the learning rate to 0.0001 and then 0.00001 to improve the accuracy of the model.

The model described is very similar to the one recommended by Nvidia which attempted a similar project before. The modifications I made was to better suite the Tensorflow platform and trials and errors from my own experience. My code could be summarised by the flowchart below:



Above describe the method for producing and training the model in model.py. Drive.py(modified from Naokishibuya's attempt by changing it from using his Keras model to the Tensorflow model used in this project) is then used to connect the model to the simulation.

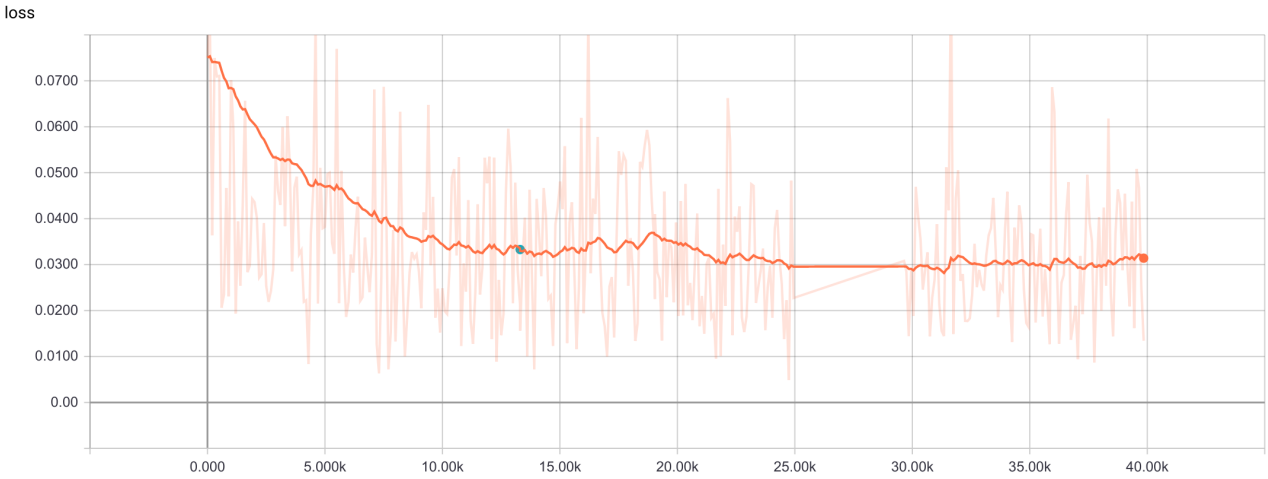
```
    if data:
        speed = float(data["speed"])
        throttle = float(data["throttle"])
        speed = float(data["speed"])
        image = Image.open(BytesIO(base64.b64decode(data["image"]))).convert('RGB')

        try:
            image = np.asarray(image)
            image = image[:, :, ::-1].copy()
            image = cv2.resize(cv2.cvtColor(image[60:-25, :, :], cv2.COLOR_RGB2YUV),(200,
66),cv2.INTER_AREA)
            image = np.array([image])

            predict_input_fn = tf.estimator.inputs.numpy_input_fn(
                x={"x": image},
                batch_size=1,
                shuffle=False)
            result = behaviour_regressor.predict(input_fn=predict_input_fn)
            steering_angle = next(result)[0]

            global speed_limit
            speed_limit = MIN_SPEED if speed > speed_limit else MAX_SPEED

            throttle = 1.0 - steering_angle**2 - (speed/speed_limit)**2

            print('{} {} {}'.format(steering_angle, throttle, speed))
            send_control(steering_angle, throttle)
        except Exception as e:
            print(e)
    else:
        sio.emit('manual', data={}, skip_sid=True)
```

As shown, if data is detected by the python script, it extracts the data, and formats the image as the input size for the neural network, it then calls the prediction function of the network and the network returns the value of the last layer of neurons. As the last layer only has 1 neuron(the steering angle), result(0) is used to get it's value. If the model is trained, this would hopefully be the desired steering angle for the situation. I then used `throttle = 1.0 - steering_angle**2 - (speed/speed_limit)**2` to calculate the desired throttle as used in Udacity's examples, and send the controls over to the simulation. This would then hopefully produce a simple self-driving car.

I also planned on writing a program to complete the challenge using Model predictive Control and making a Raspberry Pi model car using both methods, however, due to the unexpected long time needed to learn and use machine learning, these plans were halted in favour of using the results of other experts.

