ADAPTIVE CRUISE CONTROL 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

Adaptive Cruise Control 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

Adaptive Cruise Control 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].

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

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.

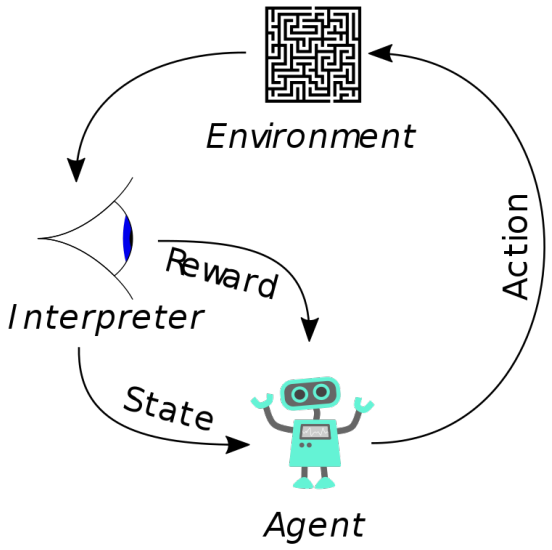


Figure 1.1. How an agent interacts with the environment.

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 Simulation

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 6 and multiplatform 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 [1], 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) 7, which is estimated as sufficiently slow physics engine [1].

Integration with ROS gives the access to a large variety of user contributed algorithms. An overview of ROS has been presented in [2].

2.1 ROS

ROS was designed to meet a specific set of challenges encountered when developing large-scale ervice robots as part of the STAIR project [2] at Stanford University1 and the Personal Robots Program [3] at Willow Garage,2 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 Nomenclature

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 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. The simplest communications are along pipelines:

2.1.2 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 unwieldly 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 dynami- cally 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.

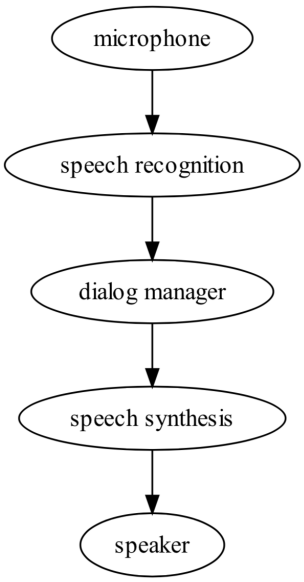


Figure 2.1. Gazebo for robot simulation.

However, graphs are usually far more complex, often containing cycles and one-to-many or many-to-many connections.

Although the topic-based publish-subscribe model is a flexible communications paradigm, its ?broadcast? routing scheme is not appropriate for synchronous transactions, which can simplify the design of some nodes. In ROS, we call this a service, defined by a string name and a pair of strictly typed messages: one for the request and one for the response. This is analogous to web services, which are defined by URIs and have request and response documents of well-defined types. Note that, unlike topics, only one node can advertise a service of any particular name: there can only be one service called ?classify image?, for example, just as there can only be one web service at any given URI.

2.1.3 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. 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.

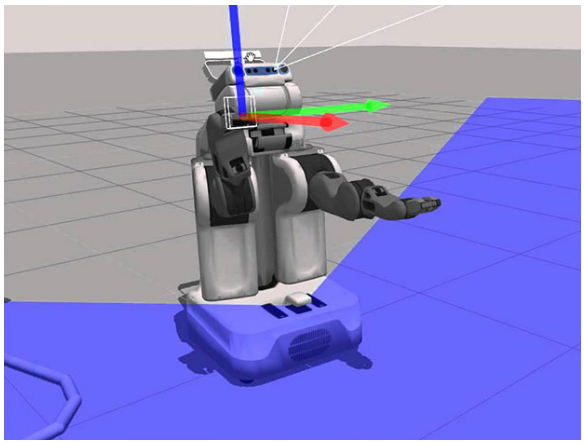


Figure 2.2. Gazebo for robot simulation.

Typical uses of Gazebo include:

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

A few key features of Gazebo include:

- 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.

Robot simulation is an essential tool in every roboticist's toolbox. A well-designed simulator makes it possible to rapidly test algorithms, design robots, perform regression testing, and train AI system using realistic scenarios. Gazebo offers the ability to accurately and efficiently simulate populations of robots in complex indoor and outdoor environments. At your fingertips is a robust physics engine, high-quality graphics, and convenient programmatic and graphical interfaces. Best of all, Gazebo is free with a vibrant community.

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.

This architecture is graphically depicted in Figure 1. The World represents the set of all models and environ- mental 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.

Physics Engine. The Open Dynamics Engine [8], created by Russel Smith is a widely used physics engine in the open source com- munity. It is designed to simulate the dynamics and kine- matics associated with articulated rigid bodies. This engine includes many features such as numerous joints, collision detection, mass and rotational functions, and many geome- tries including arbitrary triangle meshes (Figure 6). 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.

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)

[9] were chosen as the default visualization tools. OpenGL is a standard library for the creation of 2D and 3D interactive applications. It is platform indepen- dent, 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 indepen- dent.

