Mechatronics Summer School 2025

Prepared by:

The Tech Heads

Jaideep Nirmal AJ, *Head, Mechatronics*

Kaushal Morankar, *Head, Machine Learning*

and *Perplexity.ai*

Tinkerers' Lab, IIT Hyderabad

Between A, B, G, H Hostel Block, IIT Hyderabad

IITH Main Road, Near NH-65, Sangareddy, Telangana 502285

.

Module 5
Intelligence

Contents

1 Neural Networks

A neural network is a computational model, inspired by the biological neural networks of the human brain, designed to recognize patterns and relationships in data. Every neural network consists of layers of nodes or artificial neurons, an input layer, one or more hidden layers, and an output layer.Neural networks rely on training data to learn and improve their accuracy over time. Once they are fine-tuned for accuracy, they are powerful tools in computer science and artificial intelligence, allowing us to classify and cluster data at a high velocity.

How do neural networks work?

Think of individual node as its own linear regression model, composed of input data, weights, a bias and an output.

- **Input Layer**
  Information from the outside world enters the artificial neural network from the input layer. Input nodes process the data, analyze or categorize it, and pass it on to the next layer.
- **Hidden layer**
  Hidden layers take their input from the input layer or other hidden layers. ANN can have a large number of hidden layers. Each hidden layer analyzes the output from the previous layer, processes it further, and passes it on to the next layer.
- **Output Layer**
  The output layer gives the final result of all the data processing by the ANN. It can have single or multiple nodes. For instance, if we have a binary (yes/no) classification problem, the output layer will have one output node, which will give the result as 1 or 0. However, if we have a multi-class classification problem, the output layer might consist of more than one output node.

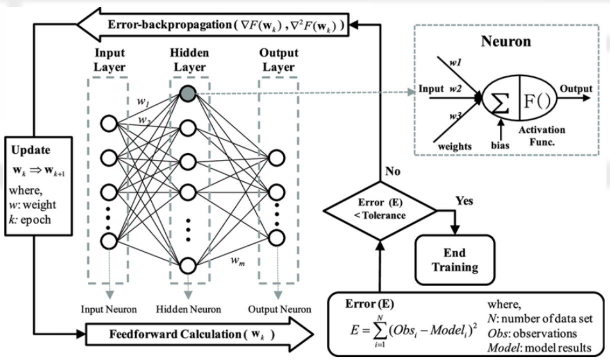

Figure 1: Neural nets

Mathematical Representation

The output of a single artificial neuron can be expressed as:

$$y = f\left(\sum_{i=1}^{n} w_i x_i + b\right) \tag{1}$$

where:

- x_i are the input features,
- w_i are the corresponding weights,
- b is the bias,
- f is the activation function,
- y is the output.

Types of Neural Networks

Artificial neural networks can be categorized by how the data flows from the input node to the output node. Below are some examples:

Feedforward Neural Networks

Feedforward neural networks process data in one direction, from the input layer to the output layer. Every node in one layer is connected to every node in the next layer, and there are no cycles or loops.

- Data moves only forward through the network.
- Each neuron performs a weighted sum of its inputs and applies an activation function.

Mathematically, the output of a feedforward neural network can be represented as:

$$\mathbf{y} = f^{(L)}\left(\mathbf{W}^{(L)} f^{(L-1)}\left(\mathbf{W}^{(L-1)} \cdots f^{(1)}\left(\mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}\right) + \cdots + \mathbf{b}^{(L-1)}\right) + \mathbf{b}^{(L)}\right) \tag{2}$$

where:

- $\mathbf{x}$ is the input vector,
- $\mathbf{W}^{(l)}$ and $\mathbf{b}^{(l)}$ are the weight matrix and bias vector for layer l,
- $f^{(l)}$ is the activation function at layer l,
- $\mathbf{y}$ is the final output.

Backpropagation Algorithm

Backpropagation is a supervised learning algorithm used for training neural networks. It adjusts the weights of the network based on the error rate obtained in the previous epoch (iteration). The goal is to minimize the loss function.

