A LEARNING BASED ADAPTIVE CRUISE AND LANE CONTROL SYSTEM

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

August, 2018

A Learning based Adaptive Cruise and Lane Control System

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. Francis Merat

Committee Member — Date
Department of Electrical Engineering and Computer Science

Dr. Kathryn Daltorio

Committee Member — Date
Department of Mechanical and Aerospace Engineering

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

Dedicated to my parents.

Table of Contents

List of Tables

List of Figures

Acknowledgements

It has been a long road to get to this point. Two years' working in Robotics Lab and one year's COOP training in Silicon Valley both gave me a strong foundation to go deeper in Robotics and AI. 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. Francis Merat and Dr. Kathryn Daltorio for their guidance and being a part of my committee. I also want to give an extra thank to Dr. Cenk for the teaching in Algorithmic Robotics and Robotic Manipulation which prepare me a software engineer in Robotics. The amount of knowledge in the DaVinci project 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 Supervisor, Dr. Wyatt Newman. You were an excellent mentor and navigator when it came to discussing the possible solutions especially on new problems. Your passion for teaching, emphasis on education, and advice really influenced me continuing on to being a thinker and doer.

Last, but certainly not least, thank you to my parents who are far away in China. 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

A Learning based Adaptive Cruise and Lane Control System

Abstract

by

PENG XU

The present thesis describes a study that aims at design of a learning based autonomous driving system mainly on adaptive cruise control (ACC) and lane control (LC). In this study a simulated highway environment is created for the vehicles to capture online training data. A Deep Q-Learning algorithm is proposed to train two agents respectively, an ACC agent and an integrated agent (ACC and LC). The results show behavioral adaptation with an ACC in terms of the speed of the preceding vehicle and emergence brake and LC in terms of two lanes. The learning based system has a nice adaption in different scenarios and can be reinforced by continuous training.

1 Introduction

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].

1.1 Motivation and Background

Complex functions like highly autonomous driving with combined longitudinal and lateral control will definitely appear first on highways, since traffic is more predictable and relatively safe there (one-way traffic only, quality road with relative wide lanes, side protections, good visible lane markings, no pedestrians or cyclists, etc.). As highways are

the best places to introduce hands-free driving at higher speeds, one could expect a production vehicle equipped with a temporary autopilot or in other words autonomous highway driving assist function as soon as the end of this decade.

Autonomous highway driving means the autonomous control of the complex driving tasks of highway driving, like driving at a safe speed selected by the driver, changing lanes or overtaking front vehicles depending on the traffic circumstances, automatically reducing speed as necessary or stopping the vehicle in the right most lane in case of an emergency. Japanese Toyota Motor have already demonstrated their advanced highway driving support system prototype in real traffic operation. The two vehicles communicate each other, keeping their lane and following the preceding vehicle to maintain a safety distance[4] as shown in Fig. 1.1.

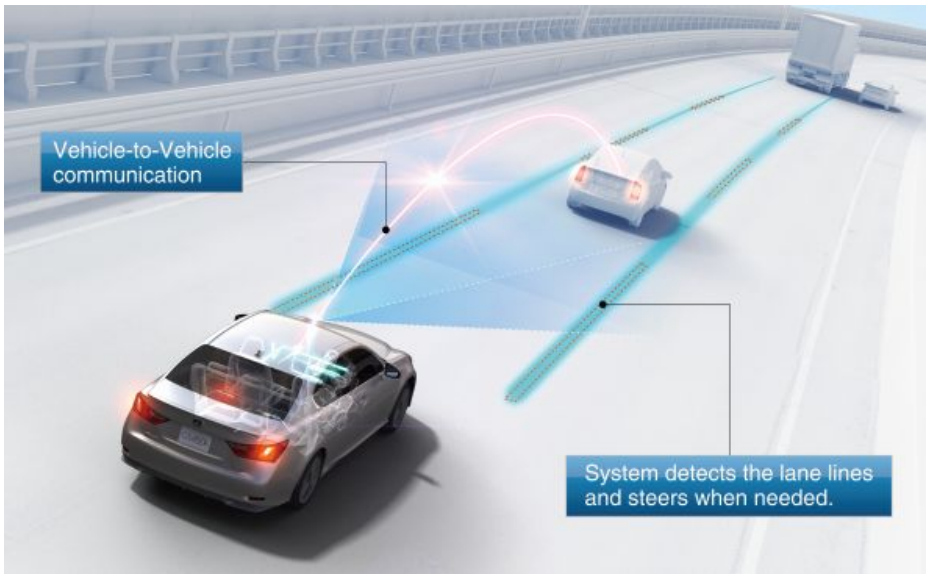

Figure 1.1. Toyota's advanced highway driving support system.

An important aspect in developing an autonomous driving system is the development of a simulator. To get to mass production, autonomous vehicles must drive 10 billion kilometers for proof of concept. With 100 vehicles driving 24 hours seven days

a week , this would still mean 225 years of driving. Simulators can accelerate that and reduce costs[5]. Both Uber and Waymo recently gave a glimpse into the workings of their simulators, with which they "drive" billions kilometer per year.

In this thesis, we are aiming for developing an autonomous driving system allowing the vehicle to smartly choose the driving behaviors, such as adjusting the speed and changing the lane. For the longitudinal motion, Adaptive cruise control (ACC), a radar-based system, is 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)[6]. As for the lateral motion, Autonomous lane control system, relying on position sensors, identifies a current lane where the vehicle is located, provides instructions to the steering system and speed control system to maneuver the vehicle in either the lane-keeping mode or the lane-changing mode[7]. 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 learning based adaptive cruise and lane control system is proposed for autonomous vehicles with the help of deep reinforcement learning.

1.2 Literation Review

The past few years have seen many breakthroughs using reinforcement learning (RL). The company DeepMind combined deep learning with reinforcement learning to achieve above-human results on a multitude of Atari games and, in March 2016, defeated Go

champion Le Sedol four games to one. Though RL is currently excelling in many game environments, it is a novel way to solve problems that require optimal decisions and efficiency, and will surely play a part in machine intelligence to come.

Google's DeepMind published its famous paper Playing Atari with Deep Reinforcement Learning[8], 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. 1.2, the robot 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.

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[9][10][11]. This approach has been applied to a range of problems. 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

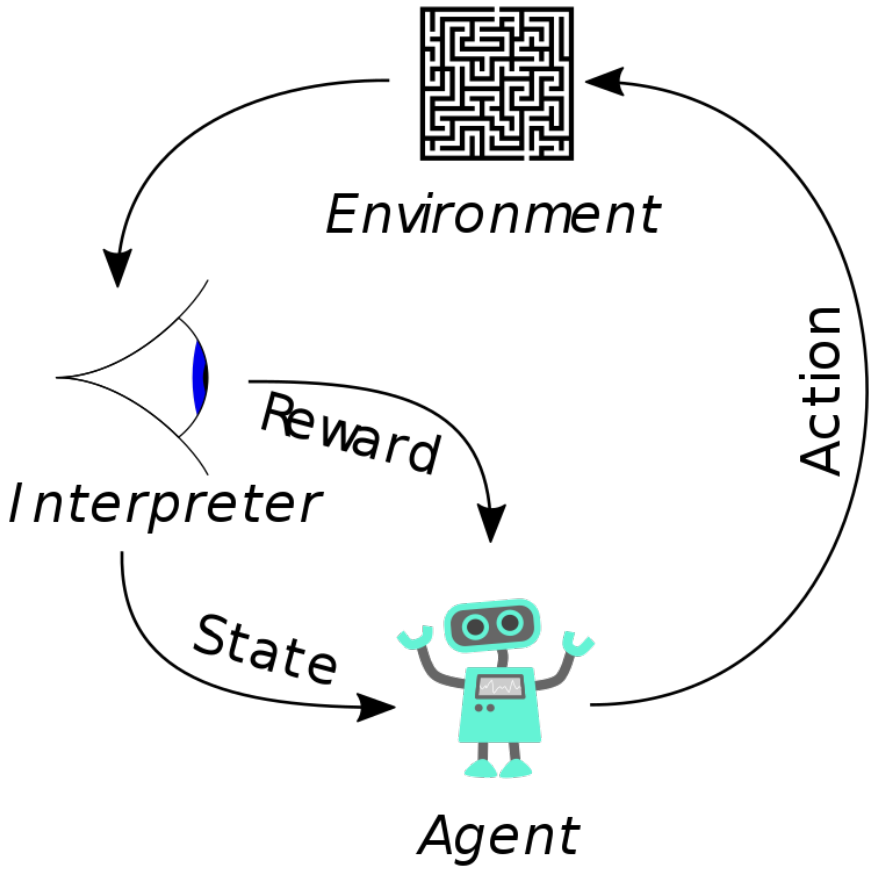

Figure 1.2. How an agent interacts with the environment.

an overcomplete representation of the physical world and complex simulation calculations. A learning based framework could 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.[10] 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 ACC 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 Cooprated-ACC 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.

Regarding the reinforcement learning in a lane control problem, Unsal, Kachroo and Bay[23] have used multiple stochastic learning automata to control the longitudinal and lateral path of a vehicle. 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 thesis. 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 state representation.

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.

1.3 Thesis Outline

The goal of this thesis is to develop a learning framework for the autonomous driving system. This section will present the chapters and subsections of this thesis and provide a brief summary of those sections.

Chapter 2 details the stack architecture of the simulating environment and the integration of the simulating tools.

- **Section 2.1 ROS:** A general introduction of Robotic Operating System (ROS).
- **Section 2.2 Gazebo:** A general introduction of Gazebo, a robot dynamic simulator.
- **Section 2.3 OpenAI-Gym:** A general introduction of OpenAI-Gym, an open source AI playground and environment builder.
- **Section 2.4 Vehicle Model:** A close-to-reality vehicle model.

Chapter 3 dives into the autonomous driving system. Chapter 3 is laid out as follows:

- **Section 3.1 Behavior Planning:** A description of the planning module on behavior selecting.
- **Section 3.2 Path Planning:** A description of the planning module on path generating and selecting.
- **Section 3.3 Path Flowing:** A description of the lane keeping or path following module.
- **Section 3.4 Drive By Wire:** A description of the drive by wire vehicle.

Chapter 4 describes the core algorithm in the framework, namely Deep Q-Learning. Chapter 4 is a follows:

- **Section 4.1 Architecture.**
- **Section 4.2 Reinforcement Learning for Longitudinal Motion.**
- **Section 4.3 Reinforcement Learning for Lateral Motion.**
- **Section 4.4 Q Learning.**
- **Section 4.5 Policy Representation.**
- **Section 4.6 Deep Neural Network.**

Chapter 5 displays the experiment results in some basic scenarios. Chapter 5 sections are as follows:

- **Section 5.1 Simulation Setup:** Explanation of how the simulation is set by parameters.
- **Section 5.2 Training for Longitudinal Motion:** An experiment on ACC agent.
- **Section 5.3 Training for Combined Motion:** An experiment on ACC and LCC agent.
- **Section 5.4 Main Evaluation:** Analysis on the results and indication of the performance.

Chapter 6 contains the conclusion. Chapter 6 sections are as follows:

- **Section 6.1 Free-Form Visualization.**
- **Section 6.2 Analysis.**
- **Section 6.2 Reflection.**

Chapter 7 indicates several promising areas for future work.

2 Simulated Environment

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

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

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

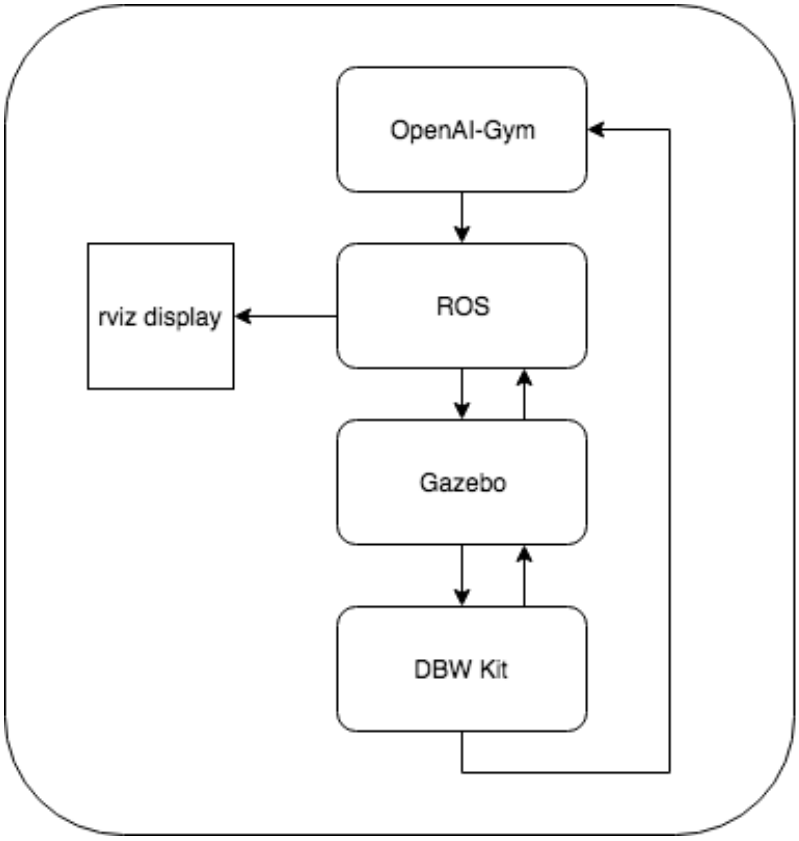

Figure 2.1. Architecture of the simulation environment.

.

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

The usual basic requirements to robot simulators are an accurate physics simulation (such as object velocity, inertia, friction, position and orientation, etc.), high quality rendering (for shape, dimensions, colors, and texture of objects), integration with the Robot Operating System (ROS) framework and multi-platform performability. It provides great opportunities for modeling robots and their sensors together with developing robot control algorithms, realizing mobile robot simulation, visualization, locomotion and navigation in a realistic 3D environment. As mentioned in the paper[27], the high graphical fidelity in a robot simulation is important because the sensory input to the robot perceptual algorithms comes from virtual sensors, which are also provided by the simulation. For example, virtual cameras use the simulator rendering engine to obtain their images. If images from a simulated camera have incorrect similarity to real camera ones, then it is not possible to use them for object recognition and localization.

