Behavioral Cloning

Writeup

Behavioral Cloning Project

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

Rubric Points

Here I will consider the rubric points individually and describe how I addressed each point in my implementation.

Files Submitted & Code Quality

1. Submission includes all required files and can be used to run the simulator in autonomous mode

My project includes the following files:
* model.py containing the script to create and train the model
* drive.py for driving the car in autonomous mode
* model.h5 containing a trained convolution neural network
* writeup_report.md and writeup_report.pdf summarizing the results

2. Submission includes functional code

Using the Udacity provided simulator and my drive.py file, the car can be driven autonomously around the track by executing `python drive.py model.h5`

3. Submission code is usable and readable

The model.py file contains the code for training and saving the convolution neural network, as well as code for generating training and validation data. The file shows the pipeline I used for training and validating the

model, and it contains comments explaining how the code works.

Model Architecture and Training Strategy

1. An appropriate model architecture has been employed

My model consists of 3 blocks of [batch normalization, dropout, and 2 succesive 3x3 convolutions with elu nonlinearities] followed by 3 dense layers that also include batch normalization and dropout. Convolutions use [24, 36, 36, 48, 48, 64] filters, following NVIDIA's end to end learning paper.

2. Attempts to reduce overfitting in the model

Prediction model contains a generous helping of batch normalization layers followed by dropout layers, lines 49-83.

Model was trained and validated on different data sets to ensure that it wasn't overfitting (code line 367). Finally model was tested by running it through the simulator and ensuring that the vehicle could stay on the track.

3. Model parameter tuning

My model used an adam optimizer. Since batch normalization was used, initial learning rate is quite large, 0.1. Training procedure reduces it when validation loss stops improving.

4. Appropriate training data

I used a combination of center lane driving, recovering from the left and right sides of the road. I use left side frames only when turning angle is at least 0.1 (i.e. car is turning to right at least a bit) and right frames only when turning angle is -0.1 or less (i.e. car is turning to left at least a bit). For left and right frames angles are bumped up slightly.

I used plenty of data augmentation. Images are randomly translated, rotated, brightened and flipped (functions **get_augmented_image()** and **get_single_dataset_generator()**).

In **get_balanced_paths_angles_tuples()** I make sure (frame, steering angles) tuples returned by generators are approximately uniformly distributed across whole steering angles spectrum.

Finally I created different datasets concentrating on different parts of the task - smooth driving, curves, recovery - and on training generator uses all these datasets (rotating through separate generators for each dataset).

Model Architecture and Training Strategy

1. Solution Design Approach

Preprocessing: Input frames are cropped to remove most of pixels above road surface, then resized to half their original size. Input is also normalized to (0, 1) range.

My first prediction model used 3x3 convolutions with filter sizes in 64~256 range and occasional maxpoolings until output was small (about 20x5), then a single dense layer with linear activation and mse error. After some trials with different variations of above model I found that decreasing number of convolutional filters didn't hurt performance at all, while increasing training speed. This brought me to a model very similar to that from NVIDIA's paper.

This model was able to navigate some parts of both tracks, but had a large overfit to training data. Adding plenty of dropout and batch normalization helped to bring training and validation loss closer to each other and also allowed me to use larger initial learning rate.

Note on validation data - it was obtained from totally different runs that training data. This is important since just splitting single run data randomly would mean we might end up with nearly identical frames in training and validation sets, thus the validation loss would be low, but without necessary indicating true generalization power of the model - low loss would be purely due to validation data including frames nearly identical to training data frames.

A very important part in getting a working model was to make sure training data is evenly distributed between all steering angles.

Using my final architecture and enough good training samples, my model was able to drive the vehicle around track 1. A demonstration can be found in the video attached with this submission, as well as in the following Youtube video.

2. Final Model Architecture

The final model architecture (**get_preprocessing_pipeline()** lines 16-29 for preprocessing part and **get_prediction_pipeline()** lines 49-83 for prediction part) consisted of a convolution neural network with the following layers and layer sizes:

- Preprocessing:
    - cropping 50 pixels from top and 20 from bottom of the image
    - scaling - done with average pooling
    - normalization - scaling images to (0, 1) range
- Prediction:
    - 3 blocks of batch normalization, dropout, convolution 3x3, convolution 3x3. Convolutions have [24, 36, 36, 48, 48, 64] filters, going from start to end
    - 4 dense layers - sizes: 1000, 100, 50, 1

All dropout layers use keep probability of 0.5.