The steps include:

1. Forward pass: Compute the output.
2. Compute loss: Use a loss function $L(\hat{y}, y)$.
3. Backward pass: Calculate gradients of loss with respect to weights.
4. Update weights using gradient descent.

Mathematically, the gradient of the loss with respect to weights w is calculated using the chain rule:

$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z} \cdot \frac{\partial z}{\partial w} \tag{3}$$

And the weights are updated as:

$$w := w - \eta \cdot \frac{\partial L}{\partial w} \tag{4}$$

where:

- w is the weight,
- η is the learning rate,
- L is the loss function,
- $\hat{y}$ is the predicted output,
- z is the weighted input to a neuron.

Resources:
Pytorch
Stanford NN
3B1B

Neural Networks in ROS and Robotics

Neural networks are increasingly used in robotics applications to enable perception, decision-making, and control. Within the Robot Operating System (ROS), neural networks can be integrated into robotic systems through packages that support deep learning models.

- **Perception:** Neural networks are widely used for image recognition, object detection, and semantic segmentation in ROS-enabled robots. For example, Convolutional Neural Networks (CNNs) can process camera input to detect objects in the environment.
- **Control:** Deep Reinforcement Learning (DRL) techniques are used in robotics to learn control policies. ROS can work with tools like OpenAI Gym and Gazebo to simulate environments where robots learn via trial and error.
- **Integration with ROS:** Neural networks can be deployed in ROS using nodes written in Python or C++ that interface with machine learning frameworks such as TensorFlow, PyTorch, or ONNX. The 'ros_numpy', 'cv_bridge', and 'sensor_msgs' packages help in passing sensor data to models and acting on their output.
- **Example Use Cases:**
  - Autonomous navigation using LiDAR and camera-based scene understanding.
  - Robotic arms using neural nets for grasp detection and manipulation.
  - Speech and gesture recognition for Human-Robot Interaction (HRI).

2 Computer Vision

Computer Vision (CV) is a subfield of Artificial Intelligence (AI) that enables computers and systems to derive meaningful information from digital images, videos, and other visual inputs. It empowers machines to interpret, analyze, and make decisions based on visual data, allowing automation of tasks such as image classification, face recognition, object detection, segmentation, and scene understanding.

If AI enables computers to think, computer vision enables them to see, observe, and understand.

How Computer Vision Works

Computer vision systems work by training computers to recognize and extract patterns from visual data. This is typically achieved through machine learning (ML) and deep learning algorithms, which are trained on large datasets consisting of labeled images or videos. During this process, the algorithms learn to identify features and patterns associated with specific objects, scenes, or actions.

For example, when a computer vision model is trained on millions of images of cars from different angles and backgrounds, it gradually learns the characteristic features of cars—such as shape, color, edges, and textures. As a result, the system becomes capable of accurately detecting and recognizing cars in previously unseen images.

Key Technologies in Computer Vision

Modern computer vision relies heavily on deep learning techniques, which have significantly advanced the field. Some of the core technologies used include:

- **Deep Learning:** Neural networks with many layers (deep neural networks) that can learn hierarchical representations of data, capturing complex and abstract visual patterns.
- **Convolutional Neural Networks (CNNs):** Specialized neural networks particularly effective for analyzing and classifying visual data. CNNs automatically learn spatial hierarchies of features from input images, such as edges, shapes, and textures.
- **Recurrent Neural Networks (RNNs):** Neural networks designed to handle sequential data. In computer vision, RNNs are often combined with CNNs to analyze video data, model temporal dynamics, and understand actions across frames.

Computer Vision Processing Pipeline

The process of computer vision generally follows a pipeline that starts from raw visual data and ends in actionable insights or decisions. The main stages include:

1. **Image Acquisition:** Capturing images or video streams using cameras or other sensors.
2. **Preprocessing:** Enhancing image quality and preparing data for analysis. This can include:
   - Noise reduction (e.g., using filters)
   - Normalization and resizing
   - Color space conversion (e.g., RGB to grayscale)
   - Image augmentation (rotations, flips, scaling) for training robustness
3. **Feature Extraction:** Identifying meaningful patterns or descriptors in the images. Traditionally, this involved algorithms like SIFT or HOG. Today, deep learning models like CNNs automatically learn and extract hierarchical features, from edges and textures in early layers to complex object parts in deeper layers.
4. **Recognition / Classification:** Using machine learning models (such as CNN classifiers) to categorize the visual input into predefined classes, or detect and localize specific objects within the image.
5. **Post-processing and Decision Making:** Refining the model outputs (e.g., applying non-maximum suppression in object detection) and integrating them into higher-level tasks, such as tracking, control commands for a robot, or triggering alerts.

