AUTONOMOUS HIGHWAY DRIVING SYSTEM WITH DEEP Q LEARNING

by

PENG XU

Submitted in partial fulfillment of the requirements

For the degree of Master of Science

Department of Electrical Engineering and Computer Science

CASE WESTERN RESERVE UNIVERSITY

May, 2018

Autonomous Highway Driving System with Deep Q Learning

Case Western Reserve University
Case School of Graduate Studies

We hereby approve the thesis[1] of

PENG XU

for the degree of

Master of Science

Dr. Wyatt Newman

Committee Chair, Adviser — Date
Department of Electrical Engineering and Computer Science

Dr. M. Cenk Cavusoglu

Committee Member — Date
Department of Electrical Engineering and Computer Science

[1]We certify that written approval has been obtained for any proprietary material contained therein.

*Dedicated to *your*
*dedication message goes here**

Table of Contents

List of Tables

List of Figures

Acknowledgements

0.1 Acknowledgements

It has been a long road to get to this point. Three years working through community college, another four for undergrad at VT, and another two for graduate school at VT. There are many people who have made a difference along the way, especially in this final stretch.

I want to thank my committee members, Dr. Alfred Wicks, Dr. Alan Asbeck, and Dr. Steve Southward for their guidance and being a part of my committee. I want to give an extra thank you to Dr. Wicks for the guidance you have provided, as well as always encour- aging me to do the things that I did not prefer to do that made me a better engineer. I owe a huge thank you to all those in the Mechatronics Lab. The amount of knowledge in the lab is immense and never ceases to amaze me. You all have made work enjoyable and I hope those in the next workplace are just as fun.

I want to thank my ?Intro to Thermodynamics? Professor, Dr. Anthony ?Tony? Ferrar. You were an excellent mentor when it came to discussing the decision on whether or not to pursue a graduate degree. Your passion for teaching, emphasis on education, and advice really influenced me continuing on to graduate school. I also want to thank my best friend, Katey Smith. Since meeting you sophomore year, you have continued to inspire me. You were another individual who encouraged me to attend graduate school, and in doing so, further inspired me to be the best that I can be.

Last, but certainly not least, thank you to my parents Denver C. and Norma, and my siblings Austin, Dylan, and Daniel. I could not have had a better family to inspire, support, and encourage me through undergrad, graduate school, and life in general. It is through your love and support that I am who I am today.

Abstract

Autonomous Highway Driving System with Deep Q Learning

Abstract

by

PENG XU

0.2 Abstract

The present paper describes a study that aims at assessment of driver behavior in response to new technology, particularly Adaptive Cruise Control Systems (ACCs), as a function of driving style. In this study possible benefits and drawbacks of Adaptive Cruise Control Systems (ACCs) were assessed by having participants drive in a simulator. The four groups of participants taking part differed on reported driving styles concerning Speed (driving fast) and Focus (the ability to ignore distractions), and drove in ways which were consistent with these opinions. The results show behavioral adaptation with an ACC in terms of higher speed, smaller minimum time headway and larger brake force. Driving style group made little difference to these behavioral adaptations. Most drivers evaluated the ACC system very positively, but the undesirable behavioral adaptations observed should encourage caution about the potential safety of such systems.

1 Introduction

1.1 Motivation and Background

Modern technologies have the potential to create a paradigm shift in the vehicle-driver relationship with advanced automation changing the driver role from "driving" to "supervising". To design new driver environments that caters for these emerging technologies, traditional approaches identify current human and technical constraints to system efficiency and create solutions accordingly. However, there are two reasons why such approaches are limited within the technologically-evolving automotive domain. First, despite significant progress in the development of system theory and methods, the application of these methods is largely constrained to the existence of a current system. Second, there are few structured approaches for using the analysis results to support design. In this paper, an attempt is made to overcome these challenges by developing and implementing a method for analyzing and designing a highly autonomous commercial vehicle.

An autonomous vehicle has great potential to improve driving safety, comfort and efficiency and can be widely applied in a variety of fields, such as road transportation, agriculture, planetary exploration, military purpose and so on[1]. The past three decades have witnessed the rapid development of autonomous vehicle technologies, which have

attracted considerable interest and efforts from academia, industry, and governments. Particularly in the past decade, contributing to significant advances in sensing, computer technologies, and artificial intelligence, the autonomous vehicle has become an extraordinarily active research field. During this period, several well-known projects and competitions for autonomous vehicles have already exhibited autonomous vehicles' great potentials in the areas ranging from unstructured environments to the on-road driving environments[2 3].

Complex functions like highly automated driving with combined longitudinal and lateral control will definitely appear first on highways, since traffic is more predictable and relatively safe there (one-way traffic only, quality road with relative wide lanes, side protections, good visible lane markings, no pedestrians or cyclists, etc.). As highways are the best places to introduce hands-free driving at higher speeds, one could expect a production vehicle equipped with a temporary autopilot or in other words automated highway driving assist function as soon as the end of this decade.

Automated highway driving means the automated control of the complex driving tasks of highway driving, like driving at a safe speed selected by the driver, changing lanes or overtaking front vehicles depending on the traffic circumstances, automatically reducing speed as necessary or stopping the vehicle in the right most lane in case of an emergency. Japanese Toyota Motor have already demonstrated their advanced highway driving support system prototype in real traffic operation. The two vehicles (shown on Figure 83) communicate each other, keeping their lane and following the preceding vehicle to maintain a safety distance. (Source: [70])

In this paper, we are aiming for an autonomous driving system allowing the vehicle to smartly choose the driving behaviors, such as adjusting the speed and changing the

lane. In fact, Adaptive cruise control (ACC), a radar-based system, has been designed to enhance driving comfort and convenience by relieving the need to continually adjust the speed to match that of a preceding vehicle. The system slows down when it approaches a vehicle with a lower speed, and the system increases the speed to the level of speed previously set when the vehicle upfront accelerates or disappears (e.g., by changing lanes). Traditional methods have proved the reliability in several cases though the use is still quite limited and only for expected scenarios. While, currently, artificial intelligence especially deep reinforcement learning is aggressively expanding the border of human's imagination and machine's autonomy. Thus, a new highway adaptive cruise control (HACC) is proposed for autonomous vehicles with the help of deep reinforcement learning.

1.2 Literation Review

The past few years have seen many breakthroughs using reinforcement learning (RL). The company DeepMind combined deep learning with reinforcement learning to achieve above-human results on a multitude of Atari games and, in March 2016, defeated Go champion Le Sedol four games to one. Though RL is currently excelling in many game environments, it is a novel way to solve problems that require optimal decisions and efficiency, and will surely play a part in machine intelligence to come.

Google's DeepMind published its famous paper Playing Atari with Deep Reinforcement Learning[4], in which they introduced a new algorithm called Deep Q Network (DQN for short) in 2013. It demonstrated how an AI agent can learn to play games by just observing the screen without any prior information about those games. The result

turned out to be pretty impressive. This paper opened the era of what is called "deep reinforcement learning", a mix of deep learning and reinforcement learning.

Reinforcement Learning is a type of machine learning that allows you to create AI agents that learn from the environment by interacting with it. Just like how we learn to ride a bicycle, this kind of AI learns by trial and error. As seen in Fig. 1.1, the brain represents the AI agent, which acts on the environment. After each action, the agent receives the feedback. The feedback consists of the reward and next state of the environment. The reward is usually defined by a human. If we use the analogy of the bicycle, we can define reward as the distance from the original starting point.

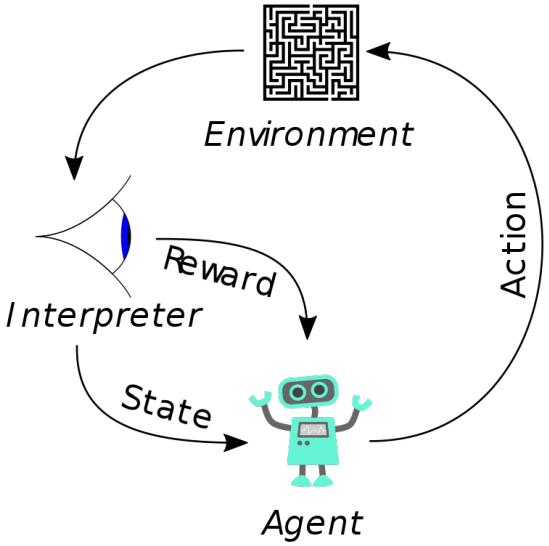


Figure 1.1. How an agent interacts with the environment.

Standard model-based methods for robotic manipulation might involve estimating the physical properties of the environment, and then solving for the controls based on the known laws of physics[5][6][7]. This approach has been applied to a range of problems.

Despite the extensive work in this area, tasks like pushing an unknown object to a desired position remain a challenging robotic task, largely due to the difficulties in estimating and modeling the physical world[8]. Learning and optimization-based methods have been applied to various parts of the state-estimation problem, such as object recognition[9], pose registration[10], and dynamics learning[11].

However, estimating and simulating all of the details of the physical environment is exceedingly difficult, particularly for previously unseen objects, and is arguably unnecessary if the end goal is only to find the desired controls. For example, simple rules for adjusting motion, such as increasing force when an object is not moving fast enough, or the gaze heuristic[12], can be used to robustly perform visuomotor control without an overcomplete representation of the physical world and complex simulation calculations. Our work represents an early step toward using learning to avoid the detailed and complex modeling associated with the fully model-based approach.

Several works have used deep neural networks to process images and represent policies for robotic control, initially for driving tasks[13,14], later for robotic soccer[15], and most recently for robotic grasping[16,17] and manipulation[18]. Although these model-free methods can learn highly specialized and proficient behaviors, they recover a task-specific policy rather than a flexible model that can be applied to a wide variety of different tasks. The high dimensionality of image observations presents a substantial challenge to model-based approaches, which have been most successful for low-dimensional non-visual tasks[19] such as helicopter control[20], locomotion[21], and robotic cutting[22]. Nevertheless, some works have considered modeling high-dimensional images for object interaction. For example, Boots et al.[6] learn a predictive model of RGB-D images of a robot arm moving in free space.

A lot of related work has been done in recent years in the design of CACC systems. Regarding the vehicle-following controller, Hallouzi et al. [8] did some research as part of the CarTalk 2000 project. These authors worked on the design of a longitudinal CACC controller based on vehicle-to-vehicle communication. They showed that inter-vehicle communication can help reduce instability of a platoon of vehicles. In the same vein, Naranjo and his colleague [14] worked on designing a longitudinal controller based on fuzzy logic. Their approach is similar to what we did with reinforcement learning for our low-level controller. Forbes has presented a longitudinal reinforcement learning controller [5] and compared it to a hand-coded following controller. He showed that the hand-coded controller is more precise than its RL controller but less adaptable in some situations. However, Forbes did not test explicitly communication between vehicles to improve its longitudinal controller to a multi-vehicle environment (which will be the focus of our future work). Our approach will also integrate our low-level controllers with a high-level multiagent decision making algorithm, which was not part of Forbes' work.

Regarding the reinforcement learning in a vehicle coordination problem, Unsal, Kachroo and Bay[23] have used multiple stochastic learning automata to control the longitudinal and lateral path of a vehicle. However, the authors did not extend their approach to the multi-agent problem. In his work, Pendrith[24] presented a distributed variant of Q-Learning (DQL) applied to lane change advisory system, that is close to the problem described in this paper. His approach uses a local perspective representation state which represents the relative velocities of the vehicles around. Consequently, this representation state is closely related to our 1-partial state representation. Contrary to our algorithms, DQL does not take into account the actions of the vehicles around and updates

Q-Values by an average backup value over all agents at each time step. The problem of this algorithm is the lack of learning stability.

On the other hand, our high level controller model is similar to Partially Observable Stochastic Games (POSG). This model formalizes theoretically the observations for each agent. The resolution of this kind of games has been studied by Emery-Montermerlo[25]. This resolution is an approximation using Bayesian games. However, this solution is still based on the model of the environment, unlike our approach which does not take into account this information explicitly since we assume that the environment is unknown. Concerning the space search reduction, Sparse Cooperative Q-Learning[26] allows agents to coordinate their actions only on predefined set of states. In the other states, agents learn without knowing the existence of the other agents. However, unlike in our approach, the states where the agents have to coordinate themselves are selected statically before the learning process. The joint actions set reduction has been studied by Fulda and Ventura who proposed the Dynamic Joint Action Perception (DJAP) algorithm[27]. DJAP allows a multi-agent Q-learning algorithm to select dynamically the useful joint actions for each agent during the learning. However, they concentrated only on joint actions and they tested only their approach on problems with few states.

Introducing communication into decision has been studied by Xuan, Lesser, and Zilberstein[28] who proposed a formal extension to Markov Decision Process with communication where each agent observes a part of the environment but all agents observe the entire state. Their approach proposes to alternate communication and action in the decentralized decision process. As the optimal policy computation is intractable, the authors proposed some heuristics to compute approximation solutions. The main

differences with our approach is the implicit communication and the model-free learning. More generally, Pynadath and Tambe[29] have proposed an extension to distributed POMDP with communication called COM-MTDP, which take into account the cost of communication during the decision process. They presented complexity results for some classes of team problems. As Xuan, Lesser, and Zilberstein[28], this approach is mainly theoretical and does not present model-free learning. The locality of interactions in a MDP has been theoretically developed by Dolgov and Durfee[30]. They presented a graphical approach to represent the compact representation of a MDP. However, their approach has been developed to solve a MDP and not to solve directly a multi-agent reinforcement learning problem where the transition function is unknown.

1.3 Thesis Outline

The goal of this thesis is to provide operational specifications for the development of a level 4 capable AVRP. This section will present the chapters and subsections of this thesis and provide a brief summary of those sections.

Chapter 2 details the system specifications required to develop an AVRP, based off feedback from faculty and researchers at Virginia Tech.

- **Section 2.1 Interdisciplinary Design Needs:** Looks at the feedback given by faculty and researchers at Virginia Tech and what kinds of research they would like to utilize an AVRP for. Breaks down the research desires into the basic needs for an AVRP. Some of the feedback is broken down with more background information.

Chapter 3 dives into the hardware and sensing side of an AVRP and lays out a discussion of the technology needed, such as DBW, navigation, sensing, communication,

computing, power bus, and external mounting racks. It then reviews additional design considerations that need to be taken into consideration when designing an AVRP. Chapter 3 is laid out as follows:

- **Section 3.1 Drive By Wire:** Discusses the DBW system and significant design parameters to be considered. Lays out the specifications of the by wire system on a base platform at Virginia Tech. Presents a high level overview of DBW system.
- **Section 3.2 Navigation:** Discusses different combinations for navigation, such as GPS/ INS and GPS/IMU, and some of the characteristics associated with each, as well as navigation via dead reckoning. Correction techniques such as DGPS and DGPS with RTK are reviewed. Hardware mounting is briefly mentioned.
- **Section 3.3.1 LIDAR:** Discusses how LIDAR works. 2D and 3D LIDAR is discussed. Discusses certain characteristics such as range, accuracy, field of view, number of re- turns, intensity, advantages and disadvantages. Reviews possible mounting locations.
- **Section 3.3.2 Radar:** Discusses how radar works and its usefulness on an AVRP. Talks about the different operation frequencies used in autonomy and how they affect performance, as well as other advantages and disadvantages. Possible mounting locations are discussed.
- **Section 3.3.3 Ultrasonic:** Discusses the use of ultrasonic sensors and their advantages and disadvantages. Reviews their range and mounting locations.

- **Section 3.3.4 Cameras:** Discusses different camera types and capability they bring to an AVRP. Goes over camera specifications for consideration. Reviews potential mounting locations.
- **Section 3.3.5 Wheel Speed and Steering Angle Sensors:** Discusses wheel encoders and steering angle sensors and how they complement other sensing and systems.
- **Section 3.4.1 Communication:** Presents different communication buses for intra-vehicle communication. Provides an overview, specs, and explanation of an AVRP communication bus structure. Reviews the need for an emergency stop system.
- **Section 3.4.2 Computers:** Addresses computing for the end users by presenting bench- marks and guide posts for consideration when determining computational needs for the end user of an AVRP.
- **Section 3.5 Base Sensor Suite Design:** Specs out the base sensor suite for an AVRP and the capabilities it allows for the AVRP without user added sensors.
- **Section 3.6 Power Bus:** Discusses AVRP power bus layout to provide power to all internally and externally mounted hardware and sensors. Provides an estimate of power need for base sensor suite and an example sensor suite.
- **Section 3.7.1 Front Universal Mounting Racks:** Reviews details and modifications made for adding front universal mounting rack to AVRP.
- **Section 3.7.2 Rear Universal Mounting Racks:** Reviews details and modifications made for adding rear universal mounting rack to AVRP.
- **Section 3.7.3 Roof Universal Mounting Racks:** Reviews details and modifications made for adding roof top universal mounting rack to AVRP.

