Alma Mater Studiorum – Università di Bologna
Cesena Campus

Department of Computer Science and Engineering

Master's Degree in Computer Science and Engineering

Integrating BDI and Reinforcement Learning: the Case Study of Autonomous Driving

Supervisor:
Prof. Alessandro Ricci
Co-supervisor:
Prof. Giovanni Pau

Author:
Michael Bosello

Academic Year 2019-2020

Abstract

Recent breakthroughs in machine learning are paving the way to the vision of software 2.0 era, which foresees the replacement of traditional software development with such techniques for many applications. In the context of agent-oriented programming, we believe that mixing together cognitive architectures like the BDI one and learning techniques could trigger new interesting scenarios. In that view, our previous work presents Jason-RL, a framework that integrates BDI agents and Reinforcement Learning (RL) more deeply than what has been already proposed so far in the literature. The framework allows the development of BDI agents having both explicitly programmed plans and plans learned by the agent using RL. The two kinds of plans are seamlessly integrated and can be used without differences. Here, we take autonomous driving as a case study to verify the advantages of the proposed approach and framework. The BDI agent has hard-coded plans that define high-level directions while fine-grained navigation is learned by trial and error. This approach – compared to plain RL – is encouraging as RL struggles in temporally extended planning. We defined and trained an agent able to drive in a track with an intersection, at which it has to choose the correct path to reach the assigned target. A first step towards porting the system in the real-world has been done by building a 1/10 scale racecar prototype which learned how to drive in a simple track.

Contents

Chapter 1

Introduction

Machine Learning (ML) and cognitive computing techniques have been getting a momentum in recent years, thanks to several factors, including theoretical developments in contexts such as deep Neural Networks (NNs), Reinforcement Learning (RL), Bayesian networks, the availability of big data and the availability of more and more powerful parallel computing machinery (GPU, cloud) [1, 2, 3, 4]. Their deeper and deeper impact in real-world applications is celebrating a new "AI Spring" era, which is generating a strong debate in the literature as well [5]. Actually, the impact is not only about the kind of applications but also about *how applications are programmed and engineered*. In particular, a vision of *Software 2.0* era is emerging [6], in which traditional programming and software development is meant to be more and more replaced by e.g. machine learning and cognitive computing techniques, towards the "the end of programming" era [4, 7].

If we consider *Agent-Oriented Programming* (AOP) [8], and more in general Multi-Agent Systems (MAS) engineering, we believe that blending cognitive architectures such as the Belief-Desire-Intention (BDI) one [9] with learning techniques will lead agent-oriented programming evolve toward the software 2.0 era. In that view, in our previous work we have proposed the Jason-RL framework [10], based on the Jason agent platform [11, 12], which integrates BDI agents and reinforcement learning [13] so as to systematically exploit RL in the agent development stage. The framework allows developers to integrate the explicit programming of plans – when developing a BDI agent – with the possibility that, for some goals, it would be the agent itself to learn the plans to achieve them, by exploiting RL based techniques. This is not only for a specific ad hoc problem but as a general feature of the agent platform. The development of certain plans moves from programming to designing an environment suitable for learning. In this sense, the development of an agent is metaphorically similar to an *education*

process, in which first an agent is created with a set of basic programmed plans and then *grow up* in order to learn plans to achieve the goals for which the agent is meant to be designed.

In this dissertation, the Jason-RL framework is applied to the interesting case study of autonomous driving to assess the advantages and effectiveness of the proposed approach, framework, and model. In the case of autonomous driving, the above-mentioned education process resembles our idea of robot-drivers as teen-drivers [14], which learns new patterns and behaviors by trial and error in a fashion that resembles young human driving students. In [14], 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 so as to open the doors to the use of reinforcement learning techniques in the real world.

In the problem of autonomous driving, the agent has to follow high-level directions while relying on environment perceptions to take decisions and navigate without incidents. In this case study, the hard-coded plans handle the high-level planning of the path so as to choose the right direction to the target, while low-level control, that uses sensors and actuators to ensure moving without incidents, is achieved by using the learning capabilities of the framework. This approach is promising as RL struggles in *temporally extended planning* [15, 16] meanwhile humans struggles in defining hard-to-engineer behaviors like fine-grained navigation. Thanks to Jason-RL, we can benefit from the strengths of both methods. The environment used in the experiment is a track with an intersection and two targets (randomly alternating). The agent has to choose the right direction at the intersection to reach the proper target. The agent uses its approximate position for high-level planning, while it uses LIDAR measurements for low-level control. In the end, we successfully defined and trained a BDI agent able to drive in the track and able to handle the intersection.

In our experiments, we used the F1/10 (or f1tenth) platform [17]. F1tenth is an open-source 1/10 scale racecar featuring *realistic dynamics* and hardware/software stacks similar to full-scale solutions. The platform offers a simulator that can run the same code of the physical f1tenth. At this stage, we were not able to build a real-world track with an intersection due to space constraints, so, we performed the main experiment on the simulator. Nevertheless, we did a first step toward the use of our system in the real-world. We built an f1tenth 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. In the real-world experiment, we have trained the agent

to drive on a simple race track (without intersections and thus without the BDI part) using *real* LIDAR data as input, which has been considered an *open problem* [18]. As a next step, we plan to perform also the intersection experiment in the real-world.

Chapter 2

Background

2.1 Belief-Desire-Intention

Agent-oriented programming was proposed as a new programming paradigm based on the cognitive and societal view of computation [8] thus providing an abstraction tool as a convenient shorthand for talking about complex systems. Since then, multiple models, languages, and frameworks were submitted. The AOP approach is suitable for developing complex systems by treating the various system components as autonomous and interactive agents, and its benefits have been illustrated in various works [19]. One of the most successful models for developing rational agents is the Belief-Desire-Intentions model [9]; the central idea is that one can talk about computer programs as if they have a "mental state" made of (1) beliefs, the information the agent has about the world that could be out of date or inaccurate; (2) desires, all the possible states of affairs that the agent might like to accomplish; (3) intentions, the states of affairs that the agent has decided to work towards [12]. These are the basic data structures while the actual mechanism that defines the agent behavior lies in the theory of human practical reasoning [20]. In essence, it provides a way for the agent to choose which desires to pursue (deliberation) and deciding how to achieve an end (means-ends reasoning). This last activity, i.e. defining the course of actions to reach a goal, also known as *planning* in the AI community, is left to the developer because of its computational complexity thereby reducing the agent task to the selection of available plans. We must also emphasize that a characteristic of agents is that they are situated in their environment i.e, they can perceive the environment to some extent and act on it.

2.2 Jason

Jason [12] is an interpreter for an agent programming language that extends AgentSpeak [9], a popular AOP language. Jason is based on Java and it is a state-of-the-art platform for AOP. For our purposes, we consider Jason as a representative instance of AOP based on the BDI model. The Jason platform implements a customizable agent architecture that allows programmers to build and run multi-agent systems. The developer writes agent programs in Jason, while the environment, the internal actions, and the custom architecture components are written in Java. The interpretation of the agent program and the underneath architecture effectively determines the agent's reasoning cycle. For our purpose, we can assume that the reasoning loop contains the following steps: the agent receives some percepts from the environment and updates its beliefs; it selects a pending event; it chooses a plan to handle the event; it executes the next action of an instantiated plan. The cycle is further discussed in chapter 4.

The agent program consists of initial beliefs, goals, and plans. Beliefs are expressed thorough Prolog-like facts and rules, and they are updated during the reasoning cycle according to the agent's perceptions. Goals represent the properties of the world that the agent wants to accomplish. There are two types of goals: achievement goals and test goals. Achievement goals lead to the execution of plans with the purpose of achieving that goal. Test goals, instead, are used to retrieve information from the agent's belief base. Plans are courses of actions that the agent expects they will lead to the corresponding goal. A plan is composed of a triggering event, a context, and a body. Changes in the agent's beliefs or goals create events. Triggering events tell the agent which events a plan can handle. The plan's context defines the conditions in which the plan is more likely to succeed. Those conditions will be checked against the belief base to determine if the plan is suitable for the current situation. The plan's body is composed of the actions that the agent will execute to hopefully achieve its goal. Actions can be environment actions, the addition of achievement or test goals, the addition of mental notes (i.e. beliefs), or internal actions that are a process executed within the agent. Plans can also have a label that associates meta-level information about it.

2.3 Reinforcement Learning

In this section, we provide a short review of RL taking Sutton and Barto's textbook [13] as the main reference – for a comprehensive overview, we suggest to consult that book.

RL is a machine learning method in which an agent learns how to fulfill a task *by interacting* with its environment. Any problem of learning *goal-directed* behavior can be reduced to three signals passing back and forth between an agent and its environment: one signal to represent the choices made by the agent (the *actions*), one signal to describe the basis on which the choices are made (the *states*), and one signal to define the agent's goal (the *rewards*). A state is defined as whatever information is available to the agent about its environment, some of what makes up a state could be based on the memory of past perceptions or even be entirely mental or subjective. More precisely, the state is the set of information useful to predict the future. 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 discounted 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 actions too might be either internal – changing the agent's mental state – or external – affecting the environment. For example, some actions might control what an agent chooses to think about, or where it focuses its attention.

We refer to each successive stages of interaction between the agent and the environment as *time steps*. At every time step, the agent performs an action and the environment provides a state and a reward. In the case of a BDI agent, a reasoning cycle can be pretty assumed as a step. In some applications, there is a natural notion of final time step, that is, when the agent-environment interaction breaks naturally into subsequences that are referred as *episodes*. Related to BDI, this is the case of agents pursuing *achievement goals*. In many cases, the agent-environment interaction does not break naturally into identifiable episodes but goes on indefinitely —these are called *continuing tasks*— i.e., *maintenance goals* in the BDI case.

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 come to our help. A NN can *approximate* the Q-function and possibly converge to the real function. This approach is known as Deep Q-Network (DQN) [21]. 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.

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'. The fundamental property of a state, known as the Markov property, is that it can be used to predict the future. A stochastic process (i.e., an environment) has the Markov property if the conditional probability distribution of the future state s_{t+1} depends only upon the current state s_t and action a_t, not on the sequence of events that preceded it. In many cases of interest, the agent has only *partial* information about the state of the world, so, the states signal is replaced by an *observations signal* that depend on the environment state but provide only partial information about it. In the BDI case, this is directly modelled by percepts and, therefore, by beliefs about the environment. From the observations, the agent recovers an approximate state i.e., a state that may not be Markov. Actually, we can partially drop the Markov property; however, this implies that long-term prediction performance can degrade dramatically if appropriate measures are not taken. 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). An approximate state will play the same role in RL algorithms as before, so we simply continue to call it a state.

Chapter 3

Related Works

Generally speaking, the integration of learning capabilities has been a main research topic in agents and MAS literature since their roots [22]. A first work providing preliminary results about integrating learning in BDI multi-agent systems is [23], proposing an extension of the BDI architecture to endow agents with learning skills, based on induction of logical decision trees. Learning, in that case, is about plan failures, that an agent should reconsider after its experience.

Other works in literature exploits RL to improve plan/action selection capability in BDI agent, making them more adaptive [24, 25, 26]. In [27] an extension of BDI is proposed so as to get a model of decision making by exploiting the ability to learn to recognise situations and select the appropriate plan based upon this.

In [28, 29], Jason is used to realise Temporal Difference (TD) and SARSA, two reinforcement learning methods, in order to face the RL problem with a more appropriate paradigm which has been remarkably effective. [30] proposes a hybrid framework that integrates BDI and TD-FALCON, a neural network based reinforcement learner. BDI-FALCON architecture extends the low-level RL agent with a desire and intention modules. In this way, the agent has explicit goals in the desire module instead of relying on an external signal, enhancing the agent with a higher level of self-awareness. The intention module and its plan library permit to reason about a course of actions instead of individual reactive responses. If there isn't a plan for a situation, the agent performs the steps suggested by the FALCON module and, if the sequence succeeds, a new plan is created with indications about the starting state, the goal state, and the actions sequence. When the agent uses a plan, it updates the confidence of the plan according to the outcome.

Also in [31] a hybrid approach BDI-FALCON is proposed. Here, the focus is on the abstrac-

tion level: BDI provides a high-level knowledge representation but lacks learning capabilities, meanwhile low-level RL agents are more adept at learning. The layered proposal wants to retain both advantages. An alternative vision is provided in [32], where a policy is learned and then is used to generate a BDI agent.

Compared to this literature, the Jason-RL framework aims to extend BDI agents with RL from a slightly different perspective, more focused on programming and the software 2.0 vision, that is: how learning – reinforcement learning, in this case – could be exploited by a developer in the process of developing an agent, so as to integrate plans explicitly written by the agent programmer with plans designed to be learned.

Chapter 4

Jason-RL Framework

4.1 The Basic Idea

The idea of Jason-RL is to extend the BDI agent development process with a learning stage in which we can specify plans in the plan library whose body is not meant to be explicitly programmed but learned, using a learning by experience technique. In so doing, the development of an agent accounts for: *(i)* explicitly programming plans as in the traditional way—we will refer to them as *hard plans*[1]; and *(ii)* implicitly defining plans by allowing the agent itself to learn them by experience—referenced as *soft* plans. Soft plans are meant to become part of the plan library like hard plans and can be selected and used at runtime – in terms of instantiated intentions – without any difference (but allowing for *continuous learning*, if wanted). At runtime, soft and hard plans are treated in a uniform way: intentions are created to carry on plan execution, hard plans could trigger the execution of soft plans and vice versa.

