CS 490 Project Report:

Implementation of a Convolutional Neural Network and Visualization Tool for Learning of Steering Angles in Self-Driving Cars

Vincent Huang (vwh5)

Advisor: Man-Ki Yoon

Faculty Supervisor: Zhong Shao

Abstract

As the self-driving car industry grows and vehicle autonomy plays a larger and larger role in mainstream cars, safety becomes an increasingly important issue. The FLINT group at Yale has established Y Driving, a self-driving car lab with the purpose of researching new methods in autonomous vehicular safety, navigation, and deployment. This semester, I explored solutions for learning steering angles that could be used for the lab's model cars, and I implemented a convolutional neural network (CNN) based on the NVIDIA team's seminal paper. Neural nets are commonly critiqued for their black boxes opacity, and a major area of machine learning today seeks to increase their transparency. With safety such a critical issue, it is important to maintain a level of transparency in neural nets involved in autonomous vehicles. To accomplish this, I built out an implementation of VisualBackProp, a visualization technique also designed by the NVIDIA team, used to visualize which pixels and features in an input image contribute most to the predictions made by a CNN. I adapted my CNN and VisualBackProp implementations to run independently on Udacity's self-driving car simulator and on real world street data. The systems are generalized such that any data set can be used to train the CNN model used to generate predictions on steering angles, after which the VisualBackProp program can be used to identify the key activation pixels in any input image for the neural net. As such this solution can be easily ported to work on driving data collected from Y Driving's model cars to autonomously steer the vehicles and generate visual representations of the net's decision-making processes.

Background

In my initial project proposal submitted at the start of the semester, I planned to build out the basic driving functionality of Y Driving's model cars such that they could drive autonomously while properly performing lane following, stopping at traffic signs, object avoidance, and lane changing. This proved challenging for two primary reasons. First, the cars' embodied nature made it sensitive to the physical conditions in the environment. All sensory input was noisy, and solutions required an adaptivity that was difficult to design and encode manually. Second, the scope of the project, in needing to properly exhibit four different behaviors, proved somewhat too large for just one semester's worth of work.

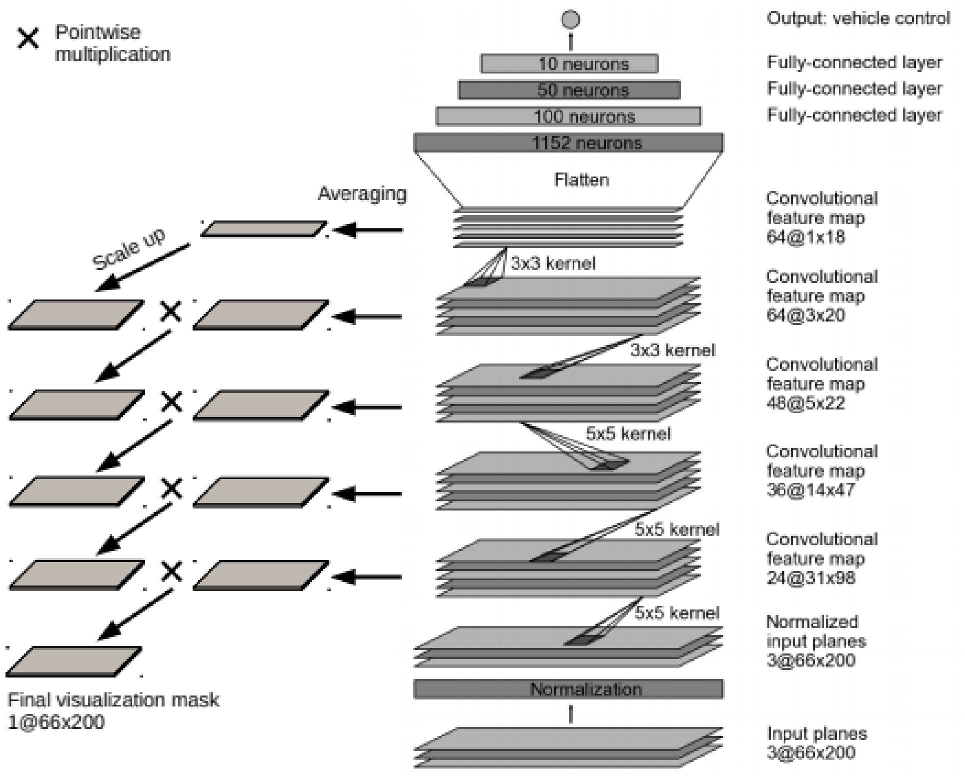
As such, I turned my attention to searching for an end-to-end solution, which I recognized would address these issues that I've described. By opting for an end-to-end solution, I pass the job of encoding key features and details to the neural net itself. For example, one distinct challenge of accomplishing lane following is the variation in light conditions that could affect perception and recognition of lane dividers. In shifting to an end-to-end solution, the programmer only needs to provide a sufficient amount of raw data and augmentation to produce a solution that will be able to generalize and account for the spectrum of light conditions that could be encountered. The difficulties of understanding the underlying perceptual mechanisms are bypassed. Meanwhile, the system is able to encompass a range of individual behaviors as long as those behaviors are exhibited in the training data. Simply by training the network using a reasonable dataset, the system can learn lane-following and object avoidance simultaneously (potentially as well as stopping, if a velocity factor is included). End-to-end solutions circumvent the need to implement separate modules for different capacities, as all of them are all encoded in the network after training.

