Udacity Self Driving Car Project 3- Deep Learning to Clone Driving Behavior

This paper is made possible with the help of my outstanding October cohorts, John Chen, who believed in my getting this project completed, Vivek Yadov, who is a world class machine learning teacher and is most passionate about the Udacity self driving class, Matthew Zimmer, who kept in contact after he completed project 3. Thanks to the awesome Udacity staff, Ryan Keenan, Dhruv Parthasarathy and David Silver for their support. I love you all. Another thanks to Ryan Keenan for his expertise on this article. Grateful to everyone that answered or asked a question in the slack #p-behavioral-cloning channel. The NVIDIA document was key to the getting this project to succeed. It is listed in the reference section. I started out hacking with python to learning keras model functions, convolutional neural networks and above all python generators.

The universe aligned and I become a part of the Udacity Self Driving Car, SDC, pilot program that launched in October 2016. SDC is a vast improvement over in person graduate courses I have taken, because of the responsive Udacity staff, world class students admitted, the class lectures with quizzes and well designed projects. My colleagues are knowledgeable about machine learning, enjoy learning and are kind people. The forum for questions and the interactive slack channels assure my questions are answered thoroughly and quickly.

This article is about collecting training data , processing this data, testing this data with the provided interface python file on the autonomous track. If the car drives without crossing the yellow lines, success! Figure 1 illustrates this. If you use my training data go back to step one collecting new training data. If you use Udacity's provided training data this step one may be skipped. I found hundreds of ways where a car does not stay in the yellow lines on the autonomous track.

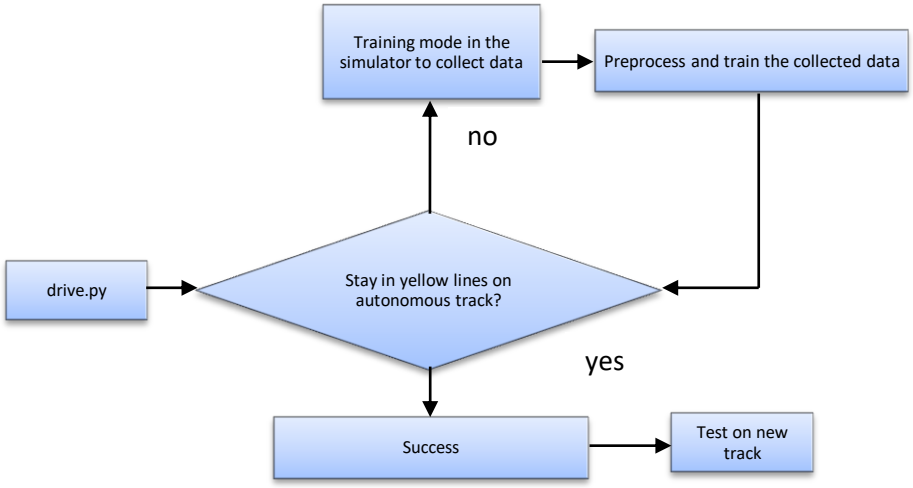

Figure 1 - Process of Deep Learning and Behavioral Cloning Project

Collect Training Data

The first step is to collect the training data using a Udacity vehicle driving simulator. There are three virtual cameras in the car at center, left and right locations. All three cameras take a picture at an optimal frames per second. The car is driven around the track two or three times using a joystick or keyboard arrows.

Preprocess Training Data

The images that are collected are preprocessed to remove some of the straight steering wheel angles to help the car turn when necessary, normalization for speed in compiling the model, cropping the picture for relevant driving range, flipping images to create more useful data, transforming right and left images and resizing the image to make it similar to the NVIDIA model I emulated.

A left image from the collected training images is shown in figure 2.

