Hichem Dhouib
Seif Daknou
Feriel Amira

Human Behavior Recognition for Autonomous Robot Navigation

Project Report
IGNC Project Distributed Industrial Systems
M.Sc. Computer Engineering

submitted to

Technische Universität Berlin

Supervisor: M.Sc. Linh Kästner

Industry Grade Networks and Clouds

April 12, 2022

Abstract

Human action recognition is an important research area in the field of computer vision that can be applied in surveillance, assisted living, and robotic systems. It heavely relies on machine learning. Deep Learning has emerged as a promising framework for solving action recognition tasks. As a result, several field experts have adopted deep learning approaches to solve these problems and achieve state-of-the-art performance. This project contains a dockerized pipeline of a spatio-temporal human action recognition on a GPU. A pre-trained model from the ZED2 camera is downloaded for the human detection and tracking and a pre-trained model by Mmaction2 is used to detect and categorize human actions. Finally, the resulting dynamic human action recognition module is integrated into the ROS ecosystem, dockerized and deployed on a Jeton Nvidia Xavier AGX.

Kurzfassung

Die Erkennung menschlicher Handlungen ist ein wichtiges Forschungsgebiet auf dem Gebiet des Computerbildes, das bei der Überwachung, beim betreuten Wohnen und bei Robotersystemen eingesetzt werden kann. Sie stützt sich in hohem Maße auf maschinelles Lernen. Deep Larning hat sich als vielversprechender Rahmen für die Lösung von Aufgaben der Handlungserkennung erwiesen. Infolgedessen haben mehrere Bereiche und Experten Deep-Learning-Ansätze übernommen, um diese Probleme zu lösen und Spitzenleistungen zu erzielen. Dieses Projekt enthält eine angedockte Pipeline für die räumlich-zeitliche Erkennung menschlicher Handlungen auf einer GPU. Für die Erkennung und Verfolgung von Menschen wird ein vortrainiertes Modell von der ZED2-Kamera heruntergeladen, und das vortrainierte Modell von Mmaction2 wird zur Erkennung der Aktionen verwendet. Schließlich wird das resultierende dynamische Modul zur Erkennung menschlicher Handlungen in das ROS-Ökosystem integriert, angedockt und auf einem Jeton Nvidia Xavier AGX eingesetzt.

Contents

List of Abbreviations

DL	Deep Learning
NN	Neural Network
CNN	Convolutional Neural Network
ROS	Robot Operating System
IGNC	Industry Grade Networks and Clouds
RVIZ	ros visualization

List of Figures

1 Introduction

1.1 Motivation

Human action recognition is an active research area in computer vision that widely impacts applications in robotics. Despite the progress of visual recognition in the past decade, performance on action recognition remains relatively low. The main reason being that past methods focus on architectures for recognizing visual appearance [1], whereas action recognition often requires reasoning about the actor's relationship with the scene, e.g. with objects and other actors, both spatially and temporally.

In this paper, we propose a human action recognition model that learns spatiotemporal relationships as given by [1]. We will deploy MMaction2's pre-trained models and then integrate the module in a mobile robot platform including an NVIDIA Jetson XAVIER embedded board and a ZED2 camera to monitor the scene.

2 Fundamentals

This section will present some of the theoretical concepts and fundamentals this project is based on. A brief introduction of deep learning as well as Convolutional Neural networks, the object detection module, the action recognition algorithm and the depth calculation process as well as other used technologies such as Robot Operating System (ROS) ecosystem, and Docker.

2.1 Machine Learning

Arthur Samuel first promoted the term Machine Learning (ML) in 1959. It is based on probability theory, statistics and optimization. He described it as a field that equips computers, similarly to humans, with the ability to learn without being explicitly programmed. In other words, ML algorithms aim at learning to solve complex tasks based on prior experiences [7]. Machine learning algorithms represent the basis for big data, data science, predictive modeling, and have potential in a wide variety of real-world applications such as game theory, robotics4, finance5, and health-care. For example, ML was recently used during COVID-19 pandemic to identify patients at high risk [3] [8]. The field is mainly sub-divided into three types: supervised, unsupervised, and reinforcement learning schemes. All three types are concerned with a learning agent, which is software programs that make intelligent decisions. In supervised learning, each input feature is associated with a target variable, and the goal is for the agent to learn a function that maps a relationship between the input features and the targets. In unsupervised learning, the dataset has no been given targets for the input features, so the agents attempt to determine the unknown structure of the dataset by grouping similar samples together. Reinforcement learning agents take sequences of actions in an environment in order to maximize the notion of cumulative reward [9]. In this project, we will focus our attention on reinforcement learning, which will be further elaborated.

2.2 Deep neural Network

Deep learning is a particular machine learning method used in reinforcement learning, usually as a function approximator. It utilizes hierarchical levels of

artificial neural networks inspired by the structure of the human brain, with neuron nodes connected to each other like a web. The simplified structure of these networks is a sequence of one input layer, one or more hidden layers, and one output layer. Each layer is composed of a number of artificial neurons, as seen in the figure below. The input layer represents the input in the form of a column vector. The hidden layers transform these values through or activation function, such as the hyperbolic tangent alias *tanh*, or rectified linear unit to obtain a new representation. The output layer contains the output values. The links between nodes from layer to layer are weighted, which means that at each layer except the first one, the input to each node is computed as the weighted sum of nodes from the previous layer [10]. After computations flow forward from input to output layer, the error derivatives are computed and the gradients propagate back towards the input layer, which updates the weights to optimize some loss function [11].

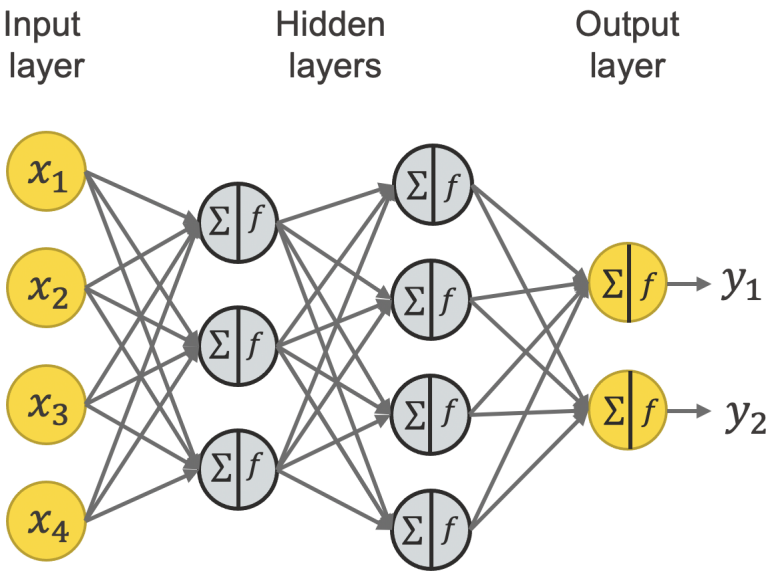


Figure 2.1: Example of Deep neural Network with multiple hidden layers

2.3 Convolutional Neural Network

A convolutional neural network, also called CNN, is an artificial neural network. While in primitive methods, the filters responsible for extracting the features from the pictures are created manually, convolutional neural networks learn to extract complex and simple features when trained long enough and using a rich dataset. The convolutional neural network can successfully capture spatial and temporal dependencies in the image by

applying appropriate filters. CNNs can be trained to learn and extract various features and understand the complexity of the image. In other words, a CNN has the capability of reducing images into an easier form to process without losing spatial and contextual features, which are critical for getting a good prediction. A CNN accepts the input frame as matrix and applies several processing steps to extract and learn complex features. The neural network is typically composed of three types layers: convolutional layers, pooling layers, and fully connected layers. Typically, the first convolution layer captures the low-level features such as edges, color, and gradient orientation. Deeper convolution layers, can extract more complex high-level features, like faces and compound shapes. The network can then be trained to learn to differentiate between the various elements of the frames and learn to classify and detect the objects correctly. Each convolutional layer executes a convolution operation by a kernel or filter. The kernel hovers over the frame and with each movement, it performs a dot product. The convolution operation is a linear operation, to introduce non-linearity and capture complex features from the input images, the Relu is used. The output is featured matrix, which is then fed to the subsequent layers of the network.

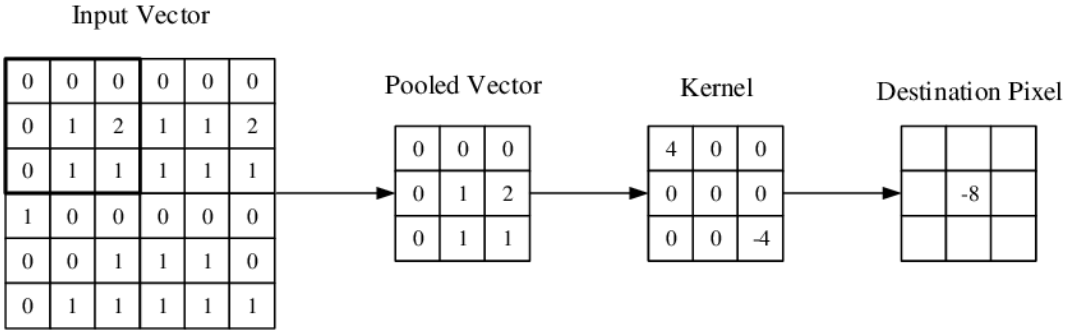


Figure 2.2: CNN Architecture, source: [17]

In the same way as the convolutional layer, the pooling layer is responsible for reducing the spatial extent of the convolved features. The purpose of this is to reduce the computational power required to process the data through dimensionality reduction. In addition, it helps extract dominant features that are rotationally and positionally invariant. Fully connected layers are based on the idea that every neuron or perceptron from the output coming from the previous layer is connected to every neuron of the next layer. Learning non-linear combinations of the high and low-level features represented by the output of a convolutional layer is done by introducing a fully connected layer into our network after flattening the output fed by the previous convolution layers. For every iteration during training,

backpropagation is applied over a series of epochs to adjust the weights to correct the model. The classification is then produced on the output of the last layer using SoftMax is the classes are mutually exclusive, and logistic regression, if they are not . [9]

