Technische Universität Berlin

Verteilte Industrielle Steuerungssysteme

RGB-D Object Detection for Robot Perception

Kim Schwarz, Vivien Jiranek & David Hoffmann

May, 2021

Contents

List of Figures

List of Tables

1 Introduction

1.1 Motivation

With a continuing demand for automation robots have become more common in industrial workplaces in the recent years, accelerating the growth of completely automated production facilities (Tilley, 2021). However many sectors such as health care, education, entertainment, and business benefit from utilizing robots. For such use cases, new models such as the Pepper robot were developed to provide functionalities to interact with humans on a more personal level and respond to behaviour such as body language and facial expressions. This creates an interaction that feels more comfortable to participants and generates a more widespread acceptance of the use of robotics in sectors that rely on some form of customer interaction (Pandey and Gelin, 2018).

The significant amounts of research and development that are being conducted in this field have produced refined novel technologies that enable a higher degree of functionality and provide more flexible use cases. At the same time the cost of deploying robots is decreasing, making them more viable for wide spread use, especially since for instance the cost of labor in the manufacturing sector in the US, which was equal to the cost of deploying robots in 1990, has more than doubled on average over the last 30 years while the cost of deploying robots has decreased by half (Tilley, 2021). So economically speaking, the deployment of robots has become increasingly beneficial. Furthermore the deployment of robots in industrial or harsh environments that would affect humans negatively ensures the safety of human workers while providing an high degree of automation, increasing efficiency of production processes in return. However in order to carry out their tasks sufficiently without endangering participants or having their respective behaviour result in misconduct, robots need to be provided with an extensive perception of their environment at high speeds and a significant level of accuracy to enable them to dynamically detect and react to changes in their environment.

These detection systems can rely on a multitude of sensors to detect their environment but visual perception has the key advantages of enabling a full mapping of their surroundings as well as the possibility of classifying different objects from the same input. The capability of extracting meaningful information from image data has also significantly increased in recent years with the advancements in machine learning systems. These have enabled very fast and highly accurate classification as well as segmentation systems which can be utilized to inform a robot about objects present in their local vicinity. This can allow a robot to make predictions on dynamic changes of these objects and act accordingly to adopt more efficient path planning, avoid potential collisions and prevent harm to humans in their environment.

1.2 Objectives

In order to fulfill their tasks in an industrial setting, robots that are deployed in collaborative environments with other robots need to be able to correctly detect and classify the participants to later adjust their behaviour. This project's objective was to develop an object detection system to discern three different robot models from their environment and locate them in relation to the robot's camera. The robots to be detected were the models Pepper, TIAGo, and the KUKA youBot. To enable a fully aware navigation in an indoor environment this system has to be extended with a human pose detection system. The corresponding information pipeline from data acquisition to deploying the trained high speed detection model was then implemented in the Robot Operating System (ROS) framework. To fulfill the previously mentioned tasks, two different depth camera models, namely the Intel Realsense Depth Camera D435i and the Stereolabs Zed2 Camera, were provided as input devices for the development of the RGB-D object detection system for robot perception.

The projects contributions are:

- Establish an object detection system to detect and locate a robot in relation to a camera.
- Create and utilize 3D models of the robots for artificial training data generation.
- Train neural network models to detect Pepper, TIAGo, and the KUKA youBot in images.
- Integrate the detection system into a ROS infrastructure.
- Compare the capabilities of the Intel Realsense Depth Camera D435i and the Stereolabs Zed2 camera models.

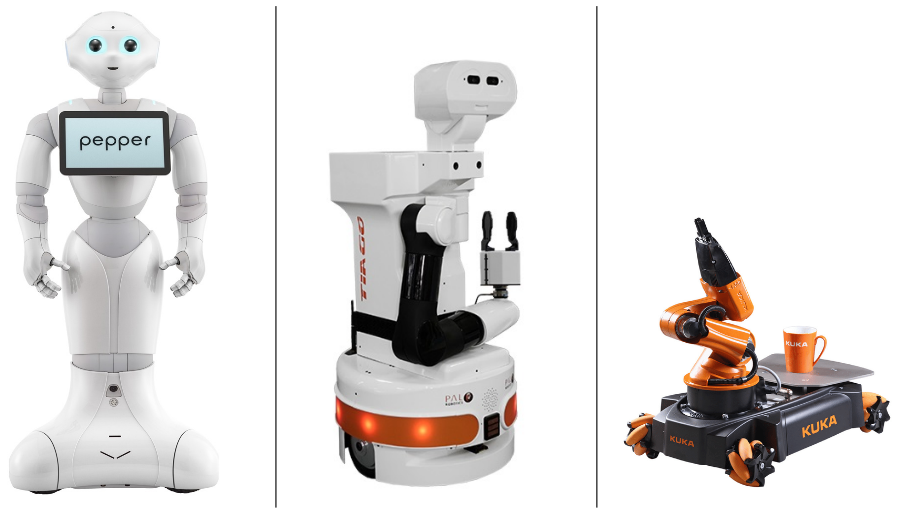


Figure 1: Robots Pepper(left), TIAGo(middle) and KUKA youBot(right)

2 Background

This chapter offers an introduction of the neural network models utilized in this project. First, we introduce Convolutional Neural Networks (CNN) in general, which are the base for all other models utilized in this project. Then we will present two networks for 2D object detection, namely Faster region-based convolutional neural networks (Faster R-CNN) and YOLO and finally Single Shot Pose Estimation (SSPE), which predicts the 2D projection of 3D bounding boxes in 2D images.

2.1 CNN

A CNN is a special type of deep learning network commonly used for image related tasks. A neural network is able to automatically extract features from given data, such as an image and makes predictions based on these features. For example, in image classification the goal is to predict if an image belongs to a certain class (e.g. cat or dog). The pixel values of the image are stored as an array, which can then be processed by the CNN. The CNN outputs a probability vector with an element for each possible class.
Figure 2 shows an example of a typical CNN for digit classification. It contains an input layer, an output layer, convolutional layers, pooling layers, fully-connected layers and a dropout layer.
Each layer is connected with the one before. The layers in between the input and output layer are often called hidden layers. The hidden layers process the input in a way, so that the data can be classified in the output layer.
A convolutional layer contains feature maps, which are generated by applying a mathematical operation, called convolution, on local areas in the previous layer with a kernel or filter. The output is then passed through a non-linear function, for example a rectified linear unit (ReLU) and afterwards passed to the next layer.
A pooling layer is a type of layer that decreases the complexity of a feature map by down-sampling it. For example, a max-pooling layer computes the maximum value in a local patch and discards all other values.
In a classification task, the final layers usually are fully-connected layers. The high-dimensional features are flattened to an array and passed into a fully-connected layer which is connected to every input and output neuron. The final layer, called output layer, contains as many neurons as there are classes. In many CNNs a softmax function is applied on the output layer, which turns the values into probabilities that sum up to 1. The class with the highest probability will be predicted.
Once the architecture is defined, the model has to be trained with a large amount of labelled data. The data is predicted by the model and the predictions are compared with the actual ground truth labels. The error is measured with a loss function. The goal of training is to minimize the loss by adjusting the weights in the kernels. This is done with a technique called Gradient Descent. A gradient vector predicts how the error would increase or decrease, if the corresponding weights were changed. The negative gradient vector points into the direction of steepest descent to the minimum error. Therefore, the weight vector is adjusted according to the negative gradient vector.

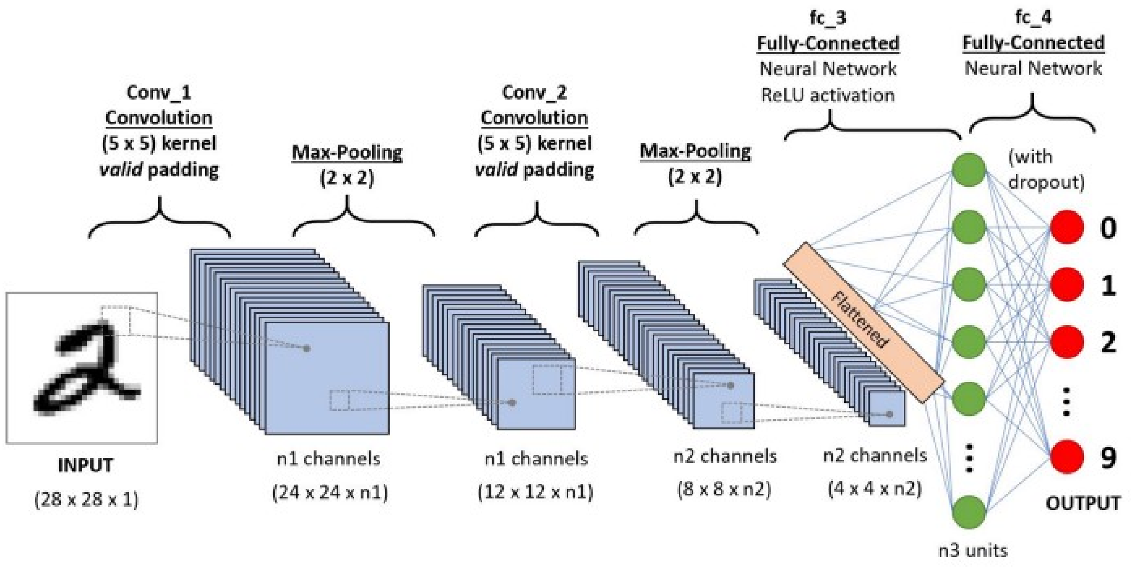


Figure 2: A CNN architecture for digit classification (Saha, 2018)

After training is finished the model is evaluated on an independent test set (LeCun et al., 2015).

