Documentation

ITK Engineering GmbH | www.itk-engineering.de | info@itk-engineering.de

Autonomous Lane Following in a Simulated Environment

At the TUM Data Innovation Lab

Raju, Sukanya (03692840)

Tabari, Azadeh (03635866)

Autonomous Lane Following in a Simulated Environment

“Only those who dare to fail greatly can ever achieve greatly.”

Robert F. Kennedy

Table of Contents

List of Figures

List of Tables

1 Introduction

Many years ago, self-driving cars existed only in science fiction films and books. Nowadays, there are lots of private and governmental companies and research institutes that have invested in the production of autonomous cars and indeed, some of them like Nvidia were already successful in this area. Based on the official definition of autonomous driving given on the website of Daimler AG,

„[s]elf-driving means the autonomous driving of a vehicle to a specific target in real traffic without the intervention of a human driver." [1]

A self-driving vehicle receives the required data from different sources such as cameras or sensors installed on the car. To drive autonomously, the car should be able to receive the data from available sensors, analyze it and send the controlling commands to the engine, brake and steering system, accordingly. In other words, the tasks done by the driver such as receiving the data by eyes, analyzing the images with brain and sending the steering commands to the foots and hands should be automated. Experts have defined in total five different level of automation which differ from another basically in the contribution level of the driver. These five levels are:

1- Driver assistance
2- Partly automated driving
3- Full automation
4- Highly automated driving
5- Full automation (No driver) [2]

The main bottleneck in autonomous driving is to train the cars to recognize the objects and obstacles they should avoid. Among many external objects for a car, the yellow or white lanes which define the borderlines of the streets and highways can be used in a positive way to train the cars to drive autonomously. If the car can follow the lanes, it can be assured to a high extent that the car stays on the street. To examine this concept, the project "Autonomous lane following in a simulated environment" was established by ITK Engineering GmbH [3]. As it can be realized from the project's name, the project uses a simulated environment generated by the game engine "Unity". This car simulator was built for Udacity's Self-Driving Car Nanodegree program to clone the behaviour of the car and is freely available for everyone at GitHub repository of the project [8]. The overall idea of the project is to set up a virtual simulation environment utilizing the before mentioned simulator. Using the images collected from cameras mounted on the car and other controlling variables, a Convolutional Neural Network (CNN) is defined and trained. The complementary information provided by the simulator are steering angles, throttle, brake and speed. Based on the trained CNN, a model is generated which is aimed to be able to drive the car autonomously in the virtual simulated environment.

The whole project is structured and performed using the SCRUM agile project management [20]. The students participating in the project were obliged to participate in the introductory course on SCRUM project management organized and performed by ITK Engineering GmbH for two days.

The rest of this documentation is organized as follows. In part two, the system and software setup is explained. Then, it continues with generation of the environment to train the network. Section four provides the CNN related definitions and the training process of the neural network. In the fifth section, the implementation of an interface to communicate between the simulated environment and the model is discussed. Finally, section six will demonstrate the obtained results and suggestions for future work and completion of the project.

2 System and Software Setup

As already mentioned in the introduction part, the whole project is formulated and built using a pre-existed car simulator which was developed for Udacity's Self-Driving Car Nanodegree program. Udacity is an educational organization offering massive open online courses (MOOCs) [4]. The self-driving car simulator is developed originally to teach the concepts of deep learning and neural networks to the students. This project is available freely on the GitHub platform – world's most famous platform for software development and version control [5].

2.1 Simulator Installation

In order to set up the environment, we first need to install the followings:

- Unity game engine
- Git LFS (Optional)
- Udacity Self-Driving Car Simulator

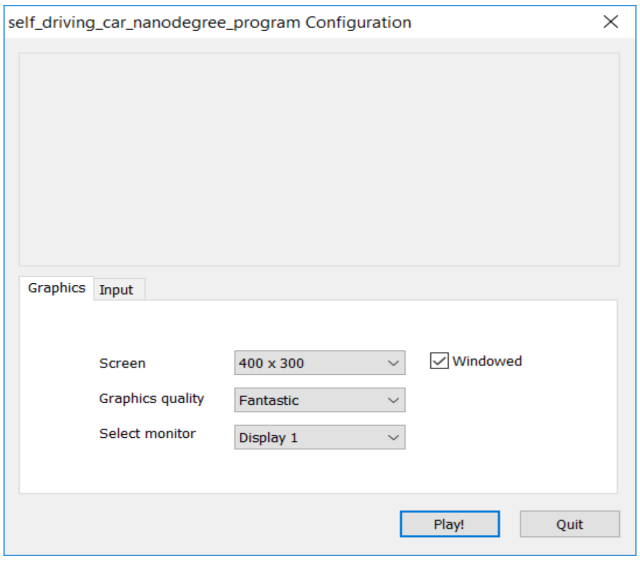
As stated above, Unity is a game engine which was used by Udacity simulator. To be able to open the simulator, first, we need to install this game engine. The installation can be done by visiting the Unity website [10]. The Personal license is enough and works well with the Udacity simulator. For the purpose of this project, version 5.5.5f is installed. It is needed to mention that by installing the Unity engine, the Visual Studio Community will be also installed automatically. This platform is free of charge and is necessary to open the scripts written in the Unity framework. Nevertheless, it was advised by mentors to use the Visual Studio Code instead of the Community version.

Downloading the Udacity Self-driving car simulator is possible in two ways: first, one can go to the GitHub repository of the project [8] and download the ZIP file. The alternative option is to use the Git Large File Storage (LFS) [9]. But to be able to use it, first, one needs to install the Git command line extension. This can be installed either by downloading from the Git website [9] or by installing the GitHub Desktop [5]. Now the system is ready to clone the project from the GitHub repository [8].

In order to use the simulator, it has to be built using the Unity engine. In this regard, one should open the Unity and open the simulator project by clicking on **file > open project** and give the address where the project is stored. When the project is loaded, one need to click on **file > Build settings**. After choosing the correct platform and pressing **Build**, Unity asks for the name and the location that the instance of the simulator should be saved. Finally, one can start the simulator by executing the .exe file saved in the defined path under the defined name.

2.2 Simulator in Details

By clicking on the generated .exe file from the previous section, the configuration screen will show up as it is displayed in Figure 1. In this page, one can select the graphics quality of the simulator as well as the input keys to steer the car.


self_driving_car_nanodegree_program Configuration

Graphics | Input

Screen	400 x 300	✓ Windowed
Graphics quality	Fantastic	
Select monitor	Display 1	

Figure 1- Configuration Page of the Udacity Simulator

By pressing the play button, the main screen of the simulator appears. This simulator is designed to perform in two different modes, naming training mode and autonomous mode as it is shown in Figure 2.

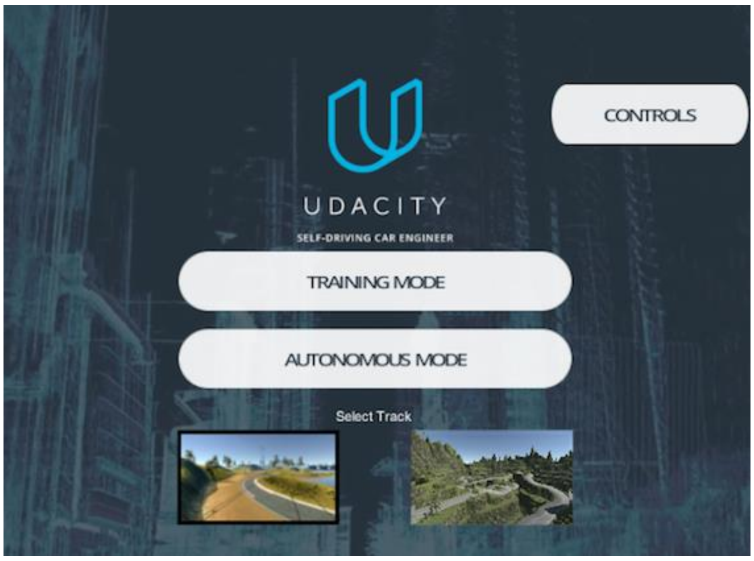


Figure 2- Udacity Simulator Main Screen

In addition to the modes, the simulator uses two tracks, the lake-side track and the jungle track. The user can choose each of the tracks that are desirable for training the car or for autonomous driving. By choosing one of the modes, a car is located at the start point of the selected track as it can be seen for the lake-side track in Figure 3.

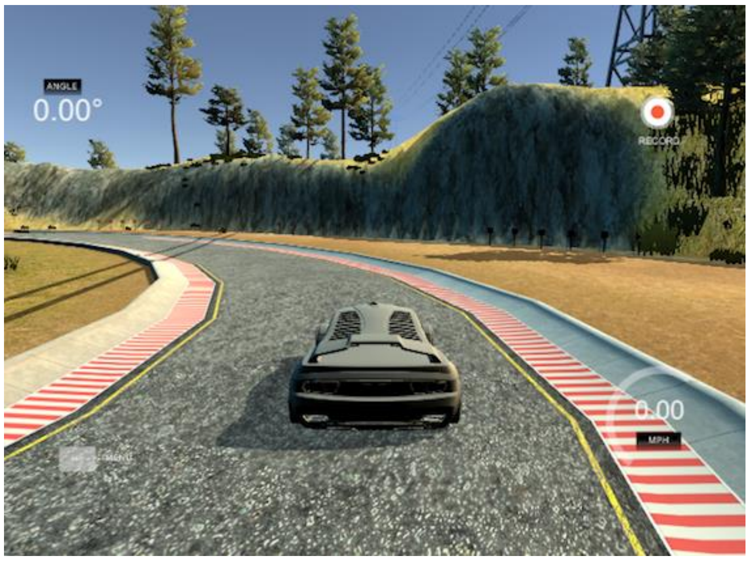


Figure 3- Starting Scene in the Training Mode for Lake-side Track

2.2.1 Training Mode

As illustrated in Figure 3, on the right corner of the screen there is a "Record" button. By pressing this button, the simulator asks for the path to save the recorded information. This information will be used later for training the neural networks in section three. After choosing the desired path, the car is ready to drive using the defined keys from the configuration screen. These keys are always accessible by selecting the controls button from the main screen. To pause or stop the recording, the record button has to be pressed again. At this point, the simulator starts to store the data in the selected directory. The data is stored in two formats. First, the images received from three cameras mounted on the car are saved in a folder called IMG. The images are labelled by the camera location and the epoch that the image is recorded: [camera_location_year_month_date_hour_minute_second_microsecond] [7]. As an example, the image obtained from the central camera at one exact epoch is illustrated in Figure 4.

