DQN-based Deep Reinforcement Learning for Autonomous Driving

Óscar Pérez-Gil, Rafael Barea, Elena López-Guillén, Luis M. Bergasa, Pedro Revenga, Rodrigo Gutiérrez, Alejandro Díaz

Electronics Department, Universidad de Alcalá, Spain
{o.perezg, rafael.barea, elena.lopezg, luism.bergasa, pedro.revenga, rodrigo.gutierrez, alejando.diazd}@uah.es

Abstract. The goal of this work is to evaluate the task of autonomous driving in urban environment using Deep Q-Network Agents. For this purpose, several approaches based on DQN agents will be studied. The DQN agent learn a policy (set of actions) for lane following tasks using visual and driving features obtained from sensors onboard the vehicle and a model-based path planner. The policy objective is to drive as fast as possible following the center of the lane avoiding collisions and road departures. A dynamic urban simulation environment will be designed using CARLA simulator to validate our proposal. The results show that a DQN agent can be a successful technique for self-driving a vehicle in a urban environment.

Keywords: Autonomous Driving · Reinforcement Learning · Convolutional Neural Network · Deep Q-Network Agent · CARLA Simulator

1 INTRODUCTION

In recent years, autonomous driving plays a pivotal role to solve traffic and transportations problems in urban areas (traffic congestions, accidents, etc) and it is going to change the way of travelling in our world in the future [1]. In the last decade, various challenges, such as the well-known DARPA Urban Challenge and the Intelligent Vehicle Future Challenge (IVFC) have proven that autonomous driving can be a reality in the near future. The teams participating in these events have demonstrated numerous technical frameworks for autonomous driving [2-5]. Nowadays, most self-driving vehicles are geared up with multiple high-precision sensors such as LIDAR and cameras. LIDAR-based detection methods provide accurate depth information and obtain robust results in location, object detection and scene understanding [6] while camera-based methods provide much more detailed semantic information [7]. Lately, Reinforcement Learning (RL) have been used to solve Markov Decision Problems (MDPs). This method tries to calculate the optimal policy of an agent to choose actions in an environment with the goal of maximize a reward function. The results obtained to solve computer games or simple decision-making system have been very successful [8,9]. Regarding to autonomous driving, recently, Deep RL approaches have been developed to learn how to drive using sensory system onboard the vehicle [10, 11].

In this paper, we are going to focus our study in the performance of Deep Q-Network agents for autonomous driving. The goal is to follow a route as fast as possible avoiding collisions and road departures in a dynamic urban simulation environment. The discrete nature of DQN makes a complex tune for a continuous problem like self-driving, due to the infinite possibles of movement by the car in each step. Studying DQN and the obtained results, we will be able to discuss whether this algorithm is the right one for this navigation purpose. On the other hand, an important part of this work, is the simulator that will be used, CARLA Simulator [12]. CARLA is a hyper-realistic simulator that helps in the development of navigation techniques and in the exportability of the simulated models to real systems. In addition, CARLA provides a bridge with ROS [13] for standardization, and a PythonAPI for easy programming. Both modules allow an easy connection between simulation and the algorithms implemented to control the vehicle. Finally, although working with a real vehicle is not the objective of this work at the moment (safety matters prevent these tests from being performed in real environments without a previous and exhaustive simulation stage), our final goal is that the developed control can be exported to our robotic platform.

2 RELATED WORK

In the last years, several notable approaches to solve autonomous driving challenge have been proposed. These approaches can be classified in three main types: classical or mapping-based, imitation learning (IL) and reinforcement learning (RL).

a) **Classical methods**: Classical autonomous driving systems usually use advanced sensor for environment perception and complex control algorithms for safety navigation in complex scenarios. Typically, these frameworks use a modular architecture where individual modules process information asynchronously. The sensing module captures information from the surroundings using different sensors such cameras, LiDAR, GPS, IMUs, etc. Perception module plays a high-priority role and is responsible for calculating the position of the vehicle and recognizing and detecting the objects present in the surroundings of the environment. Decision module relies on perception module. Once the module understands the behaviour of the scene, it makes the appropriate decision according to the external conditions and the established objective [14,15].

b) **Imitation learning**: This approach tries to learn the optimal policy by following and imitating the expert's decisions. This way, an expert (typically a human) provides a set of driving data [15,16]. Using these labelled data, the driving policy (agent) is easy to train in a supervised learning. The main advantage of this method is its simplicity and it achieves very good results in end-to-end applications. On the other hand, 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 observed.

c) **Reinforcement learning**: This approach is a type of machine learning technique that enables an agent to learn an interactive environment by trial and error using feedback from its own actions and observations.This approach have been successfully tested for solving Markov Decision Problems (MDPs). In recent years, it has been combined with deep learning techniques and have proved its potential to solve autonomous driving problems such as decision making and planning . Deep Reinforcement Learning (DRL) algorithms include: Deep Q-learning Network (DQN), Double-DQN, actor-critic (A2C, A3C), Deep Deterministic Policy Gradient (DDPG) and Twin Delayed DDPG (TD3). All these algorithms have been used to perform autonomous driving tasks, obtaining promising results [18-22].

