Car-Behavioral-Cloning

Project Description

In this project, I used a neural network to clone car driving behavior. It is a supervised regression problem between the car steering angles and the road images in front of a car.

Those images were taken from three different camera angles (from the center, the left and the right of the car).

The network is based on The NVIDIA model, which has been proven to work in this problem domain.

As image processing is involved, the model is using convolutional layers for automated feature engineering.

Files included

- model.py The script used to create and train the model.
- drive.py The script to drive the car
- model.h5 The model weights.
- run1.mp4:

I have trained the model on GPU machine at my university's cluster (NYU HPC).

Model Architecture Design

The design of the network is based on the NVIDIA model, which has been used by NVIDIA for the end-to-end self driving test. As such, it is well suited for the project.

It is a deep convolution network which works well with supervised image classification / regression problems. As the NVIDIA model is well documented, I was able to focus how to adjust the training images to produce the best result with some adjustments to the model to avoid overfitting and adding non-linearity to improve the prediction.

I've added the following adjustments to the model.

- I used Lambda layer to normalized input images to avoid saturation and make gradients work better.
- I've added an additional dropout layer to avoid overfitting after the convolution layers.
- I've also included RELU for activation function for every layer except for the output layer to introduce non-linearity.

In the end, the model looks like as follows:

- Image normalization
- Convolution: 5x5, filter: 24, strides: 2x2, activation: RELU
- Convolution: 5x5, filter: 36, strides: 2x2, activation: RELU
- Convolution: 5x5, filter: 48, strides: 2x2, activation: RELU
- Convolution: 3x3, filter: 64, strides: 1x1, activation: RELU
- Convolution: 3x3, filter: 64, strides: 1x1, activation: RELU
- Fully connected: neurons: 100, activation: RELU
- Dropout .50
- Fully connected: neurons: 50, activation: RELU
- Fully connected: neurons: 10, activation: RELU
- Fully connected: neurons: 1 (output)

As per the NVIDIA model, the convolution layers are meant to handle feature engineering and the fully connected layer for predicting the steering angle. However, as stated in the NVIDIA document, it is not clear where to draw such a clear distinction. Overall, the model is very functional to clone the given steering behavior.

The below is an model structure output from the Keras which gives more details on the shapes and the number of parameters.

Layer (type)	Output Shape	Params
lambda_1 (Lambda)	(None, 66, 200, 3)	0
convolution2d_1 (Convolution2D)	(None, 31, 98, 24)	1824

convolution2d_2 (Convolution2D)	(None, 14, 47, 36)	21636
convolution2d_3 (Convolution2D)	(None, 5, 22, 48)	43248
convolution2d_4 (Convolution2D)	(None, 3, 20, 64)	27712
convolution2d_5 (Convolution2D)	(None, 1, 18, 64)	36928
flatten_1 (Flatten)	(None, 1152)	0
dense_1 (Dense)	(None, 100)	115300
Dropout_1 (Dropout)	(None,100)	0
dense_2 (Dense)	(None, 50)	5050
dense_3 (Dense)	(None, 10)	510
dense_4 (Dense)	(None, 1)	11

Data Preprocessing

Image Sizing

- the images are cropped so that the model won't be trained with the sky and the car front parts
- the images are resized to 66x200 (3 YUV channels) as per NVIDIA model
- the images are normalized (image data divided by 255 and subtracted 0.5). As stated in the Model Architecture section, this is to avoid saturation and make gradients work better)

Image Augmentation

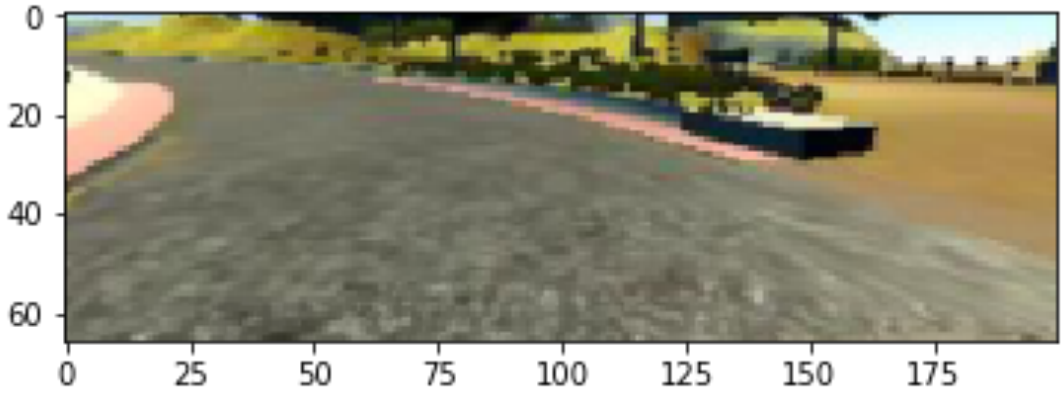
For training, I used the following augmentation technique along with Python generator to generate unlimited number of images:

- Randomly choose right, left or center images.
- For left image, steering angle is adjusted by +0.15
- For right image, steering angle is adjusted by -0.15
- Randomly flip image

