CS230

Let's Drive: Learning Autonomous Driving through Behavior Cloning

Mario Srouji
Department of Computer Science
Stanford University
msrouji@cs.stanford.edu

Jenna Lee
Department of Engineering
Stanford University
yunjlee@stanford.edu

Abstract

Autonomous driving has become one of the biggest challenges in the field of Artificial Intelligence and Machine Learning. Many companies and start-ups alike are taking up the challenge to build the first fully autonomous vehicles, and deploy them on the road. However not only is autonomous driving an extremely difficult problem to solve, but it is also one that could transform the way we view transportation as a whole. We decided to explore this high-impact field through the use of deep learning. In this work, we train a Convolutional Neural Network (CNN) to map raw pixels from 3 different camera angles directly to a steering angle. We used Udacity's self driving simulator, which is a physics-based simulator that allows you to steer a car around a track to collect data, and test your implementation. Our CNN uses image data and steering angles to clone the behavior of a human driver, who generates the data by driving around the track. We do not explicitly train the model to detect explicit decomposition of the problem, such as lane marking detection, path planning, and control. We think that this end-to-end approach will lead to better performance, due to the fact that we are not optimizing human-selected intermediate criteria (like lane detection - which doesn't guarantee maximum system performance), as our system attempts to optimize the objective as a whole.

1 Introduction

Convolutional Neural Networks (CNNs) [1] have not only revolutionized pattern recognition and computer vision [2], but have also opened up a wide-array of applications in deep learning. Before the use of CNNs, most computer vision tasks were performed using an initial stage of hand-crafted feature extraction, followed by a classifier. The main difference benefit of CNNs is that the features of the input images are learned automatically from training examples. This is especially powerful in image recognition tasks because the convolution operation captures the 2D nature of images, and since kernels are used to scan the entire image, relatively few parameters need to be learned.

In this paper we use a CNN to learn a functional mapping from camera images, to the steering angle needed to navigate an automobile. Autonomous driving has become one of the biggest challenges that the automotive industry has faced, and many companies are taking up the challenge to build the first fully autonomous vehicles. A classical approach to solving this problem includes training the model piece-by-piece, by first training the model to detect lanes, then to follow the lane, etc... However in this work, we would like to take a more holistic approach by training the model to learn 'safe driving' as a whole feature, instead of learning each necessary component piece by piece. Previous work has already showed the potential of end to end learning for self driving solely based on videos, and we leverage these results and apply them to our driving simulator.

Our method involves learning to drive autonomously by cloning the behavior of a human driver in a simulator. The driver provides the steering angles necessary given three different camera views on the vehicle (front, right side, left side). Our model will have to learn how to steer the vehicle appropriately on a simulated track.

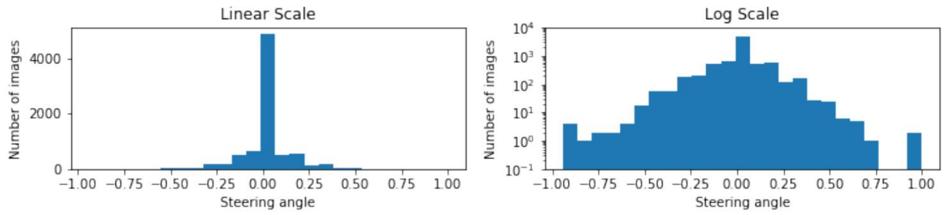
Learning to drive safely is not easy. While there are obvious contextual cues, such as abrupt changes in steering angle, and driver-induced abrupt deceleration, video recording of driving doesn't necessarily provide such numerical data. However this is exactly where deep learning can excel, as the criteria of being safe/not safe is often subtle, and is hard to extract from numerical data alone. The challenge lies in training the CNN to be able to understand certain features of driving without being explicitly told to do so (for example to avoid crashing into obstacles). There are intrinsic difficulties in learning from labeled visual data, such as generalization of learned behavior, and data cleaning/augmentation that is necessary to allow the CNN to properly learn the desired behavior.

