A DEEP LEARNING AND ROBOTICS PLATFORM FOR AUTONOMOUS DRIVING, DEVELOPED FOR TEACHING AND RESEARCH

A Project
Submitted to
The Temple University Graduate Board

---

in Partial Fulfillment
of the Requirements for the Degree of
MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

---

by
Animesh Bala
December 2021

Examining Committee Members:

Dr. Li Bai, Department of Electrical and Computer Engineering, College of Engineering
Dr. Saroj K. Biswas, Department of Electrical and Computer Engineering, College of Engineering

ABSTRACT

A Deep Learning and Robotics Platform for Autonomous Driving, Developed for Teaching and Research

by

Animesh Bala

Deep Learning, a subsection of Machine Learning (ML) has become very popular in recent years. A rapid growth of appliance of deep learning models and frameworks is visible in both industry and research. While making predictions on a dataset, compared to handcrafted models, data driven deep learning models are performing significantly better, thanks to the availability of high-performance Graphical Processing Units (GPU) developed in recent years, and accumulated piles of data collected from different digital sensors and devices. Therefore, it is unavoidable to learn the basics of the pipeline of deep learning framework for anyone who have focused their career on industry and research. Inspired by the importance of this new framework, we developed an end-to-end deep learning platform, which can be used as a tool to teach the pipeline and basics of a deep learning framework to the people who are completely new to this environment. To solve a specific problem with this framework as demonstration we chose autonomous driving, which is a rapidly growing industry and depends majorly on data driven deep learning models for making decisions. The platform also utilizes real robot cars, which will make this platform more interesting to young minds, who are aspired to learn. The autonomous driving technology is also at its cradle right now with a potential of massive growth; therefore, this platform can also be utilized developing computer vision and machine learning algorithms for self-driving cars. Our developed robotic and simulation software along with the deep learning framework will aid the researchers avoiding less important tasks regarding tuning and developing wrapper packages, and they can focus more on specific sectors related to their research. Keeping these all in consideration, we can say that our developed platform will be a great addition as a tool for both teaching and research regarding autonomous driving and deep learning based technologies.

ACKNOWLEDGEMENTS

I want to thank my advisor Dr. Li Bai, without his suggestions and constant support, developing this complex platform could never be possible. He always came up with novel ideas whenever I got stuck and didn't find any solution even on the internet. The courses I attended taught by him also helped me learn the basics of Robotic Operating System, Gazebo Simulation and Amazon Web Service, which played a major role in this project.

I also want to thank Dr. Saroj Biswas, without his advice and constant support I could never face the difficulties I had during my academic years. The course I attended taught by him also helped me obtain the mathematical foundation to learn the basics of Image Processing, Computer Vision and Deep Learning, which played significant role in this project.

Finally, I want to thank Michael Nghe, who took his valuable time to refine the dataset visually, which was indeed a tiresome task. That dataset was a major foundation for the deep learning framework we developed in this project.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

GPU	Graphics Processing Unit
CPU	Central Processing Unit
ROS	Robot Operating System
CSI	Camera Serial Interface
URDF	Unified Robot Description Format
ONNX	Open Neural Network Exchange
CNN	Convolutional Neural Network

CHAPTER 1

INTRODUCTION

Autonomous vehicle industry has gained massive popularity in recent years due to the paradigm shift of computer vision algorithms. Since 2010, during ImageNet large scale visual recognition challenge [1] on ImageNet dataset [2], deep learning based models achieved very high accuracy compared to handcrafted computer vision models [3]–[6]. The availability of large image dataset and high performance GPU paved this path for deep learning based computer vision algorithms. Since then, various end-to-end deep learning based models have been developed for driving vehicles autonomously [7]–[11]. The major advantage of deep learning based computer vision algorithms is, a vehicle can make decisions depending only on camera images in real time, instead of using numerous sensors. Which makes the whole system very simple compared to non-deep learning based model approaches. Inspired by this advantage, different miniature robot car based teaching platforms have been developed in recent years, which are simple for young minds. MIT has developed a robot car platform along with a $1/10^{th}$ scale robot car to teach students about the robotic software system [12]. DeePicar, build on Raspberry Pi 3, a similar robotic platform is developed replicating Nvidia's DAVE-2 system focusing on autonomous driving [13] for same purpose. Brigham Young University also developed another self-driving robot car platform, focusing on learning and research [14]. Apart from education, amazon has developed DeepRacer [15] and arranged a competition focusing on reinforced learning. Moreover, there are different competitions such as Micromouse competition [16], which are developed based on similar robot car platforms.

Inspired by these works, we developed a generalized deep learning and robotic software platform which can be used on any robot car with a microprocessor capable of running Linux operating system. We tested the robotic software on both Raspberry Pi 4 and Jetson Nano, two majorly used microprocessors for robot development. However, considering the availability of GPU in Jetson Nano [17], and Nvidia already made commercially available a robot car schematic called JetBot [18] with Jetson Nano as microprocessor, we developed our default robot software focusing on this robot car.

CHAPTER 2

OVERVIEW OF THE PLATFORM

The whole platform is divided into two major subsections. One of them is dedicated for robotic software development, and another is for deep learning applications. These two platforms work separately and independently. They are related only by the annotated dataset and generated deep learning models. The overview of these two platforms is discussed in following subsections.

ROBOTICS PLATFORM:

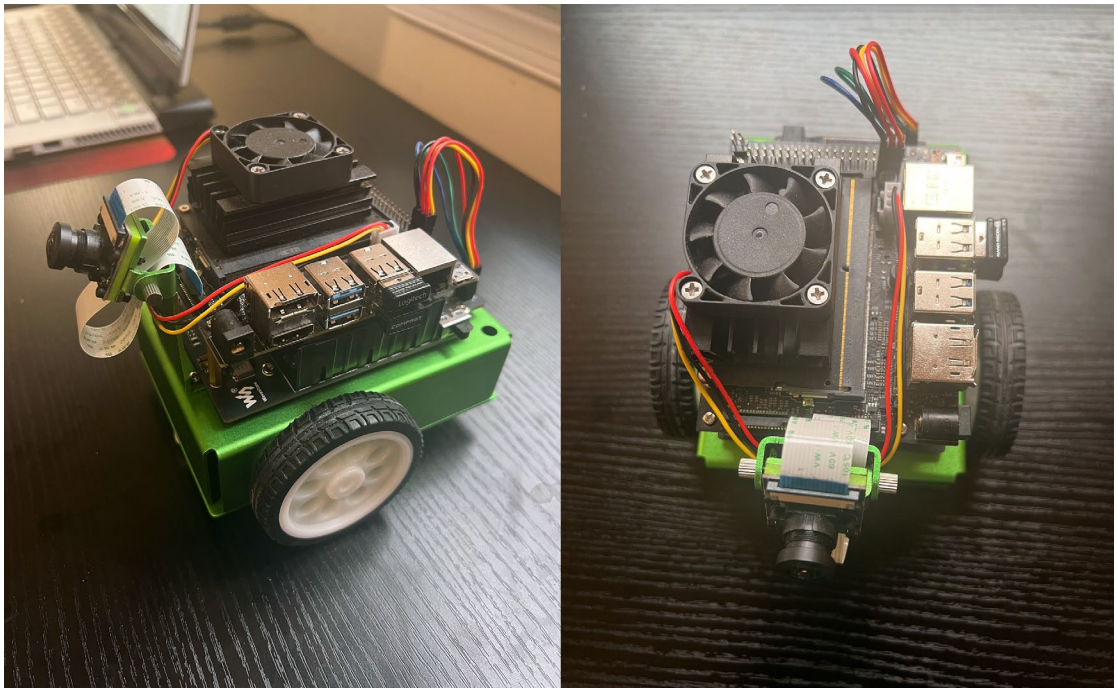

Figure 1: Robot Car JetBot.