- **Section 3.7.4 Mounting Rack Vibration:** Discusses vibration concerns and performs example natural frequency calculations for the front and rear universal mounting racks.
- **Section 3.7.5 Interchanging Hardware:** Discusses design considerations for hardware and sensors to be added to the mounting racks, such as roof top penetrations, routing wires, adapter brackets, etc.
- **Section 3.8 Additional Design Considerations:** Discusses other AVRP design considerations, such as funding, team structure, base vehicle variability, hardware selection properties, etc.

Chapter 4 reviews the base vehicle research capabilities and contains a testing plan ideas for an AVRP. Chapter 4 is a follows:

- **Section 4.1 Base Vehicle Research Capabilities:** Reiterates on the base platform capabilities without any user added sensors.
- **Section 4.2 Testing Plan Overview:** Discusses testing plan options for an AVRP and how it can be validated for use for its researchers.

Chapter 5 contains the conclusion and areas for future work. Chapter 5 sections are as follows:

- **Section 5.1 Conclusion:** Reviews the design of an AVRP, hardware, sensors, and modifications to make it adaptable to a researcher?s needs.
- **Section 5.2 Future Work:** Looks at areas relevant to an AVRP in which further research can be conducted.

2 Simulated Environment

In order for autonomous vehicles to operate safely in the real world, they must be able to adapt to a multitude of changing conditions. Before fully self-driving vehicles hit the road, they undergo a lengthy period of testing where the vehicle's sensors and artificial intelligence are tested in a variety of simulated real-world environments. Simulation has become the backbone of the autonomous driving industry, providing a means to collect extensive amounts of data for model training as well as providing a safe testbed to crash-test these models. In this chapter, we would like to create a simulator for training autonomous diving algorithms.

A simplified highway environment would be created to train deep learning models for the autonomous vehicle to gain better decision making ability of speed control and lane changing.

The simulated environment is constructed based on ROS and Gazebo and the Reinforcement Learning Algorithms are implemented by OpenAI-Gym. An open source car model kit, Drive-by-Wire (DBW) Kit, is adopted to play the role of the autonomous vehicle. An overall architecture of the simulation environment is shown as Fig. 2.1.

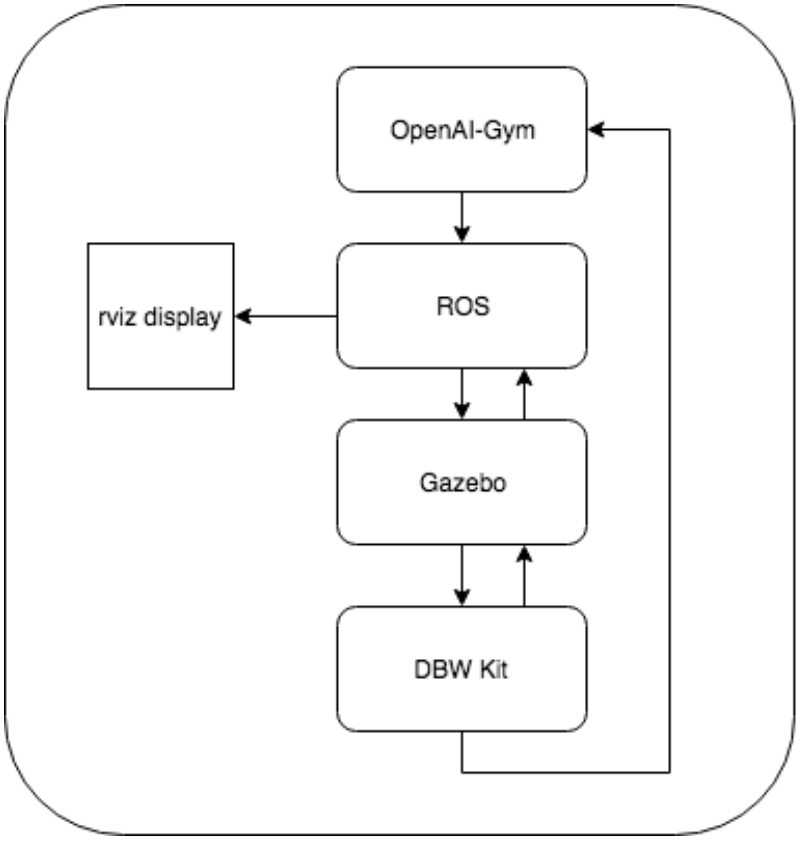


Figure 2.1. Architecture of the simulation environment.

.

OpenAI was founded in late 2015 as a non-profit with a mission to "build safe artificial general intelligence (AGI) and ensure AGI's benefits are as widely and evenly distributed as possible." In addition to exploring many issues regarding AGI, one major contribution that OpenAI made to the machine learning world was developing both the Gym and Universe software platforms. OpenAI-Gym is a collection of environments / problems designed for testing and developing reinforcement learning algorithms — it saves the user from having to create complicated environments. Gym is written in Python, and there are multiple environments such as robot simulations or Atari games. There is also an online leaderboard for people to compare results and code.

The usual basic requirements to robot simulators are an accurate physics simulation (such as object velocity, inertia, friction, position and orientation, etc.), high quality rendering (for shape, dimensions, colors, and texture of objects), integration with the Robot Operating System (ROS) framework and multi-platform performability. It provides great opportunities for modeling robots and their sensors together with developing robot control algorithms, realizing mobile robot simulation, visualization, locomotion and navigation in a realistic 3D environment. As mentioned in the paper[31], the high graphical fidelity in a robot simulation is important because the sensory input to the robot perceptual algorithms comes from virtual sensors, which are also provided by the simulation. For example, virtual cameras use the simulator rendering engine to obtain their images. If images from a simulated camera have incorrect similarity to real camera ones, then it is not possible to use them for object recognition and localization.

To avoid such a sort of problems, we use the robust and high graphical quality robot simulator — Gazebo, which is an open source robotic simulation package that closely integrated with ROS. Gazebo uses the open source OGRE rendering engine, which produces good graphics fidelity, although it also employs the Open Dynamics Engine (ODE), which is estimated as sufficiently slow physics engine[31].

2.1 ROS

Writing software for robots is difficult, particularly as the scale and scope of robotics continues to grow. Different types of robots can have wildly varying hardware, making code reuse nontrivial. A wide variety of frameworks were created to liberate researchers and developers from those way beyond their interests. Among them, Robot Operating

System (ROS) framework[32] gained more popularity because of its generality and expansibility. ROS was designed to meet a specific set of challenges encountered when developing large-scale service robots as part of the STAIR project[33] at Stanford University and the Personal Robots Program[34] at Willow Garage, but the resulting architecture is far more general than the service-robot and mobile-manipulation domains.

The philosophical goals of ROS can be summarized as:

- Peer-to-peer
- Tools-based
- Multi-lingual
- Thin
- Free and Open-Source

2.1.1 Implementation

The fundamental concepts of the ROS implementation are nodes, messages, topics, and services.

Nodes are processes that perform computation. ROS is designed to be modular at a fine-grained scale: a system is typically comprised of many nodes. In this context, the term "node" is interchangable with "software module". Our use of the term "node" arises from visualizations of ROS-based systems at runtime: when many nodes are running, it is convenient to render the peer-to-peer communications as a graph, with processes as graph nodes and the peer-to-peer links as arcs.

Nodes communicate with each other by passing messages. A message is a strictly typed data structure. Standard primitive types (integer, floating point, boolean, etc.) are

supported, as are arrays of primitive types and constants. Messages can be composed of other messages, and arrays of other messages, nested arbitrarily deep.

A node sends a message by publishing it to a given topic, which is simply a string such as "odometry" or "map". A node that is interested in a certain kind of data will subscribe to the appropriate topic. There may be multiple concurrent publishers and subscribers for a single topic, and a single node may publish and/or subscribe to multiple topics. In general, publishers and subscribers are not aware of each others' existence.

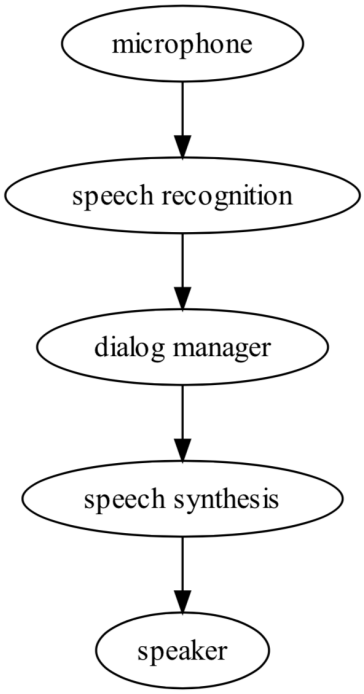


Figure 2.2. Communication pipeline.

2.1.2 Collaborative Development

Due to the vast scope of robotics and artificial intelligence, collaboration between modules is necessary in order to build large systems. To support collaborative development,

the ROS software system is organized into packages. Our definition of "package" is deliberately open-ended: a ROS package is simply a directory which contains an XML file describing the package and stating any dependencies.

A collection of ROS packages is a directory tree with ROS packages at the leaves: a ROS package repository may thus contain an arbitrarily complex scheme of subdirectories. For example, one ROS repository has root directories including "nav", "vision" and "motion planning" each of which contains many packages as subdirectories.

The open-ended nature of ROS packages allows for great variation in their structure and purpose: some ROS packages wrap existing software, such as Player or OpenCV, automating their builds and exporting their functionality. Some packages build nodes for use in ROS graphs, other packages provide libraries and standalone executables, and still others provide scripts to automate demonstrations and tests. The packaging system is meant to partition the building of ROS-based software into small, manageable chunks, each of which can be maintained and developed on its own schedule by its own team of developers.

2.1.3 Visualization and Monitoring

While designing and debugging robotics software, it often becomes necessary to observe some state while the system is running. Although *printf* is a familiar technique for debugging programs on a single machine, this technique can be difficult to extend to large-scale distributed systems, and can become unwieldy for general-purpose monitoring.

Instead, ROS can exploit the dynamic nature of the connectivity graph to "tap into" any message stream on the system. Furthermore, the decoupling between publishers and subscribers allows for the creation of general-purpose visualizers. Simple programs

can be written which subscribe to a particular topic name and plot a particular type of data, such as laser scans or images. However, a more powerful concept is a visualization program which uses a plugin architecture: this is done in the *rviz* program, which is distributed with ROS. Visualization panels can be dynamically instantiated to view a large variety of datatypes, such as images, point clouds, geometric primitives (such as object recognition results), render robot poses and trajectories, etc. Plugins can be easily written to display more types of data.

A native ROS port is provided for Python, a dynamically-typed language supporting introspection. Using Python, a powerful utility called *rostopic* was written to filter messages using expressions supplied on the command line, resulting in an instantly customizable "message tap" which can convert any portion of any data stream into a text stream. These text streams can be piped to other UNIX command-line tools such as *grep*, *sed*, and *awk*, to create complex monitoring tools without writing any code.

Similarly, a tool called *rxplot* provides the functionality of a virtual oscilloscope, plotting any variable in real-time as a time series, again through the use of Python introspection and expression evaluation.

2.1.4 Transformations

Robotic systems often need to track spatial relationships for a variety of reasons: between a mobile robot and some fixed frame of reference for localization, between the various sensor frames and manipulator frames, or to place frames on target objects for control purposes.

To simplify and unify the treatment of spatial frames, a transformation system has been written for ROS, called *tf*. The *tf* system constructs a dynamic transformation tree which relates all frames of reference in the system. As information streams in from the

various subsystems of the robot (joint encoders, localization algorithms, etc.), the *tf* system can produce streams of transformations between nodes on the tree by constructing a path between the desired nodes and performing the necessary calculations.

For example, the *tf* system can be used to easily generate point clouds in a stationary "map" frame from laser scans received by a tilting laser scanner on a moving robot. As another example, consider a two-armed robot: the *tf* system can stream the transformation from a wrist camera on one robotic arm to the moving tool tip of the second arm of the robot. These types of computations can be tedious, error- prone, and difficult to debug when coded by hand, but the *tf* implementation, combined with the dynamic messaging infrastructure of ROS, allows for an automated, systematic approach.

2.2 Gazebo

Gazebo is a 3D dynamic simulator with the ability to accurately and efficiently simulate populations of robots in complex indoor and outdoor environments, which makes it possible to rapidly test algorithms, design robots, perform regression testing, and train AI system using realistic scenarios. While similar to game engines, Gazebo offers physics simulation at a much higher degree of fidelity, a suite of sensors, and interfaces for both users and programs. Fig. gives a typical display of Gazebo and in the window is a PR2 robot[34] with its LIDAR sensor range displayed in blue.

Typical uses of Gazebo include:

- testing robotics algorithms,
- designing robots,
- performing regression testing with realistic scenarios

A few key features of Gazebo include:

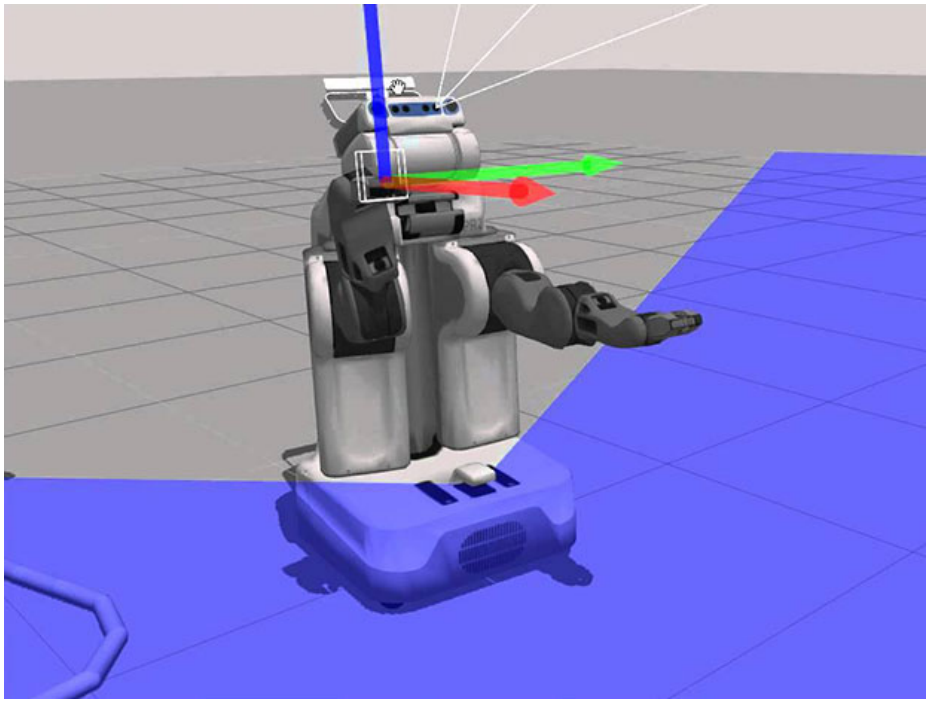


Figure 2.3. Gazebo for robot simulation.

- multiple physics engines,
- a rich library of robot models and environments,
- a wide variety of sensors,
- convenient programmatic and graphical interfaces

Gazebo is far from being the only choice for a 3D dynamics simulator. It is however one of the few that attempts to create realistic worlds for the robots rather than just human users. As more advanced sensors are developed and incorporated into Gazebo the line between simulation and reality will continue to blur, but accuracy in terms of robot sensors and actuators will remain an overriding goal.

2.2.1 Architecture

Gazebo's architecture has progressed through a couple iterations during which we learned how to best create a simple tool for both developers and end users. We realized from the start that a major feature of Gazebo should be the ability to easily create new robots, actuators, sensors, and arbitrary objects. As a result, Gazebo maintains a simple API for addition of these objects, which we term models, and the necessary hooks for interaction with client programs. A layer below this API resides the third party libraries that handle both the physics simulation and visualization. The particular libraries used were chosen based on their open source status, active user base, and maturity.

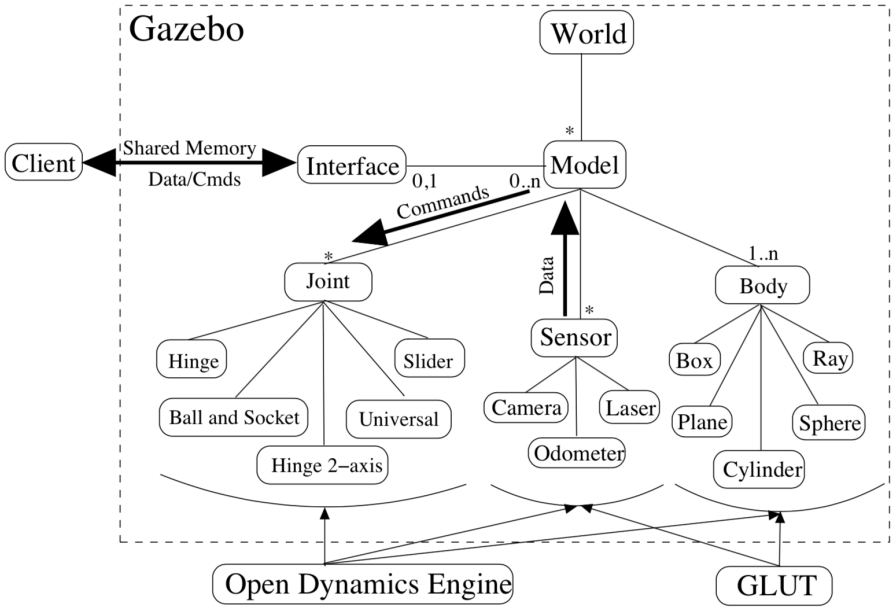


Figure 2.4. Gazebo for robot simulation.