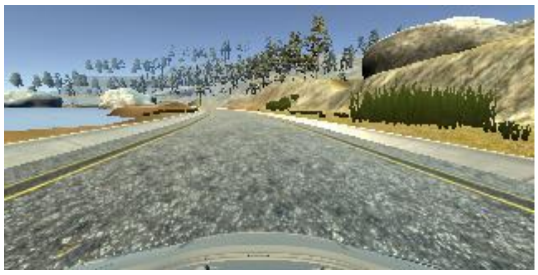


Figure 4- Image Naming Sample (center_2018_01_18_11_17_53_634)

The second type of data is saved in a CSV format and it is called "driving_log". In this file, the location of the saved images from all three cameras is given as well as the steering angle, throttle and speed of the car in that particular epoch. Also, information on the existence of a braking event is provided [8]. A sample of this CSV file for four consecutive epochs is shown in Table 1.

Table 1- CSV File Containing the Steering Information of the Car

Center camera	Left camera	Right camera	Steering Angle	Throttle	Brake	Speed
C:\...\IMG\center_2018_01_18_11_17_53_634.jpg	C:\...\IMG\left_2018_01_18_11_17_53_634.jpg	C:\...\IMG\right_2018_01_18_11_17_53_634.jpg	0	0	0	0.491714
C:\...\IMG\center_2018_01_18_11_17_53_750.jpg	C:\...\IMG\left_2018_01_18_11_17_53_750.jpg	C:\...\IMG\right_2018_01_18_11_17_53_750.jpg	0	0	0	0.48678
C:\...\IMG\center_2018_01_18_11_17_53_870.jpg	C:\...\IMG\left_2018_01_18_11_17_53_870.jpg	C:\...\IMG\right_2018_01_18_11_17_53_870.jpg	0	0	0	0.479955
C:\...\IMG\center_2018_01_18_11_17_53_985.jpg	C:\...\IMG\left_2018_01_18_11_17_53_985.jpg	C:\...\IMG\right_2018_01_18_11_17_53_985.jpg	0	0	0	0.475139

2.2.2 Autonomous Mode

In this mode, the simulator uses the trained model to steer the car on the selected track autonomously [6]. This concept will be addressed in more details in later in section four where we explain our neural networks and section six, the project result.

2.3 Programming Platform

2.3.1 Programming Languages

The whole process of programming and training the neural networks were performed with Python programming language. In this regard, Anaconda package is used. Anaconda is an open source distribution of Python which is suitable for large-scale data processing [4] Anaconda is simply a very useful set of Python packages and a package manager called conda. This package can be installed from the Anaconda website. For the purpose of this project, the conda version of 4.4.8 is chosen – is the current version at the date of this documentation. In addition, the programming language utilized to generate the roads and other assets of the simulator is C#.NET which is one of the main programming platforms for Unity. In this regard, a predefined Unity framework is used.

2.3.2 TensorFlow

Tensorflow is an open-source Machine Learning framework used for numerical computations. Its state of the art design is suitable to deploy the computational effort on one or more Central Processing Unit (CPUs) or Graphics Processing Unit (GPUs). Tensorflow was born within Google's Machine Intelligence research organization for the purpose of machine learning and deep neural networks research [11] and is up to date, one of the most used frameworks in Machine Learning. In this project, the Tensorflow version 1.4.0 with GPU support is installed. The GPU capability helps to run the training process significantly faster [11].

2.3.3 OpenDrive

Every car needs a road to drive on. The car used in this project is not exceptional. To be able to generate the underneath road and its surrounding environment, the open file format specification from openDRIVE® version 1.4 is applied [12]. In an openDRIVE® file, the geometry of roads is described by Extensible Markup Language (XML) syntax. For more details, one can refer to the specification file available on the website of openDRIVE®. The generated environment is saved in the .xodr format [12]. This format is fed to the Unity game engine and translated to a 3D model of the environment.

2.3.4 Tools Interaction

Considering all presented tools above, the following Figure shows the work logic of the project and interaction of the tools in one frame.

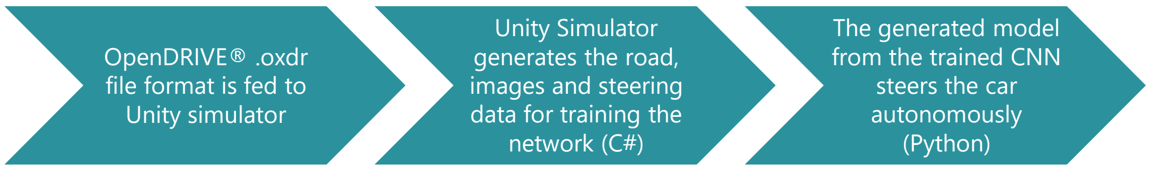

OpenDRIVE® .oxdr file format is fed to Unity simulator
Unity Simulator generates the road, images and steering data for training the network (C#)
The generated model from the trained CNN steers the car autonomously (Python)

Figure 5- Interaction of the Tools

As shown above, the environment required for driving the car is generated based on the specifications of openDRIVE® and fed to the Unity simulator. Unity simulator uses this environment to generate the required steering data to train the model. Later the model steers the car autonomously in the generated environment.

3 Road Generation Process

As stated in section 2.3.3, the generation of the road and the surrounding environment was performed using the specifications of the openDRIVE® in XML format. This process consists of the following steps as it is shown in Figure 6:

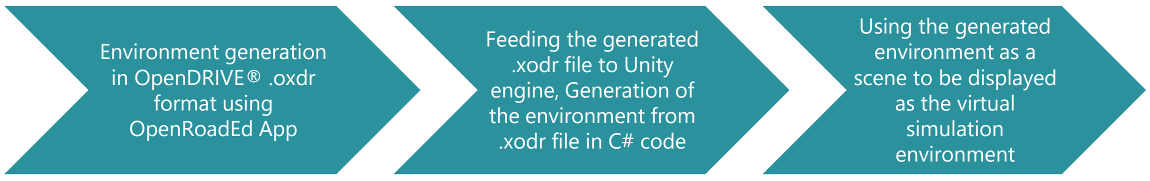
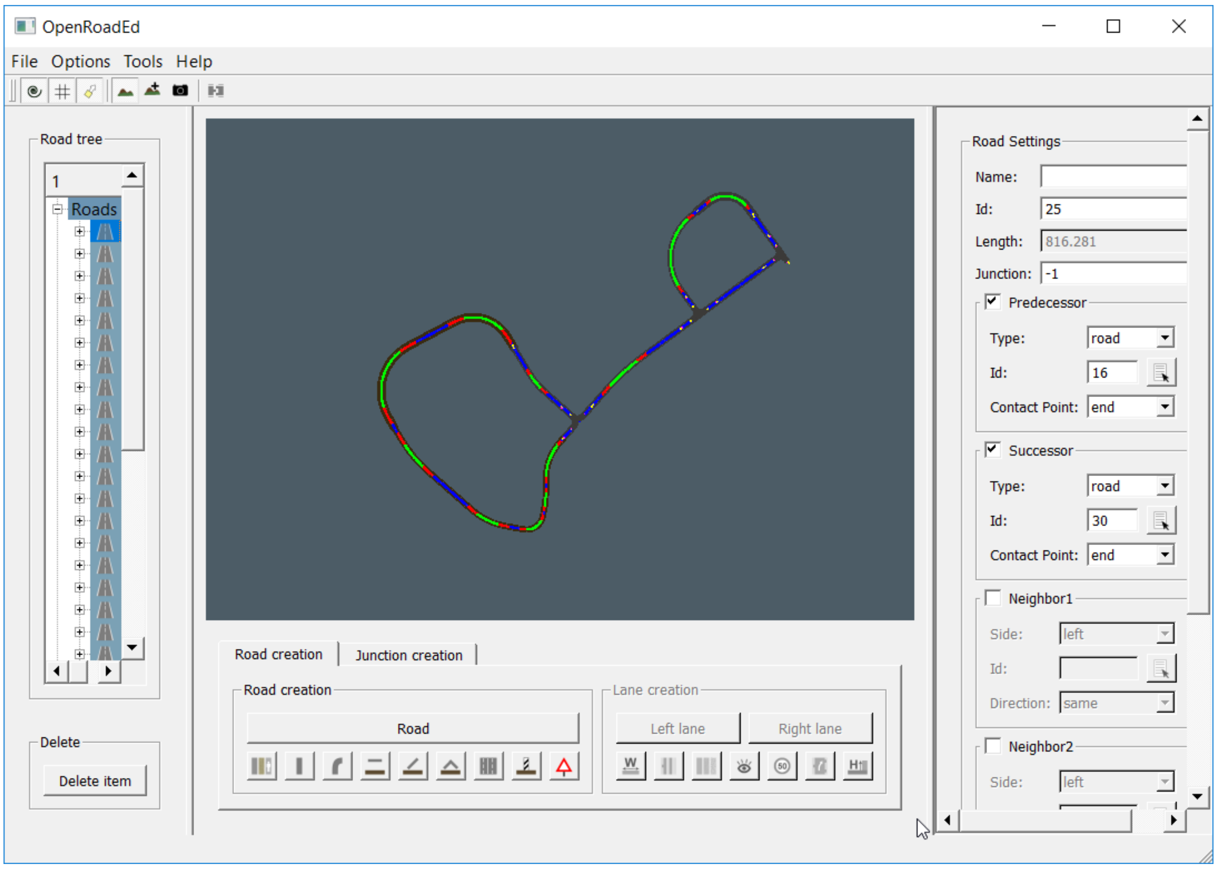

