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 30th, 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. Also worked on image segmentation compatibility.
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 integrative framework can be found in Figure 2.

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.

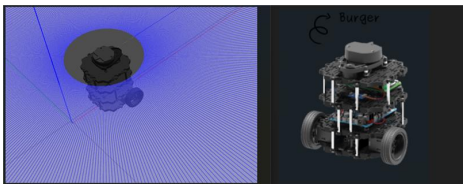

Figure 1: TurtleBot3

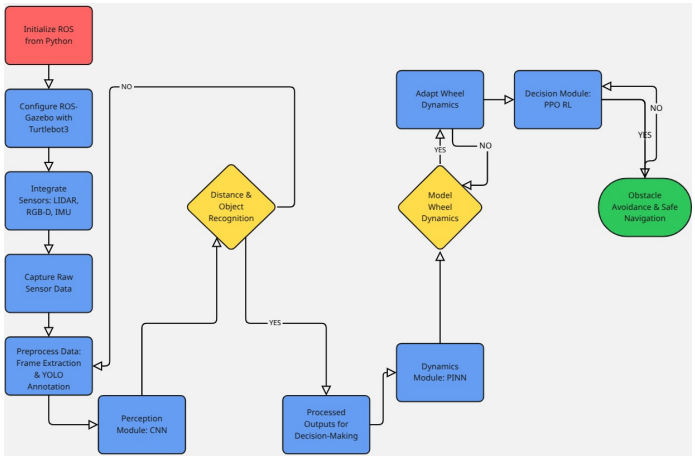

Figure 2: Integrated Vision-Physics-RL Framework Architecture

2.2 Implemented Methods

2.2.1 CNN Component

First of all, the last time we tested the effect of the modified CNN model on the basis of YOLO, the accuracy is not high. Our idea is to use the images obtained by the RGB camera, after the model processing not only to be able to identify the target box and confidence, but also to be able to judge the distance. For the customized YOLO format labeled data, a sixth column is added to the original category and bounding box coordinates, which is used to describe the actual distance between each target and the camera.

Eventually this distance value can help the robot to have good feedback based on the PPO model, for example:

- If the robot recognizes through the RGB camera that the distance ahead to the obstacle is less than a threshold value, then a negative function reward value is given.
- If the robot recognizes through the RGB camera that a goal (e.g., a person) is in front of it to be reached, and it keeps approaching the goal point during training, a positive function reward is given.

But how to get the actual distance value? We thought about three methods:

- Distance measurement by monocular camera algorithm
- We get the distance from the different depth values of each target point in the depth picture.

• We use radar to get the distance to the obstacle in front of us.

Each of these three methods has advantages and disadvantages. For example, if the distance of the recognized obstacles is obtained by radar, it is difficult to determine which points need to be clustered into a category. In the following radar picture (Figure 3), the range of the camera is [-40 degrees, +40 degrees] (between the two straight lines in the picture is the recognition range of the camera).

The radar can obtain all the surrounding obstacle points, but it cannot know which points are clustered into humans and which points are clustered into chairs, which will have a negative impact on our labeling of obstacles.

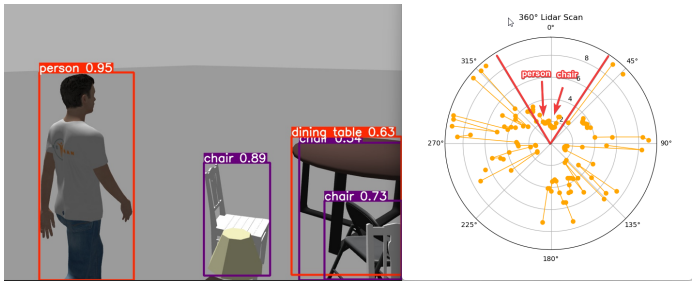

Figure 3: Recognition and radar maps for the same time frame

The distance is obtained by a monocular camera algorithm, referring to the principle of small hole imaging (camera imaging principle), as shown in the figure (Figure 4), and we want to get the actual distance. But, there are some parameters that we must know, such as the camera's internal reference (there are toolkits available in MATLAB to calculate the camera's internal reference), but we also need to know the height of the car in the real world, and this algorithm is suitable for stationary cameras, such as surveillance cameras.

We are installing the camera on a moving robot, and it is not suitable for using this algorithm.