3 FRAMEWORK OVERVIEW

This section shows the proposed framework for autonomous driving based on DRL (Fig.1). This approach is based on [18] where a hybrid of a model-based planner and a model-free DRL agent is proposed. The general architecture is based on CARLA simulator, which provides a hyper-realistic and dynamic urban simulation environments. CARLA provides a powerful API that allows the users to control all aspects related to the simulation and permits to configure diverse sensors for environment perception (cameras, LiDAR, GPS, etc). This way, CARLA provides environment and ego-vehicle data. With this data (monocular camera image, and vehicle position and speed), it is possible to obtain visual and driving features. This two set of features can be introduced in a Deep Learning Reinforcement Agent to generate control commands (actions).

4 METHOD

Autonomous driving tasks can be modeled as a Markov Decision Process (MDP) and a great amount of reinforcement learning algorithms have been developed recently for solving MDP [10,19]. Therefore, Our approach aims to develop a DRL agent that generates autonomous vehicle control actions.

4.1 MDP formulation

A MPD consists of an agent that observes the state (s_t) of the ego-vehicle (environment state) and generates an action (a_t). This causes 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. A Markov decision process is a 4-tuple (S, A, P_a, R_a) where the goal is to find a good "policy", that is, a function $\pi(s)$ that the decision maker will choose when is in state s.

a) **State space (S)**: This term refers the information which is received from the environment in each algorithm step. In our case, we model s_t as a tuple

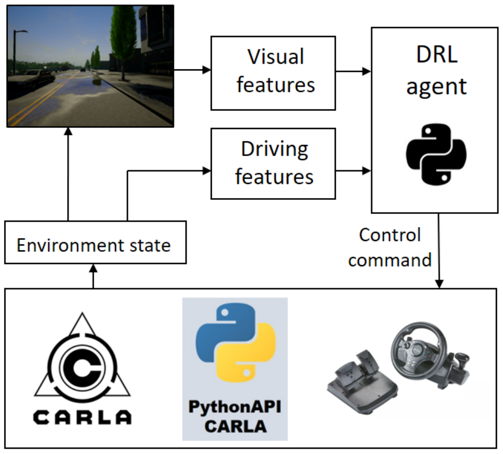

Fig. 1. Framework overview

$s_t = (vf_t, df_t)$ where vf_t is the visual features vector associated to the image I_t or a set of visual features extracted from the image, typically a set of waypoints w_t obtained using a model-based path planner $vf_t = f(I_t, w_t)$. df_t is the driving features vector consisting of an estimation of vehicles speed v_t, distance to the center of the lane d_t and angle between the vehicle and the centre of the lane ϕ_t, $df_t = (v_t, d_t, \phi_t)$. Fig.2 shows the state space where the waypoints are published by CARLA from the planning module.

b) **Action space (A)**: To interact with the vehicle available in the simulator, the commands for throttle, steering and brake must be provided in a continuous way. Throttle and brake range is [0,1] and steering range is [-1,1]. Therefore, at each step the DRL agent must publish an action $(a_t) = (acc_t, steer_t, brake_t)$ with the commands between the commented ranges.

c) **State transition function** (P_a) is the probability that action a in state s at time t will lead to state s_{t+1} at time t+1. $P_a = P_r(s_{t+1}|s_t, a_t)$.

d) **Reward function** $R(s_{t+1}, s_t, a_t)$: This function generates the immediate reward of translating from s_t to s_{t+1}. The goal in a Markov decision process is to find a good "policy" $\pi(s) = a_t$ that will choose an action given a state. This function will maximize the expectation of cumulative future rewards.

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

4.2 Deep Q-Learning algorithm for Reinforcement Learning

In recent years, there have been important advances in machine learning. Reinforcement Learning techniques have proved that an agent can learn robust

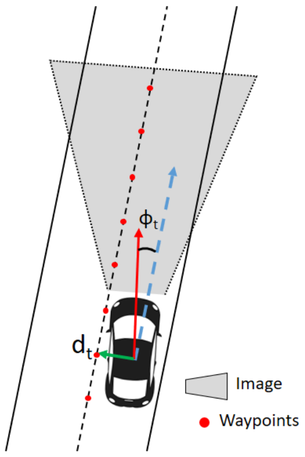

Fig. 2. State space.

policies in an interactive environment. In this work we are going to focus our effort in DQN [23] to solve our MDP problem.

Q-Learning. This process creates an exact matrix for the agent to maximize its reward in the long run. This approach is only practical for restricted environment, with limited space for observation, due to a high number of states or actions causes a wrong algorithm behaviour. Q-learning is an off-policy, model-free RL based on the Bellman Equation.

The goal of Q-learning is to maximize the Q-value though policy iteration, which runs a loop between policy evaluation and policy improvement. Policy evaluation estimates the value function V (equivalent to the referenced reward function (R)) with the greedy policy which has been obtained from the last policy improvement. On the other hand, policy improvement updates the policy with the action that maximize V function for each of state.

