Finding better learning algorithms for self-driving cars

An overview of the LAOP platform

Jihene REZGUI, Clément BISAILLON, Léonard OEST O'LEARY
Laboratoire Recherche Informatique Maisonneuve (LRIMa)
Montreal, Canada
jrezgui@cmaisonneuve.qc.ca

Abstract— **Cars are becoming more and more intelligent, embedded with a range of sensors to give them local perception of their environment (LIDARs, cameras, etc.). Trendy companies like Google and Tesla are actively testing cars on American roads that can drive without any human interaction [1]. Neural networks are the modern approach for autonomous cars. However, an inefficient neural network algorithm will make the learning process slower and will result in a less reliable autonomous vehicle. In this paper, we will introduce a platform built in JAVA named LAOP (Learning Algorithm Optimization Platform) [2] while explaining the solutions we found to make it easy for researchers to test and compare their own algorithms. Then, we will show how we have integrated a natural selection algorithm with a neural network in order to improve them. Moreover, we will demonstrate how the Fully Connected Neural Network and the NEAT [3] (NeuroEvolution of Augmenting Topologies) algorithms are implemented in the context of vehicular learning on LAOP. Finally, we will display the different results extracted from LAOP by tuning several various parameters such as the weight mutation chance and the car density in the simulation.**

Keywords— Neural networks; evolution; self-driving cars; natural selection; platform.

I. INTRODUCTION

Autonomous vehicles could solve most current world problems regarding cars. Driven by Artificial Neural Networks (ANN), these cars will reduce the amount of traffic and accidents related to human error. For this reason, companies like Google and Tesla are actively conducting researches and innovating in this field. However, research in vehicular neural networks seems to be slowed because there is no centralised method to easily develop and compare new algorithms. With a uniform method to compare them, the research aiming to find better learning algorithms would be more efficient. To the best to our knowledge, we believe to be the first to develop an open-source environment, called LOAP, that lets other people implement their algorithms and easily compare them with others in a vehicular environment that simulates real world conditions.

Our contributions can be summarized as follows: (1) We introduce an open-source environment named LAOP where researchers can easily develop, test and compare their learning algorithms; (2) we propose a Genetic **N**atural **S**election algorithm, called GNS* to improve the learning experience of the algorithms; (3) We implement several algorithms on LAOP such as the NEAT algorithm and the Fully Connected Algorithm so that researchers can start to compare their algorithms right away and (4) We demonstrate through extensive simulation the results we obtained by tuning some parameters, like the weight mutation chance, which could improve the learning experience. For the purpose of this paper, the weight mutation chance is defined as the chance that the connection's weight from the neural network changes randomly.

The second part of the paper provides a brief overview of related work and compares them with our platform. In section III, the platform will be examined by explaining the simulation process and how the cars are learning to avoid obstacles. In section IV, we showcase the algorithms added to the platform so that researchers can start working immediately by comparing their algorithms to ours. Section V shows the results we got by simulating the Fully Connected algorithm for two scenarios. Finally, conclusions are drawn in Section VI.

II. COMPARISON WITH OTHER TOOLS

Other tools already exist to simulate a car driving in an environment. For example, CARLA [4] proposes an open-source driving simulator that allows people to develop programs on top of it. It is used by people learning to drive and to validate autonomous systems. The force of CARLA resides in its alikeness to the real world. However, the problem with this simulator is the difficulty to compare multiple algorithms. Our proposed platform LAOP provides a quick comparison between several algorithms according to different parameters.

Another tool like our platform is described in [5] proposed by S. Arzt. This project demonstrates how to implement an ANN coupled with algorithms of natural selection. As in our LAOP platform, the author simulates cars and makes them learn to avoid walls using artificial networks algorithms. The problem with this tool is the difficulty to add your own algorithms and to improve them.

The pole balancing problem [6] can also be compared with our platform since both are used to test and improve learning algorithms. The problem with this technique is that is doesn't depict a real-world problem. LAOP offers to improve learning algorithms in the context of vehicular mobility and allows to test the algorithms with real settings.