3. Creation of the Training Set & Training Process

My total amount of data used for training is about ~18k frames for track 1 and ~7k frames from track 2. This is out of a total of about 140k captured frames. As explained earlier, frames are chose so that angles distribution is roughly uniform and dropped frames are mostly 0 and -1/+1 (or extreme values) inputs frames. Adding more training data would of course help generalization, but it would also push training times longer than I am happy to wait for.

I recorded three types of actions:
* smooth driving
* concentrating on turns
* concentrating on recoveries - both from entering road boudaries and from escaping them if entered

Here are some examples of captured frames with no data augmentation applied:

```
Steering angle: [-0.01886792, -0.2735849, -0.49622644, 0.63207544, -0.98301892, -0.5566038]
```

Here are some examples of captured frames with data augmentation applied:

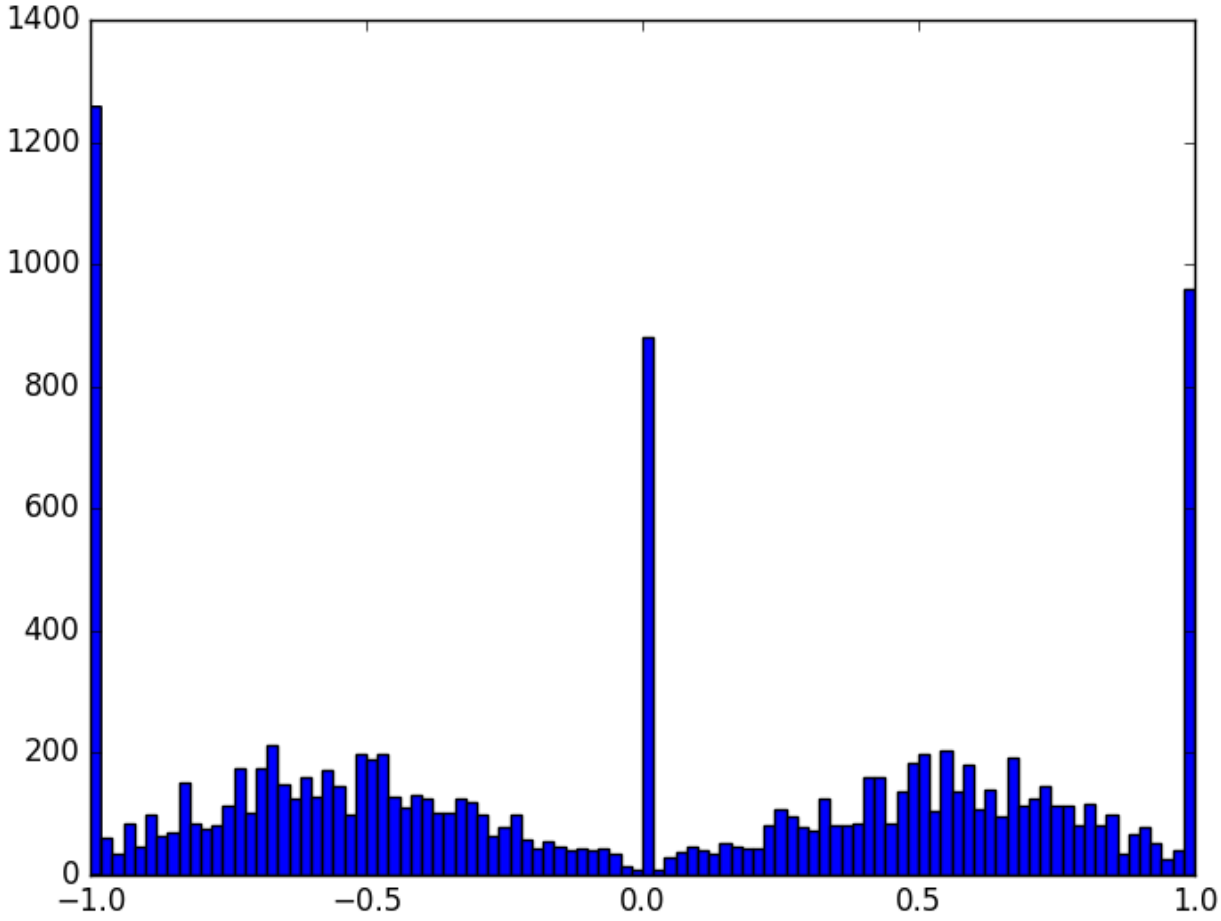
```
Steering angle: [-0.46226416000000004, -0.90377356, -0.34905664000000003, -1.0, 1.0, -0.4716981]
```

Augmentations are rather small, thus not necessary clearly visible, perhaps apart for brightness changes.

