Simon Fraser University
ENSC 424 Multimedia Communications

# Steering Direction of Self-Driving Cars Via Deep Learning

Project Report
December 5th, 2017

Group 1

| | |
|---|---|
| Fatemah Darbehani | 301212869 |
| Kimberly Kunimoto | 301220262 |
| Princess Macanlalay | 301232462 |
| Ying Hsin (Cindy) Lan | 301194824 |

## Abstract

Human error is the main cause of accidents on the road; therefore, self-driving cars are an alternative automotive option to reduce collisions. The following document contains information on how cars can learn to steer through the use of deep learning. A dashboard camera films an example path a typical car would take, which is acquired to be analyzed by the code. Using many clear images of the road, the code can compute the angle the steering wheel should turn. There are many limitations to this code, including weather restrictions, lighting of the road, and the type of road it is on. This document reveals one possible method that can assist in getting self-driving cars on the road and making it a leading option in the automotive industry.

## 1.0 Introduction and Background

The idea of having driverless cars has been around for a few decades, with experiments going as far back as the 1920s. It was 60 years later that the first truly autonomous car was built, which was the Navlab 1 developed in Carnegie Mellon University in 1986 [1]. However, Navlab had a lot of software drawbacks due to the technology back then. Nablab 1 had a top speed of 20 miles per hour - which translate to 32 kilometer per hour - was only to be allowed in semi-autonomous control. It consisted of multiple racks of computing equipment, including a Warp supercomputer.

Despite some of the technological constraints in the 1980s, it was in Carnegie Mellon University that introduced the use of neural networks in controlling autonomous vehicles. Since Navlab was semi-autonomous, neural networks were used to control the steering and brakes had to be human-controlled. Even to this day, the use of neural networks continue to be the basis of modern autonomous driving strategies.

Since the 2010s, several automotive manufacturers are testing their own self-driving car systems. Generally, they are equipped with multiple visible-light and infrared cameras, RADAR, LiDAR, and more in order to keep track of lanes, avoid collisions, and handle cruise control. [2]. More research are put into the improvement of machine learning and many companies are still working on building a fully autonomous car to be used publicly. While no self-driving cars are available to consumers at the moment, many are currently being tested by various companies all over the globe.

## 1.1 Literature Review

Convolutional neural networks (CNNs) has opened the world into many applications in image and video recognition. As tested in one of NVIDIA's research papers [3], CNN proves to be a powerful end-to-end approach in self-driving cars. NVIDIA also states that this method can provide minimal training data from humans and result with better performance and smaller systems. This removes the need to depend on human-designated features on the road, such as public markings, guardrails, or other cars as well as a collection of “if” and “else” cases.

NVIDIA trained a CNN with images from a single front-facing camera and with the angles applied on the steering wheel from a human driver. The weights are trained to minimize the mean squared error between steering angle output from the network and the steering angles of the human driver. Once trained, the network generates steering commands based from video inputs from a front-facing camera.

The network architecture used by NVIDIA is shown in Figure 1. The inputted video images are first converted into YUV and then is passed into the network. The network consists of nine layers: the first being a normalization layer, then five convolutional layers, and then three fully connected layers. The addition of a normalization layer in the network allows modification with the normalization scheme and faster processing. The convolutional layers was designed to be the most optimal based from a series of experimentation. These layers are meant to perform feature extraction. Lastly, the fully connected layers function to control the steering angles in which leads to the output vehicle control.

