Bharat Forge - Synchronised Wayfinding and Autonomy for Robot Multiagent Systems

Inter IIT Tech Meet 13.0

Final Submission Report

Author:

Team 67

Contents

1 Abstract

This project presents a centralized intelligence system for autonomous swarm localization and mapping, built on the ROS framework. Using Reinforcement Learning, the system optimizes swarm coordination, enabling efficient exploration and mapping in dynamic environments. The centralized architecture aggregates data from multiple robots for global decision-making while ensuring scalability. Simulation and testing demonstrate significant improvements in localization accuracy and mapping efficiency, highlighting the potential of RL-driven centralized control for multi-robot systems in real-world applications.

2 Problem Understanding

- **Scalable Swarm System for Autonomous Mapping and Navigation:** Develop a solution that maps and navigates a given environment efficiently and quickly, regardless of the size and characteristics of the given environment or the size of the swarm.
- **GPS independent platform:** Develop a solution that does not depend on GPS or any markers that require human intervention.
- **Dynamic Mapping:** Develop a solution to dynamically map and track changes in the environment in real-time.
- **LLM Interface:** Develop an intuitive chat-based user interface for easy use.

Each solution stage aims to create a robust and scalable system to achieve all the listed aims above.

3 Approach

In this section, we outline the approach adopted for developing a centralized intelligence system to enhance autonomous swarm localization and mapping. We begin by describing the map used to train multiple robots within a unified system. This is then followed by an overview of the LLM-based interface, and how it interacts with the RL logic to assign tasks. We then explore how Reinforcement Learning (RL) is employed to optimize swarm behavior, enabling robots to autonomously explore and map dynamic environments and track changes observed, while maintaining scalability and robustness in various real-world scenarios.

Figure 1 demonstrates a high-level overview of the logic our solution works on.

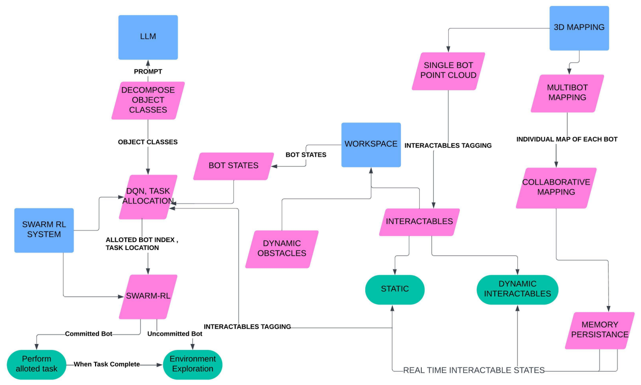


Fig. 1: Logic Diagram

3.1 Exploring the Worlds

We have considered 2 worlds as outlined below, namely a Storage warehouse and an Office

1. Storage Warehouse:

1.1 Environment Overview:

- **Type:** Closed Indoor Space
- **Dimensions:** 35m (Length) × 8m (Width) × 7m (Height).
- **Emergency Exit:** 1 (for evacuation in case of danger).

1.2 Environment Components:

1.2.1. Storage Space (6-storey Structure):

- **Purpose:** Used for storing manufactured products, with shelves and racks. The lowest level includes first aid kits for workers.
- **Impact on Navigation:** Multi-storey structure requires agents to navigate vertically and avoid cluttered areas.

1.2.2. Lathe Machines (3 units):

- **Dimensions per Machine:** 300cm × 80cm × 100cm.
- **Purpose:** Part of the production line.
- **Impact on Navigation:** Machines take up floor space and create potential hazards. Agents must avoid moving parts and human workers.

1.2.3. Chairs (4 units):

- **Dimensions:** Standard-sized chairs (50cm × 50cm × 100cm).
- **Purpose:** For workers' rest and supervisory tasks.

- **Impact on Navigation:** Small obstacles that may block paths in tight spaces.

1.3. Key Considerations for Autonomous Swarm Navigation and RL Training:

- **Space:** The environment offers a confined area, ideal for training swarm bots to navigate in a structured space with multi-vertical levels and emergency exit, which adds complexity to evacuation scenarios.
- **Obstacles:** Lathe machines and storage racks are static obstacles while moving machines and human workers introduce dynamic elements. RL agents need to avoid these while navigating efficiently.
- **Swarm Coordination:** A centralized control system optimizes swarm movement, manages tasks, and directs agents toward emergency exits. The RL model is trained for path-finding, collaborative behavior, and emergency response.
- **RL Training Complexity:** The environment's mix of static and dynamic obstacles, as well as multi-level navigation, offers a challenging setting for training RL agents to handle real-world coordination and safety protocols.

