Deep Reinforcement Learning based control algorithms: Training and validation using the ROS Framework in CARLA Simulator for Self-Driving applications

Óscar Pérez-Gil[1], Rafael Barea[1], Elena López-Guillén[1],
Luis M. Bergasa[1], Carlos Gómez-Huélamo[1], Rodrigo Gutiérrez[1], Alejandro Díaz[1]

***Abstract*— This paper presents a Deep Reinforcement Learning (DRL) framework adapted and trained for Autonomous Vehicles (AVs) purposes. To do that, we propose a novel software architecture for training and validating DRL based control algorithms that exploits the concepts of standard communication in robotics using the Robot Operating System (ROS), the Docker approach to provide the system with portability, isolation and flexibility, and CARLA (CAR Learning to Act) as our hyper-realistic open-source simulation platform. First, the algorithm is introduced in the context of Self-Driving and DRL tasks. Second, we highlight the steps to merge the proposed algorithm with ROS, Docker and the CARLA simulator, as well as how the training stage is carried out to generate our own model, specifically designed for the AV paradigm. Finally, regarding our proposed validation architecture, the paper compares the trained model with other state-of-the-art traditional control approaches, demonstrating the full strength of our DL based control algorithm, as a preliminary stage before implementing it in our real-world autonomous electric car.**

I. INTRODUCTION

Nowadays, Autonomous Vehicles (AVs) are considered as one of the greatest challenges in the automotive industry, they are expected to play a key role [3] to solve the most common traffic and transportation problems, such as traffic jams or accidents. In the last decades, some of the most famous Intelligent Vehicles (IVs) challenges, such as the DARPA Urban Challenge or the Intelligent Vehicle Future Challenge (IVFC), have proven that autonomous driving can be a reality in the near future, demonstrating well-established hardware and software frameworks for ITS purposes [25].

Considering a typical AV architecture, the control layer consists of a set of processes that implements the vehicle control and navigation functionality. A well-defined control layer makes the vehicle robust regardless the varying environment situations, such as the traffic participants, weather conditions or traffic scenario, on the premise of guaranteeing vehicle stability and covering the route provided by the other layers of the vehicle.

In this context, Artificial Intelligence (AI) is increasingly being involved in processes such as detection, Multi-Object Tracking (MOT) and environment prediction. DRL based algorithms are used to solve Markov Decision Processes (MDPs), where the scope of the algorithm is to calculate the optimal policy of an agent to choose actions in an environment with the goal of maximize a reward function, obtaining quite successful results in fields like solving computer games or simple decision-making system [18]. In terms of autonomous driving, DRL approaches have been developed to learn how to use the vehicle onboard sensors [11].

The scope of this paper is to formulate, train and validate the Deep Deterministic Policy Gradient (DDPG) [14] algorithm for AV purposes. To accomplish this task, we propose a novel software architecture for training and validating DRL algorithms that exploits the concepts of portability, isolation and flexibility in terms of software development using Docker [16] containers, standard communications in robotics using the Robot Operating Systems (ROS) and CARLA [7], a novel open-source autonomous driving simulator, featured by its hyper-realism, flexibility and real-time working. To the best of our knowledge, our DDPG implementation is the first Deep Learning based control pipeline that has been validated in a hyper-realistic open-source simulator using the ROS and Docker approaches. On top of that, though implementing and testing the algorithm to our real-world vehicle is beyond the scope of this paper (safety matters prevent these tests from being performed in real environments without a previous and exhaustive simulation stage). Our final goal is that the proposed model is exported to our electric vehicle [1] using a NVIDIA embedded system for that purpose. We hope that our distributed system can serve as a solid baseline on which future research can build on to advance the state-of-the-art in validating Deep Learning based control pipelines using hyper-realistic simulation, as a preliminary stage before implementing the algorithms in real-world prototypes.

