# Overview

This project applies imitation learning to robot motion control in **Gazebo + ROS.** Expert trajectories are generated with the built-in *move_base* stack in Gazebo. A dataset is automatically collected in 2.5 hours, and extracted from a rosbag, a NN is trained **with five input features** (x, y, θ, goal_x, goal_y) and finally deployed in a **hybrid NN + sequential PID** controller that runs onboard the TurtleBot3.

The emphasis of this document is on the data pipeline and the **new hybrid controller** that appears in the attached nn_controller.py.

## Objective

The goal was to automatically collect navigation data from a TurtleBot3 in Gazebo, preprocess it, train a NN controller, and implement it as a **control system** in ROS to navigate to user-defined setpoints.

## Implementation Process

### 1. Data Collection: Automating Navigation Goals

- **File**: goals_sender.py
- **Purpose**: To automate the process of sending navigation goals to the TurtleBot3 in Gazebo, thereby generating a diverse dataset of expert navigation trajectories provided by the move_base stack.
- **Operation Flow**:
    - **Goal Definition**: A list of **43** predefined (x, y, theta) goals is created. The orientation theta is fixed at 0.0 to standardize the final heading.

- **ROS Action Client**: An actionlib.SimpleActionClient is initialized to communicate with the /move_base action server, which handles the robot's navigation.
- **Execution Loop**:
  - On the first run, the script iterates sequentially through all 43 goals.
  - On subsequent runs, it selects a random goal from the list.
- **Goal Transmission**: The send_goal function converts the yaw angle to a quaternion, sends the goal, and waits up to **150 seconds** for completion, sleeping for **1.5 seconds** on success to prevent system overload.

- **Logic and Rationale**:
  - **Diverse Trajectories**: Iterating through all goals and then selecting randomly ensures the dataset includes a wide variety of paths, improving the NN's ability to generalize across the map.
  - **Robust Goal Management**: Using actionlib provides reliable goal handling with feedback on success or failure.
  - **Dependency Management**: Manual quaternion conversion avoids external libraries like tf, simplifying setup.
  - **System Stability**: The timeout and sleep periods ensure stable navigation and prevent overwhelming the system.

## 2. Data Recording: Capturing ROS Topics

- **Command**: rosbag record -O Recorded_Robot /gazebo/model_states /cmd_vel /move_base/goal

- **Purpose**: To record the robot's state, control inputs, and navigation goals during the automated runs.
- **Operation Flow**:
    - **Recorded Topics**:
        - **/gazebo/model_states**: Provides the robot's current pose (x, y, θ).
        - **/cmd_vel**: Captures the control inputs (linear velocity $v$ and angular velocity $w$).
        - **/move_base/goal**: Logs the goals being pursued.
    - **Execution**: This command runs in a terminal while goals_sender.py operates, saving the data into a .bag file.
- **Logic and Rationale**:
    - **Comprehensive Data**: These topics capture the complete information needed for imitation learning: the state (pose), the expert action (cmd_vel), and the reference (goal).
    - **ROS Bag Format**: This standard format preserves timestamps and message structures, facilitating accurate preprocessing.

## 3. Data Preprocessing: Extracting and Synchronizing Data

- **File**: preprocess_bag.py
- **Purpose**: To convert the recorded ROS bag file into a synchronized CSV dataset suitable for NN training.
- **Operation Flow**:

- **Data Extraction**: The script iterates through the rosbag messages. It extracts the robot's pose from /gazebo/model_states, the goal pose from /move_base/goal, and the control commands from /cmd_vel.
- **Synchronization Logic**: A new data row [x, y, theta, goal_x, goal_y, goal_theta, v, w] is created and appended to the dataset **every time a /cmd_vel message is received**. This row uses the most recently recorded values for current_pose and goal_pose.
- **Output**: The synchronized dataset is saved as training_data.csv.

- **Logic and Rationale**:
  - **Temporal Alignment**: This synchronization approach ensures that every expert control action (v, w) is paired with the state and goal that produced it, which is critical for supervised learning.
  - **Consistent Orientation**: Using euler_from_quaternion ensures a consistent representation of the robot's and goal's orientation (yaw).
  - **CSV Format**: This universal format simplifies data loading for the training phase.

## 4. Neural Network Training: Learning from Collected Data

- **File**: training_notebook.ipynb
- **Purpose**: To train a NN to predict control actions (*v*, *w*) based on the robot's current state and goal.

- **Operation Flow**:
  - **Data Loading**: The script loads training_data.csv, shuffles it randomly, and splits it into an 80% training and 20% validation set.
  - **Feature and Target Selection**:
    - **Inputs (X)**: 6 features representing the state and goal: x, y, theta, goal_x, goal_y, and goal_theta.
    - **Outputs (y)**: 2 target variables: v and w.
  - **Model Architecture**: A **7-layer** deep NN was defined:
    - **Input Layer**: 6 neurons
    - **Hidden Layers (ReLU)**: 256 → 128 → 128 → 64 → 64 → 32 neurons
    - **Output Layer (Linear)**: 2 neurons for $v$ and $w$
  - **Training**:
    - **Optimizer**: Adam with a learning rate of **0.0005**.
    - **Loss Function**: Mean Squared Error (MSE).
    - **Regularization**: Early stopping with a **patience of 25 epochs** was used to prevent overfitting.

