Craig Vargas

Behavioral Cloning write up:

Files and Submitted Code Quality:

My project contains the following files:

1. model.py – python file to train my model
2. drive.py – python file to drive the car in autonomous mode
3. model.h5 – trained CNN
4. writeup_report.pdf – pdf file summarizing results

Functional Code

My model is able to complete track 1

Readable Code

My python code is clear and documented

Model Architecture and Training Strategy

Model Architecture

My model consists of a basic convolutional neural network (Lines 253-311).

Convolutional layers (Model.py: Lines 272-291)

The network has two 5x5 convolutional layers with 2x2 strides.

Dense layers (Model.py: Lines 296-309)
The output from the last convolutional layer is flattened into an array of length 400 and fed into four dense layers with dimensions: 100, 50, 10, and 1. The output of the last layer is a floating point number that acts as our steering angle

Activations
Rectified linear units (relu) were used after every layer, except the final, layer to introduce non linearity into the model.

Regularization
Dropout layers were used after each convolutional and dense layer, not including the final layer. There were five dropout layers. The dropout probability used after each convolutional layer was 0.5. The dropout probabilities used in the dense layers were 0.4, 0.3, and 0.2.

Diagram:

1. Original Image: 160x320x3
2. Preprocessing: 160x320x3 -> 32x128x1
    a. Shrink (40%)
    b. Grayscale
    c. Cropping (50%)
    d. Normalization
3. Layer 1 - Conv(5x5x4, 2x2 strides) + Relu + Dropout(0.5): 32x128x1 -> 14x62x4
4. Layer 2 - Conv(5x5x4, 2x2 strides) + Relu + Dropout(0.5): 14x62x4-> 5x29x8
5. Flatten: 5x29x8 -> 1,160
6. Layer 5 – Dense(100) + Relu + Dropout(0.4): 400 -> 100
7. Layer 6 – Dense(50) + Relu + Dropout(0.3): 100 -> 50
8. Layer 7 – Dense(10) + Relu + Dropout(0.2): 50 -> 10
9. Layer 8 – Dense(1): 10 -> 1

Overfitting

Dropout was used to reduce overfitting. Dropout was extremely important in my model. In the beginning of the experimentation phase I had no dropout slowly added more and more dropout if I saw the model was performing poorly on the validation data. At

some point I increased the dropout probabilities markedly and trained the model over many fewer epochs than the prior trial in an effort to test my changes. Surprisingly to me my car drove much smoother with the increased dropout even though I ran over fewer epochs and the training loss was much higher. At first I only considered dropout in order to perform better on the validation data but I realize now that even if the training and validation losses are comparable, dropout is still very important for generalization of the model. For the most part, as I increased my dropout probabilities, the driving performance increased as well.

Parameter Tuning

The Adam optimizer was used since it has an algorithm to adjust the learning rate over time. I tested other hyper parameters by running experimental training over 2-10 epochs and determining if changes in other variables had negative or positive impacts on driving despite the fact that none of those models could drive the entire course. The parameters for each layer started off with inspiration from the Nvidia model but I chose to scale down the model in hopes to not over-fit and to speed up the learning process. I started with one convolutional layer and two dense layers and after a lot of experimentation I slowly added layers and tweaked parameters. As I added layers I realized I was able to drive around the track with three less convolutional layers than Nvidia used.

Training Data

I used the Udacity training data because I wanted to get a working data pipeline and model architecture before I bothered recording my own data.

Zero angle steering data

The Udacity car drove straight a lot and I was having issues going around some turns where the car drove straight instead. I decided to omit 75% of the zero steering angle data to give the data set a better balance so that the model would not be rewarded so much for guessing zero.

