# INDUSTRIAL TRAINING REPORT

**TRAINING ORGANIZATION: VEGA INNOVATIONS**

**PERIOD OF TRAINING: FROM 05/12/2022 TO 04/05/2023**

**FIELD OF SPECIALIZATION: COMPUTER ENGINEERING**

**H.S.C. SILVA**

**E/17/331**

# ACKNOWLEDGMENTS

I would like to extend my deepest gratitude to all those who have contributed and played a significant role in making my industrial training at Vega Innovations a resounding success. During my undergraduate engineering studies, being a part of Vega Innovations offered abundant opportunities for personal growth and development. The experience at Vega Innovations equipped me with the valuable skill of creative problem-solving within time constraints, which holds immense importance. Having the chance to work at Vega Innovations during my initial industrial training is something I truly value as a significant opportunity. This section is dedicated to expressing my heartfelt appreciation to the individuals and institutions that have supported me throughout this enriching experience.

First and foremost, I am immensely grateful to the staff of the Department of Computer Engineering at the University of Peradeniya, Sri Lanka. Their unwavering commitment to providing us with comprehensive knowledge, invaluable guidance, and the necessary skills to tackle real-world industry challenges has been instrumental in shaping my professional growth. I am indebted to their dedication and passion for nurturing aspiring engineers.

I would also like to express my sincere appreciation to the Industrial Training and Career Guidance Unit (ITCGU) of the University of Peradeniya and the National Apprentice and Industrial Training Authority (NAITA). Their guidance and unwavering support throughout the internship period have been invaluable. Their efforts in bridging the gap between academia and industry have provided us with a solid foundation to excel in the professional realm.

A special word of thanks goes to Mr. Chamath de Silva, a dear schoolmate of mine and an Electronics Engineer at Vega Innovations. I am truly grateful to him for providing me with this exceptional internship opportunity within the Autonomous Vehicle Division of Vega Innovations.

I would also like to extend my gratitude to the entire HR team at Vega Innovations, especially Mr. Nisal Dinuke and Miss Nimodhi Wickramasinghe, for their support, professionalism, and seamless coordination throughout the internship. Their friendly and welcoming approach made me feel at home within the organization from day one.

Furthermore, I am deeply indebted to Mr. Nisal Bulathsinghala, the Tech Lead at Vega Innovations. His expertise, mentorship, and constant guidance have been instrumental in shaping my technical skills and broadening my understanding of the industry. His patience and willingness to share knowledge have made a lasting impact on my professional development.

Lastly, I would like to express my heartfelt appreciation to my training supervisor, Mr. Gimhan Dayarathne. His dedication, expertise, and continuous guidance throughout the 20 weeks of training were truly invaluable. His mentorship not only enhanced my technical skills but also instilled in me a sense of professionalism and a strong work ethic.

I am immensely grateful to each, and every person mentioned above, as well as all the individuals who have supported me behind the scenes. Your contributions have played an integral role in shaping my experience, and I am truly honored to have had the opportunity to learn from such remarkable individuals.

# CONTENTS

# LIST OF FIGURES

# LIST OF TABLES

## LIST OF ABBREVIATIONS

| | |
|---|---|
| **AI** | Artificial Intelligence |
| **API** | Application Programming Interface |
| **ARM** | Advanced RISC Machines |
| **ATV** | All-Terrain Vehicle |
| **BEV** | Bird’s Eye View |
| **CAN** | Controller Area Network |
| **CARLA** | Car Learns to Act |
| **CEO** | Chief Executive Officer |
| **CUDA** | Computer Unified Device Architecture |
| **DL** | Deep Learning |
| **DNN** | Deep Neural Network |
| **EV** | Electric Vehicle |
| **EVCU** | Electric Vehicle Control Unit |
| **EVX** | Electric Vehicle (model) X |
| **GIE** | GPU Inference Engine |
| **GMSL** | Gigabit Multimedia Serial Link |
| **GPU** | Graphical Processing Unit |
| **HD** | High Definition |
| **HDMI** | High-Definition Multimedia Interface |
| **IC** | Integrated Circuit |
| **ICCV** | International Conference on Computer Vision |
| **IL** | Imitation Learning |
| **LiDAR** | Light Detection and Ranging |
| **LIN** | Local Interconnect Network |
| **LPDDR** | Low-Power Double Data Rate |
| **NMC** | Nickel, Manganese and Cobalt |
| **ONNX** | Open Neural Network Exchange Format |
| **OTA** | Over-the-air |
| **PCB** | Printed Circuit Board |
| **RC** | Remote Controlled |
| **ResNet** | Residual Network |

| | |
|---|---|
| **RGB** | Red, Green, and Blue (refers to a system representing the colors used on a digital screen) |
| **RL** | Reinforcement Learning |
| **RTOS** | Real-time Operating System |
| **SDK** | Software Development Kit |
| **SMD** | Surface Mount Devices |
| **SMT** | Surface Mount Technology |
| **SoC** | System on Chip |
| **TFLOPS** | Trillion Floating-point Operations per Second |
| **TOPS** | Tera Operations per Second |
| **TRT** | Tensor Runtime |
| **UART** | Universal Asynchronous Receiver/Transmitter |
| **UE4** | Unreal Engine 4 |
| **USB** | Universal Serial Bus |
| **ViT** | Vision Transformer |

# Chapter 1

# INTRODUCTION

## 1.1 TRAINING SESSION

The training session for my industrial internship was conducted at Vega Innovations, located at Bay 6, Trace Expert City, Tripoli Market, Maradana. The internship commenced on May 12, 2022, and concluded on May 5, 2023. Throughout this 20-week period, I had the privilege of immersing myself in the dynamic and innovative environment of Vega Innovations, gaining firsthand experience in their Autonomous Vehicle Division. The training provided me with invaluable insights into cutting-edge technologies and industry practices, allowing me to enhance my skills and knowledge in a practical setting.

## 1.2 INTRODUCTION TO TRAINING ORGANIZATION

Vega Innovations, a distinguished electric automobile manufacturing organization based in Colombo, Sri Lanka, stands as a dynamic research and development branch under the esteemed umbrella of CodeGen International (Pvt.) Ltd. Founded in 2013 by Dr. Harsha Subasinghe, the visionary CEO of CodeGen International, and co-founder Dr. Beshan Kulapala, Vega Innovations embodies a collective dedication to technological advancement and engineering excellence. By leveraging the vast resources and expertise within CodeGen International, Vega Innovations has successfully manifested its commitment to crafting groundbreaking, all-electric supercars. Notably, the remarkable debut of the Vega EVX, as shown in Figure 1.3, at the prestigious 2020 Geneva Motor Show solidified their prowess in sustainable mobility and innovation. See Figure 1.1 and Figure 1.2 for the logos of Vega Innovations and CodeGen International respectively.

VEGA INNOVATIONS
INFINITE IMAGINATION

*Figure 1.1 Logo of Vega Innovations*

CODEGEN
www.codegen.co.uk

*Figure 1.2 Logo of CodeGen International*

Renowned for its pioneering advancements in electric vehicle technology, Vega Innovations has garnered acclaim for its revolutionary approach to the design and manufacturing of the all-electric supercar. Presently, the production-level specifications for this remarkable vehicle are as follows:

- TORQUE: A formidable 760Nm.
- POWER: Impressive 804hp.
- ACCELERATION: Swift 0-100km/h in just 3.1 seconds.
- RANGE: A substantial 300km, facilitated by the utilization of a 55kWh NMC Gen 2 Battery Pack.

*Figure 1.3 Vega EVX and the Team at Geneva Motorshow 2020*

As Vega Innovations continues to amass acclaim through the production of groundbreaking electric supercars, its horizons expand further to encompass the development of an array of vehicles. In the pipeline for imminent market release are ATVs and TUK-TUKs, showcasing the company's commitment to diversifying their product portfolio. Moreover, Vega Innovations actively conducts extensive research in areas such as EV charging, energy storage, and generation technologies. It is worth noting that chargeNET and AIGrow, similar to Vega Innovations, serve as subsidiary research and development branches under the esteemed umbrella of CodeGen. chargeNET focuses on advancements in EV charging technologies, while AIGrow specializes in agricultural growth and farming innovations. Together, these three entities synergistically contribute to CodeGen's multifaceted expertise and innovation-driven approach.

### 1.2.1 Organizational Structure

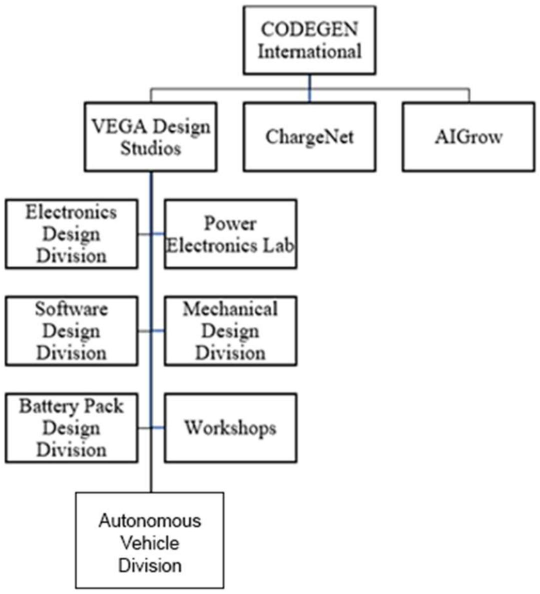

*Figure 1.4 Organizational Structure*

Figure 1.4 visually presents the organizational structure of the CodeGen International Group of Companies. Within the premises of Vega Innovations, several departments and divisions collaboratively operate to drive innovation and excellence. These include the Electronics Design Division, Power Electronics Lab, Software Design Division, Mechanical Design Division, Battery Pack Design Division, Workshop, and Autonomous Driving Division. Together, these specialized units form a cohesive framework that enables Vega Innovations to effectively pursue its mission and deliver cutting-edge solutions in the field of electric vehicle technology.

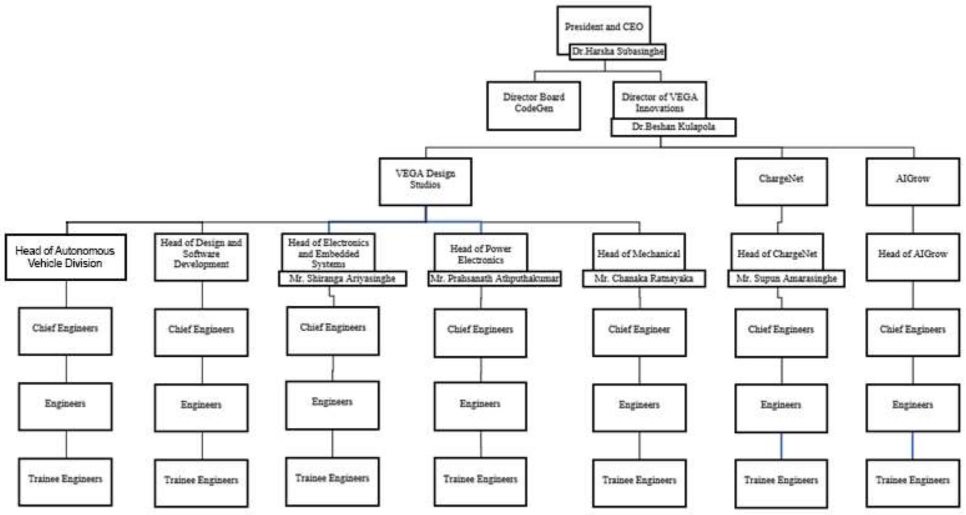

*Figure 1.5 Structure of the Engineering Team within Vega Innovations*

Figure 1.5 illustrates the composition of the engineering team at Vega Innovations, showcasing a well-defined structure within Vega Design Studios. The engineering team is organized into five distinct teams, each specializing in a specific area:

1. Software Development
2. Electronics and Embedded Systems
3. Power Electronics
4. Mechanical Engineering
5. Autonomous Vehicle (AI)

Within each team, a hierarchy is established, led by a head engineer overseeing the operations. Accompanying them are chief engineers, engineers, and trainees, collectively contributing their expertise and skills to propel the innovative endeavors undertaken by Vega Innovations. This structured approach ensures efficient coordination and collaboration among the engineering teams, fostering an environment conducive to technological advancement and groundbreaking achievements.

### 1.2.2 Mission

Powering An Extraordinary Future with Infinite Possibilities. Crafting high quality, intelligent technology solutions by a community of dynamic professionals through continuous research and knowledge sharing to optimize business processes of corporate enterprises.

### 1.2.3 Vision

"With Infinite Imaginations We Can Achieve Greatness Together."

## 1.3 SUMMARY OF THE WORK ENGAGED IN TRAINING

During my training at Vega Innovations, I gained diverse experiences. I started by soldering an SMD PCB for the Vega ETX (the EVCU of the Vega ETX), followed by reading a research paper called NEAT: Neural Attention Fields for Autonomous Driving. This led me to explore various deep learning approaches to autonomous driving and work with a simulator called CARLA using the Python API. I then developed autonomous driving architectures and worked with GPU-accelerated embedded systems. Additionally, I was able to attempt to make a commercial RC car drive autonomously as a prototype. This training provided valuable hands-on experience and deepened my passion for the field of autonomous vehicles.

# Chapter 2

# SMD SOLDERING

## 2.1 INTRODUCTION

During the initial week of my internship at Vega Innovations, I had the invaluable opportunity to delve into the world of SMD soldering under the expert guidance of Eng. Chamath de Silva. The focal point of this chapter is the immersive experience of soldering a prototype testbench PCB for the EVCU of the Vega ETX. As a Computer Engineering trainee, this hands-on encounter with SMD soldering and SMT components proved to be an exceptional and rare opportunity, considering that Computer Engineering interns typically focus on software-related tasks. This chapter explores the significance of SMD soldering, highlighting the practical application of engineering knowledge in assembling PCBs and showcasing the unique experience of stepping into the world of hardware engineering.

