LEONARDO TECHNICAL TRAINING

Robotic Software
Lezione 9

NVIDIA ISAAC ROS WITH ROS2
ROS2

Example 1.8

- Writing simple Publisher/Subscriber in C++
- Create a node able to send/get data via topics
  - $ ros2 pkg create --build-type ament_cmake cpp_pubsub
  - Add the source code
  - Modify the dependencies’ list from the package.xml file
  - Modify the CMakeLists.txt to allow the compilation of the package

Example 2.8

- Writing simple Publisher/Subscriber in python
- Create a node able to send/get data via topics
  - $ ros2 pkg create --build-type ament_python py_pubsub
  - Add the source code
    - Must be put in the py_pubsub/py_pubsub
  - Modify the dependencies' list from the package.xml file
    - <exec_depend>rclpy</exec_depend>
    - <exec_depend>std_msgs</exec_depend>
  - Modify the setup.py file:
    - entry_points={
    -         'console_scripts': [
    -             'talker = py_pubsub.publisher_node:main',
    -         ],
    -     },
  - Modify the CMakeLists.txt to allow the compilation of the package

ROS2 Commands

- All the commands used in ROS to handle the topics work similarly for ROS2
  - $ ros2 topic list
  - $ ros2 topic publisher
  - …
- Nodes can be inspected and handled via ROS2 node commands
  - $ ros2 node list
  - $ ros2 node info <node_name>
    - ros2 node info returns a list of subscribers, publishers, services, and actions (the ROS graph connections) that interact with that node

Example 1.9

- Create custom services
    - Create a new package
        - $ ros2 pkg create --build-type ament_cmake custom_msgs
    - Modify the CMakeLists.txt
        - find_package(rosidl_default_generators REQUIRED)
        - rosidl_generate_interfaces(${PROJECT_NAME}
            - "srv/AddTwoInts.srv"
        -)
    - Modify the package.xml
        - <export>
            - <build_type>ament_cmake</build_type>
            - <build_depend>rosidl_default_generators</build_depend>
            - <exec_depend>rosidl_default_runtime</exec_depend>
        - </export>
        - <member_of_group>rosidl_interface_packages</member_of_group>

Example 1.9

- Create custom services
    - Compile the custom services
        - $ cd ros2_ws
        - $ colcon build
        - $ source install/setup.bash
    - Check the compiled service
        - $ ros2 srv show custom_msgs/srv/AddTwoInts
        - The relative header file will be named as the message file name without the Camel notation
        - Example: AddTwoInts.srv -> add_two_ints.h

Example 2.9

- Create a new service using cpp
- We can add the dependencies in the package creation
  - $ ros2 pkg create --build-type ament_cmake cpp_srvcli --dependencies rclcpp custom_msgs
- After written the code we need to add the dependencies
  - add_executable(service_server src/srv_server.cpp)
  - ament_target_dependencies(service_server rclcpp custom_msgs)
  - add_executable(service_client src/srv_client.cpp)
  - ament_target_dependencies(service_client rclcpp custom_msgs)
  - install(TARGETS service_server service_client DESTINATION lib/${PROJECT_NAME})

Example 3.9

- Create a new service using python
- We can add the dependencies in the package creation
  - $ ros2 pkg create --build-type ament_python py_srvcli --dependencies rclpy example_interfaces
- Add the entry points
  - 'service = py_srvcli.service_member_function:main'
  - 'client = py_srvcli.client_member_function:main’
- Compile the package
  - $ colcon build --packages-select py_srvcli

Example 4.9

- Use the ROS2 parameter server
- Create a package
    - ros2 pkg create --build-type ament_cmake cpp_parameters --dependencies rclcpp
- Add the source code
- Run the source code
- Change the param
    - ros2 param set /parameter_node my_parameter earth

ROS2 Launch file

- The launch system in ROS 2 is responsible for helping the user describe the configuration of their system and then execute it as described
  - programs to run
  - where to run them
  - what arguments to pass
- It is also responsible for monitoring the state of the processes launched, and reporting and/or reacting to changes in the state of those processes.

- Launch files written are in Python and they can start and stop different nodes as well as trigger and act on various events
- The package providing this framework is launch_ros, which uses the non-ROS-specific launch framework underneath.
- One way to create launch files in ROS 2 is using a Python file
  - $ ros2 launch is the command responsible for starting launch files
- To use launch files, launch must be a dependency of our package
- the appropriate state.

ROS2 Launch file

- The typical usage in ROS2 is to have launch files invoked by ROS2 tools
- We need to compile the package after edited a proper launch file (very differently from ROS)
- After running colcon build and sourcing your workspace, you should be able to launch the launch file as follows:
  - $ ros2 launch <PACKAGE_NAME> <LAUNCH_FILE_NAME>
- In the launch file a Node object is created
  - Node (package, executable, output, parameters, …)

Example 5.9

- Create launch file for the Cpp parameter package
- Import the necessary python libraries
- Use the «generate_launch_description» function to generate the launch file structure
- Define a new node from:
  - Package: cpp_parameters
  - Node_executable: parameter_node
  - Output: screen
  - Prefix: stdbuf –o L   (this is used to force the flush of command line buffer)
  - Parameters: a new parameter value for: «my_parameter»

ROS2 bagfile

- ros2 bag is a command line tool for recording data published on topics in your system
- It accumulates the data passed on any number of topics and saves it in a database
- You can then replay the data to reproduce the results of your tests and experiments
- Recording topics is also a great way to share your work and allow others to recreate it
    - $ sudo apt-get install ros-dashing-ros2bag
    - $ sudo apt-get ros-dashing-rosbag2-converter-default-plugins
    - $ sudo apt-get ros-dashing-rosbag2-storage-default-plugins
- Record a bagfile
    - $ ros2 bag record <topic_name>
    - $ ros2 bag record -o <bagname> <topic-1> <topic-2>
- Get info from a bagfile
    - $ ros2 bag info <bagname>
- Play the bagfile
    - $ ros2 bag play <bagname>

ROS2 Components

- In ROS1 you can write your code as a ROS node
- ROS1 nodes are compiled into executables
- In ROS 2 the recommended way of writing your code is defining shared libraries called Components
- Components are very similar to the ROS2 nodes written so far
- This makes it easy to add common concepts to existing code, like a life cycle
    - Load/unload component
    - Share the component
    - …
- By making the process layout a deploy-time decision the user can choose between:
    - running multiple nodes in separate processes with the benefits of process/fault isolation as well as easier debugging of individual nodes
    - running multiple nodes in a single process with the lower overhead and optionally more efficient communication
    - ros2 launch tool can be used to automate these actions through specialized launch actions

ROS2 Components

- Since a component is only built into a shared library it doesn't have a main function
- A component is commonly a subclass of rclcpp::Node
  - Since it is not in control of the thread it shouldn't perform any long running or blocking tasks in its constructor
  - It can use timers to get periodic notification
  - It can create publishers, subscribers, servers, and clients.

