Deep Reinforcement Learning Autonomous Driving Using Lidar in the Physical World

Michael Bosello*

* Università di Bologna – DISI: Department of Computer Science and Engineering, Cesena, Italy

***Abstract*—Autonomous cars have been in the making for over 15 years. Skepticism has taken the place of initial hype and enthusiasm. The dream of fully autonomous cars has been delayed as current self-driving systems, like ones from major players such as Tesla and Waymo, rely on supervised learning which requires countless labeled data and humans in the loop during the training phase. In this paper as well as in our previous work, we consider robot-drivers as teen-drivers eager to learn how to drive but prone to mistakes in the beginning. The question we are trying to investigate is "what if we allow autonomous cars to make mistakes like young human drives do?" The deep reinforcement learning approach gives us newer possibilities to solve complex control tasks. It let the agent learn by interacting with the environment and from its mistakes. Unfortunately, transferring learning from simulations to the real world is a hard problem. In this paper, we explore the use of lidar data as input of a Deep Q-Network on a realistic 1/10 scale car prototype capable of performing training in real-time. The robot-driver learns how to run in a circuit by exploiting the experience gained in the real world through a mechanism of rewards designed to quickly help our robot-teen to learn its driving skills.**

***Index Terms*—Reinforcement Learning, Autonomous Driving, LIDAR, f1tenth, real-time systems.**

I. Introduction

Autonomous driving has been a societal goal since the automotive infancy. RCA engineers envisioned driver-less cars on US highways by 1975 [1]. While the prediction has not been fulfilled in time; the dream is still alive and was re-ignited by the DARPA Grand Challenge [2] for autonomous vehicles in 2004. Driving a vehicle becomes natural and almost semi-automatic for the large majority of humans. The necessary skills are usually acquired by humans in their early 20s and kept for life. In contrast, driving is a fairly complex task for computers as they need to interpret scenes, make predictions, and take actions based on the input of many sensors providing different information for a given scene, and this is repeated hundreds of times every second. Autonomous driving involves three main problems [3]:

- *recognition*: the ability to detect and identify the environment's feature and components e.g., line marks and traffic sign detection, pedestrians and obstacles recognition;
- *prediction*: predicting the future evolution of the surrounding;
- *planning*: taking decisions about which actions to perform.

Interpreting a scene and reacting to it is natural for humans; they learn how to understand and react to stimuli during their infancy and make this experience as part of their natural behaviors; fully replicating this behavior by an artificial agent is one grand challenge of this century.

Autonomous vehicles are classified by the US National Highway Traffic Safety Administration (NHTSA) according to their automation level from Level 0 to Level 5. Figure 1

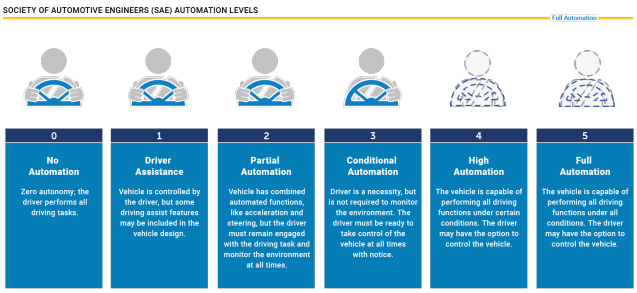

Figure 1. Current Status in Vehicle Automation (credits to SAE & NHTSA)

shows the current progress towards fully autonomous vehicles. In Level 0 the human driver does all the driving; in Level 5 an Automated Driving System (ADS) on the vehicle can do all the driving in all circumstances. The human occupants are just passengers and need never be involved in driving [4]. As the technology approaches level 5 the path to success becomes more difficult: vehicles need to predict *ALL* possible driving scenarios, known, and unknown [4]. Ça va sans dire, the general public expects driverless cars to be extremely safe (i.e. virtually 0 accidents) and no forgiveness is granted to robots[1]. In contrast, human drivers are given a fair amount of forgiveness for failures.

According to CBINSIGHT research group over 40 companies are in the race for L5 autonomous cars including well-established manufacturers such as Audi or relative newcomers like Tesla [5]. A common approach to driver-less cars involves instrumenting the vehicle with a wide range of sensors, constantly updating very high definition multi-layer maps, developing and employing Machine Learning (ML) models to interpret the scene and consequently controlling the vehicle. Car manufacturing costs, as well as operational costs, are driven higher in the process due to the expensive sensors and the constant need for data-updates that lead to a burden on the

[1] A single casualty in a Tesla driven by Autopilot has drowned more attention than all the other road-casualties in the same week.

communication and computational infrastructure. All players optimized their systems for safety through the introduction of very accurate high-definition sensors such as LIDARs and supervised machine learning models trained with humongous amounts of labeled data. Supervised learning, while effective is very expensive and time-consuming as it requires humans in the loop in the training phase, either to label the data or to provide training signals. Covering all possible driving scenarios becomes a very hard challenge and is today the conundrum of autonomous driving. Some researchers thus believe that given the fast-paced changes in driving scenarios L5 autonomous driving is a chimera and propose remotely-assisted alternatives [6].

In our works, we depart from the assumption of infallible self-driving vehicles to accept the idea of a limited amount of time granted to robot-drivers to learn how to drive in certain scenarios. Relaxing the constraint of an immediately perfect driving behavior opens the doors to unsupervised learning techniques. Unsupervised learning relies on trial and error to learn new patterns and behaviors in a fashion that resembles young human driving students. While supervised learning techniques learn input-output associations (i.e. they are not able to learn the dynamics of an agent's environment), Reinforcement Learning (RL) is specifically formulated to handle the agent-environment interaction [7]. It is therefore a natural approach for learning robotics (and autonomous driving) control policies. In the context of robotics applications, RL has taken an increasingly important role as it allows us to design hard-to-engineer behaviors [8] and optimizing problems that have no closed-form solutions.

In our previous work [9], we have explored the use of RL (DQN) directly in the real-world using a small-size robot-car instrumented with a wide-angle camera, and we have shown that DQN can be trained in a real-time system using real-world camera frames. In this work, we have built a very realistic 1/10 scale racing car (namely, an f1tenth [10]), equipped with a LIDAR, and capable of performing DQN training in real-time. In this setting, the driver agent faces all the challenges of a real driving scene including sensors noise and actuators' unpredictability that lead to imprecise vehicle control. We have then trained the agent to drive on a race track using *real* LIDAR data as input, which has been considered an *open problem* [11]. To strengthen our findings, we provide a comparison of the performance of three different Neural Networks (NNs): a one-dimensional convolutional neural network (1D CNN), a two-dimensional convolutional neural network (2D CNN) which uses images – grid maps – generated from LIDAR data, and a fully-connected network.