However, one of the main drawbacks of end-to-end solutions is their opacity. With their intermediate states never being made explicit, they do not readily offer insight into how they both succeed and fail. Given the high stakes that the act of driving involves, it's critical that we are able to understand what's going on within such a system. Advancement of autonomous vehicular technology hinges on our ability to improve through analysis and evaluation, which visualization may help with. These considerations were what inspired the choice to VisualBackProp [2, 3] to generate insight into the hidden workings of our car's neural net.

End-to-end solutions for learning steering are not broadly new. ALVINN (Autonomous Land Vehicle in a Neural Network) [4] was a system built in 1989 that demonstrated the potential for an end-to-end neural network to steer a car. In developing their new system, named both PilotNet and DAVE-2 [1], the NVIDIA team leveraged the vast technological advancements in hardware and software that have arisen over the last three decades to improve on models that have come before them. The system had remarkably good performance with minimum training data. The authors posited that "better performance will result because the internal components self-optimize to maximize overall system performance, instead of optimizing human-selected intermediate criteria" [1].

Models

The architecture for my implementation of the neural net is nearly identical to that described in the NVIDIA team's paper, which is depicted in Figure 1. The described network consists in order of one input normalization plane, three convolutional layers with kernel size 5x5 and stride 2x2, two non-strided convolutional layers with kernel size 3x3, and finally three fully-connected layers. The one modification I made was to the kernel sizes used in the strided



× Pointwise multiplication
Output: vehicle control
10 neurons
Fully-connected layer
50 neurons
Fully-connected layer
100 neurons
Fully-connected layer
1152 neurons
Flatten
Averaging
Scale up
Convolutional feature map 64@1x18
3x3 kernel
Convolutional feature map 64@3x20
3x3 kernel
Convolutional feature map 48@5x22
5x5 kernel
Convolutional feature map 36@14x47
5x5 kernel
Convolutional feature map 24@31x98
5x5 kernel
Normalized input planes 3@66x200
Normalization
Input planes 3@66x200
Final visualization mask 1@66x200

Figure 1: CNN Architecture (right) and VisualBackProp Method (left) [3]

layers which I'll justify during my discussion of the VisualBackProp architecture. The designation of the architecture, training of the model, and VisualBackProp was all performed in Keras. However, whereas VisualBackProp was run in Keras 2, training proceeded much more quickly in Keras 1 and so was conducted using that instead.