The Robotics Platform is developed in Linux operating system. This platform is dedicated to run a robot car both for data collection and autonomously. A simulation environment with simulated robot car is also developed to test the performance of computer vision algorithms and deep learning models before deploying them on real robot. Therefore, the

complete robotics platform is divided into two separate workspaces. They are robotic workspace and simulation workspace. The robotic workspace runs a real robot car controlled by gamepad during data collection session and runs the robot autonomously with a trained deep learning model. The simulation workspace has the same functionality but performs them on simulated robots.

We used JetBot as robot car to develop our default software in robot workspace. JetBot AI Kit Accessories manufactured by Waveshare is used to build this robot and Jetson Nano 4GB developed by Nvidia is used a microprocessor for this robot. Figure 1 shows the JetBot we've built for this project. The advantage of using a Jetson Nano over commonly used Raspberry Pi 4 is Jetson Nano has a dedicated GPU, which aids deep leaning processes run lot faster by doing multiprocessing, which is impossible with regular CPUs. This is a major requirement for autonomous vehicles to make steering decisions as quick as possible using deep neural networks. For developing robot software tools, we used Robot Operating System 2 (ROS2), a set of software libraries and tools that helps to build robot applications and packages [19]. ROS2 can be installed in any supported Linux operating system.

The simulation workspace also does the same thing, but inside a Gazebo simulation world. Gazebo is an open-source 3D robotics simulator which is commonly used to develop robotic software. We developed a generalized simulation platform with ROS2 support to simulate any simulation robot inside a given simulation world environment. While developing this simulation workspace, we used AWS RoboMaker [20], a web service provided by Amazon. The advantage of this web service is it comes with all robot and simulation libraries (ROS2, Gazebo etc.) already installed. Therefore, the user doesn't have to put his valuable time on setting up the whole environment from scratch. Moreover, it provides a very user friendly graphical interface, which is important for a teaching platform developed for young minds.

We also developed different software tools and ROS2 packages to run simulation robot cars both with keyboard commands for data collection, and autonomously with a trained deep neural network. The ROS2 packages dedicated for robot car control are interchangeable with robot workspace. The keyboard command tools are developed

because AWS doesn't give gamepad controllers direct access to their instances. However, if anyone wants to use this platform in personal computer with Linux operating system, he/she can totally use the ROS2 packages developed for gamepad controller in the simulation workspace.

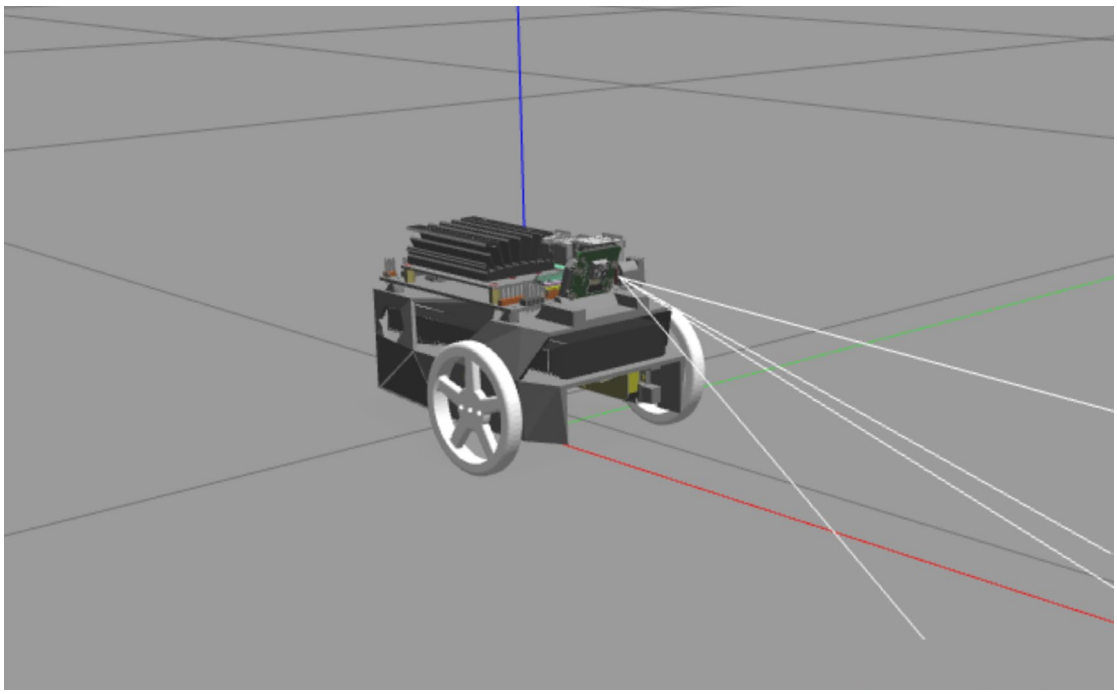

Figure 2: JetBot in Gazebo Simulation.

While developing these simulation packages, we used a simulation robot mimicking the JetBot provided by Nvidia to test our developed ROS2 packages. Figure 2 demonstrates the simulated JetBot in Gazebo simulation world. We used an empty world here. The JetBot simulation files provided by Nvidia was developed for ROS1 platform. Therefore, we had to convert it to ROS2 to make it available for our simulation software packages. The reason for using ROS2 over ROS1 for developing our robot and simulation packages is, ROS2 supports Python 3, unlike Python 2 for ROS1. It was unavoidable because most of the deep learning and computer vision tools are being developed in Python 3.

