Deep Reinforcement Learning for Autonomous Driving: A Survey

B Ravi Kiran[1], Ibrahim Sobh[2], Victor Talpaert[3], Patrick Mannion[4], Ahmad A. Al Sallab[2], Senthil Yogamani[5], Patrick Pérez[6]

***Abstract*—With the development of deep representation learning, the domain of reinforcement learning (RL) has become a powerful learning framework now capable of learning complex policies in high dimensional environments. This review summarises deep reinforcement learning (DRL) algorithms and provides a taxonomy of automated driving tasks where (D)RL methods have been employed, while addressing key computational challenges in real world deployment of autonomous driving agents. It also delineates adjacent domains such as behavior cloning, imitation learning, inverse reinforcement learning that are related but are not classical RL algorithms. The role of simulators in training agents, methods to validate, test and robustify existing solutions in RL are discussed.**

***Index Terms*—Deep reinforcement learning, Autonomous driving, Imitation learning, Inverse reinforcement learning, Controller learning, Trajectory optimisation, Motion planning, Safe reinforcement learning.**

I. Introduction

Autonomous driving (AD)[1] systems constitute of multiple perception level tasks that have now achieved high precision on account of deep learning architectures. Besides the perception, autonomous driving systems constitute of multiple tasks where classical supervised learning methods are no more applicable. First, when the prediction of the agent's action changes future sensor observations received from the environment under which the autonomous driving agent operates, for example the task of optimal driving speed in an urban area. Second, supervisory signals such as time to collision (TTC), lateral error w.r.t to optimal trajectory of the agent, represent the dynamics of the agent, as well uncertainty in the environment. Such problems would require defining the stochastic cost function to be maximized. Third, the agent is required to learn new configurations of the environment, as well as to predict an optimal decision at each instant while driving in its environment. This represents a high dimensional space given the number of unique configurations under which the agent & environment are observed, this is combinatorially large. In all such scenarios we are aiming to solve a sequential decision process, which is formalized under the classical settings of Reinforcement Learning (RL), where the agent is required to learn and represent its environment as well as act optimally given at each instant [1]. The optimal action is referred to as the policy.

In this review we cover the notions of reinforcement learning, the taxonomy of tasks where RL is a promising solution especially in the domains of driving policy, predictive perception, path and motion planning, and low level controller design. We also focus our review on the different real world deployments of RL in the domain of autonomous driving expanding our conference paper [2] since their deployment has not been reviewed in an academic setting. Finally, we motivate users by demonstrating the key computational challenges and risks when applying current day RL algorithms such imitation learning, deep Q learning, among others. We also note from the trends of publications in figure 2 that the use of RL or Deep RL applied to autonomous driving or the self driving domain is an emergent field. This is due to the recent usage of RL/DRL algorithms domain, leaving open multiple real world challenges in implementation and deployment. We address the open problems in VI.

The main contributions of this work can be summarized as follows:

- Self-contained overview of RL background for the automotive community as it is not well known.
- Detailed literature review of using RL for different autonomous driving tasks.
- Discussion of the key challenges and opportunities for RL applied to real world autonomous driving.

The rest of the paper is organized as follows. Section II provides an overview of components of a typical autonomous driving system. Section III provides an introduction to reinforcement learning and briefly discusses key concepts. Section IV discusses more sophisticated extensions on top of the basic RL framework. Section V provides an overview of RL applications for autonomous driving problems. Section VI discusses challenges in deploying RL for real-world autonomous driving systems. Section VII concludes this paper with some final remarks.

II. Components of AD System

Figure 1 comprises of the standard blocks of an AD system demonstrating the pipeline from sensor stream to control actuation. The sensor architecture in a modern autonomous driving system notably includes multiple sets of cameras,

[1]Navya, Paris. ✉ ravi.kiran@navya.tech
[2]Valeo Cairo AI team, Egypt. ✉ ibrahim.sobh, ahmad.elsallab@{valeo.com}
[3]U2IS, ENSTA Paris, Institut Polytechnique de Paris & AKKA Technologies, France. ✉ victor.talpaert@ensta.fr
[4]School of Computer Science, National University of Ireland, Galway. ✉ patrick.mannion@nuigalway.ie
[5]Valeo Vision Systems. ✉ senthil.yogamani@valeo.com
[6]Valeo.ai. ✉ patrick.perez@valeo.com

[1]For easy reference, the main acronyms used in this article are listed in Appendix (Table IV).

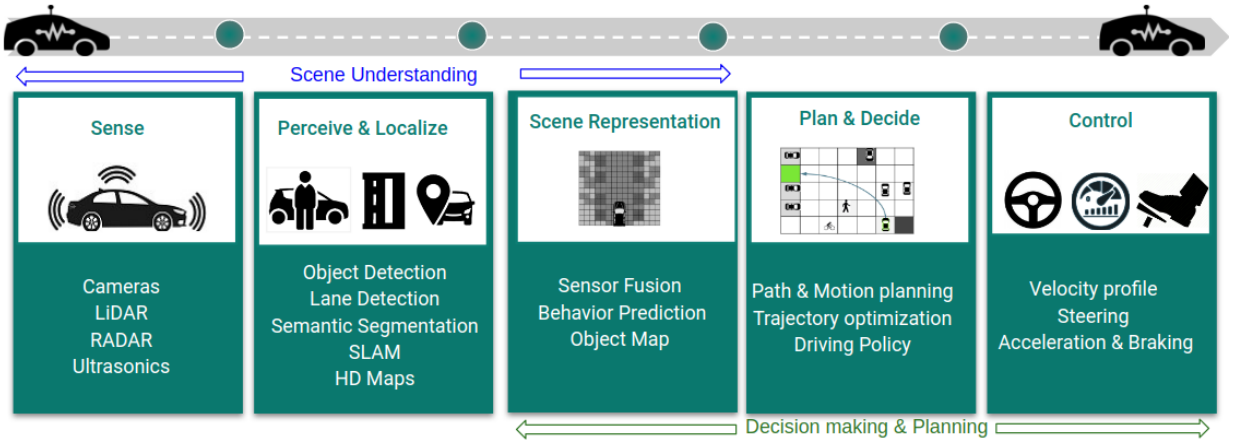

Fig. 1. Standard components in a modern autonomous driving systems pipeline listing the various tasks. The key problems addressed by these modules are Scene Understanding, Decision and Planning.

radars and LIDARs as well as a GPS-GNSS system for absolute localisation and inertial measurement Units (IMUs) that provide 3D pose of the vehicle in space.

The goal of the perception module is the creation of an intermediate level representation of the environment state (for example bird-eye view map of all obstacles and agents) that is to be later utilised by a decision making system that ultimately produces the driving policy. This state would include lane position, drivable zone, location of agents such as cars & pedestrians, state of traffic lights and others. Uncertainties in the perception propagate to the rest of the information chain. Robust sensing is critical for safety thus using redundant sources increases confidence in detection. This is achieved by a combination of several perception tasks like semantic segmentation [3], [4], motion estimation [5], depth estimation [6], soiling detection [7], etc which can be efficiently unified into a multi-task model [8], [9].

A. Scene Understanding

This key module maps the abstract mid-level representation of the perception state obtained from the perception module to the high level action or decision making module. Conceptually, three tasks are grouped by this module: Scene understanding, Decision and Planning as seen in figure 1 module aims to provide a higher level understanding of the scene, it is built on top of the algorithmic tasks of detection or localisation. By fusing heterogeneous sensor sources, it aims to robustly generalise to situations as the content becomes more abstract. This information fusion provides a general and simplified context for the Decision making components.

Fusion provides a sensor agnostic representation of the environment and models the sensor noise and detection uncertainties across multiple modalities such as LIDAR, camera, radar, ultra-sound. This basically requires weighting the predictions in a principled way.

B. Localization and Mapping

Mapping is one of the key pillars of automated driving [10]. Once an area is mapped, current position of the vehicle can be localized within the map. The first reliable demonstrations of automated driving by Google were primarily reliant on localisation to pre-mapped areas. Because of the scale of the problem, traditional mapping techniques are augmented by semantic object detection for reliable disambiguation. In addition, localised high definition maps (HD maps) can be used as a prior for object detection.

C. Planning and Driving policy

Trajectory planning is a crucial module in the autonomous driving pipeline. Given a route-level plan from HD maps or GPS based maps, this module is required to generate motion-level commands that steer the agent.

Classical motion planning ignores dynamics and differential constraints while using translations and rotations required to move an agent from source to destination poses [11]. A robotic agent capable of controlling 6-degrees of freedom (DOF) is said to be holonomic, while an agent with fewer controllable DOFs than its total DOF is said to be non-holonomic. Classical algorithms such as A^* algorithm based on Djisktra's algorithm do not work in the non-holonomic case for autonomous driving. Rapidly-exploring random trees (RRT) [12] are non-holonomic algorithms that explore the configuration space by random sampling and obstacle free path generation. There are various versions of RRT currently used in for motion planning in autonomous driving pipelines.