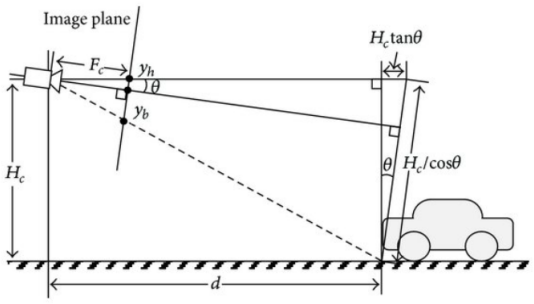

Figure 4: Schematic diagram of the monocular camera ranging algorithm

Eventually, we would like to recognize normal pictures (RGB pictures) based on the model, get the target frame, combine it with the depth picture (which has the depth of the object in the photo stored), estimate the target distance, and combine the labels in YOLO format to form a new dataset of labels.

However, since only one depth is stored, we will also encounter problems at the algorithmic level in measuring the depth of the objects in the target frame. Our first idea was to directly calculate the mean depth/median depth of the pixels in the target frame to represent the distance from this obstacle to the camera through the existing target frame, and encountered problems in implementing it.

For example, the target box labeled 4.1m distance in the picture (Figure 5) is composed of the median depth of pixels of the white chair, the lamp in front, and the ground in the distance, not the median depth of pixels of the white chair, and the error is especially obvious in some other pictures.

The segmentation algorithm is used to obtain the rough outline of the chair in the target frame alone, and then the median depth value of the pixel points in the area within this outline is taken as the distance from the recognized chair to the camera.

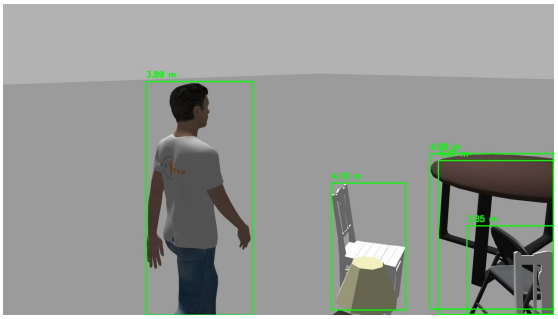

Figure 5: Direct depth image distance estimation

In terms of model structure, by inheriting Faster R-CNN model (Region-based Convolutional Neural Network, which excels at object detection tasks by combining region proposals with feature extraction), we extend the data enhancement and add a new branch of distance regression based on RoI (Region of Interest) features (distance head), which performs linear regression on the features of each target and outputs continuous distance values.

In the training stage, smooth L1 loss is introduced as the distance regression loss, which participates in back propagation together with the original classification loss and edge regression loss; in the inference stage, the model outputs the category, confidence and distance information corresponding to each detection frame. The evaluation function extends the metric of distance prediction performance, and adds the calculation of Average Distance Error.

We use the region growth segmentation algorithm, by taking the depth picture, set different thresholds for different categories. The red area is the contour range obtained by the segmentation algorithm. The left picture (Figure 6) is the contour obtained by setting the same threshold, and the right picture (Figure 7) is the contour range obtained by adjusting the threshold, and then take the median of the depth of the pixels in the contour.

That way, the distance value is obtained with good effect.

Segmentation for depth cameras The ultimate goal of this section is to provide scene distance information.

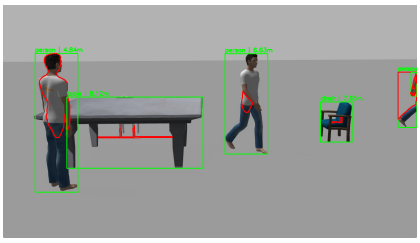

Figure 6: Region growing algorithms for classes using uniform threshold

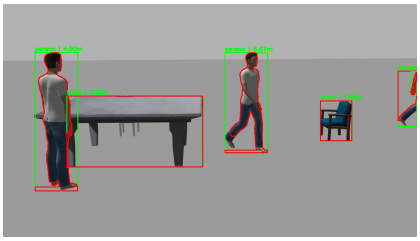

Figure 7: Region growing algorithms for classes using categorical threshold

1. **Vertical declination calculation**

   In order to get better results for object recognition, our RGB and depth cameras have a pitch angle of 0.2 rad (as illustrated in Figure 8), which is used to avoid not being able to see the top surface of higher objects (for CNN training) caused by too low viewing angle.

   So if we want to characterize the actual position of the object using the data from the depth camera, we need to overcome the vertical distortion. Translate all depth quantities to the horizontal z-axis. (The origin of the coordinate system is the upper left corner of the depth image, at which point the z-axis is the depth data. The horizontal z-axis assumes that the center optical axis is horizontal.)