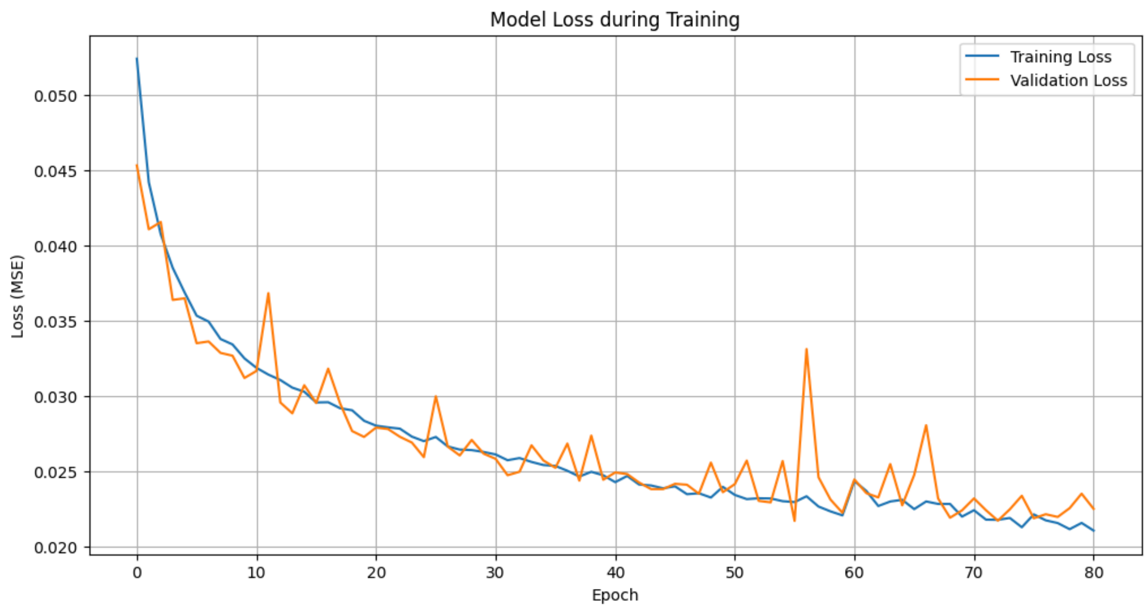


**Figure 4.** Model Loss Plot

- **Output**: The trained model is saved as model.h5.

- **Logic and Rationale**:
  - **Data Randomization**: Shuffling and splitting ensure an unbiased evaluation of the model's performance on unseen data.
  - **Network Design**: The deep architecture is designed to capture the complex, non-linear relationship between the robot's state/goal and the expert's control commands. ReLU activation aids gradient flow, and early stopping ensures the model generalizes well.
  - **Regression Task**: The linear output layer and MSE loss are appropriate for this regression task, where the goal is to predict continuous velocity values.

## 5. Hybrid NN+PID Controller Implementation

- **File**: nn_controller.py
- **Purpose**: To deploy the trained NN in a robust, hybrid control scheme. The NN handles long-range navigation, while a sequential PID controller takes over for precise docking at the goal.
- **Operation Flow**:
    - **Model Loading**: The node loads the weights and biases from model.h5 directly into NumPy arrays using the h5py library, avoiding a full TensorFlow/Keras dependency.
    - **Forward Pass**: A pure NumPy nn_forward function is implemented to perform the NN's forward propagation.
    - **ROS Node Setup**: The node subscribes to /gazebo/model_states for the current pose and /reference_pose for the goal coordinates [x_r, y_r]. It publishes commands to /cmd_vel and a boolean flag to /goal_reached.
    - **Hybrid Control Logic**:
        - The distance to the goal is calculated at each step.
        - **NN Mode (Far)**: If the distance is greater than **0.35m**, the node uses the NN. It constructs a 6D input vector [x, y, th, gx, gy, 0.0] and calls nn_forward to predict $v$ and $w$.
        - **PID Mode (Near)**: If the distance is less than or equal to **0.35m**, the node switches to a sequential PID controller. This controller executes three phases in order:

        1. align1: Rotate to face the goal.
        2. drive: Move straight toward the goal.
        3. align2: Rotate to the final desired orientation (0 radians).
    - Once the PID controller finishes, the node publishes True to /goal_reached.

- **Logic and Rationale**:
    - **Hybrid Approach**: This design combines the strengths of both control methods. The NN, trained on expert data, provides efficient, human-like navigation over long distances. The classical PID controller ensures guaranteed precision and stability for the final approach, a task where NNs can sometimes struggle.
    - **State Representation**: The 6D input vector for the NN matches the format of the training data, ensuring accurate predictions.
    - **Safe Operation**: The predicted velocities are clamped to the TurtleBot3's maximum limits to ensure safe and realistic operation.
    - **Modularity**: Publishing a /goal_reached flag allows for easy integration with a higher-level planner like motion_planner.py.

## 6. Motion Planning: User-Defined Setpoints

- **File**: motion_planner.py
- **Purpose**: To provide a simple user interface for testing the controller by sending sequential goals.

- **Operation Flow**:
  - **User Input**: The script prompts the user to enter target X and Y coordinates.
  - **Goal Publishing**: It publishes the [x_r, y_r] coordinates to the /reference_pose topic as a Float64MultiArray.
  - **Sequential Execution**: The script subscribes to the /goal_reached topic and waits until it receives a True message before prompting the user for the next goal.
- **Logic and Rationale**:
  - **User-Friendly Testing**: This node allows for easy, interactive testing of the nn_controller with any custom goal within the map.
  - **Task Management**: By waiting for the /goal_reached signal, it ensures that goals are executed one at a time, mimicking a real-world sequential task planner.

# Discussion

Automated data collection and careful preprocessing were critical to the success of this problem. The key innovation was the implementation of a **hybrid controller**. The NN generalized well across diverse navigation scenarios, but by handing over control to a deterministic PID controller near the goal, the system achieved high precision and reliability at the goal boundaries. This hybrid approach proved to be a robust solution, combining the learned efficiency of the NN with the predictable accuracy of classical control