Final Machine Learning Project Autonomous Driving Training for Obstacle Avoidance using Reinforcement Learning

Bruno Mendes 68411 Pedro Silva 72645

Abstract—This document contains a description of the process of developing a project for the subject of Machine Learning, which is part of the fourth year of University of Aveiro's course Engenharia de Computadores e Telemática, and falls within the scope of Computer Vision. It showcases the work done, what was considered most important, the main goals, along with the problems and solutions found in order to develop this tool. Suggested and accepted as a final assessment item for the subject, this tool was created with the intention that it would allow for object detection, obstacle avoidance and its uses for real-life remote-control cars that uses distance sensors like the ATLASCAR project. Through an approach that is intended to be simple and organised, this report starts by setting the context and motivation that back this project up. The features, ideas, problems and solutions are then presented, following roughly the real chronological order in which they came about throughout the development of the project. Finally, the main conclusions are discussed, going over the acquired knowledge, the main goals and their completion, and the enriching characteristics of the project, technical-wise and social-wise.

Index Terms—Machine Learning, Computer Vision, Autonomous Driving, Object Detection, Obstacle Avoidance, Pygame, Neural Network, Reinforcement Learning, Imitation Learning, Deep Q-Network, Q-Learning.

1 Introduction

In the context of developing the final project for Machine Learning, a subject that is part of the fourth year of University of Aveiro's course Engenharia de Computadores e Telemática, this idea of creating a tool that can be used for training an autonomous vehicle to drive indenpently around a world created by an autonomous driving simulator was suggested and accepted as a final assessment item. It is interesting to create tools that have real use, particularly in the scope of a bigger project. In this case, the work shall prove of particular interest in the context of the master's thesis of one of the authors, focusing on the study and adaptation of autonomous driving simulators for the ATLASCAR project of the LAR personnel of the DEM department of Aveiro University.

This document is made up of 4 sections. After this introduction, the motivation behind this project is explained in section 2, contextualising the work carried out. Then, in section 3, a review of different techniques used to handle problems similar to the main problem shall be presented, detailing some of the results and their basic limitations. After that, in section 4, the description, visualization and statistical analysis of the dataset tool is exposed, describing the main problem that we want to solve and discussing some of the problems found and solutions developed in order to obtain the results showcased. Afterwards, in section 5, the machine learning algorithm used in this project will be showcased as well as its results for the experiment at hand. Finally, in section 6, some conclusions about the development of this project are presented.

2 The motivation

In the scope of the master's thesis of one of the authors, which focuses on studying and adapting autonomous driving simulators for

the ATLASCAR, implementing the sensor setup of the ATLASCAR2 described in figure 2 into a vehicle and spawn this vehicle in the world created by the autonomous driving simulator in order to perform data acquisition and test out the algorithms that are going to be implemented in the ATLASCAR2 before implementing these algorithms in the real platform. The autonomous driving simulator chosen for the purposes of this dissertation was a simulator called CARLA for its ability to render realistic scenarios and controlled environments as described in figure 3. This simulator spawns a vehicle equipped with sensors at a random position together with a Pygame Window that can be used to control the vehicle and report the vehicle stats as seen in figure 4. This simulator provides some state-of-the-art sensors that can be used for data acquisition and it also provides two driving methods for the vehicles in which one is based on manual control using the keyboard keys and the other one is based on an autopilot system that was created by the CARLA's research and development team in the Computer Vision Center in Barcelona and it was implemented based on three approaches to autonomous driving: modular pipeline, imitation learning and reinforcement learning. However, this autopilot is currently being developed and tested by its developers and so it still presents some problems namely when it comes to avoiding obstacles that come in the way of the vehicle. This is mainly due to the complexity factors that come with this problem, such as the number of obstacles, their size and their current position in relation to the vehicle. As such, in order to resolve this problem, creating a new autopilot system based on reinforcement learning seemed like the way to go.

The tools created in this project were built in a way that would best suit their use in the context of the master's thesis of the student Pedro Silva. Although the work in this dissertation is pretty much finished, we hope to add this tool as a new plot point that can be used by anyone that want to continue the work done in this dissertation.

Fig. 1: ATLASCAR2[1]

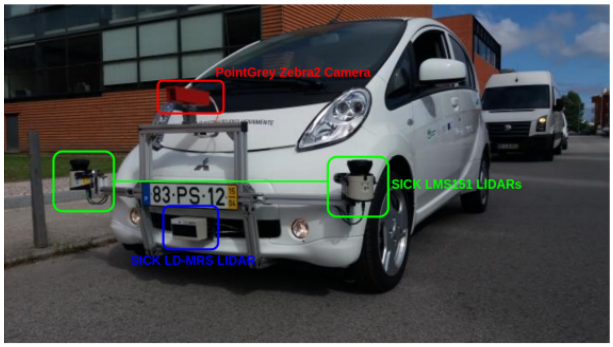

Fig. 2: ATLASCAR2 Sensors[2]

Fig. 3: CARLA Simulation Environments[4]

Fig. 4: CARLA Pygame Window[11]

3 State of the Art Review

Sensorimotor control in autonomous driving remains a major challenge in machine learning and robotics. The development of autonomous ground vehicles has been a long-studied instantiation of this problem and this particularly shown in navigation in densely populated urban environments where there are multiple object obstacles that need to be avoided.

One of the main tasks tasked to any machine learning algorithm implemented in a self-driving car is a continuous rendering of the surrounding environment and the prediction of possible changes to those surroundings and this can be achieved by performing these four sub-tasks: object detection, object identification or recognition using object classification, object localization and predict the movements of these objects. Machine learning algorithms, such as regression, pattern recognition, clustering and decision matrix algorithms can be applied on these sub-tasks.

3.1 Related Work based on Regression Algorithms

Regression algorithms can be used for developing image-based models that can be used for prediction and feature selection for object detection. These algorithms usually use images as input data and they play a very important role in localizing an object in an image and act accordingly. This is extremelly important in working with datasets with labelled data that are used to develop autonomous vehicles. One of example of these datasets, is the KITTI(Karlsruhe Institute of Technology 2019)[13], which is one of the main datasets used in the fields of autonomous driving. This dataset was recorded using a Volkswaggen station waggon and has grown significantly over the years by adding more results and more sensors to the vehicle.

This dataset consists of a series of image sequences followed by a text file in which, for each frame the various objects in the field of view are depicted with an identification number, a label and a set of 2D and 3D space coordinates about their position[18]. Using the information from this dataset, it is possible to compute trajectories based on visual odometry, image disparity using image techniques such as Optical Flow and 3D object label information as seen in figure 5.[17]

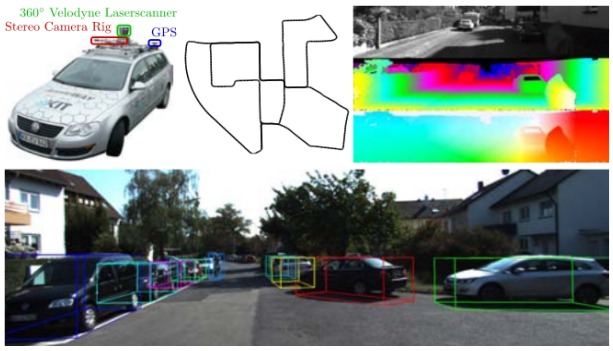

Fig. 5: KITTI Recording platform with sensors (top-left), trajectory based on visual odometry (top-center), disparity maps computed with Optical Flow (top-right) and 3D object label information (bottom)

The trajectory computation is based on a technique called Visual SLAM(VSLAM) that is used in order to generate graphs for global path planning. VSLAM generates a trajectory network which is usually in the form of a spare graph(if it is odometry based) or in the form of probabilistic relations on landmark estimates relative to the vehicle. Running an regression algorithm to perform object detection in parellel, has been proven to add more information to this trajectory network, and it can be seen as a good solution for multi-objective path planning, by the thesis of Ami Woo in 2019.[10] The main algorithm used for object detection in this thesis was `YOLO V2` which is a unified detection method that localizes an object in an image using regression, writting its result using a bounding box. This algorithm proposed a solution to solve the object detection problem using a single neural network to predict the location as well as classification of the object and it was chosen for its need to look at the image region only once unlike other approaches which need to check the same region several thousand times over. The overall process of this algorithm can be seen in figure 6.