DEEP LEARNING PLATFORM:

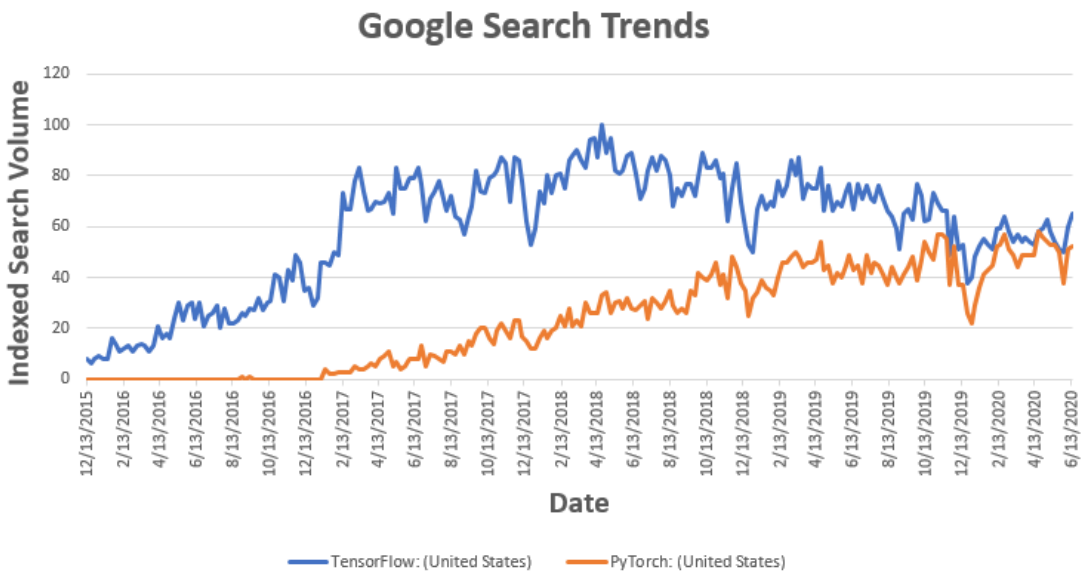

Figure 3: Google Search Trend of PyTorch and TensorFlow in US.

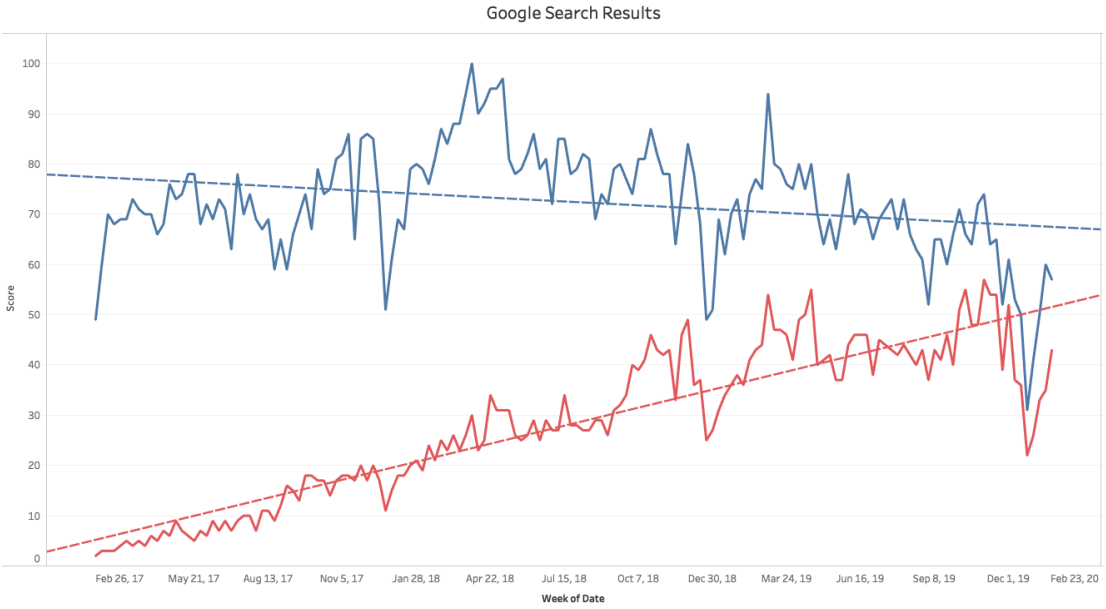

Figure 4: How PyTorch is catching up TensorFlow.

The deep learning platform contains all required software and tools to create proper dataset with annotations, preprocess dataset by discarding outliers, train a deep neural network with the refined dataset, inspect different parameters during training session, test performance of the trained model etc. We also developed visualization tools to inspect the

performance of trained models visually. The platform is developed to run it on a laptop or desktop, which makes the platform significantly easy to use, especially for young minds. For software development we used Python as core language. OpenCV module is used for the development of computer vision tools, which is an open source package developed in Python language. For developing the deep learning tools, we used PyTorch, which is one of the commonly used deep learning framework developed in Python. The reason why we preferred PyTorch over TensorFlow is, PyTorch framework is more Pythonic. Therefore, and it is lot easier to debug any deep learning software developed in PyTorch. The environment setup in any device regarding installation with GPU support is also straightforward for PyTorch, unlike TensorFlow. Figure 3 shows how PyTorch is becoming more popular in recent years [21]. Figure 4 shows how the trend line of PyTorch is catching up TensorFlow in 2020 [22]. Considering all these, PyTorch is obvious choice for a platform developed for teaching and research.

Apart from the PyTorch models, ONNX models are also generated. The advantage of ONNX models is, they can be used directly as a filter in a supported runtime, which takes images as input and generates predictions as output. This straightforward way of generating results makes the prediction software very simple and lightweight. This approach comes handy while developing ROS2 packages. The complexity of the packages decreases significantly if an ONNX model is used, unlike going with a PyTorch model. Moreover, ONNX runtime also provides GPU support. Which means we won't have to worry about processing speed even with ONNX model. We can take full advantage of the GPU, which comes with a Jetson Nano.

CHAPTER 3

ROBOTICS PLATFORM

```
Robotics Platform
    ├── Robot Workspace
    │   └── src
    │       ├── ROS2 Gamepad to Twist Message
    │       ├── ROS2 Deep Learning to Twist Message
    │       ├── ROS2 Twist Message to Robot Motion
    │       ├── ROS2 CSI Camera Publish
    │       ├── ROS2 Save Camera Image
    │       └── Robot App
    └── Simulation Workspace
        └── src
            ├── ROS2 Keyboard to Twist Message
            ├── ROS2 Deep Learning to Twist Message
            ├── ROS2 Save Camera Image
            ├── ROS2 World Simulation
            ├── ROS2 Robot Simulation
            └── Simulation App
```

Figure 5: Directory tree of ROS2 packages in Robotics Platform.

As we mentioned in previous chapter, the robotics platform is developed on ROS2, which is a set of software libraries and tools for building robot applications. Considering the procedure of application, we divided the robotics platform into two different workspaces. The Robot Workspace, which contains all packages to run a real robot. And Simulation Workspace, which contains all packages to spawn and run a simulated robot inside a Gazebo simulation world. Figure 5 shows the directory tree of ROS2 packages in Robotics Platform. The details of these packages will be discussed in following subsections.

ROBOT WORKSPACE:

Figure 6 shows the overview and data flow of the Robot Workspace. The Robot Workspace contains ROS2 packages which runs a robot car with a gamepad controller for data collection. It also contains ROS2 packages to run a robot car autonomously with trained deep neural networks. The function of these packages is provided in following sections.

