Tuning Trajectory Following Controllers for Autonomous Cars Using Reinforcement Learning

Ana Vilaça Carrasco
ana.vilaca.c@tecnico.ulisboa.pt

Instituto Superior Técnico, Lisboa, Portugal

December 2021

Abstract

This paper proposes the use of a *Q-Learning* algorithm to train an agent to tune the parameters of a trajectory controller. This controller is used on a system designed to perform two maneuvers - lane changing and roundabout navigation - and is also capable of avoiding frontal collisions and controlling velocity and steering angle range. This work integrates a Reinforcement Learning agent with the CARLA simulation environment to test the efficacy of a simplified Q-learning algorithm.

The results after the training demonstrate an appropriate implementation of the algorithm. A custom environment was created to test all the functions of the system, as well as the quality of the tuned gains. The results show the system performing all the functions as intended. They also suggest the gains were appropriately tuned, attesting to the success of the proposed algorithm.

Future works include real-world experimenting with the system, upgrading the Q-learning algorithm to a Double Q-learning, and adding a Neural Network for image processing of the environment.

Keywords: Machine Learning, Reinforcement Learning, Autonomous Driving Systems, Q-Learning

1. Introduction

Autonomous Driving technologies have a vast range of applications, the most notorious being fully autonomous vehicles, but also other explored technologies like Vehicle-to-vehicle and Vehicle-to-Infrastructure[10]. Partially autonomous technologies can be found in many applications (Adaptive Cruise Control, Parallel Parking Assist). Massive implementation of fully autonomous vehicles can have countless advantages: avoiding accidents, reducing congestion, optimizing resources like energy, time and money, and a more Eco-friendly impact in the world.

Machine Learning (ML) offers many benefits to the field of **Autonomous Driving Systems (ADS)**: self-optimisation based on collected data and new environments, the ability to specify the desired behaviour of the system, and increased generalisation capacity. Machine Learning has become the primary interest of the autonomous driving industry.

This work proposes a system with a modular architecture and a **Reinforcement Learning (RL)** agent for lateral control of a vehicle. This agent is trained for **offline tuning of the trajectory control gains**, to minimize the trajectory errors. The RL agent, using a discretized tabular **Q-learning algorithm**, is trained to fine-tune these gains to perform two maneuvers: **lane changing** and **roundabout navigation**. The system includes a module that identifies when to perform each maneuver and changes the gains appropriately, and also a safety module that controls the vehicle's velocity and steering angle range and avoids frontal collisions. The **CARLA Simulator**[4] is used to train and test the proposed system. The simulator communicates with the system through a **ROS**[12] bridge.

This work presents an alternative to the popular online gain tuning: instead of tuning the gains while driving the vehicle through live updates of the Q-table values, this thesis proposes updating the Q-table and tuning the gains every time a specific maneuver is performed by the vehicle - the data is collected and processed afterward, and the gains adjusted accordingly. This would mean that the gain tuning would occur in an extended period of time, rather than at each iteration. Offline tuning avoids possible sudden gain changes while driving, which can cause safety issues. It also allows the user to tune the gains to specific environments inside a city, which can be beneficial to use in well-known dangerous zones. This work demonstrates that a Q-learning algorithm with limited state-action space and constricted combination of gains can still produce satisfying results. A simplified algorithm brings advantages like: code errors are less likely, easier to troubleshoot, easier to implement in a project with limited computational resources, and easier transition to a physical system.

2. Background

The **perception system**, an essential part of an ADS, is made up of a set of sensors that use fusion techniques to combine the gathered information about the environment. The general functioning of an ADS is heavily based on the perception of its surroundings, hence the perception system has a significant impact on the system's performance. The sensors that usually make up perception systems are: **Ultrasonic**, **RADAR**, **LiDAR** and **cameras**. **Sensor fusion** manages the advantages and disadvantages of each sensor to improve the quality of the data perceived. **Ultrasonic** sensors are especially used in short-distance measurements at low velocities. They are cheap sensors that work well with any material and harsh environments but are also prone to false positives. **RADARs** are high accuracy sensors that detect targets in a wide range around the vehicle. They are especially good in all weather conditions but are also known for a reduced individual Field of View and a lack of precision. **LiDARs** are known for their high resolution and great accuracy in perception of the environment, even in the dark. These are very expensive sensors that are affected by weather conditions such as rain, snow, fog or dust. **Cameras**, the most commonly used sensors, are especially well suited for object recognition and are also very cost-efficient. The most common cameras (RGB) are highly affected by variations in lighting conditions, rain, snow, or fog, and lack depth perception.

Three domains can be distinguished when designing an intelligent agent: **learning strategies**, **neural network architecture** and **training algorithm**. In Machine Learning, most training algorithms fall into one of three categories: **Supervised Learning (SL)**, **Unsupervised Learning (UL)** and **Reinforcement Learning (RL)**. **Reinforcement Learning** is distinguished for the self-improvement and generalisation capabilities it offers. This strategy is suitable for typical time-sequential problems, which is the case for autonomous driving. Unlike SL, it does not require a **labeled dataset** and instead offers the possibility of explicitly defining the desired behaviour for the agent (reward function), which may be an advantage or a disadvantage, depending on the complexity of that task. RL agents take notoriously more time to train compared to SL agents, and its training in the real world may be challenging, potentially raising safety and cost complications. **Supervised Learning** offers a high-speed training convergence and does not require specifications on how the agent must reach its goal. An agent's training is based on labeled data, which can be unfavorable for multiple reasons: acquiring and labeling data can be a very slow process; labeled data may contain biases towards certain actions or situations; agents trained with labeled data usually present less adaptability to new environments than agents trained with RL. While with SL, ADS are able to imitate a human driver, with RL the vehicle is able to learn to drive better than a human, since it's learning by itself. On the other hand, choosing RL can exhaust the resources of a project, without guaranteeing applicability to the real world. **Hybrid learning methods**, like using SL as a pre-training step to a RL agent, try to take advantage of both faster learning processes and better self-adaptation to new environments.