With soft plans we obtain a notion of *learning module*, that is, a soft plan is like a module that defines the boundaries of a learned behavior. A learning module can be *replaced* without affecting the rest of the agent. It can be *reused* in other agents and for other tasks as happens with standard hard plans. It can be *tested* at the unit level and in a natural way with the support of the BDI abstractions.

To support the learning stage, we would need to run the agent in a proper environment supporting this learning by experience, like the simulated environments typically used in RL scenarios. Besides, before deploying the agent, there could be some *assessment* of the soft plans, eventually using an assessment environment which could be different from the one used

[1] *hard* in this case stands for *hard-coded*.

for training. The assessment is actually very similar to a traditional validation stage, including tests that consider the soft plans and their integration with hard plans. If the agent overcomes the assessment, it can be deployed—otherwise, the process goes back to the learning stage, possibly changing also plans in the hard part.

Given this general idea, in the remainder of the section we first introduce the model integrating key concepts of RL into a BDI framework, and then we describe an extension of the BDI reasoning cycle supporting the learning process. We will use AgentSpeak(L) [9] and Jason [11] as concrete abstract and practical BDI-based languages—nevertheless, we believe that the core idea is largely independent of the specific BDI agent programming language or framework adopted.

To exemplify the approach, we will use the simple gridworld example (p. 80, [13]), in which an agent is located into a bi-dimensional grid environment, where it has to reach some destination cells without knowing in advance the best path for doing it.

4.1.1 A First Model

In devising the model, we aim at abstracting from the specific RL algorithm that can be used. To that purpose, we consider key common RL concepts – observations, actions, rewards, episodes – and how they are represented into a BDI setting (see Fig. 4.1, top). These concepts are used by a component – referred as *RL reasoner* – extending the BDI reasoner (interpreter) (see Fig. 4.1, bottom). The classic BDI reasoner handles the hard plans – i.e., with a body – and the RL reasoner handles the soft plans—whose behavior is learned.

Observations are modeled as a subset of the agent beliefs that will be used by the particular algorithm to construct the state, including those that are necessary to understand when a goal is achieved. Recall that the observations are a generalization of states and if we want to represent a Markov state we just need to include all its aspects as observations. In the gridworld case, for instance, the agent must reach a specific position moving towards four directions. In this case, the observations include the current position of the agent (`pos(X, Y)`) and a belief about having reached the target (`finish_line`)—if this belief is missing, it means that the agent has not achieved the target in the current state.

The *action set* can contain both primitive actions (a BDI agent action) and compound actions (a BDI agent plan) so that representing different levels of planning granularity. To have a common representation for both cases, we represent actions as plans, i.e. the set of actions selectable by

RL	Proposed BDI+ construct	Representation in BDI
Observations	Belief about Learning	Belief subset
Actions	Actions	Relevant Plans
Rewards	Motivational Rule	Belief Rule
Episode	Terminal Rule	Belief Rule
Policy	Learning Module	Plan
Behavior	Behavior	Intention

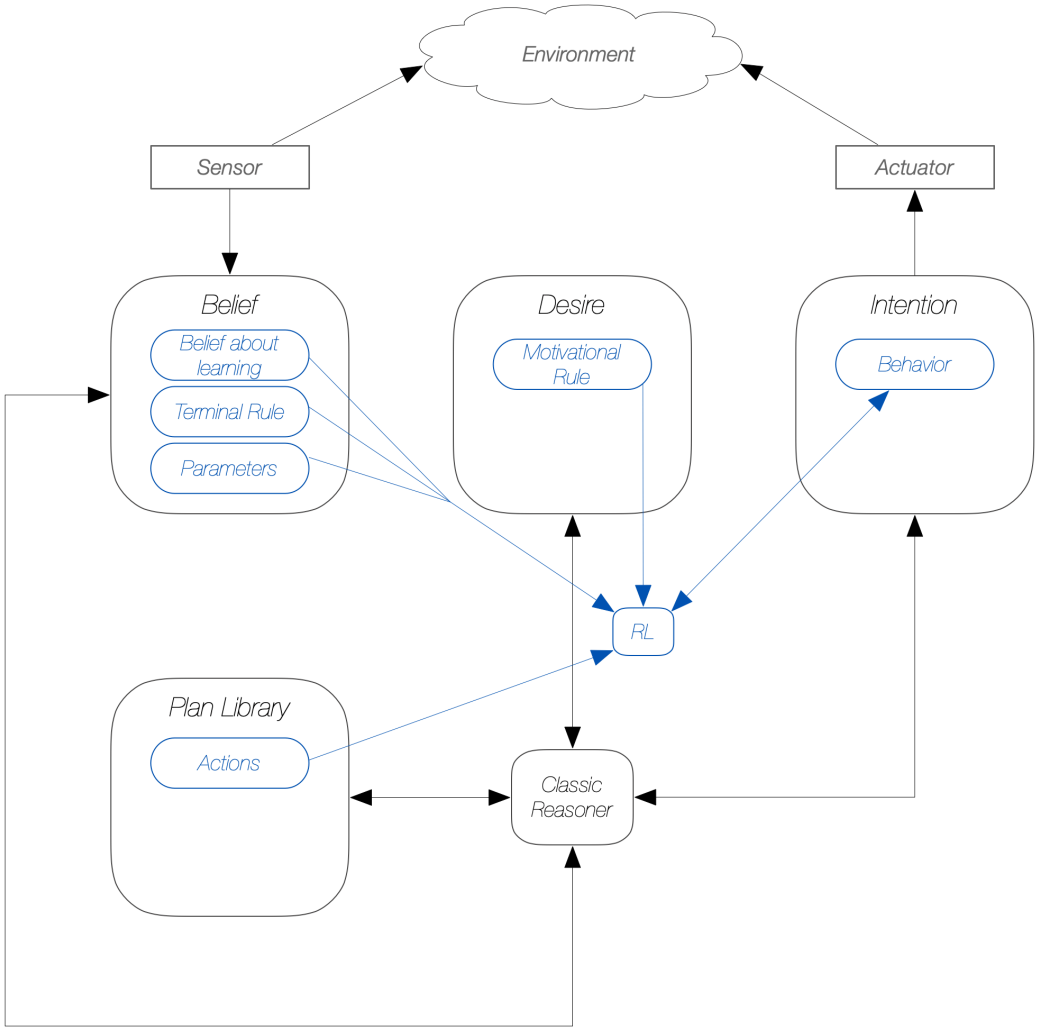

Figure 4.1: (Top) The mapping between RL concepts and their counterpart in the BDI model. (Bottom) A graphic representation of the BDI model with the addition of our constructs.

the RL reasoner is a subset of the plan library defined by the plans which are relevant for the goal and applicable in the current context. In Fig. 4.1, these plans are still referred as *Actions* in the plan library, and as *Behavior* (i.e., the learned policy) when instantiated at runtime, wrapped into an intention. In the gridworld example, the action set has one parametric plan to move the agent in one direction: `!move(D)`, where `D` could be `up`, `down`, `left`, `right`.

Rewards are represented by rules reflecting agent desires—we call them *Motivational Rules*. These rules make it possible to weight the current situation of an agent according to some goal to be achieved. We can see these rules as the generators of internal stimuli in the agent like a reward signal in neuroscience, which is a signal internal to the brain that influences decision making and learning. The signal might be triggered by the external environment, but it can also be triggered by things that do not exist outside the agent and which can be represented as beliefs as well. We move the reward, that in RL comes from the environment, into the agent. This is crucial to separate the agent desires from the environment so as to allow an agent to define its own goals and rewards about them.

In the AgentSpeak(L)/Jason model that will be detailed in next section, we can represent Motivational Rules as Prolog-like rules in the belief base:

```
reward(Goal, Reward) :- < some condition over the belief base >
```

In the gridworld example, the motivational rule will give a positive reward when the finish line is reached (while pursuing a `reach_end` goal) and a negative reward in other steps.

```
reward(reach_end, 10) :- finish_line.
reward(reach_end, -1) :- not finish_line.
```

It is worth remarking that Motivational Rules are meant to model also the reward signal that comes from the environment, as in the classic view. In this case the reward can be seen more like suggestion coming from a teacher (or a coach) which is assisting the agent during the learning/training process. The teacher conceptually knows both the goal of the agent and the structure of the (task) environment. Since the agent has the objective of learning how to achieve the goal, it is part of its motivation to follow the suggestions given by its teacher/coach.

Finally, we include a notion of *episode*. An episode is an event or a group of events occurring as part of a sequence. Like in the case of rewards, we can assume that agent designers may have their vision about how to define episodes, starting from a relevant ensemble of situations. This condition is well established by a rule that asserts in which belief state a coherent group of

events ends up in an episode i.e., it defines when the task ends, regardless of whether the agent accomplishes or fails the goal. We refer to this rule as a *Terminal Rule*:

```
episode_end(Goal) :- < some condition over the belief base >
```

In the gridworld example, the episode for achieving a `reach_end` goal ends when the finish line is reached:

```
episode_end(reach_end) :-  finish_line.
```

4.1.2 Extending the Reasoning Cycle

In our framework, a BDI agent is then equipped with general-purpose learning capabilities that are triggered as soon as a soft plan must be learned or adapted, for some goal. Fig. 1 shows the pseudo-code of a classic BDI agent reasoning cycle (as defined in [12, 33]) extended with such learning capabilities. This pseudo-code does not depend on a specific RL algorithm, which is encapsulated inside a separate module – referenced in the following as *learning strategy module* – whose methods are called by the reasoning cycle.

The beginning (lines 1–8) and the end (lines 22–36) of the reasoning cycle have not been modified. We briefly recall those parts, please refer to [12, 33] for more details. At the beginning of the reasoning cycle, the agent reviews its beliefs, desires, and intentions according to the new perception from the environment. At the end of the plan cycle, after each action execution, the agent acquires a new percept and uses the new knowledge to reconsider its desires, intentions, and the plan itself.

In the standard cycle, the function *plan* (line 8) generates a plan to achieve the intention I. In our extension, the plan could be either a soft plan or a hard plan. If it is a soft plan and we are in the learning stage, then a new learning episode is initialized (line 10), producing the initial action to be executed. The initialization of the episode depends on the specific learning strategy adopted.

The execution loop condition remains the same for the hard plans (continue until the plan is empty or it succeeds or fails). Instead, a soft plan (in the learning stage) continues until the learning episode finishes – i.e., a terminal state is met. A soft plan in the exploitation stage continues to produce new actions until the intention succeeds or becomes impossible to reach. Inside the loop, in the case of a soft plan in the learning stage, the currently selected action is

Algorithm 1 BDI practical reasoning extended with RL, in pseudo-code.

1: $B \leftarrow B'$ ▷ B' are the initial beliefs
2: $I \leftarrow I'$ ▷ I' are the initial intentions
3: **loop**
4: $\rho \leftarrow$ getNextPercept
5: $B \leftarrow$ brf(B, ρ) ▷ Belief Revision Function
6: $D \leftarrow$ options(B, I)
7: $I \leftarrow$ filter(B, D, I)
8: $\pi \leftarrow$ plan(B, I, Ac) ▷ Ac is the set of available actions
9: **if** softPlan$(\pi) \wedge$ learning(π, B) **then**
10: $A \leftarrow$ initEpisode(π, B, I)
11: **end if**
12: **while** $((\text{softPlan}(\pi) \wedge \text{learning}(\pi, B) \wedge \neg\text{episodeFinished}(\pi, B, I)) \vee (\text{softPlan}(\pi) \wedge \neg\text{learning}(\pi, B)) \vee (\neg\text{softPlan}(\pi) \wedge \neg\text{empty}(\pi))) \wedge \neg((\text{succeeded}(I, B) \vee \text{impossible}(I, B)))$ **do**
13: **if** isSoftPlan(π) **then**
14: **if** learning$(\pi, B) \wedge \neg$episodeFinished(π, B, I) **then**
15: execute(A)
16: $A \leftarrow$ doLearnStep(π, B, I, A)
17: **else**
18: $A \leftarrow$ chooseLearnedAction(π, B, I)
19: execute(A)
20: **end if**
21: **else**
22: $A \leftarrow$ head(π)
23: execute(A)
24: $\pi \leftarrow$ tail(π)
25: **end if**
26: $\rho \leftarrow$ getNextPercept
27: $B \leftarrow$ brf(B, ρ)
28: **if** reconsider(I, B) **then**
29: $D \leftarrow$ options(B, I)
30: $I \leftarrow$ filter(B, D, I)
31: **end if**
32: **if** $\neg$sound(π, I, B) **then**
33: $\pi \leftarrow$ plan(B, I, Ac)
34: **end if**
35: **end while**
36: **end loop**

executed (line 15). and then a learning step is performed (line 16) which also returns the next action to be executed. The learning step depends on the specific learning strategy adopted. If we are not in the learning stage but instead we are in the exploitation stage, the soft plan is used almost like a hard plan: the soft plan is used to decide which action to do (line 18) and the action is executed (line 19). In the case of a hard plan, the action is retrieved from the plan and executed as in the original cycle (lines 22–24).