2.2 Faster R-CNN

Faster R-CNNs are a type of object detection network that includes a region-proposal network to extend and improve the design of Fast R-CNNs (Girshick, 2015). This is achieved by introducing a so called region proposal network (RPN) which handles the region proposals that were calculated algorithmically before. This leads to significantly faster processing times than the previous proposals could achieve.
Faster R-CNN is a unified object detection system composed of a deep convolutional RPN and a Fast R-CNN. The Fast R-CNN classifies the region proposals made by the RPN. Both networks share a set of convolutional layers including weights, which makes computation less expensive.
An image is fed into the RPN which then outputs rectangular object proposals with an individual objectness score (probability of object or not object). The region proposals are made on a feature map outputted by a backbone CNN. A small network slides over the map and produces a low dimensional feature, which is then fed into two sibling fully-connected layers as shown in Figure 3. The *reg* layer, also called box-regression layer, predicts the bounding boxes and the *cls*, namely box-classification layer, the objectness score.
Objects come in different sizes and shapes. Instead of using an image pyramid, where a resizing of the whole image is necessary, Faster R-CNN utilizes anchor boxes. For each sliding-window location k proposals are made by using k manually selected anchor boxes. An anchor box has a scale and an aspect ratio and is centered at the sliding window. Examples for such anchors can be seen in Figure 3. The bounding boxes are then regressed with respect to the anchor boxes.
For training a binary class label is assigned to each anchor. An anchor box is

labelled positive, if it has the highest Intersection-over-Union (IoU) with a ground-truth bounding box, or an IoU higher than 0.7. It is labelled negative if the IoU is lower than 0.3.
The proposals are then fed into a Fast R-CNN (Girshick, 2015). 4-Step alternating training is used to train the Faster R-CNN. The RPN is pre-trained with ImageNet and then fine-tuned. Additionally, Fast R-CNN is also pre-trained on ImageNet and then fine-tuned on the proposals of RPN. In the third step RPN is initialized with the previously trained Fast R-CNN. However, the shared weights are fixed and only the layers unique to RPN are fine-tuned. Finally, the unique layers of Fast R-CNN are fine-tuned, while still keeping the shared weights fixed to ensure that both networks share convolutional layers (Ren et al., 2016).

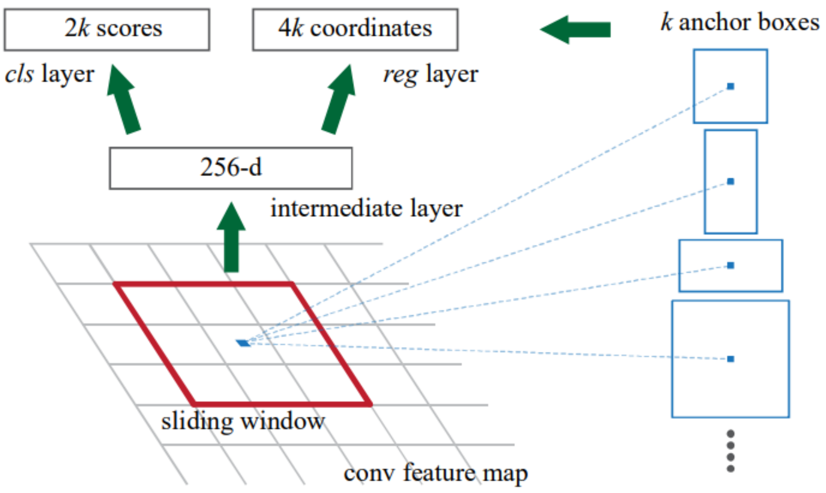


Figure 3: Region Proposal Network (RPN) (Ren et al., 2016)

2.3 YOLO

YOLO is an object detection network that, unlike Faster R-CNN, is not based on region proposals, but simultaneously predicts the location and class probabilities of each bounding box in the entire image. This results in much faster detection speeds. A first version of the network was published in 2016 (Redmon, Divvala et al., 2016) and since then many improvements have been made and new versions released. We will be using the third version of the network (YOLOv3), realeased in 2018 (Redmon and Farhadi, 2018).

The Output Predictions

Figure 4 shows how the image is divided into a SxS grid to detect the objects. For each object in the image, the grid cell that contains the center of the object is responsible for it's detection. Each grid cell predicts B bounding boxes and C class probabilities. Every bounding box has a center (x, y), a height (h), a width (w) and a confidence score, which indicates how likely it is that the box actually contains an

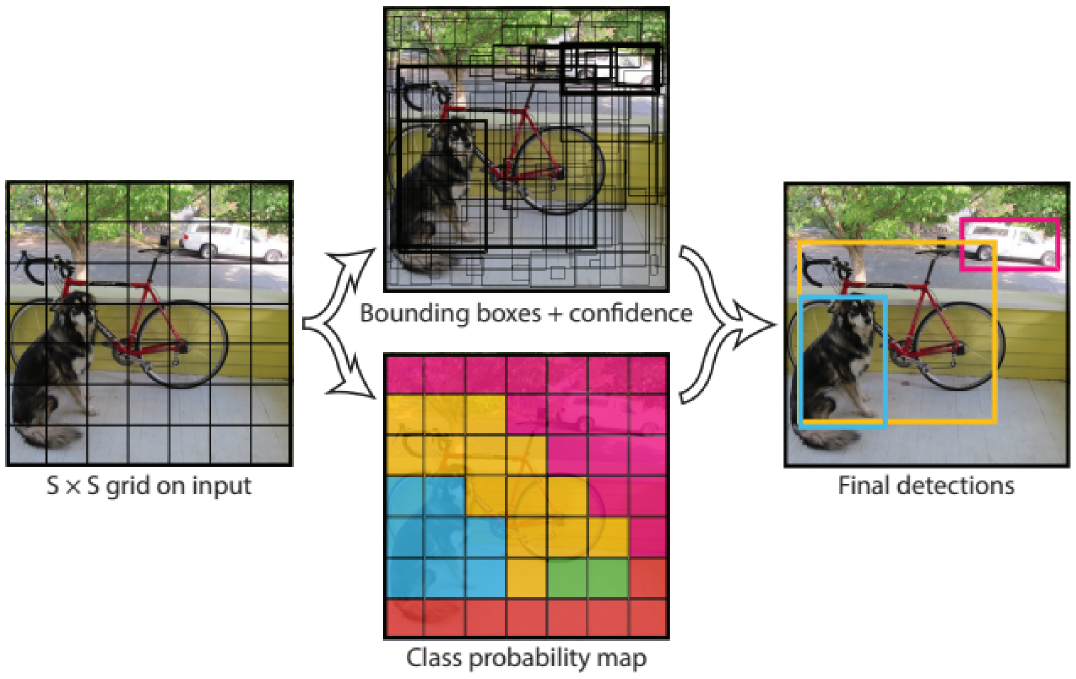


Figure 4: The image is divided into an SxS grid. Each cell predicts B bounding boxes, confidence scores and C class probabilities (Redmon, Divvala et al., 2016).

object and also how good the bounding box is. The confidence is defined as:

$$\text{Confidence} = Pr(\text{Object}) * IOU_{pred}^{truth} \tag{1}$$

$Pr(\text{Object})$ is 0, if the cell does not contain an object and 1 if any type of object is present. IOU_{pred}^{truth} is defined as the intersection over union between the predicted and the ground truth bounding box.
In the first YOLO version, regardless on how many bounding boxes a cell predicts, each cell predicts one set of C conditional class probabilities.

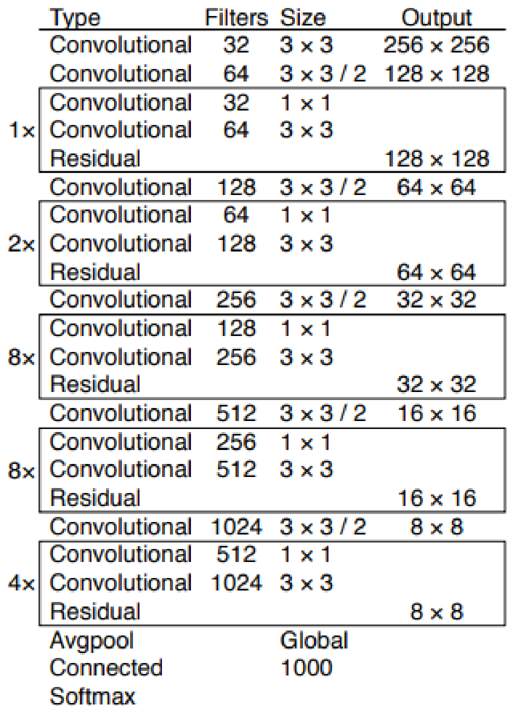
$$\text{Probability of Class i} = Pr(\text{Class}_i|\text{Object}) \tag{2}$$

Thus, the probability of class i is conditioned on the presence or absence of any kind of object in the grid cell (Redmon, Divvala et al., 2016).

However, one major drawback of YOLO was, that if smaller objects (e.g. a flock of birds) are close to each other, the network struggles to detect these objects, since it can only detect one object per cell. Thus, in version two (YOLOv2 and YOLO9000) anchor boxes, which were inspired by Faster R-CNN, were introduced. These anchor boxes allow a grid cell to handle multiple overlapping objects. However, unlike Faster R-CNN, YOLO does not use hand-picked priors, but dimension clusters as anchor boxes. Instead of manually picking the priors, k-means clustering is applied on the dimensions (aspect ratios) of the training set bounding boxes. Additionally, YOLO does not predict offsets, but constrains the location predictions to the location of the grid cell. Furthermore, the class probabilities are not predicted for every grid cell anymore, but for every anchor box (Redmon, Divvala et al., 2016).

The Network Architecture

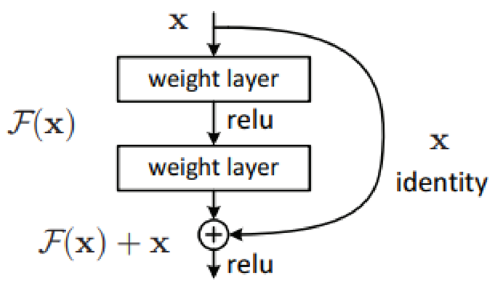
The original YOLO consisted of a feature extractor with 24 convolutional layers and fully connected layers on top to extract location and probabilities (Redmon, Divvala et al., 2016). Since the second version, the fully connected layers for detection are removed and replaced by anchor boxes (Redmon and Farhadi, 2016). YOLOv3 also uses anchor boxes for detection and a 53 layer convolutional network for feature extraction with shortcut connections, as shown in Figure 5 (Redmon and Farhadi, 2018).



