CS 241 Final Project

Sahil Bambulkar

May 12, 2022

1 Background

Self Driving Cars

While a self driving car has been the stuff of science fiction for decades, it is now firmly within our grasp. With the most famous being Tesla, today there are numerous companies developing self driving cars with many available for consumer purchase. From technology companies like Apple, Google, Nvidia, Samsung, Huawei and Baidu, to traditional car companies like Audi, BMW, Ford, GM, Hyundai and Honda, to ride-sharing companies like Uber and Lyft. One notable entrant is Comma.AI, a company that sells a small smartphone sized driving assistant that attaches to your car and converts it into a self driving car. The code is open source and freely available to view and contribute to.

Advances in deep learning have been the primary technology that have enabled these advances. Previously, it was nearly impossible to attempt to hard code individual decisions for each of an nearly infinite set of test cases that occur when driving on the roads. Deep learning allowed for the development and training of advanced models that learn the best driving solution through large amounts of data. Some implementations of self driving deep learning in use right now are HydraNet at Tesla, ChauffeurNet at Google Waymo and the self-titled Nvidia Self Driving Car at Nvidia.

Section 1.1: Background - History of Self Driving Cars

The invention of a self driving car is so core to human desire that it may have even preceded the invention of the car itself. In the middle ages, Leonardo De Vinci sketched a blueprint for a theoretical self-propelled cart using springs for propulsion. Then in 1776 Nicolas Cugnot of France actually developed a self-driving steam engine that could navigate the streets of Paris at an extremely slow two miles an hour.

With the invention of the automobile some one-hundred years later, this idea saw renewed public interest. The Houdina Radio Control company demonstrated the first driver-less car in 1925, by having it navigate the trafficked streets of Broadway and Fifth Avenue in New York City. But as their name suggests, Houdina Radio Control company was simply controlling the car remotely through radio with another car following close behind.

Of note, all of these early attempts from Leonardo to the Houdina Control Company were not truly self driving cars we think of today. They were designed for novelty not practicality, and thus had to have a set of steering and acceleration instructions pre-programmed or remotely sent to them before or during their rides. They did not do much to solve problems of an actual driver as they all still required a navigator to input instructions for each of the vehicle's rides.

But as the automobile began seeing widespread propagation in the twentieth century, the idea went from an interesting pet project to having vast practical applications for driver safety and speed of travel. The more realistic vision shifted the focus of self driving cars to solving the practical needs of consumers and ultimately, to invent a truly self driving, navigator-less car.

The first real attempt at this was in 1958 by General Motors and an industrialist named Norman Bel Geddes. As Bel Geddes put it: "These cars of 1960 and the highways on which they drive will have in them devices which will correct the faults of human beings as drivers." Thus, the original vision was for a centralized logic system integrated into the pavement that used electronic circuits to gauge road conditions and steer the vehicles. The prototype was successfully deployed in Princeton, New Jersey in 1960 on a 400-foot stretch of automated highway. General Motors even invented an experimental

Firebird model of cars that would work in tandem with the circuits of the road. Ultimately, this project was scrapped due to $ 931,069.02 adjusted for inflation per mile cost of the project.

Thus, what is interesting about the self driving car problem is that it is a problem that we have manufactured because of the widespread use and relatively cheap cost of traditionally operated human driven cars. Given no legacy cars or existent road infrastructure, it would be relatively easy and affordable to invent and deploy a system of transportation nodes with centralized logic that could avoid collisions, obey traffic laws and optimize routes. It would be nearly identical to our existing train infrastructure invented in 1804 and such a car based system was even successfully completed by General Motors in 1958.

Section 1.2: Background - Nature of Problem

Since the 1960s, we are in the current era of self driving car technology and with a clear outline of the nature of the problem. Key to the problem are that it requires a decentralized network of cars using externally mounted visualization hardware and using onboard processing to make independent decisions in an attempt to model, and ultimately outperform, a human driver. It also has extremely high stakes, as the cost of failure is the loss of a human life. And thus simulating realistic driving conditions to collect data for training your model is often just as important as developing the self driving car. The self driving problem can be broken down into three main components.

- Its **perception problem** refers to collecting environmental information primarily through cameras, LIDAR and GPS. This is needed to perceive the setting surrounding the car from the front, sides and back. Because of conditions like sun glare and inclement weather cameras are used alongside LIDAR and GPS technologies. Other sub decisions in the process involve what parts of the camera feeds are valuable, like the road and sidewalk, and which parts can be ignored and masked off, like the sky.

- Its **localization problem** refers to detecting and classifying objects in the input feeds. These are objects relevant to driving like lanes, obstacles, traffic lights, street signs, other transport vehicles and most importantly pedestrians. This is a detection problem for lanes and a classification problem for street signs, traffic lights and obstacles. Basic edge detection is often used for lane boundaries and convolutional neural networks are used for the more advanced classification tasks.

- And finally its **decision making problem** which is one, or multiple, deep learning approaches used to process all the variables to make driving decisions. These decisions consist of determining the magnitude and timing of acceleration, turning, braking and signaling. This is done through reinforcement learning which is using neural networks to solve complex goals. This involves an agent, the car, taking an action and changing its state. The environment responding to the agent by giving it rewards which changes its value, or expected long-term reward, which in turn changes its trajectory or sequence of actions.

Section 1.3: Summary - Potential Consequences of Solving Problem

George Holtz, founder and CEO of self driving company Comma.AI said that solving chess and internet search were the biggest technological problems solved in the early part of this decade. And that self driving is the preeminent technology problem that will be solved in the next two decades. This is because the potential implications of solving the self driving car problem are so profound and far reaching.

Increasing Accessibility

They would provide widespread travel accessibility to young, disabled or elderly individuals who are unable to drive. This would extend to eligible drivers with deteriorating eyesight, physical or mental restrictions that reduce their ability to drive effectively.

Reducing Resources and Saving the Environment According to a Department of Energy report, there are huge potential environmental benefits to self driving cars. During travel they could use 60 percent less gasoline by taking shorter routes, eliminating inefficient behaviors like idling or keeping the engine running and closely following other cars to avoid drag. As only 12 percent of today's vehicles are on the road any given time, ridesharing autonomous vehicles could eliminate the

need for universal car ownership reducing manufacturing and shipping costs by nearly 90 percent. And none of this considers the fact that many self driving cars are electric eliminating their dependency on fossil fuels altogether.

Economic Savings

Along with an environmental benefits, less cars would also extend substantial economic benefits. Eliminating the need to purchase a car would save consumers money, along with the additional savings on insurance, maintenance, parking and storage. While there would be additional costs imposed like increased maintenance for the fewer cars being used more and ride-sharing costs they would be far exceeded by the savings. While a tangential problem to cars, self driving trucks, ambulances, farm and construction vehicles would revolutionize industries and automate entire supply chains.