We must stress that learning and exploitation are two distinct phases at different times. At first, the developer needs to design a teaching stage performed in a simulated environment in which the agent learns how to perform the soft plan. After that, the developer must validate the learning result to ensure it is suitable for the real scenario. From the agent cycle point of view, there is no difference between the two phases: it is the developer's job to assess the agent's capabilities before deploying it. The developer can also freely decide if the agent, after the deployment, will only exploit plans or also adapt to new experiences.

Fig. 2 shows the implementation of a learning strategy module based on SARSA [13] as concrete RL algorithm. SARSA is based on an action-value function Q which is built during the learning process. Besides the Q function the module keeps track of the current state, which is built from the observations (line 4 during the initialisation of an episode, line 13 in performing a learning step and line 21 when choosing an action during the exploitation stage). Observations are extracted from the belief base according to the goal to achieve (line 3 and line 12). The reward in a learning step is obtained from the motivational rule that quantifies the fulfilment of the goal in accordance with the current beliefs (line 11). Action selection follows the RL policy and the current value function Q (line 14 and line 22). The core part of the learning step is the value-action function update (line 15).

4.2 Jason Implementation

Jason-RL is implemented on top of Jason, exploiting its extensibility. Knowledge required by the RL part is uniformly represented by specific beliefs, referred as *beliefs about learning*. The framework abstracts from the specific RL algorithm but, depending on the characteristics of the problem, there will be different constraints on it. Critical factors are the knowledge about the environment and the state/action space dimensionality: if the state/action space dimensionality increases or more environment features are hidden (Markov property), then the constraints on

Algorithm 2 Learning strategy module based on SARSA

$Q(S,A), S, \alpha, \epsilon, \gamma$ ▷ Module state variables and params

procedure INITEPISODE(π, B, I) ▷ To initialise a learning episode
  $O \leftarrow \pi_{obs}(B, I)$
  $S \leftarrow$ STATE(O)
  return $Q_{policy}(S, \epsilon)$ ▷ action chosen using a policy derived from Q (e.g., ϵ-greedy)
end procedure

function EPISODEFINISHED(π, B, I) ▷ To check if an episode is finished
  return $S = \pi_{term}(B, I)$
end function

procedure DOLEARNSTEP(π, B, I, A) ▷ To do a learning step of an episode, using SARSA
  $R \leftarrow$ MR(π, B, I) ▷ Getting the reward using the motivational rules
  $O \leftarrow$ OBS(π, B, I) ▷ Getting the observations relevant for the learning task
  $S' \leftarrow$ STATE(O) ▷ Reconstruct the state given the observations
  $A' \leftarrow Q_{policy}(S', \epsilon)$ ▷ Choose A' from S' using a policy derived from Q
  $Q(S,A) \leftarrow Q(S,A) + \alpha[R + \gamma Q(S',A') - Q(S,A)]$ ▷ Q update, according to SARSA
  $S \leftarrow S'$ ▷ Updating the new current state S
  return A'
end procedure

function CHOOSELEARNTACTION(π, B, I)
  $O \leftarrow \pi_{obs}(B, I)$
  $S \leftarrow$ STATE(O)
  return $Q_{policy}(S)$ ▷ action chosen from S using Q action-value function
end function

the algorithm will be more stringent. To deal with this problem, we consider the possibility for the programmer to specify some domain knowledge so as to reduce state/action space and thus obtaining a more efficient/effective learning.

All the RL items are represented as beliefs, including rules, and plans. In this way, the agent can control everything related to the reinforcement learning process. For the BDI agent, RL is a black box and vice versa. We can see the black box as a block that we can change without affecting the agent and that can implement any RL algorithm.

4.2.1 RL Concepts Representation in Jason

All beliefs about learning include as the first parameter a ground term representing the goal for which we want a soft plan, i.e. whose plan is learned. This is useful to support multiple goals with soft plans at the same time. In the gridworld, for instance, we identify the goal with `reach_end`.

In order to reduce the state space, we need to declare which beliefs shall be considered relevant for a goal, so that they will be used as observations. We do this with the beliefs `rl_observe(G, O)`, where *G* is a ground term that refers to the goal and *O* is the list of the beliefs that will be considered for the goal. In gridworld example:

```
rl_observe(reach_end, [ pos ]).
```

As introduced in the previous section, Motivational Rules define the rewards for some goal given the current context:

```
rl_reward(G, R) :- ...
```

where *G* is the goal and R is a real number. The body of the rule represents the state for which this reward must be generated. In the gridworld example:

```
rl_reward(reach_end, 10) :- finish_line.
rl_reward(reach_end, -1) :- not finish_line.
```

At each execution, the RL reasoner gets the sum of all the rewards of the Motivational Rules for which the body is true on the basis of the agent beliefs.

Similarly, Terminal Rules are in the form of `rl_terminal(G) :- ...`, asserting when the end of an episode is reached. In the gridworld example:

```
rl_terminal(reach_end) :- finish_line.
```

The action set is represented as a set of (hard) plans, identified by an `@action` annotation: `@action[rl_goal(`g_1`, ..., `g_n`)]` where g_1, ..., g_n is the list of goals for which the plan/action can be used. In the gridworld example:

```
@action[rl_goal(reach_end),
     rl_param(direction(set(right, left, up, down)))]
+!move(Direction) <- move(Direction).
```

This is used to inform the RL reasoner that the `move` action, wrapped into the corresponding plan, is relevant for learning how to achieve the `reach_end` goal. The annotation allows for specifying also parameters that are used in the action/plan, specifying the range of the values: `@action[rl_goal(`g_1`, ..., `g_n`), rl_param(`p_1`, ..., `p_m`)]` where p_1, ..., p_m is the list of literals whose names match the names of the variables—these literals must contain a predicate that defines the type of the parameter and its range. To define an action space in which the action set is not the same in all states we can use the context of the plan—if the context is not satisfied for the current state, the plan will not be considered by the RL algorithm.

RL algorithm parameters can be specified as beliefs, enabling the complete control of the learning process by the programmer and the agent. In the gridworld example, some parameters are:

```
rl_parameter(alpha, 0.26).
rl_parameter(gamma, 0.9).
rl_parameter(policy, egreedy).
```

Finally, a couple of internal actions are provided to drive and inspect the learning process: `rl.execute(G)` and `rl.expected_return(G,R)`.

The internal action `rl.execute(G)` makes it possible to perform one run (episode) of the learning process – if we are in the learning stage – or execute the soft plan – in the exploitation stage. The belief about policy parameter `rl_parameter(policy,P)` can be used to specify if the agent is in learning stage or exploitation stage. Exploitation occurs when the policy `P` is set to `exploit`. When `rl.execute(G)` is executed, if the policy is set to `exploit`, then the soft plan is executed without learning.

The action `rl.execute(G)`, implemented in Java, wraps the core part of the RL algorithm. To that purpose, the Java bridge makes it possible to reuse existing RL libraries, when useful,

including libraries written in other languages such as Python ones. The action carries on and improves the soft plan under learning and its execution completes when the episode is completed (or until an action failure). The soft plan intention is carried out like any other intention—so the RL execution competes with the other intentions for the agent attention and further execution. Typically, a full learning process involves the repeated execution of learning episodes, in this case by executing multiple times the `rl.execute` action.

The internal action `rl.expected_return(G, R)` gets the estimate of future rewards `R` for the goal `G` on the basis of the current state and learned policy, i.e. the *expected return*. This could be used to understand the performance of the learned soft plan for some goal, given the current situation of the agent. For instance, if the expectation of the learned behavior in the current state is poor, we can fall back on another plan. As a result, we obtain a notion of *context* for soft plans.

A simple example of learning process, related to the gridworld example, using `rl.execute(G)` and `rl.expected_return(G, R)` follows:

```
+!reach_end: rl.expected_return(reach_end, R) & R >= 10
  <- rl.execute(reach_end).

+!reach_end: rl.expected_return(reach_end, R) & R < 10
  <- -+rl_parameter(policy, egreedy);
     !do_learning(reach_end,10).

+!do_learning(G,TargetRew) :
    rl.expected_return(reach_end, R) & R < TargetRew
  <- rl.execute(reach_end);
     !do_learning(G,TargetRew).

+!do_learning(G,TargetRew) :
    rl.expected_return(reach_end, R) & R >= TargetRew
  <- -+rl_parameter(policy, exploit).
```

In this case, if a new `reach_end` goal is requested and we have a good expected return, then we execute the soft plan (which is supposed to be in exploitation mode). Otherwise, we start a learning process, until the target reward is achieved.

4.2.2 Jason-RL Reasoning Cycle

The Jason reasoning cycle defines how the Jason interpreter runs an agent program, it can be seen as a refinement of the BDI decision loop [12]. There are ten main steps: in our framework, some of these steps are extended to include learning aspects. Figure 4.2 shows our extended architecture based on the original one; the red components are the extensions. The detailed description of the original cycle can be found in [12]. In the following we focus on our extensions.

In a learning agent, after the update of the belief base (2a), the maps that track the *Belief about Learning* are updated to reflect the new belief; we call this process *Observation Update Function* OUF (2b). In this way, when the observations are required, the agent doesn't need to iterate multiple times over the belief base. In step (7a), when the plan's context is bound to the expected reward, the value is asked to the RL black box and then verified against the threshold (7b). Finally, a new step (11) shall be added to the sequence when the next action of the intention selected in (9) is the RL execution. At this stage, the information that the RL process needs in order to continue shall be provided to it. The observations and the parameters are taken from the belief base, plus, the Motivational Rules and the Terminal Rules are checked against the belief base to retrieve the reward and the terminal status.

The RL reasoner needs also the set of relevant actions; this is formed by the action set defined in the plan library after the non-applicable actions are eliminated through the same check context function of (7). So, the agent provides these data to the RL black box and then obtain the next suggested action (12). This action is pushed on top of the intention queue and, if the state is not terminal, under this action is put a new call to the RL execution action (13). The next time this intention is further executed, the selected action will be performed, at the subsequent intention execution, another action will be requested and so on.

The full implementation of the framework, along with some documentation, is available as open-source project[2].

[2] https://github.com/MichaelBosello/jacamo-rl

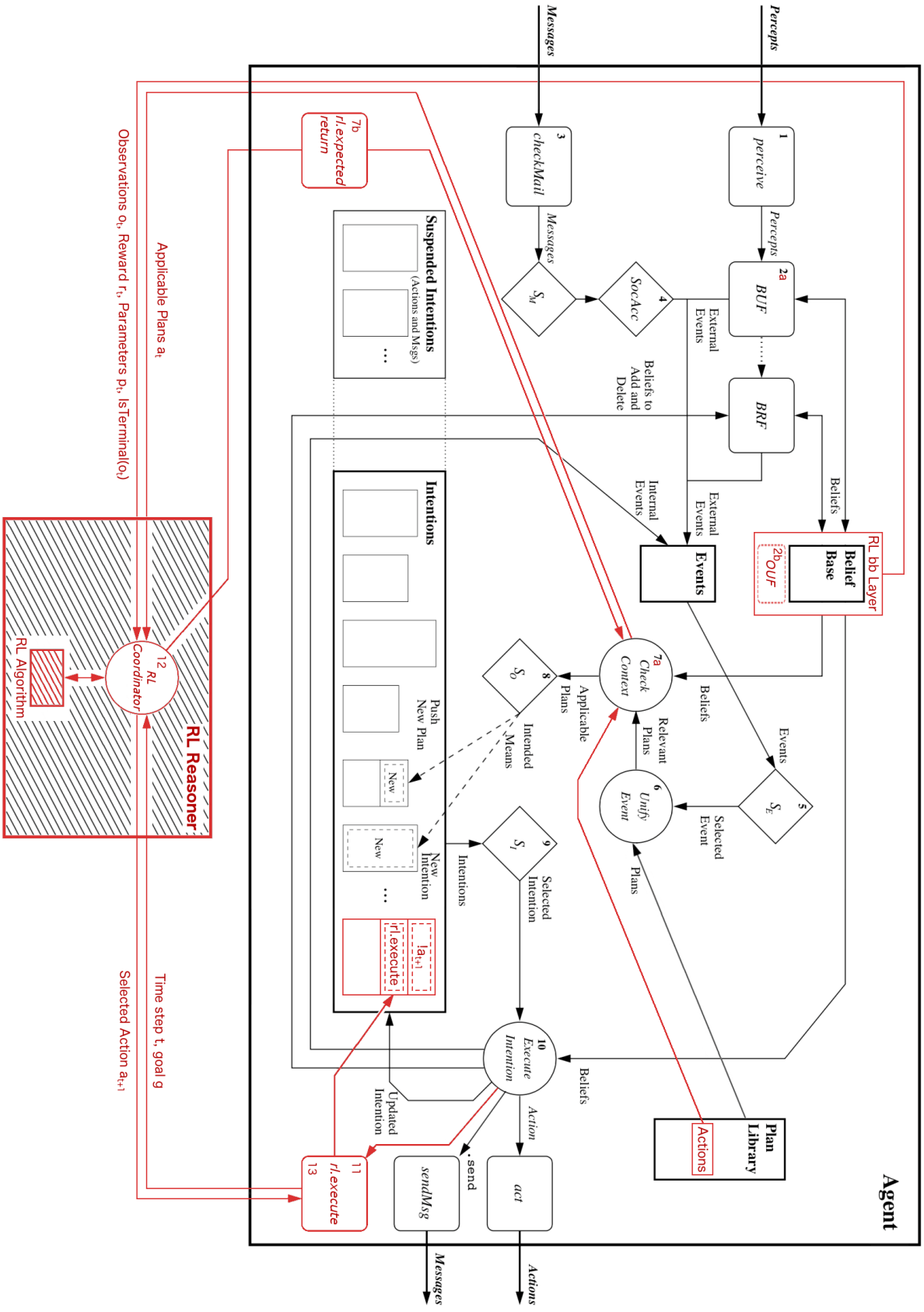

Figure 4.2: The Jason Reasoning Cycle extended with learning aspects.