	Type	Filters	Size	Output
	Convolutional	32	3 × 3	256 × 256
	Convolutional	64	3 × 3 / 2	128 × 128
1×	Convolutional	32	1 × 1	
	Convolutional	64	3 × 3	
	Residual			128 × 128
	Convolutional	128	3 × 3 / 2	64 × 64
2×	Convolutional	64	1 × 1	
	Convolutional	128	3 × 3	
	Residual			64 × 64
	Convolutional	256	3 × 3 / 2	32 × 32
8×	Convolutional	128	1 × 1	
	Convolutional	256	3 × 3	
	Residual			32 × 32
	Convolutional	512	3 × 3 / 2	16 × 16
8×	Convolutional	256	1 × 1	
	Convolutional	512	3 × 3	
	Residual			16 × 16
	Convolutional	1024	3 × 3 / 2	8 × 8
4×	Convolutional	512	1 × 1	
	Convolutional	1024	3 × 3	
	Residual			8 × 8
	Avgpool		Global	
	Connected		1000	
	Softmax			

Figure 5: Architecture of YOLOv3's feature extractor (Redmon and Farhadi, 2018).

A shortcut connection is an identity mapping x that allows the solver to skip layers. It simply adds the output from the previous layer to the following layer. However before adding, the identity mapping is multiplied by a linear projection that ensures that the identity mapping has the same output dimensions as expected by the following layer. This results in a residual block as depicted in Figure 6 (He et al., 2015).



x
weight layer
$\mathcal{F}(\mathbf{x})$
relu
x
identity
weight layer
$\mathcal{F}(\mathbf{x}) + \mathbf{x}$
relu

Figure 6: Residual Network building block (He et al., 2015)

Each residual block in YOLOv3 contains two convolutional layers: The first one uses a 1x1 kernel and the second one a 3x3 kernel.
For regularization and convergence speed, batch normalization is performed after each convolutional layer (Redmon and Farhadi, 2016). Meaning, that the outputs of these layers are normalized across each mini-batch (Ioffe and Szegedy, 2015).
In YOLOv3, more layers are stacked on the feature extractor for detection and the bounding boxes are predicted at three different scales with three anchor boxes for each scale. Thus, the detection made at the last scale benefits from the fine-grained features in the early layers as well and the network can detect smaller objects more easily. YOLOv3 also does not use a softmax in the final output layer, but independent logistic classifiers. Softmax implies that there is only one class per object, however, classes do not always have do be mutually exclusive, e.g. woman and person (Redmon and Farhadi, 2018).

2.4 SSPE

The SSPE architecture developed by Tekin et al., 2018 is a CNN architecture which is largely based on the YOLOv2 architecture. In difference to YOLO which outputs a 2D bounding box to denote the location of an object within an image the SSPE architecture outputs the 2D projection of a 3D bounding box projected onto the image plane. This projection consists of the 8 corner points of the 3D bounding box as well as the center point. Therefore, this architecture solely relies on standard color images and is able to make accurate estimations of the position and pose of specific objects without significant post-processing, which is often required in similar approaches. It does however require information on the precise dimensions of the 3D bounding box around the objects shape to calculate its 3D position using the Perspective-n-Point algorithm (PnP).
Upon receiving a color input image the network first divides the image area into a grid consisting of SxS cells with S being a hyperparameter chosen before the training process. Each of these cells is then assigned a multidimensional vector, which contains data for an estimated prediction for one of the possible objects that this model is trained to classify, as well as a class probability for each of the objects it can classify. This results in a full tensor with the dimensions $SxSxD$ with $D = 9 * 2 + C + 1$ corresponding to the $9 * 2$ bounding box coordinate values, the list of class probability values and one confidence value. From all predictions in all cells the ones with confidence values below a certain threshold are discarded leaving only high confidence predictions. The confidence is calculated according to formula 3. This is in contrast to the YOLOv2 architecture which relies on an IoU score. The confidence exponentially declines with the euclidean distance between the predicted bounding box points and the ground truth points.

$$c(x) = \begin{cases} \exp \alpha (1 - \frac{\text{distance}(x)}{d_{th}}) & \text{if distance}(x) \leq d_{th} \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

The base neural network is very similar to the YOLOv2 architecture and features 23 convolutional layers and 5 max-pooling layers overall. For the training process the authors designed a custom loss function taking into account all relevant losses for the different properties of a detection. As shown in Equation 4.

$$L = \lambda_{pt} * L_{pt} + \lambda_{conf} * L_{conf} + \lambda_{id} * L_{id} \tag{4}$$

Here L_{pt} denotes the coordinate loss, L_{conf} denotes the confidence loss, L_{id} denotes the classification loss and the different λ factors are hyperparameters. The first two of these losses are calculated using the mean-squared error function and the last one is calculated using the cross entropy error. By optimising this loss function all relevant aspects of the SSPE models output can be simultaneously improved.

The result can then be transformed into 3D position and pose estimate using the PnP algorithm and precise knowledge of a bounding box around the objects shape. The advantages of an SSPE model lie in its relatively simple architecture design and its independence from the actual 3D model for the models predictions leading to significantly faster predictions than other models with a similar objective. This is partly because it relies on a standard YOLO architecture and therefore profits from its inherent benefits of very fast predictions and high accuracy.

3 Related Work

The use of intelligent robotic systems for various tasks provides many newfound benefits and possibilities. In industrial or harsh environments, the deployment of robots to perform repetitive, hazardous or arduous tasks effectively reduces the burden on human workers and provides valuable opportunities to carry out jobs in challenging environments such as the CERN accelerator complex (di Castro. et al., 2017). Deploying robots to carry out work in personal or business environments also provides merits such as counteracting a shortage of workers as for example currently found in Japan due to its increasing age demographic, leading to the development of robot models designed to interact with humans on a social level such as the Pepper robot. Novel innovations such as the Pepper model perform their tasks in collaborative environments in sectors such as health care, education, entertainment, and business (Pandey and Gelin, 2018). For Robots to perform their tasks correctly however without endangering participants regardless of their specific field of use, they need to be able to detect, perceive and intelligently navigate their environments.

Object detection is a necessary step for a robot to perceive its environment and is a fundamental challenge of computer vision. Different approaches have been made over the years to find solutions to this problem. One approach focuses on object detection based on a detection system comprised of discriminately trained part-based models using partially labeled data. New methods for discriminative training are used in this approach which combine a margin-sensitive approach for data-mining hard negative examples with a reformulation of an MI-SVM using latent variables, defined as a latent SVM (Felzenszwalb et al., 2010).
Another approach for object detection uses RGB-D image and depth data. Here geocentrically embedded depth images are used to train convolutional neural networks. The pixels of the images detected by the network were then labeled using a decision forest approach and the output was used in an existing superpixel classification framework for semantic scene segmentation (Gupta et al., 2014).

Robots in industrial environments that share the same workspace with humans face the more extensive challenge of detecting and tracking the movements of workers in the vicinity. Real-time people detection and tracking can be performed by utilizing algorithms using both color and depth data in a cascade organization for people detection and using particle filters for tracking (Munaro et al., 2016).
A real-time RGB-D people detection and tracking system for mobile robots can also be facilitated combining visual odometry estimation, target feature extraction and nearest point position information into a vision system run on a robot operating system (Fang et al., 2017).

To extend the robots' perception for more delicate and demanding use cases that warrant intelligent robotic systems, a pose estimation system can be utilized to gain a more complex knowledge of the robots' surroundings and be able to adapt its behaviour accordingly (Liu et al., 2016). This was accomplished by di Castro. et al., 2017, where a 6D pose detection algorithm was developed that detects the position and rotation of an object using 3D camera input data and works in unstructured, difficult environments with varying luminosity values, light reflections and obstructed accessibility. Real-time human activity recognition was also achieved using a convolutional neural network which combines spatial and temporal information extracted from images acquired from RGB cameras. The trained network in this scenario was tested using a TIAGo robot for carrying out the activity recognition and performed at an accuracy of 87.05% in real-time (Mocanu et al., 2018).

4 Components

This chapter contains a short description of the various software and hardware components utilized in this project.

4.1 Software

4.1.1 MMDetection

MMDetection is a framework developed by the OpenMMLab project with the aim to simplify the process of designing and training neural networks for detection systems. It is based on the MMCV library developed by the same project and fundamentally relies on PyTorch for implementing and training the varying neural network architectures.
It provides a sizable collection of premade configuration files for different neural networks which have been proposed and tested in the field of automatic object detection in recent years. These can serve as the basis to train similar architectures or their framework can be utilized to design entirely new architectures without requiring extensive programming beforehand.

4.1.2 ROS

The Robot operating system (ROS) is a flexible framework consisting of a collection of tools and libraries to provide software developers with the necessary functionality to create robot applications and encourage collaborative software development(Open Robotics, 2021c,Open Robotics, 2021a). It is licensed under an open source BSD license and provides device drivers, hardware abstraction, visualizers, libraries, message-passing, package management among other semantics expected of an operating system(Open Robotics, 2021c).

Thee peer-to-peer network of ROS processes called the computation graph relies on different concepts to provide data to the graph (Open Robotics, 2021b). The ROS Master provides name registration and lookup to the parts of the computation graph called Nodes (Open Robotics, 2021b). The ROS Nodes perform the computation and communicate with one another via topics, RPC services, and the Parameter Server and usually represent different areas of a system. The Nodes communicate with each other over messages that are routed over a transport system with a publish/ subscribe functionality. To send a message, a Node publishes a ROS Topic (Open Robotics, 2021b). A Topic classifies which kind of data is transmitted in the content of the message (Open Robotics, 2021b). A Node that processes incoming data then subscribes to the topic containing the data it requires (Open Robotics, 2021b).