Environment generation in OpenDRIVE® .oxdr format using OpenRoadEd App
Feeding the generated .xodr file to Unity engine, Generation of the environment from .xodr file in C# code
Using the generated environment as a scene to be displayed as the virtual simulation environment

Figure 6- Environment Generation Process

The generation of the environment in .xodr is done using a free application called OpenRoadEd [13]. As can be seen in Figure 7, different parts of the roads consisting of straight roads, arcs and spirals are generated using the road settings panel. Later, the roads are connected to each other to form the whole track. In this part, the definition of the environment can be also done using the available textures or by adding the desired textures to the library of images. By saving the geometry, the generated environment is transformed automatically to .xodr format.



Road Settings

Name:
Id: 25
Length: 816.281
Junction: -1

[✓] Predecessor
Type: road
Id: 16
Contact Point: end

[✓] Successor
Type: road
Id: 30
Contact Point: end

[ ] Neighbor1
Side: left
Id:
Direction: same

[ ] Neighbor2
Side: left

Figure 7- OpenRoadEd Interface

In the second step, the generated .xodr files are converted to roads, arcs and spirals in the Unity engine. This is done using the C# script called RoadGenerator.cs, which was written in the Unity specifications format. In the code, the .xodr file is read line by line and each XML tag is converted into its appropriate data with its x coordinate value, y coordinate value, heading and road length. A new environment is subsequently built utilizing these extracted features, and shown on the GUI. From this environment, the road part is mainly considered for the training of the CNN. This environment is saved as a scene in the Unity engine and appears as the virtual simulation environment to collect the ground truth data for CNN training and later to test the generated model in the autonomous mode. Figures 8 and 9 provide two samples of the generated roads in Unity.

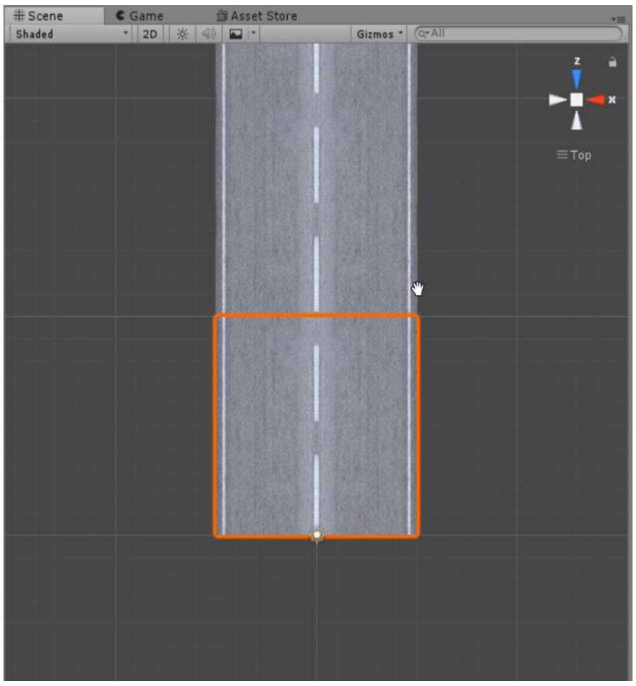


Figure 8- Straight road generated in Unity

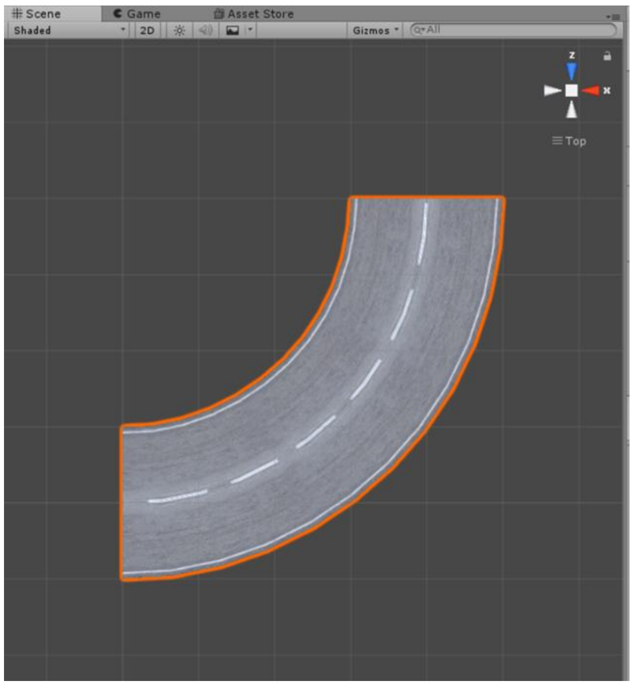


Figure 9- Curved road generated in Unity

4 Convolutional Neural Networks

4.1 Introduction

The idea of Machine Learning is to provide computers with the ability to build a model from data observations (image, audio, etc.), that can make decisions and predictions based on similar data. We will now introduce some machine learning terminologies, which we will use in the project.

4.1.1 Supervised Learning

Given a space X that yields a representation of the image space containing the images from the unity framework. We consider a finite subset of training examples $X_{Train} = \{x^{(1)}, \ldots, x^{(n)}\} \subseteq X$. Let further $y^{(i)}$ be the corresponding ground truth data with respect to $x^{(i)}$, e.g., the steering angle, brake and throttle. For supervised learning, the model M_w is trained with the training set $T = \{(x^{(i)}, y^{(i)}) \mid i = 1, \ldots, n\}$ which includes the training examples and the corresponding ground truth data.

4.1.2 Artificial Neural Networks

Artificial Neural Networks (ANNs) are a group of models in Machine Learning, which are inspired by the structure and functions of biological neural networks [17]. The central neural system in the mammals' brain contains a huge number of neurons, which are strongly interconnected. With the help of this structure, information is processed and evaluated across neurons by electrochemical interactions. The ANNs are then an attempt at implementing a similar way of information processing on computers.

We illustrate below the internal structure of an artificial neuron, which is the constitutive unit of ANNs. The artificial neuron receives one or more inputs x_i. All inputs are then multiplied by corresponding weights w_i and summed up together with a threshold term θ known as bias. After that, this input is passed through a non-linear function φ known as an activation function to produce a certain output y. We summarize the whole operation applied by an artificial neuron in the following formula:

$$y = \varphi\left(\sum x_i \cdot w_i + \theta\right)$$

We can represent the ANN as a model M_w, that is composed of learnable weights w, and activation functions φ.

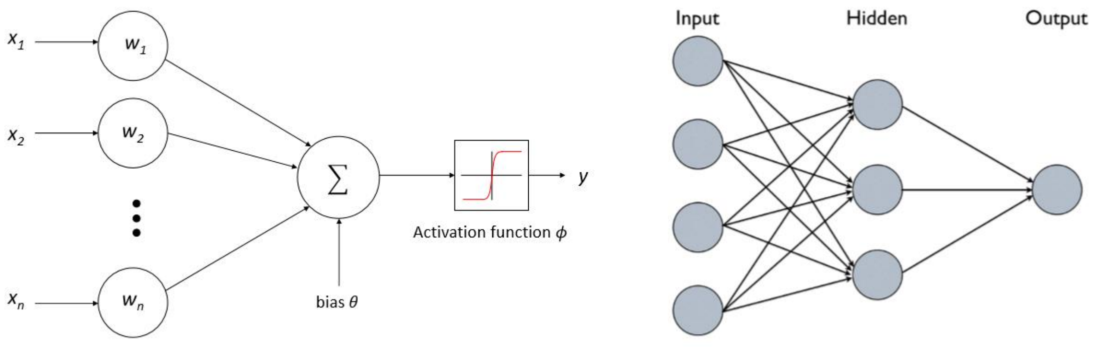


Figure 10- Artificial Neural Networks

The above Figure 10 shows how the architecture of this model is constructed based on neurons interconnected with each other. With the correct parametrization, M_w can approximate a certain function G, to realize a specific task (e.g. emulating a Turing Machine [18]). These networks grow big quickly, using a great number of parameters and thus are hard to train. This represents the major restriction of their usefulness.

The **size of the neural network** is calculated by the sum of total connections and number of biases in all the layers of the network.