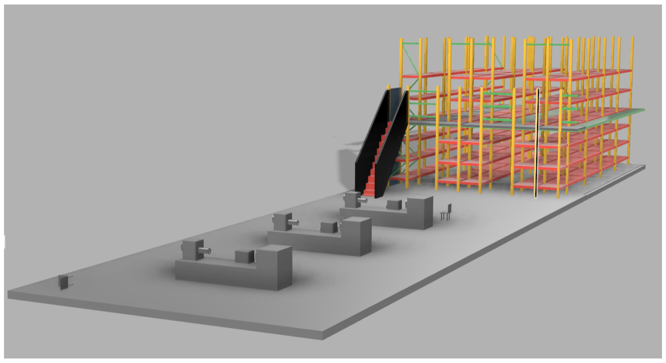


Fig. 2: Storage Warehouse Environment

2. Office Space:

2.1 Environment Overview:

- **Type:** Closed indoor space
- **Dimensions:** 59m (Length) × 38m (Width) × 5m (Height), totaling 11,210 cubic meters.
- **Multiple Exits:** Several exits for evacuation in emergency situation.

2.2 Environment Components and their impact:

- **Tables and Chairs:** Create static obstacles, requiring agents to navigate around them.
- **Cubicles:** Partitioned spaces that divide the office, creating narrow pathways and complex navigation.
- **Flower Pots:** Small static obstacles that can block narrow paths.
- **Fire Extinguishers:** Must be avoided to ensure safety and prevent collisions. **Humans:** Dynamic obstacles that require agents to navigate around or cooperate with them.

2.3 Key Considerations for Autonomous Swarm Navigation and RL Training:

- **Space:** The large but confined office space with multiple exits is suitable for swarm training in navigation and emergency response.
- **Obstacles:** Static obstacles (tables, chairs, cubicles) and dynamic ones (humans) present challenges for RL models in path-finding and obstacle avoidance.
- **Swarm Coordination:** Centralized control can help agents navigate efficiently, avoid collisions, and coordinate tasks like delivering items or evacuating.
- **RL Complexity:** Agents must learn to navigate complex layouts, avoid obstacles, and interact with humans while optimizing task execution.

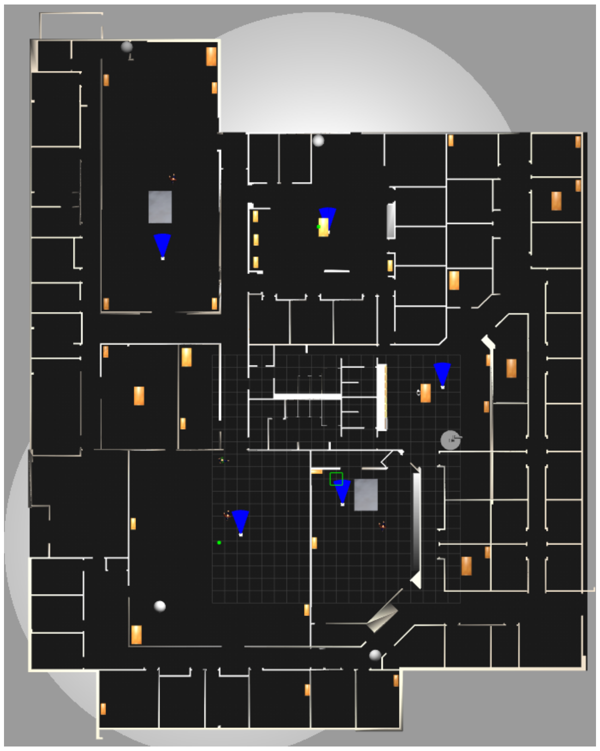


Fig. 3: Office space environment

Why these Environments are suitable for RL and Swarm Navigation:

These warehouse setups offer practical challenges like spatial constraints, dynamic obstacles, and the need for agent coordination. This is ideal for training RL models to avoid obstacles and navigate effectively, collaborate in product handling and emergency situations, and develop sophisticated swarm intelligence for real-world applications like search and rescue, or industrial automation. By simulating centralized swarm control, these environments allow us to test and optimize multi-robot coordination in a complex and dynamic setting.

3.2 User Interface

Overview: The app offers a modern, intuitive interface built with React for seamless navigation, dynamic form submissions, and real-time feedback.

Page Breakdown:

- **Home Page:** The main landing page uses glassmorphism effects. Users submit commands through an input form, and results are displayed instantly without page reloads.
- **Solution Page:** A clean, consistent page explaining the app's features and functionalities.