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha[r_{t+1} + \gamma maxargQ(s_{t+1}, a_t) - Q(s_t, a_t)] \tag{2}$$

Deep Q-Learning. Q-learning is lack of generality when space of observation increases. Imagine one situation with 10 states and 10 possible actions, we will need a 10x10 Q-matrix. If now the number of states increases to 1000, the Q-matrix increases too. To solve this issue, Deep Q-Learning get rid of the two-dimensional array by introducing a Neural Network.

So now, DQN estimates the Q-value by using a Neural Network, where the state is used as input, and the output is the corresponding Q-value for each action. The difference between D-Learning and Deep Q-Learning lies in:

$$y_j = r_j + \gamma maxargQ(\phi_{j+1}, a'; \theta^-) \tag{3}$$

Where ϕ is equivalent to the state s, while the θ stands for the parameters in the Neural Network. A detailed explanation of DQN algorithm can be found in [21].

5 DEEP Q-NETWORK AGENTS

We have developed various agents that cover a wide variety of model architectures for the Deep Q-Network agents. Models will be first developed in simulation for safety reasons. Therefore, the agent will interact with CARLA [24] and the code will be programmed in Python based on several open-source RL frameworks [25, 18] (see Fig.3) .

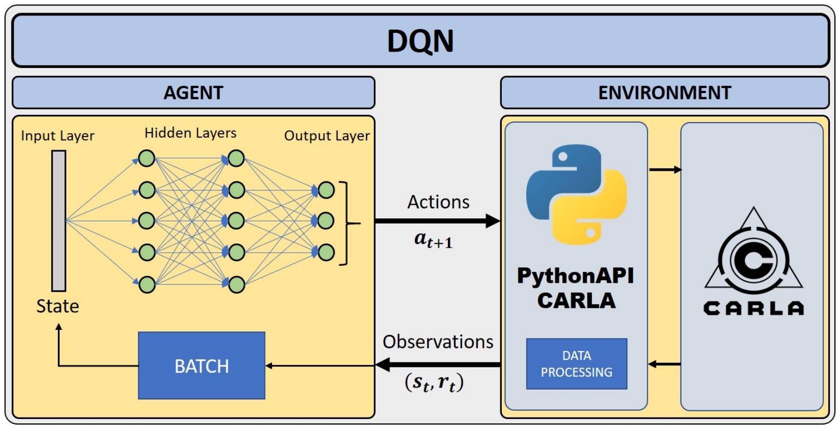

Fig. 3. Reinforcement Learning architecture DQN-based.

CARLA is an hyper-realistic simulator which provides a complete set of maps, in addition to the possibility of creating your own maps and import them into it. Fig. 10 depicts the map corresponding to ”Town01”. CARLA also provides a PythonAPI from which different actors (vehicles or pedestrians) in the simulator can be created and controlled by an external python agent. Different types of sensors are allowed to be attached to these actors, many of them will be used in this work, such as collision sensor, odometry, or camera image, being this last the most important for our application.

The cameras can be fixed to a vehicle using its centre as reference point, so that is placed in a position relative to the vehicle’s position by using a set of coordinates (x, y, z, pitch, yaw and roll). Moreover, there is a fourth post-processing effect available for some specific cameras like raw depth or semantic segmentation.

Using this PythonAPI, points belonging to the map can be obtained easily, as well as actual vehicle position and speed. These points are called spawn points and they are determined by how the map was created. But what is really important is that CARLA also provides waypoints, which do not depend on how the map was built, and can be used for navigation. The simulator contains a path planning algorithm, both global and local. The global route planner is based on A* algorithm, and is able to built a route between two map points and returning the set of waypoints that joins them, forming a path. Furthermore, thanks to the debugging tools, this path can be drawn in CARLA scene as shown in Fig. 4.

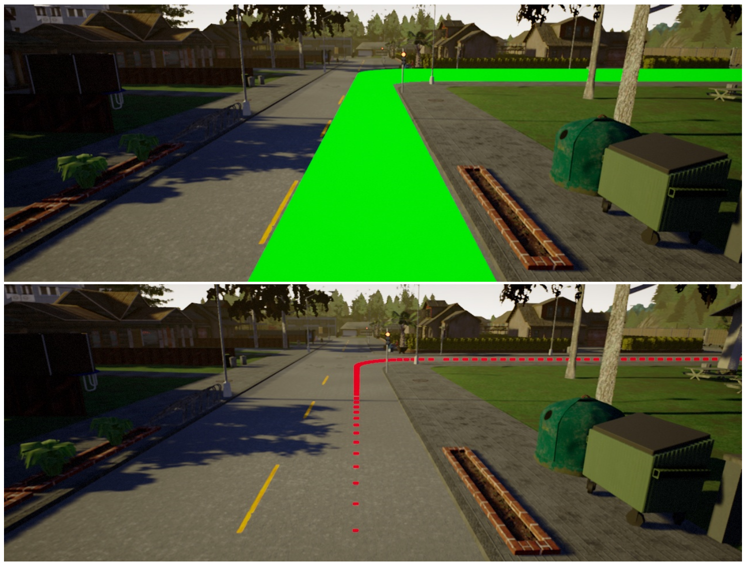
Fig. 4. Route with waypoints.

