Behavioural Cloning Project Writeup

Charlie Wartnaby

Applus IDIADA

charlie.wartnaby@idiada.com

Udacity Self-Driving Car Engineer Nanodegree

Term 1 Computer Vision and Deep Learning

29 Aug 2017

Submission 2: colour handling mismatch between model.py and drive.py now fixed (thank you), which gave much improved driving performance without model retraining.

Abstract

A convolutional neural network model based on the NVIDIA self-driving car architecture was trained to successfully drive a simulated car around a virtual track, providing steering input at a controlled speed. The NVIDIA model was modified a) to reduce the size of the convolutional layers given the (presumably) lower information content in the low image resolution images in this project, and b) to add dropout layers to avoid overfitting. Variants of LeNet were also tried, but proved less successful.

Training data was collected in the same simulation. It was found necessary to augment the training data with many cases of the car recovering from heading on an off-track trajectory to make the model robust against such scenarios. In addition, general successful driving in both directions around the track was recorded and used as model training input.

To double the dataset size and avoid left/right bias, a horizontally flipped copy of each image and its negated steering value was also added to the training set.

It was noted that this exercise did not really apply independent test data; the final test was to navigate the track on which the model was trained, and that training was influenced by whether the test was successful, and in how it failed when it did not. So as a future enhancement, it would be better to train on different tracks, and only test on the target track as a last step, treating that as held-back final test data. As it is, the model is likely to be excessively well fitted to this particular track. Although a second track was available in the program, that would not avoid the problem of using data from the final test track as a model input, and lack of time precluded training on the second track in this case.

Write-Up

This document is the project write-up, meeting the rubric points specified.

Quality of Code

The project code is in the accompanying code file, **`model.py`**, and is generously commented. Key fragments are shown in this write-up, but please see the actual source file for full detail.

It was not found necessary to use a Python generator function as all of the training data could be loaded comfortably into memory for training, which thus proved more efficient, so this optional point was not explored further.

Training Data

Training Process: Keyboard versus Mouse

A significant decision was the choice between using key presses or the mouse to control steering input when recording training data.

Subjectively, it was found easier to control the car using the mouse, which provides a continuous steering angle if the left button is held down throughout and then moved gently left or right as required, allowing small and appropriate steering angles. By contrast, the left and right arrow keys on the keyboard gave an immediate strong steering angle in the corresponding direction or zero very quickly after the key was released.

As well as being more difficult to control with the keyboard, it was apparent that the resulting logged data was much more strongly quantised. For example, a continuous curve followed using the keyboard would result in bursts of strong steering separated by 'gaps' of zero steering angle. But using the mouse, a smooth appropriate value of steering angle would be recorded. Here are some example fragments of logged data taking a left curve (not the *same* curve):

Keyboard steering angle	Notes on keyboard angle	Mouse steering angle	Notes on mouse angle
0	'gap' of zero steering	-0.15094	Almost continuous value, just a little quantised from mouse movement across discrete screen pixels
0		-0.15094	
0		-0.15094	
0		-0.15094	
0		-0.13208	
-0.1	'burst' of strong steering from key being held down for short time	-0.13208	
-0.3		-0.13208	
-0.2859351		-0.13208	
-0.09991371		-0.13208	
0	gap	-0.13208	
0		-0.13208	
0		-0.13208	
0		-0.13208	
0		-0.13208	
0		-0.11321	
-0.1	keypress burst	-0.11321	
-0.3		-0.11321	
-0.2277855		-0.11321	
-0.04404172		-0.09434	
0	gap, etc	-0.07547	
0		-0.07547	
0		-0.07547	
0		-0.0566	

0		-0.0566	

Although fitting a model to the bursty keyboard data through least-mean-squares optimisation should smooth those bursts and gaps into some kind of averaged response, it is surely better to fit the model to the higher-quality mouse-generated data from the outset.

Training Process: Speed

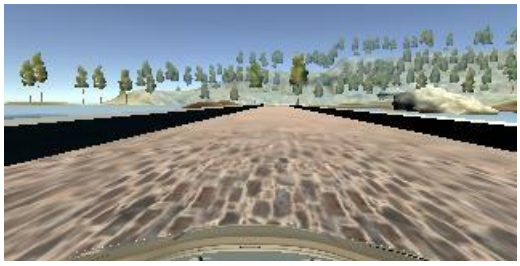
It was noted from drive.py that by default, it used a fixed set-point of 9 mph as the car speed, adjusting the throttle demand using a proportional-integral (PI) closed-loop controller to maintain that speed. This speed is fairly modest.

Training as a human, it was difficult to maintain accurate steering at 30 mph, but quite easy to do so at 10-15 mph. As this was similar to the speed that would be used in autonomous mode anyway, the car dynamics (e.g. slipping tyres) should be much the same (and in fact should be negligible at such low speed). By contrast, the simulator might include some tyre slippage when cornering hard at higher speed, which would cause the human driver to steer differently, yet that different steering would be inappropriate for the same corner played slowly in autonomous mode. So it could introduce a source of error training at too high a speed, as well as being difficult to steer well.