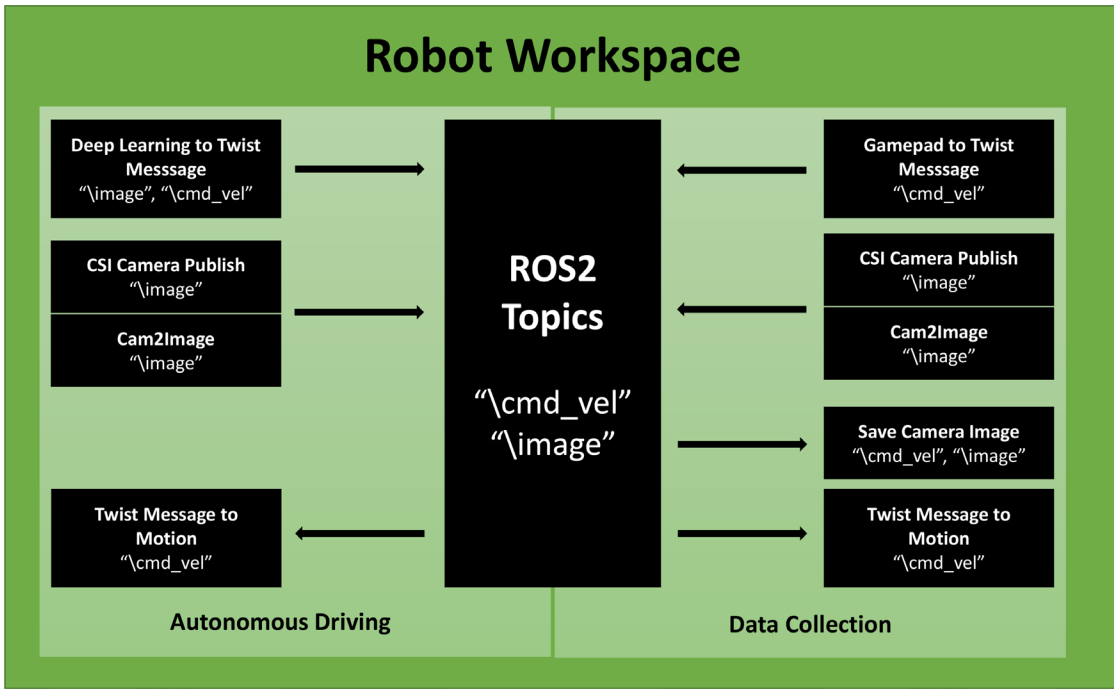

Figure 6: Overview of Robot Workspace.

ROS2 Gamepad to Twist Message: The $ros2\_gamepad\_to\_twist$ package is developed to collect data from a gamepad controller and publish that data to a ROS2 topic. A topic is a method of communication among different packages in ROS2 environment [23]. Any package which is subscribed to that topic can read and utilize that published data. Our default developed packages collects data from a Logitech or Waveshare gamepad controller and publishes that data in default $/cmd\_vel$ topic as ROS2 $geometry\_msgs/Twist$ [24] format. The $settings.json$ file can be modified to change the default publish topic. Table 1 shows the key mappings from gamepad controller data to $geometry\_msgs/Twist$ format. The values for $linear.x$ and $angular.z$ are ranged from -1 to $+1$.

Gamepad Controller	$geometry\_msgs/Twist$
$Left\ Joystick\ Up$	$+\ linear.x$
$Left\ Joystick\ Down$	$-\ linear.x$
$Right\ Joystick\ Right$	$+\ angular.z$
$Right\ Joystick\ Left$	$-\ angular.z$

Table 1: Key mappings from Gamepad to $geometry\_msgs/Twist$ format.

ROS2 Deep Learning to Twist Message: The $ros2\_deep\_learning\_to\_twist\_message$ package is developed to run a deep learning model converted to ONNX format and make steering and robot movement decision. The package is subscribed to $/image$ topic to read camera images and make prediction from computer vision algorithms and deep learning models. The predicted values are converted to $geometry\_msgs/Twist$ format and published to $/cmd\_vel$ topic. The functionality is completely same as Gamepad to Twist Message package. The only difference is, we are generating control data using deep learning models and computer vision algorithms instead of a gamepad controller. The $settings.json$ file can be modified to change the input image shape, subscription topic for camera image, and publish topic, along with the input image shape for ONNX model. Although, the default executable uses ONNX models as deep learning models, the package can be used as wrapper to implement support for other model types.

ROS2 Twist Message to Robot Motion: The $ros2\_twist\_message\_to\_robot\_motion$ package is developed to move the robot in real world. The default package is developed for Adafruit Motor HAT module controlled robots, and we finetuned it for JetBot, which is developed with same motor controller. However, the package can be used as a wrapper to create robot movement package for other robots. The package subscribes to $/cmd\_vel$ topic, reads that data generated from either gamepad controller or deep learning model, and runs the robot accordingly. The $settings.json$ file can be modified to change the subscription topic along with some calibrations on robot movement. Table 2 shows the key mappings of $geometry\_msgs/Twist$ data for the JetBot movement.

$geometry\_msgs/Twist$	JetBot Movement
$+ linear.x$	Forward
$- linear.x$	Backward
$+ angular.z$	Right
$- angular.z$	Left

Table 2: Key mappings of $geometry\_msgs/Twist$ data for JetBot movement.

ROS2 CSI Camera Publish: ROS2 by default provides a package called $image\_tools$ with $cam2image$ extension, which is dedicated to publishing images collected by camera to $/image$ topic. This default package works properly if we use it in Raspberry Pi or Jetson Nano with a webcam. However, problem occurs when we use CSI (Camera Serial Interface) camera in Jetson Nano to collect images. Because Jetson Nano uses GStreamer Pipeline [25], which is incompatible with the default ROS2 package. To resolve this issue, we developed a similar package $ros2\_csi\_camera\_publish$ for publishing camera images with support for CSI camera. Moreover, the $settings.json$ file provides some modifications available on the output image, unlike the default ROS2 package.

ROS2 Save Camera Image: To train a deep learning model, it is important to collect annotated data. The $ros2\_save\_camera\_image$ package is dedicated to performing that job. The package is subscribed to two topics, $/cmd\_vel$ and $/image$ topic. It collects robot control data from $/cmd\_vel$ topic and use them as annotation for images collected from $/image$ topic. The control values are collected in range of -1 to $+1$, which are mapped from 0 to 10 in integer format.

$$mapped\ value = 10 * (control\ value + 1)$$

Then these values are used as annotations for respective images. Since this process is linear, we can always go back to actual $geometry\_msgs/Twist$ values. While saving the annotations, instead of creating a separate file we put this values in the names of saved images. For example, if we obtain $linear.x = 0.2$ and $angular.z = 0.6$, the package

will save the image with $0000000\_z08\_x06.jpg$ name. This not only simplifies the database, but also saves memory spaces. In the Deep Learning Platform section, we will demonstrate how linear mapping doesn't affect the accuracy of deep learning models.

Robot App: The Robot App launches multiple packages and performs a particular task. For example, *Gamepad to Twist Message*, *Twist Message to Robot Motion*, *CSI Camera Publish*, and *Save Camera Image* packages are required to run the JetBot with a gamepad controller and collect annotated data. To run the robot autonomously, *Deep Learning to Twist Message*, *Twist Message to Robot Motion*, and *CSI Camera Publish* are required. These packages are launched at the same time with the Robot App.

SIMULATION WORKSPACE:

Figure 7 shows the overview of the Simulation Workspace. The Simulation Workspace contains following ROS2 packages we developed, which runs a simulated robot car inside a Gazebo simulation world both for data collection and autonomously. The simulation world can be run on any Linux operating system with Gazebo and ROS2 installed. However, instead of developing the package in a Linux operated computer, we developed them in AWS RoboMaker, which comes with all required software installed and provides a better user interface. The details of the developed packages are discussed in following sections.