To avoid such a sort of problems, we use the robust and high graphical quality robot simulator — Gazebo, which is an open source robotic simulation package that closely integrated with ROS. Gazebo uses the open source OGRE rendering engine, which produces good graphics fidelity, although it also employs the Open Dynamics Engine (ODE), which is estimated as slow physics engine[27].

2.1 ROS

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

System (ROS) framework[28] gained more popularity because of its generality and expansibility. ROS was designed to meet a specific set of challenges encountered when developing large-scale service robots as part of the STAIR project[29] at Stanford University and the Personal Robots Program[30] at Willow Garage, but the resulting architecture is far more general than the service-robot and mobile-manipulation domains.

The philosophical goals of ROS can be summarized as:

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

2.1.1 Implementation

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

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

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

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

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

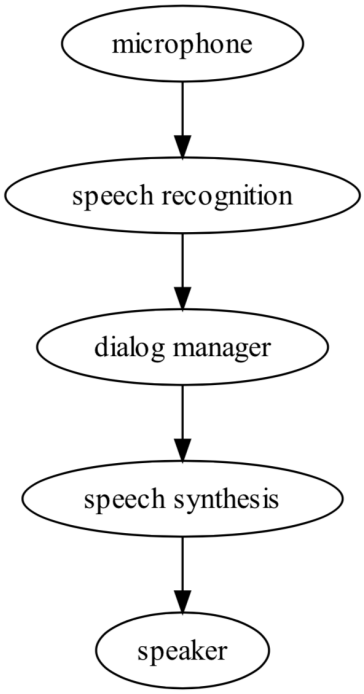

Figure 2.2. Communication pipeline.

2.1.2 Collaborative Development

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

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

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

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

2.1.3 Visualization and Monitoring

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

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

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

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

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

2.1.4 Transformations

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

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

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

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

2.2 Gazebo

Gazebo is a 3D dynamic simulator with the ability to accurately and efficiently simulate populations of robots in complex indoor and outdoor environments, which makes it possible to rapidly test algorithms, design robots, perform regression testing, and train AI system using realistic scenarios. While similar to game engines, Gazebo offers physics simulation at a much higher degree of fidelity, a suite of sensors, and interfaces for both users and programs. Fig. 2.3 gives a typical display of Gazebo and in the window is a PR2 robot[30] with its LIDAR sensor range displayed in blue.

Typical uses of Gazebo include:

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

A few key features of Gazebo include:

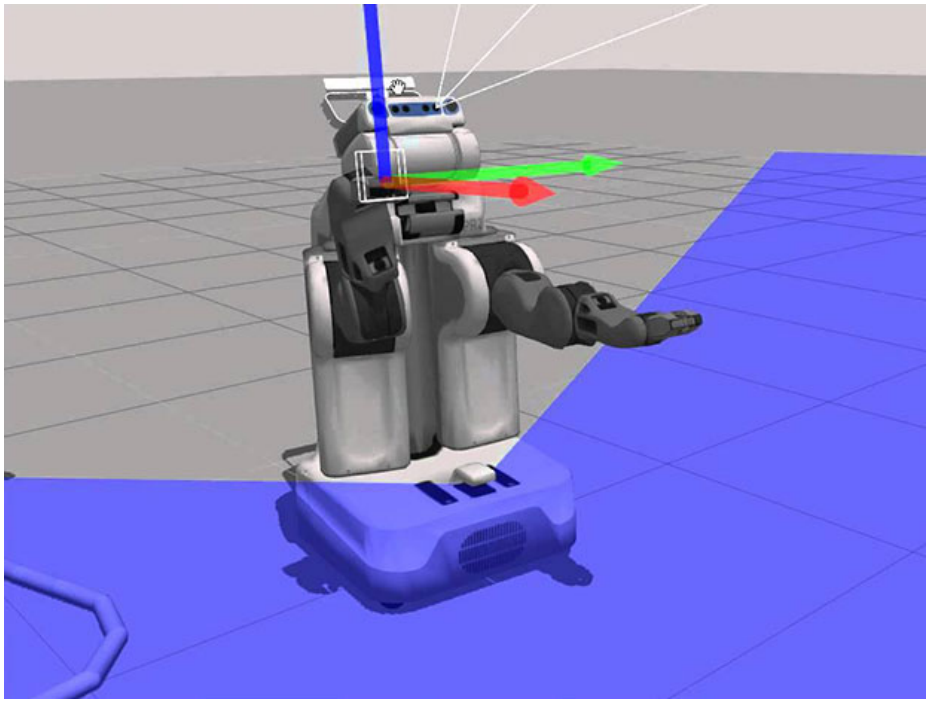

Figure 2.3. Gazebo for robot simulation.

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

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

2.2.1 Architecture

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

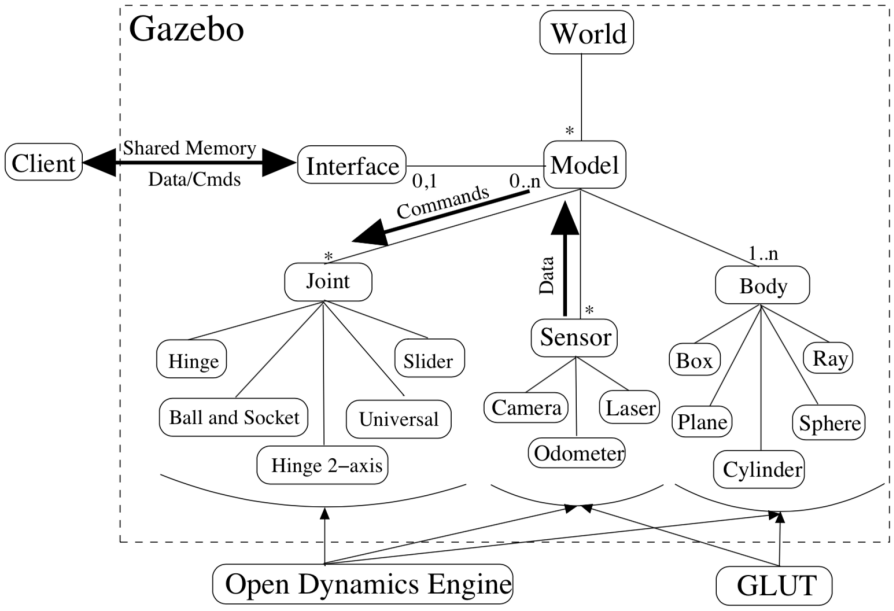

Figure 2.4. Gazebo for robot simulation.

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

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

2.2.2 Physics Engine

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

2.2.3 Visualization

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

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

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

2.2.4 Customized Environment

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

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

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

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

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

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

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

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

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

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

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

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

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

2.2.5 Test Bed for Algorithm Design

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

2.3 OpenAI-Gym

Reinforcement learning assumes that there is an agent that is situated in an environment. Each step, the agent takes an action, and it receives an observation and reward from the environment. An RL algorithm seeks to maximize some measure of the agent's total reward, as the agent interacts with the environment. In the RL literature, the environment is formalized as a partially observable Markov decision process (POMDP)[31].

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

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

2.3.1 Design For Environment

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

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

2.3.2 Interfacing with ROS and Gazebo

In the context of robotics, reinforcement learning offers a framework for the design of sophisticated and hard-to-engineer behaviors[32]. The challenge is to build a simple environment where this machine learning techniques can be validated, and later applied in a real scenario.

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

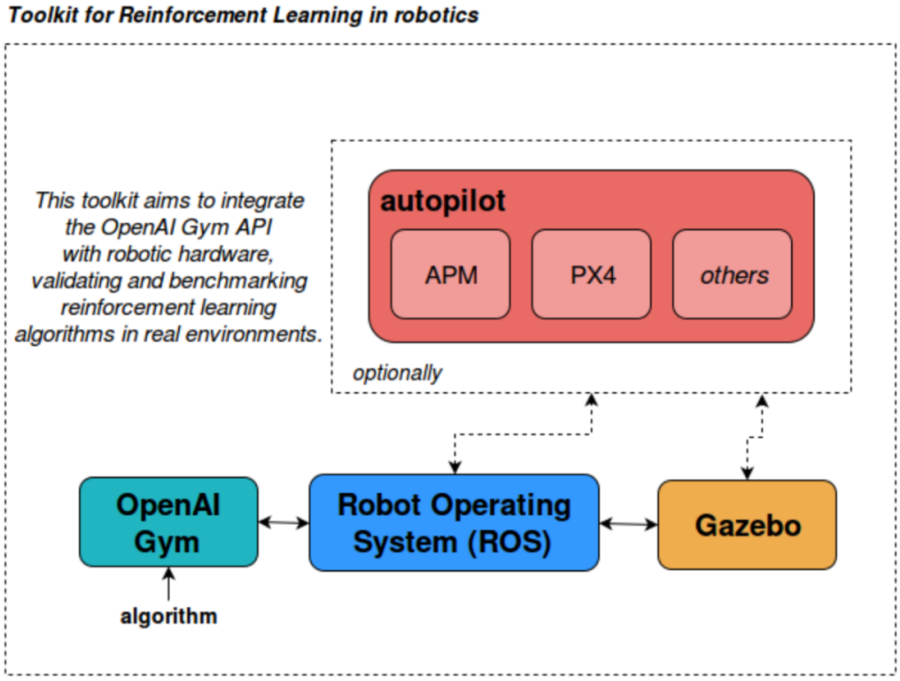

Figure 2.5. Simplified software architecture used in OpenAI Gym for robotics.

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

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

2.3.3 Example Use: Train a Cart-pole agent

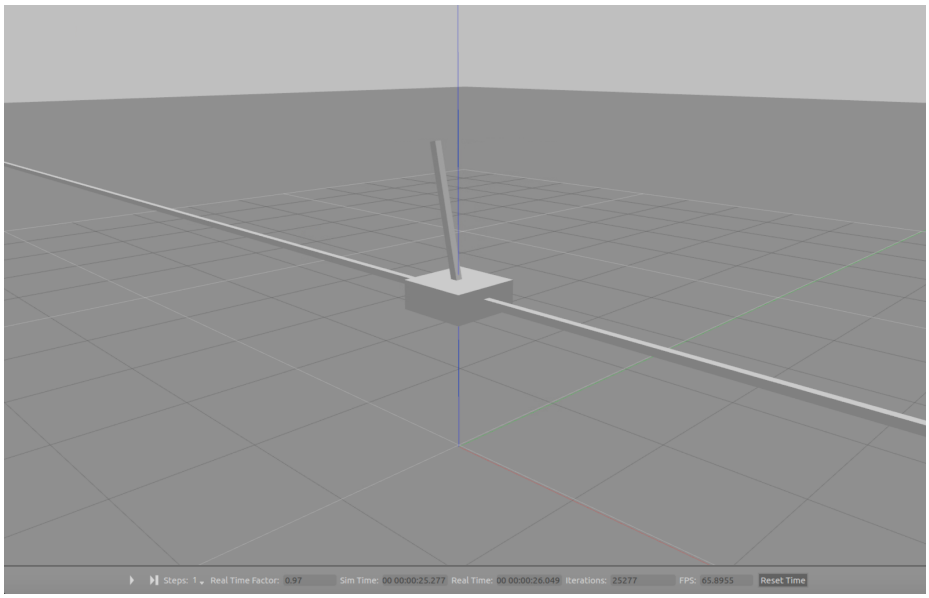

Figure 2.6. A Cart-pole model created in Gazebo.

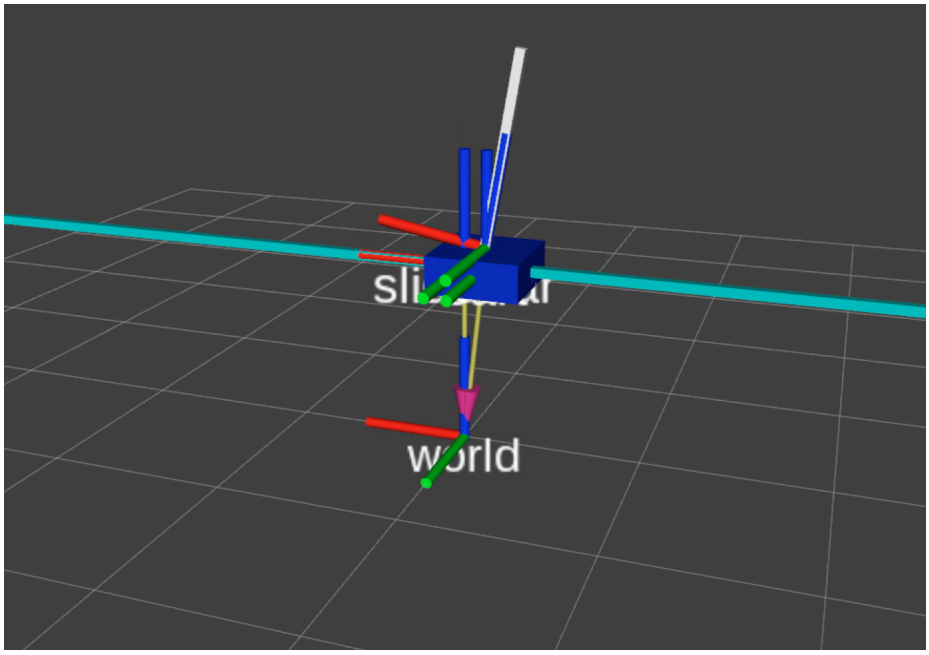

Figure 2.7. A Cart-pole model displayed in ROS-Rviz.

Before we dive into a complex autonomous driving problem on highways, it is helpful to apply the Integrated OpenAI Gym on a simple enough problem, for example, Cart-Pole problem. In this problem or game, the pole needs to keep its balance purely relying on moving the cart in a one-degree axes. A Cart-Pole model was created in Gazebo as shown in Fig. 2.6. The acceptable state follows that the cart is with ±2.4 meters range and pole is with ±12 degree angle range. Otherwise, it loses balance and will be initialized to the initial position and angle.

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

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

- **State Space:** $[pos_{cart}, vel_{cart}, pos_{pole}, vel_{pole}]$
- **Action Space:** $[vel_{cart} + 1, vel_{cart} - 1]$
- **Reward Function:** Each step it gains a 1 point reward until it loses balance and reinitialize.

where pos_{cart} is the position along the translation axes, vel_{cart} is the linear velocity, pos_{pole} is the angle of the pole compared to the initial pose, and vel_{pole} is the angular velocity of the pole.

