Master's in Computational and Mathematical Engineering

Final Master Thesis

An open-source development environment for Self-driving vehicles

Aitor Ruano Miralles
Computational and Mathematical Engineering
Artificial Intelligence

Samir Kanaan Izquierdo
Carles Ventura Royo

05/2017

FICHA DEL TRABAJO FINAL

Título del trabajo:	*An open-source development environment for Self-driving vehicles*
Nombre del autor:	*Aitor Ruano Miralles*
Nombre del consultor/a:	*Samir Kanaan Izquierdo*
Nombre del PRA:	*Carles Ventura Royo*
Fecha de entrega (mm/aaaa):	05/2017
Titulación::	*Ingenieria Computacional y Matematica*
Área del Trabajo Final:	*Inteligencia Artificial*
Idioma del trabajo:	*Ingles*
Palabras clave	*Self-driving vehicles, deep learning*

Resumen del Trabajo (máximo 250 palabras): *Con la finalidad, contexto de aplicación, metodología, resultados y conclusiones del trabajo.*

Durante el proyecto, un entorno de desarrollo para vehiculos autonomos ha sido desarrollado, conjuntamente con dos modelos de Deep Learning que hacen uso de el, uno para deteccion de objetos y el otro para regresion del angulo del volante.

El entorno de desarrollo ha sido implementado sobre el famoso videojuego Grand Theft Auto V, mientras que los modelos de Deep Learning han sido desarrollados usando la popular libreria Tensorflow.

La tesis ha tenido exito global y diversas universidades y centros de investigacion hacen uso del entorno de desarrollo para llevar a cabo su propia investigacion. El proyecto tiene mas de 260 estrellas y 70 'forks' en GitHub.

Abstract (in English, 250 words or less):

During this thesis, a development environment for self-driving vehicles has been developed, along with two Deep Learning models that make use of it, one for object detection and the other for steering angle regression.

The development environment has been built in top of the famous game Grand Theft Auto V, while the Deep Learning models have been developed by using the popular Tensorflow library.

The thesis has had worldwide success and several universities and research centers are using the development environment to conduct their own research. The project has more than 260 stars and 70 forks on GitHub.

Index

List of figures

1. Introduction

1.1 Context and justification of the thesis

Self-driving vehicles are the promised technology of the near future, with some people arguing that together with AI they will ignite the fourth industrial revolution.

Their development has been trending in Silicon Valley for more than 10 years, leaded at the beginning by Stanford University (fig. 1) and followed by Google. In the past few years, several companies have joined the race, including Uber, Tesla, NVIDIA, Drive.ai, Cruise Automation and Auro Robotics along the most of the common traditional automakers such as Mercedes, Audi, Nissan, Toyota, BMW and General Motors. Most of them, setting their autonomous driving headquarters in the San Francisco Bay Area, California.

Illustration 1: Stanley, one of the very first self-driving cars

Just as it happened with the development of the digital revolution, the self-driving car revolution is being concentrated in just one place and it is difficult for other countries to keep pace with it. This is mostly, not for a lack of talent but for a lack of resources. Reducing the cost of developing self-driving cars could greatly speed-up development, not only in Silicon Valley but also in the rest of the world and democraticize the access to this technology.

For self-driving vehicles to become a reality, several things have to fall into place. Some of them are already there, but others are still pending:

Done:

- Enough computational power
- Good deep learning and traditional robotic algorithms

To do:

- Availability of data to train self-driving models
- Reduction of the cost of hardware
- User acceptance (prove it is safer than a human driver)

The main enabling technology behind self-driving cars is Deep Learning, which is in itself an extension of Machine Learning that uses bigger models and much more data to fit its models. This enables this algorithms to learn from very unstructured and high variability data (enormous state space), like are the frames taken by a camera in a vehicle dashboard, the number of different scenarios and situations you can encounter while driving is overwhelming: multiple shapes and colors of cars, erased lane lines, different weather conditions, weird looking pedestrians... In a very general sense, self-driving cars are learning to drive by seeing humans do it and by learning to detect objects, traffic signs, lane lines and other important features which previously had to be hand labeled and stored in a dataset consisting of thousands if not millions of samples.
Collecting, parsing and hand-labeling this amount of data requires an immesurable amount of resources (fig. 2) and this is why this is still a major point to solve by all the self-driving vehicle industry. And not only that, when a control algorithm is developed, the only way of testing it is by loading it in the vehicle and drive it on the street, which is also a time and money consuming activity, especially for trucks.

Illustration 2: Example of and-labeled object detection images

Moreover, self-driving vehicles are not only relying on vision, but also in other sensors such as Lidar and/or Radar, this gives the system much more accuracy on its location and distance and relative velocity to other objects by applying Data Fusion algorithms such as the Unescented Kalman Filter. Still, human drivers don't make use of radars to drive and still are able to reliably compute the distance to other vehicles and figure out where they are located, with this premise some engineers argue that self-driving vehicles should one day be able to drive by using cameras only, just as humans do, further reducing the cost of self-driving

technology, just as Tesla is trying to prove. For comparison, the price of a camera is on the hundreds, while the price of a Lidar sensor is on the tens of thousands, this is two orders of magnitude.