The remaining content of this work is organized as follows. The next sections presents a comparison between Classic, Imitation Learning and Deep Reinforcement Learning based control algorithms in terms of autonomous driving. Section 3 studies the DDPG algorithm, proposes a novel software architecture in order to train and validate our model, highlighting the steps to merge the DDPG algorithm with ROS, Docker and the CARLA simulator, and formulates our Markov Decision Process (MDP) proposal for ITS purposes. Section 4 illustrates the proposed model performance obtain-

*This work has been funded in part from the Spanish MICINN/FEDER through the Techs4AgeCar project (RTI2018-099263-B-C21) and from the RoboCity2030-DIH-CM project (P2018/NMT- 4331), funded by Programas de actividades I+D (CAM) and cofunded by EU Structural Funds.

[1]All authors are with the Electronics Department, University of Alcalá (UAH), Spain {rafael.barea, elena.lopezg, luism.bergasa}@uah.es, {o.perezg, carlos.gomezh, rodrigo.gutierrez, alejando.diazd}@edu.uah.es

ing both qualitative and quantitative experimental results by comparing the fine-tuned DDPG method against other state-of-the-art control algorithms. Finally, Section 5 deals with the future works and concludes the paper.

II. RELATED WORKS

As mentioned in the previous section, several approaches for the control layer of an AV have been developed, which are commonly classified into classic controller and AI based controllers. The basics of control systems state that the transfer functions decides the relationship between the output and the input given the plant. Some of the most relevant algorithms used in the control layer are:

A. Classic controllers

Classic autonomous driving systems usually use advanced sensor for environment perception and complex control algorithms for safety navigation in arbitrarily challenging scenarios. Typically, these frameworks use a modular architecture where individual modules process information asynchronously. Regarding the control layer, some of most used control methods are PID, predictive control [12], Fuzzy Control [4], Adaptive control [27], Fractional-Order control [30], Pure-Pursuit (PP) path tracking control and the Linear-Quadratic Regulator (LQR) [9] algorithm. However, despite their good performance, these types of controllers are often environment dependent, so their corresponding hyperparameters must be properly fine-tuned according to the path to be followed in order to obtain the expected behaviour, which is not a trivial task to do.

B. Imitation learning

This approach tries to learn the optimal policy by following and imitating a expert system decisions. In that sense, an expert system (typically a human) provides a set of driving data [3], which is used to train the driving policy (agent) through supervised learning. The main advantage of this method is its simplicity, since it achieves very good results in end-to-end applications. Nevertheless, its main drawback is the difficulty of imitating every potential driving scene being unable to learn behaviors that have not been provided. This drawback causes this approach can be dangerous in some real driving situations that have not been previously observed.

C. Deep Reinforcement Learning

Reinforcement learning (RL) algorithms have been sucessfully tested for solving Markov Decision Problems (MDPs) and the combination of Deep Learning techniques and RL algorithms have demonstrated its potential solving some of the most challenging tasks of autonomous driving, such as decision making and planning [28]. Deep Reinforcement Learning (DRL) algorithms include: Deep Q-learning Network (DQN) [15], Double-DQN, actor-critic (A2C, A3C) [13], Deep Deterministic Policy Gradient (DDPG) [26] and Twin Delayed DDPG (TD3) [29]. Our approach, based on the DDPG algorithm, is explained in the following section.

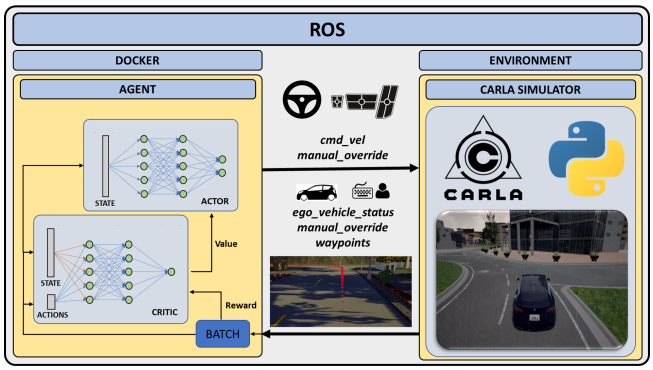

Fig. 1. Proposed architecture for training and validating DRL algorithms. On the left, the **Docker** image contains the Agent which is responsible of managing the observations received from the simulator, by using a DRL based network, and the CARLA ROS bridge to communicate with it. On the right, the **Environment** uses CARLA simulator to generate control commands. Bidirectional communications are managed by ROS.