   In actual calculations, we actually convert in rows of pixels.

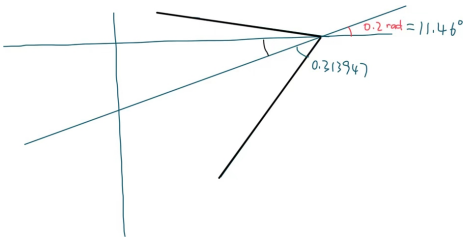

Figure 8: Camera pitch angle illustration

2. **8-neighborhood region growth based on mutation thresholding**

   We consider two neighboring pixels to be the same object when they are less than the mutation threshold. All labeled pixels are given after traversing all pixels.

Then create multiple regions and initialize region seeds based on the marked pixels, traverse all neighboring pixels in the current region by breadth-first search (BFS), and use 8-neighborhood to detect all pixels that meet the threshold.

3. **Background Filtering**

   Based on the results of the previous step, we found that the largest regions in the results are earth and sky. This is not the distance we expect to recognize, so filtering the background is necessary. We actually tried a number of approaches, which we will detail.

   We used the geometric method at the beginning to exclude the ground by calculating the corresponding value of the ground in each pixel (in this case the unit is each pixel point, so we not only calculate the vertical declination but also give the horizontal declination). The problem was that the difference between the theoretical and actual values made it difficult to achieve the desired result even with the addition of a tolerance.

   Then we tried the 3D point cloud method with a single frame depth image, and we have to say that the 3D point cloud achieved very good results. However, considering that our final project requires real-time computation, the 3D point cloud will undoubtedly greatly increase the computational demand, which indicates that this is not a strategy that meets the conditions of our hardware.

   On the basis of the above, we considered that in depth images, the maximum value of each row of pixels is always the ground or the sky, and the actual detected object is always before the background. Based on this idea, we came up with the idea of using a difference algorithm for the maximum value.

   This difference algorithm not only requires very little computation, but also works very well: first, the maximum value of each row of depth pixels (the unit here is each row of pixels) is extracted to form an array. Since the depth of the ground background varies uniformly, in order to prevent a row of pixels from having no background at all, we form a difference array of the neighboring differences of the array of maximum values.

   Once the difference exceeds the threshold value, the corresponding value in the original maxima array is replaced by 0. This results in a very good representation of the processed segmentation depth image. Although a V-shaped splinter appears in all images, it is almost negligible after the mutation threshold is increased to 0.3.

4. **Horizontal projection**

   After the segmentation of the depth images is complete, we want to project these regions onto the ground to provide scene distance information to the decision maker. So first of all the depth image system is transformed to the robot system using 3D coordinate transformation. In this step: the depth system with the origin in the upper left corner of the depth image is transformed to the robot system on the ground. The computational procedure is not repeated.

During the transformation we found that the xy-axis of the original depth system is in pixels, only the z-axis is in meters. But the robot coordinate system is in meters. In order to correct this problem we introduced the pinhole camera model and solved this unit problem by consulting the optical center coordinates and the horizontal and vertical focal lengths.

Finally, in this projection form we tested two models: ray projection and parallel projection. After testing, it was found that the ray projection does not magnify the object equivalently to improve accuracy, but rather distorts the original object. So parallel projection was chosen for the final solution.

The xy-plane projection of each pixel of the depth image in the robot coordinate frame is faithfully reflected. In the final step, we added outer rectangles to each regions to facilitate the later processing.

The following image is the original image visualization of the depth image in npy format (Figure 9):

Figure 9: Original image visualization of depth image

The set of images shown below (Figure 10) are the original RGB image, segmentation of the depth image, horizontal projection, horizontal projection of the outer rectangle.