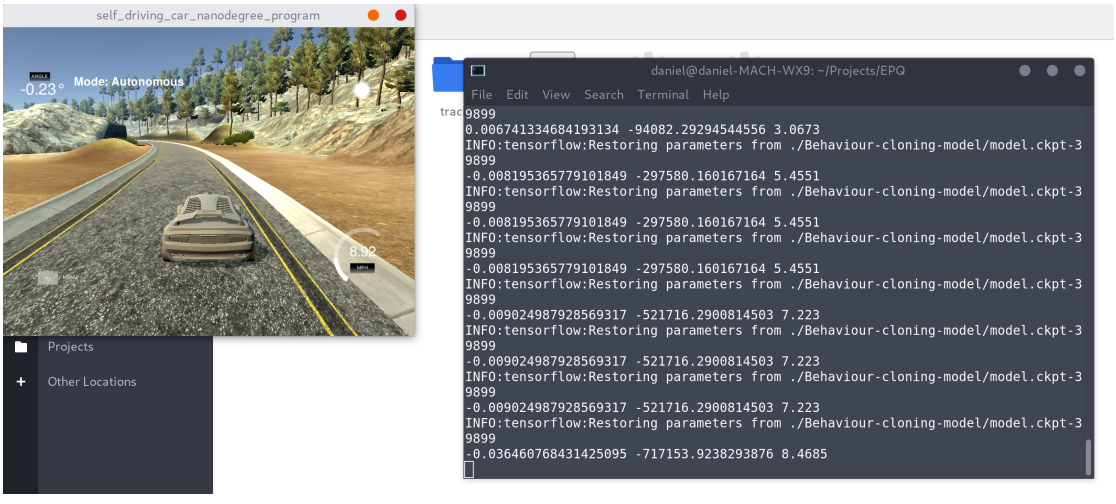
Results and analysis

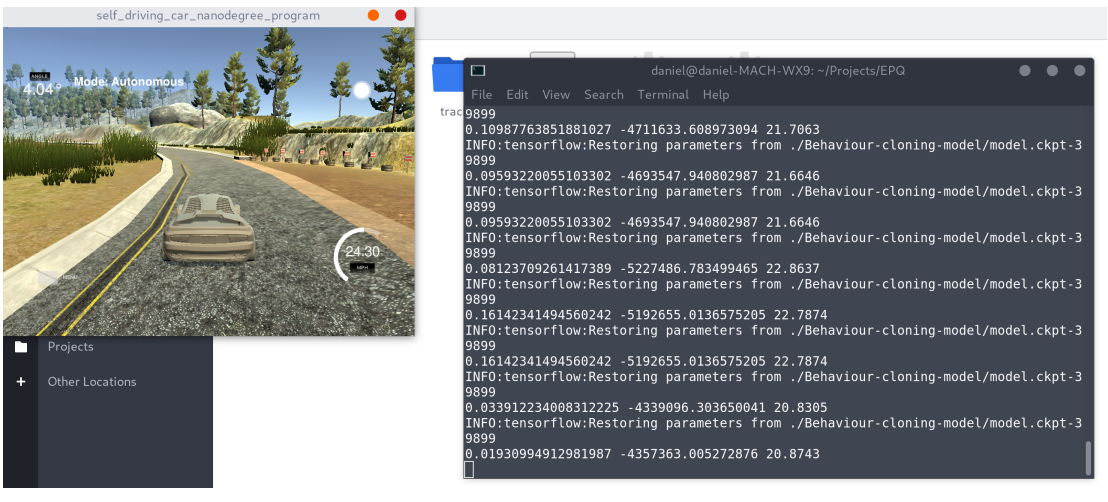
The results from this project is different from what I expected and proves the difficulty of creating such a system. After several attempts at modifying the model including varying the learning rate, changing the dropout rate, changing the whole structure of the fully connected layers, below is the best cost function I could get:



The graph shows the cost function in relation to the number of global steps, the cost function drops dramatically from 0 to 10k, before flattening at 10k to 15k, then starting to increase back up from 15k to 19k possibly due to over-fitting. To counter the issue I lowered the learning rate from 0.001 to 0.0001, which dropped the learning rate down to an average of 0.03, then I lowered the learning rate once more to 0.0001 but the cost function stayed at 0.03 for 10k.

Using this cost function of around 0.03, the model was able to drive the car at low speeds (<=25mph) around track 1 with human interventions at tight edges and recovery situations, the program was not able to drive without major human intervention around track 2.