D. Control

A controller defines the speed, steering angle and braking actions necessary over every point in the path obtained from a pre-determined map such as Google maps, or expert driving recording of the same values at every waypoint. Trajectory tracking in contrast involves a temporal model of the dynamics of the vehicle viewing the waypoints sequentially over time.

Current vehicle control methods are founded in classical optimal control theory which can be stated as a minimisation of a cost function $\dot{x} = f(x(t), u(t))$ defined over a set of states $x(t)$ and control actions $u(t)$. The control input is usually defined

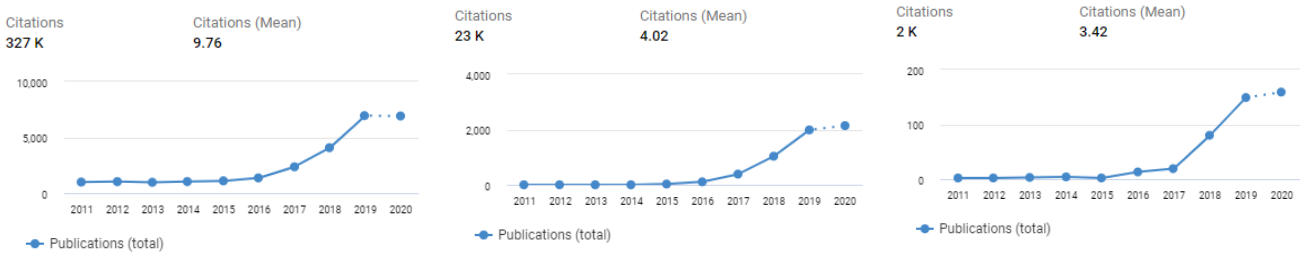

Fig. 2. Trend of publications for keywords 1. "reinforcement learning", 2."deep reinforcement", and 3."reinforcement learning" AND ("autonomous cars" OR "autonomous vehicles" OR "self driving") for academic publication trends from this [13].

over a finite time horizon and restricted on a feasible state space $x \in X_{\text{free}}$ [14]. The velocity control are based on classical methods of closed loop control such as PID (proportional-integral-derivative) controllers, MPC (Model predictive control). PIDs aim to minimise a cost function constituting of three terms current error with proportional term, effect of past errors with integral term, and effect of future errors with the derivative term. While the family of MPC methods aim to stabilize the behavior of the vehicle while tracking the specified path [15]. A review on controllers, motion planning and learning based approaches for the same are provided in this review [16] for interested readers. Optimal control and reinforcement learning are intimately related, where optimal control can be viewed as a model based reinforcement learning problem where the dynamics of the vehicle/environment are modeled by well defined differential equations. Reinforcement learning methods were developed to handle stochastic control problems as well ill-posed problems with unknown rewards and state transition probabilities. Autonomous vehicle stochastic control is large domain, and we advise readers to read the survey on this subject by authors in [17].

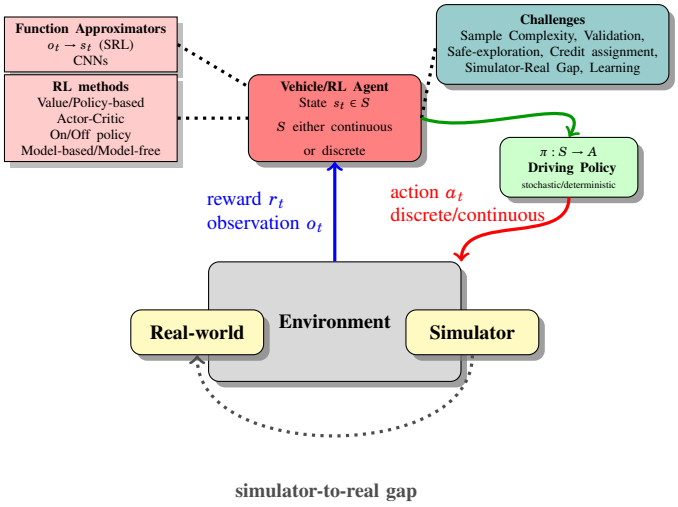

Fig. 3. A graphical decomposition of the different components of an RL algorithm. It also demonstrates the different challenges encountered while training a D(RL) algorithm.

III. Reinforcement Learning

Machine learning (ML) is a process whereby a computer program learns from experience to improve its performance at a specified task [18]. ML algorithms are often classified under one of three broad categories: supervised learning, unsupervised learning and reinforcement learning (RL). Supervised learning algorithms are based on inductive inference where the model is typically trained using labelled data to perform classification or regression, whereas unsupervised learning encompasses techniques such as density estimation or clustering applied to unlabelled data. By contrast, in the RL paradigm an autonomous agent learns to improve its performance at an assigned task by interacting with its environment. Russel and Norvig define an agent as "anything that can be viewed as perceiving its environment through sensors and acting upon that environment through actuators" [19]. RL agents are not told explicitly how to act by an expert; rather an agent's performance is evaluated by a reward function R. For each state experienced, the agent chooses an action and receives an occasional reward from its environment based on the usefulness of its decision. The goal for the agent is to maximize the cumulative rewards received over its lifetime. Gradually, the agent can increase its long-term reward by exploiting knowledge learned about the expected utility (i.e. discounted sum of expected future rewards) of different state-action pairs. One of the main challenges in reinforcement learning is managing the trade-off between exploration and exploitation. To maximize the rewards it receives, an agent must exploit its knowledge by selecting actions which are known to result in high rewards. On the other hand, to discover such beneficial actions, it has to take the risk of trying new actions which may lead to higher rewards than the current best-valued actions for each system state. In other words, the learning agent has to exploit what it already knows in order to obtain rewards, but it also has to explore the unknown in order to make better action selections in the future. Examples of strategies which have been proposed to manage this trade-off include ϵ-greedy and softmax. When adopting the ubiquitous ϵ-greedy strategy, an agent either selects an action at random with probability $0 < \epsilon < 1$, or greedily selects the highest valued action for the current state with the remaining probability $1-\epsilon$. Intuitively, the agent should explore more at the beginning of the training process when little is known about the problem environment. As training progresses, the agent may gradually conduct more exploitation than exploration. The design of exploration strategies for RL agents is an area of active research (see *e.g.* [20]).

Markov decision processes (MDPs) are considered the de facto standard when formalising sequential decision making problems involving a single RL agent [21]. An MDP consists of a set S of states, a set A of actions, a transition function T and a reward function R [22], i.e. a tuple $< S, A, T, R >$. When in any state $s \in S$, selecting an action $a \in A$ will result in the environment entering a new state $s' \in S$ with a transition probability $T(s,a,s') \in (0,1)$, and give a reward $R(s,a)$. This process is illustrated in Fig. 3. The stochastic policy $\pi : S \rightarrow \mathscr{D}$ is a mapping from the state space to a probability over the set of actions, and $\pi(a|s)$ represents the probability of choosing action a at state s. The goal is to find the optimal policy π^*, which results in the highest expected sum of discounted rewards [21]:

$$\pi^* = \underset{\pi}{\operatorname{argmax}} \underbrace{\mathbb{E}_\pi \left\{ \sum_{k=0}^{H-1} \gamma^k r_{k+1} \mid s_0 = s \right\}}_{:= V_\pi(s)}, \tag{1}$$

for all states $s \in S$, where $r_k = R(s_k, a_k)$ is the reward at time k and $V_\pi(s)$, the 'value function' at state s following a policy π, is the expected 'return' (or 'utility') when starting at s and following the policy π thereafter [1]. An important, related concept is the action-value function, a.k.a.'Q-function' defined as:

$$Q_\pi(s,a) = \mathbb{E}_\pi \left\{ \sum_{k=0}^{H-1} \gamma^k r_{k+1} \mid s_0 = s, a_0 = a \right\}. \tag{2}$$

The discount factor $\gamma \in [0,1]$ controls how an agent regards future rewards. Low values of γ encourage myopic behaviour where an agent will aim to maximise short term rewards, whereas high values of γ cause agents to be more forward-looking and to maximise rewards over a longer time frame. The horizon H refers to the number of time steps in the MDP. In infinite-horizon problems $H = \infty$, whereas in episodic domains H has a finite value. Episodic domains may terminate after a fixed number of time steps, or when an agent reaches a specified goal state. The last state reached in an episodic domain is referred to as the terminal state. In finite-horizon or goal-oriented domains discount factors of (close to) 1 may be used to encourage agents to focus on achieving the goal, whereas in infinite-horizon domains lower discount factors may be used to strike a balance between short- and long-term rewards. If the optimal policy for a MDP is known, then V_{π^*} may be used to determine the maximum expected discounted sum of rewards available from any arbitrary initial state. A rollout is a trajectory produced in the state space by sequentially applying a policy to an initial state. A MDP satisfies the Markov property, i.e. system state transitions are dependent only on the most recent state and action, not on the full history of states and actions in the decision process. Moreover, in many real-world application domains, it is not possible for an agent to observe all features of the environment state; in such cases the decision-making problem is formulated as a partially-observable Markov decision process (POMDP). Solving a reinforcement learning task means finding a policy π that maximises the expected discounted sum of rewards over trajectories in the state space. RL agents may learn value function estimates, policies and/or environment models directly. Dynamic programming (DP) refers to a collection of algorithms that can be used to compute optimal policies given a perfect model of the environment in terms of reward and transition functions. Unlike DP, in Monte Carlo methods there is no assumption of complete environment knowledge. Monte Carlo methods are incremental in an episode-by-episode sense. Upon the completion of an episode, the value estimates and policies are updated. Temporal Difference (TD) methods, on the other hand, are incremental in a step-by-step sense, making them applicable to non-episodic scenarios. Like Monte Carlo methods, TD methods can learn directly from raw experience without a model of the environment's dynamics. Like DP, TD methods learn their estimates based on other estimates.