The remainder of this paper is organized as follows: section II discusses the state of the art and background on machine learning applied to autonomous driving; section III offers a brief primer to reinforcement learning and defines the autonomous driving problem in the context of RL. In section IV, we examine the challenges of using LIDAR data in the context of RL, and the work done on this subject. The experimental setup is described in V while the results are presented and discussed in VI. The final remarks and future works in section VII concludes the paper.

II. ML in Autonomous Driving

Autonomous driving is based on machine learning models that are used to make decisions and predictions on the driving scene and control the vehicle consequently. ML has been of paramount importance in providing scene perception to robot-drivers. In the early days trough object detection algorithms based on hand-crafted feature extraction followed by a classifier i.e. Viola-Jones Classifier [12]. Feature-based classifiers are however inadequate for autonomous driving scenarios as they lack the necessary robustness. Advances in Deep Learning (DL), the availability of large data sets [13], are fueling the dream of self-driving cars. The potential offered by deep learning techniques has led to extensive use of ML in the autonomous driving arena. Specifically, Convolutional Neural Networks [14] fostered a revolution in pattern recognition [15] thus enabling advanced real-time applications. CNN training requires large amounts of data and can take even several weeks; however, the inference model created is relatively small usually in the order of a few hundred Megabytes [15] and can run in near-realtime on relatively small computational units. Two are the main strategies used when applying ML to autonomous driving: Single task handling and end-to-end learning. In the first case, every single task is designed with human aid and optimized per task. In the latter case, the system is modeled from driving scene to the appropriate vehicle control action (i.e. steering angle) [16], and it is designed to achieve the ultimate goal of controlling the vehicle and single tasks have no independent value in the design. Considering the single-task approach, CNN made it possible and resilient several perception tasks such as in-lane and vehicle detection. CNN successfully perform such tasks in realtime at frame rates required for real vehicles [17].

End-to-end autonomous driving agents have been also proposed as an alternative to the approach of decoupling the system into single-tasks. The aim is to improve performances by directly optimizing the final goal of self-driving instead of optimizing human-selected intermediate tasks. This approach has been initially proposed by NVIDIA with the DAVE-2 System [18]. In DAVE-2 a CNN goes beyond pattern recognition; the system takes raw pixel of a front-facing camera as input and returns the steering angle of the wheel directly. DAVE-2 learns internal representations of the necessary processing steps with only the human steering angle as the training signal. The system was trained on a video labeled with the steering angle applied by the human driver through a wide variety of roads.

The majority of the methods currently in use for autonomous driving are based on *supervised learning* techniques. They require *very large labeled data-sets* or significant human effort to provide training signals. A radically different vision advocates for the use of unsupervised learning techniques.

Reinforcement Learning is an *unsupervised learning* technique which *learns by trial-and-error*, and it does not require

explicit supervision from humans. Applications of RL to autonomous driving have been inspired by Deep Mind in (super) human-level control demonstrated on Atari games with the Deep Q-Network (DQN) method[19]. In particular, [3] proposes an end-to-end framework, based on RL techniques, for autonomous driving. There, the input is the environment's state (camera frames) and its aggregations over time, and the output is the driving action. Instead, [20] split the task into two modules: the perception module uses deep learning to extract the track features from images, and the control module uses RL to make decisions based on these features. Both the works use TORCS, an open-source car racing simulator, to train and evaluate the frameworks. In [21] the authors investigate the use of DQN – as is – in a racing game. In contrast with supervised learning, that can be applied to real vehicles, training of Reinforcement Learning based driving agents is based on *simulated environments*. The trial-and-error nature of the paradigm that makes it impossible (or at least very expensive) to perform the training in the real world. Unfortunately, however, an agent trained in a virtual environment *will not perform well in the real setting*, regardless of the sensors used. Both low-dimensional sensors like cameras and high-dimensional sensors like LIDARs are affected by the issue. In the case of images, the visual appearance of a virtual simulation is different from that of a real-world driving scene, especially for textures [22]. The literature offers methods to transfer the learning from a virtual world to the real one. In particular, [22] proposes a framework that converts the images rendered by the simulator to realistic ones, and then the agent is trained on those synthesized scenes, thus adding potential noise to the model.

III. Reinforcement Learning Primer

In this section, we provide a short review of RL taking the second edition of Rich Sutton's textbook[7] as the main reference – For a comprehensive overview, we suggest to consult that book.

RL is a machine learning framework in which an agent learns how to fulfill a task *by interacting* with its environment. At every time step, the environment provides a *state* and a *reward*. The state is the set of information useful to predict the future, and the reward is a value that quantifies the goodness of the agent behavior. The reward signal is thus your way of communicating to the agent what it must achieve. Based on the state, the agent decides which action to take to maximize the cumulative reward, i.e., the sum of *rewards over time*. The agent's actions affect the evolution of the environment, thereby affecting opportunities available to the agent at later times. Thus, correct choices require the balance of immediate and future rewards.

The objective of the agent is to learn the optimal policy. A *policy* defines the agent behavior, and it's based on a map between states and actions. The optimal policy is the one that maximizes the cumulative reward. A way to produce a policy is to *estimate* the *action-value function* which associate state and action pairs to a value called *expected return*. The expected return is the expectation of future rewards that the agent will get starting from a state S and following a policy π. *Q-learning* is an online algorithm that estimates the optimal action-value function and that refine the estimation through iterations on the new experience. Every time the evaluation becomes more precise, the policy gets closer to the optimal one. Given the value function, the agent need only to choose the action with the greatest expected return. A table can represent the Q-function, but, when the number of states increases, it becomes hard or even impossible to use this method. Here, Neural Networks (NN) come to our help. A NN can *approximate* the Q-function and possibly converge to the real function. This approach is known as DQN.

One of the RL issues is the exploration/exploitation dilemma. We want the agent to behave optimally, but the only way to enhance the precision of the value function is to *explore* the environment by choosing different actions – and possibly *making mistakes*. A typical method to balance the exploitation and exploration phases is to use an ϵ-greedy policy with which the agent behave greedily, but there is a (small) ϵ probability to select a random action.