The training and prediction processes for the network are depicted in Figure 2. Training data includes one or more images taken by front-facing cameras, paired with a steering angle. In my implementation, the data is augmented using a variety of random translations, flips, brightness adjustments, and additions of shadows. These augmentation methods were adapted from Naoki Shibuya's implementation (see acknowledgements). Given the augmented data, the CNN undergoes standard supervised learning using backpropagation. Once trained, the system

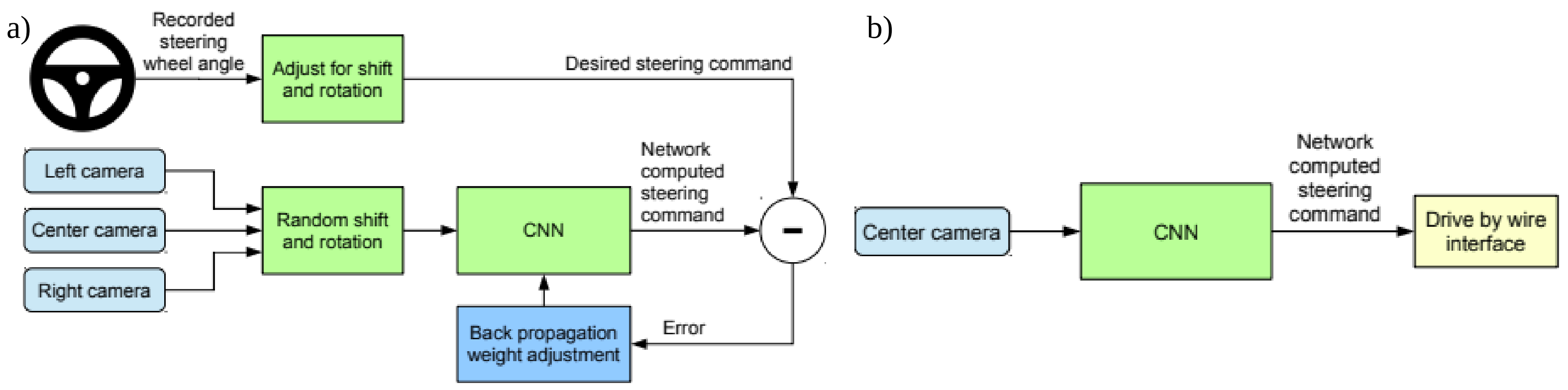

a)
Recorded steering wheel angle
Adjust for shift and rotation
Desired steering command
Left camera
Center camera
Right camera
Random shift and rotation
CNN
Network computed steering command
Back propagation weight adjustment
Error
b)
Center camera
CNN
Network computed steering command
Drive by wire interface

Figure 2: a) Training DAVE-2 / PilotNet b) Generation of steering commands using trained network. [1]

can be tested either using data that was previously separated from the training data, or images that are generated live from a simulator or real-life driving scenario.

The VisualBackProp process is depicted in the left half of Figure 1. From a broad perspective, VisualBackProp produces a representation of each convolutional layer's state via its feature map, then relies on deconvolution and multiplication to aggregate this information to determine which sets of pixels are most potent across the layers. More specifically, the process occurs as such: the final convolutional layer's feature map is averaged (depth becomes 1) and undergoes a transposed convolution (synonymous with deconvolution; however, the effect is not the exact reverse of convolution) so that results dimensions scale up and match those of the penultimate layer's averaged feature map. These two entities are multiplied pointwise. The resulting mask (the same dimensions as the latter's averaged feature map) is in turn deconvolved and multiplied by the next previous layer's averaged feature map, and this process repeats until the final multiplication, involving the first convolutional layers, occurs. The result of this operation undergoes one final deconvolution such that it matches the dimensions of the input image, and this final product is the mask that highlights the key pixels in the image.