The most common **neural network architectures** used in the field of autonomous driving are **Convolutional Neural Networks (CNN)**, **Recurrent Neural Networks (RNN)**, **Long Short-Term Memory (LSTM)** RNNs and **Fully Connected Neural Networks (FCNN)**, since these have exceptionally good results in solving image processing and temporal sequence data processing problems. Although RNN and LSTM applications produce good results, CNNs are still the broadly used architecture.

SL problems can be broadly grouped into two types of problems: **classification** problems, which consist of predicting a discrete class label, and **regression** problems, which consist of predicting a continuous quantity. In RL, algorithms are broadly divided into three classes: **value-based**, **policy gradient** and **actor-critic** algorithms. One of the most commonly used RL algorithms in the ADS context is **Q-learning** or variations, some of the most relevant being **Hierarchical Q-Learning**[2], **Double Q-Learning**[5], and **SARSA**[14]. In the context of autonomous driving, **Deep Q-Networks**[7] are a popular choice, as they make it possible to apply Q-learning methods to a high dimensional state-action space, and optimize the training process with Neural Networks as function approximations.

Other popular RL actor-critic algorithms include **Supervised Actor-Critic** (SAC)[16] and **Asynchronous Advantage Actor-Critic** (A3C)[11]. By having a combination of value-based and policy gradient approaches, these algorithms find a way to have a better performance than the other two algorithms separately. There is another subset of RL called **Imitation Learning** [6], with some methods, like **Inverse Reinforcement Learning (IRL)**[7], surging to overcome obstacles like dataset biases and learning complex reward functions. **Neural Networks** are a popular approach, used to solve either classification or regression problems, or are used as function approximators for the RL algorithm. When using NNs, the most commonly used **optimizers** are **Stochastic Gradient Descent** (SGD) and its variants, like **AdaBoost**, **AdaGrad** and **Adam**. Although some works claim better results from one or two optimizers, it is still a highly argued subject.

2.1. Q-Learning

Q-learning[15] defines a table shaped $[state, action]$, with an utility value for each pair. It initializes it randomly and then updates those utility values (known as the ***Q-values***) by greedily maximizing a new kind of **value function**, called the **Q-function**:

$$\begin{aligned} Q^{new}(s_t, a_t) \leftarrow (1-\alpha) \cdot Q(s_t, a_t) + \\ \alpha \cdot (r_t + \gamma \cdot \max_a Q(s_{t+1}, a)) \end{aligned} \tag{1}$$

$Q(s_t, a_t)$ is the Q-value of the state-action pair before the update, α is the **learning rate** ($0 < \alpha \leq 1$), $r_t = R(s_t, a_t)$ is the **expected reward value** associated with taking the action a_t in state the state s_t, γ is the **discount factor** and $\max\limits_a Q(s_{t+1}, a)$ is the **estimate of the optimal future value**. This equation can be understood as: the **new Q-value** of the (s, a) pair is the **expected reward** for executing action a in state s plus **the best Q-value** it can produce in the new state, s', weighted by γ. The **Convergence Theorem** for this algorithm is defined in [15].

Full vehicle autonomy is a relatively new area of research. Robust high-performing full vehicle autonomy is still out of reach, the main constraint being the significant complexity of the tasks performed by the agent. A variety of learning strategies have been explored, however showing a preference for end-to-end architectures with SL and CNNs. The inputs are generally camera images and the outputs are steering and acceleration variables. Although the results show progress towards full vehicle autonomy, it still shows a lack of adequate performance and robustness compared to partial autonomy.

2.2. Challenges

Over the last years, there has been a lot of research around the applications of Machine Learning to autonomous vehicle control, which has been supported by tremendous technological growth in many related areas. However, it still faces a significant amount of challenges, especially with real-world applications [9]. Two of the biggest obstacles faced by Machine Learning applications are the **high computation complexity** of the methods and the size of the **dataset**. While bigger and more diverse datasets provide more robust and reliable systems, computational complexity grows exponentially with the number of dimensions of an environment (**Curse of Dimensionality**), making the training process slower for more complex environments. It is also important to ensure the **datasets** are diverse and don't include biases, which often happens in simulated environments. In Reinforcement Learning applications, defining an adequate reward function is essential but usually very difficult. IRL is known to help with this obstacle since it provides a natural system for optimizing the reward function without the need to define it empirically. However, IRL is usually expensive to run and needs a lot of human driving data to obtain decent results. An autonomous vehicle encounters a very wide range of inconsistent environments, which is why it is vital to ensure the system can generalise well to new environments. **Generalisation** is difficult to guarantee, and it's virtually impossible to test the vehicle in all the scenarios it could encounter. Simulators are usually used to train the system on a diverse set of environments, but these systems often fail to generalise well in real environments. To properly **validate** the performance and safety of the system, rigorous testing must be applied. Testing in the real world can often be expensive, time-consuming and unsafe, so that is why simulation studies are becoming the dominant method of study in this field [9]. However, despite the advantages, simulators difficult the process of transferring the system to the real world. The developer must ensure the agent's policies don't overfit the simulation environment, which is why **it's important to validate both the model and the simulation environment**. **Safety** is, undoubtedly, one of the most important properties of an autonomous vehicle. Guaranteeing that these systems are safe is a top priority since a failure or malfunction can have devastating consequences. However, the solutions provided by systems operating with Machine Learning methods are increasingly difficult to interpret and control. The **lack of transparency** of these systems is known as the **black box problem** [9]. Approaches to ensure functional safety include combining reinforcement methods with traditional control methods, increasing safety through traditional control theory.