Increasing Productivity

There would also be a substantial increase in productivity as, according to the Census Beureu, the average American spends a total of 19 full work days in their car commute each year. While current self driving cars require the driver stare at the road, freeing up that time would allow for it to be used for leisure or work.

Reducing or Eliminating Motor Vehicle Deaths

And perhaps the greatest potential benefit is decreasing motor vehicle deaths which take over 35,000 lives in the United States each year and put 2.3 million in the hospital with injuries. Driving is more dangerous in other parts of the world with their being 1.25 million traffic-related deaths each year or one every 25 seconds. The implications of this technology is so meaningful that self driving vehicles could drastically reduce or altogether eliminate vehicle and traffic deaths. Self driving cars would be given more information through all their sensors, be able to process faster and make perfect decisions. Principally of these accidents, DUI crashes could be avoided by removing the option for intoxicated drivers to put others at risk. This also applies to distracted drivers or dangerous drivers as well as those those experiencing a medical episode who would otherwise put others at risk. In addition, the onboard processing of many of these cars could take conscious actions to swerve to avoid local environmental accidents like fallen branches, rock slides or wild animals or avoid areas with broader environmental accidents like floods, hurricanes and tornadoes.

Section 1.2: Results: Math Behind Self Driving Cars (prior proofs/experiments)

Loss Functions and Optimization

The relevant results section, called CNNs used in Self-Driving, is included at the end of this, but to understand the implications a general introduction to deep learning and neural networks is needed. As a general introduction to the learning problem, we will briefly describe the fundamentals. Please note this is not a thorough explanation and briefly explains the outline so we can move onto convolutional neural networks and behavioral learning and other topics more relevant to self driving.

$$f(x, W) = Wx$$

Given a learning problem of classifying images by assigning probability of their likeliness, we can use X as input data, W as the weights applied to the inputs and Y as predicted labels. Then we can create a loss function that gives a quantifiable score on how to grade how good the weights are at predicting each image. This is useful for optimization as we can then aim to minimize this loss and improve the weights of our algorithm to better predict labels.

Again, say we are classifying images. Given an an RGB image defined as an array of n pixels with each pixel being three values of 255x255x255 representing RGB values. Our model applies weights to the this input data and returns a probability of each label from cat, dog, frog, fish.

A potential loss function could be:

$$L_i = \sum_{j \neq y_i} max(0, s_j - s_{y_i} + 1)$$

This would essentially only sum values where the label is incorrect and subtract the difference between the correct prediction percentage and the incorrect prediction percentage.

Additionally, if we only tried to fit data we would experience overfitting where the model too perfectly fits our training data. As our training data is only sampled from a much larger subset of the actual data, this would not be ideal. It may perform very well and provide low training loss, or loss calculated with on our training data set. But when presented with outside data it will have very high loss or high test loss. This is called overfitting.

To prevent this, we add an additional regularization term $R(W)$ to our loss function that punishes the function for complexity. This can be implemented by punishing the number of terms in the loss function by punishing polynomials of high degree. You can also artificially constrain your model class to not contain more complex models.

$$L_i = \sum_{i=1} L_i(f(x_i, W), y_i) + \lambda R(W)$$

To optimize this loss function, we can solve for the derivative (or gradient) to find how changes to each weight contribute to the loss and then minimize it. It seems like a simple calculus optimization problem. As you can probably guess, this is not as trivial a problem as it seems because each loss function can consist of many variables. In more complicated learning problems, like self driving, there can be millions to billions of variables. Thus, the beautiful solution of backpropagation can be used to solve this. Backpropagation is a recursive application of the chain rule that does this step by step to make solving for the gradient relatively easy. We also use stochastic gradient descent where we do many calculations far from the optimal value and make smaller and smaller changes as we reach the true optimal value for the optimization problem. This is extremely useful when it is difficult or not possible to solve for where the derivative is 0.

Neural Networks Right now, we have presented a potential model to predict classes as a simple linear function.

$$f = Wx$$

Consider if we stacked two of these functions ontop of each other. Though, we cannot only do that, as two linear functions will simply collapse into another simple linear function once again. So we can use a non-linear function to fix that issue.

$$f = W_2 max(0, W_1 x)$$

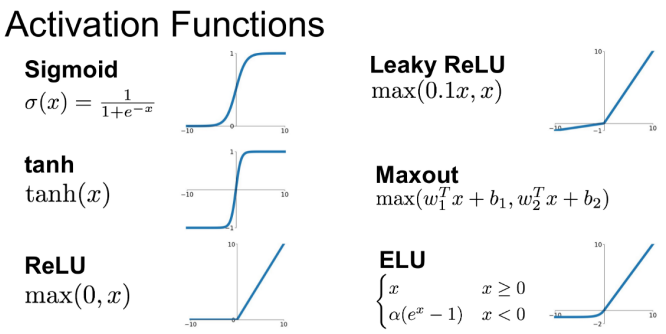
This is the simplest form of a 2-Layer Neural Network. And performing this again will give us a 3-Layer Neural Network. As we add layers, each resultant output is fed into the next. Neural Networks are a class of functions where we have simpler functions, hierarchically stacked on top of each other to create a more complex non-linear function. The term deep neural network can now be visualized as us adding more and more layers. Below is an example of a simple 3 layer neural network.

$$f = W_3 max(W_2 max(0, W_1 x))$$

Activation Functions

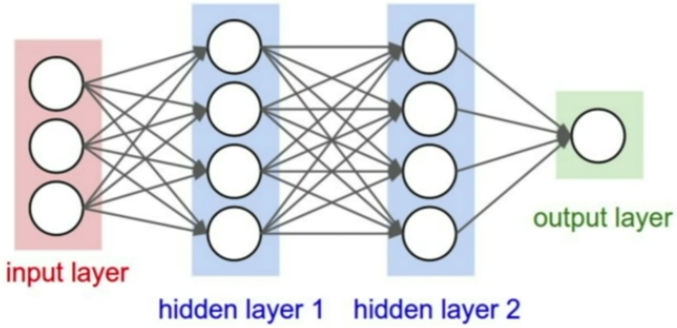
These activation functions, which at each neuron take all the inputs coming in and output one number going out, have to be chosen carefully. Some examples of this are the sigmoid activation function and the tanh Activation function. Another very popular activation function, that many neurologists say are similar to the way brain neurons fire, is the ReLU or rectified linear unit activation function, that is 0 for non-negative values and is linear for positive values.