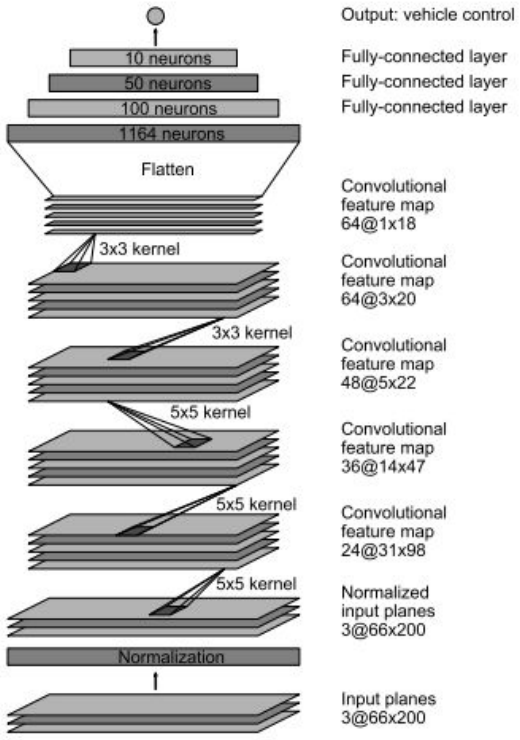


Figure 1: NVIDIA's CNN architecture for self-driving car

## 2.0 Project Description

This project focuses on the visible-light camera - as it is one of the more economical options out of the other sensors - to predict the steering angle of end-to-end self-driving cars using deep learning. We used a convolutional neural network (CNN) for analyzing the visual imagery and for controlling the output for steering. The main purpose of the project is to explore deep learning and to improve on the open source solution provided by MIT [4].

## 3.0 Initial Implementation Details

The implementation started on following the published NVIDIA's network architecture for end-to-end self-driving cars in the paper. To train this CNN we used the data from the project "DeepTesla" provided by The Massachusetts Institute of Technology (MIT) [5], which includes a set of 10 videos of over one hour of driving footage and corresponding steering angles with time stamps. The initial computation was done on a personal computer with a CPU. The entire group has decided to work on the project as a whole as we researched and discussed together.

## 3.1 Preprocessing

The original input data, which was in the form of a set of 10 videos, contains a total of 27000 frames of size 1280 by 720. Figure 2 is an example of a frame from the data.

Figure 2: Example of an Original Frame

To speed up the process of loading input images into memory for training, we converted these video files into individual images. We have then cropped the irrelevent areas such as pixels corresponding to the sky and pixels representing the dashboard as shown in Figure 3. We then saved them as png images of size of 200 by 66 for future processing. We kept 2700 frames for testing and divided the remaining frames into 80% training and 20% evaluation sets.

Figure 3: Example of a Cropped Data Image

Steering angles are saved in a text file along with the image they correspond to. Figure 4 is a screenshot of how our dataset is organized.

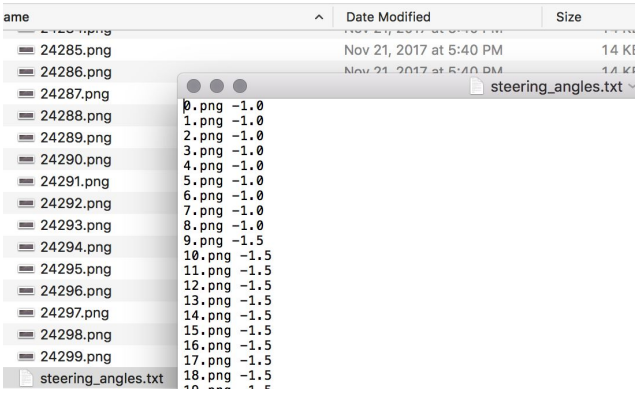


Figure 4: Data Values Stored in a Text File

The file “load_data.py” is responsible for loading only a certain number of input images for batch processing. We noticed that sometime we received “BAD MEMORY” error if we loaded the whole dataset instead for batch processing.

Then, we have converted the cropped images from RGB to YUV to reduce redundancy. Lastly, we have normalized the range to be in between of 0 to 1, so the data can be put into prescriptive; it would not be just purely value dependent. Only then, we are ready to feed the images as our input data set.