Finally, there is the political counterpart that must allow self-driving vehicles to be sold and used by the public. This probably won't happen until it is proven that a self-driving vehicle is safer than the average human, and probably, by a large margin.

The aim of this project is to solve the first and second TO-DO's and let the third be solved by society itself.

To solve this issues, an open-source development platform for self-driving vehicles will be developed, and we will also show that it can be successfully used to train self-driving deep learning models for instance. Moreover, once trained, this models can be plugged-in the open-source tool to test them before loading them into production.

This tool will allow for automated dataset generation, which means that the user will be able to collect an infinite amount of labeled data without even being present. Moreover, the tool will allow to simulate input commands such as steering angle, throtle and brake to see how algorithms behave in a simulated environment without the need of loading them into the vehicle to test them.

Finally, being open-source it will hopefully speed-up the research in vision-based localization algorithms all around the globe, without being restricted by the cost of testing this algorithms in a real vehicle with expensive hardware.

1.2 Objectives of the thesis

- Develop a self-driving vehicle development environment
- Try to guarantee it is easy to use and build a community of users around it
- Prove that developing and testing self-driving models is possible with this tool. Particularly, develop an object-detection and auto-steering model.

1.3 Followed strategy and methods

For the tool to be useful, it needs to somehow capture the variability of the real word driving scenarios, this includes among other things:

- Urban and inter-urban driving
- Different type, size and model of vehicles, pedestrians and animals
- Traffic signs and lights
- Unexpected behaviour of other vehicles, animals or pedestrians
- Different weather conditions

- Time cycle
- Variability of lane width and pavement state and texture
- Traffic density

Developing a virtual world that includes all this possible scenarios (and more) is an impossible task for the resources given for the project, luckily this world already exists and it is called Grand Theft Auto V, a very famous videogame that allows the player to freely move around a ficticious region called Los Santos, based on the real Los Angeles, California.

The player can not only walk, but also drive any kind of vehicle, from bicycles to aeroplanes, in montain, sea or urban areas under different weather and light conditions in a very decent replica of the west-coast US infrastructure.

The project takes benefit of the videogame, which is for now considered the most realistic simulation of the real-world ever created, to get all the necessary information to build the tool on top of it.

To do so, it will make use of a third-party library created by Alexander Blade to access the internal memory registers of the game, which will allow to use C++ code to call internal native methods and get and set information such as surrounding vehicle positions, current speed and throttle level in order to build our development environment, that will allow us to set-up scenarios, collect labeled datasets and programatically drive the vehicle.

1.4 Thesis plan

See attached Gantt chart

1.5 Briefing of the obtained products

In this project an open-source self-driving vehicle development environment has been developed with the aim of speeding up the development of vision-based self-driving vehicles, reducing the cost associated and democratizing the access to it.

The tool allows the user to automatically collect labeled datasets and test his models in the simulation.

Since its release, the development environment has been forked 70 times and starred 250 on GitHub. Its users include universities, companies, research centers and self-driving technology aficionado groups from all around the globe, including Spain, China, Taiwan, South Korea, Russian Federation, India, US, Germany, UK, Japan, The Netherlands...

Some of them are actively using it for research and have built self-driving models that run in the game.

1.6 Briefing of other chapters in the memory

The chapter 2 includes the main content of this memory, which is an explanation of the developed environment and agents.

Next, chapter 3 discusses the results and impacts of the project and finally in chapter 4 I reflect my personal conclutions.

Chapters 5 and 6 correspond to the glossary of terms and bibliography respectively.

2. Development process

2.1 The initial idea

It was a sunny workday of July 2016, at that time I was working as a project engineer in the electronics department of Applus+ IDIADA, an Spanish company that offers engineering services to OEM car manufacturers.

At that time I was assigned to R&D European projects so I had the luxury of travelling a lot around Europe. One of these projects, the i-Game project [2], brought my attention, it was about organising a contest of self-driving vehicles, where participating universities could build their own automated cars and try to solve typical self-driving scenario cases such as platoong forming, lane merge and intersection speed adjustement without traffic lights!

The contest, the Grand Cooperative Driving Challenge was successfully celebrated in the Netherlands (fig. 3). I was so amazed by the technology behind self-driving vehicles and the enourmous social and economic implications it would have that I decided at that moment that my thesis had to somehow be related to this.

Illustration 3: A colleague and I at the GCDC

Since I was a child I always had a passion for computer science, concretely for the field of AI, the brain is the ultimate general-purpose computing machine, knowing how it works and building such a system is the dream of any Alan Turing contemporary.

The drawback is that this is still a long distant achievement, however, self-driving vehicles are the nearest intersection of AI and social disruption, so before trying to build an Artificial General Intelligence agent, the most reasonable and pragmatic step is to invest time on discovering how vehicles could drive themselves.