Chapter 5

The Self-Driving Problem

Autonomous driving has been a societal goal since the automotive infancy. RCA engineers envisioned driver-less cars on US highways by 1975 [34]. While the prediction has not been fulfilled in time; the dream is still alive and was re-ignited by the DARPA Grand Challenge [35] 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 [36]:

- *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.

As the technology approaches level 5 the path to success becomes more difficult: in Level 5 an Automated Driving System (ADS) on the vehicle can do all the driving in all circumstances, thus the vehicles need to predict *ALL* possible driving scenarios, known, and unknown [37].

5.1 ML in Autonomous Driving

Advances in Deep Learning (DL), the availability of large data sets [38], 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 (CNNs) [39] fostered a revolution in pattern recognition [40] thus enabling advanced real-time applications. NN 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 [40] 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 and throttle) [41], 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 [42].

End-to-end autonomous driving agents have been 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 [43]. 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.

5.1.1 Supervised vs Reinforcement Learning

The majority of the methods currently in use for autonomous driving are based on *supervised learning* techniques that require *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. A radically different vision advocates for the use of reinforcement learning techniques. As reinforcement learning is a machine learning paradigm which *learns by trial-and-error*, it does not require explicit supervision from humans. Besides, supervised learning techniques

learn input-output associations (i.e. they are not able to learn the dynamics of an agent's environment), while RL is specifically formulated to handle the agent-environment interaction [13]. 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 [44] and optimizing problems that have no closed-form solutions.

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 method [45]. In particular, [36] 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, [46] 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 [47] 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 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* [48], 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 [49]. The literature offers methods to transfer the learning from a virtual world to the real one. In particular, [49] 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. In our previous work [14], we faced the simulation to real (sim2real) gap with a different perspective: instead of transferring learning from simulation to the real world, 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.

5.2 Modeling the Driving Problem

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) [48]. Moreover, the driving problem is open to subjective definitions as different behaviors in similar situations may be equally acceptable [38]. 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 [48]. A set-up of driving as a MDP is given in [48, 50], 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. This should be considered when designing the driving agent, and appropriate measures should be taken as explained in 2.3. 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.

5.3 LIDAR-based RL

LIDAR [51, 52] (an acronym for light detection and ranging) has become an established method for measuring distances by emitting focused beams of light and measuring the time it takes for the reflections to be detected by the sensor. The main principle – the return time is used to compute ranges to objects – is similar to the one used in radars, except that LIDAR is based on laser light instead of radio waves. In a basic LIDAR system, a rotating mirror reflects the laser range finder to gather distance measurements at specified angle intervals. The measured distances can be quite dense, resulting in and high-resolution maps. LIDAR sensors can be either 2D or 3D. 2D LIDARs produce a map of the azimuth at fixed height while 3D LIDARs produce 3D point clouds. In our case study, we employ a 2D LIDAR. Examples of LIDAR data sensed in different contexts are shown in chapter 9.

Thanks to the progress in the LIDAR technology and the use of deep learning, LIDAR has become widely used in autonomous driving [53], but not in the context of RL-based driving. 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.

Authors of [18] 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 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.

[54] 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 the 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 [55], a set of OpenAI Gym environments for RL-robotic research are proposed, complemented by experiments that used basic RL algorithms. A simulated TurtleBot [56] (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 [57], 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.

Chapter 6

Experiment Setup

6.1 F1tenth Platform

We used the F1/10 (or f1tenth) platform [17] 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 [14] 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 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 [56] which deliver slow-speeds and differential steering.

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° compared to 270° of the real one. Despite this,

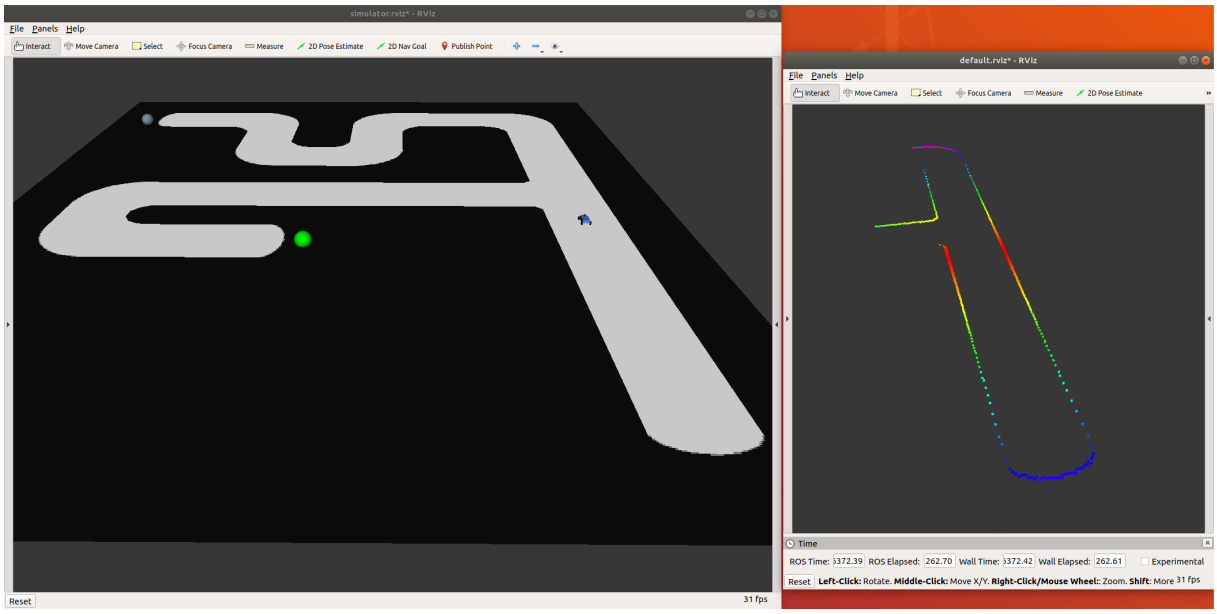

Figure 6.1: A frame of the simulated environment (left). The visual representation of the sensed lidar data (right).

simulated and real experiments are conducted with the same conditions as we reduced the field of view to 180° so as to reduce the state space. The simulated car, like the real one, is equipped with a VESC (Vedder Electronic Speed Controller) and a LIDAR. The VESC controls the speed of the motor and the direction of the servo. It also provides the odometry data (velocity vector and position) that are updated according to the motor speed and servo angle. The simulated LIDAR has 1080 scan rays with an angular resolution of $0.\overline{3}°$ (360° field of view). On the software side, sensors, actuators, and controllers are represented as nodes and coordinated by ROS (Robot Operating System) [58], a middleware strongly based on the publish/subscribe paradigm. Even though ROS provides several facilities, only hardware abstraction and communication have been used in our context. The simulated hardware implements sensors and actuators nodes with the same interfaces of the real car. The simulated environment is rendered using rviz, a visualization tool for ROS. The simulated car and a visual representation of sensed lidar data are shown in figure 6.1.

6.2 Agent Environment

The custom track has an intersection, some hairpin turns, and two possible targets (`End 1` and `End 2`). The map of the track is shown in figure 6.2. The agent starts at the bottom-right

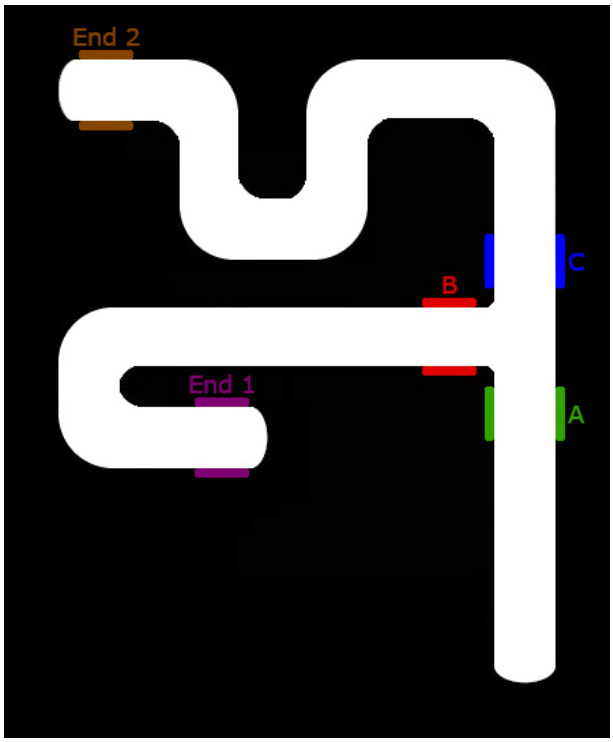

Figure 6.2: The map of the simulated track used in the experiment.

of the map. First, it has to reach the intersection (area `A`). At the intersection, the agent has to choose whether going forward (reaching `C`) or turning left (reaching `B`), according to the current target. After the intersection, the agent has to follow the street until the target.

As mentioned above, the software is coordinated through ROS. The RL environment implements the controller node that communicates with the – simulated – hardware nodes. The node uses rospy, the Python controller for ROS. In rospy, the callbacks associated to a subscriber are executed in a separate thread, while a publisher sends messages from the main thread. Publishers and subscribers are Python objects that a node can create. The RL environment is built as follows. Two subscribers receive respectively LIDAR and odometry data. The RL environment uses the latest stored data, that are updated asynchronously by the subscribers at every new measurement. A publisher, in a separate thread, sends the latest driving command chosen by the RL agent to the VESC. The publisher sends the command at fixed time intervals and continues to send it until the agent asynchronously updates the command. A safety callback 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 by the RL agent. The time-to-collision is calculated using the following

equation (the car's velocity is obtained from odometry data):

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

- r_i distance measured by ray i
- v car's velocity
- θ_i beam angle

At each interaction step, the RL environment executes the provided action and returns a state, a reward, and a boolean that indicates if the emergency braking has been activated. If the emergency braking has been activated, before returning the control to the agent, the car goes in reverse to ensure it has enough space to start again. The state is formed by the latest raw LIDAR distances, the target label (`End 1` or `End 2`), and the position label. The position label gives a sort of coarse-grained position useful to plan the high-level directions. Each strategic area has a label associated. The presence of the car in an area is verified thanks to the odometry data. The label could be `A`, `B`, `C`, `End 1`, `End 2`, or empty string if the car is not in a labeled area. The map in figure 6.2 shows the various areas. The agent has three driving 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. In addition to the driving actions, the agent can perform other two actions (used in the learning stage): it can ask for a new target randomly chosen and restart in the initial position; it can reset its position in the center of a labeled area (`A`, `B`, `C`). The reward function has been designed differently in the intersection experiment and in the real car experiment. Either way, rewards are clipped between $[-1, 1]$ because it could dramatically affect the learning efficiency [45]. In the real-world experiment, 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 intersection track, 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.