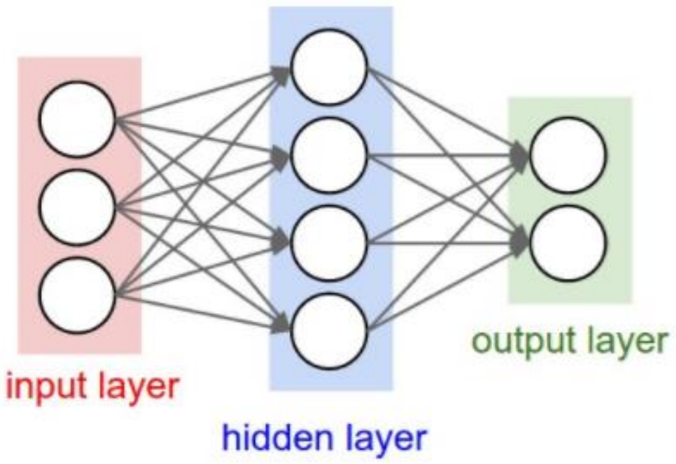


Figure 11- Neural Network

The above network, shown in Figure 11, has [3 x 4] + [4 x 2] = 20 weights and 4 + 2 = 6 biases for a total of 26 learnable parameters. The number of parameters can be up to 100 million, but the parameter size should be appropriate to the size of the network. The more parameters used by the CNN, the more time and memory it needs. Using fewer parameters, the greater the chances are for the network being insufficient to yield a good model to the data.

4.1.3 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) [19] belong to the approaches known as deep learning approaches and can be applied to various problems of computer vision and digital image processing. These networks were re-interpreted in 1998 for recognizing digits in a document recognition application.

Similarly, to the classical neural networks, CNNs are composed of neurons arranged in a specific structure and holding learnable weights and biases. However, CNNs have the following characteristics:

• **3D volumes of neurons**: Neurons are arranged in CNNs in a 3D structure (width, height and depth), which makes this type of networks practical for image processing. While the internal features of classical neural networks are computed in hidden layers, CNNs compute 2D feature maps stacked together in a 3D structure which is also known as feature hierarchy. As in Fgure 8 (a) neurons inside a layer are only connected to a small region of the previous layer, which is known as receptive field.

• **Local connectivity**: Since we work with a high dimensional input such as images, it is computationally very expensive to connect all neurons from one layer to all neurons from the previous layer. CNNs exploit the fact that the input (image) is spatially correlated and enforce a local connectivity pattern between neurons. The connections are local in space (along width and height) but always extend along the depth, Figure 12. This architecture ensures that the learned network weights produce a strong response to a spatially local input pattern.

• **Shared weights**: As known from the convolution operation, a convolutional filter applies on a certain input in a sliding window approach. Building on this property, each filter in the CNN is replicated across the entire layer input, which implies that the weights of the neurons are shared. This highly reduces the number of parameters and hence increases the learning efficiency, Figure 12 (c).

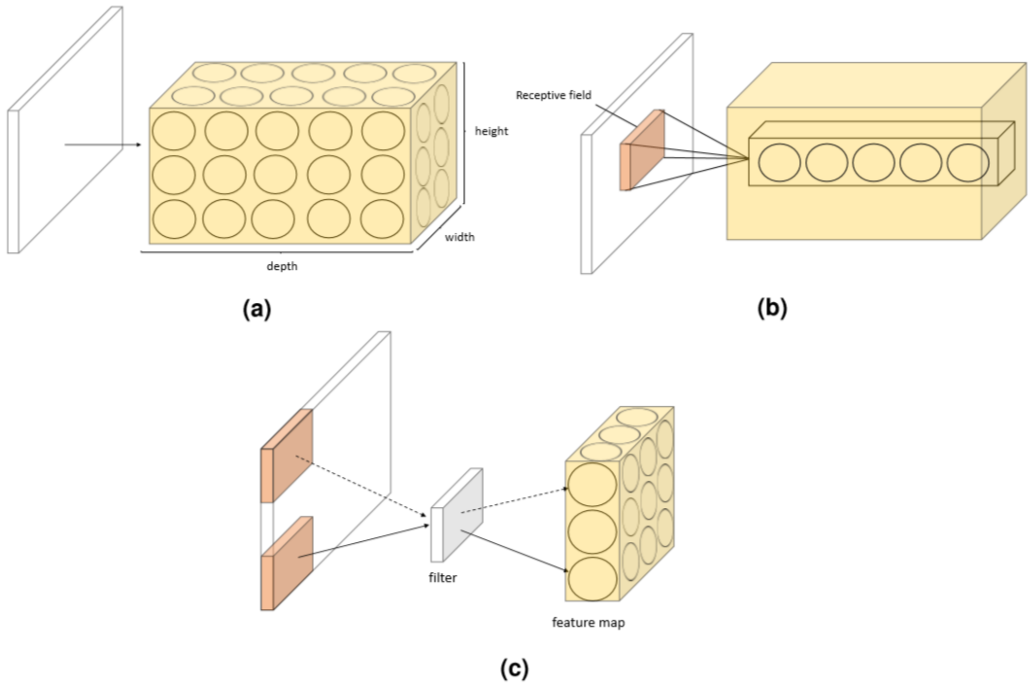


Figure 12- Convolutional Neural Network

A CNN can be represented by the model M_w, which is characterized by the weights parameter w. It consists of n_L layers, where each layer can be represented by the function f. More precisely, M_w can be formulated as the sequence of functions f_i with i ∈ {1, ..., nL}.

$$M_w = f_{nL} \circ \ldots \circ f_i \circ \ldots \circ f_1$$

Let x_i be the output of the i-th layer. Each x_i is computed from the previous output $x_i - 1$ by applying the function f_i with weights parameter w_i:

$$x_i = f_i\,(wi_;\, x_i - 1),\ \forall\ 1 \leq i \leq n_L$$

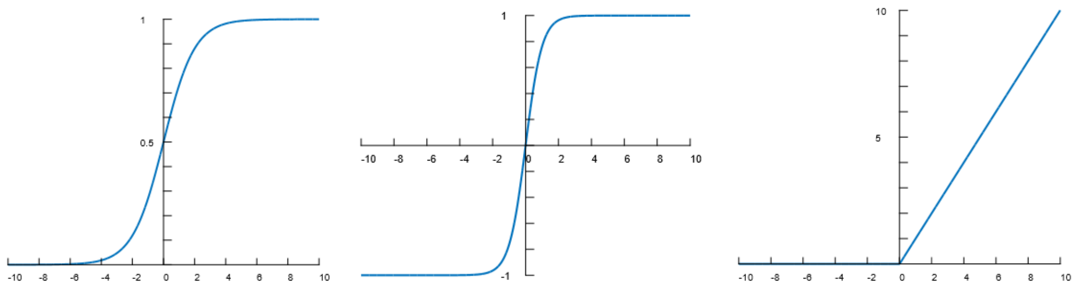
We can see the inferred data throughout the network has a 3D structure. Therefore, the layer outputs are of the form $x_i \in R^{Wi \times Hi \times Di}$, with $(Wi \times H_i)$ representing the spatial dimensions and D_i representing the depth. Each spatial extent of x_i is called feature map in the context of CNNs.

4.1.4 Layers of Convolutional Neural Networks

A CNN [15] architecture is formed by several distinct layers that transform an input volume x to an output volume y. The most commonly used layer types are:

• **Convolution** layer is parametrized by a learnable filter bank and the biases. Each filter is spatially small with a size typically between 3 and 7. By applying the convolution, we slide each filter spatially across the input volume (in height and width directions), and produce a 2-dimensional feature map. Intuitively, the network learns filters that activate when they see a specific type of features at some spatial position in the input. For a given input volume the convolution implies the computation of the dot product between the entries of the input and the sliding filter entries.

• **Activation Functions** layers are commonly set after a convolution layer of the network. They are needed to make the input more separable and to increase the computational power while keeping the input volume dimensions unchanged. Below, we illustrate the non-linear functions that are commonly used as activation functions in CNNs.



(a) Sigmoid **(b)** Tanh **(c)** Rectified linear units

Figure 13- Activation Functions

a) Sigmoid: uses the mathematical formula:

$$\sigma(x)=1/(1+e^{-x})$$

It converts the input value to [0,1]. 1 is when the data is sent with maximum frequency to next neuron, and 0 is when it is not sent at all.

b) Tanh: This function is similar to the Sigmoid function, but it converts the input to the range [-1,1], there is a possibility of getting a saturation level from this activation function. This function gives zero centred and is not always positive.

$$f(x) = \tanh(x) = 2 / (1+e^{-2x}) - 1$$

The Tanh in terms of Sigmoid function:

$$f(x) = \tanh(x) = 2\sigma (2x) - 1$$

c) ReLu: operator applies a thresholding at zero. f (x) = max (0, x). The advantage of the ReLU operator over sigmoid and tanh is the convergence acceleration of the learning algorithm to the optimal parameters. In practice, sigmoid and tanh can be computationally expensive (due to exponentials, etc), whereas the ReLU only applies a thresholding of input volume at zero. The disadvantage of ReLU units is that their effect can stop in some cases during learning. For instance, if a large update flows through the ReLU unit, the weights can be changed in such a way that the activation is always zero and from that point the corresponding neuron is dead.

• **Pooling** layers are usually inserted between convolution layers, reduces the spatial size (W × H) of the feature maps produced by the convolutional layers. The most common form of pooling uses a filter with spatial extent (2×2), this filter is applied with a moving step of 2 for each filter. This operation down-samples every depth slice in the input by 2 along width and height of images. However, the depth dimension remains unchanged. W and H.

The advantage of the pooling layer is to increase translation and distortion invariance of the feature maps i.e the activation for a feature can have the same result at different positions in the input. In the early years of the CNNs, the widely used method for pooling was the average pooling, which applies the average of the feature response in a widow of predefined size. We use max pooling in our project that forwards the maximal value of each window.

• **Merge** Layer is used when multiple inputs need to be fed into the network. For each input, a layer is created and each of these layers is merged using the Merge layer. The Merge layer is also used to input an intermediate or auxiliary output of any of the previous layers.