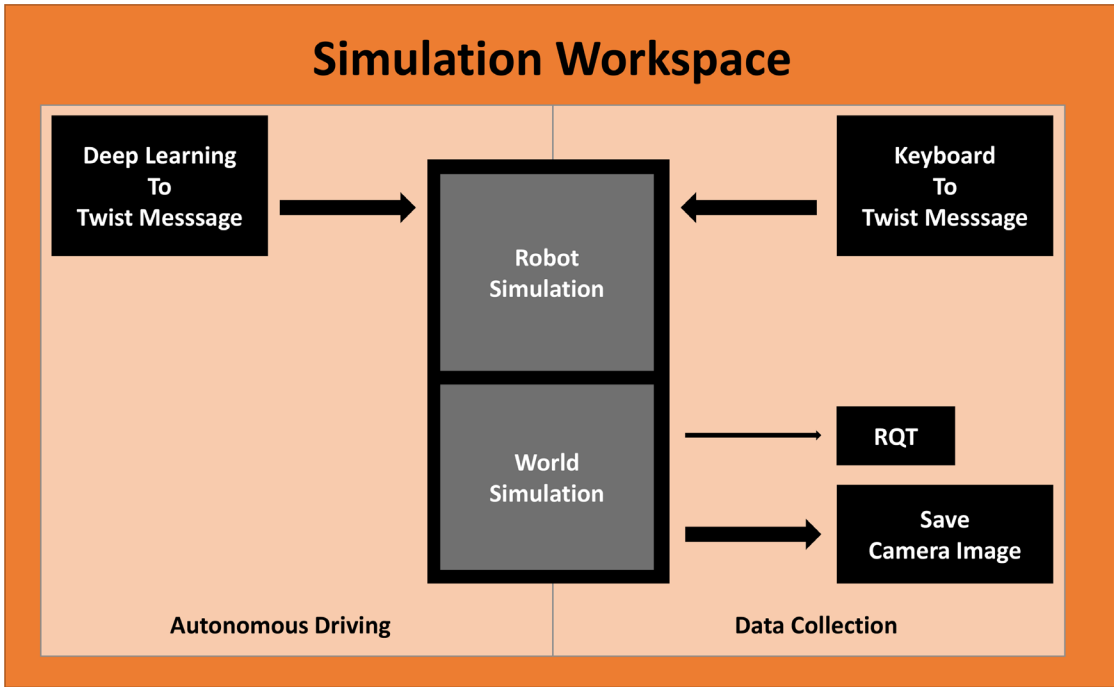

Figure 7: Overview of Simulation Workspace.

ROS2 Keyboard to Twist Message: Although we already developed a package *ros2_gamepad_to_twist_message* for gamepad controllers, however AWS doesn't let gamepads to connect to their instances directly. In that case, during simulation in AWS RoboMaker, we cannot operate the simulated robots with gamepads. Therefore, we developed this additional *ros2_keyboard_to_twist_message* package which is allowed to run in AWS instances. The key mapping of this package is shown in Table 3. The

keyboard input data is published to $/cmd\_vel$ topic in $geometry\_msgs/Twist$ format. This is completely in same format as how $ros2\_gamepad\_to\_twist\_message$ package and $ros2\_deep\_learning\_model\_to\_twist\_message$ package publish data. The $settings.json$ file can be modified to change the subscription topic. Since the package takes input from command line, it needs to be run independently instead of launching it with another ROS2 package launcher.

Keyboard Buttons	$geometry\_msgs/Twist$
W	$+ linear.x$
S	$- linear.x$
A	$+ angular.z$
D	$- angular.z$
L	$linear.x = 0.0$
K	$angular.z = 0.0$

Table 3: Key mappings from Keyboard to $geometry\_msgs/Twist$ format.

ROS2 World Simulation: It is important to simulate a world first inside the Gazebo simulator. And everything is simulated in that world afterwards. We developed the $ros2\_world\_simulation$ package to take care of this world simulation job. The package is developed to launch any world file in Gazebo simulation. Only a world file and related model files are required to be provided inside specific directory. This flexibility allows the users to develop any kind of simulation worlds for robot simulation. The package can be use for development of any simulation project, which could be even completely unrelated to robot car driving. For default simulation, we used both Empty World and AWS RoboMaker Racetrack World.

ROS2 Robot Simulation: Once a world is simulated inside Gazebo simulator, we can spawn anything in that simulation world. The $ros2\_robot\_simulation$ package is developed to take care of this job. The package not only spawns a robot, but also monitors its status and publish them to different default ROS2 topics at constant rate. The package

is developed in a way that any robot can be spawned inside a simulation world. Providing a robot URDF file inside a specific model directory is enough to spawn that robot. Once a robot is spawned, this package publishes its status from Gazebo simulator constantly with $joint\_state\_publisher$ and $robot\_state\_publisher$, two default ROS2 packages dedicated for this job. For default robot simulation, we used the simulation file mimicking JetBot, as we've mentioned in previous chapter.

Packages from Robot Workspace: For autonomous driving of real robot, we used $ros2\_deep\_learning\_to\_twist\_message$ in robot workspace. We used the same package for autonomous driving of robots in simulation world. Since we are using same ROS2 libraries as foundation to build both robot and simulation packages, it allows us to interchange packages between these workspaces. The same thing we did for saving annotated camera images, which is being done by $ros2\_save\_camera\_image$ package developed in robot workspace. The advantage of this approach is the generated annotated data is totally similar for both robot and simulation workspace. Therefore, we don't need two different pipelines for training deep learning models. Moreover, we can use same trained models in both robot and simulation workspace.

Simulation App: Same as the $robot\_app$ package, the $simulation\_app$ launches multiple packages to perform a particular task. During data collection session, it launches *Save Camera Image, World Simulation*, and *Robot Simulation* packages at the same time. While running the robot autonomously, *World Simulation*, and *Robot Simulation* packages are launched instead. The $simulation\_app$ package takes care of this job.

CHAPTER 4

DEEP LEARNING PLATFORM

```
Self Driving CV Repository
    ├── data process
    │   ├── output
    │   ├── settings
    │   ├── batch_rename.py
    │   ├── create_dataset.py
    │   ├── create_final_dataset.py
    │   ├── refine_dataset.py
    │   └── visually_refine_dataset.py
    ├── dataset
    │   ├── images
    │   └── lists
    │       ├── cross-validation
    │       ├── debug
    │       └── random
    │           ├── train.csv
    │           ├── val.csv
    │           └── test.csv
    ├── deep learning
    │   ├── train
    │   │   ├── checkpoints
    │   │   ├── output
    │   │   ├── config.py
    │   │   ├── utils.py
    │   │   ├── dataset.py
    │   │   ├── model.py
    │   │   └── train.py
    │   └── test
    │       ├── test.py
    │       └── test_onnx.py
    ├── results
    └── visualization
        ├── prediction_visualization.py
        └── prediction_video.py
```

Figure 8: Directory Tree of Deep Learning Platform.