With all these facilities, some parameters have to be determined for the correct Trajectory with waypoints

a) **Reward function**. The proposed architecture obtains a driving features vector $df_t = (v_t, d_t, \phi_t)$ from the simulator. This vector is composed of the velocity of the vehicle in the direction of its heading v_t , the distance to the center of the lane d_t and the angle regarding the lane direction ϕ_t. Considering that the objective is to go as fast as possible through the center of the lane without leaving the lane and avoiding collisions, the reward function rewards the longitudinal velocity and penalize the transverse velocity and divergence from the center of the lane. This approach is similar to the proposal made in [torcs].

$$R = \sum_t |v_t cos\phi_t| - |v_t sin\phi_t| - |v_t||d_t| \text{ if car in lane} \tag{4}$$

$$R = -200 \text{ if collision or lane change or roadway departure} \quad (5)$$

b) **Control commands (actions)**. CARLA needs control commands for steering [-1,1] and throttle [0,1]. Brake has not been implemented in this first version because the environment is free of obstacles and the regenerative braking of the vehicle is enough to stop the vehicle. The DQN policy allows generating discrete actions, so it is necessary to simplify the continuous control of actions to a discrete control. Taking this into account, the number of control commands has been simplified to a set of 27 discrete driving actions, discretizing steering angle and throttle position in an uniform way. Table 1 shows the set of control commands where there are 9 steering wheel positions and 3 throttle position.

Table 1. Policy network. 27 classes.

Control commands		
Classes	Steering	Throttle
27	**-1,-0.75,...0.75,1**	**0,0.5,1**

5.1 DQN ARCHITECTURES

This section explains different implemented model architectures. The main difference among them are the features extraction methods. All agents have been programmed using Keras [26], which is a high-level neural networks library, that runs on the top of TensorFlow (open source platform for machine learning) [27].

DQN-CNN Agent. DQN-Convolutional Neural Network is an ambitious agent model, as it covers the whole problem straightforwardly, as shown in Fig. 5. This agent takes data from an image as visual features $vf_t = I_t$ and a set of parameters obtained from the vehicle as driving features $df_t = (v_t, d_t, \phi_t)$. Camera image is connected into a CNN and its output feeds a Fully-connected module, whereas the driving features are directly connected into the Fully-connected module. What is expected to have as CNN output are the road features at that moment, serving as a kind of waypoints for the training of the DRL. Then, both road and driving features are concatenated and introduced on set of Fully-Connected layers from which final action would be obtained. The CNN consist of three convolutional layer with 64 filters of size [7x7], [5x5] and [3x3] respectively. All layers use RELU as activation function. Each layer is followed by an average pooling. The output of the last convolutional layer is flattened and concatenates with driving features vector and fed into 2-fully-connected layers with 300 and 600 hidden units respectively. Finally a classification layer with linear activation outputs the predicted Q value.

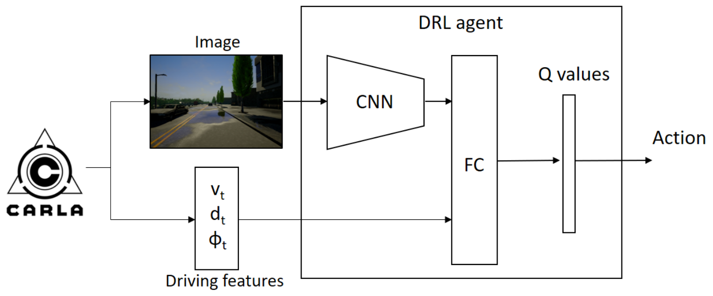

Fig. 5. DQN-CNN Agent.

DQN-FC agent. DQN-Fully-Connected agent differs the previous one in the CNN suppression. In this case, the input of the agent are the waypoints provided by the global planner of CARLA. Once A* algorithm is called, the complete path can be used, so only the next 14 waypoints from actual vehicle position will be introduced into the Fully-connected layers, so as the car moves the waypoints that feed the network will be updated accordingly. Although global planner's waypoints contain 3 position and 3 rotation components, only X and Y values will be used (referenced to the local map of the vehicle). This is due to only variations regarding the lane plane are taken into account. As in the previous agent, this vector will be concatenated with the driving features vector into the Fully-connected layer in order to obtain the action to take (see Fig.6).

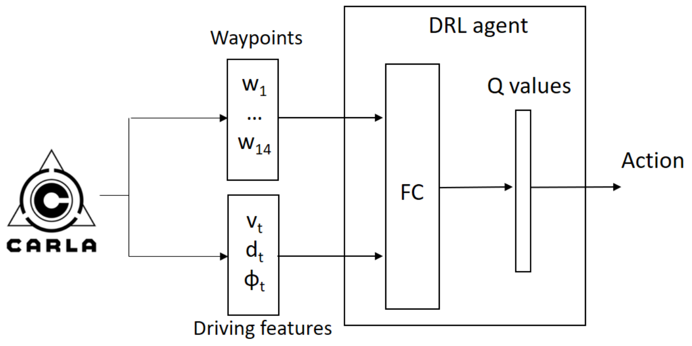

Fig. 6. DQN-Fully-Connected Agent.