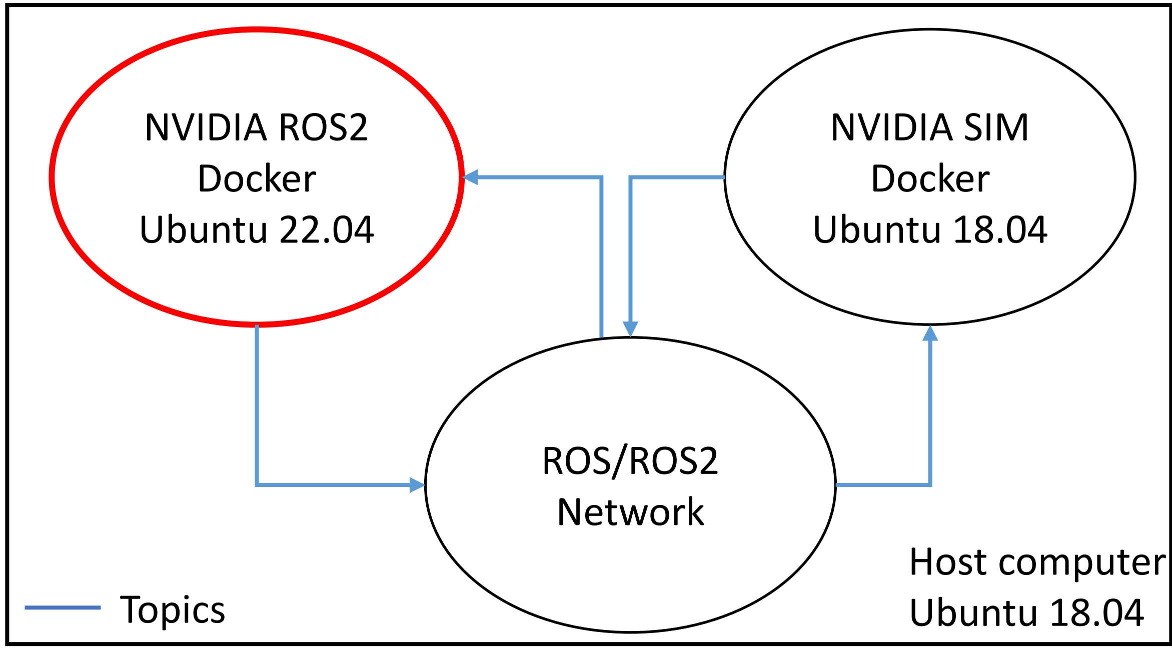
- An important aspect of making such a class a component is that the class registers itself using macros from the package rclcpp_components
- This makes the component discoverable when its library is being loaded into a running process - it acts as kind of an entry point.
- Once a component is created, it must be registered with the index to be discoverable by the tooling (the name of the component)

ROS2 Components

- To use the components, we must rely on the composition package.
- The composition package contains a couple of different approaches on how to use components
- The three most common ones are:
  1. Start a (generic container process) and call the ROS2 service load_node offered by the container
     - The ROS2 service will then load the component specified by the passed package name and library name and start executing it within the running process
     - Instead of calling the ROS2 service programmatically you can also use a command line tool to invoke the ROS2 service with the passed command line arguments
  2. Create a custom executable containing multiple nodes which are known at compile time
     - This approach requires that each component has a header file (which is not strictly needed for the first case).
  3. Create a launch file and use ROS2 launch to create a container process with multiple components loaded.

NVIDIA ISAAC ROS

- Install NVIDIA ROS docker
- Configure nvidia-container-runtime as the default runtime for Docker.
  - Install nvidia container
    - $ curl -s -L https://nvidia.github.io/nvidia-container-runtime/gpgkey | sudo apt-key add -
    - $ distribution=$(. /etc/os-release;echo IDVERSION_ID)
    - $ curl -s -L https://nvidia.github.io/nvidia-container-runtime/$distribution/nvidia-container-runtime.list | sudo tee /etc/apt/sources.list.d/nvidia-container-runtime.list
    - $ sudo apt-get update
    - $ sudo apt-get install nvidia-container-runtime

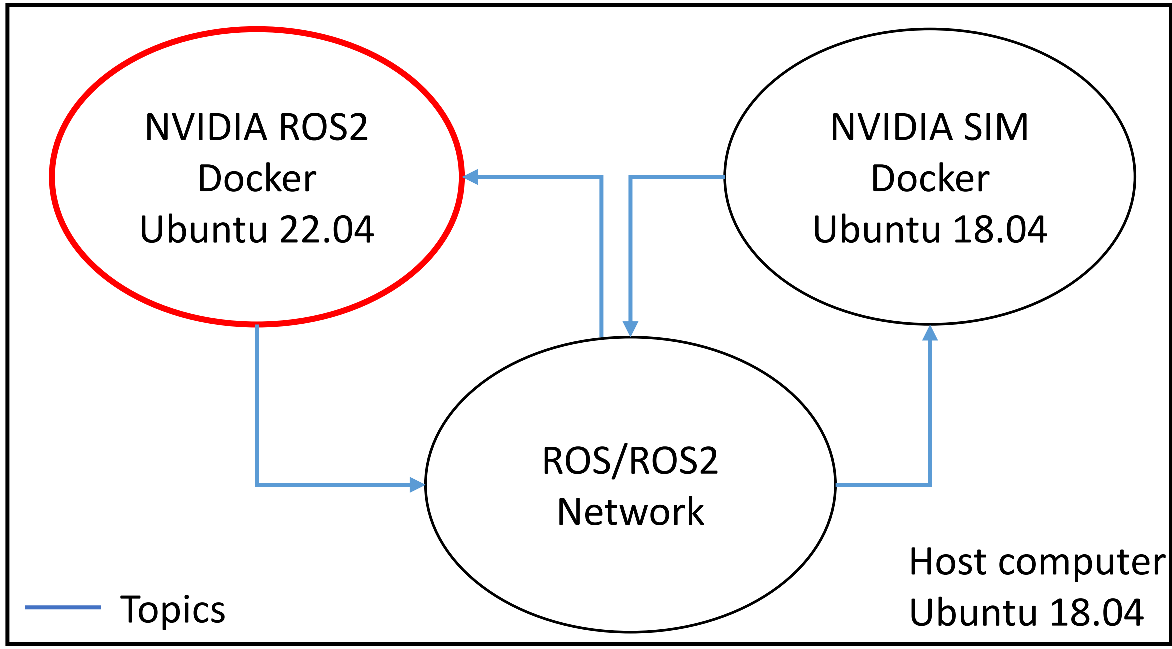


NVIDIA ISAAC ROS

- Install NVIDIA ROS docker
- Configure the docker
  - $ sudo touch /etc/docker/daemon.json
  - $ sudo gedit /etc/docker/daemon.json
  - Fill with:

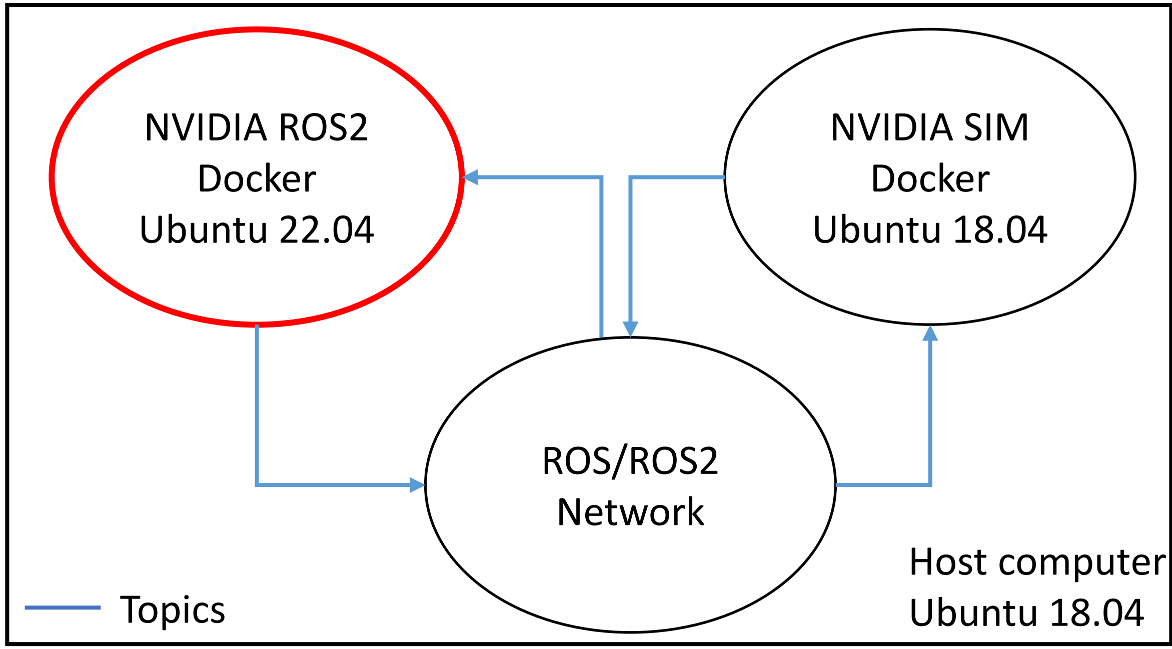
```
{
  "runtimes": {
    "nvidia": {
      "path": "nvidia-container-runtime",
      "runtimeArgs": []
    }
  },
  "default-runtime": "nvidia"
}
```

  - Install the Git Large File Storage to download large dimensions files
    - $ sudo apt-get install git-lfs



NVIDIA ISAAC ROS

- Create the ROS2 virtual workspace
  - $ mkdir -p ~/workspaces/isaac_ros-dev
  - $ cd ~/workspaces/isaac_ros-dev
- Clone the ISAAC ROS COMMON package
  - $ git clone https://github.com/NVIDIA-ISAAC-ROS/isaac_ros_common
- Run the docker
  - Set the correct permissions for docker commands
    - $ sudo usermod -aG docker $USER && newgrp docker
  - Start the running script…?



NVIDIA ROS

- NVIDIA ROS packages can be divided into
  - Utilities
  - GEMs
- Utilities
  - A set of packages supporting the execution of the ISAAC ROS and ROS nodes
- GEMs
  - The implementation of AI based and algebric based robotic applications
  - New GEMs will be released in the future
  - GEMs are fully implemented as ROS nodes

Isaac_ros_common

- The Isaac ROS Common repository contains a number of scripts and Dockerfiles to help streamline development and testing with the Isaac ROS suite
- The Docker images included in this package provide pre-compiled binaries for ROS2 Humble on Ubuntu 20.04 Focal
- Docker containers allow us to quickly setup a sensitive set of frameworks and dependencies to ensure a smooth experience with Isaac ROS packages
- Docker Development Scripts
  - run_dev.sh sets up a development environment with ROS2 installed and key versions of NVIDIA frameworks prepared for both x86_64 and Jetson
  - Running this script will prepare a Docker image with supported configuration for the host machine and deliver you into a bash prompt running inside the container
  - If you run this script again while it is running, it will attach a new shell to the same container.

Isaac_ros_common

- By default, the directory /workspaces/isaac_ros-dev in the container is mapped from ~/workspaces/isaac_ros-dev on the host machine if it exists, or the current working directory from where the script was invoked otherwise
- The host directory the container maps to can be explicitly set by running the script with the desired path as the first argument:
  - scripts/run_dev.sh <path to workspace>
- run_dev.sh prepares a base Docker image and mounts your target workspace into the running container
- Create the ROS2 virtual workspace
  - $ mkdir -p ~/workspaces/isaac_ros-dev/src
  - $ cd ~/workspaces/isaac_ros-dev/src

Isaac_ros_common

- Clone the ISAAC ROS COMMON package
  - $ git clone https://github.com/NVIDIA-ISAAC-ROS/isaac_ros_common
- Run the docker
  - Set the correct permissions for docker commands
    - $ sudo usermod -aG docker $USER && newgrp docker
  - Start the running script
    - $ cd ~/workspaces/isaac_ros-dev/src
    - $ touch TEST
    - $ cd ~/workspaces/isaac_ros-dev/src/isaac_ros_common/
    - $ ./scripts/run_dev.sh
    - This last step will take several minutes
  - Do the same in another terminal and see what's happen

Isaac_ros_nitros

- ROS2 Humble introduces new hardware-acceleration features, including type adaptation and type negotiation, that significantly increase performance for developers seeking to incorporate AI and machine learning and computer vision functionality into their ROS-based applications
  - Type adaptation is common for hardware accelerators, which require a different data format to deliver optimal performance
  - Type adaptation allows ROS nodes to work in a format better suited to the hardware
  - Processing pipelines can eliminate memory copies between the CPU and the memory accelerator using the adapted type
  - Unnecessary memory copies consume CPU compute, waste power, and slow down performance, especially as the image size increases.
  - Type negotiation allows different ROS nodes in a processing pipeline to advertise their supported types so that formats yielding ideal performance are chosen
  - The ROS framework performs this negotiation process and maintains compatibility with legacy nodes that don't support negotiation.

Isaac_ros_nitros

- Accelerating processing pipelines using type adaptation and negotiation makes the hardware accelerator zero-copy possible
- This reduces software/CPU overhead and unlocks the potential of the underlying hardware
- The NVIDIA implementation of type adaption and negotiation are called NITROS (NVIDIA Isaac Transport for ROS)
- ROS processing pipelines made up of NITROS-based Isaac ROS hardware accelerated modules (a.k.a. GEMs or Isaac ROS nodes) can deliver promising performance and results

Isaac_ros_nitros

- The design of NITROS makes the following assumptions of the ROS2 applications:
  - To leverage the benefit of zero-copy in NITROS, all NITROS-accelerated nodes must run in the same process.
  - For a given topic in which type negotiation takes place, there can only be one negotiating publisher.
  - For a NITROS-accelerated node, received-frame IDs are assumed to be constant throughout the runtime.
- Most Isaac ROS GEMs have been updated to be NITROS-accelerated.
- The acceleration is in effect between NITROS-accelerated nodes when two or more of them are connected next to each other
- In such a case, NITROS-accelerated nodes can discover each other through type negotiation and leverage type adaptation for data transmission automatically at runtime.
- NITROS-accelerated nodes are also compatible with non-NITROS nodes:
  - A NITROS-accelerated node can be used together with any existing, non-NITROS ROS2 node, and it will function like a typical ROS2 node.