4.2 Hardware

4.2.1 Intel Realsense Depth Camera

The Intel Realsense Depth Camera D435i is a stereo vision depth camera system that offers high resolution 1920x1080 RGB image data as well as refined 1280x720 active stereo depth data with a depth diagonal field of view of over 90° and a range of up to 10 meters, varying with lighting conditions (Intel Corporation, 2021a). Offering both a comparatively low price of 199$ and its easy-to-use setup design for portable indoor and outdoor use cases, it is an optimal component for educators, makers, hardware prototypes and software development (Intel Corporation, 2021a). It is comprised of a stereo depth module, a complete depth camera integrating D4 vision processor, RGB sensors with color image signal processing and an intertial measurement unit (IMU) to refine its depth awareness and detect movement and rotations in 6 degrees of freedom (Intel Corporation, 2021b). As a model of the Intel RealSense D400 series, it is supported with a cross plattform and open source Intel RealSense SDK 2.0 (Intel Corporation, 2021a). Overall the Intel RealSense Depth Camera D435i is an ideal component for applications such as object recognition and robot navigation, providing both high quality image and depth data as well as a solid ROS and ROS2 integration and an easy integration with various programming environments (Intel Corporation, 2021c).

4.2.2 ZED 2 Camera

The ZED 2 stereo vision depth camera is the first stereo vision camera that offers an on-board neural network with a body-tracking and a spatial object-detection functionality that provides the user with 2D as well as 3D bounding boxes marking the location of the detected object or person, aiming to reproduce human vision and bring stereo perception to new heights at a price of 499$ (StereoLabs, 2021b). It offers a high quality video stream with a resolution of up to 4416x1242 at 15 frames per second and a maximum frame rate of 100 frames per second at a resolution of 1344x376. At a range of up to 20 meters, it offers equally high resolution depth data at a frame rate of up to 100 Hz and a maximum field of view of up to 110°(H)x70°(V)x120°(D) (StereoLabs, 2021b). It is also equipped with an accelerometer, a gyroscope, a barometer, a magnetometer, a temperature sensor and a six degree of freedom visual inertial stereo simultaneous localization and mapping(SLAM) with advanced sensor fusion and thermal compensation (StereoLabs, 2021b). The ZED 2 stereo Camera is provided with a ZED ROS wrapper to provide access to camera data such as the depth map, a colored 3D point cloud, pose tracking, spatial mapping and other data (StereoLabs, 2021a). Considering the capabilities provided by its depth camera sensing and AI functionality and compatibility with ROS, it is a valuable component for use cases involving object detection and robot navigation. However an external processing unit is required to run the extensive on-board network, in this project an Nvidia Jetson GPU workstation was used for this purpos (StereoLabs, 2021b)

4.2.3 Nvidia Jetson

The Nvidia Jetson AGX Xavier Developer Kit is a small GPU workstation that was designed to ”deploy end-to-end AI robotics applications” (NVIDIA Corporation, 2018). It supports important software libraries and frameworks, such as the NVIDIA JetPack SDK, CUDA, cuDNN, PyTorch and TensorFlow (NVIDIA Corporation, 2018).

5 Conceptual Design

Our objective is to develop a real-time 3D object detection system in a ROS environment that is able to detect three different types of robots and humans in a laboratory environment. This chapter will give an overview of the conceptual design including requirements, tasks and components. We will explain a detailed implementation of the system in a later chapter.

5.1 Requirements

For our system to work we have four major requirements that can be further divided:

- First, we need a reliable object detection module that can classify the three different robots, as well as humans, and outputs the exact location of the objects in the camera image.
- The second big requirement is, that we can obtain depth information from the camera and convert the pixel coordinates of the objects into real world coordinates that can be used for navigation.
- Additionally, all of the components have to be integrated in ROS and function together.
- Finally, this has to be done in real-time since the robot has to make quick decisions when navigating in a collaborative environment.

5.2 Tasks and Components

To fulfill the requirements we divide the project into individual tasks. The first task is to generate an annotated data set of robot images for detection. We then need to implement an object detection network and train it on the generated data set. We want to set up two different camera systems and compare their performance. Therefore, we need to integrate both cameras as well as the object detection module into two different ROS environments and detect the objects in the camera images in real-time. Once we obtain the 2D pixel coordinates of the objects, we convert them into 3D real world coordinates. Finally, we compare the performance of both setups and decide which is more suited for real-time applications. An overview about the most important task is also pictured in Figure 7.

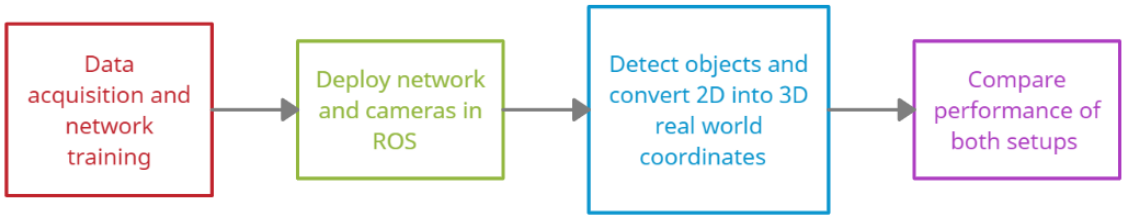

Data acquisition and network training
Deploy network and cameras in ROS
Detect objects and convert 2D into 3D real world coordinates
Compare performance of both setups

Figure 7: Overview of the major tasks in this project. We first generate a data set and train a network. Then we deploy the network and the cameras in ROS. Afterwards, we use the network to predict the locations of objects in the camera videos and convert the 2D into 3D coordinates. Finally, we compare the setups.

Generating a data set and training an object detection module

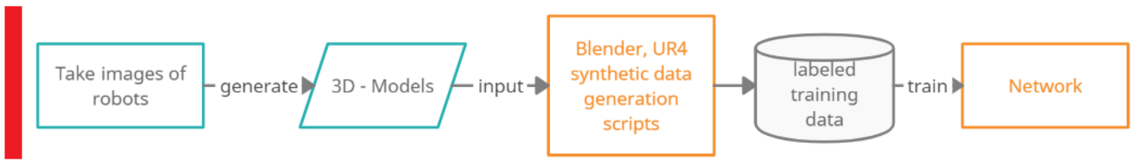


Figure 8: Data acquisition and network training pipeline. Generate a data set and train a object detection model

Figure 8 shows the components of the data acquisition and network training pipeline. We generate 3D models of robots from images input them into the Blender data generation tool, which outputs a data set of annotated images. We then use this data set to train an object detection network.

Integrating the cameras and the object detection model into ROS

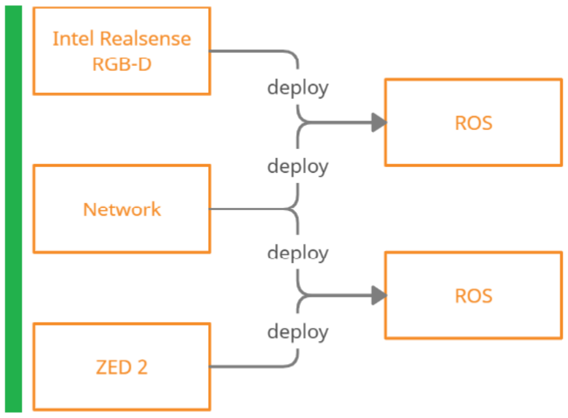


Figure 9: Deploy each camera (Intel Realsense and ZED2) in two different ROS environments with the previously trained network.

We will work with the RealSense™ Depth Camera D435i and ZED2 stereo stereo camera, which we integrate into ROS. Since we want to compare them, we use two different independent systems, as shown in Figure 9. However, we use the same trained network.

Detect objects and convert 2D coordinates into 3D world coordinates

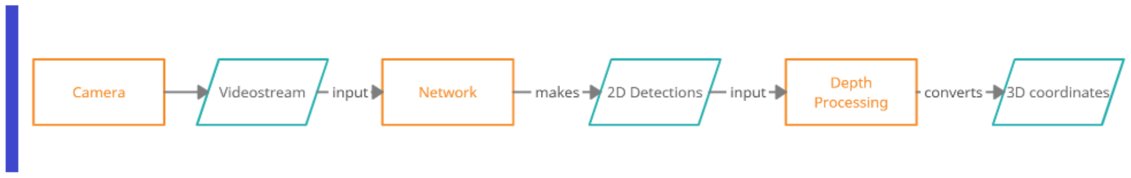


Figure 10: Pipeline for making 3D object detections within ROS

Figure 10 shows a pipeline for 3D object detection in ROS that is similar for both systems. The camera outputs a video stream which is the fed into the trained network. The network outputs the predicted 2D coordinates in the image and these are then processed by another computed, which extracts the depth and converts the 2D pixel coordinates into 3D real world coordinates.

6 Implementation

The implementation of this project consisted in a large part of integrating multiple complex software components into a workflow to create a full pipeline ranging from the acquisition of a robots 3D model up to the deployment of a position estimation system in ROS. To achieve this the varying system components had to be set up, configured and tested to enable the transition of the various outputs into the next component.

6.1 Data Generation

To supply the training process of the neural networks with a sufficient amount of training data an artificial data generation tool was utilized. This tool was developed by Leon Eversberg at the IGNC chair and uses a set of Python scripts to interface with Blender to generate training images and corresponding labels. Blender is a 3D editing and animation software which, using the aforementioned tool, was utilized to create artificial 3D scenes in which an arbitrary object of interest could be positioned. The tool then positions a camera at randomly selected positions to image the 3D object from different angles and store an image of this view as a training image. Because Blender manages the positions of all involved objects it can then export the labeling information corresponding to the generated image.
To utilize this tool effectively a 3D model of the object of interest with a high enough degree of detail was required. To achieve this we relied on Autodesk ReCap™ which is a cloud based processing tool that takes a video file as input and outputs a corresponding 3D model which maps every pixel in the videos frames to a 3D coordinate. By taking a separate video for each robot in which we would slowly move in a circle around the robot we could therefore generate accurate 3D models from real world footage. The resulting 3D models surface information usually contained some imprecisions and bumps, but the mapping of color information is performed at a high degree of detail resulting in accurate visual representations of the original objects. The resulting models can be seen in Figure 11.
To enable the generation of training data using these models both the model and the data generation tool had to be configured accordingly. The models generated by Autodesk ReCap™ were non-aligned with the respective coordinate system. To achieve an alignment they had to be edited with a 3D editing tool so that their frontal lowest point would coincide with the origin and all faces of the 3D model would be located at positive coordinates. This is important because blender data generation tool estimates the extend of a 3D bounding box around the model by taking the maximum and minimum coordinates of all vertices in the model and the direction of the bounding box is assumed to be along the z-axis. Additionally, it might be necessary to adjust configuration parameters in the file ”config.py” in the base path of the Blender data-generation tool to optimally align the camera and the object of interest. This is also where the reference path, from which the 3D model will be loaded, needs to be updated to correspond with the current local file structure. The output training data and labels are then stored to the path ”./DATASET/object/”. The model simultaneously generates 3D label information