The Neural Network applied here has one layer with 10 units. The other hyperparameters in the experiment are set as below,

- Epochs: 2000.
- Learning Rate in policy gradient: 0.01.
- Learning Rate in value gradient: 0.1.

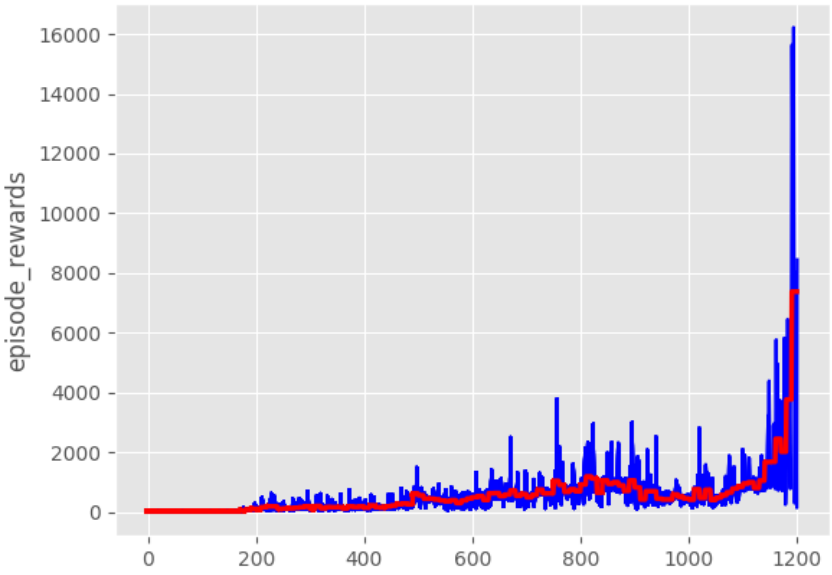

Figure 2.8. Reward history of training A Cart-Pole Agent. (Blue line: the rewards of each time step; Red line: the rewards of every 10 time steps)

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

2.4 Vehicle Model

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

2.4.1 URDF Models

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

Figure 2.9. 4 vehicle models in ADAS Kit.

.

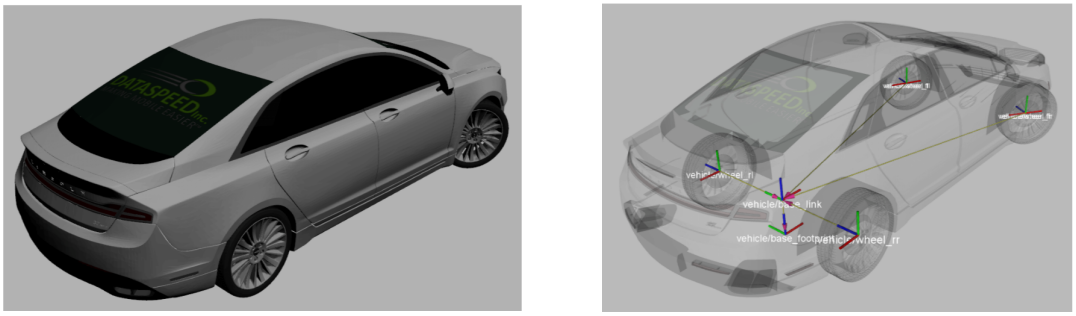

Figure 2.10. Simulation model and corresponding TF tree.

2.4.2 Simulated CAN Message Interface

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

Topic Name	Msg Type
$<name>/can\_bus\_dbw/can\_rx$	$can\_msgs/Frame$
$<name>/can\_bus\_dbw/can\_tx$	$can\_msgs/Frame$

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

The simulator only implements a subset of the complete Dataspeed CAN message specification. The supported command messages are listed in Table 2.3, and the supported report messages are listed in Table 2.3. See the ADAS Kit datasheets for complete CAN message information.

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

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

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

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

2.4.3 Simulating Multiple Vehicles

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

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

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

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

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

3 System Design

Current autonomous vehicles use the same architecture as the Urban DARPA Challenge vehicles did[33 34 35 36]. This architecture comprises three main processing modules, described below and illustrated in Fig. 3.1.

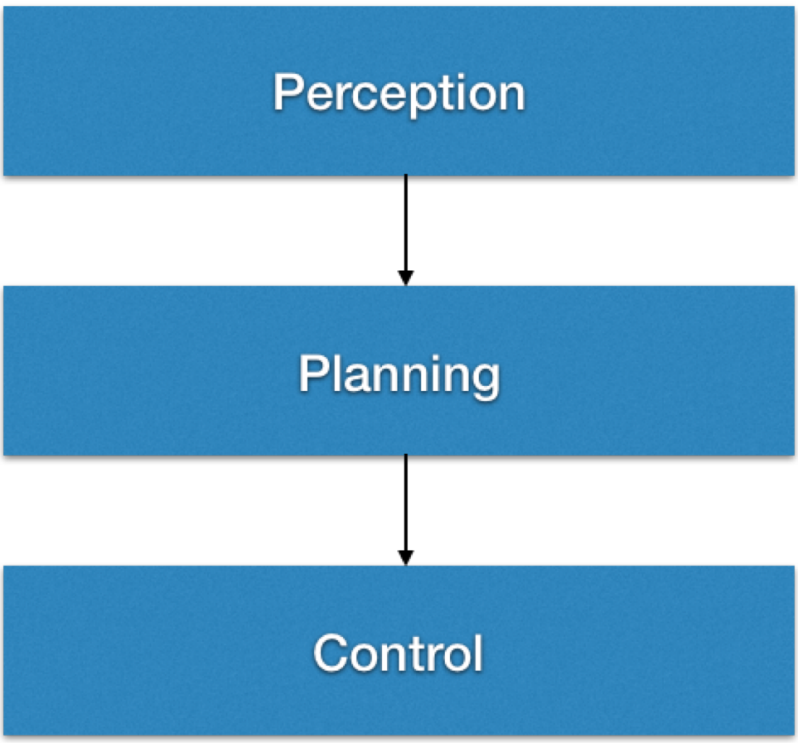

Figure 3.1. Standard Architecture of Autonomous Vehicle.

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

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

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

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

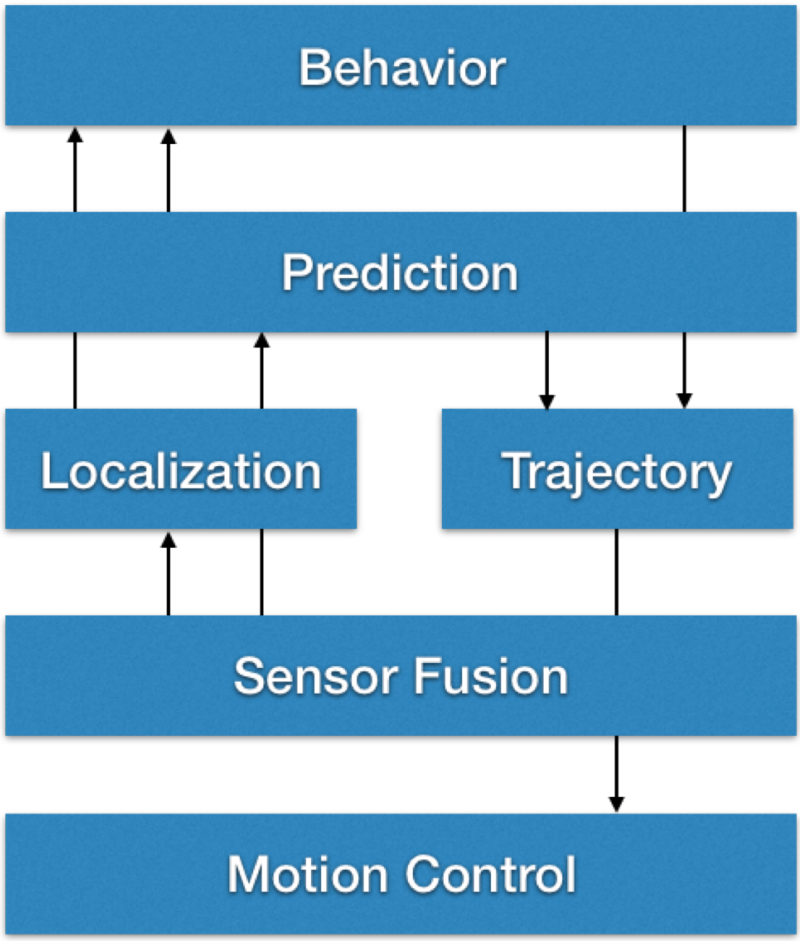

Figure 3.2. Standard Architecture of Autonomous Vehicle Control.

- **Lane Detecting:** Detect the lane line and
- **Adaptive Cruise Control:** Detect a leading vehicle vehicle and maintain a safe distance and speed.
- **Adaptive braking:** Braking system adapts braking to different driving conditions to improve response time, overall safety, etc.
- **Lane Control:** Automatically switch between Lane Keep mode and Change Lane mode with a good timing.

- **Trajectory / Path Generating:** Given the decision making result of ACC and LC, generating a trajectory (a series of waypoints containing the pose and velocity information) for the Motion Control layer to excute.
- **Path Following:** Detecting and determining a path/lane to follow, and following it.

3.1 Behavior Planner

3.1.1 Longitudinal Control: Adaptive Cruise Control

We define the following variables to represent the relative motion of the autonomous vehicle (referred to as the “ego vehicle”) and the vehicle located ahead in the same lane as the autonomous vehicle (referred to as the “preceding vehicle").

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

At each time step t , the driver model generates an acceleration command. This acceleration command is used by the trajectory generator to compute a target velocity as a reference. The controller solves a constrained optimization problem over the prediction

horizon , and generates a planned velocity sequence which guarantees the safety of the vehicle.

3.1.2 Lateral Control: Lane Control

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

- $lane = [true, false]$, for example, indicates the left lane is available but the right lane is not at time t.

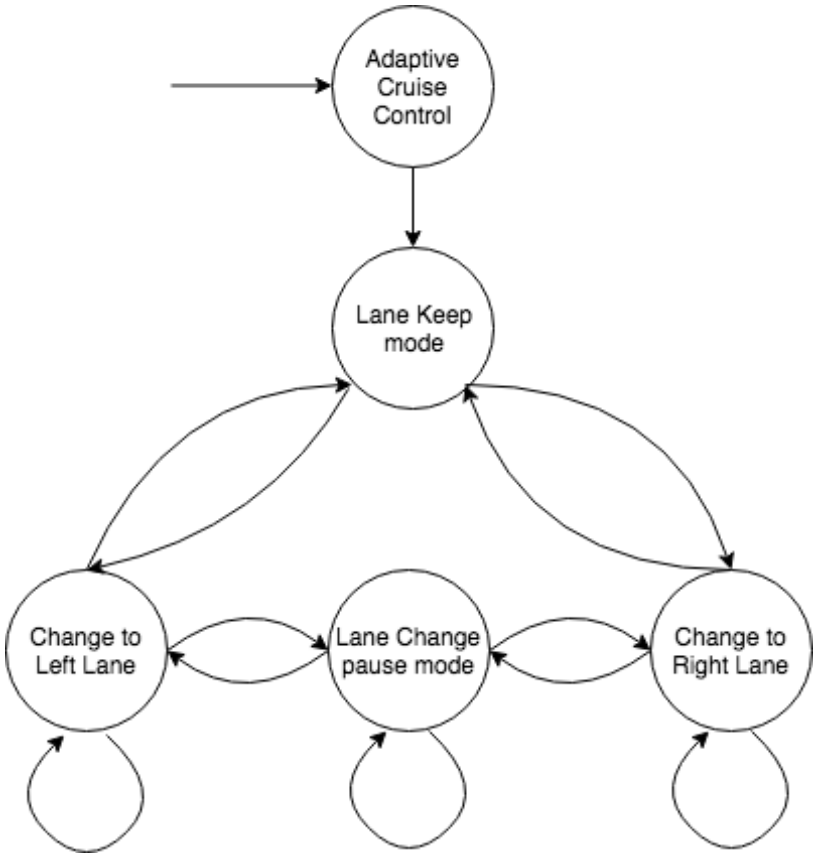

Figure 3.3. The finite state machine for lateral motion control.

At each time step t , the driver model, with the information above together with the other vehicle in its left and right lanes, generates a change lane or a keep lane command. The proposed algorithm will encourage the ego vehicle to change to a different lane neighboring to it when the target lane has better driving condition and to keep the current lane if else. To avoid conflict two lane changing commands which are too close, the lateral control would stay the current state for a period of time for the lane change to finish. Within the time period, new lane change or lane keep commands would be ignored. By this idea, it becomes a rule-based controller, sometimes referred to as “Finite state machine (FSM)”, as shown in Fig. 3.3. FSM has its own advantages, including:

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

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

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

3.2 Path Planner

3.2.1 Reference Path Generator

The proposed Reference Path Generator is used to generate a safe and comfortable path (with an appropriate speed and acceleration) from an initial position towards a destination, while complying with a global route and map. Our method aims to resolve local path planning problems based on a global route and map. The global route is obtained

by the high precision navigation system, and the map is downloaded from the Internet. The map is composed of a set of waypoints on the road edges and topology that describes the relationships between connected roads, as shown in Fig. 3.4. The process by which the map is obtained falls outside the scope of this thesis. Therefore, the maps used in this paper are predefined in our simulations.