A. Modeling the driving problem

The RL setting is formalized as a Markov Decision Process (MDP); a MDP is defined by a tuple containing the set of states, the set of actions, the set of immediate rewards associated to a state transition $s \rightarrow s'$, and the state transition probability function which describe the probability that an action a in the state s will lead to state s'. An environment displays the Markov property if the probability distribution of $s_t + 1$ depends solely on s_t and a_t.

Properly characterizing the driving task requires close attention as developers have a great deal of freedom in choosing input features (states) and control signals (actions). Moreover, the driving problem is open to subjective definitions as different behaviors in similar situations may be equally acceptable. As a result, designing a suitable reward function could be tricky. The correct design of the reward function is crucial to obtain the desired behavior, especially in real-world systems. A first set-up of driving as a MDP is given in [23], which is compatible with our definition stated below.

Driving can be modeled as a multi-agent interaction problem in which the driver seeks to behave optimally (reaching the target point in a fast and safe way, ensuring passengers comfort, and so on) while taking into account of others. The action set could be either discrete or continuous. In the first case, the actions available may be: go forward, turn right, turn left, speed up, slow down, brake. A wider set of discrete actions could be also considered, for example, it could contain an action for each x degree of steering and an action for y steps in the accelerator pedal. In the second case, the output will be the continuous control signal for the steering wheel, throttle, and brake. The environment state consists of information about car conditions and surroundings, but it is not directly accessible to the agent which needs to reconstruct it by sensor readings. The developer decides

which *observations* will be given; they may include the car's position, orientation, velocity, acceleration as car status, while the surroundings may be represented either by raw input like camera frames, LIDAR maps or by high-level features such as ones derived from object and sign recognition. All the useful information available are put together and presented as a state; the chosen RL process will perform the *sensor fusion* task by weighting each sensor feature according to its relevance. Since the environment evolution depends not only on the driver actions but also on the behavior of the other road users, the Markov property is lost. Furthermore, the environment is only partially observable. One possible solution is to use Partially Observable MDP (POMDP) in which we assume the Markov property, but the agent receives only observations regarding the current state. From observations, the agent seeks to reconstruct states by information integration over time, for example, by keeping a history of states or by using a Recurrent Neural Network (RNN). Finally, the reward signal has to define the goals of the driver by taking into account both correct driving and planning – in order to reach the destination. For instance, We could give a negative reward if a distance sensor indicates an obstacle too close or a positive reward if the car moves towards the target. As stated above, the definition of the reward function is not a simple task and it could be designed in several ways thus the provided examples are not comprehensive.

IV. LIDAR-BASED RL

While the use of camera frames in RL driving tasks has become rather common, little work has been done to use LIDAR data to guide RL agents in driving contexts, even considering researches based solely on simulated environments.

Authors of [11] used RL to train several NNs having LIDAR data as input in the context of NN verification. Even though their main interest was NN verification to ensure safety in safety-critical systems, their findings are useful also for understanding LIDAR data in ML. They took the f1tenth platform as a testbed. The system has been trained and verified in a simulated environment, after that, the simulation to real (sim2real) gap has been analyzed. The environment consisted of a single 90-degree right turn, with the car approaching and passing the turn at constant throttle. The use of a simplified environment was necessary to keep the verification task manageable. The results show that state-of-the-art verification tools can handle at most 40 rays (while 2D LIDARs have typically 1081 rays) and that only 10% of the time is used to verify the NN, while the rest of it is used to elaborate the car's dynamic and observation models (that are needed in the verification process). This informs us about the complexity of the car's dynamics as well as the complexity of the policy that our agent seeks to learn. When they moved to the real environment, they found that LIDAR measurements are greatly affected by reflective surfaces (a problem that we faced as well). When a ray gets reflected, it appears as it is no obstacle in that direction. The agent performed well in the ideal simulated environment. In a real environment specifically designed to reduce reflections (where all the reflective surfaces were covered), LIDAR faults still caused crashes in 10% of the runs (they emphasized that LiDAR faults occurred even in such an environment). In the unmodified real environment, they were not able to train a robust controller using state-of-the-art RL algorithms. They have therefore claimed it has been an open problem. Interestingly, fault patterns that caused unsafe behaviors were identified and reproduced in simulation.

[24] presents one of the first approaches that use LIDAR data in RL contexts. The input of the system is composed of camera frames and grid maps (images produced using LIDAR data that display the perceived borders). Two 2D CNNs process the images and their outcomes are fused together through a dense layer. Although it is one of the first works applying RL to LIDAR measurements, the testing scenario was quite simple: a car on a highway that can only change line or accelerate/brake. A physical car prototype similar to an f1tenth is mentioned, but no tests or results performed on it are shown.

In [25], a set of OpenAI Gym environments for RL-robotic research are proposed, complemented by experiments that used basic RL algorithms. A simulated TurtleBot [26] (a two-wheel robotic-car) learned how to move in a circuit using "highly discretized" LIDAR readings. We should notice the Turtlebot has much simpler dynamics compared to the f1tenth (we will elaborate on that in the next section).

Lastly, we should mention 3D LIDARs. While they are not adopted for end-to-end control, they could be employed to provide high-level features in the context of autonomous driving. In [27], deep RL is used to perform semantic parsing of large-scale 3D point clouds. DQN, with a 3D CNN as NN, decides the size and position of the observation window, then a RNN takes the features and produces the classification result. This approach outperformed other state-of-the-art classification methods.

V. EXPERIMENTAL STUDY SETUP

A. Hardware and track setup

We used the F1/10 (or f1tenth) platform [10] as a testbed for our experiments. F1tenth is an open-source 1/10 scale racecar designed for autonomous systems and cyber-physical systems research. This platform offers high-performance and hardware/software stacks similar to full-scale solutions while limiting costs and dangers typical of real vehicles. With the aim to conduct real-world experiments, the f1tenth features *realistic dynamics* like Ackermann steering and the ability to achieve high speeds. Remarkably, authors of the platform have shown that it is possible to port the stack on a real car. Ackermann steering is the geometric arrangement of linkages used in the steering of a typical car. Compared to differential steering, which allows a vehicle to turn by applying different drive torque to its sides, Ackermann steering has a more complex dynamic model and requires a wider operating space. In our previous experiment [9] we used differential steering which required less engineering efforts to design the learning environment and that resulted in less training time. For example, a vehicle with differential steering can recover