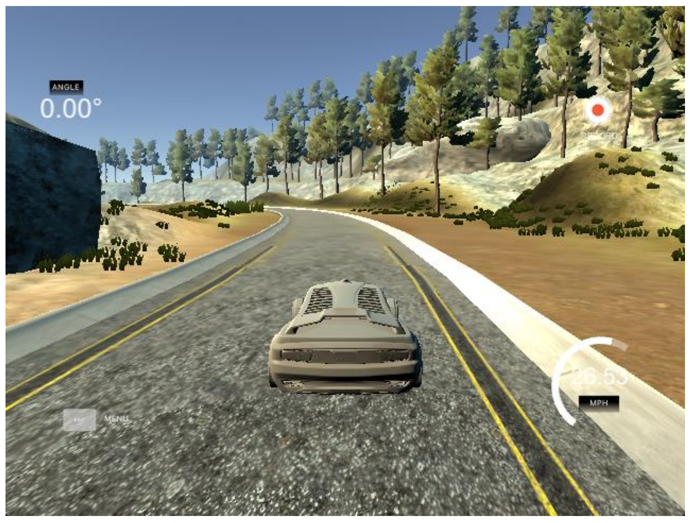
A major stumbling block that I faced while working on the project involved the setting of dimensions in the deconvolution operation. For a long period, I struggled to understand why the output of a deconvolution operation was producing dimensions that were one off (i.e. (13, 47, 1) vs. (14, 47, 1)) from the dimensions of the following layer. Ultimately, the issue existed in the fact that while convolution shrinks an image and does so in a deterministic way, the reverse operation in deconvolution fails to if rounding previously occurs. The output dimension (height or width) of a non-padded convolution operation is

```
floor((input – kernel) / stride) + 1
```

With a stride of two or greater, multiple distinct input dimensions can thus produce the same output dimension. For deconvolution, then, it becomes ambiguous what the output dimensions (i.e. input dimensions of the corresponding convolution) should be. Yet even after arriving at the problem, I had issues circumventing it. While the deconvolution method in Keras 1 required an explicit specification of the output dimensions, Keras 2 removes this syntax altogether and calculates it based on the input, kernel, and stride parameters. The formula is as such:

```
dim_size * stride_size + max(kernel_size - stride_size, 0)
```

Given the deterministic output of this formula, it is in fact impossible to arrive at certain output dimensions in a strided deconvolution operation. For example, with stride 2 and kernel size 5, convolutional inputs of 13 and 14 both result in an output of 5. However, 5 fed into the

a)


b)

Figure 3: Self-Driving Car Sim a) Data Collection Phase b) Autonomous Driving

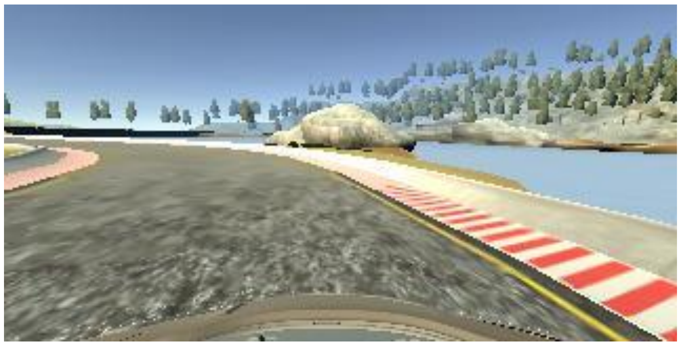
corresponding deconvolution will only produce 13, which was the source of my problem. After a multitude of attempts at trying to modify the neural net's input dimensions to fit properly, I finally realized that I could use a rectangular kernel for strided layers. That way I could always get to the values did not end up being rounded. This was the reason for the single architectural modification to the NVIDIA network's model.

Results

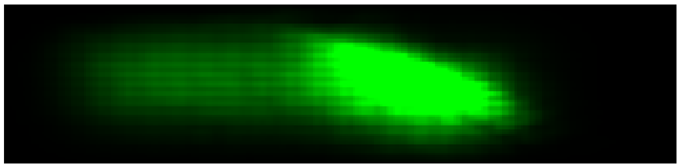
The network was first trained and tested on Udacity's self-driving car simulator. The data collection and testing processes are shown in Figure 3. Data collection very closely mirrored that outlined in the NVIDIA paper, with three individual front-facing cameras (center, left, and right) taking images simultaneously. During training, one of these three was randomly selected and the steering angle adjusted if necessary (if it was left or right). The network was trained using 10 epochs of 20,000 samples each with batch size 40.