III. Our proposed platform LAOP

LAOP is a tool that lets people build, test and compare their own learning algorithms in a stable environment in the context of vehicular learning. Our platform can be used to accelerate the development of algorithms and to try to improve the best performing ones. The platform is divided into three parts: (A) settings, (B) simulation and (C) learning.

A. The settings

It is important to be able to easily configure and compare multiple variations of the same algorithm, as the goal of the platform is to improve them. LAOP lets the user configure three types of settings: (1) simulation settings, (2) genetic settings and (3) algorithm settings. Simulation settings (1) are specific for each simulation batch and contain variables such as the car density and the number of sensors. Genetic settings (2) are similar to the simulation settings, as they are specific for a simulation batch. However, they are responsible for all the genetic variables, such as the chance of mutation and the chance of having a changed connection weight. The algorithm settings (3) are the ones created by the designer of an algorithm. They are easily accessible within the code. The platform lets the designer choose from the beginning the algorithms to compare and their settings. This way, the user can compare the same algorithms multiple times with those settings modified and find the settings that perform best. As an example, we tested multiple times the Fully Connected Algorithm with different mutation chances to try to find the optimal configuration.

B. The simulation batches

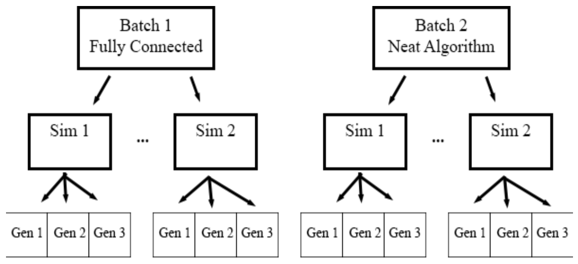

Fig. 1: A simulation batch is created for every compared algorithm (Fully Connected and Neat algorithms).

LAOP separates the update loop in three layers: (1) simulation batches, (2) simulations and (3) generations, as illustrated in Fig. 1. When the user runs one or multiple algorithms on the platform, a simulation batch (1) is created for every pair of learning algorithms and settings chosen. There could be two of the same learning algorithms with different settings in the two different batches. The number of generations in each simulation is determined by the simulation settings. Fig. 1 shows that the first batch runs simulations for the Fully Connected algorithm and the second batch runs simulations for the NEAT algorithm.

The role of the simulation batch (1) is to coordinate multiple simulations. It tells the simulations which algorithms and settings are required to initiate the cars. All simulations under a specific batch run the same settings and algorithms. The simulations (2) compute everything related to moving the cars according to the algorithm and changing its state when colliding with walls. The generations (3) are used to keep track of the average, the median, the highest and the lowest performing car. We use a function called *fitness function* to determine the performance of a car. A generation ends when all the cars hit an obstacle or when the time limit for a generation is reached. The information contained in the generations is saved to be analysed. The separation of the generation allows to easily store and retrieve the performance data and easily populate the graphs.

The different simulations let us run multiple times the same algorithms and settings. There can be errors related to randomness when dealing with neural networks and genetic algorithm. The same algorithms are tested multiple times with the same settings in order to have an average performance score, thus reducing the error related to chance. For example, one simulation might be performing better than another only because of the initial random values, and not because it is more efficient. Having multiple simulations solves this problem.

1) *The simulation and the environment*

A simulation contains two important elements: the environment and the cars. When a simulation runs, the cars are updated according to the environment and the simulation batch's algorithm (see section B.2 for more details).

The environment has also two important elements: the starting location and the obstacles. The starting point is where the cars spawns. The obstacles are lines that act like walls. If a car hits one of those, it is eliminated. An eliminated car is not updated anymore. When the generation ends, the data about all the cars get stored in the generation.

