CSCI 5302: Final Project Report (Team Cruella)

Aaptha Boggaram, Nick Zarilla, Amiy Yadav, Srikrishna Bangalore Raghu, Suhas Srinivasa Reddy

I. INTRODUCTION

The CSCI 5302 course is designed in such a way that we can incorporate almost everything learned in class into our project. The goal of this project is to create an autonomous car using the AWS DeepRacer hardware provided to us, along with ROS2-Foxy. Aside from that, there are no restrictions on how we implement any of the challenges. With that said, to begin with, the following are the hardware and software tools provided for this project:

A. Hardware

The following hardware components were given to us (main components):

- AWS DeepRacer chassis with attached cameras and Lidar.
- Batteries, adapters, and cables.
- Portable power bank for the car.

B. Software

The following software tools were pre-installed on the provided DeepRacer:

- Robot Operating System 2 - Foxy
- Ubuntu 20.04
- AWS simulator using the DeepRacer IP address

Additionally, we had the freedom to install any libraries and packages needed for our final project.

In summary, after receiving the hardware and tools, our initial tasks involved setting up the DeepRacer and commencing the implementation of challenges. The subsequent sections provide detailed insights into the work undertaken, and comprehensive literature surveys.

II. SETUP

Upon receiving the DeepRacer, we set up the car's WiFi to utilize AWS features via the car's IP address. Within the AWS DeepRacer portal, various options were provided to test the Lidar sensor, cameras, and wheels of the DeepRacer. Additionally, tools were supplied for manual calibration of wheel speeds and alignment, ensuring optimal car movement.

With the initial setup complete, to get the car up and running, you can follow one of the two setups:

- Wired - Connect the DeepRacer to a monitor and run the Ubuntu 20.04 installed on it to write all codes and test.
- Wireless - Use PuTTY or any other software to SSH into the car's IP address and wirelessly edit the files and run the DeepRacer.

In our final project, we employed a combination of both setups. The wired setup was predominantly used during development to write scripts, while the wireless setup was utilized for ground testing of the DeepRacer.

III. GOALS

There were many challenges given to us that we could implement, but we were required to choose and implement a total of 8 points. As graduate students, we had to tackle at least one 5-point challenge. With that said, the following sections explain the workings of each challenge we implemented for this project. To view them in a list, the challenges we will be implementing are provided below:

- SLAM Implementation (5 pts)
- Stop-Sign Detection (1 pts)
- Dynamic Collision Avoidance (2 pts)
- Vehicle Telemetry Visualization (1 pts)
- Literature Survey
  - Deep Learning for Autonomous Control (1 pts)
  - Learning from Demonstration (1 pts)
- Reverse Driving (2 pts)

In addition to completing these challenges, we also had to navigate our DeepRacer around a race course under time constraints.

IV. SLAM IMPLEMENTATION

For this challenge, we used an open-source package for creating the map that publishes the generated map to the `/map` topic. This package is called **gmapping**. While this package is not available for ROS2, we used a wrapper that allows us to use it for the software we use in this project. The map can be visualized either in RviZ or via our visual telemetry implementation (specified in section VIII). This is the "mapping" portion of our SLAM implementation challenge. The working images of mapping part is given below:

For our planning controller, we used a PID controller. The implementation details are given in the following subsection.

A. PID Controller

PID stands for **P**roportional, **I**ntegral, and **D**erivative controller. It is a feedback-based control system commonly used in various automation tasks, particularly in robotics. This controller continuously adjusts the output based on three parameters: P (Proportional), I (Integral), and D (Derivative).

In our implementation, we used a PID controller to adjust the steering angle of the DeepRacer to center it along a