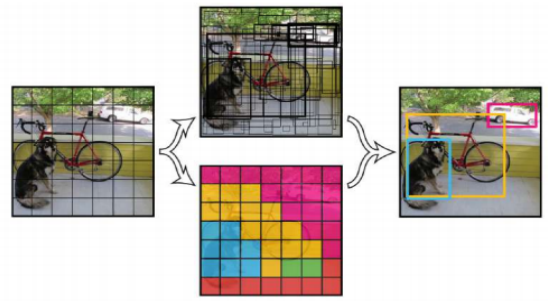

Fig. 6: YOLO structure design showing the grid, bouding boxes, confidence and class probabilities used in its implementation

Using the information provided by this algorithm, it is possible to create a map for the trajectory based on a sequence of dots and these results can be seen in figure 7.

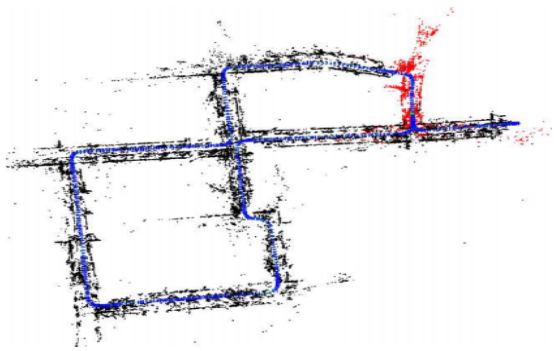

Fig. 7: Map created and visualized using VSLAM with a sequence of points. The blue frames indicate the estimated camera poses for the keyframes. The map points are visualized with the black and red dots. The red points belong to the current local map.

3.2 Related Work based on Pattern Recognition Algorithms

Pattern recognition algorithms are used for ruling out unusual or sometimes irrelevant data points from the images obtained by the sensors which can possess all types of environmental data. These algorithms help in reducing the dataset by detecting object edges by fitting in line segments and circular arcs to each one of the edges of the object, and these can then be combined in various ways to form the image features that are used for recognizing the object. The support vector machines (SVM) with histograms of oriented gradients (HOG) and principal component analysis (PCA) are some of the most common recognition algorithms used in autonomous driving systems but other algorithms such as Bayes decision and K nearest neighbor (KNN) algoritghms can also be used.
One example that uses this type of algorithm is Junior[22], as seen in figure 8, which was one of the autonomous driving platforms developed by the people over at Stanford University and its main goal was to perform pratical object recognition of specfic objects namely pedestrians, plants and other materials that the vehicle must interact with, as seen in figure 10. The laser-based object recognition(seen in figure 9) algorithm used on Junior can be broken down into three main components(segmentation, tracking and track classification) and was based on two methods: a supervised method and a semi-supervised method.

Fig. 8: Junior Autonomous Driving Platform

On the supervised method, the objects were segmented from the environment using depth perception, then tracked with a simple Kalman filter and the classification of these tracked objects was achieved with a boosting algorithm that was appiled accross several high dimensional descriptor spaces which encoded the size, shape and other motion properties of the object. The semi-supervised method used model-free segmentation and tracking which enabled a highly effective

method of learning objects models without the need for massive quantities of hand-labeled data. This method iteratively learned a classifier and collected new useful training instances by using the tracking information and these training instances can be seen in figure 11.

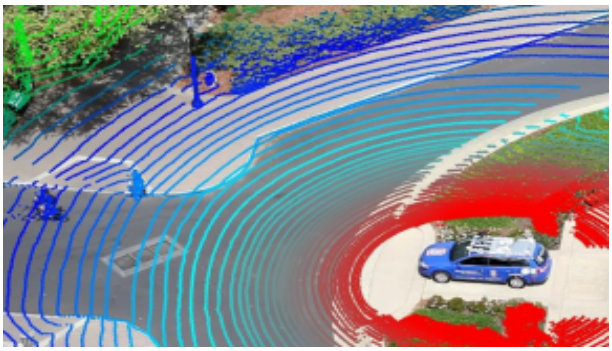

Fig. 9: Junior Laser-Based Object Recognition

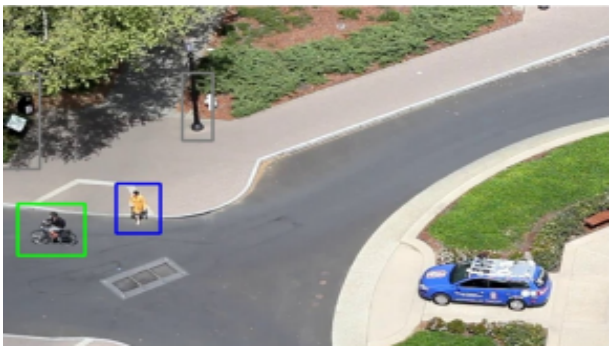

Fig. 10: Junior Specific Object Recognition

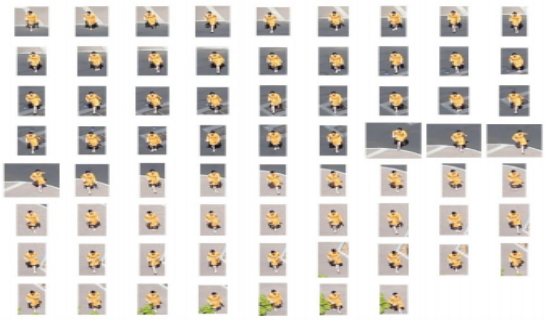

Fig. 11: Junior Labelled Training Data for the vision-based classifier to learn from.

3.3 Related Work based on Clustering Algorithms

The clustering algorithms are good for discovering the data structure from the data points since sometimes the images obtained by the system are not that clear and it is difficult to detect and locate objects. The clustering methods are typically organized by modelling approaches such as centroid-based and hierarchical and all of these methods are concerned with using the inherent structures in the data to best organize the data into groups of maximum commonality. The K-Means and the Multi-class Neural Network are some of the most common algorithms used in autonomous driving systems.
This type of algorithm can also be used for cluster driving encounter scenarios using connected with multiple vehicle trajectories in which the clustering algorithm is applied to extract representations and gather homogeneous driving encounters into groups. Some of these driving encounters are detailed in figure 12.[23] Two vehicle trajectories with the same length represent a driving encounter, where the vehicle trajectories could be continuous or discrete.

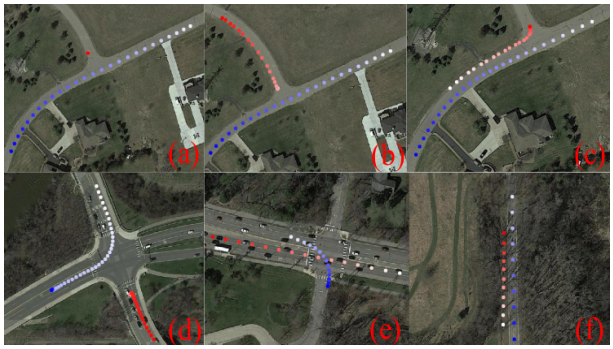

Fig. 12: Examples of driving encounters at T-shape intesections, cross-intersections and straight roads

Clustering similar driving encounters into groups requires to extract the features capable of capturing their primary characteristics. Unlike extracing features of images with specific and explicit labels, there is limited prior knowledge about the feature space of

driving encounters. Fortunely, many other approaches to learning representations of time-series trajectories have been developed and based on the results of these approaches, two methods to achieve multi-vehicle dynamic interactions in spatial and temporal spaces have been developed using representation learning: deep autoencoders, distance-based measure.

The deep autoencoders are used to generate samples from the learned representation that are as close as possible to the original input. Each one of these autoencoders, corresponds to a specific neural network (NN) consisted of an encoder and a decoder based on representation learning, as shown in figure 13, that attemmpts to copy the input value `x` to its output value $\tilde{x}$ in which the hidden layer `h` will describe the code to represent the input value `x`. By training this model to minimize the difference between `x` and $\tilde{x}$ it is possible for the hidden layer `h` to capture the underlying characteristics of the driving encounter.