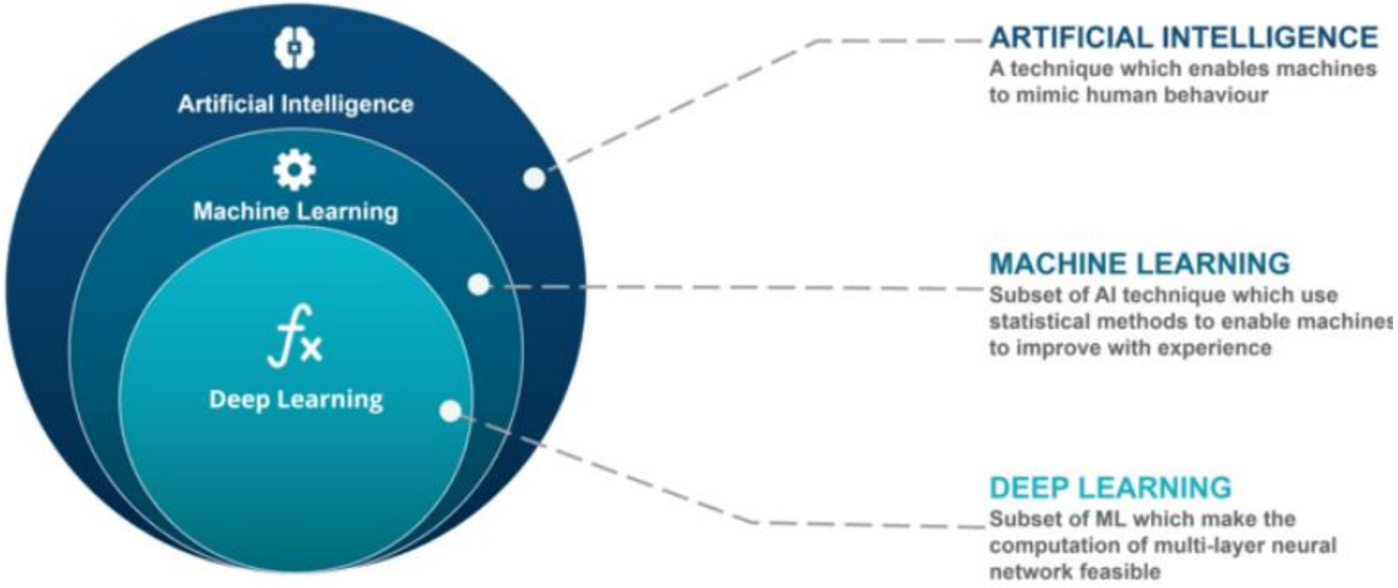
Isaac_ros_nitros

- NITROS supports transporting various common data types with zero-copy in its own NITROS types
- Each NITROS type is one-to-one-mapped to a ROS message type, which ensures compatibility with existing tools, workflows, and codebases
- A non-NITROS node supporting the corresponding ROS message types can publish data to or subscribe to data from a NITROS-accelerated node that supports the corresponding NITROS types.

NITROS Interface	ROS Interface
NitrosImage	sensor_msgs/Image
NitrosCameraInfo	sensor_msgs/CameraInfo
NitrosTensorList	isaac_ros_tensor_list_interfaces/TensorList
NitrosDisparityImage	stereo_msgs/DisparityImage
NitrosPointCloud	sensor_msgs/PointCloud2
NitrosAprilTagDetectionArray	isaac_ros_apriltag_interfaces/AprilTagDetectionArray

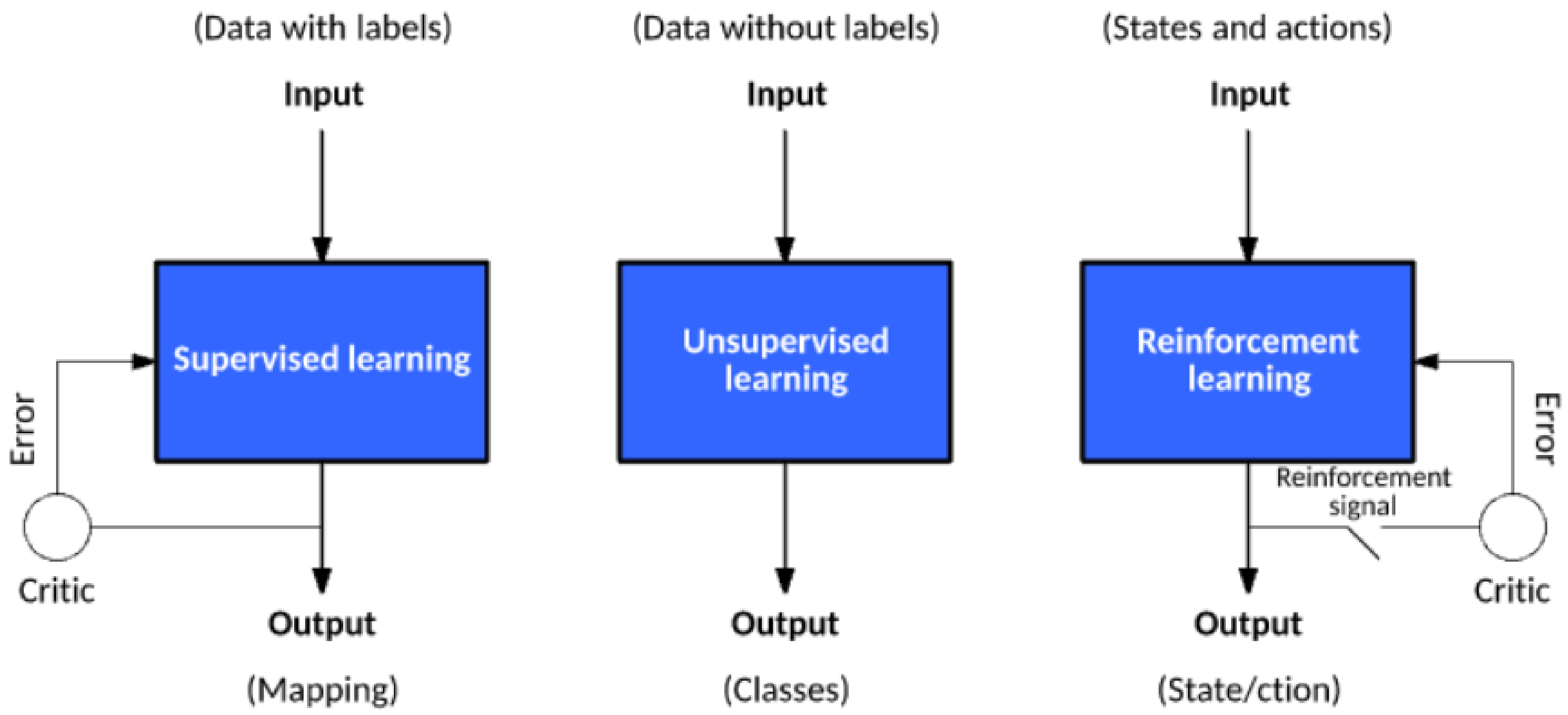
Deep Learning

- Deep learning is a subset of machine learning (ML), which is itself a subset of artificial intelligence (AI)
- The concept of AI has been around since the 1950s, with the goal of making computers able to think and reason in a way similar to humans
- ML is focused on how to the machines can learn without being explicitly programmed
- Deep learning goes beyond ML by creating more complex hierarchical models that are meant to mimic how humans learn new information
- Model
  - a mathematical algorithm that is trained to come to the same result or prediction that a human expert would when provided with the same information
  - In deep learning, the algorithms are inspired by the structure of the human brain and known as neural networks
  - Deep neural networks are built from interconnected network switches designed to learn to recognise patterns in the same way the human brain and nervous system does



