Behavioral Cloning

The goals / steps of this project are the following:

- Use the simulator to collect data of good driving behavior
- Build, a convolution neural network in Keras that predicts steering angles from images
- Train and validate the model with a training and validation set
- Test that the model successfully drives around track one without leaving the road
- Summarize the results with a written report

Files Submitted:

- drive.py *(unedited – set speed changed to 8, 9, 10, 15, 20 and back to 9 for consecutive tries)*
- video.py *(unedited)*
- Write up
- Model.py (file that lists the keras model created for the lab)
- Behaviourcloning_model.ipynb (Jupyter notebook for final model)
- run1 (video of initial run of model)
- run3 (Video of final model run on track 1)
- run3_track2 (Video of final model run on track 2)
- test.py (used to translate data to pickle file before uploading to AWS server)

Function Code File

- python drive.py model3.h5

Submission code

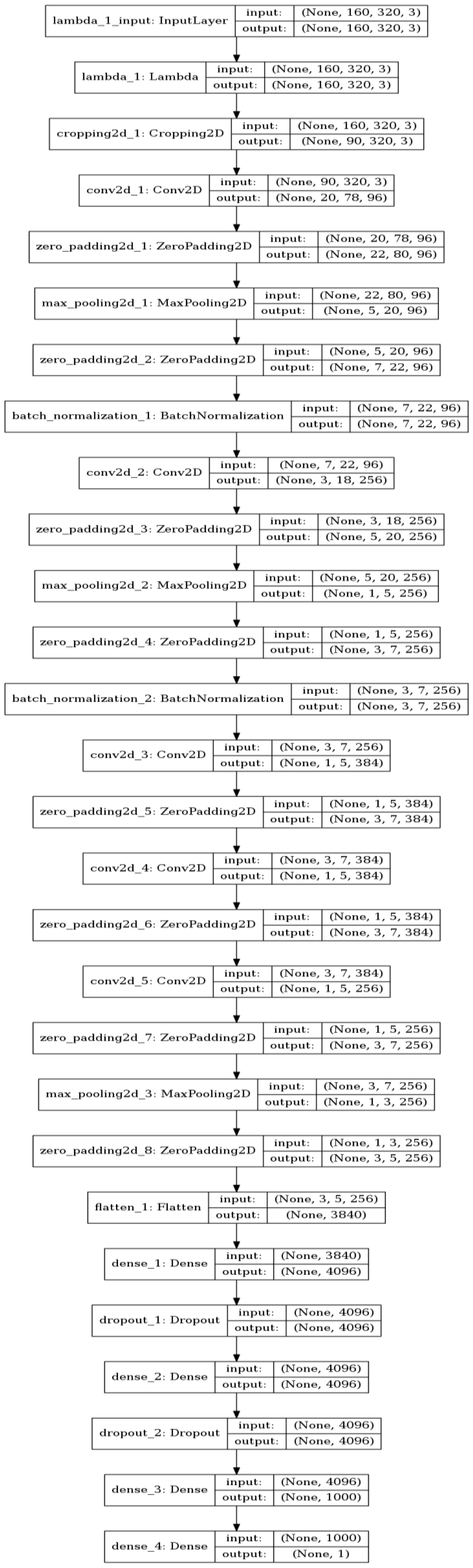
- The model.py file contains the code for training and saving the convolution neural network. The file shows the pipeline I used for training and validating the model, and it contains comments to explain how the code works.

Model Architecture and Training Strategy

- Initially I started out with the basic three layer model provided by the Udacity Class.
- Then I moved on to attempting to implement the AlexNet Caffenet version.
- Which was 9 layers that involved 5 Convolution layers followed by 3 FC layers.
- I proceeded to further modify it involving Batch normalization layers, Zero Padding Layers and a few dropout layers between the FC layers to prevent Over-fitting.
- To also prevent overfitting, I included Callbacks (Early Stopping) into model compilation to monitor the Loss variable for a min of 5 epochs and discern a loss of 0.1 percent or less as no progress in the gradient descent.

Also, the fully connected layers involved Relu's to attempt to speed up training.

Please refer to the following page on the network finally implemented.