This pipeline transforms raw visual data into structured information that robots and AI systems can use to perceive and interact with their environment effectively.

Convolution Neural Nets

Convolutional Neural Networks (CNNs) are a powerful and widely adopted tool for feature extraction in computer vision. While CNNs can perform end-to-end classification and detection, their most critical contribution lies in their ability to automatically learn and extract meaningful features from raw visual data.

In this section, we focus specifically on feature extraction, as it is the most crucial step in the computer vision pipeline.

CNNs excel at capturing hierarchical features through successive layers:

- **Lower layers** detect simple patterns such as edges, corners, and color gradients.

- **Intermediate layers** identify more complex shapes, textures, and parts of objects.
- **Deeper layers** abstract high-level representations like full objects or semantic regions.

To detect whether or not a feature is in an image patch we can use a filter. The key behind a filter is a kernel, which for grayscaled images is a matrix, for colored images a tensor with as many color channels as in the image. Let's consider the grayscaled case.

The Convolution Operation

The core of a CNN is the convolution operation, where a small matrix (kernel) slides across the input image, computing dot products to generate a feature map. Each kernel is trained to detect a specific feature, such as a vertical edge or a texture pattern. During training, the network learns which features (and their corresponding filters) are most relevant for the task.

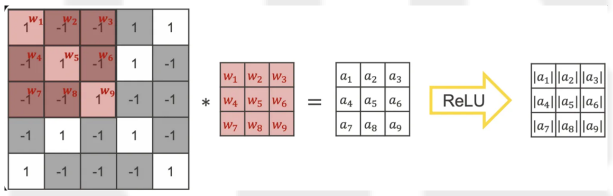

Figure 2: CNN

Pooling Layers

Pooling layers, such as max pooling or average pooling, reduce the spatial size of feature maps and help in making the representations invariant to small translations. This reduces computational cost and prevents overfitting by summarizing feature activations in regions.

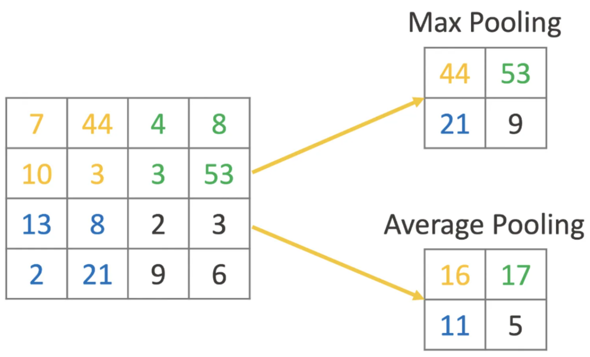

Figure 3: Pooling layers

End-to-End CNN Architecture

A typical CNN architecture for image classification includes:

- **Input layer:** Accepts image data (e.g., 28×28×1 for grayscale).
- **Feature extractor:** Multiple convolution and pooling layers stacked to learn hierarchical features.
- **Flattening layer:** Transforms the final feature maps into a vector.
- **Fully connected layers:** Learn final decision boundaries for classification.
- **Output layer:** Usually a softmax activation for multiclass classification.

Training a CNN from scratch requires large amounts of labeled data and computational resources. To overcome this, practitioners often use **pretrained CNN models**, which are models that have already been trained on large benchmark datasets such as *ImageNet*.

These pretrained models can be used in two main ways:

- **Feature Extraction:** The pretrained CNN is used as a fixed feature extractor. The input image is passed through the convolutional base of the network, and the resulting feature maps are used as input to a separate classifier (e.g., SVM or a shallow neural network).
- **Fine-tuning:** The pretrained model is partially retrained on a new dataset. This involves freezing some layers (typically early ones that learn general features like edges and textures) and retraining later layers to adapt the model to the specific task or domain.