$$f(x) = max(0, x)$$



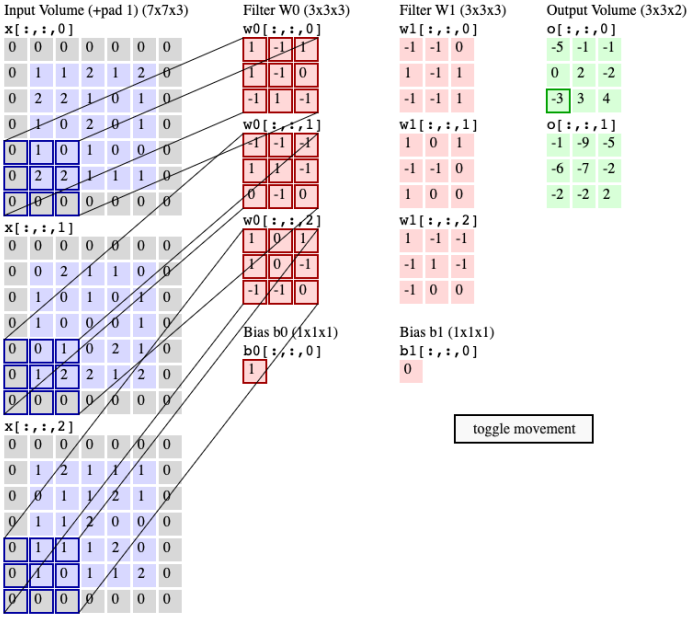
Types of Layers

The types of layers are the input layer that takes in the raw data at the beginning. As well as the hidden layers, or the layers between the input and output that apply non-linear functions to process the data. They are called hidden because the user will only see the input and output not seeing the hidden layers involved in the model. And finally the output layer that takes the processed data and produces final results. Additionally there are many other types of layers that have different purposes.



Convolutional Neural Networks

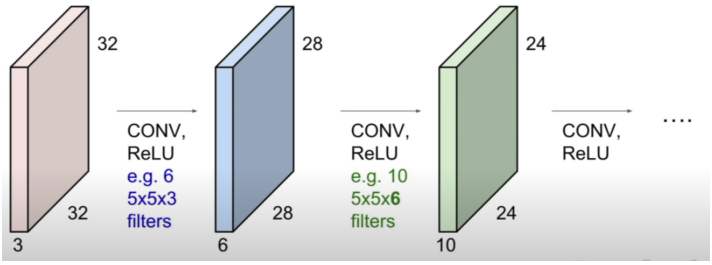
So far in our discussion above of neural network layers we presented fully connected layers where all the inputs were essentially stretched out and applied equally. So, for example, if we are processing an image a 32x32x3 pixel image, to apply the dot product with a weight we would effectively treat it as a 3072x1 image to be multiplied by weights to determine the output of the neuron.



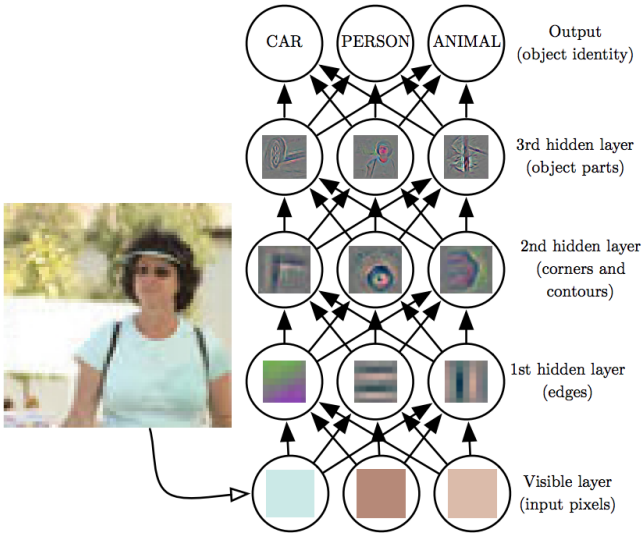
The main difference between for a convolutional neural network is that we want to preserve structure. This is done by convolving a filter, or sliding a filter over the image spatially and computing dot products at each point. The standard definition of a convolution is:

$$f[x,y] * g[x,y] = \sum_{n_1=-} \sum_{n_2=-} f[n_1,n_2] * g[x-n_1, y-n_2]$$

As the image demonstrates, each layer can consist of multiple filters. That produce an output with a width of the number of filters. Also of note, the depth of each filter must match the depth of the input. Each of these layers is interspersed with an activation function. There are also some pooling layers that downsample the size of activation maps. And there is one final fully connected layer to get a final score.



The way this turns out is that early layers define low-level features, middle layers define mid-level features and later layers define high-level features.



CNNs used in Self-Driving

- **HydraNet by Tesla** HydraNet is a semantic segmentation implementation for self-driving cars first proposed in 2018 by Ravi et al.

  In a broad sense, Tesla aims to optimize each individual part of self driving rather than taking an end-to-end approach. They are separately optimizing models like lane detection, collision avoidance and breaking distance instead of putting them together.

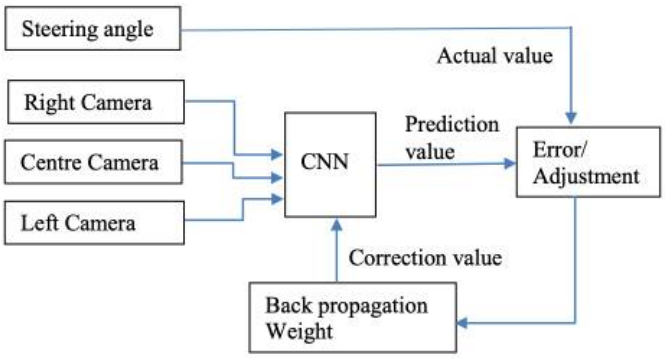
  It has a dynamic architecture so different CNNs can be assigned to different jobs. So for example, a dataset can be trained on static environments of a plain road with no weather or obstacles, another can be trained on heavy traffic. Then at inference time, the gate chooses which branch to run and the combiner aggregates branch outputs to make a final decision. This multi-pronged approach gives it its name, HydraNet, based off of the mythological snake-like monster with nine heads.

- **ChauffeurNet by Google Waymo** ChauffeurNet is an RNN, recurrent neural network, implementation used by Google Waymo. Still CNN is again a core component of this system used to extract features from the perception system. These two work together as the convolutional feature network extracts contextual feature representation which is then fed into a recurrent agent network. Unlike the end-to-end process where the perception system is part of a deep learning algorithm, here the system is a mid level system and left detached.