Tech Stack:

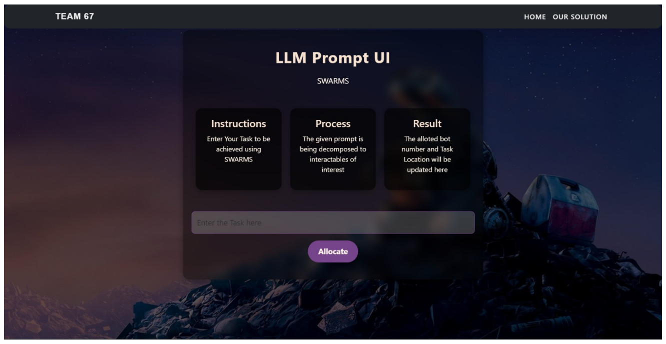
- **React.js:** For building dynamic UIs with reusable components.
- **CSS:** Custom styling using modern design patterns like glassmorphism.
- **React Router:** For client-side routing, avoiding page reloads.
- **Bootstrap:** Ensures consistent UI components and responsiveness.
- **Axios:** For asynchronous HTTP requests to the backend.

Backend Integration:

The frontend communicates with the backend to process user input and return results dynamically, providing real-time feedback.

3.3 LLM Chatbot

The core functionality is structured to efficiently handle task allocation based on input descriptions. Let's take a closer look at the step-by-step process:



LLM Prompt UI

SWARMS

Instructions

Enter Your Task to be achieved using SWARMS

Process

The given prompt is being decomposed to interactables of interest

Result

The alloted bot number and Task Location will be updated here

Fig. 4: User Interface

3.3.1 Task Description

The model begins by receiving a simple, natural language prompt that describes a task. This input could vary in complexity, but is structured in a way that is familiar and intuitive to human users. Some examples of possible inputs include:

- "Give me a fire extinguisher."
- "Go to the table."
- "Find a wrench and a chair."

3.3.2 Task Decomposition:

The model decomposes the prompt into a more structured and actionable format. It analyzes the language and identifies the core elements of the task. This step is critical because it allows the model to understand the intent behind the input and break it down into distinct "task objects". A **Task Object** represents the core action or object that needs to be addressed. For example:

- In **"Find me a fire extinguisher"**, Task Object $\Longrightarrow$ **"fire extinguisher"**.
- In **"Reach the table"**, Task Object $\Longrightarrow$ **"table"**.
- In **"Find a wrench and a chair"**, Task Objects $\Longrightarrow$ **"wrench"** and **"chair"**.

3.3.3 Searching Object Logs:

The next step after defining task objects is to locate them within a predefined list of object logs. Object logs contain information about various objects and their locations/statuses within a given environment. This list serves as a reference, helping the model pinpoint the exact object needed for the task.

For instance, if the task object is **"fire extinguisher"**, the system will search through the object logs to find the location of the fire extinguisher in the environment. If the task object is **”wrench”**, the model will reference the object logs to find where the wrench is stored or situated.

3.3.4 Position Assignment:

Once the task object and its corresponding location are identified, the model passes this information to the task allocation function, which is responsible for managing and executing tasks by assigning actions to the appropriate agents or systems based on the object’s **position** and **relevance**.

The model ensures that the **task object** is aligned with the correct physical or virtual location, and then delegates the task to the appropriate action handler or agent that will perform the task.

For example:

- For **“go to the table”**, the task allocation function will determine the agent’s current position, calculate the path to the table, and initiate the movement accordingly.
- For **“find a wrench”**, the function will first confirm where the wrench is located in the object logs and direct the agent to retrieve or use the wrench.

3.3.5 Object Logs:

The object logs contain the following information:

1. Bot States:

- **Bot Position:** Tracked in real-time using odometry for accurate navigation.
- **Committed States:** Bots transition from uncommitted (idle) to committed (task-assigned) based on task allocation.

2. Object States:

- **Last Known Position:** Tracks the last known location of objects for accurate task execution, even if they move over time.

3. Task States:

- **Task Object Name:** Unique identifier for each task.
- **Task Object Position:** Location of task objects, used for task allocation.
- **Bot Allotted to Task:** Bots are assigned to tasks based on their position, capabilities, and availability.

This structure enables efficient task allocation and real-time task execution management.

3.3.6 Overall Process Flow:

- **Bot Initialization:** Bots start in an uncommitted state with positions tracked via odometry.
- **Object Detection:** Bots gather and log object positions in the environment.
- **Task Creation:** The user defines a task, and the system locates the task object in the logs.
- **Task Allocation:** The system assigns an appropriate bot to the task based on position and availability, changing its state to ”committed.”
- **Task Execution:** The bot performs the task, updates its position, and logs any changes. After completion, it may return to ”uncommitted” or be assigned a new task.