When thinking about my Master's thesis I wondered how I would be able to get the resources necessary to develop a self-driving car, and my conclusion was that it would take me much more time than what I had to deliver my thesis. Still, I wanted to really make an input to the field and I thought that actually, thousands if not millions of people in the world may have the same issue as me. The goal was clear, democratize the access to self-driving technology.

Nowadays you don't need a car to drive one, videogames let you do so it in fairly realistic conditions, if we could simulate self-driving technology in one of these games it would allow anybody with a computer to develop and try new algorithms in the comfort of their sofa, surprisingly no company had introduced yet such features in any commercial driving game, I had to!

2.2 Maximizing the contribution

From the very beginning I thought of using the Torcs open-source racing game for the purpose [3], this game however presents several drawbacks:

1. Low quality graphics, which could make it difficult to transfer to the real-world the models developed in the game.
2. Close racing tracks, the number of tracks are very limited as well as the number of vehicles.
3. Very little variability, being a close track game there are no elements such as pedestrians or traffic signs, moreover, there is no control on the weather for example.

Briefly speaking, Torcs was an easy and limited environment to effectively develop state-of-the-art self-driving technology (fig. 4).

Illustration 4: Example Torcs screenshot

This thesis could have consisted on developing a model that successfully completed a Torcs track in record time as it was my very first initial idea. However that wouldn't had any visible contribution to the field of AI nor to the self-driving community. Instead, I decided to invest my time on developing a more useful tool.

This is when Grand Theft Auto V came to my mind, there is no actual game as complex as this one (fig. 5), and although it is not a driving game, still the game let's you hit anybody and steal his car to drive it with fairly realistic physics. Moreover, it is just super fun to spend time on it, so it would be a wonderful experience to write my thesis around it.

Illustration 5: Example GTAV screenshot

GTAV is the ideal candidate for such task, it has day/night cycles, hundreds of different cars, unpredictable weather, pedestrians, traffic signs and whole urban and inter-urban scenarios to drive in, all of that with stunning graphics and textures. The only unfortunate issue is that the game is propietary, so there is no direct evident direct access to the game's internal engine, a necessary requirement to get the output we need from the game

The solution comes thanks to the great modding community, as it will be explained in the following chapter.

2.3 Methods and tools

GTAV has a strong modding community [4], these are game's aficionados that have built hacking tools, software and new textures that let you modify the game almost in any way.

The most famous of this tools is called ScriptHookV [5] , a DLL plugin that fakes a game's shared library to access its internal registers, which includes both variables and functions. At the same time, this library allows to load external ASI plugins that make use of the public API provided by ScriptHookV, this means you can write your own C++ including the ScriptHookV header and call any of its methods and access any of its published variables, compile it and load it within the game.

In this particular case, I use ScriptHookV to gather information from the game, such as position of the surrounding vehicles and pedestrians or deviation from the center of the lane and then I provide a JSON API to the external world via socket connection.

I do this, instead of building the whole tool in the plugin itself, to give the user platform and language freedom of choice. GTAV only runs on Windows, so that would constrain the researcher to develop his agent using this OS only, also ScriptHookV is C++ only. For instance, implementing the sockets allows the researcher to have the game running on a Windows machine and the AI model implemented in Python running in a separate Linux machine. Moreover, the JSON messages allow to dynamically change the self-driving environment parameters without needing to restart the game every time.

On the agent side, I've been using Python [6] to develop the models. Python is the most common language used in the field of AI and Deep Learning in particular, and it is also the language I am more comfortable with nowadays. To develop the Deep Learning features, I've been using Keras [7] on top of Google's Tensorflow [8], also one of the most used libraries in the Deep Learning communities and that provides a wonderful Python API.

The tools used are then:

Environment side:

- Grand Theft Auto V (PC version)
- Visual Studio Express 2012
- ScriptHookV
- Standard Microsoft C++ libraries

Agent side:

- Python 3.1 (For the external interaction with the environment)
- Tensorflow 1.1
- Keras 2.0

2.5 Implementation details (Environment side)

The code for DeepGTAV can be found in my GitHub repo: https://github.com/ai-tor/DeepGTAV

DeepGTAV works the following way:

Once the game starts, DeepGTAV opens a TCP connection on port 8000 and waits for TCP clients to connect.
Clients connected to DeepGTAV are allowed to send messages to GTAV to start and configure the research environment (*Start* and *Config* messages), send driving commands to control the vehicle (*Commands* message) and to stop the environment to fallback to normal gameplay (*Stop* message).

When the environment has been started with the Start message, DeepGTAV will start sending the data gathered from the game back to the client in JSON format, so the client can use it to generate a dataset, run it through a self-driving agent and many other possible applications.

The data sent back to the client and initial conditions of the environment will depend on the parameters sent by the client with the Start or Config messages. Thins that can be configured for instance are: frame rate, frame width and height, weather, time, type of vehicle, driving style, wether to get surrounding vehicles or peds, type of reward function and many more...