## 2.2 SMT COMPONENTS

In the realm of electronic components, two main types are commonly encountered: Through-hole Components and Surface Mount Devices (SMDs). While through-hole components require leads to be inserted through drilled holes in the PCB and soldered on the opposite side, SMDs offer a more compact alternative. Most of the components employed in PCB assembly are Surface Mount Devices (SMDs). These SMDs are soldered onto a single side of the PCB, optimizing the utilization of space. Figure 2.1 showcases several commonly used SMD components, akin to the ones I soldered during my time at Vega Innovations. These components encompass passive elements like resistors, capacitors, inductors, and fuses, as well as active components such as transistors, regulators, and integrated circuits (ICs).

*Figure 2.1 Commonly used SMDs*

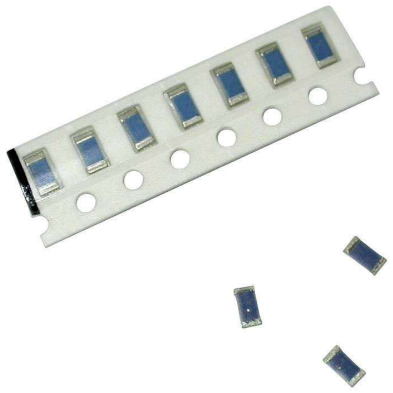
*Figure 2.2 SMD Components in a cut tape*

SMD components are typically obtained in bulk quantities, often in the form of reels. However, when procuring smaller quantities (<5000), suppliers offer the parts in "cut tapes," as depicted in Figure 2.2. As the prototype development undertaken at Vega Innovations necessitated only a limited number of components of a specific kind, the procurement process involved acquiring all the required components in this manner.

## 2.3 SOLDERING WORKBENCH

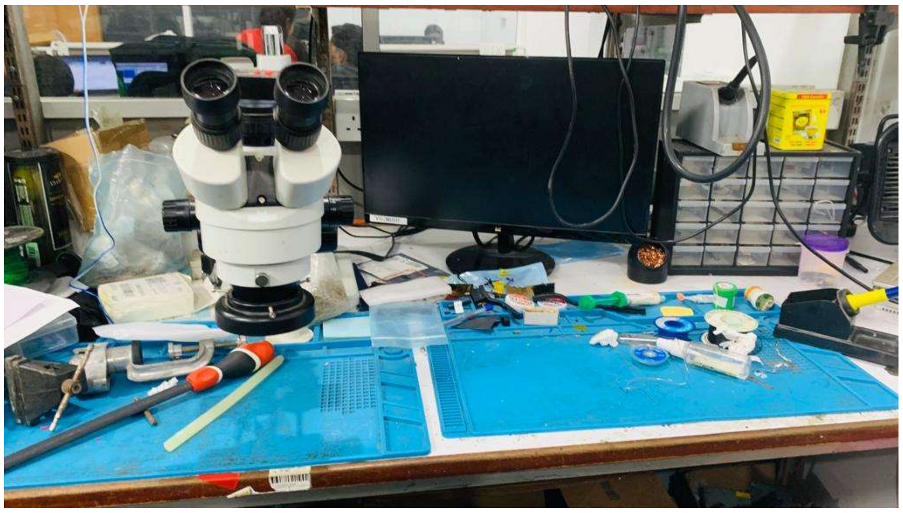
*Figure 2.3 Soldering Workbench*

Vega Innovations employed a dedicated soldering workbench, as illustrated in Figure 2.3, for the PCB soldering process. To safeguard the tabletop from heat damage, a silicone heat-resistant mat was utilized. This mat not only provided thermal insulation but also featured compartments for storing SMD components before soldering, along with a magnetic area for secure component placement. The pivotal component of the soldering workbench was the soldering station, comprising a control unit, soldering iron, and hot air gun. Soldering stations, including those utilized at Vega Innovations, often incorporated two separate seven-segment displays to indicate the temperature of the soldering iron and hot air gun. The control unit offered buttons and knobs to adjust the temperature set point for both tools and regulate the air flow intensity of the hot air gun. Additionally, anti-static tweezers were employed to handle and position components during the soldering process. These tweezers featured a specialized coating designed to prevent the accumulation of static electricity, mitigating the risk of damaging sensitive electronic components like ICs during soldering.

## 2.4 SOLDERING PROCESS

When it comes to soldering SMD components, there exist two primary methods:

1. Utilizing a soldering iron and solder wire.
2. Employing soldering paste in conjunction with a hot air gun.

The former method is commonly employed for soldering components with two or three terminals, while the latter method is preferred for components with a higher pin count, such as ICs.

### 2.4.1 Soldering iron and solder wire method

The solder wire utilized in this soldering technique comprises a blend of 63% Tin (Sn) and 37% Lead (Pb). This specific ratio is chosen due to its ability to form a eutectic alloy, wherein the mixture of metals melts at a single temperature lower than the individual melting points of each metal. Furthermore, the solder wire is equipped with a flux core, typically composed of Rosin flux. This flux acts as a reducing agent, effectively preventing the formation of metal oxides during the soldering process. As a result, a robust metal-to-metal bond can be established between the components being soldered, facilitating a reliable connection.

To apply this soldering technique, the initial step involves heating one pad using the soldering iron while simultaneously introducing a small amount of solder by feeding the wire onto the pad. As illustrated in Figure 2.4, the appropriate component is then carefully positioned onto the heated pad using tweezers. Once the component is securely attached to the pad, the remaining terminal(s) are heated using solder wire to finalize the soldering procedure. It is crucial to execute this entire process within a few seconds, as prolonged heating can potentially harm the PCB layers and the components themselves.

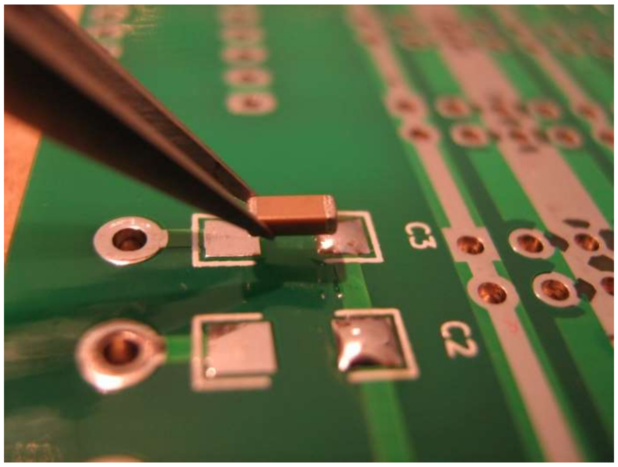

*Figure 2.4 Technique of Placing an SMT Components*

### 2.4.2 Soldering paste and hot air gun method

This method is typically used to solder components with many pins, as it is difficult to do it with a soldering iron. A soldering paste containing an emulsion lead/tin mixture similar to that found in solder wire, mixed with soldering flux is used. This paste is applied using a syringe shown in Figure 2.5, onto the soldering pads. Next the relevant component is placed on top of the pads using a tweezer, and heat is applied from a hot air gun. This will cause the lead/tin metal alloy to melt and bond the pads to the pins of the component. When applying heat from the hot air gun, it is important to hold it directly above the component in a vertical manner, to prevent the components being blown away by the air flow. The air flow intensity of the blower is also typically set to low value.

*Figure 2.5 Soldering Paste*

## 2.5 SAFETY PRACTICES

At Vega Innovations, adherence to rigorous safety protocols during soldering was of paramount importance. Recognizing the potential harm posed by inhaling fumes produced during the burning of soldering flux, a mask, as depicted in Figure 2.6, was consistently worn during the soldering process. Furthermore, to safeguard against eye irritation caused by soldering fumes, protective goggles, as illustrated in Figure 2.7, were diligently utilized. These measures ensured the well-being and protection of individuals involved in soldering activities.

*Figure 2.6 Mask worn during Soldering*

*Figure 2.7 Protective goggles worn during Soldering*

Furthermore, to enhance safety measures, a fume extractor, similar to the one depicted in Figure 2.8, was actively utilized at the soldering workbench. This apparatus effectively drew soldering fumes away from the vicinity of the individual working, minimizing their exposure to potentially harmful fumes.

*Figure 2.8 Fume Extractor*

## 2.6 QUALITY ASSURANCE & ETHICAL PRACTICES

During the first few weeks of my internship when I was working on SMD soldering, I had the opportunity to witness and participate in the rigorous Quality Assurance practices implemented by the organization. Vega Innovations upheld the highest standards of quality in soldering processes, ensuring that each solder joint met precise specifications and quality benchmarks. This commitment to quality assurance included regular inspections, testing, and calibration of soldering equipment, guaranteeing the reliability and integrity of electronic components. Additionally, ethical practices and professionalism were paramount throughout the soldering tasks. We followed strict safety protocols, such as wearing protective gear and utilizing fume extractors, to create a safe and healthy work environment. Respect for colleagues and a dedication to teamwork were consistently upheld, contributing to a collaborative and professional atmosphere conducive to effective learning and skill development.

## 2.7 EXPERIENCE GAINED

Valuable experience and insights were gained through my short period of engagement in SMD soldering during the internship at Vega Innovations. Firstly, hands-on expertise in the intricate process of soldering SMD components was acquired, familiarizing myself with the techniques and equipment involved. The complexities associated with assembling PCBs were revealed, and the importance of precision and attention to detail was highlighted. Additionally, the opportunity to work with a diverse range of SMD components, including both passive and active elements, deepened my knowledge of their characteristics and applications. An understanding of the significance of selecting appropriate soldering materials, such as the specific solder wire composition and the flux core, to achieve reliable and robust solder joints was developed. Furthermore, strict adherence to safety protocols, such as the wearing of masks, protective goggles, and utilization of fume extractors, led to a heightened awareness of the significance of maintaining a safe and healthy work environment. Overall, the experience gained in SMD soldering not only honed my technical skills but also fostered a sense of responsibility and professionalism in industrial settings.

# Chapter 3

# THE CARLA SIMULATOR, AUTONOMOUS DRIVING AND NEAT

## 3.1 CARLA SIMULATOR

The CARLA simulator, as illustrated in Figure 3.1, is a powerful open-source platform that is built upon the Unreal Engine 4 (UE4) gaming engine, offering a wide range of materials and features for comprehensive autonomous driving simulation. One notable aspect of CARLA is its inclusion of various sensor simulations, such as LiDAR, depth maps (emulating stereo cameras like ZED or Intel's D435), and semantic segmentation data. Additionally, CARLA provides a realistic environment with ten towns, complete with pedestrians, other vehicles, traffic lights, speed limits, and authentic layouts.

*Figure 3.1 A snapshot of the CARLA Simulator Software*

Notably, CARLA offers extensive resources for development and testing, including repositories for imitation learning and reinforcement learning models. It provides options for running simulations without a display, allowing for efficient processing and testing. However, CARLA's strengths extend beyond its technical capabilities. The platform boasts an active and supportive community, comprehensive documentation, and user-friendly interfaces. These factors contribute to CARLA's popularity and ease of use among researchers and developers in the autonomous driving field.

Furthermore, CARLA is specifically designed to facilitate the development, training, and validation of autonomous driving systems. Alongside its open-source code and protocols, CARLA offers a collection of open digital assets, including urban layouts, buildings, and vehicles, specifically created for autonomous driving simulations. These assets are freely available and can be utilized for various purposes.

CARLA also provides a versatile simulation platform, enabling users to specify sensor suites, define environmental conditions, exert full control over static and dynamic actors, and generate custom maps.

Overall, the CARLA simulator serves as a comprehensive tool for researchers and developers in the autonomous driving domain, offering a rich set of features, a supportive community, and a user-friendly interface. Its commitment to open-source principles, coupled with its extensive resources, make CARLA an asset for advancing autonomous driving technologies and conducting cutting-edge research in the field.

The CARLA simulator utilizes a scalable client-server architecture, as shown in Figure 3.2, ensuring efficient and realistic simulation. The server component takes charge of various simulation aspects, including sensor rendering, physics computation, and updating the world state and its actors. For optimal performance, running the server with a dedicated GPU is highly recommended, especially when working with deep learning techniques. Figure 3.3 shows the workstation computer used in Vega Autonomous Vehicle division that consists of a NVIDIA RTX 4080 GPU.

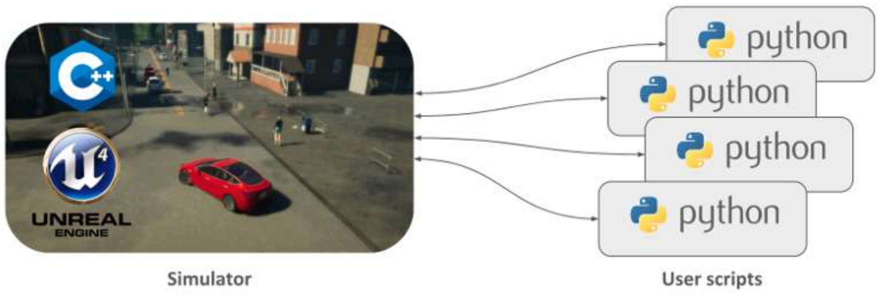

*Figure 3.2 CARLA Simulator's Client-Server Architecture*

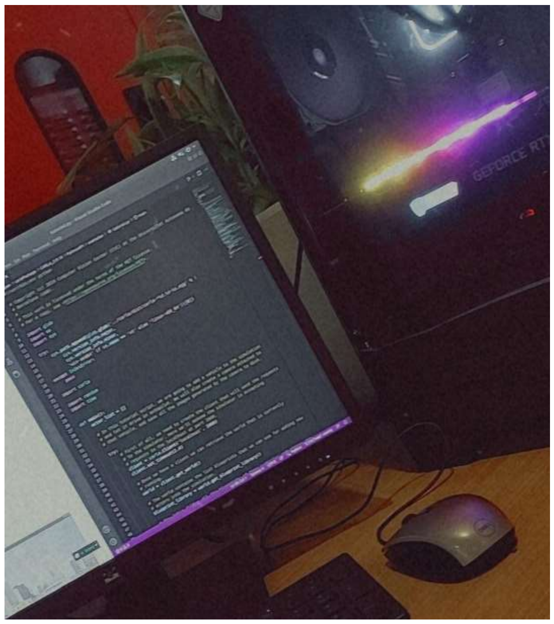

*Figure 3.3 Workstation Computer in the Vega Autonomous Vehicle Division*

On the client side, multiple client modules control the behavior of actors within the simulation and establish the desired world conditions. This is achieved through the utilization of the CARLA API, available in Python or C++. The CARLA API acts as a mediator between the server and client, facilitating seamless communication and constantly evolving to offer new functionalities.