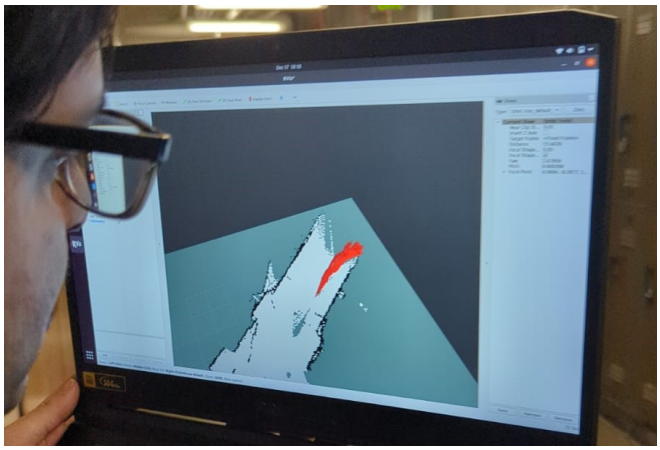

Fig. 1. Slam mapping in the basement

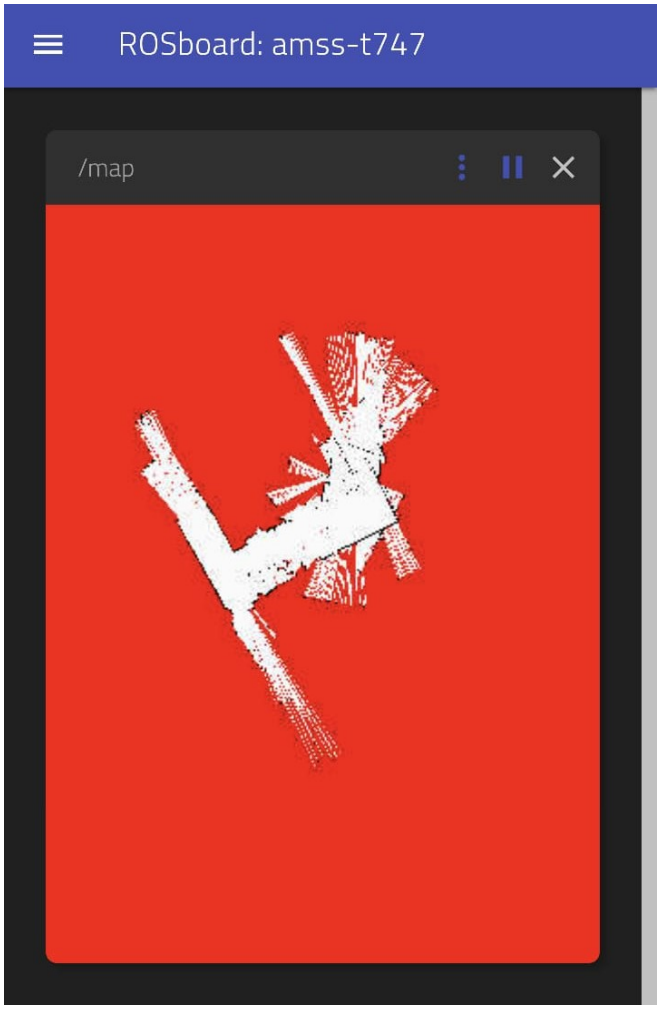

Fig. 2. SLAM map visualization on rosboard

hallway and run around a track in a loop. The pseudocode for our algorithm is given below:

```
# PID-controller pseudocode
Initialize kp, ki, kd
Initialize dt, prev_error and I(integral)

left_median = median dist on the left side of
     Lidar at a 30 degree angle interval
right_median = median dist on the right side
    of Lidar at a 30 degree angle interval

err = right_median - left_median
integral = integral + err * dt
derivative = (prev_error - err) / dt
angle = kp * err + ki * I + kd * derivative
prev_error = err
```

The range values used for this algorithm - right_median and left_median - are calculated fairly simply. First, the length of the LIDAR message being published by the RPLidar node is divided by 360 to get a ”one-degree” value for the index of the message; this index is then multiplied by the angular value we wish to examine, and rounded by transforming it into an ”int” type. Next, in a cone with a set angle to the left and right - in our implementation, a 35-degree-width cone with its initial reading at 45 degrees from center - a series of values are recorded from the RPLidar message.

These values are then added to a list (provided they are not infinity or NaN); due to the presence of floor-length glass windows to the sides of the track, it was found to be necessary to remove the highest 1/6th of the values in the list to avoid erroneously increasing the median value. Finally, the median value of the remaining readings within each list is taken as ”left_median” or ”right_median”, for the respective sections of readings.

Fine-tuning the kp, ki, and kd parameters is the most crucial part of implementing this algorithm.

- The **kp** term denotes the ”Proportional” term, and this parameter determines how aggressively the DeepRacer reacts to deviations.
- The ”Integral” parameter, i.e., **ki**, on the other hand, denotes the importance you want to give to the error term. You might want to keep this low, as we are minimizing the error and do not want to weigh it very highly.
- Lastly, **kd**, the ”Derivative” term, is a damping factor for our PID controller.

Our chosen parameters are purely based on test runs and visualizing the oscillations of the given error and steering angle. It was discovered during the course of testing that increasing the proportional scaling value above 1, regardless of the values of the other terms, results in a frequent failure to steer. This is characterized by a lack of steering angle adjustment for a period, regardless of the steering angle

published to the ServoMsg ROS topic. As such, our final tuning - and successful run - uses a fairly low proportional term of .85.