In conclusion, in preparation to use the data set, we have saved the actual steering angle results with corresponding images in a text file. The steps for image preprocessing is as follows:

1. Converted the original video files into individual frame of images
2. Cropped off irrelevant pixels from each of the images
3. Convert the images from RGB to YUV
4. Normalized the image to a range of 0 to 1
5. Divide the images into three sets: training set, evaluating set, and testing test

## 3.2 Building the Base Model

Our initial model was developed in Python using Keras 2.2.1 with Tensorflow 1.3.0 backend and was based on NVIDA’s CNN architecture shown previously in Figure 1. We used “ReLU (Rectifier)” as activation function for the convolutional layers and fully connected layers. This activation function was used by NVIDIA because it eliminates the problem of vanishing/exploding gradient descent, is linear, and only passes positive values. The output layer’s activation function is 2*arctan(x) to output an angle between -180 and 180 degrees. Unfortunately, Keras does not have a built-in activation function for that and we had to implement it using Tensorflow.

Figure 5 shows the implementation and usage of the arctan activation function.

```
18 def atan(x):
19     return tf.multiply(tf.atan(x), 2)
```

```
86     # ------------------------------------------------- #
87     # Step 5: Output Layer
88     # ------------------------------------------------- #
89     model.add(Dense(units=1, activation=atan, \
90                     kernel_regularizer=l2(l2_lambda)))
```

Figure 5: Arctan Function Implementation

We used Adoptive Moment Estimation (Adam) optimizer with learning rate of 0.0001 on Mean Square Error (MSE) loss function. We used Adam because it is the most common optimizer used in practice. Figure 6 is a summary of our model compiled with the above specifications.

| Layer (type) | Output Shape | Param # |
|---|---|---|
| conv2d_1 (Conv2D) | (None, 31, 98, 24) | 1824 |
| conv2d_2 (Conv2D) | (None, 14, 47, 36) | 21636 |
| conv2d_3 (Conv2D) | (None, 5, 22, 48) | 43248 |
| conv2d_4 (Conv2D) | (None, 3, 20, 64) | 27712 |
| conv2d_5 (Conv2D) | (None, 1, 18, 64) | 36928 |
| flatten_1 (Flatten) | (None, 1152) | 0 |
| dense_1 (Dense) | (None, 1164) | 1342092 |
| dense_2 (Dense) | (None, 100) | 116500 |
| dense_3 (Dense) | (None, 50) | 5050 |
| dense_4 (Dense) | (None, 10) | 510 |
| dense_5 (Dense) | (None, 1) | 11 |

Total params: 1,595,511
Trainable params: 1,595,511
Non-trainable params: 0

Figure 6: Summary of the Original Model

After compiling the initial model, we have trained it with a batch size of 20 and an epoch number of 30. Figure 7 shows the result obtained for our first attempt for the training.

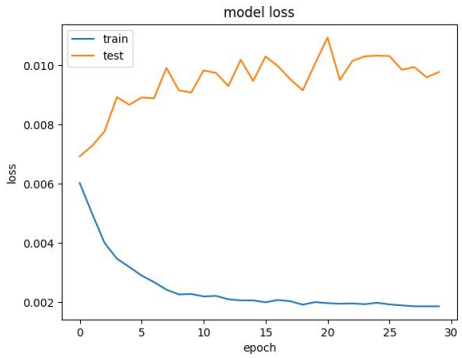


Figure 7: Model Loss of the Original Model

As observed, error increased for evaluation set as error decreased for training set. This is a typical case of overfitting as we have a large number of trainable parameters and dour dataset is not large enough for that.

## 4.0 Further Implementation Details

The first approach in improving the NVIDIA neural network for our database is to correct the overfitting issue. After we have made some changes to the neural network, we decided to compute more iterations on Amazon Web Services (AWS), which is a cloud machine learning engine with computational GPU. This allows us to create a more accurate neutral network as it can train more with less time compared to a local CPU. The corresponding output is saved in CSV format. Finally, we visualized our result and compare it with human recorded output to measure the accuracy of our trained network by recreating the video with the 2700 testing images.