Here follows a list of the messages used to control DeepGTAV:

Start:

This is the message that needs to be sent to start DeepGTAV, any other message sent prior to this won't make any effect. Along with this message, several fields can be set to start DeepGTAV with the desired initial conditions and requested *Data* transmission.

When this message is sent, the environment starts, the game camera is set to the front center of the vehicle and *Data* starts being sent back to the client until the client is disconnected or a *Stop* message is received.

Here follows an example of the *Start* message:

```
{"start": {
  "scenario": {
    "location": [1015.6, 736.8],
    "time": [22, null],
    "weather": "RAIN",
    "vehicle": null,
    "drivingMode": [1074528293, 15.0]
  },
  "dataset": {
    "rate": 20,
    "frame": [227, 227],
    "vehicles": true,
    "peds": false,
    "trafficSigns": null,
    "direction": [1234.8, 354.3, 0],
    "reward": [15.0, 0.5],
    "throttle": true,
    "brake": true,
    "steering": true,
    "speed": null,
    "yawRate": false,
    "drivingMode": null,
    "location": null,
    "time": false
  }
}}
```

The scenario field specifies the desired initial conditions for the environment. If any of its fields or itself is null or invalid the relevant configuration will be randomly set.

The dataset field specifies the data we want back from the game. If any of its fields or itself is null or invalid, the relevant Data field will be deactivated, except for the frame rate and dimensions, which will be set to 10 Hz and 320x160 by default.

Config:

This message allows to change at any moment during DeepGTAV's execution, the initial configuration set by the *Start* message.

Here follows an example of the *Config* message (identical to the *Start* message):

```
{"config": {
  "scenario": {
    "location": [1015.6, 736.8],
    "time": [22, null],
    "weather": "RAIN",
    "vehicle": null,
    "drivingMode": [1074528293, 15.0]
  },
  "dataset": {
    "rate": 20,
    "frame": [227, 227],
    "vehicles": true,
    "peds": false,
    "trafficSigns": null,
    "direction": [1234.8, 354.3, 0],
    "reward": [15.0, 0.5],
    "throttle": true,
    "brake": true,
    "steering": true,
    "speed": null,
    "yawRate": false,
    "drivingMode": null,
    "location": null,
    "time": false
  }
}}
```

In this case, if any field is null or invalid, the previous configuration is kept. Otherwise the configuration is overriden.

Commands:

As simple as it seems, this message can be sent at any moment during DeepGTAV's execution to control the vehicle. Note that you will only be able to control the vehicle if *drivingMode* is set to manual during the *Start* or *Config* message:

```
{"commands": {
      "throttle": 1.0,
      "brake": 0.0,
      "steering": -0.5
}}
```

Stop:

Stops the environment and allows the user to go back to the normal gameplay. Simply disconnecting the client produced the same effect.

```
{"stop": {}}
```

The relevant project files are organized in the following manner:

Rewarders: Directory including classes to obtain a reward function to use for example in Reinforcement Learning agents, the rewards are always in the interval [-1,1].

Rewarder: Abstract class that implements the structure of all rewarders

LaneRewarder: The closer the vehicle is to the center of the lane the greater the reward, in the case the vehicle is driving agains traffic, the reward is negative.

SpeedRewarder: Rewards mantaining a certain defined speed, if the speed is lower than the set speed, the reward is positive, otherwise it is negative (conservative reward)

GeneralRewarder: It is a weighted sum of the LaneRewarder and the SpeedRewarder. It defines an aggressivity parameter (a) such that reward = a*SpeedRearder + (1 – a)*LaneRewarder. The greater the aggressivity the agent will tend to overtake other vehicles to mantain velocity.

bin: Contains the compiled .ASI plugin and game save files

lib: Contains external libraries used during the development, such as rapidjson [9] and ScriptHookV

Project files: Files such as DeepGTAV.sdf and DeepGTAV.vccproj are files generated automatically by Visual Studio to store the configuration of the project.

Scenario: The scenario class represents the user environment configuration, such as weather, time and type of car. It also provides useful methods to fill the JSON message with the game's information requested by the user.

ScreenCapturer: Utility class to manage the game screen capture

Server: Class that manages the socket connection with the agent, interfaces between the agent and the scenario by receiving, parsing and sending the JSON messages

script: Main file, it instantiates a Server class and runs indefinetly

Most of the implementation details of the environment are out of the scope of this project as they are more related to classical computer science than to AI. Nevertheless, it is worth mentioning some of them, as deep reverse engineering hacking had to be applied to be able to obtain the features needed by self-driving agents, such as the lane reward, the throttle, brake and steering angle of the vehicle and the bounding boxes around the game's objects for automated data-set generation.

Lane reward: The position of the lane lines can't be obtained using internal game's memory neither methods, and without their position it is mathematically impossible to know how centered is the vehicle in the lane.

To obtain those I had to use an external tool called OpenIV [10] that allows to decrypt the game's installation files and check if I could find any kind of map with information about the lanes. What I found was a *XML* file were road nodes and the links between them were listed. The nodes are sparsed all around GTAV's roads and they have certain properties, including the width of the road, the type of the road (highway, narrow-road, etc...), the number of lanes in the direction of the road, and the number of lanes in the other direction. The direction of the road happens to be the direction of the directed graph formed by the node links.