Since there is no Java client for ROS, the RL environment was implemented in Python as explained above. The Jason environment uses Java instead. We developed REST APIs that act as a bridge between the two environments. The Jason environment provides to the agent all the actions mentioned above. When the agent performs an environment operation, the Jason environment updates its percepts. The percepts given by the environment are: `lidar_data`, a

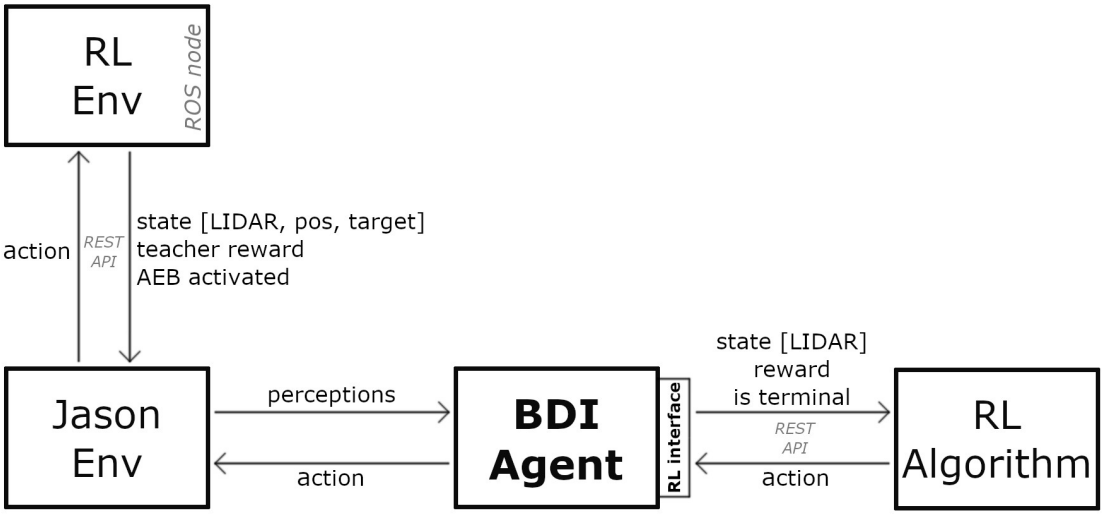

Figure 6.3: The high-level structure of agent and environment modules.

vector containing the lidar measures; `target(T)`, where `T` is the target label; `position(P)`, where `P` is the position label; `reward(R)`, which indicates the environment reward; `crash`, which inform whether the emergency braking has been activated; `new_position`, which inform whether the position label has changed. Regarding the RL algorithm, we implemented a RESTful service in Python that provides the capability of TensorFlow [59], the popular machine learning open source platform, to our framework. A Java class that implements the interface requested by our framework consumes the REST service, completing the bridge between our framework and Tensorflow. The agent/environment structure is shown in figure 6.3.

The Python-Java REST bridge adds significant overhead to the training cycle due to data conversion and communication. Compared to an end-to-end RL agent written entirely in Python (which shares the same environment/algorithm code), we measured an increase of almost 10ms per cycle in a machine equipped with an Intel core I7 (simulator) and approximately 15ms in the embedded system. This overhead could have a considerable impact in real-time scenarios, therefore, we are working to replace the REST bridge with the Java Native Interface (JNI) [60], which enables Java to interoperate with applications written in other languages.

Chapter 7

Agent Description

7.1 BDI Agent

The BDI agent has two hard-plans that defines the high-level directions, three soft-plans that manages navigation in different situations, and four hard-plans for each soft-plans that handle the soft-plans outcome. A portion of the agent code is stated in listing 3.

The directions plans react to the addition of a `target` belief (line 11 and 30) and they plan the moves to the indicated target. In the case of the target `END 1`, the agent has to follow the street until the area `A`, then turning left to reach `B`, and finally following the street to `END 1`. First, the agent add (or updates) two mental notes: one indicating the starting point, `START` (line 14); one indicating the current sub-target, `A` (line 15). Then, the achievement goal `follow_street` is added (line 16). When it concludes, the agent has successfully reached the sub-target `A`. The starting point and sub-target are updated (line 18 and 19); the agent starts from `A` and should reach `B`. The agent pursues the `turn_left` goal (line 20), then it updates the starting and sub-target points to `"B"` and `"END 1"` (line 22 and 23), and it adds the `follow_street` goal (line 24). At this point, the agent has successfully drove to the appropriate target. At the end of the plan, the car's position is reset to the initial point (line 27) and a new target is asked (line 28), restarting the trip. The plan for `END 2` works similarly, except for the achievement goal `go_forward` instead of `turn_left` and the mental notes adjusted accordingly.

The three soft-plans, `follow_street`, `turn_left`, and `go_forward` works in the same way, but they are trained in different conditions and with different goals, resulting in distinct behaviors (i.e, policies). `follow_street` learns how to reach the next position by following the path. `turn_left` learns how to turn at intersections i.e., how to go from `A` to `B`. `go_forward` learns

Algorithm 3 The Jason code of the agent able to drive in a track with an intersection.

```
rl_parameter(policy, egreedy).
rl_algorithm(follow_street, dqn).
rl_observe(follow_street, lidar_data(list(1080))).
rl_terminal(follow_street) :- crash.
rl_terminal(follow_street) :- new_position.
rl_reward(follow_street, R) :- reward(R).
rl_reward(follow_street, 50) :- new_position & position(P) & target_point(P).
rl_reward(follow_street, -50) :- new_position & position(P)
            & not target_point(P) & not starting_point(P) & not position("").

+target("END1") :  true <-
    .println("");
    .println("target:  END1");
    -+starting_point("START");
    -+target_point("A");
    !follow_street;
    .println("reached sub-target A");
    -+starting_point("A");
    -+target_point("B");
    !turn_left;
    .println("reached sub-target B");
    -+starting_point("B");
    -+target_point("END1");
    !follow_street;
    .println("reached target END1");
    .println("getting new target");
    reset_to_position("START");
    new_target.

+target("END2") :  true <-
    .println("");
    .println("target:  END2");
    -+starting_point("START");
    -+target_point("A");
    !follow_street;
    .println("reached sub-target A");
    -+starting_point("A");
    -+target_point("C");
    !go_forward;
    .println("reached sub-target C");
    -+starting_point("C");
    -+target_point("END2");
    !follow_street;
    .println("reached target END2");
    .println("getting new target");
    reset_to_position("START");
    new_target.
```

```
48: +!follow_street :  target_point(P) & position(P) <-
49:     move("stop");
50:     .println("reached ", P).
51:
52: +!follow_street :  starting_point(P) & position(P) <-
53:     rl.execute(follow_street);
54:     !follow_street.
55:
56: +!follow_street :  position("") <-
57:     rl.execute(follow_street);
58:     !follow_street.
59:
60: +!follow_street :  true <-
61:     move("stop");
62:     ?starting_point(P);
63:     .println("wrong direction taken");
64:     .println("resetting to point ", P);
65:     reset_to_position(P);
66:     !follow_street.
67:
68: @action1[rl_goal(follow_street),
69:           rl_param(direction(set(forward, right, left)))]
70: +!move(Direction) <- move(Direction).
```

how to pass intersections i.e., how to go from `A` to `C`. Only the code for the goal `follow_street` is given in the listing (line between 2 and 9, line between 48 and 66) as `turn_left` and `go_forward` repeats the exact same code with different names. The soft-plans use LIDAR data as observations (line 3) and a parametric plan, `move`, as set of actions (line 68, 69, 70). The parameter can be `forward`, `right`, and `left`. The soft-plans end when the car reaches a new position (whether it is the correct one or the wrong one) or when the automatic emergency braking activates, this is established by the terminal rules (line 4 and 5). The motivational rules represent the agent's will. They give a positive reward for reaching the current sub-target (line 7), or a negative reward for reaching the wrong target (line 8-9). The reward coming from the environment is also provided to the soft-plans (line 6). This reward signal may be intended as an external guideline given by a tutor.

A set of hard-plans ensures the achievement of the goals `follow_street`, `turn_left`, and `go_forward`. These plans use the soft-plans (which have the same name) to complete their task and they govern the various situations (and the learning cycle) by calling the soft-plans multiple times. If the car is in the starting point (line 52) or it is in an area without a label (line 56), the

soft-plan is executed (line 53 and 57), then the hard-plan is triggered again (line 54 and 58) to check the outcome of the soft-plan. If the agent reached the sub-target position (line 48), the hard-plan ends as the goal is achieved. In the remaining case, the agent drove to a wrong area (line 60), so, it resets to the starting point (line 65), and the achievement goal is added again (line 66). With this set of hard-plans, if the soft-plan ended because of an incident, the soft-plan will be executed again. If it ended because the car reached a new position, the agent will either return to the starting point or end the plan with the goal completed.

The source code, along with other resources, is available here[1]

7.2 RL algorithm

The software developed uses DQN [21] along with standard enhancements [21] that make it effective like experience replay, target network, and state history. On the RL side, 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 has two main parts: the NN and the training step.

There are two options to estimate a Q-function with a NN. (1) The network takes a state as input and gives a vector with the expected return for every possible action as output. (2) The network takes a state and an action as input and gives the expected return for that action as output. Here, we used the first approach. The NN used is a 1D CNN as our experiments, reported in appendix A, showed that 1D CNNs are very effective in processing LIDAR data compared to other NNs. As far as the authors' knowledge, this is the first documented use of 1D CNNs to process LIDAR data. The details of the 1D CNN used are in the appendix as well.

Training 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. At the subsequent step, the algorithm receives 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 at the end of the chapter 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. 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,

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

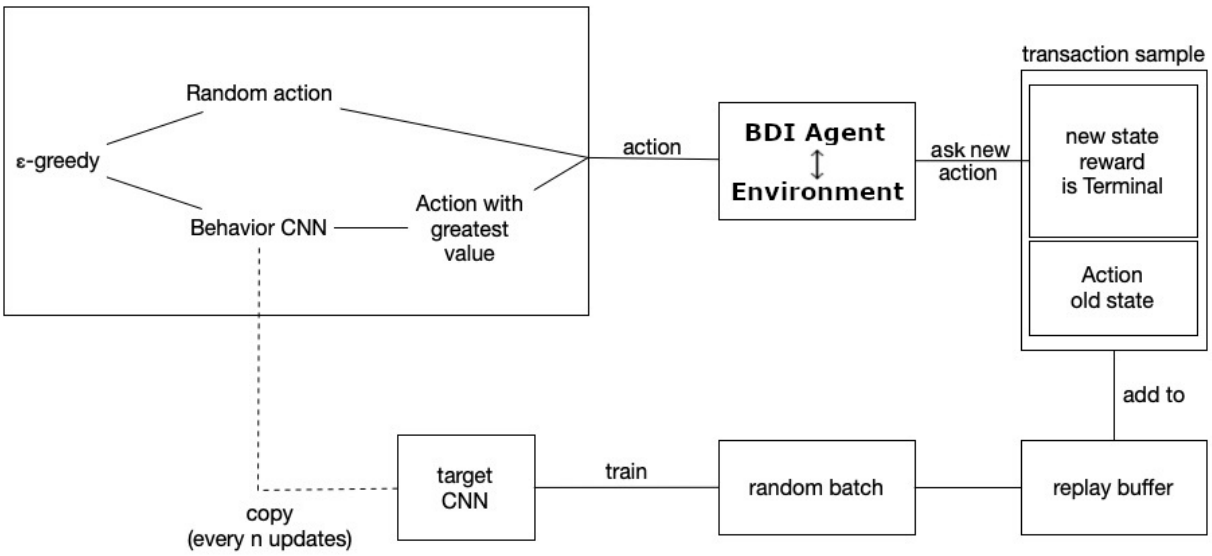

Figure 7.1: The overview of the training cycle from the RL perspective.

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. This technique, called *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 (SGD). Actually, the NN are two. One represents the target policy and is updated at every iteration. The other one 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, and the target network allows to reduce the instability. The training cycle from the RL perspective is outlined in figure 7.1

LIDAR data pre-processing

The simulated LIDAR has 1080 rays with an angular resolution of $0.\overline{3}°$. 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 360° 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 best results were obtained with 20 rays. As expected, 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 scaled between 0 and 1 (min-max normalization), with 1 corresponding approximately to the maximum value that can be sensed in the specific track. In the case of backpropagation, having input values in the range $[0, 1]$ improves learning [61].

Chapter 8

Results and Discussion

Thanks to the integration of soft and hard plans, the agent can reach its target by navigating without incidents and by choosing the proper path at the intersection. At the end of the experiment, the agent learned the three soft-plans necessary to pursue its goals. The improvement of the agent in the three tasks is made explicit in the reward plots. Results of training for each soft-plan are presented respectively in figures 8.1, 8.2, and 8.3. The plots show the average cumulative reward on 100 episodes in the learning phase. The trained models and Tensorboard logging of all experiments (intersection, real-car, NNs comparison), and a video showing the evolution of training are available in the project's repository.

Figure 8.1: Average cumulative reward on 100 episodes in the training phase of the `follow_street` soft-plan.

Figure 8.2: Average cumulative reward on 100 episodes in the evaluation phase of the `go_forward` soft-plan.

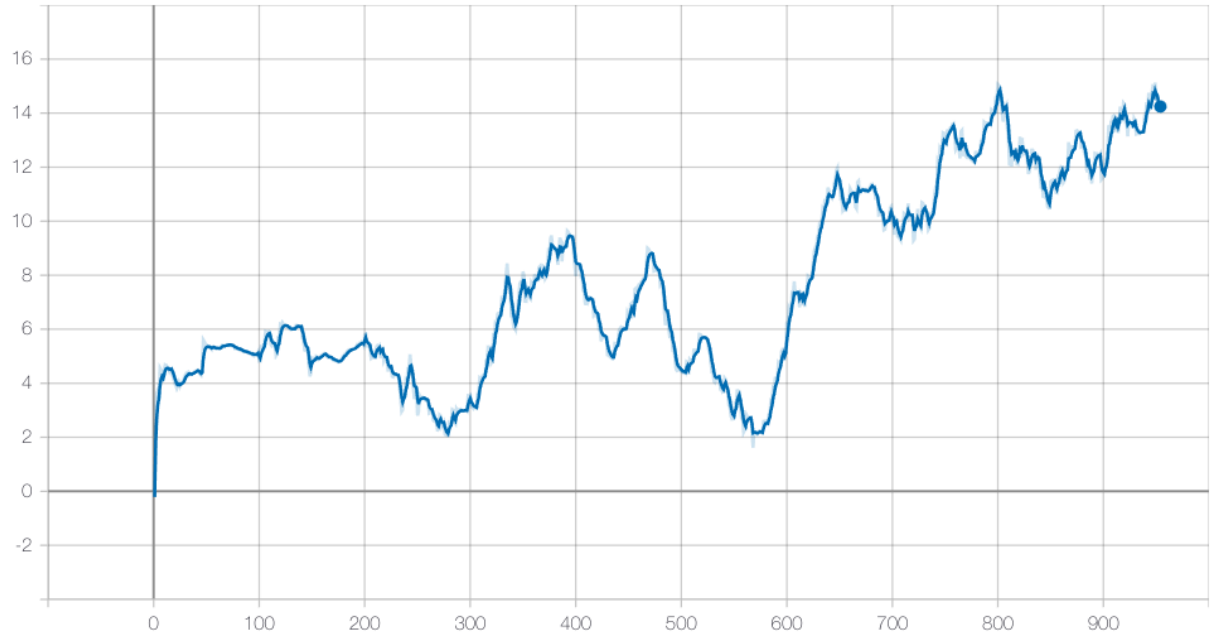

Figure 8.3: Average cumulative reward on 100 episodes in the evaluation phase of the `turn_left` soft-plan.