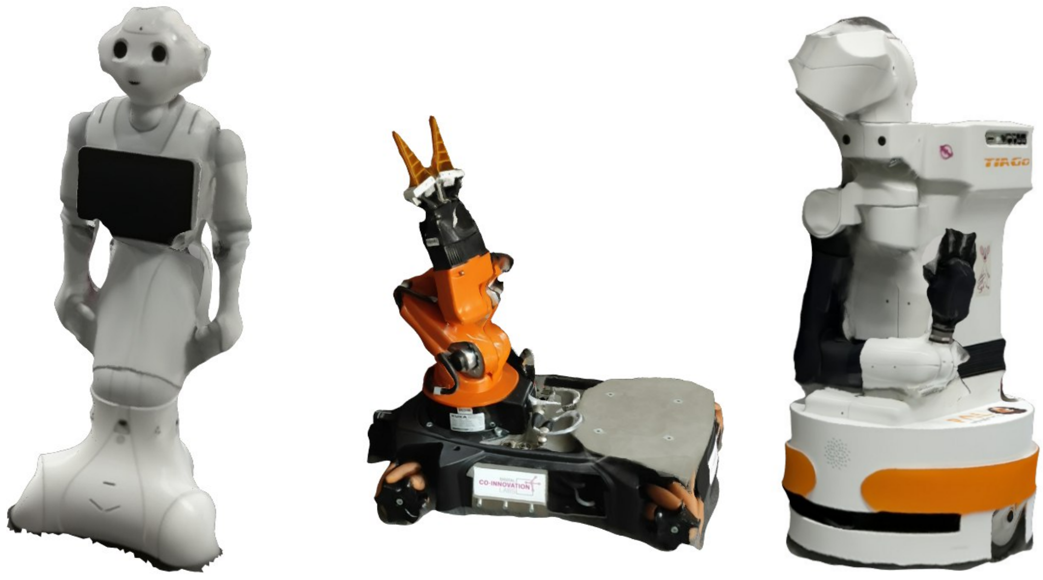


Figure 11: The generated 3D objects acquired from the three different robot models.

as well as 2D bounding box labels. While the former is stored as separate files corresponding to the respective image the latter is stored in the COCO dataset format in one JSON file called "annotation.json". The generated images as well as their respective annotation file then need to be moved to a separate folder to be used in the training process.

6.2 Detection System

In the process of designing a detection system to locate the different robot models in a 3D environment two different approaches with varying neural network architectures were tested. These detection system designs were utilized to achieve the localization of a robot within the cameras environment.
The first architecture tested in this context was a the Single Shot Pose Estimation network which has been originally developed and proposed by Tekin et al., 2018 and has been described in Chapter 2.4. To train this architecture we utilized a clone from the IGNC chairs SSPE repository and trained this network with our own generated training data. This setup manages the training process within a docker container which has to be build with the training data previously placed in the respective data folder. To specify the location of training and test images the training process relies on text files in which the paths to the images belonging to the respective class are specified. The label files corresponding to each image are stored in such a way that they can be identified and assigned to their respective image solely based on the filename and directory structure. Because a sufficiently accurate SSPE architecture would however require a huge amount of training data and training time, it was deemed to be too cost intensive to be trained on the hardware locally available in this project.
Therefore, we decided to base the detection system on a 2D bounding box detector and infer the positional information of the object of interest by extracting the depth

of pixels within the bounding box from the depth information provided by the camera. Consequently, the complexity of the detection is simplified by mainly relying on the available depth image data provided by both camera models utilized in this project. The pixels should approximately coincide with the center of mass of the robot's geometry. For Pepper we used the center of the bounding box. To train our networks we utilized the outputted 2D label information from the Blender data generation tool in the COCO data set format. The label information is stored in a JSON file which contains a multitude of definitions relevant for the training procedure. It defines a set of classes which are of interest to the detection process. In our test setup this was set to one common object class but could be extended to incorporate the various different robot types in a full data set for all robot models. It also contains a list of all images in the data set and assigns each one an ID to assign it to its corresponding bounding box. These are stored in a separate list and also contain the same ID as the corresponding image as well as a bounding box list which contains the upper left point of the bounding box and its maximum extend in x and y direction. These are then loaded during the training process as ground truth information for the optimisation procedure.
To split the full COCO annotation JSON file into smaller files containing training, validation and test set respectively we utilized a small python script by Artur Karaźniewicz called "cocosplit.py" which simply loads the respective information from one JSON file and rearranges it in specific ratios into smaller files. For our training runs we utilized a split of 80% training data, 10% validation data and 10% test data.

For training the models we relied on the MMDetection framework which is described in more detail in Chapter 4.1.1. The training configuration are defined in a python file which contains information about the model layout, various model parameters, the location of the data set JSON files and how the data should be loaded into the underlying training procedure. It also contains definitions about the location of the data set JSON files and how the data should be loaded into the underlying training procedure. We tested two different neural network architectures, namely Faster R-CNN and YOLOv3 both described in Chapter 2, and evaluated their general performance on one robot model to decide which one might be more suited to the specific task of this project. A more detailed description of their performance can be found in Chapter 7.1, but mainly due to the significantly faster inference speed of the YOLO architecture we decided to rely on this architecture for all robot detection systems. Therefore, we additionally trained two similar YOLO models with the same architecture on the two data sets for the KUKA youBot and the TIAGo respectively.
All training was performed on a Nvidia GeForce GTX 1660 SUPER graphics card which contains 1408 CUDA cores. As this is a standard middle price range graphics card with a main focus on the gaming market it does not deliver the best possible performance for artificial intelligence applications. This meant that one full training cycle consisted of several hours of training time for each separate model. These lengthy time requirements to train a particular model architecture resulted in limitations to all parameters in the training process ranging from the amount of images in a dataset up to the time slots in which training could be performed.

6.3 ROS Infrastructure

6.3.1 Intel Realsense Integration

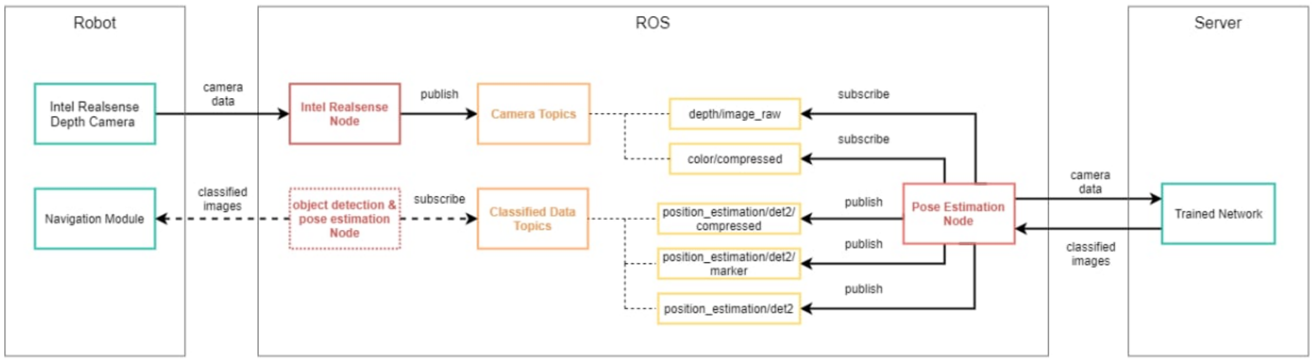


Figure 12: The communication graph showing the nodes and topics in the ROS setup for the Intel Realsense camera based detection system.

To transmit image messages and positional information between the different components a ROS infrastructure, which is visualized in Figure 12, was set up. This consisted mainly of two major nodes, one managing and sending the cameras outputs and another one interacting with the detection system.
The camera node relies on the official Realsense ROS integration and publishes a series of camera related topics over the communication network, including topics for depth and color images, camera information for both image types and inertial measurement unit outputs among others. From these topics only the depth frame and the color image topics were selected to be received by the Pose Estimation node.

The detection system was provided by a so called position estimation node which runs in a docker container of a host computer which should have significant processing capabilities and preferably contain an Nvidia GPU. The code on which this container is based on was originally developed by the IGNC chair.
The docker container first installs a number of required libraries for the functionality of the detection system and then loads the ros node from the script file "start_ros_node.sh". This script first configures the relevant environment parameters like the location of the ROS installation and the IP address of the computer hosting the roscore. This needs to be adjusted by the user manually before loading the docker container. It then runs the python code from "mmdetection_ros_node.py". This script contains the definition of a class dedicated to handle the loading of the neural network model and supply it with the input from the respective ROS nodes as well as sending the output of the network on a new topic.
It first defines a set of internal objects to interface with the respective ROS nodes for sending and receiving messages and then connects them to their respective topics. It then proceeds to load the models configuration file as well as its weights from the "./model" folder. This is followed by the definition of a callback function which runs indefinitely from startup. It gets called when something is published on both the color and depth image in a specified time frame and can then be accessed locally as

if both images would have been published synchronously. The received color image is then analysed by the respective neural network model currently loaded and if anything is detected this is stored locally into a Detection2D object from the vision messages library.
If the confidence of the detection exceeds a specified threshold it then infers the 3D position of the detected object in relation to the camera. This is calculated based on a specific pixel location within the detection area depending on the particular object of interest and can vary depending on that object's geometry. The 3D location of the object can then be calculated based on this pixel location, the previously stored camera intrinsics matrix and the depth value in the corresponding location in the depth image the. The camera intrinsic information currently needs to be stored manually for the particular camera model in use but it is also supplied by the camera itself in a topic called "/camera_info". But to prevent any delay in the receival of image frames and because this information is static it was decided to not include it in the received topics and instead store it statically. The system could however be expanded by an automatic configuration if required. The calculated 3D location is then formatted into a Marker message and published to ROS over its respective topic together with the Detection2D message and an image frame annotated with the predicted bounding box. This can then be subscribed by any node requiring such positional information to infer advanced information or calculate predictions of the respective object behaviour.