Figure 8 shows the directory tree of the deep learning platform. The deep learning platform is designed to develop computer vision and deep learning tools, which are required for autonomous driving. The platform is developed in Python language and build upon OpenCV and PyTorch framework. The whole platform can be divided into four units which works independently. The details of this units are discussed in following sections.

DATA PROCESSING UNIT:

The data processing unit contains all software and tools written in Python to prepare proper dataset for a deep learning session. At first, we need to create a list of all our collected camera images so that we can easily find them during deep learning sessions. The $create\_dataset.py$ script takes care of that job. There is another important step, which is performed with this script. Since the annotations are embedded in image names to keep the database simple, we must parse these annotations first. The $create\_dataset.py$ automatically does that for every image and stores those values in a 2 dimensional array along with the corresponding image path. Then it creates a CSV file containing a list of the path of the images along with annotations. Table 4 shows a sample of the created list with 5 entries, where z and x are annotations for $angular.z$ and $linear.x$ values.

images	z	x
03_13_2020_1\output_0003\0000720_z01_x06.jpg	1	6
03_09_2020_1\output_0024\0000519_z02_x07.jpg	2	7
03_16_2020_0\output_0005\0000146_z06_x06.jpg	6	6
03_10_2020_1\output_0006\0000621_z02_x06.jpg	2	6
03_16_2020_0\output_0001\0000012_z06_x06.jpg	6	6

Table 4: A sample of entries in CSV file.

Once a CSV file with list of the complete dataset is created, we can perform different preprocessing jobs to remove outliers for avoiding overfitting and making the trained models more robust. The $refine\_dataset.py$ and $visually\_refine\_dataset.py$ scripts take care of these jobs. The $refine\_dataset.py$ script automatically removes outliers using different algorithms. However, if human intervention is required for removing outliers, the $visually\_refine\_dataset.py$ script can be used. Once the dataset is refined from the outliers, we can create train, test, and validation dataset to run the deep learning sessions with $create\_final\_dataset.py$ script. Both random and sequential dataset can be created with this script. Additionally, with some efforts, n-fold cross validation dataset can also be created using this script as reference.

DATASET:

Figure 9: Samples from Image Database.

We created a dataset of 100,526 images collected through different robot running session with proper annotations. A $1/10^{th}$ size miniature robot car was driven on a miniature racetrack, and image data was collected with a CSI camera attached to the forehead of robot car. Figure 9 shows some samples from image database. The images were collected in different environment condition to make the trained models more robust. Then the primarily collected dataset was refined by our data processing tools to remove outliers. All images are discarded first when the robot car was stationary position. We only allowed images when the robot car was moving. Afterwards the dataset was refined by visualization tools by human inspection. Once we were satisfied with the dataset with minimum outliers, we divided the refined dataset into train, test, and validation datasets. The final datasets contain 57,417 annotated images for training, 7,178 annotated images for validation, and 7,178 annotated images to run test session. Moreover, we created a debug dataset with only 100 images to debug our deep learning and computer vision tools. Additionally, we created a 5 fold cross validation database make the trained model more robust.

Apart from the created lists, it is important to mention about the annotations. The $angular.z$ and $linear.x$ which we are using here as annotations for driving robot cars, are generated in the range from -1 to $+1$. However, we mapped these data from 0 to 10 range to embed these annotations inside corresponding image names. Before training

session, we used another transformation, normalizing the annotations by mapping them in the range from 0 to 1. Therefore, it is important to investigate how these linear transformations on annotations affect the training session and their accuracy. Figure 10 shows how different range of mapping for $angular.z$ affects the training session and the accuracy achieved in different epochs. It is obvious that linear transformation on annotation doesn't affect the accuracy. However, normalized dataset is preferred for deep learning session, which is why we used 0 to 1 mapping while training neural networks. Moreover, this 0 to 1 mapping achieved highest accuracy, although the difference is very small, and took less epochs to reach that point during training session.

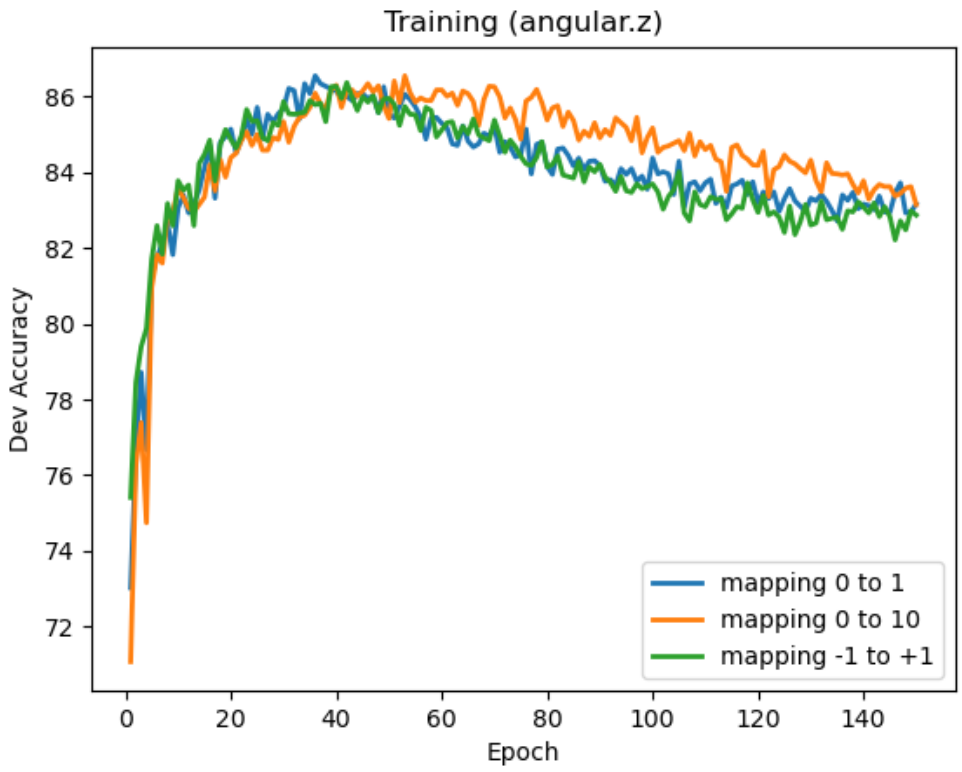

Figure 10: Accuracy achieved in different Epochs for different Annotation Mappings.

DEEP LEARNING UNIT:

The deep learning unit contains all deep learning tools to train and test a deep learning model for autonomous driving. The scripts are written in Python, and we built the deep learning pipeline on PyTorch framework. The growing popularity of PyTorch over TensorFlow and easy debugging approach are the reason why we chose PyTorch. To create an end-to-end learning pipeline, we chose regression approach with deep neural networks to predict the control values ($angular.z$ and $linear.z$) for autonomous driving. Nvidia's DAVE-2 system for autonomous driving used similar approach, therefore used their deep neural network to develop our deep learning platform. Figure 11 shows the CNN architecture of Nvidia's DAVE-2 system [26]. The $model.py$ file contains this architecture. However, one can easily modify this file with another novel or state of art deep neural network and train that model using our platform. This approach comes handy while using this platform as teaching tool for designing deep neural networks.