3.4 RL Logic - Swarm Path Planning

The **SwarmRL** logic implements swarm management using RL, specifically the **TD3** (Twin Delayed Deep Deterministic Policy Gradient) algorithm. Using ROS2 and Gazebo, multiple agents navigate to a goal position while avoiding obstacles and choosing efficient paths in the environment. Whenever a bot is allocated from the task function with a goal position, that bot heads to the newly assigned goal position. The input of this RL is bot states and intractable states (goal position).

3.4.1 Reward Function Dynamics:

The code interacts with the reward function during the evaluation and training phases:

1. Evaluation Phase:

- Reward for each agent is calculated in every step during the evaluation loop in the evaluate function.
- Total reward for all agents in an episode is computed by summing individual rewards.
- A penalty mechanism is applied when certain conditions are met, such as: $total\_reward < -90$.

2. Training Phase:

- The TD3 agent utilizes rewards to compute the Q-value in the Bellman equation: $Q(s,a) = r + \gamma \cdot Q(s',a')$, where r is the immediate reward, and $Q'(s',a')$ is the target Q-value.
- The critic networks are optimized to minimize the difference between predicted $Q(s,a)$ and target $Q(s,a)$.

3.4.2 Multi-Agent Reward Aggregation:

In a multi-agent[1] setup:

- Rewards are computed for each agent independently in `envstep()`, which is part of the `evaluate()` function.
- The aggregated reward across agents is used to assess the overall progress.
- The reward signal is directly tied to actions taken by each agent and their environment interactions.

3.4.3 Considerations in Reward Design:

The following factors are kept in mind:

- **Encourage Goal Achievement:** Reward agents for reducing the distance to the target or achieving the goal efficiently.
- **Penalize Collisions or Unsafe Actions:** A collision penalty is applied when iter_reward < -90. This could be fine-tuned based on the application's safety requirements.
- **Promote Multi-Agent Collaboration:** Incorporate rewards that encourage agents to work together or avoid conflicts (e.g., maintaining a safe distance from one another).
- **Dynamic Reward Adjustments:** Use a dynamic reward system where the magnitude of rewards or penalties adjusts based on the agent's progress over time.

Policy Definition: The bots have to move in random motion while avoiding static and dynamic obstacles, also prevent clustering, and reach the target position when allotted.

3.4.4 SwarmRL Architecture:

The TD3 architecture for training swarm robots is built upon the actor-critic model and features the following key components:

- **Actor Network:**
  - **Function:** Generates policy by mapping state inputs to continuous action outputs.
  - **Structure:** A deep neural network with state inputs (e.g., robot positions, velocities) and action outputs (e.g., movement commands).
  - **Policy Smoothing:** Adds noise to actions during training to improve robustness against over-fitting and promote exploration.
- **Critic Networks**[2]
  - **Function:** Two Q-value approximators evaluate the action-value function for a given state-action pair.
  - **Double-Q Learning:** Utilizes the minimum value of two critics during target updates.
- **Target Networks**
  - **Purpose:** Stabilize learning by providing slowly updated versions of the actor and critic networks.
  - **Delayed Updates:** Updates to the actor and target networks occur less frequently than critic updates, improving stability.
- **Loss Functions**
  - **Critic Loss:** Mean squared error between the predicted Q-values and the TD3 target values.
  - **Actor Loss:** Encourages actions that maximize the Q-value predicted by the critic.
- **Training Workflow :**
  - Gather experience from the simulated swarm environment using ROS 2 and Gazebo.
  - Store the experience in a replay buffer to decorrelate updates and stabilize training.
  - Perform gradient updates to the actor and critic networks based on mini-batches sampled from the buffer.
  - Update the target networks using a soft update rule.
- **Swarm-Specific Considerations:** Multiple robots use shared policies or separate ones, with decentralized training and execution. The system rewards agents for achieving goals, avoiding collisions, and maintaining swarm coordination.

3.5 RL Logic - Task Allocation

The task allocation system uses a Deep Q Network (DQN) to optimize the distribution of tasks among robots. By combining reinforcement learning with deep neural networks, the model enables robots to autonomously select tasks based on factors like position, availability, and task proximity.

The system incorporates reward and penalty functions to encourage quick task completion, minimize travel time, and avoid conflicts. Trained in a Gazebo environment using a pre-trained SwarmRL model, the DQN continually refines its task allocation strategies, improving efficiency and performance over time.

