AMITY UNIVERSITY

——GURUGRAM——

AMITY SCHOOL OF ENGINEERING AND TECHNOLOGY

SUMMER INTERNSHIP REPORT

UNDERSTANDING OF NEURAL NETWORKS USED FOR END-TO-END AUTONOMOUS DRIVING

SUBMITTED BY

NAME- Yoshita
ENROLMENT NUMBER- A50568421003
COURSE- M.Tech Data Science
Year/Semester- Final Year/ Third Sem

SUBMITTED TO

Dr. XYZ

ACKNOWLEDGEMENTS

ABSTRACT

Table of Contents

1	INTRODUCTION

In recent years, various self-driving technologies have been developed by Google, Uber, Tesla and other technology companies. Though still in its infancy, self-driving technology is becoming increasingly common and could radically transform our transportation system. For achieving its goals an autonomous vehicle needs to understand its surrounding, plan the route and make correct judgments when interacting with other objects on the road.

To accomplish all this, it relies on the following key technologies –

- **Front Camera –**

  A camera mounted on the windshield helps the car see objects in front of it. This camera also detects and records information about road signs and traffic lights, which is intelligently interpreted by the car's software The camera is considered to be the eye of the vehicle through which it sees the world around it.

- **Ultrasonic Sensors –**

  An ultrasonic sensor on one or more of the rear wheels help keep track of the movements of the car and will alert the car about the obstacles in the rear. It uses the concept of doppler effect to track the distance and relative velocity between it and the other stationary and non-stationary objects.

- **LIDAR –**

  An autonomous car is driven along the route and maps out the route and it's road conditions including poles, road markers, road signs and more. This map is fed into the car's software helping the car identify what is a regular part of the road. As the car moves, LIDAR generates a detailed 3D map of the environment at that particular moment which is used for semantic segmentation which means detailing objects in each and every pixel of image. The use of LIDAR is a bit controversial as some researchers believe that it could be replaced with two cameras both inclined so as to map a perspective image of what the car sees from the front. This could result in a drop in prices as LIDAR is quite expensive.

- **Aerial –**

  An aerial on the rear of the vehicle receives information about the precise location of the car. The car's GPS inertial navigation unit works with the sensors to help the car localize itself. As the vehicle moves, the vehicle's internal map is updated with new positional information displayed by the sensors which help the car understand where it is at a particular instance of time.

- **Computer Software –**

  This is the heart of self-driving cars. It is responsible for processing all of the data received in real time as input from the various sensors and acting based on it with the means of various actuators. So essentially the role of computer software is to process the inputs, plot a path and send instructions to the actuators to control acceleration, braking and steering.

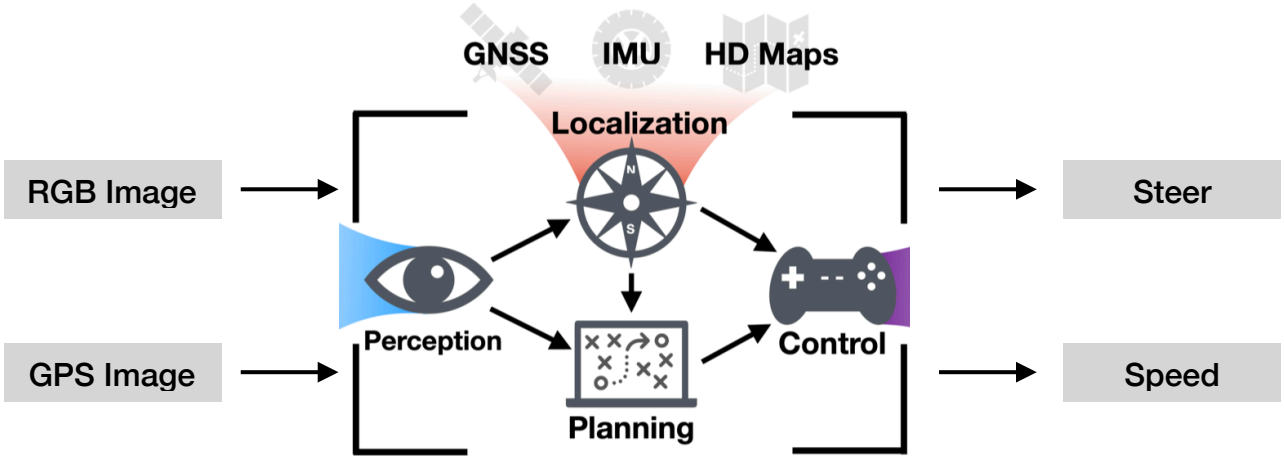
  Autonomous vehicle technology as progressed considerably in the past decade due to the increased processing capabilities of the processors and improvement in accuracy of deep learning algorithms. Sensor fusion combined with advancements in computer vision technology is probably going to change the landscape of transportation. Although there are still a lot of work to be done but researchers are convinced that it's just a matter of time when these vehicles would be a common place on the roads.