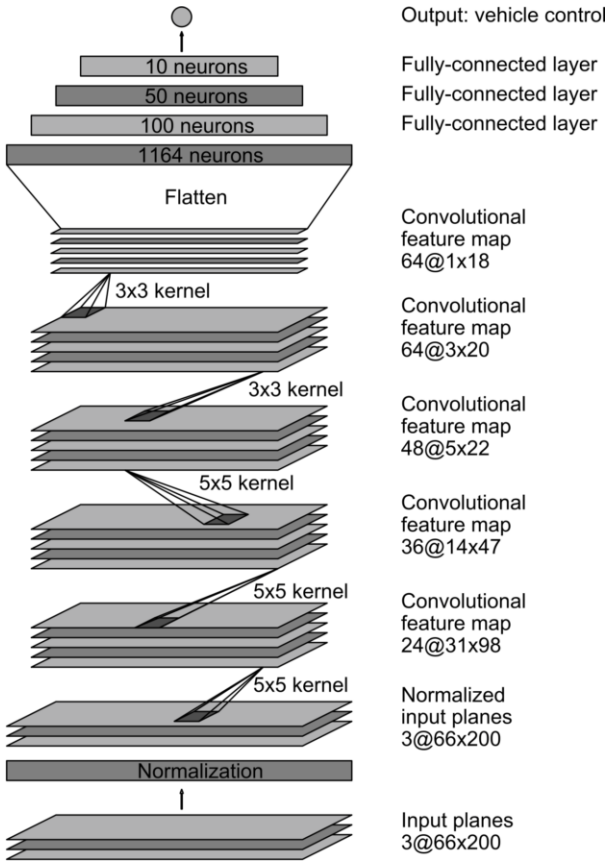

Figure 11: CNN architecture of Nvidia's DAVE-2 system.

To make the trained models more robust to different environment and lighting setup, it is important to use an augmented dataset along with the training dataset. PyTorch provides different transformations, which can be performed during training session. This approach doesn't require creating a separate database for augmented data and saves memory. Following this approach we included color jitter transformation in the pipeline, to make the training models more robust and avoid overfitting. Additionally, the annotations are normalized and mapped from the range of 0 to 1.

The whole unit is divided into two main sections. One section is dedicated for training deep learning models, another section is for testing performance. The $train.py$ script takes care of training session, and $test.py$ takes care of testing session. The $model.py$ contains the architecture of deep neural network, as mentioned earlier. The $config.py$ file contains different hyperparameters, and $utils.py$ contains different tools dedicated for showing outputs and loading or saving checkpoints. The $output$ directory contains logs and plots generated during training session. The $checkpoints$ directory contains trained weights, which are saved after every epoch. The models are saved in both default PyTorch $pth.tar$ format and ONNX format. Along with the $test.py$ script, the test section also contains a $test\_onnx.py$ script, which tests the performance of ONNX models and compare them with the PyTorch model. Both should provide similar result with same accuracy.

VISUALIZATION UNIT:

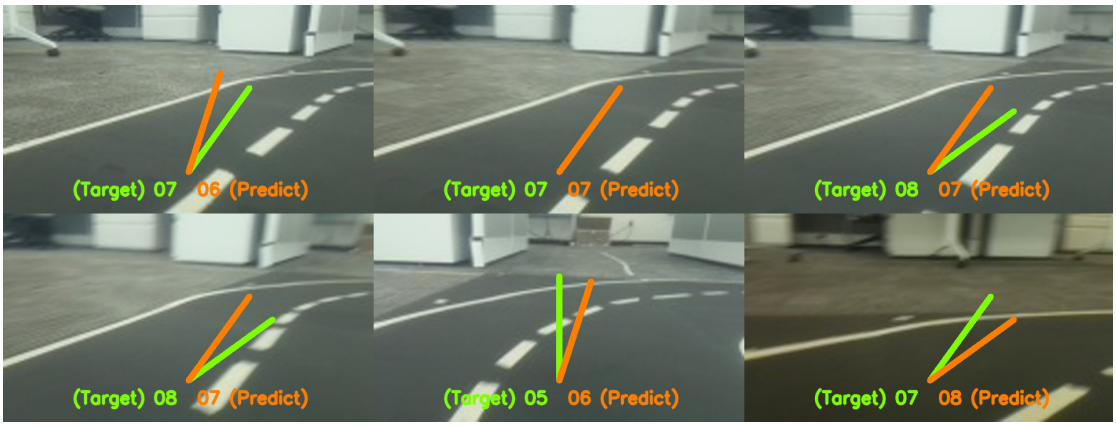

Figure 20: Samples generated with Visualization Tools.

The visualization unit contains computer vision tools to generate output in a more perceivable way for human eyes. The generated images contain proper texts and steering direction markers so that one can easily measure the performance of trained models visually. Moreover, a video clip generation tool is also developed which takes a sequential dataset and generates a video clip with steering direction from both annotation and prediction. Figure 20 demonstrates some samples generated with visualization tools. Moreover, the *visually_refine_dataset.py* script in Data Processing Unit is also developed following similar algorithms.

CHAPTER 5

CONTRIBUTION TO TEACHING AND RESEARCH

While developing this platform, our objective was to use this platform as a tool for teaching and research, instead of running a robot car autonomously with a trained neural network. Therefore, we developed the default robot packages for JetBot, a commercially available miniature robot car, which can be built under 350 USD. We also created a GitHub repository, which provides instructions to setup a JetBot with all dependencies very easily. The installed JupyterLab provides a easy to use user interface for developing robot softwares and packages. The AWS RoboMaker also provides an easy to use user interface to develop simulation software and packages. Additionally, the deep learning platform is written entirely in Python, which means one can easily use the whole platform in a personal computer or laptop. All these makes the platform very easy to use for both researchers and while using it as a tool teaching. Additionally, our whole platform is developed in three separate sections, which works separately and independently. The robot workspace, simulation workspace, and deep learning platform. Therefore, each section can be used for teaching or research independently.

While developing this platform as teaching tool, we kept two things in consideration. How someone who doesn't have any knowledge on programming language going to use the platform, and how someone with basic programming language uses it. We prepared single line command for running robot both with gamepad and autonomously. The same thing is applicable for simulation. In this way, those who are interested in learning deep learning pipeline with our platform, can test the performance of their models on a real or simulation robot very easily. Moreover, the deep learning platform already provides a annotated and refined dataset. Scripts for running training and testing sessions are also provided. Therefor one can easily modify the model.py file and learn how different deep neural networks performs by using our provided scripts. This helps someone to learn the pipeline of deep learning without advanced knowledge on Python programming language. Additionally, one can use this platform to teach robotic software too without touching the deep learning platform. We trained the neural network of Nvidia's DAVE-2 system which achieves 84%

accuracy with 10% tolerance on our prepared dataset. This model can be used for autonomous driving while using the robotic and simulation workspace for teaching robotic software development.