There are very few hard-coded navigation conditions included in our implementation, but as our environment contained odd floor layouts, reflective (or totally non-reflective) surfaces, such as steel kickplates on doors and porous concrete walls, glass surfaces, and myriad other difficult obstacles for our singular sensing mechanism (LIDAR), we found them to be necessary for the correction of rare navigation errors. The first of these circumstances is simply a condition to avoid collisions and to allow for correction due to steering errors. This condition simply checks for obstacles in a narrow section in front of the bot, and should one be discovered, causes the bot to stop, back up with a steering angle inverse of that which it was previously given, and then continue PID navigation as before.

The second condition is rarely satisfied, but one particular hallway involving all of the above problematic obstacles required special treatment - a check to ensure that the hallway in front of the bot was clear, the wall on the right was fairly close, and the wall on the left was a significant distance away allowed the bot to simply bypass this particular section by driving straight. The final check simply checks if the forward reading is below a specific threshold, and the right reading is above a second threshold; if this circumstance is satisfied, the bot is told to move forward for a small distance and then make a right turn. While this is effective in the event of a LIDAR failure during a turn, it is rarely used by the bot.

V. Reverse Driving

For this challenge, we used our PID controller with backward throttle and reversed the angles we used for PID. The DeepRacer finished the reverse PID lap in under 3 minutes, whereas the forward PID model finished the lap in our basement in under 1.5 minutes.

VI. Stop-Sign Detection

The main idea behind this challenge is to detect "red" in the live camera stream and stop the vehicle for a short while before continuing. We chose this approach as stop signs are whole red circular signs and detecting red would be enough to accommodate on our DeepRacer as the operation is not computationally intensive. For our final project, we created a publisher-subscriber model to perform this aspect of the challenge.

First, we create a subscriber to subscribe to the '/display_mjpeg' topic and access the CameraMsg images frame by frame. We achieve this by creating a subscriber using the code snippet below.

```
self.subscription =
    self.create_subscription(Image,
    '/display_mjpeg', self.msg_callback, 10)
```

The msg_callback function inside the subscriber utilizes the msg published to the '/display_mjpeg' topic.

This function then calls a function called stop_sign_detection(self, msg) that reads ROS 2 image msg and checks for the presence of a stop sign in it. If the stop sign is detected, the DeepRacer is stopped for a few seconds before it starts moving again.

These controls are given using the ServoCtrlMsg from deepracer_interfaces_pkg via another publisher that publishes to '/cmdvel_to_servo_pkg/servo_msg' topic. The code snippet for the publisher is as follows:

```
# the publisher class publishes another
    message to ServoCtrlMsg to control motors
self.publisher_servo =
    self.create_publisher(ServoCtrlMsg,
    '/cmdvel_to_servo_pkg/servo_msg', 10)
```

The control for the car stopping and moving is done using the following code.

```
def car_go(self):
        msg = ServoCtrlMsg()
        msg.angle = 0.0
        msg.throttle = self.speed_mod
        self.publisher_servo.publish(msg)

def car_stop(self):
        msg = ServoCtrlMsg()
        msg.angle = 0.0
        msg.throttle = 0.0
        self.publisher_servo.publish(msg)
```

One thing to note is that the '/display_mjpeg' topic contains a ROS 2 message. However, cv2 requires a HSV image. Therefore, the image msg is first converted into BGR format by using a the cv_bridge library. This library acts as a bridge between OpenCV and ROS2, converting the ROS messages to cv2 readable BGR format images. The following code is used to convert a ROS msg back to a cv2 image.

```
# converting ros image msg -> cv2 frame
## this is done to the message that is
    received from subscribing to topic

# importing library
from cv_bridge import CvBridge

# using the bridge in the class
    initialization
self.bridge = CvBridge()

# using the self attribute to convert img
    msg to cv2 bgra8 format
frame =
    self.bridge.imgmsg_to_cv2(msg,'bgra8')
```

The BGR image is converted into HSV format using the cv2 library using the following command.

```
# transforming an rgv image to hsv image
hsv_image = cv2.cvtColor(frame,
    cv2.COLOR_BGR2HSV)
```

Now that the entire workflow is in place, let's discuss the details of the stop sign function. As mentioned earlier, this function takes a cv2 image as input. The initial step involves converting the image to the HSV format. Within the HSV format, two ranges of red colors are identified, which led us to create two masks based on these ranges.

```
# mask 1 for lower red range
lower_red = cv2.inRange(hsv_image,
    (0,50,20), (5,255,255))

# mask 2 for upper red range
upper_red = cv2.inRange(hsv_image,
    (175,50,20), (180,255,255))
```

These masks are then combined using a bitwise OR operation. Subsequently, the `cv2.findContours()` function is employed to identify all areas containing red in the image. The red areas are then assessed against a specified threshold. If the area surpasses this threshold, the DeepRacer detects a stop sign, prompting the car to halt for a few seconds before resuming motion.