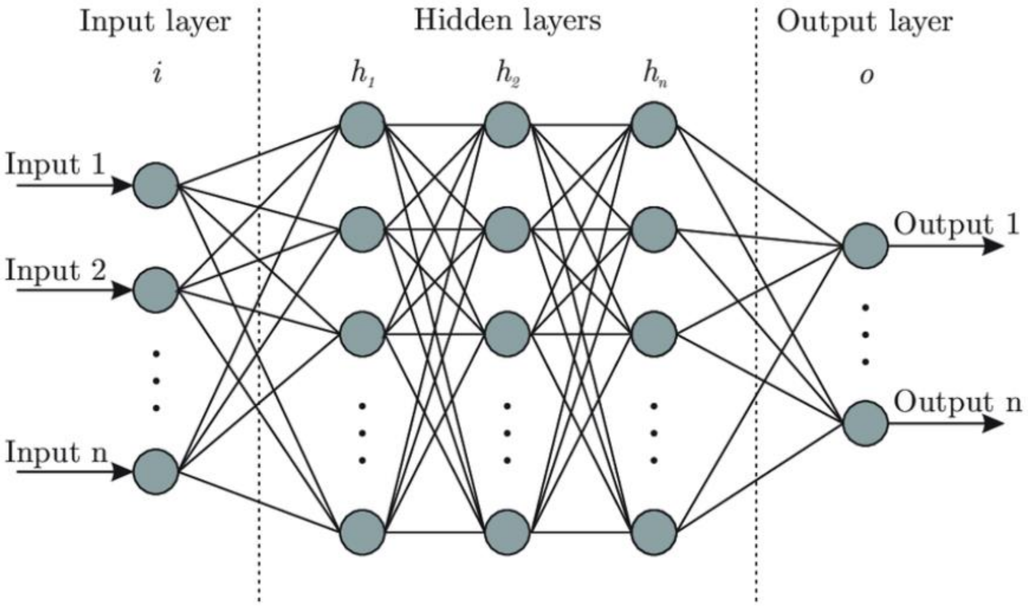
Deep Learning

- Staring from Machine Learning
  - Supervised Learning: Where we teach the machine what to learn
  - Unsupervised Learning: Where the machine will find what to learn
  - Reinforcement Learning: Where the machine learns from previous mistakes at every step



Deep Learning

- Neural Networks
    - Set of mathematical algorithms that work together to perform operations on the input. These operations then produce an output.
    - Neural networks can help us understand the relationships between complex data structures
    - The neural networks can use the trained knowledge to make predictions on the behavior of the complex structures
    - They can process images and even make complex decisions such as on how to drive a car, or which financial trade to execute next
- Feed-Forward NN:
    - Data are flowing in one direction only, from the input layer to the output layer
- Deep learning
    - Uses DNN: deep Neural Networks
        - A neural network with multiple hidden layers and multiple nodes in each hidden layer
        - Deep learning is the development of deep learning algorithms that can be used to train and predict output from complex data.
        - Deep: refers to the number of hidden layers



Deep learning data structures

- Tensors are a type of data structure used in linear algebra, and like vectors and matrices, you can calculate arithmetic operations with tensors
- Tensors are the building blocks of a machine learning model
  - For one type of machine learning, supervised learning, we provide large sets of training data for our machine learning models
  - We represent a single black and white image as a 3D Tensor of shape [width, height, color]
  - We represent a set of 1000 black and white image as a 4D Tensor of shape [sample_size, width, height, color].
    - For example, a set of 1000, 640 x 480 pixel black and white images would be represented as 4D Tensor of Shape [1000, 640, 480, 1].
- Tensors are the main information source used in Machine learning and AI applications based on vision or data analysis
- Tensors became well known to IT after google's machine learning framework tensorflow
  - They use tensors as the basic unit for calculation
  - Although the term is same, they are not entirely same
  - Tensors in programming are not the same as tensors in mathematics
  - They just inherit some of their qualities

Isaac_ros_dnn_interface

- The Isaac_ros_dnn_interface provides two NVIDIA GPU-accelerated ROS2 nodes that perform deep learning inference using custom models
- One node uses the TensorRT SDK, while the other uses the Triton SDK
- This repository also contains a node to preprocess images and convert them into tensors for use by TensorRT and Triton.
  - TensorRT is a library that enables faster inference on NVIDIA GPUs
  - TensorRT provides an API for the user to load and execute inference with their own models
  - The TensorRT ROS2 node in this package integrates with the TensorRT API, so the user has no need to make any calls to or directly use TensorRT SDK
  - Users simply configure the TensorRT node with their own custom models and parameters, and the node will make the necessary TensorRT API calls to load and execute the model
  - Triton is a framework that brings up a generic inference server that can be configured with a model repository, which is a collection of various types of models (e.g. ONNX Runtime, TensorRT Engine Plan, TensorFlow, PyTorch)
- Isaac ROS NITROS Acceleration
  - This package is powered by NVIDIA Isaac Transport for ROS (NITROS), which leverages type adaptation and negotiation to optimize message formats and dramatically accelerate communication between participating nodes.

Isaac_ros_image_pipeline

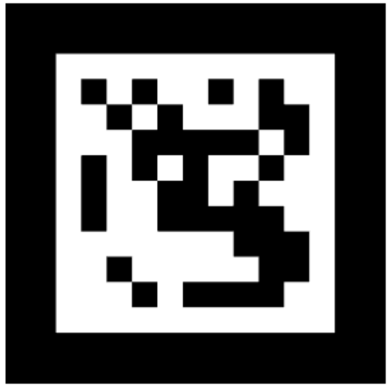
- The image_pipeline stack is designed to process raw camera images into useful inputs to vision algorithms: rectified mono/color images, stereo disparity images, and stereo point clouds
  - Calibration: Cameras must be calibrated in order to relate the images they produce to the three-dimensional world
  - Monocular processing: The raw image stream can be piped through the image_proc node to remove camera distortion
  - Stereo processing: The stereo_image_proc node performs the duties of image_proc for a pair of cameras co-calibrated for stereo vision
  - Depth processing: depth_image_proc provides nodelets for processing depth images
  - Visualization: The image_view package provides a lightweight alternative to rviz for viewing an image topic
- The Isaac_ros_image_pipeline offers similar functionality as the standard, CPU-based image_pipeline metapackage, but does so by leveraging NVIDIA GPUs and the Jetson platform's specialized computer vision hardware
- Considerable effort has been made to ensure that replacing image_pipeline with isaac_ros_image_pipeline on a Jetson device is as painless a transition as possible

ISAAC GEMs

- All the gems comes with an example bagfile
- They can be used also with Isaac sim
- They can be used with real hardware

AR Markers

- AR marker stands for:
    - Augmented reality markers
- Any object that can be placed in a scene to provide a fixed point of reference of position and/or scale
- With one camera we can not estimate the distance between the objects and the camera
    - We can solve this problem with triangulation procedures
- Defined patterns can be used to get multiple information
    - Object type – an id defines the marker
    - Position (3D)
    - Orientation (3D)


