Integrated Vision-Physics-Reinforcement Learning Framework for Dynamic Industrial Robot Navigation

Benyamain Yacoob, **Jingyuan Wang**, **Xinyang Zhang**

Machine Intelligence Enthusiasts

ELEE 5350: Machine Learning

University of Detroit Mercy

March 20th, 2025

Contents

1 Contributions of Team Members

Member	Contributions
Benyamain Yacoob	Led the initial discussions and inquired questioning about overall framework, ranging from CNN, RL, PINN. Proofread analysis from teammates, and cohesively integrated all thoughts into the report.
Jingyuan Wang	Wrote analysis on his experimentation of PINN and the actual experiment of exploring the compatability to the Turtlebot3 within Gazebo to make PINN possible.
Xinyang Zhang	Wrote analysis on his experimentation of CNN and the actual experiment of exploring the compatability to the Turtlebot3 within Gazebo to make CNN possible. Also parsed images for dataset collection.

2 Accomplishments

2.1 Project Summary

This project addresses the critical challenge of mobile robot navigation in dynamic industrial environments, where traditional navigation methods often fail due to unpredictable obstacles and changing conditions. The importance of this research lies in enhancing factory floor automation, warehouse logistics, and hazardous environment exploration, where robots must adapt to moving equipment, workers, and changing layouts without requiring constant reprogramming.

Our integrated framework combines vision, physics, and reinforcement learning to enable dynamic navigation for industrial robots. Utilizing TurtleBot3 refer to Figure 1) within a ROS2 Humble and Gazebo Fortress simulation environment, the framework integrates a Convolutional Neural Network (CNN) for perception, a Physics-Informed Neural Network (PINN) for wheel dynamics, and Proximal Policy Optimization (PPO)-based Reinforcement Learning (RL) for decision-making. The objective is to enable the robot to detect a target object and navigate to it efficiently in a dynamic setting, with this phase focusing on developing and designing these components.

The key limitations of our approach include the sim-to-real gap between Gazebo physics and real-world dynamics, the computational demands of real-time perception and decision-making on resource-constrained robotic platforms, and the challenge of generalizing to unseen environments beyond the training scenarios. Our team is currently exploring options to address the sim-to-real challenge by developing precise wheel dynamics models through ROS2 joint state service calls for torque control.

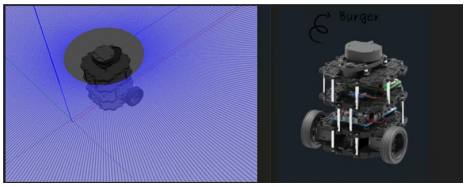

Figure 1: TurtleBot3

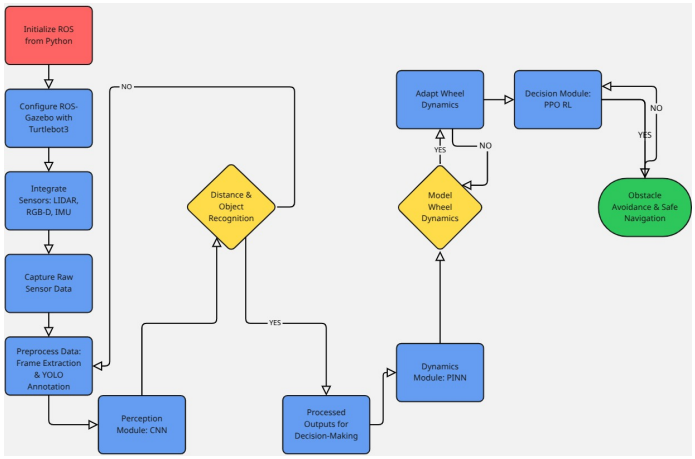

Figure 2: Integrated Vision-Physics-RL Framework Architecture

2.2 Data Processing

2.2.1 Camera Setup

Our data collection plan utilizes the TurtleBot3's RGB-D camera mounted at a fixed height of 0.3m from the ground, with a field of view of 87° horizontally and 58° vertically. The camera is configured to capture images at 640×480 resolution with a 30 fps framerate. To supplement visual data, we are integrating LiDAR sensor data with a ±5° projection overlay on the camera feed, providing depth information critical for obstacle avoidance. This sensor fusion approach will enhance the robot's environmental perception capabilities beyond what either sensor could provide independently.