## 4.1 Improving Upon Overfitting Issue

Overfitting happens when we have too many features in the fully-connected layers and that the contribution of all those features are too large. To compensate for the overfitting form the original design, we introduced pooling, L2-norm regularization, and dropouts to the model. This is done to control the capacity of the neural network. A quick background on each of the three components is as follows.

### Pooling

A certain feature that the neural network is detecting may seem different with other outside factors. For instance, the lane may look slightly tilted when the car turns to the right compared to when the car drives straight forward. Pooling builds spatial invariance for input images, which means that a feature could be recognized relative to the position of other features instead; Thus, the exact position of the testing feature in the frame does not not affect the output.

This is accomplished by reducing the dimensions of the feature map as we implement pooling at the convolutional layers. There are numerous types of pooling, such as max-pooling, mean-pooling, and sum-pooling. For our model, we have tried max-pooling to control overfitting. After the image has been convoluted, max-pooling records the maximum value for each non-overlapping subregion of size $p_{row} \times p_{column}$ of the feature map [6]. This process produces a reduced feature map while keeping the pertinent features. Therefore, the neural network can still recognize the object even if it varies in lighting, direction, shape and form to the other images.

### L2-norm regularization

Originally we have just used the mean square loss function. The equation for the L2-norm loss function is as follows:

$$S = \sum_{i=1}^{n} (y_i - f(x_i))^2$$

One of the most common method of to correct overfitting is by introducing a regularization term as shown in the equation as follows:

$$w^* = arg\ min \sum_j (t(x_i) - \sum_i w_i h_i(x_j))^2 + \lambda \sum_{i=1}^{k} w_i^2$$

The valuable $\lambda$ is the regularization strength. The remaining of the term is the sum of all weights in the network squared. This term would keep the contribution of each feature small by keeping the strength small. If we have a big weight for a feature, the equation would have a smaller first term anyways.

### Dropout

Dropout can be implemented in the fully-connected layers on top of the L2-norm regularization to effectively prevent overfitting [7] It can reduce interdependent learning amongst neuron as it evaluate the combination of various neural network architectures. Figure 8 shows the changes made to a standard neural network with two fully-connected layers by dropping the some parameters as illustrated with crossed units.

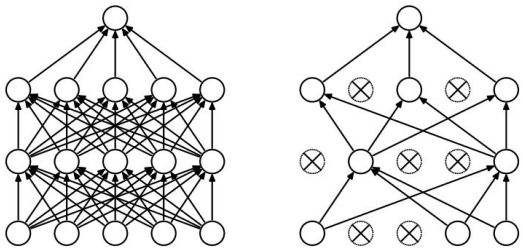

Figure 8: Standard Neural Net Model (left) Versus Dropout Neural Net Model (right) [7]

Furthermore, we also decided to add a new convolutional layer to improve accuracy. With such changes to the model, we ended up with the improved model as shown in Figure 9.

| Layer (type) | Output Shape | Param # |
|---|---|---|
| conv2d_1 (Conv2D) | (None, 33, 100, 24) | 1824 |
| max_pooling2d_1 (MaxPooling2 | (None, 17, 50, 24) | 0 |
| conv2d_2 (Conv2D) | (None, 9, 25, 36) | 21636 |
| max_pooling2d_2 (MaxPooling2 | (None, 5, 13, 36) | 0 |
| conv2d_3 (Conv2D) | (None, 3, 7, 48) | 43248 |
| max_pooling2d_3 (MaxPooling2 | (None, 2, 4, 48) | 0 |
| conv2d_4 (Conv2D) | (None, 2, 4, 64) | 27712 |
| max_pooling2d_4 (MaxPooling2 | (None, 1, 2, 64) | 0 |
| conv2d_5 (Conv2D) | (None, 1, 2, 64) | 36928 |
| max_pooling2d_5 (MaxPooling2 | (None, 1, 1, 64) | 0 |
| conv2d_6 (Conv2D) | (None, 1, 1, 128) | 73856 |
| max_pooling2d_6 (MaxPooling2 | (None, 1, 1, 128) | 0 |
| flatten_1 (Flatten) | (None, 128) | 0 |
| dense_1 (Dense) | (None, 1164) | 150156 |
| dense_2 (Dense) | (None, 100) | 116500 |
| dropout_1 (Dropout) | (None, 100) | 0 |
| dense_3 (Dense) | (None, 50) | 5050 |
| dropout_2 (Dropout) | (None, 50) | 0 |
| dense_4 (Dense) | (None, 10) | 510 |
| dropout_3 (Dropout) | (None, 10) | 0 |
| dense_5 (Dense) | (None, 1) | 11 |