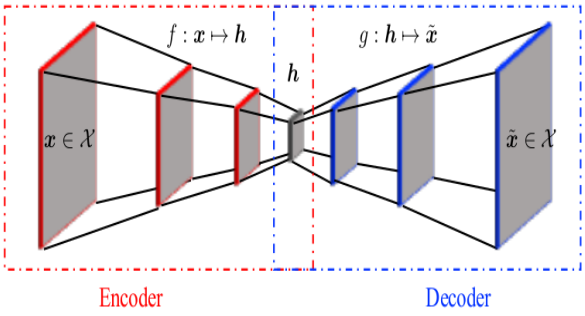

Fig. 13: Illustration of representation learning through the deep neural network-based autoencoder, with an encoder f and a decoder g to learn hidden representations h given observations x $\in X$.

The distance-based measures treat the representation learning process with a mathematically rigorous way to capture the spatial relationship between two vehicles in a driving encounter instead of treating the process as a black box. In order to achieve this, a measure that can be used to gauge the geometrical distance needed to be defined and for that two methods were used to achieve to calculate this measure. One of the methods was based on `Dynamic Time Warping(DTW)`, in which given a driving encounter observation `x`, this algorithm would measure the geometric relationship of the two-vehicle trajectories over time by looking at the length of the unified driving encounters and the local distance of the positional point of one vehicle to another positional point of the other vehicle and this distance was calculated by first calculating the Euclidean distance between these two points. The other method was based on `Normalized Euclidean Distance(NED)` which calculates the Euclidean distance directly in order to ensure the distance of temporal-pairwise positions of two vehicles at a given time and save the results in a distance vector `f` which can later be used to describe the relationship of the two vehicles. After that the results were saved in a distance vector, by applying this formula:

$$F = \frac{f}{max(f)}$$

in which `F` corresponds to a feature of the driving encounter.

In summary, the deep network-based autoencoders and the distance-based measure can be used to learn representations of driving encounters. Besides, the autoencoders can be used to capture underlying information through dimension reduction and hence they can potentially extract meaningful representations from the results of `DTW` or `NED`. In terms of clustering, the `k-means` clustering algorithm was chosen for its simplicity and scalability. During the clustering procedure, a criterion was needed in order to evaluate the clustering performance and consequently select an appropriate cluster number. The aim of this algorithm was to gather the different driving encounters with similar characteristics into one group that is different from the other groups and hence improve the quality of the clustering results that can be assessed by checking the between and within cluster variances of the obtained clusters which for the case of the elements involved in the same groups should be as small as possible. However, directly computing the mean of driving encounters proved to be nearly

impossible since it is inconceivable and even meaningless to calculate the mean of all trajectories in driving encounters. Therefore, instead of using multi-vehicle trajectories to evaluate the variances, the mean of the extracted representation was calculated by averaging all extracted representations of driving encounters that belong to a certain cluster. The variances between clusters(BC) and within clusters(WC) were computed based on these two formulas:

$$BC = \frac{1}{J-1}\sum_{j=1}^{J} D(\tilde{F}x, \tilde{F}x_j)$$

$$WC = \frac{1}{N-J}\sum_{j=1}^{J}\sum_{i}^{|Xj|}\frac{1}{|Xj|}D(Fx_{ji}, \tilde{F}x_j)$$

in which `D(.,.)` is the Euclidean Distance measure, `|Xj|` is the number of driving encounters in a cluster `Xj`, `Fx` is the feature representation of a certain driving encounter in a certain cluster and $\tilde{F}x$ corresponds to the mean value of the feature representations of all driving encounters in a certain cluster. The main goal of this algorithm is to maximize the variance between clusters and minimize the variance within clusters and make a performance comparison between the two by normalizing these variances using these formulas:

$$\lambda_{BC} = \frac{BC}{WC + BC}$$

$$\lambda_{WC} = \frac{WC}{WC + BC}$$

This way, the larger the value of $\boldsymbol{\lambda_{BC}}$ means the larger distance between clusters will be and the larger the value of $\boldsymbol{\lambda_{WC}}$ means the larger distance within clusters will be.
Using GPS data visualization the clustering results should look something like what is represented in figure 14 and figure 15.

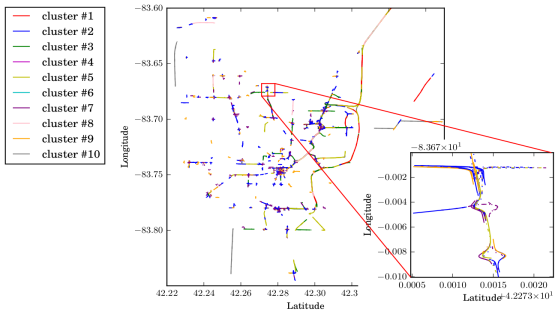

Fig. 14: GPS Data Visualization of clustering results using DTW and k-means clustering algorithm with k=10.

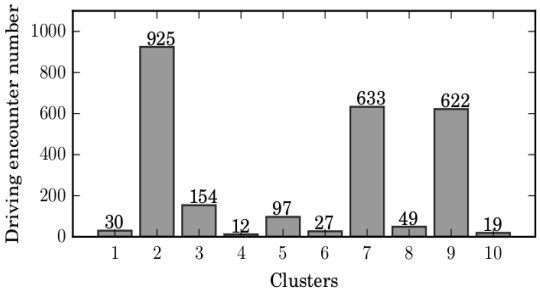

Fig. 15: Bar Chart of clustering results of using DTW and k-means clustering algorithm with k=10.

3.4 Related Work based on Decision Matrix Algorithms

The Decision Matrix Algorithms are often used for systematically identify, analyze and rate the perfornamce of relationships between sets of values and information and they are mainly used for decision making. These algorithms are models composed of multiple decision models independently trained and whose predictions are combined in some way to make the overall prediction while reducing the possibility of errors in decision making. The most commonly used algorithms are the gradient boosting (GDM) and the AdaBoosting algorithms.
Decision-making for autonomous driving is challenging due to the uncertainty on the continuous state of nearby vehicles and especially over their potential discrete intentions such as turning at an intersection or changing lanes, as seen in figure 16, and

for that reason multipolicy decision-making proved to be extremelly important when it comes to autonomous driving.[16]
The people over at the Institute of Robotics and Intelligent Systems ETH Zurich, developed an integrated inference and decision-making approach for autnomous driving based on changepoint-based behavior prediction that would model the vehicle behavior for both their vehicles and the nearby vehicles as a discrete set of closed-loop policies. In this case, each policy captured a distinct high-level values for behavior and intention of certain actions such as driving along a lane or turning at an intersection. By employing Bayesian changepoint detection on the observed history of nearby vehicles it was possible to estimate the distribution over potential policies that each nearby vehicle could be executing. After this estimation, it is possible to sample the policy assignments from these distributions in order to obtain the high-likelihood actions for each participating vehicle and perform closed-loop forward simulation in order to predict the outcome for each sampled policy assignment. After evaluating each one of these predicted outcomes, the policy with the maximum expected reward value is then executed and based on these results it is possible to validate behavioral prediction and decision-making using simulated and real-world experiments.
A system diagram describing the overall design of this algorithm is presented in figure 17, but to put it in a simple perspective, the way this system works is that it takes an input route to the user's desired destination and perceptual data, such as localization and dynamic object tracks, as input values. The model then outputs a series of low-level commands, such as forward speed and steering wheel angle, to the vehicle based on a module that performs behavioral prediction and anomaly detection and a policy selection algorithm.

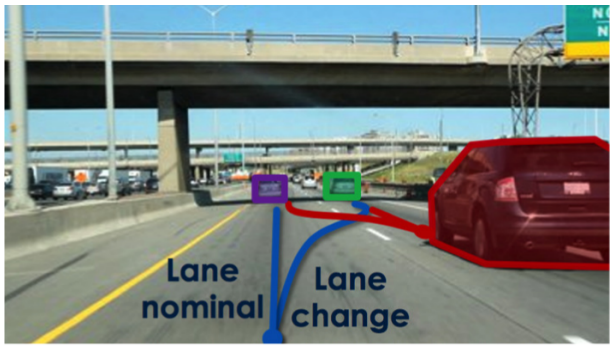

Fig. 16: Autonomous Driving Change Lane Policy

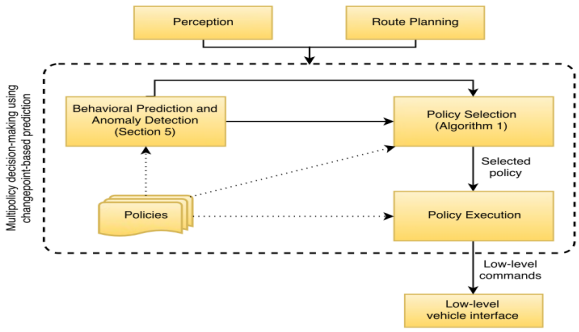

Fig. 17: Multipolicy decision-making via changepoint-based prediction system diagram.