This architecture is graphically depicted in Fig. 2.4. The World represents the set of all models and environmental factors such as gravity and lighting. Each model is composed of at least one body and any number of joints and sensors. The third party

libraries interface with Gazebo at the lowest level. This prevents models from becoming dependent on specific tools that may change in the future. Finally, client commands are received and data returned through a shared memory interface. A model can have many interfaces for functions involving, for example, control of joints and transmission of camera images.

2.2.2 Physics Engine

The Open Dynamics Engine, created by Russel Smith is a widely used physics engine in the open source community. It is designed to simulate the dynamics and kinematics associated with articulated rigid bodies. This engine includes many features such as numerous joints, collision detection, mass and rotational functions, and many geometries including arbitrary triangle meshes. Gazebo utilizes these features by providing a layer of abstraction situated between ODE and Gazebo models. This layer allows easy creation of both normal and abstract objects such as laser rays and ground planes while maintaining all the functionality provided by ODE. With this internal abstraction, it is possible to replace the underlying physics engine, should a better alternative become available.

2.2.3 Visualization

A well designed simulator usually provides some form of user interface, and Gazebo requires one that is both sophisticated and fast. The heart of Gazebo lies in its ability to simulate dynamics, and this requires significant work on behalf of the user's computer. A slow and cumbersome user interface would only detract from the simulator's primary purpose. To account for this, OpenGL and GLUT (OpenGL Utility Toolkit) were chosen as the default visualization tools.

OpenGL is a standard library for the creation of 2D and 3D interactive applications. It is platform independent, highly scalable, stable, and continually evolving. More importantly, many features in OpenGL have been implemented in graphic card hardware thereby freeing the CPU for other work such as the computationally expensive dynamics engine.

GLUT is a simple window system independent toolkit for OpenGL applications. Scenes rendered using OpenGL are displayed in windows created by GLUT. This toolkit also provides mechanisms for user interaction with Gazebo via standard input devices such as keyboards and mice. GLUT was chosen as the default windowing toolkit be- cause it is lightweight, easy to use, and platform independent.

2.2.4 Customized Environment

A complete environment is essentially a collection of models and sensors. The ground and buildings represent stationary models while robots and other objects are dynamic. Sensors remain separate from the dynamic simulation since they only collect data, or emit data if it is an active sensor.

Models: A model is any object that maintains a physical representation, which can be created by hand. The process starts with choosing the appropriate bodies and joints necessary to build an accurate model in both appearance and functionality. This encompasses anything from simple geometry to complex robots. Models are composed of at least one rigid body, zero or more joints and sensors, and interfaces to facilitate the flow of data.

Bodies represent the basic building blocks of a model. Their physical representation take the form of geometric shapes chosen from boxes, spheres, cylinders, planes, and

lines. Each body has an assigned mass, friction, bounce factor, and rendering properties such as color, texture, transparency, etc.

Joints provide the mechanism to connect bodies together to form kinematic and dynamic relationships. A variety of joints are available including hinge joints for rotation along one or two axis, slider joints for translation along a single axis, ball and socket joints, and universal joints for rotation about two perpendicular joints. Besides connecting two bodies together, these joints can act like motors. When a force is applied to a joint, the friction between the connected body and other bodies cause motion. However, special care needs to be taken when connecting many joints in a single model as both the model and simulation can easily loose stability if incorrect parameters are chosen.

Interfaces provide the means by which client programs can access and control models. Commands sent over an interface can instruct a model to move joints, change the configuration of associated sensors, or request sensor data. The interfaces do not place restrictions on a model, thereby allowing the model to interpret the commands in anyway it sees fit.

Sensors: A robot can't perform useful tasks without sensors. A sensor in Gazebo is an abstract device lacking a physical representation. It only gains embodiment when incorporated into a model. This feature allows for the reuse of sensors in numerous models thereby reducing code and confusion.

There currently are three sensor implementations including an odometer, ray proximity, and a camera. Odometry is easily accessible through integration of the distance traveled. The ray proximity sensor returns the contact point of the closest object along the ray's path.

External Interfaces: From the users point of view, the models simulated in Gazebo are the same as their real counterparts, and are treated as a normal device capable of sending and receiving data. A second key advantage to this approach is that one can use abstract drivers inside a simulation.

The low-level library provides a mechanism for any third-party robot device server interface with Gazebo. Going even further, a connection to the the library is not even necessary since Gazebo can be run independently for rapid model and sensor development. Currently the Gazebo library offers hooks to set wheel velocities, read data from a laser range finder, retrieve images from a camera, and insert simple objects into the environment at runtime. This data is communicated through shared memory for speed and efficiency.

Environments: Many environments in which robots operate are either well studied or carefully constructed. Deploying robots in a never before encountered world may cause unforeseen, and possibly negative, side effects. Lighting conditions, reflective surfaces, and odd objects can all play an effect on how a robot operates. A strategy of online testing can be extremely slow and tedious. Time can be spent much more productively by testing and modifying the robot controllers offline in preparation for the real experiments. The fine grained control of Gazebo, the ability to extrude 2D images into 3D structures, and terrain generation allow for the unique ability to hand create rough outlines of a new environment.

As a result, the development time of the algorithms employed was greatly reduced. Gazebo made it possible to continue experimentation in the environment even after the physical robots were deployed. Elevation information collected by real sensors can be imported along with relevant structures to further blur the line between simulation and

the real world. All of this culminates in the ability of Gazebo to reduce development and test time, and even allow experiments to virtually take place in almost any part of the world.

2.2.5 Test Bed for Algorithm Design

The design and implementation of new algorithms can be a difficult task that become particularly acute with the lack of convenient test environments. In situations such as this, Gazebo's sensory realism can play a time saving role. Traditionally, development of new algorithms either required custom simulators or direct testing on the hardware; Gazebo's realistic environments and simple interface can drastically reduce the turn around time from a conceptual stage to functional system.

2.3 OpenAI-Gym

Reinforcement learning assumes that there is an agent that is situated in an environment. Each step, the agent takes an action, and it receives an observation and reward from the environment. An RL algorithm seeks to maximize some measure of the agent?s total reward, as the agent interacts with the environment. In the RL literature, the environment is formalized as a partially observable Markov decision process (POMDP) [12].

OpenAI Gym focuses on the episodic setting of reinforcement learning, where the agent?s experience is broken down into a series of episodes. In each episode, the agent?s initial state is randomly sampled from a distribution, and the interaction proceeds until the environment reaches a terminal state. The goal in episodic reinforcement learning is to maximize the expectation of total reward per episode, and to achieve a high level of performance in as few episodes as possible.

OpenAI Gym aims to combine the best elements of these previous benchmark collections, in a software package that is maximally convenient and accessible. It includes a diverse collection of Environments (POMDPs) with a common interface, and this collection will grow over time. The environments are versioned in a way that will ensure that results remain meaningful and reproducible as the software is updated.

2.3.1 Design For Environment

The design of OpenAI Gym is based on the experience developing and comparing reinforcement learning algorithms, and the experience using previous benchmark collections. Below, we will summarize some of our design decisions.

Two core concepts in Reinforcement Learning are the agent and the environment. OpenAi Gym's design focuses on providing an abstraction for the environment, but not for the agent. This choice was to maximize convenience for users and allow them to implement different styles of agent interface. First, one could imagine an "online learning" style, where the agent takes (observation, reward, done) as an input at each time-step and performs learning updates incrementally. In an alternative "batch update" style, a agent is called with observation as input, and the reward information is collected separately by the RL algorithm, and later it is used to compute an update. By only specifying the agent interface, it is allowed to write customized agents with either of these styles.

2.3.2 Interfacing with ROS and Gazebo

In the context of robotics, reinforcement learning offers a framework for the design of sophisticated and hard-to-engineer behaviors [2]. The challenge is to build a simple environment where this machine learning techniques can be validated, and later applied in a real scenario.

OpenAI Gym leaves interfaces to write customized agents, which makes it possible to integrate the Gym API with robotic hardware, validating reinforcement learning algorithms in real environments. Real-world operation is achieved combining Gazebo simulator with ROS.

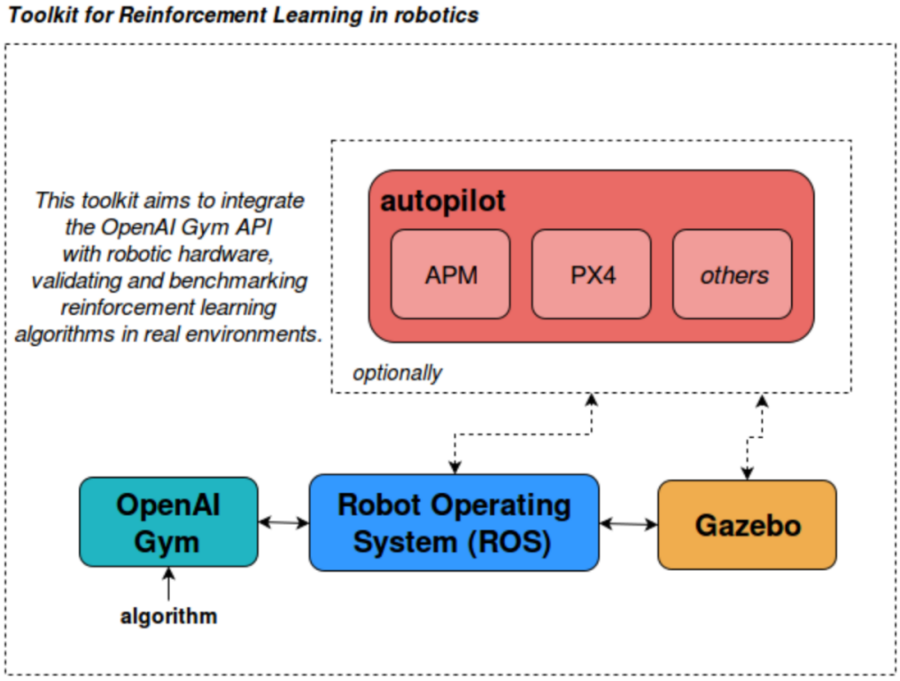


Figure 2.5. Simplified software architecture used in OpenAI Gym for robotics.

The architecture consists of three main software blocks: OpenAI Gym, ROS and Gazebo as shown in Fig. 2.5. Environments developed in OpenAI Gym interact with ROS, which is the connection between the Gym itself and Gazebo simulator. Gazebo provides a robust physics engine, high-quality graphics, and convenient programmatic and graphical interfaces.

The physics engine needs a robot definition in order to simulate it, which is provided by ROS or a Gazebo plugin that interacts with an autopilot in some cases (depends on the robot software architecture).

2.3.3 Example Use: Train a Cart-pole agent

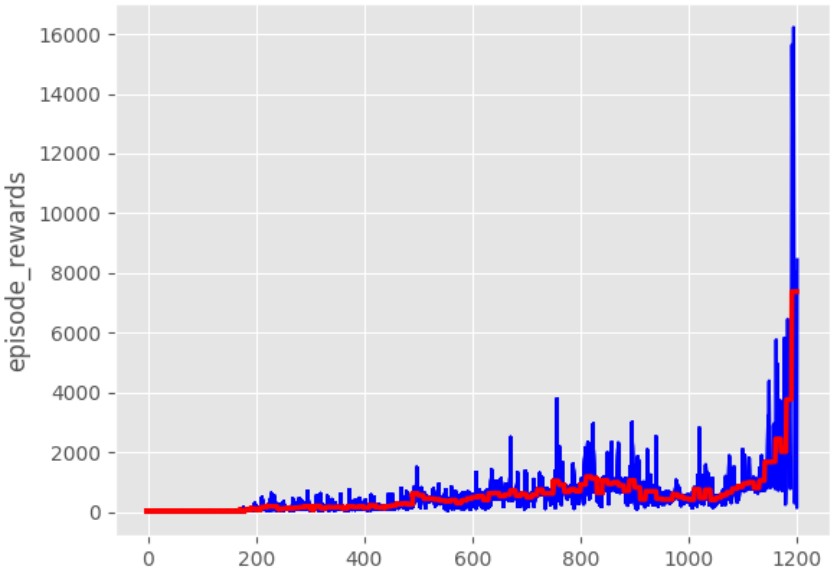


Figure 2.6. A Cart-pole model created in Gazebo.

Before we dive into a complex autonomous driving problem on highways, it is helpful to apply the Integrated OpenAI Gym on a simple enough problem, for example, Cart-Pole problem. In this problem or game, the pole needs to keep its balance purely relying on moving the cart in a one-degree axes. A Cart-Pole model was created in Gazebo as shown in Fig. 2.6.

After the controller interface and the *TF* tree are correctly defined, the model and the motion can be monitored in ROS-rviz, as shown in Fig. 2.7.

A Q-Learning Algorithm is applied and the policy are defined as below.

State Space

Action Space

Reward Function

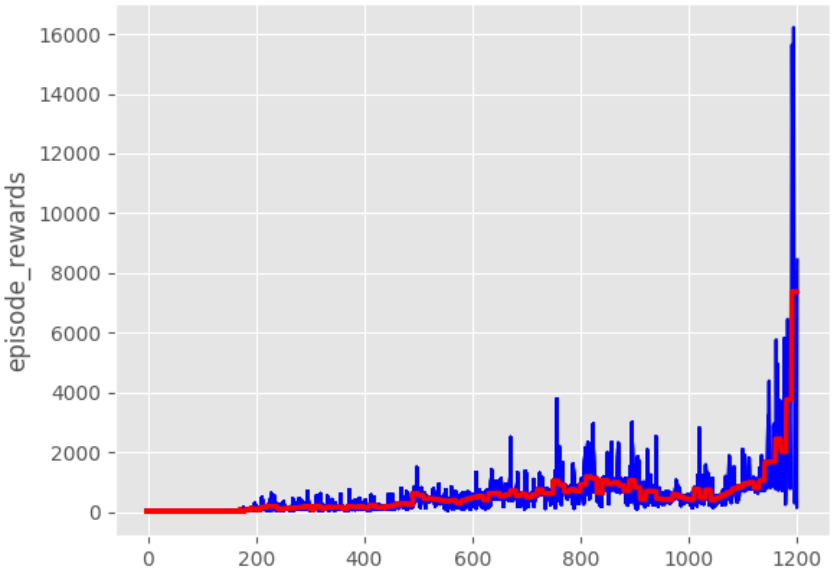


Figure 2.7. A Cart-pole model created in ROS-Rviz.

The hyperparameters are set as below,

- Epochs: 1000.
- Learning Rate: 0.005.
- Initial Exploration Rate: 1.
- Final Exploration Rate: 0.05.

As shown in Fig. 2.8, the accumulated reward in each episode was gradually increasing and had a huge jump when it came to Episode 1200. After that, it had a satisfying ability to keep balance for a considerable period of time.

2.4 Vehicle Model: DBW Kit

A well performed vehicle model kit, ADAS Kit Gazebo/ROS Simulator, is adopted here.

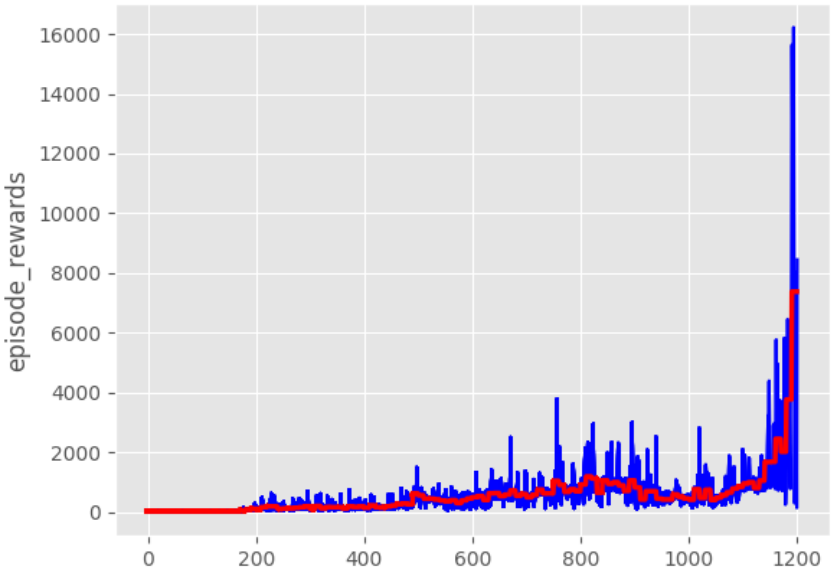


Figure 2.8. Reward history of training A Cart-Pole Agent.