2 Dataset and Features

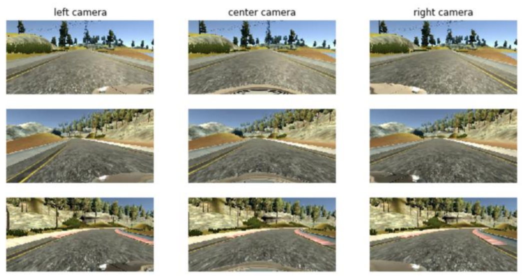
Our dataset and simulator are provided by Udacity's delf driving simulator, which was built for Udacity's Self-Driving Car Nanodegree, to teach students how to train cars to navigate road courses using deep learning. The data includes three camera frames from different angles of the car, and the associated steering angle that the driver performed for that given view. The data can be gathered with the Udacity simulator itself. When the simulator is set to training mode, the car is controlled by the human though the keyboard, and frames and steering directions are stored to disk as supervised learning examples. A link to the simulator's GitHub is the following: https://github.com/udacity/self-driving-car-sim.

We collected more data using training mode in the simulator. In training mode the simulator produces three images per frame corresponding to the left, right, and center mounted cameras. The simulator also produces a (.csv) file which includes file paths for each of these images, along with the associated steering angle, throttle, brake, and speed for each frame. Our code base loads in the file paths for all three camera views for each frame, along with the steering angle (adjusted by +0.25 for the left camera frame and -0.25 for the right to normalize the perspective), into two numpy arrays of image paths and image angles (as the inputs X and ground truth Y for our data). Images produced by the simulator in training mode are 320x160, and therefore require pre-processing prior to being fed into the Nvidia CNN because it expects input images to be of size 200x66. We crop the bottom 20 pixels and the top 35 pixels (because they mostly include the sky and the ground) from the image, and then resized the image to 200x66.

The final dataset is a combination of Udacity's released dataset, and the data that we generated. We have a total of 17,350 data points. We split our data into training and validation data sets, using 80% as training and 20% as validation. We use the validation set to test for over-fitting, and adjust hyper-parameters.



An issue that arose right away with the data is that the vehicle in most of our simulated scenarios is moving straight, and therefore our data is highly biased towards a steering angle of 0. We quantitatively visualized this phenomenon by displaying the frequency of steering angles in the data (in the charts above). The data is highly centered towards the middle steering angle of 0. To overcome this, we augment our data using the following techniques (to create more diverse samples): cropping, perspective transformation, image flipping, image blurring/Gaussian noise, image rotation, change in brightness, and image translation.

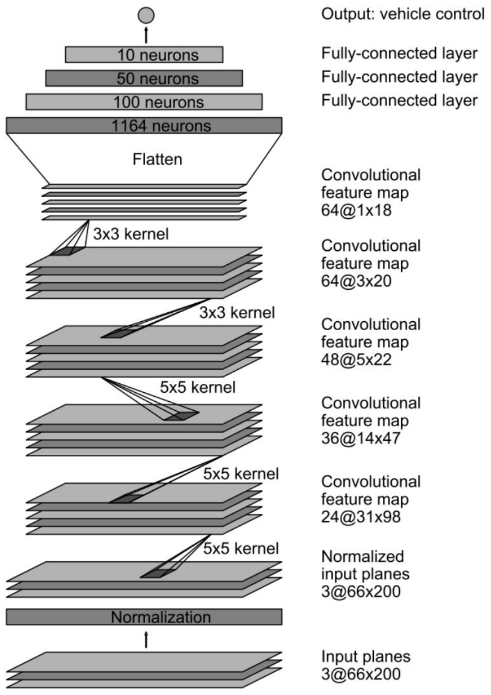


To implement these augmentation techniques strategically, we implemented a data generator to select images at random and apply appropriate augmentation techniques for the given selected images. Additionally we generated more data from the simulator, specifically for situations with curvy roads and tight curves. To avoid erratic steering we apply Gaussian smoothing to the steering data in order to best mimic true steering transitions. We also normalize all of the image pixel values to be between -1 and 1. This proved to effectively reduce convergence time during training.