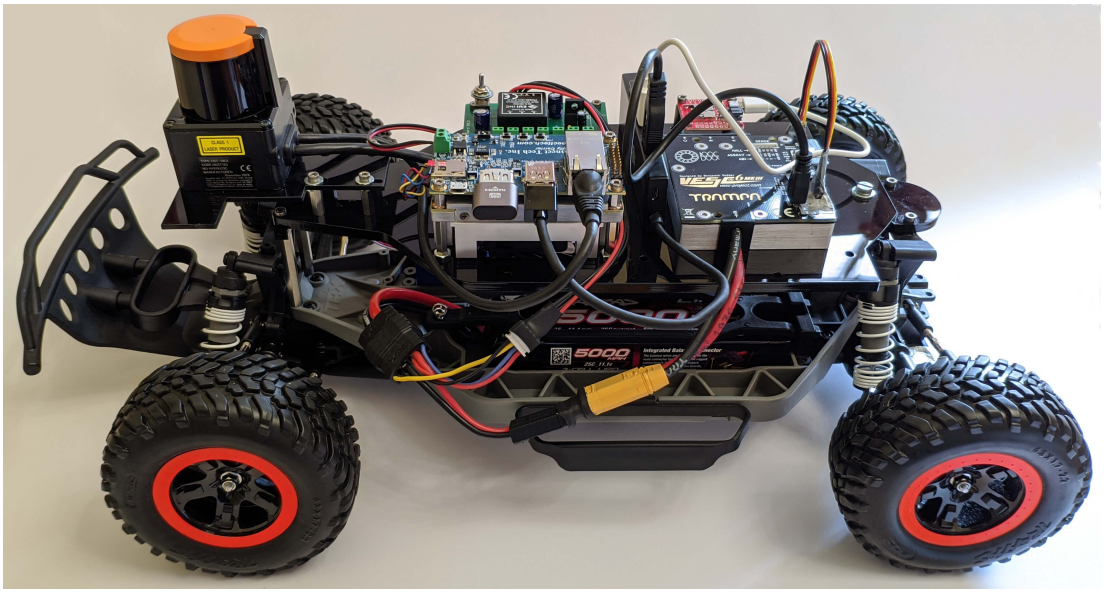

Figure 2. The f1tenth race car used in the experiments.

from bad states near an obstacle by turning on itself, whereas a car with Ackermann steering will eventually reach a fatal state. This kind of dynamics makes the agent learning slower and, if care is not taken, the agent might not learn at all. It goes without saying that the f1tenth platform is much more realistic than commercially available alternatives like TurtleBot [26] which deliver slow-speeds and differential steering.

The bottom chassis of the f1tenth comes from a 1/10 scale race car available from Traxxas. The Traxxas model has very realistic mechanisms, just like a real race car. It has a brushless motor capable of reaching (depending on the model) over 90 km/h, and a servo motor for steering control. The top chassis is a custom laser-cut ABS plate that hosts all the electronic components. The power board is a PCB designed to provide stable voltage to the car and its peripherals, and it is connected to a Lithium Polymer battery. The VESC (Vedder Electronic Speed Controller) controls the speed of the motor and the direction of the servo. The main board that controls the car is a Jetson TX2 [28], a GPU module designed to enable embedded AI. Its GPU and memory capabilities make it possible to train NNs in real-time. The TX2 module is housed on a carrier board [29] characterized by a reduced form factor. Both the hardware and software stacks of the f1tenth are modular. Several sensors are available on the f1tenth, but only the LIDAR is present in our build. Fig. 2 shows our produced f1tenth. The 2D LIDAR used is a Hokuyo UST-10LX [30]. The sensor has a 270° field of view with an angular resolution of 0.25°, for a total of 1081 scan rays. It has a detection range of 10m, an accuracy of ±40mm, and a scan speed of 25ms. According to the datasheet, a specific number is returned if a measurement error occurs like an object does not exist or object has low reflectivity. Fig. 3 displays visual representations of data produced by LIDAR scans, depicting the sensor precision in different conditions. On the software side, sensors, actuators, and controllers are represented as nodes and coordinated by ROS (Robot Operating System) [31], a middleware strongly based on the publish/subscribe paradigm.

A very handful feature of the f1tenth is the possibility to run your code on both the real f1tenth and the provided simulator, almost without changes. The biggest difference is the field of view of the simulated LIDAR which is 360°. We reduced it to 270° like the real Hokuyo so as to conduct experiments with the same conditions. We initially used the simulator to speed up development times and then we moved to the real car. Challenges addressed in the real setting are discussed in subsection V-D. Due to space constraints, the physical track is quite simple therefore the result comparison of different NNs has been done on the simulator with a complex track. The real racetrack is shown at the top left of fig. 3 and the simulated one is shown in fig. 4.

B. RL environment

As mentioned above, the software is coordinated through ROS. The software environment is built as follows. The sensor node subscribes to LIDAR and odometry data, and it stores them. The RL environment uses the latest stored data, that are updated asynchronously at every new measurement. The control node sends the latest driving command chosen by the RL agent to the VESC. The node sends the command at fixed time intervals and continues to send it until the agent asynchronously updates the command. The safety node implements the automatic emergency braking and ensures that the car will not – hardly – crash. Every time it receives a LIDAR scan message, the time-to-collision is calculated on every ray. If the time-to-collision is below a threshold, the car will brake with a higher priority compared to commands send