DQN-Flatten-Image agent. This approach is built as simplification of DQN-CNN agent and is based on converting the RBG image into a resized B/W image, where the navigation path is segmented in a pre-processing stage, so that this image is flattened and converted into visual features vector. The visual vector is concatenated with driving features vector and this data is inserted into a DQN-Fully-Connected agent (see Fig.7).

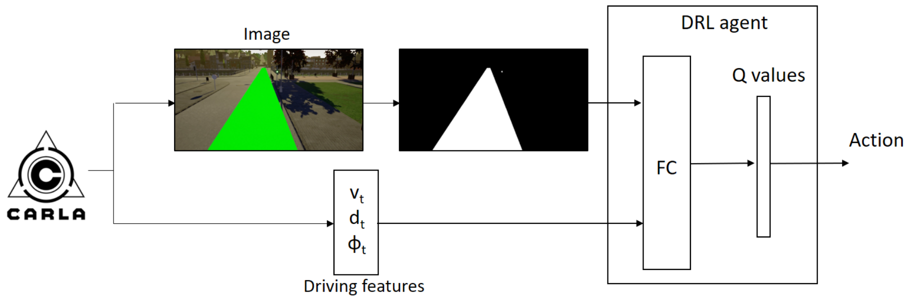

Fig. 7. DQN-Flatten-Image Agent.

DQN-PilotNet agent. Due to the really hard task of getting a good model training the DQN-CNN agent, we decided to separate the problem into two stages: the first one to obtain waypoints from the images and the second one to train the DQN from the waypoints. This architecture was built with the main objective of obtaining the waypoints through a CNN, but numerous networks can be used for this purpose not being necessary to design an ad-hoc network from scratch. A known network model that works with images is ”PilotNet” [28], which was created with the purpose of obtaining the road angle and vehicle speed from the track image. So, assuming a proper PilotNet behaviour, some necessary changes in its architecture are made in Python, and thanks to ”Tensorflow” and ”Keras” libraries, a well trained model is achieved (see Fig.8)

A PilotNet architecture as shown in Fig.9 is able to return the same waypoints that CARLA would provided thought PythonAPI from its planner. This implies having an independence of third party algorithms to obtain the waypoints and in a synchronized way with the DRL Net. Obviously, to work with this network it is necessary a pre-training stage with data recorded from the simulator. However, this is much simpler and shorter than a complete training for the DQN-CNN agent. The remaining agent is identical to the one implemented in DQN-FC agent.

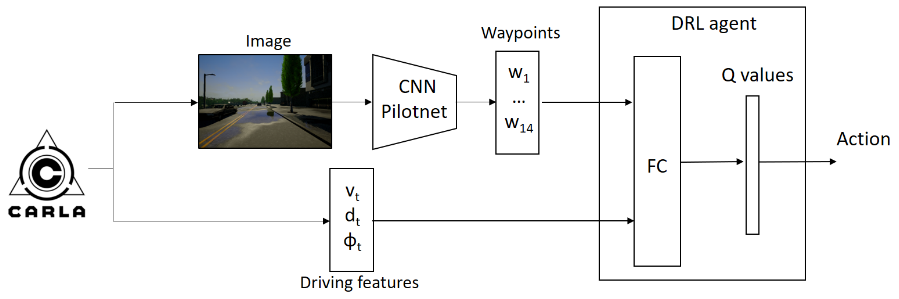

Fig. 8. DQN-PilotNet Agent.

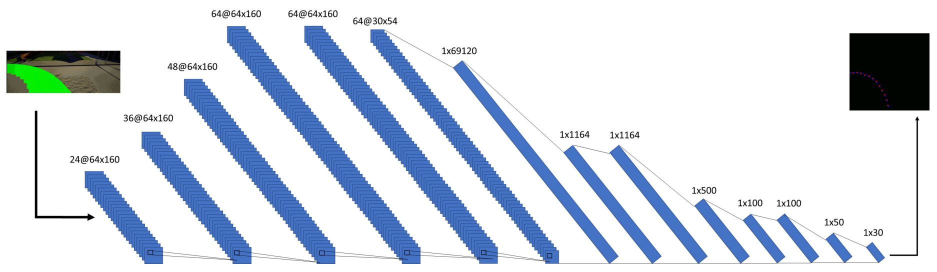

Fig. 9. CNN's architecture based on PilotNet modified to obtain waypoints to feed the Deep Reinforcement Learning training. A 160x60 RGB image is provided to the PilotNet to return a (15, 2)-shaped waypoint vector.

6 RESULTS

To test the performance of the implemented DQN Agents, we compare results with other navigation modes such as manual driving (hand-crafted) and autonomous driving using a Classical Waypoint Tracking Controller based on the LQR [29].

The experiment carried out consist on learn how to drive in a trajectory where a origin-destination pair is selected from the "Town01" of CARLA so that the trajectory includes straight sections and curves to the left and right as it is shown in Fig. 10. All the training and testing experiments have been developed using a PC (Intel Core i7-9700, 32GB) with Nvidia GeForce RTX 2080 and using CUDA-based GPU.