III. DDPG-BASED ARCHITECTURE PROPOSAL

This section describes the Deep Deterministic Policy Gradient (DDPG) algorithm used as a baseline of our model, Fig. 1 shows our novel software architecture in order to train and validate different models in countless scenarios, illustrating the importance of hyper-realistic simulation to build safe AV technology based on these models.

A. Deep Deterministic Policy Gradient

Deep Deterministic Policy Gradient (DDPG) is a DRL algorithm that concurrently learns a Q-function and a policy. It uses off-policy data and the Bellman equation to learn the Q-function, where the Q-function is in charge of the policy. The algorithm pseudocode, adapted to AV purposes, is illustrated in Algorithm 1.

The algorithm that learns and take the decisions is known as the agent, which is interacting with the environment. The agent is continuously choosing actions a_i from an Action space $A = \mathbb{R}^N$ and a State space s_{t+1}, in such a way that a reward $r(s_t, a_t)$ is returned by the environment. The agent behaviour is governed by a policy (π) which plays as a state map in the action probabilistic distribution $\pi : S \rightarrow P(A)$ in a stochastic environment E.

The two main components in the policy gradient are the policy model and the value function. It makes sense to learn the value function and the policy model simultaneously, since the value function can assist the policy update by reducing the gradient variance in vanilla policy gradients, what is actually what the Actor-Critic method does. This method consists of two models (Critic and Actor), which may optionally share some parameters: While the Critic updates the value function parameters in function of the action-value, the Actor updates the policy parameters θ according to the suggestions of the Critic. In addition, DDPG includes the experience replay method that consists in keeping a buffer of past transitions available to update the algorithm with them. This technique not only boosts the learning process and increases the efficiency of the exploration [17], but also

it has proven to be vital for the stability of the learning process [6]. Updating the agent using past iterations allows to evaluate a single iteration several times with different policies, increasing the efficiency of the initial exploration. Moreover, to deal with the oscillations or even divergence in the policy value, we modify the DDPG features in order to emulate this Actor-Critic structure. Our modified version uses a soft update with $\tau << 1$ parameter to slowly update the policy parameters.

Algorithm 1 DDPG algorithm for AV purposes

Input: State vector $S = ([wp_{t_0}...wp_{t_N}], \phi_t, d_t)$
Output: Action vector $A = (throttle, steering)$
Init randomly the Critic $Q(s, a|\theta^\mu)$ and Actor $\mu(s|\theta^\mu)$ networks with weights θ^Q and θ^μ respectively.
Init the Critic and Actor objective networks Q' and μ' with weights
$\theta^{Q'} \leftarrow \theta^Q, \theta^{\mu'} \leftarrow \theta^\mu$.
Init iterations buffer R.
for *episode = 1, M* **do**
  Init random process N for action space exploration
  Receive observation State s_1
  for *t=1, T* **do**
    Select action $a_t = \mu(s_t|\theta^\mu)$ according to policy.
    Execute the action a_t and compute the reward r_t and the new State s_{t+1}.
    Store transition (s_t, a_t, r_t, s_{i+t}) in R.
    Sample a random minibatch of N transitions (s_t, a_t, r_t, s_{t+1}) in R.
    Being $y_i = r_i + \gamma Q'(s_{i-t}, \mu'(s_{i+t}|\theta^{\mu'})|\theta^{Q'})$, update Critic by minimizing loss:

$$L = (\tfrac{1}{N}) \sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$$

    Update the Actor policy using the gradient of sampling policy: $\nabla_{\theta^\mu} J$
    Update objective networks:

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$
$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

  end
end

B. Validation Architecture

Lately, hyper-realistic virtual testing is increasingly becoming in one of the most important concepts to build safe AV technology. The use of photo-realistic simulation (virtual development and validation testing) and an appropriate design of the driving scenarios are the current keys to build safe and robust AV. Regarding Deep Learning based algorithms (found in any layer of our architecture), the complexity of urban environments requires that these algorithms must be tested in countless environments and traffic scenarios. This issue causes that the cost and development time are exponentially increased using the physical approach. Some well-known simulators in the field of AV are NVIDIA DRIVE PX [2], Microsoft Airsim [24], V-REP [22], and CARLA. The latter, is currently one of the most powerful and promising simulators for developing and testing AV technology, based on Unreal engine and is of great importance in our approaches. Since this paper is framed in an open-source project, aimed at developing techniques for an automatic electric car new concept (AgeCar), able to assist senior drivers with different automation levels [21], we decided to use the open-source hyper-realistic CARLA simulator.