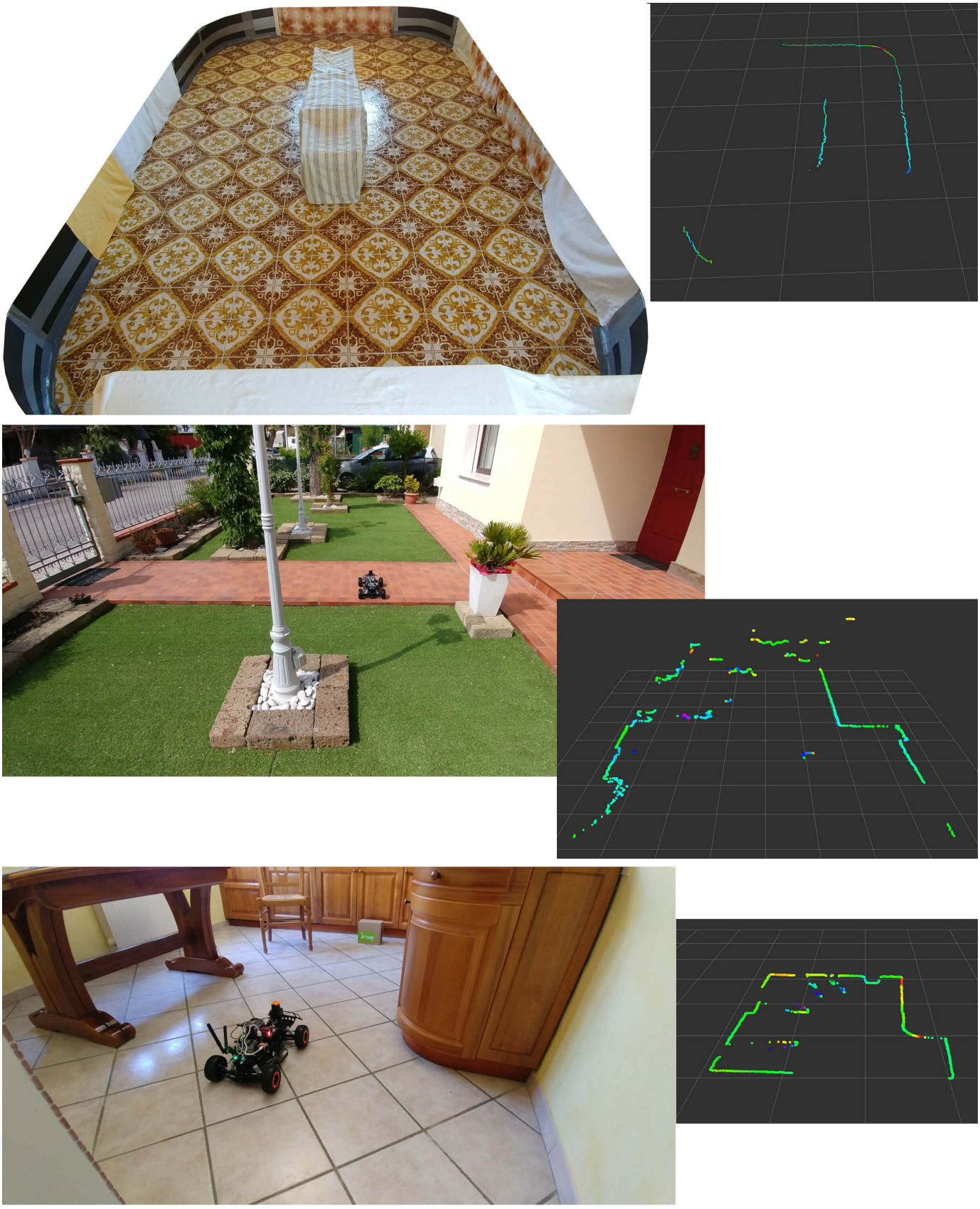

Figure 3. The circuit used in the experiments (top). An example of LIDAR data in an outdoor scenario with far away obstacles (middle). An example of LIDAR data in the presence of thin obstacles like a chair's legs (bottom).

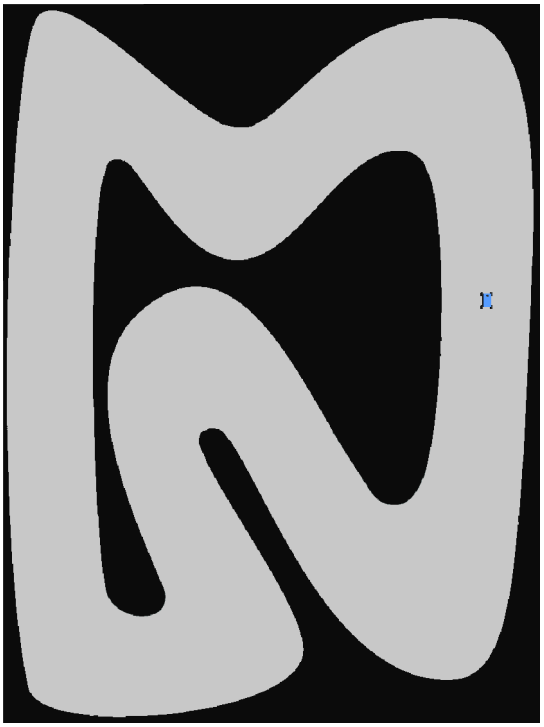

Figure 4. The simulated racetrack used in the comparison experiments.

by the RL agent. The time-to-collision is calculated using the following equation:

$$TTC_i(t) = r_i(t)/[-v_x cos(\theta_i)]_+$$

r_i	distance measured by ray i
v_x	car's linear velocity
θ_i	beam angle

At each cycle, the RL environment returns a state, a reward, and a boolean that indicates if the episode ended. The state is formed by the latest raw LIDAR distances. If the emergency braking was activated, the episode ends and a negative reward (-1) is given. Before starting a new episode and returning the control to the agent, the car goes in reverse to ensure it has enough space to start again. The agent has three actions: forward, right, and left. Having only three actions let the agent learn faster. Considering that the commands update is quite fast, there is no need for more actions as the agent can still achieve appropriate control. The reward function has been designed differently in the simulator and in the real car. Rewards are clipped between [-1, 1] because it could dramatically affect the learning efficiency [19]. In the physical f1tenth, the reward is proportional to the car velocity obtained through odometry, with a maximum of 0.09. This *optimizes* for the fastest route. A small bonus proportional to the distance to the nearest obstacle is added to the previous reward to award safer routes. In the case of the simulator, a fixed reward is assigned to each action: the forward movement gives a reward of 0.2, while turn and left give a 0.05 reward. This differentiation prevents zigzagging behaviors.

C. LIDAR data pre-processing

The Hokuyo LIDAR has 1081 rays with an angular resolution of 0.25°. Due to the curse of dimensionality, each additional value in the input vector makes the state space grow exponentially. To reduce the state size, we have cut the field of view from 270° to 180° as we are not interested in overtaken obstacles. In addition, we have further reduced the state size by grouping values together. In our experiments, the f1tenth's hardware was able to manage a NN that can process at most 40 rays, and the best results were obtained with 20 rays. Moreover, a coarse-grained resolution is sufficient for racetracks. We used the exceeding rays to strengthen the measurements by grouping the rays together. We have tested three grouping method: `averaging`, values are split in 20 groups and averaged; `sampling`, a value each x is taken; `minimum`, the minimum of the group is taken. As the averaging method seems to be the most robust, it is the method used in the experiments. Values are also normalized between 0 and 1, with 1 corresponding approximately to the maximum value that can be sensed in the specific track. In NNs, having input values in the range $[0, 1]$ improves learning.