Model Type: We use the Deep Q Network (DQN) for task allocation. The DQN approach combines Q-learning with deep neural networks to enable robots to learn optimal task allocation strategies. It allows the model to handle complex, high-dimensional spaces by approximating the Q-values, which guide robots in choosing actions that maximize rewards. The model iteratively improves by learning from experience, and adjusting its decision-making process as it encounters different task allocation scenarios in the environment.

States:

- **Bot's position:** The position of each robot in the environment is critical for task allocation, as it helps determine the proximity to various tasks. This enables the system to prioritize bots closest to tasks, optimizing efficiency and reducing travel time.
- **Task's position:** Each task's location within the environment is also tracked, influencing which robot is allocated to it. The system uses this information to assign tasks to the most suitable robots based on proximity and availability.

Bot's status: represented by t_values, where:

- $t\_values[i] < 0$: Bot i is performing a task.
- $t\_values[i] = 0$: Bot i has completed a task.
- $t\_values[i] > 0$: Bot i is free.

Penalty Functions:

- **Task Duration Penalty:** This penalty discourages long task durations by increasing the penalty as a bot spends more time on a task. The longer a robot is active on a task, the higher the penalty, encouraging quick task completion. (t_values[i] $<$ 0):

$$\text{task\_duration\_penalty} = \begin{cases} -\log(1 + \text{mean}([-t \text{ for } t \in t\_values \text{ if } t < 0])) & \text{if any } t < 0, \\ 0 & \text{otherwise.} \end{cases}$$

Fig. 5: Task Duration Penalty

- **Completion Reward:** A reward of 10 is given bots that transition from doing a task (t_values[i] $<$ 0) to completing it (t_values[i] $=$ 0). This incentivizes robots to finish tasks and ensures that the allocation system values task completion.
- **Distance Penalty:** The penalty is based on the distance between a robot and the task. Greater distances incur higher penalties, encouraging task allocation to nearby robots to minimize travel and improve efficiency.

$$\text{completion\_reward} = \sum_{i=1}^{\text{NUM\_BOTS}} \begin{cases} 10 & \text{if previous\_t\_values}[i] < 0 \text{ and } t\_values[i] = 0, \\ 0 & \text{otherwise.} \end{cases}$$

Fig. 6: Completion Reward Function

- **Double Allocation Penalty:** A penalty is applied if a robot that is free (t_values[i] $>$ 0) is also found to be performing a task (t_values[i] $<$ 0). This prevents conflicting task assignments and ensures that each robot handles only one task at a time.

$$\text{double\_allocation\_penalty} = \sum_{i=1}^{\text{NUM\_BOTS}} \begin{cases} 10 & \text{if } t\ \ values[i] < 0, \\ 0 & \text{otherwise.} \end{cases}$$

Fig. 7: Double Penalty

Reward Functions:
The total reward is the average of all the individual components, balancing task duration, completion, distance, and conflicts. This aggregate reward drives the system toward efficient task allocation strategies, considering all factors simultaneously.

$$\text{total\_reward} = \frac{\text{task\_duration\_penalty} + \text{completion\_reward} + \text{distance\_penalty} - \text{double\_allocation\_penalty}}{4}.$$

Fig. 8: Total Reward

Training: The task allocation model is trained using an already trained SwarmRL model, which focuses on path optimization in a Gazebo environment. This ensures that the task allocation algorithm is fine-tuned in a realistic setting, with robots capable of navigating the environment and handling tasks. By leveraging SwarmRL, the DQN model learns to allocate tasks while taking into account optimized robot paths, minimizing overall task completion time.

3.6 Dynamic Mapping

Our objective is to achieve comprehensive 3D mapping of an environment using LIDAR point clouds, employing a swarm of robots for efficiency.

3.6.1 Approach:

To accomplish 3D mapping, we have selected RTAB-Map (Real-Time Appearance-Based Mapping), an open-source framework for 3D mapping and localization in ROS2. RTAB-Map supports real-time 3D SLAM (Simultaneous Localization and Mapping) with various sensors, including RGB-D cameras, LIDAR, and stereo cameras.

Advantages of RTAB-Map:

- **Multi-sensor Compatibility:** It supports a wide range of sensor configurations.
- **Ease of Use:** Prebuilt ROS 2 packages simplify setup with intuitive launch configurations.
- **Scalability:** Performs well in environments of varying sizes.
- **Accuracy:** Incorporates techniques like loop closure and graph optimization to produce high-quality maps.