LAOP makes it easy for algorithm developers to create their own environment. The platform comes with a built-in map editor. It offers different tools to help create an environment with obstacles. The ability to create maps is useful to test how algorithms perform in a different context. For example, by testing multiple maps, we found that only relying on the maximum distance from the start is not the best option to compute the fitness score. In the future, the platform will be able to import a shapefile [8] containing real-world road information. This lets the algorithm test its capabilities in real-world environments.

2) *The car*

While a simulation is running, rectangles with different colors will move on the screen. These are the cars being updated by the simulation. This section describes how the cars retrieve information from their environment and then move through the environment.

The car has three components: (1) the proximity sensors, (2) the ANN and (3) the wheels. Each component is represented in Fig. 2.

The proximity sensors (1) let the car gather information about its environment. The user determines the number of car sensors in the settings. The proximity sensors provide the distance in a given line between the car and an obstacle. The result is then normalized to a value between 0 and 1. A value of 1 indicates that no wall is detected in the direction of the sensor; the more the value approaches 0, the more the obstacle is approaching the car.

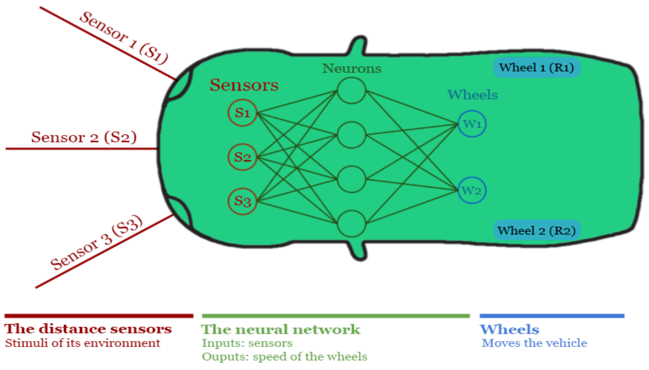

Fig. 2. A car with its different elements. The sensors give information to the ANN that determines the values of the wheels.

The wheels (3) control the direction in which the car will go. They can each receive a value between 0 and 1 and this will determine the force of the wheel. If the value received on the right wheel is greater than the one on the left, the car will go left, and vice versa. If the value of the two wheels are the same, the car will go in a straight line.

The ANN (2) makes the link between the sensors (1) and the wheels (3). It is the algorithm given by the simulation batch. It takes as inputs the values of the different sensors and outputs the values of the two back wheels. - in other words, on every update of the simulation, the car's ANN computes the speed at which each wheel should go based on the value of each of its sensors.

In the scenarios we considered, we only used proximity sensors. In the future, we plan on adding more types of sensors. According to the MEMS Journal [7], vehicles have between 60 and 100 sensors on board. This number is expected to increase to an average of 200 sensors. We could add a temperature, a light, a pressure, an acceleration and a speed sensor to make the car more alert of its environment.

*C. Making the cars learn with GNS**

When all the cars hit an obstacle or when the simulation time exceeds the time in the settings, the generation finishes. Then, an algorithm is used to modify the cars and make them learn.

To achieve this, we created the Genetic *Natural Selection** algorithm (GNS*). This algorithm was inspired by the concept of *natural selection*. It takes a set of cars as inputs and returns a modified one. The GNS* algorithm works in three phases: (1) the attribution of *fitness*; (2) the elimination process and (3) the repopulation process as displayed in the *Fig. 3*. This algorithm is used each time a generation is finished. Our platform lets the user easily change this algorithm in the code if desired.

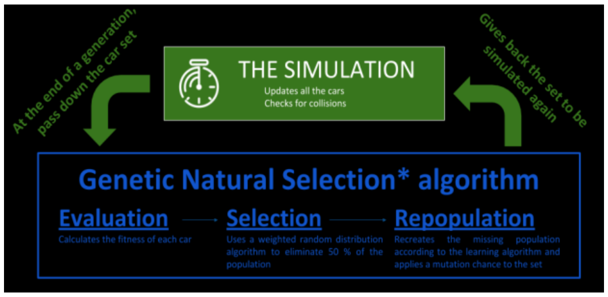
THE SIMULATION
Updates all the cars
Checks for collisions
At the end of a generation, pass down the car set
Gives back the set to be simulated again
Genetic Natural Selection* algorithm
Evaluation
Calculates the fitness of each car
Selection
Uses a weighted random distribution algorithm to eliminate 50 % of the population
Repopulation
Recreates the missing population according to the learning algorithm and applies a mutation chance to the set

Fig. 3: The process of simulating consists of three phases: (1) the evaluation, (2) the selection and (3) the repopulation.