The trained net was very successful in steering the car around the map, which consisted of a looping track circling a lake. The car is able to drive continuously forever without diverging

a)



b)



c)

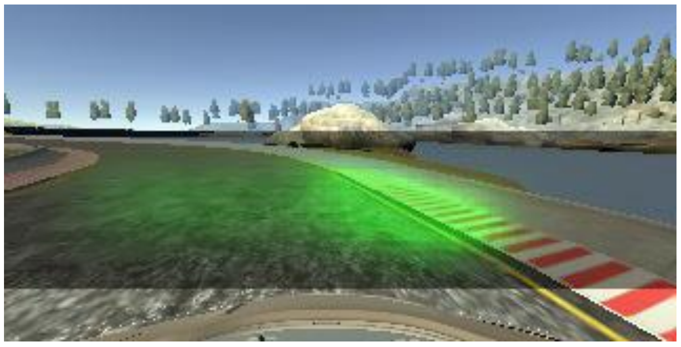


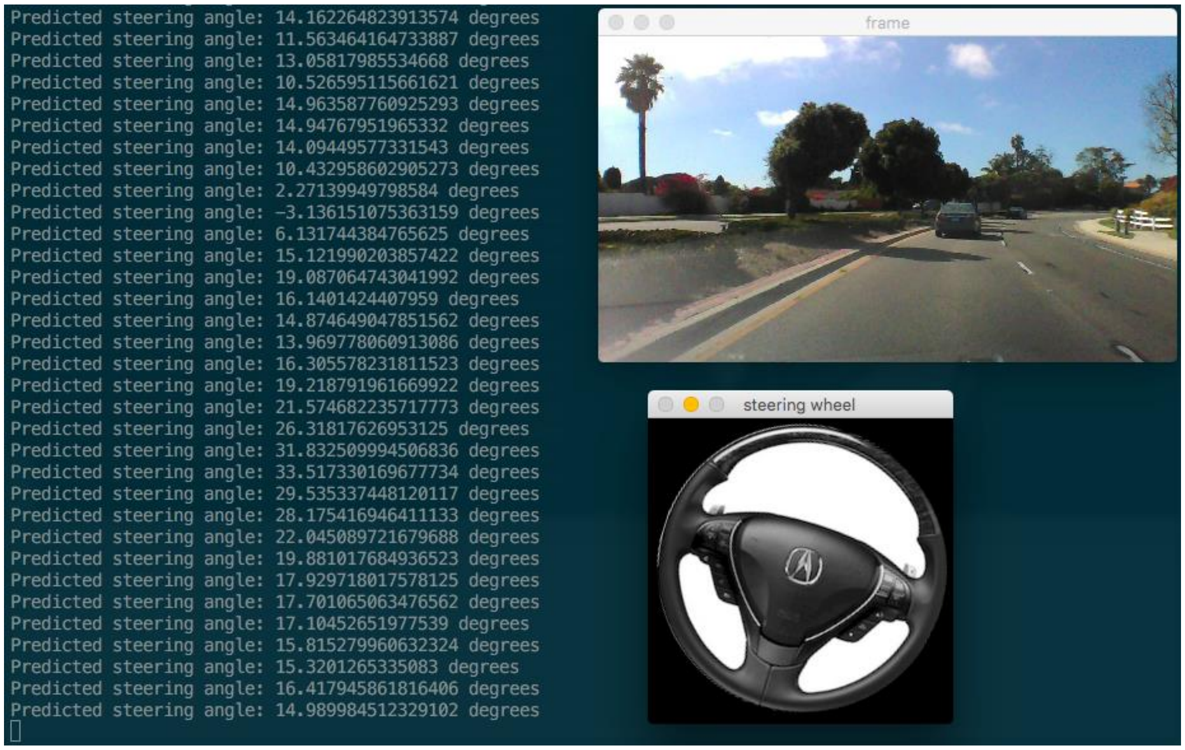
Figure 4: VisualBackProp on self-driving car image

a) Original Image

b) VisualBackProp Mask (Active Pixels)

c) Visualization Overlay of Mask onto Original from the road and falling into a hazard. The VisualBackProp implementation was similarly successful at producing a sensible representation of the active pixels in the inputs. Figure 4 depicts a) the original image, b) the mask generated by VisualBackProp, and c) the overlayed image. The dark shaded rectangle represents the image that was fed into the neural net controller—all the input images were cropped to remove the sky and the front of the car. In the figure, the car is encountering a bend that, if dealt with incorrectly, would cause it to drive into the lake. We see clearly that the outside curve of the road is the feature that is most potent in the network's decision making process. This makes a lot of sense. When looking at any image similar to this we immediately identify the bend or turn based on the fact that one side of the road extends sharply towards the middle.