2.3. Related Work

When designing an intelligent ADS, the most popular approach is a SL-based end-to-end architecture with a complex neural network. Bojarski et. al [3] proposes **an end-to-end architecture with a 9-layered CNN trained with Supervised Learn-**

ing methods to map RGB images directly to steering commands for the vehicle, which revealed to be surprisingly powerful. However, the proposed NN architecture adds great computational complexity to the system, while also requiring large datasets to be created for training. As an alternative to an end-to-end architecture, Bari S. et. al [1] proposes the use of **a simple 3-layered NN for online tuning of a PID controller's parameters** of a Flight Control of a Quadcopter. This work showcases the benefits of applying ML directly in the control module, over conventional offline PID tuning methods. Despite that, the testing environment is simplified and the proposed system's robustness is not comparable to an autonomous driving system. The works from Panagiotis Kofinas et. al [8] and Shi et. al[13] both offer an alternative RL approach for tuning PID controllers. Shi et. al[13] proposes the **tuning of two PID controllers' gains in a simulated cart-pole system, using a Q-Learning algorithm with 6 discretized Q-tables**. This work argues in favour of introducing the adaptive properties of the Q-Learning algorithm directly into the control module through PID tuning. However, the proposed system requires extensive training times. Panagiotis Kofinas et. al [8] proposes a **hybrid Zeigler-Nichols (Z-N) Reinforcement Learning-based PID tuner** to control the speed of a DC motor, where the PID gains are set by the Z-N method and then are tuned online through a **fuzzy Q-learning agent**. The authors argue that using a NN as a function approximator would result in a very slow learning rate and, although simpler, discretizing the action-state space can lead to sudden gain changes, which may cause dangerous behaviours. Instead, they opt for Fuzzy logic to allow the Q-Learning to be applied in a continuous state-state, despite increasing the complexity of the system.

The work of [13] and [8] signify the viability of using a Q-learning algorithm to tune a controller's gains in a simulated environment. It is argued that working with a higher-level controller (**trajectory controller**) instead of the PID controller, defining **control laws** and setting **gain limits** dismisses the issues raised in [8] against using a discretized action-state space. Furthermore, discretizing and limiting the state-action space reduces **training times**. Finally, applying these ML methods only on the motion control module allows for a high level of control over the system, as other modules may be added to build a "safety net", giving the developer even more control over the system.

3. Methodology

The proposed system's architecture is presented in fig.1. This architecture can be divided into 4 modules: the **vehicle simulator**, a **high-level controller**, a **Reinforcement Learning agent** and a **low-level controller**. The **simulation** consists of a vehicle with multiple sensors attached. The **low-level controller** drives the vehicle through a predefined reference path by calculating and imposing values for velocities and steering angle. The goal is to optimize the vehicle's behavior while performing certain maneuvers. The optimization process consists of adjusting the **set of gains**, used to calculate the velocities and steering angle values, to minimize the average error between the vehicle's pose and the reference path. The **RL agent** is trained to find the best set of gains for each maneuver. The maneuvers that were tested were **lane changing** (to the right) in a straight road and **circulating in a roundabout**. The **high-level controller** processes information from the simulator and communicates with the other modules, making a bridge between them.

The system works as follows: while the vehicle follows the reference path, the high-level controller collects and processes data from the simulation sensors. If necessary, it sends **high-level control actions** to the low-level controller. If the system needs to perform any of the maneuvers, the high-level controller sends that information to the RL agent, which will then set the gains to the appropriate fine-tuned values for that maneuver. Those **fine-tuned set of gains**, (K_v, K_l, K_s), are then sent to the low-level controller to adjust the **steering angle**, ϕ, the **linear and the angular velocities**, v and w_s - these three values define a **low-level control action**. The simulator also communicates directly with the low-level controller, sending an estimation of the vehicle's current pose. This estimation is made based solely on the vehicle's odometry.

The tuning process is offline, therefore the set of gains are constant while the vehicle follows the reference path, and will only change if the vehicle is prompted by the High-Level controller to perform one of the mentioned maneuvers.

Essentially, the vehicle is expected to behave as a human driver would when facing different types of environments. The gain adjustments should mimic the adjustment in the driving style a human driver would display.

3.1. Simulator

CARLA is an open-source simulator designed for autonomous driving research. It simulates urban driving environments, with multiple vehicle models, sensors, objects and pedestrians. The simulator has a bridge with ROS, which allows direct communication with the simulated vehicle, through publishers and subscribers from this bridge, and

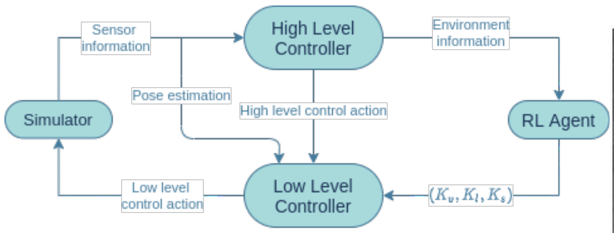


Figure 1: Full system architecture - the simulator module can be substituted by a real vehicle