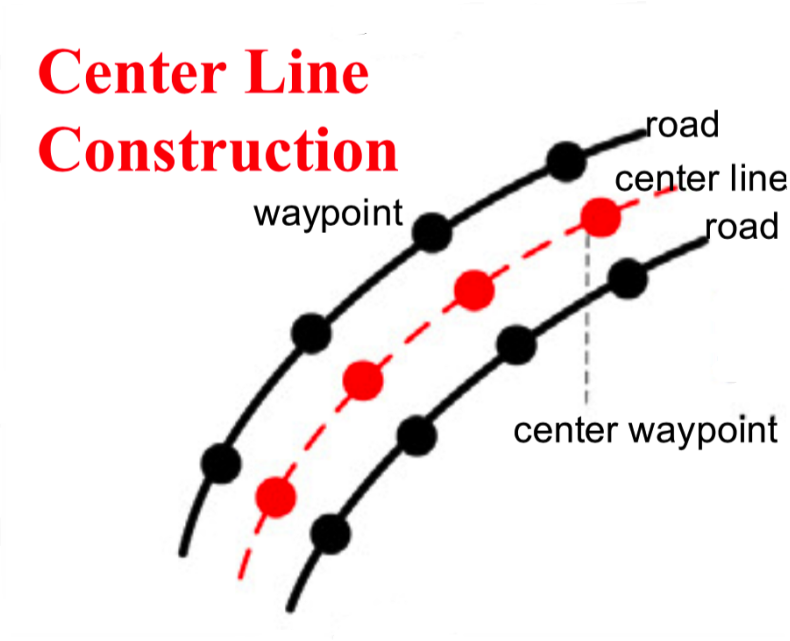

Figure 3.4. The global route and the waypoints.

The dynamic path planning process includes three stages: center line construction, path candidate generation, and path selection. These are performed on the basis of perceived information. The center line of the road is constructed from the center waypoints, which are several waypoints captured from the map and aligned to the center line of the lane, using the method of cubic spline fitting. The path candidates, which are also described by the cubic spline, are generated by adjusting the lateral offset to the center line using the information for the current vehicle position, speed, and direction in a road-aligned coordinate system. During path selection, the costs of static safety,

comfortability, and dynamic safety are taken into account, and are combined with information on road edges, and static and moving obstacles for selecting the optimal path. Our method provides not only the selected path, but also the appropriate speed for the vehicle maneuvering system. In this project, the proposed dynamic path planning algorithm is executed 50 times per second, and a new path is generated from the current vehicle position at every time step.

3.2.2 A road-aligned coordinate system: Frenet Coordinate

As we have mentioned several times in the previous section, a road-aligned coordinate system is needed to have a better view on the global and local paths. A well known one is the Frenet coordinate, which asserts invariant tracking performance under the action of the special Euclidean group $SE(2) := SO(2) \times \mathbb{R}^2$. Here, we will apply this coordinate system in order to be able to combine different lateral and longitudinal cost functionals for different tasks as well as to optimise driving behavior on the highway. As depicted in Fig. 3.5, the moving reference frame is given by the tangential and normal vector $\vec{t_r}$, $\vec{n_r}$ at a certain point of some curve referred to as the center line in the following. This center line represents either the ideal path along the free road, in the most simple case the road center, or the result of a path planning algorithm for unstructured environments[37]. Rather than formulating the trajectory generation problem directly in Cartesian Coordinates $\vec{x}$, we switch to the proposed dynamic reference frame and seek to generate a one-dimensional path for both the root point $\vec{r}$ along the center line and the perpendicular offset d with the relation

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

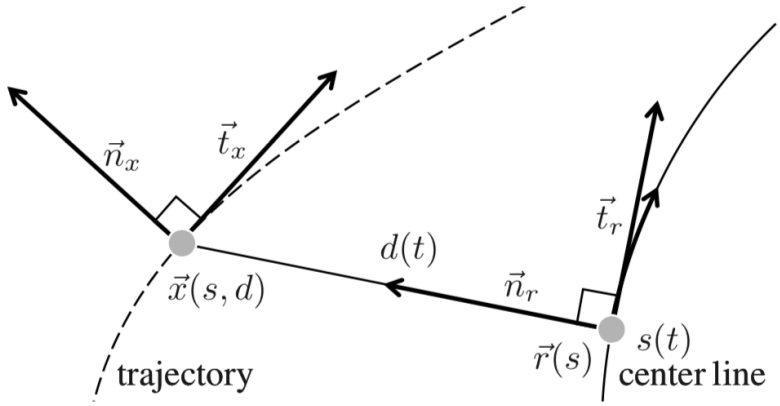

Figure 3.5. Reference path generation in Frenet coordinate.

s coordinate represents distance along the road (also known as longitudinal displacement) and the *d* coordinate represents side-to-side position on the road (also known as lateral displacement). Imagine a curvy road like in Fig. 3.6 with a Cartesian coordinate system laid on top of it.

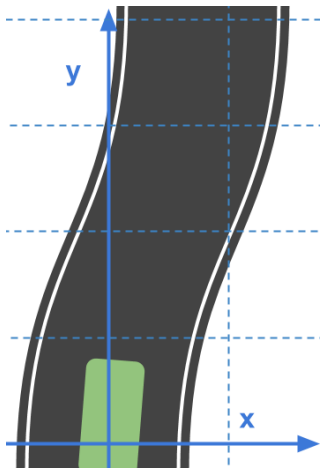

Figure 3.6. A curve road in Cartesian coordinate.

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

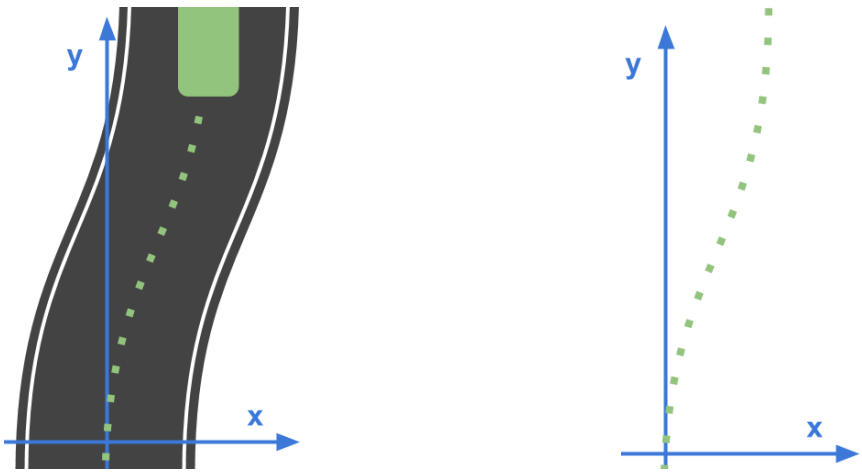

Figure 3.7. A curve road in Cartesian coordinate with waypoints.

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

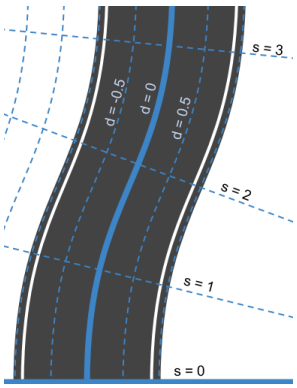

Figure 3.8. Curve in Frenet coordinate system.

Here, we've defined a new system of coordinates. At the bottom we have s=0 to represent the beginning of the segment of road we are thinking about and $d = 0$ to represent the center line of that road. To the left of the center line we have negative d and to the

right d is positive. Then a typical path would look like in Fig. 3.9 when presented in Frenet coordinate.

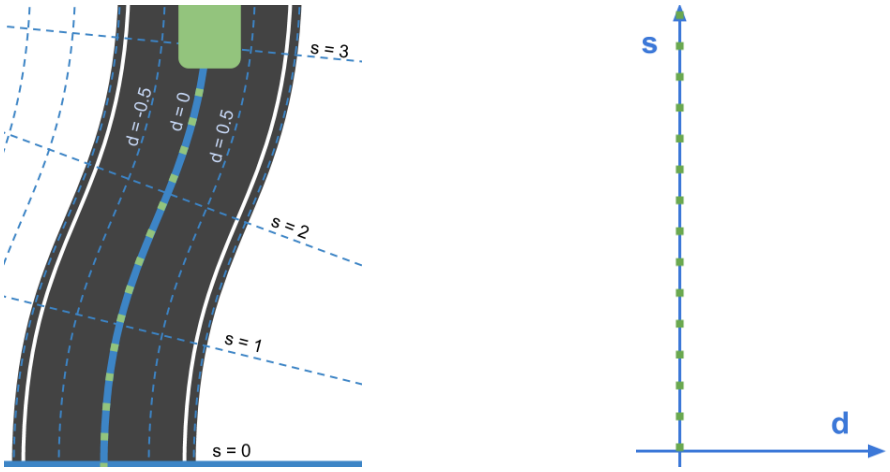

Figure 3.9. A curve road in Frenet coordinate with waypoints.

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

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

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

Straight lines are so much easier than curved ones.

3.2.3 Path Selecting and Optimizing

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

To use the direction and curvature of the center line, it is necessary to find the position of the vehicle on the reference waypoints. We first map the vehicle position from

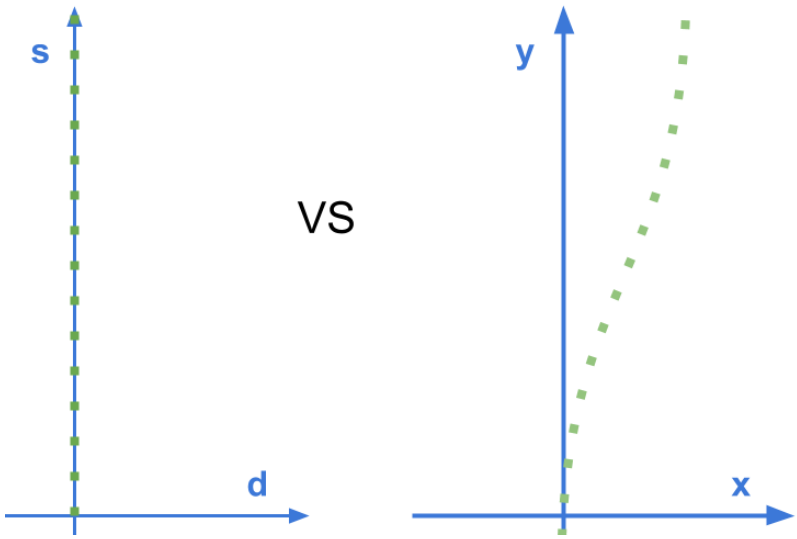

Figure 3.10. Comparison display in Frenet and Cartesian coordinate systems.

the Cartesian coordinate system to the Frenet coordinate system, and then determine the closest point of the center line p_0, which has the minimum distance ρ_{min}. In this paper, a method combining quadratic minimization and Newton's method is used to find p_0.

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

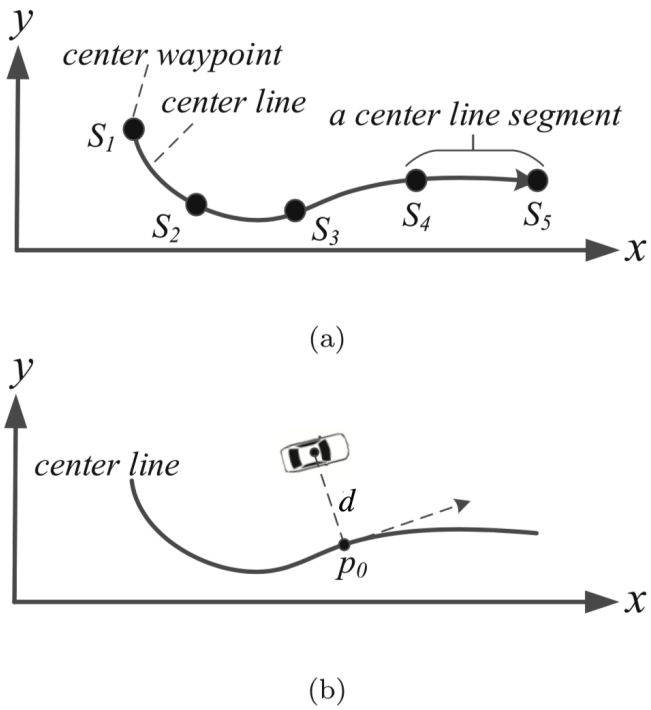

Figure 3.11. Vehicle localization on the center line. (a) Center waypoints and center line segments, (b) localization on the center line.

Path candidates are generated in the Frenet coordinate system, but path planning results must be mapped into a Cartesian coordinate system to convey to the maneuvering system. Path candidate points in the Cartesian coordinate system can be represented with respect to the arc length of the center line.

3.3 Path Following

The path following layer is implemented by referring to the existing Pure Pursuit algorithm. Pure Pursuit is a well-known algorithm for following a given path. We have designed a path-following algorithm based on a previously proposed Pure Pursuit algorithm[38]. It computes the angular velocity command that moves the vehicle from its

current position to reach some look-ahead point in front of the vehicle. The linear velocity is assumed constant, hence you can change the linear velocity of the vehicle at any point. The algorithm then moves the look-ahead point on the path based on the current position of the vehicle until the last point of the path. Given the pose (position and orientation) of the vehicle as an input, the object can be used to calculate the linear and angular velocities commands for the vehicle. How the vehicle uses these commands is dependent on the Motion Control layer.

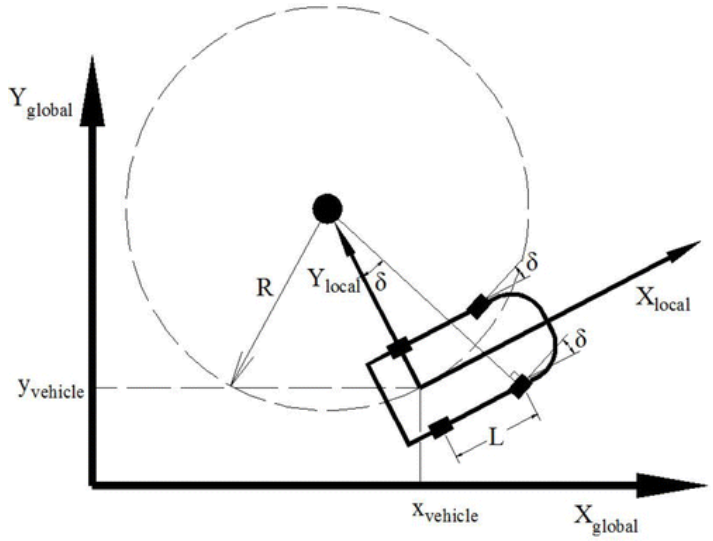

Figure 3.12. Global coordinate system (Xglobal, Yglobal), local coordinate system (xlocal, Ylocal).