2.4 Spatio-temporal Action Detection

Action recognition mainly focuses on classifying actions in short trimmed clips. When addressing long untrimmed videos, as webcam streams with multiple actors, temporal and spatial localization becomes a key point in action detection. Spatio-temporal action detection, assigns different labels to different actors in the same scene. For example, consider a scene with two actors A and B, the first action is sitting down and the second is standing up. In order to fully understand the scene, one must localize both actions to know which of A and B is executing it. Furthermore, incorporating a temporal context from multiple frames would disambiguate the actions of standing up and sitting down, which may appear identical at the single frame level. Spatio-temporal action detection has two key components: actor localization and action classification, which will be discussed in the upcoming sections. For the action detection, an actor-centric relation network (ACRN) is used. As seen on Figure 3, actorcentric relation features are extracted and inputted to the action classification network. The approach performs relation reasoning given only action annotations, and automatically retrieves the regions that are most related to the action. [9] [2] Spatio-temporal action recognition suffers the same challenges as other action recognition algorithms, e.g tracking the action throughout the frames, localizing the time frame of the action occurrence. Furthermore, it sets additional challenges: background clutter, object occlusion, high spatial complexity with a rising number of proposed actors,etc. The problem is not linear, as detection algorithms usually require region proposals to later on classify the action. Furthermore, accounting for temporality would cause an exponential increase in the number of proposals which would render any such approach impractical for use. [9]

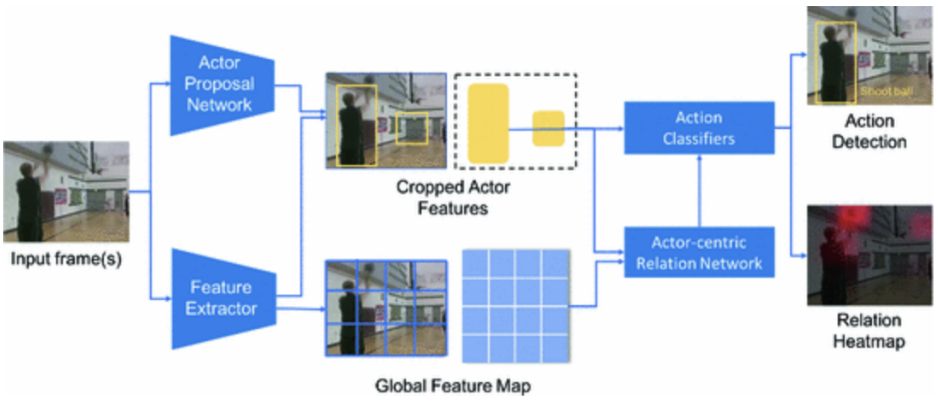


Figure 2.3: Overview of the action detection framework [2] , source: [17]

2.5 Action Recognition

To use actor-centric relations for action detection, we need to define actors, objects and their relations. The actors are proposed by the actor localization model. To avoid generating object proposals, the objects are represented by individual feature cells in the convolutional feature map, as seen in Figure 6. Both the feature extractor function as well as aggregation of features from all pair-wise relations can be implemented with standard convolutions and pooling operations. Since spatiotemporal orientations are not equally likely, spatial structures and temporal events should be treated separately. Thus, a SlowFast model, as given by [12] is used. A SlowFast network is single stream architecture with two pathways with different temporal speeds: a Slow pathway and Fast pathway. The two pathways are fused by lateral connections.

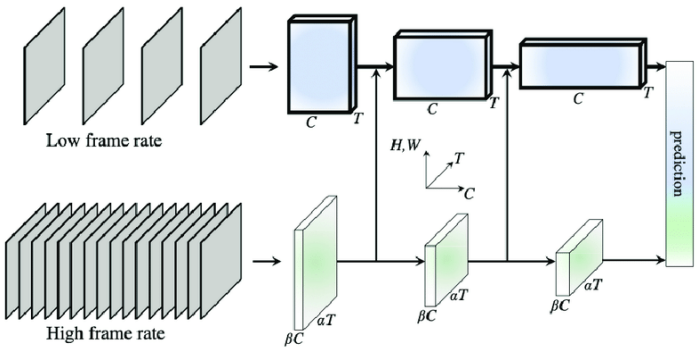


Figure 2.4: Slowfast network for video recognition, source: [12]

The Slow pathway is a convolutional model; namely a temporally strided 3D ResNet. It handles semantic information that can be given by images or a few sparse frames, and it operates at low frame rates. To achieve this, a large temporal stride τ is used on the input frames i.e., it processes only one out of τ frames. The Fast pathway is also a 3D ResNet. It is responsible for capturing rapidly changing motion. It has a fast refreshing speed and high temporal resolution. The pathway is made lightweight through a low channel capacity. The logic being that this pathway is designed to weaker ability to process spatial information, since those are handled by the Slow pathway. Let $\alpha > 1$, be the frame rate ratio between the Fast and Slow pathways, then the Fast pathway works with a temporal stride of τ/α. At every stage, the information of the two pathways is fused through lateral connections, which are a popular technique for merging different levels of spatial resolution and semantics. Since, the pathways have different temporal resolutions, the lateral connections reshapes and transposes the feature of the Fast pathway to match the slow one. A global average pooling is performed on each pathway's output. Finally, two pooled feature vectors are concatenated as the input to the fully-connected classifier layer.

2.6 AVA Dataset

The AVA dataset [10] focuses on spatiotemporal localization of human activity. The data is is taken from the 15th to 30th minute time intervals of 437 different movies. Spatiotemporal labels are provided for one frame per second, with every person annotated with a bounding box and (exhaustively many) actions. The vocabulary consists of 80 different atomic visual actions. The action could correspond to the actor's pose (i.e standing, sitting, etc.), interactions with objects (i.e carrying, riding, etc.) and interactions with other persons (i.e talking to, listening to, etc.) [10]

2.7 Robot Operating System (ROS)

The Robot Operating System [18] contains a collection of libraries and packages that help structure and organize robot-oriented software and applications. It is an open-source project that permits the integration of different software and algorithms. ROS offers visualization and simulation tools. The core concept behind ROS is the built-in message-passing infrastructure, which acts as a middleware between different software and hardware components. This well-tested communication system permits a smooth information exchange. ROS allows the developer to focus on implementing the desired functionality rather than spending time trying to create these different means of communication. In addition, creating isolated software

applications and functionalities results in systems that can be easily maintained and reused. The ZED2 Camera provides a ZED ROS wrapper. It is used in this project to integrate the Camera into the ROS ecosystem. The data will be retrieved from and published to ROS.

The following ROS concepts and structures are required to understand the architecture of the ROS workspace. First, a base unit in ROS is called a node. Nodes are in charge of running the computations and functionalities. Each node is responsible for one task. In addition, Nodes can communicate with each other using topics or services. In ROS, a topic is a data stream used to exchange information between nodes. They are used to exchange messages of different types, e.g., a sensor readout, where each topic is registered under a unique name and with a defined message type. Nodes can connect with it to publish or subscribe to them to retrieve messages. For a given topic, one node can not publish and subscribe to it at the same time, but there are no restrictions in the number of different nodes publishing or subscribing [15]. Roslaunch is a tool for quickly launching multiple ROS nodes locally and remotely via SSH. Besides launching multiple ROS nodes, it is also used for setting parameters on the Parameter Server. Roslaunch files takes in one or more XML configuration files (with the .launch extension) that specify the parameters to set and nodes to launch, as well as the hosting machines. In addition, it includes several options to control the execution flow. For example, there is an option to automatically respawn processes that have already died [20].

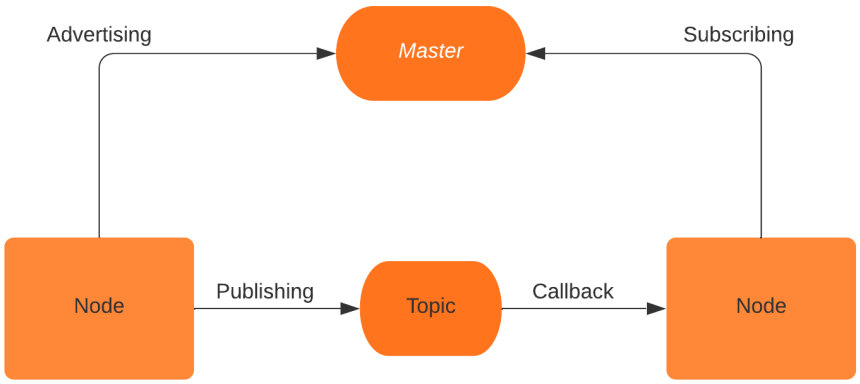


Figure 2.5: ROS concept

2.8 Docker

Docker [13] is a platform for running software applications in containers. It is a deployment tool used by developers to focus more on the development of the software rather than the deployment. They increase the flexibility and portability of applications. In other words, it accelerates the deployment of software applications by packaging an application and its dependencies into an isolated environment called a docker container. This container can run reliably and predictably on various platforms, such as servers, cloud providers, or local machines. It is important to note that Docker is not a virtualization tool but a way to containerize or encapsulate software. Containers rely on some part of the host machine to run [7]. Docker is written in Go programming language and takes advantage of several features of the Linux kernel to deliver its functionality. E.g., control croups (cgroups) are a feature of the Linux kernel that allow you to limit the access different processes and containers have to system resources such as CPU, RAM, disk input and ouput and network. Cgroups are a key component when multiple processes are running in a container [16]. Docker uses a technology called namespaces to provide the isolated workspace called the container. A set of namespace is automatically created when a container is run. These namespaces provide a layer of isolation. Each aspect of a container runs in a separate namespace and its access is limited to that namespace [8]. Dockerfiles are used to sequentially define the content of each layer of the docker image. The final docker image is then used to start or create a docker container. The figure 2.6 visualizes the docker container creation process.

Figure 2.6: Docker Structure Overview

3 hardware

This section introduces the hardware used for the implementation. First, the ZED2 stereo camera is presented. Then the Jetson Nvidia Xavier AGX board used for the deployment is described.

3.1 ZED 2 camera

The camera has two wide-angle dual lenses with optically corrected distortion, an aperture of $f/1.8$ and a 120° field of view. It has many built-in sensors such as an IMU, barometer, accelerometer, gyroscope, and magnetometer with a data rate ranging from 25Hz to 50Hz. Moreover, the camera can deliver between 15 and 30 frames per second at a resolution of 3840x1080. It includes depth calculation thanks to the neural stereo depth-sensing technology with an FPS up to 100 Hz. The ZED SDK provides a custom detector that can be used to detect and track humans, animals and objects. The 2D detection is processed and combined with the 2D or 3D information such as position and a 2D or 3D bounding box to deliver accurate and fast human tracking performance. The detection range is up to 20m for 3D and up to 40m for 2D. It is also able to detect and track a human skeleton with 18x body key points with up to 15m range. If the positional tracking module is activated, the ZED SDK can track the objects within the environment with an ID. The SDK computes a 2D mask that indicates which left image pixels belong to the object. From there, the ZED can output the 2D bounding boxes of the objects and accurately compute the 3D bounding boxes with the help of a depth map. The bounding boxes of the objects can be displayed in RVIZ using the 2D image input or a 3D point cloud [2]. The ZED SDK offers several pre-implemented and pre-trained human detection models based on PyTorch and TensorFlow. Furthermore, The camera processes the computations locally which makes it a good fit for several robotic use cases [22].