2.4.1 URDF Models

Figure 2.9. 4 vehicle models in ADAS Kit.

.

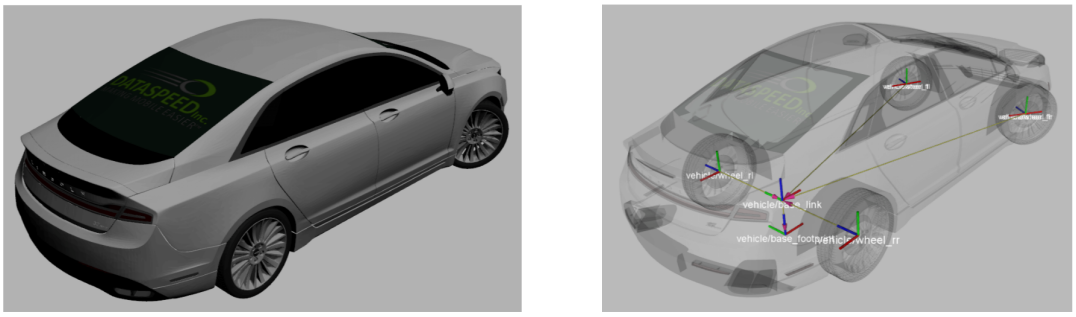


Figure 2.10. Simulation model and corresponding TF tree.

Four URDF models representing the different vehicles supported by the Dataspeed ADAS Kit are included in the simulation, as shown in Fig. 2.9. The TF trees of the simulation models are all the same, and this common TF tree is shown in Fig. 2.10.

2.4.2 Simulated CAN Message Interface

The simulator emulates the CAN message interface to the real ADAS Kit. Therefore, there are only two ROS topics used to interact with the simulated vehicle: can bus dbw can tx to send CAN messages to the vehicle, and can bus dbw can rx to receive feedback data from the vehicle. These topics and their corresponding message types are listed in Table 2.1.

Topic Name	Msg Type
$<name>/can_bus_dbw/can_rx$	$can_msgs/Frame$
$<name>/can_bus_dbw/can_tx$	$can_msgs/Frame$

Table 2.1. CAN message topics to interact with simulated ADAS Kit.

The simulator only implements a subset of the complete Dataspeed CAN message specification. The supported command messages are listed in Table 2.2, and the supported report messages are listed in Table 3. See the ADAS Kit datasheets for complete CAN message information.

Command Msg	CAN ID
Brake	0x060
Throttle	0x062
Steering	0x064
Gear	0x066

Table 2.2. Command CAN messages supported by the ADAS Kit simulator.

Report Msg	CAN ID	Data Rate
Brake	0x061	50 Hz
Throttle	0x063	50 Hz
Steering	0x065	50 Hz
Gear	0x067	20 Hz
Misc	0x069	50 Hz
Wheel Speed	0x06A	100 Hz
Accel	0x06B	100 Hz
Gyro	0x06C	100 Hz
GPS1	0x6D	1 Hz
GPS2	0x6E	1 Hz
GPS3	0x6F	1 Hz
Brake Info	0x074	50 Hz

Table 2.3. Report CAN messages supported by the ADAS Kit simulator.

2.4.3 Simulating Multiple Vehicles

The parameters of the ADAS Kit Gazebo simulation are set using a single YAML file. This section describes the options and formatting of the YAML file.

To simulate multiple vehicles simultaneously, simply add more dictionaries to the array in the YAML file. Below is an example:

```
- vehicle1:
x: -2.0
y: 0.0
color: red
model: mkz
year: 2017
```

```
- vehicle2:
x: 0.0
y: 2.0
color: green
model: fusion
```

This would spawn two vehicles: one red 2017 MKZ with model name vehicle1 spawned at (0.0, -2.0, 0.0) and one green 2013 Fusion with model name vehicle2 spawned at (0.0, 2.0, 0.0). Both vehicles would have the default values of the parameters not set in the individual dictionaries.

3 System Design

Current autonomous vehicles use the same architecture as the Urban DARPA Challenge vehicles did[35 36 37 38]. This architecture comprises three main processing modules, described below and illustrated in Fig. 3.1.

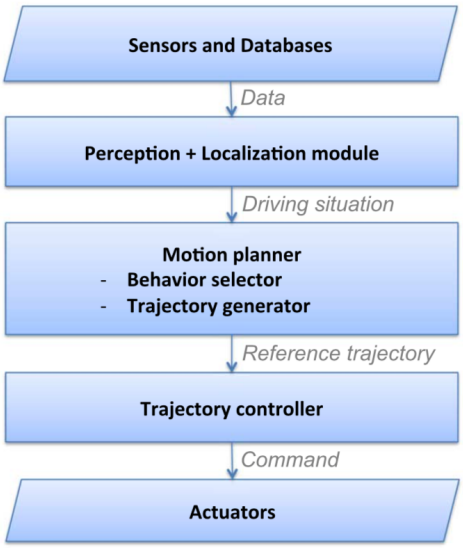


Figure 3.1. Standard Architecture of Autonomous Vehicle.

- The "Perception + Localization" module combines data received from sensors and digital maps to estimate some relevant features representing the driving situation (e.g. position of other vehicles, road geometry).

- The "Motion planner" selects the appropriate high-level behavior (e.g. car following, lane changing) and generates a trajectory corresponding to that behavior.
- The "Trajectory controller" computes the steering and acceleration commands with the objective to follow the reference trajectory as closely as possible. These commands are sent to the actuators.

This architecture has been successfully used in the field of terrestrial robotics for decades. Our autonomous driving framework inherits the most of it and replaces the motion planner with a trained driver model, which will be described in detail in Chapter 4.

The driver model can be designed independently and can be modified at any time without impacting the performance of the other. This feature is particularly useful if one wants to adjust the car's driving style over time: the driver model can learn continuously, or be replaced, without having to readjust any other module. One could also imagine extending the architecture in Fig. 3.2 to take advantage of cloud-based computing and learn new models based on data collected from millions of drivers.

The framework proposed above is general and can be applied to a variety of driving scenarios. We will implement and test it for the longitudinal and lateral control of an autonomous vehicle during lane keeping and lane changing. In this scenario, the commanded input is the acceleration and steering of the vehicle. The system is aiming for constructing at least the following modules or functions.

- **Path following** Detecting and determining a path/lane to follow, and following it.

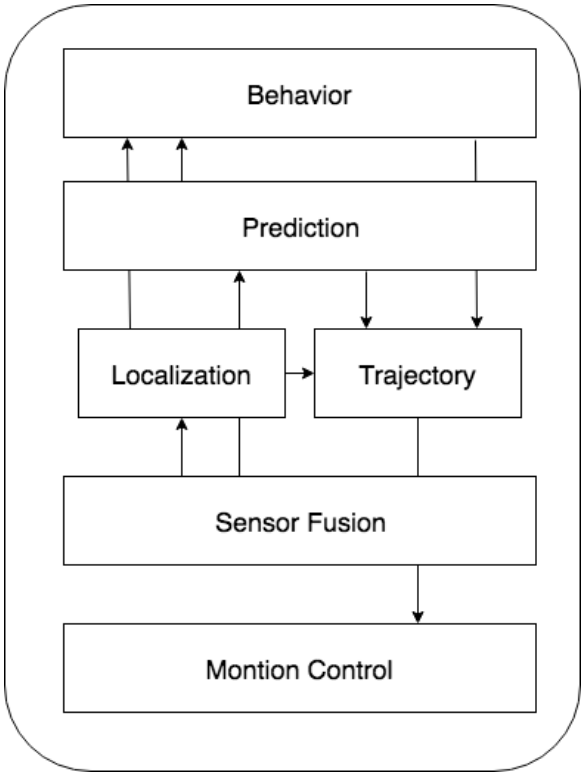


Figure 3.2. Standard Architecture of Autonomous Vehicle Control.

- **Lane and obstacle detection** Detecting driving lanes and obstacles to successfully navigate them.
- **ACC** Detect a leading vehicle or a trailing vehicle and maintain a safe distance and speed.
- **Adaptive braking** Braking system adapts braking to different driving conditions to improve response time, overall safety, etc.

3.1 Motion Planner

3.1.1 Longitudinal Control: Adaptive Cruise Control

We define the following variables to represent the relative motion of the autonomous vehicle (referred to as the "ego vehicle") and the vehicle located ahead or behind in the

same lane as the autonomous vehicle (referred to as the "preceding vehicle" or the "trailing car").

- $\epsilon = [d_t, v_t]$ is the state of the ego vehicle at time t, where $d_t \in R^+$ is the longitudinal position of the ego vehicle in a road-aligned coordinate system, and $v_t \in R^+$ is the longitudinal velocity of the ego vehicle.
- $\epsilon = [d_t^p, v_t^p]$ is the state of the preceding vehicle at time t, where $d_t \in R^+$ is the longitudinal position of the preceding vehicle in a road-aligned coordinate system, and $v_t \in R^+$ is the longitudinal velocity of the preceding vehicle.
- $\epsilon = [d_t^t, v_t^t]$ is the state of the trailing vehicle at time t, where $d_t \in R^+$ is the longitudinal position of the trailing vehicle in a road-aligned coordinate system, and $v_t \in R^+$ is the longitudinal velocity of the trailing vehicle.
- $z_t = [d_t^{pr}, d_t^{tr}, v_t^p, v_t^t, v_t^t]$ are the features representing the current driving situation at time t , to be used by the driver model to generate an appropriate acceleration command. $d_t^{pr} = d_t^p - d_t$ is the relative distance to the preceding vehicle and $d_t^{tr} = d_t^t - d_t$ is the relative distance to the preceding vehicle.

At each time step t , the driver model generates an acceleration command. This acceleration command is used by the trajectory generator to compute a target velocity as a reference. The controller solves a constrained optimization problem over the prediction horizon , and generates a planned velocity sequence which guarantees the safety of the vehicle.

3.1.2 Lateral Control: Lane Change Control

We define a one-dimensional array with boolean values to represent the lateral position of the autonomous vehicle in a 3-lane highway scenario.

- $lane = [false, true, false]$, for example, represents the vehicle locates at the middle lane at time t.

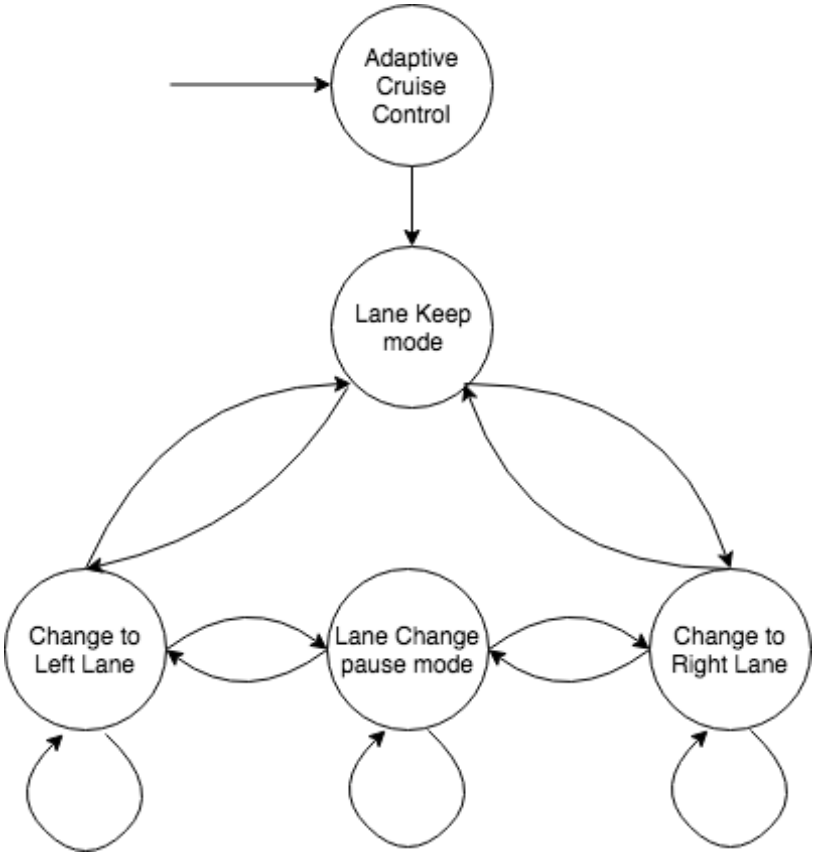


Figure 3.3. The finite state machine for lateral motion control.

At each time step t , the driver model, with the information above together with the other vehicle in its left and right lanes, generates a change lane or a keep lane command. The proposed algorithm will encourage the ego vehicle to change to a different lane neighboring to it when the target lane has better driving condition and to keep the current lane if else. To avoid conflict two lane changing commands which are too close, the lateral control would stay the current state for a period of time for the lane change to finish. Within the time period, new lane change or lane keep commands would be

ignored. By this idea, it becomes a rule-based controller, sometimes referred to as "?nite state machine (FSM)", as shown in Fig. 3.3. FSM has its own advantages, including:

(1) **Clear in structure:** the controller is based on "if-then-else" logic, which is explicitly readable, so that the controller's behavior can be relatively easily predicted;

(2) **Easy to calibrate:** a FSM usually has a ?nite number of parameters, so that it is easy to calibrate and optimize;

(3) **More reliable:** A well-calibrated FSM is usually more reliable, compared to some other frameworks, e.g., based on function approximation techniques as used in machine learning-based approaches.

3.2 Trajectory Generator

3.2.1 Trajectory Planning

The proposed Trajectory Generator is used to generate a safe and comfortable path (with an appropriate speed and acceleration) from an initial position towards a destination, while complying with a global route and map. Our method aims to resolve local path planning problems based on a global route and map. The global route is obtained by the high precision navigation system, and the map is downloaded from the Internet. The map is composed of a set of waypoints on the road edges and topology that describes the relationships between connected roads, as shown in Fig. 3.4. The process by which the map is obtained falls outside the scope of this thesis. Therefore, the maps used in this paper are predefined in our simulations.

The dynamic path planning process includes three stages: center line construction, path candidate generation, and path selection. These are performed on the basis of

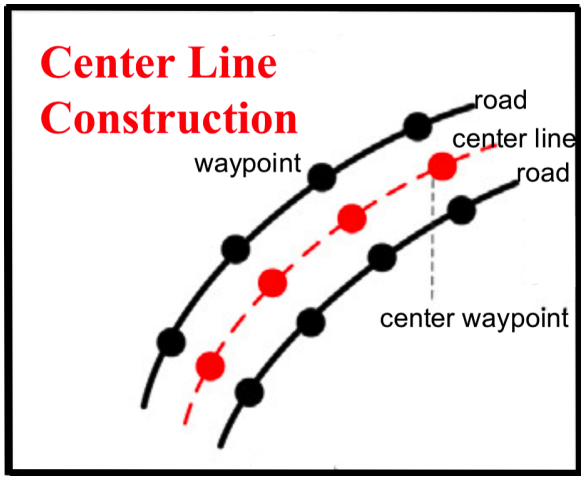


Figure 3.4. The global route and the waypoints.

perceived information. The center line of the road is constructed from the center waypoints, which are several waypoints captured from the map and aligned to the center line of the lane, using the method of cubic spline fitting. The path candidates, which are also described by the cubic spline, are generated by adjusting the lateral offset to the center line using the information for the current vehicle position, speed, and direction in a road-aligned coordinate system. During path selection, the costs of static safety, comfortability, and dynamic safety are taken into account, and are combined with information on road edges, and static and moving obstacles for selecting the optimal path. Our method provides not only the selected path, but also the appropriate speed for the vehicle maneuvering system. In this project, the proposed dynamic path planning algorithm is executed 50 times per second, and a new path is generated from the current vehicle position at every time step.

3.2.2 A road-aligned coordinate system: Frenet Coordinate

As we have mentioned several times in the previous section, a road-aligned coordinate system is needed to have a better view on the global and local trajectories. A well known

one is the Frenet coordinate, which asserts invariant tracking performance under the action of the special Euclidean group $SE(2) := SO(2) \times \mathbb{R}^2$. Here, we will apply this coordinate system in order to be able to combine different lateral and longitudinal cost functionals for different tasks as well as to mimic human-like driving behavior on the highway. As depicted in Fig. 3.5, the moving reference frame is given by the tangential and normal vector $\overrightarrow{t_r}$, $\overrightarrow{n_r}$ at a certain point of some curve referred to as the center line in the following. This center line represents either the ideal path along the free road, in the most simple case the road center, or the result of a path planning algorithm for unstructured environments[39]. Rather than formulating the trajectory generation problem directly in Cartesian Coordinates $\overrightarrow{x}$, we switch to the proposed dynamic reference frame and seek to generate a one-dimensional trajectory for both the root point $\overrightarrow{r}$ along the center line and the perpendicular offset d with the relation

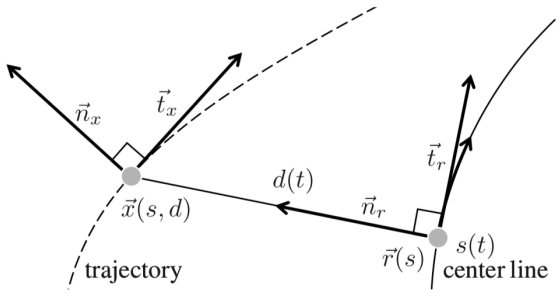


Figure 3.5. Trajectory generation in Frenet coordinate.

