Behavioral Cloning

Model architecture

I used the Nvidia paper as a starting point, using larger filters and strides in the upper layers, eg, 8x8 with a stride of 4 and 5x5 with a stride of 2, in the first two layers, before using 3x3 filter with a stride of 1, as I have typically seen more often in practice. However, the larger earlier two layers seem to have helped in this project, which is after all, about finding features that represent curves on a road, which are likely to be significantly larger than 3x3.

This is a summary of my submitted model. I used one max pooling layer between the last two convolutions, which brought the parameters down from 27 million or so to just over 7 million, resulting in much faster training, and no apparent loss of the ability to drive round the track.

Layer (type)	Output Shape	Param #
cropping2d_1 (Cropping2D)	(None, 90, 320, 3)	0
lambda_1 (Lambda)	(None, 90, 320, 3)	0
conv2d_1 (Conv2D)	(None, 23, 80, 24)	4632
conv2d_2 (Conv2D)	(None, 12, 40, 36)	21636
conv2d_3 (Conv2D)	(None, 12, 40, 48)	15600
conv2d_4 (Conv2D)	(None, 10, 38, 64)	27712
max_pooling2d_1 (MaxPooling2	(None, 5, 19, 64)	0
conv2d_5 (Conv2D)	(None, 3, 17, 128)	73856
flatten_1 (Flatten)	(None, 6528)	0
dense_1 (Dense)	(None, 1000)	6529000
dropout_1 (Dropout)	(None, 1000)	0
dense_2 (Dense)	(None, 500)	500500
dropout_2 (Dropout)	(None, 500)	0
dropout_3 (Dropout)	(None, 500)	0
dense_3 (Dense)	(None, 10)	5010
dropout_4 (Dropout)	(None, 10)	0
dense_4 (Dense)	(None, 1)	11

Total params: 7,177,957
Trainable params: 7,177,957
Non-trainable params: 0

I also experimented with 3x3 kernels, with stride 1 and max pooling between each

layers, with fairly similar, but slightly worse results. I used 4 fully connected layers with dropout of 0.2 between the first two and 0.1 before the output layer in attempt to avoid overfitting, and more or less line with what Nvidia had done. I found that changing the number of nodes, or dropping a layer, had little effect in practice.

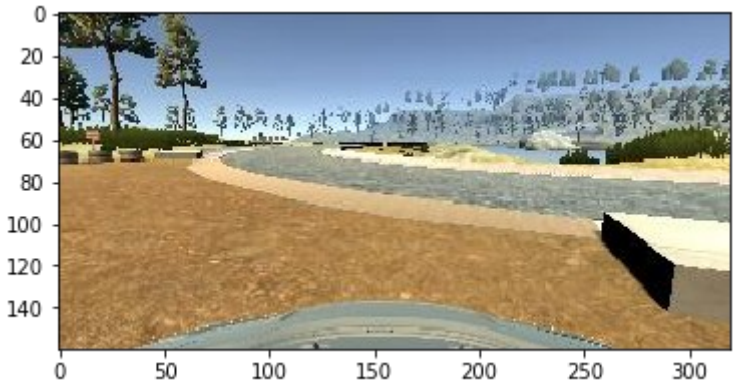
.

Data collection

I drove three times round the track, not paying too much attention to smooth driving, as I am rubbish at video games these days. I then turned round I did a couple in the other direction. I looked at the average value of the steering measurement and was satisfied when it got reasonably close to zero that I had removed the left-hand bias.

Despite getting losses at or below 0.01, I was finding that I was always crashing on the first right hand turn. After looking at a few images for steering angle adjustments, I realized that it was because I was training in GBR space, rather than RGB – note to self, read the docs more closely.

Following the one line fix to convert back to RGB, I was pleased do discover that I could drive fairly smoothly around the track. The only adjustment I made was to remove all absolute steering measurement of 1, as I discovered these occurred when I was turning round to go in the other direction. Here is an example of -1 steering.



After removing the values of 1, I was left with 8476 images. I flipped each of the center camera images, negating the measurement and then did the same for the left and right cameras, so, ultimately ended up with around 50,000 images. I made some attempts at selecting more images where the car was steering back towards the center, but ultimately, didn’t find any logic that made much difference, and by this time I was having no difficulty driving fairly cleanly round the track.