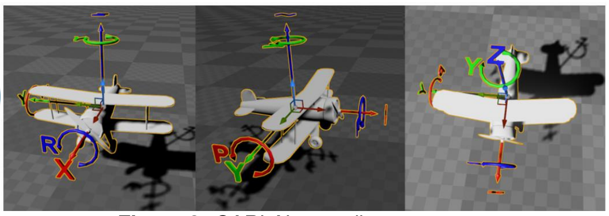


Figure 2: CARLA's coordinates system

also provides a way to customize the vehicle setup, through a *.launch* file. The proposed setup is a "Tesla Model 3" vehicle with a speedometer, a collision detector sensor, an odometry sensor and an obstacle detector sensor located on the vehicle's bumpers. This sensor is initialized separately from the rest of the setup, since it is not included in the **CARLA ROS bridge** packages. The system communicates with the controller inside the simulator through **Ackermann Messages**. Each iteration, the system uses the publisher *"/carla/ego_vehicle/ackermann_cmd"* to send a message with updated values for the *steering angle*, ϕ, for *speed*, v and for *steering angle velocity*, ω_s. Then, the simulation controller converts the Ackermann Messages into commands that control acceleration and velocity through an integrated Python-based PID. The simulator communicates back with the client through subscribers to topics such as *"/carla/ego_vehicle/odometry"* for the vehicle's odometry, *"/carla/ego_vehicle/vehicle_status"* to check the vehicle's current linear velocity, *"/carla/ego_vehicle/collision"* to check for collisions and *"/carla/status"* to keep track of the simulation time. The obstacle detector sensor's data is sent to the system by a publisher created in a separate script. Figure 2 shows the coordinate system of the vehicles inside the CARLA simulator - this will be referred to as the **vehicle frame**. The simulation has its own clock, conducted by the server. This **simulation time** is generally slower than real time. The time span between two simulation frames, called the time-step, can be configured to be either variable - running as fast as it can - or fixed. For this project, the simulator runs with a **fixed time-step of 0.01 simulated seconds**, which is equivalent to 100 frames per simulated second. CARLA is built over a client-server architecture. The communication between client and server can be asynchronous or synchronous. In this case, these communications run in **synchronous mode**, which means the simulation runs as fast as the client can process the information, and never faster. The simulator was also configured to wait for a new vehicle control command before processing each new frame.

3.2. Low-level controller

The controller module is in charge of controlling the trajectory of the vehicle by adjusting the values of the **steering angle** $\boldsymbol{\phi}$, **linear velocity**, $\boldsymbol{v}$, and **angular velocity**, $\boldsymbol{\omega_s}$, in real-time, with the goal of minimizing the error between the reference and the actual pose of the vehicle. Figure 3 illustrates how the **error in the world frame**, $\boldsymbol{^w e}$,is obtained.

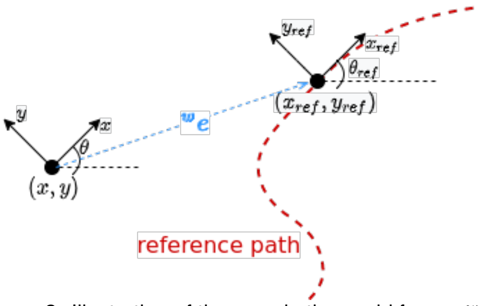


Figure 3: Illustration of the error in the world frame, $^w e$.

The control laws that will determine the values of v, ω_s and ϕ are defined by the equations:

$$v = K_v {}^b e_x, \quad \omega_s = K_s {}^b e_\theta + K_l {}^b e_y, \quad \phi = \int \omega_s \, dt \tag{2}$$

where $\boldsymbol{^b e}$, the **error in the vehicle frame**, is obtained from $^w e$ by means of a **rotation matrix** of a θ degree rotation around the Z axis.The values for v, ω_s and ϕ are then converted into control actions to communicate with the vehicle. The **trajectory controller gains** are the **linear velocity gain**, $\boldsymbol{K_v}$, the **steering gain**, $\boldsymbol{K_s}$, and the **linear gain**, $\boldsymbol{K_l}$, which are the focus of this work. The diagram in fig.4 explains how the controller functions.

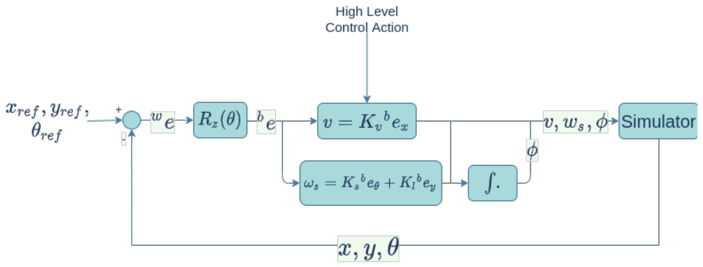

High Level Control Action
$x_{ref}, y_{ref}, \theta_{ref}$
+
-
$^w e$
$R_z(\theta)$
$^b e$
$v = K_v {}^b e_x$
$\omega_s = K_s {}^b e_\theta + K_l {}^b e_y$
$\int .$
ϕ
v, w_s, ϕ
Simulator
x, y, θ

Figure 4: Diagram of the controller

3.3. Reinforcement Learning agent

The RL agent is responsible for tuning the trajectory controller gains. The proposed method aims to automatize this process, which is usually manual, using a **simple Q-learning algorithm** with a **discretized Q-table** that takes an interval of gains and finds the set of gains with which the vehicle presents the best performance. Performance evaluation is translated in the reward function of the algorithm. The RL environment is defined as follows:

States: An array with the **average of the absolute value of the lateral and orientation errors**, $S = [E_y, E_\theta]$. Each of the error values have low and high limits, E_{LOW} and E_{HIGH}, and are discretized into 40 units;

Actions: Each action, $A = [a_0, a_1, a_2]$, is represented by an array that defines either an **increase, decrease or invariance of each of the controller gains**. There are 27 different actions. The gains are adjusted by the action array through the following equation:

$$K_v = K_v + h_0 a_0, K_l = K_l + h_1 a_1, K_s = K_s + h_2 a_2 \tag{3}$$

The variables a_0, a_1 and a_2 can take the values 1, 0 or -1. The values h_0, h_1 and h_2 are positive constants that will either be ignored, subtracted or added to the previous value of the gain.