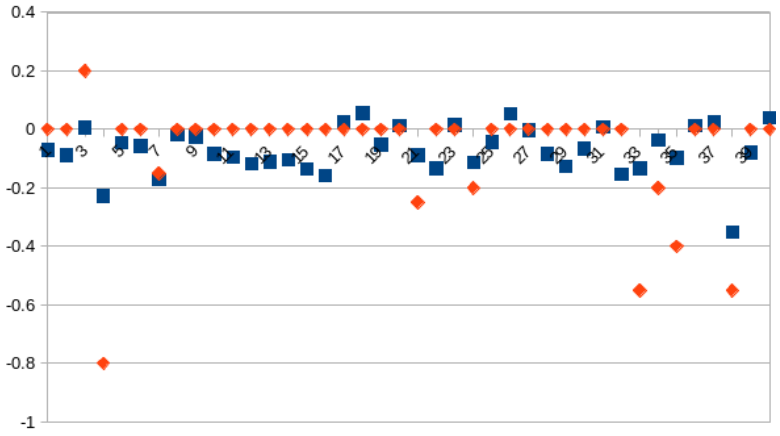




Looking at some raw data from the output could help with finding improvements to the model:

Calculated Steering angle	Human Steering angle
-0.06983535	0
-0.08703621	0
0.00726715	0.2
-0.22668575	-0.80000007
-0.04502907	0
-0.05557102	0
-0.16989344	-0.15000001
-0.01911068	0
-0.02458413	0
-0.08368663	0
-0.09508058	0
-0.11673108	0
-0.11090682	0
-0.1024816	0
-0.13582982	0
-0.15849783	0
0.0253439	0
0.05664245	0
-0.0517484	0
0.0138239	0
-0.08827768	-0.25
-0.13208538	0
0.01616539	0
-0.1124405	-0.2
-0.04356452	0
0.05476038	0
-0.00300911	0
-0.08274044	0
-0.12564878	0
-0.06615399	0
0.0078903	0
-0.1537438	0
-0.13353935	-0.55000007
-0.0368935	-0.2
-0.09589692	-0.40000001
0.01274048	0
0.02415559	0
-0.35064209	-0.55000007
-0.07864147	0
0.04028311	0

Just by looking at the data, one could see that the human angles are more discrete, that is it only peaks at certain values, presumably when I press the arrow keys. The calculated angles are more averaged, possibly due the the inability to differentiate the given images substantially, graphing the data only confirms the suspicion:

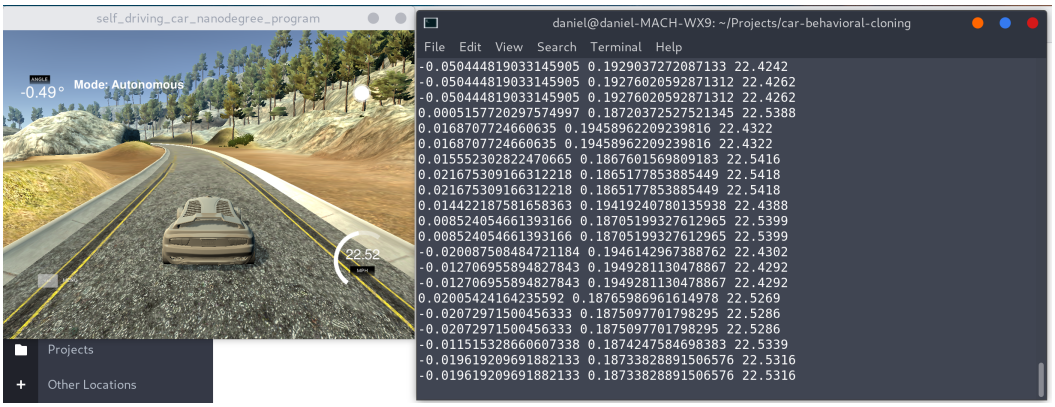


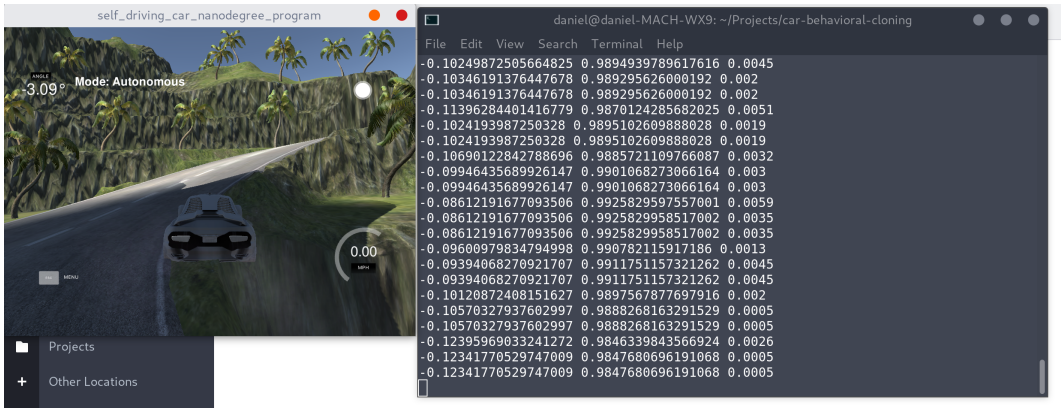
(orange being human and blue being the model)

To improve upon the model's ability to drive successfully, it is in my opinion that a smoother human input is needed, possibly by using a joy stick. I could also improve the model by using a larger and better dataset by asking people who are better at driving than I am to generate the data, and tweak the hyper-parameters more so that the model could better differentiate between the generated images. I could have also trained the network with more recovery situations and sharp turns as they cause more trouble to the network than mild but continuous turns.

So are end-to-end neural networks that emulate human behaviour suitable for use in autonomous vehicles? My experience suggests that it is a flawed approach as humans are not the perfect driver, and a program that learns to drive like humans end up inevitably driving even worse than a human, however to conclusively give an answer, the data from experts in the field needs to be reviewed.

Using the model built and trained by Naokishibuya, which is the best behaviour cloning model I could find for this particular challenge, the car could drive at an average speed of 22 mph around track 1 with no human intervention, and it could drive at speeds less than 15 mph around track 2 with human intervention at hard edges and recoveries.

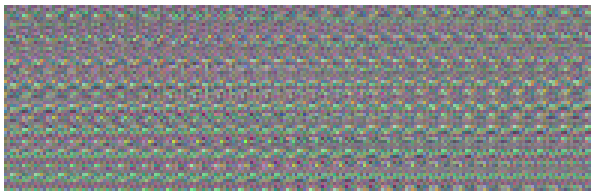


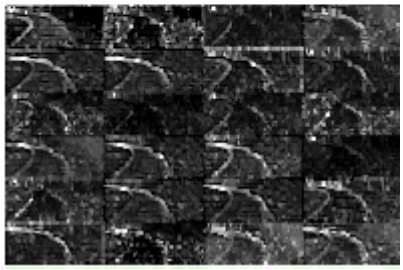


Compared to my attempt, the attempt by Naokishibuya more closely resembles Nvidia model and he found success in artificially expanding the dataset by shifting, rotating, inverting, and changing the brightness of the image to avoid over-fitting. He also spent time to teach the model about specific recovery scenarios so less human intervention is needed.

Nvidia's network based on end-to-end machine learning which is trained enough to drive cars on public roads is perhaps the limit of what such a system could do. Their network has learnt how to steer the car in various conditions with hundreds of hours of driving data, which is very compelling considering the simplicity of the model and the relatively short training time.

Looking at the first convolution layer of my model and comparing it to Nvidia's model(that has been trained for a significantly longer time on faster machines), it is clear that mine did not pick up the edges of the road enough for driving real vehicles.