Working of RTAB-Map:
RTAB-Map[3] processes data from various sensors to generate 3D point clouds:

- **RGB-D Cameras:** Depth data is combined with RGB images to create colored point clouds. Disparity maps are used to calculate depth, forming 3D points.
- **LIDAR:** Directly provides 3D point clouds based on laser reflections. Each sensor's data is processed in its respective coordinate frame, and RTAB-Map integrates this information to build a cohesive point cloud.

3.6.2 Swarm Robot Deployment

Post successful mapping with a single robot, the system was scaled to incorporate multiple robots working simultaneously. The process involved the following steps:

- **Robot Spawning:** Multiple robots were spawned in a simulated environment, each assigned a unique identifier and namespace to avoid communication conflicts. ROS2 launch files were utilized to streamline the spawning process.
- **Node Assignment:** Each robot was assigned a set of nodes for core functions, such as odometry, sensor data publishing, and RTAB-Map SLAM processes. For instance, the first robot's topics were named `/robot1/odom`, `/robot1/camera/depth`, and `/robot1/camera/image_raw`, with similar configurations for additional robots.

3.6.3 Handling Environment Map

Namespace Isolation:
To ensure independent operation, each robot operated within its own namespace, leveraging ROS 2's namespace functionality to prevent topic collisions. To improve efficiency, we implemented collaborative mapping and memory persistence mechanisms.

Collaborative Mapping:

- Robots exchanged data, including pose graphs, point clouds, and map updates, using ROS 2's DDS (Data Distribution Service) communication protocols.
- Real-time updates ensured that each robot's 3D map data was synchronized within global coordinates, accommodating both static and dynamic obstacles.
- Exploration tasks were distributed to minimize redundancy and maximize efficiency.
- Collaborative mapping allows for seamless scalability, enabling faster and more accurate mapping in larger environments by incorporating additional robots.

Memory Persistence:

- Each robot uses PointNet on its RGB-D camera to identify and tag interactable locations.
- Intractable locations are stored in a real-time updating JSON file.
- An algorithm was developed to maintain memory persistence by continuously monitoring intractable states. If a robot detected changes in the position or absence of an intractable object (compared to the JSON file), it is updated or removed the corresponding entry.
- This approach ensured efficient tracking of real-time changes in the environment, maintaining the integrity of the mapped data.

By combining RTAB-Map's capabilities with collaborative mapping and robust memory management, we created a scalable and efficient system for 3D mapping with a swarm of robots.

3.6.4 Object Detection

The 3D object detection pipeline for object detection in raw point-clouds is directly taken from the robots' LiDAR data. The pipeline is implemented using PointNet and custom sampling methods on the ModelNet40 dataset, with classes including bookshelf, chair, desk, door, and person.

The primary objective is to enhance object detection by considering partial point clouds generated from specific viewpoints, simulating real-world scenarios where objects are often occluded or partially visible.

Overview of PointNet:

PointNet[4] is a neural network designed for 3D point cloud data, directly processing raw points without requiring intermediate data representations like voxels or meshes.

- **Permutation Invariance:** PointNet treats point clouds as unordered sets, making the model robust to different point ordering.
- **Efficient Feature Extraction:** It extracts both local and global features using a shared MLP (Multi-Layer Perceptron) and global max-pooling.
- **Scalability:** PointNet can handle point clouds with varying densities and sizes, which is essential for real-world applications.

Architecture:[5]

- **Input Transform:** Learns a 3x3 transformation matrix to align the input point cloud.
- **Feature Transform:** Learns a 64x64 transformation matrix to align features for better classification.
- **Global Feature Extraction:** Aggregates local features using max-pooling to obtain a global feature vector representing the entire point cloud.
- **Classification Layers:** Fully connected layers followed by batch normalization and dropout are used for classification.

Why PointNet?

Direct Processing of Raw Point Clouds: Eliminates the need for complex data preprocessing.

- **Adaptability to Partial Data:** The model performs well even with incomplete point clouds, making it ideal for occluded or cluttered environments.
- **Lightweight and Efficient:** Suitable for real-time applications

Point Cloud Sampling: To create the point clouds while training, we need to sample points from the meshes provided in the ModelNet dataset. The dataset includes the file containing the vertices and faces of each object.

- Each face of the mesh is assigned a sampling probability based on its surface area.
- Points are sampled uniformly from selected faces, ensuring diverse coverage of the mesh surface
- A random vector is generated to define the frontal view, simulating a sensor's perspective in a real-world scenario.
- Only points on the visible side of the object (i.e., those facing the frontal vector) are sampled. This mimics partial point cloud data from occluded viewpoints.
- Points are dynamically sampled based on their alignment with the specified frontal vector ensuring that the final point cloud represents a realistic sensor capture, with a minimum threshold of visible points.