For our dataset development, we're planning to extract 15 frames per second from the TurtleBot3 camera's 30 fps stream to build a comprehensive training set.

The dataset organization plan includes three main components: raw images (unprocessed frames captured directly from the TurtleBot3 camera during Gazebo simulation), annotated data (YOLO-formatted labels with bounding boxes and class identifiers for ground, obstacles, and target objects), and processed files (Darknet-compatible files for training, including data augmentation variants). All dataset files will be indexed consistently for streamlined access and will be available in our project's GitHub repository upon completion.

2.3 Implemented Methods

2.3.1 CNN Component

We investigated whether we can implement RL inputs based on combining object detection and segmentation implementations. During practice, we encountered problems such as trying to implement the judgment of walls, floors, people, etc. However, the segmentation method is not applicable in the Gazebo world because we tested two combined object detection and segmentation models: YOLO with detectron2 and YOLO with self-segmentation. As shown in

Figures 3, 4, and 5, detectron2 does not work well in Gazebo. Since detectron2 only has segmentation, we tested using YOLO object recognition first, followed by detectron2 segmentation for the object box and peripheral parts. The results were not as good as with YOLO with the best possible segmentation. Both models struggled with wall recognition, and obtaining a dataset is challenging (requiring manually labeled images, which is time-consuming). This hindered the design of logic code for obstacle avoidance judgment. Based on the segmentation capabilities of YOLO, we can recognize existing categories effectively. One idea is to use LiDAR for wall obstacle avoidance and the camera for obstacle avoidance.

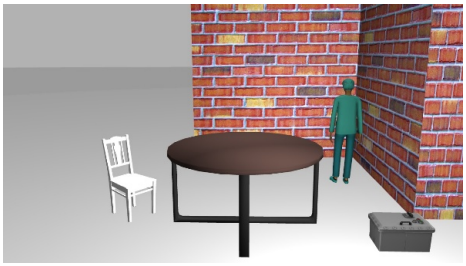

Figure 3: Original image from Gazebo simulation

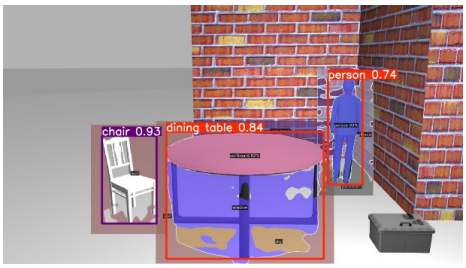

Figure 4: Object detection with Detectron2

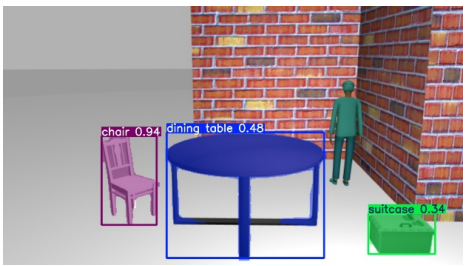

Figure 5: Object detection with YOLO-seg

We designed a modified network structure based on YOLO, characterized by simultaneous prediction of target class, distance, and navigability. Inputs include RGB images and LiDAR data, with key features extracted by the Backbone and multi-scale information integrated by the Neck. You can find the

framework in Figure 6.