The World. A complete environment is essentially a collection of models and sensors. The ground and buildings represent stationary models while robots and other objects are dy- namic. Sensors remain separate from the dynamic simula- tion since they only collect data, or emit data if it is an active sensor. The following is a brief description of each general component involved in the simulator. 1) Models, Bodies, and Joints: A model is any object that maintains a physical representation. 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. 2) 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. This generic sensor has been incorporated into a Sick LMS200 model to simulate a scanning laser range finder, and also into a sonar array for the Pioneer 2. Finally, the camera renders a scene using OpenGL from the perspective of the model it is attached to. Currently the camera sensor is used for both a Sony VID30 camera and the ?god?s eye? view of the world. 3) External Interfaces: Gazebo is generally used in conjunction with Player. The Player device server treats Gazebo as a normal device capable of sending and re-ceiving data. From the users point of view, the models simulated in

Gazebo are the same as their real counterparts. A second key advantage to this approach is that one can use abstract drivers inside a simulation. For example, it is possible to use Player?s VFH (Vector Field Histogram) [10] or AMCL (Adaptive Monte-Carlo Localization) [11] interchangeably between real and simulated environments. The interface to Gazebo is not limited to Player alone. The low-level library provides a mechanism for any third- party robot device server (Player or otherwise) to 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.

Construction of Models. Models are currently created by hand. The process starts with choosing the appropriate bodies and joints necessary to build an accurate model in both appearance and func- tionality. Following Figure 2, we will use the construction of the Pioneer2 AT model as an example. The entire set of bodies for this model encompass four cylinders for the wheels and a rectangular box body.

The next step attaches the bodies together using joints. The end result is a complete physical representation of our model. For the Pioneer2 AT, each wheel has a single axis of rotation. Hinge joints match this requirement perfectly. They are connected to the sides of the rectangular base and the wheels such that the axis of rotation allows for proper wheel spin.

Our Pioneer2 AT model now only lacks an interface for user control. A few functions provided by the Gazebo library resolve this issue, and allow a user to apply velocity changes to the wheels and retrieve odometric data. The Pioneer2 AT model base can

also be retro-fitted with any number of devices such as a sonar ring, gripper, and laser range finder.

2.3 Vehicle Model

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

2.3.1 URDF Models

Figure 2.3. A general highway case display.

Four URDF models representing the different vehicles supported by the Dataspeed ADAS Kit are included in the simulation. These are shown in Figure 1. The TF trees of the simulation models are all the same, and this common TF tree is shown in Figure 2.

2.3.2 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.

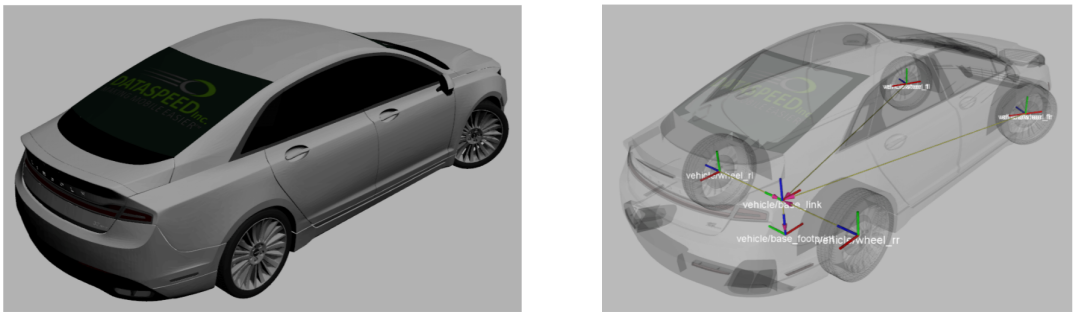


Figure 2.4. Simulation model and corresponding TF tree.

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.

2.3.3 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.

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.

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.

2.4 OpenAI-Gym

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 tasks (called environments) 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.

Alongside the software library, OpenAI Gym has a website (gym.openai.com) where one can find score- boards for all of the environments, showcasing results submitted by users. Users are encouraged to provide links to source code and detailed instructions on how to reproduce their results.

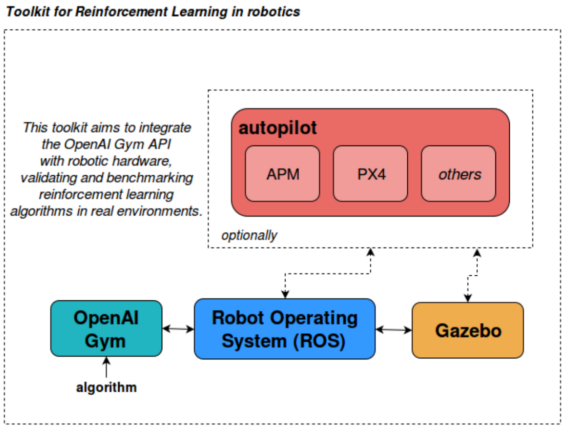


Figure 2.5. Simplified software architecture used in OpenAI Gym for robotics..

2.4.1 Architecture

The architecture consists of three main software blocks: Ope- nAI Gym, ROS and Gazebo (Figure 1). Environments developed in OpenAI Gym interact with the Robot Operating System, 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 definition1 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). The Turtlebot is encapsulated in ROS packages while robots using an autopilot like Erle-Copter and Erle-Rover are defined using the corresponding autopilot. Our policy is that every robot needs to have an interface with ROS, which will maintain an organized architecture.

Figure 1 presents the a simplified diagram of the soft- ware architecture adopted in our work. Our software structure provides similar APIs to the ones presented initially by OpenAI Gym. We added a new collection of environments called gazebo where we store our own gazebo environments with their corresponding assets. The needed installation files are stored inside the gazebo collection folder, which gives the end-user an easier to modify infrastructure.

