# Project 3: Use Deep Learning to Clone Driving Behavior

## Goals

* 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

## Summary of results

The project was executed to completion. The main challenge I found was translating the knowledge I had from categorization into more of a regression problem.

The second issue I faced was that I started with a model that wasn’t complex enough to keep the car in the driving lane as expected. I ended up replacing my simple model by one based on the NVidia whitepaper referenced in the project definition titled “End to End Learning for Self Driving Cars”.

In the process, I collected my own driving data which I used, along with the Udacity demo data, to train the neural network. The data was then preprocessed, augmented, and normalized before being split into training and validation data, and getting used to train the model. The resulting model was then saved to file model.h5.

Finally, the model could smoothly drive around track one without leaving the road, and reacting very safely.

# Rubric points

Here’s my justification for each of the rubric points for this project, as specified in <https://review.udacity.com/#!/rubrics/432/view>.

## Required files

### Are all the required files submitted?

This project submission includes the following files:

* Model.py
* Drive.py
* Model.h5
* README

They can also be retrieved from <https://github.com/DavidObando/carnd/tree/master/Term1/Project3>.

## Quality of code

### Is code functional?

The code is functional, it can be tested by executing:

python model.py

and

python drive.py model.h5

### Is the code usable and readable?

The rubric suggested using a Python generator, and one is included in the code. That said, in my local machine I found that not using a generator provided me with better results given the memory requirements of the model and the size of the input data. I ran my model in a GPU-accelerated machine with 8 GBs of video memory, which could accommodate and execute the model in training with batches of 512 images per cycle.

The use of Python generators allows us to not be memory constrained, but its tradeoff is time. When I resorted to using a Python generator, each epoch took around 85 seconds to complete, as the processing of the images required the generation of the entire set on each epoch, and used CPU cycles that were slower than the GPU alone could do with a predefined data set. I used the [fit\_generator](https://keras.io/models/sequential/#fit_generator) function to train the model using a Python generator.

Given that my model could train in roughly 1 second per epoch on the GPU with almost twenty thousand images, I decided to ultimately not use Python generators in the final implementation. I’m unsure if I could have improved the performance I obtained using the Python generator, but given that the results obtained without it were much better (performance-wise, in my computer) I decided to not invest more in going this route for now. I do appreciate knowing of this going forward as I’m sure I’ll run into datasets that won’t be easy to handle in my computer.

The code has been organized in a logical manner, divided in sections that make it easy to read.

## Model architecture and training strategy

### Has an appropriate model architecture been employed for the task?

This project challenged my understanding of neural networks in that, so far, I’ve only been doing categorization problems, such as the traffic sign categorization done in P2. When I started P3 it wasn’t clear to me how I was going to build a model to do what appeared to be a regression model. The Keras documentation for sequential model compilation [[here](https://keras.io/getting-started/sequential-model-guide/#compilation)] was useful in telling me two things:

1. Loss calculation for regression problems differs from that of categorization problems, and
2. Mean squared error is what I was looking for.

In retrospective, this should have been intuitive from the material we’ve seen. Given that I could recognize the need for mean squared error, I went and created a simple sequential model with this architecture:

1. Convolution (32 filters, 3x3 kernel, 1x1 stride, valid mode)
2. Activation (ReLu)
3. Max pooling (2x2 pool size)
4. Dropout
5. Convolution (64 filters, 2x2 kernel, 1x1 stride, valid mode)
6. Activation (ReLu)
7. Max pooling (2x2 pool size)
8. Dropout
9. Flatten
10. Fully connected (128 output)
11. Activation (ReLu)
12. Dropout
13. Fully connected (1)
14. Activation (Relu)
15. Compile (Adam optimizer, MSE loss)

Notice the mistake I made? I added an activation layer in step 14, before compilation, meaning that whatever value the fully connected layer was giving me in step 13 was going to get “activated” in the way we usually produce categorization problems, yielding a zero when the value from the previous layer was negative, and a one otherwise.

My first run was problematic, of course, but after going back to my notes I realized what I had made. I based this model from what we had done in the introduction to Keras, which was a categorization problem. After it became clear that I was looking for a result set in the range [-1, 1], I realized that the activation layer was damaging my results. After removing it, the model started to produce values that seemed plausible.

I trained the model with the data provided by Udacity, and then added my own data from runs in the simulator where I was driving the car myself. The resulting model could indeed drive the car but not safely, as it would sometimes go off track.