## 3.2 DEEP LEARNING APPROACHES TO AUTONOMOUS DRIVING

Self-driving vehicles encompass six different levels of autonomy, ranging from drivers being in full control to full automation, as illustrated in Figure 3.4. According to Statista, the market for autonomous vehicles in levels 4 and 5 is projected to reach $60 billion by 2030. This research also indicates that before fully autonomous vehicles become widespread, approximately 73% of the total number of cars on the roads will possess some degree of autonomy.

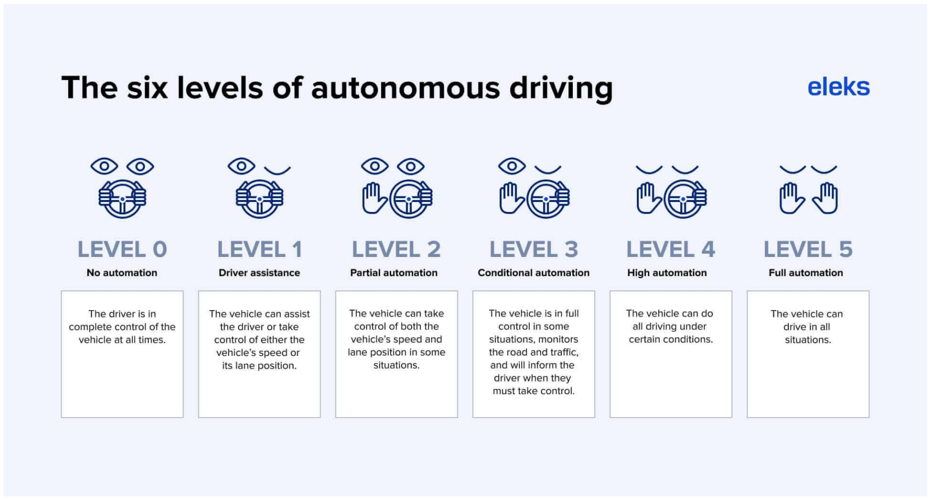
The six levels of autonomous driving
eleks
LEVEL 0
No automation
The driver is in complete control of the vehicle at all times.
LEVEL 1
Driver assistance
The vehicle can assist the driver or take control of either the vehicle's speed or its lane position.
LEVEL 2
Partial automation
The vehicle can take control of both the vehicle's speed and lane position in some situations.
LEVEL 3
Conditional automation
The vehicle is in full control in some situations, monitors the road and traffic, and will inform the driver when they must take control.
LEVEL 4
High automation
The vehicle can do all driving under certain conditions.
LEVEL 5
Full automation
The vehicle can drive in all situations.

*Figure 3.4 The six levels of Autonomous Driving*

As the autonomous driving industry continues to advance, machine learning and deep learning technologies have emerged as key enablers for achieving more sophisticated and capable autonomous vehicles. These advancements are underpinned by the four fundamental pillars of autonomous driving: perception, localization, planning, and control. In this section, we will explore how deep learning has become the go-to approach for implementing these pillars, offering enhanced perception, precise localization, intelligent planning, and adaptive control capabilities in autonomous driving systems.

The utilization of deep learning in autonomous driving has proven to be highly effective due to its ability to handle complex and unstructured data, such as images from cameras, LiDAR point clouds, and sensor readings. Traditional approaches to autonomous driving relied on handcrafted algorithms and rules, which often struggled to capture the intricacies of real-world scenarios. Deep learning models, on the other hand, can learn directly from raw data, automatically extracting relevant features and representations, enabling the development of more accurate and adaptable autonomous systems.

### 3.2.1 The Autonomous Driving Stack

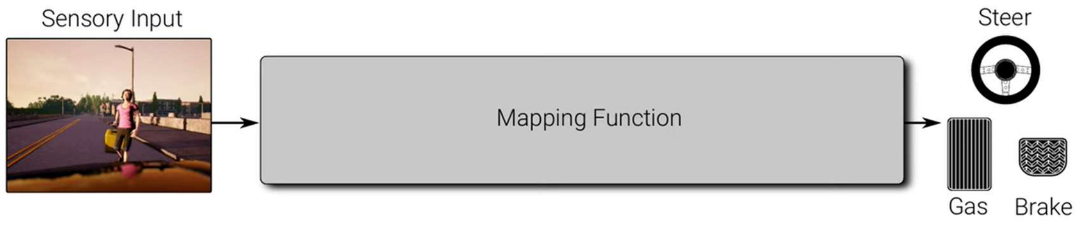

*Figure 3.5 The Autonomous Driving Stack*

Before exploring the deep learning approaches for self-driving, it is important to grasp the overall structure of the autonomous driving stack. The autonomous driving stack, as depicted in Figure 3.5, can be conceptualized as a function that receives a high-dimensional sensory input, including elements like RGB images, LiDAR scans, Radar data, Ultrasonic measurements, or a combination of these inputs. The objective of this function is to generate a low-dimensional output that is both highly robust and dependable. Building this black-box mapping function from the intricate high-dimensional sensory input to the concise low-dimensional output is the fundamental task at hand.

### 3.2.2 Dominating Paradigms

In the deep learning-based autonomous driving literature, three dominating paradigms have emerged to tackle the task of constructing the black-box mapping function within the autonomous driving stack: *Modular Pipeline*, *End-to-End learning*, and *Direct Perception*. The following sections will delve into the details of each paradigm, providing a comprehensive understanding of their principles and applications in the context of autonomous driving.

(1). Modular Pipeline

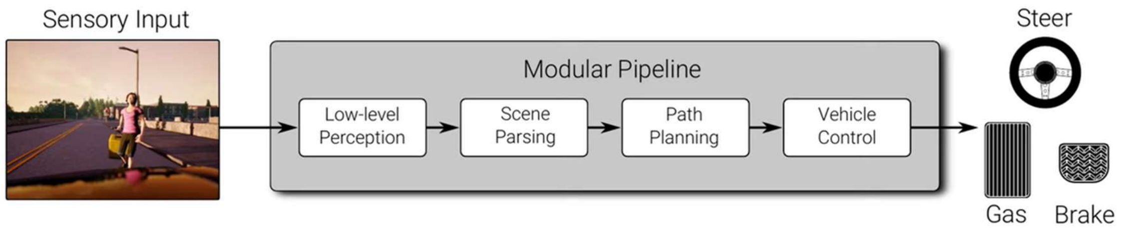

*Figure 3.6 Modular Pipeline Workflow*

The Modular Pipeline paradigm involves breaking down the autonomous driving problem into modular components such as perception, localization, planning, and control. Each component is individually designed and optimized using deep learning techniques.

*Table 3.1 Pros and Cons of Modular Pipeline*

| **Advantages** | **Drawbacks** |
|---|---|
| Small components, easy to be developed individually, in parallel. | Piece-wise training is needed (joint End-to-End training is not allowed). |
| Interpretable since all the modules are developed by humans and the interfaces between the modules are well-defined. | All the modules are trained independently on different sets of data than the actual self-driving task. i.e., the modules learn to optimize different tasks rather than optimizing the unified driving task and thus the model do not generalize well. |
| | Localization and planning heavily relies on ***HD maps**. |

***HD maps** are highly accurate maps used in autonomous driving containing details not normally present on traditional maps such as, centimeter precise lanes, markings, traffic lights/signs. Since these maps are human annotated, they are expensive to create.

(2). End-to-End Learning

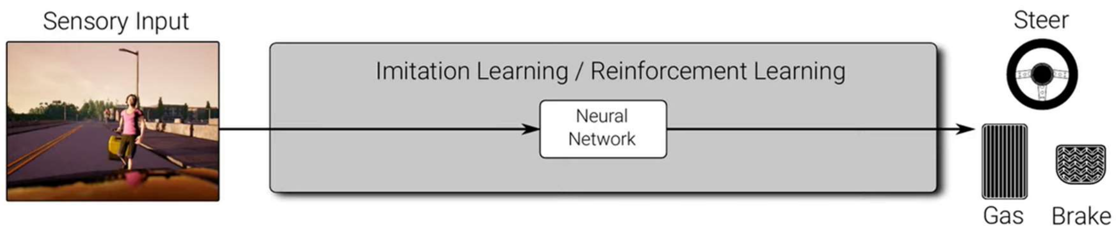

*Figure 3.7 End-to-End Learning Workflow*

The end-to-end learning paradigm in deep learning-based autonomous driving aims to simplify the autonomous driving stack by directly learning a monolithic function that maps high dimensional sensory inputs, such as images, LiDAR scans, or sensor fusion data, to low dimensional output commands, such as gas and brake commands. This approach typically involves training a neural network (using Imitation Learning or Reinforcement Learning) to directly learn the complex mapping from input to output.

*Table 3.2 Pros and Cons of End-to-End Learning*

| **Advantages** | **Drawbacks** |
|---|---|
| End-to-End training is possible. This allows for a more holistic optimization process where the focus is on optimizing the system's performance on the driving task itself rather than optimizing individual components separately. | Do not generalize as well as modular pipelines since the model becomes highly specialized in reproducing the behavior observed during training. |
| Cheap annotations. | Interpretability is lacking as the model learns a monolithic function. |

(3). Direct Perception

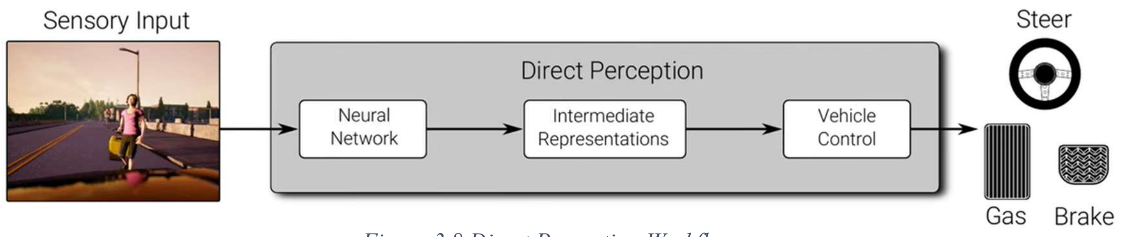

*Figure 3.8 Direct Perception Workflow*

Direct Perception is a paradigm that combines the advantages of both modular pipelines and end-to-end learning models in the context of autonomous driving. It offers a hybrid approach where a modular pipeline is utilized, but with the added benefit of end-to-end trainability. In this paradigm, the autonomous driving system consists of multiple modules that work in conjunction to achieve the desired outcome.

The first neural network in the Direct Perception framework takes the high-dimensional sensory input, and processes it to generate an intermediate representation. This intermediate representation captures important features and information from the input data, acting as a condensed and informative representation of the driving environment. This step allows for a level of abstraction and feature extraction, similar to the modular pipeline approach.

The vehicle control module, which is responsible for generating low-dimensional vehicle controls such as steering, acceleration, and braking, receives the intermediate representation as its input. This module can be implemented using classical vehicle control algorithms or neural networks, depending on the specific requirements and design choices. By using the intermediate representation, the vehicle control module can make informed decisions based on the processed sensory data.

*Table 3.3 Pros and Cons of Direct Perception*

| **Advantages** | **Drawbacks** |
|---|---|
| Compact representation. | Vehicle control is not typically learned jointly. |
| Interpretability. | How to choose the intermediate representations? |

### 3.2.3 Reinforcement Learning and Imitation Learning

In the field of Reinforcement Learning (RL), the agent-environment interaction revolves around following a policy to make decisions. The agent, in each state of the environment, takes actions based on the policy and receives a reward, subsequently transitioning to a new state. The objective in RL is to learn an optimal policy that maximizes the long-term cumulative rewards. However, RL teaching processes face certain challenges.

One of the challenges is the difficulty in determining an appropriate reward function that accurately reflects the true performance objectives. Designing a reward function that effectively captures the desired behavior can be a complex task. Additionally, rewards in RL can often be sparse, meaning that they are only received in specific situations or at certain milestones. For example, in a game, a reward may be given only when the game is won or lost, making it challenging to provide continuous feedback to guide learning.

Imitation Learning (IL) offers a feasible solution to these challenges. Instead of relying on sparse rewards or manually specifying a reward function, IL leverages demonstrations provided by an expert, typically a human. The expert demonstrates the desired behavior by showcasing a set of actions in various states. The agent then aims to learn the optimal policy by imitating and following the expert's decisions (i.e., IL has a data-driven learned policy instead of a hard-coded one). This approach enables the agent to learn from the expert's knowledge and expertise, reducing the complexity associated with reward engineering. Figure 3.9 provides an overview of the imitation learning process, highlighting the interaction between the expert and the learning agent.