Fig. 3.12 shows the global coordinate system and the local coordinate system the origin of which is located at the center of the rear wheels. We used X_{global}, Y_{global} to represent the global coordinate, and x_{local}, y_{local} to represent the local coordinate. Ignoring the motion of z axis, we can express the motion state of the vehicle as position and orientation (heading), $q = [x, y, \theta]$. The kinematic model of the robotic vehicle is expressed as:

$$\dot{x} = v\cos(\theta) \tag{3.2}$$

$$\dot{y} = v\sin(\theta) \tag{3.3}$$

$$\dot{\theta} = v\kappa \tag{3.4}$$

where x, y is the position of the vehicle, θ is the vehicle's heading, as shown in Fig. 3.12. According to Eq. 3.2, 3.3 and 3.4, the main task regarding tracking is to calculate the curvature κ, velocity v. The backward motion is not taken into consideration and the dynamic effects such as side slip are ignored for control design purposes. As a result, the Ackerman geometric relationship, Eq. 3.5 can be employed to calculate the curvature. And Eq. 3.4 can be rewritten as Eq. 3.6:

$$\kappa = 1/R = \tan(\delta)L \tag{3.5}$$

$$\dot{\theta} = v\tan(\delta)L \tag{3.6}$$

where the δ is the steering angle of the front wheel, L is the vehicle's wheelbase, R is the turning radius and κ is the turning curvature. Rather than analyze the whole vehicle model in Fig. 3.12, we may simplify the vehicle into a classical bicycle model, shown in Fig. 3.13 so that Eq. 3.5 can be applied.

When the vehicle turns with a fixed steering front-wheel angle, the vehicle's path will be a circle with an approximately constant radius. Conversely, when you specify a

circular path for the vehicle, you can also calculate a steering angle for the vehicle to track the circle.

The principle of Pure-Pursuit is to calculate the instantaneous curvature of the path that the vehicle intends to generate with the current velocity and heading. We use Fig. 3.13 to show the detail of Pure-Pursuit.

We can detail the implementing process of Pure-Pursuit as follows:

(1) Find the current location of the vehicle in the global coordinate system ($x_{vehicle}$, $y_{vehicle}$);

(2) Find the closest point on the path to the vehicle, (X_{cv}, Y_{cv}) which is used to locate the vehicle on the path, at which point we can search from it;

(3) Choosing a constant look-ahead distance and then search the goal point (X_{la}, Y_{la});

(4) Transform the goal point to vehicle coordinates (x_{la}, y_{la});

(5) Calculate the curvature and then acquire the steering angle from Eq. 3.7;

$$Error_{cte} = Error_{calculate} + Error_{tracking} \tag{3.7}$$

In the equation, there are two different errors, $Error_{calculate}$ and $Error_{tracking}$. We define the $Error_{tracking}$ as the the errors brought by the procedure when the vehicle executes the control commands. Another $Error_{calculate}$ is the error caused by the truth control value and calculated value through algorithms.

(6) Update the vehicle's position and repeat.

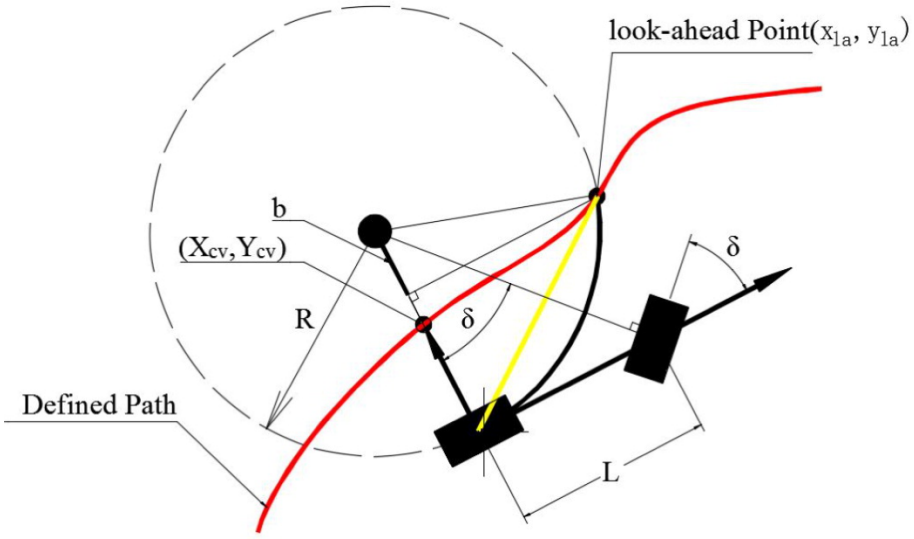

Figure 3.13. Geometric explanation of Pure-Pursuit.

3.4 Drive By Wire

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

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

The advantage of using drive-by-wire rather than mechanical systems is that reduction of cost, moving parts and weight can be achieved. Since the steering rack can be removed, the car?s shock impact, in case of a collision, can be improved. Using an electrical based system will also increase the information flow and ease up the interconnect

between different components in the car, facilitating the use of safety functions such as; ABS (anti-lock brake system), ESP (electronic stability programme), etc.

4 Deep Q-Learning

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

4.1 General Architecture

Our system follows the basic RL structure. The agent performs an action A_t given state S_t under policy π. The agent receives the state as feedback from the environment and gets the reward r_t for the action taken. The state feedback that the agent takes from sensors consists of the velocities of the neighboring vehicles $v_{veh}[i](i = 1,2,...,N)$ and the relative positions of the neighboring vehicles to the ego vehicle $dist_{veh}[i](i = 1,2,...,N)$. Possible actions that agent can choose are 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 within $dist_{safe}$.
- **Time Out** The maximal number of time steps are reached.
- **Bump** The ego vehicle is turned over for some reason.
- **Off Lane** The ego vehicle is out of lanes.

The ego vehicle continuously detects the space information in 6 areas around itself as shown in Fig 4.1 and the maximal detecting range is 80 meters in front and behind. The variable $s_i(i = 0, 1, ..., 5)$ represents the detected vehicle distance or obstacle distance along the path with the ego vehicle. If none is detected in the range of 80 meters, 80 would be put in the variable. For example, Vehicle 0 is located in the left lane of the ego vehicle and is in front of it. Vehicle 0 is defined in Area 0. When there are multiple vehicles in the same area, only the closest is considered. Later we can use the information to check the state of the left lane or the right lane. A function is provided to compute if the lane change is safe or not.

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

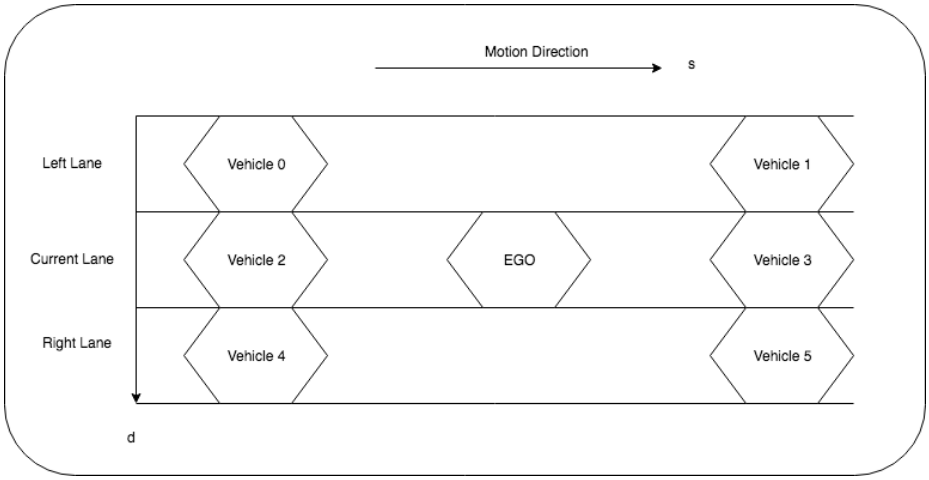

Figure 4.1. A general highway case display.

The general architecture is as shown in Fig. 4.2.

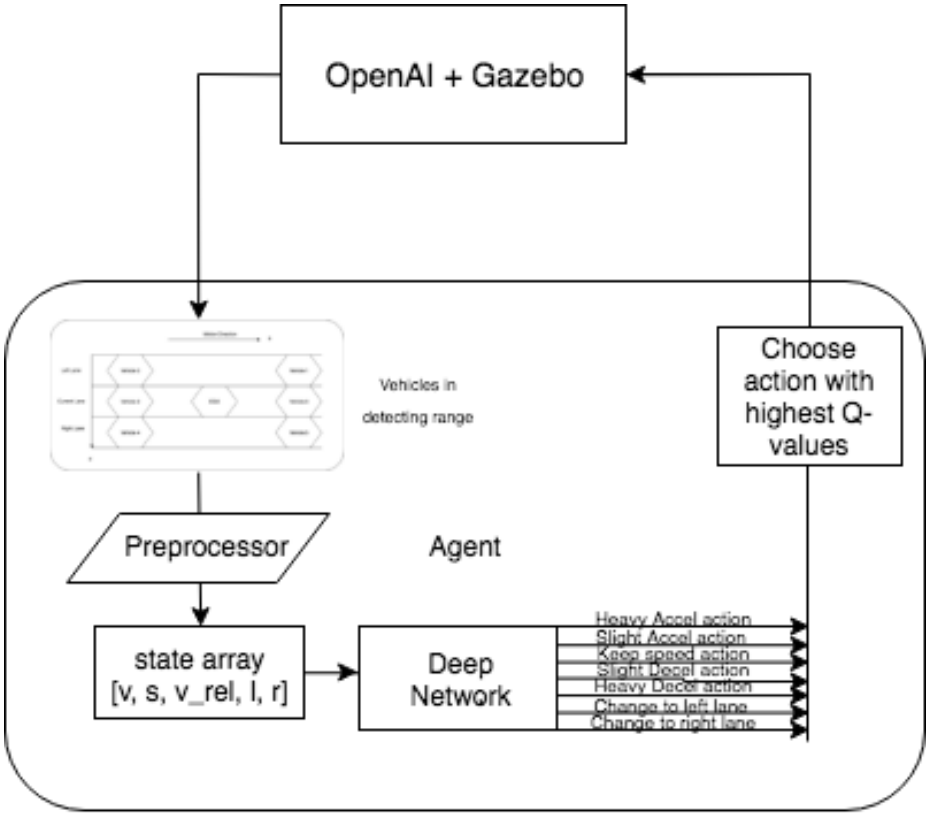

Figure 4.2. A general architecture of training.

OpenAI and Gazebo are providing online environment state information including vehicle's state information. With the global path or the map given, we construct an array indicating the poses and velocities from the point of view of the ego vehicle with the detecting range.

The preprocessor is to simplify and separate the in lane information for ACC and neighboring lane information for LC. Finally the state array to be fed into the Deep Network is a small and normalized array. The first three items "v", "s" and "v_{rel}" are representing the speed of the ego vehicle v_{ego}, the distance between the ego vehicle and the preceding vehicle $s_{pre} - s_{ego}$ and the relative speed between the two, $v_{rel} = v_{pre} - v_{ego}$. The three are key variables for adaptive cruise control. The remaining two, l and r, indicate the availabilities of the left lane and the right lane. The two are binary variables with "True" representing that the lane is safe to reach. We define that only when there no vehicle detected within some range to the ego vehicle the target lane is thought to be available. Here we set 30 meters as the range limit. With this setting, we don't need to consider adaptive control problems when changing lane. After it change to a new lane, the first three variables will also update to the state in the new lane and it will have enough time to adjust its speed with ACC.

The Deep Network will be trained to output Q-values for different classes of activities. The action with highest Q-value will be thrown into OpenAI / Gazebo to generate the next state.

4.2 Reinforcement Learning for Longitudinal Motion

To apply this RL framework, we first had to model the problem by defining the states, actions, goals and rewards. Our first approach was to use variables such as the position

of a leading car, their velocities and accelerations, etc. Clearly, this state definition put us up against the curse of dimensionality, and it became impossible to have a discrete state space precise enough to learn a valuable policy. We modified our state definition by consolidating numerous state variables. This allowed us to use a smaller discretization and to reach a better precision with only two variables. Since driving can be seen as a sequential decision problem, there is no problem in modeling it using a MDP and discrete state variables. For now, our discrete state space was built around a state definition containing variables as we defined our states by the relative distance and velocity between two vehicles.

This relative movement between vehicles is needed to take an informed decision on the action to take at the next time step. Those actions were taken directly on the brakes or throttle (only one action per time step is chosen), closely simulating human interaction. The actions were discretized, according to a percentage of pressure on the pedal, from accelerating, keeping to decelerating.

The goal is to maintain a high speed on highway and to slow down when detecting a slow vehicle ahead. To reach the goal, we set the rewards accordingly, with a positive reward proportional to the velocity (for a given time step it is the distance moved along the path). We also set negative rewards when wandering too close with the preceding vehicle. The behavior the agent is supposed to learn is to maintain high speed and safe as long as possible.

Those elements were put together in a RL framework, and the policy obtained, learned in a simulated environment, formed the core of our longitudinal controller. The environment, a simulated highway system built in previous work, featured complex vehicle physics and dynamics as close as possible to reality. Since the simulation environment

was using continuous time, we had to define the time interval at which action decisions would be taken. The action chosen at the specified time frame would be taken for the whole frame. To observe an accurate behavior of the vehicle, we had to set the time step between each action decision to a small value (50 milliseconds). But in such conditions, the observation of real vehicle acceleration needed many consecutive acceleration actions, a behavior that could not be learned in a decent time with normal state space exploration. To overcome this problem, we had to use a heuristic to speed up learning. The heuristic specified that every time the car was behind the desired time ratio, the best acceleration action known from experience was taken. By ignoring in that case the braking actions, this action selection technique directed rapidly the agent towards more rewarding locations of the state space.

