Near-Field Maritime Obstacle Detection for Small USV Collision Avoidance System

Author:
David Fosca Gamarra

Project Report
NTNU
Trondheim, 2022

Abstract

This work proposes an end-to-end solution for maritime obstacle detection and proximity alert for USV vehicles that require an extra layer of on-board intelligent navigation. The NTNU FishOtter USV is a mobile platform for autonomous robotic search, localization and tracking of fish. The platform aims to facilitate investigation and understanding of fundamental biological processes in the ocean such as the migration and movement ecology of fishes. The FishOtter has been deployed several times over the past two years in Norwegian fjords, and during this time it became evident that potential collisions with medium to small obstacles are a threat that can only be mitigated if the USV is self-aware of its surroundings by detecting obstacles and its proximity during runtime.

In this context, obstacles are defined as surface objects of unknown origin and character that randomly appear in the vehicle's path. Small boats, buoys and unmapped piers and rocks are examples of such obstacles. Since the FishOtter is a small USV with limited space and power resources, LIDAR, radar or a sensor fusion based systems are sometimes not a viable option to implement an efficient anti-collision system. However, stereo-camera based solutions provide accurate short distance depth estimation and allow for powerful image processing approaches such as image segmentation using deep learning, with little space and power overhead.

The stereo-camera used for this project is provided by Stereolabs, the deep learning component is a U-Net segmentation neural network developed using Python and Tensorflow, and the obstacle detection and proximity estimation application, which integrates the stereo vision and the neural network solution, is developed in C++ and deployed on the edge using a Jetson Xavier-NX. In addition, the application is also deployed in a Jetson Nano to assess its performance on a lower cost hardware.

As mentioned, the OtterNet segmentation model is evaluated with 100 images manually annotated (Otter Dataset), resulting in a f1-score of 0.83. In addition, the accuracy of the stereo-depth estimates is also assessed, showing a relative error no larger than 5% for distances no longer than 10 meters. Finally, considering the resource constraint nature of the edge computing devices such as the Jetson Xavier-NX and Jetson Nano, performance tests are conducted to assess the latency of the application as well as the memory utilization and power consumption during run-time. These tests are relevant in order to discover potential optimization opportunities and calculate the processing speed of the application, which is approximately 1 frame per second and 2 frames per second for the Jetson Nano and Jetson Xavier-NX respectively.

Overall, the obstacle detection and proximity alert application, designed to be deployed on an edge Jetson device, shows promising results to potentially be used as part of a collision avoidance protocol. It is important to mention that this project sets a technical baseline for further optimizations, by providing the details of a unique pipeline that goes from design to implementation and deployment using the different platforms required to build the application.

Acknowledgments

I want to thank Professor Jo Arve Alfredsen for supervising this project, and PhD candidate Nikolai Lauvås for co-supervising it. Special thanks for trusting me with the newly acquired hardware, and investing time helping me integrating it in the FishOtter platform, as well as organizing the sea trials that took place in Børsa, Norway. Finally, thanks to the European Master in Embedded Computing Systems (EMECS) program and the coordinators for the opportunity of studying this master.

TTK4550 - Specialization Project Assignment

- **Name:** David Fosca Gamarra
- **Program:** EMECS/ Cybernetics and Robotics

Project description: The master project is an integral part of the NTNU robotic fish tracking project where the overall goal is to create a mobile platform for autonomous robotic search, localization and tracking of free-ranging fish (NTNU FishOtter). This will be achieved by unifying two technological domains: acoustic fish telemetry and cooperative unmanned surface vehicles. As a novel enabling technology within fisheries research, the platform will facilitate investigation and understanding of fundamental biological processes in the ocean such as the migration and movement ecology of fishes. The project objective is to design and develop a safety system that implements automatic obstacle detection and anti-collision for the USV FishOtter when it operates in autonomous mode. Obstacles are in this context defined as surface objects of unknown origin and character that randomly appears in the vehicle's path with which collision should be avoided to prevent damage to the object or the vehicle itself. Small boats, buoys and unmapped piers and rocks are examples of such obstacles. The system should be based on a suitable camera and computer vision solution that enable robust object detection under varying sea states and light conditions and provide situational awareness and anti-collision capacity to the vehicle's local guidance system. The system should also provide situational awareness for a supervisory remote vehicle operator in terms of live images, annotations and alert messaging. The project consists of the following tasks:

- Study the physical and technical properties (hardware/software/control) of the USV FishOtter system.
- Literature survey of camera-based obstacle detection and anti-collision systems for autonomous systems. Focus on solutions considered relevant for the FishOtter case.
- Selection of a suitable camera system and other required components (in collaboration with FishOtter team members).
- Design and implement a solution for computer vision-based obstacle detection for anti-collision strategies.
- Define relevant test cases, conduct experiments, evaluate, and discuss the performance and robustness of the solution.
- Write technical documentation and report.