Frenet Coordinates. Frenet Coordinates are a way of representing position on a road in a more intuitive way than traditional (x, y) Cartesian Coordinates. With Frenet coordinates, we use the variables s and d to describe a vehicle's position on the road. The s coordinate represents distance along the road (also known as longitudinal displacement) and the d coordinate represents side-to-side position on the road (also known as

lateral displacement). Imagine a curvy road like in Fig. 3.6 with a Cartesian coordinate system laid on top of it.

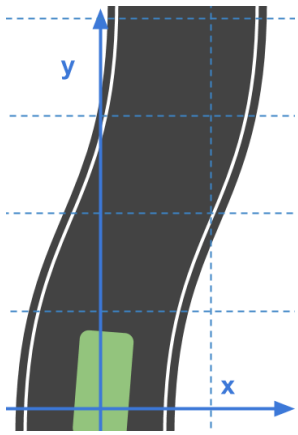


Figure 3.6. A curve road in Cartesian coordinate.

Using these Cartesian coordinates, we can try to describe the path a vehicle would normally follow on the road as shown in Fig. 3.7.

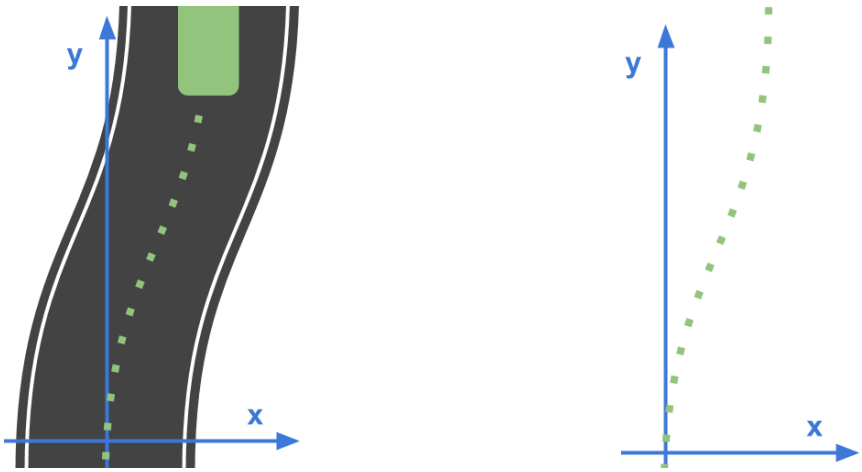


Figure 3.7. A curve road in Cartesian coordinate with waypoints.

And notice how curvy that path is. If we wanted equations to describe this motion it wouldn't be easy. Ideally, it should be mathematically easy to describe such common driving behavior. Now instead of laying down a normal Cartesian grid, we would refer to Frenet coordinate system as below in Fig. 3.8.

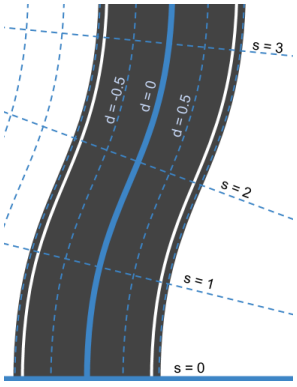


Figure 3.8. Curve in Frenet coordinate system.

Here, we've defined a new system of coordinates. At the bottom we have s=0 to represent the beginning of the segment of road we are thinking about and $d = 0$ to represent the center line of that road. To the left of the center line we have negative d and to the right d is positive. Then a typical trajectory would look like in Fig. 3.9 when presented in Frenet coordinate.

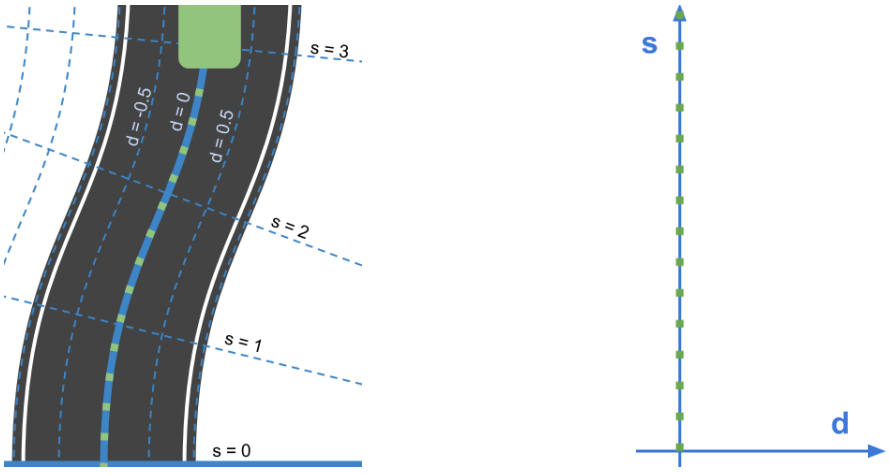


Figure 3.9. A curve road in Frenet coordinate with waypoints.

It looks straight! In fact, if this vehicle were moving at a constant speed of v_0 we could write a mathematical description of the vehicle's position as:

$$s(t) = v_0^t \tag{3.1a}$$

$$d(t) = 0 \tag{3.1b}$$

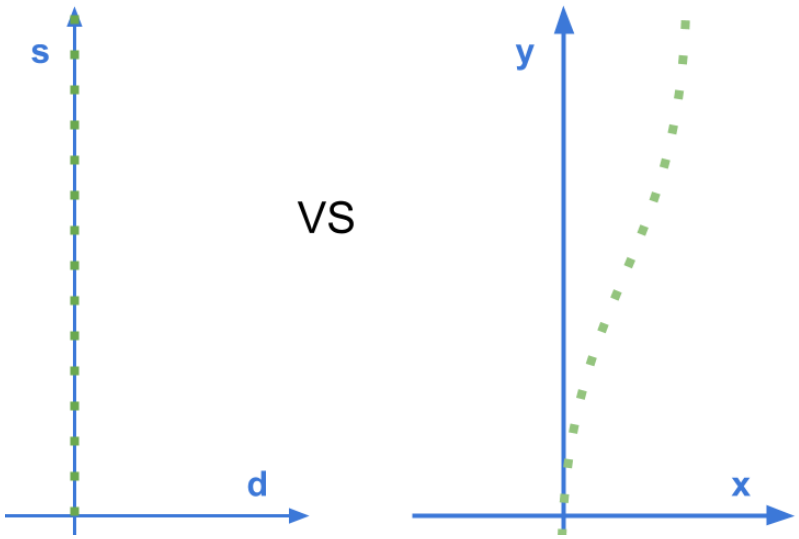


Figure 3.10. Comparison display in Frenet and Cartesian coordinate systems.

Straight lines are so much easier than curved ones.

3.2.3 Reference path generation

The path is the trajectory guiding the vehicle to follow a global route and avoid obstacles. The arc length s indicates the traveling distance on the global route, and the offset ρ can be used to measure the distance between the vehicle and the road edge.

To use the direction and curvature of the center line, it is necessary to find the position of the vehicle on the reference waypoints. We first map the vehicle position from the Cartesian coordinate system to the Frenet coordinate system, and then determine the closest point of the center line p_0, which has the minimum distance ρ_{min}. In this

paper, a method combining quadratic minimization and Newton's method is used to find p_0.

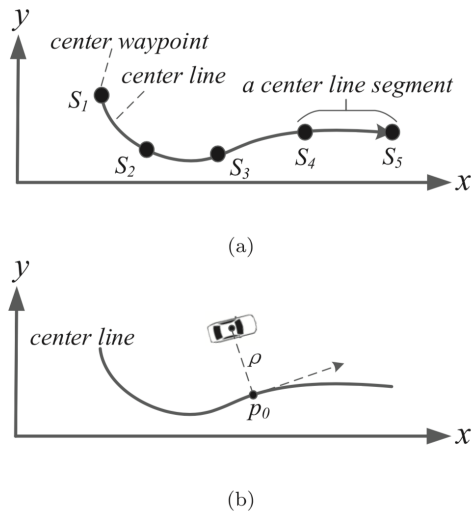


Figure 3.11. Vehicle localization on the center line. (a) Center waypoints and center line segments, (b) localization on the center line.

To generate path candidates, the curvature of each path is determined by the lateral offset d of the path, based on the curvature of the center line. As shown in Fig. 3.11 (a), p_{init} is the original point on the center line. p_{start} and p_{end} are the start and end points on the center line, respectively, for one step of planning. p_{veh} is the start point of the vehicle. p_1 to p_5 are the end points of five path candidates, and are indicated by r_1 to r_5. It is obvious that only r_2; r_4, and r_5 are available and free of obstacles. The reason for this availability lies with the differences between the offset from the path candidate to the center line and the offset from the obstacle to the center line. Meanwhile, the positions of the obstacle and the vehicle on the center line can be expressed by the arc length s.

Path candidates are generated in the Frenet coordinate system, but path planning results must be mapped into a Cartesian coordinate system to convey to the maneuvering

system. Path candidate points in the Cartesian coordinate system can be represented with respect to the arc length of the center line as Eq. (6) [43].

3.3 Path Flowing

bla

3.4 Drive By Wire

An autonomous car require that actuators that control the motion of the vehicle, can be interacted with electronically. Therefore a drive-by-wire system is needed. A drive-by-wire system replaces the mechanical systems in a traditional vehicle by using electrical/electronic (E/E) systems to perform fundamental vehicle functions.

The drive-by-wire system includes steer-by-wire, brake-by-wire and throttle-by-wire. The "by-wire" expression means that the information, from the sensor to the actuator of the different systems, is transferred electronically through wires and not by traditional hydraulic systems or mechanically through struts or shafts.

The advantage of using drive-by-wire rather than mechanical systems is that reduction of cost, moving parts and weight can be achieved. Since the steering rack can be removed, the car?s shock impact, in case of a collision, can be improved. Using an electrical based system will also increase the information flow and ease up the interconnect between different components in the car, facilitating the use of safety functions such as; ABS (anti-lock brake system), ESP (electronic stability programme), etc.

3.5 Experiment

3.5.1 Lane Keep

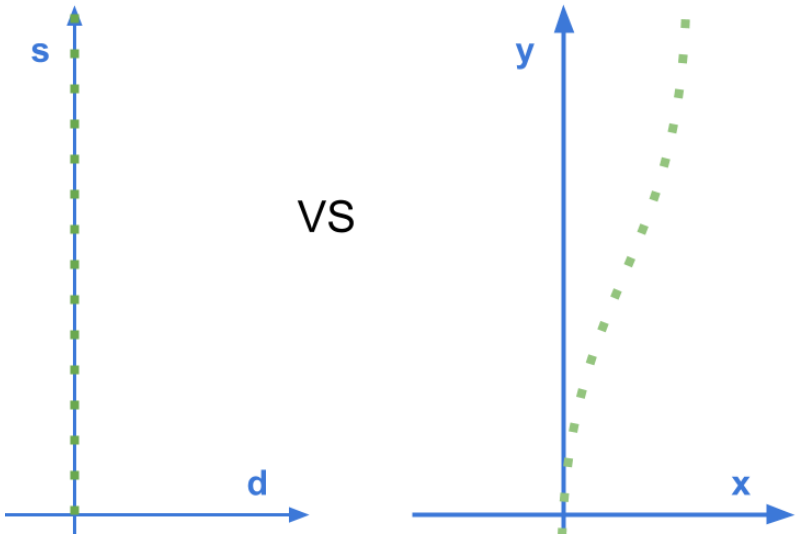


Figure 3.12. Comparison display in Frenet and Cartesian coordinate systems.

3.5.2 Lane Change

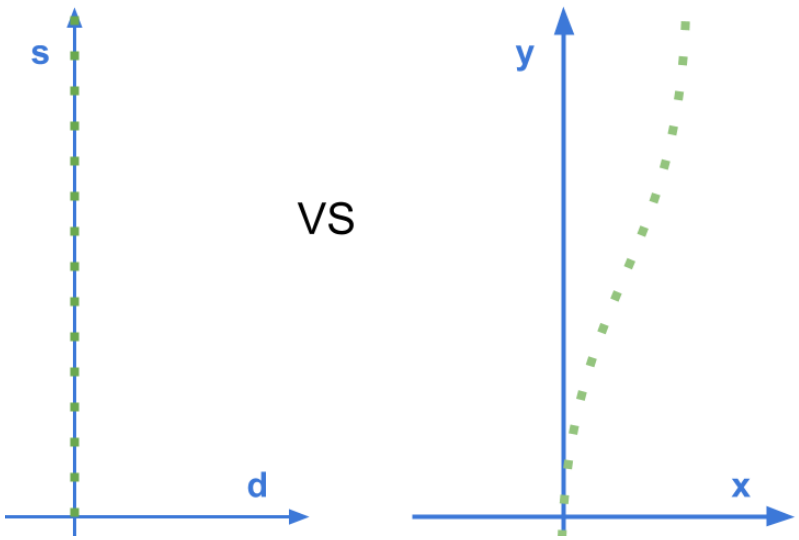


Figure 3.13. Comparison display in Frenet and Cartesian coordinate systems.

4 Deep Q-Learning

With algorithms such as Q-Learning, one can learn by choosing the actions and observing their results directly in the environment. Put into an ACC context, it is possible to learn an acting policy in a simulated highway system by taking ac- tions on the cars? brakes and throttle, and observing the results. The policy obtained can be used as a longitudinal controller to safely follow a preceding vehicle.

4.1 General Architecture

Our system follows the basic RL structure. The agent performs an action A_t given state S_t under policy π. The agent receives the state as feedback from the environment and gets the reward r_t for the action taken. The state feedback that the agent takes from sensors consists of the velocities of the neighboring vehicles $v_{veh}[]$ and the relative positions of the neighboring vehicles to the ego vehicle $dist_{veh}[]$. Possible action that agent can choose is among 4 levels of accelerations, 4 levels of decelerations and keeping the current speed. The goal of our proposed Adaptive Cruise System is to maximize the expected accumulated reward called "value function" that will be received in the future

within an episode. Using the simulations, the agent learns from interaction with environment episode-by-episode. One episode starts when the vehicle and road state information are detected. The vehicle drives on a standard circular track . If the distance between the ego vehicle and the front vehicle or the behind vehicle is less than the safety distance $dist_{safe}$, it is considered as a collision event. The episode ends if at least one of the following events occurs

- **Collision** The ego vehicle detects the distance with the vehicle in front or behind within $dist_{safe}$.
- **Time Out** The ego vehicle failed to finished N laps within a specific time.
- **Finishing** The ego vehicle successfully finished N laps within a specific time.
- **Bump** The ego vehicle is turned over for some reason.
- **Off Lane** The ego vehicle is out of lanes.

The ego vehicle continuously detect the vehicles around itself as shown in Fig 4.1. The vehicle i (i = 0,1,...,5) do not represent any specific vehicle but a detected vehicle in that area. For example, Vehicle 0 and Vehicle 1 represent vehicles in the left lane of the ego vehicle and Vehicle 0 is behind and Vehicle 1 is in front of it. When there are multiple vehicles in the same area, only the closest is remained. When there is no vehicle there, a fake and safe vehicle would be used to maintain the state structure, which would be described in more detail later.

Once one episode ends, the next episode starts with the state of environment and the value function reset.

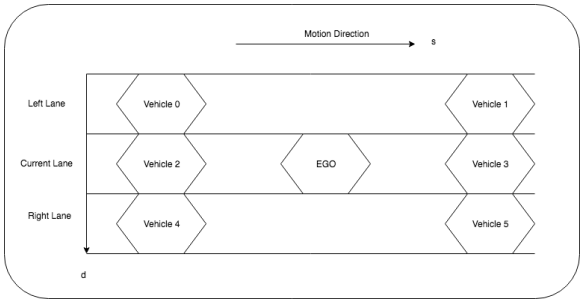


Figure 4.1. A general highway case display.

4.2 Reinforcement Learning for Longitudinal Motion

Reinforcement Learning (RL) is an interesting technique for the design of a longitudinal controller because it enables us to abstract from the complexity of car physics and dynamics that have an important computing cost. With algorithms such as Q-Learning, one can learn by choosing the actions and observing their results directly in the environment. Put into an ACC context, it is possible to learn an acting policy in a simulated highway system by taking actions on the cars? brakes and throttle, and observing the results. The policy obtained can be used as a longitudinal controller to safely follow a preceding vehicle.