Installation and setup consits of a ROS catkin workspace containing the ROS packages required for the robots (e.g.: Turtlebot, Erle-Rover and Erle-Copter) and optionally the appropriate autopilot that powers the logic of the robot. In this particular case we used the APM autopilot thereby the source code is also required for simulating Erle-Rover and Erle-Copter. Robots using APM stack need to use a specific plugin in order to communicate with a ROS/Gazebo simulation.

2.4.2 Environments and Robots

We have created a collection of six environments for three robots: Turtlebot, Erle-Rover and Erle-Copter. Following the design decisions of OpenAI Gym, we only provide an

abstraction for the environment, not the agent. This means each environment is an independent set of items formed mainly by a robot and a world.

Figure 5 displays an environment created with the Turtlebot robot which has been provided with a LIDAR sensor using and a world called Circuit. If we wanted to test our reinforcement learning algorithm with the Turtlebot but this time using positioning information, we would need to create a completely new environment.

The following are the initial environments and robots pro- vided by our team at Erle Robotics. Potentially, the amount of supported robots/environments will will grow over time.

Turtlebot. TurtleBot combines popular off-the-shelf robot components like the iRobot Create, Yujin Robot?s Kobuki, Microsoft?s Kinect and Asus? Xtion Pro into an integrated development platform for ROS applications. For more information, please see turtlebot.com.

Erle-Rover. A Linux-based smart car powered by the APM autopilot and with support for the Robot Operating System erlerobotics.com/blog/erle-rover .

Erle-Copter. A Linux-based drone powered by the open source APM autopilot and with support for the Robot Oper- ating System erlerobotics.com/blog/erle-copter .

2.4.3 Design Decisions

The design of OpenAI Gym is based on the authors? experience developing and comparing reinforcement learning algorithms, and our experience using previous benchmark collections. Below, we will summarize some of our design decisions. Environments, not agents. Two core concepts are the agent and the environment. We have chosen to only provide an abstraction for the environment, 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 timestep 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, we allow users to write their agents with either of these styles. Emphasize sample complexity, not just final performance. The performance of an RL algorithm on an environment can be measured along two axes: first, the final performance; second, the amount of time it takes to learn?the sample complexity. To be more specific, final performance refers to the average reward per episode, after learning is complete. Learning time can be measured in multiple ways, one simple scheme is to count the number of episodes before a threshold level of average performance is exceeded. This threshold is chosen per-environment in an ad-hoc way, for example, as 90 percent of the maximum performance achievable by a very heavily trained agent.

Both final performance and sample complexity are very interesting, however, arbitrary amounts of computation can be used to boost final performance, making it a comparison of computational resources rather than algorithm quality. Encourage peer review, not competition. The OpenAI Gym website allows users to compare the performance of their algorithms. One of its inspiration is Kaggle, which hosts a set of machine learning contests with leaderboards. However, the aim of the OpenAI Gym scoreboards is not to create a competition, but rather to stimulate the sharing of code and ideas, and to be a meaningful benchmark for assessing different methods. RL presents new challenges for benchmarking. In the supervised learning setting, performance is measured

by prediction accuracy on a test set, where the correct outputs are hidden from contestants. In RL, it?s less straightforward to measure generalization performance, except by running the users? code on a collection of unseen environments, which would be computationally expensive. Without a hidden test set, one must check that an algorithm did not ?overfit? on the problems it was tested on (for example, through parameter tuning).

We would like to encourage a peer review process for interpreting results submitted by users. Thus, OpenAI Gym asks users to create a Writeup describing their algorithm, parameters used, and linking to code. Writeups should allow other users to reproduce the results. With the source code available, it is possible to make a nuanced judgement about whether the algorithm ?overfit? to the task at hand. Strict versioning for environments. If an environment changes, results before and after the change would be incomparable. To avoid this problem, we guarantee than any changes to an environment will be accompanied by an increase in version number. For example, the initial version of the CartPole task is named Cartpole-v0, and if its functionality changes, the name will be updated to Cartpole-v1.

Example Use.

3 System Design

Current autonomous vehicles use the same architecture as the Urban DARPA Challenge vehicles did [1]?[4]. This architecture comprises three main processing modules, de- scribed below and illustrated in Fig. 1.

? 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 combines two of the methods presented in the previous section: learning-by-demonstration, which is able to generate commands which feel natural to the passengers, and predictive control, which relies on model-based predictions to make decisions, and provides safety and stability. Our framework inherits the

advantages of both. It is illustrated in Fig. 2, and the differences with the architecture in Fig. 1 are explained below.

? The trajectory generator is replaced by a driver model. The driver model uses learning-by-demonstration: it is trained using real driving data and can generate commands which imitate the driving style of the driver it learned from. In addition to the reference command, the motion planner out- puts a confidence value representing how reliable the reference command is. The confidence will be higher for driving situations which are similar to the training dataset.

? The trajectory controller is replaced by a model predictive controller which takes as an input the reference command generated by the driver model, and the associated confidence value. The role of the controller is to guarantee safety while trying to match the reference command sent by the driver model. When the confidence value is high and the safety constraints are respected, the command computed by the controller will match the reference command. The controller's command will deviate from the reference command when the confidence value is low or when a safety constraint violation is anticipated.

The driver model and the model predictive controller can be designed independently, and one 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 re- placed, without having to readjust any other module. One could also imagine extending the architecture in Fig. 2 to take advan- tage 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. Thus far, we have implemented and tested it

for the longitudinal control of an autonomous car during lane keeping. In this scenario, the commanded input is the acceleration of the vehicle. The problem is defined formally below.