A. Value-based methods

Q-learning is one of the most commonly used RL algorithms. It is a model-free TD algorithm that learns estimates of the utility of individual state-action pairs (Q-functions defined in Eqn. 2). Q-learning has been shown to converge to the optimum state-action values for a MDP with probability 1, so long as all actions in all states are sampled infinitely often and the state-action values are represented discretely [23]. In practice, Q-learning will learn (near) optimal state-action values provided a sufficient number of samples are obtained for each state-action pair. If a Q-learning agent has converged to the optimal Q values for a MDP and selects actions greedily thereafter, it will receive the same expected sum of discounted rewards as calculated by the value function with π^* (assuming that the same arbitrary initial starting state is used for both). Agents implementing Q-learning update their Q values according to the following update rule:

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma \max_{a' \in A} Q(s',a') - Q(s,a)], \tag{3}$$

where $Q(s,a)$ is an estimate of the utility of selecting action a in state s, $\alpha \in [0,1]$ is the learning rate which controls the degree to which Q values are updated at each time step, and $\gamma \in [0,1]$ is the same discount factor used in Eqn. 1. The theoretical guarantees of Q-learning hold with any arbitrary initial Q values [23]; therefore the optimal Q values for a MDP can be learned by starting with any initial action value function estimate. The initialisation can be optimistic (each $Q(s,a)$ returns the maximum possible reward), pessimistic (minimum) or even using knowledge of the problem to ensure faster convergence. Deep Q-Networks (DQN) [24] incorporates a variant of the Q-learning algorithm [25], by using deep neural networks (DNNs) as a non-linear Q function approximator over high-dimensional state spaces (e.g. the pixels in a frame of an Atari game). Practically, the neural network predicts the value of all actions without the use of any explicit domain-specific information or hand-designed features. DQN applies experience replay technique to break the correlation between successive experience samples and also for better sample efficiency. For increased stability, two networks are used where the parameters of the target network for DQN are fixed for a number of iterations while updating the parameters of the online network. Readers are directed to sub-section III-E for a more detailed introduction to the use of DNNs in Deep RL.

B. Policy-based methods

The difference between value-based and policy-based methods is essentially a matter of where the burden of optimality resides. Both method types must propose actions and evaluate the resulting behaviour, but while value-based methods focus on evaluating the optimal cumulative reward and have a policy follows the recommendations, policy-based methods aim to estimate the optimal policy directly, and the value is a secondary if calculated at all. Typically, a policy is parameterised as a neural network π_θ. Policy gradient methods use gradient descent to estimate the parameters of the policy that maximise the expected reward. The result can be a stochastic policy where actions are selected by sampling, or a deterministic policy. Many real-world applications have continuous action spaces. Deterministic policy gradient (DPG) algorithms [26] [1] allow reinforcement learning in domains with continuous actions. Silver *et al.* [26] proved that a deterministic policy gradient exists for MDPs satisfying certain conditions, and that deterministic policy gradients have a simple model-free form that follows the gradient of the action-value function. As a result, instead of integrating over both state and action spaces in stochastic policy gradients, DPG integrates over the state space only leading to fewer samples in problems with large action spaces. To ensure sufficient exploration, actions are chosen using a stochastic policy, while learning a deterministic target policy. The REINFORCE [27] algorithm is a straight forward policy-based method. The discounted cumulative reward $g_t = \sum_{k=0}^{H-1} \gamma^k r_{k+t+1}$ at one time step is calculated by playing the entire episode, so no estimator is required for policy evaluation. The parameters are updated into the direction of the performance gradient:

$$\theta \leftarrow \theta + \alpha \gamma^t g \nabla \log \pi_\theta(a|s), \tag{4}$$

where α is the learning rate for a stable incremental update. Intuitively, we want to encourage state-action pairs that result in the best possible returns. Trust Region Policy Optimization (TRPO) [28], works by preventing the updated policies from deviating too much from previous policies, thus reducing the chance of a bad update. TRPO optimises a surrogate objective function where the basic idea is to limit each policy gradient update as measured by the Kullback-Leibler (KL) divergence between the current and the new proposed policy. This method results in monotonic improvements in policy performance. While Proximal Policy Optimization (PPO) [29] proposed a clipped surrogate objective function by adding a penalty for having a too large policy change. Accordingly, PPO policy optimisation is simpler to implement, and has better sample complexity while ensuring the deviation from the previous policy is relatively small.

C. Actor-critic methods

Actor-critic methods are hybrid methods that combine the benefits of policy-based and value-based algorithms. The policy structure that is responsible for selecting actions is known as the 'actor'. The estimated value function criticises the actions made by the actor and is known as the 'critic'. After each action selection, the critic evaluates the new state to determine whether the result of the selected action was better or worse than expected. Both networks need their gradient to learn. Let $J(\theta) := \mathbb{E}_{\pi_\theta}[r]$ represent a policy objective function, where θ designates the parameters of a DNN. Policy gradient methods search for local maximum of $J(\theta)$. Since optimization in continuous action spaces could be costly and slow, the DPG (Direct Policy Gradient) algorithm represents actions as parameterised function $\mu(s|\theta^\mu)$, where θ^μ refers to the parameters of the actor network. Then the unbiased estimate of the policy gradient gradient step is given as:

$$\nabla_\theta J = -\mathbb{E}_{\pi_\theta}\Big\{(g-b)\log \pi_\theta(a|s)\Big\}, \tag{5}$$

where b is the baseline. While using $b \equiv 0$ is the simplification that leads to the REINFORCE formulation. Williams [27] explains a well chosen baseline can reduce variance leading to a more stable learning. The baseline, b can be chosen as $V_\pi(s)$, $Q_\pi(s,a)$ or 'Advantage' $A_\pi(s,a)$ based methods. Deep Deterministic Policy Gradient (DDPG) [30] is a model-free, off-policy (please refer to subsection III-D for a detailed distinction), actor-critic algorithm that can learn policies for continuous action spaces using deep neural net based function approximation, extending prior work on DPG to large and high-dimensional state-action spaces. When selecting actions, exploration is performed by adding noise to the actor policy. Like DQN, to stabilise learning a replay buffer is used to minimize data correlation. A separate actor-critic specific target network is also used. Normal Q-learning is adapted with a restricted number of discrete actions, and DDPG also needs a straightforward way to choose an action. Starting from Q-learning, we extend Eqn. 2 to define the optimal Q-value and optimal action as Q^* and a^*.

$$Q^*(s,a) = \max_\pi Q_\pi(s,a), \quad a^* = \operatorname*{argmax}_a Q^*(s,a). \tag{6}$$

In the case of Q-learning, the action is chosen according to the Q-function as in Eqn. 6. But DDPG chains the evaluation of Q after the action has already been chosen according to the policy. By correcting the Q-values towards the optimal values using the chosen action, we also update the policy towards the optimal action proposition. Thus two separate networks work at estimating Q^* and π^*.

Asynchronous Advantage Actor Critic (A3C) [31] uses asynchronous gradient descent for optimization of deep neural network controllers. Deep reinforcement learning algorithms based on experience replay such as DQN and DDPG have demonstrated considerable success in difficult domains such as playing Atari games. However, experience replay uses a large amount of memory to store experience samples and requires off-policy learning algorithms. In A3C, instead of using an experience replay buffer, agents asynchronously execute on multiple parallel instances of the environment. In addition to the reducing correlation of the experiences, the parallel actor-learners have a stabilizing effect on training process. This simple setup enables a much larger spectrum of on-policy as well as off-policy reinforcement learning algorithms to be applied robustly using deep neural networks. A3C exceeded the performance of the previous state-of-the-art at the time on the Atari domain while training for half

the time on a single multi-core CPU instead of a GPU by combining several ideas. It also demonstrates how using an estimate of the value function as the previously explained baseline b reduces variance and improves convergence time. By defining the *advantage* as $A_\pi(a,s) = Q_\pi(s,a) - V_\pi(s)$, the expression of the policy gradient from Eqn. 5 is rewritten as $\nabla_\theta L = -\mathbb{E}_{\pi_\theta}\{A_\pi(a,s)\log\pi_\theta(a|s)\}$. The critic is trained to minimize $\frac{1}{2}\left\|A_{\pi_\theta}(a,s)\right\|^2$. The intuition of using advantage estimates rather than just discounted returns is to allow the agent to determine not just how good its actions were, but also how much better they turned out to be than expected, leading to reduced variance and more stable training. The A3C model also demonstrated good performance in 3D environments such as labyrinth exploration. Advantage Actor Critic (A2C) is a synchronous version of the asynchronous advantage actor critic model, that waits for each agent to finish its experience before conducting an update. The performance of both A2C and A3C is comparable. Most greedy policies must alternate between exploration and exploitation, and good exploration visits the states where the value estimate is uncertain. This way, exploration focuses on trying to find the most *uncertain* state paths as they bring valuable information. In addition to advantage, explained earlier, some methods use the entropy as the uncertainty quantity. Most A3C implementations include this as well. Two methods with common authors are energy-based policies [32] and more recent and with widespread use, the Soft Actor Critic (SAC) algorithm [33], both rely on adding an entropy term to the reward function, so we update the policy objective from Eqn. 1 to Eqn. 7. We refer readers to [33] for an in depth explanation of the expression

$$\pi^*_{MaxEnt} = \underset{\pi}{\operatorname{argmax}}\mathbb{E}_\pi\{\sum_t[r(s_t,a_t) + \alpha H(\pi(.|s_t))]\}, \quad (7)$$

shown here for illustration of how the entropy H is added.