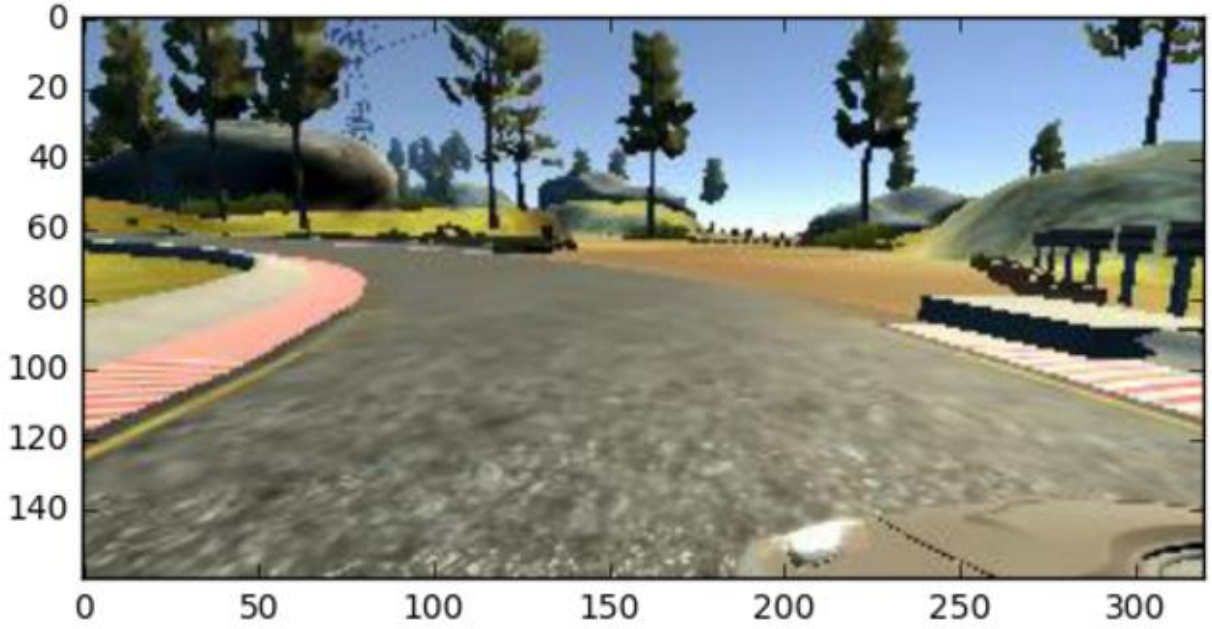

Figure 2 - Left Image From Training Data

The front of the car, the hood, is not relevant in the training data. Twenty percent or 32 pixels are removed from the bottom of the image. Twenty five pixels are removed from the vertical top of the image to not train the trees and sky but the road. The new cropped image is show in figure 3.

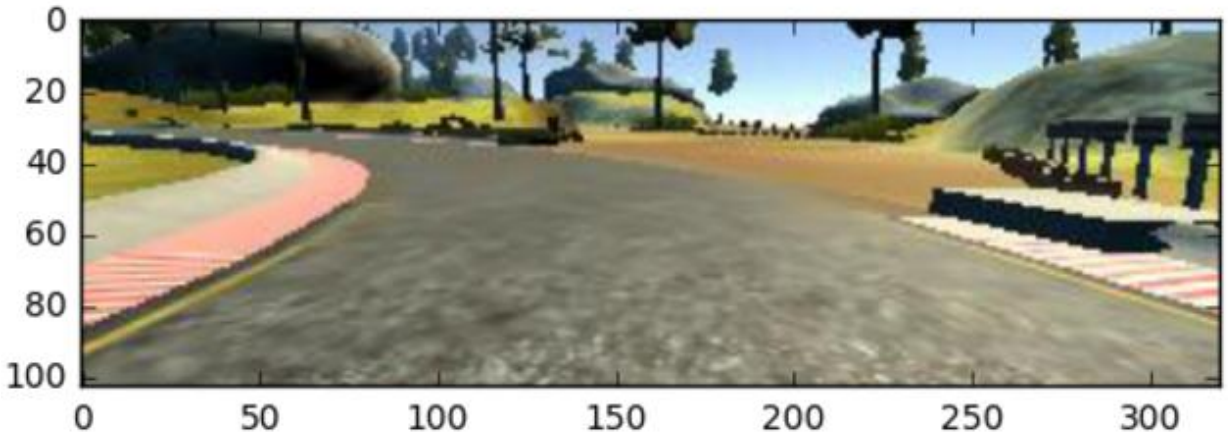

Figure 3 - Cropped Left Image From Training Data

The cropped image is resized to closely match the known working NVIDIA model. This image has 8 more pixels in the horizontal direction than the NVIDIA model. Taking less pixels may not work with this model.

The cropped image is resized to closely match the NVIDIA image size to (66, 208, 3) size. This is the input shape to be sent to the keras sequential model. Figure 4 shows a picture of the resized image.