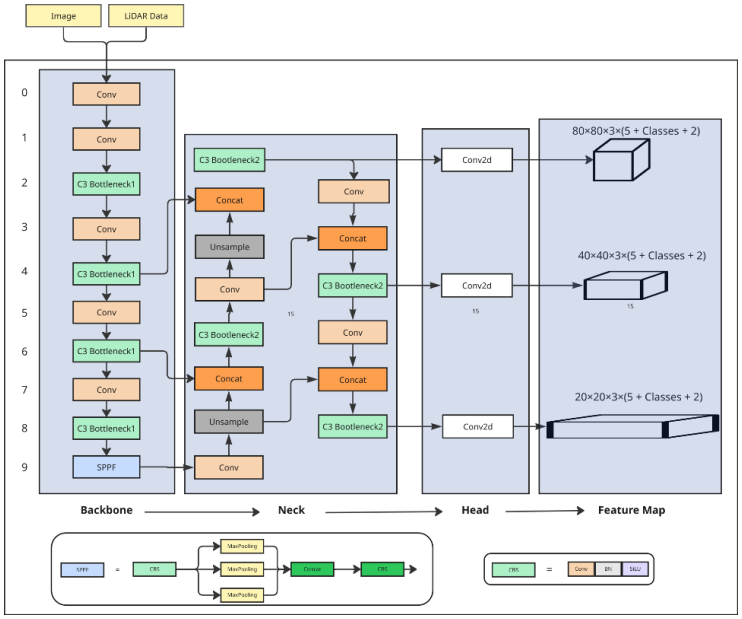

Figure 6: CNN architecture and design

The entire feature network is divided into four main parts:

- **Backbone**: Responsible for extracting features from input images and LiDAR data.
  - Conv: Extracts basic image features such as edges and texture.
  - C3 Bottleneck: Optimizes feature expression through residual connections to improve detection.
  - SPPF: Increases the sensory field to enhance the network's ability to adapt to targets at different scales.
- **Neck**: Performs multi-scale feature fusion to detect targets of different sizes.
  - Upsample: Increases the resolution of the feature map for better detection of small targets.
  - Concat (stitching): Fuses features from different layers to enhance multi-scale information.
  - C3 Bottleneck2 (residual module): Optimizes the fused features again to improve detection.
- **Head**: Detects small, medium, and large targets using 80×80, 40×40, and 20×20 feature maps, predicting x, y, w, h, confidence, class, distance, navigability, etc.

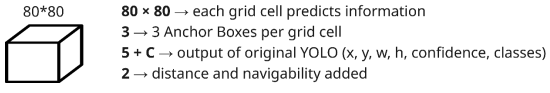

Figure 7: Grid cell instance

Finally, the Head generates 80×80, 40×40, 20×20 feature maps, with each grid cell predicting 3 Anchor Boxes for detecting targets of different sizes (refer to Figure 7). The number of output channels is expanded from (5 + C) in YOLO to (5 + C + 2), adding target distance and navigability. In the post-processing stage, distance (based on monocular camera, LiDAR, depth camera to calculate the optimal distance) and navigability (based on LiDAR to determine path feasibility) are resolved to generate target detection information for the robot. This enables both object recognition and environment navigation information, providing usable input for reinforcement learning (RL). You can see a sample of the detection worflow in Figure 8.

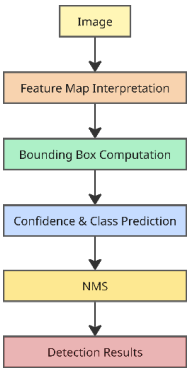

Figure 8: Image detection workflow

2.3.2 PINN Component

Our project aims to use two subcomponents: CNN image recognition (for object detection and obstacle distance measurement) and PINN wheel control (dynamics wheel control). These provide conclusions to a reinforcement learning module, PPO, which makes decisions and gives feedback to the CNN and PINN. The goal is to recognize and find the target (person) in an indoor environment while avoiding obstacles (table, chair, suitcase) during travel. PPO acts as the reinforcement learning decision layer, CNN as the image recognition and distance determination layer providing environment information to PPO, and PINN controls the dynamics layer (applying torque to the wheel).

This week's PINN work focused on basic control in Gazebo, without involving neural network development yet. Since our project requires controlling TurtleBot3's attitude by torque, but Gazebo defaults to controlling via robot linear and angular velocity, we chose to call the ROS2 service `/apply_joint_effort` for torque control. We commented out the velocity masking effect of the `libgazebo_ros_diff_drive.so` plugin in the SDF file for TurtleBot3_burger to make sure the torque effect works properly.

After modifications, we found TurtleBot3 still couldn't move. Upon checking the SDF file, we discovered the default joint μ value was abnormally large ($\mu = 10000$), which we corrected to $\mu = 0.5$. Sending the `/apply_joint_effort` service call via terminal, TurtleBot3 exhibited torque action but flipped and flew out due to excessive torque. Although uncontrollable, this verified the feasibility of torque control.