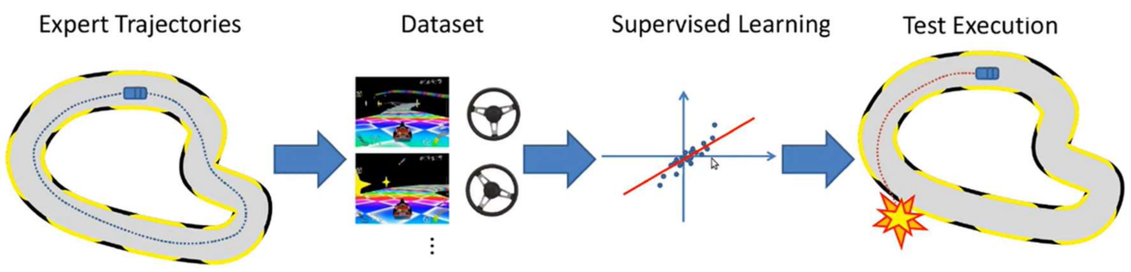

*Figure 3.9 Imitation Learning Overview*

IL encompasses three main variants: behavioral cloning, direct policy learning via an interactive demonstrator, and inverse reinforcement learning. These variants offer different approaches to leverage expert demonstrations in the learning process. IL offers a valuable alternative when it is easier for an expert to demonstrate the desired behavior compared to specifying a reward function or directly learning the policy. It leverages the expertise of the demonstrator to guide the learning process, enabling the agent to acquire the desired behavior efficiently.

## Formal Definition of Imitation Learning

The main component of IL is the environment, which is essentially a Markov Decision Process (MDP). In other words, the environment has,

- State: $s \in S$ (may be fully observed or partially observed – typically partially observed)
- Action: $a \in A$ (may be discrete/continuous. Example: turn angle, speed)
- Transition Model: $P(s'|s, a)$ (which is the probability that an action $a$ in the state $a$ leads to state $s'$)
- Unknown Reward: $R(s, a)$
- Policy: $\pi_\theta : S \to$ A (need to learn the policy parameters $\theta$)
- Expert's Demonstrations (trajectories): $\tau = \{(s_0^*, a_0^*), (s_1^*, a_1^*), \cdots\}$
- Optimal Policy: $\pi^* : S \to A$ (provided by the expert demonstrator)
- Rollout: Given $s_0$, sequentially execute $a_i = \pi_\theta(s_i)$ and sample $s_{i+1} \sim P(s_{i+1}|s_i, a_i)$ yields trajectory $\tau = \{(s_0^*, a_0^*), (s_1^*, a_1^*), \cdots\}$
- Loss function: $\mathcal{L}(a^*, a)$

## General Imitation Learning

The general imitation learning problem can be mathematically formulated as follows,

$$argmin_{\theta}\ E_{s\sim P(s|\pi_{\theta})}[\mathcal{L}(\pi_{\theta}^{*}(s), \pi_{0}(s)]$$

In other words, the problem is to find the parameters $\theta$, by minimizing the expectation of the loss that compares the expert's actions at a particular state, to the decision that's made by the agent's policy, over all states that are reachable by the policy $\pi_{\theta}$.

## Behavioral Cloning

Behavioral cloning represents the simplest form of imitation learning, aiming to replicate the expert's policy through supervised learning techniques. The key idea behind behavioral cloning is to treat the imitation learning problem as a supervised learning problem. The process involves dividing the expert's demonstrations into state-action pairs, considering these pairs as independent and identically distributed (IID) examples. Subsequently, supervised learning algorithms are applied to learn the mapping between observed states and the corresponding expert actions.

One of the advantages of behavioral cloning is that it does not require active querying of the expert during the training phase. Instead, it relies solely on the available expert demonstrations. By treating the problem as a supervised learning task, the agent can learn directly from the labeled state-action pairs without the need for further interaction with the expert. This approach simplifies the learning process and enables the agent to acquire the desired behavior by mimicking the expert's actions. However, it is important to note that behavioral cloning inherits the challenges of supervised learning, such as the need for high-quality and diverse expert demonstrations to ensure generalization and robustness in real-world scenarios.

$$\begin{aligned} &argmin_{\theta}\ E_{s\sim P(s|\pi_{\theta})}[\mathcal{L}(\pi_{\theta}^{*}(s), \pi_{0}(s)] \\ &= \sum_{i=1}^{N} \mathcal{L}(a_{i}^{*}, \pi_{\theta}(s_{i}^{*}) \end{aligned}$$

Another significant drawback of behavioral cloning is its limited ability to reason beyond the immediate next step. In Markov Decision Processes (MDPs), the outcome of an action in a particular state depends on the subsequent states and actions, breaking the Independent and Identically Distributed (IID) assumption. As a result, errors made by the agent can accumulate and propagate throughout the decision-making process. Even a small mistake early on can lead the agent into states that the expert has never encountered, and consequently, the agent may lack the necessary training to handle such situations.

This lack of exposure to unvisited states can have severe consequences, potentially resulting in catastrophic failures. Since the agent has not been trained on these unvisited states, its behavior in such scenarios becomes undefined. The agent may exhibit unpredictable or unsafe behaviors, which can jeopardize the performance and safety of the autonomous system.

## 3.3 NEAT: NEURAL ATTENTION FIELDS FOR END-TO-END AUTONOMOUS DRIVING

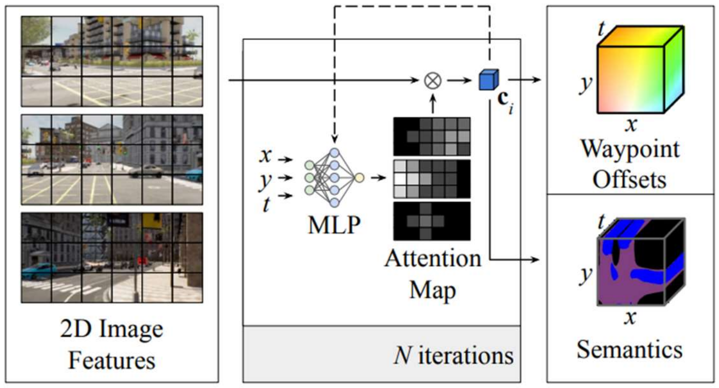

*Figure 3.10 Neural Attention Fields Overview*

NEAT, a paper published in 2021 at the ICCV conference by A. Geiger et al., is a vision-based End-to-End autonomous driving architecture that uses imitation learning. It can drive through urban environments by taking in multi-camera inputs through monocular cameras and by predicting the trajectory. NEAT jointly learns to predict the trajectory in the form of waypoint offsets and the Bird's Eye View semantics. NEAT differs from other autonomous driving research by directly learning in the BEV space. It can be interpreted as a continuous function that maps locations in BEV scene coordinates to waypoints and semantics by selectively attending to the perspective camera views. The paper does this mapping by building their novel Attention Field using so-called *Spatiotemporal BEV queries*. NEAT does behavioral cloning by *conditional imitation learning* to learn the mapping function.

Vega Autonomous Vehicle division followed this paper to deploy it in a prototype car and to improve building upon it. Therefore, a proper understanding of the NEAT architecture and its background was required before touching upon implementations and deployment.

### 3.3.1 NEAT Architecture

The NEAT (Neural Encoder-Decoder for Autonomous Driving) architecture is an innovative End-to-End learning (however the overall autonomous driving system follows the Direct Perception paradigm) approach that combines encoder and decoder components to enable effective imitation learning for autonomous driving tasks. Figure 3.11 illustrates the overall structure of the NEAT architecture.

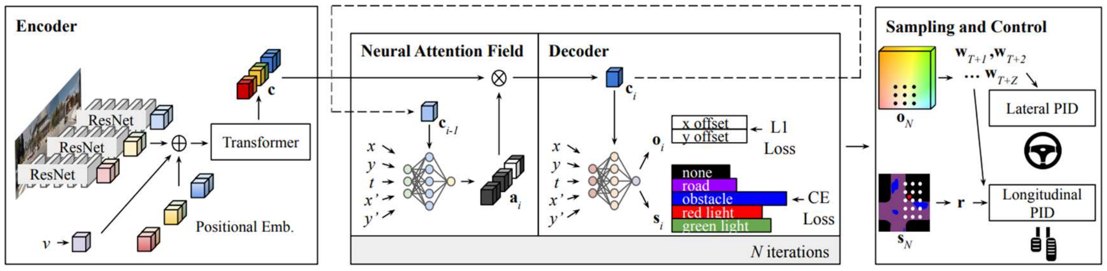

*Figure 3.11 NEAT Architecture*

At the core of NEAT is the encoder, which processes input data to extract relevant features. In this case, the encoder takes in three monocular camera images that collectively provide a 180° view of the scene ahead of the vehicle. Each image is independently passed through a ResNet feature extractor, which captures important visual representations from the images. The ResNet outputs a grid of 8x8 features per image, with each feature representing a "patch feature" corresponding to a specific region of the image. This approach draws inspiration from the Vision Transformer (ViT) model.

To enhance the image embeddings with spatial information, the patch features are combined with positional embeddings using addition. Additionally, a projection of the ego-vehicle's current velocity is incorporated into the image embeddings through another additional operation. These enriched embeddings are then fed into a Transformer, a powerful self-attention mechanism, to generate a comprehensive "global context" latent vector that captures the essential information from the input images.

On the other hand, the decoder component of NEAT is responsible for predicting the trajectory of the vehicle as the primary task. It achieves this by generating "waypoint offsets" that represent the relative positions of waypoints with respect to the vehicle's current position. Additionally, the decoder tackles an auxiliary task by predicting the BEV semantics, which provide a top-down view of the scene from above.

To jointly learn these two tasks in the decoder, the NEAT architecture incorporates a technique called "*spatiotemporal query*". This mechanism allows the model to effectively utilize both spatial and temporal information to make accurate predictions. By leveraging the spatiotemporal query, NEAT exploits the correlations between different time steps and their corresponding spatial representations to enhance the overall performance and robustness of the system.

### 3.3.2 Spatiotemporal Query

A crucial aspect of the NEAT architecture is the incorporation of spatiotemporal queries, which enable the model to effectively utilize both spatial and temporal information for accurate predictions. During training, the model is exposed to a sequence of BEV frames, including the current frame and several future frames within a predefined prediction horizon. Each BEV frame is associated with a specific timestamp and contains spatial coordinates (x, y) representing the real-world BEV scene.

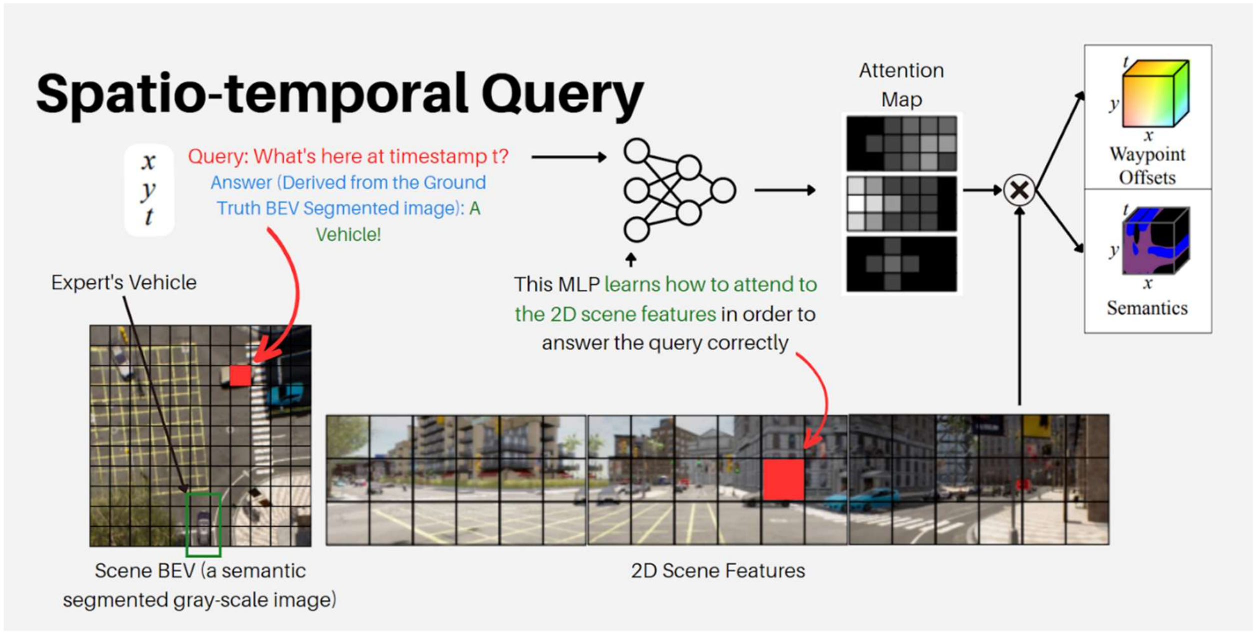
Spatio-temporal Query
$x$
$y$
$t$
Query: What's here at timestamp t?
Answer (Derived from the Ground Truth BEV Segmented image): A Vehicle!
Expert's Vehicle
This MLP learns how to attend to the 2D scene features in order to answer the query correctly
Attention Map
Waypoint Offsets
Semantics
Scene BEV (a semantic segmented gray-scale image)
2D Scene Features

*Figure 3.12 Spatiotemporal Query Mechanism*

To facilitate the spatiotemporal queries, the future BEV frames are transformed into the vehicle's local BEV frame, aligning them with the current frame. From this mixture of BEV frames at different timestamps, a set of spatiotemporal query locations is sampled uniformly at random. For each semantic class, 64 points are sampled, resulting in a total of 320 points across the five timestamps. These query points essentially ask, "*What is present at the spatial BEV location (x, y) at timestamp t*?", as depicted in Figure 3.12.

These spatiotemporal queries are then passed through a Multi-Layer Perceptron (MLP) module, which learns to map the BEV locations to the 2D scene features observed by the camera. The output of the MLP represents spatiotemporal features and is treated as attention weights for the Encoder's scene context output, denoted as $c$. The attention map, weighted with the Encoder's context output, is utilized for the actual predictions.

Another MLP within the Decoder is responsible for performing predictions based on the spatiotemporal queries. This MLP takes the same queries as input and produces two sets of predictions: waypoint offset predictions and BEV semantics predictions. The waypoint offset predictions represent the relative positions of waypoints with respect to the vehicle's current position, while the BEV semantics predictions provide probability distributions over the five semantic classes for the queried points.