D. Model-based (vs. Model-free) & On/Off Policy methods

In practical situations, interacting with the real environment could be limited due to many reasons including safety and cost. Learning a model for environment dynamics may reduce the amount of interactions required with the real environment. Moreover, exploration can be performed on the learned models. In the case of model-based approaches (*e.g.* Dyna-Q [34], R-max [35]), agents attempt to learn the transition function T and reward function R, which can be used when making action selections. Keeping a model approximation of the environment means storing knowledge of its dynamics, and allows for fewer, and sometimes, costly environment interactions. By contrast, in model-free approaches such knowledge is not a requirement. Instead, model-free learners sample the underlying MDP directly in order to gain knowledge about the unknown model, in the form of value function estimates for example. In Dyna-2 [36], the learning agent stores long-term and short-term memories, where a memory is defined as the set of features and corresponding parameters used by an agent to estimate the value function. Long-term memory is for general domain knowledge which is updated from real experience, while short-term memory is for specific local knowledge about the current situation, and the value function is a linear combination of long and short term memories.

Learning algorithms can be on-policy or off-policy depending on whether the updates are conducted on fresh trajectories generated by the policy or by another policy, that could be generated by an older version of the policy or provided by an expert. On-policy methods such as SARSA [37], estimate the value of a policy while using the same policy for control. However, off-policy methods such as Q-learning [25], use two policies: the behavior policy, the policy used to generate behavior; and the target policy, the one being improved on. An advantage of this separation is that the target policy may be deterministic (greedy), while the behavior policy can continue to sample all possible actions, [1].

E. Deep reinforcement learning (DRL)

Tabular representations are the simplest way to store learned estimates (of *e.g.* values, policies or models), where each state-action pair has a discrete estimate associated with it. When estimates are represented discretely, each additional feature tracked in the state leads to an exponential growth in the number of state-action pair values that must be stored [38]. This problem is commonly referred to in the literature as the "curse of dimensionality", a term originally coined by Bellman [39]. In simple environments this is rarely an issue, but it may lead to an intractable problem in real-world applications, due to memory and/or computational constraints. Learning over a large state-action space is possible, but may take an unacceptably long time to learn useful policies. Many real-world domains feature continuous state and/or action spaces; these can be discretised in many cases. However, large discretisation steps may limit the achievable performance in a domain, whereas small discretisation steps may result in a large state-action space where obtaining a sufficient number of samples for each state-action pair is impractical. Alternatively, function approximation may be used to generalise across states and/or actions, whereby a function approximator is used to store and retrieve estimates. Function approximation is an active area of research in RL, offering a way to handle continuous state and/or action spaces, mitigate against the state-action space explosion and generalise prior experience to previously unseen state-action pairs. Tile coding is one of the simplest forms of function approximation, where one tile represents multiple states or state-action pairs [38]. Neural networks are also commonly used to implement function approximation, one of the most famous examples being Tesuaro's application of RL to backgammon [40]. Recent work has applied deep neural networks as a function approximation method; this emerging paradigm is known as deep reinforcement learning (DRL). DRL algorithms have achieved human level performance (or above) on complex tasks such as playing Atari games [24] and playing the board game Go [41].

In DQN [24] it is demonstrated how a convolutional neural network can learn successful control policies from just raw video data for different Atari environments. The network was trained end-to-end and was not provided with any game specific information. The input to the convolutional neural

network consists of a $84\times84\times4$ tensor of 4 consecutive stacked frames used to capture the temporal information. Through consecutive layers, the network learns how to combine features in order to identify the action most likely to bring the best outcome. One layer consists of several convolutional filters. For instance, the first layer uses 32 filters with 8×8 kernels with stride 4 and applies a rectifier non linearity. The second layer is 64 filters of 4×4 with stride 2, followed by a rectifier non-linearity. Next comes a third convolutional layer of 64 filters of 3×3 with stride 1 followed by a rectifier. The last intermediate layer is composed of 512 rectifier units fully connected. The output layer is a fully-connected linear layer with a single output for each valid action. For DQN training stability, two networks are used while the parameters of the target network are fixed for a number of iterations while updating the online network parameters. For practical reasons, the $Q(s,a)$ function is modeled as a deep neural network that predicts the value of all actions given the input state. Accordingly, deciding what action to take requires performing a single forward pass of the network. Moreover, in order to increase sample efficiency, experiences of the agent are stored in a replay memory (experience replay), where the Q-learning updates are conducted on randomly selected samples from the replay memory. This random selection breaks the correlation between successive samples. Experience replay enables reinforcement learning agents to *remember* and reuse experiences from the past where observed transitions are stored for some time, usually in a queue, and sampled uniformly from this memory to update the network. However, this approach simply replays transitions at the same frequency that they were originally experienced, regardless of their significance. An alternative method is to use two separate experience buckets, one for positive and one for negative rewards [42]. Then a fixed fraction from each bucket is selected to replay. This method is only applicable in domains that have a natural notion of binary experience. Experience replay has also been extended with a framework for prioritising experience [43], where important transitions, based on the TD error, are replayed more frequently, leading to improved performance and faster training when compared to the standard experience replay approach.

The max operator in standard Q-learning and DQN uses the same values both to select and to evaluate an action resulting in over optimistic value estimates. In Double DQN (D-DQN) [44] the over estimation problem in DQN is tackled where the greedy policy is evaluated according to the online network and uses the target network to estimate its value. It was shown that this algorithm not only yields more accurate value estimates, but leads to much higher scores on several games.

In Dueling network architecture [45] the state value function and associated advantage function are estimated, and then combined together to estimate action value function. The advantage of the dueling architecture lies partly in its ability to learn the state-value function efficiently. In a single-stream architecture only the value for one of the actions is updated. However in dueling architecture, the value stream is updated with every update, allowing for better approximation of the state values, which in turn need to be accurate for temporal difference methods like Q-learning.

DRQN [46] applied a modification to the DQN by combining a Long Short Term Memory (LSTM) with a Deep Q-Network. Accordingly, the DRQN is capable of integrating information across frames to detect information such as velocity of objects. DRQN showed to generalize its policies in case of complete observations and when trained on Atari games and evaluated against flickering games, it was shown that DRQN generalizes better than DQN.

IV. Extensions to reinforcement learning

This section introduces and discusses some of the main extensions to the basic single-agent RL paradigms which have been introduced over the years. As well as broadening the applicability of RL algorithms, many of the extensions discussed here have been demonstrated to improve scalability, learning speed and/or converged performance in complex problem domains.

A. Reward shaping

As noted in Section III, the design of the reward function is crucial: RL agents seek to maximise the return from the reward function, therefore the optimal policy for a domain is defined with respect to the reward function. In many real-world application domains, learning may be difficult due to sparse and/or delayed rewards. RL agents typically learn how to act in their environment guided merely by the reward signal. Additional knowledge can be provided to a learner by the addition of a shaping reward to the reward naturally received from the environment, with the goal of improving learning speed and converged performance. This principle is referred to as reward shaping. The term shaping has its origins in the field of experimental psychology, and describes the idea of rewarding all behaviour that leads to the desired behaviour. Skinner [47] discovered while training a rat to push a lever that any movement in the direction of the lever had to be rewarded to encourage the rat to complete the task. Analogously to the rat, a RL agent may take an unacceptably long time to discover its goal when learning from delayed rewards, and shaping offers an opportunity to speed up the learning process. Reward shaping allows a reward function to be engineered in a way to provide more frequent feedback signal on appropriate behaviours [48], which is especially useful in domains with sparse rewards. Generally, the return from the reward function is modified as follows: $r' = r + f$ where r is the return from the original reward function R, f is the additional reward from a shaping function F, and r' is the signal given to the agent by the augmented reward function R'. Empirical evidence has shown that reward shaping can be a powerful tool to improve the learning speed of RL agents [49]. However, it can have unintended consequences. The implication of adding a shaping reward is that a policy which is optimal for the augmented reward function R' may not in fact also be optimal for the original reward function R. A classic example of reward shaping gone wrong for this exact reason is reported by [49] where the experimented bicycle agent would turn in circle to stay upright rather than reach its goal. Difference

rewards (D) [50] and potential-based reward shaping (*PBRS*) [51] are two commonly used shaping approaches. Both D and *PBRS* have been successfully applied to a wide range of application domains and have the added benefit of convenient theoretical guarantees, meaning that they do not suffer from the same issues as the unprincipled reward shaping approaches described above (see *e.g.* [51]–[55]).