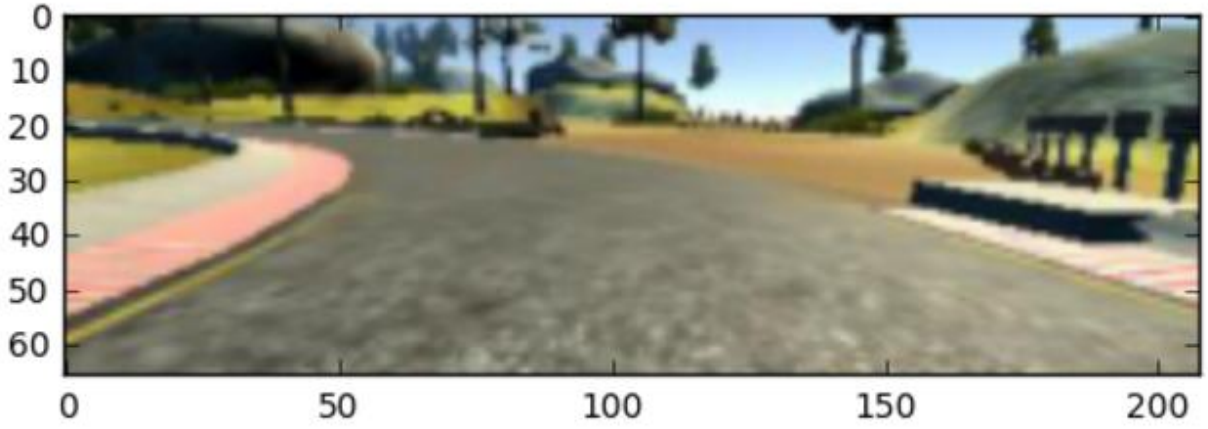

Figure 4 - Resized Left Image From Cropped Image

Half of the training images in each epoch are flipped. If the training data is mostly left or right turns then it will train the car with a bias towards left or right steering. The images are flipped to counteract that bias to simulate driving in the reverse direction. Figure 4 shows the horizontally flipped image.

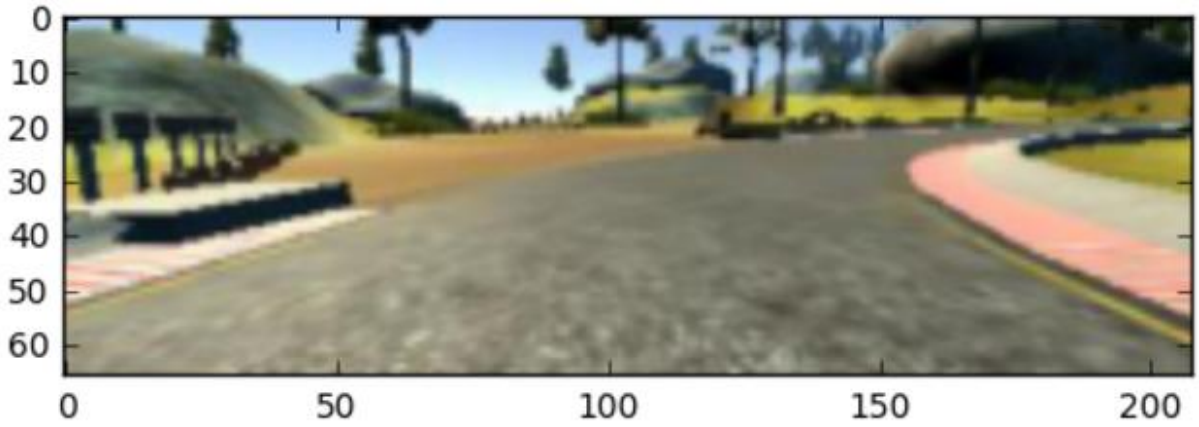

Figure 5 - Flipped Image From Resized Image

After the preprocessing is completed the images are trained with a keras convolutional neural network. A keras fit_generator is used in this project.