Figure 3.1: ZED 2 camera, source: [22]

3.2 Jetson Nvidia Xavier AGX

The ZED2 camera requires a processing unit to run the integrated computations for the AI networks. In this project, the NVIDIA Jetson AGX Xavier developer kit 3.2 is used to deploy the end-to-end AI robotics software. It supports the NVIDIA Jetpack and DeepStream SDKs, as well as CUDA®, cuDNN, and TensorRT software libraries. The NVIDIA Xavier processor powers it. [11]

Figure 3.2: Jetson Xavier, source: [12]

4 Conceptual Design

This chapter will present the conceptual design of the software. The section starts by describing the goal of this project, the required components and modules as well as their interactions is shown in various design flowcharts. The goal of this project is to implement a deep learning based action recognition algorithm with RGBD data, integrate the tracking software into the ROS ecosystem and deploy it on a Jetson Nvidia Xavier board. For these purposes, we will be using the provided human tracking service of the ZED2 camera. A trained neural network will be used to recognize actions in 2D. The human detection results are then combined with the predicted labels provided by the camera sensor to construct 3D coordinates for detected humans and their respective action recognition results in 3D.

4.1 Action Recognition Algorithm

The ActionDetector class must complete the following actions:

1. The stereo camera publishes the sensor data to ROS topics with an approximately constant average rate.
2. The ActionDetector script subscribes to the data topics to retrieve the RGB values of each pixel of the input picture.
3. The ActionDetector script subscribes to the results of the human detections provided by the pre-implemented ZED SDK service.
4. The ActionDetector script converts the RGB frames, the 2D bounding boxes of the detected humans to the required tensor format of the inference
5. The ActionDetector script builds 3D bounding box to visualize the detected humans and the predicted action recognition label results.
6. A marker array containing a list of the 3D bounding boxes and predicted actions is then published to RVIZ with a constant average rate for visualization.
7. Each time a new marker array is published, it updates the previously published markers.

4.2 Human tracking - ZED2 Camera

The object detection service provided by the ZED2 camera can be used in 2D or 3D. It can identify and track objects present in images and videos thanks to depth sensing and 3D information. The ZED SDK provides a custom detector that can be used to detect and track humans. The 2D detection is processed and combined with the 3D information such as position and 3D bounding box to deliver accurate and fast human tracking performance. The ZED SDK uses AI and neural networks to determine which objects are present in the left and right images. This pre-implemented service uses PyTorch and TensorFlow to run deep learning pre-trained models to detect humans. It computes the 3D position of each human present in the frame and produces a surrounding 3D bounding box, using data from the depth module. If the positional tracking module is activated, the ZED SDK can track the objects within the environment. This means that the detected object will keep the same ID throughout the sequence, displaying the object's path over time. The distance of the object from the camera is expressed in meters. The SDK computes a 2D mask that indicates which left image pixels belong to the object. From there, the ZED can output the 2D bounding boxes of the objects and accurately compute the 3D bounding boxes with the help of the depth map. The bounding boxes of the objects can be displayed in RVIZ using the 2D image input or a 3D point cloud [1]. The detection model that enables human body detection can be chosen from various object detection models with different performances:

1. fast: for a real-time performance even on Jetson or low-performance GPU cards but does not deliver the best performance.
2. medium: which is a compromise between accuracy and speed.
3. accurate: represents the state of the art accuracy but requires powerful GPU resources.

4.3 Integration into the ROS ecosystem

The Robot Operating System is collection of open-source libraries and packages that help organize and integrate different robot-oriented software and applications. ROS offers visualization and simulation tools. The system heavily relies on a built-in message-passing infrastructure, which acts as a middleware between different software and hardware components. This well-tested communication system permits a smooth information exchange. ROS allows developers to create isolated software applications and functionalities resulting in systems that can be easily maintained and reused. In order understand the architecture of the ROS workspace, one needs to know

the following concepts: First, a base unit in ROS is called a node. Each node is responsible for running the computations and functionalities of one task. These nodes are combined into a graph. In ROS, a topic is a data stream used to exchange information e.g., a sensor readout, between nodes. Each topic is registered under a unique identifier and with a defined message type. Nodes can connect with it to publish or subscribe to them to retrieve messages. There are no restrictions in the number of different nodes publishing or subscribing to a given topic, however one nodes can only publish and subscribe sequentially. Another important concept is Roslaunch, which is a tool for launching multiple ROS nodes locally or remotely via SSH as well as setting initialization requirements such as initial parameters. Roslaunch files takes in one or more XML configuration files with the .launch extension that specify the parameters to set and the nodes to launch. In addition, it includes several options to control the execution flow. The ZED2 Camera provides a ZED ROS wrapper for integration into the ROS ecosystem. The data is retrieved from the camera and published to ROS: The human detection results and the RGB data are published continuously with a constant frequency on pre-specified topics. The ActionDetector node subscribes to these topics to retrieve and pre-process the data before passing the RGB frames as well as the human detection results in the form of tensors to the inference detector. The 3D coordinates of the 3D bounding boxes as well as the predicted actions are then used to create ROS marker objects, appended to a ROS markers array, and published to the ROS topics. The 2D detection results are used to draw a 2D bounding box on the RGB input image. The processed RGB image, which contains the predicted 2D bounding boxes, is published to a specific ROS topic. The bounding boxes can be automatically visualized in RVIZ. The following Figure represents how the ROS workspace is built on top of the docker container and the three main ROS nodes that constitute the software workspace.

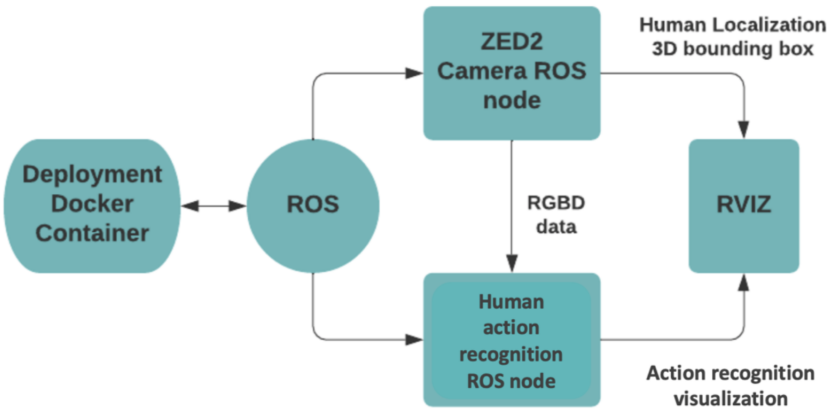


Figure 4.1: Action Recognition Software Flowchart

5 Implementation

This section provides an overview of the different modules and their interconnections. It aims at demonstrating the implementation and logic of the software. First, we will start with the human detection in 2D and 3D then moving on to the action detection and recognition algorithm followed by the software deployment with Docker and finally the integration into the ROS ecosystem.

5.1 Bounding boxes of detected and localized humans

The object detection service provided by the ZED2 camera can be used both in 2D and 3D. It is capable of identifying and tracking objects using its depth-sensing module. Moreover, the ZED SDK provides a custom detector specifically adapted and used to detect in 2D and localize and track humans amongst other objects in 3D. The 2D detection process takes as an input the RGB frames published by the camera at a high frame rate on the following ROS topic: "/zed2/zed_node/rgb/image_rect_color". The average publication rate ranges from 25 to 40 frames per second. The results are delivered in the form of a 2D and 3D bounding boxes defined by the by four corners for the 2D bounding boxes represented by x and y coordinates in pixels and eight corners for the 3D bounding boxes represented by x, y and z coordinates. The 2D human detection results are combined with the 3D information such as the depth information in order to calculate the corresponding 3D bounding boxes. The ZED2 camera is able to detect and localize multiple humans present in the RGB frames. The camera is able to deliver accurate and fast human detection and localization performance. The ZED SDK uses a neural network to determine which objects are present in the left and right images. The pre-implemented service uses PyTorch and TensorFlow to run deep learning pre-trained models to detect the human body. The object detection is automatically started when the parameter object_detection.od_enabled is set to true in the file zed2.yaml. The configuration files are present in the ROS ZED wrapper node. The camera computes the 3D position of each human present in the frame and produces a surrounding 3D bounding box, using data from the depth module. If the positional tracking module is activated, the ZED SDK can track

the objects within the environment. This means that the detected object will keep the same ID through the sequence, displaying the object's path over time. The distance of the object from the camera is expressed in meters. The SDK computes a 2D mask that indicates which pixels from the left image belong to the object. From there, the ZED camera can output the 2D bounding boxes of the objects and accurately compute the 3D bounding boxes with the help of the depth map. The bounding boxes of the objects can be displayed in RVIZ using the RGB image input for the 2D bounding boxes or a 3D point cloud for the 3D bounding boxes [2]. The detection model that enables human body detection and localization can be chosen from various object detection models with different performances. For the purposes of this project, the performance of the ZED is set on medium, which allows for a good balance between accuracy and speed. The prediction score threshold for the human detection is set to 90 to only process confident detections. The chosen human detection model can be changed by modifying the corresponding ROS parameter present in the ROS configuration file provided and used by the ZED ROS wrapper. The results of the human detection in 2D and 3D are published by the ZED2 camera on the following ROS topic: "/zed2/zed_node/obj_det/objects". The objects ROS message contains the predicted object label, which is in this case person, the ID, the confidence of the prediction that ranges from 1 to 99 and the 2D bounding box projected to the camera image contained in "zed_interfaces/BoundingBox2Di bounding_box_2d" as well as the 3D bounding box in world frame present in "zed_interfaces/BoundingBox3D bounding_box_3d".

```
#      0 ------- 1
#      |         |
#      |         |
#      |         |
#      3 ------- 2
zed_interfaces/Keypoint2Df[4] corners
```

Figure 5.1: zed_interfaces/BoundingBox2Df [1]

```
#      1 ------- 2
#     /.        /|
#    0 ------- 3 |
#    | .       | |
#    | 5.......| 6
#    |.        |/
#    4 ------- 7
zed_interfaces/Keypoint3D[8] corners
```

Figure 5.2: zed_interfaces/BoundingBox3D [1]

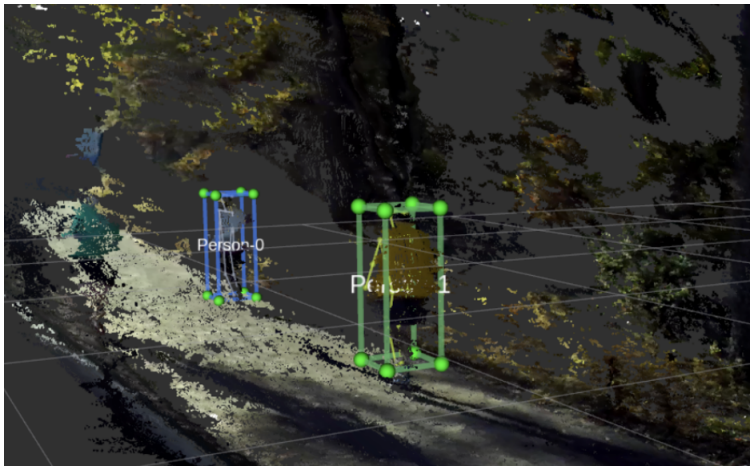


Figure 5.3: Example of detected and tracked humans in 3D [1]