CARLA provides quite interesting features to develop and test self-driving architectures. However, regarding this work focused on the control layer, we highlight the following: 1. A Powerful PythonAPI, that allows the user to control all aspects related to the simulation, including weathers, pedestrian behaviours, sensors and traffic generation, 2. Fast simulation for planning and control, where rendering is disabled to offer a fast execution of road behaviors and traffic simulation for which graphics are not required, 3. Traffic scenarios simulation based on Scenario Runner and 4. ROS integration provided by the CARLA ROS Bridge.

This simulator is grounded on Unreal Engine (UE4) [23], one of the most opened and advanced real-time 3D creation tools nowadays and OpenDrive standard [8] is used to define the roads and urban settings, allowing CARLA to have an incredible realistic appearance. Regarding this, the simulator plays a crucial role in this paper for several reasons. First of all, it allows us to perform as many tests as required, avoiding putting lives or goods at risk as well as decreasing the development and cost time. It would be virtually impossible to carry out a project of this nature (training a DRL algorithm for AV purposes in arbitrarily complex scenarios) directly in a real environment, as it would represent a risk to both the ego-vehicle and its surrounding environment, specially at the beginning, due to the randomness of the first actions taken by the algorithm. Secondly, in order to validate the effectiveness of a control algorithm, it is mandatory to compare against the ideal route the vehicle should perform. In terms of the control layer, CARLA provides the user the actual odometry of the vehicle as well as the groundtruth of the route, what makes easier to evaluate the performance of the proposed system pipeline.

Considering this, we propose a novel software architecture for training and validating DRL algorithms for AV purposes. Fig. 1 illustrates the architecture, highlighting two different parts: On the left, it is appreciated the Agent, which is a Docker image based on Ubuntu18.04 that contains the Actor-Critic system pipeline. On the right, the Environment represents the simulators utils, that is, the CARLA simulator that provides the world and the Scenario Runner that specifies a specific map and traffic situation. The communication between the Agent and the Environment is done via ROS topics, thanks to the CARLA ROS bridge provided by CARLA and fine-tuned for our purposes. The Docker approach help us to conduct a CI/CD (Continuous Integration / Continuous Delivery), providing flexibility and isolation, encapsulating the algorithm and required dependencies for future real-

world purposes. In order to connect the Docker image with the host, we share the *host network* when creating writeable container layer over the specified image. As observed in Fig. 1, there are three main data that are sent from the Environment to the Agent: The **ego_vehicle_status**, which content is made up by the current vehicle position, collision, lane invasor status and current vehicle speed respectively, the **waypoints**, that contains the trajectory points list, and the **manual_override**, which represents a bidirectional signal that changes between manual and autonomous mode. On the other hand, from the Agent to the Simulator, we send the **cmd_vel**, that refers to the DRL network output, being the commanded velocity that have to play over the vehicle, and the current state of the **manual_override** signal.

C. MDP formulation

The final objective of the DDPG algorithm is to compute an action to send to the simulator as a commanded velocity. Considering that the problem of autonomous navigation can be modelled as a Markov Decision Process (MDP), to be able to generate these actions, the features of these processes must be considered. A MDP is a discrete-time stochastic control process that provides a mathematical framework for modeling decision making in situations where outcomes are partly random and partly under the control of a decision-maker. It can be represented as a 4-tupple (S, A, P_a, R_a) where the goal is to find a good policy (function $\pi(s)$) that the decision-maker will choose when is in a certain state s. Regarding this context, it corresponds to agent that observes the state (s_t) of the ego-vehicle (environment state) and generates an action (a_t), leading the vehicle to move to a new state (s_{t+1}) producing a reward $(r_t = R(s_t, a_t))$ based on the new observation. The four components of our formulated MDP tuple are explained below.

a) **State space (S)**: Information which is received from the Environment in each algorithm step. In our case, we model s_t as a tuple $s_t = (w_t, d_t)$ where w_t represents the waypoints vector associated to the current step, corresponding to the next N waypoints from the vehicle position and d_t represents the driving features vector (Fig. 2) made up by the vehicle speed estimation v_t, the distance to the center of the lane d_t and the angle between the vehicle and the centre of the lane ϕ_t, so $d_t = (v_t, d_t, \phi_t)$.