Training

I clipped the image using:

```
    model.add(Cropping2D(cropping=((60, 10), (0, 0)), input_shape=(160, 320, 3)))
```

to avoid including trees and other distractions from the top of the image and then used a lambda function to normalize the images.

I run the models with various iterations between 0 and 0.2 for the offset for the left and right camera. The value that gave the best results was around 0.04, though I realize that there should probably be a different offset value for left and right, I decided this was a form of image distortion similar to random

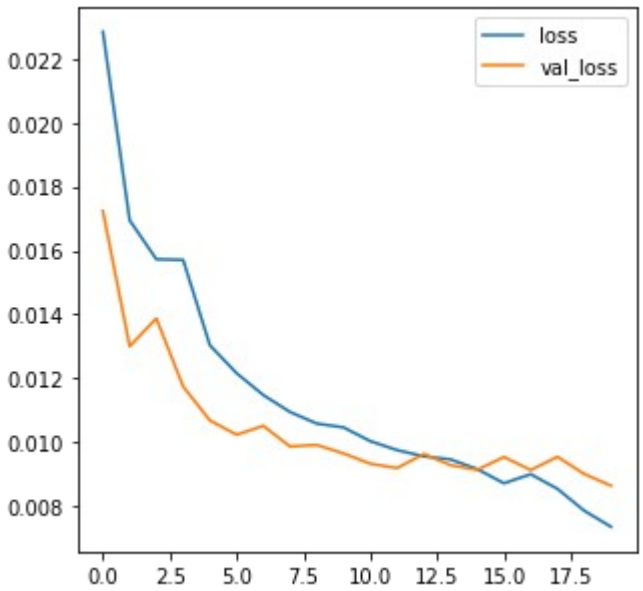
offset or rotation, and decided not to investigate further.

I used an adam optimizer, as I found there was so many varieties of possible networks, all leading to mse for training and validation of 0.01 or less, which was always enough to drive around the track quite easily, that I didn't want to have to play with any more hyperparameters.

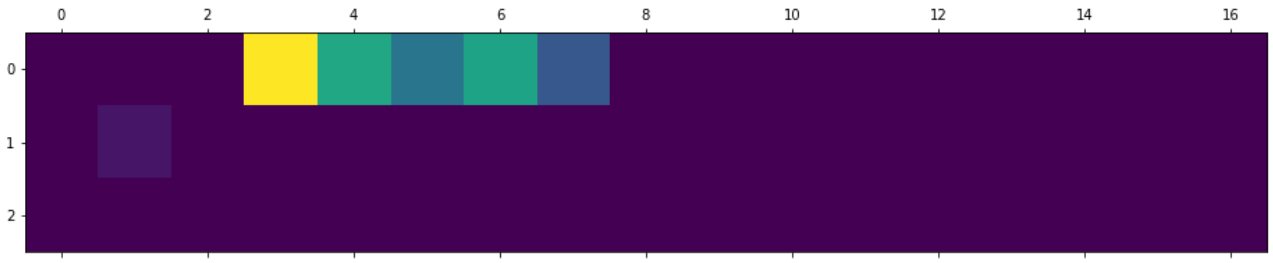
I used a batch size of 64, as this seemed to lead to fairly stable training. The chart below shows the training and validation losses during the submitted run, which show both a fairly stable learning rate and not much evidence of overfitting. I used an early stopping callback, with a patience of 3 and an epoch size of 20.