3 Methods

In order to learn how to clone a human driver, we use a Convolutional Neural Network. The specific CNN we implemented is based on The Nvidia model [3], which has been proven to work in this problem domain. Since the Nvidia model is well documented, we were able to focus on adjusting the training images to produce the best results, with only minor adjustments to the model to avoid over-fitting. We also add non-linearity to improve the prediction.

A CNN is able to successfully capture the Spatial and Temporal dependencies in an image through the application of relevant kernels or filters. The objective of the convolution operation is to extract the high-level features such as edges, from the input image. CNNs need not be limited to only one convolutional layer however. Conventionally, the first convolutional layer is responsible for capturing the low-level features such as edges, color, gradient orientation, etc... With added layers, the architecture adapts to the high-level features as well, and begins to learn more complex features (such as obstacles on the road, road curvatures, etc...).



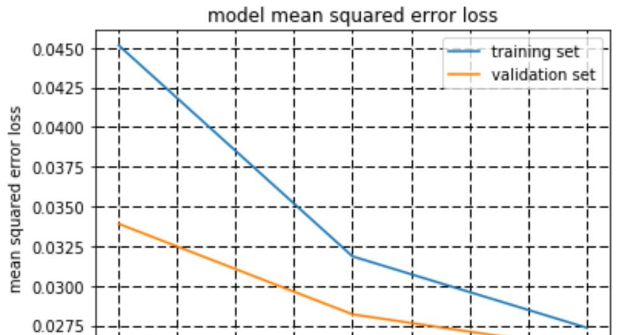
We train the weights of the network to minimize the mean squared error (MSE) between the steering angle output of the network, and the actual steering angle inhibited by the human driver. The network consists of 9 layers, including a normalization layer, 5 convolutional layers, and 3 fully connected layers. The input image is split into YUV planes and passed into the network. The first layer of the network performs image normalization. The normalizer is hard-coded and is not adjusted in the learning process. As noted in the Nvidia paper [3], the convolutional layers were designed to perform feature extraction and were chosen empirically through a series of experiments that varied layer configurations. They use strided convolutions in the first three convolutional layers with a (2, 2) stride and a (5, 5) kernel and a non-strided convolution with a (3, 3) kernel size in the last two convolutional layers. Following the five convolutional layers, we have three fully connected layers leading to an output node, which is the steering angle for the vehicle. The fully connected layers are designed to function as a controller for steering, but we note that by training the system end-to-end, it is not possible to make a clean break between which parts of the network function primarily as a feature extractor, and which serve as a controller.

We define an epoch as one whole pass through the training data. Our training consisted of running Mini-Batch Gradient Descent for 5 epochs. We found this to be a good number of epochs to avoid over-fitting the training data, while also reducing the training error and validation error. We used an Adam optimizer with a learning rate of .0001 which is smaller than the default value. We used a smaller learning rate because we noticed that with the default value, the validation loss stopped improving fairly quickly, and hence we were not able to make proper steps towards a better optima. Our batch size was 32 data points per batch. We used a MSE (Mean Squared Error) loss function to minimize the difference between the output of our CNN, and the true steering angles for a given image. We decided upon this loss function because it is very efficient in regression type problems (which in our case we are trying to fit our model to output steering angles close to that of the human driver).

4 Results

We evaluated our CNN's performance in two ways: quantitatively and qualitatively.

Quantitatively we tracked the MSE on both the training and the validation set for 5 epochs. What we found is that we were able to get our training error down to about 0.025, and our validation error to about 0.022. This demonstrates both low bias (getting very close to human error and best achievable error), and low variance (good generalization on unseen data). We chose to stop training the model at 5 epochs, because we noticed that our model began to over-fit to the training data with each additional epoch, with little gain in performance on the training set. We show the training plot for 2 epochs below to show how quickly our model was able to converge to a good solution.



In addition to tracking the loss, we estimate what percentage of the time the CNN could drive the car fully autonomously without human intervention. This metric is determined by counting simulated human interventions, which occur when the simulated vehicle departs from the current line it is driving on of the track by more than 1.5 meters (to simulate a road lane encroachment). Since the simulator runs at a default FPS (frames per second) of 60, we decided to track the trained vehicle's trajectory for a whole run around the track, measuring how many times an intervention was required.