6.3.2 ZED 2 Integration

We implemented a different setup in ROS for the ZED 2. Figure 13 shows the communication between the nodes in the ZED 2 setup. One major difference between the ZED 2 and the Realsense setup is, that all calculations are done locally, since we utilize the Nvidia Jetson and both the camera and the Jetson are directly mounted on the robot.
The Jetson runs a ROS environment with a Zed camera node, a YOLO object detection node and a depth processing node.
Stereolabs already offers a ZED ROS wrapper that can be directly downloaded

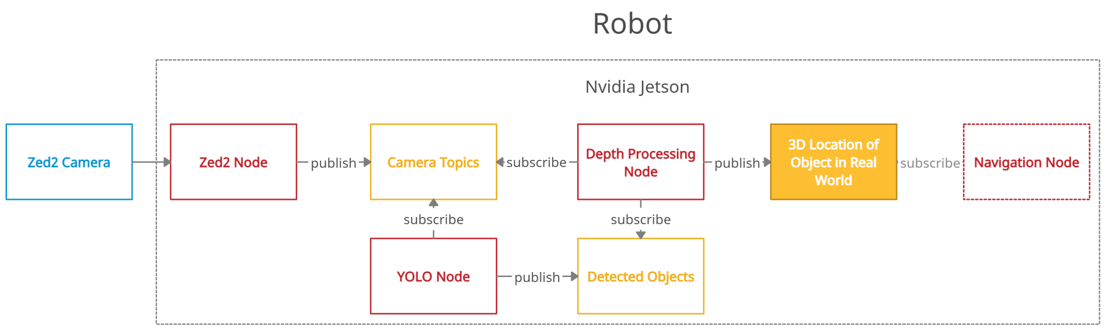


Figure 13: ZED 2 Set-up

from GitHub. The wrapper allows to use the camera in ROS and provides access to many types of camera data, including general camera information, a depth map and

the left and right camera images. Furthermore, the camera settings can be changed in the wrapper config files. For example, image resolution, fps and depth quality can be adjusted to improve the overall setup performance. We used the HD720 resolution, a depth perception up to 0.5m and set the depth quality to "QUALITY", which is not the highest quality mode, but a good trade-off between performance and sensibility.
We chose YOLOv3 as the object detection model and used a ROS package by legged-robotics that offers a YOLO integration for ROS. The package can be cloned from GitHub. We adjust the config files so that the node subscribes to the left camera image, which is published by the ZED node. YOLO then detects the objects and publishes the bounding boxes. We also enabled CUDA and GPU for faster detection since we have both available.
The third node is responsible for computing the depth of detected objects and publishes the 3D coordinates of all objects found. The depth processing node subscribes to four topics in total.

- zed_node/depth/depth_registered
- zed_node/depth/camera_info
- darknet_ros/bounding_boxes

The first two topics are published by the ZED node. *zed_node/depth/depth_registered* contain the depth map and *zed_node/depth/camera_info* information about the camera, such as the intrinsic parameters. The final topic *darknet_ros/bounding_boxes* is published by the YOLO node and contains the bounding boxes of the detected objects.
To compute the depth of each object we need the depth map, the camera information and the bounding boxes. Therefore, we use the ApproximateTimeSynchronizer of the message_filters library in ROS. The approximate time synchronizer can receive different types of messages from different sources and only outputs them, if they have approximately the same time stamp. Once all the necessary information arrived, the depth can be calculated.
We calculate the center of each bounding box that arrives and access the depth value in the depth map. However, the depth can be $-\text{inf}$ (negative infinity), inf (positive infinity), or NaN (not a number), if the object is too close to the camera, too far away or if the pixel is occluded. One workaround would be to activate the FILL mode in the ZED 2 camera settings. This mode computes a fully dense depth map and approximates a depth value for every pixel. It fills holes and occlusion in a depth map, but since this is only a approximation it can distort the actual distance. Furthermore, it requires more computation and leads to a drop in FPS. Thus, using the FILL mode is not suitable for real-time detection tasks where an exact distance is necessary. Instead we iterate the five pixels to the right and below the center pixel and pick the first depth value that is not NaN. If the depth value is still NaN we will not publish the coordinates, since they do not contain processable information. In most cases, however, one of the pixels has a valid depth value.
The depth is already outputted in meters, but the x and y variables are only pixel coordinates and have to be converted into real world coordinates X and Y. Since

we already have the real world depth value, converting the two values is easy as shown in Equations 5 and 6. cx and cy are the coordinates of the principal point of the camera and fx and fy is the focal length. These intrinsic camera parameters are stored in the calibration matrix which is contained in message of the *zed_node/depth/camera_info* topic.

$$X = \frac{(x - cx)}{fx} * depth \tag{5}$$

$$Y = \frac{(y - cy)}{fy} * depth \tag{6}$$

Once we obtained the 3D real world coordinates of the object we define a ROS marker for each object and append it to a MarkerArray. This MarkerArray is then published as "visualization_marker_array". However, all the markers in the array are deleted as soon as new 3D object coordinates are computed to keep the array up to date. We use a ROS timer to manage the publishing of topics. A callback function processes the messages of the subscribed topics and updates the MarkerArray, while the node publishes the last data it received every 0.01 seconds. The markers can then be visualized in RVIZ.

7 Evaluation

We will now present and evaluate the results of our implementation. First, we focus on the performance of the individual network architectures on our data set. Finally, we will evaluate and compare the set-ups for both cameras, the Intel Realsense and the ZED 2.

7.1 Networks

In the scope of this project multiple neural network architectures to detect the different robots were utilized and their resulting performance is presented in the following. At first a SSPE network was setup and trained on a dataset consisting of 1500 images of the Pepper robot. While this model was able to detect the location of the Pepper robot in most images, it could often not correctly predict the rotational pose information. This was especially true for real world data as can be seen in Figure 14.

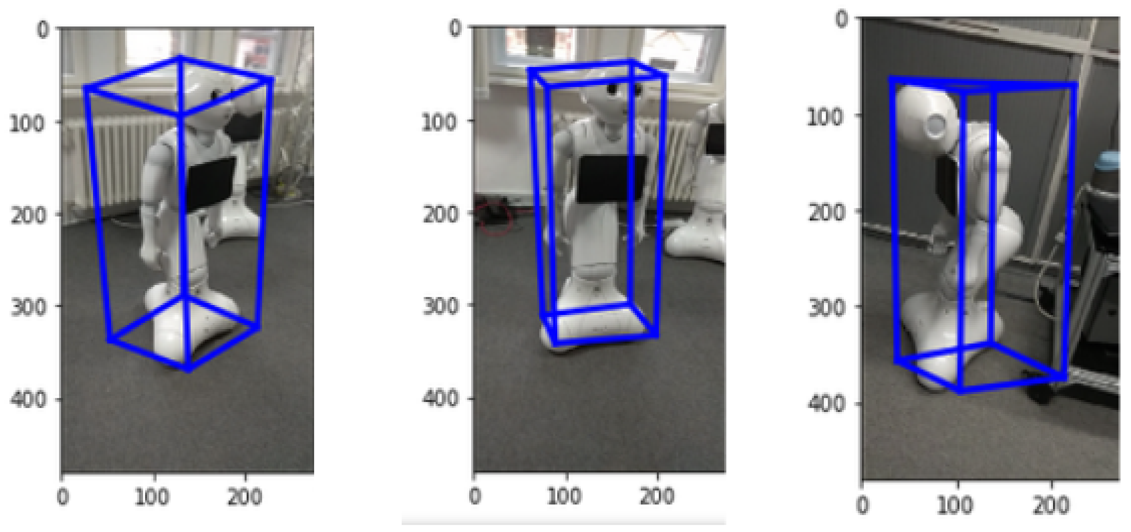


Figure 14: Example predictions of the SSPE model on real world images from the laboratory environment. Notice the misaligned bounding boxes.

The performance of this SSPE design would improve significantly with larger and more diverse training data sets. This would however require such an amount of images and training time that it would likely exceed the computation and potentially even storage capabilities of the system utilized in this project as both of these were only available with a relatively limited capacity.
Due to these limitations we decided to utilize a 2D detection system to detect the robot in color images and infer the positional information from the depth data provided by both cameras. For this we selected the YOLO and the Faster R-CNN architecture, because they are two of the most popular object detection models. While testing these two different 2D detector architectures for the Pepper robot on the respective test data set a few important differences can be noticed. The results

can be seen in Table 1. Firstly, it is directly apparent that the YOLO architecture processes inferences significantly faster then the Faster R-CNN architecture with a difference of more than three times the speed. At the same time the average precision and recall indicate a better approximation of the ground truth bounding boxes by the Faster R-CNN model. The average precision is calculated by taking the ratio of true positive predictions to the sum of all positive predictions. A prediction is counted as positive here when the IoU of the ground truth and the predicted bounding box overlap to more than 75%. Mean average recall on the other hand depends on the ratio of true positives to the sum of true positives and false negatives but it expands upon the simple IoU constraint by taking the average value over multiple IoU definitions.

	FasterRCNN	YOLO
Inference Speed	9.1	23.7
Average Precision (IoU≥0.75)	0.927	0.836
Mean Average Recall	0.858	0.744

Table 1: The performances of the FasterRCNN and the YOLO neural network architectures trained to detect the Pepper model.

Binding detection accuracy to the overlap of the ground truth and predicted bounding boxes does however not necessarily directly correlate with a better capability to detect the robots precise location within the image. This can be seen when analysing some example outputs from the models as are shown in Figure 15. Here, a few of the bounding boxes do not overlap to a sometimes significant degree which we noticed in a large amount of test images. The predicted bounding box does however still correctly locate the robot which implies that this metric does not necessarily directly correlate with a higher detection accuracy.

Because of these factors and to a significant part due to the speed requirement we decided to adopt YOLO as the detection system for all robot models. The different YOLOs were then trained without significant modifications to the architecture and on data sets of the same size and with background features from the same environment while only the different 3D robot models were exchanged. Table 2 shows the same performance indicators as Table 1 for the different models.