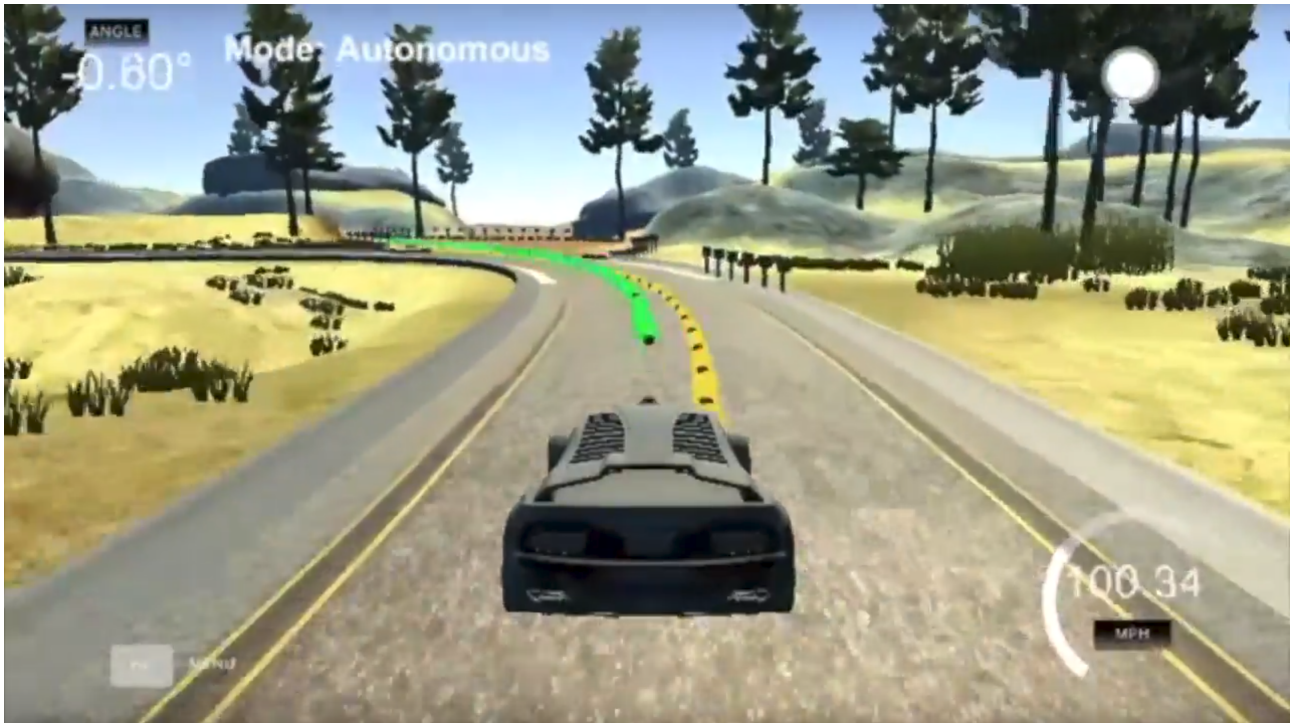




(left mine right Nvidia's)

To objectify the suitability of an end-to-end system in self driving cars, we could compare it to other approaches such as Model Predictive Control(MPC), a more complex system that breaks down the environment for the computer, before sending the data to determine how the car should be steered.

YouTube user Bstppldrmr managed to use MPC drive around the Udacity lake track at around 90-100 mph without human intervention, significantly higher than the 20-22 mph possible with MPC. The car also managed to stay in the center of the track unlike what happens with behavioural cloning where the car wobbles from side to side occasionally. Unfortunately I was not able to find any attempts of using MPC around the jungle track, however, the fact that this method is used in more complex situations is evidence that it works well.



Conclusion

After looking at data from my own project, as well as the data from people attempting similar challenges, I think it's safe to say in terms of accuracy, average speed, safety of an vehicle, Model Predictive Control is clearly superior to end-to-end learning as a mean drive vehicles autonomously. It could keep the car in the center of the lane better, interact with situations it has never met before, and because of these it could drive faster and safer than the end-to-end approach.

However, an end-to-end system like the one developed in this project is easier to write, easier to train, and is still very useful when developed well like the Nvidia's model. Behavioural cloning means that such a system could potentially interact better with human drivers as it behaves in a more predictable way, a problem that Waymo's self driving taxis are facing now. In the end I believe a system that combines the advantages of both approaches is needed to bring us to the age of full autonomy.

Bibliography:

Bertozzi, M., Bombini, L., Broggi, A., Buzzoni, M., Cardarelli, E., Cattani, S., Cerri, P., Debattisti, S., Fedriga, R.I., Felisa, M., Gatti, L., Giacomazzo, A., Grisleri, P., Laghi, M.C., Mazzei, L., Medici, P., Panciroli, M., Porta, P.P., Zani, P., n.d. The VisLab Intercontinental Autonomous Challenge: 13,000 km, 3 months,… no driver 11.

Broggi, A., Cerri, P., Felisa, M., Laghi, M.C., Mazzei, L., Porta, P.P., 2011. The VisLab Intercontinental Autonomous Challenge: an Extensive Test for a Platoon of Intelligent Vehicles," Intl. Journal of Vehicle Autonomous Systems, special issue for 10 th Anniversary.

Build a Convolutional Neural Network using Estimators | TensorFlow [WWW Document], 2018. URL https://www.tensorflow.org/tutorials/estimators/cnn (accessed 8.20.18).