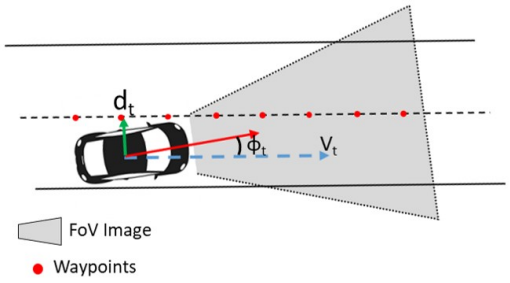

Fig. 2. Driving features vector illustration

In order to obtain these waypoints, we use the A* algorithm [10] that generates the global route at the beginning of each episode, taking the next N waypoints from the car position to form the state vector. Since we model our algorithm to process local waypoints (referred to our ego-vehicle), we transform their global coordinates to local ones using a homogeneous transformation matrix.

Then, the S State vector is made up by a composition from waypoints vector and driving features vector as follows:

$$S = ([wp_{t_0}...wp_{t_N}], \phi_t, d_t) \tag{1}$$

b) **Action space (A)**: Information which is send to the Environment to interact with the ego-vehicle in the simulator. It is required to send some command to the throttle, steering wheel and brake in a continuous way, which ranges are [0,1], [0,1] and [-1,1] respectively. Therefore, at each step the DRL agent must publish an action $(a_t) = (throttle_t, steer_t, brake_t)$ with the commands in the corresponding ranges. In this work, only the throttle and steering wheel are considered, representing the output of the DRL network.

c) **State transition function (P_a)**: Probability that an action a in state s at time t lead to state s_{t+1} at time $t+1$.

$$P_a = P_r(s_{t+1}|s_t, a_t) \tag{2}$$

d) **Reward function** $R_a(s_{t+1}, s_t, a_t)$: Function that generates the immediate reward of updating s_t to s_{t+1}. Since the goal in a MDP is to find a good "policy" $\pi(s) = a_t$ that will choose an action given a state, we use eq. 3 to maximize the expectation of cumulative future rewards.

$$E = \sum_{t=0}^{\infty} \gamma^t R(s_t, s_{t+1}) \tag{3}$$

D. Training Stage

Our proposal to build the DRL network is made up by two parts, the Agent and the Critic network. The Actor predicts an action based on the State and a Value that is obtained from the Critic. The Critic calculates this Value based on the State, the previous action obtained by the Actor and the reward associated with that action, as shown in Fig. 1. The architecture of Actor network consists of two Fully-Connected layers (300 and 600 hidden units and ReLu activation function respectively) that receives the State as input and an output layer formed by 2 neurons and $Tanh$ activation function, returning the commanded velocity and the automation state (manual or automatic). On the other hand, the Critic Network consists of two Fully-Connected layers (300 and 600 hidden units and ReLu and Linear activation functions respectively), receiving the State, previous action and reward as inputs and using a simple output obtains the Value from a Linear activation function.

We designed a simple but accurate training workflow:

1) Launch the simulator and iterate over M episodes and T steps.
2) At the beginning of the episode, call the A* based global planner to obtain the complete route from two random points on the map.

3) At each episode, take an observation corresponding to the State S by concatenating the next $Nwaypoints$ (as a subset of the total route) in local coordinates and the $drivingfeaturesvector$. The State $S = ([wp_{t_0}...wp_{t_N}], \phi_t, d_t)$ is introduced to the DRL network, which predicts the actions as output $A = (throttle, steering)$. Then, the predicted actions are sent to the simulator and the reward is calculated in function of this actuation.
4) The $lane\_invasor$ and $collision\_sensor$ topics are read in each step. If any of these sensors are active, the episode ends by sending the $manual\_override$ topic, which give back the control to manual mode. The vehicle must then be located in the centre of the lane and the topic enabled again so that the control starts learning again in another new episode. If these topics are not activated, the training process iterates over another new step.
5) The training stage finishes when the maximum number of episodes is reached.