Creating data

The Udacity training set comes with three camera images for every recorded steering response. I decided to multiply the overall dataset by 3 by using the left and right camera images and augmenting their associated steering angles. I thought this would both provide more data as well teach the car to straighten itself when it veered off a bit to the left or the right. The correction factor used to augment the data came from trial and error. I started with the average steering magnitude and tried different multiples of this

number in several different training sessions. By time I got to 3.0 * AVG_MAGNITUDE the car was visibly swerving around the road and it seemed logical that my correction angles were causing this. I settled on 2.0 * AVG_MAGNITUDE as my correction.

I also created data later on in the pipeline by flipping all images and inverting the associated steering angles. I thought this would be helpful to both add more data to the data set and equalize the proportion of left and right turns; thus possibly help the model generalize more.

Image Cropping

The last thing I did to the data was to crop the top and bottom of the image. Cropping the top allowed me to ensure that the network decided how to steer based on objects that were closer to the car rather than the noise further away. Also, the network focused on the portion of the image that contained the road. Cropping the bottom of the image served to not bother with processing the portion of the image that contained the hood of the car which was likely to not add much information.

Solution Design Approach

My approach to the overall solution was to start off with a simple model and build up to the Nvidia architecture while carefully selecting the hyper-parameters through a long process of trial and error. I drew inspiration from the Nvidia model because they solved a similar problem and proved that the architecture worked. My problem seemed like a smaller problem than theirs so I had intensions to scaled down their architecture in the hopes to speed up the training process. This involved shrinking the images, having smaller depths in the convolutional layers, and possibly omitting a convolutional layer. After building from the ground up I realized I only needed two convolutional layers to be able to circle the track.

Images

As stated above, the first thing I wanted to do was to scale the images down so that less data needed to be processed overall. I scaled it down to 40% of the original image, anything smaller than that limited how deep I could design my CNN architecture. I also employed cropping to focus on areas of interest and used a grayscale image again to reduce the feature set.

Image processing examples:

Full image

After Shrinking

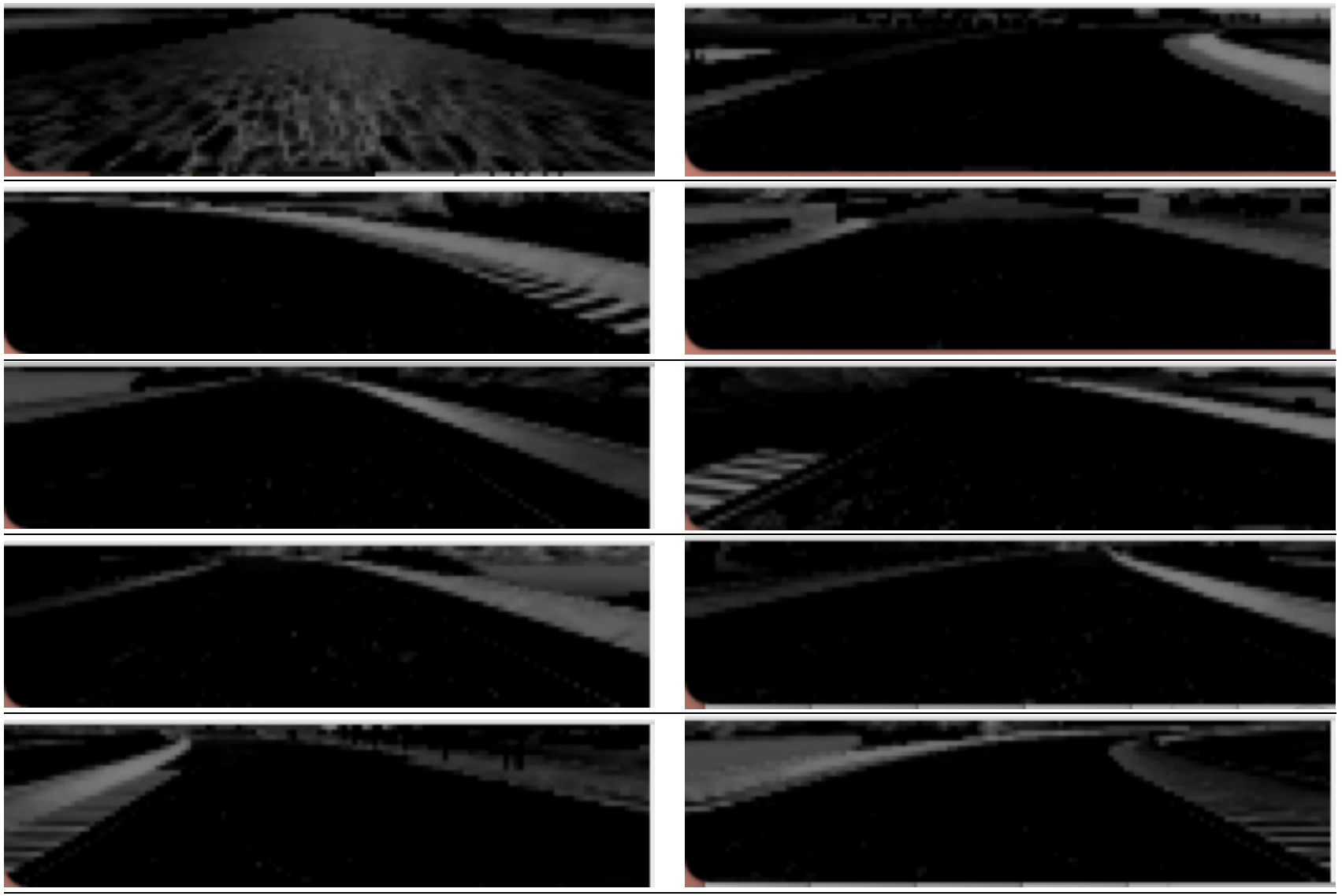
Image processing examples (continued):