This concludes the description of the stop sign detection challenge in our final project. The next section will elaborate on the dynamic collision avoidance challenge.

VII. Dynamic Collision Avoidance

For this challenge, we employed the main idea of accessing 60 different indices at a 45-degree angle to the left and right of the DeepRacer. If the Lidar reading is within a certain distance of the DeepRacer, it indicates an obstacle encounter, and therefore, the car is steered away from that index before steering itself back into position. To achieve this, we implemented a publisher-subscriber model.

The publisher file basically controls the movement of the car (stopping, moving, steering) and the subscriber checks for the presence of an obstacle. The functions in the publisher that controls the movement and stopping of the car remains the same as the ones mentioned in the stop sign detection section (`car_go()` and `car_stop()`).

However, there are two new functions that are implemented in the publisher and they are the `car_steer_right()` and `car_steer_left()` functions. The `car_steer_right()` checks for an obstacle on the left of the DeepRacer at a 60 degree window. If there is a change in Lidar readings in that range, then the car steers right and steers left again to avoid that obstacle. Similarly, `car_steer_left()` function also works the same way.

The publisher, on the other hand, communicates with the vehicle motors through the '`/cmdvel_to_servo_pkg/servo_msg`' topic. Meanwhile, the subscriber receives Lidar data by subscribing to the '`/scan`' topic.

In the subscriber file, we applied a threshold of 0.5 meters. If a Lidar reading is less than this threshold, it signifies the detection of an obstacle. Consequently, we publish messages to steer right, left, stop, or move, depending on the specific scenario.

VIII. Visual Telemetry

Visual telemetry was implemented using an open-source ROS2 package named rosboard. This package, at its most basic level, functions very similarly to RViz. Its main function is to connect to all available ROS topics and directly visualize the published data. In the case of text strings, it provides a terminal-like scroll of the published text. For maps and other plots, such as 2D Lidar/scan data or maps produced by packages like gmapping, it employs the same plotting methodology as RViz.

The significant advantage of this package comes from its utilization of the Deepracer platform's ability to host a webserver. Utilizing this capability, the package reserves a port, instantiates a custom webserver using Tornado, and then uses it to publish the collected data directly to a customizable UI in a web browser. The opened websocket can be connected to from any device with a functional browser on the same network.

As an additional layer of convenience, the UI is designed to run correctly on mobile devices, allowing one to walk beside their robot while simultaneously viewing map data.

The visualization of the same can be seen in the figures below:

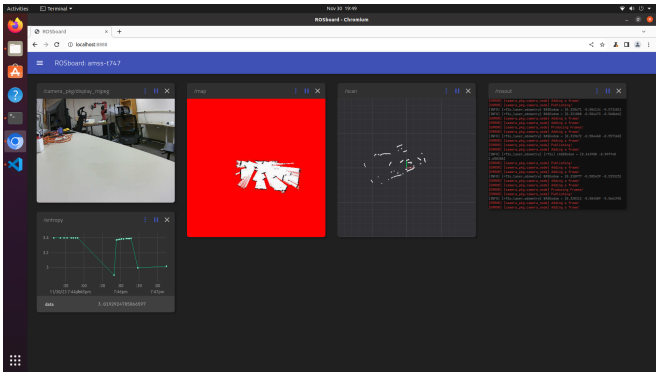

Fig. 3. Rosboard visualization

In figure, we observe the camera output, entropy, SLAM-generated map, Lidar readings, and the console output from the bring-up process. Additionally, rosboard also gives the flexibility to customize the displayed messages by utilizing

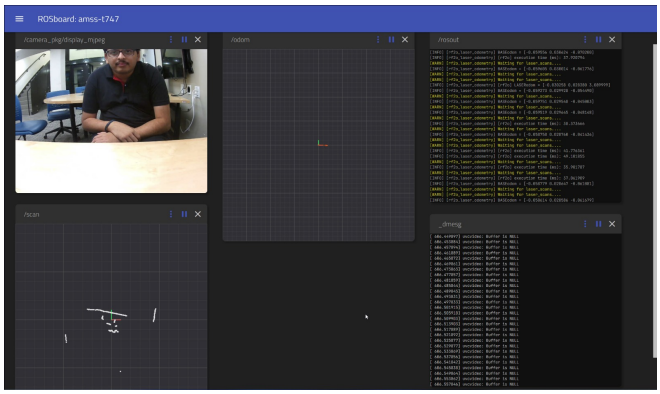

Fig. 4. Rosboard 2

a JavaScript file and specifying the published messages you wish to visualize.