1) *The fitness function*

To be able to make the difference between well-performing cars and the others, we use a function called the *fitness function* (Eq. 1). We deduced this formula through extensive simulations. It determines how well the car performed. The fitness function can be easily changed within our platform to best suit another algorithm. For our scenarios, we proposed this fitness function:

$$f = x + d \tag{1}$$

Where f is the fitness of the car, x is the maximum distance from the start the car (in pixels) and d is the total distance traveled by the car (also in pixels). The maximum distance is defined as the maximum distance between the start and the car in a straight line.

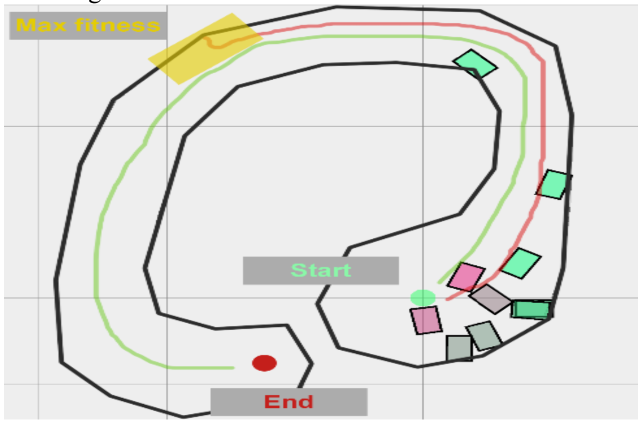

Fig. 4: If we rely only on the maximum distance from the start, the cars would get stuck in the yellow area since they would have a better fitness there than those reaching the end.

Our goal with this formula is to give a higher fitness to the cars going the furthest along the "path". If we only rely on $f = x$, the car will get stuck because it attains the maximum value of x when it reaches the furthest point from the map and not by going the furthest on the path. Figure 4 shows this problem. Represented with the red line is the path that the cars take if we only rely on the maximum distance from the start. In green is the path that the cars take if we add the total distance traveled to the equation.

2) *The elimination algorithm*

The goal of this algorithm is to eliminate the worst performing cars and keep the best ones. After the attribution of fitness, this algorithm uses a weighted random distribution algorithm to eliminate the worst performing cars. Our implementation of the weighted random distribution algorithm is as follows: we first sort the array of cars depending on their fitness score, then the algorithm gives each car a weight depending on its index in the sorted list of cars. It assigns a weight of 99.0 to the worst car and a weight of 0.5 to the best one. The weights of the cars in between are determined by a reverse exponential function. Then, the algorithm generates a random floating-point number between 1 and the total of all the weights. For each car, it removes its weight from the randomly chosen number and this car is removed from the array. This algorithm is repeated until the population size is half the size of the initial population. Fig. 5 shows that even if a car performs poorly in the simulation, it still has a chance to survive the elimination process.

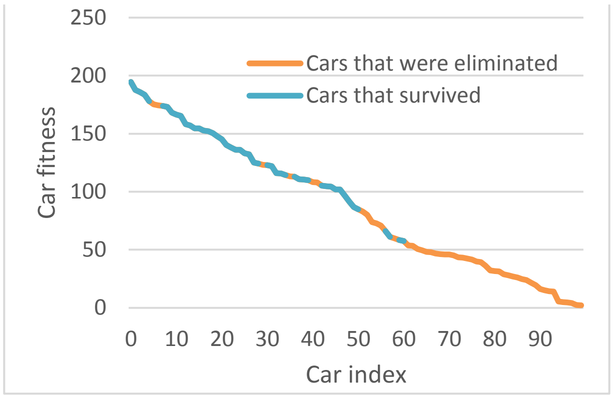

Fig 5. The distribution of the cars that survived the elimination process (orange) versus the cars before the elimination process (blue).