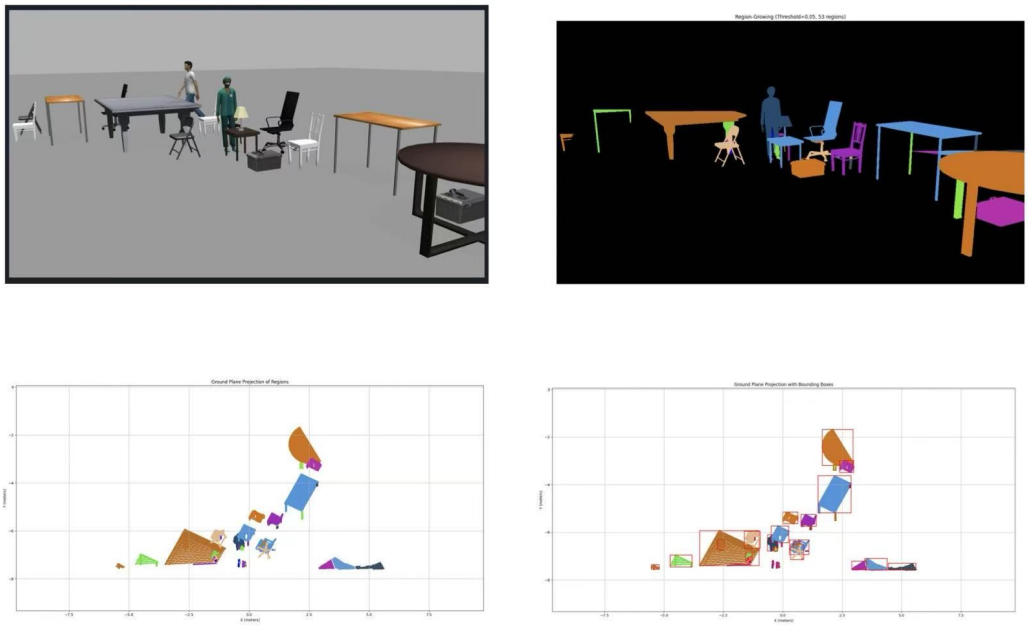

Figure 10: Original image, segmented image, horizontal projection, and rectangle image

Finally, we got the distance value of each target box and composed the dataset to train based on modified Faster RCNN model. There are 10077 images and labels in the dataset, 8:2 ratio to divide the training set and validation, the batch size in the training is 4, i.e., $8064/4 = 2016$, epochs $= 50$, and each epoch takes around 18 minutes to finish.

Table 1 shows the detailed architecture of our Faster R-CNN implementation, illustrating how the input image is processed through various layers and how the feature map dimensions change throughout the network.

You can find the framework represented in image form in Figure 11.

Stages	**Roles**	**Feature map changes (input 3×720×1280)**
Conv1+bn+relu	Initial convolution + activation	[64, 360, 640]
Maxpool	Downsampling	[64, 180, 320]
Layer1	Residual module × 3	[256, 180, 320]
Layer2	Residual module × 4 + down-sampling	[512, 90, 160]
Layer3	Residual module × 23 + down-sampling	[1024, 45, 80]
Layer4	Residual module × 3 + down-sampling	[2048, 23, 40]
Avgpool	Pooling to 1x1	[2048, 1, 1]
FC	Classification output	[1000]

Table 1: Faster R-CNN tabular feature map representation and its respective roles to each layer

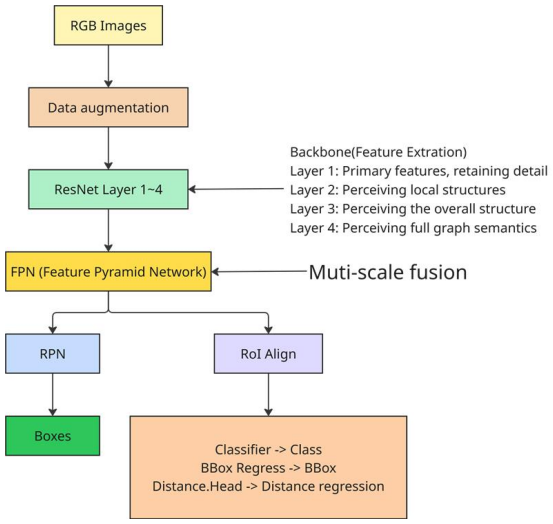

Figure 11: Faster R-CNN framework

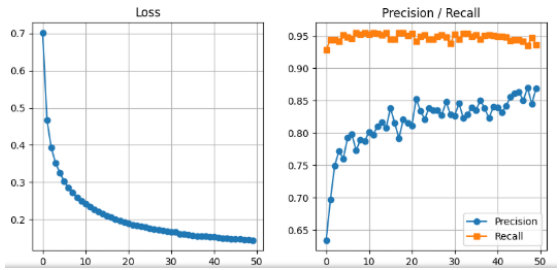

Figure 12: Model Evaluation with Low Confidence Thresholds

A high recall indicates that our model "almost detects all target frames", but a low precision indicates that there are many false positives; therefore, our model tends to favor the strategy of "better to over-report than to under-report", which is similar to our goal, as shown in Figure 12.