  It is based off a paper by Bansal et al. called ChaufferNet: Learning to Drive by Imitating the Best and Synthesizing the Worst. Key to this paper, is the argument that solving the problem requires impossibly large amounts of data and thus we must use synthetic data with artificial scenes to optimize. They found that this approach was able to train the car much more efficiently.

- **Nvidia Self Simplistic Driving Car Model**

  Nvidia uses a convolutional neural network as the primary algorithm with three cameras, one on each side and one in the front. It is an end-to-end system that optimizes all the processing steps at once. In this way it is traditional compared to Tesla which breaks these into different parts.



  Later in this paper, we will use NVDIA's architecture in an attempt to train a model to steer a car around a track. Its architecture will be discussed more in depth at that point.

2 Implementation (Playing with the Problem)

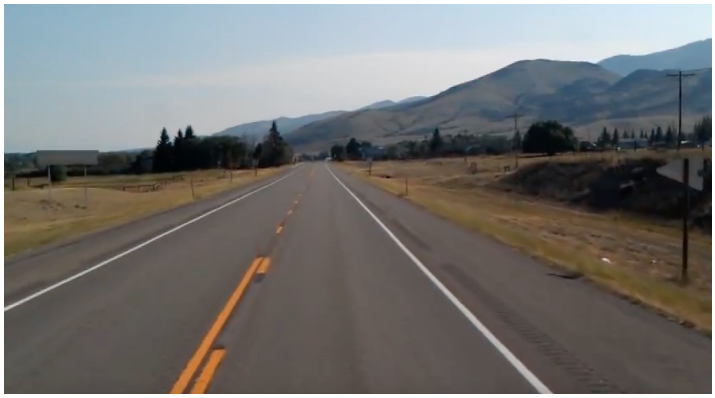
Due to limitations on resources like time, processing tools and hardware the scope of this project had to be restricted. Thus, my attempt at playing with the problem involved attempting to simulate and then solve subsets of the self driving car problem to give insight into the process and its challenges. Specifically, the perception/localization and decision making aspects of the problem.

Section 2.1: Localization and Perception: Line Finding and Masking

Our first attempt at playing with the data will involve simple edge detection for a self driving car. While this is basic, performing this manually lays a foundation to understand the tasks being performed by each neuron in a convolutional neural network.

We will perform all of the operations in Python using libraries which abstract many of the underlying concepts for simplicity. As we are more interested in these concepts, I will explain these underlying concepts and provide image results. And then at the end I will provide the entire code as a file rather than walking through the code line by line.

Link to Lane Finding and Masking Python Code



Grayscale Conversion

After loading our image as a matrix of pixels, the first step of edge detection is to convert the image to grayscale. This will reduce each pixel of the image which normally represents three RGB values of an image from 255x255x255 to one value of 0 to 255. 0 would be fully black and 255 is fully white. This can be accomplished with a very simple function call to the cv2 library: cv2.cvtColor(image, cv2.COLOR_RGB2GRAY).

Gaussian Blur

Next we will add gaussian blur. This is important as smoothing or blurring the image is important to remove false edges. This is accomplished in a very similar method as one filter of a convolutional layer of a convolutional neural network discussed earlier. This is because it also uses kernel convolution. A kernel of normally distributed numbers is run across the image and sets each pixel value equal to the weighted average of its neighboring pixels. This can also be done with a function call to the cv2 library: cv2.GaussianBlur(gray, (5,5), 0).

Gradient: Deriving Multi-Variable Function

Now using this image, we can find the gradient or the change in brightness over adjacent pixels. A strong gradient would be a change from white to black or vice versa and a weak gradient would be going from black to black or white to white. This can be easily calculated by finding the absolute value when you subtract two adjacent pixel values.

Edges can now be easily found as they are defined by extreme changes in brightness or places with high gradients. It is important to note that these changes can happen in all directions, not just horizontally. So instead of only looking at our image as a matrix of values, we can also look at it as a continuous function of $f(x, y)$. This allows us to calculate gradients in all directions of the image to find strong gradients which again, are extreme changes in the values of adjacent pixels representing edges. This is also provided for us in a simple cv2 library call: cv2.Canny(blur, 50, 150).

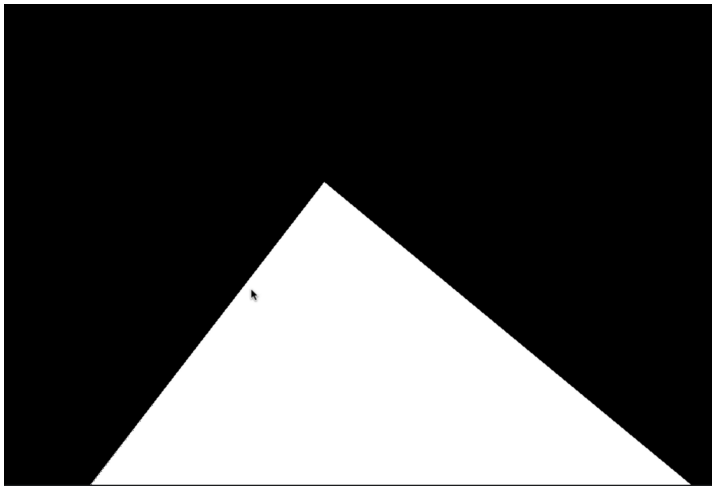
This image below demonstrates the ability to accurately capture all edges or pixels with high gradients.

Masking off Irrelevant Regions

While this information is valuable, it is a long way from isolating lane information. So the first thing we can do is a simple of task of masking off parts of the image we do not need. Using the car size, we can estimate the relevant space it will take up on the image. This can be done by plotting the image on any image editing software and looking for the relevant pixels. But to accomplish this in code, we can use the matplotlib library, load the image into a graph and estimate an area that is relevant.

$$f'(x, y)$$

This region of interest will be estimated and added in as a triangle with vertices at (200, 704), (1100, 704) and (550,250). Rather than hardcoding these values, we can make this code slightly more modular and adaptable to other images by getting the height of the image. Still, this applies well to cars of average size but trucks or larger vehicles will need additional changes to account for their size. We can now create a copy of our image and apply the triangle we created ontop of it before filling in the polygon white. After defining the triangle array of values in the variable triangle and the copy of the image in mask this can be done through the cv2 library call: cv2.fillPoly(mask, triangle, 255).