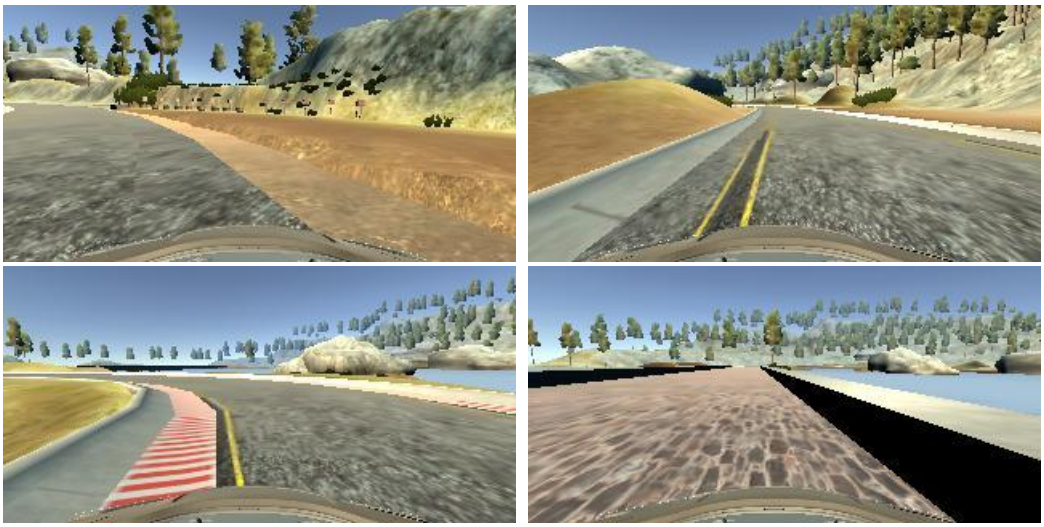
Therefore the training was done at a modest 10-15 mph, going more slowly round tricky corners to maintain accuracy. The lower cornering speed also meant more sample images in tricky areas, giving the model relatively more exposure to those difficult areas to train on.

Choice of Training Data

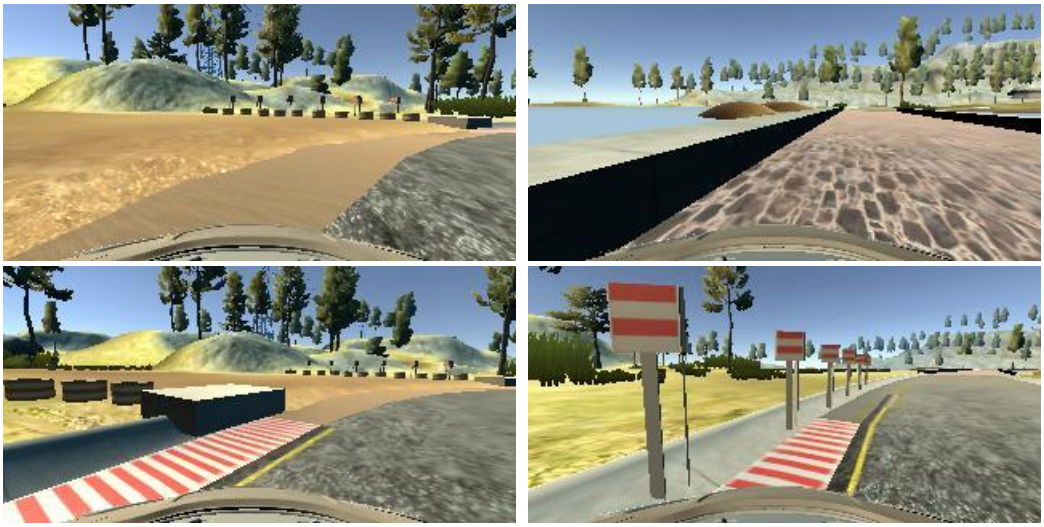
To start with, slightly more than a complete track circuit was recorded both going anticlockwise (the default), and clockwise. In both cases the bridge, which has very different appearance to the rest of the track, was crossed twice to increase the relative number of images for the model to work with, in the hope that its different textures and colour would then be captured more effectively. Here are some typical images:

A large number of 'recoveries' were recorded in both directions, where the car was driven to the edge (or slightly off the edge, where possible) of the track, and then data recorded as it was steered strongly back towards the centre of the track. Care was taken to do this with all the kinds of border encountered on the track (e.g. red/white hatching, white kerb, dirt, bridge wall), for example:



Initially however it was found that the trained model still tended to go off-track. This could be understood in that it had been given data on how to recover *from* being on the edge of the track, but no data on how to *avoid* getting onto the edge in the first place. So additional 'recoveries' were recorded, this time aiming the car towards the edge of the track, and then recording the strong steering input required to veer away from the edge before it was hit. This much improved the ability of the car to stay on-track when running autonomously. Here are some examples where the car is heading off-track, but actually being steered back towards centre:

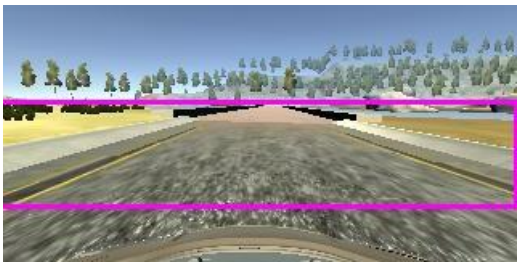


With that data added, the model did a fair job of following the track, even using LeNet (which is not really intended for this task).

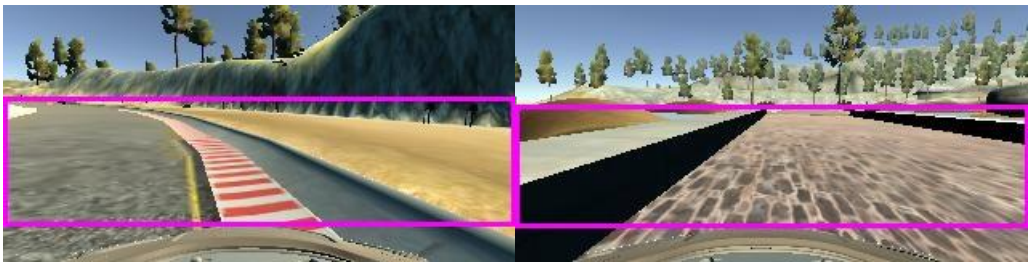
Cropping

Image cropping was applied to remove data from the image that the model should not attempt to fit (e.g. the sky, trees, hills etc), with the added bonus of shrinking the processing load.

A sample of images were examined to decide where to crop. In straight-ahead driving, a relatively large area of the bottom of the image seems to convey little information, as well as the portion above the immediate road horizon (vanishing point). As the track edges intersect the image edges well above the bottom of the picture here, it seems anything below that can be discarded, as suggested by the highlighted rectangle (which extends from 60 to 122 pixels from the top):



However, images taken from 'recovery' data where the car was going off track show that image data further down the image then becomes significant, because the track edge intersects the image edge towards the bottom, or even along its bottom edge. Here a reasonable cropping rectangle extends further down, to about 136 pixels from the top only to exclude the car bonnet/hood:



Recovering from track edges is important, so the model kept data further down to include that additional information in track-edge images.

At the top, the useful limit was always about 60 pixels from the top; although the car dipped and bounced a little when moving onto the bridge surface or hitting a kerb, this was not very significant. So overall, all images were cropped as shown in the track-edge pictures above, i.e. from 60 to 136 pixels from the top edge.

Model Architecture (Solution Design)

As suggested I started with the LeNet architecture from the previous course exercise. This was moderately successful, and overtraining was reduced by adding dropout layers. Please see this chunk in the submitted **`model.py`**:

```
# Final solution used modified NVIDIA architecture, but also experimented with LeNet
# with some success:
lenet = False
if lenet:
    # Try LeNet
    print("Using adapted LeNet architecture")
    #  Layer 1: Convolutional with 5x5 filter patch
    model.add(Convolution2D(6,5,5,activation="relu"))
    # Pooling across 2x2 patch
    model.add(MaxPooling2D())
    model.add(Dropout(0.3)) # CW added to avoid overfitting
    # Layer 2: Convolutional with 5x5 filter patch
    model.add(Convolution2D(6,5,5,activation="relu"))
    # Pooling across 2x2 patch
    model.add(MaxPooling2D())
    model.add(Dropout(0.3)) # CW added to avoid overfitting
    model.add(Flatten())
```

(...etc)