IV. Experimental results.

Once the DRL algorithm is trained, it is evaluated and compared against other state-of-the-art control algorithms using the facilities provided by CARLA, that is, the actual trajectory driven by the vehicle and ideal route by interpolating the waypoints [9] provided by the A* based global planner. The goal of the algorithms is to follow the corresponding route as fast as possible avoiding collisions and road departures in an arbitrarily complex dynamic urban simulation environment. The metrics considered to evaluate the performance of the trained models are: The Root Mean Squared Error (RMSE) between the performed trajectory and ideal trajectory, the Maximum Error when driving the routes and the time spent by the vehicle to complete the navigation. Considering these metrics, a total of 20 trajectories have been randomly defined in the Town01 map of CARLA. Both training stage and experimental results have been developed using a desktop PC (Intel Core i7-9700k, 32GB RAM) with CUDA-based NVIDIA GeForce RTX 2080 Ti 11GB VRAM.

Table I shows the ablation study, illustrating our different DDPG based models trained using different input data sources, in order to select the best one to compare against other state-of-the-art control algorithms. DDPG-Im-Waypoints uses a binary image with segmented lane to calculate N waypoints by proccessing the image. DDPG-Flatten-Im flattens the binary image with the segmented lane. DDPG-PreCNN uses a pre-trained-CNN to obtain the waypoints from the on-board RGB image. DDPG-Waypoints is our final model, explained in Section 3. For a deeper explanation of our models we refer the readers to [19]. The ablation study shows that the Waypoints based proposal achieves the best performance on each evaluated metric, justifying its choice as baseline in the proposed architecture.

Based on the best model from the ablation study, we perform a comparison with other well-known control algorithms, not only classic control methods (LQR and Pure-Pursuit) but also other AI based control algorithm like the Deep Q-Network (DQN). The trajectories are again evaluated over 20 routes in the Town01 map. Table II demonstrates that the algorithm works in an optimal way, beating the other AI based method and the Pure-Pursuit method by a large margin, and achieving minimally differentiated results with respect to one of the best classic controllers as is the LQR algorithm, illustrating the performance of our proposed model and the architecture for training and validating DRL based algorithms.

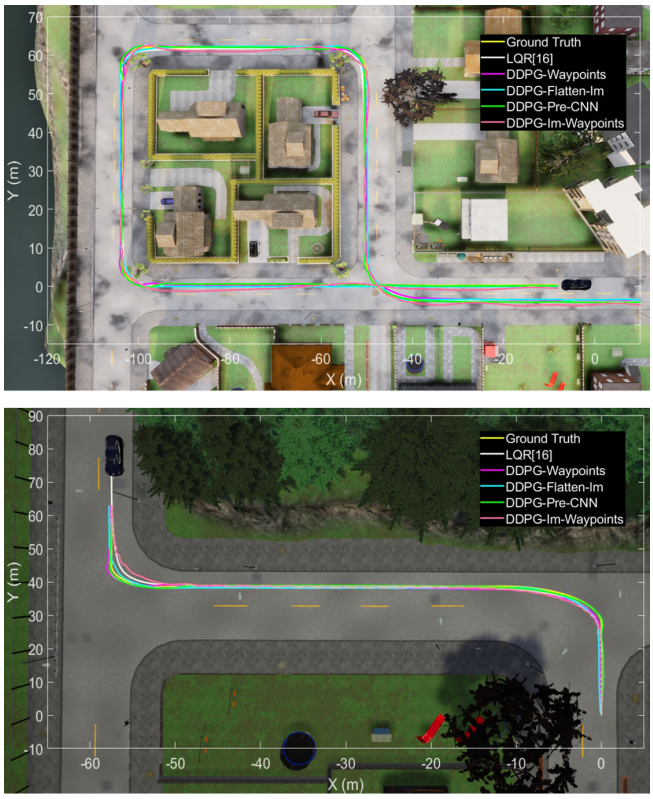

Fig. 3. Qualitative results on two different routes. Ground truth route (yellow); LQR-Controller (white); DDPG-Waypoints (magenta); DDPG-Flatten-Im (cyan); DDPG-Pre-CNN (green); DDPG-Im-Waypoints (pale pink).