8.1 Discussion

Agentspeak has both a declarative nature (the reasoning) and an imperative one (plans, i.e sequences of actions). We worked towards automating the development of the imperative side while maintaining the declarative one. In the developed framework, the concept of soft plan allows the agent to learn plans while preserving the reasoning capabilities of a BDI agent: the framework act on the body of the plan, retaining the declarative attitude derived from event triggering and plans' context, as with traditional plans. As a result, it is not just about calling a policy learned through RL, but it concerns the inclusion of that kind of plan in the reasoning flow of the agent. This leads to the opportunity to manage and use learning in a declarative way with powerful abstractions, to support learned policies with the developer's domain-knowledge, and to have multiple RL contexts coordinated inside the same agent to achieve complex and time-extended tasks.

The presented case study provides several insights on the BDI-RL integration. As future work, we will quantify the advantages of the approach by comparing the BDI-RL agent with an end-to-end RL agent. Nevertheless, we can draw some qualitative considerations. Mixing hard and soft plans have been fruitful as it allowed us to face a problem that requires both time-extended planning and learned policies. Long-term plans are achieved thanks to the BDI reasoning, which uses hard-coded plans and learned policies as building blocks. The reasoning takes into account the various situations, so, hard-plans do not only manage planning but also recovering from soft-plan failures. For example, when the agent takes the wrong direction, it is brought back to the previous point (in a realistic scenario, it will re-plan the path – the concept is still valid). As RL struggles in temporally-extended planning [15, 16], we foresee that a plain RL agent will need more training time (episodes) to learn how to reach the two targets, or even it may not be able to learn such behavior at all. This is even more true if we consider broader scenarios with complex intersections and roundabout where, in the case of the BDI-RL agent, we can simply add a few soft-plans. Considering multiple RL contexts, while the BDI reasoning provides an effective way to achieve planning and coordination of different learned policies, the BDI abstractions allow to split – and enhance – the RL signals among the different contexts. **States**: Observations are appropriately represented and selected by percepts and mental notes. **Actions**: Even though only simple actions are used in the case study, using plans as action representation enables the options framework [13], a RL setting in which time-extended actions and sub-routines are considered. **Rewards**: In the case examined, the agent has both self-

rewards and environmental rewards. In this way, the agent increases its self-awareness, with a reward signal (and thus goals) dependent on the desires that it is currently pursuing. Having both internal and external rewards allows to shape the rewards according to the specific agent goals, but also to take into account the suggestion from the environment (or from artifacts [62] built for the purpose) that is viewed as a teacher. In some cases, besides splitting the problem in different RL contexts, it could be useful to use different RL methods – or different NNs – for different tasks. Since every soft-plan has its own RL context, multiple RL algorithms can coexist and be used by the same agent allowing to use the most appropriate algorithm for each sub-task. As the framework aims at modeling the three fundamental RL signals without any assumption on the RL algorithm, it is sufficient to plug the chosen RL component, without changing the whole interpreter architecture. In conclusion, the BDI reasoning as temporally-extended planning, the availability of multiple RL contexts for different sub-tasks, and the enhanced abstractions could be an effective set of tools to tackle complex – multi-agent system – problems.

Chapter 9

From Simulation to Reality

As a first step towards the use of the system in the real-world, we built a physical f1tenth and trained it to drive in a simple racetrack (as we were limited by the room dimension). The track is shown at the top left of fig. 9.2. The test aims to verify whether DQN could be used to train a robust controller in a real environment with noisy LIDAR data — a problem considered *open* [18]. In the future, we plan to bring the full system in the real f1tenth and perform also the intersection experiment in the real-world.

As explained in section 6.1, the f1tenth hardware is similar to full-scale solutions to conduct real-world experiments. 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 brush-less 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 mainboard that controls the car is a Jetson TX2 [63], 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 [64] 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. 9.1 shows our produced f1tenth. The 2D LIDAR used is a Hokuyo UST-10LX [65]. 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

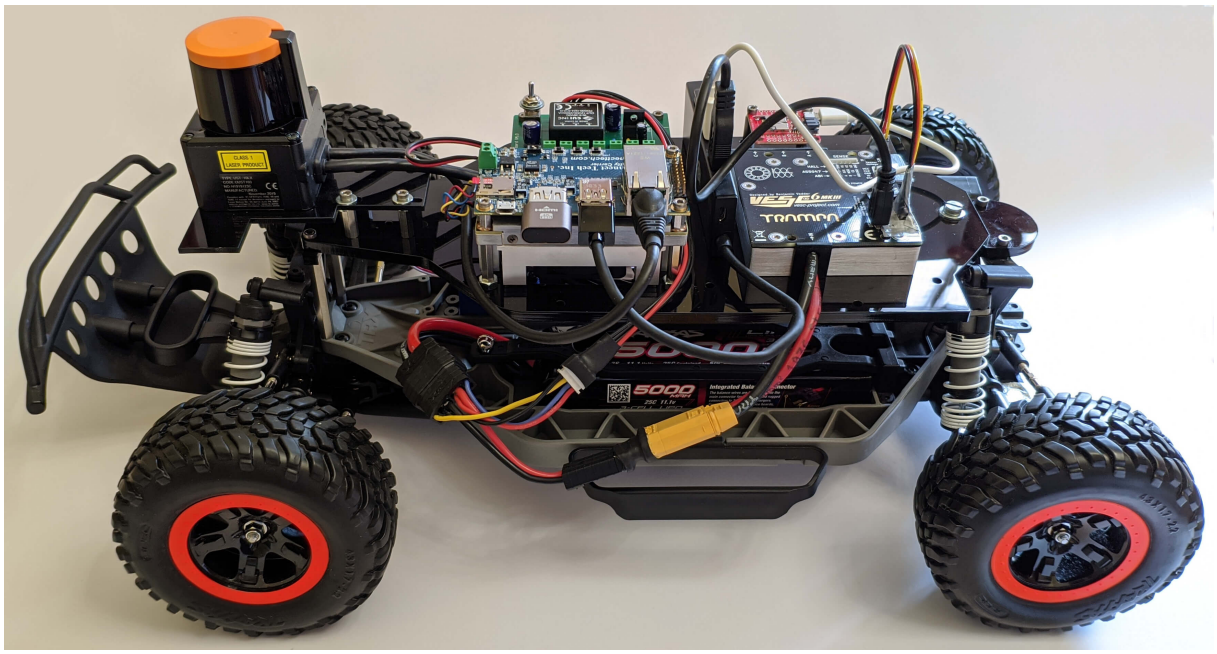

Figure 9.1: The f1tenth race car used in the experiments.

returned if a measurement error occurs like an object does not exist or object has low reflectivity. Fig. 9.2 displays visual representations of data produced by LIDAR scans, depicting the sensor precision in different conditions.

Moving to the real car required four adjustments: addressing LIDAR faults [18], facing continuous time [66], providing a reset mechanism [67], and specific hyper-parameter tuning. As explained in section 5.3, 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. The cycle speed is a problem also for the reward signal. The faster

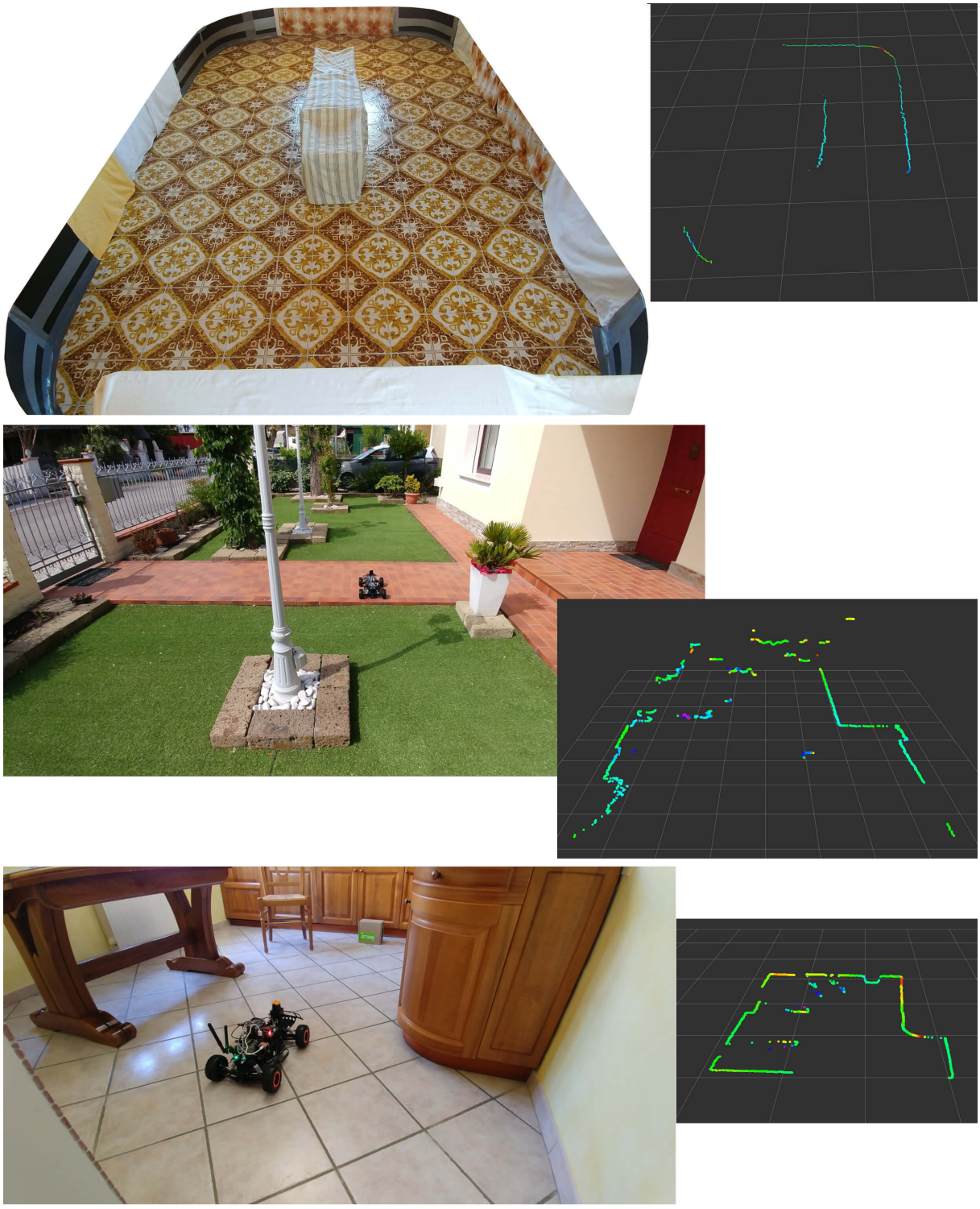

Figure 9.2: The circuit used in the experiment (top). An example of LIDAR data in an outdoor scenario with far away obstacles (middle). An example of LIDAR data in the presence of thin obstacles like a chair's legs (bottom).

the cycle, the more the reward received; vice-versa, the slower the cycle, the fewer the rewards received. In the real car, the reward had to be scaled according to the cycle timing. Regarding the reset mechanism, when an episode ends because of emergency braking activation, the car goes backward for a fixed time before starting a new episode. This is feasible in both simulation and real-world. When we will perform the intersection experiment in the real world, a new reset mechanism will have to be conceived instead.

We used the 1D CNN also in the real car as it is the best option (see appendix A). The plots of the average cumulative reward for training and evaluation are shown in fig. 9.3 and fig. 9.4. 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.