AR Markers

- Information
  - Where the robot is: which floor? Which room? (like QRCodes)
  - Where to push the button to open the elevator
- Elaboration is based on a binary classification of the squares of the marker
- Common problems:
  - Precision
  - Low-speed image elaboration
  - Marker detection
- Different software
  - Each software has its marker library

Isaac_ros_apriltag

- Install the apriltag NVIDIA packages
  - cd ~/workspaces/isaac_ros-dev/src
  - git clone https://github.com/NVIDIA-ISAAC-ROS/isaac_ros_apriltag
  - git clone https://github.com/NVIDIA-ISAAC-ROS/isaac_ros_image_pipeline
- Get the bagfile
  - cd ~/workspaces/isaac_ros-dev/src/isaac_ros_apriltag
  - git lfs pull -X "" -I "resources/rosbags/quickstart.bag“
- Compile the package
  - cd ~/workspaces/isaac_ros-dev/src/isaac_ros_common
  - ./scripts/run_dev.sh
  - cd /workspaces/isaac_ros-dev
  - colcon build
  - source install/setup.ba sh

Isaac_ros_apriltag

- Test the apriltag package
- Terminal 1: launch the apriltag package
  - $ cd ~/workspaces/isaac_ros-dev/src/isaac_ros_common
  - $ ./scripts/run_dev.sh
  - $ ros2 launch isaac_ros_apriltag isaac_ros_apriltag.launch.py
- Terminal 2: start the bagfile
  - $ cd ~/workspaces/isaac_ros-dev/src/isaac_ros_common
  - $./scripts/run_dev.sh
  - $ ros2 bag play --loop src/isaac_ros_apriltag/resources/rosbags/quickstart.bag
- Terminal 3: see the output
  - $ cd ~/workspaces/isaac_ros-dev/src/isaac_ros_common
  - $./scripts/run_dev.sh
  - $ ros2 topic echo /tag_detections
  - $ rviz2 -> check the tfs
- Terminal 4:
  - Use rqt:
    - rqt -> visualization -> image

Isaac_ros_apriltag

- Apriltag input

ROS Parameter	Type	Default	Description
`size`	`double`	`0.22`	The tag edge size in meters, assuming square markers. E.g. `0.22`
`max_tags`	`int`	`64`	The maximum number of tags to be detected. E.g. `64`

ROS Topic	Interface	Description
`image`	sensor_msgs/Image	The input camera stream.
`camera_info`	sensor_msgs/CameraInfo	The input camera intrinsics stream.

ROS Topic	Type	Description
`tag_detections`	isaac_ros_apriltag_interfaces/AprilTagDetectionArray	The detection message array.
`tf`	tf2_msgs/TFMessage	Pose of all detected apriltags(`TagFamily:ID`) wrt to the camera topic frame_id.

Isaac_ros_visual_slam

- SLAM:
  - Estimate the robot’s pose starting from odometry information and sensor information
- GMAPPING is a SLAM ROS1 package performing 2D slam from encoder odometry and laser scanner

Isaac_ros_visual_slam

- This repository provides a ROS2 package that performs stereo visual simultaneous localization and mapping (VSLAM)
- Estimates stereo visual inertial odometry using the Isaac Elbrus GPU-accelerated library
  - It takes in a time-synced pair of stereo images (grayscale) along with respective camera intrinsics to publish the current pose of the camera relative to its start pose
- Elbrus is based on two core technologies: Visual Odometry (VO) and Simultaneous Localization and Mapping (SLAM).
- Visual SLAM is a method for estimating a camera position relative to its start position
- This method has an iterative nature
  - At each iteration, it considers two consequential input frames (stereo pairs)
  - On both the frames, it finds a set of keypoints
  - Matching keypoints in these two sets gives the ability to estimate the transition and relative rotation of the camera between frames

Isaac_ros_visual_slam

- Simultaneous Localization and Mapping is a method built on top of the VO predictions
- It aims to improve the quality of VO estimations by leveraging the knowledge of previously seen parts of a trajectory
- It detects if the current scene was seen in the past and runs an additional optimization procedure to tune previously obtained poses.
- Along with visual data, Elbrus can optionally use Inertial Measurement Unit (IMU) measurements
- It automatically switches to IMU when VO is unable to estimate a pose; for example, when there is dark lighting or long solid featureless surfaces in front of a camera
- Elbrus allows for robust tracking in various environments and with different use cases: indoor, outdoor, aerial, HMD, automotive, and robotics.

Isaac_ros_visual_slam

- Download the packages
  - cd ~/workspaces/isaac_ros-dev/src
  - git clone https://github.com/NVIDIA-ISAAC-ROS/isaac_ros_visual_slam
- Get the bagfile
  - cd ~/workspaces/isaac_ros-dev/src/isaac_ros_visual_slam
  - git lfs pull -X "" -I isaac_ros_visual_slam/test/test_cases/rosbags/
- Compile the workspace
  - cd ~/workspaces/isaac_ros-dev/src/isaac_ros_common
  - ./scripts/run_dev.sh
  - cd /workspaces/isaac_ros-dev
  - colcon build
  - source install/setup.bash

Isaac_ros_visual_slam

- Start the nodes
- Terminal 1:
  - $ ros2 launch isaac_ros_visual_slam isaac_ros_visual_slam.launch.py
- Terminal 2:
  - $ source /workspaces/isaac_ros-dev/install/setup.bash
  - $ rviz2 -d src/isaac_ros_visual_slam/isaac_ros_visual_slam/rviz/default.cfg.rviz
- Terminal 3:
  - $ source /workspaces/isaac_ros-dev/install/setup.bash
  - $ ros2 bag play src/isaac_ros_visual_slam/isaac_ros_visual_slam/test/test_cases/rosbags/small_pol_test/
- Terminal 4:
  - $ ros2 topic echo visual_slam/tracking/vo_pose_covariance

Isaac_ros_pose_estimation

- This packages performs object pose estimation
- Starting from the model of an object, estimate its pose
  - Detect the object in the scene
  - Understand how it is placed
  - Estimate its pose in the 3D world

- This repository provides NVIDIA GPU-accelerated packages for 3D object pose estimation
- Using a deep learned pose estimation model and a monocular camera it can estimate the 6DOF pose of a target object
- We need also the DNN interface

- Download the packages
  - $ cd ~/workspaces/isaac_ros-dev/src
  - $ git clone https://github.com/NVIDIA-ISAAC-ROS/isaac_ros_pose_estimation
  - $ git clone https://github.com/NVIDIA-ISAAC-ROS/isaac_ros_dnn_inference