The versatile keras fit_generator() function

The keras `fit_generator()` function is mainly used to save memory space. A custom generator was created to send a batch size of one to the keras `fit_generator()`. This generator has some great features that are used in the this project. The `callbacks` function in the `fit_generator()` is used to save the weights after each epoch. The weights of each epoch may be run on the simulator to determine which works best. I found one thousands ways where the keras model did not work. `Fit_generator()` allows you to use the same custom generator for validation data and training data. The `nb_val_samples` attribute is only relevant if `validation_data` is a generator. The autonomous program tests my training data, that is considered the validation data. Therefore `validation_data` is set to **None**. Below is the command to use model.fit_generator.

```
history = model.fit_generator(batchgen(x_train_less, y_train_less),
        samples_per_epoch = samples_per_epoch, nb_epoch = nb_epoch,
           verbose=1, max_q_size = max_q_size, callbacks=callbacks_list,
        validation_data=None, class_weight=None,
            pickle_safe=False)
```

The custom generator, `batchgen()`, sends a batch size of one to the keras `model.fit_generator`. The custom generator batch size must be an integer multiple of the `samples_per_epoch`. I chose one sample size to use every Udacity training data sample. The training data may be on a server half across the world and will be downloaded one sample at a time saving local memory for other processes. The `max_q_size` is the cache size of samples the `fit_generator()` holds. In this project number 32 was chosen.

The callbacks function in the `fit_generator()` is versatile This makes it possible to test each epoch on the autonomous track to determine which is best. I suspect there is much more than can be done with this callbacks function that I named it a list, `callbacks_list` as a reminder of its possibilities. Here is the callbacks function I used.

```
checkpoint = ModelCheckpoint("model-{epoch:02d}.h5", monitor='loss',
verbose=1, save_best_only=False, mode='max')

callbacks_list = [checkpoint]
```

The keras `fit_generator()` provided the opportunity to become proficient in python.

Preprocessing the data and training a model is illustrated in figure 6, Keras Convolutional Neural Network Architecture.