Since the Encoder's context output has high dimensionality, the Decoder performs computations iteratively. The Decoder iteratively computes lower-dimensional context features, denoted as $c_i$ by applying attention computed using the spatiotemporal queries. These filtered context features serve as the basis for the predictions made by the Decoder.

The Decoder predicts the outputs for multiple iterations, and these predictions are averaged to obtain the final output predictions. The waypoint offset predictions have a size of (2, 320), where 320 represents the number of queries made, and the BEV occupancy predictions have a size of (5, 320), reflecting the probability distributions of the queried points being assigned to each of the five semantic classes.

By leveraging the spatiotemporal queries and the iterative decoding process, the NEAT architecture effectively captures the temporal dynamics and spatial relationships necessary for accurate trajectory predictions and BEV semantics estimation in autonomous driving scenarios.

## 3.4 EXPERIENCE GAINED

In my engagement with the Vega Autonomous Vehicle division, significant experience was gained in various aspects of autonomous driving, particularly through the utilization of the CARLA simulator and the exploration of the NEAT architecture. The CARLA simulator, an open-source platform built on Unreal Engine 4, provided a powerful tool for comprehensive autonomous driving simulation.

Delving into the realm of autonomous driving, the study of deep learning approaches provided a comprehensive understanding of the dominating paradigms in the field. The exploration of modular pipelines, end-to-end learning, and direct perception shed light on their strengths and limitations in building the black-box mapping function within the autonomous driving stack. Furthermore, the discussion of imitation learning and its variants, such as behavioral cloning, direct policy learning, and inverse reinforcement learning, highlighted the importance of learning from expert demonstrations to overcome challenges associated with sparse rewards and defining reward functions. I was assigned to write bash scripts for automating the data collection process using CARLA for training NEAT, and the source codes that I wrote can be found in Annexes I and II.

In particular, the NEAT architecture, an end-to-end learning encoder-decoder model, was examined. The Encoder component processed monocular camera images and utilized a ResNet feature extractor to obtain image embeddings. These embeddings, combined with positional information and the vehicle's velocity, were then passed through a Transformer module to derive a global context latent vector. On the other hand, the Decoder component predicted trajectory waypoints and BEV semantics as auxiliary tasks. Notably, the introduction of spatiotemporal queries facilitated the incorporation of spatial and temporal information in the prediction process, enabling the model to effectively learn and generate accurate outputs.

Overall, the experience gained in the Vega Autonomous Vehicle division provided a comprehensive understanding of the CARLA simulator, deep learning approaches, and the NEAT architecture. This exposure not only enhanced technical skills in areas autonomous driving simulation but also fostered a sense of responsibility, professionalism, and adaptability in the rapidly evolving field of autonomous vehicles. The knowledge and insights gained through this experience serve as a strong foundation for further exploration and advancements in autonomous driving technologies.

# Chapter 4

# DEVELOPMENT OF THE AUTONOMOUS VEHICLE

## 4.1 INTRODUCTION

During the early stages of my internship at Vega, the Vega Autonomous Vehicle division was at the initial phase of development. Immersed in the world of autonomous driving, the team delved into the research literature, with a particular focus on the NEAT paper, and began testing NEAT within the CARLA simulator. Vega Innovations possessed the powerful Nvidia DRIVE PX2 supercomputer, purpose-built for autonomous driving applications. With newfound knowledge about deep learning approaches in autonomous driving, I was assigned the task of deploying NEAT on the Drive PX2. Despite months of dedicated effort, it became evident that achieving this objective with the Drive PX2 was not feasible, leading to this realization through rigorous testing and analysis. Additionally, plans to construct a prototype small car with autonomous capabilities experienced delays, prompting the Autonomous Driving division to pivot and pursue the implementation of NEAT on an RC car as a proof of concept. The subsequent sections will delve into the detailed work carried out in pursuit of these tasks, offering insights and explanations of the progress made within each area.

## 4.2 NVIDIA DRIVE PX2

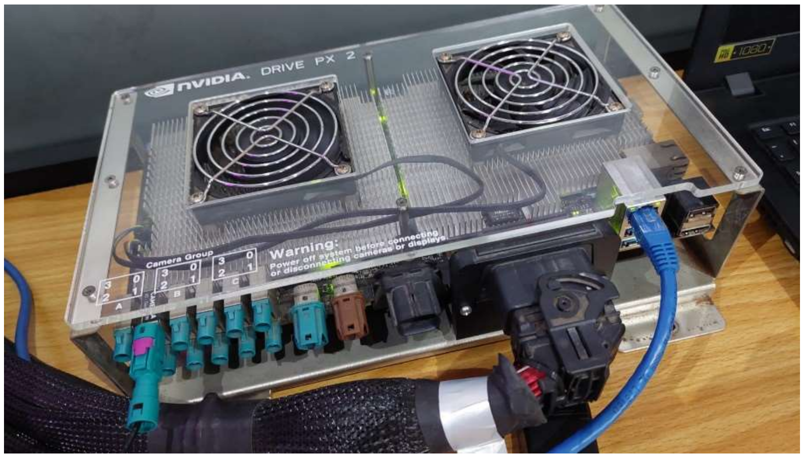
NVIDIA. DRIVE PX 2
Camera Group
Warning:
Power off system before connecting or disconnecting cameras or displays

*Figure 4.1 Nvidia DRIVE PX2 Development Kit*

The Nvidia DRIVE PX2 is a cutting-edge computer platform utilized for the accelerated development of automated and autonomous vehicles. This platform is equipped with powerful hardware components, including one or two Tegra X2 System-on-Chips (SoCs). Each Tegra X2 SoC incorporates two Denver cores, four ARM A57 cores, and a Pascal generation GPU. There are two real-world board configurations available: AutoCruise, featuring one Tegra X2 SoC and one Pascal GPU, and AutoChauffeur, equipped with two Tegra X2 SoCs and two Pascal GPUs. Vega Innovations is in possession of the AutoChauffeur configuration, as depicted in Figure 4.1.

The Nvidia DRIVE PX2 platform is specifically designed to facilitate the deployment of deep neural networks (DNN) and handle sensor data fusion and processing tasks. It boasts two integrated and two discrete GPUs based on the Pascal architecture, resulting in a total computational performance of 8 TFLOPS or 24 DL TOPS (deep learning tera-operations per second). The platform's two Tegra X2 SoCs, featuring ARM v8 architecture CPUs, provide a combined total of 12 cores, with each SoC housing four ARM-A57 and two Denver cores. The system is equipped with 8 GB LPDDR4 128-bit memory and an integrated GPU based on the Pascal architecture.

In addition to its robust computing capabilities, the Nvidia DRIVE PX2 platform offers a comprehensive range of connectivity options. It provides UART, CAN, LIN, FlexRay, USB 3.0 and 2.0, 1 and 10 Gbit Ethernet, HDMI, and 12 GSML camera ports. These features cater to the diverse requirements of autonomous vehicle development, ensuring seamless integration and compatibility with various peripherals and sensors. The Nvidia DRIVE PX2 platform stands as a versatile and high-performance solution for enabling advanced deep learning algorithms and sensor data processing in the pursuit of autonomous driving.

### 4.2.1 Nvidia DRIVE OS

Nvidia DRIVE OS is the reference operating system and associated software stack designed specifically for developing autonomous vehicle applications on DRIVE hardware. Nvidia DRIVE OS delivers,

- A safe & secure execution environment for safety-critical applications
- Services:
    - Secure boot
    - Security services
    - Firewall
    - Over-the-air (OTA) updates

This foundational software stack for autonomous vehicles consists of an embedded real-time operating system (RTOS), Nvidia Hypervisor, Nvidia CUDA libraries, Nvidia TensorRT and other components, as shown in Figure 4.2, optimized to provide direct access to DRIVE hardware acceleration engines.

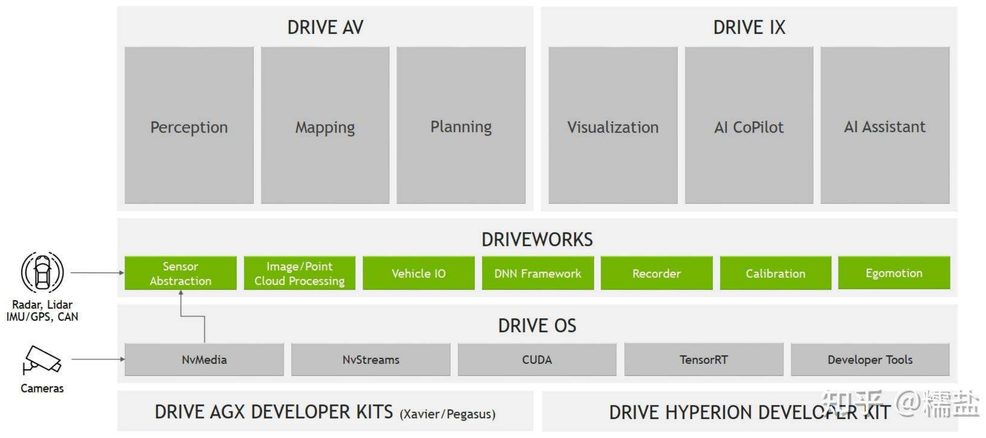

*Figure 4.2 Nvidia DRIVE Software Stack*

### 4.2.2 Nvidia GIE

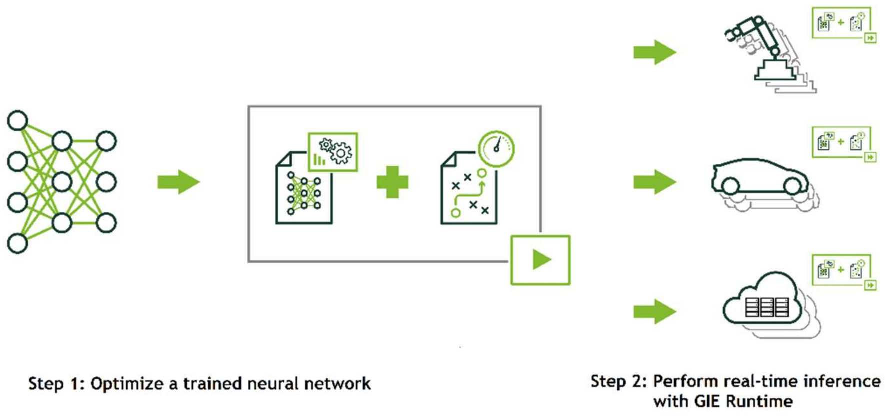
Step 1: Optimize a trained neural network
Step 2: Perform real-time inference with GIE Runtime

*Figure 4.3 The GPU Inference Engine Workflow*

The Nvidia GPU Inference Engine (GIE) is a powerful system designed to enable efficient and high-performance inference for deep learning applications. Inference engines play a crucial role in extracting new insights, rules, and relationships from knowledge bases or deep learning models. By leveraging the facts and rules contained within these knowledge bases, an inference engine can make informed decisions.

Previously known as Nvidia TensorRT, GIE represents the earlier version of this inference engine. GIE, or TensorRT, offers exceptional inference throughput and efficiency for a variety of common deep learning tasks, including image classification, segmentation, and object detection. Its primary objective is to optimize trained neural networks for runtime performance, providing GPU-accelerated inference capabilities for web, mobile, embedded, and automotive applications.

GIE is specifically engineered to maximize the utilization of GPU resources and ensure efficient execution of inference tasks. By leveraging the parallel processing capabilities of GPUs, it delivers accelerated performance and enables real-time inference for a wide range of deep learning applications. GIE empowers developers to optimize their models and deploy them in production environments, where inference efficiency and throughput are critical factors. With GIE, developers can harness the power of Nvidia GPUs to achieve high-performance and efficient inference for various domains, driving advancements in web, mobile, embedded, and automotive applications.

The use of the NVIDIA GPU Inference Engine (GIE) involves two distinct phases: the build phase and the deployment phase, as depicted in Figure 2. During the build phase, GIE undertakes crucial optimizations on the network configuration to enhance its performance. It generates an optimized plan that specifies the computations required to perform the forward pass through the deep neural network. This plan is transformed into an optimized object code that can be serialized and stored in memory or on disk, ready for deployment.

In the deployment phase, GIE is utilized within a long-running service or a user application. This phase involves accepting batches of input data, executing the optimized plan on the input data to perform inference, and returning batches of output data, such as classification results or object detection outcomes. Notably, with GIE, there is no need to install and run a deep learning framework on the deployment hardware, simplifying the deployment process.

While the details of batching and the pipeline of the inference service are beyond the scope of this discussion, it is important to highlight that GIE streamlines the process of using deep learning models for inference. It provides developers with the means to optimize their network configurations, generate efficient execution plans, and seamlessly deploy these plans for real-time inference tasks. By leveraging GIE, developers can focus on leveraging the power of the inference engine without the need for deep learning framework installations on the deployment hardware.

### 4.2.3 TensorRT (TRT)

Nvidia TensorRT is a powerful software development kit (SDK) designed to optimize trained deep learning models, facilitating high-performance inference. With TensorRT, developers can significantly enhance the efficiency and speed of deep learning models during inference. The SDK comprises two key components: a deep learning inference optimizer and a runtime for execution.

Once deep learning models are trained using a framework of choice, TRT steps in to optimize them further. By leveraging various techniques such as layer fusion, precision calibration, and kernel auto-tuning, TRT streamlines the computational graph and reduces unnecessary operations, resulting in highly optimized models. This optimization process enables superior throughput and lower latency during inference. The TRT workflow is shown in Figure 4.4.