We use this algorithm instead of removing the worst half because even a less performing car might be useful for the learning process. It might have some traits that, when coupled with other cars, will result in a better performing neural network. However, the weighted random distribution allows us to have more chance to keep the best performing cars as shown in Fig. 5.

3) *The repopulation process*

When the elimination process ends, the GNS* algorithm populates the array with new cars having similar traits with the ones that survived. To create a new car from two parents, the GNS* algorithm is as follows: it first choses two cars randomly in the surviving cars population. Then, in the case of the Fully Connected Algorithm, a new neural network is created with the same topology of its parents. It iterates through each of its connections and it takes the weight from a randomly chosen parent at the same location in the topology. At the end of the reproduction, the child has weights coming randomly from both of its parents.

When the set of cars has been repopulated, the simulation then runs the newly created generation of cars. The process of simulation, selection and repopulation is repeated until the maximum number of generations specified in the settings is reached. The process is represented in Fig. 3.

IV. PROPOSED ALGORITHMS

The LAOP platform comes with two premade algorithms for the user to compare against. We offer a version of a fully connected neural network algorithm and an implementation of the NEAT algorithm proposed by Kenneth O. Stanley and Risto Miikkulainen [3]. We also made it possible and easy for developers to add their own algorithm to the LAOP platform. This section describes how we implemented the fully connected and the NEAT algorithm in our platform and how to add an algorithm.

A. The fully connected Algorithm

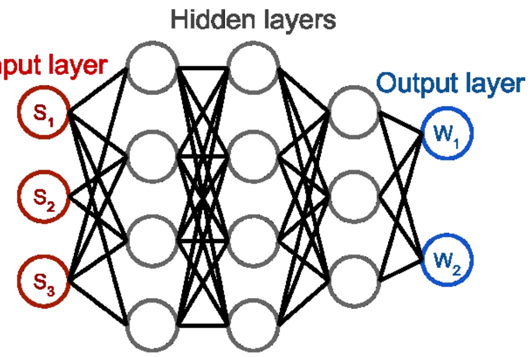

Fig. 6: A neural network contains multiple layers. The first one, in red, represents the nodes receiving the information from its environment. The last layer, in blue, gives it outputs to the car's wheels.

A fully connected neural network, as represented in Fig. 6, is a network starting with a fixed number of layers each containing a fixed number of neurons. In this type of neural network, each node is connected to every node in the next layer. To compute the value at a specific neuron, we do the sum of the value of each neuron (x_i) pointing to this specific neuron, multiplied by the weight of the connection (w_i) linking the two neurons. Then we normalise this value according to an *activation function* (φ). Here, we used the sigmoid function (3). The nodes in the first layer is not calculated, as they receive their input from the environment. In our case, the inputs were the value of each proximity sensors.

$$y = f(x, w) = \varphi\left(\sum_{i=1}^{m} w_i x_i\right) \qquad (2)$$

$$\varphi(x) = \frac{1}{1 + e^{-x}} \qquad (3)$$

In equation 2, y is the computed value of the node. x_i is the value of the previous layer at the position i. w_i is the value of the connection at the position i. m is the number of neurons in the previous layer. φ is the *activation function* that outputs a value between 0 and 1. The *activation function* we used is the sigmoid function (3).

We compute the value of all the nodes of the neural network. Then, we extract the values of each node in the last layer and use them as inputs for the wheels.

B. The Neat algorithm

The problem with the Fully Connected neural network algorithm is the fixed topology throughout the simulation. It is limiting the neural network to find a solution with the best arrangement of weights. If the topology is altered throughout the simulation, the network is not only able to find the best weight arrangement, but also the best topology. For example, a network with 3 layers might be performing better than a network with 4 layers. If the number of layers is fixed, two simulations must be run to test the 3-layer option and the 4-layer option. This optimal neural network could be found in just one run with a variable topology.