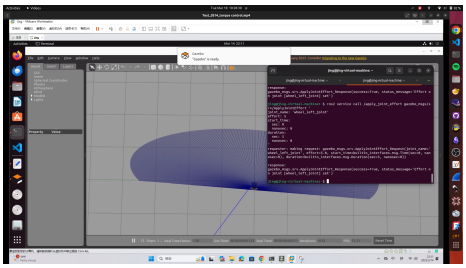

Figure 9: TurtleBot3 torque wheel dynamics control 1

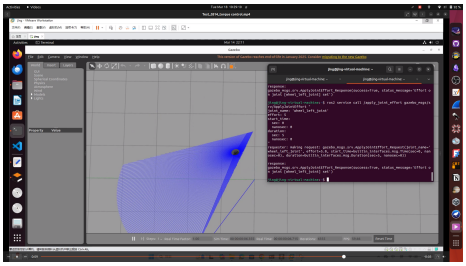

Figure 10: TurtleBot3 torque wheel dynamics control 2

To maintain a normal attitude, we increased the friction coefficient (μ from 0.5 to 1) and decreased the torque (from 20 to 1). Through experiments, TurtleBot3's behavior changed to flipping back and forth before flying out, rather than immediately flying out. We also found that the torque command only took effect when Gazebo was just started, and after a while, Gazebo stopped responding to torque service requests. After checking, we realized the `starting_time` of the torque service corresponds to Gazebo's simulation time.

Having solved the time problem, we needed to make TurtleBot3 controllable. By monitoring the `/joint_states` topic, we realized the torque invocation service doesn't zero out after the duration, which caused improper motion. We manually reset the torque to 0 after the action ended, greatly improving TurtleBot3's performance with no more flying out (refer to Figures 9, 10, 11).

At this point, to realize TurtleBot3 turning or rotating in place, we needed to apply different torques to the two wheels, possibly with different times for complex turns. We transferred the torque parameters from terminal to Python for control, using `MultiThreadedExecutor` for parallel two-threaded control of both wheels and asynchronous torque zeroing.

The result is that TurtleBot3 can run controllably but cannot overcome inertia to stop immediately after the torque effect ends (since applying 0 torque doesn't limit speed). This phase of PINN work focused on modifying Gazebo to meet PINN's basic requirements. You can find our efforts visualized in Figure 12.

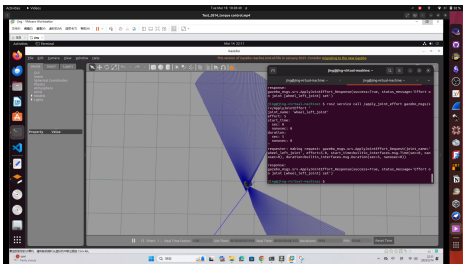

Figure 11: TurtleBot3 torque wheel dynamics control 3

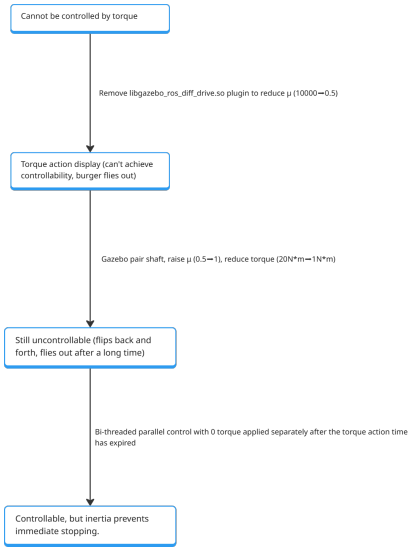

Figure 12: Gazebo PINN compatibility

3 Challenges Faced and Plans/Goals

The team is in the theoretical stage of addressing the sim-to-real challenge for the TurtleBot3, particularly focusing on torque control of the two-wheel dynamics to better model real-world conditions.

Beginning the implementation of the physics-constrained autoencoder for wheel dynamics modeling and our custom PPO will move our PINN and RL components from theoretical to practical application.

These efforts will prepare us to move from the theoretical framework to practical implementation in the next phase, bringing us closer to a functional prototype for real-world testing.