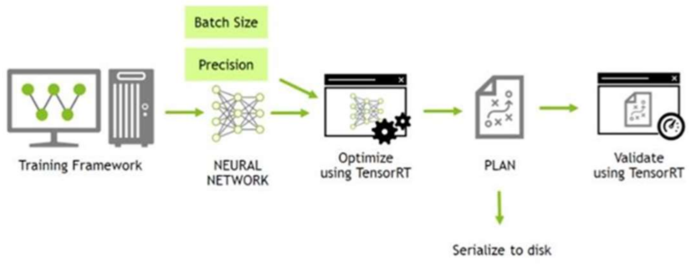

*Figure 4.4 TensorRT Workflow*

TRT takes over the optimization process by analyzing the already trained network (this would imply the model definition along with the learned parameters) and applying various optimization techniques. It also takes other hyperparameters such as inference batch size and precision. TRT does optimization and builds an execution plan. This execution plan can be directly used for inference purposes or serialized and stored on disk for future use. During inference, there is no requirement to have deep learning frameworks like PyTorch or TensorFlow installed or running. TRT provides a standalone runtime environment, leveraging the optimized execution plan to efficiently process input data and generate inference results.

### 4.2.4 ONNX (Open Neural Network Exchange Format)

ONNX is an open and interoperable format designed to facilitate the exchange of deep learning models between different frameworks. It provides a standardized way to represent and transfer trained models across various platforms and tools. With ONNX, users can seamlessly move their models between different deep learning frameworks, enabling greater flexibility and collaboration. ONNX stores data in a format called *protocol buffers*. The version of the ONNX file format is specified in the form of an “opset”.

The main advantage of ONNX is its ability to bridge the gap between different frameworks, allowing models to be trained in one framework and deployed in another. This interoperability simplifies the model development and deployment process, as it eliminates the need for reimplementation or conversion of models between frameworks.

ONNX defines a common intermediate representation for deep learning models, which captures the structure and parameters of the model in a framework-agnostic manner.

To use ONNX, one can export a trained model from a supported framework, such as PyTorch, TensorFlow, or Caffe, into the ONNX format. This exported model can then be imported into another framework that supports ONNX, enabling seamless integration and inference across different environments. Additionally, ONNX supports the execution of models on various hardware platforms, including CPUs, GPUs, and specialized accelerators.

### 4.2.5 UFF (Universal Framework Format)

UFF (Universal Framework Format) is a file format developed by NVIDIA that provides a standardized representation for deep learning models to be used with the TRT inference engine. UFF is designed to optimize the deployment of trained models on NVIDIA GPUs, enabling high-performance inference. The main purpose of UFF is to bridge the gap between different deep learning frameworks and TRT, allowing models trained in popular frameworks such as TensorFlow, Caffe, and TRT to be easily converted and optimized for efficient execution using TRT.

To convert a model to UFF format, one typically starts with a trained model in a framework-specific format, such as a TensorFlow SavedModel or a Caffe model definition. The UFF converter tool provided by NVIDIA can then be used to convert the model into the UFF format, which represents the model's architecture, weights, and operations in a framework-agnostic manner.

Once the model is converted to UFF, it can be further optimized by TRT to take advantage of the GPU's computational capabilities. TRT performs various optimizations such as layer fusion, precision calibration, and kernel auto-tuning to maximize the inference performance of the model on NVIDIA GPUs.

### 4.2.6 Attempt to deploy NEAT into Nvidia DRIVE PX2

The deployment of NEAT into Nvidia DRIVE PX2 involved exploring two paths: ONNX and UFF. Typically, PyTorch models can be exported to ONNX, while TensorFlow models can be exported to UFF. However, since the NEAT model was defined in PyTorch, the initial approach was to export NEAT to ONNX and deploy it on the PX2 platform. To test the UFF models, a simpler autonomous driving model called PilotNet by Nvidia was implemented in TensorFlow (see Annex III). Additionally, a CIFAR10 classification network was implemented (see Annex IV) to validate the conversion process from PyTorch to ONNX. General Python scripts were developed to convert PyTorch models to ONNX and TensorFlow models to

UFF (see Annex V and VI). After training and exporting these models to ONNX and UFF, DriveWorks scripts in C++ were written on the Drive PX2 to load these saved models and generate TensorRT (TRT) plan files, enabling inference using GMSL camera feeds. These initial tests were successful, and the plan was to proceed with exporting NEAT to ONNX. However, issues arose during the ONNX export due to limited support for ONNX opset versions in the DriveWorks version of the DRIVE PX2. The minimum supported ONNX opset version in DriveWorks SDK v4 was 7, which was insufficient for exporting the NEAT model. Consequently, the alternative approach was to export NEAT into UFF, requiring the implementation of NEAT in TensorFlow from scratch. Although the UFF models were successfully exported, the UFF parser in DriveWorks SDK v4 encountered difficulties parsing the computational graph, similar to the previous issue with ONNX. As a result, it was concluded that deploying NEAT on a DRIVE PX2, which is a 7-year-old computer, was not feasible.

## 4.3 AUTONOMOUS RC CAR

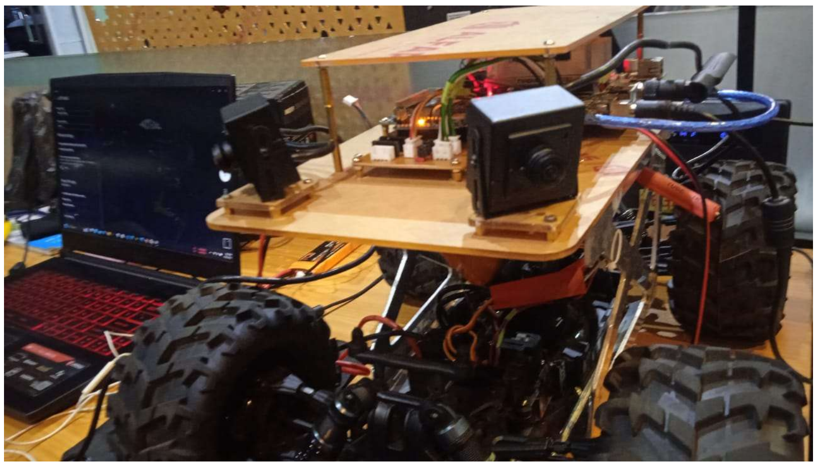

*Figure 4.5 The RC Autonomous Vehicle Prototype of Vega Innovations*

Due to the limitations of deploying NEAT on the existing DRIVE PX2 computer, Vega Innovations' autonomous driving division made the decision to purchase an RC car (see Figure 4.5) and implement autonomous driving using NEAT as a proof of concept. However, NEAT requires GPS navigational commands to compute waypoint offsets. Since the RC car is a scaled version of a real car, it cannot be driven on actual roads. As a solution, a scaled arena was designed for the car to drive in (see Figure 4.6). This presented the challenge of obtaining GPS signals within the scaled arena. Since the arena is small and local, using an IMU to gather GPS positional data would be ineffective. To address this issue, a vision-based localization system was developed. The brain of the RC car was powered by a Nvidia Jetson TX2 development kit. An overhead camera was strategically placed to capture the map arena and localize both the map and the car. To facilitate localization, ArUco markers, a type of fiducial markers developed by

Augmented Reality University of Cordoba, were placed on the vehicle as well as at the four corners of the map arena.

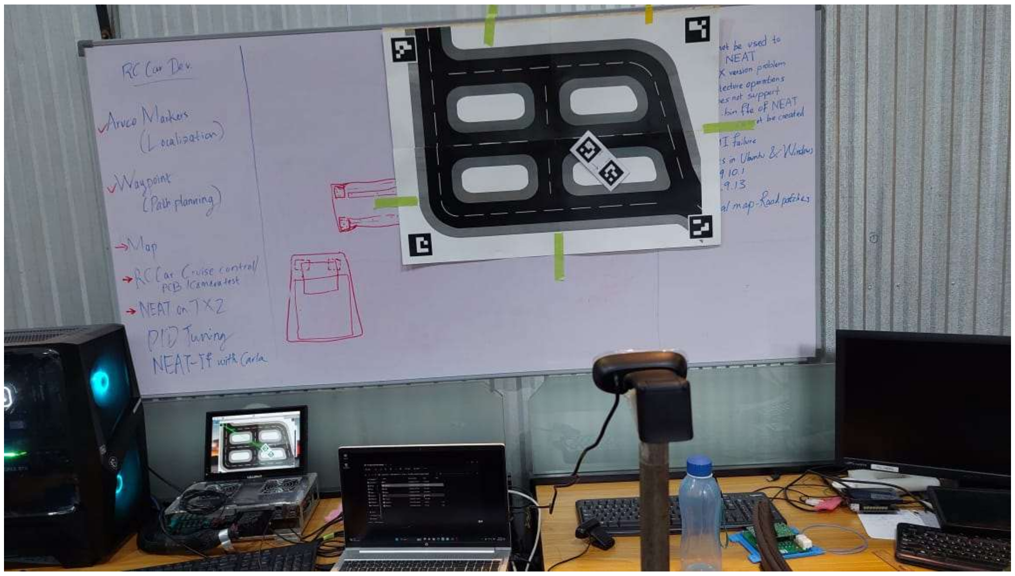

*Figure 4.6 The RC Car Map Arena Testing Setup*

To overcome the limitations of the Nvidia DriveWorks SDK in deploying NEAT on the Nvidia Jetson TX2, the jetson-inference library was leveraged. This library is designed for various Nvidia embedded systems, including the Jetson TX2, and provides ONNX and UFF parsers that are compatible with the latest operator sets. With this in mind, C++ scripts were developed using the jetson-inference library to import the NEAT ONNX model and perform inference on the Jetson TX2. However, unlike Nvidia DriveWorks, the jetson-inference library does not offer a built-in capability to read multiple camera feeds. To address this, Nvidia GStreamer was utilized to handle the task of acquiring and processing multiple camera feeds, enabling a comprehensive input for the NEAT-based autonomous driving system.

In accordance with company rules and guidelines, specific references to the codes and scripts developed during this project cannot be provided in this documentation.

## 4.4 QUALITY ASSURANCE & ETHICAL PRACTICES

In my role within the development of the autonomous driving vehicle's software stack, I played a proactive part in enforcing robust quality assurance practices. These encompassed comprehensive testing procedures, meticulous code reviews, and thorough documentation, all aimed at guaranteeing the software's reliability and adherence to safety standards. By actively participating in these processes, I contributed to the creation of a software ecosystem that was not only cutting-edge but also maintained the highest levels of quality and dependability.

Furthermore, I was unwavering in my commitment to upholding ethical practices throughout the software's development lifecycle. This involved safeguarding data privacy, ensuring secure handling of sensitive information, and maintaining transparency in our algorithms and processes. By conscientiously following these ethical guidelines, I helped foster a culture of responsibility and integrity within the team. This commitment to ethical practices was pivotal in building trust and confidence in our autonomous driving technology, both internally within the organization and among our end-users.

## 4.5 EXPERIENCE GAINED

Driven by the desire to implement NEAT on the Nvidia DRIVE PX2 computer platform, significant efforts were devoted to integrating NEAT with the platform's capabilities. However, after months of diligent work, it became apparent that deploying NEAT on the DRIVE PX2 posed insurmountable challenges due to hardware constraints. Undeterred, the autonomous driving division pivoted its approach, opting to implement NEAT on an RC car as a proof of concept. This shift necessitated the creation of a scaled arena, designed specifically for the RC car's navigation and testing. To ensure accurate localization, a vision-based system leveraging fiducial markers, known as ArUco markers, was developed. The Nvidia Jetson TX2 development kit served as the brain of the RC car, providing the necessary computing power for real-time decision-making.

While progress was being made with the NEAT implementation on the RC car, it was essential to find a suitable framework for model deployment. The team discovered the jetson-inference library, tailored for Nvidia embedded systems such as the Jetson TX2, which provided compatibility with the latest operator sets in ONNX and UFF formats. By leveraging this library, the NEAT ONNX model was successfully imported, enabling efficient inference on the RC car. To handle the multiple camera feeds required for comprehensive perception, the team integrated Nvidia GStreamer, a flexible multimedia framework, ensuring synchronized and real-time data processing.

This transformative journey presented numerous challenges, each a valuable learning opportunity. It deepened our understanding of autonomous driving, deep learning deployment, and real-world system integration. Collaboration spurred innovative thinking and problem-solving. Our dedication and adaptability led to a wealth of expertise shaping Vega Innovations' future in autonomous vehicle tech. Adhering to company rules on code references preserved intellectual property and confidentiality.

# CONCLUSION

The training experience at Vega Innovations has been a transformative journey, encompassing SMD soldering techniques, the CARLA simulator, and the development of autonomous vehicles using DL techniques. This comprehensive program allowed me to acquire valuable skills and insights, while encountering various challenges along the way. The academic exposure and coding practices gained at the faculty played a vital role in effectively following the training curriculum, providing a solid foundation for understanding the underlying concepts and methodologies. Additionally, the hands-on experience at Vega Innovations deepened my understanding of real-world constraints and practical application development, bridging the gap between theory and implementation.

There are several suggestions to enhance the productivity and effectiveness of the training program at Vega Innovations. Firstly, fostering a culture of collaboration among participants can greatly enrich the learning experience. Encouraging participants to engage in group projects, discussions, and knowledge-sharing sessions can facilitate the exchange of ideas and perspectives. This collaborative environment can stimulate creativity, problem-solving skills, and teamwork, which are essential in the field of autonomous driving.

Moreover, incorporating regular feedback and evaluation mechanisms throughout the training program can help identify areas for improvement and ensure the program remains relevant and responsive to the evolving needs of the participants. This feedback loop can involve regular assessments, surveys, and discussions to gather insights on the effectiveness of the curriculum, training methodologies, and overall program structure. By actively seeking input from participants, the training program can be continuously refined and tailored to address specific learning needs and expectations.