The width of the road is not a measure of distance but an enumeration, and it also depends on the type of road, this means that the distance in meters for a highway road with width zero, is not the same than the distance in meters for a normal road with the same width parameter. So it was not enough to deduce the meaning of this properties, it was also necessary to do some trial and error to map their combination to real distances.

Knowing the real width distance of the road and the number of lane lines, it was straight forward by using some linear algebra to deduce how far was our vehicle from each of the lane lines, and also, by knowing the direction of the link joining the two closest nodes and the direction of our car with respect to the world coordinates it was also easy to know if we are going in the wrong way.

Details of the implementation can be seen in *LaneRewarder.cpp*

The following video, created by an OpenAI engineer shows the result of this implementation, which was merged with OpenAI's DeepDrive project: https://www.youtube.com/watch?v=6zIClaSCTcc

Vehicle's throttle, brake and steering: When manually driving a vehicle (using the keyboard or a game controller) it is fairly easy to get vehicle's throttle, brake and steering by reading the controller inputs, however this is not the case when the vehicle is driven by the game's internal AI, because it directly inputs the variables into the game's internal registers. It was important to get this values even when the game's AI was driving, because that would allow to automatically generate regression values without needing the user to drive for hours.

There is no method in ScriptHookV's API to access this variables. In this case I had to rely in another tool called CheatEngine [11] to read the game's memory pointers. Again, by trial and error, I looked for values that changed when the vehicle was accelerating or turning, until I eventually found the pointer addresses of the correct values and read them in the C++ code.

Details of the implementation can be seen in *Scenario.cpp*

Bounding boxes: Like in the Lane Reward, this is something that still hadn't been done before and it is a feature that has been integrated in a lot of other projects due to its applicability. Bounding boxes of objects allow to train object-detection algorithms that are able to locate and classify objects in an image, a major problem in computer vision. For the case of self-driving vehicles we were interested in doing so for vehicles and pedestrians.

In this case, through ScriptHookV's methods it is possible to know dimensions, position and orientation vectors of objects, this means that by correctly understanding this data it is possible to easily compute the top-right and bottom-left corners of a 3D bounding box surrounding the objects.

Details of the implementation can be seen in *Scenario.cpp*

I made a video of this feature to announce its release, it can be seen in the following link: https://www.youtube.com/watch?v=StR9tuaIwYw

2.6 Implementation details (Agent side)

The code for VPilot can be found in my GitHub repo: https://github.com/ai-tor/VPilot

VPilot makes use of DeepGTAV with three main purposes:

1. Provide an example to DeepGTAV users on how to communicate with the environment, in this case using Python.
2. Use DeepGTAV to store and object-detection and regression dataset
3. Use the trained Deep Learning model with the generated dataset along with DeepGTAV to test it

The project structure goes as follows:

deepgtav: Implements utility classes to easily interface with DeepGTAV

client: Implements a socket client class that understands DeepGTAV messages and implements methods to send/receive them and to store them in a Pickle Python file to generate a dataset

messages: Implements the codification/decodification of the DeepGTAV JSON messages.

models: Includes the object-detection and steering regression Deep Learning models

CROWD: Object detection model based on SSD [12]

NASA: Steering regression model based on NVIDIA's PilotNet [13]

dataset: Starts a DeepGTAV session and starts capturing a dataset to be used by the models

drive: Starts a DeepGTAV session and starts sending the commands generated by the models back to the game to see the result

Again, I will focus on explaining the inner workings of the code that relate to AI, these are basically the two deep learning models: CROWD (Cheap Real-Time Object Window Detector) and NASA (Neural Auto-Steering Algorithm)

CROWD

Cheap Real-Time Object Window Detector is an end-to-end Deep Learning object detection model. Object detection is a common problem in computer vision, which given an arbitrary image as input, tries to identifity objects, surround them with bounding boxes and classify its type accordingly (fig. 6).

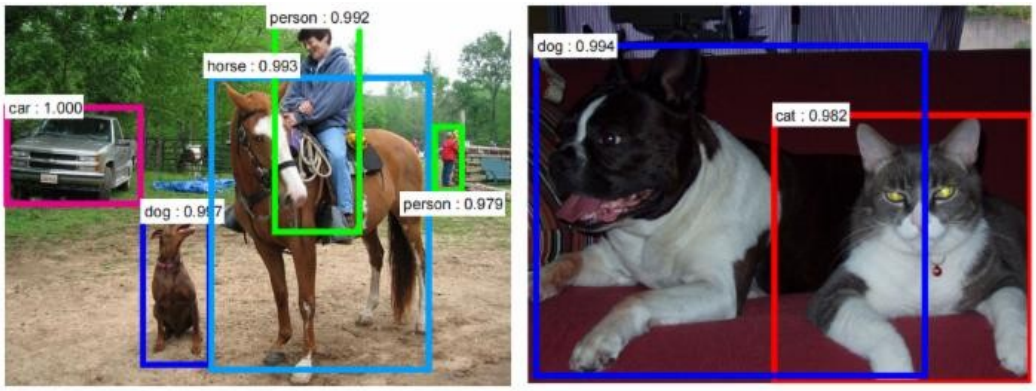

Illustration 6: Bounding boxes example

Note the difference between a traditional classification problem and object detection. The latter case is much more complicated, because we aren't just classifying an image in different categories, we are also trying to deduce the number of different objects in an image, their precise location and their category. This usually makes object detection models

much bigger and computationally more expensive compared to simple classification architectures.