My final model used an adapted version of the NVIDIA self-driving car model shown in the tutorial video, but with modifications a) to reduce the depth of the convolutional layers given that we have less total information to start with than the model was originally intended for, given our(320x160) resolution here, and b) to reduce overfitting using dropout. Here is the code fragment, where significant changes to the NVIDIA original are highlighted in red:

```
else:
    # Nvidia example based on project tutorial video, but with some dropout added to
    # avoid overtraining
    print("Using adapted NVIDIA architecture")

    # Assuming Nvidia used higher resolution than 320x160, I experimented with cutting down the
    # size of the convolutional layers, as there is surely less input information from them to
    # learn from (this also sped up modelling of course!)
    resolution_divide_factor = 2

    # Also tried changing subsample here to 1,1 as another way of adjusting the model to work
    # with a coarser starting resolution (but settled on factor used throughout instead):
    model.add(Convolution2D(int(24/resolution_divide_factor),5,5,subsample=(2,2),activation='relu'))

    model.add(Convolution2D(int(36/resolution_divide_factor),5,5,subsample=(2,2),activation='relu'))
    model.add(Dropout(0.1)) # CW added to avoid overfitting
    model.add(Convolution2D(int(48/resolution_divide_factor),5,5,subsample=(2,2),activation='relu'))
    model.add(Convolution2D(int(64/resolution_divide_factor),3,3,activation='relu'))
    model.add(Dropout(0.1)) # CW added to avoid overfitting
    model.add(Convolution2D(int(64/resolution_divide_factor),3,3,activation='relu'))
    model.add(Flatten())
    model.add(Dense(100))
    model.add(Dropout(0.1)) # CW added to avoid overfitting
    model.add(Dense(50))
    model.add(Dropout(0.1)) # CW added to avoid overfitting
    model.add(Dense(10))
    model.add(Dense(1)) # CW to get final steering angle, not shown in tutorial I think
```

In both cases the Adam optimiser was used to fit the data:

```
# Using Adam optimiser as that seems to work nicely, hence no explicit learning rate needed
model.compile(loss='mse', optimizer='adam')

# This was a sufficient number of epochs in the end, though validation accuracy was still
# improving slightly so could have made it higher -- and that suggested it wasn't overfitting,
# perhaps thanks to the Dropout layers added
model.fit(X_train, y_train, validation_split=0.25, shuffle=True, nb_epoch=6)
```

So the final model consisted of the following layers:

Layer	Description
Input	320x160x3 RGB image (and corresponding steering angle as output)
Convolution + RELU	kernel depth=12, 5x5 patch, 2x2 strides
Convolution + RELU	kernel depth=18, 5x5 patch, 2x2 strides
Dropout	discarding random 10%
Convolution + RELU	kernel depth=24, 5x5 patch, 2x2 strides
Convolution + RELU	kernel depth=32, 3x3 patch, no strides
Dropout	discarding random 10%
Convolution + RELU	kernel depth=32, 3x3 patch, no strides
Fully connected	output size=100
Dropout	discarding random 10%
Fully connected	output size=50
Dropout	discarding random 10%
Fully connected	output size=10
Fully connected	output size=1 (in order to get single steering angle value)

Avoiding Overfitting

In early experiments using the LeNet architecture, it was noted that the training loss sometimes became very low, but the validation loss was quite high, symptomatic of overfitting to the training subset.

Dropout layers were added to the keras model to address this.

Similarly in further development with the NVIDIA model, several dropout layers were introduced. The final output shows that validation loss was similar to training loss and still decreasing, so overtraining had been avoided successfully:

```
(C:\udacity\Miniconda3\envs\carnd-term1) C:\svn\courses\udacity\self_driving_car\CarND-
Behavioral-Cloning-P3>python model.py
Adding horizontally flipped version of each image with negated steering value too
Loading index data from recordings\left_circuit_centred\driving_log.csv retaining 1 in 1 images
... 4061 lines retained from index file
Now have 8122 images and corresponding steering values
Loading index data from recordings\right_circuit_centred\driving_log.csv retaining 1 in 1 images
... 4452 lines retained from index file
Now have 17026 images and corresponding steering values
Loading index data from recordings\recoveries\driving_log.csv retaining 1 in 1 images
... 12545 lines retained from index file
Now have 42116 images and corresponding steering values
Using TensorFlow backend.
Using adapted NVIDIA architecture
Train on 31587 samples, validate on 10529 samples
Epoch 1/6
31587/31587 [==============================] - 490s - loss: 0.0702 - val_loss: 0.0430
Epoch 2/6
31587/31587 [==============================] - 460s - loss: 0.0403 - val_loss: 0.0407
Epoch 3/6
31587/31587 [==============================] - 476s - loss: 0.0314 - val_loss: 0.0270
Epoch 4/6
31587/31587 [==============================] - 484s - loss: 0.0269 - val_loss: 0.0222
Epoch 5/6
31587/31587 [==============================] - 486s - loss: 0.0234 - val_loss: 0.0212
Epoch 6/6
```

Parameter Tuning

The Adam optimizer was used, and therefore no tuning of learning rate was required.

Conclusions

Given more time, it would have been more robust to include training data from the second track available in the simulator. As it is, the model has been effectively tested in the same domain in which it was trained. For a production system, that would be fine if the aim was to drive on that specific track; but not if should handle all tracks.

The car also tended to 'hunt' or wander from the centre line sometimes. This perhaps reflected that the training data became dominated by recoveries to prevent off-track excursions, with perhaps not enough 'ordinary' driving round the track for it to learn how to do that smoothly. Lack of time precluded further training, however.