To improve model robustness and simulate real-world data, several pre-processing steps were applied:

- **Point Cloud Normalization:** Normalizes the point cloud to fit within a unit sphere, ensuring consistent scale across different objects.
- **Random Rotation:** Applies random rotations around the Z-axis to increase model invariance to object orientation.
- **Random Noise Addition:** Adds Gaussian noise to simulate sensor noise and improve generalization
- **Model Training :** The PointNet model is trained on the ModelNet40 dataset using a custom training loop. Key components of the training pipeline include:
  - **Triplet Loss:** A triplet loss function is implemented to improve feature learning, ensuring that similar objects are closer in the feature space.
  - **Data Loader:** The TripletPointCloudData class generates triplet samples (anchor, positive, negative) for training, enhancing the model's ability to distinguish between similar and dissimilar objects.

4 Future Scopes and Applications

Adaptability to Various Environments: The model can be scaled to diverse settings like of-

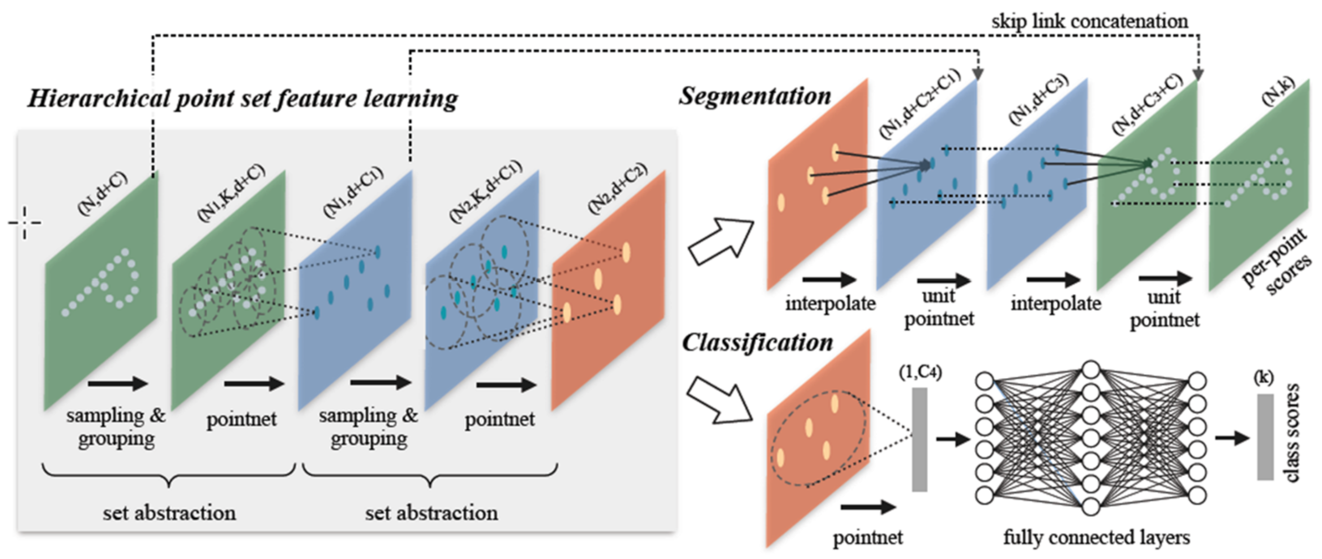


Fig. 9: Architecture of the point Net

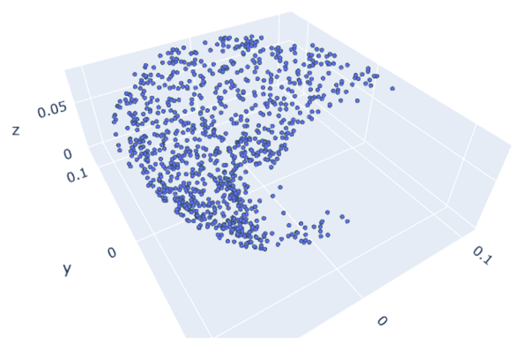
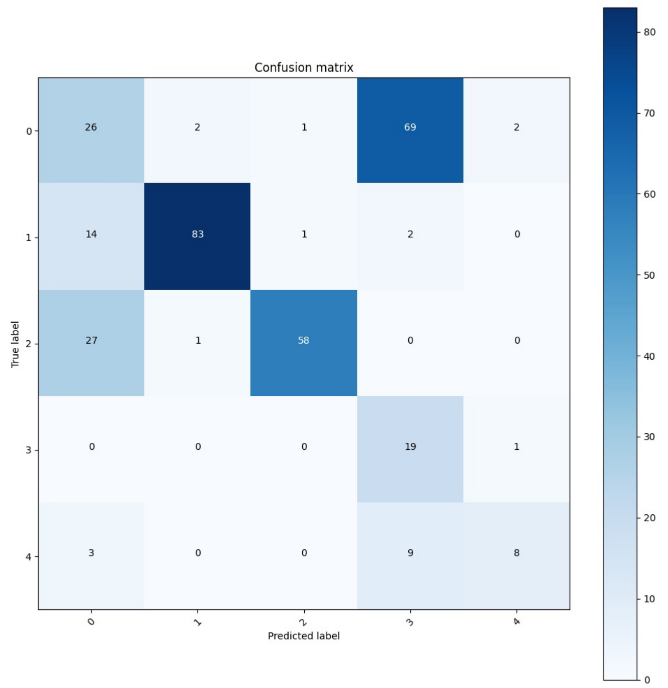