Index

1 Introduction

As a country that gains much value from the ocean, Norway cares greatly about maritime life and ocean conditions through studies and research activities that are carried out to better understand the conditions below the sea surface. One example of such deployed efforts is the “NTNU FishOtter Project”, a fish tracking system to observe and analyze the movement and condition of fishes. The project aims to develop an autonomous multi-agent system for search, localization and continuous tracking of acoustic fish tags beneath the water surface.

In the pursuit of building the FishOtter as an autonomous system that will be navigating the Norwegian fjords on its own for several days without any continuous human monitoring, the system must be capable of perceiving its surroundings in order to avoid collisions with static or moving objects that could endanger its own safety or that of the others. Data coming from "Global Navigation Satellite Systems (GNSS)" or "Marine Automatic Identification System (AIS)" are very useful for general navigation awareness and to avoid collision with ships that are in the system (AIS), but many other obstacles such as buoys, fish farms (aquaculture), small boats (kayaking), and other small objects are not perceived by it, and thus require field-in-situ recognition and reaction.

Object detection and collision avoidance systems are a critical part of autonomous vehicles and as such, much work is being done in that area. The most relevant sensors that are considered for such endeavors, working individually or in a hybrid sensor-fusion mode are: LIDAR, radar, and cameras. Nevertheless, due to factors such as weight, power consumption, and price, some of these systems may not be suitable for an efficient deployment in small USVs such as the FishOtter.

The next section will present some of the most important concepts behind obstacle detection for collision avoidance at sea with special focus on camera-based solutions.

2 Theoretical Background

2.1 Unmanned and Autonomous Surface Vehicles

Unmanned Surface Vehicles (USVs), are maritime vehicles that do not rely on humans on board for navigation, but are remotely controlled by an operator. On the other hand, Autonomous Surface Vehicles (ASVs) are vehicles that operate autonomously on the surface of the water with limited or non-existent human intervention, meaning that they are capable of making decisions during run-time based on its situation and surroundings. Nevertheless, it is possible to provide similar autonomous capabilities to a USV if some technical considerations are met, such as automatic route generation and path planning, object detection for collision avoidance, along with other autonomous decision-making systems that need to work in-situ and in real-time for most of the cases. The applications of USVs and ASVs are wide, from military, civilian, and research applications, they represent a big asset for maritime exploration and transportation due to several advantages such as: i) longer duration performance specially for potentially hazardous environments for humans, ii) reduced human error and communication dependability, iii) reduced personnel cost, iv) increased available space for transport, among others [75]. Much work has been done in this area on different industry levels, from the demonstration of the first "Fully Autonomous Ferry by Rolls-Royce in 2018" [15], to a multi-purpose USV platform with ASV capability from *Kongsberg Maritime* [20]. Overall, several global companies are working towards the development of ASVs and USVs and aime to have fully autonomous ships by the end of 2025.

2.1.1 Study Case: NTNU USV-FishOtter

The NTNU FishOtter is a small unmanned surface vehicle (USV) that is mostly composed of a catamaran structure chassis, two electrical fixed thrusters and power-distribution components from *Maritime Robotics* [10], but with control and sensing platform designed at NTNU [11]. The system is based on the "Underwater Systems and Technology Laboratory Toolchain (LSTS)", with "DUNE: Unified Navigation Environment", a runtime environment for unmanned systems on-board software [2]. The project has the objective of developing an autonomous multi-agent system for search, localization and persistent tracking of acoustic fish tags beneath the water surface [35].

- **Hardware Architecture**

  Figure [1] shows the main components of the USV, which includes: i) two electrical fixed thrusters that provide a maximum propulsive power of 1 HP, ii) a signal light for communicating control status to the user, iii) a GPS antenna, iv) four batteries with a total capacity of 3660Wh, for 20 hours of continuous operations, and v) the control box which includes a Raspberry Pi 4 with 4GB of RAM, connected to a *Strato Pi CAN* board for power supply and communication. In addition, the Raspberry

Pi is connected through CAN bus to the *Torqeedo Interface* card which controls the thrusters. Finally, it includes an industrial 4G LTE Wi-Fi router *Teltonika RUT950*. The interior of the control box can be seen in **Figure** [2].