We group 120 frames together (to simulate 2 seconds of driving) to determine if the car left the line it is following by more than 1.5 meters. We assume that in real life an actual intervention would require a total of six seconds to complete (retake control of the vehicle, re-center it, and then restart the self-steering mode). We calculate the percentage of autonomy by counting the number of interventions and multiplying by 6 seconds, dividing this by the elapsed time of the simulated test (which is number of frames that occurred in the test divided by 60). We then subtract this fraction from 1 to get the fraction of time that the trained car drove fully autonomously. The equation can be summarized as: $1 - \frac{interventions * 6seconds}{totaltimeelapsed}$. We performed this test on 10 separate full runs on the simulated track, and averaged the results. What we found was that our trained CNN was able to drive fully autonomously for about 86% of the time, which is a good result, but can use improvement. We believe that this number is not as high as it can be because the learned driving behavior is not as smooth as we would like (the car tends to swerve side to side / jitters along the driving line at certain times). We discuss this qualitative observation below.

Qualitatively we observed the following behaviors after training. Firstly, the car drives well on the track it was trained on, but performs rather poorly on a second track that it was not trained on. This motivates including data from multiple different tracks in both the training and validation sets. We would have liked to explore this change given more time. Additionally, we observed that the learned behavior of the vehicle tends to not be as smooth as a human driver. Our data augmentation and smoothing techniques help, however the car still jitters around the driving line. The best solution to this would have been to include a "smoothness" notion into the loss function itself, to better optimize for smooth driving during training. We would have liked to explore this given more time.

5 Conclusion, Discussion, and Future Work

We have used CNNs to demonstrate that these ML models are able to learn the entire task of autonomous driving (through behavior cloning), without manual decomposition into road or lane marking detection, semantic abstraction, path planning, or control. A relatively small amount of training data was sufficient to train the car to operate quite well on a simulated track, even on camera frames that were not seen during training. The CNN was able to learn meaningful road features from a very sparse training signal (steering angle alone).

For example, the CNN learns to detect the outline of the road (including turns and curvatures) without the need for explicit labels during training. Despite the good results, more work is needed to improve the robustness of the network, and to find methods to verify the robustness. Additionally, we need a method to allow the CNN to not only generalize well on unseen examples for a specific track, but also on unseen tracks.

According to the MSE, our model worked well. However it didn't perform as smoothly as expected when we tested the model visually using the simulator. So this is a clear indication that MSE is not the best loss function/metric to use when assessing the performance of the model in this domain. For future work, we would like to experiment with looking at a historical sequence of camera frames to make steering decisions, versus simply looking at a single time frame. This is because when we are driving a car, our actions such as changing steering angles, and applying brakes are not just based on instantaneous driving decisions. In fact, current driving decisions are likely based on the past few seconds of driving. Hence, it would be really interesting to see how Recurrent Neural Networks (RNNs) could be applied in this problem domain. Additionally it would be interesting to see how we could build a Reinforcment Learning agent to learn how to drive autonomously in this same setup.

6 Code

Code is available at (and has been shared with the CS230 staff): https://github.com/msrouji/CS230-Final-Project

7 Contributions

Both team members contributed equallty on all delivarables of the project (code, final paper, and poster).

References

[1] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. Backpropagation applied to handwritten zip code recognition. Neural Computation, 1(4):541–551, Winter 1989. URL: http://yann.lecun.org/exdb/publis/pdf/lecun-89e.pdf.

[2] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, Advances in Neural Information Processing Systems 25, pages 1097–1105. Curran Associates, Inc., 2012. URL: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.

[3] M. Bojarski, D.D. Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L.D. Jackel, M. Monfort, U. Muller, J. Zhang, X. Zhang, J. Zhao, K. Zieba, End to End Learning for Self-Driving Cars, 2016.

[4] Udacity Simulator and Starter Code: https://github.com/udacity/CarND-Behavioral-Cloning-P3