5.2 Software Logic

In the subsequent section, the structure and logic of the human detection module is described in detail. The ROS callback, the data processing steps, the 2D detection inference, the calculation of the 3D bounding boxes, and the 2D as well as 3D visualization in RVIZ. The human action detector class, which contains the ROS callback responsible for pre-processing the sensor data to produce the 2D and 3D bounding boxes, is instantiated in the main function. The ROS callback is responsible for the preparation of the input required by the human action recognition inference. The model configuration file and the trained SlowFast and SlowOnly model relative paths are defined as constants in the python script. Followed by other predefined constants needed throughout the code.

```
CONFIG_PATH = '/workspace/zed_catkin_ws/src/mmdetection_ros/
    ↪ mmaction2/configs/detection/ava/
    ↪ slowfast_kinetics_pretrained_r50_8x8x1_20e_ava_rgb.py'
MODEL_PATH = '/workspace/zed_catkin_ws/src/mmdetection_ros/
    ↪ mmaction2/checkpoints/
    ↪ slowfast_kinetics_pretrained_r50_8x8x1_20e_ava_rgb_20201217
    ↪ -ae225e97.pth'
```

The parameters required by the neural network for the inference as well as the variables used to form the input and to control the execution of the software are defined in the following way in the __init__ function of the ActionDetector class along the ROS variables used to retrieve the data from the ROS data topics and publish the results to ROS visualization topics:

```
class ActionDetector:
    def __init__(self, model, device, config):

        self.bridge = CvBridge()

        # 2D
        self.pub_topic_color = "/mmdet/pose_estimation/det2d/
            ↪ compressed"
        self.image_pub = rospy.Publisher(self.pub_topic_color,
            ↪ CompressedImage, queue_size=1)

        # 3D
        self.pub_topic_marker_array = "/mmdet/
            ↪ visualization_marker_array"
        self.marker_array_pub = rospy.Publisher(self.
            ↪ pub_topic_marker_array, MarkerArray, queue_size=1)

        # Subscribe to ZED 2 object detection data
        self.sub_topic_det_results_data = "/zed2/zed_node/obj_det
            ↪ /objects"
        self.hum_det_results_sub = message_filters.Subscriber(
            ↪ self.sub_topic_det_results_data, ObjectsStamped)

        # Subscribe to ZED 2 rectified RGB image
        self.sub_topic_color = "/zed2/zed_node/rgb/
            ↪ image_rect_color"
        self.rgb_image_sub = message_filters.Subscriber(self.
            ↪ sub_topic_color, Image)
```

```
# Visualization parameters in RVIZ
self.visualization_3d = rospy.get_param("visualization_3d
    ↪ ")
self.visualization_2d = rospy.get_param("visualization_2d
    ↪ ")

self.compressed_cv_image = None
self.compressed_cv_images_list = []
self.marker_array_msg = MarkerArray()

# Action detection model from MMaction
self.model = model
self.config = config
self.device = torch.device(device)

# Action detection model from MMaction
self.model = model
self.config = config
self.device = torch.device(device)

# stdet sampling strategy (fron neural network config)
val_pipeline = self.config.data.val.pipeline
sampler = [x for x in val_pipeline if x['type'] == '
    ↪ SampleAVAFrames'][0]
self.clip_len, self.frame_interval = sampler['clip_len'],
    ↪  sampler['frame_interval']
self.predict_stepsize = 40
self.window_size = self.clip_len * self.frame_interval
self.buffer_size = self.window_size - self.
    ↪ predict_stepsize
frame_start = self.window_size // 2 - (self.clip_len //
    ↪ 2) * self.frame_interval
self.frames_inds = [frame_start + 1 + self.frame_interval
    ↪  * i for i in range(self.clip_len)]

self.stdet_input_shortside=256

img_norm_cfg = config['img_norm_cfg']
if 'to_rgb' not in img_norm_cfg and 'to_bgr' in
    ↪ img_norm_cfg:
    to_bgr = img_norm_cfg.pop('to_bgr')
    img_norm_cfg['to_rgb'] = to_bgr
```

```
        img_norm_cfg['mean'] = np.array(img_norm_cfg['mean'])
        img_norm_cfg['std'] = np.array(img_norm_cfg['std'])
        self.img_norm_cfg = img_norm_cfg

        self.label_map = load_label_map(LABEL_MAP_PATH)
        self.new_h = 0
        self.new_w = 0
        self.w_ratio = 0
        self.h_ratio = 0
```

The main function of the python script mm-action-detector.py first loads the configuration file of the pretrained human action recognition model using the *fromfile* function provided by the MMCV package. Then the build_detector function from the mmdetection [5] package is used to initialize the inference used for the action recognition. This function takes the modified configuration file of the pre-trained SlowFast neural network and the configuration path of the neural network as arguments. The testing pipeline of the configuration file is used for the inference. By using the torch.device('cuda') function the input tensors provided to the neural networks are automatically routed to the CUDA current device, which by default is the 0 device. The ActionDetector class is instantiated in the main function and takes as arguments the pre-trained model, its corresponding neural network configuration file and CUDA as a device. The rospy.init_node is called to initialize the ROS node. The message_filters is a method defined by rospy and used here to collect the human detection results and RGB data published on the ROS source topics. The rectified RGB frame produced by the right lens and the depth data are respectively stored in sub_topic_color and sub_topic_det_results_data.

```
def main():
    config = Config.fromfile(CONFIG_PATH)
    checkpoint = MODEL_PATH
    config.model.backbone.pretrained = None
    model = build_detector(config.model, test_cfg=config.get('
        ↪ test_cfg'))
    _ = load_checkpoint(model, checkpoint, map_location='cpu')

    device = torch.device('cuda')
    model.to(device)
    model.eval()

    detector = ActionDetector(model, device, config)
```

```
    rospy.init_node('mmdetector', log_level=rospy.DEBUG)
    ts = message_filters.ApproximateTimeSynchronizer([detector.
        ↪ rgb_image_sub, detector.hum_det_results_sub],
        ↪ queue_size=1, slop=0.6, allow_headerless=True)
    ts.registerCallback(detector.callback)
    rospy.spin()
```

The message.filters.ApproximateTimeSynchronizer function takes in the ROS data source topics as arguments, which in turn takes in messages of different types from multiple sources and outputs in case it receives a message on each source respectively with the corresponding timestamp. The slop parameter defines the delay in seconds in which messages are synchronized. The queue_size flag determines how many sets of messages should be stored from each emanating ROS topic, while waiting for messages to be processed by the callback function. The callback function is registered with the registerCallback() method and is executed whenever data is available on both sources i.e., whenever it is possible to start processing the next set of data. The function rospy.spin() is used to prevent the main python thread from leaving or terminating the program's execution.

The callback is defined with the parameter RGB image and detected objects delivered by the ZED2 camera. If the ZED2 camera results are empty, which means that no human body is detected in the current RGB frame then the callback is skipped

```
def callback(self, image, objects_stamped):
    if objects_stamped.objects == []:
        # 'We would like to process only frames with detected
            ↪ huamns'
        return

    if self.control_idx in self.frames_inds:
        image_np = self.image_preprocessing(image)
        self.processing_human_detection_results(image_np,
            ↪ objects_stamped)
        if self.clip_frames_counter == self.buffer_size:
            self.action_recogntion()
            self.marker_array_pub.publish(self.marker_array_msg.
                ↪ markers)
            self.reset()
    self.control_idx += 1
```

The variable control_idx is used to select the frames we are interested in depending on the prediction step size used to control the size of the buffer

which stores the delivered RGB frames by the ZED2 camera. The function image_preprocessing takes as an input the raw RGB frame and delivers back the resized image which corresponds to the size of the images used by the neural network in the training phase. The images are rescaled in order to have the short side of the image set to 256.

```
def image_preprocessing(self, image):
    image_np = np.frombuffer(image.data, dtype = np.uint8).
        ↪ reshape(image.height, image.width, -1)

    custom_width = int(376 / 2)
    custom_height = int(1344 / 8)
    image_np = cv2.resize(image_np, (custom_width,custom_height))

    self.new_w, self.new_h = mmcv.rescale_size((custom_width,
        ↪ custom_height), (self.stdet_input_shortside, np.Inf))
    image_np = mmcv.imresize(image_np, (self.new_w, self.new_h))

    self.w_ratio, self.h_ratio = self.new_w / image.width , self.
        ↪ new_h / image.height
    return image_np
```

The function processing_human_detection_results take as parameters the resized image and human detection results.

```
def processing_human_detection_results(self, image_np,
    ↪ objects_stamped):
    self.bboxes_proposal_frame = []
    for counter, detected_human in enumerate(objects_stamped.
        ↪ objects):
        # 2D bounding box
        bbox_2d_pixel_0 = detected_human.bounding_box_2d.corners
            ↪ [0]
        bbox_2d_pixel_1 = detected_human.bounding_box_2d.corners
            ↪ [1]
        bbox_2d_pixel_2 = detected_human.bounding_box_2d.corners
            ↪ [2]
        bbox_2d_pixel_3 = detected_human.bounding_box_2d.corners
            ↪ [3]

        self.bbox = [bbox_2d_pixel_0.kp, bbox_2d_pixel_1.kp,
            ↪ bbox_2d_pixel_2.kp, bbox_2d_pixel_3.kp]
        rospy.logdebug("appending␣3d␣bbox␣and␣text␣marker:␣%s",
```

```
            ↪ counter)
        self.bbox = self.zed2_2d_bbox_preprocessing(self.bbox,
            ↪ bbox_2d_pixel_0.kp, bbox_2d_pixel_2.kp)
        self.bbox = np.array(self.bbox, dtype=np.float32)

        self.bbox[0] = self.bbox[0] * self.w_ratio
        self.bbox[1] = self.bbox[1] * self.h_ratio
        self.bbox[2] = self.bbox[2] * self.w_ratio
        self.bbox[3] = self.bbox[3] * self.h_ratio

        self.bboxes_proposal_frame.append(self.bbox)
        if self.visualization_2d : self.pre_vis_2d(image_np,
            ↪ bbox_2d_pixel_0.kp, bbox_2d_pixel_2.kp)

        bbox_3d = self.processing_3d_bboxes(detected_human.
            ↪ bounding_box_3d)
        marker_3d = make_3d_marker(bbox_3d)
        marker_text = make_text_marker(bbox_3d)
        self.marker_3d_list.append(marker_3d)
        self.marker_text_list.append(marker_text)

    tensor_bboxes_frame_proposal = torch.from_numpy(np.asarray(
        ↪ self.bboxes_proposal_frame)).to(self.device)
    self.bboxes_proposal_clip.append(tensor_bboxes_frame_proposal
        ↪ )

    # Append converted values to float32, normalized and
        ↪ resized images
    image_np = image_np.astype(np.float32)
    image_np = image_np[:,:,:3]
    image_np = mmcv.image.imnormalize(image_np, self.img_norm_cfg
        ↪ ['mean'], self.img_norm_cfg['std'])

    self.clip.append(image_np)
    self.clip_frames_counter +=1
```