This is another issue, because object detection algorithms in self-driving vehicles have to run in real-time (min 10 Hz) in order to be useful. Till recently, object detecion models have been splitted in two steps. First, locate the objects in the image (region proposal) and second, classify those objects (classification). These are the cases for instance of RCNN, Fast-RCNN and Faster-RCNN [14]. Although version after version the computational cost of these models has been reduced and so they have been able to run faster, still at prediction time the best frame rate was of 5 Hz for Faster-RCNN.

This trend changed when SSD (Single Shot Detector) was introduced, which is the base model architecture we are taking for CROWD. SSD is end-to-end, which means that is not a two step process, but both region proposal and classification are trained at the same time in a single neural network architecture. This is extremely beneficious, because weights from region proposal and classification are shared within the layers thus reducing by orders of magnitude the number of neural network parameters. In the case of SSD, we reach frame rates of 50 Hz on NVIDIA's Geforce 1080 Ti.

The ideas that made this possible are basically two:

- The use of Fully Convolutional Neural Networks: They replace the traditional fully-connected layers, typically used in the last layers of classification, with convolutional layers, thus making the models fully convolutional. Convolutions represent a saving in terms of network parameters because the kernel weights in convolutions are shared espatially across the input dimensions. [15]

- The introduction of priors (or anchors): These are a set of manually defined bounding boxes prior to training. The predicted bounding boxes are then computed relatively to this prior boxes. This helps reduce the search space during training of such models, thus making them feasible in what respect to training time.

Finally, the loss function (the output of which is used to compute the network gradients and update the weights) also differs from the classification problem, as in this case we don't only need to account by the error in the classification but also in the object localization.

After SSD, other similar architectures such as YOLO [16] and SqueezeDet [17] also appeared, however there are no major break-throughts among these, just small differences in what respects to architecture configuration and loss functions.

A major common property among these models is that all of them are builded on top of an already trained classification neural network in the

ImageNet dataset [18], in the case of SSD and YOLO, this architecture is VGG-16 [19], while in SqueezeDet it is SqueezeNet. Using pre-trained weights in part of the neural network also helps to speed-up convergence, as models such as VGG-16 can take weeks to be fully trained on the ImageNet dataset. Building on top of it, can reduce the training of object detection from weeks to days (fig. 7).

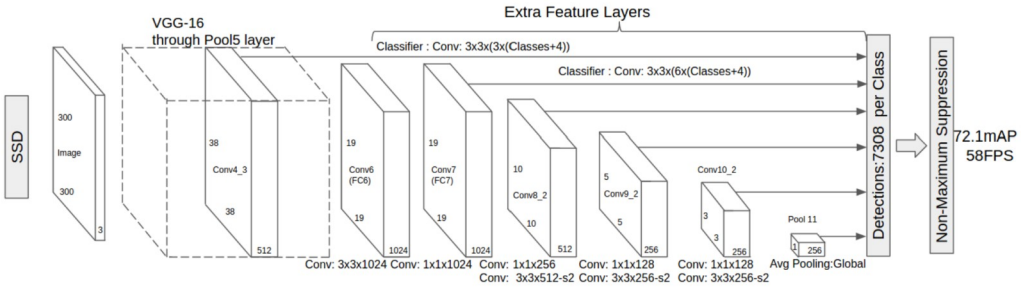

Illustration 7: SSD architecture

In our case, we use a pre-trained SSD on the PASCAL VOC dataset [20] and re-train it using the DeepGTAV dataset, the training takes only a few hours! For that purpose, the official SSD Caffe architecture has been ported to Keras according to the official paper for the purpose of our project.

During training we add intensive data augmentation to improve overall performance and reduce overfitting. Basically we randomly change brightness, contrast, saturation and lighting while also randomly shifting the image vertically and horizontally.

The resulting code is structured as follows:

models/SSD

SSD: Implements the Keras model according to the original paper

utils: Implements useful methods such as loss function, Keras visualization callbacks and data augmentation

utilities/ImageTransformer: Implements image utilities for the data augmentation

crowd: Implements basic methods around SSD and utils to setup training and testing of the model and provide an API for VPilot.

train: Starts a model training session

NASA

Neural Auto-Steering Algorithm is a regression model that maps from an input 200x66 RGB (or YUV) image to a steering angle. Thus, when trained, the neural network should be able to recognize the most important features in a road to compute the necessary steering angle to stay within the lane and take curves.