B. Multi-agent reinforcement learning (MARL)

In multi-agent reinforcement learning, multiple RL agents are deployed into a common environment. The single-agent MDP framework becomes inadequate when multiple autonomous agents act simultaneously in the same domain. Instead, the more general stochastic game (SG) may be used in the case of a Multi-Agent System (MAS) [56]. A SG is defined as a tuple $< S, A_{1\ldots N}, T, R_{1\ldots N} >$, where N is the number of agents, S is the set of system states, A_i is the set of actions for agent i (and A is the joint action set), T is the transition function, and R_i is the reward function for agent i. The SG looks very similar to the MDP framework, apart from the addition of multiple agents. In fact, for the case of $N = 1$ a SG then becomes a MDP. The next system state and the rewards received by each agent depend on the joint action a of all of the agents in a SG, where a is derived from the combination of the individual actions a_i for each agent in the system. Each agent may have its own local state perception s_i, which is different to the system state s (i.e. individual agents are not assumed to have full observability of the system). Note also that each agent may receive a different reward for the same system state transition, as each agent has its own separate reward function R_i. In a SG, the agents may all have the same goal (collaborative SG), totally opposing goals (competitive SG), or there may be elements of collaboration and competition between agents (mixed SG). Whether RL agents in a MAS will learn to act together or at cross-purposes depends on the reward scheme used for a specific application.

C. Multi-objective reinforcement learning

In multi-objective reinforcement learning (MORL) the reward signal is a vector, where each component represents the performance on a different objective. The MORL framework was developed to handle sequential decision making problems where tradeoffs between conflicting objective functions must be considered. Examples of real-world problems with multiple objectives include selecting energy sources (tradeoffs between fuel cost and emissions) [57] and watershed management (tradeoffs between generating electricity, preserving reservoir levels and supplying drinking water) [58]. Solutions to MORL problems are often evaluated using the concept of Pareto dominance [59] and MORL algorithms typically seek to learn or approximate the set of non-dominated solutions. MORL problems may be defined using the MDP or SG framework as appropriate, in a similar manner to single-objective problems. The main difference lies in the definition of the reward function: instead of returning a single scalar value r, the reward function $\mathbf{R}$ in multi-objective domains returns a vector $\mathbf{r}$ consisting of the rewards for each individual objective $c \in C$. Therefore, a regular MDP or SG can be extended to a Multi-Objective MDP (MOMDP) or Multi-Objective SG (MOSG) by modifying the return of the reward function. For a more complete overview of MORL beyond the brief summary presented in this section, the interested reader is referred to recent surveys [60], [61].

D. State Representation Learning (SRL)

State Representation Learning refers to feature extraction & dimensionality reduction to represent the state space with its history conditioned by the actions and environment of the agent. A complete review of SRL for control is discussed in [62]. In the simplest form SRL maps a high dimensional vector o_t into a small dimensional latent space s_t. The inverse operation decodes the state back into an estimate of the original observation $\hat{o}_t$. The agent then learns to map from the latent space to the action. Training the SRL chain is unsupervised in the sense that no labels are required. Reducing the dimension of the input effectively simplifies the task as it removes noise and decreases the domain's size as shown in [63]. SRL could be a simple auto-encoder (AE), though various methods exist for observation reconstruction such as Variational Auto-Encoders (VAE) or Generative Adversarial Networks (GANs), as well as forward models for predicting the next state or inverse models for predicting the action given a transition. A good learned state representation should be Markovian; i.e. it should encode all necessary information to be able to select an action based on the current state only, and not any previous states or actions [62], [64].

E. Learning from Demonstrations

Learning from Demonstrations (LfD) is used by humans to acquire new skills in an expert to learner knowledge transmission process. LfD is important for initial exploration where reward signals are too sparse or the input domain is too large to cover. In LfD, an agent learns to perform a task from demonstrations, usually in the form of state-action pairs, provided by an expert without any feedback rewards. However, high quality and diverse demonstrations are hard to collect, leading to learning sub-optimal policies. Accordingly, learning merely from demonstrations can be used to initialize the learning agent with a good or safe policy, and then reinforcement learning can be conducted to enable the discovery of a better policy by interacting with the environment. Combining demonstrations and reinforcement learning has been conducted in recent research. AlphaGo [41], combines search tree with deep neural networks, initializes the policy network by supervised learning on state-action pairs provided by recorded games played by human experts. Additionally, a value network is trained to tell how desirable a board state is. By conducting self-play and reinforcement learning, AlphaGo is able to discover new stronger actions and learn from its mistakes, achieving super human performance. More recently, AlphaZero [65], developed by the same team, proposed a general framework for self-play models. AlphaZero is trained entirely using reinforcement learning and self play, starting from completely random play, and requires no prior knowledge

of human players. AlphaZero taught itself from scratch how to master the games of chess, shogi, and Go game, beating a world-champion program in each case. In [66] it is shown that given the initial demonstration, no explicit exploration is necessary, and we can attain near-optimal performance. Measuring the divergence between the current policy and the expert policy for optimization is proposed in [67]. DQfD [68] pre-trains the agent and uses expert demonstrations by adding them into the replay buffer with additional priority. Moreover, a training framework that combines learning from both demonstrations and reinforcement learning is proposed in [69] for fast learning agents. Two policies close to maximizing the reward function can still have large differences in behaviour. To avoid degenerating a solution which would fit the reward but not the original behaviour, authors [70] proposed a method for enforcing that the optimal policy learnt over the rewards should still match the observed policy in behavior. Behavior Cloning (BC) is applied as a supervised learning that maps states to actions based on demonstrations provided by an expert. On the other hand, Inverse Reinforcement Learning (IRL) is about inferring the reward function that justifies demonstrations of the expert. IRL is the problem of extracting a reward function given observed, optimal behavior [71]. A key motivation is that the reward function provides a succinct and robust definition of a task. Generally, IRL algorithms can be expensive to run, requiring reinforcement learning in an inner loop between cost estimation to policy training and evaluation. Generative Adversarial Imitation Learning (GAIL) [72] introduces a way to avoid this expensive inner loop. In practice, GAIL trains a policy close enough to the expert policy to fool a discriminator. This process is similar to GANs [73], [74]. The resulting policy must travel the same MDP states as the expert, or the discriminator would pick up the differences. The theory behind GAIL is an equation simplification: qualitatively, if IRL is going from demonstrations to a cost function and RL from a cost function to a policy, then we should altogether be able to go from demonstration to policy in a single equation while avoiding the cost function estimation.

V. Reinforcement learning for Autonomous driving tasks

Autonomous driving tasks where RL could be applied include: controller optimization, path planning and trajectory optimization, motion planning and dynamic path planning, development of high-level driving policies for complex navigation tasks, scenario-based policy learning for highways, intersections, merges and splits, reward learning with inverse reinforcement learning from expert data for intent prediction for traffic actors such as pedestrian, vehicles and finally learning of policies that ensures safety and perform risk estimation. Before discussing the applications of DRL to AD tasks we briefly review the state space, action space and rewards schemes in autonomous driving setting.

A. State Spaces, Action Spaces and Rewards

To successfully apply DRL to autonomous driving tasks, designing appropriate state spaces, action spaces, and reward functions is important. Leurent *et al.* [75] provided a comprehensive review of the different state and action representations which are used in autonomous driving research. Commonly used state space features for an autonomous vehicle include: position, heading and velocity of ego-vehicle, as well as other obstacles in the sensor view extent of the ego-vehicle. To avoid variations in the dimension of the state space, a Cartesian or Polar occupancy grid around the ego vehicle is frequently employed. This is further augmented with lane information such as lane number (ego-lane or others), path curvature, past and future trajectory of the ego-vehicle, longitudinal information such as Time-to-collision (TTC), and finally scene information such as traffic laws and signal locations.

Using raw sensor data such as camera images, LiDAR, radar, etc. provides the benefit of finer contextual information, while using condensed abstracted data reduces the complexity of the state space. In between, a mid-level representation such as 2D bird eye view (BEV) is sensor agnostic but still close to the spatial organization of the scene. Fig. 4 is an illustration of a top down view showing an occupancy grid, past and projected trajectories, and semantic information about the scene such as the position of traffic lights. This intermediary format retains the spatial layout of roads when graph-based representations would not. Some simulators offer this view such as Carla or Flow (see Table V-C).