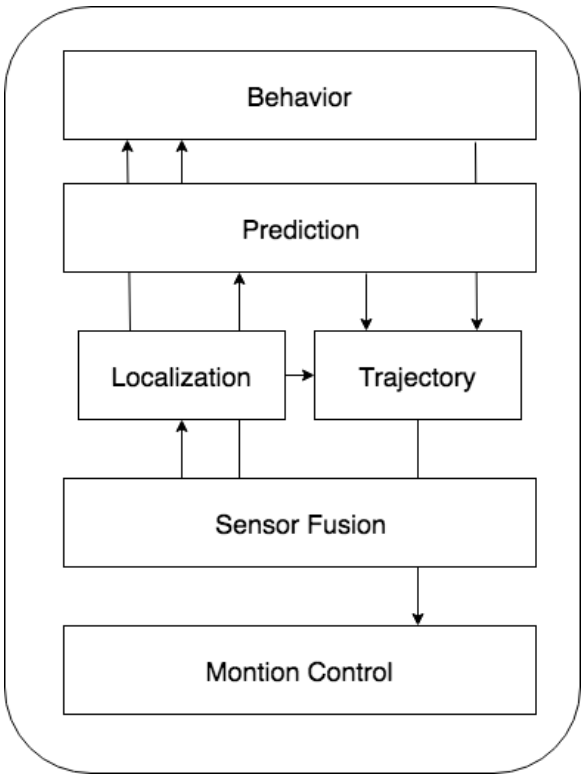


Figure 3.1. Standard Architecture of Autonomous Vehicle Control.

3.1 Behavior Planner

3.1.1 States for Autonomous Vehicles

3.2 Trajectory Generation

The proposed path planning method 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. As shown in Fig. 1, the map is composed of a set of way- points on the road edges and topology that describes the relationships between connected roads. The process by which the map is obtained falls outside the scope of this paper; therefore, the maps used in this paper are predefined in our simulations.

Fig. 1 shows the proposed dynamic path planning method, which includes three stages: center line construction, path candidate generation, and path selection. These are performed on the basis of perceived information and the proposed algorithms. As shown in Fig. 1, the center line of the road is constructed from the center waypoints, which are the center posi- tions of a pair of waypoints, 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 posi- tion, speed, and direction in the s q 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 and acceleration for the vehicle maneuvering system. In this study, the proposed dynamic path planning algorithm is executed 15 times per second, and a new path is generated from the current vehicle position at every time step.

A well known approach in tracking control theory is the Frenet Frame method, which asserts invariant tracking performance under the action of the special Euclidean group $SE(2) := SO(2)\text{Œ}R2$. Here, we will apply this method 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. 2, the moving reference

frame is given by the tangential and normal vector ?tr , ?nr 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 [20]. Rather than formulating the trajectory generation problem directly in Cartesian Coordinates ?x, we switch to the pro- posed dynamic reference frame and seek to generate a one- dimensional trajectory for both the root point ?r along the center line and the perpendicular offset d with the relation

3.2.1 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). Why do we use Frenet coordinates? Imagine a curvy road like the one below with a Cartesian coordinate system laid on top of it...

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

And notice how curvy that path is! If we wanted equations to describe this motion it wouldn't be easy! x(t)=? y(t)=? Ideally, it should be mathematically easy to describe such common driving behavior. But how do we do that? One way is to use a new coordinate system. Now instead of laying down a normal Cartesian grid, we do something like you see below...

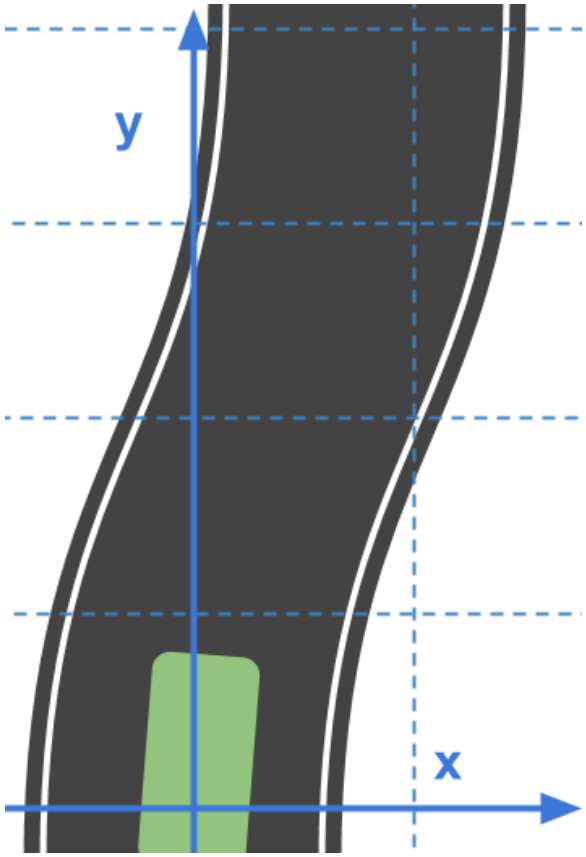


Figure 3.2. Cropped input image.

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. So what does a typical trajectory look like when presented in Frenet coordinates?

It looks straight! In fact, if this vehicle were moving at a constant speed of v0? we could write a mathematical description of the vehicle's position as: s(t)=v0?t d(t)=0

...straight lines are so much easier than curved ones.