The 2D detection results which come in the form of four coordinates of the 2D bounding box that surrounds the detected human body are processed in order to form a 2D bounding box in the coco format. These are defined by the center coordinates x and y and the height and width of the bounding box. The zed_2d_bbox_preprocessing is used to calculate the center coordinates x and y of the bounding box as well as the height and width from the 2D bounding box extracted from the human detection results.

```
#extract center, height and width of 2d human bbox.
def zed2_2d_bbox_preprocessing(self, subResult, kp0, kp2):
    y = int(subResult[0][0] / 2)
    x = int(subResult[0][1] / 2)
    width = int((subResult[2][0] - subResult[0][0]))
    height = int((subResult[2][1] - subResult[0][1]))
    return [x,y,width,height]
```

The processed 2D bounding boxes are multiplied by the corresponding height and width ratios calculated when resizing the images and are then appended to the variable bbox_proposal_frame which is used to buffer the 2D bounding boxes of the detected humans for every buffered frames which are used for the inference. The images are normalized using the mean and the standard deviation specified in the testing pipeline of the configuration file of the neural network. The images are stacked and transferred to a tensor that is required by the neural network for the inference. The images tensor is then transposed where the first dimension represents the channels that correspond to the number of the filters in the neural network, the second dimension represents the number of frames stacked in the tensor, the third dimension represents the height of the images and the fourth dimension represents the width of the images. The action recognition inference is performed once the required number of frames is buffered. The neural networks take as parameters the stacked processed images in a tensor format, the processed 2D bounding box in a tensor format, a python dictionary that contains the height and width of the images, and the last parameter indicates if the loss value should be returned or not.

```
def action_recogntion(self):
    return_loss = False
    # Stacked processed BGR Frames - 2nd parameter: THWC ->
        ↪ CTHW -> 1CTHW
    input_array = np.stack(self.clip).transpose((3, 0, 1, 2))[np.
        ↪ newaxis]
    input_tensor = torch.from_numpy(input_array).to(self.device)
    self.images_metas = dict(img_shape=(self.new_h, self.new_w))

    self.inference_number += 1
    with torch.no_grad():
        result = self.model(
            return_loss=False,
            img=[input_tensor],
            img_metas=[[self.images_metas]],
```

```
            proposals=[[self.bboxes_proposal_clip]]
            )[0]
```

The pre_vis_2d function is used to build the start and end point of the 2D boundingbox used for the visualization in RVIZ. The visualization can be enabled by setting the corresponding ros parameter defined in the roslaunch file to true. The function rectangle provided by the opencv library is used to draw the 2D bounding boxes and cv2_to_compressed_imgmsg provided by cv_bridge is used to compress the image which is then appended to a list and used to publish the images with the 2D bounding boxes of the detected images to the following topic: "/mmdet/pose_estimation/det2d/compressed"

```
def pre_vis_2d(self, image_np, kp0, kp2):
    start_point = (int(kp0[0] * self.w_ratio) ,int(kp0[1] * self.
        ↪ h_ratio))
    end_point = (int(kp2[0] * self.w_ratio) ,int(kp2[1] * self.
        ↪ h_ratio))
    cv_img = cv2.rectangle(image_np, start_point, end_point,
        ↪ COLORS_2D, 3)
    self.compressed_cv_image = self.bridge.
        ↪ cv2_to_compressed_imgmsg(cv_img)
    self.compressed_cv_images_list.append(self.
        ↪ compressed_cv_image)
```

For the 3D visualization in RVIZ the 3D bounding boxes of the detected humans are preprocessed in order to calculate the location of the ROS marker object used to localize the detected humans in the 3D point cloud.

```
bbox_3d = self.processing_3d_bboxes(detected_human.
    ↪ bounding_box_3d)
marker_3d = make_3d_marker(bbox_3d)
marker_text = make_text_marker(bbox_3d)
self.marker_3d_list.append(marker_3d)
self.marker_text_list.append(marker_text)
```

The ROS marker objects are appended to a ROS marker array and published at the end of the inference to the following topic "/mmdet/visualization_marker_array". The function processing_3d_bboxes takes as an input the 3D bounding boxes delivered by the ZED2 camera defined with 8 corners. Each corner is represented by a set of three points which correspond to the x, y and z coordinates in world frame. Two opposite corners are used to calculate the center of the 3D bounding box coordinate x and y as well as the associated depth value.

```
def processing_3d_bboxes(self, bbox_3d_corners):
    # 3D bounding box
    bbox_3d_pixel_0 = bbox_3d_corners.corners[0]
    bbox_3d_pixel_1 = bbox_3d_corners.corners[1]
    bbox_3d_pixel_2 = bbox_3d_corners.corners[2]
    bbox_3d_pixel_3 = bbox_3d_corners.corners[3]
    bbox_3d_pixel_4 = bbox_3d_corners.corners[4]
    bbox_3d_pixel_5 = bbox_3d_corners.corners[5]
    bbox_3d_pixel_6 = bbox_3d_corners.corners[6]
    bbox_3d_pixel_7 = bbox_3d_corners.corners[7]

    x_center = (bbox_3d_pixel_1.kp[0] + bbox_3d_pixel_7.kp[0])/ 2
    y_center = (bbox_3d_pixel_1.kp[1] + bbox_3d_pixel_7.kp[1])/ 2
    z_center = (bbox_3d_pixel_1.kp[2] + bbox_3d_pixel_7.kp[2])/ 2

    bbox_3d = [y_center, z_center, x_center]
    return bbox_3d
```

The function make_3d_marker is used to create the ROS marker object. The type CUBE is used to define the shape of the marker, the action ADD to specify that we want to add new markers when the publication occurs. The ID of the markers are reinitialized at the end of the action recognition inference in order to replace the old markers with the new calculated ones.

```
def make_3d_marker(marker_location):
    marker = Marker()
    marker.header.frame_id = "base_link"
    marker.type = Marker.CUBE
    marker.action = Marker.ADD
    marker.scale.x = SCALE / 2
    marker.scale.y = SCALE / 2
    marker.scale.z = SCALE
    marker.header.stamp = rospy.get_rostime()
    marker.id = MARKER_3D_COUNTER_ID +1
    marker.pose.position.y, marker.pose.position.z, marker.pose.
        ↪ position.x = marker_location
    marker.color.r, marker.color.g, marker.color.b, marker.color.
        ↪ a = COLORS_MARKER_3D
    return marker
```

The function make_text_marker is used to create a text marker which contains the predicted human action. This function is similar to the make_3d_marker

function. The TEXT_VIEW_FACING type is used to visualize the predicted labels in RVIZ. Both ros markers use the same marker location.

```
def make_text_marker(marker_location):
    marker = Marker()
    marker.header.frame_id = "base_link"
    marker.type = Marker.TEXT_VIEW_FACING
    marker.action = Marker.ADD
    marker.scale.x = SCALE/10
    marker.scale.y = SCALE/10
    marker.scale.z = SCALE/10
    marker.header.stamp = rospy.get_rostime()
    marker.id = MARKER_TEXT_COUNTER_ID +1
    marker.pose.position.y, marker.pose.position.z , marker.pose.
        ↪ position.x = marker_location
    marker.pose.position.z = - marker.pose.position.z * 10
    marker.color.r, marker.color.g, marker.color.b, marker.color.
        ↪ a = COLORS_TEXT_3D
    return marker
```

To evaluate the inference results, the last image from the buffer used for the inference and its corresponding bounding boxes are used to evaluate the results of the human action prediction. The inference results contain the predicted label and prediction score for every class present in the label map. A Threshold is used to select the highest prediction scores which are assigned to the ROS text marker objects. The results are appended to a list which is then published at the end of the evaluation.

```
proposal = self.bboxes_proposal_clip[-1]

for class_id in range(len(result)):
    if class_id + 1 not in self.label_map: continue
    for bbox_id in range(proposal.shape[0]):
        if result[class_id][bbox_id, 4] > self.action_score_thr:
            predicted_label = self.label_map[class_id + 1]
            prediction_score = result[class_id][bbox_id,4]
            self.marker_text_list[bbox_id].text = self.
                ↪ marker_text_list[bbox_id].text + """
            """ + predicted_label + ' : ' + str(prediction_score)

for bbox_id in range(proposal.shape[0]):
    self.marker_array_msg.markers.append(self.marker_3d_list[
        ↪ bbox_id])
```

```
self.marker_array_msg.markers.append(self.marker_text_list[
    ↪ bbox_id])
```

5.3 Roslaunch

For organizing and launching the final ROS workspace, a couple of roslaunch files are used. Figure 5.4 represents the hierarchy of the roslaunch files which structure the workspace. A Roslaunch files can be used to launch multiple ROS nodes locally and set parameters on the Parameter Server. With a roslaunch file, multiple XML configuration files can be recursively manipulated to change the ROS parameters to set and nodes to launch.