A vehicle policy must control a number of different actuators. Continuous-valued actuators for vehicle control include steering angle, throttle and brake. Other actuators such as gear changes are discrete. To reduce complexity and allow the application of DRL algorithms which work with discrete action spaces only (*e.g.* DQN), an action space may be discretised uniformly by dividing the range of continuous actuators such as steering angle, throttle and brake into equal-sized bins (see Section VI-C). Discretisation in log-space has also been suggested, as many steering angles which are selected in practice are close to the centre [76]. Discretisation does have disadvantages however; it can lead to jerky or unstable trajectories if the step values between actions are too large. Furthermore, when selecting the number of bins for an actuator there is a trade-off between having enough discrete steps to allow for smooth control, and not having so many steps that action selections become prohibitively expensive to evaluate. As an alternative to discretisation, continuous values for actuators may also be handled by DRL algorithms which learn a policy directly, (*e.g.* DDPG). Temporal abstractions options framework [77]) may also be employed to simplify the process of selecting actions, where agents select *options* instead of low-level actions. These options represent a sub-policy that could extend a primitive action over multiple time steps.

Designing reward functions for DRL agents for autonomous driving is still very much an open question. Examples of criteria for AD tasks include: distance travelled towards a destination [78], speed of the ego vehicle [78]–[80], keeping the ego vehicle at a standstill [81], collisions with other road users or scene objects [78], [79], infractions on sidewalks [78], keeping in lane, and maintaining comfort and stability while avoiding extreme acceleration, braking or steering [80], [81],

and following traffic rules [79].

B. Motion Planning & Trajectory optimization

Motion planning is the task of ensuring the existence of a path between target and destination points. This is necessary to plan trajectories for vehicles over prior maps usually augmented with semantic information. Path planning in dynamic environments and varying vehicle dynamics is a key problem in autonomous driving, for example negotiating right to pass through in an intersection [87], merging into highways. Recent work by authors [89] contains real world motions by various traffic actors, observed in diverse interactive driving scenarios. Recently, authors demonstrated an application of DRL (DDPG) for AD using a full-sized autonomous vehicle [90]. The system was first trained in simulation, before being trained in real time using on board computers, and was able to learn to follow a lane, successfully completing a real-world trial on a 250 metre section of road. Model-based deep RL algorithms have been proposed for learning models and policies directly from raw pixel inputs [91], [92]. In [93], deep neural networks have been used to generate predictions in simulated environments over hundreds of time steps. RL is also suitable for Control. Classical optimal control methods like LQR/iLQR are compared with RL methods in [94]. Classical RL methods are used to perform optimal control in stochastic settings, for example the Linear Quadratic Regulator (LQR) in linear regimes and iterative LQR (iLQR) for non-linear regimes are utilized. A recent study in [95] demonstrates that random search over the parameters for a policy network can perform as well as LQR.

C. Simulator & Scenario generation tools

Autonomous driving datasets address supervised learning setup with training sets containing image, label pairs for various modalities. Reinforcement learning requires an environment where state-action pairs can be recovered while modelling dynamics of the vehicle state, environment as well as the stochasticity in the movement and actions of the environment and agent respectively. Various simulators are actively used for training and validating reinforcement learning algorithms. Table V-C summarises various high fidelity perception simulators capable of simulating cameras, LiDARs and radar. Some simulators are also capable of providing the vehicle state and dynamics. A complete review of sensors and simulators utilised within the autonomous driving community is available in [105] for readers. Learned driving policies are stress tested in simulated environments before moving on to costly evaluations in the real world. Multi-fidelity reinforcement learning (MFRL) framework is proposed in [106] where multiple simulators are available. In MFRL, a cascade of simulators with increasing fidelity are used in representing state dynamics (and thus computational cost) that enables the training and validation of RL algorithms, while finding near optimal policies for the real world with fewer expensive real world samples using a remote controlled car. CARLA Challenge [107] is a Carla simulator based AD competition with pre-crash scenarios characterized in a National Highway Traffic Safety Administration report [108]. The systems are evaluated in critical scenarios such as: Ego-vehicle loses control, ego-vehicle reacts to unseen obstacle, lane change to evade slow leading vehicle among others. The scores of agents are evaluated as a function of the aggregated distance travelled in different circuits, and total points discounted due to infractions.

D. LfD and IRL for AD applications

Early work on Behavior Cloning (BC) for driving cars in [109], [110] presented agents that learn form demonstrations (LfD) that tries to mimic the behavior of an expert. BC is typically implemented as a supervised learning, and accordingly, it is hard for BC to adapt to new, unseen situations. An architecture for learning a convolutional neural network, end to end, in self-driving cars domain was proposed in [111], [112]. The CNN is trained to map raw pixels from a single front facing camera directly to steering commands. Using a relatively small training dataset from humans/experts, the system learns to drive in traffic on local roads with or without lane markings and on highways. The network learns image representations that detect the road successfully, without being explicitly trained to do so. Authors of [113] proposed to learn comfortable driving trajectories optimization using expert demonstration from human drivers using Maximum Entropy Inverse RL. Authors of [114] used DQN as the refinement step in IRL to extract the rewards, in an effort learn human-like lane change behavior.

VI. Real world challenges and future perspectives

In this section, challenges for conducting reinforcement learning for real-world autonomous driving are presented and discussed along with the related research approaches for solving them.

A. Validating RL systems

Henderson *et al.* [115] described challenges in validating reinforcement learning methods focusing on policy gradient methods for continuous control algorithms such as PPO, DDPG and TRPO as well as in reproducing benchmarks. They demonstrate with real examples that implementations often have varying code-bases and different hyper-parameter values, and that unprincipled ways to estimate the top-k rollouts could lead to incoherent interpretations on the performance of the reinforcement learning algorithms, and further more on how well they generalize. Authors concluded that evaluation could be performed either on a well defined common setup or on real-world tasks. Authors in [116] proposed automated generation of challenging and rare driving scenarios in high-fidelity photo-realistic simulators. These adversarial scenarios are automatically discovered by parameterising the behavior of pedestrians and other vehicles on the road. Moreover, it is shown that by adding these scenarios to the training data of imitation learning, the safety is increased.

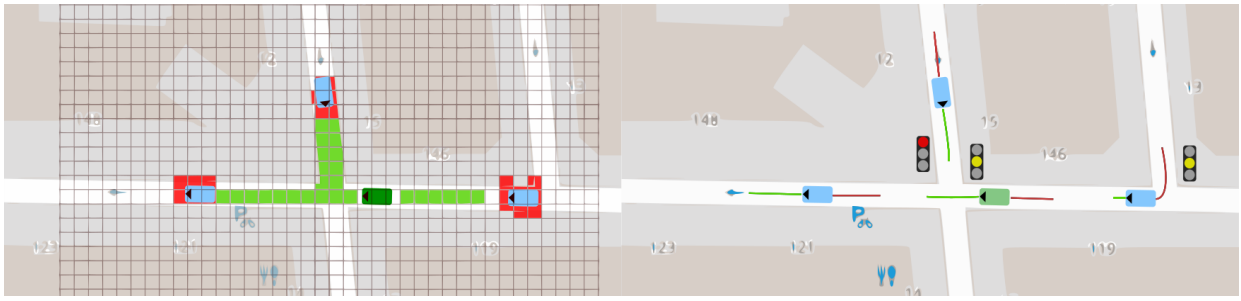

Fig. 4. Bird Eye View (BEV) 2D representation of a driving scene. Left demonstrates an occupancy grid. Right shows the combination of semantic information (traffic lights) with past (red) and projected (green) trajectories. The ego car is represented by a green rectangle in both images.

AD Task	(D)RL method & description	Improvements & Tradeoffs
Lane Keep	1. Authors [82] propose a DRL system for discrete actions (DQN) and continuous actions (DDAC) using the TORCS simulator (see Table V-C) 2. Authors [83] learn discretised and continuous policies using DQNs and Deep Deterministic Actor Critic (DDAC) to follow the lane and maximize average velocity.	1. This study concludes that using continuous actions provide smoother trajectories, though on the negative side lead to more restricted termination conditions & slower convergence time to learn. 2. Removing memory replay in DQNs help for faster convergence & better performance. The one hot encoding of action space resulted in abrupt steering control. While DDAC's continuous policy helps smooth the actions and provides better performance.
Lane Change	Authors [84] use Q-learning to learn a policy for ego-vehicle to perform no operation, lane change to left/right, accelerate/decelerate.	This approach is more robust compared to traditional approaches which consist in defining fixed way points, velocity profiles and curvature of path to be followed by the ego vehicle.
Ramp Merging	Authors [85] propose recurrent architectures namely LSTMs to model longterm dependencies for ego vehicles ramp merging into a highway.	Past history of the state information is used to perform the merge more robustly.
Overtaking	Authors [86] propose Multi-goal RL policy that is learnt by Q-Learning or Double action Q-Learning(DAQL) is employed to determine individual action decisions based on whether the other vehicle interacts with the agent for that particular goal.	Improved speed for lane keeping and overtaking with collision avoidance.
Intersections	Authors use DQN to evalute the Q-value for state-action pairs to negotiate intersection [87],	Creep-Go actions defined by authors enables the vehicle to maneuver intersections with restricted spaces and visibility more safely
Motion Planning	Authors [88] propose an improved A^* algorithm to learn a heuristic function using deep neural netowrks over image-based input obstacle map	Smooth control behavior of vehicle and better peformance compared to multi-step DQN

TABLE I
LIST OF AD TASKS THAT REQUIRE D(RL) TO LEARN A POLICY OR BEHAVIOR.