IX. Bluetooth

As a personal goal (and to ease the use of the robot, as the AWS-provided manual control and manual control using a teleop twist keyboard had several seconds of latency), a pygame script was written to allow full robot control using a standard PlayStation 4 controller over Bluetooth. The first step of this was the installation of Bluetooth drivers. During this process, it was discovered that Ubuntu does not have any support for Bluetooth 5.0 or greater - the purchase of a second, older USB Bluetooth dongle (and another reinstallation and reconfiguration of the Bluetooth drivers on the robot) solved this issue.

The pygame package written to operate the controller is fairly simple, and uses only inbuilt pygame functions. The controller map is fairly intuitive - the vertical axis of the left stick controls the throttle, with the tilt mapped from -1 to 1 with a scaling value applied. The horizontal axis of the right stick operates similarly with the steering angle. The ”triangle” button, ”circle” button, and left trigger are mapped to stop commands, with the former mapped to a full emergency stop and shutdown of the ServoMsg publisher service, and the latter two simply acting as overrides that publish a 0-velocity 0-angle ServoMsg. For functionality and ease of use (especially when dealing with the bot experiencing lower motor torque due to battery depletion), the scaling factor for both the steering and the linear speed can be adjusted via the controller. The ”square” and ”cross” buttons raise and lower the linear speed multiplier by .02 (with a default value of -.80) respectively, while the ”L1” and ”R1” buttons do the same for the angular steering multiplier.

X. Obstacles and Issues

Several major obstacles were encountered during the implementation of the software for this bot. The first of these involved the constant buildup of ROS2 log files - it was discovered that every line of text produced by a ROS2 package was logged in a text file. The creation of (and constant writing to) these files during operation was causing significant slowdowns on the robot’s operating system; additionally, within one hour of operation, it was discovered that this particular log folder was accumulating over 12GB of text files - a massive issue on a robot with only 32GB of space in its storage. This was solved by instating a crontab event in Linux that deleted the contents of this folder on startup.

The next issue involved the LIDAR itself - it appeared that the performance of the hardware module was slowly degrading as we conducted tests, and we discovered fairly late in the testing and implementation process that our LIDAR had (for all intents and purposes) completely failed; in this state, it was reading either .15m or 5m to the right, and would not read any other values in that direction. Replacing the LIDAR module solved this problem, though this occurred only within the last few days of the progress.

The next problem we encountered was an issue with the drive motor and corresponding motor driver overheating. After extended use - slightly less than one lap of the track, or so - the motor begins overheating severely, resulting in poor performance and eventually total motor failure. Allowing it to cool and power-cycling the motor driver appears to fix this, but it slows down our lap times considerably.

Next, in the last two weeks of the project period, the AWS service web-server (used to calibrate the motors and steering) failed entirely, and remained inaccessible for the remainder of the project. Whether this was the fault of the AWS services themselves or the web-server running on the robot was never ascertained, but the result is that a steering offset had to be hard-coded into the PID controller to ensure consistently straight driving, and all motor calibration from that point onward had to be performed by creatively publishing ServoMsgs to the motors and then power cycling them while they were either at 0 signal or a speed of our choosing.

Finally, as discussed below, the steering driver itself had a tendency to fail to operate the steering mechanism if certain values were published as steering angles. It was found that drastically reducing the values we were publishing fixed this issue.

XI. Conclusion

In conclusion, our research endeavor encompassed the meticulous implementation of challenges worth 11 points, each integral to the overarching objective of constructing an autonomous car utilizing the AWS DeepRacer hardware and ROS2-Foxy. The CSCI 5302 course provided a comprehensive foundation, enabling the seamless integration of theoretical concepts into practical applications. Our success in testing and executing all challenges attests to the effectiveness of the coursework in facilitating the translation

of acquired knowledge into real-world autonomous driving scenarios.

APPENDIX I
DEEP LEARNING FOR AUTONOMOUS CONTROL

Autonomous Vehicle Control incorporates the use of sensor readings to generate actuator commands for running vehicles autonomously. Not to mention, the number of algorithms in this domain has been increasing at an alarming rate. Ever since the grand autonomous vehicle challenge held by the Defense Advanced Research Projects Agency (DARPA) in 2004, more and more people have delved into the topic, and very interesting research has been emerging ever since.

In the past few years, however, deep learning approaches have taken a front seat, and the same has been tried and tested in use cases pertaining to autonomous vehicle control systems. This document emphasizes and elucidates the deep learning techniques that have been used in this field, their key aspects, research gaps, and much more.

A. Key Aspects

Some of the key aspects to consider in the scope of deep learning for autonomous vehicle control is that most of the work has a selective set of implementation approaches. They include: Feed-forward neural networks, reinforcement learning and supervised learning networks. The following section elaborates about the related work in this domain and the kind of algorithms used for different use cases.