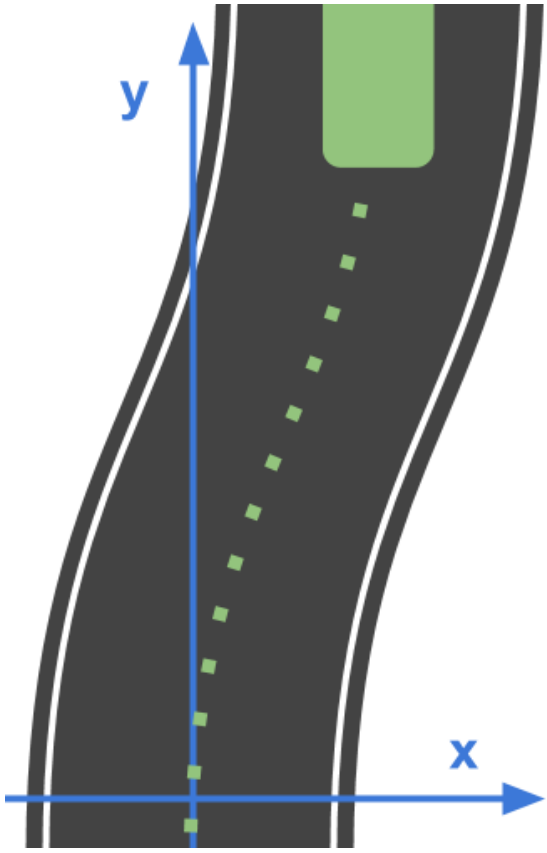


Figure 3.3. Cropped input image.

3.2.2 Path candidates 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 q can be used to measure the distance between the vehicle and an obstacle. Consequently, the proposed path planning method for avoiding obstacles is designed in the s q coordinate system rather than the Cartesian coordinate system.

To use the direction and curvature of the center line, it is necessary to find the position of the vehicle on the center line, as shown in Fig. 2(b). We first map the vehicle position from the Cartesian coordinate system to the s q coordinate system, and then determine the closest point of the center line p0, which has the minimum distance q. In

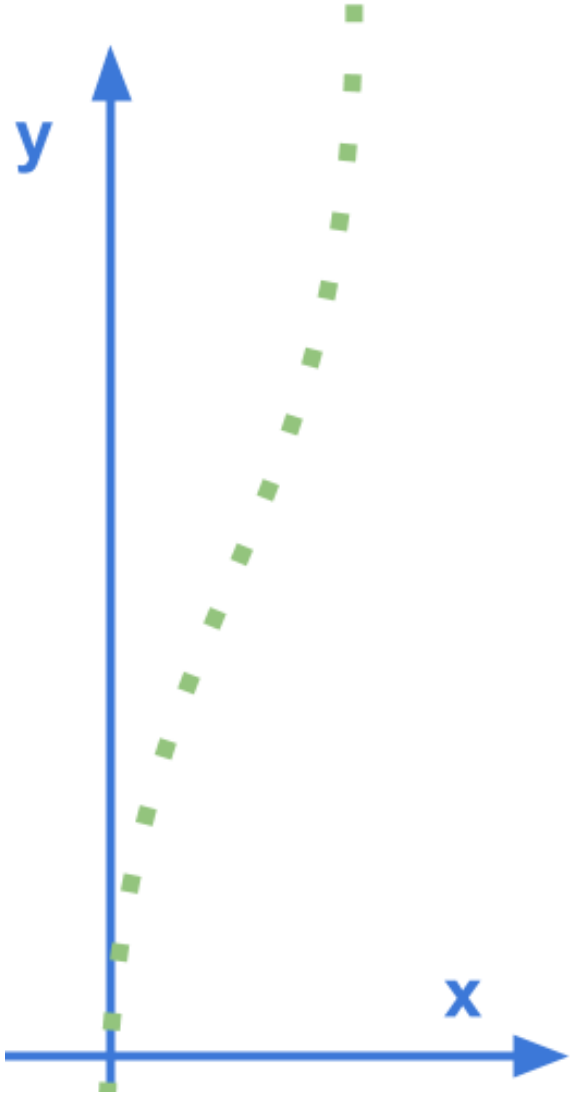


Figure 3.4. Cropped input image.

this paper, a method combining quadratic minimization and Newton?s method is used to find p0 [41].

3.2.3 Path candidates generation in the s q coordinate system

To generate path candidates, the curvature of each path is determined by the lateral offset q of the path, based on the curvature of the center line. As shown in Fig. 3(a), Pinit is the original point on the center line. Pstart and Pend are the start and end points

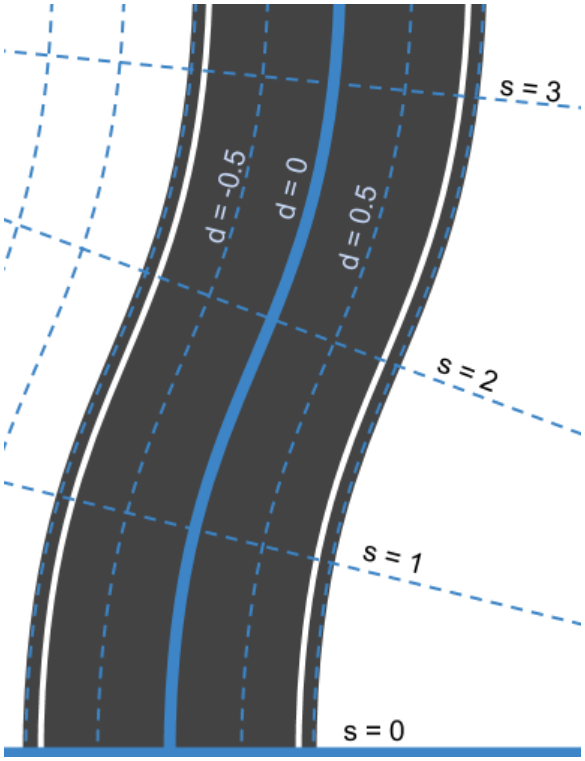


Figure 3.5. Cropped input image.

