Udacity: Self Driving Car Engineer Nanodegree

Project 4 – Behavioural Cloning

Submitted by Ayeda Sayeed

A) INTRO AND SUMMARY

1. Submitted files

In this project, I tried and saved multiple methods. The workspace submission contains the best model and video for Track 1. My project includes the following files:

- `modeld.py` containing the script to create and train the model, with comments explaining functionality
- `modeld.h5` containing a trained convolution neural network
- `video.mp4` record of the network model performing on Track 1 (corresponds to `video4-singled_track1)`
- `Writeup.pdf` the report which you are reading

2. Additional files

Additional videos and models can be found here:
https://www.dropbox.com/sh/nppusjwynr8836e/AAATdSwuikc60tnVQRFBxnJPa?dl=0

Table A.1: Files associated with each model that can be found at the above link

Model	Model Script	Model File	Track 1 videos	Track 2 videos
Original	`model.py`	`model.h5`	• `video1-og_track1` • `video2-og_track1_overrides`	`video7-og_track2`
Single Dropout ***BEST MODEL	`modeld.py`	`modeld.h5`	• `video4-singled_track1` • `video3-singled_track1_overrides`	`video5-singled_track2`
Double Dropout	`modeldd.py`	`modeldd.h5`	`video8-doubled_track1_overrides`	`video6-doubled_track2`

**"overrides" means that the video includes durations where I took over to drive the vehicle incorrectly and then let the model correct for my behaviour*

3. Appropriate model architecture

I used a two-part CNN architecture for my model, with initial layers for normalization and image cropping, and multiple final fully-connected layers.

4. Overfitting

To correct for the model overfitting with Track 1 data, I experimented with dropout layers, so that the model might have improved performance on Track 2, before training with Track 2 data.

5. Model parameter tuning

I used the Adam optimizer to automatically tune my parameters like the learning rate, instead of tuning them manually.

6. Training data

Because I am not skilled at video games, I opted to use the sample data of Track 1, provided by Udacity, to begin my training. I had the intention of implementing a sort of transfer learning, saving the weights from Track 1, and continuing to train with Track 2, and I discovered that I am not unskilled at driving video games; I am terrible, and would cause serious errors in my model. Section B3 goes into further detail of my augmentation of the sample data, and what I would do if I had access to good Track 2 data.

B) MODEL ARCHITECTURE DESIGN AND TRAINING

1. Solution design approach

1.1 Concept

As a basis for my model, I referred to NVIDIA's paper titled "End to End Learning for Self-Driving Cars". Their architecture features a CNN consisting of 5 layers:

Table B.1: Basic architecture of the model used by NVIDIA

Convolution layer 1	# of filters = 24	Filter size = 5	Stride = 2
Convolution layer 2	# of filters = 36	Filter size = 5	Stride = 2
Convolution layer 3	# of filters = 48	Filter size = 5	Stride = 2
Convolution layer 4	# of filters = 64	Filter size = 3	Stride = 1
Convolution layer 5	# of filters = 64	Filter size = 3	Stride = 1

And 3 additional dense layers. My initial architecture, before I discovered the sample code in the lessons: *I implemented these layers by pairing a convolutional layer whose stride is 2 with a max pooling layer instead. This resulted in a large flattened layer (and thus more parameters), but I opted not to max pool any further for fear of losing too much information. My dense layers reduced in size by nearly the same ratios as the NVIDIA paper to yield a final layer of 10 neurons. I also had to give some pooling layers valid paddings to ensure integer output dimensions.*

1.2 Original Model

I quickly realized that it was safer to follow exactly the code provided in the lessons, and my architecture now looked almost identical to the NVIDIA architecture. First, I applied image normalization using a Lambda layer (*LINE 70* of `model.py`). I also used a Cropping2D layer to remove the unwanted features of the sky and the hood of the car (*LINE 72*). This is more effective than cropping when batching the images, because it ensures that the images fed to the final model in autonomous mode are of the same format. The CNN begins at *LINE 76*.

This model gave me very good MSE for both training and validation (around 0.02), implying minimal overfitting, and so I tested it in autonomous mode without any further tuning.

This first model, `model.h5`, works beautifully on track 1. It can even recover from manual overrides that send the car swerving towards the shoulders, where the model is not accustomed to those particular camera angles. As expected, however, this model did not perform well on track 2, which is completely unfamiliar to it.

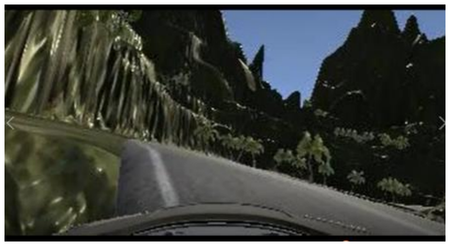

Figure 1.1: Final resting place for model.h5 on Track 2

1.3 Tuning for Track 2

I tried adding some additional dropout layers and ReLu activation layers, to reduce the model overfitting for Track 1, but inevitably, these subsequent models were still unable to make it far past the first turn of track 2, because of the vast differences between the visualization of the roads. I would need driving data from Track 2 to continue to train the model.