4.1.5 Training of Convolutional Neural Networks

To train a CNN we need to calculate the prediction error between the predicted data $\hat{y} \in L^{H\times W}$ and the ground truth labels $y \in L^{H\times W}$. Therefore, a loss function is added at the end of the network, to provide this error

• **The mean squared error loss** gives the estimate of the average error squares, that is, the difference between the estimator and what is estimated. This is always non-negative, values closer to zero show a good model. The basic principle of this loss is to minimize the quadratic sum. The standard form of MSE loss function is defined as

$$L = 1/n \sum (y_{(i)} - \hat{y}_{(i)})^2$$

where $(y_{(i)} - \hat{y}_{(i)})$ $(y_{(i)} - \hat{y}_{(i)})$ is named as residual, and the target of MSE loss function is to minimize the residual sum of squares.

• **Cross-Entropy** is commonly-used in binary classification (labels are assumed to take values 0 or 1) as a loss function (For multi-classification, use Multi-class Cross Entropy), which is computed by

$$L=-1/n \sum [\, y_{(i)} \log (\hat{y}_{(i)}) + (1 - y_{(i)}) \log (1 - \hat{y}_{(i)})]$$

Cross-entropy measures the divergence between two probability distribution, if the cross-entropy is large, which means that the difference between two distributions is large, while if the cross-entropy is small, which means that two distributions are like each other.

4.1.6 Regression

Regression [16] is a mapping of one set of independent continuous input to another set of continuous outputs. The inputs are features passed forward from the network's previous layer. Many inputs will be fed into each node of the last hidden layer, and each input will be multiplied by a corresponding weight, w.

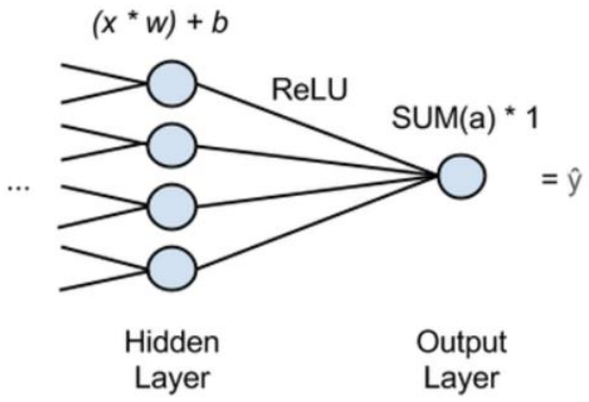


Figure 14- Regression Network

The sum of those products is added to a bias and fed into an activation function. (eg. ReLU)

For each hidden node, the activation function outputs an activation, a, and the activations are summed going into the output node, which simply passes the activations' sum through.

A neural network performing regression will have one output node, and that node will just multiply the sum of the previous layer's activations by 1. The result will be the results ŷ, the network's estimate, the dependent variable that all the Xs are mapped to.

To perform backpropagation and make the network learn, ŷ is compared to the ground-truth value of y and the weights and biases of the network are adjusted until the error is minimized, this is similar to a classifier.

In this way, the neural network can be used to get the function relating an arbitrary number of independent variables x to an arbitrary number of dependent variables y that can be predicted.

4.1.7 Data Pre-processing

The images can be pre-processed in many ways before being fed into the network. The image can be replicated with various changes. Example shifted horizontally/vertically, vertical/horizontal flip, rotation, centre, normalize, zoom, rescale. A user-defined pre-processor function can also be used. All the above can be done using the ImageDataGenerator from Keras [14]. The images produced here can also be stored in a file location to be analysed before feeding into a CNN.

The data needs to be normalized before being fed into the network. Normalization of data can be done for images and the data used as ground truth. Normalization of data is converting all the data to the same range, so when all the parameters are in the same range, the same activation function can be used in one layer over all the parameters. This ensures that all the parameters have the same influence on the model.

4.2 Implementation of Convolution Neural Network

For implementing the CNN, we implemented a few basic projects to understand the different features of a convolution neural network. Some of them are:

1. To classify a 'tree' image from a 'non-tree' image. The purpose of this classification is to set up a simple neural network and understand the various layers used for classification. The MNIST data classification was used as a reference for this classification.
2. The above classification was then extended to use images produced by unity, of size 160*320*3. The ground truth for this data set was 1 if there was a road in the image and 0 if there was no road in the image in front of the car.
3. To train a CNN model to steer the car in autonomous mode. The model read the images, 'Speed' as input and to predicts the 'Steering Angle', 'Throttle', 'Brake'. All the above data is sent from unity to the CNN.

4.2.1 Classification

To start with, we trained the model to classify images received from unity.

4.2.2 Data Generated from Unity

The two classes of classification are 'road' and 'no road', 'road' class is when the car is on the road and the lane is in front of the car. The 'no road' class is when the car is not on road, but on the side path, so there is grass or sand in front of the car.

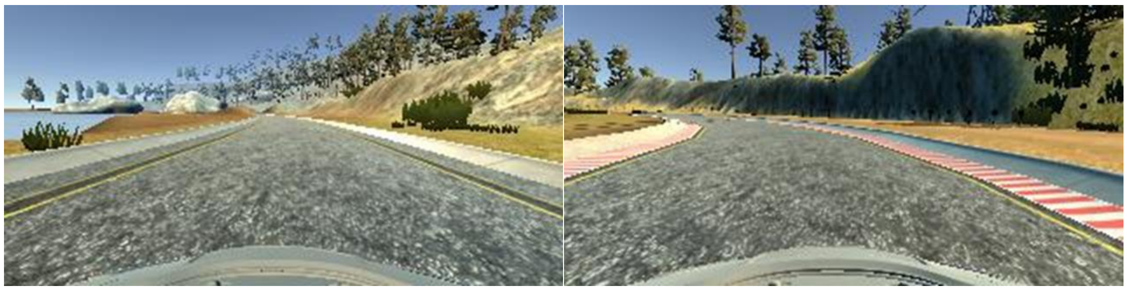


Figure 15- Class 'road'

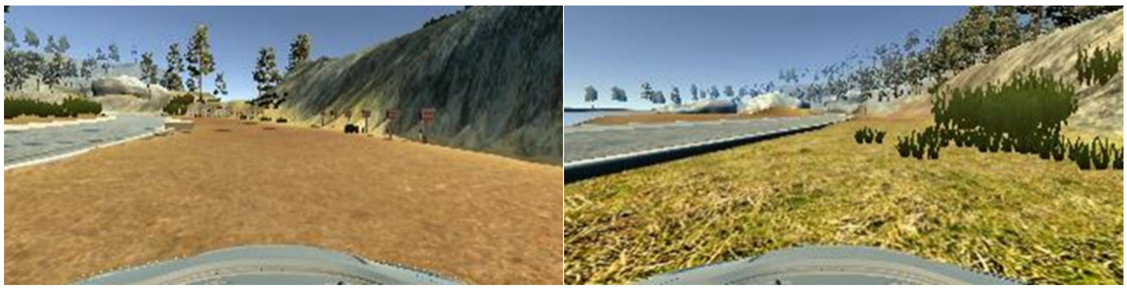


Figure 16- Class 'no road'

4.2.2.1 CNN Details

- Size of Images used for training: 160 * 320 * 3
- The training set consists of 750 images and its respective ground truth
- The testing set consists of 150 images and its respective ground truth
- Images are being read from a location in file browser using ImageDataGenerator from Keras
- Loss used for this model: categorical_crossentropy
- Optimizer used for this model: SGD
- Batch size to train the model: 5
- Number of epoch to train the model: 20

4.2.2.2 Convolution Layers Details

Table 2- Convolution Layers Summary

Layer (type)	Output Shape	Param #	Activation Function
conv2d_3 (Conv2D)	(None, 149, 309, 32)	13856	'relu'
max_pooling2d_3	(None, 37, 77, 32)	0	
conv2d_4 (Conv2D)	(None, 35, 75, 32)	9248	'relu'
max_pooling2d_4	(None, 8, 18, 32)	0	
flatten_2 (Flatten)	(None, 4608)	0	
dropout_3 (Dropout)	(None, 4608)	0	
dense_3 (Dense)	(None, 64)	294976	'relu'
dropout_4 (Dropout)	(None, 64)	0	
dense_4 (Dense)	(None, 2)	130	'softmax'
Total Parameters: 318,210			
Trainable Parameters: 318,210			
Non-trainable Parameters: 0			

4.2.2.3 Network Results

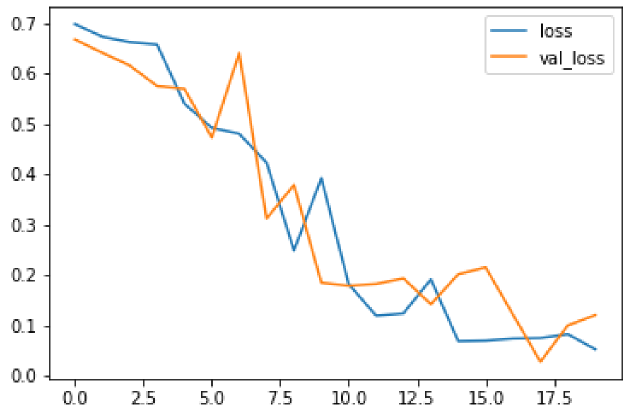


Figure 17- Network Loss

Table 3- Result Metrics

Metrics	1st Epoch	Last Epoch
Loss	0.698	0.0522
Accuracy	0.575	0.9938
Validation loss	0.6675	0.12
Validation accuracy	0.5312	0.9688

4.2.3 Training the Model to Drive the Car Using Regression

4.2.3.1 Data Generated from Unity

From Unity, we receive the image and speed which is fed to CNN as input, and the model outputs shall be steering angle, brake and throttle.

All the data used in the CNN needs to be manipulated such that the most important information is enhanced, this will help the CNN learn better.