Total params: 477,431
Trainable params: 477,431
Non-trainable params: 0

Figure 9: Summary of the Final Model

We have trained this new model with the same batch size and epoch number - which is 20 and 30 respectively. We can see the improved result in the Figure 10. The result shows a loss of 0.43 with this model. We can also visually observe that the trend of the loss of the testing data set is following the loss the training data set better with the improvements.

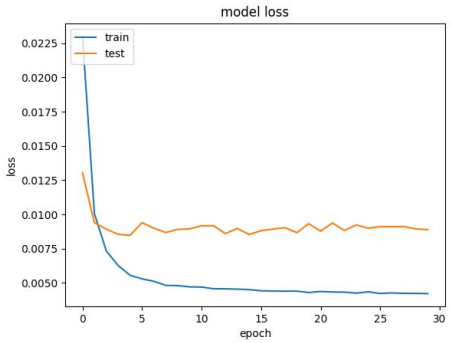


Figure 10: Model Loss of the Improved Model

## 4.2 Increasing Accuracy

With the improved model, we wanted to increase the accuracy by running more iterations. However, our local CPU is not powerful enough as it takes approximately 10 minutes to run an iteration of batch size 20. We decided to try with a cloud machine learning engine. Initially, we originally attempted to use the Google Cloud Machine Learning Engine (GCMLE); however, due to the lack of documentation, we ended up using the Amazon Web Services (AWS). We were able to set up an E2C instances with a Tesla K80 GPU [8].

Then we tested our neural network with the same testing and validation test data set. This time, we ran a batch size of 100 and epoch size of 2000. As predicted, this training result has a smaller loss of 0.05. It is a good improvement from the previous trainings as shown in Figure 11.