To apply this RL framework, we first had to model the problem by defining the states, actions, goals and rewards. Our first approach was to use variables such as the position of a leading car and of a follower, their velocities and accelerations, etc. Clearly, this state definition put us up against the curse of dimensionality, and it became impossible to have a discrete state space precise enough to learn a valuable policy. We modified our state definition by consolidating numerous state variables. This allowed us to use a smaller discretization and to reach a better precision with only two variables. Since

driving can be seen as a sequential decision problem, there is no problem in modeling it using a MDP and discrete state variables. As seen in section 7, part of our future works will be to implement techniques to better approximate the continuous aspects of the problem. For now, our discrete state space was built around a state definition containing variables similar to those used in [14] for a fuzzy logic controller, as we defined our states by the relative distance in time between two vehicles and by the difference between those distances at two consecutive steps.

As seen in Eq. (1) and Eq. (2), the time distance takes into ac- count the relative position between the two vehicles and also the velocity of the follower, while the differences of the time distance between two consecutive time steps gives a signal about the movement of the vehicles relative to each other (whether they are closing up since last step, or getting farther). The time distance is the main variable for identifying the follower?s position related to the secure distance, while the difference in time completes the Markovian signal, as it adds to the state definition an evaluation of the relative acceleration or deceleration. This relative movement between vehicles is needed to take an informed decision on the action to take at the next time step. Those actions were taken directly on the brakes or throttle (only one action per time step is chosen), closely simulating human interaction. The actions were discretized, according to a percentage of pressure on the pedal, from 0 to 100 by increments of 20.

The goal was defined as a secure distance to reach behind a pre- ceding vehicle. That distance was specified as a time range and was defined as 2 seconds (ś 0.1 sec.), as it is a value often used as a se- cure distance in today?s ACC systems [2]. To reach the goal, we set the rewards accordingly, with a positive reward given when the ve- hicle was located in the specified time range. We also set negative rewards when wandering too far or too

close from the time ratio we were looking for. The behavior the agent was supposed to learn was to reach the secure distance specified as the goal, and to stay in that range for as long as possible.

Those elements were put together in a RL framework, and the policy obtained, learned in a simulated environment, formed the core of our longitudinal controller. The environment, a simulated highway system built in previous work, featured complex car physics and dynamics as described in [9]. Since the simulation environment was using continuous time, we had to define the time interval at which action decisions would be taken. The action chosen at the specified time frame would be taken for the whole frame. To observe an accurate behavior of the vehicle, we had to set the time step between each action decision to a small value (50 milliseconds). But in such conditions, the observation of real vehicle acceleration needed many consecutive acceleration actions, a behavior that could not be learned in a decent time with normal state space exploration. To overcome this problem, we had to use a heuristic to speed up learning. The heuristic specified that every time the car was behind the desired time ratio, the best acceleration action known from experience was taken. By ignoring in that case the braking actions, this action selection technique directed rapidly the agent towards more rewarding locations of the state space.

Put into context, Figure 3 shows that using RL simplifies the de- sign of a longitudinal controller. The closed-loop controller takes as inputs the vehicle?s state as described earlier, and selects the appropriate action according to the policy that was learned. Such a technique is obviously simpler than the complex mathematical analysis needed to predict precise car physics and dynamics for act- ing, as our controller basically hides in a black box vehicle physics and dynamics. It is possible for the agent to learn the optimal

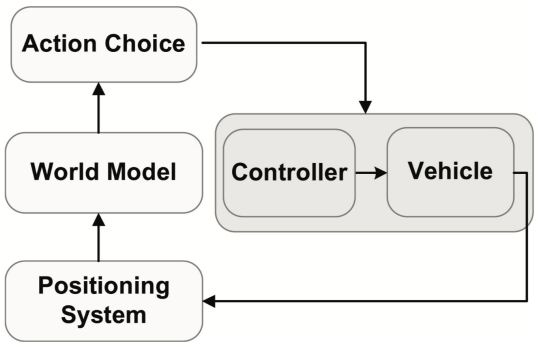


Figure 4.2. Reinforcement applied on ACC system.

behavior by taking driving actions and observing their results on the time distance and its difference between two time steps. In the next section, we will show results obtained by using this policy for longitudinal vehicle control. As drivers spend a great amount of their time in heavy traffic, such systems could reduce the risk of rear-end collisions and protect the drivers mentally by relieving them from stressful driving. (Source: [67])

4.3 Reinforcement Learning for Lateral Motion

Lateral motion control without longitudinal motion control hardly exists. Besides parking another good example of low speed combined longitudinal and lateral control is the traffic jam assist system. At speeds between zero and 40 or 60 km/h (depending on OEMs), the traffic jam assist system keeps pace with the traffic flow and helps to steer the car within certain constraints. It also accelerates and brakes autonomously. The system is based on the functionality of the adaptive cruise control with stop & go, extended by adding the lateral control of steering and lane guidance. The function is based on the built-in radar sensors, a wide-angle video camera and the ultrasonic sensors of the parking system.

In this section, we describe the design of the Management layer and, more precisely, the design of the policy to select the most efficient and safest lane for each vehicle according to their current state and action.

Lane changes are stressful maneuvers for drivers, particularly during high-speed traffic flows.

4.4 Q Learning

Q-learning is one of the popular RL methods which searches for the optimal policy in an iterative fashion. Basically, the Q-value function $q_\pi(s,a)$ is defined as

$$q_\pi(s,a) = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | S_t = s, A_t = a \right] \tag{4.1}$$

For the given state s and action a, where r_t is the reward received at the time step t. The Q-value function is the expected sum of the future rewards which indicates how good the action a is given the state s under the policy of the agent π. The contribution to the Q-value function decays exponentially with the discounting factor π for the rewards with far-off future. For the given Q-value function, the greedy policy is obtained as

$$\pi(s) = arg\max_a q_\pi(s,a) \tag{4.2}$$

One can show that for the policy in Eq. 4.2, the following Bellman equation should hold,

$$q_\pi(s,a) = \mathbb{E}_\pi[r_{t+1} + \gamma \max_a q_\pi(S_{t+1},a) | S_t = s, A_t = a] \tag{4.3}$$

In practice, since it is hard to obtain the exact value of $q_\pi(s,a)$ satisfying the Bellman equation, the Q-learning method uses the following update rule for the given one step backups S_t, A_t, r_{t+1}, S_{t+1};

$$q_\pi(S_t,A_t) \leftarrow q_\pi(S_t,A_t) + \alpha\left[r_{t+1} + \gamma \max_a q_\pi(S_{t+1},a) - q_\pi(S_t,A_t)\right] \tag{4.4}$$

However, when the state space is continuous, it is impossible to find the optimal value of the state-action pair $q_\pi(s,a)$ for all possible states. To deal with this problem, the DQN method was proposed, which approximates the state-action value function q(s, a) using the DNN, i.e., $q(s,a) = q_\theta(s,a)$ where θ is the parameter of the DNN. The parameter θ of the DNN is then optimized to minimize the squared value of the temporal difference error δ_t

$$\delta_t = r_{t+1} + \gamma \max_{a'} q_\theta(S_{t+1},a') - q_\theta(S_t,A_t) \tag{4.5}$$

For better convergence of the DQN, instead of estimating both $q(S_t,A_t)$ and $q(S_{t+1},a')$ in Eq. (4.5), we approximate $q(St,At)$ and $q(S_{t+1},a')$ using the Q-network and the target network parameterized by θ and θ', respectively. The update of the target network parameter θ', is done by cloning Q-network parameter θ, periodically. Thus, Eq. 4.5 becomes

$$\delta_t = r_{t+1} + \gamma \max_{a'} q_{\theta'}(S_{t+1},a') - q_\theta(S_t,A_t) \tag{4.6}$$

To speed up convergence further, replay memory is adopted to store a bunch of one step backups and use a part of them chosen randomly from the memory by batch size. The backups in the batch is used to calculate the loss function L which is given by

$$L = \sum_{t \in B_{replay}} \delta_t^2 \tag{4.7}$$

where B_{replay} is the backups in the batch selected from replay memory. Note that the optimization of parameter θ for minimizing the loss L is done through the stochastic gradient decent method.

One of the most basic and popular methods to estimate action-value functions is the Q-learning algorithm. It is model-free online off-policy algorithm, whose main strength is that it is able to compare the expected utility of the available actions without requiring a model of the environment. Q-learning works by learning an action-value function that gives the expected utility of taking a given action in a given state and following a fixed policy thereafter.

A value function estimates what is good for an agent over the long run. It estimates the expected outcome from any given state, by summarizing the total amount of reward that an agent can expect to accumulate into a single number. Value functions are defined for particular policies.

The state value function (or V-function), is the expected return when starting in state s and following policy π thereafter[40],

$$V^{\pi}(s) = \mathbb{E}_{\pi}\left[R_t | s_t = s\right] \tag{4.8}$$

The action value function (or Q-function), is the expected return after selecting action a in state s and then following policy π,

$$q^{\pi}(s,a) = \mathbb{E}_{\pi}\left[R_t | s_t = s, a_t = a\right] \tag{4.9}$$

The optimal value function is the unique value function that maximizes the value of every state, or state-action pair,

$$Q^*(s,a) \quad = \quad \max_{\pi} Q^{\pi}(s,a), \forall s \in \mathscr{S}, a \in \mathscr{A} \tag{4.10}$$

An optimal policy $\pi^*(s,a)$ is a policy that maximizes the action value function from every state in the MDP,

$$\pi^*(s,a) = \underset{\pi}{\operatorname{argmax}} Q^{\pi}(s,a) \tag{4.11}$$

The update rule uses action-values and a built-in max-operator over the action-values of the next state in order to update $Q(s_t,a_t)$ as follows,

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha \left[r_{t+1} + \gamma \max_{a} Q(s_{t+1},a) - Q(s_t,a_t) \right] \tag{4.12}$$

The agent makes a step in the environment from state s_t to s_{t+1} using action a_t while receiving reward r_t. The update takes place on the action-value a_t in the state s_t from which this action was executed. This version of Q-learning works well for tasks with a small a state-space, since it uses arrays or tables with one entry for each state-action pair.

In this project the policy is using the ϵ-**greedy** policy:

- ϵ-**greedy.** Selects the best action for a proportion $1-\epsilon$ of the trials, and another action is randomly selected (with uniform probability) for a proportion,

$$\pi_{\epsilon}(s) = \begin{cases} \pi_{\text{rand}}(s,a) & \text{if } rand() < \epsilon \\ \pi_{\text{greedy}}(s,a) & \text{otherwise} \end{cases} \tag{4.13}$$

  where $\epsilon \in [0,1]$ and $rand()$ returns a random number from a uniform distribution $\in [0,1]$.

4.5 Policy Representation

A policy is a mapping between a state space S and an action space A, i.e., $\pi(s) : S \rightarrow A$. For our framework, S is a continuous space that describes the state of the ego vehicle and neighboring vehicles. The action space A is represented by a 27 sized 1D discrete space where each action specifies a behavior the ego vehicle could do. The following sections provide further details about the policy representation.

4.5.1 State

A state s consists of features describing the state of the ego vehicle and relative positions and velocities with its neighboring vehicles. The state is represented by its pose q and velocity q?, where q records the positions of the center of mass of each link with respect to the root and q? records the center of mass velocity of each link. The terrain features, T , consist of a 1D array of samples from the terrain height- field, beginning at the position of the root and spanning 10 m ahead. All heights are expressed relative to the height of the terrain immediately below the root of the character. The samples are spaced 5 cm apart, for a total of 200 height samples. Combined, the final state representation is 283-dimensional. Figure 5 and 6 illustrate the character and terrain features.

4.5.2 Actions

A total of 27 controller parameters serve to define the available policy actions. These include specifications of the target spine curvature as well as the target joint angles for the shoulder, elbow, hip,

4.5.3 Reward Function

Unlike video games, the reward should be appropriately defined by a system designer in Adaptive Cruise System. As mentioned, the reward function determines the behavior of the adaptive cruise. Hence, in order to ensure the reliability of the adaptive cruise control, it is crucial to use the properly defined reward function. In our model, there is conflict between two intuitive objectives for cruise control; 1) collision should be avoided no matter what happens and 2) the vehicle should get out of the risky situation quickly. If it is unbalanced, the agent becomes either too conservative or reckless. Therefore, we should use the reward function which balances two conflicting objectives. Taking this into consideration, we propose the following reward function

$$r_t = \alpha * vel_{ego} + \beta * (s_{ego} - s_{behind}) + r \tag{4.14}$$

where v_t is the velocity of the vehicle at the time step t, decel is difference between v_t and v_{t1} and 1(x = y) has a value of 1 if the statement inside is true and 0 otherwise. The first term ?(?(pedposx ? vehposx)2 + ?)decel in the reward function prevents the agent from braking too early by giving penalty proportional to squared distance between the vehicle and pedestrian. It guides the vehicle to drive without deceleration if the pedestrian is far from the vehicle. On the other hand, the term ?(?vt2 + ?)1(St = bump) indicates the penalty that the agent receives when the accident occurs. Note that this penalty is a function of the vehicle?s velocity, which reflects the severe damage to the pedestrian in case of high velocity at collision. Without such dependency on the velocity, the agent would not reduce the speed in situation when the accident is not avoidable. The constants α, β, ϕ and ψ are the weight parameters that controls the trade-off between two objectives.

4.5.4 Relay Memory

In reinforcement learning (RL), the agent observes a stream of experiences and uses each experience to update its internal beliefs. For example, an experience could be a tuple of (state, action, reward, new state), and the agent could use each experience to update its value function via TD-learning. In standard RL algorithms, an experience is immediately discarded after it?s used for an update. Recent breakthroughs in RL leveraged an important technique called experience replay (ER), in which experiences are stored in a memory buffer of certain size; when the buffer is full, oldest memories are discarded. At each step, a random batch of experiences are sampled from the buffer to update agent?s parameters. The intuition is that experience replay breaks the temporal correlations and increases both data usage and computation efficiency Lin (1992).

Combined with deep learning, experience replay has enabled impressive performances in AlphaGo Silver et al. (2016), Atari games Mnih et al. (2015), etc. Despite the apparent importance of having a memory buffer and its popularity in deep RL, relatively little is understood about how basic characteristics of the buffer, such as its size, affect the learning dynamics and performance of the agent. In practice, a memory buffer size is determined by heuristics and then is fixed for the agent.

As mentioned in the previous section, the adaptive cruise system should learn both of the conflicting objectives. However, when we train the DQN with the reward function in Eq. 4.14, we find that the learning performance is not stable since collision events rarely happen and thus there remains only a few one-step backups associated with the collisions in the replay memory. As a result, the probability of picking such one-step backups is small and the DQN does not have enough chance to learn to avoid accidents in practical learning stage. To solve this issue, we propose so called "trauma" memory

which is used to store only the one-step backups for the rare events (e.g., collision events in our scenario). While the one step backups are randomly picked from the replay memory, some fixed number of backups associated with the collision events are randomly selected from the trauma memory and used for training together. In other words, with the trauma memory, the loss function L is modified to

equation ... (Autonomous Braking System via Deep Reinforcement Learning)

where B_{trauma} is the backups randomly picked from trauma memory. Trauma memory persistently reminds the agent of the memory on the accidents regardless of the current policy, thus allowing the agent to learn to maintain speed and avoid collisions reliably.

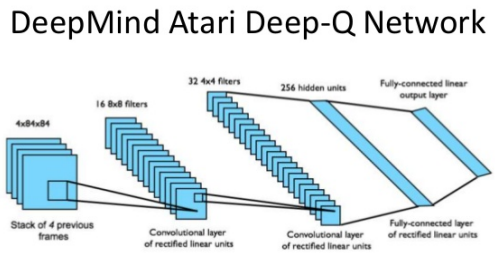
4.6 Deep Neural Network Layer

Many of the successes in DRL have been based on scaling up prior work in RL to high-dimensional problems. This is due to the learning of low-dimensional feature representations and the powerful function approximation properties of neural networks. By means of representation learning, DRL can deal efficiently with the curse of dimensionality, unlike tabular and traditional non-parametric methods [15]. For instance, convolutional neural networks (CNNs) can be used as components of RL agents, allowing them to learn directly from raw, high-dimensional visual inputs. In general, DRL is based on training deep neural networks to approximate the optimal policy ??, and/or the optimal value functions $V^\star$, $Q^\star$ and $A^\star$.