TABLE I
Ablation study to build the State vector (S).

Method	RMSE (m)	ME (m)	Time (s)
DDPG-Im-Waypoints	0.295	2.3875	222.18
DDPG-Flatten-Im	0.134	1.522	63.97
DDPG-PreCNN	0.115	1.5125	65.12
DDPG-Waypoints(ours)	**0.10**	**1.46**	**62.25**

Figure 3 shows two qualitative results obtained when comparing the groundtruth (yellow line) calculated by interpolating the corresponding waypoints against the LQR method and the different DDPG proposals of the ablation study shown in Table I. It is observed how our proposal (DDPG based on waypoints) is able to complete the specified route in a way that is similar to the LQR controller.

TABLE II

DDPG INTEGRATED PROPOSAL VS OTHER APPROACHES COMPARISON. (TESTED OVER 20 DIFFERENT ROUTES IN TOWN01)

Method	RMSE(m)	MaxError(m)	Time(s)
DQN [20]	0.198	1.625	87.1
Pure-Pursuit[5]	0.20	1.747	80.7
LQR[9]	**0.095**	**1.305**	65.60
DDPG (ours)	0.10	1.46	**62.25**

V. CONCLUSIONS AND FUTURE WORKS

In this paper we have presented a Deep Reinforcement Learning (DRL) based control algorithm adapted to Autonomous Vehicles (AVs) purposes as well as a novel software architecture for training and validating DRL based algorithms integrating the open-source simulator CARLA, the Robot Operating System (ROS) and Docker. The proposed architecture demonstrates its robustness, effectiveness and manageability in both performing the training stage and its proper validation, obtaining quite similar results than LQR method, which one of the best classical controllers. However, the major advantage of our AI based algorithm, is that once a model is trained in a scenario, it can be directly reproduced on any other scenario, while classic controllers depends on a fine-tuning stage that is relatively complex depending on the environment in which it is located. We hope that our validation architecture will serve as a solid baseline in the state-of-the art of AVs validation using simulation environments testing. As future work, due to the modularity and portability of this approach provided by ROS and Docker, we plan to integrate the proposed algorithm in a NVIDIA embedded system in our real-world prototype, carrying out the fewest modifications required to make it possible.

REFERENCES

[1] J Felipe Arango, Luis M Bergasa, Pedro A Revenga, Rafael Barea, Elena López-Guillén, Carlos Gómez-Huélamo, Javier Araluce, and Rodrigo Gutiérrez. Drive-by-wire development process based on ros for an autonomous electric vehicle. *Sensors*, 20(21):6121, 2020.

[2] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, et al. End to end learning for self-driving cars. *arXiv preprint arXiv:1604.07316*, 2016.

[3] Jianyu Chen, Bodi Yuan, and Masayoshi Tomizuka. Deep imitation learning for autonomous driving in generic urban scenarios with enhanced safety. *arXiv preprint arXiv:1903.00640*, 2019.

[4] Rerngwut Choomuang and Nitin Afzulpurkar. Hybrid kalman filter/fuzzy logic based position control of autonomous mobile robot. *International Journal of Advanced Robotic Systems*, 2(3):20, 2005.

[5] R Craig Coulter. Implementation of the pure pursuit path tracking algorithm. Technical report, Carnegie-Mellon UNIV Pittsburgh PA Robotics INST, 1992.

[6] Tim De Bruin, Jens Kober, Karl Tuyls, and Robert Babuška. The importance of experience replay database composition in deep reinforcement learning. In *Deep reinforcement learning workshop, NIPS*, 2015.

[7] Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. Carla: An open urban driving simulator. *arXiv preprint arXiv:1711.03938*, 2017.

[8] Marius Dupuis, Martin Strobl, and Hans Grezlikowski. Opendrive 2010 and beyond–status and future of the de facto standard for the description of road networks. In *Proc. of the Driving Simulation Conference Europe*, pages 231–242, 2010.