D. From simulation to reality

Moving to the real car required three adjustments: addressing LIDAR faults, facing continuous time, and specific hyper-parameter tuning.

As explained in section IV, LIDAR faults could be tedious for correct control. Since we already achieved resilience to noise with the grouping method, at this stage we needed only to filter out the bad readings (the ones with the error value specified by the manufacturer) from the groups. This was sufficient to train a robust controller.

Videogames are a popular testbed for RL, but in videogames and in several simulators one has complete control on time. In those setting time is *discrete* as the agent-environment interaction is synchronous. In other words, the agent can not "miss" frames, the observations obtained from the environment are the same regardless of the time used to choose the next action to be performed. This is not true in *continuous time* settings where the clock is ticking independently of the agent behavior. As a result, in a real-time system, the agent cycle should have the right timing or it will not learn a robust policy. If the cycle is too fast, the agent will not be able to see the changes in states and to relate the consequences of its actions. If the cycle is too slow, the agent will not be able to correlate states as changes would be too substantial. This required to review the code to speed up the agent cycle and fine-tuning some parameters.

E. Driver-agent software

The software developed uses DQN [32] along with standard enhancements that make it effective like experience replay, target network, and state history. Significant work was done in the engineering of the RL environment, in the shaping of the reward signal, and in hyper-parameters tuning. The code,

excluding the RL environment already described, has two main parts: the NN and the training cycle.

The training cycle evolves as follows. First, an action to carry out is selected with an ϵ-greedy policy. The action is random with probability ϵ; otherwise, it is the one with the greatest value inferred by the NN. Then, the agent performs that action, and the environment returns a new observation vector, a reward, and a boolean value that indicates if the state is terminal or not. Before becoming a state, the observations are pre-processed as stated above and two subsequent observations are stacked together to form the state. This integration over time allows to detect the movements of surrounding objects but also of the car itself.

There are two options to estimate a Q-function with a NN. Here, the network takes a state as input and gives the expected return for every possible action as output. The NN needs *transition samples* for training. A sample is formed by a state with its reward, an action, the new state reached after the action, and the terminal mark. With that information, one can compute the updated expected return of the state and use gradient descent to update the model. Thus, at every iteration step, a new transition is added to the *replay buffer* which contains all the encountered samples (until the maximum capacity is reached, then the older samples are discarded). After a certain number of observation steps in which the model is not updated, at every iteration, the NN is trained on a *batch* of samples retrieved randomly from the pool. *Experience replay* is essential to break the temporal relation between transitions and hence ensure that samples are i.i.d (identically and independently distributed), a necessary condition for an effective Stochastic Gradient Descent. Actually, the NN are two. One represents the target policy and is updated at every iteration. The other is used to produce the behavior and is updated every few steps by copying the weights of the target network. Having a moving target makes NNs unstable, this is the reason behind the target network technique.

The source code, with the trained models and Tensorboard logging of experiments, is available here[2].

VI. Results and Discussion

Two experiments have been performed. One aims to verify whether DQN could be used to train a robust controller in a real environment with noisy LIDAR data. The other one compares the results of three NNs. As already mentioned, the comparison experiment has been conducted in simulation in order to use a complex track. The NNs tested has been: 1D CNN, a fully connected network, and a 2D CNN.

The 1D CNN has two convolution layers with respectively 16 filters with kernel size 4 and 32 filters with kernel size 2. It follows a flatten layer and a dense layer with 64 neurons. The fully connected network has two dense layers with 128 units each. The 2D CNN operates on black and white images generated from LIDAR data (grid maps) obtained by outlining the measured borders. The coordinates of the white pixel

[2]https://github.com/MichaelBosello/f1tenth-RL

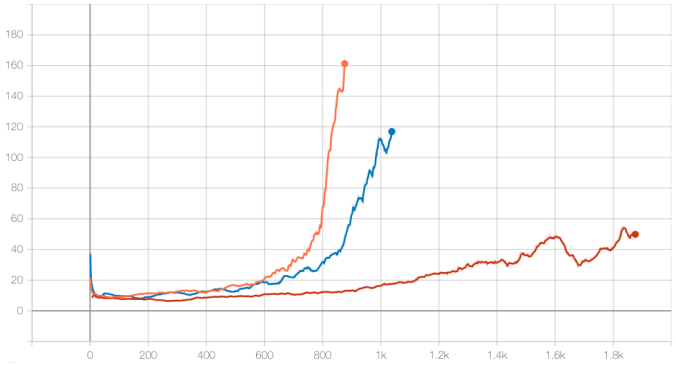

Figure 5. Comparison of average cumulative reward on 100 episodes in the training phase of the simulator experiment. 1D CNN: orange, dense net: blue, 2D CNN: red.

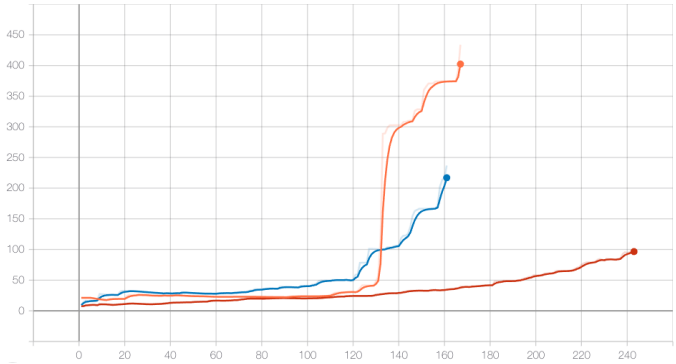

Figure 6. Comparison of average cumulative reward on 100 episodes in the evaluation phase of the simulator experiment. 1D CNN: orange, dense net: blue, 2D CNN: red.

corresponding to one beam are calculated as follows (where the zoom factor helps to separate borders):

$$x = value_i * zoomfactor * cos(angle_i) + offset$$
$$y = value_i * zoomfactor * sin(angle_i) + offset$$

The network is composed by a convolution layer with 16 filters and kernel $(4, 4)$; a max-pooling layer with kernel $(2, 2)$; another convolution layer with 8 filters and kernel $(2, 2)$; another max-pooling layer with kernel $(2, 2)$; a flatten layer; a dense layer with 64 units.