Terminal condition: Given how difficult it is to reach the state $S_0 = [0, 0]$, the adopted approach was to consider any state that would come closer to S_0 to be the terminal state. As a result, if the current state is closer to S_0 than the closest state recorded so far, then a terminal state was reached. To determine the distance of a state to the state S_0, $d(S, S_0)$, the algorithm uses a **weighted euclidean distance**. Since the lateral error values, E_y, are generally 10 times greater than the orientation error values, E_θ, the weight array used was $[1, 10]$:

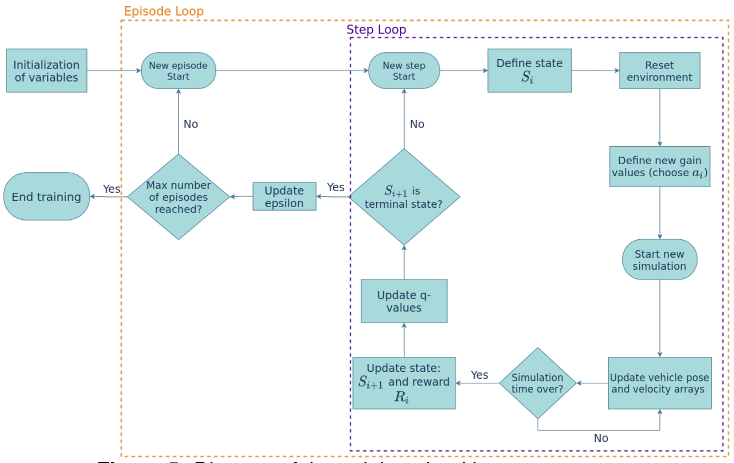
$$d(S, S_0) = \sqrt{|E_y|^2 + 10 * |E_\theta|^2} \tag{4}$$

Reward function: The reward function chosen for this work is defined by the equation:

$$R = \frac{1}{1 + d(S', S_0)} - \frac{1}{1 + d(S, S_0)}, \tag{5}$$

where $d(S', S_0)$ is the distance between the new state S' and S_0 and $d(S, S_0)$ is the distance between the current state S and S_0. This function is based on the one used in [8]. Also, if a collision is registered the reward is decreased by a defined value.

Training Algorithm: The RL agent was trained to perform two different maneuvers: a lane changing maneuver in a straight road and driving in a roundabout.The **algorithm used to train** the RL agent



Episode Loop
Step Loop
Initialization of variables
New episode Start
New step Start
Define state S_i
Reset environment
Define new gain values (choose a_i)
Start new simulation
Update vehicle pose and velocity arrays
Simulation time over?
No
Yes
Update state: S_{i+1} and reward R_i
Update q-values
S_{i+1} is terminal state?
No
Yes
Update epsilon
Max number of episodes reached?
No
Yes
End training

Figure 5: Diagram of the training algorithm

is shown by the diagram in fig.5. The agent was trained over a certain number of **episodes**, each of which is divided by **steps**. Each step, a current state, S_i, is defined based on the last error average registered. Then, an action is taken and the new gains are defined, after which a new simulation starts, with the system's controller guiding the vehicle through the reference path. After the simulation stops, the new state, S_{i+1}, and the reward, R_i, are updated. With these values, the Q-table is updated based on equation (1). If the new state, S_{i+1} does not satisfy the terminal condition, this cycle repeats in a new step. Otherwise, the episode ends.

3.4. High-level Controller

The high-level controller works as both an **event manager** and a **safety module**. It determines if the vehicle needs to perform one of the two maneuvers and enforces **high-level control actions** on the low-level controller, including enforcing a speed limit, controlling the steering angle range and avoiding collisions. The diagram on fig.6 demonstrates how this controller works. To test the system, the map was divided into **zones**, each of which associated with an event. Figure 12 shows the organization of the map. The map is divided into 4 types of zones: the **blue zone**, where the vehicle performs a lane change, the **red zone**, where the vehicle navigates in a roundabout, the **green zone**, where the collision avoidance function is tested and the **orange zones**, where the steering angle range limit is temporarily reduced. The reference path is marked by a red line. The controller uses the vehicle's position to identify which zone it is in. If a vehicle is detected in the green zone, the High-Level controller will overwrite the Low-level control actions, lowering the linear velocity and imposing $\omega = 0$ and $\phi = 0$ until the vehicle stops.

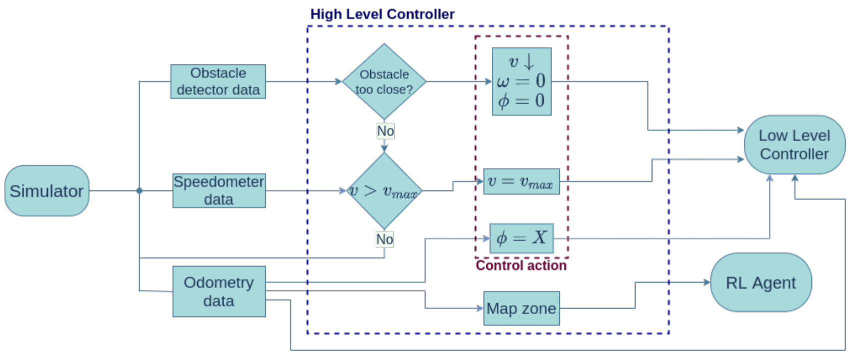


Figure 6: Diagram of the high-level controller's internal operations

4. Results & discussion

4.1. CARLA simulator validation

A series of experiments were carried out to assess the simulator's performance while working with the proposed system. The experiments were designed to evaluate the simulator's odometry precision. Many factors, such as delays in communications with the simulation server, lower the simulator's precision. Initial tests revealed a standard deviation around $\sigma = 0.072$ of the values of the average lateral error, E_y. High deviation values can compromise the Q-learning agent's quality. After some adjustments to the algorithm and to the simulator settings, the following tests registered deviation values below $\sigma = 0.035$. These tests greatly outline the stochastic behaviour of the simulator.