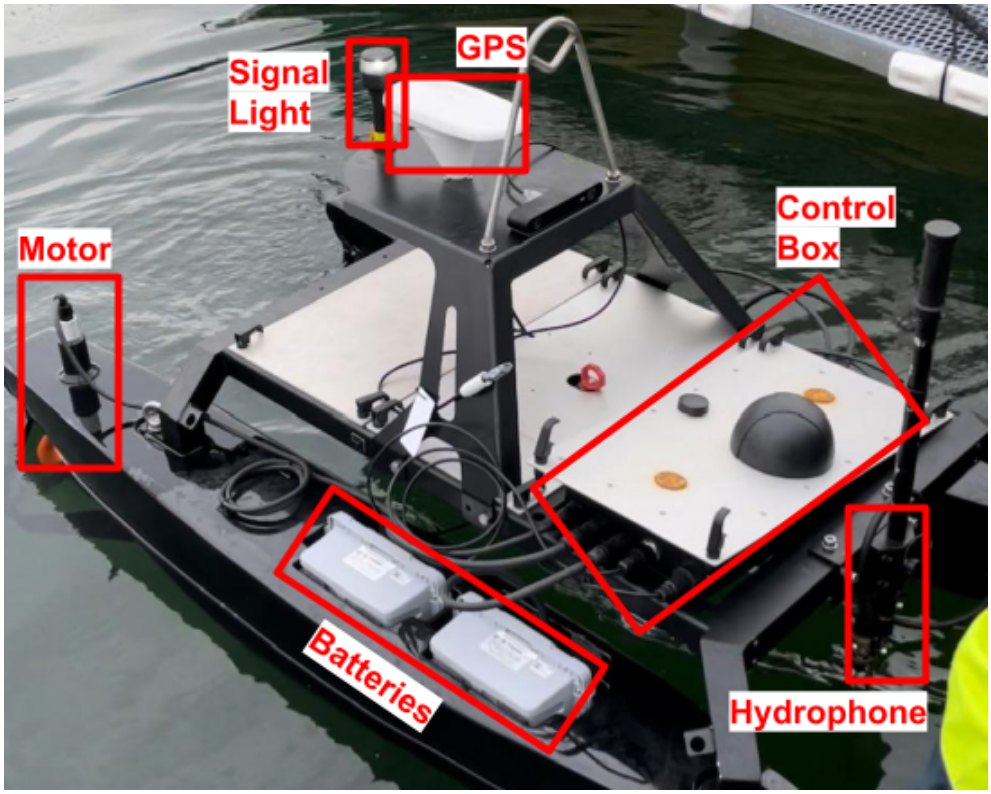

Figure 1: The FishOtter with some of its hardware features pointed out.

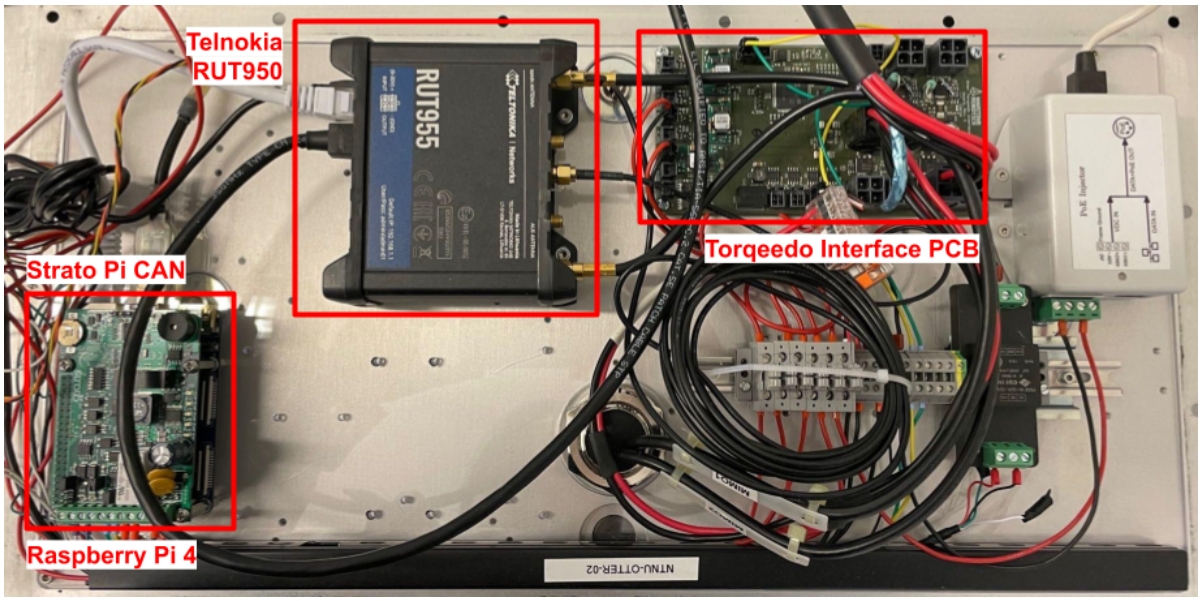

Figure 2: Control Box of the FishOtter.