[9] Rodrigo Gutiérrez, Elena López-Guillén, Luis M Bergasa, Rafael Barea, Óscar Pérez, Carlos Gómez-Huélamo, Felipe Arango, Javier Del Egido, and Joaquín López-Fernández. A waypoint tracking controller for autonomous road vehicles using ros framework. *Sensors*, 20(14):4062, 2020.

[10] Peter E Hart, Nils J Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[11] Alex Kendall, Jeffrey Hawke, David Janz, Przemyslaw Mazur, Daniele Reda, John-Mark Allen, Vinh-Dieu Lam, Alex Bewley, and Amar Shah. Learning to drive in a day. In *2019 International Conference on Robotics and Automation (ICRA)*, pages 8248–8254. IEEE, 2019.

[12] Roland Lenain, Benoit Thuilot, Christophe Cariou, and Philippe Martinet. Model predictive control for vehicle guidance in presence of sliding: application to farm vehicles path tracking. In *Proceedings of the 2005 IEEE international conference on robotics and automation*, pages 885–890. IEEE, 2005.

[13] Xiaodan Liang, Tairui Wang, Luona Yang, and Eric Xing. Cirl: Controllable imitative reinforcement learning for vision-based self-driving. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 584–599, 2018.

[14] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[15] V Matt and N Aran. Deep reinforcement learning approach to autonomous driving, 2017.

[16] Dirk Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux journal*, 2014(239):2, 2014.

[17] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[18] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

[19] Óscar Pérez-Gil, Rafael Barea, Elena López-Guillén, Luis M Bergasa, Carlos Gómez-Huelamo, Rodrigo Gutiérrez, and Alejandro Díaz. Deep reinforcement learning based control for autonomous vehicles in carla. *Multimedia Tools and Applications*, In revision, 2021.

[20] Óscar Pérez-Gil, Rafael Barea, Elena López-Guillén, Luis M Bergasa, Pedro A Revenga, Rodrigo Gutiérrez, and Alejandro Díaz. Dqn-based deep reinforcement learning for autonomous driving. In *Workshop of Physical Agents*, pages 60–76. Springer, 2020.

[21] Spain Robesafe group, University of Alcala. Techs4agecar project. *URL http://www.robesafe.uah.es/proyectos/tech4agecar/index.php*, 2019.

[22] Coppelia Robotics. V-rep user manual. *URL http://www. coppeliarobotics. com/helpFiles/. Ultimo acesso*, 13(04), 2015.

[23] Andrew Sanders. *An introduction to unreal engine 4*. AK Peters/CRC Press, 2016.

[24] Shital Shah, Debadeepta Dey, Chris Lovett, and Ashish Kapoor. Airsim: High-fidelity visual and physical simulation for autonomous vehicles. In *Field and service robotics*, pages 621–635. Springer, 2018.

[25] Fei-Yue Wang. Ai and intelligent vehicles future challenge (ivfc) in china: From cognitive intelligence to parallel intelligence. In *2017 ITU Kaleidoscope: Challenges for a Data-Driven Society (ITU K)*, pages 1–2. IEEE, 2017.

[26] Sen Wang, Daoyuan Jia, and Xinshuo Weng. Deep reinforcement learning for autonomous driving. *arXiv preprint arXiv:1811.11329*, 2018.

[27] Wei Wang, Kenzo Nonami, and Yuta Ohira. Model reference sliding mode control of small helicopter xrb based on vision. *International Journal of Advanced Robotic Systems*, 5(3):26, 2008.

[28] Ekim Yurtsever, Linda Capito, Keith Redmill, and Umit Ozguner. Integrating deep reinforcement learning with model-based path planners for automated driving. *arXiv preprint arXiv:2002.00434*, 2020.

[29] Fengjiao Zhang, Jie Li, and Zhi Li. A td3-based multi-agent deep reinforcement learning method in mixed cooperation-competition environment. *Neurocomputing*, 411:206–215, 2020.

[30] DJ Zhuang, F Yu, and Y Lin. The vehicle directional control based on fractional order pdˆ mˆ u controller. *JOURNAL-SHANGHAI JIAOTONG UNIVERSITY-CHINESE EDITION-*, 41(2):0278, 2007.