I recurred to the NVidia whitepaper titled End-to-End Learning for Self-Driving Cars [[here](http://images.nvidia.com/content/tegra/automotive/images/2016/solutions/pdf/end-to-end-dl-using-px.pdf)] which gave me a much better architecture:

1. Convolution (24 filters, 5x5 kernel, 2x2 stride, valid mode)
2. Activation (ReLu)
3. Dropout
4. Convolution (36 filters, 5x5 kernel, 2x2 stride, same mode)
5. Activation (ReLu)
6. Dropout
7. Convolution (48 filters, 5x5 kernel, 2x2 stride, same mode)
8. Activation (ReLu)
9. Dropout
10. Convolution (64 filters, 3x3 kernel, 1x1 stride, same mode)
11. Activation (ReLu)
12. Dropout
13. Convolution (64 filters, 3x3 kernel, 1x1 stride, same mode)
14. Activation (ReLu)
15. Dropout
16. Flatten
17. Fully connected (1164 output)
18. Activation (ReLu)
19. Dropout
20. Fully connected (100 output)
21. Activation (ReLu)
22. Dropout
23. Fully connected (50 output)
24. Activation (ReLu)
25. Dropout
26. Fully connected (1)
27. Activation (Relu)
28. Compile (Adam optimizer, MSE loss)

### Has an attempt been made to reduce overfitting of the model?

The architecture does contain dropout layers, which by default have been configured to be 50% during model training.

Additionally, the data used to train the model has been augmented to make the neural network both less reliant on the initial data set and more flexible so it understands variations in the input that might not have been captured initially.

### Have the model parameters been tuned appropriately?

I initially experimented with different values for batch size and number of epochs. I was using the default Adam optimizer that Keras provides. After reading some documentation I realized that the Adam optimizer can be further fine-tuned with regards to its learning rate, as well as a few other parameters. I noticed a big improvement when I chose to make the learning rate smaller than the default setting was, while also running a larger number of epochs.

I settled on the following values:

* Number of epochs: 1000 (model.py line 267)
* Batch size: 512 (model.py line 267)
* Learning rate for the Adam function: 0.00005 (model.py line 166)
* Dropout rate during training: 0.5 (model.py line 166)

### Is the training data chosen appropriately?

The model was trained from data given by Udacity, as well as data I collected by driving the car simulator in my own computer.

I loaded the images and did the following transformations:

1. Trim the image so only the road section is fed into the model
2. Resize to 64 pixels wide by 16 pixels tall.

Following that, I wanted to make use of the left and right camera data, not only the center camera data. Given that these cameras are offset from the center I modified the steering angle associated to them as indicated in the NVidia whitepaper. From the whitepaper:

Images for two speciﬁc off-center shifts can be obtained from the left and the right camera. Additional shifts between the cameras and all rotations are simulated by viewpoint transformation of the image from the nearest camera. Precise viewpoint transformation requires 3D scene knowledge which we don’t have. We therefore approximate the transformation by assuming all points below the horizon are on ﬂat ground and all points above the horizon are inﬁnitely far away. This works ﬁne for ﬂat terrain but it introduces distortions for objects that stick above the ground, such as cars, poles, trees, and buildings. Fortunately these distortions don’t pose a big problem for network training. The steering label for transformed images is adjusted to one that would steer the vehicle back to the desired location and orientation in two seconds.

I didn’t do any precise transformations the way they claimed they did, I only took the right and left camera images and adjusted the steering by 0.15. The left camera makes the road look like it’s curving to the right, so I added 0.15 to the steering value on the left camera. The right camera makes the road look like it’s curving to the left, so I subtracted 0.15. I then capped the values at the [-1,1] range originally provided. See model.py lines 118 to 125 for the implementation.

I chose 0.15 as the steering value with no basis on what number to use. I’m not sure how optimal or suboptimal this value is, given that the result was satisfactory and I didn’t adjust it much. I did play with 0.1 instead, but I reverted to 0.15 and the car was able to drive with this as expected.

After that, I ran three more transformations on the data:

1. Leaving only 10% of the center data
2. Balancing left and right data by mirroring the entire dataset
3. Normalizing the image data

I left only 10% of the center data as it originally was disproportionally large compared to any other value when visualized in a histogram. I did this so that the model wouldn’t overfit too much to the center.

The balancing of the left and right data was simple. I took all the images and flipped them using the cv2 library, and I negated its associated steering value. The resulting set was merged with the original one, thus giving me twice the amount of data I started with.

The data normalization was done with a minmax function, ensuring all the pixel values would be [-0.5 to 0.5]. See model.py lines 73 to 83 for its implementation.

## Architecture and training documentation

### Is the solution design documented?

TO DO: Readme file that thoroughly discusses the approach taken for deriving and designing a model architecture fit for solving the given problem.

### Is the model architecture documented?

TO DO: Readme file that provides sufficient details of the characteristics and qualities of the architecture, such as the type of model used, the number of layers, the size of each layer. Visualizations emphasizing qualities of the architecture are encouraged.

### Is the creation of the training dataset and training process documented?

TO DO: Readme file that describes how the model was trained and what the characteristics of the dataset are. Information such as how the dataset was generated and examples of images from the dataset should be included.

## Simulation

### Is the car able to navigate correctly on test data?

Yes.