- **Software Architecture**

  The firmware running on the FishOtter is based on the LSTS tool-chain , which is an open source software specially designed for maritime vehicle network development and interconnection of heterogeneous devices [64]. It can handle Surface (ASV), Underwater (AUV) and Aerial vehicles (UAV). The toolchain includes three components: i) NEPTUS: the operators console, ii) IMC: the communication protocol, and iii) DUNE: the control software running on-board the vehicles. **Figure [3]** presents a diagram of the interaction between the three mentioned components. The Raspberry Pi 4 uses its own Raspberry Pi OS that runs a customized version of DUNE [2].

Figure 3: LSTS Tool Chain General Overview.

- **Network Architecture**

  The FishOtter and the land station communicate using *AirMax* wireless devices from *Ubiquiti*. The expected working distance for wireless communication is around 600m to 700m. As mentioned, the FishOtter is equipped with *Teltonika RUT950* 4G routers with a *Teoglas* antenna mounted on the top of the USV as shown in the previous **Figure [1]**. It is important to mention that the connection uses a *OpenVPN* service to connect with servers from NTNU.

- **Sensors**

  The main sensor in the FishOtter is a specialized hydrophone from *Thelma Biotel AS*. It has built in signal processing for frequency band discrimination and detection of acoustics fish tags with a frequency between 63 and 77 kHz. The receiver communicates through an RS-485 interface with the controlling computer and produces messages that are transmitted using IMC communication protocol [2].

2.2 Challenges in Maritime Obstacle Detection

Obstacle detection in unmanned surface vehicles is still a relatively young research area, especially when compared to autonomous driving where more literature and public data can be found. For example, the use of stereo cameras for disparity map calculation and object detection has been evaluated by [32][85]. In addition, several hybrid sensor fusion approaches have been developed. This is the case of [73], who uses a combination of LIDAR and camera, or [29] who merges data from stereo-cameras and Radars, and [25] who proposes a combination of LIDAR and inertial measurements (IMU) for more precise obstacle detection in different environments.

However, many of these approaches that rely on detecting the ground plane do not properly work in maritime environments. In addition, the use of LIDAR and Radar can be sometimes prohibited, which is the case of small USVs with limited space and power. Camera based solutions seem to be a viable approach. Traditionally, object detection in automotive applications have been addressed by saliency-based region detectors [83] [52] or stereo detectors [77] [58]. Nevertheless, the performance of these methods [63] [28] has a strong dependence on the proper estimation of the ground plane which is challenging in maritime environments. In addition, these methods assume the objects are distinguishable enough from their background which is far from reality in several cases due to glitter, fog and several changing lighting conditions, leading to failure. Moreover, considering classical background subtraction methods that require a static camera position is overall not the best approach due to waves constantly rocking the USV [65]. Additionally, studies that rely on estimating the horizon as a reference to object proximity have their horizontal plane assumptions violated due to the rocking movement of the USV besides not working when the USV is close to coastal waters [55].

Over the last five years, deep learning based solutions have become part of the state of the art for maritime object detection. Deep convolutional networks (CNNs) are one of the most used Artificial Intelligence models. Several studies have proposed the use of object detection models such as R-CNN, YOLO and SSD for maritime applications [50][76] due to the robustness these models show towards very changing environmental conditions. However, object detection models usually have a hard time at detecting big size objects such as piers or general land structures. On the other hand, semantic segmentation CNNs appear to be a more robust solution that can address the previously mentioned limitations of object detection models. For example, the use of E-Net [62] and U-Net [71] models with skip-connections for real-time image segmentation in [49] and [82] respectively.

Overall, deep learning approaches are becoming the go-to solution for several maritime obstacle detection, especially for small USVs that can only rely on camera based solutions.

2.3 Perception Sensors

2.3.1 LIDAR

LIDAR (Light Detection and Ranging) is considered an active sensor that works with light beam emission and time-of-flight calculation at a very high frequency to generate a 3D map representation of its surroundings, called 3D point cloud. LIDARs are very accurate sensors when it comes to computing the depth or distance of an object, even at distances greater than 100 meters, depending on the system [48], in addition to not requiring daylight to work. Although it is a very popular and well known sensor, LIDARs are relatively large, power hungry, costly, and better suited for long range accurate distance estimation or when the USV moves at very high speeds that requires further away distance information to make well anticipated maneuver decisions[26].