CarND-Behavioral-Cloning-P3: Starting files for the Udacity CarND Behavioral Cloning Project, 2018. . Udacity.

CS231n Convolutional Neural Networks for Visual Recognition [WWW Document], 2018. URL http://cs231n.github.io/convolutional-networks/ (accessed 8.19.18).

Davies, B., Lienhart, R., 2006. Using CART to segment road images, in: Chang, E.Y., Hanjalic, A., Sebe, N. (Eds.), . Presented at the Electronic Imaging 2006, San Jose, CA, p. 60730U. https://doi.org/10.1117/12.650164

Early Rider Program [WWW Document], n.d. . Waymo. URL https://waymo.com/apply/ (accessed 8.27.18).

End-to-End Deep Learning for Self-Driving Cars [WWW Document], 2016. . NVIDIA Dev. Blog. URL https://devblogs.nvidia.com/deep-learning-self-driving-cars/ (accessed 8.19.18).

Guizzo, E., 2010. Autonomous Vehicle Driving from Italy to China [WWW Document]. IEEE Spectr. Technol. Eng. Sci. News. URL https://spectrum.ieee.org/automaton/robotics/robotics-software/autonomous-vehicle-driving-from-italy-to-china (accessed 8.26.18).

Levinson, J., Askeland, J., Becker, J., Dolson, J., Held, D., Kammel, S., Kolter, J.Z., Langer, D., Pink, O., Pratt, V., Sokolsky, M., Stanek, G., Stavens, D., Teichman, A., Werling, M., Thrun, S., 2011. Towards fully autonomous driving: Systems and algorithms, in: 2011 IEEE Intelligent Vehicles Symposium (IV). Presented at the 2011 IEEE Intelligent Vehicles Symposium (IV), IEEE, Baden-Baden, Germany, pp. 163–168. https://doi.org/10.1109/IVS.2011.5940562

Machine Learning Crash Course | Google Developers [WWW Document], 2018. URL https://developers.google.com/machine-learning/crash-course/ (accessed 8.20.18).

Malgonde, S., 2018. A car trained by a deep neural network to drive itself in a video game: subodh-malgonde/behavioral-cloning.

Motion Planning for Autonomous Vehicles | Dynamic Design Lab [WWW Document], n.d. URL https://ddl.stanford.edu/publication-research-theme/motion-planning-autonomous-vehicles (accessed 8.27.18).

Nielsen, M.A., 2015. Neural Networks and Deep Learning.

Premade Estimators [WWW Document], n.d. . TensorFlow. URL https://www.tensorflow.org/guide/premade_estimators (accessed 8.27.18).

Safe driving envelopes for path tracking in autonomous vehicles | Dynamic Design Lab [WWW Document], n.d. URL https://ddl.stanford.edu/publications/safe-driving-envelopes-path-tracking-autonomous-vehicles (accessed 8.27.18).

Shibuya, N., 2018. Built and trained a convolutional network for end-to-end driving in a simulator using Tensorflow and Keras: naokishibuya/car-behavioral-cloning.

Stanford Racing :: Home [WWW Document], n.d. URL https://cs.stanford.edu/group/roadrunner//old/technology.html (accessed 8.26.18).

Tartan Racing @ Carnegie Mellon [WWW Document], n.d. URL http://www.tartanracing.org/tech.html (accessed 8.26.18).

Udacity, n.d. Self-Driving Car Nanodegree Program Overview.

Urmson, C., Anhalt, J., Bagnell, D., Baker, C., Bittner, R., Dolan, J., Duggins, D., Ferguson, D., Galatali, T., Geyer, C., Gittleman, M., Harbaugh, S., Hebert, M., Howard, T., Kelly, A., Kohanbash, D., Likhachev, M., Miller, N., Peterson, K., Taylor, M., 2007. Tartan Racing: A multi-modal approach to the DARPA Urban Challenge.

Waymo, n.d. Waymo 360° Experience: A Fully Self-Driving Journey.

What are some of the technologies used in the Google self-driving car? - Quora [WWW Document], n.d. URL https://www.quora.com/What-are-some-of-the-technologies-used-in-the-Google-self-driving-car# (accessed 8.27.18).

Yadav, V., 2016. An augmentation based deep neural network approach to learn human driving behavior [WWW Document]. Chatbots Life. URL https://chatbotslife.com/using-augmentation-to-mimic-human-driving-496b569760a9#.d779iwp28 (accessed 8.27.18).s