In conclusion, the training program at Vega Innovations provided a valuable learning experience, from mastering SMD soldering to exploring the intricacies of autonomous driving with DL. The fusion of academic knowledge and practical training created a synergistic environment, enabling me to apply my skills and knowledge effectively. This training experience has not only enriched my understanding of the field but also equipped me with the necessary tools and expertise to contribute to the advancement of autonomous driving technology. I am grateful for the support and guidance received during this journey, and I am confident that the valuable lessons learned will serve as a solid foundation for my future endeavors in the dynamic and ever-evolving field of autonomous driving.

## REFERENCES

1. “NEAT: Neural Attention Fields for End-to-End Autonomous Driving,” GitHub, Aug. 29, 2023. https://github.com/autonomousvision/neat.

2. “CARLA Simulator,” carla.readthedocs.io. https://carla.readthedocs.io/en/latest/.

3. V. Ungrapalli, “Low Precision Inference with TensorRT,” Medium, Nov. 22, 2018. https://towardsdatascience.com/low-precision-inference-with-tensorrt-6eb3cda0730b#:~:text=TensorRT%20is%20Nvidia%20software%20solution.

4. Deci, “Understanding NVIDIA TensorRT for Deep Learning Inference Optimization,” Deci, Apr. 24, 2023. https://deci.ai/blog/tensorrt-framework-overview/.

5. “What every ML/AI developer should know about ONNX,” Paperspace Blog, Mar. 23, 2018. https://blog.paperspace.com/what-every-ml-ai-developer-should-know-about-onnx/.

6. “Self-Driving Cars — Andreas Geiger - YouTube,” www.youtube.com. https://youtube.com/playlist?list=PL05umP7R6ij321zzKXK6XCQXAaaYjQbzr&si=VtcZmMTTpDQlRFfK.

7. “Datasets & DataLoaders — PyTorch Tutorials 1.11.0+cu102 documentation,” pytorch.org. https://pytorch.org/tutorials/beginner/basics/data_tutorial.html.

8. lhzlhz, “PilotNet: End to End Learning for Self-Driving Cars,” GitHub, Aug. 15, 2023. https://github.com/lhzlhz/PilotNet.

9. “What are the 6 levels of autonomous vehicles?,” Faistgroup.com, Jan. 11, 2023. https://www.faistgroup.com/news/autonomous-vehicles-levels/.

10. “Introduction to Tensorflow (Graph, Session, Nodes and variable scope),” cs230.stanford.edu. https://cs230.stanford.edu/blog/moretensorflow/.

11. P. Janetzky, “A practical guide to TFRecords,” Medium, Jan. 13, 2022. https://towardsdatascience.com/a-practical-guide-to-tfrecords-584536bc786c.

# ANNEXES

**ANNEX I**

```
#!/bin/bash
# Copyright (C) 2023, Vega Innovations, Sri Lanka.
# Author: Sathira Silva
#
# This is an automated CARLA data collection script for a single routes file, used for
the NEAT algorithm.
# Usage: bash data_collection_single.sh <CARLA world port> <CARLA traffic manager port>
<path to routes .json file> <save path root> <route repetition count>
# Example: bash data_collection_single.sh 2000 8000 /home/vegaav/Desktop/autonomous-
vehicle/neat/leaderboard/data/training_routes/routes_town01_short.xml
/Disk/NeatData/TrainingData 1

# ANSI colour codes that're used in the script.
BLUE='\033[0;34m'
GREEN='\033[0;32m'
YELLOW='\033[0;33m'
RED='\033[0;31m'
NC='\033[0m'

# Store the command line arguments (usage is defined in the upper section).
arg1=$1
arg2=$2
arg3=$3
arg4=$4
arg5=$5

# Exit flag is used to indicate the script to terminate or not (used inside the main
function).
exit=false
# Initialize the eval_pid, which is the pid (process id) of the leaderboard_evaluator.py
process.
eval_pid=
# Initialize the cid, which is the container id of the docker container which runs the
CARLA server.
cid=

# Cleanup function is triggered on INT, TERM, and EXIT signals which terminates
leaderboard_evaluator.py and CARLA server
# that are spawned from the shell script if they exist when the particular signal
arrives.
cleanup() {
    printf "\nExiting...\n"
    exit=true
    sleep 2
    # Check if the leaderboard_evaluator.py process exists at the moment.
    if ps --pid $eval_pid &> /dev/null
    then
        # Kill the process if it does exist.
        kill -9 $eval_pid >/dev/null 2>&1 &
        echo "leaderboard_evaluator [PID=${eval_pid}] Terminated Successfully."
    fi
    # Check if the docker container exists at the moment.
    if docker ps -q --filter "id=$cid" &> /dev/null
    then
        # Kill the container if it does exist.
        docker kill $cid >/dev/null 2>&1 &
        echo "CARLA Server [Container ID=${cid}] Terminated Successfully."
    fi
}

# Similar to the cleanup function but calls the main function to restart the script on a
segfault signal.
restart() {
    printf "\nA Segfault detected. Restarting the Script...\n"
    sleep 2
    if ps --pid $eval_pid &> /dev/null
    then
        kill -9 $eval_pid >/dev/null 2>&1 &
        echo "leaderboard_evaluator [PID=${eval_pid}] Terminated Successfully."
    fi
    if docker ps -q --filter "id=$cid" &> /dev/null
    then
        docker kill $cid >/dev/null 2>&1 &
        echo "CARLA Server [Container ID=${cid}] Terminated Successfully."
    fi

    main
}

# Register the trap handlers.
trap cleanup INT TERM EXIT
trap restart SIGSEGV

main() {
    # Get the first argument (path to the routes file) and set it as an environment
variable.
    export ROUTES=$arg3 # routes file
    # Get the basename of the routes file.
    xbase=${ROUTES##*/}
    # Get the prefix name (basename without the extension).
    xpref=${xbase%.*}
    # Split the prefix name by underscore and get an array.
    arr=(${xpref//_/ })

     ...
    export CHECKPOINT_ENDPOINT="${NEAT_ROOT}/carla_results/ckpt_${xpref}.json" # output
results file
    export DATA_COLLECTION_LOG="${NEAT_ROOT}/data_collection_logs/log_${xpref}"
    export SAVE_PATH=$arg4/${arr[1]^} # path for saving episodes.

    # Initialize the route completion stats.
    # These will be updated from the json file that gets created by the
leaderboard_evaluator.py which contains route scenario checkpoint information.
    curr=0
    total=-1

    # If the SAVE_PATH directory doesn't exist, make one.
    if [ ! -d ${SAVE_PATH} ]
    then
        mkdir ${SAVE_PATH}
    # Otherwise, check if the route checkpoint file exists and it is not empty.
    elif [ -f $CHECKPOINT_ENDPOINT ] && [ -s $CHECKPOINT_ENDPOINT ]
    then
        # Read the length of the 'progress' field from the json file.
        # 'progress' field is a list whose first element is the current number of
completed routes and the second element
        # is the number of total routes.
        # Reading a json file can be easily done via a third-party command-line utility
called jq.
        len=$(cat ${CHECKPOINT_ENDPOINT} | jq '._checkpoint.progress | length' 2>
/dev/null)
        # Check if the 'progress' field has a non empty list.
        if (( len > 0 ))
        then
            # Read the currently completed and the total number of routes.
            curr=$(cat $CHECKPOINT_ENDPOINT | jq '._checkpoint.progress[0]' 2>
/dev/null)
            total=$(cat $CHECKPOINT_ENDPOINT | jq '._checkpoint.progress[1]' 2>
/dev/null)

            # Check if all the routes in the current map are completed.
            if (( curr == total ))
            then
                echo -e "✅ ${GREEN}Skipping${NC} routes ${GREEN}'${xbase}'${NC} since
it's already completed."
                exit 0
            else
                echo -e "🔄 ${GREEN}Continuing${NC} routes ${GREEN}'${xbase}'${NC} from
[${curr} / ${total}]."
```

```
                echo   "================================================================"
                touch ${DATA_COLLECTION_LOG}
                echo -e "Monitor the log file ${GREEN}${DATA_COLLECTION_LOG}${NC} to see
more details during data collection."
            fi
        fi
    else
        # Create the route checkpoint json file if it doesn't exist.
        touch ${CHECKPOINT_ENDPOINT}
    fi

    echo "Setting environment variables..."

    # CARLA path
    export CARLA_SERVER=${CARLA_ROOT}/CarlaUE4.sh
    export LEADERBOARD_ROOT=${NEAT_ROOT}/leaderboard
    export PYTHONPATH=$PYTHONPATH:${CARLA_ROOT}/PythonAPI
    export PYTHONPATH=$PYTHONPATH:${CARLA_ROOT}/PythonAPI/carla
    export PYTHONPATH=$PYTHONPATH:$CARLA_ROOT/PythonAPI/carla/dist/carla-0.9.10-py3.7-
linux-x86_64.egg
    export PYTHONPATH=$PYTHONPATH:${LEADERBOARD_ROOT}
    export PYTHONPATH=$PYTHONPATH:${LEADERBOARD_ROOT}/team_code
    export PYTHONPATH=$PYTHONPATH:${NEAT_ROOT}/scenario_runner
    export CHALLENGE_TRACK_CODENAME=SENSORS
    export DEBUG_CHALLENGE=0

    # Server Ports
    export PORT=$arg1 # same as the carla server port
    export TM_PORT=$arg2 # port for traffic manager, required when spawning multiple
servers/clients
    export SCENARIOS=${LEADERBOARD_ROOT}/data/scenarios/eval_scenarios.json # scenarios
file

    export REPETITIONS=1 # number of evaluation runs
    export RESUME=$arg5 # resume evaluation from partial results file

    # Agent Paths
    export TEAM_AGENT=${LEADERBOARD_ROOT}/team_code/auto_pilot.py # agent script
    export TEAM_CONFIG=${NEAT_ROOT}/model_ckpt/neat # model checkpoint (not required for
auto_pilot)

    # ${CARLA_SERVER} -RenderOffScreen -carla-server -quality-level=Epic -opengl --
world-port=${PORT} >/dev/null 2>&1 &
    cid=$(docker run -d --rm --privileged --gpus all --net=host -v /tmp/.X11-
unix:/tmp/.X11-unix:rw -e SDL_VIDEODRIVER=offscreen -e SDL_HINT_CUDA_DEVICE=0
sathiiii/carla:0.9.10.1 /bin/bash ./CarlaUE4.sh -RenderOffScreen -carla-server -
benchmark -opengl --world-port=${PORT} 2>&1)
```

```
    # Wait until all sub processes spawn.
    sleep 3

    # ppid=$!
    # carla_pid=$(ps --ppid "${ppid}" -o pid --no-headers)
    echo "Starting Carla Server in Docker [Container ID=${cid}]..."

    turn=("${BLUE}|" "${GREEN}/" "${YELLOW}-" "${RED}\\")
    j=0
    k=0

    sleep 8

    # Loop while exit flag is not set and the routes are incomplete.
    while (( ! exit || curr < total ))
    do
        if $exit; then break; fi

        # Check if the docker container has killed.
        if [ ! $exit ] && [ ! docker ps -q --filter "id=$cid" &> /dev/null ]
        then
            printf "\n${RED}Carla simulator stopped unexpectedly.${NC}\n"
            echo   "====================================="
            i=0
            # Try to run the container again until it succeeds.
            while ! $exit
            do
                if ps --pid $eval_pid &> /dev/null
                then
                    kill -9 $eval_pid
                fi
                # ${CARLA_SERVER} -RenderOffScreen -carla-server -quality-level=Epic -
opengl --world-port=${PORT} >/dev/null 2>&1 &
                cid=$(docker run -d --rm --privileged --gpus all --net=host -v
/tmp/.X11-unix:/tmp/.X11-unix:rw -e SDL_VIDEODRIVER=offscreen -e SDL_HINT_CUDA_DEVICE=0
sathiiii/carla:0.9.10.1 /bin/bash ./CarlaUE4.sh -RenderOffScreen -carla-server -
benchmark -opengl --world-port=${PORT} 2>&1)

                echo -ne "\r\033[1KRestarting Carla Server [Container ID=${cid}]...
[Attempt ${i}]"
                ((i++))

                sleep 10

                if docker ps -q --filter "id=$cid" &> /dev/null
                then
```

```
                    break
                fi

                # Give up and reboot after 10 consecutive retries!
                # if (( i == 10 ))
                # then
                #     echo "10 consecutive attempts failed. Rebooting the PC..."
                #     sleep 2
                #     echo "vega@321" | sudo -S reboot now
                # fi
            done

            if $exit; then break; fi

            # Start the leaderboard_evaluator.py process.
            CUDA_VISIBLE_DEVICES=0 python3
${LEADERBOARD_ROOT}/leaderboard/leaderboard_evaluator.py \
            --scenarios=${SCENARIOS}  \
            --routes=${ROUTES} \
            --repetitions=${REPETITIONS} \
            --track=${CHALLENGE_TRACK_CODENAME} \
            --checkpoint=${CHECKPOINT_ENDPOINT} \
            --agent=${TEAM_AGENT} \
            --agent-config=${TEAM_CONFIG} \
            --debug=${DEBUG_CHALLENGE} \
            --record=${RECORD_PATH} \
            --resume=${RESUME} \
            --port=${PORT} \
            --trafficManagerPort=${TM_PORT} > ${DATA_COLLECTION_LOG} 2>&1 &

            # Record the pid of the leaderboard_evaluator.py process.
            # Note that $! is the process ID of the last job run in the background.
            eval_pid=$!
            printf "\nRestarting data collection script [PID=${eval_pid}]...\n"
        # Check if the leaderboard_evaluator.py process has terminated.
        elif [ -z "${eval_pid}" ]
        then
            # Spawn the leaderboard_evaluator.py process again if it had terminated.
            CUDA_VISIBLE_DEVICES=0 python3
${LEADERBOARD_ROOT}/leaderboard/leaderboard_evaluator.py \
            --scenarios=${SCENARIOS}  \
            --routes=${ROUTES} \
            --repetitions=${REPETITIONS} \
            --track=${CHALLENGE_TRACK_CODENAME} \
            --checkpoint=${CHECKPOINT_ENDPOINT} \
            --agent=${TEAM_AGENT} \
            --agent-config=${TEAM_CONFIG} \
            --debug=${DEBUG_CHALLENGE} \
            --record=${RECORD_PATH} \
            --resume=${RESUME} \
            --port=${PORT} \
            --trafficManagerPort=${TM_PORT} > ${DATA_COLLECTION_LOG} 2>&1 &

            eval_pid=$!
            echo "Starting data collection script [PID=${eval_pid}]..."

            echo -e "${GREEN}***********[Data Collection In Progress]***********${NC}"
        fi

        # Read the route checkpoint file to update the curr and total variables.
        # Keep the following commented lines commented if you run the script in
background and redirect the stdout to a file.
        # Otherwise, the log output file will be too lengthy with progress texts.
        # They're added only to make the UI informative and not boring when the script
is running in foreground.
        if [ -f ${CHECKPOINT_ENDPOINT} ] && [ -s ${CHECKPOINT_ENDPOINT} ]
        then
            len=$(cat ${CHECKPOINT_ENDPOINT} | jq '._checkpoint.progress | length' 2>
/dev/null)
            if (( len > 0 ))
            then
                curr=$(cat ${CHECKPOINT_ENDPOINT} | jq '._checkpoint.progress[0]' 2>
/dev/null)
                total=$(cat $CHECKPOINT_ENDPOINT | jq '._checkpoint.progress[1]' 2>
/dev/null)
                # If the json is updating while reading from it, the above commands
might generate errors.
                # Therefore, wait until the json file settles down.
                while [ -z $curr ] || [ -z $total ]
                do
                    # echo -ne "⏳ Calculating data collection progress. Please Wait
${turn[j]} ${NC}"
                    # ((j++))
                    # j=$(($j%4))
                    # sleep 0.08
                    # echo -ne "\033[2K\r"
                    curr=$(cat ${CHECKPOINT_ENDPOINT} | jq '._checkpoint.progress[0]' 2>
/dev/null)
                    total=$(cat $CHECKPOINT_ENDPOINT | jq '._checkpoint.progress[1]' 2>
/dev/null)
                done
                if (( curr == total ))
                then
```

```
                    echo -ne "\033[2K\rCompleted routes ${GREEN}[${curr} /
${total}]${NC}"
                    printf "\nData collection for routes ${GREEN}${xbase}${NC}
completed.\n"
                    break
                # else
                #     echo -ne "\033[2K\rCompleted routes [${curr} / ${total}]
${turn[j]} ${NC}"
                #     ((j++))
                #     j=$(($j%4))
                #     sleep 0.08
                #     echo -ne "\033[2K\r"
                fi
            # else
            #     echo -ne "⌛ Calculating data collection progress. Please Wait
${turn[j]} ${NC}"
            #     ((j++))
            #     j=$(($j%4))
            #     sleep 0.08
            #     echo -ne "\033[2K\r"
            fi
        fi
    done
}

main
```