4.2. Agent training and testing

The values chosen for the parameters, for each of the maneuvers, are shown in table 1. The **Loop time** represents the time of each training step, in simulated seconds. The values $[h_0, h_1, h_2]$ are the positive constants that define the action values (eq. (3)). ϵ decay and start value refer to the **ϵ-greedy policy**[14] used in the Q-learning algorithm. ϕ range is the default steering angle range used during training. **Step limit** refers to the maximum number of steps an episode can have, a condition that prevents unfeasible training times. $K_{max/min}$ refers to the limits of each of the gains, K_v, K_l and K_s.

Table 1: Table of parameter values for each of the training environments,where n is the number of episodes

	Lane Change	**Roundabout**
Loop time	5	30
$\boldsymbol{\alpha}$	$1/(n+1)^{0.6}$	$1/(n+1)^{0.6}$
$\boldsymbol{\gamma}$	0.9	0.9
$\boldsymbol{E_{HIGH}}$ **(m)**	[4 , 0.4]	[1 , 0.1]
$\boldsymbol{E_{LOW}}$ **(m)**	[0 , 0]	[0 , 0]
$\boldsymbol{K_{min}}$	[0.1, 1, 1]	[1, 1, 1]
$\boldsymbol{K_{max}}$	[2.42, 21, 21]	[5.8, 21, 21]
$\boldsymbol{[h_0, h_1, h_2]}$	[0.58, 5, 5]	[1.2, 5, 5]
$\boldsymbol{\phi}$ **range (°)**	30	30
ϵ (start)	1	1
ϵ **decay**	1/(n/2)	1/(n/2)
Step Limit	120	100

Using the algorithm illustrated in fig.5, the agent is trained to find the set of gains that minimize the error while the vehicle performs the two maneuvers. The agent was trained for **30 episodes** for the lane changing maneuver and for **20 episodes** for the roundabout navigation. The **training times** of these tests were 18.5 hours and 16 hours, respectively. Figures 7-8 show the sum of rewards of each episode of the two trainings, also known as **the learning curve**, which is an indicator of the efficiency of the algorithm applied [13]. Both figures show a **convergence** of the learning curve, which implies the success of the algorithm. Episodes 21, 23, 27 and 22 on fig.7 and episode 4 on fig.8 reveal a drop in value. In these episodes, the agent reaches the **step limit** without finding a terminal state and is forced to move to the next episode, causing the drop in value. The center of mass was chosen to **define the set of gains** used in the blue and red zones. All terminal gains obtained after the convergence of the algorithm are considered, i.e., the gains from episode 15 onward for the lane changing maneuver and from episode 10 onward for the roundabout navigation. The episodes where a terminal state is not reached are discarded. The center of mass of each of these groups of gains are $\mathbf{(2.42, 11.67, 1.33)}$ and $\mathbf{(2.2, 12.5, 1.0)}$, respectively. These are the sets of gains used in the validation tests.

The **validation process** consists of evaluating the performance of these sets of gains while performing the corresponding maneuver and comparing them with the performance of other sets of gains. The tests lasted for 10 sim.secs for the lane changing and 50 sim.secs for the roundabout navigation. **Figures 9 and 10** show the trajectory performed by the vehicle in blue and the reference path in orange. The figures suggests that the system can perform the maneuvers efficiently, successfully correcting the initial errors imposed.

Table 2 presents the **average Mean Square Error (MSE)** of the lane changing maneuver for some sets of gains spread through the range of values.

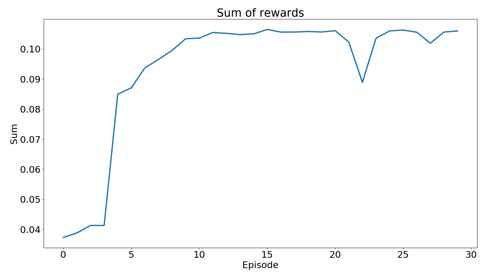


Figure 7: Lane changing training: Sum of rewards per episode

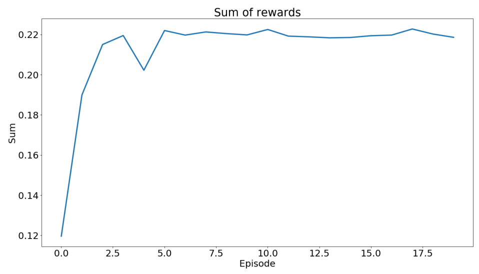


Figure 8: Roundabout training: Sum of rewards per episode

As mentioned before, some dispersion is observed in the error values produced by the simulator. For that matter, the table presents **the highest average MSE registered out of 10 samples**, excluding the first second of the simulation.The third column presents the **standard deviation** values, σ, from the 10 samples gathered from each set of gains.

	Average MSE	σ
$(0.68, 1, 1)$	2.723	0.407
$(0.68, 21, 21)$	2.194	0.327
$(1.26, 11, 11)$	1.294	0.174
$(1.84, 16, 16)$	0.842	0.135
$(1.84, 11, 1)$	0.7809	0.1203
$\mathbf{(2.42, 11.67, 1.33)}$	**0.4658**	**0.0328**
$(2.42, 21, 21)$	0.5628	0.05

Table 2: Average MSE of set of gain from the lane changing validation tests

A qualitative analysis of these values reveal that the chosen gains present the lowest average MSE, with gains $(2.42, 21, 21)$ following close behind. Also, the closer to the chosen gains, the lower the σ values are. The table also suggests that **differences in the value of K_v have more impact on the results**, with the biggest decreases observed between $K_v = 0.68$ and $K_v = 1.26$.