Majorly, the research on autonomous driving can be divided into two main approaches:

(i) Modular approach

(ii) End- to-End approach

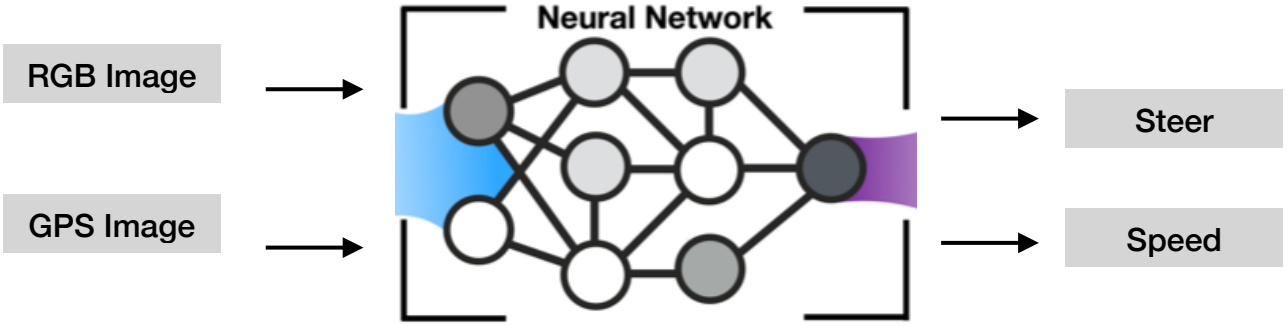
The modular approach, also known as the mediated perception approach, is widely used by the industry and is nowadays considered the conventional approach. The modular systems stem from architectures that evolved primarily for autonomous mobile robots and that are built of self-contained but inter-connected modules such as perception, localization, planning and control . As a major advantage, such pipelines are interpretable – in case of a malfunction or unexpected system behavior, one can identify the module at fault. Nevertheless, building and maintaining such a pipeline is costly and despite many man-years of work, such approaches are still far from complete autonomy.



End-to-end driving, also known in the literature as the behavior reflex, is an alternative to the aforementioned modular approach and has become a growing trend in autonomous vehicle research.

This approach proposes to directly optimize the entire driving pipeline from processing sensory inputs to generating steering and acceleration commands as a single machine learning task. The driving model is either learned in supervised fashion via imitation learning to mimic human

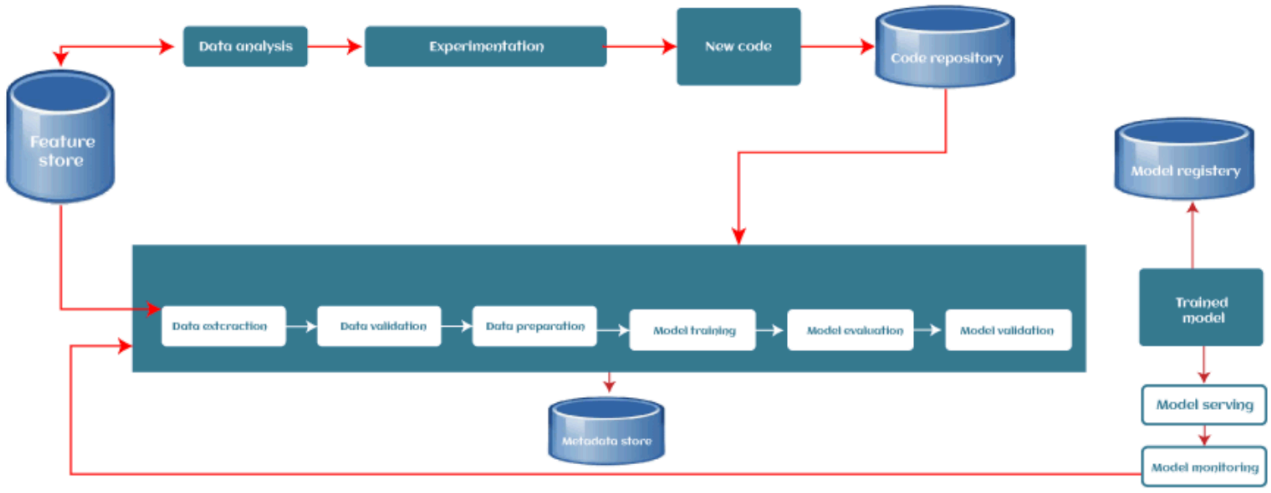
drivers , or through exploration and improvement of driving policy from scratch via reinforcement learning. Usually, the architecture is simpler than the modular stack with much fewer components . While conceptually appealing, this simplicity leads to problems in interpretability – with few intermediate outputs, it is difficult or even impossible to figure out why the model misbehaves.