3.5 Related Work on Autonomous Driving Simulators

The use of autonomous driving simulators is becoming more and more important to test out the learning algorithms that are going to be implemented in the autonomous vehicle. One of these simulator is CARLA(described in section 2) that implements an autonomous driving method that was implemented based on three approaches to autonomous driving. The first one being a modular pipeline that relies on dedicated subsystems for visual perception, planning and control. The second one being a deep network trained end-to-end via imitation learning and third one being another deep network but this one was trained end-to-end via reinforcement learning.[14]
The modular pipeline approach decomposed the driving task among 3 subsystems: perception, planning and control.
The perception stack uses semantic segmentation to estimate lanes, road limits, dynamic

objects and other hazards using a semantic segmentation network based on RefineNet.[19]In addition to this, a classification model was used to determine the proximity to intersections. The local planning subsystem used a rule-based state machine that implements simple predefined polices tuned for urban environments and the continuous control module was based on a PID controller that was used to control the vehicle(steering, throttle and brake).

The second approach was based in conditional imitation learning, which is a form of imitation learning that uses high-level commands in addition to perceptual input data. The input dataset consisted in a list of tuples, each of which containing an observation, a command and an action and the high-level commands used were correlated to actions such as following the lane, driving straight at an intersection and turning left or right at the next intersections. The observations were images that were picked from the front facing camera. This method amounted to around 14 hours of driving data for training. This network was trained using an optimizer and procedures such as data augmentation and dropout were used to improve its generalization.

The deep reinforcement learning was trained using a deep network based on a reward signal provided by the simulated environment with no human driving traces and it used the `Asynchronous Advantage Actor-Critic(A3C)` algorithm. This algorithm had already been shown to perform well in simulated three-dimensional environments such as racing and navigation in three-dimensional mazes.[20] This algorithm was used for its ability to run multiple simulation threads in parallel, which was important given the high sample complexity of deep reinforcement learning.

This model was trained on goal-directed navigation, in which in each training episode the vehicle had to reach a certain goal guided with high-level commands and each one of these episodes was terminated when the vehicle reached the goal, or when the vehicle collided with an obstacle or simply when the time budget was expired. This network was trained with 10 parallel actor threads for a total of 10 million simulation steps and this process took up to roughly 12 days of non-stop driving at 10 frames per second.

Each one of these approaches was evaluated using a series of tasks and at the end it was found out that none of the methods performs perfectly even on the simplest of tasks such as driving straight. The modular pipeline performance was very close to the performance of the end-to-end learning methods and the agent trained with reinforcement learning was proven to perform significantly worse than the one trained with imitation learning in all tasks despite being trained with larger amounts of data. The results for each one of the experiments can be seen in table of figure 18.

	Training conditions			New town			New weather			New town & weather		
Task	MP	IL	RL	MP	IL	RL	MP	IL	RL	MP	IL	RL
Straight	**98**	95	89	92	**97**	74	**100**	98	86	50	**80**	68
One turn	82	**89**	34	**61**	59	12	**95**	90	16	**50**	48	20
Navigation	80	**86**	14	24	**40**	3	**94**	84	2	**47**	44	6
Nav. dynamic	77	**83**	7	24	**38**	2	**89**	82	2	**44**	42	4

Fig. 18: Quantitative evaluation results for the three autonomous driving system approaches in CARLA.

Other autonomous driving simulators make use of other approaches to autonomous driving with one of them being deep learning. One example that implements this approach is the `COGNATA`[6] simulator which uses state-of-the-art deep learning algorithms for its simulation engine and leverages reality-grade city mesh combined with `DNN (Deep Neural Network)` and AI capabilities as seen in figure 19. This system is based on three layers of different deep-learning algorithms in which the first two lauers make up the objects in the surrounding environment as well as other elements like cars, time of day and weather conditions and the third layer is a sensing layer which behaves as if the car is powered by real-life sensors.[5]

Fig. 19: COGNATA reality-grade city mesh combined with DNN and AI capabilities.

3.6 Overview

Autonomous vehicles are always very closely associated with industrial IoT technology and machine learning algorithms implemented for artificial intelligence.[7]
There are many approaches to implement autonomous driving methods but almost every one involves machine learning algorithms in which the model is trained to perform various actions in a set environment. However, it is important to note that these driving methods are not perfect and they can easily underperform under certain conditions which sometimes can have drastic consequences. Therefore these models have to be subjected and trained in different conditions each time they are to be implemented in an autonomous vehicle. Some of these models need more time to train than others depending on the complexity of the problem and the desired objectives and they need to be best tested several times in simulated environments before testing the results in a real world scenario and that is why the use of autonomous driving simulators is so important in the implementation of autonomous driving systems. When it comes to autonomous driving, tasks such as obstacle avoidance are of utmost importance and many autonomous driving research teams developed and trained their algorithms based on this task. However, due to the complexity of this task, the algorithms can sometimes underperform in certain conditions and because of that they need more training under those conditions in order to learn and improve their performance.

4 Data Description

In autonomous driving, obstacle avoindance is very important for training the model to understand their own spatial boundaries and not impact with other obstacles which can have very negative consequences to the growth of the model. This is a very complex problem for machine learning because it involves many factors such as the number of obstacles present in the world, the size of these obstacles and the distance of the incoming obstacles when compared to the vehicle, all of which can increase the complexity of the problem immensely. These are factors that can greatly affect the performance of the model to learn and adapt in certain situations and because of this we considered this problem as the focus point for this project.
In order to implement a solution to this problem, we decided to handle this approach as a 2D game instead of the typical 3D scenario mainly to reduce the computational resources needed to implement the simulation. The game was designed using a `Pygame` window and the main goal of this was move the vehicle around the map without hitting any of the obstacles. Every model used in this pygame is based on circle shapes from the pygame module and these shapes were chosen for the vehicle to move around more easily within the map. The green circle represents the vehicle model, the blue circles represents the obstacles and the orange represents another vehicle that represents another obstacle for the vehicle model to avoid. The shapes can be seen in figure 20.

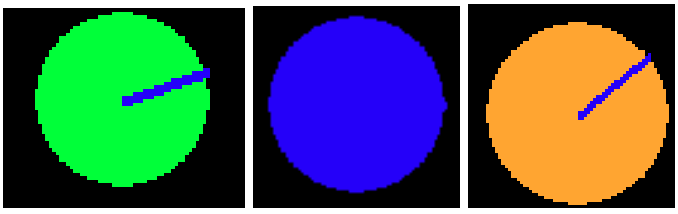

Fig. 20: Pygame Shapes

The number of obstacles chosen for this simulator was 3, each one with different sizes moving around in random positions at random speeds and this factor was chosen to prevent problems for the model such as over fitting.

The different sizes of the obstacles can be seen in figure 21.

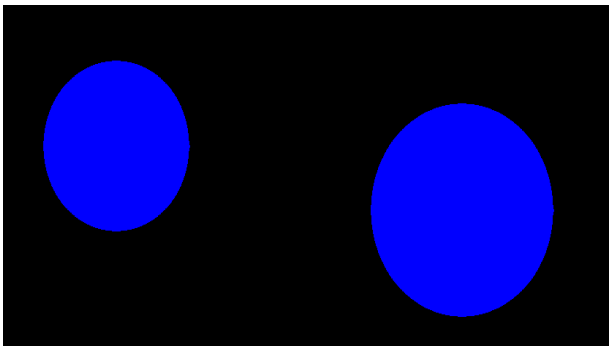

Fig. 21: Pygame Obstacles with Different Sizes.