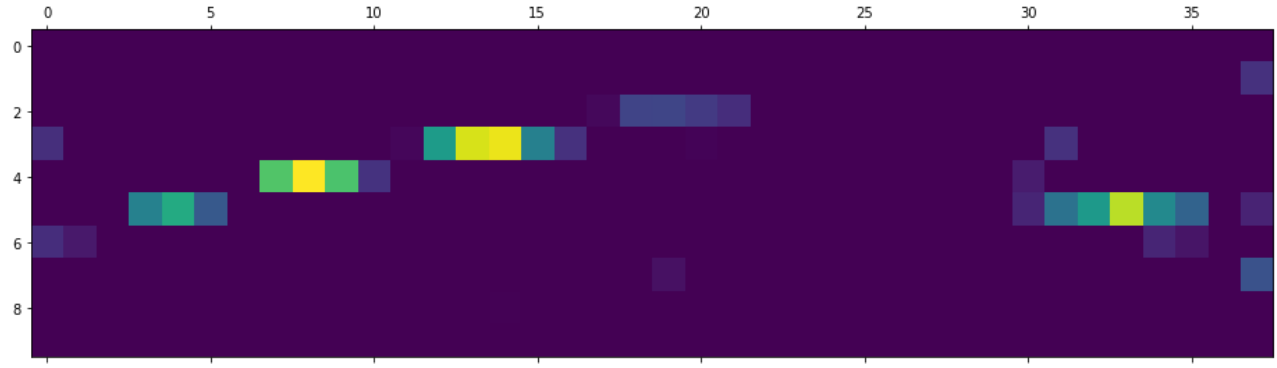
I used an adam optimizer, as I found there was so many varieties of possible networks, all leading to mse for training and validation of 0.01 or less, which was always enough to drive around the track quite easily. I used a 20% split for validation and used shuffle inside the generator, to ensure different samples were presented each time.

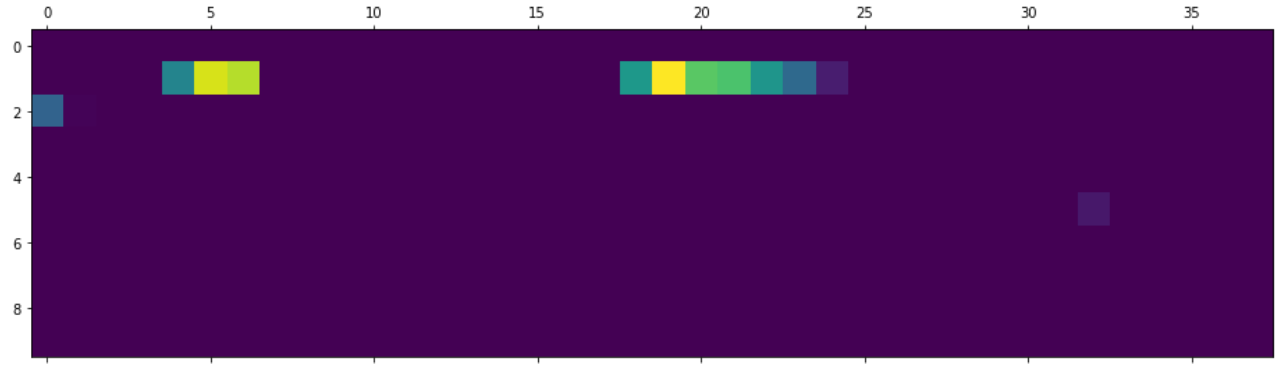
As can be seen from the following image, the dropout appeared to help and there is little evidence of overfitting.



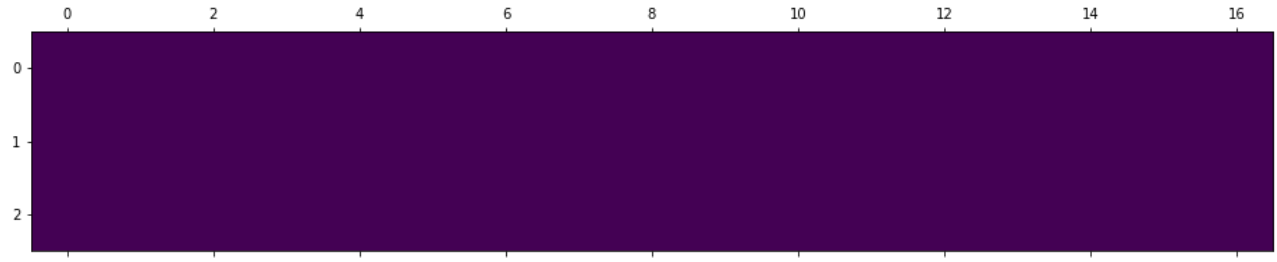
I did some visualitions of some of the lower convolutional layers, for a couple of images with high steering measurements, to ensure that they were still of benefit. Here are a couple of examples from the final two convolutional layers, for a sample with a steering measure above 0.5.







However, I also found quite a few ones that looked like this:



suggesting that quite a few of the filters in the final layer were adding very little to the model. I suspect that it would be possible to significantly reduce the number of parameters, but as self-driving cars is not my primary work area, I will save this effort for another day.