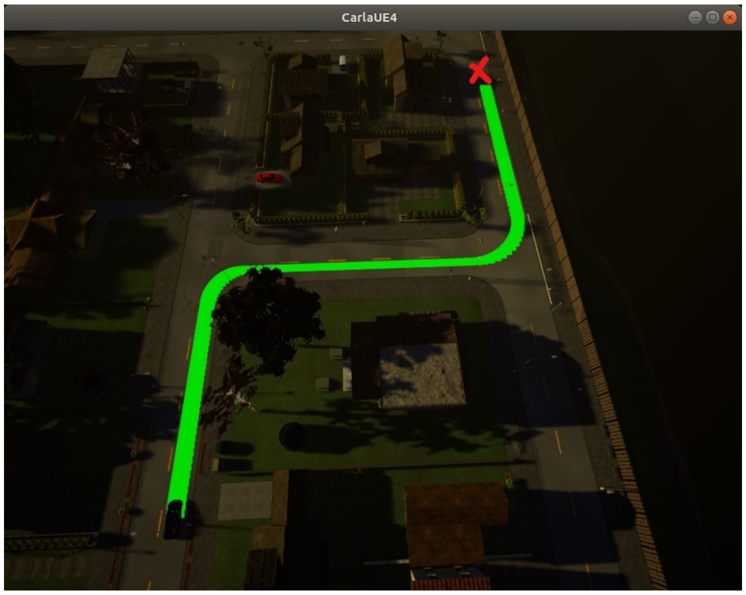

Fig. 10. Navigation-trajectory on CARLA's Town01 scene obtained by Global Path Planning A* algorithm provided by Carla PythonAPI.

The following steps have been developed in the training process:

1) At the beginning of each episode, the CARLA Python-API generates a path using a global planner (origin, destination and waypoints).

2) Each timestamps (step), the corresponding visual and driving features vector are obtained, which are introduced in the DQN agent.

3) In each timestamp of the DQN training process, the DQN weights are updated, the reward is calculated and the Q matrix is updated.

4) The episode ends when the destination is reached or a collision or lane departure occurs.

5) Throughout the process, the accumulated reward is calculated. This value will be used to determine which episode has made the best trajectory (best-reward-episode).

6) Training ends when the maximum number of episodes is reached.

Table 2 shows the results obtained in DQN agents training process. The number of training episodes is 20,000 with the exception of the DQN-CNN where it has been expanded to 120,000. This is so because we take RGB images directly as input instead of waypoints, which supposes a much larger order of data, and consequently, requires a higher number of iterations for the system to converge.

Table 2. Training performance

Model	Training Episodes	Best Episode
DQN-FC	**20000**	**8300**
DQN-FLATTEN-IMAGE	**20000**	**16500**
DQN-PILOTNET	**20000**	**13200**
DQN-CNN	**120000**	**108600**

As you can see, all agents reach the final destination in the established episodes, although this does not guarantee that the performed trajectory is adequate. Those DQNs that reached their best-reward-episode earlier (shorter learning time) are DQN-FC and DQN-PILOTNET with 8300 and 13200 episodes respectively. These two models are the ones that use a set of waypoints as visual feature vector as input to the agent's full-connected layer. The other agent that also reaches its best-reward-episode before reaching the maximum number of episodes is the DQN-Flatten. In this case the RGB image information is reduced, converted to b/n and flattened. Finally, the most complex model, the DQN-CNN requires many more episodes to achieve a correct result. In this case it has taken 108,600 episodes. To summarize, those DQN agents with a simpler features learn to drive much faster than those with more complex visual features.

Once the training process of the different DQN agents is completed, the objective is to compare how well the trajectory performed by each one fits to an ideal driving following the center of the lane (ground-truth). To do this, the different DQNs will be compared with each other and also with manual driving (hand-crafted) and an autonomous driving method using by the authors in its real prototype [28,29].

Fig.11 shows the trajectories followed by the different DQN agent as well as the manual and classical models. As can be seen, all approaches are close the ideal path and manage to reach the final destination. In general terms, it can be seen that better results are obtained using classical methods. DQN-based approaches show greater oscillations when following the ideal path and present higher errors compared with the classical approach. DQN-CNN and DQN-Flatten agents that use an image as visual feature vector obtain a trajectory with more oscillation and move further away from the ideal path. On the other hand, DQN-FC and

DQN-Pilotnet follow a trajectory that is much closer to the ideal and have hardly any oscillations.

To get a more quantitative idea of errors made with respect to the ground-truth (obtained interpolating by means of a spline-line the route points provided by CARLA) we calculate the mean square error (MSE) and the maximum error along the whole path. Numbers are depicted in Table 3.

The best result are obtained by Waypoint Tracking Controller based on LQR Optimal Controller with an MSE of 6 cm, being the maximum distance to the ground-truth 0.74 m in the middle of the first curve.

DQN-FC and DF-Pilonet, which use waypoints as inputs present an small error ranging from MSE=0.21 m to MSE=0.33 m. These two agents correctly follow the ideal path in straight sections, but in curves they have maximum errors of 1.32 m and 1.72 m respectively. With these errors and taking into account the lane width is 3.5m both models model can be applied for self-driving a car.