on the center line, respectively, for one step of planning. Pveh is the start point of the vehicle. P1 to P5 are the end points of five path candidates, and are indicated by r1 to r5. It is obvious that only r2;r4, and r5 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. In this case, it is necessary to design the algorithm to avoid obstacles using the parameters s and q. As shown in (4), a function describing the relationship between the arc length s and offset q is designed to provide a continuous change in the offset.

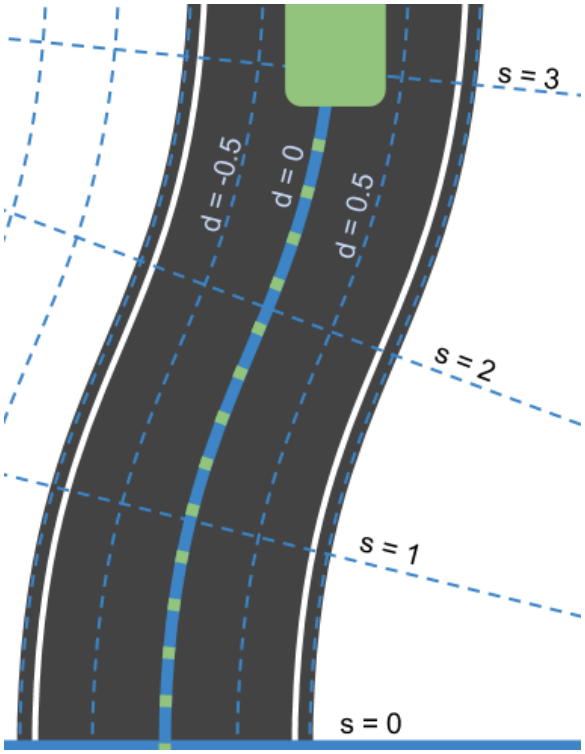


Figure 3.6. Cropped input image.

3.2.4 Coordinates conversion of path candidate points

Path candidates are generated in the s q 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 (6) [43].

3.3 Montrol Control

3.3.1 Waypoint Follower Node

The waypoint follower node uses the $/final_waypoints$ data to publish $/twist_cmd$ for the dbw node to use. The node implementation uses an open source implementation of pure pursuit algorithm from Autoware. (https://github.com/CPFL/Autoware)

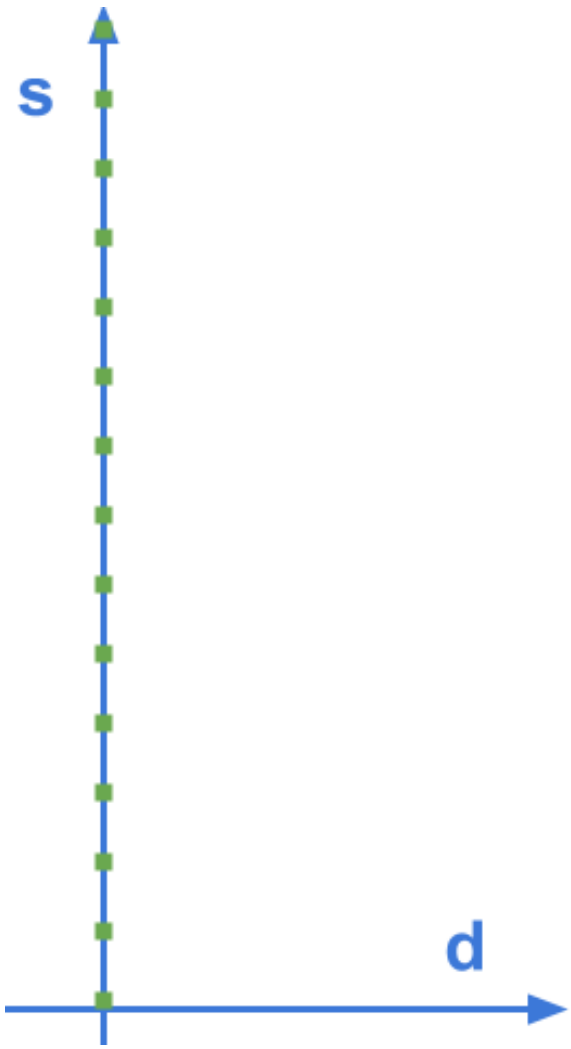


Figure 3.7. Cropped input image.

3.3.2 Drive By Wire (DBW) Node

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

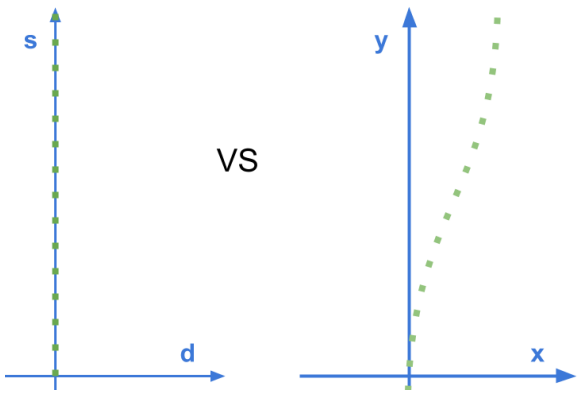


Figure 3.8. Cropped input image.