B. Literature Review

The following section gives brief descriptions of all articles considered for this literature review. Their working, usage, advantages, and downsides are elucidated.

1) ***Study 1****:* Zhao et al. [1] implemented an image-based end-to-end deep reinforcement learning framework designed to assist Unmanned Aerial Vehicles (UAVs) during landing. Within their framework, two deep neural networks were utilized to model the actuator commands and the Q-value for the UAV. Consequently, they applied two deep reinforcement learning models for policy and value networks to calculate rewards. At each timestep t, the target was computed using the target value network, and the behavior value and policy network were updated accordingly. Subsequently, the target network underwent further updates, and the timestep was incremented until the termination condition was met.

The authors explored three use cases involving the fixed-wing landing of UAVs and conducted tests using their framework. In all three cases, the UAV successfully landed on the test runway without crashing. However, it is worth noting that the framework exhibits limitations in generalizing to noisy environments. Furthermore, the testing encompassed a minimal number of use cases. Additionally, the framework may face challenges if the state of the UAV changes mid-flight and the actuator commands remain unchanged, potentially leading to landing failures. These scenarios need careful consideration and addressing for comprehensive applicability.

2) ***Study 2****:* Muller et al. [2] employed a 6-layer Convolutional Neural Network (CNN) to develop an end-to-end framework for collision avoidance in an off-road unmanned robot vehicle. The CNN framework processes image inputs with dimensions of 149×58. As a result, the neural network produces two outputs: the first corresponds to the left steering command, and the second corresponds to the right steering command.

Notably, their data collection process incorporated significant diversity to ensure generalization across all terrains, given that their robot was designed for off-road use. However, the complexity of the convolutional neural network was crucial to handling diverse cases effectively. Models dealing with images exhibiting substantial diversity require increased complexity to identify patterns and achieve optimal performance. On a related note, the primary drawback of their work likely lies in the computational limitations imposed by their robot as well as the large error rates their model posed.

3) ***Study 3****:* Wulfmeier et al., [3] implemented a Deep Inverse Reinforcement Learning (DeepIRL) algorithm that uses Lidar point clouds mapped into a grid as input. The output of their implementation is the set of discrete motions based on the rewards computed. Moreover, their work is very robust to noisy environments. The performance of the IRL was tested with various other algorithms such as maximum entropy reinforcement learning, GPIRL etc,. And their expected value differences are compared and plotted. While DeepIRL performed better than Max-Ent, it could not match up to GPIRL in terms of learning complex non-linear cost functions. Besides that, the computational complexity of the algorithm cannot be ignored when it comes to real-time testing and the computational limitations of robots. Therefore, it is crucial to acknowledge these conditions and address these concerns in the future.

4) ***Study 4****:* Paxton et al., [4], combined Monte Carlo Tree Search (MTCS) with deep neural networks to estimate steering angle, rate, and acceleration by incorporating the following features:

- Features of neighboring vehicles.
- Vehicle states.
- Vehicle position.
- Vehicle priority (at intersections).

They used reinforcement learning to identify the neural network parameters for the control and task policies. Since the inputs are numerous, they may not always be available at all points in time. This could lead to the system behaving unexpectedly or not functioning at all in some situations. Furthermore, the algorithm they designed does not guarantee

collision avoidance. By focusing on features consistently available to these algorithms, some of these minor issues can be mitigated.

5) ***Study 5****:* In this study, Hecker et al. [5] developed an end-to-end framework that outputs the steering wheel angle as well as velocities based on a unique sensor setup. This setup takes data from 8 different cameras that capture a 360-degree view around the autonomous vehicle, along with a route planner that includes a valid route from a given location to a dedicated destination. They used a novel deep learning model to train on the data collected from their sensor setup and tested their results in a simulation.

While this work proves to be very useful, the lack of real-time testing with physical robots makes it less facilitative. One way this can be achieved is by integrating this model with a physical robot and testing it in general situations to check how well the model generalizes and generates output with respect to the environment.

6) ***Study 6****:* Rausch et al., [6] created a deep learning neural network algorithm that takes in image input and gives the steering commands for autonomous driving. The neural network takes pixel by pixel in the input image and maps it to the steering wheel angle.

C. Research Gaps and Trends

Visual-based approaches for autonomous control can also be very risky, as malicious input can be given to the deep learning models, thus yielding unexpected results. These considerations must be taken into account when designing such architectures. The observed trend is that, although most people use convolutional neural networks in their implementations, reinforcement learning approaches are being considered lately.

Similar modeling techniques are used to generate various outputs, such as:

- Steering Angle
- Acceleration
- Yaw rate
- Rewards (for reinforcement learning problems), and many more.