Although there have been DRL successes with gradient free methods [37, 23, 64], the vast majority of current works rely on gradients and hence the backpropagation algorithm [162, 111]. The primary motivation is that when available, gradients provide

a strong learning signal. In reality, these gradients are estimated based on approximations, through sampling or otherwise, and as such we have to craft algorithms with useful inductive biases in order for them to be tractable.The other benefit of backpropagation is to view the optimization of the expected return as the optimization of a stochastic function [121, 46]. This function can comprise of several parts?models, policies and value functions?which can be combined in various ways. The individual parts, such as value functions, may not directly optimize the expected return, but can instead embody useful information about the RL domain. For example, using a differentiable model and policy, it is possible to forward propagate and backpropagate through entire rollouts; on the other hand, inacuracies can accumulate

Convolutional Neural Networks, or CNNs, are a special type of neural network that has a known grid-like topology. Like most other neural networks they are trained with a variant of the back-propagation algorithm. CNN's strength is pattern recognition directly from pixels of images with minimal processing. We use a convolutional network as a function mapping the preprocessed images to Q values, since the actions are highly based on what would be seen as pixel matrix.



Layer	Input	Filter size	Stride	Num filters	Activation	Output
conv1	84x84x4	8x8	4	32	ReLU	20x20x32
conv2	20x20x32	4x4	2	64	ReLU	9x9x64
conv3	9x9x64	3x3	1	64	ReLU	7x7x64
fc4	7x7x64			512	ReLU	512
fc5	512			18	Linear	18

Figure 4.3. Deep Neural Network model from DeepMind paper.

5 Results

In this chapter, we evaluate the performance of the proposed autonomous highway driving system via computer simulations. It also briefly explains and discusses some characteristics of the results, whereas a more general discussion follows in the next Chapter. As described in Chapter 4, two agents with different action spaces were investigated. Agent 1 only decided when to change lanes, whereas Agent 2 decided both the speed and when to change lanes.

5.1 Simulation Setup

In simulations, we used the open source software OpenAI-Gym and ROS / Gazebo which models highway environment in real time. We generated the environment in order to train the DQN by simulating the behaviors of the cars on highway. In the simulations, we assume that the positions and velocities of the ego vehicle's neighboring vehicles is detected and stored by the ego vehicle. In each episode, the initial position of ego vehicle is set to (0, -6) in the middle lane. The initial velocities of the other vehicle are set based on their lane. With different velocities, the other vehicles have chances to create several different scenarios for the DQN model to learn.

- Scenario 1: No vehicle in the ego vehicle's detecting range.

- Scenario 2: One vehicle is detected in the same lane of the ego vehicle.
- Scenario 3: One vehicle is detected but in a different lane of the ego vehicle.
- Scenario 4: Vehicles are detected in the left, right and the ego's lanes.

5.2 Training for Longitudinal Motion only

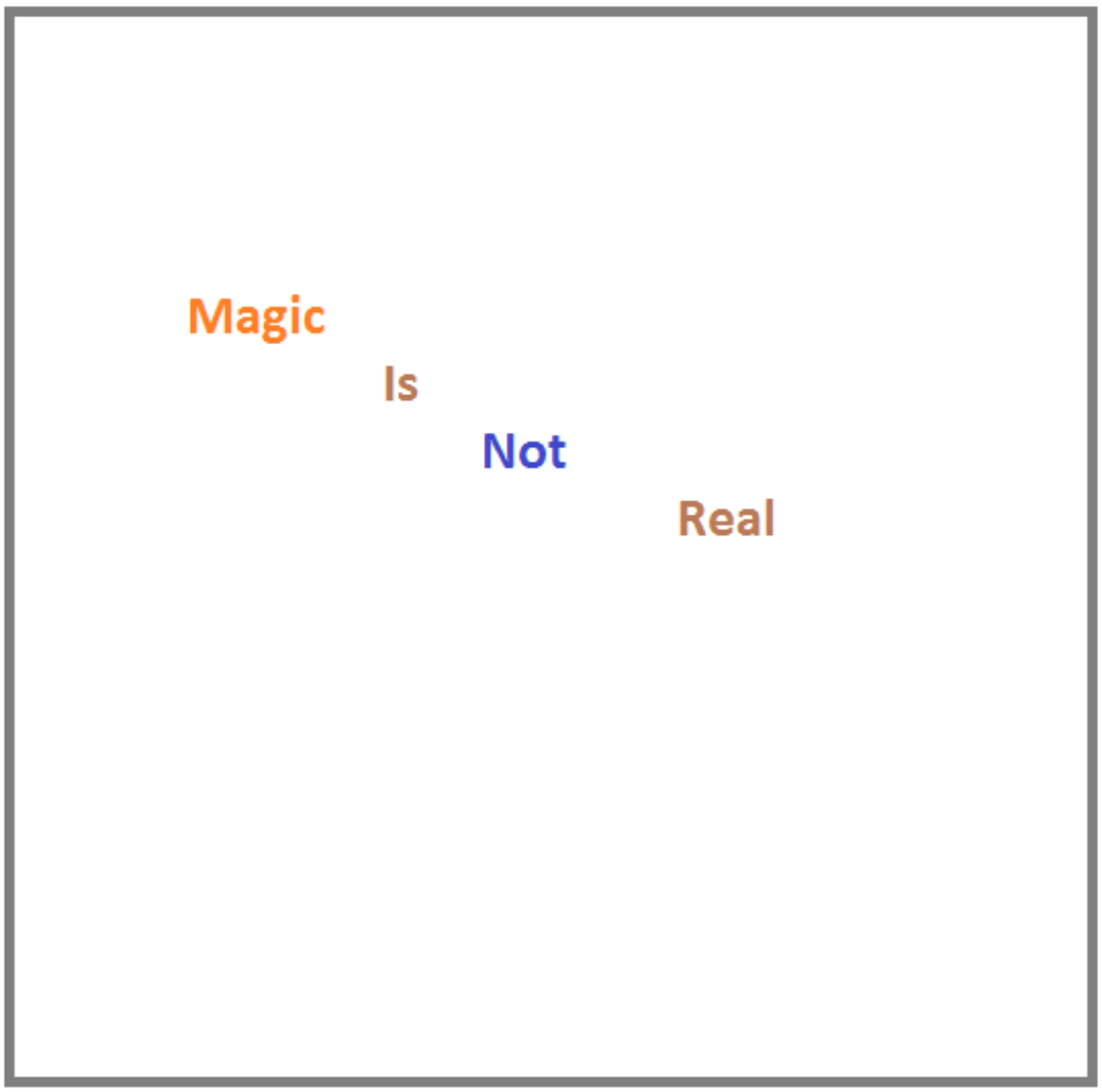


Figure 5.1. The architecture of Deep Neural Network.

The neural network used for the DQN consists of the fully-connected layers with five hidden layers. RMSProp algorithm [14] is used to minimize the loss with learning rate ? = 0.0005. The number of position data samples used as a state is set to n = 5. We set the size of the replay memory to 10,000. We set the replay batch size to 32 and trauma batch size to 10. The summary of the DQN configurations used for our experiments is provided below:

- State buffer size: n = 5
- Network architecture: fully-connected feed-forward network
- Nonlinear function: leaky ReLU [13]
- Number of nodes for each layers : [17 (Input layer), 100, 70, 50, 70, 100, 27 (Output layer)]
- RMSProp optimizer with learning rate 0.0005 [14]
- Replay memory size: 10,000
- Replay batch size: 32

5.2.1 Case 1: The neighboring vehicles are at constant speeds.

The parameters are set as below,

- Cruise speeds in Lane 0, 1and 2 are $vel_{lane0}, vel_{lane1}, vel_{lane2} = 12, 10, 8m/s$ which means Lane 0, 1 and 2 are Fast Lane, Medium Lane and Slow Lane.
- Safety distance is $dist_{safety} = 5m$
- $accel_{high}, accel_{low}, accel_{zero}, decel_{low}, decel_{high} = 2, 1, 0, -1, -2m/s^2$

Fig. 5.2 provides the plot of the total accumulated rewards i.e., value function achieved for each episode. We observe that the value function converges after 750 episodes and high total reward is steadily attained after convergence.

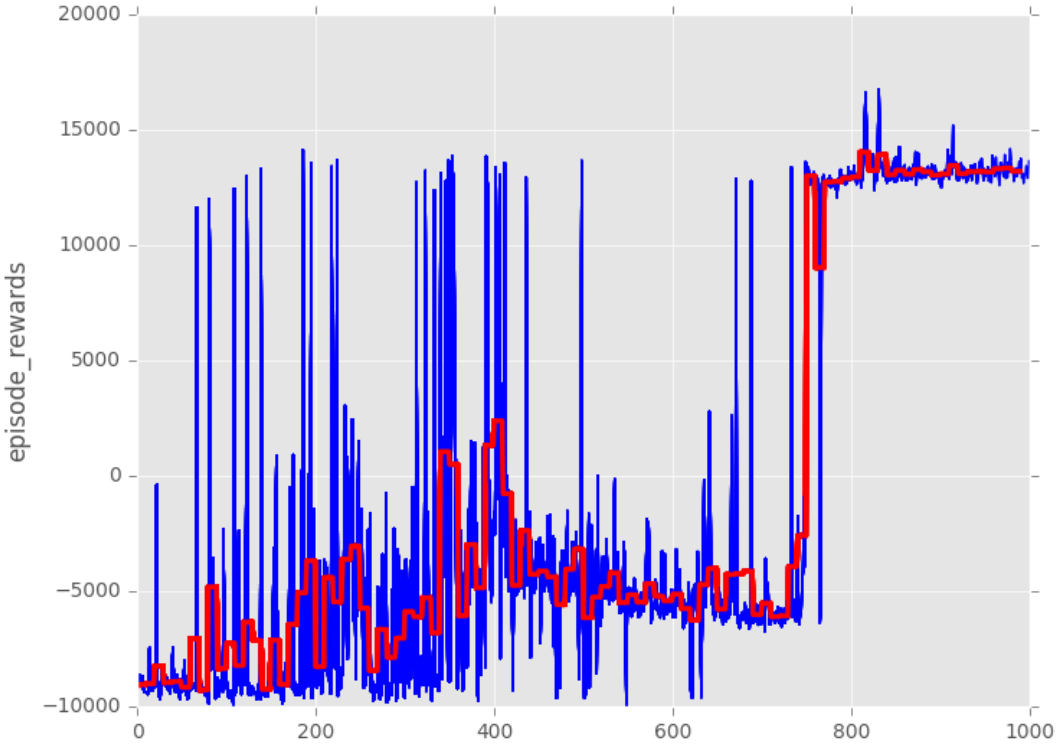


Figure 5.2. The reward history of training for Adaptive Cruise Control in Case 1.

5.2.2 Case 2: The neighboring vehicles are at erratic speeds.

The parameters are set as below,

- The speed of the vehicle except for the ego vehicle in each lane would follow a sine curve, a period of zero value and a period of costant non-zero value as shown in Fig. 5.3.
- Safety distance is $dist_{safety} = 5m$
- $accel_{high}, accel_{low}, accel_{zero}, decel_{low}, decelhigh = 2, 1, 0, -1, -2m/s^2$

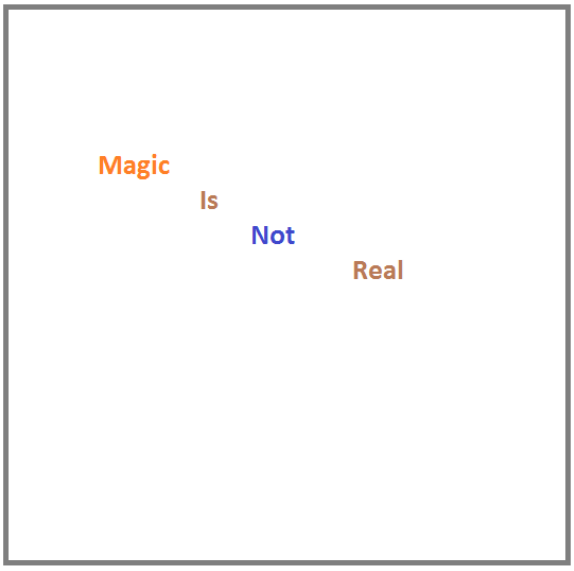


Figure 5.3. The speed variance of vehicle in each lane.

Fig. 5.4 provides the plot of the total accumulated rewards i.e., value function achieved for each episode. We observe that the value function converges after 1,000 episodes and high total reward is steadily attained after convergence.

5.3 Training for Combined Motion

The neural network used for the DQN consists of the fully-connected layers with five hidden layers. RMSProp algorithm [14] is used to minimize the loss with learning rate ? = 0.0005. The number of position data samples used as a state is set to n = 5. We set the size of the replay memory to 10,000. We set the replay batch size to 32 and trauma batch size to 10. The summary of the DQN configurations used for our experiments is provided below:

- State buffer size: n = 5
- Network architecture: fully-connected feed-forward network

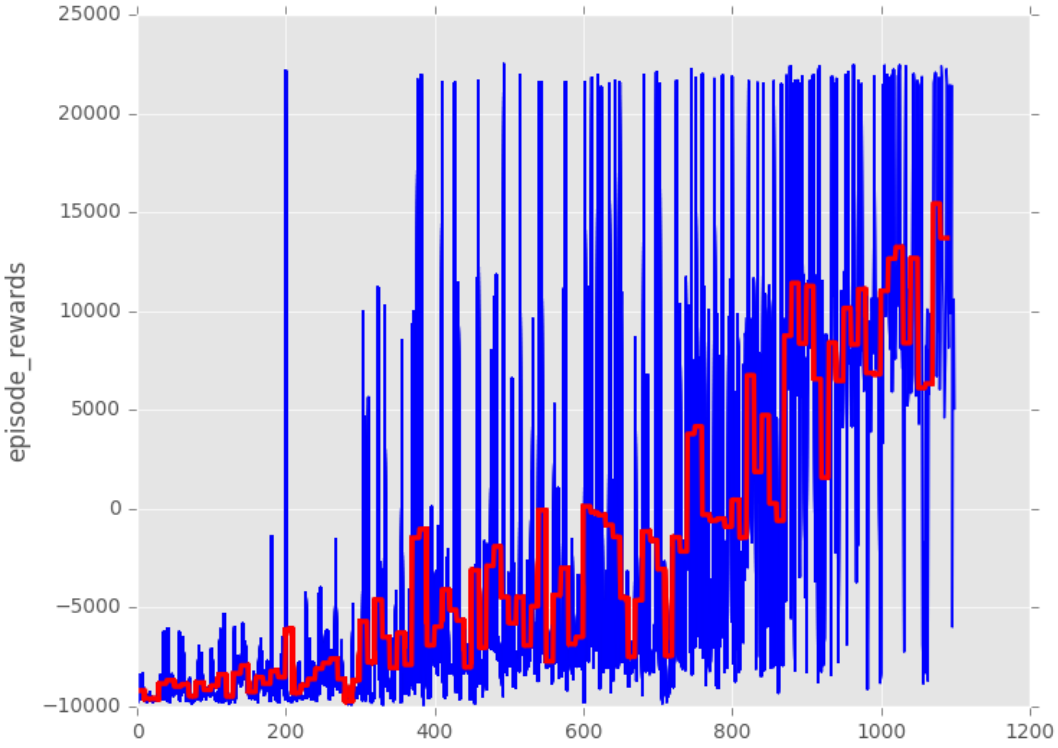


Figure 5.4. The reward history of training for Adaptive Cruise Control in Case 2.

- Nonlinear function: leaky ReLU [13]
- Number of nodes for each layers : [17 (Input layer), 100, 70, 50, 70, 100, 27 (Output layer)]
- RMSProp optimizer with learning rate 0.0005 [14]
- Replay memory size: 10,000
- Replay batch size: 32

The parameters are set as below,

- Cruise speeds in Lane 0, 1and 2 are $vel_{lane0}, vel_{lane1}, vel_{lane2} = 12, 10, 8 m/s$ which means Lane 0, 1 and 2 are Fast Lane, Slow Lane.

- Safety distance is $dist_{safety} = 5m$
- $accel_{high}, accel_{low}, accel_{zero}, decel_{low}, decelhigh = 2, 1, 0, -1, -2m/s^2$

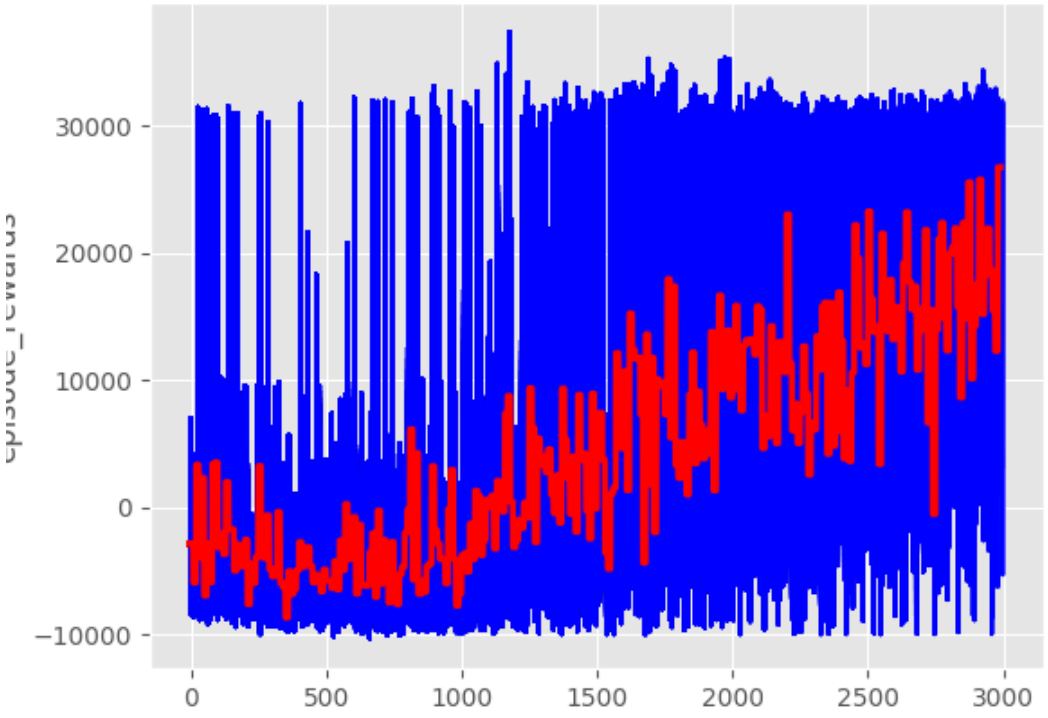


Figure 5.5. The reward history of training for full autonomous highway driving.