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.

The purpose of the drive by wire node is to publish commands to the vehicle actuators: steering wheel, accelerator, and break. The node subscribes to the following nodes:

$/current_velocity$

$/twist_cmd$

$/vehicle/dbw_enabled$

It uses information from $current_velocity$ and $twist_cmd$ to determine the values to be sent to the actuators, while $/vehicle/dbw_enabled$ topic is only used to determine if drive-by-wire is being overridden, in which point it will stop publishing any message, and ignore the received messages.

The node delegates the actuator's value to the $twist_controller$ class, which returns a value for each of the actuators.

Twist Controller. The twist controller is initialized with values based on the vehicle configuration, as steer ratio and vehicle mass, and implements a single method control, which receives the vehicle target position and current velocity, and will return a value for acceleration, braking, and steering. It will in turn, delegate the calculation for each of those values to the $throttle_controller$, $brake_controller$, and $yaw_controller$. The $yaw_controller$ calculates a steering angle for every update, while the twist controller will calculate the position error (current - target) to determine if braking or acceleration should be engaged. If the error is positive, the throttle controller is used, while a negative error will be sent to the brake PID.

1) Throttle Controller ($throttle_controller.py$) The throttle is initialized with a min and max acceleration values. It uses a PID controller to determine the amount of acceleration or decelaration to be given based on the difference between the target velocity and the current velocity.

2) Braking Controller ($braking_controller.py$) The brake controller calculates the amount of torque to be sent to the brake by multiplying the vehicle mass, wheel radius and aceleration.

3) Steering Controller ($yaw_controller.py$) The steering controller calculates the amount of steering it should send to the actuator using the target linear and angular velocity, taking into account the steer ratio of the vehicle.

4 Deep Reinforcement Learning

4.1 Structure of DRL

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.

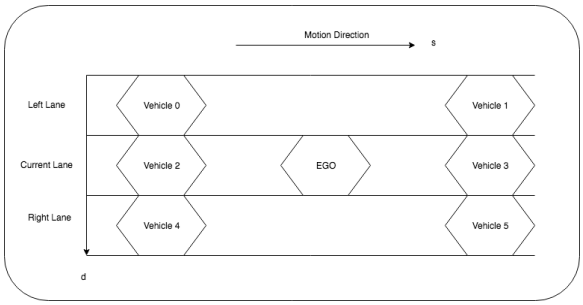


Figure 4.1. A general highway case display.

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

4.2 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[31],

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

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 $\in [0,1]$.

4.3 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.

4.3.1 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.

4.3.2 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,

4.3.3 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.

4.3.4 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)

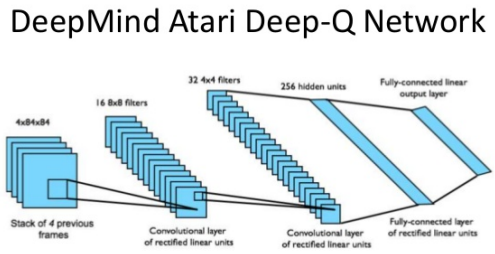
where B_{trauma} is the backups randomly picked from trauma memory. Trauma memory persistently reminds the agent of the memory on the accidents regardless of the current policy, thus allowing the agent to learn to maintain speed and avoid collisions reliably.

4.4 Deep Neural Network

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 4.2. Deep Neural Network model from DeepMind paper.

5 Results

In this chapter, we evaluate the performance of the proposed adaptive cruise system via computer simulations.

5.1 Simulation Setup

In simulations, we used the commercial software PreScan which models vehicle dynamics in real time [15]. We generated the environment in order to train the DQN by simulating the random behavior of the pedestrian. In the simulations, we assume that the relative location of the pedestrian is provided to the agent. To make the system practical, we add slight measurement noise to it. In each episode, the initial position of vehicle is set to (0, 0). Time-to-collision T T C is chosen according to the uniform distrubution between 1.5 s and 4 s. The initial velocity of the vehicle is uniformly distributed between $v_{initmin} = 2.78m/s(10km/s)$ and $v_{initmax} = 16.67m/s(60km/h)$. At the beginning of the episodes, the position of the pedestrian is fixed to 5 ? v_{init} meters away from the position of the vehicle. The pedestrian stands either at the far-side or at near-side of the vehicle with equal probability. The behavior of the pedestrian follows one of two scenarios below;

- Scenario 1 : Cross the road

- Scenario 2 : Stay at initial position. During training, either of two scenarios is selected with equal probability. In Scenario 1, the pedestrian starts to move when the vehicle is crossing at the ?pedestrian crossing point? ptrig = (5?TTC)?vinit. (see Fig. 4.) The safety distance l for the pedestrian is set to 3 m. The agent chooses the brake control among ahigh = ?9.8 m/s2, amid = ?5.9 m/s2, ahigh = ?2.9 m/s2 and anothing = 0 m/s2 every ?T = 0.1 second. The detailed simulation setup is summarized below.
- Initial velocity of vehicle vinit ? U (2.78, 16.67) m/s ? Velocity of pedestrian vped ? U (2, 4) m/s
- Time-to-collision T T C ? U (1.5, 4) s
- Initial pedestrian position pedposx = 5 ? vinit m
- Trigger point $p_{trig} = (5TTC)?v_{init}m$
- Safety dist = 3 m
- T=0.1s
- $a_{high}, a_{mid}, a_{low}, a_{zero} = 9.8, 5.9, 2.9, 0m/s^2$

5.2 Training of DQN

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 and that of the trauma memory to 1,000. 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
- Trauma memory size: 1,000
- Trauma batch size = 10
- Reward function: ? = 0.001, ? = 0.1, ? = 0.01, ? =100

Fig. 5 provides the plot of the total accumulated rewards i.e., value function achieved for each episode when training is conducted with and without trauma memory. We observe that with trauma memory the value function converges after 2,000 episodes and high total reward is steadily attained after convergence while without trauma memory the policy does not converge and keeps fluctuating.