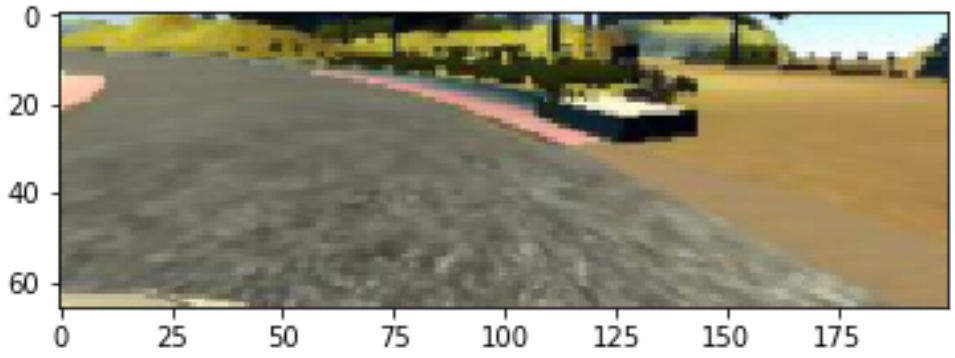
Examples of Augmented Images

The following is the example transformations:

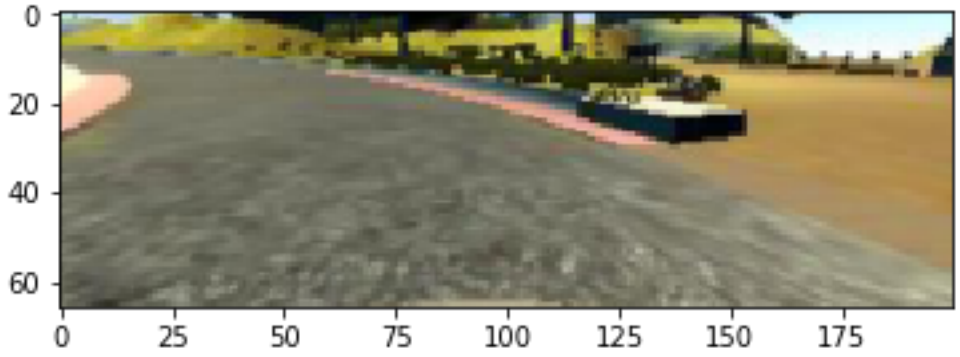
Left Image:



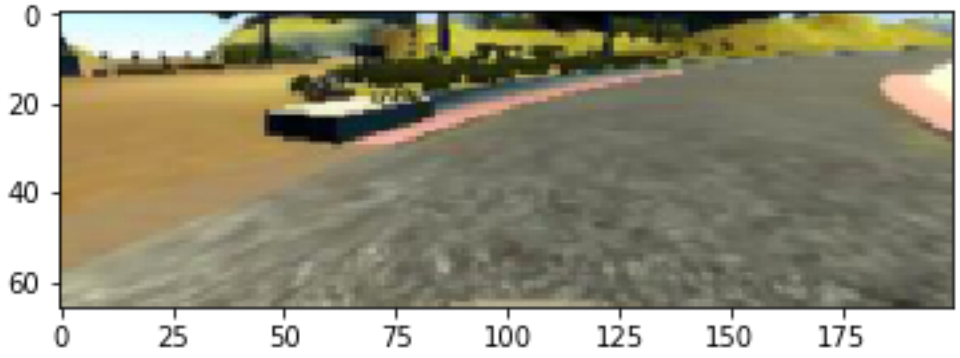
Right Image:



Center Image:



Flip Image:



Training, Validation and Test

I split the images into train and validation (90%,10% respectively) set in order to measure the performance at every epoch. Testing was done using the simulator.

As for training,

- I used mean squared error for the loss function to measure how close the model predicts to the given steering angle for each image.
- I used Adam optimizer for optimization with default learning rate.
- I used Batch Size of 128.

I ran the model for 10 epochs. This is the result of model performance. My final validation loss was 0.103.

```
Epoch 1/10
56/56 [==============================] - 76s - loss: 0.0193 - val_loss: 0.0127
Epoch 2/10
56/56 [==============================] - 67s - loss: 0.0149 - val_loss: 0.0117
Epoch 3/10
56/56 [==============================] - 66s - loss: 0.0140 - val_loss: 0.0106
Epoch 4/10
56/56 [==============================] - 66s - loss: 0.0126 - val_loss: 0.0106
Epoch 5/10
56/56 [==============================] - 66s - loss: 0.0124 - val_loss: 0.0119
Epoch 6/10
56/56 [==============================] - 66s - loss: 0.0120 - val_loss: 0.0106
Epoch 7/10
56/56 [==============================] - 65s - loss: 0.0120 - val_loss: 0.0094
Epoch 8/10
56/56 [==============================] - 66s - loss: 0.0113 - val_loss: 0.0101
Epoch 9/10
56/56 [==============================] - 66s - loss: 0.0111 - val_loss: 0.0101
Epoch 10/10
56/56 [==============================] - 67s - loss: 0.0107 - val_loss: 0.010
```

References

- NVIDIA model: https://devblogs.nvidia.com/parallelforall/deep-learning-self-driving-cars/
- RGB2YUV: http://www.pythonexample.com/code/rgb-to-yuv-conversion-formula/