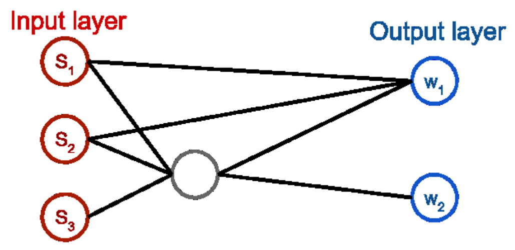

Fig. 7: In the NEAT algorithm, the network's nodes are not connected to all the nodes in the next layers. New connections are created and removed between generations.

The NEAT algorithm resolves this problem as its topology is variable. As the Fully Connected neural network, the NEAT network has an input layer and an output layer. The difference resides in the fact that the topology of a network using NEAT is not fixed, as shown in Fig. 7. In its simplest form, the algorithm has only one connection going from a random input to a random output. It evolves throughout the simulation by randomly creating new nodes and connecting them to the network. The evolution of the topology in NEAT makes it possible to find other solutions that can be hidden in the topology.

C. Creating an algorithm in LOAP

We built our platform for anybody (researcher, student, etc...) to easily add their own algorithms. We designed an easy-to-use abstract class called the Neural Network that can be used to create a Learning Algorithm. Once extended and added to the array of current algorithms, the new class will control the behaviour of the cars. Two methods are required to be redefined: *feedForward()* and *crossOver()* as shown in *diagram 1*. The *feedForward()* method is where the computation happens. This function retrieves the value of the different Transmitters (the sensors) and assign a value to the Receivers (the wheels).

For example, the fully connected algorithm takes the values of all the sensors (the transmitters), puts them in each of the input nodes, activates all the layers and applies the value of the two output nodes to the two back wheels of the car (the receivers).

V. Simulations Results

In this section, we compare the Fully Connected Algorithm with different parameters to see their effects on the learning performance of the algorithm. We evaluated (1) the impact of changing the car density and (2) the connection mutation chance.

Table I. Parameters of the simulation

Parameter	**Value**
Number of simulations	10
Number of generations	10
Number of sensors	7
Chance of weight mutation	1%, 3%, 5%, 7%, 9% and 11%
Car density	20, 50, 100, 150 and 200

5.1 Simulations configuration

Table I shows the different settings we used in our experiments and the variable settings for our two scenarios. We decided to run 10 simulations per algorithm variation to reduce the error related to randomness. When we test the different weight mutation chance, we keep the car density at 25. When we vary the car density, we keep the chance of weight mutation to 9%.

In the first scenario, we highlighted the impact of changing the chance of a connection weight, replacing it by a random value during the mutation process. In Fig. 8, each line represents the Fully Connected algorithm with a different weight modification chance.

Through extensive simulations, we conclude that the Fully Connected algorithm performs better when its connection weights have 9% and 11% chance of getting modified. We expect that going higher than a certain threshold would result in a performance drop because of the higher amount of randomness.

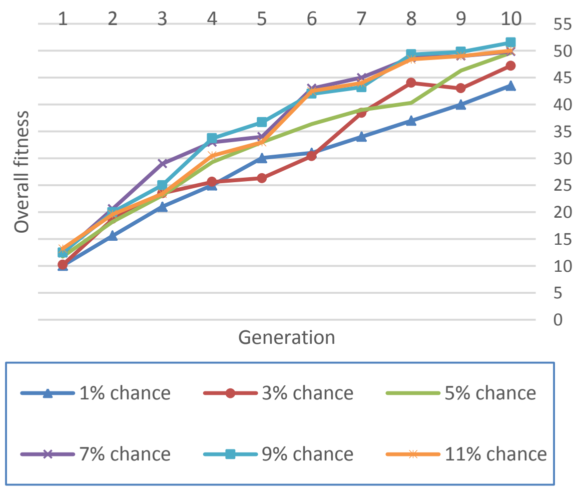

Fig 8. *Scenario 1*: How the performance of the Fully Connected Algorithm is influenced by the weight modification chance.