The images received from Unity are of the size 160*320*3. The image contains trees and mountains in the background, all this data is not important to drive the car and will confuse the model during training. The image received from Unity:

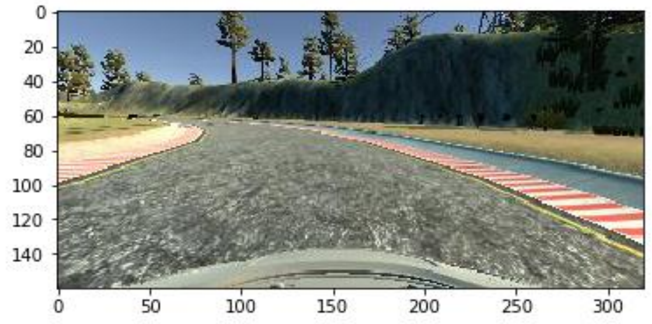


Figure 18- Image from Unity (160*320*3)

4.2.3.2 Data-preprocessing

The trees, mountains and the front part of the car are not required by model and is cropped. Image after cropping:

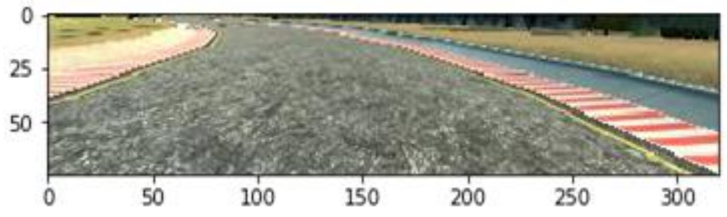


Figure 19- Cropped Image (75*320*3)

To take into consideration all the scenarios possible, the images are flipped horizontally. This helps the model learn for an unbiased data set of steering angles.

The images are then normalized. The images can be normalized about the 0 axes and in each of the R, G and B matrix. The easiest way to do this is to divide each value of matrix by 225. Keras also provides a way to do this using the 'samplewise_std_normalization=False', of the ImageDataGenerator function.

Below is the range of all the parameters used in the model:

- Steering Angle: [-1, 1] (negative values for left steering angle and positive values for right steering angle)
- Throttle/Speed: [0, 1]
- Brake: [0, 1]
- Speed: [0, 31]

Below is the sample of steering angle, throttle, brake and speed:

Table 4- Data Sample

Steering angle	Throttle	Brake	Speed
0	0	0.4606265	13.05383
0	0	0.3536329	11.81833
0	0	0.1376654	10.90094
-0.1	0	0	9.877307
-0.25	0	0	9.801888
-0.45	0	0	9.677054
-0.3205477	0	0	9.583974
-0.1090438	0	0	9.527431
-0.2079084	0	0	9.447891
-0.3579084	0	0	9.375011
-0.3219513	0	0	9.279725
-0.113026	0	0	9.225466
0	0.05866113	0	9.15232
0	0.2672702	0	9.211304
0	0.4814161	0	9.511002
0	0.5307518	0	9.850943
0	0.2689583	0	10.13124

As the speed has a large range, it will affect the model more than any other parameter. To normalize this value, the mean and standard deviation of each of the samples of the ground truth is calculated, and then (sample data - mean) / standard deviation formula is used to normalize the data. The steering angle is also normalized. After normalization of the data, all the data is in the range [0, 1]. The same normalization is also done for the real-time data received from Unity during the autonomous mode. This data is then scaled back to original level before being sent back to unity to drive the car.

4.2.3.3 CNN Details

- Size of Images used for training: 75 * 320 * 3
- The training set consists of 14252 images and its respective ground truth.
- The validation set consists of 1583 images and its respective ground truth.
- The testing set consists of 1759 images and its respective ground truth.
- Loss used for this model: mean_squared_error
- Metrics used for this model: mean absolute error
- Optimizer used for this model: Adam
- Batch size to train the model: 32
- Number of epoch to train the model: 100

4.2.3.4 Convolution Layers Details

To feed the image and speed as input, first the image is fed in the layer 1 of the network, then after convolution, max-pooling and flattening layers, a Merge layer is used to input the speed. As the model is no longer Sequential, the functional API layers of keras are used.

To train the model, each of the output parameters is considered independent of the other, by considering three arrays as output. And an external weight is applied to each of the output parameters. This weight is added to the loss function and the parameter with more weight is given more importance. For this model, the external weight of 2 for Steering angle, 5 for Throttle, and 1 for Brake was added.

Table 5- Convolution Layers Summary

Layer (type)	Output Shape	Parameter #	Connected to
main_input (InputLayer)	(None, 75, 320, 3)	0	
conv2d_185 (Conv2D)	(None, 75, 320, 12)	336	main_input[0][0]
max_pooling2d_185 (MaxPooling2D)	(None, 37, 160, 12)	0	conv2d_185[0][0]
conv2d_186 (Conv2D)	(None, 35, 158, 24)	2616	max_pooling2d_185[0][0]
max_pooling2d_186 (MaxPooling2D)	(None, 17, 79, 24)	0	conv2d_186[0][0]
conv2d_187 (Conv2D)	(None, 75, 320, 12)	6944	max_pooling2d_186
max_pooling2d_187 (MaxPooling2D)	(None, 8, 39, 32)	0	conv2d_187[0][0]
conv2d_188 (Conv2D)	(None, 4, 35, 64)	512645	max_pooling2d_187[0][0]
max_pooling2d_188 (MaxPooling2D)	(None, 2, 17, 64)	0	conv2d_188[0][0]
flatten_64 (Flatten)	(None, 2176	0	max_pooling2d_188[0][0]
speed_input (InputLayer)	(None, 1)	0	
concatenate_44 (Concatenate)	(None, 2177)	0	speed_input[0][0], flatten_64[0][0]
dropout_127 (Dropout)	(None, 2177)	0	concatenate_44[0][0]
dense_150 (Dense)	(None, 64)	139392	dropout_128[0][0]
dropout_128 (Dropout)	(None, 64)	0	dense_150[0][0]
dense_151 (Dense)	(None, 1)	65	dropout_128[0][0]
dense_152 (Dense)	(None, 1)	65	dropout_128[0][0]
dense_153 (Dense)	(None, 1)	65	dropout_128[0][0]
Total Parameters: 200,747			
Trainable Parameters: 200,747			
Non-trainable Parameters: 0			

4.2.3.5 Network Results

The network gives the following graphs:

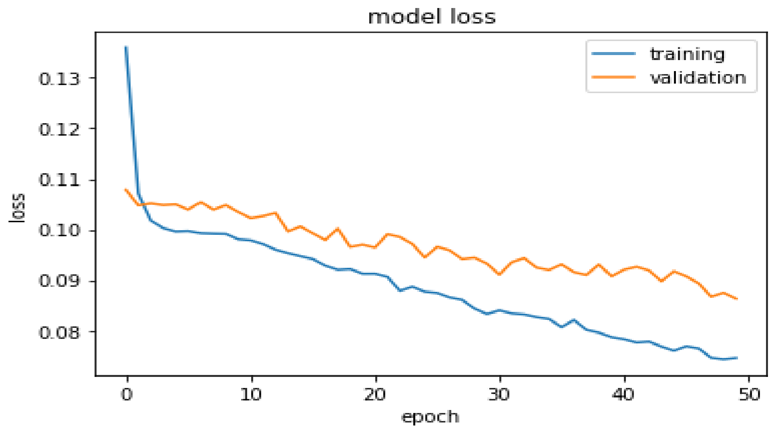


Figure 20- Loss of Model

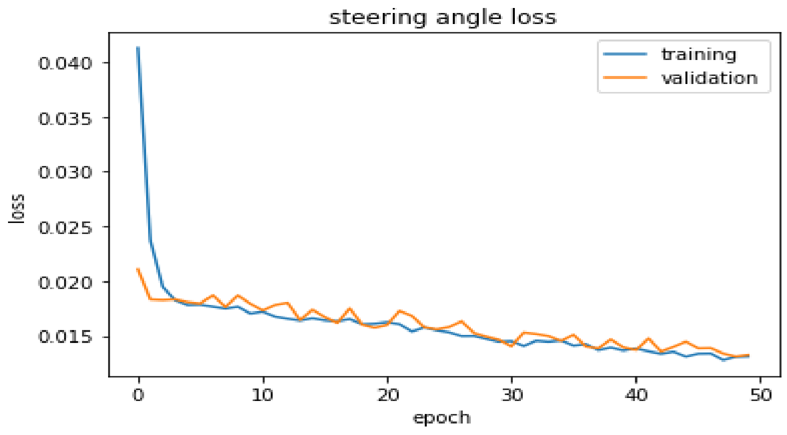


Figure 21- Loss of Steering Angle

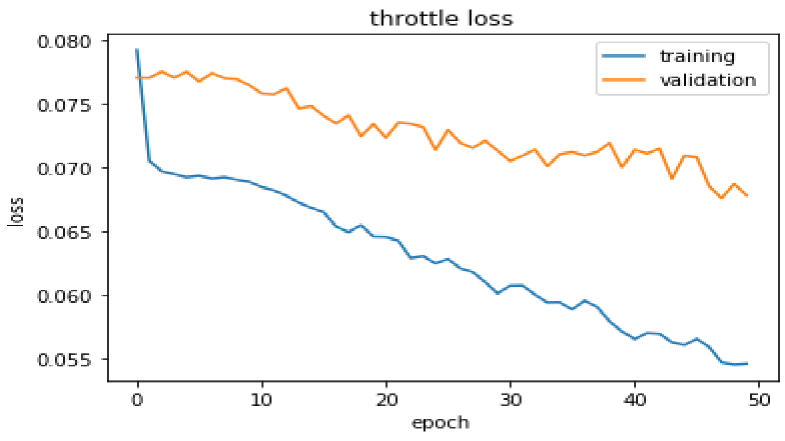


Figure 22- Loss of Throttle

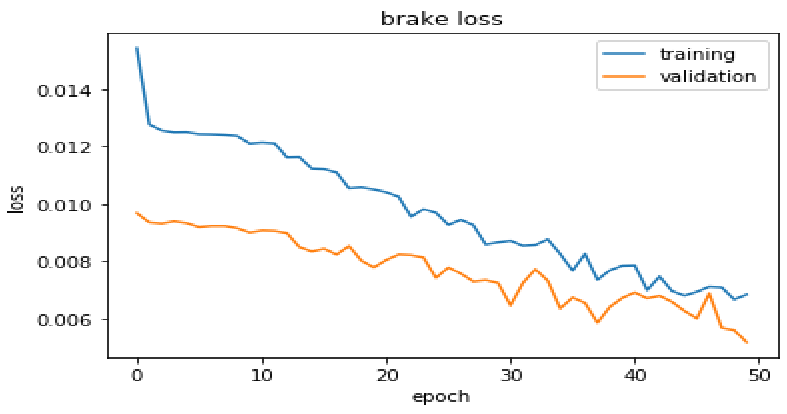


Figure 23- Loss of Brake

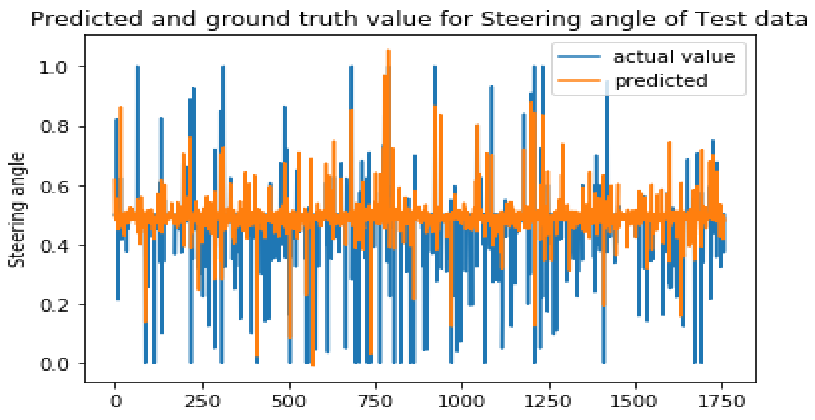


Figure 24- Steering Angle, Predicted vs. Actual

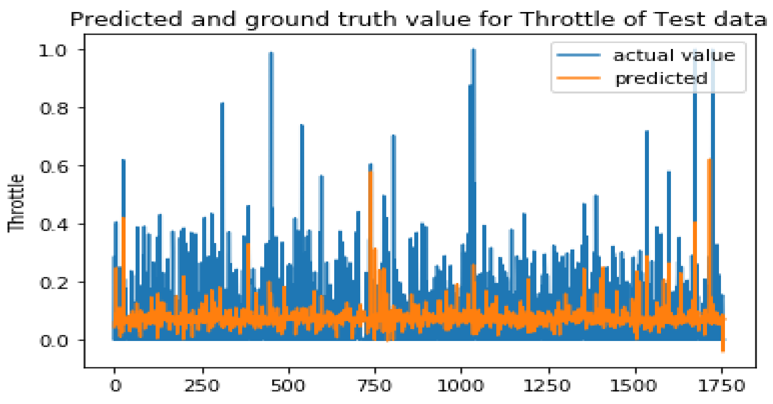


Figure 25- Throttle, Predicted vs. Actual

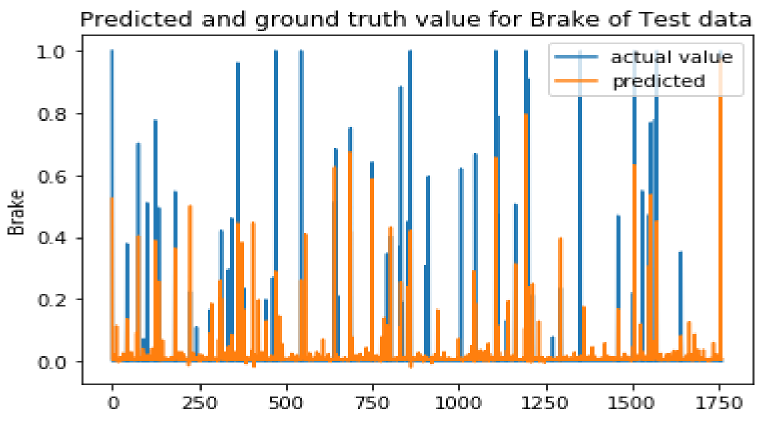


Figure 26- Brake, Predicted vs. Actual

From the graphs, we see the CNN's predictions are good. When tested in the simulator, the car drives well and stays in the lane for a good period.

5 Interface Implementation

In order to steer the car autonomously using the generated model, communication links should be established between the Udacity framework and the autonomous car model. Therefore, a Python script is written to act as an interface between the simulator and the model. As mentioned before, the model is designed such that it receives as input the images from the central camera mounted on the car, and the speed of the car. As output, it returns an array of three values required to control the car, meaning the steering angle, the acceleration throttle and the brake. The model needs to communicate with the simulator as fast as possible and reliable. Otherwise, any delay in communication would cause a malfunctioning of the whole system. In this regard and to generate a real-time bidirectional communication between the model and the simulator, the socket-IO package is used for both C# and Python platforms. Socket-IO library is originally written in JavaScript and is suitable for real-time applications [4]. It has two sides: a client-side which is written for this project in Python and sends the request as a client to the server and a server-side which is written in C# and is implemented in the simulator. This server accepts the request from the client (the model in this project) and establishes the connection. This process is illustrated with more details in Figure 27.

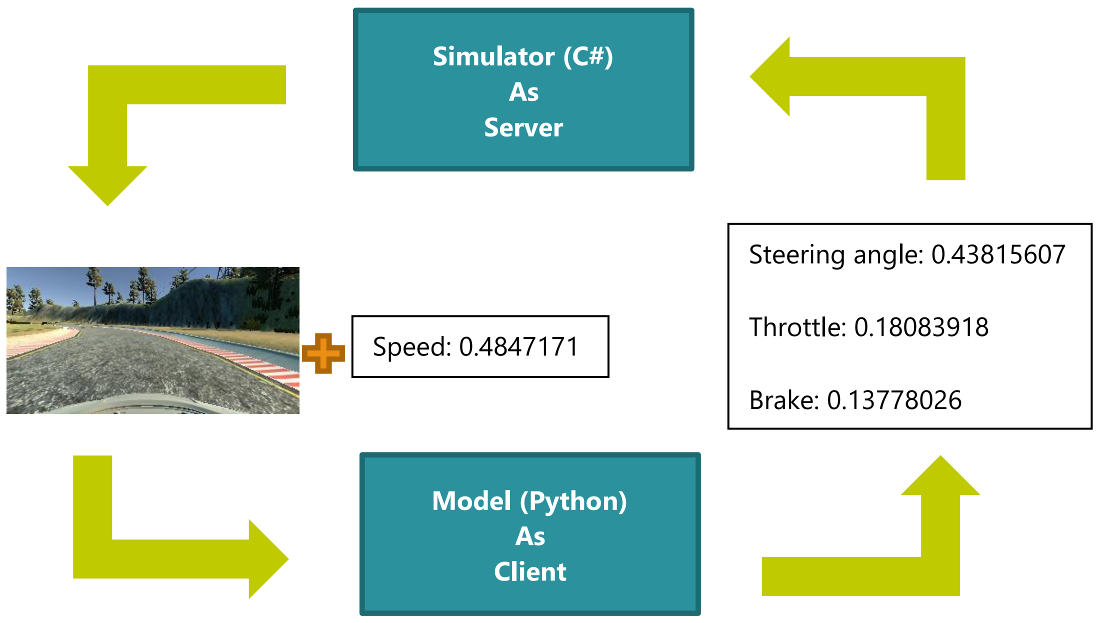
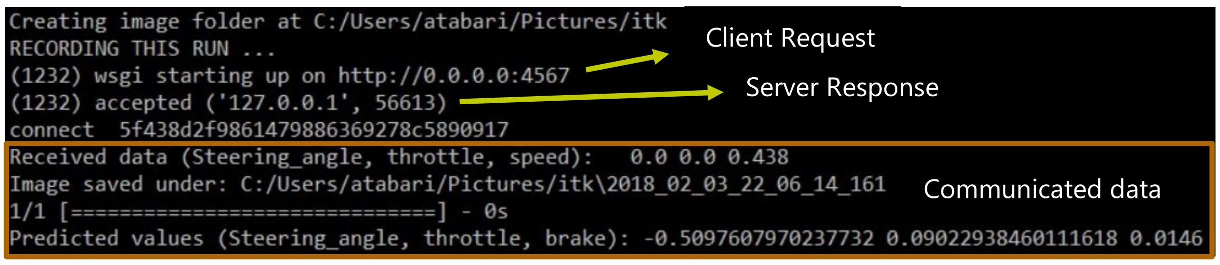