Keras Convolutional Neural Network Architecture

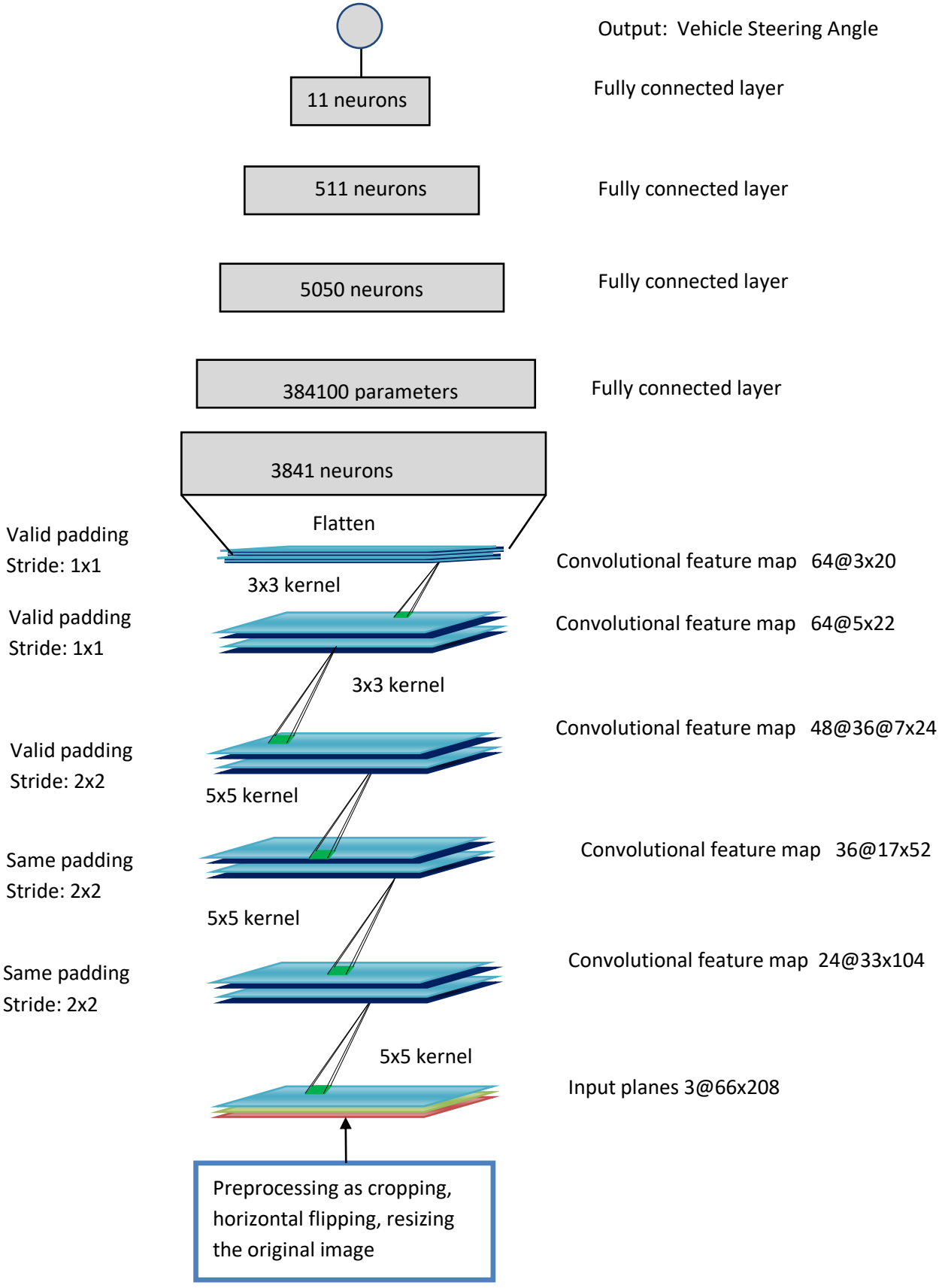

Figure 6 - Model Architecture

model.summary() command provides the following output:

Layer (type)	Output Shape	Parameters	Connected to
conv1 (Convolution2D)	(None, 33, 104, 24)	1824	input
elu_1 (ELU)	(None, 33, 104, 24)	0	conv1[0][0]
conv2 (Convolution2D)	(None, 17, 52, 36)	21636	elu_1[0][0]
elu_2 (ELU)	(None, 17, 52, 36)	0	conv2[0][0]
conv3 (Convolution2D)	(None, 7, 24, 48)	43248	elu_2[0][0]
elu_3 (ELU)	(None, 7, 24, 48)	0	conv3[0][0]
conv4 (Convolution2D)	(None, 5, 22, 64)	27712	elu_3[0][0]
elu_4 (ELU)	(None, 5, 22, 64)	0	conv4[0][0]
conv5 (Convolution2D)	(None, 3, 20, 64)	36928	elu_4[0][0]
elu_5 (ELU)	(None, 3, 20, 64)	0	conv5[0][0]
flatten_1 (Flatten)	(None, 3840)	0	elu_5[0][0]
hidden1 (Dense)	(None, 100)	384100	flatten_1[0][0]
elu_6 (ELU)	(None, 100)	0	hidden1[0][0]
hidden2 (Dense)	(None, 50)	5050	elu_6[0][0]
elu_7 (ELU)	(None, 50)	0	hidden2[0][0]
hidden3 (Dense)	(None, 10)	510	elu_7[0][0]
elu_8 (ELU)	(None, 10)	0	hidden3[0][0]
output (Dense)	(None, 1)	11	elu_8[0][0]

Total parameters: 521,019

First Convolution Layer - conv1

For the input planes with a shape of 66x208x3, (HxWxD)

conv1 has 24 filters of shape 5x5x3 (HxWxD)

A stride of 2 for both the height and width (S)

Same padding of size 1 (P)

The formula for calculating the new output height or width of conv1 layer is:

new_height = (input_height - filter_height + 2 * P)/S + 1

new_width = (input_width - filter_width + 2 * P)/S + 1

new_height = (66-5 +2*1)/2 +1 = 33

new_width = (208-5+2*1)/2 + 1 = 104

Output shape of conv1 layer is (33, 104, 24)

To calculate the parameters with parameter sharing, each neuron in an output channel shares its weights with every other neuron in that channel. So the number of parameters is equal to the number of neurons in the filter, plus a bias neuron, all multiplied by the number of channels in the output layer.