Popular pretrained models include:

- **VGG16 / VGG19:** Known for their simple and deep architecture.
- **ResNet:** Introduces residual connections that help in training very deep networks.
- **Inception / GoogLeNet:** Efficient at capturing multi-scale features.
- **MobileNet / EfficientNet:** Lightweight models suitable for mobile and edge devices.

Resources

[CS231n Notes on CNNs (Stanford)]
3B1B intution
Complete guide Andrew Ng
medium Knime

Applications of Computer Vision

Computer vision has a wide range of real-world applications, including:

- **Object Detection and Tracking:** Identifying and localizing objects in images or videos and tracking their movement over time.
- **Medical Imaging:** Assisting doctors by analyzing medical scans to detect anomalies.
- **Autonomous Vehicles:** Enabling vehicles to perceive and understand their surroundings for safe navigation.
- **Augmented Reality:** Overlaying digital content onto real-world scenes by recognizing and tracking objects and environments.

3 YOLO

YOLO : - You Only Look Once

YOLO (You Only Look Once) is a real-time **Single short** object detection algorithm that frames detection as a single regression problem — directly from image pixels to bounding boxes and class probabilities. Unlike traditional two-stage detectors (e.g., R-CNN), YOLO performs detection in one forward pass of the network, making it extremely fast and suitable for real-time applications.

The YOLO algorithm divides the input image into a grid of cells, and for each cell, it predicts the probability of the presence of an object and the bounding box coordinates of the object. It also predicts the class of the object. Unlike two-stage object detectors such as R-CNN and its variants, YOLO processes the entire image in one pass, making it faster and more efficient.

YOLO is widely used in various applications such as self-driving cars and surveillance systems. It is also widely used for real-time object detection tasks like in real-time video analytics and real-time video surveillance.

YOLO algorithm

The basic idea behind YOLO is to divide the input image into a grid of cells and, for each cell, predict the probability of the presence of an object and the bounding box coordinates of the object. The process of YOLO can be broken down into several steps:

1. Input image is passed through a CNN to extract features from the image.
2. The features are then passed through a series of fully connected layers, which predict class probabilities and bounding box coordinates.
3. The image is divided into a grid of cells, and each cell is responsible for predicting a set of bounding boxes and class probabilities.
4. The output of the network is a set of bounding boxes and class probabilities for each cell.
5. The bounding boxes are then filtered using a post-processing algorithm called non-max suppression to remove overlapping boxes and choose the box with the highest probability.
6. The final output is a set of predicted bounding boxes and class labels for each object in the image.

YOLOv8: Mathematical Foundations & Workflow

YOLOv8 is a real-time object detection system that significantly improves upon its predecessors by removing anchor boxes and using advanced loss functions and architectural innovations.

1. Grid and Predictions

The input image is divided into an $S \times S$ grid. Each grid cell is responsible for predicting object presence and its bounding box:

- $\sigma(t_x), \sigma(t_y)$: center offset of the predicted box within the grid cell.
- e^{t_w}, e^{t_h}: predicted width and height.
- p_{obj}: objectness score.
- $p(c_i|\text{obj})$: conditional class probabilities.

YOLOv8 is **anchor-free**, meaning it does not rely on predefined anchor boxes. Predictions are made directly from feature map points.

2. Bounding Box Decoding

The model predicts bounding boxes using the following transformations:

$$\begin{aligned} b_x &= \sigma(t_x) + c_x, \\ b_y &= \sigma(t_y) + c_y, \\ b_w &= e^{t_w} \cdot p_w, \\ b_h &= e^{t_h} \cdot p_h \end{aligned}$$

where (c_x, c_y) is the top-left offset of the grid cell, and (p_w, p_h) are the stride-based scales or reference values.