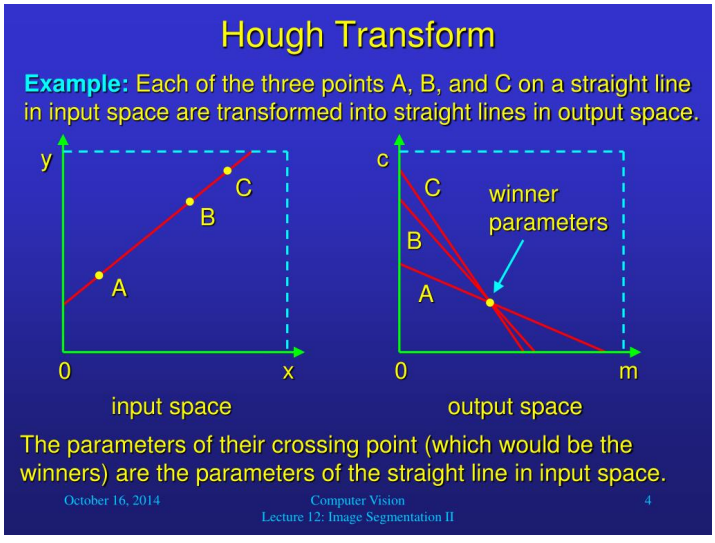
Now that we have another array image with white (or value 255) for each pixel that is relevant and black (value 0) for each pixel that is irrelevant. We can mask off our area of interest by simply doing a bit-wise AND operation.

$$0 \text{ (in binary)} = 00000000 \text{ and } 255 \text{ (in binary)} = 11111111$$

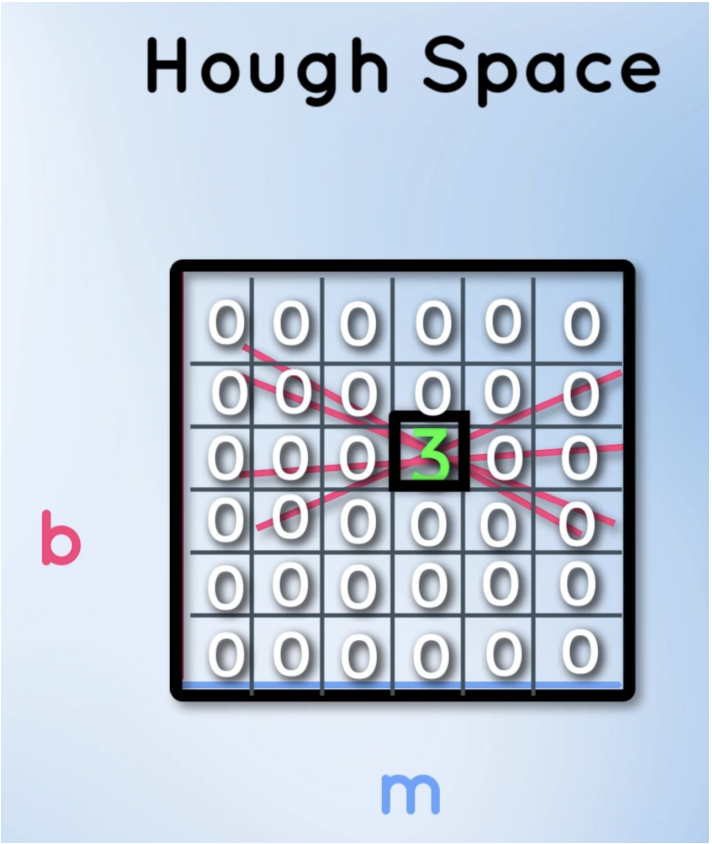
The bitwise AND operation takes two operands and performs an AND for each bit. This means that it results in 0 if either of the two bits are 0 and 1 only if both are 1. In practice, this will turn the values of all pixels in the black area to 0 or black while not affecting all pixels in the white area. This is done by using cv2.bitwiseand(image, mask).

Detecting Lines using Hough Transform

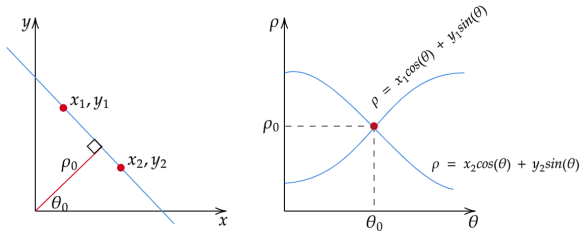
A popular application of hough transform is to detect lines. As an introduction to this method, let us consider modeling a line $y = mx + b$ in hough space as m and b where m is the slope and b is the y-intercept. Now, for any point in hough space, all the lines that intersect a point represent different points on that line. For us, the more lines intersecting a point in hough space will be a line of interest in the cartesian grid because it represents a line.

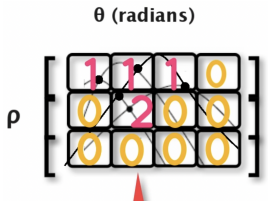


Now we can find the best candidate lines by creating "bins" or grids on the hough space graph and "voting". This process will segment the hough space into grids and then calculate the number of lines that intersect within it. The grid spots with the most intersections will be our lines, by taking the slope from m-axis and y-intersect from b-axis to create line y=mx+b on our image.



One clear issue with this method is that it is unable to account for vertical lines where the slope is infinite. So the Hesse Normal Form was proposed for more practical calculations which represents hough space as: $r = xcos(\theta) + ysin(\theta)$.