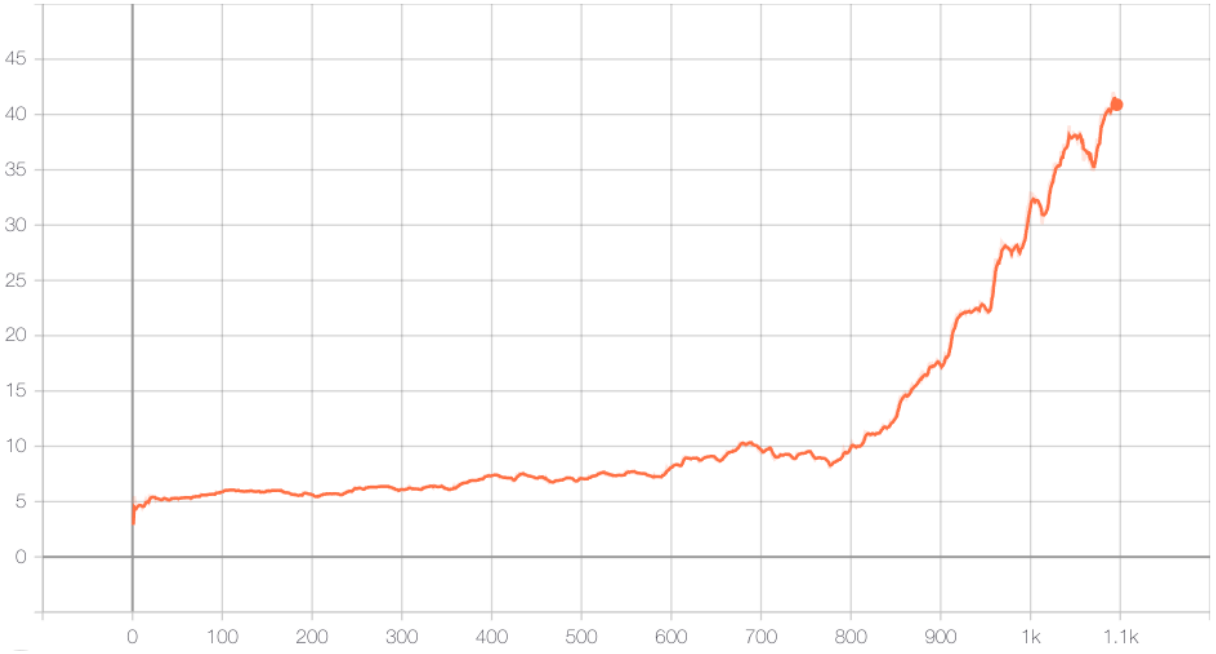

Figure 9.3: Average cumulative reward on 100 episodes in the training phase of the real car experiment.

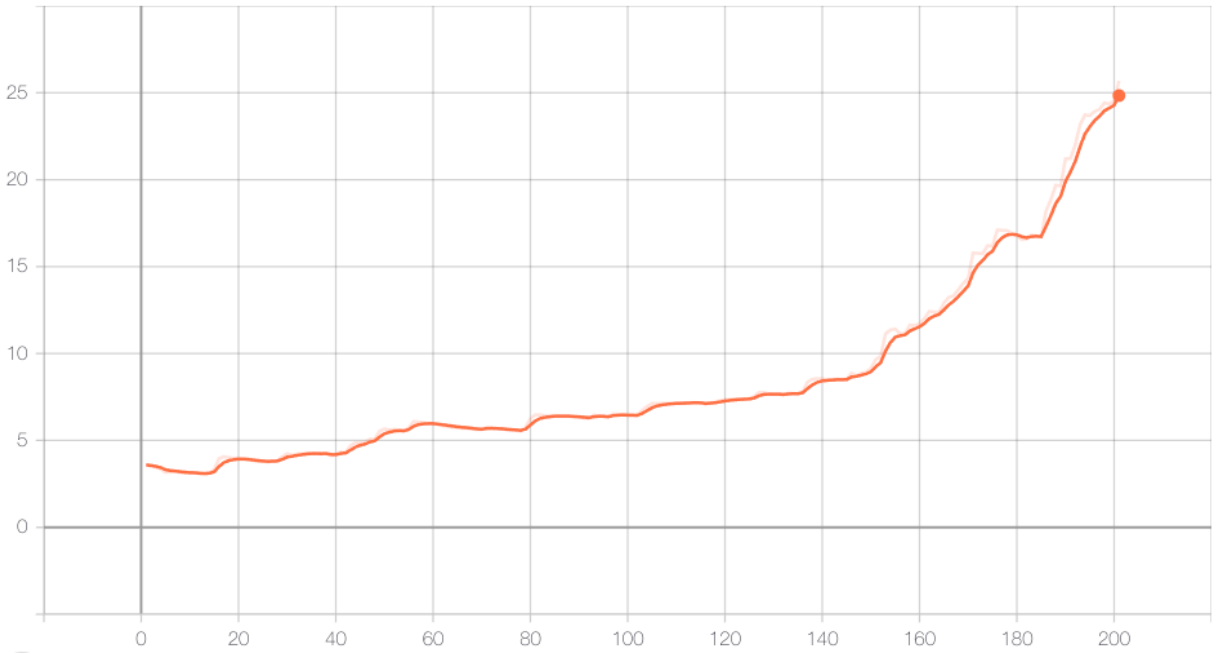

Figure 9.4: Average cumulative reward on 100 episodes in the evaluation phase of the real car experiment.

Chapter 10

Conclusions and Future Works

In this work, we have shown the effectiveness of the BDI-RL blending in an interesting case study and discussed its implications and advantages. The agent was able to follow hard-coded directions while the challenging task of navigation has been learned autonomously. This sheds light on the usefulness of machine learning tools to support (agent) developers instead of replacing them with end-to-end solutions, possibly taking the best of both worlds.

Considering the case study, 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 on both the BDI-RL integration and the case study. In the first case, we intend to further explore and develop this perspective in its many facets. In the latter case, we seek to understand how close to real urban scenarios it is possible to push this approach without sacrificing safety. Specifically, many interesting aspects – from our point of view – are worth to be investigated in future work:

Sim2real. Performing the intersection experiment with the real f1tenth using Bluetooth beacons to retrieve the car position.

BDI-RL vs plain RL *Quantifying* the expected benefits of BDI-RL integration compared to plain RL by reproducing the intersection experiment with an agent that uses only RL so as to verify whether it could actually learn how to handle the intersection and, in that case, checking the differences in terms of learning speed and quality of the final behavior.

Options. In this case study, hard-plans use soft-plans and soft-plans, in turn, uses only simple actions. In some scenarios, it could be useful to have soft-plans that use hard-plans like compound actions. Extended actions are called *options* in the RL setting [13]. Hard plans in this case can be framed as the predefined practical knowledge that the agent could exploit, without the need of learning everything necessarily from scratch. We believe that this may have a positive impact on both the time required to learn and of the quality of the learned plans itself. To effectively explore this point, we need to identify and implement a case study where options could be useful.

Hierarchical RL and Reward Shaping. Further extending the set of RL concepts covered, eventually doing a rigorous analysis of the computational complexity and properties of the computations performed by the extended reasoning cycle. Among the large spectrum of RL-based approaches, two are particularly interesting with respect to the objective of our research line. The first is about Hierarchical Reinforcement Learning, extending the reinforcement learning paradigm by allowing the learning agent to aggregate actions into reusable subroutines or skills [68]. In the BDI case, reusable subroutines or skills are modeled as plans triggered by subgoals. The second one is about reward shaping in reinforcement learning [69]. There, "education" is realized through the creation of a proper learning environment and, in particular, through demonstration.

Education Process. Exploring further the *development/education process life-cycle*, analysing how the different stages – development/training, validation/assessment, deployment/monitoring – are related.

New Generation of Tools and IDE. Designing and developing proper tools to be embedded in existing IDEs – or extending them – to support this process. Including simulators, which become an essential part of the picture.

Software Engineering. Exploring how software engineering aspects such as modularity, extensibility, reusability, composability can be framed when dealing with soft plans, aside from hard plans. Can we introduce a kind of *incremental learning* to extend existing soft plans?

Artifact-Based Environments. Exploring how environment first-class abstractions such as artifacts [62] could be useful to better structure, modularise and make the way in which

actions – and observations as well – are considered more dynamic.

Beyond the Single Agent Perspective. What does it mean for an education process for a multi-agent system? What does it mean an education process for an agent *organisation*?

Methodologies. What is the impact on Agent-Oriented Software Engineering (AOSE) [70] methodologies? Can we exploit existing AOSE methodology to effectively support this process or do we need to extend them?

Urban environment. Moving to an urban scenario (e.g. by using a realistic simulator like Carla [71]) and make the agent follows general driving directions to reach a target (i.e. directions given by a maps service).

Cooperative learning. In a *multi-agent system*, exploring the possibility of *cooperative learning* through the exchange of experience between agent — with efficient communication[72].

Meta-learning. Experimenting with meta-learning [73] approaches as they could improve the learning stage (and verify if they could be useful in the context of self-driving).

Continuous action space. Implementing a deep RL technique for continuous action space like Asynchronous Advantage Actor-Critic (A3C) [74].

Safe RL and RL with bounded risk. Exploring *safe Reinforcement Learning* models [75] and *Reinforcement Learning with bounded risk* [76] (as they could be useful in the context of self-driving) and how they fit in the framework's model.

1/10 to full scale. Attempt to transfer the experience acquired by 1/10 vehicles to actual vehicles in a controlled environment.

Appendix A

NNs Comparison for LIDAR Data Processing

We have conducted an experiment to compare the effectiveness of three NNs in the driving task. The NNs tested has been: 1D CNN, a fully connected network, and a 2D CNN. The comparison has been conducted in simulation with a complex track which is displayed in figure A.1. The track has no intersections as we want to test only the navigation capabilities of the NNs.

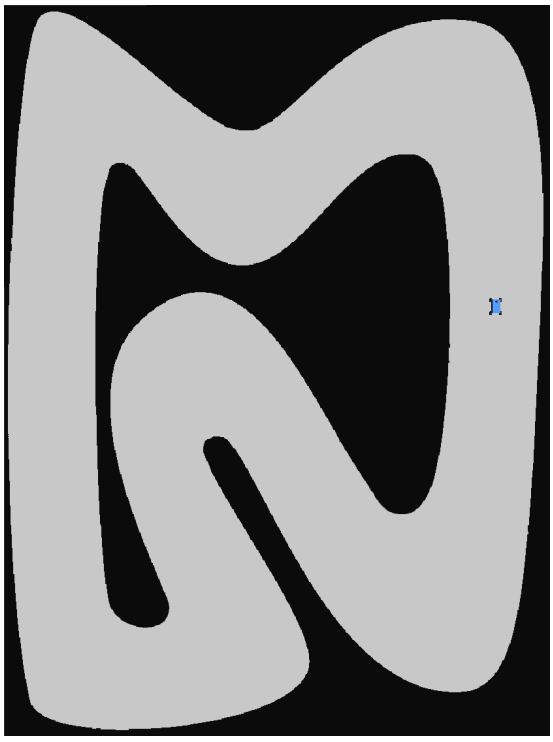

Figure A.1: The simulated racetrack used in the comparison experiment.

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 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. For all the networks, 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 [77] had a greatly positive effect on learning.

The plot in fig. A.2 shows the average cumulative reward on 100 episodes of the three NNs during training stages. Fig. A.3 shows the cumulative reward during the evaluation phases, that 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.

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, audio, and text [78]. 1D CNNs have proven to be 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.

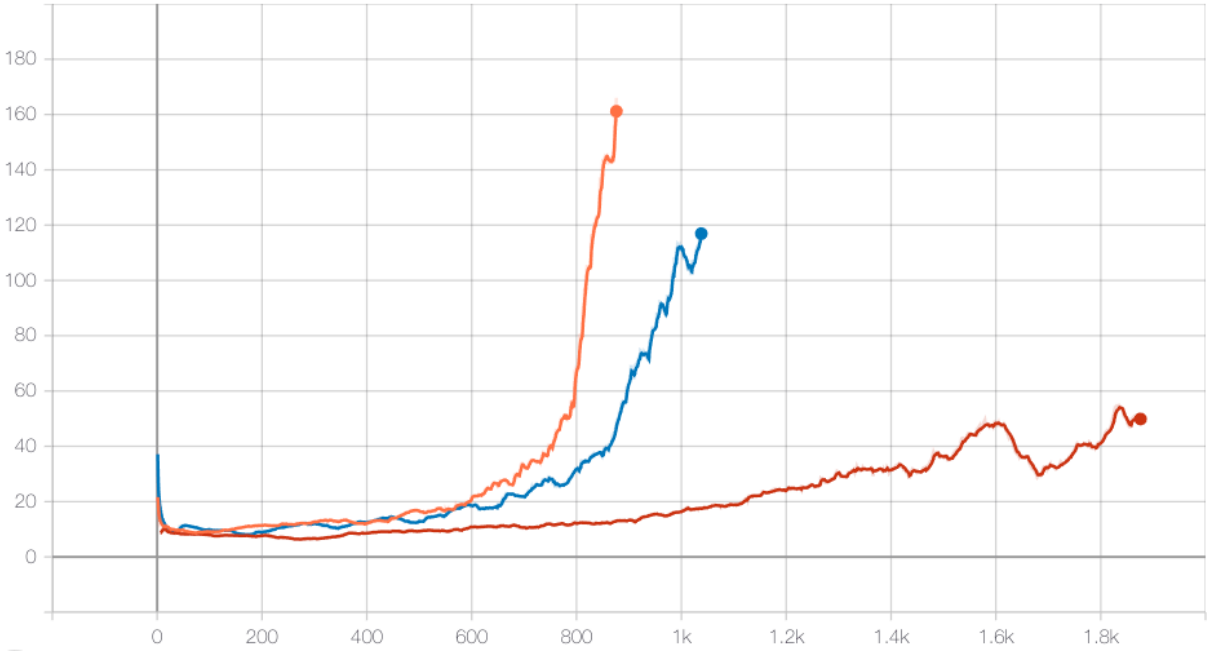

Figure A.2: Comparison of average cumulative reward on 100 episodes in the training phase of the NNs effectiveness experiment. 1D CNN: orange, dense net: blue, 2D CNN: red.