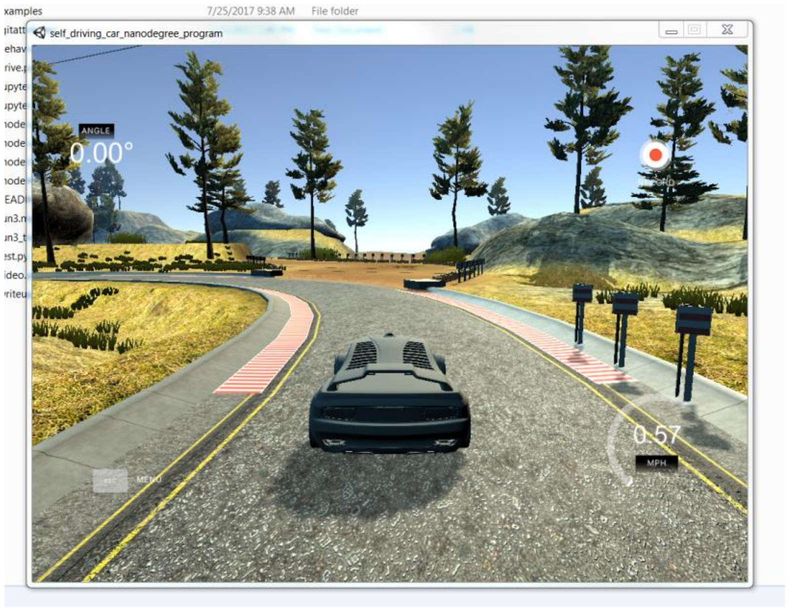

lambda_1_input: InputLayer | input: (None, 160, 320, 3) | output: (None, 160, 320, 3)
lambda_1: Lambda | input: (None, 160, 320, 3) | output: (None, 160, 320, 3)
cropping2d_1: Cropping2D | input: (None, 160, 320, 3) | output: (None, 90, 320, 3)
conv2d_1: Conv2D | input: (None, 90, 320, 3) | output: (None, 20, 78, 96)
zero_padding2d_1: ZeroPadding2D | input: (None, 20, 78, 96) | output: (None, 22, 80, 96)
max_pooling2d_1: MaxPooling2D | input: (None, 22, 80, 96) | output: (None, 5, 20, 96)
zero_padding2d_2: ZeroPadding2D | input: (None, 5, 20, 96) | output: (None, 7, 22, 96)
batch_normalization_1: BatchNormalization | input: (None, 7, 22, 96) | output: (None, 7, 22, 96)
conv2d_2: Conv2D | input: (None, 7, 22, 96) | output: (None, 3, 18, 256)
zero_padding2d_3: ZeroPadding2D | input: (None, 3, 18, 256) | output: (None, 5, 20, 256)
max_pooling2d_2: MaxPooling2D | input: (None, 5, 20, 256) | output: (None, 1, 5, 256)
zero_padding2d_4: ZeroPadding2D | input: (None, 1, 5, 256) | output: (None, 3, 7, 256)
batch_normalization_2: BatchNormalization | input: (None, 3, 7, 256) | output: (None, 3, 7, 256)
conv2d_3: Conv2D | input: (None, 3, 7, 256) | output: (None, 1, 5, 384)
zero_padding2d_5: ZeroPadding2D | input: (None, 1, 5, 384) | output: (None, 3, 7, 384)
conv2d_4: Conv2D | input: (None, 3, 7, 384) | output: (None, 1, 5, 384)
zero_padding2d_6: ZeroPadding2D | input: (None, 1, 5, 384) | output: (None, 3, 7, 384)
conv2d_5: Conv2D | input: (None, 3, 7, 384) | output: (None, 1, 5, 256)
zero_padding2d_7: ZeroPadding2D | input: (None, 1, 5, 256) | output: (None, 3, 7, 256)
max_pooling2d_3: MaxPooling2D | input: (None, 3, 7, 256) | output: (None, 1, 3, 256)
zero_padding2d_8: ZeroPadding2D | input: (None, 1, 3, 256) | output: (None, 3, 5, 256)
flatten_1: Flatten | input: (None, 3, 5, 256) | output: (None, 3840)
dense_1: Dense | input: (None, 3840) | output: (None, 4096)
dropout_1: Dropout | input: (None, 4096) | output: (None, 4096)
dense_2: Dense | input: (None, 4096) | output: (None, 4096)
dropout_2: Dropout | input: (None, 4096) | output: (None, 4096)
dense_3: Dense | input: (None, 4096) | output: (None, 1000)
dense_4: Dense | input: (None, 1000) | output: (None, 1)

<u>Model parameter tuning</u>

- The model used an 'adam' optimizer, so the learning rate was not tuned manually
- Also, I attempted to switch to adagrad optimizer but the model started increasing the loss value instead of decreasing.
- Switching it to rmsprop also got the loss to increase.
- So I finally switched back to Adam.

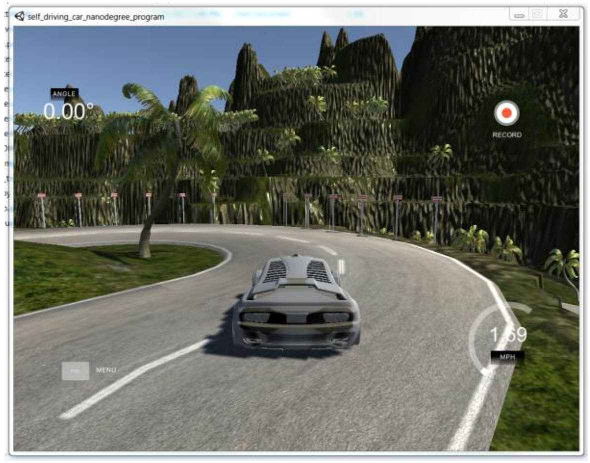
<u>Appropriate training data</u>

- Training data initially consisted of one run in the forward direction and one run in the reverse direction.
- After successive tests, I noticed a few points.
  On track on the car did not do well on the following turn.



So I went back and reran the training on this section for ways to return from edge of the track.

Also, on track 2 it would not recognize all turn that had the red bars as shown below.



So I retrained it only on one curve and hoped it would learn for the others, and it worked 85 percent of the time. The other percent it would not run off, but it got stuck slightly on the edge.

Reference video run3_track2.

In addition to the above steps, I also did the following to improve th emodel, **But they did not work.**

1) I tried implementing a data generator (code commented out) but as the data set increased it kept throwing out memory errors.
2) Second in order to increase training features I attempted to add an extra channel to the image input containing the Canny transform. But this worked for trianing data but the simulator returned only 3 channel data and I couldn't get it to convert the data to 4 channels within the model layers.
3) Also, I attempted to modeify different filter sizes for each layer but ended up sticking to the values shown by alexnet (https://github.com/dmlc/minerva/wiki/Walkthrough:-AlexNet)

The final model is as shown in the image shown before.

NOTE: One thing I forgot to implement was shuffling of data.