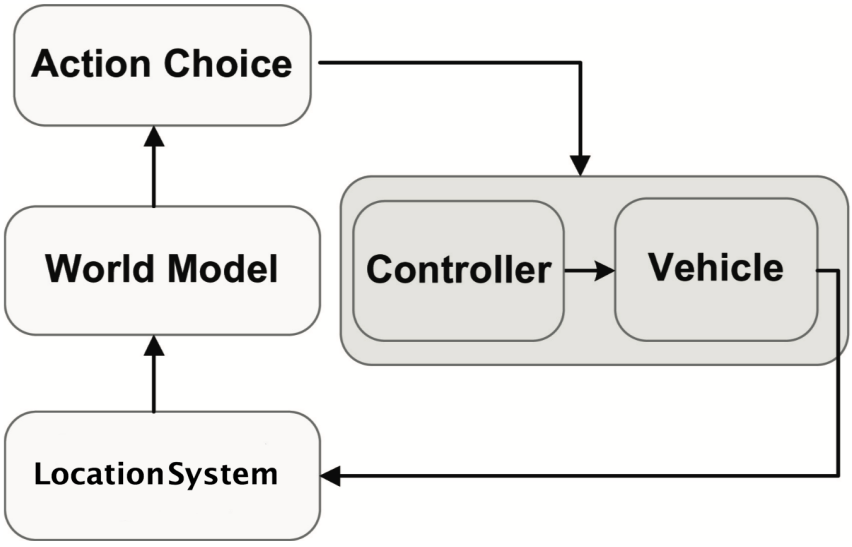

Figure 4.3. Reinforcement applied on ACC system.

Put into context, Fig. 4.3 shows that using RL simplifies the design of a longitudinal controller. The closed-loop controller takes as inputs the vehicle's state as described earlier, and selects the appropriate action according to the policy that was learned. Such a

technique is obviously simpler than the complex mathematical analysis needed to predict precise car physics and dynamics for acting, as our controller basically hides in a black box vehicle physics and dynamics. It is possible for the agent to learn the optimal behavior by taking driving actions and observing their results on the time distance and its difference between two time steps. In the next section, we will show results obtained by using this policy for longitudinal vehicle control. As drivers spend a great amount of their time in heavy traffic, such systems could reduce the risk of rear-end collisions and protect the drivers mentally by relieving them from stressful driving.

4.3 Reinforcement Learning for Lateral Motion

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

In this section, we describe the design of the Lane Control layer and, more precisely, the design of the policy to select the most efficient and safest lane for each vehicle according to their current state and action. Lane changes are stressful maneuvers for drivers, particularly during high-speed traffic flows. As designed in the last Chapter, the

Lane Control system is using a Finite State Machine (FSM) and hence RL is only responsible for choosing the optimal behavior or next Finite State. The action space consists of three actions, Change to Left, Keep Lane and Change to Right. To make such kind of decision, the agent needs to be aware of the nearby vehicle's position and state in both its current lane and nearby lane. We maintain an array combining a 6-dimensional relative position array which captures space information in 6 nearby locations as shown in Fig. 4.1 for the Lateral Control agent. With the array, we can easily check if the neighboring lane is available to reach.

4.4 Q Learning

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

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

For the given state s and action a, where r_t is the reward received at the time step t. The Q-value function is the expected sum of the future rewards which indicates how good the action a is given the state s under the policy of the agent π. The contribution to the Q-value function decays exponentially with the discounting factor γ 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 descent 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 $\in [0,1]$.

4.5 Policy Representation

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

4.5.1 State

A state s consists of features describing the state of the ego vehicle and relative positions and velocities with its neighboring vehicles. The state is represented by its pose p and velocity q, where p 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.

4.5.2 Actions

For a longitudinal controller, an action space of 5 actions is defined,

- **Heavy acceleration.** Increase 2 m/s from current speed.
- **Slight acceleration.** Increase 1 m/s from current speed.
- **Keep speed.** Keep current speed.
- **Slight deceleration.** Decrease 1 m/s from current speed.

- **Heavy deceleration.** Decrease 2 m/s from current speed.

For a longitudinal controller, an action space of 5 actions is defined,

- **Change to Left.**
- **Keep Lane.**
- **Change to Right.**

Usually, a total of 15 controller parameters serve to define the available policy actions. These include specifications of the 5 level of speed control action and 3 kinds of lane control action. To simplify the action space, we here can assume the speed is constant when changing lanes. By this, the action space is downsized as 7 possible actions, i.e. 5 level of speed control action and 2 kinds of lane change action (left or right).

4.5.3 Reward Function

The reward should be appropriately defined by a system designer in Adaptive Cruise System or Lane Control System.

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 finish the trip 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

$$r_t = \alpha * r_s + \beta * r_{action} + r_{collision} \tag{4.14}$$

where r_s is the incremental travel distance along the path of the ego vehicle at the time step t and r_{action} is the reward (or penalty if negative) of the action at the time step t. The first term $\alpha * r_s$ in the reward function encourages the vehicle to drive as fast as possible within the speed limit to achieve high reward. It guides the vehicle to drive without deceleration if the preceding vehicle is far from the ego vehicle. The second term $\beta * r_{action}$ is to penalize frequent speed changes. On the other hand, the term $r_{collision}$ indicates the penalty that the agent receives when the accident occurs. Once the collision happens, a huge penalty is received and the episode ends here. The constants α and β are the weight parameters that controls the trade-off between two objectives.

As for the lane control, it is all about timing of changing to the next finite state defined in previous chapter. The reward function should be designed to encourage the ego vehicle to change to a safer lane than the current one. When the current lane is safe enough or the preceding car is far away enough, it is then encouraged to keep in the lane since it is not reasonable to change lanes quite often.

The reward function for the ACC and LC agent is almost the same as in Eq. 4.14 except that the second item $\beta * r_{action}$ is dealing with more actions and is to penalize frequent speed and lane changes. The lane change actions have been forbidden when the target lane is not available (vehicles or obstacles are detected in some specified ranges which might potentially leads to an accident). The agent would make lane changes in different scenarios namely with different state arrays. It would get penalty when hitting the forbidden changing lane trigger. Also every time it changes lane successfully, it gives a price. After all these, the agent is expected to learn to change lanes when safe and efficient.

4.5.4 Relay Memory

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

Combined with deep learning, experience replay has enabled impressive performances[8]. 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.

4.6 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. For instance, fully connected neural networks (FCNNs) can be used as components of RL agents, allowing

them to learn directly from raw, high-dimensional visual or lidar range inputs. In general, DRL is based on training deep neural networks to approximate the optimal policy π, and/or the optimal value functions $V^{\star}$. For example, in this thesis a Deep Neural Network is trained to map the state and action arrays to Q-values which points to the optimal policy, as shown in Fig. 4.4.

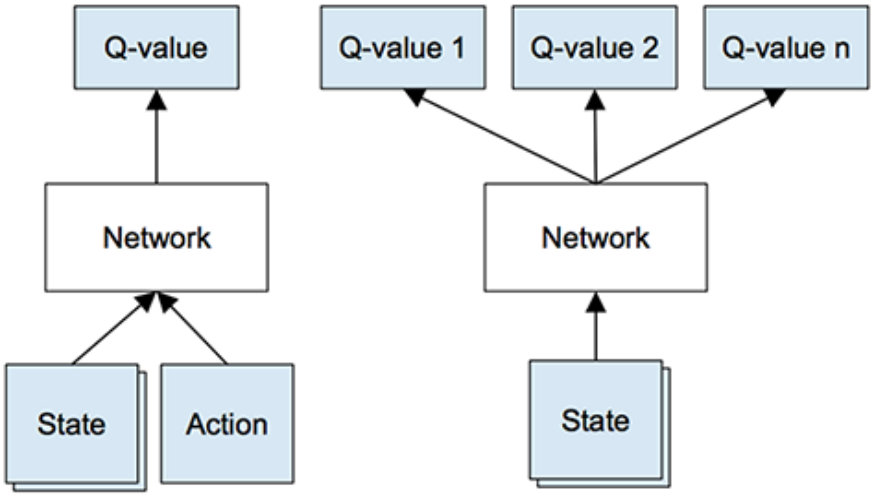

Figure 4.4. Architecture of Deep Q-Network.

Although there have been DRL successes with gradient free methods, the vast majority of current works rely on gradients and hence the backpropagation algorithm. 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. 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, inaccuracies can accumulate.

If we do all calculations of Deep Neural Network, we will end up with an output, which is actually incorrect. So knowing this we want to update neuron weights and biases so that we get correct results. That?s exactly where backpropagation comes to play. Backpropagation is an algorithm which calculates error gradients with respect to each network variable (neuron weights and biases). Those gradients are later used in optimization algorithms, such as Gradient Descent, which updates them correspondingly. The process of weights and biases update is called Backward Pass.

In order to start calculating error gradients, first, we have to calculate the error (in other words, loss) itself. We will use standard classification loss, cross entropy. However, the loss function could be any differentiable mathematical expression. The standard choice for regression problem would be a Root Mean Square Error (RMSE). The cross entropy loss looks as following:

$$L = -\sum_{j}^{M} y_j ln p_j \tag{4.15}$$

5 Results

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

5.1 Simulation Setup

In simulations, we used the open source software OpenAI-Gym and ROS / Gazebo which models a highway environment in real time. We generated the environment in order to train the DQN by simulating the behaviors of the cars on highway. In the simulations, we assume that the positions and velocities of the ego vehicle's neighboring vehicles is detected in real time by the ego vehicle. In the beginning of each episode, the position of ego vehicle is set in a defined position but the velocity is initilaized with 0. The initial positions and velocities of other vehicles are initialized based on their lanes. Along with

time growing, the relative positions and relative velocities among the vehicles are changing and there have chances to create several different scenarios for the DQN model to learn as below,

- Scenario 1: No vehicle in the ego vehicle's detecting range.
- Scenario 2: One vehicle is detected in the same lane of the ego vehicle.
- Scenario 3: One vehicle is detected but in a different lane of the ego vehicle.
- Scenario 4: Vehicles are detected in both the ego's lane and the neighboring lane.

5.2 Training for Single-lane Motion

The Single-lane Motion means that actions here are all changing the speed. This environment with one lane as shown in Fig. 5.1 would focus on training an Adaptive Cruise Control agent for the ego vehicle. The ego vehicle and the preceding vehicle would be initialized with defined the positions and headings and surely the relative position is enough safe. The dimension of the state space is set to $n = 3$ and the variables are the velocity of the ego vehicle v_{ego}, the longitudinal distance of the vehicles s_{rel} and the relative velocity $v_{rel} = v_{pre} - v_{ego}$. To accelerate the training, they are normalized before feeding to the network.

The neural network used for the DQN consists of the fully-connected layers with five hidden layers. RMSProp algorithm is used to minimize the loss with learning rate $\epsilon = 0.0005$. We set the size of the replay memory to 10,000. We set the replay batch size to 32. The summary of the DQN configurations used for our experiments is provided below:

- State buffer size: n = 3

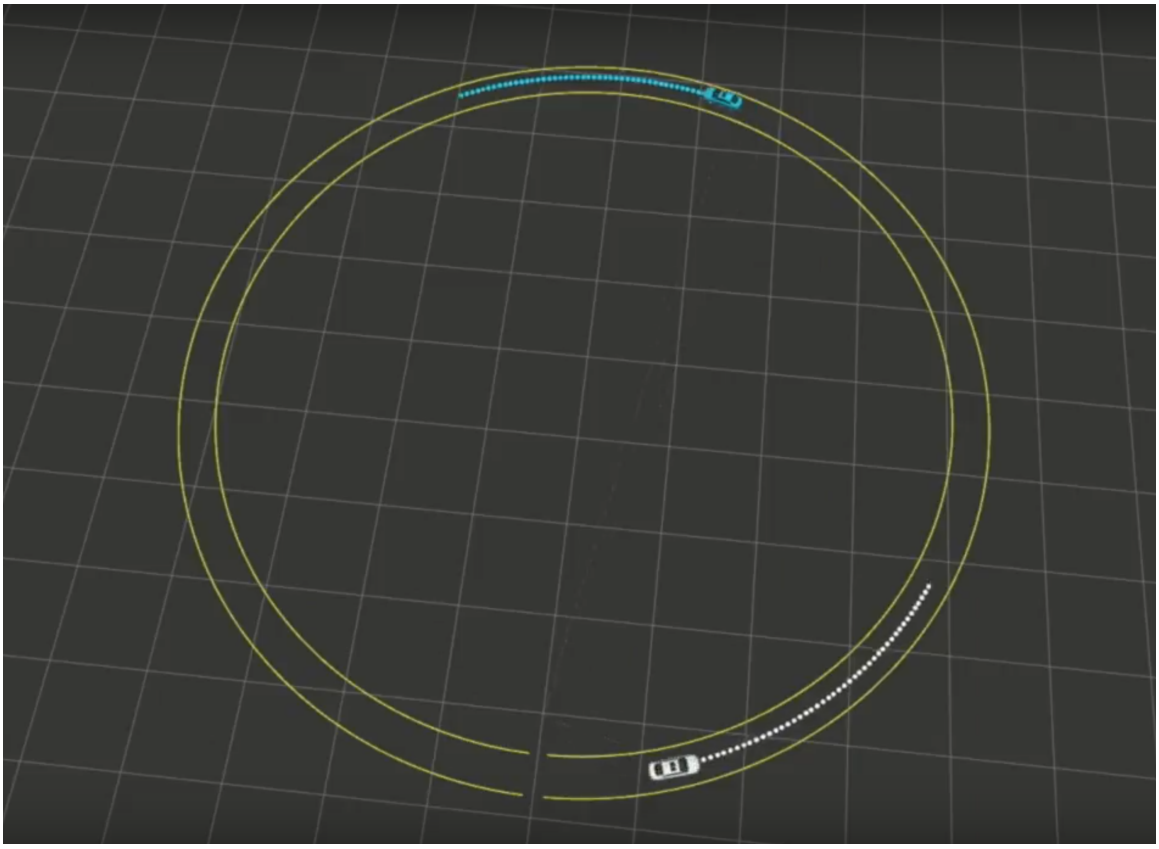

Figure 5.1. Training environment for Adaptive Cruise Control.

- Network architecture: fully-connected feed-forward network
- Nonlinear function: leaky ReLU
- Number of nodes for each layers : [3 (Input layer), 100, 70, 50, 70, 100, 5 (Output layer)]
- RMSProp optimizer with learning rate 0.0005
- Replay memory size: 10,000
- Replay batch size: 32

The parameters are set as below,

- Cruise speed of the preceding vehicle is $v_{pre} = 10m/s$.
- Safety distance is $dist_{safety} = 10m$

- $accel_{high}, accel_{low}, keep_{zero}, decel_{low}, decel_{high} = 2, 1, 0, -1, -2 m/s^2$

The reward function is as in 4.14 and we set $\alpha = 1.0$ and $\beta = 4.0$.