	Pepper	Kuka Youbot	Tiago
Inference Speed	23.7	24	23.9
Average Precision (IoU≥0.75)	0.836	0.911	0.857
Mean Average Recall	0.744	0.813	0.728

Table 2: The performances of the YOLO models trained to detect the three different robots.

By testing the dedicated networks for each robot model on their original video from which their 3D model was generated the confidence of the network in its predictions could be partly tested in a real world scenario. Table 3 shows the average confidence with which the model detected a robot in all frames of one of these videos. This

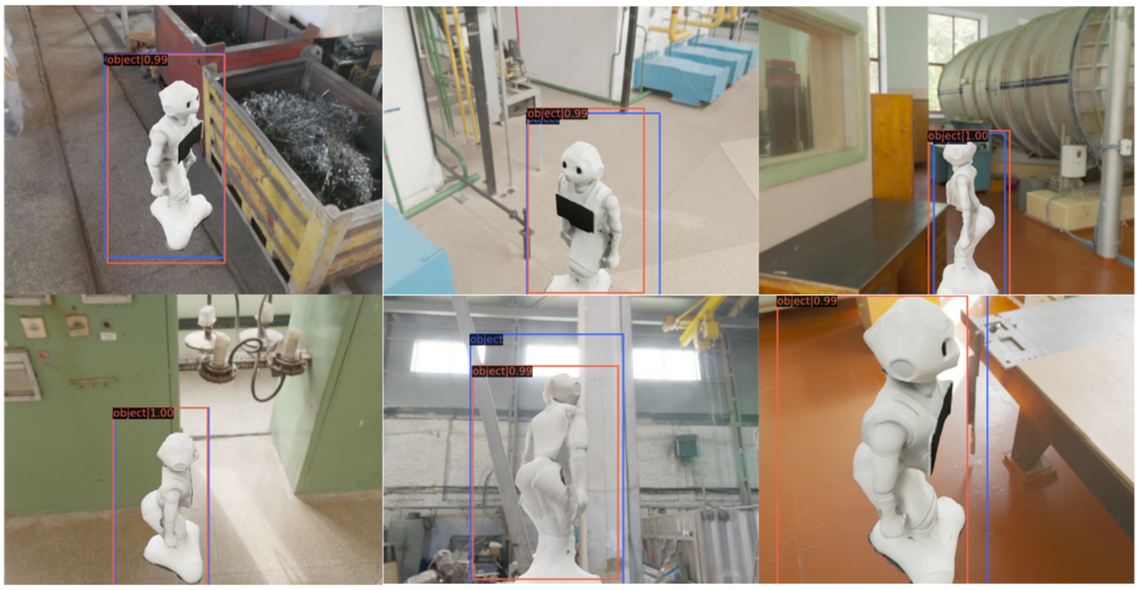


Figure 15: Example predictions of the YOLO Pepper model on the respective test data set. The predictions are marked in orange and the ground truth bounding boxes in blue.

implies that the networks are able to generalize on the 3D models geometry and that it is able to detect the robot from varying angles and viewpoints. This is because the view on the robot constantly changes within these videos with a very high temporal sampling so that the average confidence is calculated for a multitude of slight variations in viewing angle and distance relative to the robot. The different images in Figure 16 show a few examples from the predictions derived from these video frames.

In summary, the 2D Detectors seem to be able to generalize to the respective robots

	Pepper	Kuka Youbot	Tiago
Average Confidence	0,9057	0,987	0,933

Table 3: The average confidence of the three models for tested on their respective original video from which their 3D model was generated.

geometry, but this is currently limited to only one pose and is also not necessarily invariant to illumination changes. This could have a significant impact on actual operations, however tests with Pepper images from very different environments have mostly resulted in appropriate predictions by the respective network indicating that the extend of these problems could not easily be measured and would have to be taken into account in future setups. Both of these problems could be addressed by introducing variations of the pose of the 3D object and the lighting conditions, or optimally both, to incorporate these differences in the training process.

Another problem with the current setup is that each robot is at the moment only detected by one dedicated network trained for this specific task. That could be addressed by unifying the separate data sets into one by generating one JSON file encompassing a collection of paths to all images. This would also require adjustments to the definition of the network model and to the amount of classes the model is able to detect. This was however not tested in the scope of this project as increasing the data sets size by at least three times would likely exceed the capabilities of our available training hardware and lead to significant increased training times.

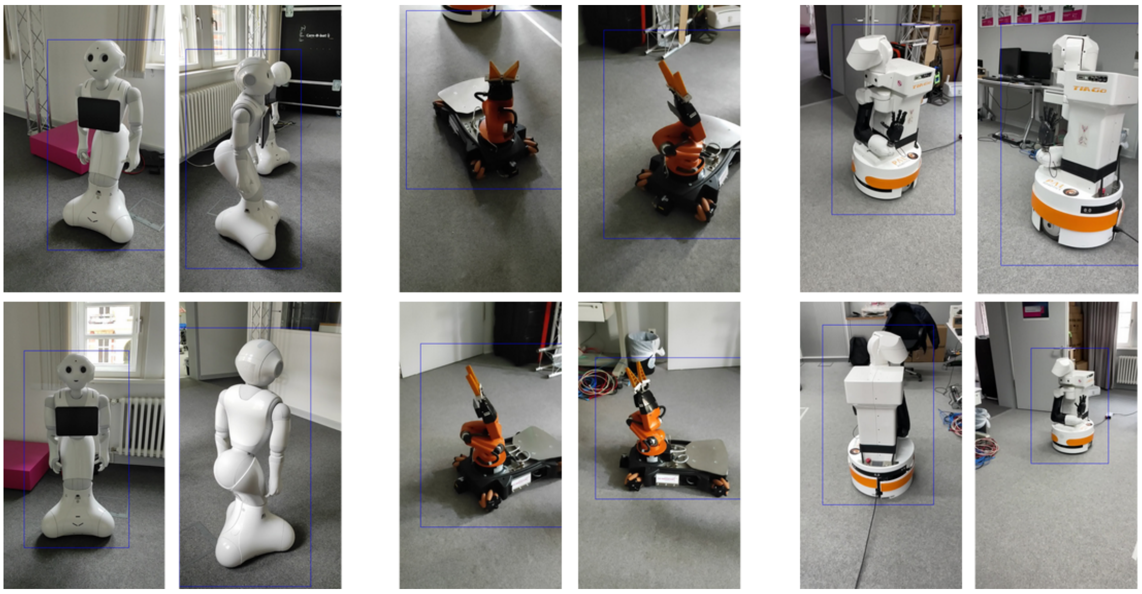


Figure 16: Example predictions on images from the laboratory environment showing the networks capabilities from different angles and viewpoints.

7.2 Cameras

The ZED 2 is with a price point of 449$ relatively expensive compared to the Intel Realsense, which only cost 199$. Additionally it requires a more expensive setup with a GPU.

The ZED 2 offers a higher resolution, many more sensors, can perceive depth up to double the range of the Intel Realsense and comes with plenty of on-board software. However, those features are not necessarily needed for all applications. Thus, one objective of this project is to compare both cameras and figure out which is best for what kind of application.

We were able to implement a working setup for both Realsense and ZED 2. Nevertheless, the setup for the ZED 2 is much faster than the Realsense setup as shown in Figure 17. The left part of the chart shows the publishing frequency of the 2D object coordinates for the Realsense setup with Faster R-CNN and YOLO and for the ZED 2 setup with YOLO. Faster R-CNN is slower than both YOLO implementations. However, ZED 2 YOLO is with a frequency of around 28Hz more than twice as fast as Realsense YOLO with a frequency of 13,71Hz.

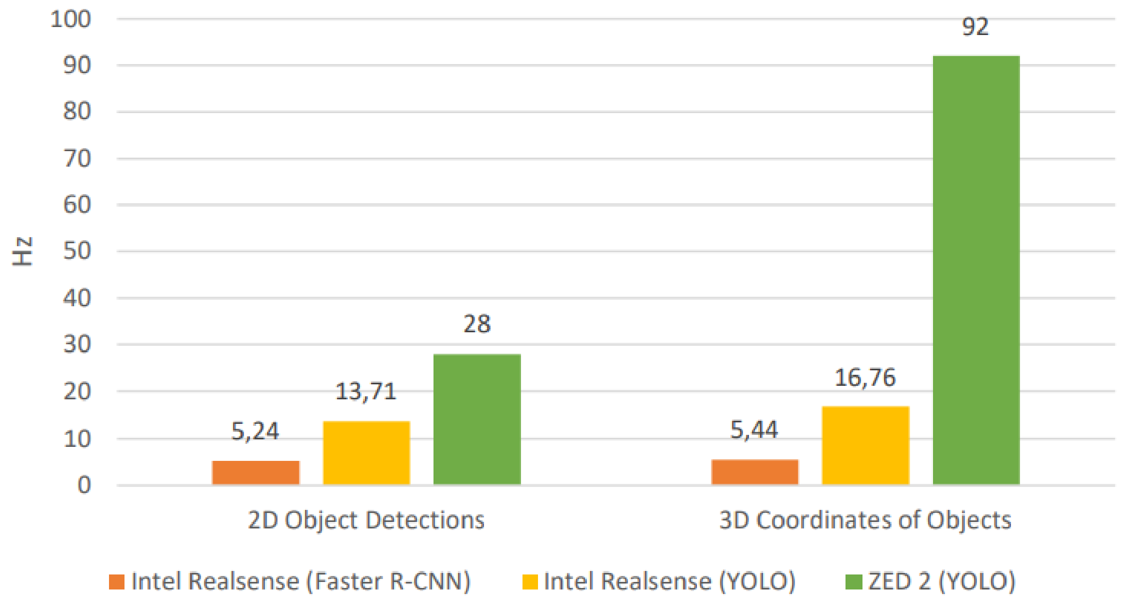


Figure 17: Comparison of publishing freqency for different ROS-topics in the Realsense and ZED 2 setup.

This is due to the fact that in the Realsense setup the camera video has to be send to a server first before detections can be made. This results in a delay, that depends on the speed and bandwidth of the given network. On the other hand, the ZED 2 is directly connected to the Nvidia Jetson and all computations can be made locally on the robot, which speeds up detection.