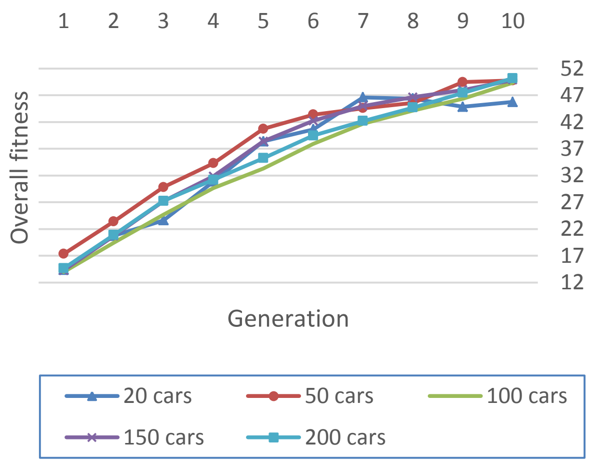

Fig 9. *Scenario 2*: How the car density affects the performance of the Fully Connected algorithm.

In the second scenario, we investigated a possible correlation between the car density and the performance of the Fully Connected algorithm. Fig. 9 shows the evolution of the average fitness for each algorithm variation tested.

We cannot conclude that the car density influences the performance of the algorithm, but we note that it has an impact on the smoothness of the curve. The blue line is very shattered and has less cars, while the orange one with 200 cars has a very smooth line. This is probably due to the stability of the population.

VI. Conclusion and Future Work

In this paper, we offer a platform that makes it easy for anybody to test and compare their learning algorithms. The platform uses learning algorithms with a modified genetic natural selection algorithm named GNS* to train them. Multiple algorithms are built in the platform, such as the fully connected ANN and a version of the NEAT algorithm. Through extensive simulation, we found out that our platform LOAP could be used to better understand learning algorithms and to find better settings. For example, we could use LAOP to find the optimal number of proximity sensors.

In the future, the platform will have major improvements, like a better respect of the real-world conditions, more sensors, the ability to import a shapefile and a web platform where users can share their algorithms.

Acknowledgment

This research was financially supported by the "Fonds Québécois de la recherche sur la nature et les technologies (FRQNT)."

References

[1] "On the Road to Fully Self-Driving", Waymo safety report, 2017, https://storage.googleapis.com/sdc-prod/v1/safety-report/waymo-safety-report-2017-10.pdf [last visit 06/01/2019].

[2] L. Oest O'Leary, C. Bisaillon and J. Rezgui, "LAOP: Learning Algorithm Optimization Platform" on GitHub, https://github.com/lool01/LAOP, 2019 [last visit 06/01/2019].

[3] K. O. Stanley and R. Miikkulainen, "Evolving Neural Networks through Augmenting Topologies", Evolutionary Computation Journal, 10(2):99-127, 2002.

[4] A. Dosovitskiy et al. "CARLA: An Open Urban Driving Simulator", http://proceedings.mlr.press/v78/dosovitskiy17a/dosovitskiy17a.pdf, [last visit 06/01/2019]

[5] Samuel Arzt, "Applying Evolutionary Artificial Neural Networks on GitHub, https://github.com/ArztSamuel/Applying_EANNs, [last visit 06/01/2019].

[6] J. Brownlee, "The pole balancing problem - A Benchmark Control Theory Problem", Technical Report 7-01 July, 2005, https://pdfs.semanticscholar.org/3dd6/7d8565480ddb5f3c0b4ea6be7058e77b4172.pdf, [last visit 06/01/2019].

[7] *M*EMS Journal Automotive sensors and electronics 2015. http://www.automotivesensors2015.com/[last visit 06/01/2019].

[8] "ESRI Shapefiles technical description", an ESRI white paper, July, 1998, https://www.esri.com/library/whitepapers/pdfs/shapefile.pdf, [last visit 06/01/2019].