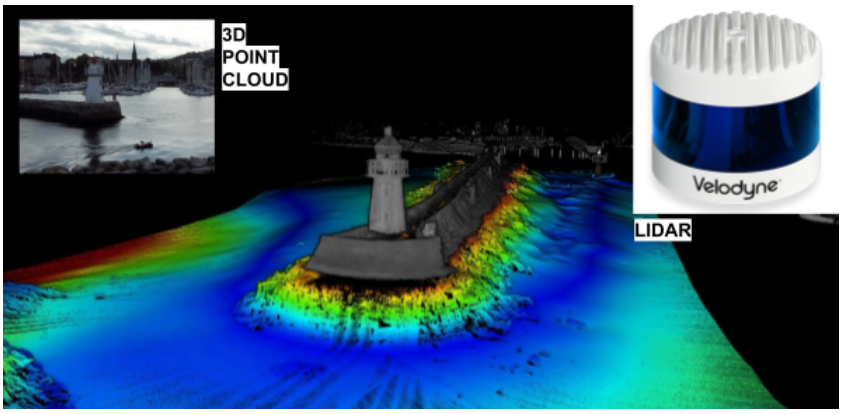

Figure 4: Example of LIDAR sensor and 3D point cloud [9].

2.3.2 Radar

Radar (Radio Detection and Ranging) is considered an active sensor that works by emitting electromagnetic waves and analyzing the reflection on object surfaces to generate a map of the surroundings. Along with the fact that radars are well known for providing accurate speed information about the detected obstacle by exploiting the well known doppler effect, they are strongly robust even against heavy environmental conditions such as rain and fog, compared to LIDAR and camera based solutions. Nevertheless, radars commonly require space for installation and carry a considerable weight, making them a difficult choice for a small USV [46][26].

Figure 5: Example of a radar sensor in a maritime vessel [43].

2.3.3 Camera

Cameras are passive sensors that have gained popularity as an interesting alternative to the previous mentioned active sensors because of their attributes, such as: light-weight, low-power, lower-cost, and information rich sensors [80]. Unlike LIDAR, cameras do not contain mechanical moving parts, which makes them inherently more robust to vibration or atypical movement stress. Furthermore, cameras are passive, meaning that there is no risk of them interfering with other critical systems, such as GPS or radio [41]. In addition, there is a vast research background [56][23][47][61] and available commercial solutions developed for image processing. Moreover, image processing with artificial intelligence has shown to be a very promising approach for complex tasks such as object detection at greater computational speed than traditional image processing approaches.

In addition to visual image inspection, using two cameras in a stereo setup makes it possible to estimate geometric information such as depth and build up 3D point clouds in real-time with low computational complexity. Such is the case of [76], that uses stereo vision for a USV auto-docking solution, or [79] for 3D object detection from stereo images. Although LIDAR geometric information is usually more accurate, researchers are evaluating how to increase the accuracy of stereo measurements in a direct way, which depends of the camera parameters such as the resolution, or in an indirect way by using it along with the LIDAR to continuously train and tune image classifiers [78]. Overall, visual data has become a key stone for image-based navigation techniques for advanced driver-assistance systems (ADAS) and autonomous driving (AD), currently available and used in industry.

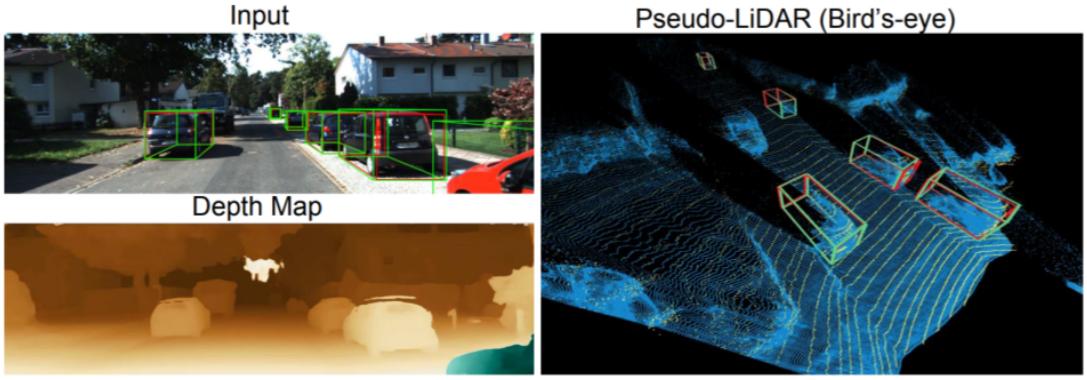

Figure 6: Stereo-depth estimations and 3D object detection [78].

2.3.4 Sensor-fusion based solutions

A sensor-fusion approach, integrates data from active and passive sensors, to enhance the overall system by merging the advantages but also disadvantages of both [74][60]. Nevertheless, sensor fusion requires careful consideration of multiple sources of data, which brings its own difficulties and complexities, being the most critical one the calibration and synchronization of several sensors [84]. In addition, each sensor brings data from different domains which usually requires complicated data processing and to achieve a homogeneous input needed by most of the systems.

Figure 7: Example of sensor fusion for autonomous navigation at sea [42].