This is a much simpler problem than object detection, but still, it requires some important tricks to guarantee generalization. For instance, if the neural network is simply trained with images and steering angles captured from a vehicle driving in the center of the lane, the model won't have a clue what to do when the vehicle deviates a bit from the center of the lane, because it has never seen such a case. It is important then, to also teach the neural network to recover. To do so, images are randomly shifted and rotated to simulate this deviation, and the steering angle is adjusted so that the vehicle returns to the center of the lane in two seconds.

NASA is based on NIVIDIA's PilotNet (fig. 8) but we include a some improvements to the original implementation:

- We change the ReLu activations by ELU activations, which are shown to solve the «dead neuron» problem [21]
- We add Batch Normalization after each convolutional layer, to speed-up convergence [22]
- We make use of Dropout in the final fully-connected layers to improve generalization (reduce overfitting) [23]

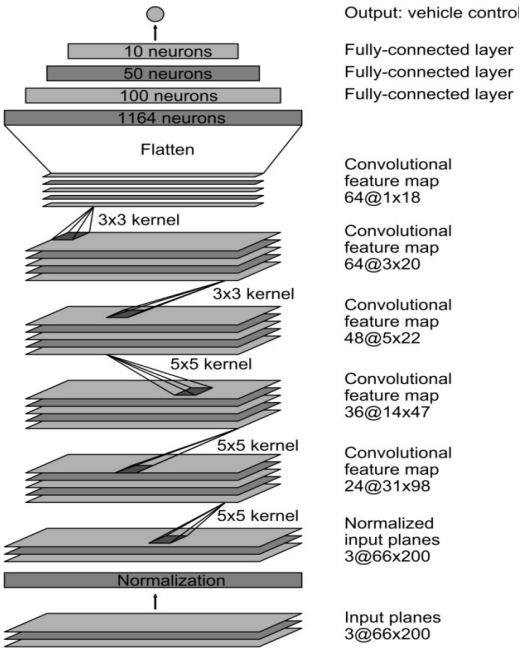

Illustration 8: PilotNet architecture

NASA's code structure is very similar to that of CROWD:

models/PilotNet

PilotNet: Implements the Keras model according to the original paper

utils: Implements useful methods such as Keras visualization callbacks and data augmentation

utilities/ImageTransformer: Implements image utilities for the data augmentation

nasa: Implements basic methods around PilotNet and utils to setup training and testing of the model and provide an API for VPilot.
train: Starts a model training session

3. Results and impact

In this project an open-source self-driving vehicle development environment (DeepGTAV) has been developed with the aim of speeding up the development of vision-based self-driving vehicles, reducing the cost associated and democratizing the access to it.

DeepGTAV is available on GitHub with detailed instructions of installation and usage, along with companion libraries to interface with it using Python.

Moreover, two Deep Learning models have been implemented and successfully trained to make use of this data, one for road understanding (object detection) and another for auto-steering. In both models we adapt them to our self-driving platform, we make improvements and show the potential of the tool.

The tool allows the user to automatically collect labeled datasets and test his models in the simulation. Among other things, it is possible to:

- Set the weather
- Set the location
- Set the time
- Set the vehicle to use
- Set the frame rate of dataset recording
- Set the type of driving style (in automated mode)
- Set the type of objects to be labeled
- Get 3D bounding boxes for the objects
- Get ego information about the vehicle
- Get reward score (for reinforcement learning models)

Moreover, it is possible to control the vehicle without the need of a game controller, so an agent can send the commands to the game as it was a car so that it can be quickly tested inside the game.

Since its release, the development environment has been forked 70 times and starred 260 on GitHub. Its users include universities, companies and research centers from all around the globe, including Catalonia, China, Taiwan, South Korea, Russian Federation, India, US, Germany, UK, Japan, The Netherlands...

Some of them are actively using it for research and have built self-driving models that run in the game. Here follows a list of *known* active DeepGTAV users:

- Centre de Visió per Computador (Catalonia)
- Huawei (Canada)
- National Sun Yat-sen University (Taiwan)

- ETH Zürich (Switzerland)
- Linnaeus University (Sweden)
- Texas Instruments (India)
- University of Modena and Reggio Emilia (Italy)
- Panasonic R&D Center (Singapore)
- University of Wisconsin-Madison (US)
- Autobon.ai (US)
- OpenAI (US)
- Uber/OTTO (US)
- Clemson University (US)
- University of Pennsylvania (US)

During the development of DeepGTAV, I had close collaboration with OpenAI to bring the environment to life inside Universe. Unfortunately, the project had to be taken down soon after it was released, due to legal actions from Rockstar (company behind GTAV)

Luckily, DeepGTAV could continue its development thanks to the fact that it is a non-profit open-source project.

At a personal level, due to the repercussion of this project I eventually got several job interviews, mostly from Silicon Valley but I finally accepted a job offer from a company that is building self-driving trucks, from Illinois, Chicago.