3. Loss Function

The total YOLOv8 loss $\mathcal{L}$ is a weighted sum of three components:

$$\mathcal{L} = \lambda_{\text{loc}} \cdot \mathcal{L}_{\text{box}} + \lambda_{\text{obj}} \cdot \mathcal{L}_{\text{obj}} + \lambda_{\text{cls}} \cdot \mathcal{L}_{\text{cls}}$$

- **Localization Loss** ($\mathcal{L}_{\text{box}}$): Based on Complete IoU (CIoU) which considers overlap, center distance, and aspect ratio:

$$\mathcal{L}_{\text{box}} = 1 - \text{CIoU}(b, \hat{b})$$

- **Objectness Loss** ($\mathcal{L}_{\text{obj}}$): Binary Cross Entropy (BCE) loss between predicted objectness score and ground truth.

- **Classification Loss** ($\mathcal{L}_{\text{cls}}$): Cross-entropy loss across the predicted class probabilities, optionally enhanced with Focal Loss:

$$\mathcal{L}_{\text{cls}} = -\sum_{i=1}^{C} \alpha(1 - p_i)^{\gamma} \log(p_i)$$

4. Architecture Highlights

YOLOv8 incorporates the following improvements:

- **Backbone:** CSPDarknet with deeper layers for better feature extraction.
- **Neck:** PANet for feature aggregation across scales.
- **Head:** Anchor-free prediction and regression head for bounding boxes and classification.

5. Post-Processing: Non-Maximum Suppression (NMS)

After prediction:

1. Discard boxes with objectness score $< \tau$ (threshold).
2. Sort remaining boxes by confidence.
3. Apply Non-Max Suppression: remove overlapping boxes with IoU $> \theta$.

6. Summary Table

Component	**Details**
Grid	$S \times S$ cells
Bounding Box	(b_x, b_y, b_w, b_h) decoded from (t_x, t_y, t_w, t_h)
Loss Function	CIoU + BCE + Focal/Cross-Entropy
Anchor Mechanism	Anchor-free
Post-processing	NMS with IoU filtering

Resources

DeepLearning AI

YOLO in Robotics and ROS

1. Role of YOLO in Robotics

YOLO (You Only Look Once) is widely used in robotic systems for real-time object detection, enabling machines to perceive and react to their environment. Applications include autonomous navigation, pick-and-place tasks, drone surveillance, warehouse automation, and quality inspection.

2. Integration with ROS

ROS (Robot Operating System) allows YOLO to be integrated into a modular robot system. A typical setup involves:

- A camera node publishing images.
- A YOLO node processing frames and detecting objects.
- Publishing detection results as ROS messages.

3. ROS-Compatible YOLO Packages

- **darknet_ros**: Integration of YOLOv2/v3 with ROS.
- **ros-yolov5**: ROS wrapper for YOLOv5 using PyTorch.
- **vision_msgs**: Standardized detection message types.

4. Practical Uses in Mechatronics

- Detecting and tracking objects using a webcam or robot-mounted camera.
- Interfacing detections with motors or actuators for pick-and-place.
- Using detections for robotic decision-making in real-time.

5. Visualization and Simulation

Detections can be visualized in **RViz**, and simulated using **Gazebo** with YOLO output controlling robot behavior in a virtual environment.

4 YOLO on ROS

Running a CV Model on ROS is a rather straightforward task. First, download the pre-trained YOLO model from this link (access through IITH Email ID only). This is roughly 54MB in size and save this in your workspace subfolder
`src/PACKAGE_NAME/PACKAGE_NAME`
This should be at the same level as your python file for nodes. This `.pt` file is nothing but the weights for different nodes and for different classes of the model. Now all that is left to do is instantiate an object of YOLO and make inferences by passing the camera data as a numpy array to it.

The workflow will be as follows:

- Instantiate a YOLO object using the weights from the `.pt` file
- Subscribe to a camera topic
- Set a minimum time difference for making an inference call. This is a simulation thing. Without setting a minimum time difference, inference calls would be made at every instant, increasing the workload on CPU and GPU and thus resulting in a slow/laggy simulation
- The msg of the camera has three things - the height (`int`), the width (`int`) and the image array (`int[height*width*3]`). This needs to be reshaped to a shape `(height, width, 3)`
- Make an inference call to the model. BOOM! Then get the list of results and print them one by one
- Publish these to an appropriate topic if needed

WATCH OUT!