2.4 Stereo Vision

Stereo vision or stereoscopic vision is a technique that mimics human depth perception by combining two cameras to extract geometric information from the perceived images. To better understand how this is done, it is first important to have a general idea of what a camera model is. A camera model describes how 3D coordinates perceived from the world are mapped into 2D coordinates in the form of an image. The pinhole model is one of the most used geometric camera models due to its simplicity and popularity to estimate geometric quantities. For example, a relevant estimation problem is Triangulation. A simple mathematical equation describes how a 2D image pixel (u_i) is the projection of a 3D world coordinate (X_w) after rotation (R_{iw}) and translation (tiw/i), as presented in **Equation (1)**. Given the camera parameters ($p_{camera,i}$) and the exact camera poses, it is possible to calculate the 3D point coordinates (X_w). A more specific application is that of stereo-vision, which uses two equal and synchronized cameras with known relative poses and overlapping views in order to estimate the 3D coordinates of the pixels presented in both images.

$$u_i = project(R_{iwXw} + t_{iw/i}, p_{camera,i}) \tag{1}$$

Considering two cameras that are positioned parallel to each other as shown in **Figure** [8]. The following **Equations (2)** and **(3)** describe the projection of the 3D point into a 2D image, where "b" is the baseline, "f" is the focal length of camera, u_L and u_R are the 2D projections of the 3D point "P", X_A is the X-axis of a camera, Z_A is the optical axis of a camera. The distance between the two points (u_L and u_R), also known as disparity, is described in **Equation (4)**. From this equation is it possible to clear out the depth of the pixel if the disparity value is known.

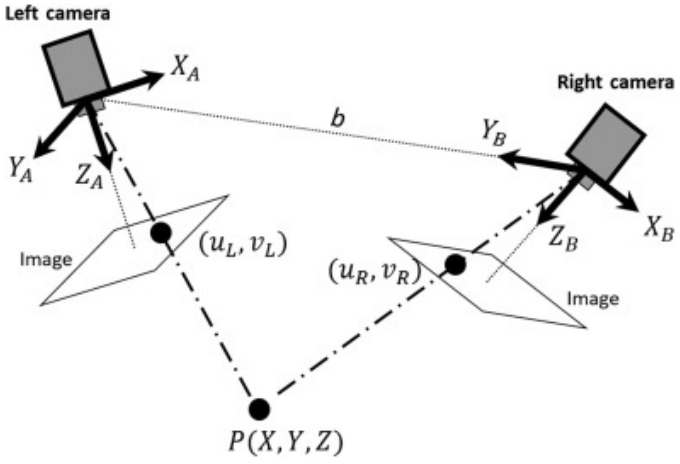

Figure 8: Stereo-vision system with geometric reference [40].

$$u_L = f\frac{X_A}{Z_A} \tag{2}$$

$$u_R = f\frac{X_A - b}{Z_A} \tag{3}$$

$$d = (u_L - u_R) = f\frac{b}{Z_A} \tag{4}$$

2.4.1 Correspondence Problem

To compute the disparity for all the pixels, also known as disparity map, the correspondence problem has to be solved. This task aims to find the matching pair of pixels in the stereo images that are projections of the same real points in 3D space. One basic approach to solve this problem is the so-called blocking matching algorithm, which is based on comparing small windows around a pixel in the first image with multiple small windows in the other stereo image. An important constraint to find a solution in this problem is given by the epipolar geometry [24], that simplifies the search of a given 3D point "X" by stating that it should be found in the epipolar line. So, if the 3D point "X" is observed in the left image view, the same point has to be located along the epipolar line in the right image view, and vice versa.

The problem is defined as a minimization problem, where a loss function is computed for each pair of windows that are compared. So, the point in the second image with the minimum loss value is defined as the match to the evaluated point in the first image. The difference between both points will result in the disparity, as previously stated. There are two main functions used to find the matching points: Sum of Absolute Differences (SAD), and Sum of Squared Differences (SSD). In general, SAD is preferable to SSD as it is faster and more robust to noise and outliers [72].