On the other hand, DQN-Flatten and DQN-CNN present higher errors, ranging from MSE=0.64 m for DQN-Flatten to MSE=0.83 m for DQN-CNN. Although the mean error may be acceptable, the maximum error reach 3.15 m and 2.15 m on curves, which are unacceptable because generates road departures.

It must be remarked that the manual driving gets a MSE=0.4 m and a maximum error in curves of 0.74 m. In consequence, if we compare autonomous driving behaviours with the performed by a human errors would be lesser.

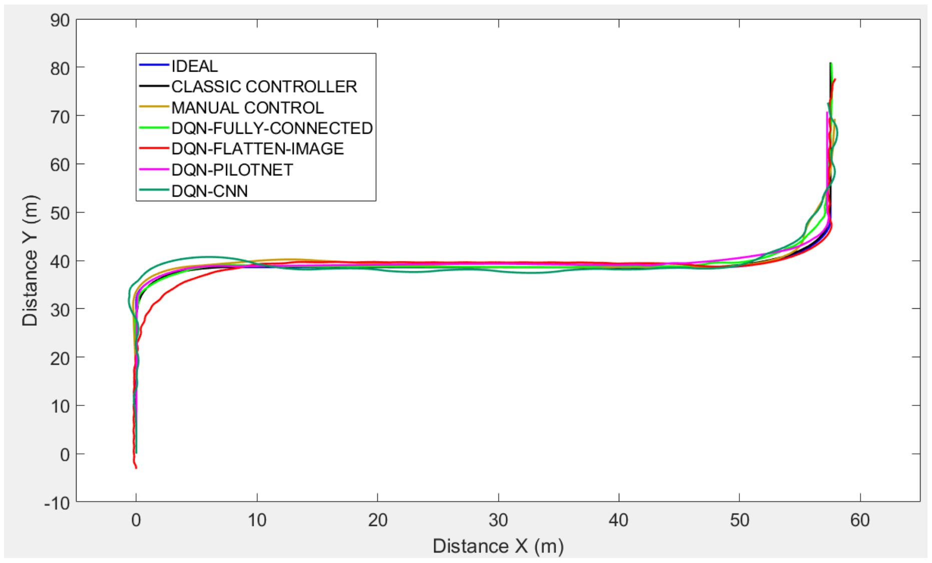

Fig. 11. Trajectories-comparison.

Summarising, although all the implemented agents have reached the destination, it is observed that the agents that use directly images as inputs obtain

Table 3. Lateral Mean Square Error.

Model	Mean Square Error	Maximum Error
CLASSIC CONTROLLER	**0.06 m**	**0.74 m**
MANUAL CONTROL	**0.40 m**	**1.80 m**
DQN-FC	**0.21 m**	**1.32 m**
DQN-PILOTNET	**0.33 m**	**1.72 m**
DQN-FLATTEN-IMAGE	**0.64 m**	**3.15 m**
DQN-CNN	**0.83 m**	**2.15 m**

worse results (greater errors following the center of the lane) than those that work directly with waypoints obtained from a external global planner or generated from pre-processing of the images using a CNN. This is due to work with images is more complex (there are more inputs to the neural network) and requires much more agent training time to obtain similar results.

7 CONCLUSIONS

In this work, an approach for integrating some Deep Reinforcement Learning algorithms into a navigation system was proposed, more specifically DQN-based algorithms. In spite of the results shown in previously section where each proposed model is able to follow a certain trajectory to a greater or lesser extent, DQN algorithm presents several limitations for this purpose.

One of them is the real difficulty to tune the necessary outputs of the algorithm, as well as its discrete values related to the actions to take by the simulated car, to get a model that performs a complete trajectory. This is due to the discrete character of the DQN algorithm itself. Several factors depends on the amount of selected classes as shown in Table 1. The higher the number of actions, the higher the training time of the model anf the better the performed trajectory. This parameter depends very much of the map where the vehicle has to travel.

Other limitation is the training time, controlled by Epsilon parameter and that decays as training progresses. Epsilon decay selection establishes the time from which a valid model can be obtained, and how good it will be. Smaller decay values causes fewer training times but worse models, and vice versa.

Finally, the worst limitation found in DRL algorithms is the huge training time for a complete online training like was described with DQN-CNN Agent. Each test involves a long waiting time to get a model, which can or cannot work. DQN-PilotNet Agent was built in order to solve the large DQN-CNN training times by including a pre-trained neural network, as explained above, that decouple the features extraction in two steps. According to the results achieved by this agent, we can claim that including the modified PilotNet is a good practical solution for our application.

Despite the above difficulties, simulated trajectories has been carried out, unlike many of the articles in the state of the art. These results are promising, since even though DQN is a discrete algorithm, the final objective of the work has been achieved, which was that a car could learn by itself how to follow a trajectory on a urban map in simulation, efficiently and in a safety way.

The reported results leaves the door open to the implementation of a continuous Deep Reinforcement Learning algorithm to solve the limitations observed in the DQN, with the idea of reaching better models with shorter training times. Another future work is to export our best simulated models to our real car prototype to really check the potential of these DRL techniques.