- r_s = incremental travel distance from the last time step (we can simply use euclidean distance between the current position and the previous one since the time is short and curve road can be ignored)
- r_{action} = {accelerate or decelerate with 2 m/s: -1.0, accelerate or decelerate with 1 m/s: -0.5, keep the speed: 0.0}
- $r_{collision}$ = {No collision: 0.0, Collision: -100}

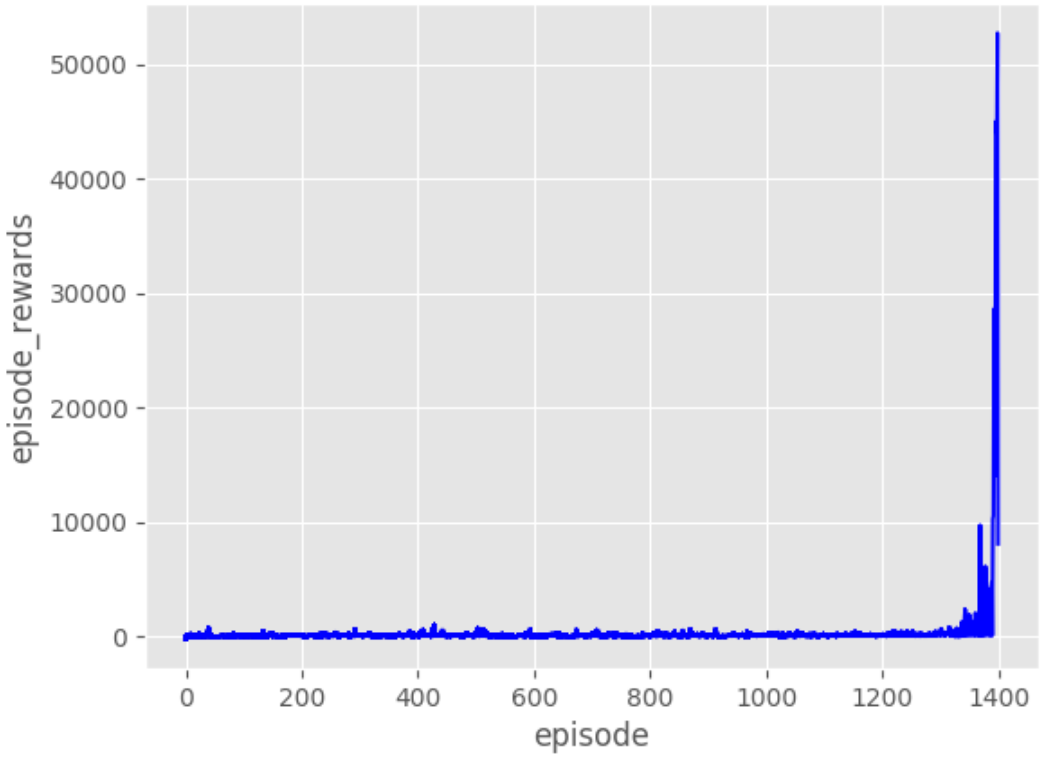

Figure 5.2. The reward history of training for Adaptive Cruise Control.

Fig. 5.2 provides the plot of the total accumulated rewards i.e., value function achieved for each episode. After 1400 episodes, there shows a high jump of the reward which means ACC is working and it takes forever to have a collision.

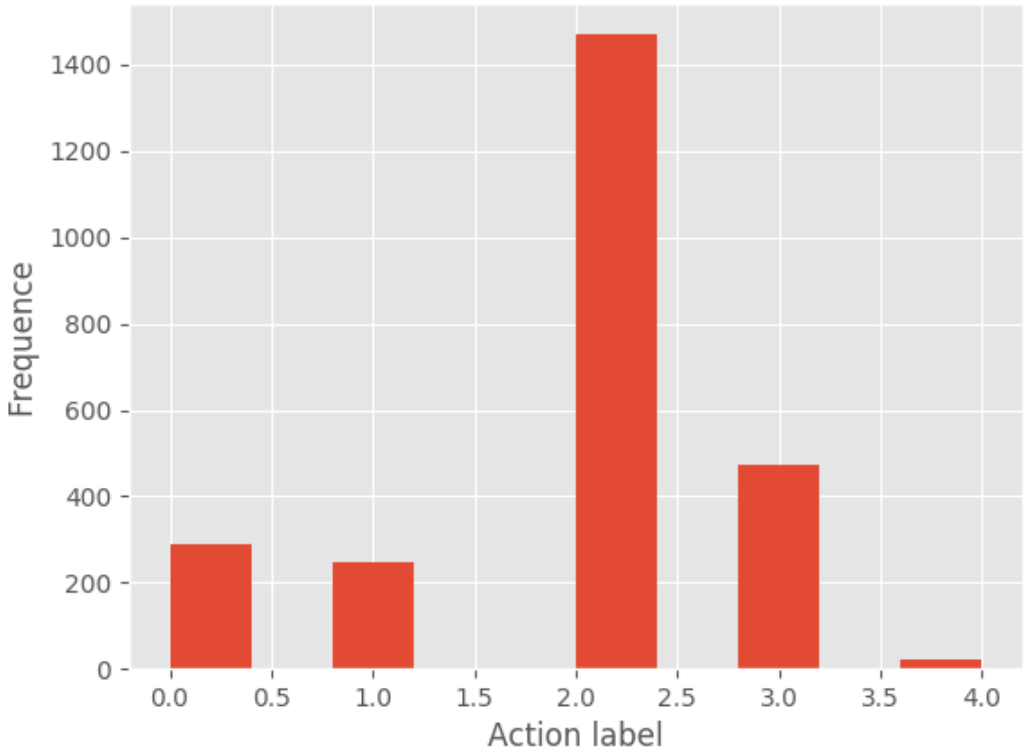

Figure 5.3. The action distribution after training.

To quantify the performance after 1400 episodes' training, we recorded a piece of time of driving behavior using the trained agent. As shown in Fig. 5.3, the actions are mostly chosen as "Keep the current velocity" since our reward function penaties any speed changes. About 75% percent actions generated by the agent during the piece of time are "Keep velocity" (labeled as "2"). Accordingly, the speed variance shown in Fig. 5.4 proves that the ego vehicle is closely following the speed of the preceding vehicle.

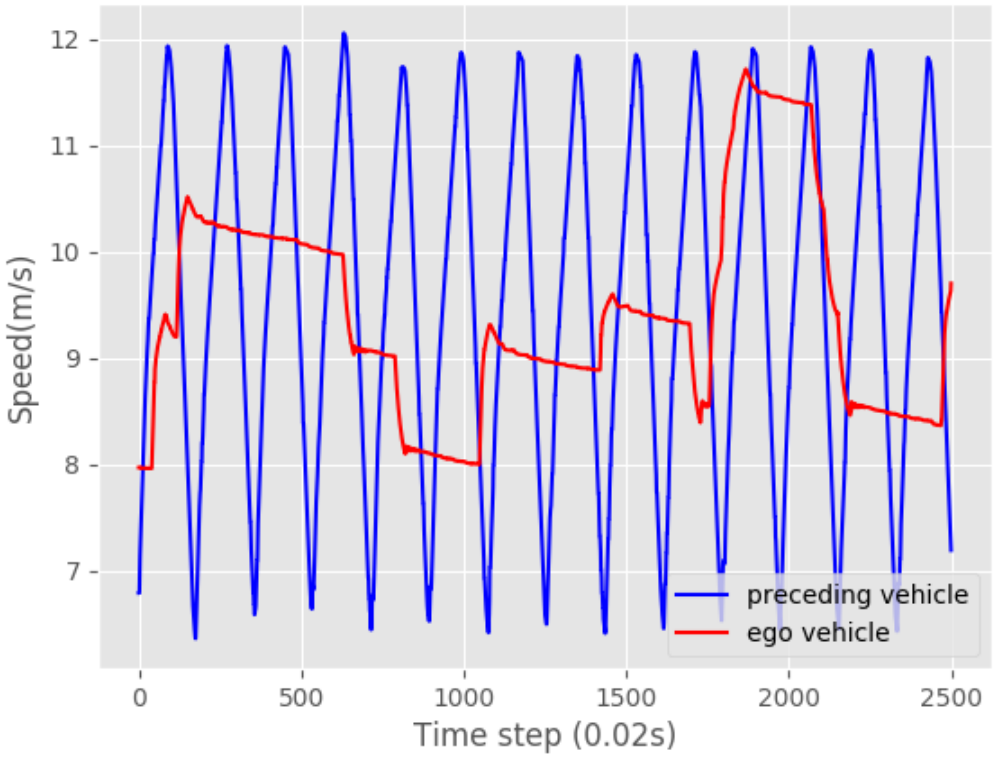

Figure 5.4. The speed variance after training.

The speed of the preceding vehicle is varying by 2 m/s positively or negatively around 9.5 m/s. The ego vehicle learned to maintain the same average speed but with less variance and for some duration it was relatively stable at a speed, for instance, from 200s to 600s.

5.3 Training for Multi-lane Motion

The Multi-lane Motion means that changing the speed and changing lanes are both available as driving behaviors. This environment with two lanes as shown in Fig. 5.5 would focus on training an Adaptive Cruise Control and Lane Control agent for the ego vehicle. The ego vehicle and the other two vehicles would be initialized with defined the

positions and headings and surely the relative position is enough safe. The dimension of the state space is set to $n = 5$ and the variables are the velocity of the ego vehicle v_{ego}, the longitudinal distance of the vehicles s_{rel}, the relative velocity $v_{rel} = v_{pre} - v_{ego}$, the flag showing if the left lane is available $flag_{left}$ and the flag showing if the left lane is available $flag_{right}$. To accelerate the training, they are normalized before feeding to the network.

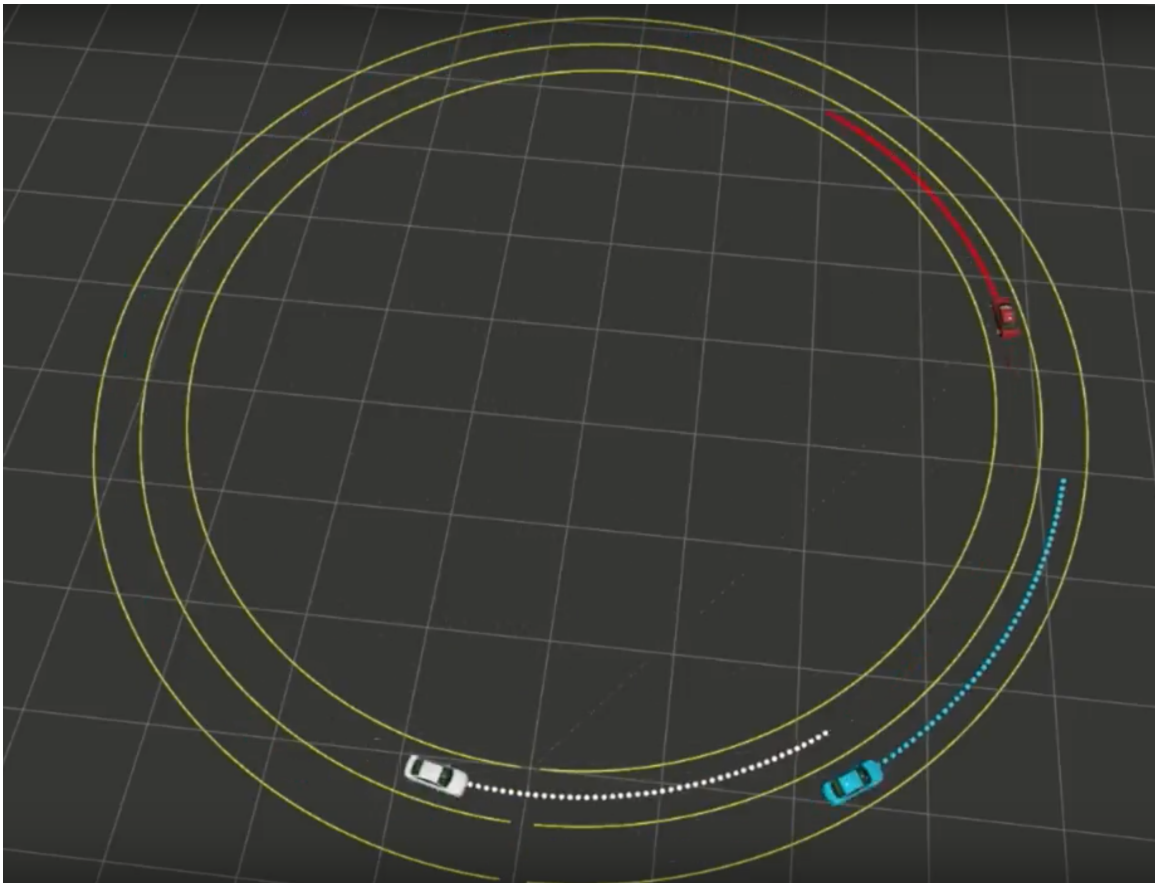

Figure 5.5. Training environment for Adaptive Cruise Control.

The summary of the DQN configurations used for our experiments is provided below:

- State buffer size: n = 5

- Action Space size: 7 = 5+2. (5 level of speed control and 2 modes of lane change)
- Network architecture: fully-connected feed-forward network
- Nonlinear function: leaky ReLU
- Number of nodes for each layers : [5 (Input layer), 100, 70, 50, 70, 100, 7 (Output layer)]
- RMSProp optimizer with learning rate 0.0005
- Replay memory size: 10,000
- Replay batch size: 32

The parameters are set as below,

- Cruise speeds in Lane 0 and 1 are v_{lane0}, v_{lane1} = $6m/s$, $5m/s$, which means Lane 0 and 2 are Fast Lane and Slow Lane.
- Safety distance is $dist_{safety} = 10m$
- $accel_{high}, accel_{low}, keep_{zero}, decel_{low}, decel_{high} = 2m/s^2, 1m/s^2, 0m/s^2, -1m/s^2, -2m/s^2$.

The reward function is as in 4.14 and we set $\alpha = 1.0$ and $\beta = 2.0$.