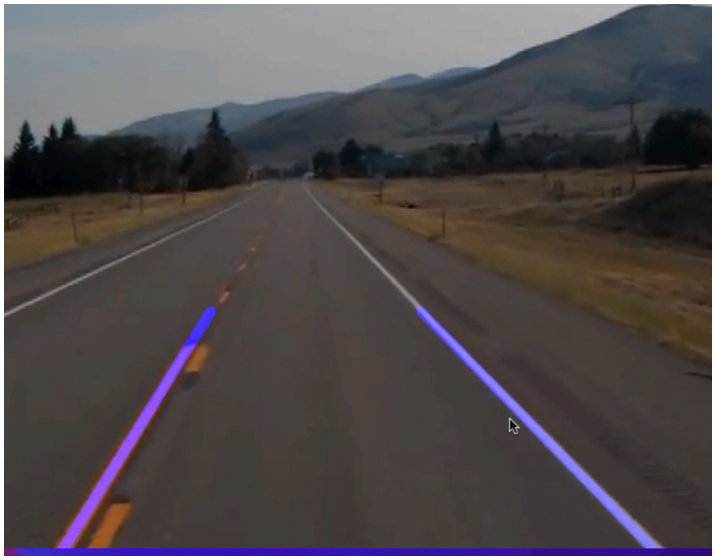




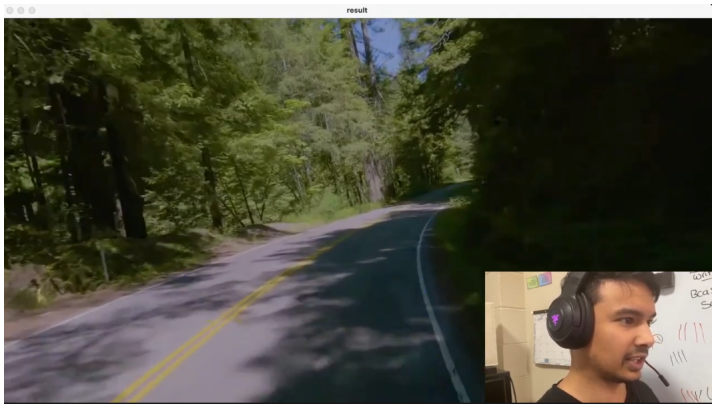
The smaller the bins the higher our precision, so we can choose 2px by 1px grids. The 1px needs to be converted into radians or $\pi/180$. We also need to set the threshold or minimum number of intersections to qualify a candidate line. After testing, we can set this to 100. Thus, we can accomplish this in code using cv2.HoughLinesP(image, 2, np.pi/180, 100, np.array([]), minLineLength=40, maxLineGap=5). We can now run through each line and draw it on a black image that we overlay over our original image. To overlay the two images, we must combine them using cv2.addWeighted(lane$_i$*mage*, 0.8, *line*$_i$*mage*, 1, 1).

Optimization and Final Product

We can add some optimizations because we know we are only looking to detect two lanes. So we can average the lines on the right and lines on the left to create two lanes. This is done simply by using np.polyfit((x1,x2), (y1,y2),1) which combines two lines and returns a linear function with a degree of one. Lines on the right and left can be told apart by their slope, this must flipped because the y-axis goes from height of the image up to 0. So confusingly, lines on the right have a negative slope and lines on the left have a positive slope. By averaging and combining the lines on the right and left we can produce cleaner lane lines.



Finally, we can apply this to every frame of a video to test the software. In this video I will test it on 3 different driving videos with different environments.



Link to video demo showing performance in different driving environments
Link to Lane Finding and Masking Python Code

Section 2.2: Decision Making Steering: and Behavior Cloning

For this section we will attempt to simulate the decision making sub problem of self driving by providing image data and using it to train a model through a neural network. To limit the scope of this, we will only try and optimize steering to keep the vehicle on the track and navigating properly. Thus, our data (x) will be the image driving files, the labels (y) will be the angle. Our goal is to create a model that will be able to accurately steer around a track using visual data. We can ignore other variables like throttle/speed and reverse and either zero them out or use the average value in our data.

The resources I will be using for this project the The Complete Self-Driving Car Course - Applied Deep Learning by Rayan Slim, the Udacity Unity-Based Self Driving Car Simulator, NVDIA's behavioral cloning architecture, called the NVDIA model, for the neural network architecture and the Stanford Convolutional Neural Network Course.

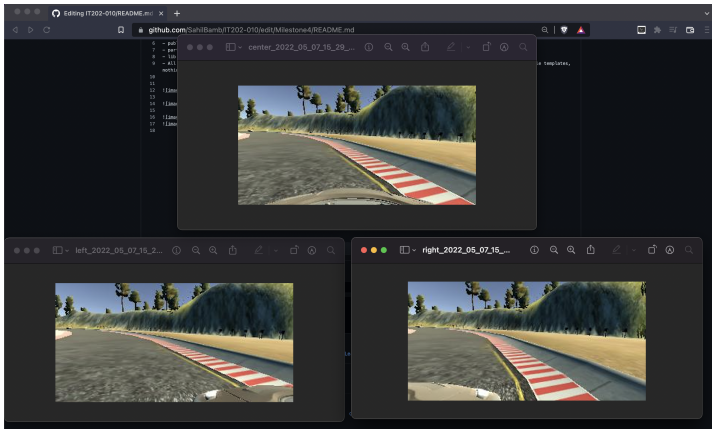
All the code in this section can be viewed in this collab notebook , where it can be run to simulate the same project. The page contains this entire document, so scroll down to the Beginning of Self Driving Car Behavioral Learning Simulation section. The simulation can also be reproduced, but requires downloading the Udacity Self Driving Car Simulation.

Collecting Data

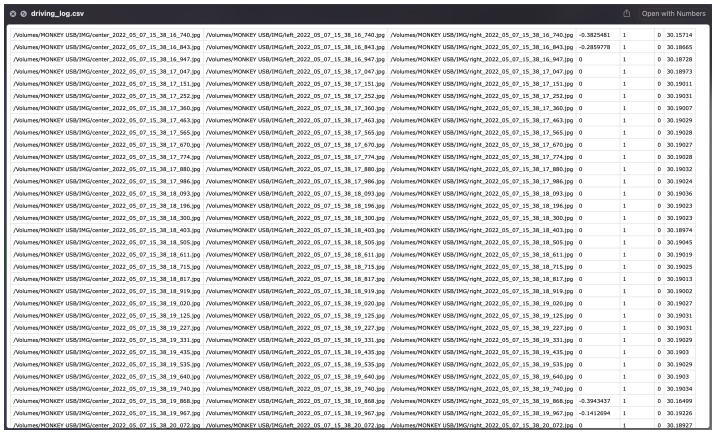
Without access to high definition mounted cameras, a device to record steering angles or even a car, it would be difficult to collect data. Luckily, Udacity's Nanodegree program on Self-Driving Cars provides a free simulation tool to collect data and test your model. This works by placing you into a Unity based game environment with a car on a track, which takes 3 pictures of your environment (left,

right and front) every second as well as recording your angle of steering. It records all of this within a csv document for our reference.

To collect data I drove around the track three times and then turned the car around and drove around three more times to avoid left or right turning bias. I aimed to stay in the middle of the lane for as long as possible.



The csv file data can very easily be downloaded and sorted into variables. The picture image needs a little more work, but the format they are saved in lends itself toward very easily differentiating them by separating the path and returning center, right and left for each screenshot.

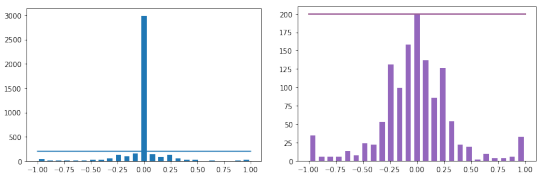


Data Visualization Preprocessing

After downloading the data, we can prepare it into a histogram to visualize any biases that may affect the model. We can do this with 25 bins as well as centering the steering data around 0. We will now plot each steering angle during simulation.