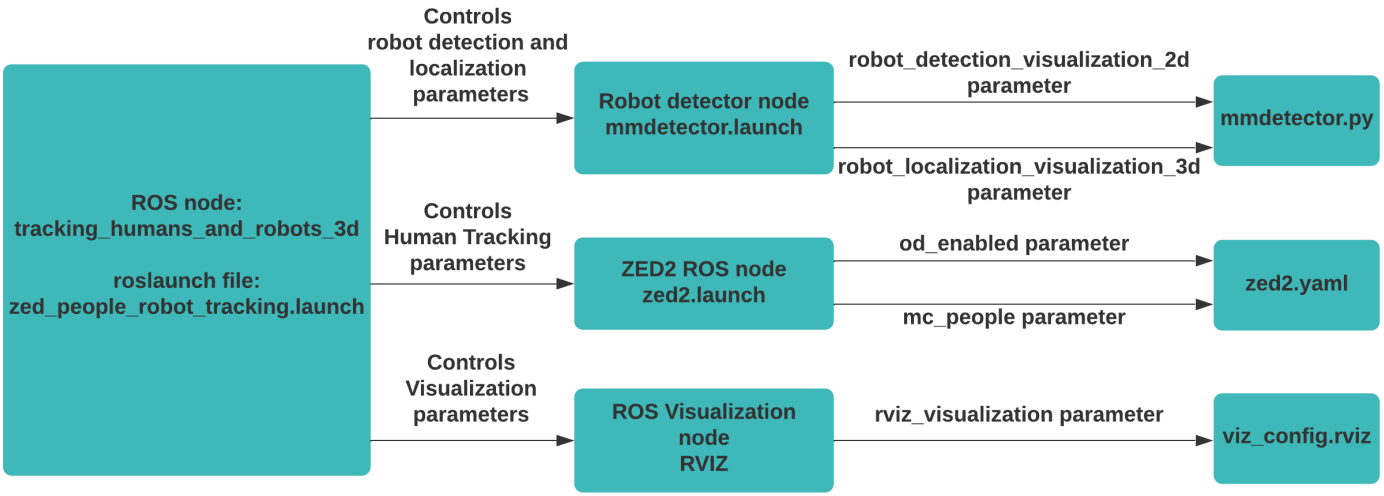


Figure 5.4: Overview of the ROS nodes, the responsible ROS launch file and the corresponding yaml configuration file

The principal roslaunch file is called zed_people_robot_tracking and can be found in the Github repo provided with this project. These ROS parameters are used to control the different processes. These parameters will decide the nodes to start and the parameters to set. They are accessed in the python script mm-action-detector.py to decide which visualization mode between 2D and 3D bounding box to start.

```
<arg name = "rviz_visualization" value = "1" />
<arg name = "human_tracking" value = "1" />
<arg name = "launch_ZED2" value = "1" />
```

To start the ZED2 camera, the value of "launch_ZED2" is set to 1 (true). If the parameter is set to true then the roslaunch file searches for the corresponding XML configuration file called "zed2.launch" that can be found under "3D-dynamic-object-localization-ros/zed_catkin_ws/ src/zed-ros-wrapper/zed_wrapper/launch/". The zed2 launch file start the ZED2 camera node and the human tracking service if activated.

```
<!-- Launch zed 2 and human tracking service -->
  <group if = "$(eval␣arg('launch_ZED2')␣==␣1)">
    <include file = "$(find␣zed_wrapper)/launch/zed2.launch"/>
  </group>
```

The ZED2.launch file includes the following XML file:
"zed_wrapper/launch/include/zed_camera.launch.xml" which includes the following file:
"zed-ros-wrapper/zed_wrapper/launch/include/zed_camera.launch.xml" which loads the respective yaml configuration file for the zed2 camera "zed-ros-wrapper/zed_wrapper/params/zed2.yaml" which contains the ROS parameters to enable and modify the human tracking service provided by the ZED2 camera. The parameter "od_enabled" is set to true to enable the object detection service. The model used is "MULTI_CLASS_BOX_ACCURATE". In the code snippet, the model is selected by choosing a number between *0* and *3*, where *1* stands for *MULTI_CLASS_BOX_ACCURATE*. It allows accurate 3D detection and 3D bounding box visualization. This model allows setting the parameter "mc_people" to true to detect humans and object_tracking_enabled to *true* to enable tracking of the detected humans.

```
object_detection:
    od_enabled: true
    model: 1
    object_tracking_enabled: true
    mc_people: true
```

This code section is responsible for enabling the bounding box visualization in 2D on the RGB picture and in 3D on a 3D point cloud computed by the ZED2 camera.

```
  <group if = "$(eval␣arg('human_detection')␣==␣1)">
       <include file = "$(find␣mmdetection_ros)/launch/
          ↪ mmdetector.launch">
            <arg name = "human_detection_visualization_2d" value
               ↪  = "true" />
```

```
            <arg name = "human_localization_visualization_3d"
               ↪ value = "true" />
        </include>
    </group>
```

The parameter human_detection is used to enable or disable launching the ROS node responsible for the detection of the humans in 2D and 3D. The parameters "human_detection_visualization_2d" and "human_localization_visualization_3d" are passed to the mmdetector launch file which start the detector node. Figure 5.4 visualizes the different ROS nodes and parameters used to control the execution of the software. They are respectively used to enable or disable the visualization of the 2D and 3D bounding box as well as the predicted labels in RVIZ. The rviz_visualization parameter is used to start the visualization in RVIZ with a predifined RVIZ configuration file called which can be found under the directory "zed_catkin_ws/src/tracking_humans_and_robots_3d)/rviz/".

```
<!-- Visualization in RVIZ -->
   <group if = "$(eval␣arg('rviz_visualization')␣==␣1)">
           <node name = "tracking_visualization_rviz" pkg="rviz"
              ↪  type="rviz" args="-d␣$(find␣
              ↪ tracking_humans_and_robots_3d)/rviz/rviz_config.
              ↪ rviz" if="$(arg␣rviz_visualization)"/>
   </group>
```

5.4 Deployment within ROS with Docker

For the deployment on the Nvidia Jetson board, a docker container [2] is used to facilitate and automate the setup of the environment and launch the human localization software and visualize the results in RVIZ. The docker default runtime must be set to Nvidia runtime during docker build operations to enable access to the CUDA compiler (nvcc). To do so, "default-runtime": "nvidia" must be added to the daemon.json configuration file found under "/etc/docker/" before attempting to build the containers. The docker service must be restarted in order for the effects to take place [9]. The base docker image used is the nvcr.io/nvidia/l4t-base:r32.5.0, which is an official docker image hosted on NVIDIA GPU Cloud [6] based on the operating system Linux4tegra of the Nvidia Jetson boards.

```
   ARG BASE_IMAGE=nvcr.io/nvidia/l4t-base:r32.5.0
```

This list contains the docker environment variables used in the dockerfile.

```
ARG PYTHON=python3.7
ARG ROS_PKG=desktop-full
ENV ROS_DISTRO=melodic
ENV ROS_ROOT=/opt/ros/${ROS_DISTRO}
ENV ROS_PYTHON_VERSION=2
ENV OPENCV_VERSION=4.5.3
ENV OPENBLAS_CORETYPE=ARMV8
ENV DEBIAN_FRONTEND=noninteractive
```

The docker environment and argument variables enhance the readability, the maintenance and further development of the dockerfile. The environment variable OPENBLAS_CORETYPE set to ARMV8 is used to avoid an Illegal instruction (core dumped) on import caused by the numpy package. The debian frontend is set to noninteractive to skip user interactions when installing various packages or upgrading the system via apt. Various required system packages, such as git, cmake, curl, and wget must be installed to setup the docker container.

```
RUN apt-get update \
    && apt-get install -y --no-install-recommends \
    git cmake build-essential curl wget gnupg2 \
    lsb-release ca-certificates software-properties-common \
    && add-apt-repository universe && apt-get update \
    && rm -rf /var/lib/apt/lists/*
```

The version 3.7 of python is installed and configured as the default python version used by the docker system inside the container. The initial 3.6 python version provided by the base image is uninstalled and removed. Subsequently, the corresponding pip and setuptools python 3.7 versions of the tools are installed.

```
RUN add-apt-repository ppa:deadsnakes/ppa \
    && apt-get update && apt-get install -y ${PYTHON} \
    && wget https://bootstrap.pypa.io/get-pip.py \
    && ${PYTHON} get-pip.py \
    && rm get-pip.py

RUN ln -sf /usr/bin/${PYTHON} /usr/local/bin/python3 \
    && ln -sf /usr/local/bin/pip /usr/local/bin/pip3 \
    && pip3 --no-cache-dir install --upgrade pip setuptools \
    && ln -s $(which ${PYTHON}) /usr/local/bin/python \
```

```
    && apt-get install -y ${PYTHON}-dev \
```

Afterwards the python packages: cython, numpy, scickit-build, wheel, matplotlib pyopengl and funcy are installed with pip. These packages are required by the mmdetector python script.

```
RUN python3 -m pip install setuptools wheel matplotlib cython
    ↪ numpy
    scikit-build pyopengl funcy
```

Following this, version 3.5.5 of the ZED SDK for Linux4Tegra with Jetpack Nvidia SDK version 4.5.0 is installed in the following way:

```
COPY ./ZED_SDK_Tegra_JP45_v3.5.5.run .
RUN apt-get update -y && apt-get install -y --no-install-
    ↪ recommends lsb-release wget less udev sudo apt-transport-
    ↪ https libqt5xml5 libxmu-dev libxi-dev build-essential
    ↪ cmake
    && chmod +x ZED_SDK_Tegra_JP45_v3.5.5.run \
    && /ZED_SDK_Tegra_JP45_v3.5.5.run -- silent
```

For a fast inference and to leverage the power of the GPUs, the Pytorch and MMCV CUDA-enabled versions are installed. For this purpose, both are built from source [4]. These two steps take a long time to complete. Instead of having to build pytorch and mmcv from source each time the docker container has to be rebuild, the wheels are saved and extracted outside of the docker container at the end of the successful installations from source. Following this, the newly built wheels are copied to the docker image when building the next docker image and pip is used to install MMCV and Pytorch cuda enabled versions directly using the extracted wheels. This step saves a lot of time while debuging and testing the Software. The links which contain the built wheels can be found in the README of the Github repository of this project. Subsequently, torchvision, pycocotools, terminaltables and mmdet [14] are installed using precompiled wheel files.

```
# Installing cuda enbaled pytorch
WORKDIR /pytorch_wheel
COPY pytorch_wheel/torch-1.7.0a0-cp37-cp37m-linux_aarch64.whl .
RUN pip3 install torch-1.7.0a0-cp37-cp37m-linux_aarch64.whl \
    && pip3 install torchvision \
    && apt-get clean all \
    && rm -rf /var/lib/apt/lists/*
```

```
# Install cuda enabled mmcv (mmcv-full version)
WORKDIR /mmcv_wheel
COPY mmcv_wheels/mmcv_full-1.3.14-cp37-cp37m-linux_aarch64.whl /
    ↪ mmcv_wheel
RUN pip3 install mmcv_full-1.3.14-cp37-cp37m-linux_aarch64.whl \
    && pip3 install pycocotools terminaltables mmdet \
    && pip3 install rospkg \
    && apt-get clean all \
    && rm -rf /var/lib/apt/lists/*
```