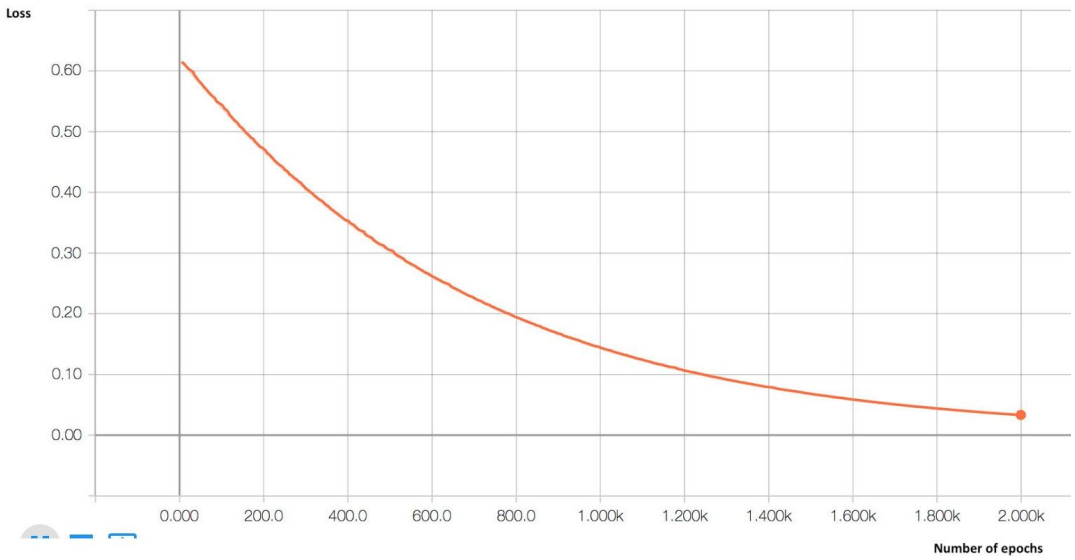


Figure 11: Model Loss for 2000 epochs

We saved the corresponding outputs in CSV format. Finally, we combined the testing images frames of 2700 images into a video and then added an image of a steering wheel to visualize the predicted steering angle along with the actual angles values, as shown in Figure 12 below.

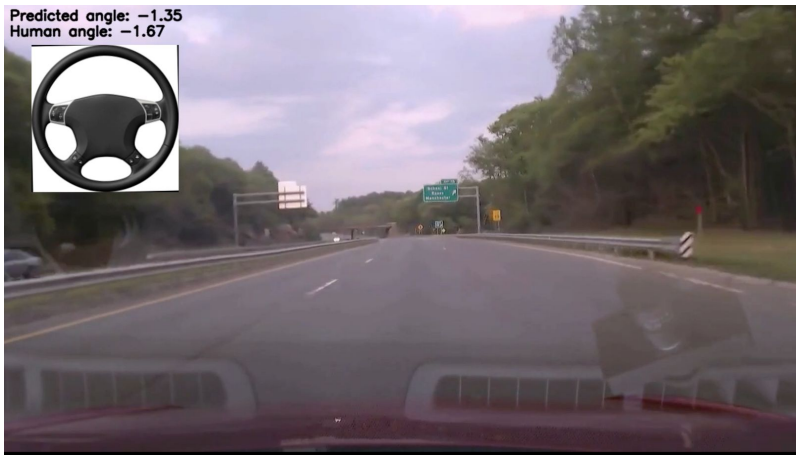


Figure 12: Screenshot of the Final Video Output

The neural network is capable of predicting the steering angle accurately most of the time. Unfortunately, the other cars on the road does affect the neural network. We assume that it may be caused by the changes in the shape of the edges of the lane.

## 5.0 Future Improvements

As the prototype neural network cannot predict the steering angle accurately with other cars on the road, especially if there is a car in the same lane as our testing car. We wish to test with more data sets and images to reduce the percentage error. Yet, this would require some changes made to our pre-processing in order to account for the larger data input. We propose to do image augmentation and to have video recorded at various driving condition for extra training data. Lastly, we would reduce frame rate of our acquired data set to reduce the train time.

### Image Augmentation

With the same set of data acquired, we could create the illusion of having extra input data when we train. For example, we can trick the neural network for turning in the opposite direction by flipping the image horizontally. If we want to use this data, we would have to alter the corresponding actual steering angle data by multiplying -1 to the value. Similarly, we can pretend to have a bigger car driving on wider roads by expanding the image horizontally. Vice versa with the shrinking the input frame horizontally, it may seem to have a smaller car driving on narrower roads. Furthermore, we can rotate or title the image to create uneven road condition. Such image augmentation should still be detectable with the pooling, which helps creating spatial invariance.

**Larger Data Set with Various Driving Condition**

In general, the neural network can have a more concrete result with a larger set of data. The ten videos provided by MIT are all recorded in dry road condition and with either sunny or mildly cloudy weather. Throughout most frames, there is no other cars directly in front or in the same lane; thus, our model does not know how to predict the steering angle when there is big obstacles.

To better prepare the neural network for daily driving condition, we should collect video footage of busy highway, night time, and light drizzle. Potentially, we will have to add more convolutional layers with max-pooling to categorize the different situations.