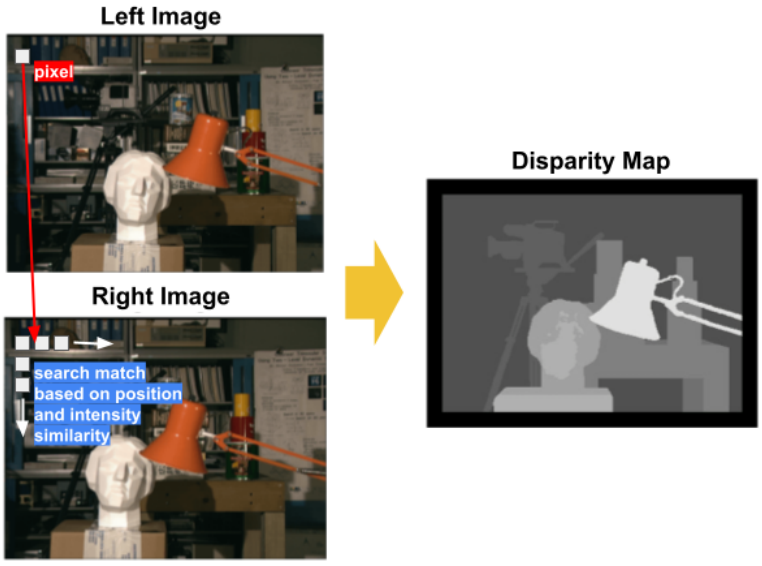

Figure 9: Correspondence problem and disparity map example [36].

2.4.2 Stereo Camera Setup

There are three main factors that influence the depth estimate: i) baseline length, ii) resolution, iii) focal length [86]. The baseline is the distance between the two cameras and as defined in the previous equation, it is directly proportional to the depth. So, the longer the baseline the longer the depth it can cover with better accuracy. Typically, the ratio used to estimate the baseline distance with respect to the working distance for depth estimation is approximately 1:30, depending on the camera's angle [34]. On the other hand, the resolution of the two cameras is also an important factor in depth estimation. The higher the number of pixels to search in the correspondence problem, the higher the number of disparity levels which in turn provides better depth estimation accuracy. However, at higher resolutions, the computational complexity along with the execution time also increases. Furthermore, the focal length of the lenses is also an important factor when it comes to the field of view. The focal length is the distance between the point of convergence of the lens and the array sensor where the image is formed. The lower the focal length, the farther the camera is able to perceive, but with reduced field of view. On the contrary, with greater focal length the less depth it is able to perceive, but at the same time the field of view increases. Finally, the recommended minimum overlap for the two cameras' fields of view is 2/3 for proper point correspondence and matching.

Figure 10: Stereo-camera setup

2.5 Deep Learning for Object Detection

Object detection is a fundamental problem in computer vision and can be performed using either traditional image processing techniques or modern deep learning approaches. Artificial Intelligence has gained great popularity in a wide range of fields and applications due to its capacity to tackle complex problems. Deep learning, a subset of machine learning, takes its inspiration from the structure of the human brain to conceptualize the so-called artificial neural networks. These networks are capable of predictive analysis when trained with a substantial amount of data specific to the application. Among the different deep learning algorithms, Convolutional Neural Networks (CNNs) are one of the most important ones when it comes to feature extraction [54]. CNNs are widely used for image processing applications such as classification, object recognition and semantic segmentation.

2.5.1 Convolutional Neural Networks

Convolutional Neural Network (CNN), as shown in **Figure** [11], is a type of deep learning network that is suitable for processing data that has a well defined spatial and/or temporal structure. CNNs are specially used for image processing due to their feature extraction ability. This is accomplished by the use of filters, which in traditional methods are hand-engineered, CNNs have the ability to learn the filter weights to solve much more complex problems such as image classification among several classes. The main layers that build a CNN architecture are explained below.