## ANNEX II

```
#!/bin/bash
# Copyright (C) 2023, Vega Innovations, Sri Lanka.
# Author: Sathira Silva
#
# The following shell script runs multiple data_collection_single.sh scripts in
background, each running a different route file, with different world and traffic
# manager ports.
# Change the following environment variables as your needs.

...

train_files=($TRAIN_ROUTES/*)
val_files=($VAL_ROUTES/*)

echo "Found ${#train_files[@]} training route files and ${#val_files[@]} validation
route files."

echo "============================"
echo "Generating Training Data..."
echo "============================"
i=1
# NOTE: The world_port and tm_port should be different for each of the CARLA servers
that're run simultaneously.
# If you happen to have some CARLA servers already running in background, make sure that
the ports you are using aren't used already.
world_port=2000
tm_port=8000

# Following loops through the train_files and executes a separate CARLA server and the
data collection script for each of them.
for route_file in "${train_files[@]}"
do
    xbase=${route_file##*/}
    xpref=${xbase%.*}
    echo "⌛ Starting training routes - ${route_file##*/} [${i} / ${#train_files[@]}]"
    nohup bash ./data_collection_single.sh $world_port $tm_port $route_file $TRAIN_PATH
2 &> "${DATA_COLLECTION_LOG}/nohup_${xpref}.out" &
    sleep 3
    echo "PID=${!}" >> "${DATA_COLLECTION_LOG}/nohup_${xpref}.out"
    ((i++))
    world_port=$(($world_port+2))
    tm_port=$(($tm_port+2))
done

# echo "============================"
# echo "Generating Validation Data..."
# echo "=============================="
# i=1
# world_port=2000
# tm_port=8000
# for route_file in "${val_files[@]:3:1}"
# do
#     xbase=${route_file##*/}
#     xpref=${xbase%.*}
#     echo "⌛ Starting validation routes - ${route_file##*/} [${i} / ${#val_files[@]}]"
#     nohup bash ./data_collection_single.sh $world_port $tm_port $route_file $VAL_PATH
2 &> "${DATA_COLLECTION_LOG}/nohup_${xpref}.out" &
#     sleep 3
#     echo "PID=${!}" >> "${DATA_COLLECTION_LOG}/nohup_${xpref}.out"
#     ((i++))
#     world_port=$(($world_port+2))
#     tm_port=$(($tm_port+2))
# done
```

## ANNEX III

```
def get_nvidia(input_dim, training=True):
    model = tf.keras.models.Sequential()
    model.add(tf.keras.layers.InputLayer(input_shape=list(input_dim),
name="InputLayer"))
    model.add(tf.keras.layers.Lambda(lambda x: tf.transpose(x, perm=(0, 2, 3, 1)),
trainable=training)) # Convert to NHWC again for efficiency.
    model.add(tf.keras.layers.Conv2D(filters=24, kernel_size=(5, 5), strides=(2, 2),
activation='relu', trainable=training))
    model.add(tf.keras.layers.Conv2D(filters=36, kernel_size=(5, 5), strides=(2, 2),
activation='relu', trainable=training))
    model.add(tf.keras.layers.Conv2D(filters=48, kernel_size=(5, 5), strides=(2, 2),
activation='relu', trainable=training))
    model.add(tf.keras.layers.Conv2D(filters=64, kernel_size=(3, 3), activation='relu',
trainable=training))
    model.add(tf.keras.layers.Conv2D(filters=64, kernel_size=(3, 3), activation='relu',
trainable=training))
    if training:
        model.add(tf.keras.layers.Dropout(0.7))
    model.add(tf.keras.layers.Flatten(trainable=training))
    model.add(tf.keras.layers.Dense(units=100, activation='relu', trainable=training))
    model.add(tf.keras.layers.Dense(units=50, activation='relu', trainable=training))
    model.add(tf.keras.layers.Dense(units=10, activation='relu', trainable=training))
    model.add(tf.keras.layers.Dense(units=1, name="OutputLayer", trainable=training))

    return model

def get_nvidia_mod(input_dim, training=True):
    # Input layers.
    image = tf.keras.layers.Input(shape=list(input_dim), name="InputImage")
    velocity = tf.keras.layers.Input(shape=(1,), name="InputVelocity")

    # Feature extraction CNN backbone.
    x = tf.keras.layers.Lambda(lambda x: tf.transpose(x, perm=(0, 2, 3, 1)))(image) #
Convert to NHWC again for efficiency.
    x = tf.keras.layers.Conv2D(filters=24, kernel_size=(5, 5), strides=(2, 2),
activation='relu', trainable=training)(x)
    x = tf.keras.layers.Conv2D(filters=36, kernel_size=(5, 5), strides=(2, 2),
activation='relu', trainable=training)(x)
    x = tf.keras.layers.Conv2D(filters=48, kernel_size=(5, 5), strides=(2, 2),
activation='relu', trainable=training)(x)
    x = tf.keras.layers.Conv2D(filters=64, kernel_size=(3, 3), activation='relu',
trainable=training)(x)
    x = tf.keras.layers.Conv2D(filters=64, kernel_size=(3, 3), activation='relu',
trainable=training)(x)
    if training:
        x = tf.keras.layers.Dropout(0.25)(x)
    x = tf.keras.layers.Flatten(trainable=training)(x)

    # Steering head.
    steer = tf.keras.layers.Dense(units=100, activation='relu', trainable=training)(x)
    if training:
        steer = tf.keras.layers.Dropout(0.1)(x)
    steer = tf.keras.layers.Dense(units=50, activation='relu',
trainable=training)(steer)
    if training:
        steer = tf.keras.layers.Dropout(0.1)(x)
    steer = tf.keras.layers.Dense(units=10, activation='relu',
trainable=training)(steer)
    if training:
        steer = tf.keras.layers.Dropout(0.1)(x)
    steer = tf.keras.layers.Dense(units=1, name="Steer", trainable=training)(steer)

    # Throttle head.
    throttle = tf.keras.layers.concatenate([x, steer, velocity], trainable=training)
    if training:
        throttle = tf.keras.layers.Dropout(.5)(throttle)
    throttle = tf.keras.layers.Dense(10, activation='relu',
trainable=training)(throttle)
    throttle = tf.keras.layers.Dense(1, name='Throttle', trainable=training)(throttle)

    model = tf.keras.models.Model(inputs=[image, velocity], outputs=[steer, throttle],
name='pilotnet_backbone_net')

    return model

def get_vgg16(input_dim, training=True):
    image = tf.keras.layers.Input(shape=list(input_dim), name="InputImage")
    velocity = tf.keras.layers.Input(shape=(1,), name="InputVelocity")

    # Load pretrained VGG16 as the backbone.
    backbone =
tf.keras.applications.vgg16.VGG16(input_tensor=tf.keras.layers.Permute((2, 3,
1))(image), weights='imagenet', include_top=False) # Convert to NHWC again for
efficiency.
    for layer in backbone.layers:
        layer.trainable = False

    # Flattening for fully-connected.
    x = backbone.output
    x = tf.keras.layers.Flatten(trainable=training)(x)
    if training:
        x = tf.keras.layers.Dropout(.25)(x)
    x = tf.keras.layers.Dense(100, activation='relu', trainable=training)(x)

    # Steering head.
    steer = tf.keras.layers.Dense(50, activation='relu', trainable=training)(x)
    steer = tf.keras.layers.Dense(10, activation='relu', trainable=training)(steer)
    steer = tf.keras.layers.Dense(1, name='Steer', trainable=training)(steer)

    # Throttle head.
    throttle = tf.keras.layers.concatenate([x, steer, velocity], trainable=training)
    if training:
        throttle = tf.keras.layers.Dropout(0.5)(throttle)
    throttle = tf.keras.layers.Dense(10, activation='relu',
trainable=training)(throttle)
    throttle = tf.keras.layers.Dense(1, name='Throttle', trainable=training)(throttle)

    # Define model with multiple inputs and outputs.
    model = tf.keras.models.Model(inputs=[image, velocity], outputs=[steer, throttle],
name='vgg16_backbone_net')

    return model
```

## ANNEX IV

```
class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = torch.flatten(x, 1) # flatten all dimensions except batch
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

## ANNEX V

```
# Export the model
x = torch.randn(batch_size_test, 1, 28, 28).cuda() # Dummy input
torch.onnx.export(network,                   # model being run
                  x,                         # model input (or a tuple for multiple
inputs)
                  "cifar10.onnx",            # where to save the model (can be a file or
file-like object)
                  export_params=True,        # store the trained parameter weights
inside the model file
                  opset_version=7,           # the ONNX version to export the model to
                  verbose=True,
                  input_names = ['input'],   # the model's input names
                  output_names = ['output'], # the model's output names
                  dynamic_axes={'input' : {0 : 'batch_size'},    # variable length axes
                                'output' : {0 : 'batch_size'}})
```

## ANNEX VI

```
def save_model(model):
    input_names = [input.op.name for input in model.inputs]
    output_names = [output.op.name for output in model.outputs]

    sess = tf.keras.backend.get_session()

    graphdef = sess.graph.as_graph_def()
    graphdef = TransformGraph(
                            graphdef,
                            input_names,
                            output_names,
                            [
                                'remove_nodes(op=Identity, op=Switch, op=CheckNumerics,
op=StopGradient, op=Placeholder)',
                                'merge_duplicate_nodes',
                                'strip_unused_nodes(type=float, shape="1,66,200,3")',
                                'fold_constants(ignore_errors=true)',
                                'fold_batch_norms',
                                'fold_old_batch_norms',
                                'strip_unused_nodes',
                                'sort_by_execution_order'
                            ])
    frozen_graph = tf.graph_util.convert_variables_to_constants(sess, graphdef,
output_names)
    frozen_graph = tf.graph_util.remove_training_nodes(frozen_graph)

    model_path = os.path.join(OUTPUT_DIR, "nvidia_mod.pb")
    with open(model_path, "wb") as ofile:
        ofile.write(frozen_graph.SerializeToString())

    print(graphdef)

save_model(model)

def model_path_to_uff_path(model_path):
    uff_path = os.path.splitext(model_path)[0] + ".uff"
    return uff_path

model_path = os.path.join(OUTPUT_DIR, "nvidia_mod.pb")
dynamic_graph = gs.DynamicGraph(model_path)
graph_def = dynamic_graph.as_graph_def()
json_graph = json_format.MessageToJson(graph_def)
with open(f"{model_path.split('.')[0]}.json", 'w') as f:
    json.dump(json.loads(json_graph), f)
```

```
graph_def = tf.compat.v1.GraphDef()
with open(f"{model_path.split('.')[0]}.json", 'r') as f:
    json_format.Parse(json.dumps(json.load(f)), graph_def)

# Save resulting graph to UFF file
output_uff_path = model_path_to_uff_path(model_path)
uff.from_tensorflow(
    graph_def,
    [output.op.name for output in model.outputs],
    output_filename=output_uff_path,
    text=True
)
```