As soon as its prepared a glaring problem seems to be that there is far more 0 angle data than all the other data combined. This makes sense, as most of driving is the car going straight. Still this needs to be adjusted, and we can do this by limiting the range of values in each bin. We can pick a maximum value of 200.

When removing data for each steering angle, we must be careful not to remove them in order, as that will remove all driving data for a certain part of the drive (most likely the beginning of the drive). Thus, we utilize shuffling to shuffle the data each time before removing it. This will randomly remove data throughout the drive. Looking at the histogram after removing the 0 bias data, it looks acceptable without any outliers or biases so we can go forward.

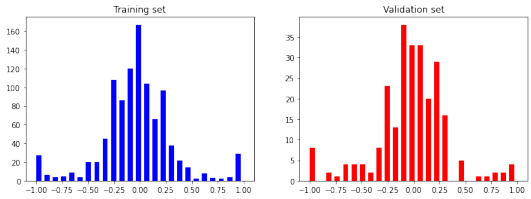


Splitting into Training and Validation Data

Now we must better organize the data by running through the data and getting the center, right and left images as well as the steering angle. To be clear, the data consists of a steering angle (the label) with three corresponding images (the input). We will load them in two two arrays with one being the images and the other its corresponding label.

Now with our data properly organized we can split our data into training and test data. Training data is the data we will run through, calculating our loss functions and aim to minimize through stochastic gradient descent. Test data is unseen data we will judge our model against after each epoch to make sure that the model is getting better against outside data as well as make sure it is not overfitting.The library function train_test_split does a great job at splitting our data into training and test data for us. Our x input data are images within x_train and x_valid and our y labels are steering angles within y_train and y_valid.

We have an estimated 1010 training samples and 253 validation samples. This is appropriate because we will need far more samples to train with then to test against. We should also visualize our train and validation data to make sure there is no bias or outliers.



Further Preprocessing

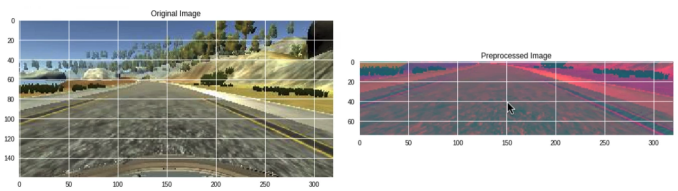
First we can mask off, or rather crop, a certain part of our image as it will not be relevant to determining our steering angle. This will be the sky, and we can remove the top of the image or approximately removing all pixels not within 60 to 135 pixels.

Later, we will be using the NVDIA model for the architecture of our neural network. NVDIA architects recommend using the YUV color space when using their NVDIA model (Y is luminosity while UV represents chromiums which adds color). We can convert our images to this color space with cv2.COLOR_RGB2YUV

We will also be using guassian blur because it reduces noise, prevents false edges and helps find features. Finally we will reduce the size of our image to increase the efficiency of our model as larger images will take more time to process.

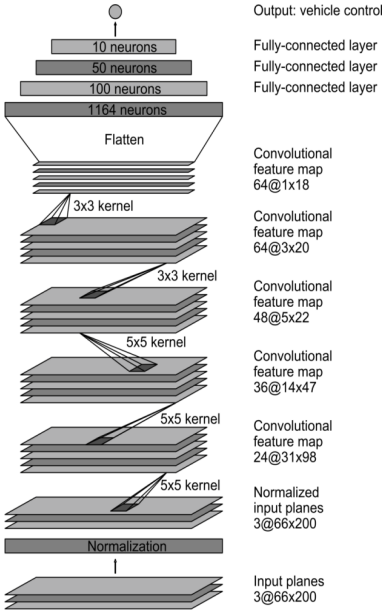
The visualization of the original image and preprocessed image shows the changes made to the image. This will be applied to every image in our data to make them preferable for our training task. We can run through our data with a map applying our function to the array to accomplish this.

After preprocessing, the shape of our data is (1010, 66, 200, 3) or 1010 images, each 66x200 with 3 color channels (YUV).



Convolutional Neural Network Architecture

For the Neural Network architecutre, we will be using the popular NVDIA model that has been shown to be effective for behavioral cloning. Most neural network architectures are not developed from scratch and are borrowed from existing proven models invented by researchers. Still, many need to be altered after applying them to specific the data to correct for overfitting or lack of convergence.



The first hidden layer will be a convolution layer with 24 filters, a 5x5 kernel, a subsample (stride length) of 2x2. The subsample / stride length means that it will convolve 2 pixels at a time horizontally and 2 pixels at a time vertically (skipping a pixel each time). Also we can ignore padding as there is not much relevant data at the edges of our driving images. Being our first layer we also need to specify the input shape (66, 200, 3). And we will use the ReLU activation function.

The second convolution layer has 36 filters, 5x5 kernel, 2x2 subsampling and the same activation function. The third layer is also a convolution layer with 48 filters.

After that, the third and fourth convolution layers reduce their kernal size to 3x3. Due to all the convolutions we can also decrease the subsample or stride length back to 1x1 as its dimensions have decreased enough to justify the change.

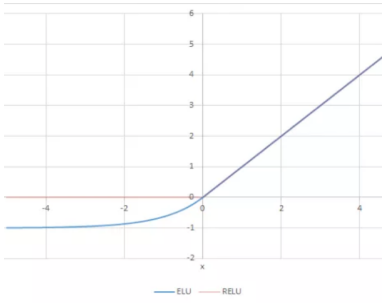
The flatten layer will is important to take our output array and convert into a one dimensional array so that it is able to be fed into the fully connected layers that follow. We then have three dense layers with 100, 50 and 10 nodes with the same ReLu activation functions as all of our other layers. Our final layer will be a dense layer with a single output node, to output the predicted steering angle. For our loss function we will be using the mean squared error and use the popular Adam optimizer.