Isaac_ros_pose_estimation

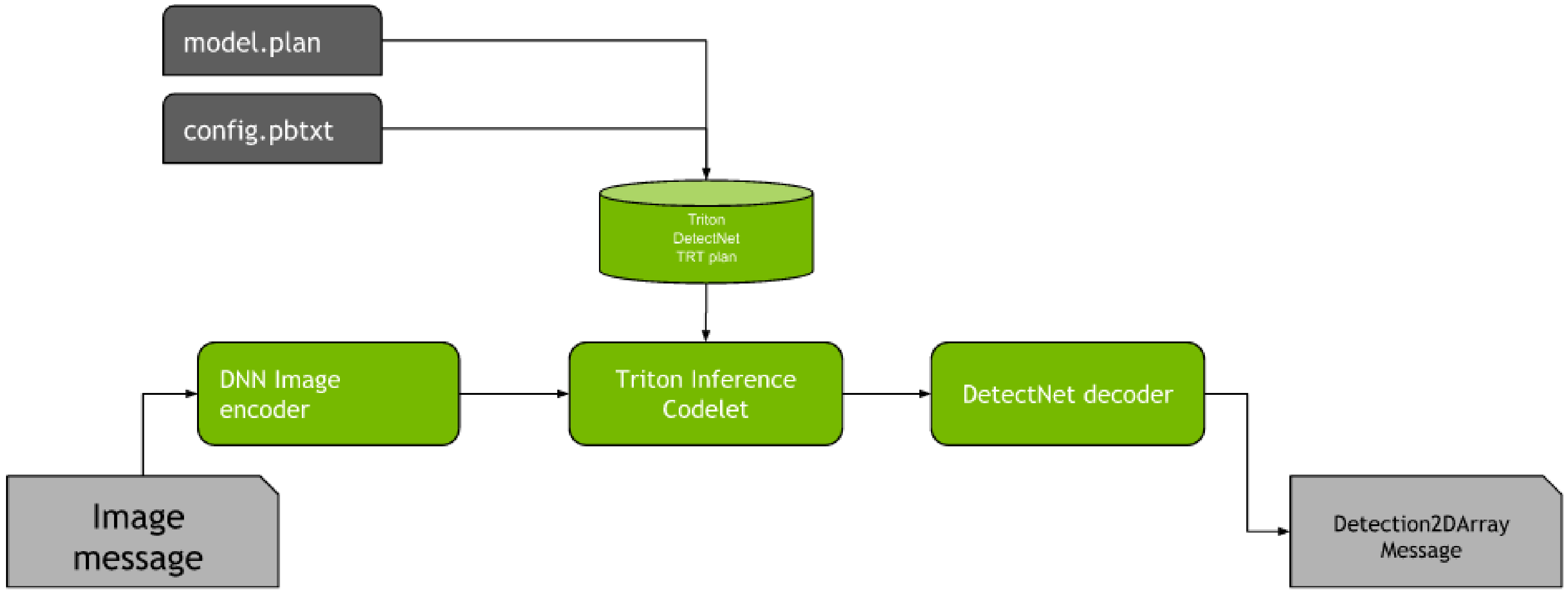
- Get the bagfile
  - $ cd ~/workspaces/isaac_ros-dev/src/isaac_ros_pose_estimation
  - $ git lfs pull -X "" -I "resources/rosbags/“
- Compile the workspace
  - $ cd ~/workspaces/isaac_ros-dev/src/isaac_ros_common
  - $ ./scripts/run_dev.sh
- Training the mode
  - mkdir -p /tmp/models/
  - cd ~/Downloads
  - Use the Ketchup.pth from the DOPE object repo (https://drive.google.com/drive/folders/1DfoA3m_Bm0fW8tOWXGVxi4ETlLEAgmcg)
- Copy the model into the docker container
  - docker cp Ketchup.pth isaac_ros_dev-x86_64-container:/tmp/models
- Convert the model
  - python3 /workspaces/isaac_ros-dev/src/isaac_ros_pose_estimation/isaac_ros_dope/scripts/dope_converter.py --format onnx --input /tmp/models/Ketchup.pth
- Compile the workspace
  - colcon build
  - source install/setup.bash

Isaac_ros_pose_estimation

- Run the package
- Terminal 1:
  - $ ros2 launch isaac_ros_dope isaac_ros_dope_tensor_rt.launch.py model_file_path:=/tmp/models/Ketchup.onnx engine_file_path:=/tmp/models/Ketchup.plan
- Terminal 2:
  - $ ros2 bag play -l src/isaac_ros_pose_estimation/resources/rosbags/dope_rosbag/
- Terminal 3:
  - $ ros2 topic echo /poses
  - $ rviz2
  - Then click on the Add button, select By topic and choose PoseArray under /poses
  - Finally, change the display to show an axes by updating Shape to be Axes, as shown in the screenshot below
  - Make sure to update the Fixed Frame to camera.

Isaac_ros_object_detection

- Object detection
  - This package provides a GPU-accelerated package for object detection based on DetectNet
  - Using a trained deep-learning model and a monocular camera, the isaac_ros_detectnet package can detect objects of interest in an image and provide bounding boxes.
  - DetectNet is similar to other popular object detection models such as YOLOV3



Isaac_ros_object_detection

- Get the package
  - $ cd ~/workspaces/isaac_ros-dev/src
  - $ git clone https://github.com/NVIDIA-ISAAC-ROS/isaac_ros_object_detection
- Get the bagfile
  - $ cd ~/workspaces/isaac_ros-dev/src/isaac_ros_object_detection/isaac_ros_detectnet
  - $ git lfs pull -X "" -I "resources/rosbags"
- Compile the workspace
  - $ cd ~/workspaces/isaac_ros-dev/src/isaac_ros_common
  - $ ./scripts/run_dev.sh
  - $ cd /workspaces/isaac_ros-dev
  - $ colcon build
  - $ source install/setup.bash

Isaac_ros_object_detection

- Train the model
  - $ cd /workspaces/isaac_ros-dev/src/isaac_ros_object_detection/isaac_ros_detectnet
  - $ ./scripts/setup_model.sh --height 632 --width 1200 --config-file resources/quickstart_config.pbtxt
- Launch the detector
  - $ ros2 launch isaac_ros_detectnet isaac_ros_detectnet_quickstart.launch.py
- Visualize and validate the output of the package in the rqt_image_view window. After about a minute, your output should look like this:

Isaac_ros_object_detection

- Input:
  - Params define the object to detect

`label_list`	`string[]`	`{"person", "bag", "face"}`	The list of labels. These are loaded from labels.txt(downloaded with the model)