Fig. 5.5 provides the plot of the total accumulated rewards i.e., value function achieved for each episode. We observe that the value function converges after 2,000 episodes and high total reward is steadily attained after convergence.

Both Agent1FCNN and Agent2FCNN failed to complete all the evaluation episodes without collisions, see Fig. 4 and Table VI. Naturally, Agent1FCNN solved a significantly higher fraction of the episodes and performed better than Agent2FCNN, since it only needed to decide when to change lanes, and not control the speed. In the beginning,

it learned to always stay in its lane, and thereby solved all episodes without collisions, but reached a lower performance index than the reference model, see Fig. 5. With more training, it started to change lanes and performed reasonably well, but sometimes caused collisions. Agent2FCNN performed significantly worse and collided in 14% of the episodes by the end of its training. A longer training run was carried out for Agent1FCNN and Agent2FCNN, but after 20 million iterations, the results were the same.

5.4 Main Evaluation

We compare our results with the best performing methods from the RL literature [3, 4]. The method labeled Sarsa used the Sarsa algorithm to learn linear policies on several different feature sets hand-engineered for the Atari task and we report the score for the best performing feature set [3]. Contingency used the same basic approach as Sarsa but augmented the feature sets with a learned representation of the parts of the screen that are under the agent?s control [4]. Note that both of these methods incorporate significant prior knowledge about the visual problem by using background sub- traction and treating each of the 128 colors as a separate channel. Since many of the Atari games use one distinct color for each type of object, treating each color as a separate channel can be similar to producing a separate binary map encoding the presence of each object type. In contrast, our agents only receive the raw RGB screenshots as input and must learn to detect objects on their own.

6 Conclusions

6.1 Free-Form Visualization

A Youtube video has also been uploaded with the link https://youtu.be/VdVA3od4tVs here and it shows how it performs after 17000 time steps' training. It shows a capacity to stay in the road though it has some difficulty choosing right actions when blocked by the guard bars. The guard covered part of its view and the always acceleration actions made it even hard to get out of the stuck.

6.2 Analysis

As expected, using a CNN architecture resulted in a significantly better performance than a FCNN architecture, see e.g. Table VI. The reason for this is, as mentioned in Sect. II-C, that the CNN architecture creates a translational invariance of the input that describes the relative state of the different vehicles. This is reasonable, since it is desirable that the agent reacts the same way to other vehicles' behavior, independently of where they are positioned in the input vector. Furthermore, since CNNs share weights, the complexity of the network is reduced, which in itself speeds up the learning process. This way of using CNNs can be compared to how they previously were introduced and

applied to low level input, often on pixels in an image, where they provide a spatial invariance when identifying features, see e.g. [26]. The results of this paper show that it can also be beneficial to apply CNNs to high level input of interchangeable objects, such as the state description shown in Sect. II-C.

As mentioned in Sect. II-C, a simple reward function was used. Naturally, the choice of reward function strongly affects the resulting behavior. For example, when no penalty was given for a lane change, the agent found solutions where it constantly demanded lane changes in opposite directions, which made the vehicle drive in between two lanes. In this study, a simple reward function worked well, but for other cases a more careful design may be required. One way to determine a reward function that mimics human preferences is to use inverse reinforcement learning [27].

he method pre- sented in this paper requires no such hand crafted features, and instead uses the measured state, described in Table I, directly as input. Furthermore, the method in [10] achieved a similar performance when it comes to safety and average speed, but the number of necessary training episodes was between one and two orders of magnitude higher than for the method that was investigated in this study. Therefore, the new method is clearly advantageous compared to the previous one. An important remark is that when training an agent by using the method presented in this paper, the agent will only be able to solve the type of situations that it is exposed to in the simulations. It is therefore important that the design of the simulated traffic environment covers the intended case. Furthermore, when using machine learning to produce a decision making function, it is hard to guarantee functional safety. Therefore, it is common to use an underlying safety layer, which verifies the safety of a planned trajectory before it is executed by the vehicle control system, see e.g. [28].

6.3 Reflection

6.3.1 Diffculties

The most difficult aspect of this project was that is extremely hard to stabilize reinforcement learning with non-linear function approximators. There are plenty of tricks that can be used and hyper-parameters that need to be tuned to get it to work, such as exploration policy, discount factor, learning rate, number of episodes, batch size, experience pool size and initial value.

All these techniques and parameters were selected by trial and error, and no systematic grid search was done due to the high computational cost. More than once it seemed that the implementation of the algorithms and techniques was incorrect, and it turned out that the wrong parameters were being used. A "simple" change such as decreasing ϵ, or changing the neural network optimizer made big changes in the performance of the value function.

Also a huge difficulty of a reinforcement learning problem could be the time lag between the action and the reward. When training with grouped actions off of the heuristic reward function, the reward for a given action was immediate, and this network showed the best performance. The next best performance came from a network based off of grouped actions, where actions were only a few steps removed from the next reward. Our worst performance came from the networks trained to estimate actions, where actions were several dozen steps removed from the next rewards.

6.3.2 General Pipeline

In a Deep Q Network setting, there are several elements which we have to be careful to define.

- **Environment:** An environment defines what the agent interacts with. It receives states and actions and generate new states and plays a role as an online data generator.
- **State Space:** A state is the input of the Deep Q Network. In this project, the stats is an image or a pixel array. It will be trained by a Deep Neural Network and predict the next actions.
- **Action Space:** An action space could be discrete and continuous. It defines the all classes that would be generated from the Deep Q Network. A bigger action space indicates a bigger room for an agent to learn and improve but also means a much complexity to train.
- **Reward Functions:** The reward function determines in which way we would like the agent to grow. For example, we would define a bigger reward for the car to stay in the middle of the road than in the side of the road. It can be a discrete or a continuous function.
- **Deep Neural Network:** The Deep Neural Network is responsible to map the states to the Q values, which are corresponding to different actions by Q functions.
- **Fine tune the Hyperparameters:** By fine tuning the hyperparameters, we try to maximize the ability of the defined Deep Q Network. It can be subtle to modify the hyperparameter values which might change the output in different ways, like effecting the time of the convergence, the prediction accuracy, overfitting or underfitting and robustness.

7 Future Work

The main results of this paper show that a Deep Q-Network agent can be trained to make decisions in autonomous driving, without the need of any hand crafted features. The generality of the method was demonstrated by applying it to a highway environment with longitudinal motion control and combined longitudinal and lateral motion control. In both cases, the trained agents handled all episodes without collisions.

Topics for future work include to further analyze the generality of this method by applying it to other cases, such as crossings and roundabouts, and to systematically investigate the impact of different parameters and network architectures. Moreover, it would be interesting to apply prioritized experience replay [29], which is a method where important experiences are repeated more frequently during the training process. This could potentially improve and speed up the learning process.

For future work on hardware, an analysis and testing of several different brands and models of LIDAR, radar, ultrasonic, and cameras can be carried out. This will further assist in determining what actual products will be adequate for the scope of the platform to be developed, particularly what is best to outfit the base sensor suite with. Research into developing a standard platform conversion kit that could be adaptable to a variety

of autonomous base vehicles would be great for bringing research platform capability to those who want it.

For an Autonomous Driving System, technology will always be improving and the needs of the researcher will always be varying and changing. This in turn will require constant research and learning in order to keep the systems of an Autonomous Driving System up to date and functional for its users.

Besides, there are at least two aspects we can improve in the next stage,

(1) **Tuned Reward Functions:** In this thesis, the learning process is highly relying on the regulation of the reward function.

(2) **Better benchmarks:** In most of this thesis we used simple benchmarks, such as playing against random agents. While testing against random is probably the first thing to test against (if you can't beat a random player your learning algorithm is not working), it would be better to find a few heuristics and better players that can be used for testing.

(3) **Incorporate other RL techniques:** The field of RL has been advancing fast in recent years. There are a few new and old techniques that I would like to try, such as asynchronous RL, double Q-learning, prioritized experience replay and Asynchronous Actor-Critic Agents (A3C).

Bibliography

[1] R. Wang et al. Integrated optimal dynamics control of 4wd4ws electric ground vehicle with tire-road frictional coefficient estimation. Mechanical Systems and Signal Processing, 60-61:727 – 741, 2015.

[2] A. Vatavu R. Danescu S. Nedevschi. Autonomous driving in structured and unstructured environments. In Intelligent Vehicles Symposium, Tokyo, Japan, September 2006. IEEE.

[3] Chris Urmson etc. Autonomous driving in urban environments: Boss and the urban challenge. Journal of Field Robotics, 25, 2008.

[4] V. Mni et al. Human-level control through deep reinforcement learning. Nature, pages 529–533, 2015.

[5] O. Khatib. A unified approach for motion and force control of robot manipulators: The operational space formulation. IEEE Journal on Robotics and Automation, 1987.

[6] B. Boots A. Byravan and D. Fox. Learning predictive models of a depth camera and manipulator from raw execution traces. International Conference on Robotics and Automation (ICRA), 2014.

[7] M. R. Dogar and S. S. Srinivasa. A planning framework for non-prehensile manipulation under clutter and uncertainty. Autonomous Robots, 2012.

[8] K. T. Yu M. Bauza N. Fazeli and A. Rodriguez. More than a million ways to be pushed: A high-fidelity experimental data set of planar pushing. International Conference on Intelligent Robots and Systems (IROS), 2016.

[9] H. Murase and S. K. Nayar. Visual learning and recognition of 3-d objects from appearance. International Journal of Computer Vision (IJCV), 1995.

[10] A. Collet D. Berenson S. S. Srinivasa and D. Ferguson. Object recognition and full pose registration from a single image for robotic manipulation. International Conference on Robotics and Automation (ICRA), 2009.

[11] F. Endres J.Trinkle and W. Burgard. Learning the dynamics of doors for robotic manipulation. nternational Conference on Intelligent Robots and Systems (IROS), 2013.

[12] P. McLeod N. Reed and Z. Dienes. Psychophysics: How fielders arrive in time to catch the ball. Nature, 2003.

[13] D. Pomerleau. Alvinn: an autonomous land vehicle in a neural network. Neural Information Processing Systems (NIPS), 1989.

[14] R. Hadsell P. Sermanet J. Ben A. Erkan M. Scoffier K. Kavukcuoglu U. Muller and Y. LeCun. Learning long-range vision for autonomous off-road driving. Journal of Field Robotics (JFR), 2009.

[15] M. Riedmiller T. Gabel R. Hafner and S. Lange. Reinforcement learning for robot soccer. Autonomous Robots, 2009.

[16] L. Pinto and A. Gupta. Supersizing self-supervision: Learning to grasp from 50k tries and 700 robot hours. International Conference on Robotics and Automation (ICRA), 2016.

[17] S. Levine P. Pastor A. Krizhevsky and D. Quillen. Learning hand-eye coordination for robotic grasping with deep learning and large-scale data collection. International Symposium on Experimental Robotics (ISER), 2016.

[18] S. Levine C. Finn T. Darrell and P. Abbeel. End-to-end training of deep visuomotor policies. Journal of Machine Learning Research (JMLR), 2016.

[19] M. Deisenroth and C. E. Rasmussen. Pilco: A model-based and data-efficient approach to policy search. nternational Conference on Machine Learning (ICML), 2011.

[20] P. Abbeel A. Coates M. Quigley and A. Y. Ng. An application of reinforcement learning to aerobatic helicopter flight. Neural Informa- tion Processing Systems (NIPS), 2007.

[21] Y. Tassa T. Erez and E. Todorov. Synthesis and stabilization of complex behaviors through online trajectory optimization. International Conference on Intelligent Robots and Systems (IROS), 2012.

[22] I. Lenz R. Knepper and A. Saxena. Deepmpc: Learning deep latent features for model predictive control. Robotics Science and Systems (RSS), 2015.

[23] C. Unsal P. Kachroo and J. S. Simulation study of multiple intelligent vehicle control using stochastic learning automata. IEEE Transactions on Systems, Man and Cybernetics - Part A : Systems and Humans, 29(1):120–128, 1999.

[24] M. D. Pendrith. Distributed reinforcement learning for a traffic engineering application. the fourth international conference on Autonomous Agents, pages 404 – 411, 2000.

[25] R. Emery-Montermerlo. Game-theoretic control for robot teams. Technical report, Technical Report CMU-RI-TR-05-36, Robotics Institute, Carnegie Mellon University, August 2005.

[26] J. R. Kok and N. Vlassis. Proc. of the 21st int. conf. on machine learning. In R. Greiner and D. Schuurmans, editors, Sparse Cooperative Q-learning, pages 481–488, Banff, Canada, July 2004. ACM.

[27] N. Fulda and D. Ventura. Dynamic joint action perception for q-learning agents. International Conference on Machine Learning and Applications, 2003.

[28] P. Xuan V. Lesser and S. Zilberstein. Communication decisions in multi-agent cooperation: model and experiments. In the Fifth International Conference on Autonomous Agents, pages 616–623, Montreal, Canada, 2001. ACM.

[29] D. V. Pynadath and M. Tambe. The communicative multiagent team decision problem: Analyzing teamwork theories and models. Journal of AI research, 16:389–423, 2002.

[30] D. Dolgov and E. H. Durfee. Graphical models in local, asymmetric multi-agent markov decision processes. Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS-04), 2004.

[31] A. Haber, M. McGill, and C. Sammut. jmesim: An open source, multi platform robotics simulator. ICRA Workshop on Open Source Software, 01 2012.

[32] A.Y. Quigley M. Conley K. Gerkey B. P. Faust J. Foote T. Leibs J. Wheeler R. Ng. Ros: an open-source robot operating system. ICRA Workshop on Open Source Software, 2009.

[33] M. Quigley E. Berger and A. Y. Ng. Stair: Hardware and software architecture. In AAAI 2007 Robotics Workshop, Vancouver, B.C, August 2007.

[34] K. Wyobek E. Berger H. V. der Loos and K. Salisbury. Towards a personal robotics development platform: Rationale and design of an intrinsically safe personal robot. roc. of the IEEE Intl. Conf. on Robotics and Automation (ICRA), 2008.

[35] M. Buehler K. Iagnemma and S. Singh. The DARPA Urban Challenge — Autonomous Vehicles in City Traffic. Springer, Berlin, Germany, 2009.

[36] J. Levinson J. Askeland J. Becker J. Dolson D. Held S. Kammel J. Kolter D. Langer O. Pink V. Pratt M. Sokolsky G. Stanek D. Stavens A. Teichman M. Werling and S. Thrun. Towards fully autonomous driving: Systems and algorithms. Proc. IEEE Intell. Veh. Symp, pages 163–168, 2011.

[37] U. Franke N. Appenrodt C. G. Keller E. Kaus R. G. Herrtwich C. Rabe D. Pfeiffer F. Lindner F. Stein F. Erbs M. Enzweiler C. Knöppel J. Hipp M. Haueis M. Trepte C. Brenk A. Tamke M. Ghanaat M. Braun A. Joos H. Fritz H. Mock M. Hein J. Ziegler P. Bender M. Schreiber H. Lategahn T. Strauss C. Stiller T. Dang and E. Zeeb. Making bertha drive — an autonomous journey on a historic route. IEEE Intell. Transportation Syst. Mag., 6(2):8–20, 2014.

[38] A. Broggi P. Cerri S. Debattisti M. Laghi P. Medici M. Panciroli and A. Prioletti. Proud — public road urban driverless test: Architecture and results. Proc. IEEE Intell. Veh. Symp, pages 648–654, 2014.

[39] J. Ziegler M. Werling and J. Schroder. Navigating car-like robots in unstructured environments using an obstacle sensitive cost function. IEEE Intelligent Vehicles Symposium, 2008.

[40] R. S. Sutton and A. G. Barto. Reinforcement Learning: An Introduction. MIT Press, 1998.