I had three major models: “original”, “single dropout”, and “double dropout”. Single dropout, `modeld.h5`, added 1 dropout layer with 50% dropout after the first three 5x5 convolutional layers. It performed equally well as original on track 1, even with manual overrides. It performed better than original on track 2, reaching a little bit past the top of the first hill.

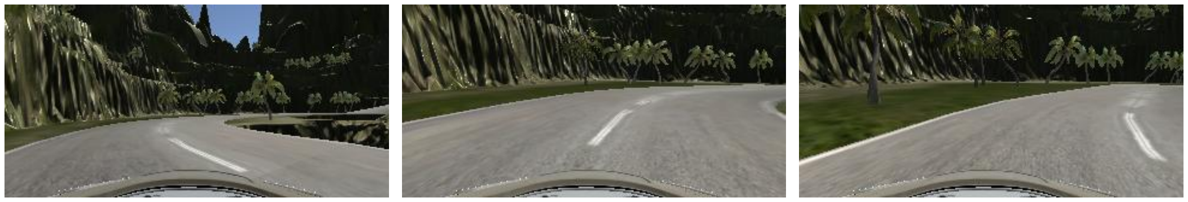

Figure 1.2: In succession, final 6 seconds before modeld.h5 runs the vehicle off of the track

With this success, I added another dropout layer, after the flattening, and ReLu activations after each dense layer. This did not improve performance for Track 2 any further, in fact it seemed to perform worse than original. Double Dropout’s performance (`modeldd.h5`) for Track 1 was the same as the other models. The MSE and validation accuracies were also very similar for all three models.

2. Final model architecture

The final, or best model, Single Dropout, is described in the following table and the subsequent image.

Table B.2: Final model architecture for this project

Layer #	Layer type	Description	# of filters	Kernel size	Stride
1	Lambda	for normalization			
2	Cropping2D	To crop sky and car hood			
3	Convolution2D		24	5	2
4	Convolution2D		36	5	2
5	Convolution2D		48	5	2
6	Dropout	50%			
7	Convolution2D		64	3	1
8	Convolution2D		64	3	1
9	Flatten	To produce 1 dimensional vector for single numerical result			

10	Dense	100 neurons			
11	Dense	50 neurons			
12	Dense	10 neurons			
13	Dense	1 neuron (to produce single numerical result)			

3. Creation of training set & training Process

The simulator has three camera available: center, left, and right. Instead of using only the center camera first, and then adding data from the other cameras, I started my model off with all three camera angles to compile a sort of comprehensive training set. The center camera image was associated with the recorded steering angle. A soft offset was applied to the steering angle to be associated with the left camera image, indicating the car should turn less towards the left and more towards center. A hard offset was applied for the right camera image, indicating the car should turn more left. I used generators to access the data, found in Lines 20-53 of model.py.

I used only the sample data provided by Udacity, which consisted of 8036 frames, providing a total of 24108 data points for training (80%) and validation (20%) of the model. The sample data consisted of “center lane driving”, where the vehicle remained in the center of the lane, effectively keeping it on the track. Since it was clear from testing that this training set proved more than satisfactory for the model to drive autonomously on Track 1, I decided that I didn’t need further data augmentation for the Track 1 data. Recording recovery driving, flipping the images and angles, or driving backwards were unnecessary when the model responded very well to manual swerving and made both left and right curves equally well.

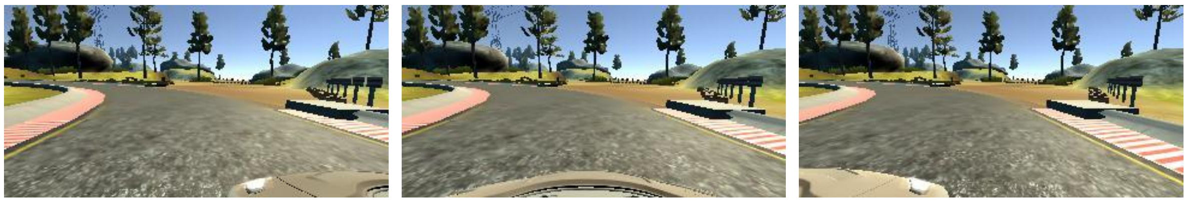

Figure 3.1: Images from left, center, and right cameras from a single frame of the sample data

I was, however, missing data from Track 2, so that the model could become familiar with the road, and further augmenting the data from Track 1 was not going to suffice as an alternative. If I was able to generate good driving data from Track 2, I would continue my training process as follows:

1. In the model script, Load both sets of data the same way, but combine the data from both tracks and shuffle.
2. Compile and train
3. Test in autonomous mode on both tracks. If performance not satisfactory, record recovery driving data for training.
4. In the model script, load only this new set of data.
5. Load the previous model and set its weights to trainable.
6. No need to build a new model or add any layers (until adding further dropouts is necessary)
7. Compile and train
8. Test again, and based on MSE and accuracy from training, decide which layers to add or remove.
9. In model script, load the previous model, and attach dropout layers to existing layers in the restored model.