Similarly to table 2, table 3 presents the information gathered about some sets of gains for the roundabout navigation. The values presented were the highest out of 10 samples, and exclude the first 10 seconds of the maneuver. The third column presents the standard deviation, σ, of those 10 samples:

	Average MSE	σ(e−3)
$(1, 1, 1)$	8.8e−3	0.31
$(2.2, 1, 1)$	5.8e−3	0.27
$\mathbf{(2.2, 12.5, 1.0)}$	**3.0e−3**	**0.14**
$(2.2, 21, 21)$	6.7e−3	2.9
$(3.4, 1, 1)$	0.010	1.9
$(4.6, 11, 21)$	0.01612	3.0
$(5.8, 21, 21)$	0.120	30.5

Table 3: Average MSE of set of gain from the roundabout validation tests

The chosen set of gains presents the lowest MSE value, followed by $(2.2, 1, 1)$ and $(2.2, 21, 21)$. Also, comparing the gains $(1, 1, 1)$ and $(5.8, 21, 21)$, it seems that **lower gain values produce lower MSE**. Contrary to the previous maneuver, all the sets of gains presented **very low standard deviation values**. The lowest σ values are also found closer to the chosen gains.

Overall, the performance of the system working with the chosen gains, for both maneuvers, show low lateral errors and MSE values. Tables 2-3 suggest that **the agent's choice of gains is around the values that minimize the average MSE**. Although the deviation observed in the error values lowers the accuracy of the agent, the lowest MSE scores frequently belong to the chosen set of gains.

4.3. Full System Testing

The system was tested while navigating in the environment illustrated in figure 12, following the reference path defined in red. The chosen gains for the blue and red zones were, respectively, $\mathbf{(2.42, 11.67, 1.33)}$ and $\mathbf{(2.2, 12.5, 1.0)}$. The steering angle range is reduced to $10°$ inside the orange zones. For testing purposes, the speed limit imposed is 4 m/s ($\simeq 14.4$ km/h). The collision avoidance method was tested by spawning a second vehicle inside the green zone, blocking the system's path. When the vehicle detects an obstacle, the High Level Controller imposes a **new equation for the linear velocity**, defined as:

$$v = v_i \cdot (d_i/20), \tag{6}$$

where v_i is the velocity of the vehicle when the obstacle is first detected (20 meters away) and d_i is the distance to the obstacle registered by the sensor at each time-step. The results are presented in **figure 11**. It shows the trajectory performed by the system, in blue, and the reference path, in orange, with the second vehicle marked by a red rectangle. The figure shows the system successfully follows the reference path, without any major errors or collisions. The last portion of the trajectory also shows the **collision avoidance function** in action, identifying the second vehicle and coming to a full

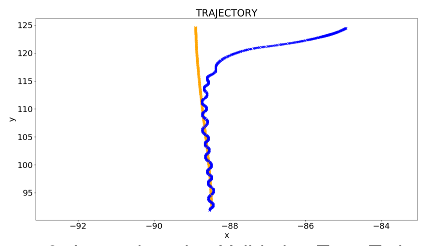


Figure 9: Lane changing Validation Test: Trajectory

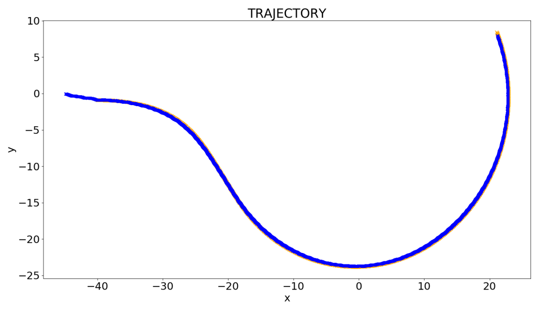


Figure 10: Roundabout Validation Test: Trajectory - the reference and the trajectory are superimposed

Table 4: Average MSE values of the system test using the chosen sets of gains and sets of gains outside the cluster

	Average MSE	σ(e−3)
$(0.68, 21, 21)$, $(2.2, 21, 21)$	0.0742	5.5
$\mathbf{(2.42, 11.67, 1.33)}$, $\mathbf{(2.2, 12.5, 1.0)}$	$\mathbf{0.0711}$	$\mathbf{2.0}$
$(2.42, 21, 21)$, $(5.8, 21, 21)$	0.164	19.1

stop a few meters away from it, despite the reference path. This produces a slight deviation from the reference path, the result of imposing $\omega = 0$ and $\phi = 0$.

Table 4 shows the average MSE of the tests performed with the chosen gains and other less preferred gains. The third column shows the standard deviation, σ, from the 10 samples registered for each set of gains. The MSE values registered during the final portion (green zone) were ignored.

The table shows the chosen set of gains produces the lowest average MSE, with very low deviation values. It is worth noting that, comparing to tables 2 and 3, the differences between average MSE values of each set of gains is close to none. This suggests that this type of tuning has **more impact when applied locally** (in a specific zone or for a maneuver), rather than generally (throughout the full path).

The results demonstrate all of the proposed functions **working correctly**. The trajectory in figure 11 and the velocity values registered during this test demonstrate that the system does not engage in unsafe behaviour, like collisions or excessive velocity. It also consistently follows the reference with very little error. The MSE values in table 4 suggest the chosen gains are in the neighbourhood of values that overall minimize the trajectory error.

5. Conclusions

This work proposes a **different approach** to using a Reinforcement Learning algorithm in an autonomous driving system: a **"long-term" offline tuning** of the gains that requires the entire maneuver to be executed to gather information and learn from it. This approach not only avoids dangerous online fluctuations of the gains but also introduces a way to adjust the gains to specific maneuvers or zones inside a city. The **drawback** of this approach is the training times, which could take as long as a few hours per episode. This means that the algorithm can only be trained with a significantly reduced number of episodes, compared to the most common RL applications. For that matter, this approach would be used as a "long term" tuner, training the algorithm within several weeks, months or years.