The vehicle model is created with 3 sensors in which each sensor gives distance readings about incoming obstacles in relation to the vehicle. The way these readings works is almost as if reaching some object with an ”arm” in which this ”arm” returns a distance between you and that object. With this method it is possible to look at each point and increment the distance counter until we hit a certain obstacle. Based on these readings the vehicle can see where the obstacles are in the map and act accordingly. Using these sensor readings, the vehicle can also check if it crashed or not and if he crashed, he can then recover from said crash by driving backwards.
The vehicle model with its sensor readings can be seen in figure 22.

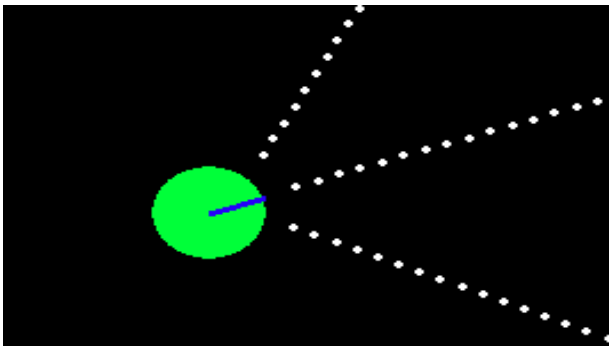

Fig. 22: Vehicle Model with Sensor Readings.

The way the game work is that in each frame, the vehicle will take one of two actions(turn left or turn right) and based on the number of steps taken, certain events such as moving of the obstacles start to occur in the game.
The driving direction is calculated based on the rotated angle of the vehicle and based on that direction the velocity of the vehicle is calculated. By calculating the current position and the sensor readings there it is possible to check the vehicle state and based on that state give rewards or penalizations to the vehicle. The way this reward system works is that as long as the vehicle does not crash, the reward count will increase but if the vehicle crashes the reward count will decrease exponentially. The full model of the pygame can be seen in figure 23 and all properties of this game can be seen in the `car_model.py` Python script.

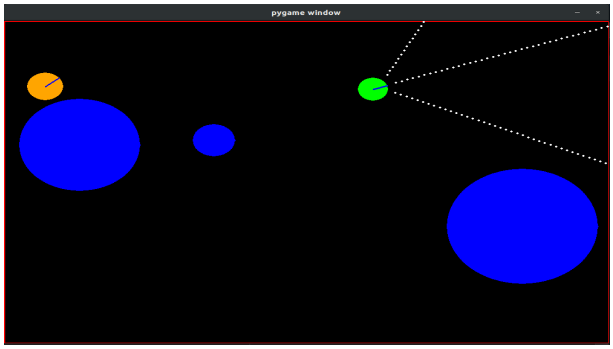

Fig. 23: Pygame Full Model.

With this pygame it is possible to test out the model and implement a machine learning algorithm for it to learn to avoid obstacles and act accordingly. The input data used to feed this algorithm will be input frames from the pygame and the information given by the sensor readings and based on this the algorithm will either get rewarded or penalized according to the vehicle’s behavior.

5 Machine Learning

In order for the vehicle to learn how to move around the screen (drive itself) without running into obstacles a `Q-learning (unsupervised)` algorithm based on `reinforcement learning` was used.
The typical machine learning approach is to train the model from scratch and as such give it a million images and some time to figure it out how to behave. The initial

approach to the problem was firstly based on `imitation learning` in which we train a policy to make decisions using demonstrations. This model is by-and-large a version of `supervised learning` with the biggest difference being that the predictions are made sequentially. However, this model needs a few key ingredients in order to train properly such as pre-collected demonstrations which are used as an oracle for the model to query; a transition model `P(s'|a,s)`, where `s` and `s'` denote states and `a` denotes an action, which is used to give information on how the environment evolves in response to the policy's actions; a policy class which constitutes a class of policies that the model should learn over; a loss function which quantifies the mismatch between the action taken by the policy in a given state versus the action taken by the pre-collected demonstration and finally and a learning algorithm which is used to minimize the loss and optimize the results for the problem. Putting it all together, it is possible to consider imitation learning as a way to learn how to make sequences of decisions in an environment where the training signal comes from the pre-collected demonstrations. The overall process involved in this learning algorithm is depicted in figure 24.[8]

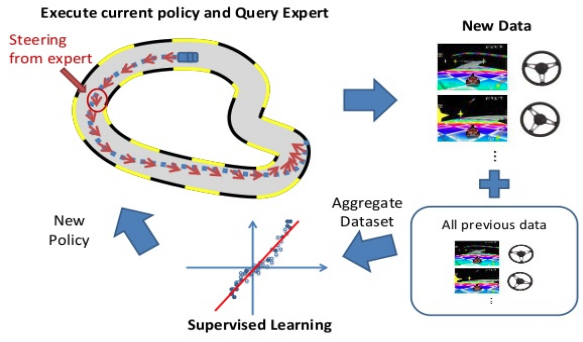

Fig. 24: Imitation Learning and Structured Prediction

Many people see `imitation learning` as a prefered way to `reinforcement learning` particularly in problems in which the reward function is difficult to specify (e.g., act "friendly") or problems in which the reward function is sparse and it is too dificult to be optimitzed directly. However, this method often requires a pre-trained network and pre-collected demonstrations in the environment which are not possible to achieve using this game since the objects in the world are spawned in random positions. Also this method needs to be supervised which can prove to be an exausting process for large amounts of data.
`Reinforcement learning` differs from this type of learning method in a way that in `supervised learning` the training data has the answer key with it so the model is trained with the correct answer itself whereas in `reinforcement learning`, there is no answer and the reinforcement agent decides what to do to perform the given task. Also in absence of a training dataset, this model can learn from its own experience since the output depends on the state of the current input and the next input depends on the output of the previous input.
Using this method it is possible to differentiate the behavior of the agent according to certain actions and give a reward score to the model as he completes those tasks. If the agent does well, his reward score will increase (positive reinforcement), otherwise a penalty, also known as, negative reward (negative reinforcement) is given to the model. Gradually, by trial and error phases, the agent will start to understand that it is better to avoid some actions and perform the actions which will bring him a higher reward. This process is detailed in figure 25[12].

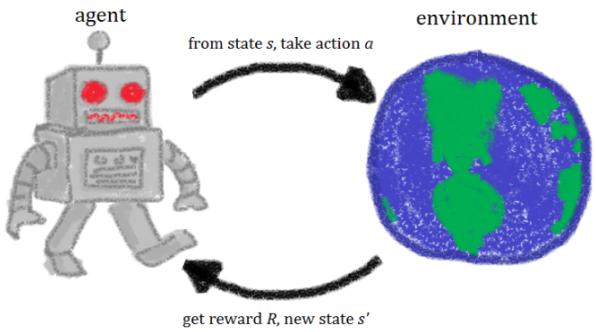

Fig. 25: Reinforcement Learning Setup

The framework described in figure 25 in which we have different stages:

$$observe \Rightarrow act \Rightarrow get(reward/nextstage)$$

and it can be described mathematically using a Markov Decision Process (MDP) [15]:

$$\langle S, A, R, T, \gamma \rangle$$

where:

- S: is the set of states,
- A: is the set of actions,
- R: is the reward function,

$$R(s, a) : S \times A \to \mathbb{R}$$

- T: is the transition function,

$$T(s, a, s') = p(s'|a, s)$$

- γ : is the discounting factor that is used to trade off the balance between the immediate reward and the future reward.

Having this MDP set, the next step is to define a policy function. The goal of this function is to return an action given the state as seen in following formula:

$$\pi(s) : S \to A$$

The final goal when solving this MDP is to learn the policy which maximizes the reward for our agent.
One example of these MDP can be seen in figure 26.

Reinforcement learning can be seen as a solution to problems in which the agents wants to find the policy which can maximize its reward while behaving in an environment. But one of the main problems with this style of learning is that it assumes that the world is Markovian when sometimes it isn't and it assumes that agents act alone in the presence of non-learning agents which is one of key problems in any OpenAI environment.
One way to help handle these problems is to use reinforcement learning in conjunction with a deep neural network and as such a solution was presented in the form of Deep Q-network (DQN).