The learning rate is 0.00042; gamma is 0.98; epsilon goes from 1 to 0.1; the other hyperparameters can be consulted in the repository. It should be mentioned that using Huber loss had a greatly positive effect on learning.

CNNs perform better on structured and spatially related data and they are typically used on inputs with local information (i.e. data structured as an array where position matters) such as images and audio. We have shown that 1D CNNs are very effective in processing LIDAR data as they have been the best performer in our tests. The agent equipped with a 1D CNN also showed a behavior oriented to cut curves so as to minimize turning actions and maximize forward actions.

The plot in fig. 5 shows the average cumulative reward on 100 episodes of the three NNs during training stages. Fig. 6 shows the cumulative reward during the evaluation phases, that

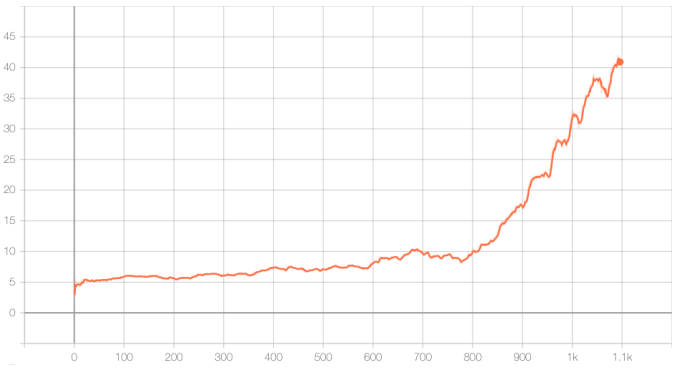

Figure 7. Average cumulative reward on 100 episodes in the training phase of the real car experiment.

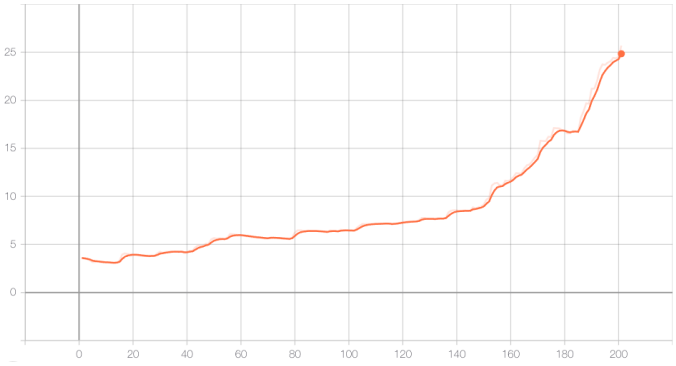

Figure 8. Average cumulative reward on 100 episodes in the evaluation phase of the real car experiment.

alternated train stages. The 1D CNN learned a robust policy in almost 870 episodes with a total training time of three hours, and it reached a maximum of 430 average cumulative rewards in the evaluation stage. The dense network performed well with 1030 training episodes in three hours and a maximum cumulative reward of 230. The 2D CNN learned a decent control policy after 1870 episodes with a total training time of four hours, and a maximum cumulative reward of 98. We have not been able to make the 2D CNN learn a robust policy. Moreover, It should be noted that the 2D CNN is much more hardware demanding, and it could be hardly suitable in embedded systems like the f1tenth.

In the real car, we used the 1D CNN as it is the best option. The plots of the average cumulative reward for training and evaluation are shown in fig. 7 and fig. 8. The physical f1tenth successfully learned a robust control policy to drive in a simple track in three hours. A demonstrative video about the evolution of the car's behavior is accessible in the project's repository. The results are evident in the plot of rewards (it should be noted that the rewards obtained in the simulator and in the real environment have different magnitudes as the cycle time is different. The average cumulative reward of 40, obtained in the real car, is quite good compared to the one of the random policy which is 5). The training has been done at 1/8 of the maximum car speed. We have verified that the policy can scale to higher velocities without re-training. In our space, the higher velocity reached has been 1/4 of the maximum car speed.

VII. Final Remarks and Future works

In this work, we have shown that it is possible to train a robust controller using DQN directly in the real world in a very realistic testbed and in the presence of LIDAR faults. A problem previously considered not solved. Pre-processing steps that led to a robust controller are described, and some alternatives are compared. 1D CNNs have been applied for the first time in the context of LIDAR data, and comparison experiments have shown that 1D CNNs are particularly effective in LIDAR data processing.

We aim at continuing our investigation in several directions that can help to understand how close to real urban scenarios it is possible to push this approach without sacrificing safety. Specifically, in the near future we plan to act on several directions:

- Instrumenting the f1tenth with a camera to compare the performance of camera and LIDAR. In our previous experiment, we used camera frame as input for the agent, but the car used in that experiment and the f1tenth are so different that a comparison would not be meaningful.
- Implementing a deep RL technique for continuous action space like Asynchronous Advantage Actor-Critic (A3C) [33].
- Experimenting with meta-learning [34] approaches and verify if they could be useful in the context of self-driving.
- Move to an urban scenario with e.g. intersections and make the agent follows high-level directions to reach a target.
- Exploring *safe Reinforcement Learning* models [35] and *Reinforcement Learning with bounded risk* [36]. The first tries to ensure functional safety through hard constrains. In the later, the Reinforcement Learning framework acquires the notion of *risk*, which is defined as the probability of entering a fatal state. The aim is to find the optimal policy with bounded risk, i.e., the best policy where the risk is under a certain threshold.
- Exploring the possibility of *cooperative learning* through the exchange of experience between cars with efficient communication[37].
- Attempt to transfer the experience acquired by small-scale vehicles to actual vehicles in a controlled environment.

References

[1] E. Ackerman, "Self-driving cars were just around the corner—in 1960," *IEEE Spectrum*, 2016.

[2] (2019, Sep) The grand challenge for autonomous vehicles. [Online; accessed 20. Sep. 2019]. [Online]. Available: https://www.darpa.mil/about-us/timeline/-grand-challenge-for-autonomous-vehicles

[3] A. E. Sallab, M. Abdou, E. Perot, and S. Yogamani, "Deep reinforcement learning framework for autonomous driving," *Electronic Imaging*, vol. 2017, no. 19, pp. 70–76, 2017.

[4] NHTSA. Automated vehicles for safety -. [Online]. Available: https://www.nhtsa.gov/technology-innovation/automated-vehicles-safety

[5] CBINSIGHT. 40+ corporations working on autonomous vehicles. [Online]. Available: https://www.cbinsights.com/research/

[6] L. Kang, W. Zhao, B. Qi, and S. Banerjee, "Augmenting self-driving with remote control: Challenges and directions," in *Proceedings of the 19th International Workshop on Mobile Computing Systems & Applications*. ACM, 2018, pp. 19–24.