The network was then trained on a set of real street data that was taken from an online source. Training was conducted in the same manner as in the computer simulator, except that

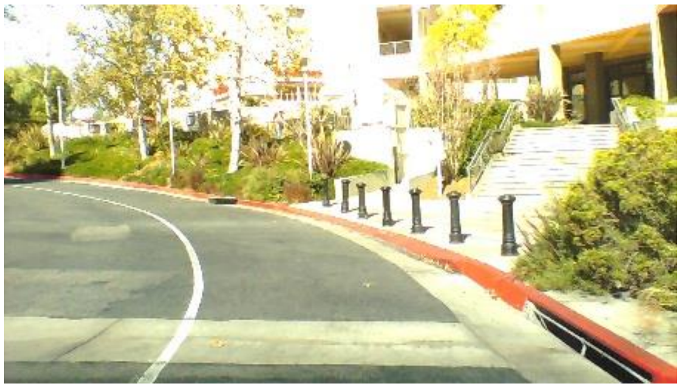


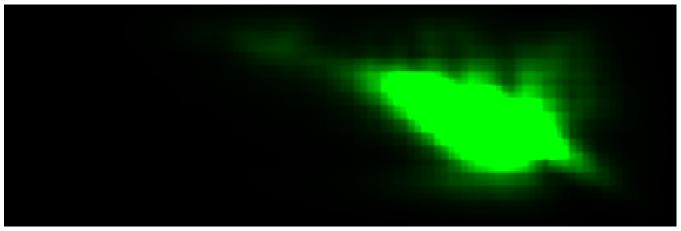
```
Predicted steering angle: 14.162264823913574 degrees
Predicted steering angle: 11.563464164733887 degrees
Predicted steering angle: 13.05817985534668 degrees
Predicted steering angle: 10.526595115661621 degrees
Predicted steering angle: 14.963587760925293 degrees
Predicted steering angle: 14.94767951965332 degrees
Predicted steering angle: 14.09449577331543 degrees
Predicted steering angle: 10.432958602905273 degrees
Predicted steering angle: 2.27139949798584 degrees
Predicted steering angle: -3.136151075363159 degrees
Predicted steering angle: 6.131744384765625 degrees
Predicted steering angle: 15.121990203857422 degrees
Predicted steering angle: 19.087064743041992 degrees
Predicted steering angle: 16.1401424407959 degrees
Predicted steering angle: 14.874649047851562 degrees
Predicted steering angle: 13.969778060913086 degrees
Predicted steering angle: 16.305578231811523 degrees
Predicted steering angle: 19.218791961669922 degrees
Predicted steering angle: 21.574682235717773 degrees
Predicted steering angle: 26.31817626953125 degrees
Predicted steering angle: 31.832509994506836 degrees
Predicted steering angle: 33.517330169677734 degrees
Predicted steering angle: 29.535337448120117 degrees
Predicted steering angle: 28.175416946411133 degrees
Predicted steering angle: 22.045089721679688 degrees
Predicted steering angle: 19.881017684936523 degrees
Predicted steering angle: 17.929718017578125 degrees
Predicted steering angle: 17.701065063476562 degrees
Predicted steering angle: 17.10452651977539 degrees
Predicted steering angle: 15.815279960632324 degrees
Predicted steering angle: 15.3201265335083 degrees
Predicted steering angle: 16.417945861816406 degrees
Predicted steering angle: 14.989984512329102 degrees
```

Figure 5: Street Data Testing Interface Visualization

there was only a single front-facing camera rather than three whose pictures could be randomly selected from. The camera images also differed in dimensional size and in not capturing the front of the car, and so the cropping process represented another minor change. The network was again trained using 10 epochs of 20,000 samples each and with batch size 40.