Fig. 10: Object point cloud



True label \ Predicted label	0	1	2	3	4
0	26	2	1	69	2
1	14	83	1	2	0
2	27	1	58	0	0
3	0	0	0	19	1
4	3	0	0	9	8

Fig. 11: Confusion Matrix

fice buildings, factories, hospitals, airports, schools, fairs, personal estates, and more, with flexible layouts and requirements.

Multi-Story Building Support: The model handles multi-storey buildings, scaling to accommodate different functions per floor and assigning the right number of bots per level.

Customization for Industry Needs: Applicable to sectors like healthcare (hospital corridors), education (school or university layouts), and manufacturing (assembly lines), adapting to specific operational needs.

Integration with Smart Systems: The model can integrate with IoT devices for enhanced task coordination and real-time updates, ensuring seamless operation in connected environments.

Collaboration with Humans: Swarm agents can work alongside humans in tasks like navigation or object handling, enhancing productivity in environments like labs or research institutes.

Energy Efficiency and Load Balancing: Bots optimize energy use by distributing tasks based on proximity and availability, ensuring efficient task completion and avoiding congestion.

Vertical Mobility Integration: Bots can interact with lifts, escalators, or elevators, ensuring smooth multi-floor navigation in complex buildings.

Multi-Tiered Task Automation: Bots can handle interdependent tasks across different levels or areas, such as moving materials in a factory, optimizing workflow in multi-tiered operations.

Multi-Modal Collaboration: The solution can be scaled to perform tasks that require both ground bots and drones to work in a collaborative fashion.

5 Results and Conclusions

We start off with 3D mapping the whole environment,marking and locating all the obstacles and interactables using cameras, LiDAR, point cloud perceptions etc. When an input (eg. find the nearest table) is given through the interface, then one robot from the swarm is chosen based on it's current state and that bot uses swarmRL to be able to reach the Goal position in the given environment.

Once the task is completed the result will be printed in the UI, and the bot gets back to environment exploration. Dynamic obstacles are evaded properly and dynamic interactables too are tagged in real time whenever a bot detects one, hence solving the problem of collaborative mapping and memory persistence. The swarm system is therefore able to map the environment and navigate through it real-time while maintaining memory persistence of interactables and obstacles.

We believe that our solution will prove to be a powerful tool and holds great potential for efficiently executing tasks on deployment in industrial landscapes.

References

[1] S. S. Kannan, V. L. N. Venkatesh, and B.-C. Min, "Smart-llm: Smart multi-agent robot task planning using large language models," *arXiv preprint arXiv:2309.10062*, 2023, Accessed: 2024-12-05. [Online]. Available: https://sites.google.com/view/smart-llm/.

[2] V. Konda and J. Tsitsiklis, "Actor-critic algorithms," *Society for Industrial and Applied Mathematics*, vol. 42, Apr. 2001.

[3] M. Labbé, "Rtab-map as an open-source lidar and visual slam library for large-scale and long-term online operation," *arXiv preprint arXiv:2403.06341*, 2024, Accessed: 2024-12-05. [Online]. Available: https://arxiv.org/abs/2403.06341.

[4] C. R. Qi, H. Su, K. Mo, and L. J. Guibas, "Pointnet: Deep learning on point sets for 3d classification and segmentation," *arXiv preprint arXiv:1612.00593*, 2017, Accessed: 2024-12-05. [Online]. Available: https://arxiv.org/abs/1612.00593.

[5] *3d point cloud classification on modelnet40*, https://paperswithcode.com/sota/3d-point-cloud-classification-on-modelnet40, Accessed: 2024-12-05.