[7] R. S. Sutton and A. G. Barto, *Reinforcement learning : an introduction*. The MIT Press, 2018.

[8] J. Kober, J. Bagnell, and J. Peters, "Reinforcement learning in robotics: A survey," *The International Journal of Robotics Research*, vol. 32, pp. 1238–1274, 09 2013.

[9] G. Pau, M. Bosello, and R. Tse, "Robot drivers: Learning to drive by trial error," in *2019 15th International Conference on Mobile Ad-Hoc and Sensor Networks (MSN)*, 2019, pp. 284–290.

[10] M. O'Kelly, V. Sukhil, H. Abbas, J. Harkins, C. Kao, Y. V. Pant, R. Mangharam, D. Agarwal, M. Behl, P. Burgio, and M. Bertogna, "F1/10: An open-source autonomous cyber-physical platform," 2019.

[11] R. Ivanov, T. J. Carpenter, J. Weimer, R. Alur, G. J. Pappas, and I. Lee, "Case study: Verifying the safety of an autonomous racing car with a neural network controller," in *Proceedings of the 23rd International Conference on Hybrid Systems: Computation and Control*, ser. HSCC '20. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3365365.3382216

[12] P. Viola, M. Jones *et al.*, "Rapid object detection using a boosted cascade of simple features," *CVPR (1)*, vol. 1, no. 511-518, p. 3, 2001.

[13] V. Talpaert, I. Sobh, B. R. Kiran, P. Mannion, S. Yogamani, A. El-Sallab, and P. Perez, "Exploring applications of deep reinforcement learning for real-world autonomous driving systems," 2019.

[14] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel, "Backpropagation applied to handwritten zip code recognition," *Neural Comput.*, vol. 1, no. 4, pp. 541–551, Dec. 1989. [Online]. Available: http://dx.doi.org/10.1162/neco.1989.1.4.541

[15] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2012, pp. 1097–1105. [Online]. Available: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf

[16] F. Leon and M. Gavrilescu, "A review of tracking, prediction and decision making methods for autonomous driving," 2019.

[17] B. Huval, T. Wang, S. Tandon, J. Kiske, W. Song, J. Pazhayampallil, M. Andriluka, P. Rajpurkar, T. Migimatsu, R. Cheng-Yue, F. Mujica, A. Coates, and A. Y. Ng, "An empirical evaluation of deep learning on highway driving," 2015.

[18] M. Bojarski, D. D. 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," *CoRR*, vol. abs/1604.07316, 2016. [Online]. Available: http://arxiv.org/abs/1604.07316

[19] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," 2013.

[20] D. Li, D. Zhao, Q. Zhang, and Y. Chen, "Reinforcement learning and deep learning based lateral control for autonomous driving [application notes]," *IEEE Computational Intelligence Magazine*, vol. 14, no. 2, pp. 83–98, 2019.

[21] A. Yu, R. Palefsky-Smith, and R. Bedi, "Deep reinforcement learning for simulated autonomous vehicle control," Stanford University, StuDocu, 2016.

[22] Y. You, X. Pan, Z. Wang, and C. Lu, "Virtual to real reinforcement learning for autonomous driving," *CoRR*, vol. abs/1704.03952, 2017. [Online]. Available: http://arxiv.org/abs/1704.03952

[23] A. Kendall, J. Hawke, D. Janz, P. Mazur, D. Reda, J.-M. Allen, V.-D. Lam, A. Bewley, and A. Shah, "Learning to drive in a day," 2018.

[24] A. R. Fayjie, S. Hossain, D. Oualid, and D. Lee, "Driverless car: Autonomous driving using deep reinforcement learning in urban environment," in *2018 15th International Conference on Ubiquitous Robots (UR)*, 2018, pp. 896–901.

[25] I. Zamora, N. G. Lopez, V. M. Vilches, and A. H. Cordero, "Extending the openai gym for robotics: a toolkit for reinforcement learning using ros and gazebo," 2016.

[26] "TurtleBot.org," [Online; accessed 31. Aug. 2020]. [Online]. Available: https://www.turtlebot.com

[27] F. Liu, S. Li, L. Zhang, C. Zhou, R. Ye, Y. Wang, and J. Lu, "3dcnn-dqn-rnn: A deep reinforcement learning framework for semantic parsing of large-scale 3d point clouds," in *2017 IEEE International Conference on Computer Vision (ICCV)*, 2017, pp. 5679–5688.

[28] "Jetson TX2 Module," Aug 2019, [Online; accessed 31. Aug. 2020]. [Online]. Available: https://developer.nvidia.com/embedded/jetson-tx2

[29] "Orbitty carrier for nvidia," [Online; accessed 31. Aug. 2020]. [Online]. Available: https://connecttech.com/product/orbitty-carrier-for-nvidia-jetson-tx2-tx1

[30] H. A. Corporation, "Scanning laser range finder smart-urg mini ust-10lx specification," 2016, [Online; accessed 31. Aug. 2020]. [Online]. Available: https://hokuyo-usa.com/application/files/3515/8947/8336/UST-10LX_Specifications.pdf

[31] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Ng, "Ros: an open-source robot operating system," vol. 3, 01 2009.

[32] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, pp. 529 EP –, 02 2015. [Online]. Available: https://doi.org/10.1038/nature14236

[33] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," ser. Proceedings of Machine Learning Research, M. F. Balcan and K. Q. Weinberger, Eds., vol. 48. New York, New York, USA: PMLR, 20–22 Jun 2016, pp. 1928–1937. [Online]. Available: http://proceedings.mlr.press/v48/mniha16.html

[34] C. Finn, P. Abbeel, and S. Levine, "Model-agnostic meta-learning for fast adaptation of deep networks," 2017.

[35] S. Shalev-Shwartz, S. Shammah, and A. Shashua, "Safe, multi-agent, reinforcement learning for autonomous driving," 2016.

[36] P. Geibel, "Reinforcement learning with bounded risk," in *ICML*, 2001, pp. 162–169.

[37] M. Kamp, L. Adilova, J. Sicking, F. Hüger, P. Schlicht, T. Wirtz, and S. Wrobel, "Efficient decentralized deep learning by dynamic model averaging," 2018.