The repository provided a program that displayed a steering wheel angle alongside the consecutive series of images, which allows us to broadly visualize the controller's output, as exhibited in Figure 5. Using the model to predict steering on the testing dataset produced consistently reasonable results. In part due to a desire not to corrupt the consecutive stream of images (effectively a video), the data was not separated into training and testing sets, and the visualization / simulator only operated on the training set. However, there are numerous data sets online that could be adopted to test the controller. Running VisualBackProp on the street data produced good results, as shown in Figure 6.

a)


b)


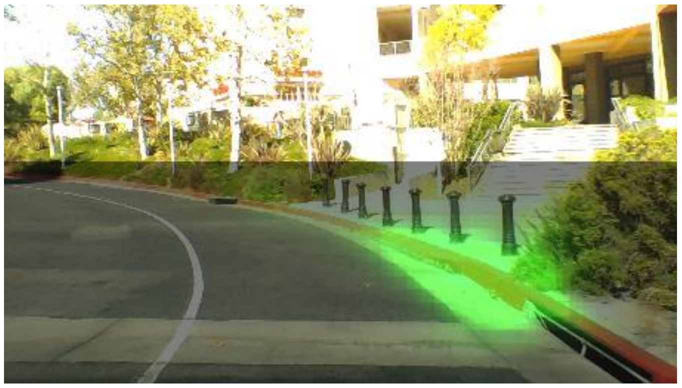
c)


Figure 6: VisualBackProp on self-driving car image

a) Original Image

b) VisualBackProp Mask (Active Pixels)

c) Visualization Overlay of Mask onto Original

Future Work

The most immediate next step is to apply the CNN and VisualBackProp to data collected from Y Driving's cars. The lab is planning to build a small model city similar to MIT's Duckietown [5], and ideally it will be completed next semester. As long as there is sufficient data, the cars should naturally be able to navigate their away around the model city, properly following lanes and avoiding other cars and objects.

The system could later be extended to control the car's velocity. Whether this would just occur as a single added output of the current architecture, or whether an entirely separate head should be added is subject to investigation. Adding a speed controller to Y Driving's cars will provide them with the capacity to obey traffic lights and stop signs, as well as stop to avoiding hitting an object in front rather than swerving around it.

The final basic capability needed for the cars is lane changing. The CNN could be posed as a solution for this potentially by simply adding a separate network of the same architecture, which is trained only over data collected during lane-changing. Once a car decides / is directed to change lanes, the lane-changing network could subsume the primary network until the lane change has been completed. It's unclear to me how effective such an architecture would be, so comparison against alternatives like reinforcement learning should be pursued.

Acknowledgements

I am deeply grateful to my advisor Man-Ki Yoon, who was endlessly accommodating throughout the semester and helped me pivot when I needed to. I would also like to thank Zhong Shao for supporting me as a faculty supervisor. I adapted a significant amount of code from Naoki Shibuya (naokishibuya)'s behavioral cloning example, Huynh Ngoc Anh (experiencor)'s

Keras adaptation of VisualBackProp, and Sully Chen (SullyChen)'s street data image engine.

Thank you to each of them for their generous open source contributions.

References

1. Bojarski, M., Del Testa, D., Dworakowski, D., Firner, B., Flepp, B., Goyal, P., ... & Zhang, X. (2016). End to end learning for self-driving cars. *arXiv preprint arXiv:1604.07316*.
2. Bojarski, M., Choromanska, A., Choromanski, K., Firner, B., Jackel, L., Muller, U., & Zieba, K. (2016). VisualBackProp: efficient visualization of CNNs. *arXiv preprint arXiv:1611.05418*.
3. Bojarski, M., Yeres, P., Choromanska, A., Choromanski, K., Firner, B., Jackel, L., & Muller, U. (2017). Explaining how a deep neural network trained with end-to-end learning steers a car. *arXiv preprint arXiv:1704.07911*.
4. Pomerleau, D. A. (1989). Alvinn: An autonomous land vehicle in a neural network. In *Advances in neural information processing systems* (pp. 305-313).
5. Paull, Liam, et al. "Duckietown: an open, inexpensive and flexible platform for autonomy education and research." Robotics and Automation (ICRA), 2017 IEEE International Conference on. IEEE, 2017.