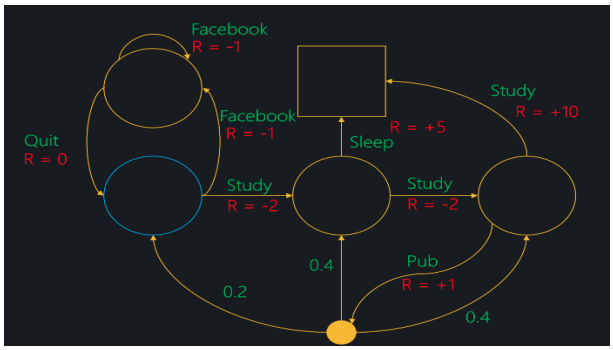

Fig. 26: Markov Decision Process Example with circles to represent the states, arrows with green labels to represent the actions, red labels to represent the rewards, squares to represent terminal states and green numeric labels to represent transition probabilities.

5.1 Deep Q-Network (DQN)

The whole deal with the deep q-network approach is built upon approximating the so-called Q function and building the agent's behavior based on that approximation. The idea behind this function is to return the entire expected discounted reward flow for the particular action and the particular state given the starting and next states following a certain policy π. Basically, this method provides an answer to the question how good is it to perform a certain action at a given state.
The Q function obeys the Bellman equation[3]:

$$Q_\pi(s, a) = \mathbb{E}_\pi[R(s, a) + \gamma . Q_\pi(s', a')]$$

and it follows the Bellman principle of optimality which states that notwithstanding what happened before, we should always take the action with the highest Q to maximize the reward flow:

$$Q_{\pi^*}(s, a) = \mathbb{E}_{s',r,a'}[R(s, a) + \gamma . max_{a'} . Q_{\pi^*}(s', a')]$$

Based on this formula, the Q value for the current step and action is equal to the maximum Q value for the next state (assuming that the agent is behaving optimally) plus the reward that we get for the transition. With this the value of the quadratic objective function becomes:

$$J_{DQ}(Q) = (R(s, a) + \gamma . max_{a'} . Q(s', a'; \theta') - Q(s, a; \theta))^2$$

The idea of using a DQN is to provide a feature called experience replay in which we make the transition and save it in a ”replay memory stack” which is an array that stores a large number of transitions with information about the reward, the states before and after the transition and information detailing if the event is terminal (game over) or not.
Based on the results from this network it is possible to implement a learning algorithm called Q-learning.

5.2 Q-Learning Algorithm

The Q-learning algorithm itself is not new[24] but the idea behind this algorithm is to use neural networks as function approximators based on neural fitted-q iteration (NFQ).[21] The NFQ is an algorithm used for efficient and effective training of a Q-value function represented in a multi-layer perceptron described in figure 27.

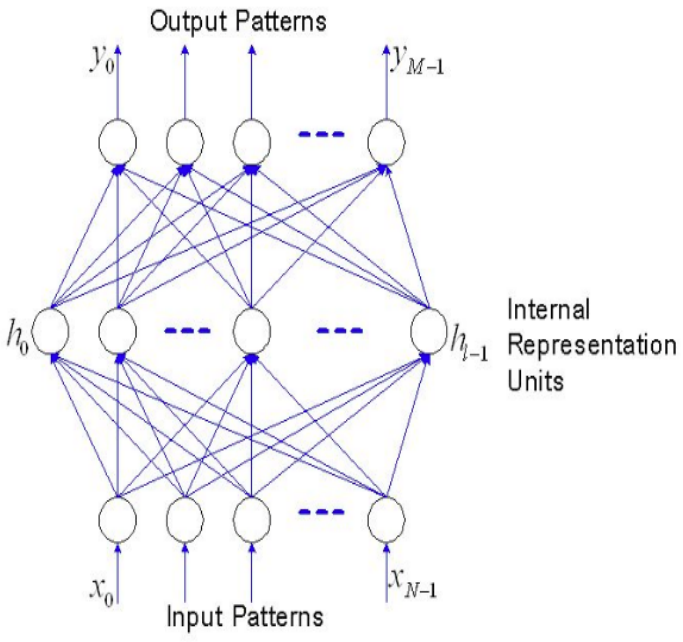

Fig. 27: Multi-Layer Perceptron Model Example

Based on the experience replay feature provided by the DQN network it is possible for the Q-learning algorithm to randomly sample mini-batches from it and learn more effectively. This presents some advantages when compared with other learning algorithms such as the potential to use each transition in several weight updates and the data to be used more efficiently; by randomly sampling it is possible to break the correlation between samples and reduce the variance between weight updates and another thing that makes this algorithm more stable when compared to other algorithms is the fact that it uses a DQN with two neural networks in which the first network is used to compute the Q value for the current state and the second network is used to compute the Q value for the next state. From this fact we get two different Q functions that use different $\boldsymbol{\theta}$ parameters for each state ($\boldsymbol{\theta}$ for the current state and $\boldsymbol{\theta}'$ for the next state). The $\boldsymbol{\theta}'$ parameteres are copied from the learned parameters $\boldsymbol{\theta}$ and this helps a lot in increasing the stability of the algorithm. It is not necessary to use two functions but the problem with using only one function is that when we try to update the weights of the model, both Q(s,a) and Q(s’,a’) functions increase and this might lead to problems such as oscillations or policy divergence. Using two separate functions, one for each network, adds a delay between the update and computation of the target Q value and reduces the oscillations and policy divergence cases.

5.3 Training Process

In order to implement the Q-learning algorithm into the pygame first we need to create a neural network for the pygame using the neural_network() constructor method in the neural_network.py file. This method takes as parameters the number of sensors of the vehicle and the nn parameters that define the number of neurons of the neural network.
This network follows a sequential model with 3 layers in which the first layer is a dense fully-connected layer with n hidden units and 3-dimensional vectors as the input data. The second layer is dense fully-connected layer witn n hidden units and it is used as an activation layer and uses a dropout rate of 0.2 which randomly sets a rate fraction of input units to 0 at each update during the training process

which helps prevent overfitting. Finally, the output layer is a dense fully-connected layer with 3 hidden units with linear activation.
After the network is created, we are free to train the model using this network and a bunch of parameters which include the batch size, a buffer and the nn parameters of the neural network. By running the `train_network()` method in `learn_model.py` file the training process can begin.
Before starting the training process, the model has to first observe 1000 frames from the pygame in order to know the features of the environment. In order to do this the model needs to check the game state, get the initial state from the vehicle object and start a timer. Once this is all set and done we can carry on with the training process.
For the purposes of this project, the training process will be played for 100000 frames from the pygame in a cicle. In this cicle, the vehicle object will choose a random action and get the Q values for each one of theese actions saving the best value in a variable. After this, the vehicle will take the action, observe the new state and get the reward score, saving the experience acquired in a replay storage. If the model is done observing the first 1000 frames the training process can begin and the first thing to do is to check if there is still space left in the replay storage buffer and if there is no more space we pop the oldest sample from the storage. Next we pick a random sample experience from the replay storage and use the `process_minibatch_train()` method to get the training values. This function takes the batch sample experience and the model as arguments and its goal is to feed the whole batch to the model which proved to be much more efficient when compared with other methods. This function reads the information present in the batch sample as (S, A, R, S') tuples in which S is the old state, A is the action, R is the reward and S' is the new state. Based on these values, the model can then predict the Q values for each one of the states, find the maximum Q value and based on that value update the value for the taken action. We then divide the data into two subsets the `x_train` which has the values from the old states and and `y_train` which has the value from the states with maximum Q value.
Using these two subsets, it is possible to train the model to fit the data the keras[9] model `fit()` function in which we record the loss history of the model by passing this history object as a list of callbacks which are functions that are applied at given stages of the training procedure and they are used to get a view on the internal states and statistics of model during the training procedure. After the loss history has been recorded, we then update the current state, decrement the epsilon value and if the vehicle has reached a terminal state we update the car distance parameters, print out the results and reset the values as seen in figure 28.

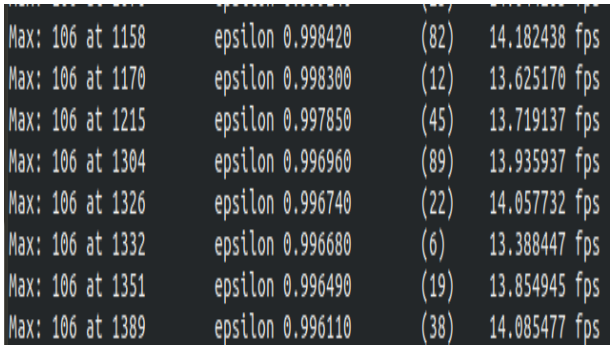

Fig. 28: Car Distance Parameters with values for maximum car distance, the frame number, the epsilon value, the current car distance and the number of fps.