ACKNOWLEDGMENT

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.

References

1. Chan, C.: Advancements, prospects, and impacts of automated driving systems. International Journal of Transportation Science and Technology. Vol. 6,3, pp. 208–216, 2017.
2. Urmson, J. Anhalt and Others.: Autonomous driving in urban environments boss and the urban challenge. Journal of field Robotics. Vol. 25, 8, pp. 425–466, 2008
3. Montemerlo, M. Becker, J. and Others.: Junior:The stanford entry in the urban challenges. Journal of field Robotics. Vol. 25, 9, pp. 569–597,2008.
4. Wang, F.: AI and intelligent vehicles future challenge (IVFC) in China: From cognitive intelligence to parallel intelligence. ITU Kaleidoscope: Challenges for a Data-Driven Society (ITU K), pp. 1–2,2017.
5. Zhao, J. et al,.: Tiev: The tongji intelligent electric vehicle. International Conference on Intelligent Transportation Systems(ITSC). Pp. 1303–1309, 2018.
6. Liang, M. Yang, B. Wang, S. Urtasun, R.: Deep continuous fusion for multi-sensor 3d object detection. European Conference on Computer Vision (ECCV), pp. 641–656, 2018
7. R. Barea et al.,: Vehicle Detection and Localization using 3D LIDAR Point Cloud and Image Semantic Segmentation. 21st International Conference on Intelligent Transportation Systems (ITSC), pp. 3481-3486, doi: 10.1109/ITSC.2018.8569962. 2018
8. Silver, D. et al.,: Mastering chess and shogi by selfplay with a general reinforcement learning algorithm. CoRR, vol. abs/1712.01815, 2017.
9. Mnih, V. et al.,: Human-level control through deep reinforcement learning. Nature, vol. 518, no. 7540, p. 529, 2015.
10. Kendall. A et al.: Learning to Drive in a Day. 2019 International Conference on Robotics and Automation (ICRA), Montreal, QC, Canada, 2019, pp. 8248-8254, doi: 10.1109/ICRA.2019.8793742.
11. Lillicrap, T. et al.,: Continuous control with deep reinforcement learning. International Conference on Learning Representations (ICLR), 2016.

12. Dosovitskiy, A. Ros, G. Codevilla, F. Lopez, A. Koltun, V.: Carla: An open urban driving simulator. arXiv preprint arXiv:1711.03938, 2017.
13. Quigley, M. et al,.: ROS: an open-source Robot Operating System. ICRA workshop on open source software. 2009. https://www.ros.org/
14. Montemerlo, M. et al.,: Junior: The stanford entry in the urban challenge. Journal of field Robotics, vol. 25, no. 9, pp. 569–597, 2008.
15. Levinson, J. et al.,: Towards fully autonomous driving: Systems and algorithms. Intelligent Vehicles Symposium (IV), 2011 IEEE. IEEE, 2011, pp. 163–168.
16. Chen, J. Yuan, B. Tomizuka, M.: Deep imitation learning for autonomous driving in generic urban scenarios with enhanced safety. arXiv preprint arXiv:1903.00640, 2019.
17. Codevilla, F. Miiller, M. Lopez, A. Koltun, V. and Dosovitskiy, A.: End-to-end driving via conditional imitation learning. 2018 IEEE International Conference on Robotics and Automation (ICRA), pp. 1–9, 2018.
18. Yurtsever, E. Capito, L. Redmill, K. Ozguner, U.: Integrating Deep Reinforcement Learning with Model-based Path Planners for Automated Driving. ArXiv:2002.00434 (2020)
19. Vitelli, M. Nayebi, A.: CARMA: A deep reinforcement learning approach to autonomous driving. Tech. rep. Stanford University, Tech. Rep., 2016.
20. Ganesh, A. Charalel, J. Sarma, Das, M. Xu, N.: Deep Reinforcement Learning for Simulated Autonomous Driving. In: Stanford University, 2016.
21. Jaritz, M, Charette, R. Toromanoff, M. Perot, E. Nashashibi, F.: End-to-End Race Driving with Deep Reinforcement Learning. 2070-2075. ICRA 2018. 10.1109/ICRA.2018.8460934.
22. Chen, J. Shengbo, E. Tomizuka, M.: Interpretable End-to-end Urban Autonomous Driving with Latent Deep Reinforcement Learning. arXiv preprint arXiv:2001.08726, 2019.
23. . Fan, J. Wang, Z. Xie, Y. Yang Z.: A Theoretical Analysis of Deep Q-Learning. ArXiv:1901.00137
24. CARLA: Open-source simulator for autonomous driving research. https://carla.org
25. Sentdex, "Carla-rl," https://github.com/Sentdex/Carla-RL, 2020.
26. Keras. https://keras.io/
27. Tensorflow. https://www.tensorflow.org
28. Bojarski, M. et al.,: End to end learning for self-driving cars. arXiv:1604.07316. 2016.
29. Gutiérrez, R. et al,.: A Waypoint Tracking Controller for Autonomous 2 Road Vehicles using ROS Framework. Send to Sensors. 2020.