Meeting the Q-learning convergence conditions, combined with the learning curves obtained, suggest **a proper implementation of the algorithm**. The validation tests performed to the chosen gains suggest that the algorithm can tune the gains to values that **minimize** the trajectory error. The results presented show that **all the modules of the system are working as intended**.

During this project, RL has shown to be a good approach when working with the CARLA simulator and limited computational resources - by not requiring labeled datasets or image processing, it **significantly reduces** the computational requirements, while still providing adaptability and robustness to the system. Using a **simplified version** of the Q-learning algorithm also helped surpass the shortage of computational resources, while seemingly not jeopardizing the viability of the agent. Although finding the best reward function was a long trial-and-error process, experiments made throughout this work suggested that multiple functions could turn out similarly good results.

CARLA offers not only a vast range of tools to develop self-driving autonomous vehicles but also a good support network for problem-solving and software improvement. Working with the CARLA simulator, however, proved to be one of the **biggest challenges** of this thesis. Configuring it to perform the necessary tasks was a slow and difficult process, but the biggest setback was the **dispersion** observed in the data acquired. This compromised the performance of the agent. Future works

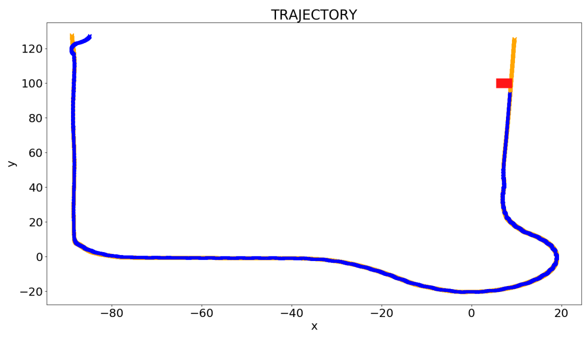


Figure 11: Full system test: Trajectory - the reference and the trajectory are superimposed

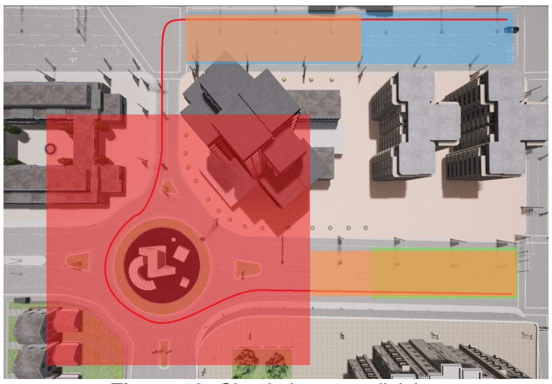


Figure 12: Simulation map division

include: **A solution to working with the dispersion** observed in the data acquired from the simulator: exploring longer in a vaster action space, achieved by lowering the learning rate and increasing the grain of gain tuning; **Implementing a Neural Network** to be used to identify the map zone the vehicle is currently in; Upgrading the Q-learning algorithm to a **Double Q-learning** algorithm, with the only downside of doubling the memory requirements; **Real-world application**.

References

[1] S. Bari, S. Shabih Zehra Hamdani, H. Ullah Khan, M. ur Rehman, and H. Khan. *Artificial Neural Network Based Self-Tuned PID Controller for Flight Control of Quadcopter; Artificial Neural Network Based Self-Tuned PID Controller for Flight Control of Quadcopter*. 2019.

[2] A. G. Barto and S. Mahadevan. Recent Advances in Hierarchical Reinforcement Learning. *Discrete Event Dynamic Systems*, 13(4), 2003.

[3] M. Bojarski, D. Del Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, X. Zhang, J. Zhao, and K. Zieba. End to End Learning for Self-Driving Cars. 4 2016.

[4] A. Dosovitskiy, G. Ros, F. Codevilla, A. Lopez, and V. Koltun. CARLA: An Open Urban Driving Simulator. 11 2017.

[5] H. Hasselt. Double Q-learning. In J. Lafferty, C. Williams, J. Shawe-Taylor, R. Zemel, and A. Culotta, editors, *Advances in Neural Information Processing Systems*, volume 23. Curran Associates, Inc., 2010.

[6] A. Hussein, M. M. Gaber, E. Elyan, and C. Jayne. Imitation Learning. *ACM Computing Surveys*, 50(2), 6 2017.

[7] B. R. Kiran, I. Sobh, V. Talpaert, P. Mannion, A. A. A. Sallab, S. Yogamani, and P. Perez. Deep Reinforcement Learning for Autonomous Driving: A Survey. *IEEE Transactions on Intelligent Transportation Systems*, 2021.

[8] P. Kofinas and A. I. Dounis. Fuzzy Q-learning agent for online tuning of PID controller for DC motor speed control. *Algorithms*, 11(10), 10 2018.

[9] S. Kuutti, R. Bowden, Y. Jin, P. Barber, and S. Fallah. A Survey of Deep Learning Applications to Autonomous Vehicle Control. 12 2019.

[10] Luigi Glielmo. Vehicle-to-Infrastructure Control: Grand Challenges for Control. Technical report, IEEE Control Systems Society, 2011.

[11] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu. Asynchronous Methods for Deep Reinforcement Learning. 2 2016.

[12] M. Quigley, B. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler, and A. Ng. ROS: an open-source Robot Operating System. Technical report.

[13] Q. Shi, H. K. Lam, B. Xiao, and S. H. Tsai. Adaptive PID controller based on Q-learning algorithm. *CAAI Transactions on Intelligence Technology*, 3(4), 2018.

[14] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, 2nd edition, 2018.

[15] C. J. C. H. Watkins and P. Dayan. Q-learning. *Machine Learning*, 8(3-4), 5 1992.

[16] D. Zhao, B. Wang, and D. Liu. A supervised Actor–Critic approach for adaptive cruise control. *Soft Computing*, 17(11), 11 2013.