Similarly, different kinds of inputs were also considered. Some of them are listed below:

- Vehicle velocity
- Position in lanes
- Camera input
- 360-degree camera input
- Acceleration
- Lidar sensor data
- Relative distance, etc.

Different inputs can be combined, and various models can be trained on them to generate different kinds of outputs. With the coming years, a lot more interesting research will surely emerge in this domain.

D. Conclusion

In conclusion, while traditional methods like motion planning have been prevalent in the field of autonomous vehicles, they are not the sole focus in recent years. With the emergence and increased adoption of deep learning techniques, autonomous vehicles have also entered this domain. Tesla's level 2 autonomous driving cars on the roads today, for instance, leverage these deep learning techniques for their products. Nevertheless, it is crucial to note that deep learning is a powerful technique, capable of effective autonomous vehicle control when provided with the right data.

E. Accommodation with our project

When it comes to the CSCI 5302 Final Project, various deep learning techniques can be employed to enable autonomous driving for our robot. For instance, we can analyze the hallway path and utilize deep learning neural network algorithms to distinguish between turns and straight paths. Additionally, these models can be applied to generate a steering angle for the robot or identify obstacles through object detection and computer vision, enabling the vehicle to come to a complete halt or avoid obstacles. The computational capability of our Deepracer is a crucial factor to consider. It is essential to note that the computational power is the primary limitation for running such powerful algorithms efficiently with good latency and performance. This trade-off must be taken into account when designing robust models for estimating numerous outputs or when used for simple binary/multi-class classification problems.

APPENDIX II
LEARNING FROM DEMONSTRATION

Learning from Demonstration (LfD) are a resource for the end users to teach the robot new tasks without programming and enable them to perform autonomously. The end-user does not have to analytically decompose and manually program a desired task, a robot controller is derived from observations and a novel task is performed.

The LfD problem is divided into two segments: gathering from examples and learning policies from such examples. The advantage of this method is that is highly flexible in performing wide variety of tasks for different world models. It is considered as a subset for Supervised Learning, where the agent is trained with labeled training data and learns the approximation for that function. In our project we are navigating our robot using a controller around the course. The sensor readings from this teaching is being processed using nav2 module to replicate a map model in rviz. This map can then be used for autonomous motion of the robot

around the track.

A. Literature Review

Following are the descriptions for the publications considered for the literature review. It encloses the summaries for these publications with their advantages and disadvantages.

1) ***Study 1****:* The Rana et al., [7] talks about the Learning from Demonstration (LfD) and Motion Planning as resources to enable robots to function in dynamic environments. Motion Planning generate trajectories which are optimal for the pre-defined goals and LfD aims at satisfying the skill-based constraints which are learned from demonstrations. The approach, a posterior distribution of successful trajectories is calculated optimally, and a likelihood is characterized in each environment. The trajectory prior is learned from the demonstration and the resulting interference based planned paradigms are like the skill reproduction.

Hence, the resulting algorithm is called Combined Learning from demonstration and Motion Planning (CLAMP) which performs probabilistic interference to compute a posterior distribution of trajectories which are encouraged to match the demonstrations while remaining feasible to any variation in the scenarios.

The CLAMP algorithm gives us the probabilistic skill model that extracts the spatio-temporal variance and the correlation among the demonstrations in terms of stochastic dynamics which required minimum tuning and a novel approach for the reproduction of trajectory via efficient probabilistic interference which is optimal to learned skill while is it feasible to different scenarios. In the end these methods are tested on three skills, box-opening, drawer opening and object picking.

2) ***Study 2****:* Gu et al., [8] presents demonstration guided motion planning (DGMP) algorithm, which combines the strengths of sample-based motion planning and robots learning from demonstrations to avoid obstacles and learn critical trajectories for effective motion. The task constraints were extracted from specific demonstrations with low variance and the motion planning is done using the multi component rapidly exploring road map (MC-RRM).

MC-RRM uses probabilistic mapping and rapidly exploring random trees for motion planning over time dependent cost maps. The downside for using this method is that is uses high computation time for high dimensional configuration space and the RRT provides a sub-optimal path. This was tested in Aldebaran Nao robot to do two household tasks, namely transferring sugar to a bowl and clean a surface.

3) ***Study 3****:* Rigter et al., [9] talks about using human teleoperation and autonomous controllers using reinforcement learning to minimize the cost of any human teaching. By analyzing the state space for each controller's performance, humans were only involved when they were highly required. The testing was done three episodic tasks where a human controller Ch or an autonomous controller Cn is selected depending on the performance prediction, using Class Constraint Bin Packing. The goal is to reduce the cumulative cost from human demonstration Cd and failure of the robot from the demonstration Cf. The chosen controller then learns from deep deterministic policy gradient to improve the autonomous performances. This method does not compensate for the errors made by the human operator and there is only a single controller is selected for a given task not a sequence of them.