Figure 27- Server (Simulator) - Client (Model) Pair

As illustrated in Figure 27, the simulator acts as a server and the model as a client. The Python script sends the request to the server (Simulator) using the socket number which is defined on the server side (4567 in this project) and waits for the response. In case that the request is accepted from the simulator, a server/client pair is formed and the data is transmitted as depicted in the figure. A screenshot from the result of this process is shown in Figure 28.



```
Creating image folder at C:/Users/atabari/Pictures/itk
RECORDING THIS RUN ...
(1232) wsgi starting up on http://0.0.0.0:4567
(1232) accepted ('127.0.0.1', 56613)
connect  5f438d2f9861479886369278c5890917
Received data (Steering_angle, throttle, speed):   0.0 0.0 0.438
Image saved under: C:/Users/atabari/Pictures/itk\2018_02_03_22_06_14_161
1/1 [==============================] - 0s
Predicted values (Steering_angle, throttle, brake): -0.5097607970237732 0.09022938460111618 0.0146
```

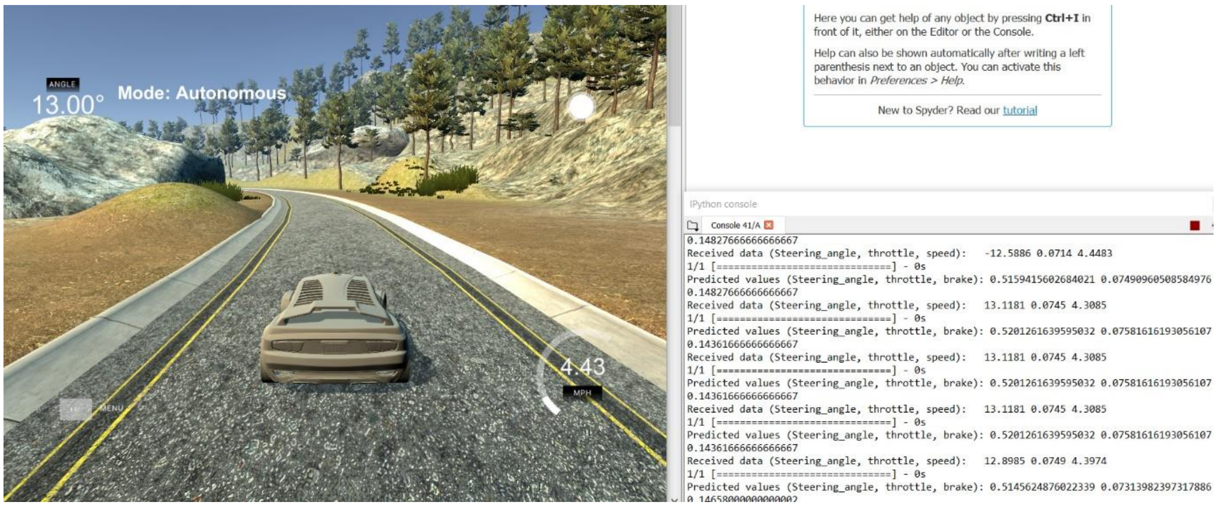
Figure 28- Interface Output

In Figure 28, the WSGI server is generated which is the standard Python approach for running web applications [7]. As it can be seen, it uses the same socket number (4567) which is defined on the server side in the simulator. Next, the connection is accepted from the simulator side on the IP address 127.0.0.1 and the server sends the session ID (sid) to the client that can be seen in front of the "connect" term in Figure 28. In the next lines, the controlling values received from the simulator are printed, meaning the steering angle, the throttle and the speed of the car. The images are also saved to a predefined address for later uses and data validation. Afterwards, the values are sent to the model and the predicted values are printed out in the following line, naming the new steering angle, throttle and the brake event magnitude. These controlling values are transmitted back to the simulator via the established link and the changes are implemented to the car behaviour by the C# code.

6 Results

6.1 Project Demo

The result of the project is illustrated by the following demo which shows the car in the simulated environment steered by the trained model. It is also possible to see the interaction of the model and the simulator from the IDE window in Figure 29. The demo is available under the following YouTube link: https://youtu.be/cXLwQQm_yLo



Here you can get help of any object by pressing **Ctrl+I** in front of it, either on the Editor or the Console.

Help can also be shown automatically after writing a left parenthesis next to an object. You can activate this behavior in *Preferences > Help*.

New to Spyder? Read our tutorial

```
0.14827666666666667
Received data (Steering_angle, throttle, speed):   -12.5886 0.0714 4.4483
1/1 [==============================] - 0s
Predicted values (Steering_angle, throttle, brake): 0.5159415602684021 0.07490960508584976
0.14827666666666667
Received data (Steering_angle, throttle, speed):   13.1181 0.0745 4.3085
1/1 [==============================] - 0s
Predicted values (Steering_angle, throttle, brake): 0.5201261639595032 0.0758161619305610‍7
0.14361666666666667
Received data (Steering_angle, throttle, speed):   13.1181 0.0745 4.3085
1/1 [==============================] - 0s
Predicted values (Steering_angle, throttle, brake): 0.5201261639595032 0.07581616193056107
0.14361666666666667
Received data (Steering_angle, throttle, speed):   13.1181 0.0745 4.3085
1/1 [==============================] - 0s
Predicted values (Steering_angle, throttle, brake): 0.5201261639595032 0.07581616193056107
0.14361666666666667
Received data (Steering_angle, throttle, speed):   12.8985 0.0749 4.3974
1/1 [==============================] - 0s
Predicted values (Steering_angle, throttle, brake): 0.5145624876022339 0.07313982397317886
0.14658000000000002
```

Figure 29- Project Demo

6.2 Outlook

As it can be seen in the attached demo, the model steers the car to stay on the road for a long period of time. It can happen that the car goes out of the road and because the speed is low and the height of the road and sidewalk is different, the car gets stuck in the corner of the road. There would be a great improvement in the autonomous driving of the car by overcoming this problem. Some suggestions to improve is to find the right amount of weight to be added to the throttle so that the speed of the car increases, but at the same time, this weight doesn't overshadow the learning of the other parameters used to drive the car. The best way to find the right weight is to try with various weights and analyze the output seen in the simulator. The network is trained with data of the car returning from the side of the lane back to the centre of the lane, another suggestion is to train it with more such data to get the best-trained model. Furthermore, we could also change the simulation to have the same height for the road and sidewalk so the car does not get stuck. It is also recommended to train the car with the new generated environment from section 4 to see how the model would behave in a new environment.

One interesting point of observation is the ground truth received from the simulator is time-correlated. In the model, this data is shuffled so the model is not trained in the same order, the continuity of the data is lost. The model is still able to drive the car autonomously based on the learning of its position on the road and the throttle, brake and steering angle at that point. The model can be improved by implementing a recurrent neural network, so the data can be fed in the same order it is received and the output of the previous unit will be considered as an input for the next unit.

7 References

[1] daimler.com. (2018). Daimler - Home. [online] Available at: https://www.daimler.com/innovation/autonomous-driving/special/definition.html [Accessed 14 Feb. 2018].

[2] bmw.com. (2018). BMW - Home. [online] Available at: https://www.bmw.com/en/automotive-life/autonomous-driving.html [Accessed 14 Feb. 2018].

[3] DataLab TUM. (2018). Data Innovation Lab - Autonomous lane following in a simulated environment project. [online] Available at: http://www.di-lab.tum.de/index.php?id=45&L=1 [Accessed 14 Feb. 2018].

[4] Wikipedia (2018). Wikipedia - Home. [online] Available at: https://en.wikipedia.org/wiki/Main_Page [Accessed 14 Feb. 2018].

[5] GitHub.com (2018). GitHub - Home. [online] Available at: https://github.com/ [Accessed 14 Feb. 2018].

[6] Introduction to Udacity Self-Driving Car Simulator (2018). Towards Data Science. [online] Available at: https://towardsdatascience.com/introduction-to-udacity-self-driving-car-simulator-4d78198d301d [Accessed 14 Feb. 2018].

[7] Python 3.5 Documentation (2018). Python documentation and libraries. [online] Available at: https://docs.python.org/3/library/ [Accessed 14 Feb. 2018].

[8] Udacity self-driving-car-simulator project at GitHub (2018). Udacity/self-driving-car-sim. [online] Available at: https://github.com/udacity/self-driving-car-sim [Accessed 14 Feb. 2018].

[9] Git-scm.com (2018). Git - Home. [online] Available at: https://git-scm.com/ [Accessed 14 Feb. 2018].

[10] Unity game engine (2018). Unity3D - Home. [online] Available at: https://unity3d.com/ [Accessed 14 Feb. 2018].

[11] TensorFlow.com (2018). TensorFlow - Home. [online] Available at: https://www.tensorflow.org/ [Accessed 14 Feb. 2018].

[12] OpenDRIVE 1.4 Specifications Document (2018). openDRIVE® - Home. [online] Available at: http://www.opendrive.org/docs/OpenDRIVEFormatSpecRev1.4H.pdf [Accessed 14 Feb. 2018].

[13] OpenRoadEd.net (2018). OpenRoadEd - Home. [online] Available at: https://sourceforge.net/projects/openroaded/ [Accessed 14 Feb. 2018].

[14] keras.io (2018). Keras - Home. [online] Available at: https://keras.io/ [Accessed 14 Feb. 2018].

[15] CS231n: Convolutional Neural Networks for Visual Recognition (Spring 2017). Courses at Stanford University. [online] Available at: http://cs231n.stanford.edu/index.html [Accessed 14 Feb. 2018].

[16] Deeplearning4J.org (2018). Deep Learning Tutorials. [online] Available at: https://deeplearning4j.org/logistic-regression [Accessed 14 Feb. 2018].

[17] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv:1409.1556, 2014.

[18] Alex Graves, Greg Wayne, and Ivo Danihelka. Neural tuning machines. arXiv preprint arXiv:1410.5401, 2014.

[19] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In Advances in neural information processing systems, pages 1097–1105, 2012.

[20] Rising, L., & Janoff, N. S. (2000). The Scrum software development process for small teams. IEEE Software, 17(4), 26-32.