B. Bridging the simulation-reality gap

Simulation-to-real-world transfer learning is an active domain, since simulations are a source large & cheap data with perfect annotations. Authors [117] train a robot arm to grasp objects in the real world by performing domain adaption from simulation-to-reality, at both feature-level and pixel-level. The vision-based grasping system achieved comparable performance with 50 times fewer real-world samples. Authors in [118], randomized the dynamics of the simulator during training. The resulting policies were capable of generalising to different dynamics without requiring retraining on real system. In the domain of autonomous driving, authors [119] train an A3C agent using simulation-real translated images of the driving environment. Following which, the trained policy was evaluated on a real world driving dataset.

Authors in [120] addressed the issue of performing imitation learning in simulation that transfers well to images from real world. They achieved this by unsupervised domain translation between simulated and real world images, that enables learning the prediction of steering in the real world domain with only ground truth from the simulated domain. Authors remark that there were no pairwise correspondences between images in the simulated training set and the unlabelled real-world image set. Similarly, [121] performs domain adaptation to map real world images to simulated images. In contrast to sim-to-real methods they handle the reality gap during deployment of agents in real scenarios, by adapting the real camera streams to the synthetic modality, so as to map the unfamiliar or unseen features of real images back into the simulated environment and states. The agents have already learnt a policy in simulation.

C. Sample efficiency

Animals are usually able to learn new tasks in just a few trials, benefiting from their prior knowledge about the environment. However, one of the key challenges for reinforcement learning is sample efficiency. The learning process requires too many samples to learn a reasonable policy. This issue becomes more noticeable when collection of valuable experience is

Simulator	Description
CARLA [78]	Urban simulator, Camera & LIDAR streams, with depth & semantic segmentation, Location information
TORCS [96]	Racing Simulator, Camera stream, agent positions, testing control policies for vehicles
AIRSIM [97]	Camera stream with depth and semantic segmentation, support for drones
GAZEBO (ROS) [98]	Multi-robot physics simulator employed for path planning & vehicle control in complex 2D & 3D maps
SUMO [99]	Macro-scale modelling of traffic in cities motion planning simulators are used
DeepDrive [100]	Driving simulator based on unreal, providing multi-camera (eight) stream with depth
Constellation [101]	NVIDIA DRIVE Constellation^{TM} simulates camera, LIDAR & radar for AD (Proprietary)
MADRaS [102]	Multi-Agent Autonomous Driving Simulator built on top of TORCS
Flow [103]	Multi-Agent Traffic Control Simulator built on top of SUMO
Highway-env [104]	A gym-based environment that provides a simulator for highway based road topologies
Carcraft	Waymo's simulation environment (Proprietary)

TABLE II
SIMULATORS FOR RL APPLICATIONS IN ADVANCED DRIVING ASSISTANCE SYSTEMS (ADAS) AND AUTONOMOUS DRIVING.

expensive or even risky to acquire. In the case of robot control and autonomous driving, sample efficiency is a difficult issue due to the delayed and sparse rewards found in typical settings, along with an unbalanced distribution of observations in a large state space.

Reward shaping enables the agent to learn intermediate goals by designing a more frequent reward function to encourage the agent to learn faster from fewer samples. Authors in [122] design a second "trauma" replay memory that contains only collision situations in order to pool positive and negative experiences at each training step.

IL boostrapped RL: Further efficiency can be achieved where the agent first learns an initial policy offline performing imitation learning from roll-outs provided by an expert. Following which, the agent can self-improve by applying RL while interacting with the environment.

Actor Critic with Experience Replay (ACER) [123], is a sample-efficient policy gradient algorithm that makes use of a replay buffer, enabling it to perform more than one gradient update using each piece of sampled experience, as well as a trust region policy optimization method.

Transfer learning is another approach for sample efficiency, which enables the reuse of previously trained policy for a source task to initialize the learning of a target task. Policy composition presented in [124] propose composing previously learned basis policies to be able to reuse them for a novel task, which leads to faster learning of new policies. A survey on transfer learning in RL is presented in [125]. Multi-fidelity reinforcement learning (MFRL) framework [106] showed to transfer heuristics to guide exploration in high fidelity simulators and find near optimal policies for the real world with fewer real world samples. Authors in [126] transferred policies learnt to handle simulated intersections to real world examples between DQN agents.

Meta-learning algorithms enable agents adapt to new tasks and learn new skills rapidly from small amounts of experiences, benefiting from their prior knowledge about the world. Authors of [127] addressed this issue through training a recurrent neural network on a training set of interrelated tasks, where the network input includes the action selected in addition to the reward received in the previous time step. Accordingly, the agent is trained to learn to exploit the structure of the problem dynamically and solve new problems by adjusting its hidden state. A similar approach for designing RL algorithms is presented in [128]. Rather than designing a "fast" reinforcement learning algorithm, it is represented as a recurrent neural network, and learned from data. In Model-Agnostic Meta-Learning (MAML) proposed in [129], the meta-learner seeks to find an initialisation for the parameters of a neural network, that can be adapted quickly for a new task using only few examples. Reptile [130] includes a similar model. Authors [131] present simple gradient-based meta-learning algorithm.

Efficient state representations : World models proposed in [132] learn a compressed spatial and temporal representation of the environment using VAEs. Further on a compact and simple policy directly from the compressed state representation.

D. Exploration issues with Imitation

In imitation learning, the agent makes use of trajectories provided by an expert. However, the distribution of states the expert encounters usually does not cover all the states the trained agent may encounter during testing. Furthermore imitation assumes that the actions are independent and identically distributed (i.i.d.). One solution consists in using the Data Aggregation (DAgger) methods [133] where the end-to-end learned policy is executed, and extracted observation-action pairs are again labelled by the expert, and aggregated to the original expert observation-action dataset. Thus, iteratively collecting training examples from both reference and trained policies explores more valuable states and solves this lack of exploration. Following work on Search-based Structured Prediction (SEARN) [133], Stochastic Mixing Iterative Learning (SMILE) trains a stochastic stationary policy over several iterations and then makes use of a geometric stochastic mixing of the policies trained. In a standard imitation learning scenario, the demonstrator is required to cover sufficient states so as to avoid unseen states during test. This constraint is costly and requires frequent human intervention. More recently, Chauffeurnet [134] demonstrated the limits of imitation learning where even 30 million state-action samples were insufficient to learn an optimal policy that mapped bird-eye view images (states) to control (action). The authors propose the use of simulated examples which introduced perturbations, higher diversity of scenarios such as collisions and/or going off the road. The *featurenet* includes an agent RNN that

outputs the way point, agent box position and heading at each iteration. Authors [135] identify limits of imitation learning, and train a DNN end-to-end using the ego vehicles on input raw image, and 2d and 3d locations of neighboring vehicles to simultaneously predict the ego-vehicle action as well as neighbouring vehicle trajectories.

E. Intrinsic Reward functions

In controlled simulated environments such as games, an explicit reward signal is given to the agent along with its sensor stream. However, in real-world robotics and autonomous driving deriving, designing a *good* reward functions is essential so that the desired behaviour may be learned. The most common solution has been reward shaping [136] and consists in supplying additional well designed rewards to the agent to encourage the optimization into the direction of the optimal policy. Rewards as already pointed earlier in the paper, could be estimated by inverse RL (IRL) [137], which depends on expert demonstrations. In the absence of an explicit reward shaping and expert demonstrations, agents can use intrinsic rewards or intrinsic motivation [138] to evaluate if their actions were good or not. Authors of [139] define *curiosity* as the error in an agent's ability to predict the consequence of its own actions in a visual feature space learned by a self-supervised inverse dynamics model. In [140] the agent learns a next state predictor model from its experience, and uses the error of the prediction as an intrinsic reward. This enables that agent to determine what could be a useful behavior even without extrinsic rewards.