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_5 (Conv2D)           (None, 31, 98, 24)        1824

 conv2d_6 (Conv2D)           (None, 14, 47, 36)        21636

 conv2d_7 (Conv2D)           (None, 5, 22, 48)         43248

 conv2d_8 (Conv2D)           (None, 3, 20, 64)         27712

 conv2d_9 (Conv2D)           (None, 1, 18, 64)         36928

 dropout_2 (Dropout)         (None, 1, 18, 64)         0

 flatten_1 (Flatten)         (None, 1152)              0

 dense_4 (Dense)             (None, 100)               115300

 dropout_3 (Dropout)         (None, 100)               0

 dense_5 (Dense)             (None, 50)                5050

 dropout_4 (Dropout)         (None, 50)                0

 dense_6 (Dense)             (None, 10)                510

 dropout_5 (Dropout)         (None, 10)                0

 dense_7 (Dense)             (None, 1)                 11

=================================================================
Total params: 252,219
Trainable params: 252,219
Non-trainable params: 0
_________________________________________________________________
```

Model Training and Issues

We trained our model with our data with 30 epochs and a batch size to 100. 30 is a standard epochs and 100 is used because it is about 10 percent of our total training data (1010).

Both overfitting and lack of convergence were issues with the data after training the model. After using the ReLU (Rectified Linear Activation Function) function in each layer the data did not converge or the loss did not decrease approaching zero as we increased in epochs. One reason for this may have been the fact that ReLU is a peicewise function that is linear for positive values and eliminates negative values by turning them into 0. This sometimes results in the issue of the 'dead ReLU' when one node turns to 0 and only outputs 0 values to the following nodes. The ELU activation function has many of the same benefits of ReLU and does not suffer from zeroing out or dying. After replacing all of our ReLU activation functions with ELU, our loss converges toward zero as desired.

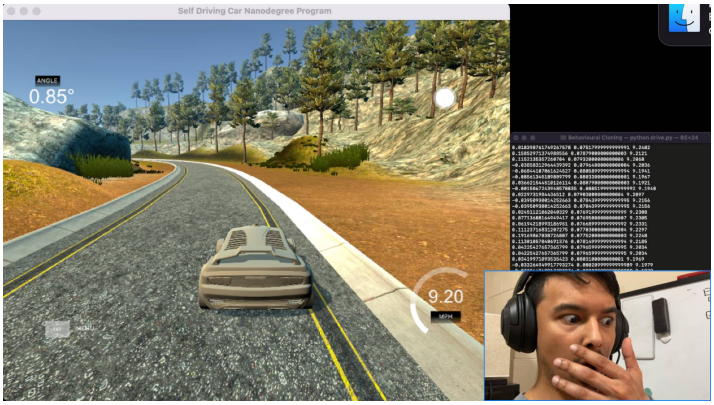


Another issue is that there is slight overfitting near the last few epochs of the model. To fix this we added additional dropout layers after the convolution and the first dense layer. The dropout layer randomly zeros out nodes at a set frequency rate (our rate is 50 percent) to stop the model from overfitting.

Testing Our Trained Model

We can download our model and test it on our simulation. There was some additional set up necessary to use flask to connect python to the simulator program's socket. As this is tangential to our task I will not go into detail to describe how this was set up.

Below is a video of the completed model driving by itself.



Link to video demo showing performance on tracks
All the code in this section can be viewed in this Collab Notebook, where it can be run to simulate the same project. (scroll down to Beginning of Self Driving Car Simulation).

3 Conclusion and Works Cited

Conclusion

This was a useful exercise in learning the underlying concepts behind the self driving car problem by trying to implement its sub-problems. While the results for both experiments were not exemplary, it is a testament to how truly difficult optimizing these algorithms in the real world is. Though I believe given more time and more adjustments to both, passable results could be reached. The solution to this problem has such lifesaving and lucrative implications it has hundreds of companies spending billions of dollars chasing the prize. While there have been low level solutions with driver monitored self driving, the larger problem has defied every attempt at being solved. Still, with the massive leaps in progress being made and the relative nascent nature of the deep learning field, I believe we will see a self driving car within our lifetime.

Resources Used

- Neural Networks and Self Driving https://medium.com/geekculture/neural-networks-and-self-driving-car-without-any-complex-mathematics-f8dcde56441c
- CS231: Deep Learning for Computer Vision http://cs231n.stanford.edu/
- Panetelis Data Science Course https://pantelis.github.io/cs301/docs/common/lectures/optimization/sgd/
- Linear Regression using Gradient Descent https://towardsdatascience.com/linear-regression-using-gradient-descent-97a6c8700931
- Self Driving Report https://www.irjet.net/archives/V7/i5/IRJET-V7I51383.pdf
- Udemy: The Complete Self-Driving Car Course - Applied Deep Learning https://www.udemy.com/course/applied-deep-learningtm-the-complete-self-driving-car-course/
- Self Driving Cars Outline (CNN) https://neptune.ai/blog/self-driving-cars-with-convolutional-neural-networks-cnn
- History of Self Driving Cars https://www.thoughtco.com/history-of-self-driving-cars-4117191
- Every Car Doing Self Driving https://www.greyb.com/autonomous-vehicle-companies/
- Top 33 Companies Doing Self Driving https://algorithmxlab.com/blog/worlds-top-33-companies-working-on-self-driving-cars/
- List of Countries By Traffic

- Car Accident Stats https://www.cdc.gov/winnablebattles/report/motor.html
- Decoding Open AI https://medium.com/@chengyao.shen/decoding-comma-ai-openpilot-the-driving-model-a1ad3b4a3612
- Comparing Human Driving and Self Driving Using Professional Driving Simulator https://www.ncbi.nlm.nih.gov/pm
- The WIRED Guide to Self-Driving Cars https://www.wired.com/story/guide-self-driving-cars/
- Implementing NVIDIA Neural Network https://towardsdatascience.com/implementing-neural-network-used-for-self-driving-cars-from-nvidia-with-interactive-code-manual-aa6780bc70f4
- Self Driving Car Simulation https://gamermusic4.itch.io/self-driving-car
- Car Crash Fatality https://www.nhtsa.gov/press-releases/2020-traffic-crash-data-fatalities
- Beginner's Guide to Deep Reinforcement Learning https://wiki.pathmind.com/deep-reinforcement-learning
- NVIDIA End-to-End Deep Learning https://developer.nvidia.com/blog/deep-learning-self-driving-cars/
- Neural Networks Deep Learning Explained https://www.wgu.edu/blog/neural-networks-deep-learning-explained2003.html
- A Beginner's Guide to Convolutional Neural Networks (CNNs) https://wiki.pathmind.com/convolutional-network
- Department of Energy on AVs https://www.nrel.gov/docs/fy13osti/59210.pdf
- Commuting Cars https://www.washingtonpost.com/business/2019/10/07/nine-days-road-average-commute-time-reached-new-record-last-year/
- Relu Activation Function Explanation https://iq.opengenus.org/relu-activation/
- Backpropogaion Explained at 5 Levels https://medium.com/coinmonks/backpropagation-concept-explained-in-5-levels-of-difficulty-8b220a939db5
- Backpropogation Intiution Explained https://towardsdatascience.com/backpropagation-intuition-and-derivation-97851c87eece
- Understanding Backpropogation https://towardsdatascience.com/understanding-backpropagation-algorithm-7bb3aa2f95fd
- Hidden Layer Explained
  https://www.baeldung.com/cs/hidden-layers-neural-network