The right part of the chart (Figure 17) shows the publishing frequency of 3D coordinates for the detected objects. The computation of 3D coordinates depends on the 2d detection of objects and can only be done, once the detected objects have been published. Both setups (Realsense and ZED 2) publish the 3D coordinate topic even when there are no new detections. However, the ZED 2 topic has a much higher

publishing frequency (around 90Hz), than the Realsense topics. This indicates that the ZED 2 setup is more sensible to changes in the object detections and therefore more suited for real-time detection.
We conclude that the Realsense is the better camera for lightweight robots with minimal processing setup. It comes at a lower price point and because it does not require a GPU on the robot, it has lower setup costs overall. However, it needs to be connected to a server at all time and maintaining a server and a network can be costly depending on the provider. Furthermore, needing an external server poses a problem for real-time detection. The speed and bandwidth of the network, that is used to exchange information between robot and server, can cause significant delays, which is not desired in a real-time task. Moreover, complete connection losses can occur, which is very problematic and makes the system less reliable.
On the other hand, the ZED 2 setup has higher hardware costs, but does not require an external server. So there are no hidden network server or network costs. Thus, the ZED 2 setup may actually be less expensive in the long run. Apart from the pricing, our results show that the ZED 2 setup is much faster and better suited for real-time application than the Realsense setup. Furthermore, it comes with a lot of features, such as more sensors and on-board software, which can be further utilized depending on the application.
In conclusion, we think that the ZED 2 is the better camera for applications where fast processing and reaction times are needed, such as robot navigation.

8 Conclusion

In this project our group integrated and tested a pipeline to train object detection systems from the data acquisition up to deploying the functioning detection models within a ROS environment.

We tested and deployed a series of tools to achieve this goal and integrated specific components with one another. This workflow included the generation of a 3D model from manually acquired video data as well as the artificial generation of training images from these models.
Furthermore, various training procedures for different neural network architectures were setup and tested and the final detection models were deployed in ROS to receive the image input from either the Intel Realsense Depth Camera D435i or the Stereolabs ZED 2.
We chose YOLO over Faster R-CNN as the preferred object detection model, because it offers the best trade-off between detection speed and accuracy for real-time applications.
Additionally, we compared the two camera systems. The Intel Realsense system relies on an external server and a network connection for computation, while all computations are done locally on the robot in the ZED 2 system, making detection more than twice as fast. Another drawback of the Intel Realsense system is that objects can not be detected if the network connection is lost, which makes it less reliable. Thus, the ZED 2 perception system is more suited for real-time applications and also more reliable, which makes it the better choice for robotic systems in collaborative environments.

8.1 Outlook

Dynamic robotic systems requiring fast and reliable perceptions systems to analyze their environments will gain increasing importance in coming years with an increase of robots utilized in open systems with non predictable input parameters.

We developed a proof of concept for such a system. However, until it can be deployed there are still improvements necessary. First, instead of training a model for each robot class, we can train one model for all robot classes and humans. This can be done by either merging the individual data sets into one or by further developing the data generation tool in a way that it can generate multi-class data sets. The combined model can then be trained on such a data set.

Furthermore, the reliability and robustness of every detection model depends on the quality of the training data. If we were to acquire more data of robots with different poses or in different environmental settings, it could lead to a significant improvement in robustness and thus make our model more reliable.
The reliability of the Intel Realsense system could also be improved by making the network set-up more robust to connection losses and delays. For example, one could utilize multiple different networks that intervene if the current network looses connection or only deploy the system in a closed environment.

So far our model can only detect and classify objects. However, for robot navigation it would also be useful to detect and analyse behavior. For example, if a robot detects a person, detecting the behaviour of the person can help the robot to make decisions on how to navigate. If the person is just sitting on a chair, the robot can be less careful and move faster, however, if the person is running, the robot should adjust its behaviour accordingly and move slower to avoid collisions. Hence, future work should extend our system to include behaviour detection.

References

di Castro., Mario et al. (2017). ‘Novel Pose Estimation System for Precise Robotic Manipulation in Unstructured Environment’. In: *Proceedings of the 14th International Conference on Informatics in Control, Automation and Robotics - Volume 2: ICINCO,* INSTICC. SciTePress, pp. 50–55. ISBN: 978-989-758-264-6. DOI: 10.5220/0006426700500055.

Fang, Fang et al. (2017). ‘Real-time RGB-D based people detection and tracking system for mobile robots’. In: *2017 IEEE International Conference on Mechatronics and Automation (ICMA)*, pp. 1937–1941. DOI: 10.1109/ICMA.2017.8016114.

Felzenszwalb, Pedro F. et al. (2010). ‘Object Detection with Discriminatively Trained Part-Based Models’. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 32.9, pp. 1627–1645. DOI: 10.1109/TPAMI.2009.167.

Girshick, Ross (2015). *Fast R-CNN*. arXiv: 1504.08083 [cs.CV].

Gupta, Saurabh et al. (2014). ‘Learning Rich Features from RGB-D Images for Object Detection and Segmentation’. In: *Computer Vision – ECCV 2014*. Ed. by David Fleet et al. Cham: Springer International Publishing, pp. 345–360. ISBN: 978-3-319-10584-0.

He, Kaiming et al. (10th Dec. 2015). ‘Deep Residual Learning for Image Recognition’. In: *arXiv:1512.03385 [cs]*. version: 1. arXiv: 1512.03385. URL: http://arxiv.org/abs/1512.03385 (visited on 31st Dec. 2020).

Intel Corporation (2021a). ‘Intel RealSense D400 Series Product Family Datasheet’. In: pp. 15–21.

— (2021b). *Intel® RealSense™ Depth Camera D435i*. URL: https://www.intelrealsense.com/depth-camera-d435i/ (visited on 25th Apr. 2021).

— (2021c). *The Intel Realsense SDK and ROS/ROS 2*. URL: https://www.intelrealsense.com/ros/ (visited on 25th Apr. 2021).

Ioffe, Sergey and Christian Szegedy (2nd Mar. 2015). ‘Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift’. In: *arXiv:1502.03167 [cs]*. arXiv: 1502.03167. URL: http://arxiv.org/abs/1502.03167 (visited on 1st May 2021).

LeCun, Yann, Yoshua Bengio and Geoffrey Hinton (May 2015). ‘Deep learning’. In: *Nature* 521.7553, pp. 436–444. ISSN: 0028-0836, 1476-4687. DOI: 10.1038/nature14539. URL: http://www.nature.com/articles/nature14539 (visited on 16th Dec. 2020).

Liu, Hengli et al. (2016). ‘People detection and tracking using RGB-D cameras for mobile robots’. In: *International Journal of Advanced Robotic Systems* 13.5, p. 1729881416657746. DOI: 10.1177/1729881416657746. eprint: https://doi.org/10.1177/1729881416657746. URL: https://doi.org/10.1177/1729881416657746.

Mocanu, Irina et al. (2018). ‘Human Activity Recognition with Convolution Neural Network Using TIAGo Robot’. In: *2018 41st International Conference on Telecommunications and Signal Processing (TSP)*, pp. 1–4. DOI: 10.1109/TSP.2018.8441486.

Munaro, Matteo et al. (2016). ‘RGB-D Human Detection and Tracking for Industrial Environments’. In: *Intelligent Autonomous Systems 13*. Ed. by Emanuele Menegatti et al. Cham: Springer International Publishing, pp. 1655–1668. ISBN: 978-3-319-08338-4.

NVIDIA Corporation (9th July 2018). *Jetson AGX Xavier Developer Kit.* NVIDIA Developer. URL: https://developer.nvidia.com/embedded/jetson-agx-xavier-developer-kit (visited on 30th Apr. 2021).

Open Robotics (2021a). *About ROS.* URL: https://www.ros.org/about-ros/ (visited on 25th Apr. 2021).

— (2021b). *ROS Concepts.* URL: http://wiki.ros.org/ROS/Concepts (visited on 25th Apr. 2021).

— (2021c). *ROS Documentation.* URL: http://wiki.ros.org/ (visited on 25th Apr. 2021).

Pandey, Amit Kumar and Rodolphe Gelin (2018). 'A Mass-Produced Sociable Humanoid Robot: Pepper: The First Machine of Its Kind'. In: *IEEE Robotics Automation Magazine* 25.3, pp. 40–48. DOI: 10.1109/MRA.2018.2833157.

Redmon, Joseph, Santosh Divvala et al. (9th May 2016). 'You Only Look Once: Unified, Real-Time Object Detection'. In: *arXiv:1506.02640 [cs]*. arXiv: 1506.02640. URL: http://arxiv.org/abs/1506.02640 (visited on 1st May 2021).

Redmon, Joseph and Ali Farhadi (25th Dec. 2016). 'YOLO9000: Better, Faster, Stronger'. In: *arXiv:1612.08242 [cs]*. arXiv: 1612.08242. URL: http://arxiv.org/abs/1612.08242 (visited on 1st May 2021).

— (8th Apr. 2018). 'YOLOv3: An Incremental Improvement'. In: *arXiv:1804.02767 [cs]*. arXiv: 1804.02767. URL: http://arxiv.org/abs/1804.02767 (visited on 1st May 2021).

Ren, Shaoqing et al. (2016). *Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks.* arXiv: 1506.01497 [cs.CV].

Saha, Sumit (17th Dec. 2018). *A Comprehensive Guide to Convolutional Neural Networks — the ELI5 way.* Medium. URL: https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53 (visited on 2nd May 2021).

StereoLabs (2021a). *Getting Started with ROS and ZED.* URL: https://www.stereolabs.com/docs/ros/ (visited on 25th Apr. 2021).

— (2021b). *Zed 2.* URL: https://www.stereolabs.com/zed-2/ (visited on 25th Apr. 2021).

Tekin, Bugra, Sudipta N. Sinha and Pascal Fua (2018). *Real-Time Seamless Single Shot 6D Object Pose Prediction.* arXiv: 1711.08848 [cs.CV].

Tilley, Jonathan (2021). *Automation, robotics, and the factory of the future — McKinsey.* URL: https://www.mckinsey.com/business-functions/operations/our-insights/automation-robotics-and-the-factory-of-the-future (visited on 8th May 2021).