4) ***Study 4****:* Hayes et al., [10] shows the effective planning of robot collaborators that work with humans that can plan around them considering that human training time is limited. The paper shows two demonstration based active learning to accelerate the learning. They used action-space graph which is a dual representation of a Semi-Markov Decision Process graph. A vertex in the graph is labeled with information of the environment and the actions are labeled in the directing edges between the transition of the tasks (Environment-space Graph). A multi-instructor task domain is generated to incorporate all the learning's from a number of robots in the same environment which leverage and benefit from each other's teaching styles. And to gain from these useful features the action-space (SMDP) graph is utilized which provides a skill-centric overview of a task.

These tasks are selected from random queries where the task is selected randomly, distance-based query where the shortest path for the vertex is selected and connectivity query. The experiment is using a construction task provided the robot inquiries about the task. Only a specific set of building blocks were used, and the training was limited for the naturalistic setting to provide a demonstration for a non-dedicated task.

5) ***Study 5****:* Gutierrez et al., [11] this paper limits the errors caused by the humans during the teaching a task. The co-present humans are allowed to rectify any errors caused by the robot while performing a task. The State-Indexed Task Updates (SITU) algorithm is used to corrects the demonstration for a local task by changing only a small subset of the model. The corrective demonstration is a combination of modeled and unmodeled segments, thus updating the unmodeled segments with their associated components.

The task is modeled as a Finite-state automaton with nodes and transitions, which have policy model describing what actions to take and the initiation classifier. Policy execution is done selecting the primitive and from the

structure of the graph the most likely primitive node is selected as the next primitive. The classifier is required to have a classification confidence. Novice teachers can provide more successful tasks by providing keyframe demonstration which reduces the noise and increases consistent results. The performance of the demonstration is related to the consistency of the segmentation across of the task demonstration. The policy takes the form of hidden Markov Model with multivariate Gaussian emissions over the end-effector pose and it is trained on a set of keyframe with hidden state as 'true' instances. The algorithm needs further testing with multiple users and tested on multiple classifiers with varied classification.

REFERENCES

[1] J. Zhao, J. Sun, Z. Cai, L. Wang, and Y. Wang, "End-to-end deep reinforcement learning for image-based uav autonomous control," *Applied Sciences*, vol. 11, no. 18, 2021. [Online]. Available: https://www.mdpi.com/2076-3417/11/18/8419

[2] U. Muller, J. Ben, E. Cosatto, B. Flepp, and Y. Cun, "Off-road obstacle avoidance through end-to-end learning," in *Advances in Neural Information Processing Systems*, Y. Weiss, B. Schölkopf, and J. Platt, Eds., vol. 18. MIT Press, 2005.

[3] M. Wulfmeier, D. Rao, D. Z. Wang, P. Ondruska, and I. Posner, "Large-scale cost function learning for path planning using deep inverse reinforcement learning," *The International Journal of Robotics Research*, vol. 36, no. 10, pp. 1073–1087, 2017. [Online]. Available: https://doi.org/10.1177/0278364917722396

[4] C. Paxton, V. Raman, G. D. Hager, and M. Kobilarov, "Combining neural networks and tree search for task and motion planning in challenging environments," in *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2017, pp. 6059–6066.

[5] S. Hecker, D. Dai, and L. Van Gool, "End-to-end learning of driving models with surround-view cameras and route planners," in *Computer Vision – ECCV 2018*, V. Ferrari, M. Hebert, C. Sminchisescu, and Y. Weiss, Eds. Cham: Springer International Publishing, 2018, pp. 449–468.

[6] 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," in *2017 American Control Conference (ACC)*, 2017, pp. 4914–4919.

[7] M. Rana, M. Mukadam, R. Ahmadzadeh, S. Chernova, and B. Boots, "Towards robust skill generalization: Unifying learning from demonstration and motion planning," 10 2017.

[8] G. Ye and R. Alterovitz, *Demonstration-Guided Motion Planning*. Cham: Springer International Publishing, 2017, pp. 291–307. [Online]. Available: https://doi.org/10.1007/978-3-319-29363-9_17

[9] M. Rigter, B. Lacerda, and N. Hawes, "A framework for learning from demonstration with minimal human effort," 06 2023.

[10] B. Hayes and B. Scassellati, "Discovering task constraints through observation and active learning," 09 2014.

[11] R. Gutierrez, E. Short, S. Niekum, and A. Thomaz, "Towards online learning from corrective demonstrations," 10 2018.