ROS Topics Subscribed

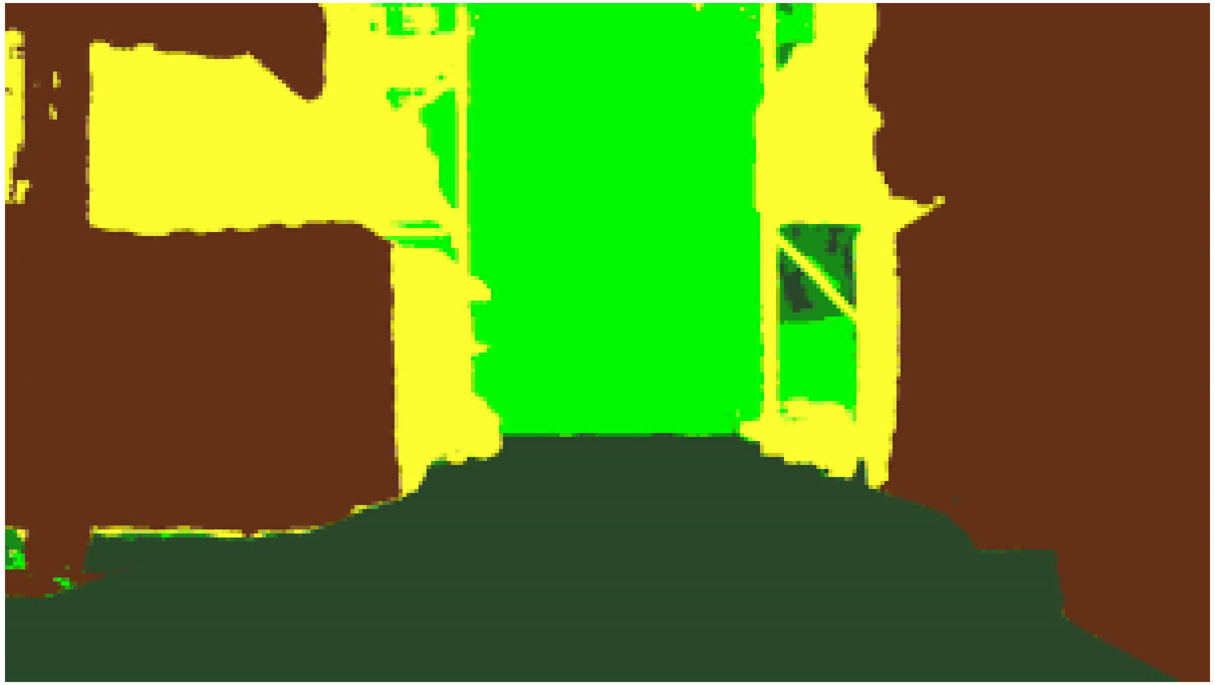
ROS Topic	Interface	Description
`tensor_sub`	isaac_ros_tensor_list_interfaces/TensorList	The tensor that represents the inferred aligned bounding boxes.

ROS Topics Published

ROS Topic	Interface	Description
`detectnet/detections`	vision_msgs/Detection2DArray	Aligned image bounding boxes with detection class.

Isaac_ros_proximity_segmentation

- Proximity segmentation predicts freespace from the ground plane
- It leverages this functionality to produce an occupancy grid that indicates freespace in the neighborhood of the robot
- This camera-based solution offers a number of appealing advantages over traditional 360 lidar occupancy scanning including better detection of low-profile obstacles



Isaac_ros_proximity_segmentation

- Get the package
  - $ cd ~/workspaces/isaac_ros-dev/src
  - $ git clone https://github.com/NVIDIA-ISAAC-ROS/isaac_ros_proximity_segmentation
- Get the bagfile
  - cd ~/workspaces/isaac_ros-dev/src/isaac_ros_proximity_segmentation
  - git lfs pull -X "" -I "resources/rosbags/bi3dnode_rosbag"
- Compile the workspace
  - cd ~/workspaces/isaac_ros-dev/src/isaac_ros_common
  - ./scripts/run_dev.sh
- Train the model
  - mkdir -p /tmp/models/bi3d
  - cd /tmp/models/bi3d
  - wget 'https://api.ngc.nvidia.com/v2/models/nvidia/isaac/bi3d_proximity_segmentation/versions/2.0.0/files/featnet.onnx’
  - wget 'https://api.ngc.nvidia.com/v2/models/nvidia/isaac/bi3d_proximity_segmentation/versions/2.0.0/files/segnet.onnx'

Isaac_ros_proximity_segmentation

  - $ /usr/src/tensorrt/bin/trtexec --saveEngine=/tmp/models/bi3d/bi3dnet_featnet.plan --onnx=/tmp/models/bi3d/featnet.onnx --int8
  - $ /usr/src/tensorrt/bin/trtexec --saveEngine=/tmp/models/bi3d/bi3dnet_segnet.plan --onnx=/tmp/models/bi3d/segnet.onnx --int8
- Compile the workspace
  - $ cd /workspaces/isaac_ros-dev
  - $ colcon build
  - $ source install/setup.bash
- Run the example
  - Terminal 1:
    - $ ros2 launch isaac_ros_bi3d_freespace isaac_ros_bi3d_freespace.launch.py featnet_engine_file_path:=/tmp/models/bi3d/bi3dnet_featnet.plan segnet_engine_file_path:=/tmp/models/bi3d/bi3dnet_segnet.plan max_disparity_values:=10
  - Terminal 2:
    - $ ros2 bag play --loop src/isaac_ros_proximity_segmentation/resources/rosbags/bi3dnode_rosbag
  - Terminal 3:
  - $ rviz2
    - In the left pane, click the Add button, then select By topic followed by Map to add the occupancy grid.

Isaac_ros_image_segmentation

- This repository provides NVIDIA GPU-accelerated packages for semantic image segmentation
- Using a deep learned U-Net model, such as PeopleSemSegnet, and a monocular camera, it can generate an image mask segmenting out objects of interest
- Get the package
  - $ cd ~/workspaces/isaac_ros-dev/src
  - $ git clone https://github.com/NVIDIA-ISAAC-ROS/isaac_ros_image_segmentation
- Get the bagfile:
  - $ cd ~/workspaces/isaac_ros-dev/src/isaac_ros_image_segmentation
  - git lfs pull -X "" -I "resources/rosbags/"
- Get and train the model:
  - cd ~/workspaces/isaac_ros-dev/src/isaac_ros_common
  - ./scripts/run_dev.sh
  - mkdir -p /tmp/models/peoplesemsegnet_shuffleseg/1
  - cd /tmp/models/peoplesemsegnet_shuffleseg
  - wget https://api.ngc.nvidia.com/v2/models/nvidia/tao/peoplesemsegnet/versions/deployable_shuffleseg_unet_v1.0/files/peoplesemsegnet_shuffleseg_etlt.etlt
  - wget https://api.ngc.nvidia.com/v2/models/nvidia/tao/peoplesemsegnet/versions/deployable_shuffleseg_unet_v1.0/files/peoplesemsegnet_shuffleseg_cache.txt

Isaac_ros_image_segmentation

- $ /opt/nvidia/tao/tao-converter -k tlt_encode -d 3,544,960 -p input_2:0,1x3x544x960,1x3x544x960,1x3x544x960 -t int8 -c peoplesemsegnet_shuffleseg_cache.txt -e /tmp/models/peoplesemsegnet_shuffleseg/1/model.plan -o argmax_1 peoplesemsegnet_shuffleseg_etlt.etlt
- $ cp /workspaces/isaac_ros-dev/src/isaac_ros_image_segmentation/resources/peoplesemsegnet_shuffleseg_config.pbtxt /tmp/models/peoplesemsegnet_shuffleseg/config.pbtxt

- Compile the workspace
  - $ cd /workspaces/isaac_ros-dev
  - $ colcon build
  - $ source install/setup.bash
- Launch the package
  - Terminal 1:
    - $ ros2 launch isaac_ros_unet isaac_ros_unet_triton.launch.py model_name:=peoplesemsegnet_shuffleseg model_repository_paths:=['/tmp/models'] input_binding_names:=['input_2:0'] output_binding_names:=['argmax_1'] network_output_type:='argmax’
  - Terminal 2:
    - $ ros2 bag play -l src/isaac_ros_image_segmentation/resources/rosbags/unet_sample_data/
  - Terminal 3:
  - $ ros2 run rqt_image_view rqt_image_view
  - Then inside the rqt_image_view GUI, change the topic to /unet/colored_segmentation_mask to view a colorized segmentation mask.