YOLO v8 requires an older version of numpy. You may have some issues with the latest version. If the errors being spit out is suggestive of this, then downgrade your numpy to an appropriate version as mentioned in the error.

5 ROS Code

```
#!/usr/bin/env/ python3

import rclpy
from rclpy.node import Node
from sensor_msgs.msg import Image
from ultralytics import YOLO
import numpy as np
import time
from std_msgs.msg import String

MIN_DELTA_T_FOR_REFRESH = 0.8 # secs

class Recogniser(Node):

    def __init__(self):
        super().__init__('recogniser')
        self.model = YOLO("yolov8m-seg.pt")
        # subscribe to the camera
        self.rgb_subscription = self.create_subscription(
            Image,
            '/robot_0/camera/image_raw',
            self.rgb_callback,
            20
        )
        self.object_publisher = self.create_publisher(String, '/vision_updates', 10)
        self.get_logger().info('Recogniser node started')
        self.last_saved_time_rgb = time.time()


    def rgb_callback(self, msg):
        current_time = time.time()
        if current_time - self.last_saved_time_rgb >= MIN_DELTA_T_FOR_REFRESH:
            # get img_array
            img_array = np.frombuffer(msg.data, dtype=np.uint8).reshape(msg.height, msg.width,
     -1)

            # recognise images & publish them
            msg = String()
            msg.data += f'Objects recognised at time {time.time()}:\n'

            # inference call (BOOM! the one line that does it all!)
            current_results = self.model(img_array, verbose=False)

            for i, result in enumerate(current_results):
                current_obj = result.names[int(result.boxes.cls.item())]
                msg.data += f'{i+1}: {current_obj}\n'
            self.object_publisher.publish(msg)

            self.last_saved_time_rgb = time.time()


def main(args=None):
    rclpy.init(args=args)
    recogniser = Recogniser()
    rclpy.spin(recogniser)
    recogniser.destroy_node()
    rclpy.shutdown()

if __name__ == '__main__':
    main()
```

6 Simultaneous Localization and Mapping (SLAM)

What is SLAM?

SLAM stands for *Simultaneous Localization and Mapping*. It is a fundamental computational technique that enables robots and autonomous vehicles to:

- **Build a map** of an unknown environment.
- **Localize themselves** within that map simultaneously.

Robots use onboard sensors (such as cameras, LiDAR, and IMUs) along with algorithms to gather and interpret environmental data. This allows them to incrementally create a map and track their own position within it in real time.

Why is SLAM Important in Robotics?

SLAM is crucial to achieving true robot autonomy. Below are some key reasons why SLAM is essential in robotics:

1. Enabling Autonomy

- **Navigation in Unknown Environments:** SLAM allows robots to operate where no prior map exists, such as disaster zones or alien terrains.
- **Dynamic Adaptation:** Robots can adjust to changes in the environment in real time.

2. Efficient and Intelligent Task Execution

- **Path Planning:** Enables robots to plan optimized routes and avoid obstacles.
- **Resource Efficiency:** Ensures that tasks (e.g., cleaning, material transport) are completed without redundancy.

3. Expanding Practical Applications

- **Industrial Automation:** Used in warehouse robots and automated forklifts.
- **Consumer Robotics:** Essential for robotic vacuum cleaners, lawn mowers, etc.
- **Autonomous Vehicles:** Forms the basis of navigation and obstacle avoidance for self-driving cars and drones.

4. Safety and Accessibility

- **Hazardous Environments:** Used in remote-controlled or autonomous robots for search and rescue, mining, or structural inspection.

Summary Table: SLAM in Robotics

Feature	**Description & Benefit**
Core capability	Map creation and self-localization simultaneously.
Key sensors	Cameras, LiDAR, Sonar, IMU (Inertial Measurement Unit).
Main applications	Autonomous vehicles, service robots, drones, industrial robots.
Primary benefits	Autonomy, efficiency, adaptability, safety.
Real-world examples	Robotic vacuums, warehouse robots, self-driving cars, exploratory drones.