The recommended full desktop version of ROS melodic is used since it contains some necessary packages such as rviz for visualization. The desktop version of ROS melodic with python 2 support is installed [21]. The desktop version contains RVIZ, which is used to visualize the 3D bounding boxes of the localized humans and the predicted action results. The setup.bash ROS file is sourced at the end of the installation.

```
RUN sh -c 'echo␣"deb␣http://packages.ros.org/ros/ubuntu␣$(
    ↪ lsb_release␣-sc)␣main"␣>␣/etc/apt/sources.list.d/ros-
    ↪ latest.list'
RUN curl -s https://raw.githubusercontent.com/ros/rosdistro/
    ↪ master/ros.asc | apt-key add -
RUN apt-get update && apt-get install -y --no-install-recommends
    ↪  \
    && apt-get clean \
    && apt-get autoremove \
    && rm -rf /var/lib/apt/lists/*

# install ROS packages
RUN apt-get update \
    && apt-get install -y --no-install-recommends \
               ros-melodic-${ROS_PKG} \
               ros-melodic-image-transport \
               ros-melodic-vision-msgs \
          python-rosdep \
          python-rosinstall \
          python-rosinstall-generator \
          python-wstool \
    && cd ${ROS_ROOT} \
    && rosdep init \
    && rosdep update \
    && rm -rf /var/lib/apt/lists/*
```

MMaction2 [3] is built from source along its requirements which are specified in the requirements/build.txt file.

```
RUN cd src/mmdetection_ros/mmaction2 \
    && pip3 install -r requirements/build.txt \
    && pip3 install scipy==1.7.0 einops \
    && pip3 install -v -e . \
        && pip3 install mmaction2 \
        && pip3 install mmcv-full
```

Afterward, the "zed_catkin_ws" is copied inside the docker container, and the catkin tool is used to build the ROS workspace.

```
COPY zed_catkin_ws /workspace/zed_catkin_ws
WORKDIR /workspace/zed_catkin_ws

# build catkin workspace
RUN /bin/bash -c '. /opt/ros/melodic/setup.bash ; catkin_make -
    ↪ DCMAKE_BUILD_TYPE=Release'
```

The "zed_catkin_ws" contains 4 other ros directories required by this project. the "zed-ros-examples" and the "zed-ros-wrapper" provided by the ZED2 camera are necessary for the human tracking service and the visualization in RVIZ. The "mmdetection_ros" directory contains the roslaunch file under the launch directory and the mmdetection folder that contains the mm-action-detector.py script for the detection and localization of the humans and the predicted human actions in python. A bash script is used as an entry point for the final docker image called "ros_entrypoint.sh" to automatically source the final ros workspace at the start of the container. The last command starts a terminal with the sourced final ROS workspace when the container is startet.

```
#!/bin/bash
set -e

PWD="$(pwd)"
ROS_ENV_SETUP="/opt/ros/$ROS_DISTRO/setup.bash"
ROS_WORKSPACE_SETUP="$PWD/devel/setup.bash"
echo "Source ROS_ENV_SETUP  $ROS_ENV_SETUP"
source "$ROS_ENV_SETUP"
echo "Source ROS_WORKSPACE_SETUP  $ROS_WORKSPACE_SETUP"
source "$ROS_WORKSPACE_SETUP"
exec "$@"
```

This instruction must be run to build the docker container

```
sudo docker build -t ros-zed2-mm-action-recognition:1.0.0 .
```

And this is the instruction to run or start the docker container

```
sudo docker run -it --runtime nvidia --gpus all --net=host --
    ↪ privileged -e DISPLAY=$DISPLAY -v /tmp/.X11-unix:/tmp.X11-
    ↪ unix:rw ros-zed2-mm-action-recognition:1.0.0 /bin/bash
```

6 Evaluation

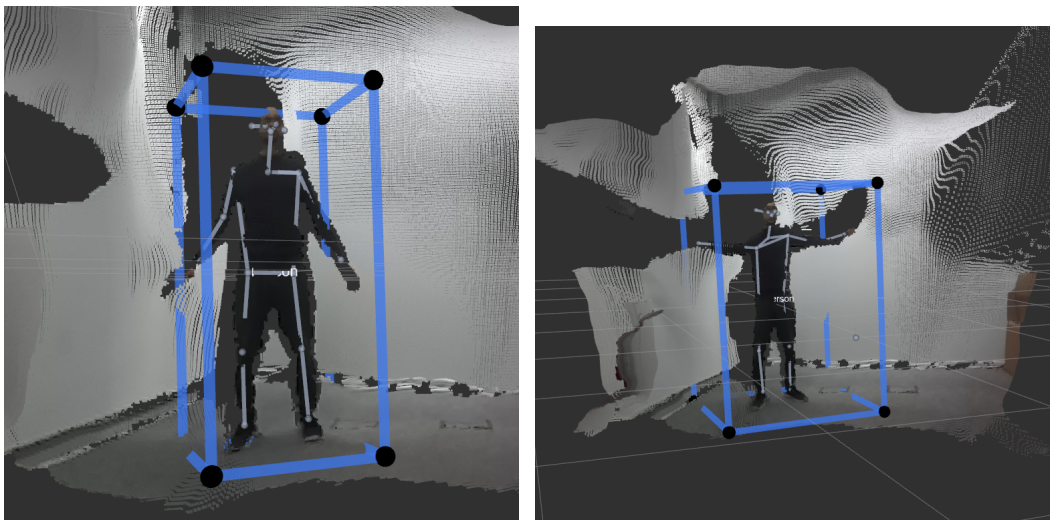
This section presents the experimental results of this project. The results for the human detection in 2D and localization in 3D and the action recognition inference.

6.1 Human Tracking

The human tracking service provided by the ZED2 camera can detect and track the human body with various different models. Different models are available via the ZED SDK. We have to keep in mind the existing trade-off between speed and accuracy:

```
'0': MULTI_CLASS_BOX
'1': MULTI_CLASS_BOX_ACCURATE
'2': HUMAN_BODY_FAST
'3': HUMAN_BODY_ACCURATE
```

The *DETECTION MODEL* parameter specifies the type of model used for the human detection. The parameter od_enabled is set to *true* to enable the object detection service. The model used is MULTI_CLASS_BOX_ACCURATE. In other words, accurate 3D detection is visualized with 3D bounding boxes. This model allows setting the parameter mc_people to *true* to detect humans. Alternatively, the HUMAN_BODY_FAST can be chosen and the body_fitting parameter is set to *true* to draw a skeleton on the human body to visualize the results. However, in crowded environment it might be more viable to detect a human head instead of a human body. In this case, the DETECTION_MODEL PERSON_HEAD_BOX might be used instead of the accurate chosen model [19]. The tracking of the detected humans over time can be enabled by setting the object_tracking_enabled to *true*. Here is an example of the visualization in RVIZ of the 3D bounding surrounding the tracked human body. The dynamic size of the bounding box visualizes the accurate detection and 3D localization of the human body using the pre-implement ZED2 human tracking service. The figures 6.1b and 6.1a show how the 3D bounding box's size adapt to the width assigned to the detect dynamic object, in this case, the human body.



a) Thin 3D bounding box

b) large 3D bounding box

Figure 6.1: 3D bounding box of tracked human body

Even when the humans are located far from the camera and partially occluded by different objects, the humans were still detected. The tracking service can detect humans at a minimum depth of 0.3 meters and a maximum depth 20 m for tracking humans. the above figures show how the object tracking model of the ZED2 camera can detect and track the distant and occluded human body in 3D.

6.2 Human Behavior Recognition

The human action recognition performance depends on the input data coming from the camera. The RGB rectified image is published on the ROS topic /zed2_node/rgb/image_rect_color with a constant average rate. The RVIZ parameters necessary for enabling and disabling each type of visualization are stored in the roslaunch file zed_people_robot_tracking.launch, which is launched by roslaunch when starting the ROS package tracking_humans_and_robots_3d. With the help of rospy the two ROS parameters human_detection_vis_2d and human_localization_vis_3d present in the launch file zed_people_robot_tracking are extracted and stored in corresponding python parameters visualization_2d and visualization_3d. These are used to control the 2D and 3D bounding box visualization, which can be activated or deactivated as the user wishes. Besides being responsible for launching the multiple nodes, the main roslaunch file is used to manipulate the type of visualization in RVIZ.
The inference running as part of the ActionDetector class delivers results for the detected humans in 2D and 3D. The center of the 2D bounding box

should correspond with the center of the detected human. The bounding box surrounding the detected object is built using opencv and the final image which contains a bounding box surrounding every detected human is published on the following ROS topic:
/mmdet/pose_estimation/det2d/compressed.

To compute the 3D coordinates of a 3D bounding box which surrounds the detect object. The x and y coordinates of the center of the detected object's bounding box are processed with the extracted z coordinate value from the 3D bounding box of the detected humans provided by the ZED2 camera. The results are 3D coordinates of the bounding box. The resulting 3D bounding boxes as well as the predicted actions are appended to a ROS marker array with a corresponding unique ID per 3D bounding box. Each new marker is used to delete the previously published markers. Consequently, the software always publishes a list of 3D bounding boxes to RVIZ when the 3D human localization and the visualization in RVIZ are activated. On one side, the 2D bounding boxes can be visualized on the original RGB picture, and on the other side, the visualization of the 3D bounding boxes along with the predicted actions occurs on a 3D point cloud generated by the ZED2 camera. The following Figures represent some examples of the visualizations and 3D localization of a human along with the corresponding action labels and prediction score given by the ActionDetector class and resulting from the action recognition inference. The used model, *SlowFast* used by the action recognition inference is able to detect several actions of humans in the same picture. Thus, the ROS ActionDetector node is able to assign several action labels to the same human. E.g., the following figures represent the 3D bounding box visualization of human body with the predicted action label in 3D in RVIZ:

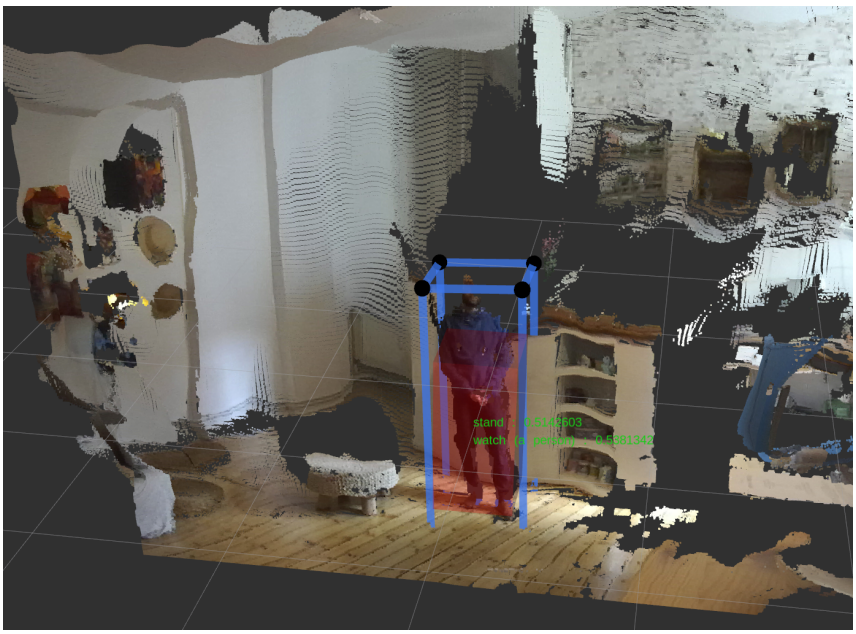


Figure 6.2: Actions recognized: stand and watch

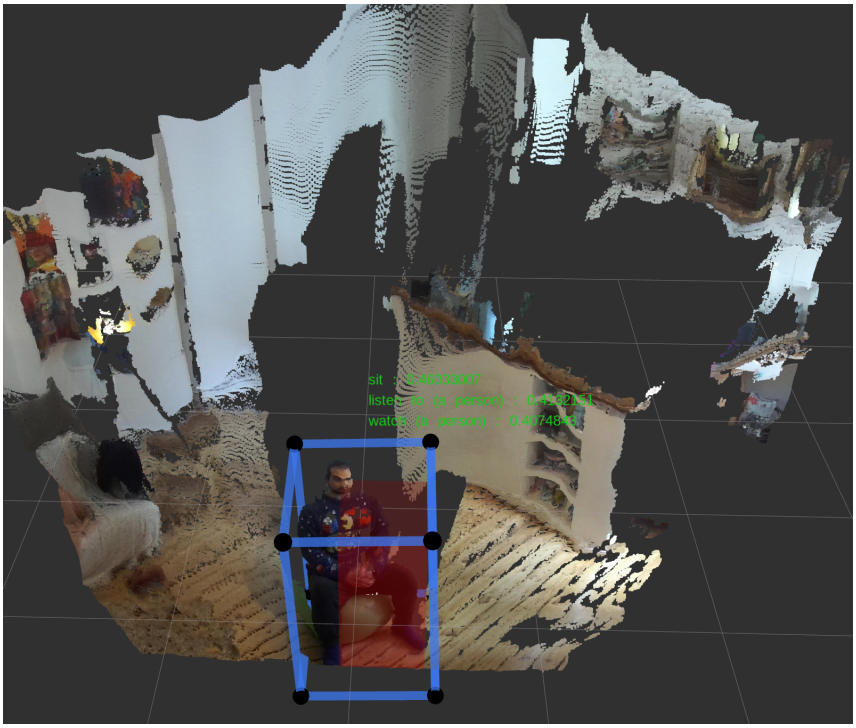


Figure 6.3: Actions recognized: sit, listen to, watch

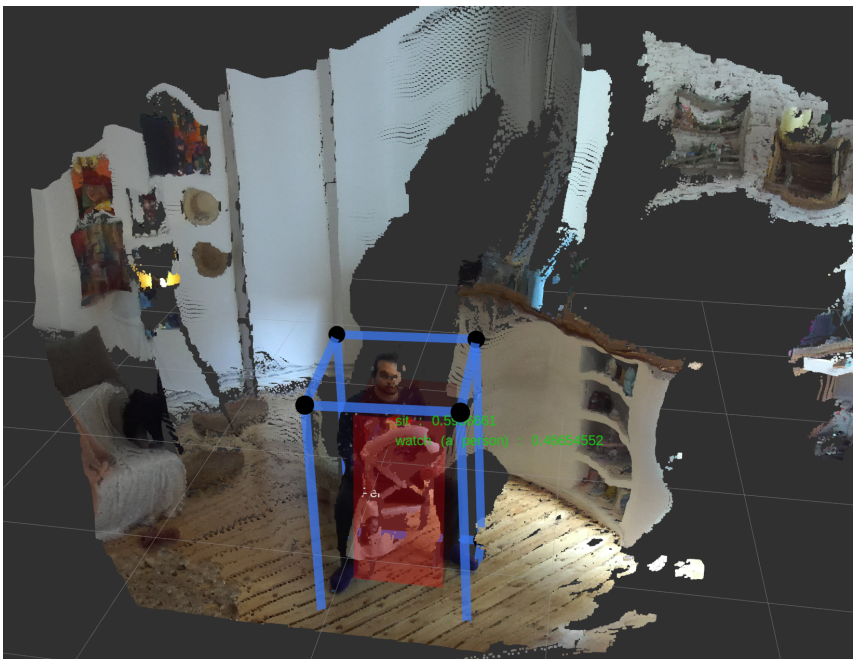


Figure 6.4: Actions recognized: sit, watch

Each time a callback is finished, a marker array is published. The ROS marker array contains the generated 3D bounding boxes for the ROS visualization as well as the predicted actions in RVIZ. In other words, the average publication rate on the markers array ROS topic depend on how fast the data is processed.

7 Conclusion

This section provides a wider perspective on the achievements of the human action recognition software. The observations and conclusion gathered during the testing of the developed human action recognition are presented. In particular, the inference accuracy and the 3D human localization speed and accuracy as well as some performance limitations are discussed.
The pre-defined detection threshold variable in the ActionDetector script is a distinctive variable. In case the prediction score is smaller than the predefined threshold, then predicted labels are not selected. Thus, the visualization of the predicted action is skipped and the ROS markers array contains just 3D bounding boxes of the localized humans. The performance of the software depends heavily on the chosen model for the human detection in 2D and the human action recognition model. This is expected since the two steps are separated and require computation resources and time. Using the threshold that ranges between zero and one in the mm-action-detector.py script, we can manipulate the accuracy of the action prediction and the human detections. For more accurate localization, the threshold of the human detection model must be increased. In this case, the human might not be localized even though it might be well-positioned in front of the camera. The threshold variable can be decreased for a higher number of detected instances. There are some performance limitations related to the action recognition inference. During testing, the trained SlowFast model failed to assign the correct action label in some cases. Several reasons might be behind this limitations. E.g., the light, the angle of the camera and the orientation of the human have a significant impact on the detection performance. Naturally, it is harder to detect objects or humans in a dark environment. Moreover, it becomes more challenging for the model to detect the occluded human with high accuracy in successive frames when one of the humans is partially occluded by any other object.
In this project, a pipeline for recognizing and categorizing actions using the results of detected human of the ZED2 camera was developed. The pipeline is integrated into the ROS ecosystem to localize humans and their behaviour in 3D. As mentioned, for tracking humans in 3D, a pre-implemented service from the ZED2 camera and a pre-trained model is downloaded and configured at runtime by the ZED2 camera. Developing the action recognition software and integrating it into the ROS ecosystem relies on the successful coordination of various components. For the action recognition algorithm

responsible for assigning the corresponding action labels, it is necessary to consider the existing trade-off between speed and accuracy. For this project, the goal is to develop a fast recognizer, and using the SlowFast model as a deep learning approach for detecting dynamic objects turned out to be a reliable option. SlowFast delivers reasonable inference results for the human action and fast inference time in comparison to other models. For future work, we suggest optimizing the input of neural network in order to achieve better inference results. For a real time performance, it is advised to migrate from ROS to ROS2 and to refactor the code in C++.

Bibliography

[1] *3D Object Detection Overview*. URL: https://www.stereolabs.com/docs/object-detection/ (cit. on pp. 13, 16, 17).

[2] *3D-robot-human-tracking-ros*. Feb. 2022. URL: https://github.com/ignc-research/3D-dynamic-object-localization-ros/tree/3D-robot-human-tracking-ros (cit. on p. 30).

[3] MMAction2 Contributors. *OpenMMLab's Next Generation Video Understanding Toolbox and Benchmark*. https://github.com/open-mmlab/mmaction2. 2020 (cit. on p. 34).

[4] MMCV Contributors. *MMCV: OpenMMLab Computer Vision Foundation*. 2018 (cit. on p. 32).

[5] MMDetection3D Contributors. *MMDetection3D: OpenMMLab next-generation platform for general 3D object detection*. 2020 (cit. on p. 20).

[6] *Data Science, Machine Learning, AI, HPC Containers: NVIDIA NGC*. URL: https://catalog.ngc.nvidia.com/ (cit. on p. 30).

[7] *Docker Overview*. Jan. 2022. URL: https://docs.docker.com/get-started/overview/ (cit. on p. 9).

[8] *Docker Overview*. Feb. 2022. URL: https://docs.docker.com/get-started/overview/#the-underlying-technology (cit. on p. 9).

[9] Dusty-Nv. *dusty-nv/jetson-containers: Machine Learning Containers for NVIDIA Jetson and JetPack-L4T*. URL: https://github.com/dusty-nv/jetson-containers (cit. on p. 30).

[10] Chunhui Gu et al. *AVA: A Video Dataset of Spatio-temporally Localized Atomic Visual Actions*. 2017. DOI: 10.48550/ARXIV.1705.08421. URL: https://arxiv.org/abs/1705.08421 (cit. on p. 7).

[11] *Jetson AGX Xavier Developer Kit*. Dec. 2021. URL: https://developer.nvidia.com/embedded/jetson-agx-xavier-developer-kit (cit. on p. 11).

[12] *Jetson AGX Xavier Developer kit*. Jan. 2022. URL: https://developer.nvidia.com/embedded/jetson-agx-xavier-developer-kit (cit. on p. 11).

[13] Dirk Merkel. "Docker: Lightweight Linux Containers for Consistent Development and Deployment." In: *Linux J.* 2014.239 (Mar. 2014). ISSN: 1075-3583. URL: http://dl.acm.org/citation.cfm?id=2600239.2600241 (cit. on p. 9).

[14] Open-Mmlab. *Mmdetection/get$_s$tarted.MDatmasteropen – mmlab/MMDETECTION*. Dec. 2021. URL: https://github.com/open-mmlab/mmdetection/blob/master/docs/en/get_started.md (cit. on p. 32).

[15] *Ros Introduction*. URL: https://husarion.com/tutorials/ros-tutorials/1-ros-introduction/?gclid=Cj0KCQiAi9mPBhCJARIsAHchl1xSlvRzHqfCMx7jHOrg5j-pCQscJiyoRbg_86RWTr232_dgKyXamWMaAj6eEALw_wcB (cit. on p. 8).

[16] *Runtime metrics*. Feb. 2022. URL: https://docs.docker.com/config/containers/runmetrics/ (cit. on p. 9).

[17] Sumit Saha. *A comprehensive guide to Convolutional Neural Networks the elite way*. Dec. 2018. URL: https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53 (cit. on pp. 4, 6).

[18] Stanford Artificial Intelligence Laboratory et al. *Robotic Operating System*. Version ROS Melodic Morenia. May 23, 2018. URL: https://www.ros.org (cit. on p. 7).

[19] *Using the object detection api*. URL: https://www.stereolabs.com/docs/object-detection/using-object-detection/ (cit. on p. 36).

[20] *Wiki*. URL: http://wiki.ros.org/roslaunch (cit. on p. 8).

[21] *Wiki*. URL: http://wiki.ros.org/melodic/Installation/Ubuntu (cit. on p. 33).

[22] *Zed 2 - AI stereo camera*. URL: https://www.stereolabs.com/zed-2 (cit. on pp. 10, 11).