Here are some examples of preprocessed data that gets fed to prediction model

```
Steering angle: [0.40566040000000003, 0.7, -0.60943396, -0.8207547, 1.0, -0.47358496000000005]
```

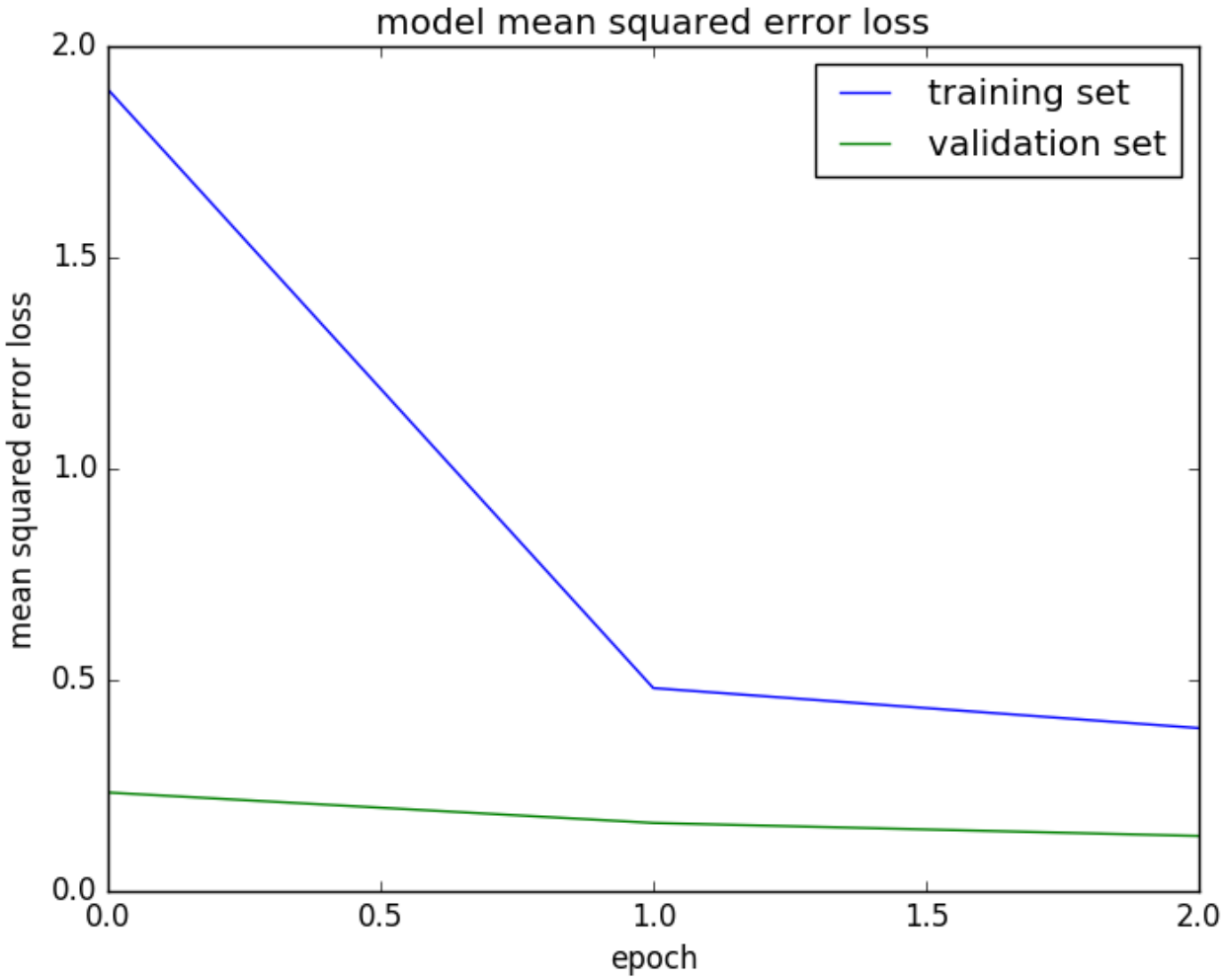
Finally here is a plot of distribution of steering angles in training data:



.

It has clear spikes on 0 and extreme steerings, but other than that has a reasonably uniform distribution.

Below is a plot of training and validation loss across epochs:


model mean squared error loss
training set
validation set
mean squared error loss
epoch
2.0
1.5
1.0
0.5
0.0
0.0
0.5
1.0
1.5
2.0

.