7 Localization

Overview

Localization is a key process within Simultaneous Localization and Mapping (**SLAM**) in robotics. It refers to a robot's capability to *estimate its position and orientation* (collectively known as its *pose*) in an environment while it is actively mapping that environment.

What is Localization?

- **Localization** estimates where a robot is, using sensor data and available environmental knowledge.
- In SLAM, the robot *simultaneously builds a map* of an unknown environment and *localizes itself* within that map.
- Both **mapping** and **localization** update in real-time, which is crucial for autonomous navigation.

*Importance in SLAM

- **Navigation**: Accurate localization enables robots to move efficiently and avoid obstacles.
- **Task Execution**: Essential for meaningful interaction with the environment, from industrial tasks to household chores.
- **Robustness**: Continuous localization corrects for errors and sensor noise, maintaining reliable operation.

How Localization Works in SLAM

- **Sensor Input**: The robot uses data from sensors (such as cameras, LiDAR, IMUs).
- **Feature Detection**: Environmental features are extracted to match with the map.

- **Estimation Algorithms**: Algorithms like particle filters or Kalman filters are used to estimate the pose.
- **Map and Pose Refinement**: Both the map and the robot's estimated pose are updated iteratively, reducing uncertainty.

8 Mapping

Mapping in SLAM refers to the process by which a robot or autonomous vehicle **constructs a representation (map) of its unknown environment** while exploring it. As the robot moves, sensors such as cameras, LiDAR, or sonar are used to collect spatial data, capturing features like walls, objects, and landmarks within the surroundings.

The mapping component of SLAM involves:

- **Recording and processing sensor measurements**: The robot continuously acquires data about distances and objects in its vicinity.
- **Extracting and associating features**: Distinctive features (e.g., corners, edges, or landmarks) are identified and tracked as the robot explores.
- **Incrementally building and updating the map**: As new measurements are integrated, the robot updates its internal map structure, linking features together based on their spatial relationships [?, ?].

Because mapping is done simultaneously with localization (estimating the robot's own position), the process must cope with uncertainty and sensor noise. Probabilistic algorithms such as the Extended Kalman Filter or Particle Filter are often used to maintain and update a consistent map in the face of imperfect data [?].

Mapping in SLAM is essential for autonomous navigation, path planning, and efficient task completion, enabling robots to operate effectively in unknown or dynamic environments without prior maps.

9 How is Localization Different from Mapping?

Localization and mapping are two fundamental, yet distinct, components of SLAM.

Localization

Localization is the process by which a robot estimates its own position and orientation (*pose*) within a given map. It relies on sensor data to compare observations of the environment with features or landmarks in a pre-built map (if available), or with a map being constructed in real time. In essence, localization answers the question: *"Where am I on the map?"*.

Mapping

Mapping refers to the creation or updating of a representation (*map*) of the environment, using sensor measurements collected as the robot moves. It involves recording features, obstacles, and spaces to build a spatial model. Here, the central question is: *"What does the world around me look like?"*.

Key Differences

- **Information dependency:** Localization requires a map as a reference, while mapping requires knowledge of the robot's accurate pose to place features correctly.
- **Goal:** The goal of localization is to estimate the robot's pose; mapping's goal is to generate or update the map of the environment.
- **Standalone operation:**
  - If a map already exists, only localization is needed.
  - If no map exists, the robot must perform both mapping and localization (SLAM) simultaneously, as each depends on the other.
- **Complexity:** Mapping in unknown environments is generally more challenging, since both the robot's pose and the environment are uncertain and must be estimated together.

Summary Table

Aspect	**Localization**	**Mapping**
Primary Question	Where am I?	What does the environment look like?
Requires Map?	Yes, as reference	No, but requires pose information
Output	Robot's pose in the map	Map of the environment
Role in SLAM	Tracks position	Builds map

10 SLAM on ROS

The `slam_toolbox` package is a widely used solution for 2D Simultaneous Localization and Mapping (SLAM) within the Robot Operating System (ROS) ecosystem. It provides both mapping and localization capabilities, robust performance, and easy integration with robotic platforms.