**Reduce Frame Rate**

The current input data set has frames per second (FPS) of 30. We have taken all frames to do the prediction. Thus, our final result video also have a FPS of 30. We hope to reduce frames since FPS of 24 is adequate to perceive images as motion according to current industry standard for a sound film [9].

Furthermore, the NVIDIA research recommended a FPS of 10 [3]. Since the neighbouring frames would have a similar steering angle prediction, a FPS of 10 is sufficient for the car to stay in approximately the center of the driving lane.

With those pre-processing changes, we can reduce files sizes for more efficient and faster computations. Thus, we would be able to train with more data in less time and would be able to acquire a more precious and accurate result.

## 6.0 Conclusion

The project began with the original NVIDIA neural network as indicated in the literature review. We have improved the model by adding one more convolutional layer, corresponding max-pooling layers, L2-norm regularization, and some dropout layers. Overall, we are satisfied with our neural network for dry road condition and when there is not a lot of cars around. The neural network is capable of predicting the correct direction and of approximating the steering angle. However, the network cannot consistently predict the accurate output as we do not have a lot of input data with a large obstacle in front of the camera.

With the further improvements - such as augmenting current input and capturing more footage, we hope to prove upon predictions and to eliminate errors due to the type of weather, time of day, and road condition. To achieve those improvements more efficiently, we would also reduce the frames per second in the way that it would still capture the important information and turning points.

## 7.0 References

[1] T. Jochem, D. Pomerleau, B. Kumar and J. Armstrong, "PANS: A Portable Navigation Platform", U.S., Sep. 25, 1995.
[2] A. Ors, "RADAR, camera, LiDAR and V2X for autonomous cars", *NXP*, 2017. [Online]. Available: https://blog.nxp.com/automotive/radar-camera-and-lidar-for-autonomous-cars. [Accessed: 02- Nov- 2017].
[3] M. Bojarski, D. Del Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. Jackel, M. Monfort, U. Muller, J. Zhang, X. Zhang, J. Zhao and K. Zieba, "End to End Learning for Self-Driving Cars", 2016.
[4] "MIT 6.S094: Deep Learning for Self-Driving Cars", *Selfdrivingcars.mit.edu*, 2017. [Online]. Available: http://selfdrivingcars.mit.edu/. [Accessed: 02- Nov- 2017].
[5] L. Fridman, “Deeptesla”, 2016, Github, Github repository, https://github.com/lexfridman/deeptesla/tree/master/epochs. [Accessed: 02- Nov- 2017]
[6] J. Masci, A. Giusti, D. Cireşan, G. Fricout and J. Schmidhuber, “A Fast Learning Algorithm for Image Segmentation with Max-Pooling Convolutional Networks”, Feb. 7, 2013
[7] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever and R. Salakhutdinov, “Dropout: A Simple Way to Prevent Neural Networks from Overfitting”, 2014
[8] Miej, “GoDeeper”, 2017, Github, Github repository, https://github.com/Miej/GoDeeper. [Accessed: 25- Nov- 2017]
[9] K. Brownlow, *Silent Films: What Was the Right Speed?*, 1980, p. 164-167.

## 8.0 Software

The development environment we used is PyCharm. We developed our code in Python using deep learning open source library Keras with Tensorflow backend. The code is divided into the 4 following Python files: load_data.py, model.py, train.py, run.py.

Load_data.py is responsible for loading training and validation images in batches and returns an array of inputs (images), and outputs (turning angles).

Model.py contains the network architecture. All the parameters regarding our network are set in that file. Train.py has two generators to return iterables for training in batches. It creates the neural network by calling the function in model.py and trains it using Keras function “fit_generator”.

Run.py takes a video file and inputs that to the train model saved in the training step. It creates a smooth transition by turning the steering wheel based on the difference of the current angle and the predicted angle. It then overlaps image of the wheel onto video frames and saves them as a video file.