As of today, there is still no simulation environment for self-driving vehicles as big as it is DeepGTAV, although probably will start to exist in the next few months to come, now that the industry has seen the importance of tools like this. In the field of self-driving drones, for example, Microsoft recently built an open-source environment.

4. Conclutions

The initial idea of this project was to use Deep Learning to build a self-driving agent that could drive autonomously in a videogame. I planned to do so using Torcs, but seeing that it was already solved I decided to use GTAV, a much more challenging environment.

I didn't expect at all such an impact in the self-driving community when I first released DeepGTAV, more than 50 bug reports and improvement suggestions have been filled as of today and at the same time I was receiving proposals of collaboration and job offers from around the globe, especially the US.

This forced me to deviate from the initial plan. I expected to invest around 70% of the project's time to Deep Learning and about 30% in building the environment, eventually it finished being the complete opposite. In this case, I unfortunately have to say say that the objectives of the final project haven't been met, although I feel very happy for the contribution I have made to the field of self-driving cars. I guess I was at the right moment at the right place [24].

The project gave me the opportunity to closely work with the self-driving car community, first collaborating with Elon Musk's OpenAI and second by getting a job as a Deep Learning engineer at a self-driving truck start-up in Illinois, Chicago. From where I am currently writing this words.

This has also helped me to learn much more about Deep Learning and the self-driving technology, which helped me to write the two Deep Learning models presented in this project in a record time.

In the future I would like to finish what I couldn't with this project, by improving my models and deep dive into the world of Reinforcement Learning, an area of Deep Learning that has still lot to deliver, especially in what regards to Artificial General Intelligence.

5. Glossary of terms

AI: Artificial Intelligence
AGI: Artificial General Intelligence
API: Application Programming Interface
ASI: Assembler Include
CROWD: Cheap Real-time Object Window Detector
ELU: Exponential Linear Unit
GTAV: Grand Theft Auto V
JSON: JavaScript Object Notation
NASA: Neural Auto-Steering Algorithm
OEM: Original Equipment Manufacturer
RCNN: Regions with CNN features
R&D: Research and Development
ReLU: Rectifier Linear Unit
SSD: Single-Shot Multibox Detector
TCP: Transmission Control Protocol
VOC: Visual Object Classes
XML: eXtensible Markup Language
YOLO: You Only Look Once

6. Bibliography

1: Tesla, Inc., Autopilot, 2017, https://www.tesla.com/autopilot
2: TNO, i-GAME Project, 2017, http://www.gcdc.net/en/i-game
3: Bernhard Wymann, Eric Espie, Torcs, 2017, http://torcs.sourceforge.net/
4: Anonymous, GTA Modding forums, 2017, http://gtaforums.com/forum/312-gta-modding/
5: Alexander Blade, ScriptHook V, 2017, http://www.dev-c.com/gtav/scripthookv/
6: Guido van Rossun, Python Programming Language, 2017, https://www.python.org/
7: Francois Chollet, Keras, 2017, https://keras.io/
8: Anonymous, Tensorflow, 2017, https://www.tensorflow.org/
9: Milo Yip, RapidJSON, 2017, https://github.com/miloyip/rapidjson
10: Anonymous, OpenIV, 2017, http://openiv.com/
11: Anonymous, CheatEngine, 2017, http://www.cheatengine.org/
12: Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, Alexander C. Berg, SSD: Single Shot MultiBox Detector, 2016
13: Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D. Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, Xin Zhang, Jake Zhao, Karol Zieba, End to End Learning for Self-Driving Cars, 2016
14: Shaoqing Ren, Kaiming He, Ross Girshick, Jian Sun, Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks, 2016
15: Yann LeCun, Leon Bottou, Yoshua Bengio, Patrick Haffner, Gradient-Based Learning Applied To Document Recognition, 1998
16: Joseph Redmon, Ali Farhadi, YOLO9000: Better, Faster, Stronger, 2016
17: Bichen Wu, Forrest Iandola, Peter H. Jin, Kurt Keutzer, SqueezeDet: Unified, Small, Low Power Fully Convolutional Neural Networks for Real-Time Object Detection for Autonomous Driving, 2016
18: Fei-Fei Li, ImageNet, 2016, http://www.image-net.org/
19: Karen Simonyan, Andrew Zisserman, Very Deep Convolutional Networks for Large-Scale Image Recognition, 2015
20: Mark Everingham, Luc van Gool, Chris Williams, John Winn, Andrew Zisserman, The PASCAL Visual Object Classes Homepage, 2017, http://host.robots.ox.ac.uk/pascal/VOC/
21: Djork-Arné Clevert, Thomas Unterthiner, Sepp Hochreiter, Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs), 2016
22: Sergey Ioffe, Christian Szegedy, Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift, 2015
23: Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, Ruslan Salakhutdinov, Dropout: A Simple Way to Prevent Neural Networks from Overfitting, 2014
24: Dana Hull, Don't Worry, Driverless Cars Are Learning From Grand Theft Auto, 2017, https://www.bloomberg.com/news/articles/2017-04-17/don-t-worry-driverless-cars-are-learning-from-grand-theft-auto