Prerequisites

Before implementing SLAM with `slam_toolbox`, ensure the following:

- A robot equipped with a LIDAR or laser scanner publishing `sensor_msgs/LaserScan` data (commonly on the `/scan` topic).
- Odometry data published on the relevant `tf` (`odom` to `base_link`) or the appropriate topics.
- ROS (ROS 1 or ROS 2) installed on your system.
- The `slam_toolbox` package installed (`sudo apt install ros-$ROS_DISTRO-slam-toolbox`).

Configuration

Key parameters in `slam_toolbox` are set using a YAML configuration file. The most important parameters include:

- **base_frame**: The robot frame name, typically `base_link`.
- **odom_frame**: Odometry frame, generally `odom`.
- **scan_topic**: Topic name for laser scans (`/scan` or `/scan_filtered`).
- **resolution**: Grid map resolution in meters (e.g., 0.05).
- **max_laser_range**: The maximum distance for processing scan points.
- **mode**: Can be `mapping` or `localization`, depending on required behavior .

Example (excerpt of `slam.yaml`):

```
slam_toolbox:
  ros__parameters:
    base_frame: base_link
    odom_frame: odom
    scan_topic: /scan
    resolution: 0.05
    max_laser_range: 16.0
    mode: mapping
```

Launching `slam_toolbox`

Launch files (ROS launch or Python files in ROS 2) are used to start `slam_toolbox`:

- **Mapping Mode:** Use `sync_slam_toolbox_node` or `async_slam_toolbox_node` executables with configuration file.
- **Localization Mode:** Use the `localization_slam_toolbox_node`, optionally with a previously saved map.

Example launch command (ROS 2):

```
ros2 launch slam_toolbox online_async_launch.py \
    use_sim_time:=false \
    slam_params_file:=/path/to/slam.yaml
```

Operation Workflow

1. **Start robot and sensors**: Ensure sensor and odometry data are being published.
2. **Launch `slam_toolbox`**: Start the node as above; this subscribes to `/scan` and odometry, and publishes the map and transforms.
3. **Drive the robot**: Move the robot through the environment to collect sufficient sensor data for mapping.
4. **Visualization**: Use `RViz` to visualize the generated map in real time.
5. **Map persistence**: To save the map, use the map saver utility:

   ```
   ros2 run nav2_map_server map_saver_cli -f ~/my_map
   ```

   This outputs map files (YAML and PGM/PNG) for later use in localization [?, ?, ?].

Brief Algorithmic Overview

`slam_toolbox` works by:

- Receiving laser scans and odometry data.
- Constructing a *pose graph* with node-to-scan associations.
- Refining odometry using scan matching and updating the pose graph.
- Detecting and closing loops for global consistency.
- Publishing the estimated robot pose and dynamic occupancy grid map in `nav_msgs/OccupancyGric` format.

Summary Table: Key Steps

Step	Description
Sensor Setup	Publish `/scan` and odometry
Config	Edit YAML with frame/topic/map parameters
Launch	Start appropriate node (mapping or localization)
Mapping	Move robot; build map in real time
Map Save/Load	Use map saver to store or reload maps
Visualization	Visualize with RViz/RViz2

Further Notes

- `slam_toolbox` supports both live robots and simulations (e.g., Gazebo).
- It is compatible with the Navigation2 stack for navigation tasks post-mapping.
- Advanced features include lifelong mapping, pose-graph editing, and more.

For detailed documentation and advanced configuration, refer to the official GitHub repository and ROS Wiki.

11 Additional Resources

- slam_toolbox - ROS Wiki
- slam_toolbox GitHub Repository
- SLAM Toolbox Mapping Tutorial for ARI Robot (ROS Wiki)
- **Nav2 Mapping with SLAM Toolbox** (YouTube, The Construct)
- 2D Mapping and Serialization for TurtleBot3 (YouTube, Nex-dynamics)
- How to install and use slam_toolbox (YouTube, ROS2 Q&A)
- Easy SLAM with ROS using slam_toolbox (YouTube)
- ROS2 SLAM Toolbox Tutorial Mobile Robot Simulation (YouTube)
- ROS2 Nav2 - Generate a Map with slam_toolbox (YouTube)
- Robotics, ROS, SLAM Toolbox Tutorials Playlist (YouTube)