- r_s = travel distance from the last time step (we can simply use euclidean distance between the current position and the previous one since the time is short and curve road can be ignored)
- r_{action} = {accelerate or decelerate with 2 m/s: -1.0, accelerate or decelerate with 1 m/s: -0.5, keep the speed: 0.0, change lane to left or right: -1.0, change lanes when lane is not available: -3.5}
- $r_{collision}$ = {No collision: 0.0, Collision: -100}

Fig. 5.6 provides the plot of the total accumulated rewards i.e., value function achieved for each episode. We observe that the reward increases along the episodes and high total

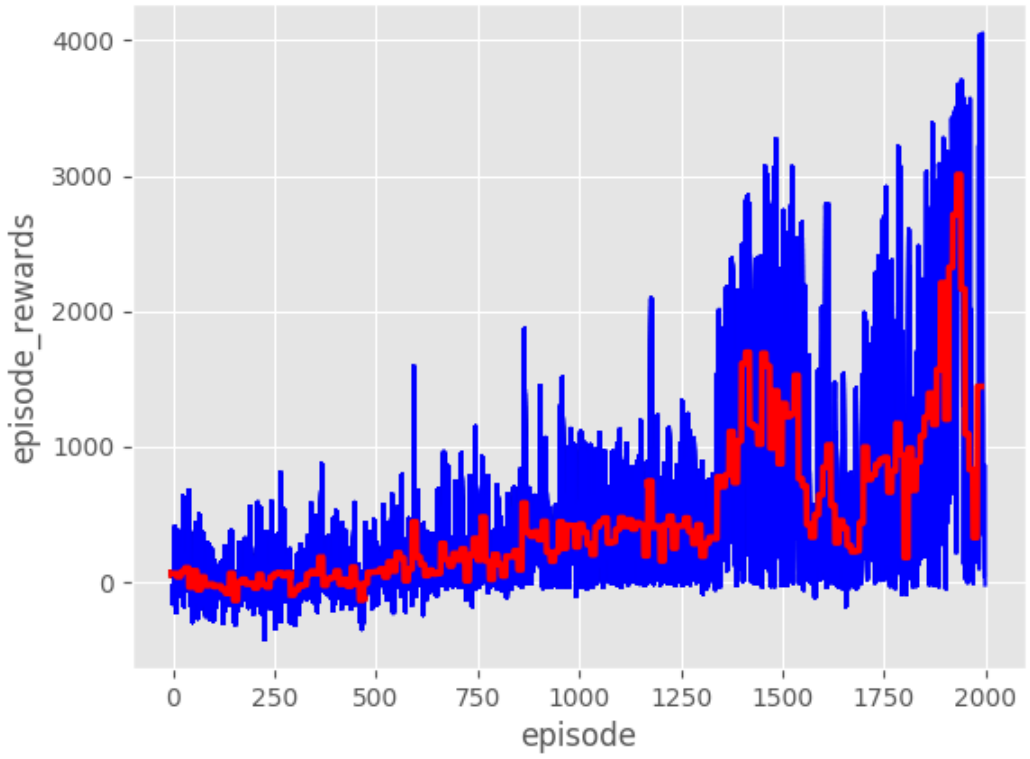

Figure 5.6. The reward history of training for full autonomous highway driving. (Blue line: the rewards of each time step; Red line: the rewards of every 10 time steps)

reward is attained after 2000 episodes. Until now, the trained agent has enough ability to drive the vehicle safely in multiple lanes using speed control and lane control. Due to the training time limit, more episodes are not provided here. Since it is still exploring the boundaries in the solution space, it has potential to gain higher reward with longer training.

Fig. 5.7 indicates that the action distribution after training. Most time it will stay at constant speed by action “Keep speed" (labeled as “2"). The agent will rarely choose to change lanes since it would be always safe to stay in the current lane as long as the ACC

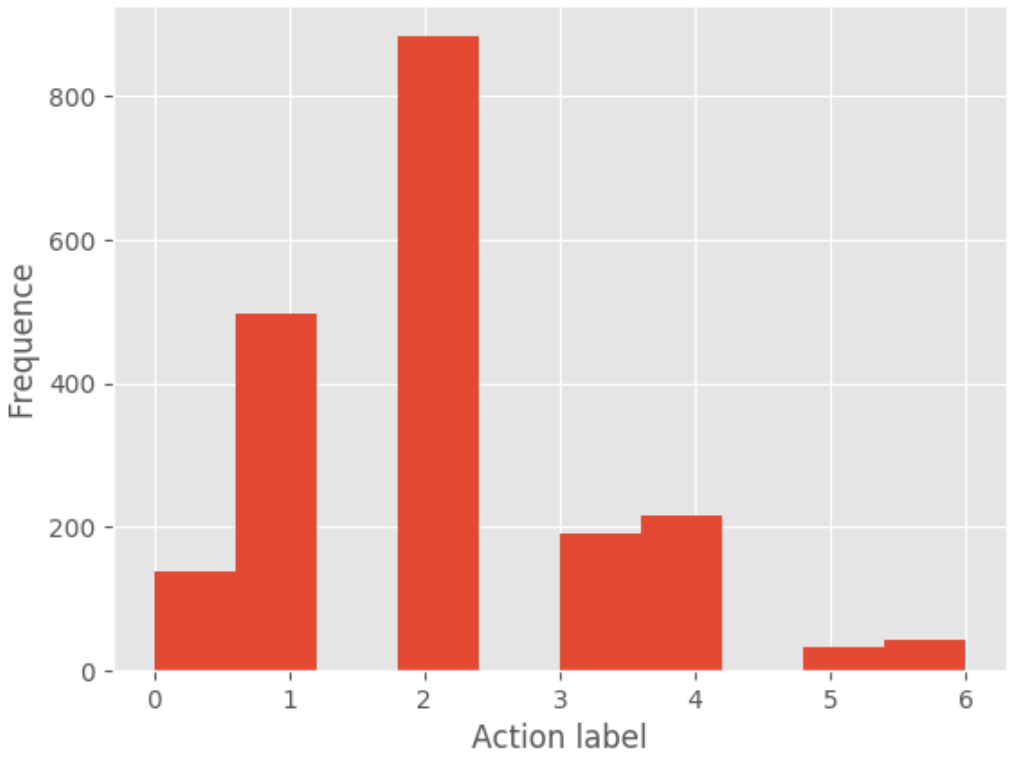

Figure 5.7. The action distribution after training.

is reliable. Sometimes it would take a try to change lanes and accelerate (labeled as “0" and “1") when no vehicle ahead in the new lane.

As shown in Fig. 5.8, it varied in a higher degree than that in Fig. 5.4. That is because when lane change is available or the action space is bigger the agent takes more time to make decisions and is more likely to make mistakes. Overall, it learned to maintain an equal average speed with preceding vehicle and accelerate when changing to a new lane and there is no vehicle right ahead. With more time training, it could be expected to perform better.

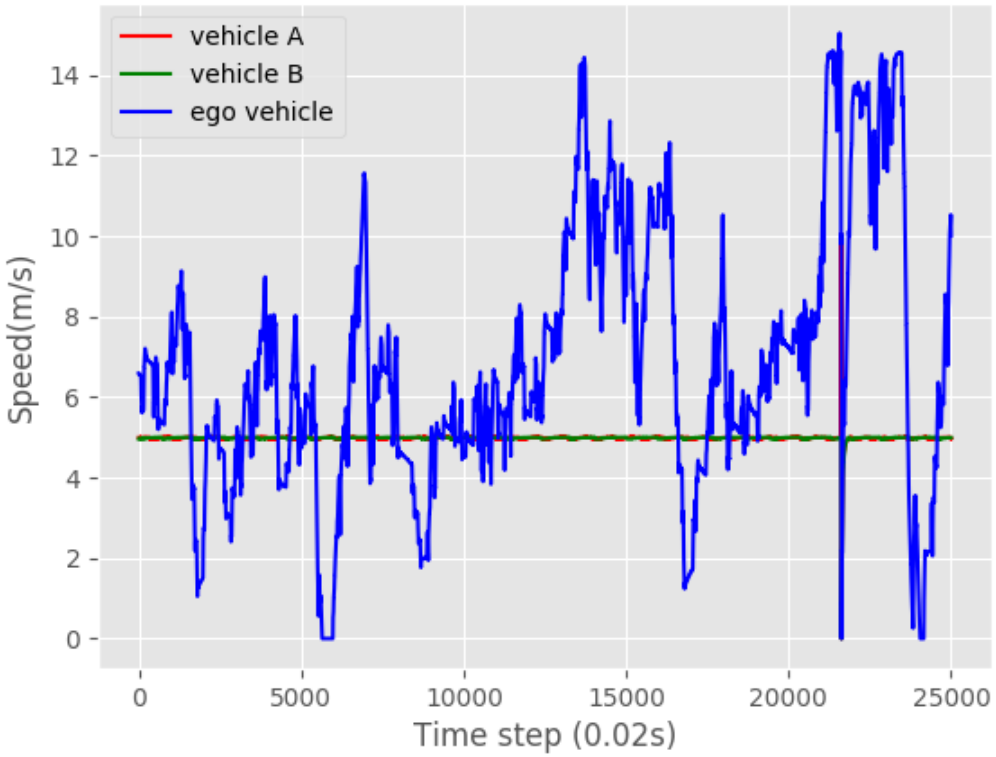

Figure 5.8. The speed variance after training.

5.4 Main Evaluation

Both $Agent1_{FCNN}$ and $Agent2_{FCNN}$ successfully drove the ego vehicle safely without collisions for a very long time and distance after enough training. Naturally, $Agent1_{FCNN}$ solved a significantly higher fraction of the episodes and performed better than $Agent2_{FCNN}$, since it only needed to control the speed, and not decide when to change lanes. In the beginning, it learned to always stay in its position to avoid an immediate collision, but quickly met a limit in receiving higher reward. With more training, it started to maintain a relatively high speed and keep stable if there is a preceding vehicle detected, but

sometimes caused collisions. $Agent2_{FCNN}$ needs much more training to stay safe. A longer training run could be carried out for better performance.

6 Conclusions

This chapter concludes the results intuitively, analyses the implementation and summarizes the difficulties and pipelines for future use.

6.1 Free-Form Visualization

Some Youtube video have been uploaded with the following links,

- https://youtu.be/RcjZSAFbHV0
- https://youtu.be/nEqEXBDs2Co

and it shows how it performs after enough training. It shows a capacity to stay in the lane adjusting its speed and choose a good timing to change lanes.

6.2 Analysis

As mentioned in Chapter 5, the simple reward functions were designed in a simple way. Naturally, the choice of reward function strongly affects the resulting behavior. For example, when no penalty was given for a lane change, the agent found solutions where it constantly demanded lane changes in opposite directions, which made the vehicle drive in between two lanes. In this study, a simple reward function worked well, but for other

cases a more careful design may be required. One way to determine a reward function is to use inverse reinforcement learning[39].

The method presented in this thesis requires no such hand crafted features, and instead uses the measured state. An important remark is that when training an agent by using the method presented in this thesis, the agent will only be able to solve the type of situations that it is exposed to in the simulations. It is therefore important that the design of the simulated traffic environment covers the intended case. Furthermore, when using machine learning to produce a decision making function, it is hard to guarantee functional safety. Therefore, it is common to use an underlying safety layer, which verifies the safety of a planned trajectory before it is executed by the vehicle control system.

6.3 Reflection

As experience which could be useful for future research on deep reinforcement learning, the difficulties and pipelines are summarized here.

6.3.1 Difficulties

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

All these techniques and parameters were selected by trial and error, and no systematic grid search was done due to the high computational cost. More than once it seemed that the implementation of the algorithms and techniques was incorrect, and it turned

out that the wrong parameters were being used. A "simple" change such as decreasing ϵ, or changing the neural network optimizer made big changes in the performance of the value function.

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

6.3.2 General Pipeline

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

- **Environment:** An environment defines what the agent interacts with. It receives states and actions and generate new states and plays a role as an online data generator.
- **State Space:** A state is the input of the Deep Q Network. In this thesis, 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 Function:** The reward function determines in which way we would like the agent to grow. For example, we would define a bigger reward for the car to stay in the middle of the road than in the side of the road. It can be a discrete or a continuous function.
- **Deep Neural Network:** The Deep Neural Network is responsible to map the states to the Q values, which are corresponding to different actions by Q functions.
- **Fine tune the Hyperparameters:** By fine tuning the hyperparameters, we try to maximize the ability of the defined Deep Q Network. It can be subtle to modify the hyperparameter values which might change the output in different ways, like effecting the time of the convergence, the prediction accuracy, overfitting or underfitting and robustness.

7 Future Work

The main results of this thesis show that a Deep Q-Network agent can be trained to make decisions in autonomous driving, without carefully preprocessed features. The generality of the method was demonstrated by applying it to a highway environment with longitudinal motion control and combined longitudinal and lateral motion control. In both cases, the trained agents are learning to stay safe and to change lanes with episodes. Collisions are rarely happen after enough long training.

Topics for future work include to further analyze the generality of this method by applying it to other cases, such as crossings and roundabouts, and to systematically investigate the impact of different parameters and network architectures. Moreover, it would be interesting to apply prioritized experience replay[40], which is a method where important experiences are repeated more frequently during the training process. This could potentially improve and speed up the learning process.

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

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

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

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

Bibliography

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

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

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

[4] A. Mukminin. Semi-autonomous nissan leaf cleared for road test in japan, 2013.

[5] Mario Herger. Baidu's open source self-driving platform apollo in detail, Oct 2017.

[6] H. Sudou and T. Hashizume. Adaptive cruise control system.

[7] D. Spero T. Pilutti M. Rupp and P. Joh. Autonomous lane control system. Ford Global Technologies LLC, 2012.

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

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

[10] 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.

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

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

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

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

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

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

[32] J. Kober J. A.Bagnell and J. Peters. Reinforcement learning in robotics: A survey. The International Journal of Robotics Research, 2013.

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

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

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

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

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

[38] R. Craig Coulter. Implementation of the pure pursuit path tracking algorithm. Technical Report CMU-RI-TR-92-01, Carnegie Mellon University, Pittsburgh, PA, January 1992.

[39] Andrew Y. Ng and Stuart J. Russell. Algorithms for inverse reinforcement learning. Proceedings of the Seventeenth International Conference on Machine Learning, pages 663–670, June 29-July 02 2000.

[40] T. Schaul et al. Prioritized experience replay. CoRR, 1511(05952), 2015.