F. Incorporating safety in DRL

Deploying an autonomous vehicle in real environments after training directly could be dangerous. Different approaches to incorporate safety into DRL algorithms are presented here. For imitation learning based systems, Safe DAgger [141] introduces a safety policy that learns to predict the error made by a primary policy trained initially with the supervised learning approach, without querying a reference policy. An additional safe policy takes both the partial observation of a state and a primary policy as inputs, and returns a binary label indicating whether the primary policy is likely to deviate from a reference policy without querying it. Authors of [142] addressed safety in multi-agent Reinforcement Learning for Autonomous Driving, where a balance is maintained between unexpected behavior of other drivers or pedestrians and not to be too defensive, so that normal traffic flow is achieved. While hard constraints are maintained to guarantee the safety of driving, the problem is decomposed into a composition of a policy for desires to enable comfort driving and trajectory planning. The deep reinforcement learning algorithms for control such as DDPG and safety based control are combined in [143], including artificial potential field method that is widely used for robot path planning. Using TORCS environment, the DDPG is applied first for learning a driving policy in a stable and familiar environment, then policy network and safety-based control are combined to avoid collisions. It was found that combination of DRL and safety-based control performs well in most scenarios. In order to enable DRL to escape local optima, speed up the training process and avoid danger conditions or accidents, Survival-Oriented Reinforcement Learning (SORL) model is proposed in [144], where survival is favored over maximizing total reward through modeling the autonomous driving problem as a constrained MDP and introducing Negative-Avoidance Function to learn from previous failure. The SORL model was found to be not sensitive to reward function and can use different DRL algorithms like DDPG. Furthermore, a comprehensive survey on safe reinforcement learning can be found in [145] for interested readers.

G. Multi-agent reinforcement learning

Autonomous driving is a fundamentally multi-agent task; as well as the ego vehicle being controlled by an agent, there will also be many other actors present in simulated and real world autonomous driving settings, such as pedestrians, cyclists and other vehicles. Therefore, the continued development of explicitly multi-agent approaches to learning to drive autonomous vehicles is an important future research direction. Several prior methods have already approached the autonomous driving problem using a MARL perspective, e.g. [142], [146]–[149].

One important area where MARL techniques could be very beneficial is in high-level decision making and coordination between groups of autonomous vehicles, in scenarios such as overtaking in highway scenarios [149], or negotiating intersections without signalised control. Another area where MARL approaches could be of benefit is in the development of adversarial agents for testing autonomous driving policies before deployment [148], i.e. agents controlling other vehicles in a simulation that learn to expose weaknesses in the behaviour of autonomous driving policies by acting erratically or against the rules of the road. Finally, MARL approaches could potentially have an important role to play in developing safe policies for autonomous driving [142], as discussed earlier.

VII. Conclusion

Reinforcement learning is still an active and emerging area in real-world autonomous driving applications. Although there are a few successful commercial applications, there is very little literature or large-scale public datasets available. Thus we were motivated to formalize and organize RL applications for autonomous driving. Autonomous driving scenarios involve interacting agents and require negotiation and dynamic decision making which suits RL. However, there are many challenges to be resolved in order to have mature solutions which we discuss in detail. In this work, a detailed theoretical reinforcement learning is presented, along with a comprehensive literature survey about applying RL for autonomous driving tasks.

Challenges, future research directions and opportunities are discussed in section VI. This includes : validating the performance of RL based systems, the simulation-reality gap, sample efficiency, designing good reward functions, incorporating safety into decision making RL systems for autonomous agents.

Framework	Description
OpenAI Baselines [150]	Set of high-quality implementations of different RL and DRL algorithms. The main goal for these Baselines is to make it easier for the research community to replicate, refine and create reproducible research.
Unity ML Agents Toolkit [151]	Implements core RL algorithms, games, simulations environments for training RL or IL based agents .
RL Coach [152]	Intel AI Lab's implementation of modular RL algorithms implementation with simple integration of new environments by extending and reusing existing components.
Tensorflow Agents [153]	RL algorithms package with Bandits from TF.
rlpyt [154]	implements deep Q-learning, policy gradients algorithm families in a single python package
bsuite [155]	DeepMind Behaviour Suite for Reinforcement Learning aims at defining metrics for RL agents. Automating evaluation and analysis.

TABLE III
OPEN-SOURCE FRAMEWORKS AND PACKAGES FOR STATE OF THE ART RL/DRL ALGORITHMS AND EVALUATION.

Reinforcement learning results are usually difficult to reproduce and are highly sensitive to hyper-parameter choices, which are often not reported in detail. Both researchers and practitioners need to have a reliable starting point where the well known reinforcement learning algorithms are implemented, documented and well tested. These frameworks have been covered in table VI-G.

The development of explicitly multi-agent reinforcement learning approaches to the autonomous driving problem is also an important future challenge that has not received a lot of attention to date. MARL techniques have the potential to make coordination and high-level decision making between groups of autonomous vehicles easier, as well as providing new opportunities for testing and validating the safety of autonomous driving policies.

Furthermore, implementation of RL algorithms is a challenging task for researchers and practitioners. This work presents examples of well known and active open-source RL frameworks that provide well documented implementations that enables the opportunity of using, evaluating and extending different RL algorithms. Finally, We hope that this overview paper encourages further research and applications.

A2C, A3C	Advantage Actor Critic, Asynchronous A2C
BC	Behavior Cloning
DDPG	Deep DPG
DP	Dynamic Programming
DPG	Deterministic PG
DQN	Deep Q-Network
DRL	Deep RL
IL	Imitation Learning
IRL	Inverse RL
LfD	Learning from Demonstration
MAML	Model-Agnostic Meta-Learning
MARL	Multi-Agent RL
MDP	Markov Decision Process
MOMDP	Multi-Objective MDP
MOSG	Multi-Objective SG
PG	Policy Gradient
POMDP	Partially Observed MDP
PPO	Proximal Policy Optimization
QL	Q-Learning
RRT	Rapidly-exploring Random Trees
SG	Stochastic Game
SMDP	Semi-Markov Decision Process
TDL	Time Difference Learning
TRPO	Trust Region Policy Optimization

TABLE IV
ACRONYMS RELATED TO REINFORCEMENT LEARNING (RL).

REFERENCES

[1] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction (Second Edition)*. MIT Press, 2018. 1, 4, 5, 6

[2] 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," in *Proceedings of the 14th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications - Volume 5 VISAPP: VISAPP,*, INSTICC. SciTePress, 2019, pp. 564–572. 1

[3] M. Siam, S. Elkerdawy, M. Jagersand, and S. Yogamani, "Deep semantic segmentation for automated driving: Taxonomy, roadmap and challenges," in *2017 IEEE 20th international conference on intelligent transportation systems (ITSC)*. IEEE, 2017, pp. 1–8. 2

[4] K. El Madawi, H. Rashed, A. El Sallab, O. Nasr, H. Kamel, and S. Yogamani, "Rgb and lidar fusion based 3d semantic segmentation for autonomous driving," in *2019 IEEE Intelligent Transportation Systems Conference (ITSC)*. IEEE, 2019, pp. 7–12. 2

[5] M. Siam, H. Mahgoub, M. Zahran, S. Yogamani, M. Jagersand, and A. El-Sallab, "Modnet: Motion and appearance based moving object detection network for autonomous driving," in *2018 21st International Conference on Intelligent Transportation Systems (ITSC)*. IEEE, 2018, pp. 2859–2864. 2

[6] V. R. Kumar, S. Milz, C. Witt, M. Simon, K. Amende, J. Petzold, S. Yogamani, and T. Pech, "Monocular fisheye camera depth estimation using sparse lidar supervision," in *2018 21st International Conference on Intelligent Transportation Systems (ITSC)*. IEEE, 2018, pp. 2853–2858. 2

[7] M. Uřičář, P. Křížek, G. Sistu, and S. Yogamani, "Soilingnet: Soiling detection on automotive surround-view cameras," in *2019 IEEE Intelligent Transportation Systems Conference (ITSC)*. IEEE, 2019, pp. 67–72. 2

[8] G. Sistu, I. Leang, S. Chennupati, S. Yogamani, C. Hughes, S. Milz, and S. Rawashdeh, "Neurall: Towards a unified visual perception model for automated driving," in *2019 IEEE Intelligent Transportation Systems Conference (ITSC)*. IEEE, 2019, pp. 796–803. 2

[9] S. Yogamani, C. Hughes, J. Horgan, G. Sistu, P. Varley, D. O'Dea, M. Uricár, S. Milz, M. Simon, K. Amende *et al.*, "Woodscape: A multi-task, multi-camera fisheye dataset for autonomous driving," in *Proceedings of the IEEE International Conference on Computer Vision*, 2019, pp. 9308–9318. 2

[10] S. Milz, G. Arbeiter, C. Witt, B. Abdallah, and S. Yogamani, "Visual slam for automated driving: Exploring the applications of deep learning," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, 2018, pp. 247–257. 2

[11] S. M. LaValle, *Planning Algorithms*. New York, NY, USA: Cambridge University Press, 2006. 2

[12] S. M. LaValle and J. James J. Kuffner, "Randomized kinodynamic planning," *The International Journal of Robotics Research*, vol. 20, no. 5, pp. 378–400, 2001. 2

[13] T. D. Team. Dimensions publication trends. [Online]. Available: https://app.dimensions.ai/discover/publication 3

[14] Y. Kuwata, J. Teo, G. Fiore, S. Karaman, E. Frazzoli, and J. P. How, "Real-time motion planning with applications to autonomous urban driving," *IEEE Transactions on Control Systems Technology*, vol. 17, no. 5, pp. 1105–1118, 2009. 3

[15] B. Paden, M. Čáp, S. Z. Yong, D. Yershov, and E. Frazzoli, "A survey of motion planning and control techniques for self-driving urban vehicles,"