This training model is supposed to train for 100000 frames but at every 25000 frames that this model runs, the weights of this model are saved in `.h5` file with a label detailing the nn parameters that were used and the number of runned frames. After the saving the process, an evaluation score is calculated using the keras model `evaluate()` function with the two subsets used for training along with the model and the loss history. This method will return a scalar test loss value which will be recorded in a log file. This data can be plotted using the `plot_evaluation.py` file. The other results such as the car distance parameters and the loss history are also recorded during the training process and these results are saved in `.csv`

data files which can later be plotted using the `plot_model.py` file.
All functions used in this process can be seen in `learn_model.py` file.

5.4 Results

Considering the size of the network 128x128 here are some of the results of the plotted data:

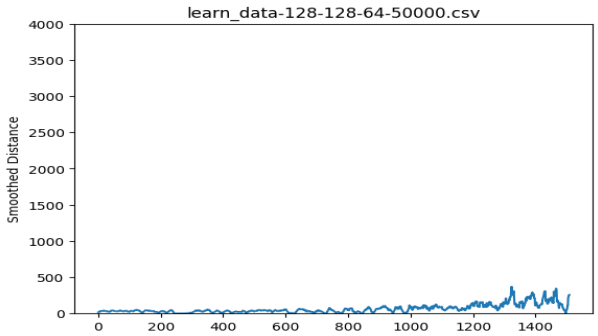

Fig. 29: Learned Data From Training with a Neural Network with size 128x128

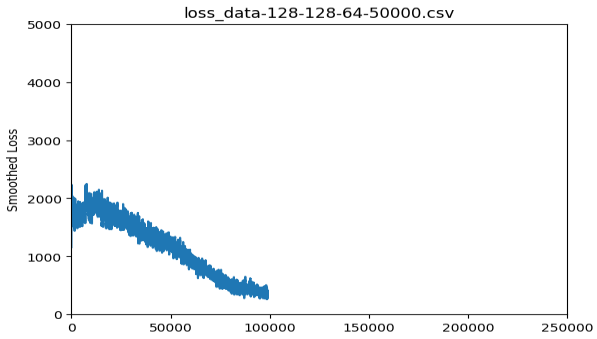

Fig. 30: Loss History Data From Training with a Neural Network with size 128x128

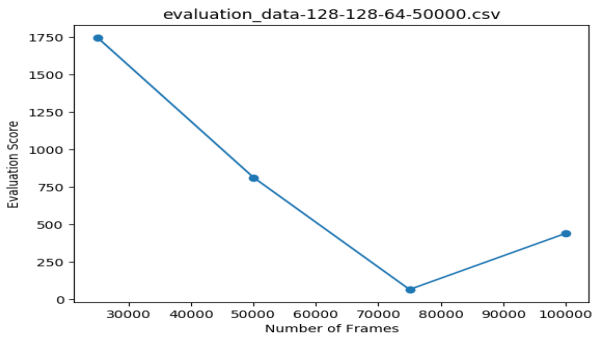

Fig. 31: Evaluation Score for the Training Model with a Neural Network with size 128x128

Based on these results it is possible to assume as the model learns more about the data the loss history values will decrease alongside the evaluation score for the training model.

5.5 Other Results

These are some of the results with different neural network sizes:

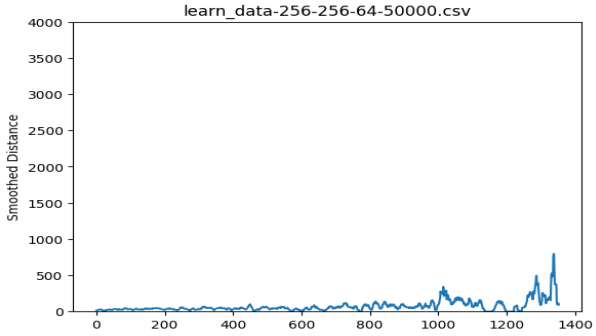

Fig. 32: Learned Data From Training with a Neural Network with size 256x256

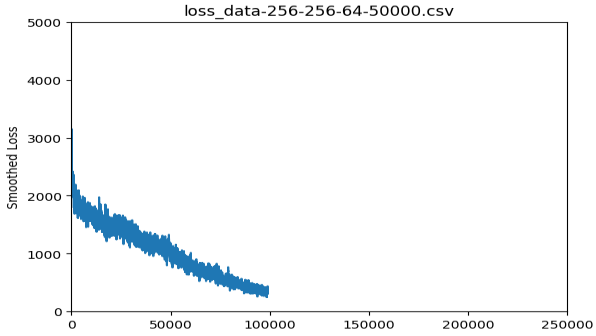

Fig. 33: Loss History Data From Training with a Neural Network with size 256x256

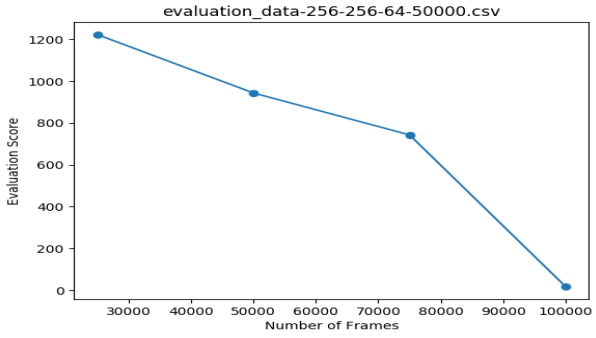

Fig. 34: Evaluation Score for the Training Model with a Neural Network with size 256x256

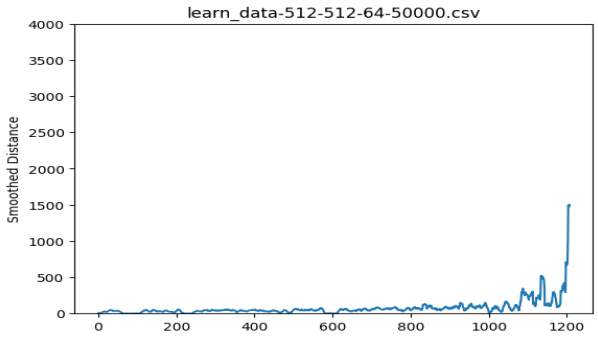

Fig. 35: Learned Data From Training with a Neural Network with size 512x512

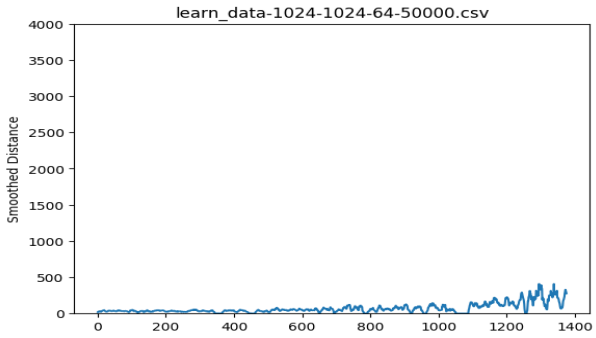

Fig. 38: Learned Data From Training with a Neural Network with size 1024x1024

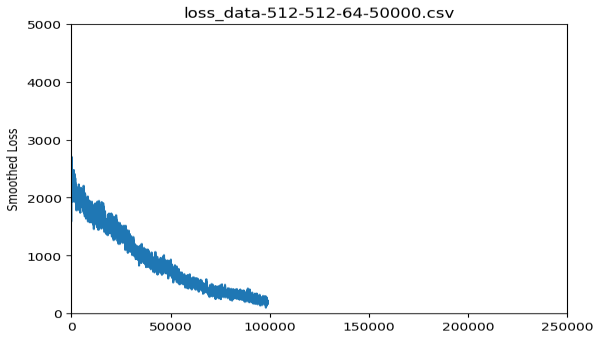

Fig. 36: Loss History Data From Training with a Neural Network with size 512x512

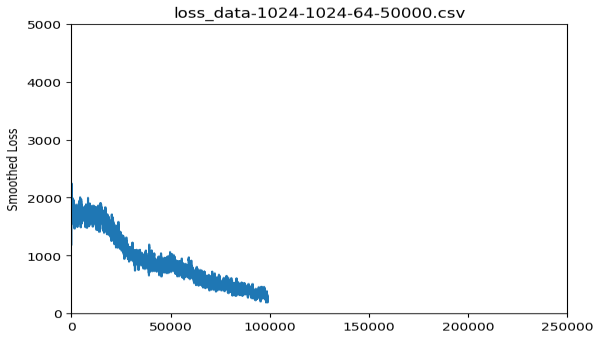

Fig. 39: Loss History Data From Training with a Neural Network with size 1024x1024