5.3 Visualizing the Value Function

Safety test was conducted for several different T T C values. Collision rate is measured for 10,000 trials for each TTC value. Table I provides the collision rate for each TTC value for the test performed for Scenario 1. The agent avoids collision successfully for TTC values above 1.5s. For the cases with TTC values less than 1.5s, we observe some collisions. According to our analysis on the trajectory of braking actions, these are the cases where collision was not avoidable due to the high initial velocity of the vehicle even though full braking actions were applied. The agent passed the pedestrian without

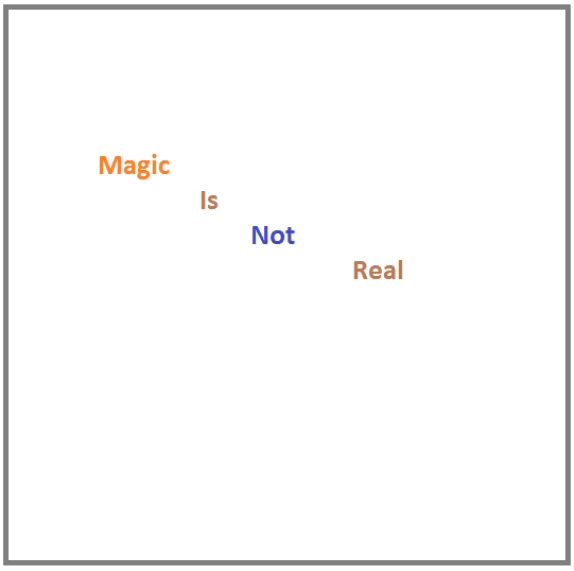


Figure 5.1. Achieved value function achieved during training.

unnecessary stop for all cases in the Scenario 2. The detailed trajectory of the brake actions for one example case is shown in Fig. 6. Fig. 6 (a) shows the trajectory of the position of the vehicle and the pedestrian recorded every 0.1 s. The velocity of the agent and the brake actions applied are shown in Fig. 6 (b) and (c), respectively. The vehicle starts to decelerate about 20 m away from the pedestrian and completely stops about 5 m ahead, thereby accomplishing collision avoidance. We observe that weak braking actions are applied in the beginning part of deceleration and then strong braking actions come as the agent gets close to the pedestrian.

Fig. 7 shows how the initial position of the pedestrian and the relative distance between the pedestrian and vehicle are distributed for 1,000 trials in the scenario 1. We see that the vehicle stops around 5 m in front of the pedestrian for most of cases. This seems to be reasonable safe braking operation considering the safety distance of l = 3 m. Note that this distance can be adjusted by changing the reward parameters. Overall,

the experimental results show that the proposed autonomous braking system exhibits consistent brake control performance for all cases considered.

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

6.0.1 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.

6.0.2 Reflection

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.

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

Given this thesis mainly covers the basic Deep Q-Learning algorithm, it leaves plenty of room for further exploration into the fancier algorithm. The admin/base layer will need to prevent user interference and act as a ?final check? to algorithm and control commands that the user level wishes to execute. The user layer will need to allow users to integrate their computers, sensors, and other hardware into the user level and allow them to control their tests. A study on sensor fusion and techniques can be carried out to determine the optimal method for continuous integration and fusion of added LIDARs, radars, and other sensors. Development of a Hardware-In-the-Loop (HIL) simulation environment can greatly help to expedite testing and verification of researchers? tests before they are carried out.

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 AVRP, 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 AVRP up to date and functional for its users.

7.1 Improvement

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 difenition of the reward function.

(2) **Better benchmarks:** In most of this thesis we used simple benchmarks, such as playing against random players. 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).

Appendix A
Pprofile

Here is a section for HHP profile stuff, whatever that is...

- Pprofile 1 stuff here
- Pprofile 2 stuff here
- Pprofile 3 stuff here
- Etc...

Appendix B
Specimens

Here is a section to describe specimens used in this research.

- Specimen 1 stuff here
- Specimen 2 stuff here
- Specimen 3 stuff here
- Etc...

Appendix C
Preparation of this document

This document was prepared using pdfLaTeX and other open source tools. The (free) programs implemented are as follows:

- LaTeX implementation:

  MiKTeX

  http://www.miktex.org/

  TeXLive

  https://www.tug.org/texlive/

  MacTeX

  https://tug.org/mactex/

- TeX-oriented editing environments:

  Vim Text Editor

  https:http://www.vim.org/

- Bibliographical:

  BibTeX

  http://www.bibtex.org/

  Zotero

  https://www.zotero.org/

Appendix D
Figures

Here is a section for figures in the document.

- Figure 1 stuff here
- Figure 2 stuff here
- Figure 3 stuff here
- Etc...

Complete References

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