While using the platform for research, it provides great flexibility. Since our simulation platform can spawn any world file and robot model file, it can be used to develop robots in Gazebo simulation world. The user can easily use our simulation workspace to link their simulated world and robot with ROS environment. For robot software development, one can either directly use our packages or use our packages as wrapper to create new robotic software packages. The deep learning platform takes care of the basic pipeline; therefore, one can easily concentrate their effort on developing novel neural networks. Moreover, the platform can use as a reference to create a software platform for autonomous driving.

CHAPTER 6

CONCLUSION AND FUTURE WORK

Although the platform is developed focusing on teaching and research, it can be used different areas. We are working on a completely different project named Micromouse 2 competition where a robot must move inside a maze, find an object, and push it to a desired location. There we are using this platform as baseline software platform. Since the same JetBot is being used as robot car for that project, we are using different packages from the robot workspace of this platform. The Micromouse 2 project also require maze world simulated in Gazebo, which is being done with our simulation workspace packages. For object detection, it is important to collect image data, preprocess them, and train deep learning models. Our deep learning platform is coming handy in that job. Additionally new robot packages and deep learning software are being developed for that project taking this platform as reference. Moreover, this platform is used to teach functionality of Robot Operating System and AWS RoboMaker service in Spring 2021 in an independent study course. It was very successful, more graduate level courses are being planned to teach robotics and deep learning pipeline. Considering these we can say, it is an important addition as a tool for teaching, research, and industry.

BIBLIOGRAPHY

[1] O. Russakovsky *et al.*, “ImageNet Large Scale Visual Recognition Challenge,” *International Journal of Computer Vision*, vol. 115, no. 3, pp. 211–252, Dec. 2015, doi: 10.1007/S11263-015-0816-Y.

[2] J. Deng, W. Dong, R. Socher, L.-J. Li, Kai Li, and Li Fei-Fei, “ImageNet: A large-scale hierarchical image database,” pp. 248–255, Mar. 2010, doi: 10.1109/CVPR.2009.5206848.

[3] K. He, X. Zhang, S. Ren, and J. Sun, “Deep Residual Learning for Image Recognition,” *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, vol. 2016-December, pp. 770–778, Dec. 2015, doi: 10.1109/CVPR.2016.90.

[4] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “ImageNet Classification with Deep Convolutional Neural Networks”, Accessed: Nov. 22, 2021. [Online]. Available: http://code.google.com/p/cuda-convnet/

[5] C. Szegedy *et al.*, “Going Deeper with Convolutions.” 2015. Accessed: Nov. 22, 2021. [Online]. Available: https://research.google/pubs/pub43022/

[6] K. Simonyan and A. Zisserman, “Very Deep Convolutional Networks for Large-Scale Image Recognition,” Sep. 2014, Accessed: Nov. 22, 2021. [Online]. Available: http://arxiv.org/abs/1409.1556

[7] M. Bojarski *et al.*, “End to End Learning for Self-Driving Cars,” Apr. 2016, Accessed: Nov. 22, 2021. [Online]. Available: https://arxiv.org/abs/1604.07316

[8] J. Kaiser *et al.*, “Towards a framework for end-to-end control of a simulated vehicle with spiking neural networks,” *2016 IEEE International Conference on Simulation, Modeling, and Programming for Autonomous Robots, SIMPAR 2016*, pp. 127–134, Feb. 2017, doi: 10.1109/SIMPAR.2016.7862386.

[9] S. Chen, S. Zhang, J. Shang, B. Chen, and N. Zheng, “Brain Inspired Cognitive Model with Attention for Self-Driving Cars,” Feb. 2017, Accessed: Nov. 22, 2021. [Online]. Available: http://arxiv.org/abs/1702.05596

[10] V. Rausch, A. Hansen, E. Solowjow, C. Liu, E. Kreuzer, and J. K. Hedrick, “Learning a deep neural net policy for end-to-end control of autonomous vehicles,” *Proceedings of the American Control Conference*, pp. 4914–4919, Jun. 2017, doi: 10.23919/ACC.2017.7963716.

[11] J. Kim and C. Park, “End-To-End Ego Lane Estimation Based on Sequential Transfer Learning for Self-Driving Cars,” *IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*, vol. 2017-July, pp. 1194–1202, Aug. 2017, doi: 10.1109/CVPRW.2017.158.

[12] S. Karaman *et al.*, "Project-based, collaborative, algorithmic robotics for high school students: Programming self-driving race cars at MIT," *ISEC 2017 - Proceedings of the 7th IEEE Integrated STEM Education Conference*, pp. 195–203, Apr. 2017, doi: 10.1109/ISECON.2017.7910242.

[13] M. G. Bechtel, E. McEllhiney, M. Kim, and H. Yun, "DeepPicar: A Low-cost Deep Neural Network-based Autonomous Car," Dec. 2017, Accessed: Nov. 22, 2021. [Online]. Available: http://arxiv.org/abs/1712.08644

[14] J. Newman, Z. Sun, and D. J. Lee, "Self-Driving Cars: A Platform for Learning and Research," *2020 Intermountain Engineering, Technology and Computing, IETC 2020*, Oct. 2020, doi: 10.1109/IETC47856.2020.9249142.

[15] "AWS DeepRacer - the fastest way to get rolling with machine learning." https://aws.amazon.com/deepracer/ (accessed Nov. 22, 2021).

[16] "Competitions | Micromouse USA." http://micromouseusa.com/?page_id=23 (accessed Nov. 29, 2021).

[17] "NVIDIA Jetson Nano Developer Kit | NVIDIA Developer." https://developer.nvidia.com/embedded/jetson-nano-developer-kit (accessed Nov. 23, 2021).

[18] "Home - JetBot." https://jetbot.org/master/ (accessed Nov. 23, 2021).

[19] "ROS 2 Documentation — ROS 2 Documentation: Foxy documentation." https://docs.ros.org/en/foxy/index.html (accessed Nov. 23, 2021).

[20] "AWS RoboMaker - Amazon Web Services." https://aws.amazon.com/robomaker/ (accessed Nov. 23, 2021).

[21] "Pytorch vs Tensorflow in 2020. How the two popular frameworks have… | by Moiz Saifee | Towards Data Science." https://towardsdatascience.com/pytorch-vs-tensorflow-in-2020-fe237862fae1 (accessed Nov. 30, 2021).

[22] "Is PyTorch Catching TensorFlow?. The State of Deep Learning Frameworks… | by Jeff Hale | Towards Data Science." https://towardsdatascience.com/is-pytorch-catching-tensorflow-ca88f9128304 (accessed Nov. 30, 2021).

[23] "Understanding ROS 2 topics — ROS 2 Documentation: Foxy documentation." https://docs.ros.org/en/foxy/Tutorials/Topics/Understanding-ROS2-Topics.html (accessed Nov. 23, 2021).

[24] "geometry_msgs/Twist Documentation." http://docs.ros.org/en/melodic/api/geometry_msgs/html/msg/Twist.html (accessed Nov. 23, 2021).

[25] "Foundations." https://gstreamer.freedesktop.org/documentation/application-development/introduction/basics.html?gi-language=c (accessed Nov. 23, 2021).

[26] “End-to-End Deep Learning for Self-Driving Cars | NVIDIA Developer Blog.” https://developer.nvidia.com/blog/deep-learning-self-driving-cars/ (accessed Nov. 30, 2021).