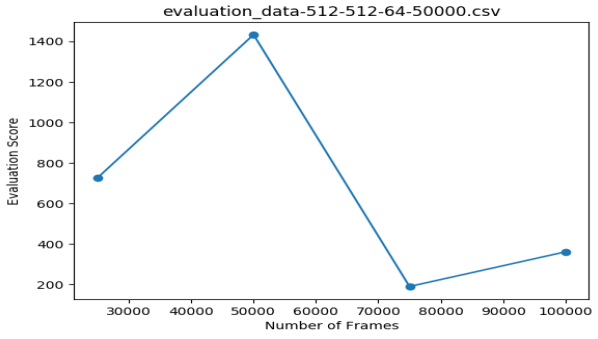

Fig. 37: Evaluation Score for the Training Model with a Neural Network with size 512x512

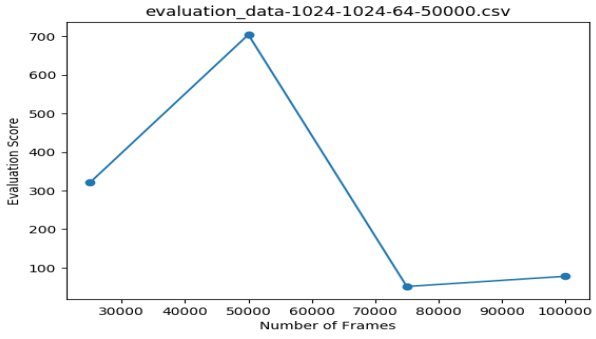

Fig. 40: Evaluation Score for the Training Model with a Neural Network with size 1024x1024

5.6 Results Discussion

It is important to note that training each one of these models takes a lot of time and a lot of data to train. It can take anywhere up from an hour to 36 hours to train the model depending on the complixity of the network and the size of the sample.
Arguably, the network size that showed better results was the `256x256` network size because it was one that showed the least amount of variances in the data. But this can also be a factor of the random properties of the pygame and the vehicle managed to learn better because the pygame allowed it. There a lot of factors to consider and a lot of tests need to be sorted before we can arrive at a conclusion for the best network size. Also we assumed simetric sizes `nxn` for the neural network in which the nn params are the same but this does not need to be the case. This alternative was not fully explored mainly due to time constraints and the fact that there are multiple variations for the neural network size that can be achieved this way.
We can also use more than 3 sensors on the vehicle but we opt with this number namely to prevent problems such as over fitting the data.
Converting the 2D pygame to a 3D scenario would also be a good idea since it would bring a new coordinate system in which we have to worry about other factors such as the width, height and depth of the objects but this would also drain the computational resources.
As for the models, each one of the weights were saved in a model in a `saved-models` folder and each one of these models can be played using the `play_model.py` python script. Then all that is left is to watch the vehicle drive itself around the world and see it avoid the obstacles.

6 Conclusions

Throughout the development of this project, it was possible to learn how a multitude of techniques work and match them together in a greater scale than merely lessons and tutorials can provide, with a greater motivation backing it up. It was all planned with a target use in mind, but the features developed can be used in other contexts. Having this said, the goals set for this project were achieved. The vehicle can correctly learn information about the world provided by the pygame and use that information to correctly avoid obstacles present in the world. It is important to note that regardless of the model that is being played, the vehicle can go to almost 100000 frames with barely hitting any obstacles unless one of these obstacles traps the vehicle in one of the corners of the pygame window.
There were of course some drawbacks to this procedure, the main one being the amount of time needed to train each one of these models and record the data. Another drawback was the inability to explore the alternative with the imitation learning algorithm.
It is also of course important to go over the whole social aspect of working in a team, even if only a two-man one, soft-skills become imperative, cooperation and understanding is crucial and something to always be worked on granted the opportunity, which this project has.

Appendix A
Acknowledgments

The authors of this report would like to thank the professor Petia Georgieva for her time, patience, and insight.

REFERENCES

[1] ATLAS atlascar hardware. http://atlas.web.ua.pt/hardwaresetup.html https://www.cars.com/articles/2018-volkswagen-atlas-car-seat-check-1420695923847/. Accessed: 2019-05-29.

[2] ATLAS atlascar onboard sensors. http://atlas.web.ua.pt/onboardsensors.html. Accessed: 2019-05-29.

[3] Bellman understanding rl : The bellman equations. https://joshgreaves.com/reinforcement-learning/understanding-rl-the-bellman-equations/. Accessed: 2019-05-29.

[4] CARLA open-source simulator for autonomous driving research. http://carla.org/. Accessed: 2019-05-29.

[5] Cognata Autonomous Driving Simulator audi partners with israeli startup cognata for autonomous car development. http://nocamels.com/2018/06/audi-cognata-autonomous-vehicles/. Accessed: 2019-05-29.

[6] Cognata Autonomous Driving Simulator deep learning autonomous and adas simulation. https://www.cognata.com/. Accessed: 2019-05-29.

[7] IIoT World machine learning algorithms in autonomous driving. https://iiot-world.com/machine-learning/machine-learning-algorithms-in-autonomous-driving/. Accessed: 2019-05-29.

[8] IL introduction to imitation learning. https://blog.statsbot.co/introduction-to-imitation-learning-32334c3b1e7a https://www.quora.com/What-is-imitation-learning. Accessed: 2019-05-29.

[9] Keras keras documentation core layers. https://keras.io/layers/core/. Accessed: 2019-05-29.

[10] Multi-objective Mapping and Path Planning unsing Visual SLAM and Object Detection by Ami Woo master thesis in applied science in mechanical and mechatronics engineering waterloo, ontario, canada, 2019. https://uwspace.uwaterloo.ca/bitstream/handle/10012/14386/Woo_Ami.pdf?sequence=3&isAllowed=y. Accessed: 2019-05-29.

[11] PYGAME pygame front page documentation. https://www.pygame.org/docs/. Accessed: 2019-05-29.

[12] RL human level control through deep reinforcement learning. https://www.nature.com/articles/nature14236?foxtrotcallback=true. Accessed: 2019-05-29.

[13] The KITTI VIsion Benchmark Suite (2019) a project of karlsruhe institute of technology and toyota technological institute at chicago. http://www.cvlibs.net/datasets/kitti/. Accessed: 2019-05-29.

[14] Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. Carla: An open urban driving simulator. 11 2017.

[15] J Frédéric Bonnans. *Markov Decision Processes*, pages 223–266. 04 2019.

[16] Enric Galceran, Alexander Cunningham, Ryan Eustice, and Edwin Olson. Multipolicy decision-making for autonomous driving via changepoint-based behavior prediction. 07 2015.

[17] Andreas Geiger. Are we ready for autonomous driving? the kitti vision benchmark suite. pages 3354–3361, 06 2012.

[18] Andreas Geiger, P Lenz, Christoph Stiller, and Raquel Urtasun. Vision meets robotics: the kitti dataset. *The International Journal of Robotics Research*, 32:1231–1237, 09 2013.

[19] Guosheng Lin, Anton Milan, Chunhua Shen, and Ian Reid. Refinenet: Multi-path refinement networks for high-resolution semantic segmentation. pages 5168–5177, 07 2017.

[20] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. 02 2016.

[21] Martin Riedmiller. Neural fitted q iteration – first experiences with a data efficient neural reinforcement learning method. volume 3720, pages 317–328, 10 2005.

[22] Alex Teichman and Sebastian Thrun. Practical object recognition in autonomous driving and beyond. pages 35–38, 10 2011.

[23] Wenshuo Wang, Aditya Ramesh, and Ding Zhao. Clustering of driving scenarios using connected vehicle datasets. 07 2018.

[24] Christopher Watkins and Peter Dayan. Technical note: Q-learning. *Machine Learning*, 8:279–292, 05 1992.