Comparison??

- A Machine Learning pipeline is a process of automating the workflow of a complete machine learning task.
- It can be done by enabling a sequence of data to be transformed and correlated together in a model that can be analyzed to get the output.
- A typical pipeline includes raw data input, features, outputs, model parameters, ML models, and Predictions. Moreover, an ML Pipeline contains multiple sequential steps that perform everything ranging from data extraction and pre-processing to model training and deployment in Machine learning in a modular approach.
- It means that in the pipeline, each step is designed as an independent module, and all these modules are tied together to get the final result.

A typical ML pipeline includes the following stages:



1. Data Ingestion

Each ML pipeline starts with the Data ingestion step. In this step, the data is processed into a well-organized format, which could be suitable to apply for further steps. This step does not perform any feature engineering; rather, this may perform the versioning of the input data.

2. Data Validation

The next step is data validation, which is required to perform before training a new model. Data validation focuses on statistics of the new data, e.g., range, number of categories, distribution of

categories, etc. In this step, data scientists can detect if any anomaly present in the data. There are various data validation tools that enable us to compare different datasets to detect anomalies.

3. Data Pre-processing

Data pre-processing is one of the most crucial steps for each ML lifecycle as well as the pipeline. We cannot directly input the collected data to train the model without pr-processing it, as it may generate an abrupt result.

The pre-processing step involves preparing the raw data and making it suitable for the ML model. The process includes different sub-steps, such as Data cleaning, feature scaling, etc. The product or output of the data pre-processing step becomes the final dataset that can be used for model training and testing. There are different tools in ML for data pre-processing that can range from simple Python scripts to graph models.

4. Model Training & Tuning

The model training step is the core of each ML pipeline. In this step, the model is trained to take the input (pre-processed dataset) and predicts an output with the highest possible accuracy.

However, there could be some difficulties with larger models or with large training data sets. So, for this, efficient distribution of the model training or model tuning is required.

This issue of the model training stage can be solved with pipelines as they are scalable, and a large number of models can be processed concurrently.

5. Model Analysis

After model training, we need to determine the optimal set of parameters by using the loss of accuracy metrics. Apart from this, an in-depth analysis of the model's performance is crucial for the final version of the model. The in-depth analysis includes calculating other metrics such as precision, recall, AUC, etc. This will also help us in determining the dependency of the model on features used in training and explore how the model's predictions would change if we altered the features of a single training example.

6. Model Versioning

The model versioning step keeps track of which model, set of hyperparameters, and datasets have been selected as the next version to be deployed. For various situations, there could occur a significant difference in model performance just by applying more/better training data and without

changing any model parameter. Hence, it is important to document all inputs into a new model version and track them.

7. Model Deployment

After training and analyzing the model, it's time to deploy the model. An ML model can be deployed in three ways, which are:

- Using the Model server,
- In a Browser
- On Edge device

However, the common way to deploy the model is using a model server. Model servers allow to host multiple versions simultaneously, which helps to run A/B tests on models and can provide valuable feedback for model improvement.

8. Feedback Loop

Each pipeline forms a closed-loop to provide feedback. With this close loop, data scientists can determine the effectiveness and performance of the deployed models. This step could be automated or manual depending on the requirement. Except for the two manual review steps (the model analysis and the feedback step), we can automate the entire pipeline.

2	SYSTEM REQUIREMENTS

SOFTWARE:

- **OS :** Ubuntu 18
- **Python Packages:**
  - Torch 1.71
  - Torchvision 0.8.2
  - PIL 8.2.0
  - Numpy 1.19.5
  - Opencv 4.4.0
  - Os-noetic
  - Rospkg
  - Rospy
  - GPU driver version 470.103.01
  - CUDA version 11.4

HARDWARE:

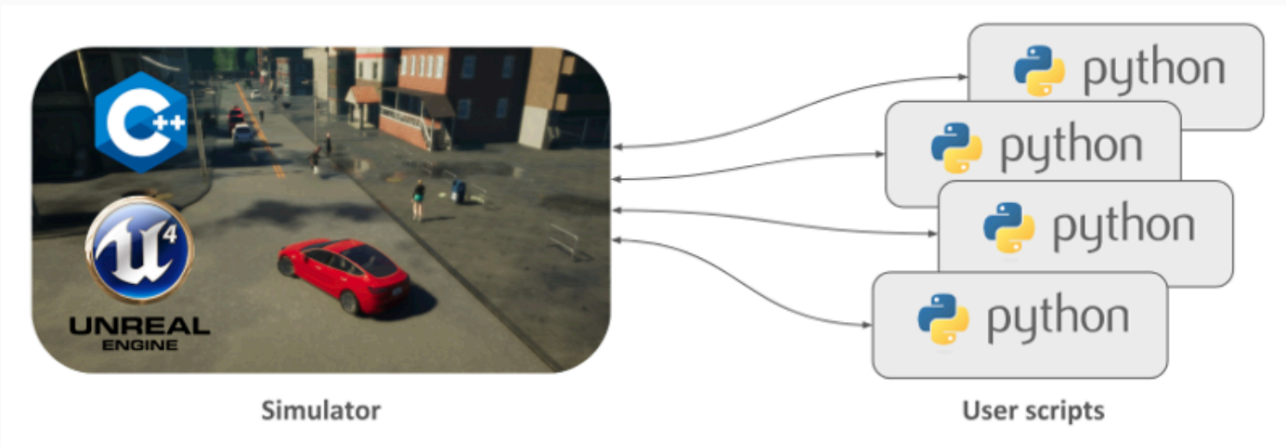
- **GPU :** Nvidia 1080Ti
- **GPU memory :** 12GB
- **System RAM :** 24 GB
- **Processor** : Intel Xeon
- **CPU cores:** 12

3	DATA COLLECTION

- CARLA is an open-source autonomous driving simulator. It was built from scratch to serve as a modular and flexible API to address a range of tasks involved in the problem of autonomous driving. One of the main goals of CARLA is to help democratize autonomous driving R&D, serving as a tool that can be easily accessed and customized by users.
- To do so, the simulator has to meet the requirements of different use cases within the general problem of driving (e.g. learning driving policies, training perception algorithms, etc.).
- CARLA is grounded on Unreal Engine to run the simulation and uses the OpenDRIVE standard (1.4 as today) to define roads and urban settings. Control over the simulation is granted through an API handled in Python and C++ that is constantly growing as the project does.

The simulator

- The CARLA simulator consists of a scalable client-server architecture.The server is responsible for everything related with the simulation itself: sensor rendering, computation of physics, updates on the world-state and its actors and much more. As it aims for realistic results, the best fit would be running the server with a dedicated GPU, especially when dealing with machine learning.
- The client side consists of a sum of client modules controlling the logic of actors on scene and setting world conditions. This is achieved by leveraging the CARLA API (in Python or C++), a layer that mediates between server and client that is constantly evolving to provide new functionalities.



- The CARLA simulator consists of a scalable client-server architecture.
  The server is responsible for everything related with the simulation itself: sensor rendering, computation of physics, updates on the world-state and its actors and much more. As it aims for realistic results, the best fit would be running the server with a dedicated GPU, especially when dealing with machine learning.
  The client side consists of a sum of client modules controlling the logic of actors on scene and setting world conditions. This is achieved by leveraging the CARLA API (in Python or C++), a layer that mediates between server and client that is constantly evolving to provide new functionalities.
- That summarizes the basic structure of the simulator. Understanding CARLA though is much more than that, as many different features and elements coexist within it. Some of these are listed hereunder, as to gain perspective on the capabilities of what CARLA can achieve.
  - **Traffic manager.** A built-in system that takes control of the vehicles besides the one used for learning. It acts as a conductor provided by CARLA to recreate urban-like environments with realistic behaviours.

  - **Sensors.** Vehicles rely on them to dispense information of their surroundings. In CARLA they are a specific kind of actor attached the vehicle and the data they receive can be retrieved and stored to ease the process. Currently the project supports different types of these, from cameras to radars, lidar and many more.

  - **Recorder.** This feature is used to reenact a simulation step by step for every actor in the world. It grants access to any moment in the timeline anywhere in the world, making for a great tracing tool.

  - **ROS bridge and Autoware implementation.** As a matter of universalization, the CARLA project ties knots and works for the integration of the simulator within other learning environments.

  - **Open assets.** CARLA facilitates different maps for urban settings with control over weather conditions and a blueprint library with a wide set of actors to be used. However, these elements can be customized and new can be generated following simple guidelines.

- **Scenario runner.** In order to ease the learning process for vehicles, CARLA provides a series of routes describing different situations to iterate on. These also set the basis for the CARLA challenge, open for everybody to test their solutions and make it to the leaderboard.