Therefore, our model is biased towards the detection strategy of "better to over-recognize than to under-recognize", which is similar to our goal, it is better to recognize the obstacles than not, but our low score_threshold setting (e.g., 0.3 0.4) results in a lot of low-confidence frames being recognized and retained.

We also tested different learning rate optimizers (SGD, Adam) to see how well the model converges, and found that SGD takes longer, but Adam tends to converge quickly into a local optimum.

2.2.2 PINN Component

PINN Basics: Inertial Processing Based on last week's unfinished inertia handling in the PINN Basics configuration, we tackled inertia in between other work. Since the /apply_joint_effort service doesn't directly control the overall speed zeroing, this gives us two new ideas: a) Force zeroing directly using the speedtopic. b) Zeroing the velocity by reversing the torque.

It is very obvious that we prefer the second option, since forcing the use of a solution other than physical is not the original intention of our PINN. On further probing, we realized that since we had commented out a speed control plugin in the previous one that was causing the torque masking effect, we needed to add an additional plugin if we wanted to invoke the speed control service.

So based on the above, we finally chose to reverse the torque to complete the closed-loop speed control.

1. **Speed monitoring and variable torque control**

   Based on the result of the previous process, we need to add a reverse torque before the torque goes to zero, and the impulse here needs to be strictly controlled to ensure that the cart can stop exactly at speed zero. We first introduce speed monitoring: the speeds of the left and right wheels are monitored in real time by subscribing to the topic /joint_states.

   But this doesn't work very well, because the moment is a constant value, the acceleration remains a constant at the end of the time of action, and even if the moment goes to zero, it doesn't stop immediately. For this reason, we introduced variable moment control.

   In practice, in Gazebo, the time-based variable moment action is not as smooth as in the hard world, but the frequency of application is determined by the refresh rate of the simulation software. And in testing it was found that the ROS service would cause port blocking if called too often.

   So we used a discrete service refresh, using a rate slightly below the Gazebo clock frequency to give the cart gradually decreasing acceleration/angular acceleration in time steps. This gave by far the best control in real-world performance.

2. **Time Synchronization and Floating Point Compatibility**

   In the previous phase of the task, we were actually using the fixed-time action command. In this phase, we realized real-time issuing of commands and real-time acting of torque through subscription/clock topic.

However, there are some variables (especially time variables) that can only be used with int-type data, so we optimized for this to make floating point numbers acceptable.

3. **Multi-thread processing**

   Because of the thread blocking situation during the test, we have handled multi-threading. First of all, the fundamental reason we chose multithreading is that when dealing with Turtlebot3_burger having angular velocity, it requires two wheels to act with different moments, or even different action times, start times, and termination times, while having completely different reverse moments acting at the same time, as shown in the flowchart (Figure 13).

   Also using a separate thread in the time listening is effective in avoiding the main thread getting stuck. And even asynchronous invocation of services can be synchronized and waited for in the future after the decision maker imposes a command.

   Meanwhile, based on the problems we encountered in the test, the safe exit of the control thread can effectively avoid thread blocking.

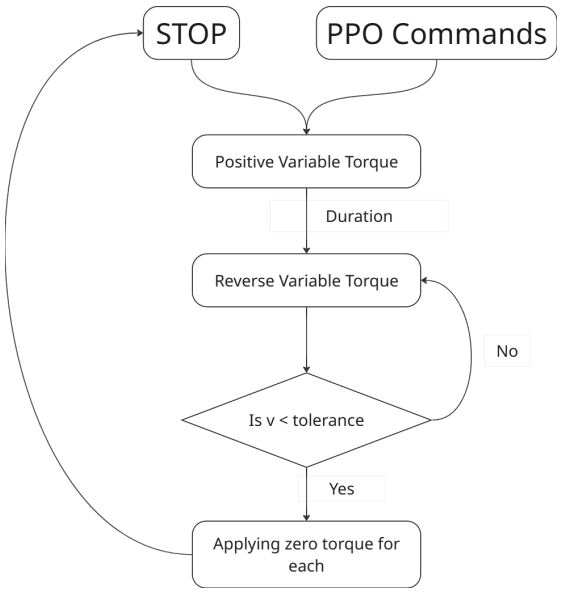

Figure 13: PINN torque control flowchart

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.