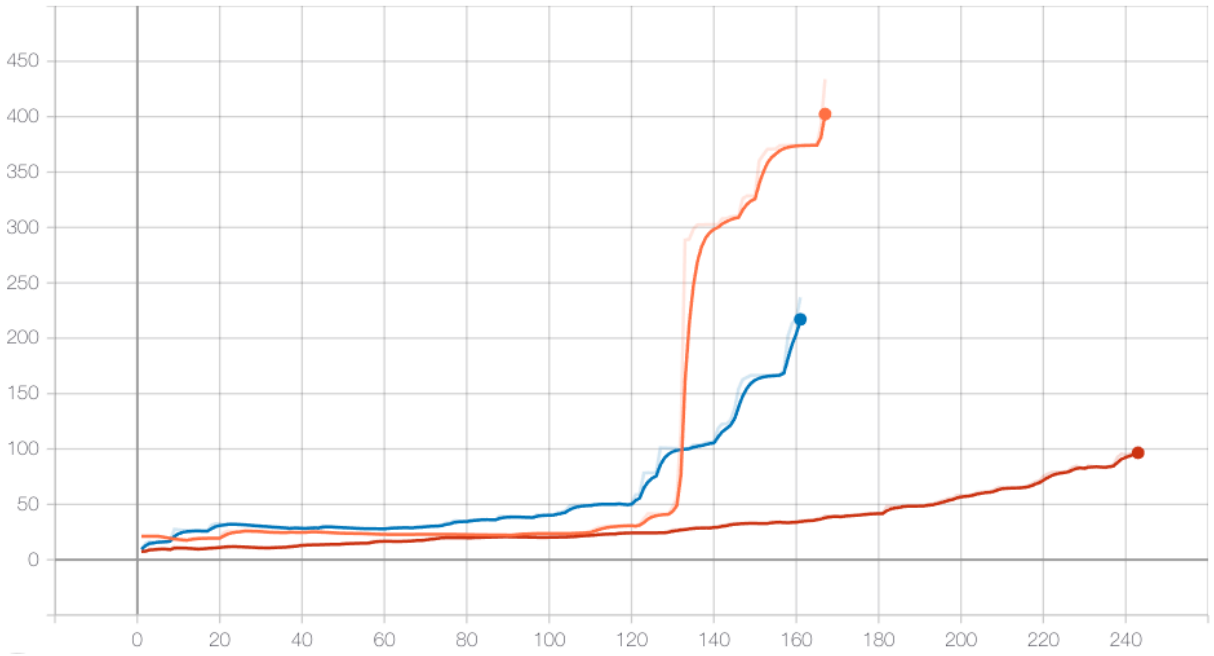

Figure A.3: Comparison of average cumulative reward on 100 episodes in the evaluation phase of the NNs effectiveness experiment. 1D CNN: orange, dense net: blue, 2D CNN: red.

Bibliography

[1] E. B. Andrew McAfee, *The Second Machine Age: Work, Progress, and Prosperity in a Time of Brilliant Technologies*. W. W. Norton & Company, 2014.

[2] M. Ford, *Architects of Intelligence: The truth about AI from the people building it*. Packt Publishing, 2018.

[3] S. Gerrish, *How Smart Machines Think*. MIT Press, 2018.

[4] J. E. Kelly, "Computing, cognition and the future of knowing," 2015, IBM Research and Solutions, white paper.

[5] D. L. Parnas, "The real risks of artificial intelligence," *Commun. ACM*, vol. 60, no. 10, pp. 27–31, Sep. 2017.

[6] E. Meijer, "Behind every great deep learning framework is an even greater programming languages concept," 2018, Invited Talk at the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE).

[7] J. Tanz, "The end of code," *Wired*, 2016.

[8] Y. Shoham, "Agent-oriented programming," *Artif. Intell.*, vol. 60, no. 1, pp. 51–92, Mar. 1993.

[9] A. S. Rao and M. P. Georgeff, "BDI agents: From theory to practice," in *Proc. of the First Int. Conf. on Multi-Agent Systems (ICMAS-95*, 1995, pp. 312–319.

[10] M. Bosello and A. Ricci, "From programming agents to educating agents – a jason-based framework for integrating learning in the development of cognitive agents," in *Engineering Multi-Agent Systems*, L. A. Dennis, R. H. Bordini, and Y. Lespérance, Eds. Cham: Springer International Publishing, 2020, pp. 175–194.

[11] R. H. Bordini, J. F. Hübner, and R. Vieira, "Jason and the golden fleece of agent-oriented programming," in *Multi-Agent Programming: Languages, Platforms and Applications*, R. H. Bordini and et al., Eds. Boston, MA: Springer US, 2005, pp. 3–37.

[12] R. H. Bordini, J. F. Hübner, and M. Wooldridge, *Programming Multi-Agent Systems in AgentSpeak Using Jason (Wiley Series in Agent Technology)*. USA: John Wiley & Sons, Inc., 2007.

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

[14] 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.

[15] B. M. Lake, T. D. Ullman, J. B. Tenenbaum, and S. J. Gershman, "Building machines that learn and think like people," *Behavioral and Brain Sciences*, vol. 40, p. e253, 2017.

[16] D. Hassabis, D. Kumaran, C. Summerfield, and M. Botvinick, "Neuroscience-inspired artificial intelligence," *Neuron*, vol. 95, pp. 245–258, 07 2017.

[17] 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.

[18] 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

[19] N. R. Jennings, "An agent-based approach for building complex software systems," *Commun. ACM*, vol. 44, no. 4, pp. 35–41, Apr. 2001.

[20] M. E. Bratman, "Intention, plans and practical reason," *Bibliovault OAI Repository, the University of Chicago Press*, vol. 100, 01 1987.

[21] 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

[22] G. Weiß, "Adaptation and learning in multi-agent systems: Some remarks and a bibliography," in *Adaption and Learning in Multi-Agent Systems*, G. Weiß and S. Sen, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 1–21.

[23] A. Guerra-Hernández, A. El Fallah-Seghrouchni, and H. Soldano, "Learning in BDI multi-agent systems," in *Proc. of the 4th Int. Conf. on Computational Logic in Multi-Agent Systems (CLIMA VI)*, ser. CLIMA IV'04. Berlin, Heidelberg: Springer-Verlag, 2004, pp. 218–233.

[24] D. Singh, S. Sardina, L. Padgham, and G. James, "Integrating learning into a BDI agent for environments with changing dynamics," in *Proc. of the Twenty-Second Int. Joint Conf. on Artificial Intelligence (IJCAI '11)*, ser. IJCAI'11. AAAI Press, 2011, pp. 2525–2530.

[25] S. Airiau, L. Padgham, S. Sardina, and S. Sen, "Enhancing the adaptation of BDI agents using learning techniques," *Int. J. Agent Technol. Syst.*, vol. 1, no. 2, pp. 1–18, Apr. 2009.

[26] D. Singh and K. V. Hindriks, "Learning to improve agent behaviours in goal," in *Programming Multi-Agent Systems*, M. Dastani, J. F. Hübner, and B. Logan, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 158–173.

[27] E. Norling, "Folk psychology for human modelling: Extending the bdi paradigm," in *Proc. of the Third Int. Joint Conf. on Autonomous Agents and Multiagent Systems (AAMAS '04)*, ser. AAMAS '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 202–209.

[28] A. Badica, C. Badica, M. Ivanovic, and D. Mitrovic, "An approach of temporal difference learning using agent-oriented programming," in *20th Int. Conf. on Control Systems and Computer Science*, May 2015, pp. 735–742.

[29] C. Badica, A. Becheru, and S. Felton, "Integration of jason reinforcement learning agents into an interactive application," in *19th Int. Symp. on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*, Sep. 2017, pp. 361–368.

[30] A.-H. Tan, Y.-S. Ong, and A. Tapanuj, "A hybrid agent architecture integrating desire, intention and reinforcement learning," *Expert Syst. Appl.*, vol. 38, no. 7, pp. 8477–8487, Jul. 2011.

[31] S. Karim, L. Sonenberg, and A.-H. Tan, "A hybrid architecture combining reactive plan execution and reactive learning," in *PRICAI 2006: Trends in Artificial Intelligence*, Q. Yang and G. Webb, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 200–211.

[32] J. L. S. Feliú, *Use of Reinforcement Learning (RL) for plan generation in Belief-Desire-Intention (BDI) agent systems*. University of Rhode Island, 2013.

[33] M. Wooldridge, *Introduction to Multi-Agent Systems*. Wiley, 2009.

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

[35] (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

[36] 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.

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

[38] 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.

[39] 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

[40] 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

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

[42] 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.

[43] 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

[44] 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.

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

[46] 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.

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

[48] B. R. Kiran, I. Sobh, V. Talpaert, P. Mannion, A. A. A. Sallab, S. Yogamani, and P. Pérez, "Deep reinforcement learning for autonomous driving: A survey," 2020.

[49] 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

[50] 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.

[51] J. Shan and C. Toth, *Topographic Laser Ranging and Scanning: Principles and Processing*. CRC Press, 2008.

[52] N. C. S. Center, "Lidar 101: An introduction to lidar technology, data, and applications." 2012.

[53] Y. Li, L. Ma, Z. Zhong, F. Liu, M. A. Chapman, D. Cao, and J. Li, "Deep learning for lidar point clouds in autonomous driving: A review," *IEEE Transactions on Neural Networks and Learning Systems*, pp. 1–21, 2020.

[54] 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.

[55] 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.

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

[57] 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.

[58] 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.

[59] M. Abadi and et al., "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: http://tensorflow.org/

[60] Oracle, "Jni apis and developer guides," [Accessed 29. Sep. 2020]. [Online]. Available: https://docs.oracle.com/javase/8/docs/technotes/guides/jni

[61] N. M. Nawi, W. H. Atomi, and M. Rehman, "The effect of data pre-processing on optimized training of artificial neural networks," *Procedia Technology*, vol. 11, pp. 32 – 39, 2013, 4th International Conference on Electrical Engineering and Informatics, ICEEI 2013.

[62] A. Ricci, M. Piunti, and M. Viroli, "Environment programming in multi-agent systems: an artifact-based perspective," *Autonomous Agents and Multi-Agent Systems*, vol. 23, no. 2, pp. 158–192, Sep. 2011.

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

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

[65] 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

[66] J. Kober, J. A. Bagnell, and J. Peters, "Reinforcement learning in robotics: A survey," *The International Journal of Robotics Research*, vol. 32, no. 11, pp. 1238–1274, 2013. [Online]. Available: https://doi.org/10.1177/0278364913495721

[67] H. Zhu, J. Yu, A. Gupta, D. Shah, K. Hartikainen, A. Singh, V. Kumar, and S. Levine, "The ingredients of real world robotic reinforcement learning," in *International Conference on Learning Representations*, 2020. [Online]. Available: https://openreview.net/forum?id=rJe2syrtvS

[68] M. Botvinick, Y. Niv, and A. C Barto, "Hierarchically organized behavior and its neural foundations: A reinforcement learning perspective," *Cognition*, vol. 113, pp. 262–80, 11 2008.

[69] T. Brys, A. Harutyunyan, H. B. Suay, S. Chernova, M. E. Taylor, and A. Nowé, "Reinforcement learning from demonstration through shaping," in *Proc. of the 24th Int. Conf. on Artificial Intelligence (IJCAI '15)*, ser. IJCAI'15. AAAI Press, 2015, pp. 3352–3358.

[70] F. Bergenti, M.-P. Gleizes, and F. Zambonelli, *Methodologies and Software Engineering for Agent Systems: The Agent-Oriented Software Engineering Handbook*.

[71] A. Dosovitskiy, G. Ros, F. Codevilla, A. Lopez, and V. Koltun, "CARLA: An open urban driving simulator," in *Proceedings of the 1st Annual Conference on Robot Learning*, 2017, pp. 1–16.

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

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

[74] 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

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

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

[77] P. J. Huber, "Robust estimation of a location parameter," *Ann. Math. Statist.*, vol. 35, no. 1, pp. 73–101, 03 1964. [Online]. Available: https://doi.org/10.1214/aoms/1177703732

[78] J. Gu, Z. Wang, J. Kuen, L. Ma, A. Shahroudy, B. Shuai, T. Liu, X. Wang, G. Wang, J. Cai, and T. Chen, "Recent advances in convolutional neural networks," *Pattern Recognition*, vol. 77, pp. 354 – 377, 2018.