After Applying Grayscale

After Cropping

After Normalization

Some examples of the images used to train on:



Model Depth

I omitted three convolutional layers from the Nvidia model because my model was able to drive around the track smoothly without the extra layers and since I was shrinking and cropping I was worried that adding more layers would lead me to run into overfitting problems. I think if I were to collect data form a few other tracks I would need to add more layers to be able to learn a much wider array of images.

Convolution Filter Depth

I started my model with a convolutional filter depth of 2 and tried doubling the depth in subsequent layers. I also doubled the depth of all layers during the experimental phase if I felt the model needed to learn more. My final convolution depths were 4 and 8. I ended up with a much shallower convolutional layers for my problem mostly because I only needed to model for one track and my data set was likely a fraction of theirs.

Regularization

During every training run I set aside data for validation and monitored how the model performed on the validation set. I did this mainly to understand when I needed to increase regularization and understand if my model was overfitting the data. During the trial and error process of building my network, I realized that overfitting was not just a simple comparison of training loss to the validation loss. Instead I saw that overfitting caused my car to drive erratically even if the training and validation losses looked legitimate.

Loss Function

I chose to use mean squared error because it seemed like a standard choice for regression. However, I also monitored the mean absolute error since the nature of our data set produced very small numbers for mean squared error even if the model had barely learned anything. With steering angles taking on values [-1,1] the magnitude of the error terms was mostly always less than one making the MSE a metric too small to make any human sense out of it.

Validation

Once validation results were good I ran the model in the simulator and watched it drive around the track. A few of the models drove off on to the dirt road and I thought I was going to have to record more data around that part of the track to fix this problem. However, I later realized how much regularization stabilized the driving of my car and once I beefed up the dropout layers the car magically stayed off of the dirt roads due to its ability to generalize better.

Training

During the experimental trials I trained the model anywhere from 2-10 epochs just to gage how the tuning of a particular parameter effected the car's driving. I didn't expect the car to make it around the track until I found a good architecture and trained it for 100-200 epochs. During this experimental phase I ended up finding a good model that completed the track with only 10 epochs of training.