Parameters = (5 x 5 x 3 + 1) * 24 = 1824

Second Convolution Layer - conv2

For the input planes with a shape of 33x104x24, (HxWxD)

conv2 has 36 filters of shape 5x5x24 (HxWxD)

A stride of 2 for both the height and width (S)

Same padding of size 2 (P)

The formula for calculating the new output height or width of conv1 layer is:

new_height = (input_height - filter_height + 2 * P)/S + 1

new_width = (input_width - filter_width + 2 * P)/S + 1

new_height = (33-5 +2*2)/2 +1 = 17

new_width = (104-5+2*2)/2 + 1 = 52

Output shape of conv1 layer is (17, 52, 36)

Parameters = (5 x 5 x 24 + 1) *36 = 21636

Third Convolution Layer - conv3

For the input planes with a shape of 17x52x36

conv3 has 48 filters of shape 5x5x36 (HxWxD)

A stride of 2 for both the height and width (S)

Valid padding of size 1 (P)

The new output height or width of conv1 layer is:

new_height = (17-5 +2*1)/2 +1 = 8

I calculate 8, the model.summary() calculates 7. I do not know why the model does not match this answer. I will use the model.summary() 7 for the new_height.

new_width = (52-5+2*1)/2 + 1 = 25

I calculate 25 model.summary() calculates 24. I am sure I am right, but I am going to use model.summary() result.

Output shape of conv1 layer is (7, 24, 48)

Parameters = (5 x 5 x 36 + 1) * 48 = 43,248

Fourth Convolution Layer - conv4

For the input planes with a shape of 7x24x48

conv4 has 64 filters of shape 3x3x48 (HxWxD)

A stride of 1 for both the height and width (S)

Valid padding of size 1 (P)

The new output height or width of conv4 layer is:

new_height = (7-3 +1*1)/1 +1 = 4

I calculate 4, the model.summary() calculates 5. I do not know why the model does not match this answer. I will use the model.summary 5 for the new_height.

new_width = (24-3+2*1)/1+ 1 = 24

I calculate 24, the model.summary() calculates 22. I do not know why the model does not match this answer. I will use the model.summary 5 for the new_height.

Output shape of conv1 layer is (5, 22, 64)

Parameters = (3 x 3 x 48 + 1) * 64 = 27,712

Fifth Convolution Layer - conv5

For the input planes with a shape of 5x22x64

conv4 has 64 filters of shape 3x3x64 (HxWxD)

A stride of 1 for both the height and width (S)

Valid padding of size 1 (P)

The new output height or width of conv5 layer is:

new_height = (5-3 +1*1)/1 +1 = 4

I calculate 4, the model.summary() calculates 3. I do not know why the model does not match this answer. I will use the model.summary 3 for the new_height.

new_width = (22-3+1*1)/1+ 1 = 21

model.summary() calculates 20.

(model.summary only uses even numbers for the last two numbers in the output shape)

Output shape of conv5 layer is (3, 20, 64)

Parameters = (3 x 3 x 64 + 1) * 64 = 36,928

Flatten

The output shape of conv5, (3, 20, 64) is flattened to 3840 neurons.

Fully Connected Layers

There are four fully connected hidden layers in this model as how in figure 2.

This simple model does not benefit from dropout layers. The biggest cost is training the data using AWS. A GPU is necessary to train this data. The saving of weights for each epoch and the final model saves computing time.

Validation on the Autonomous Track

The files to run on the autonomous track are the model output file and a drive file. Modifying the drive file allows the data to not be retrained. In this example, drive files was edited to undo a change in the model without retraining the data and using AWS again. The way to change the model output file is to retrain the data.

After training data for weeks, making progress, and still crossing the yellow lines a few times around the track, I wondered, "What would Vivek Do?". He recommended I modify the drive.py file on the steering output to put a damper on my sinusoidal driving around the track. It helped immensely. While in the drive.py file, I read, "feel free to change this". I then changed the throttle to go above 30 miles per hour on the autonomous track. The car hit the side barrier immediately. This gave me an idea to change the throttle to 15 miles per hour. It worked on the track. The car would get a ticket for driving too slow in Mountain View, California. I have seen my cohorts videos where they drive around the autonomous track above 30 miles per hour. Here is my video on youtube.

Project 3 has been the most challenging project where my learning has been exponential as I completed the project. I look forward to completing projects 4 and 5.

References

1. John Chen, Vivek Yadov, Ryan Keenan and October cohorts on the forum and slack project channels.

2. Udacity lecture movies and written material. https://www.udacity.com/self-driving-car

3. NVIDIA paper https://images.nvidia.com/content/tegra/automotive/images/2016/solutions/pdf/end-to-end-dl-using-px.pdf

4. Comma.ai https://github.com/commaai/research/blob/master/train_steering_model.py

5. Keras documentation https://keras.io/