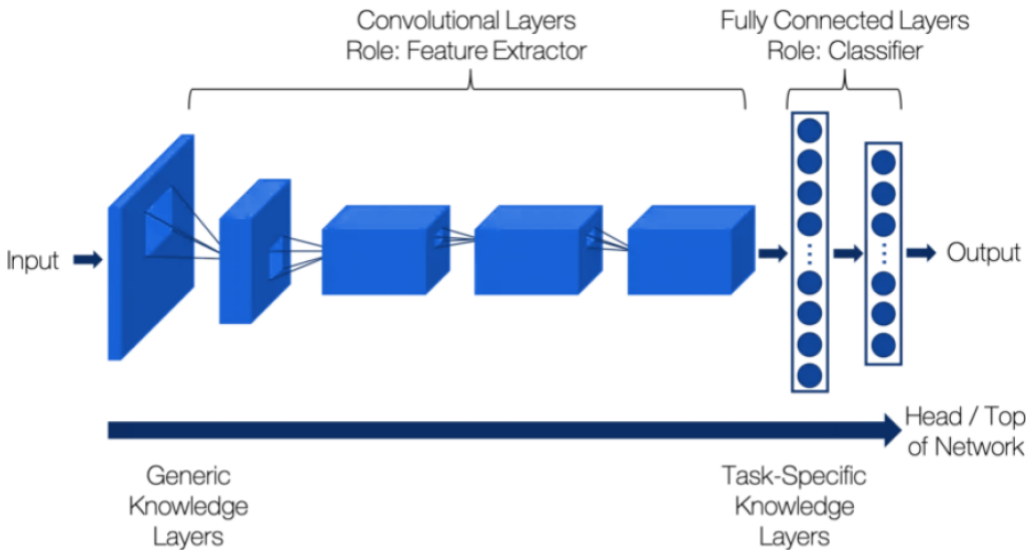

Figure 11: Convolutional Neural Network example

(i) **Convolutional Kernel and Filter**

The convolutional kernel is defined as an n-dimensional vector with values known as weights. This kernel slides over another n-dimensional data vector while performing arithmetic operations with it. The convolutional filter is a multi-dimensional version of a kernel filter, stacking two or more kernels. When processing an image, the filter slides over all the pixels in the input image and outputs the dot product of the filter and the image pixels at the filter's position. This process is called convolution and is represented in **Figure** [12]. Typically 2D kernel sizes are: 2x2, 3x3, 5x5, 7x7. The weights of each convolutional filter is learned during training.

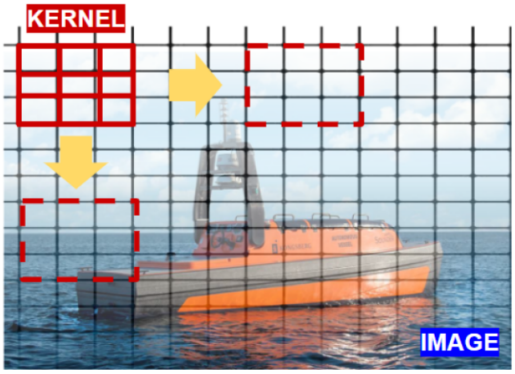

Figure 12: Example of convolutional kernel sliding through image.

(ii) **Max Pooling**

The pooling layers execute a downsampling operation. Similar to the convolutional filters, the pooling layer also works with a n-dimensional vector that slides through the data. There are different downsampling approaches, but the most common one is Max Pooling. This method selects the highest value from all the values from the intersection between the pooling vector and the data as seen in **Figure** [13]. This operation helps the next convolutional layers to pick a different information that was not analyzed by the previous layer. In addition, pooling helps the model to avoid overfitting to the training dataset.

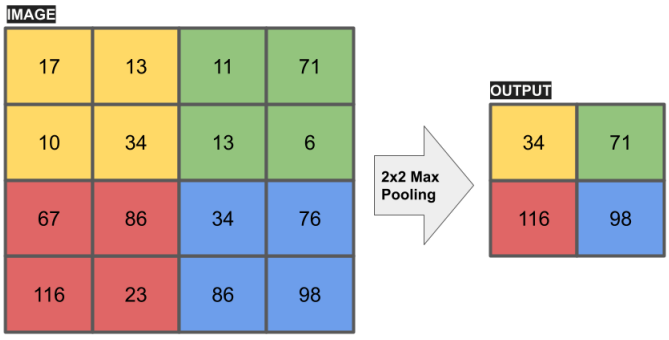

Figure 13: Example of max pooling operation.

(iii) **Batch Normalization**

Batch normalization is a popular data normalization method. This layer does not generate any dimensional modification, but applies a transformation that maintains the mean of the output close to 0 and the standard deviation of the output close to 1. This is important to assure a correct learning process for the neural network, reducing the probability of vanishing or exploding gradients that would eclipse any learning possibility.

(iv) **Dropout**

Dropout is a well known regularization technique in deep learning. It is applied after a neural layer and has the purpose of randomly and temporarily deactivating a percentage of neurons during training. This means that these neurons are ignored during forward propagation and no update is made to its learning parameters during backward propagation.

2.5.2 Object Detection

Object Detection is used to find and recognize several targets in images with the help of bounding boxes. The common algorithms approach this problem in two steps: i) classification of sub-regions of the image, and i) prediction of the bounding boxes which describes where the object is located in the image. Since deep learning was discovered to be a powerful tool for object detection, several architectures have been proposed and implemented for this purpose. Some of them are explained below.

(i) **Region-based convolutional neural networks (R-CNN)**

R-CNNs are one of the first deep learning algorithms for object detection. Based on a selective search algorithm, the model first selects various proposed regions (nearly two thousands sections) from which CNNs are used to extract features for classifying them and predict bounding boxes for determining the position. **Figure** [14] shows an example of such a process. It is important to mention that this model is actually composed of three independently trained sub-models: i) Support vector machine for classification, ii) Bounding box regressor, and iii) CNN for feature extraction. This poses two problems: i) increases inference time, and ii) increases training complexity [39]. A new version called Fast R-CNN, addresses these problems by coming up with a unified deep learning model for feature extraction, classification and bounding box regression. In addition the features are extracted once from the whole image and not from several sub-regions [38]. Finally, the latest version called Faster R-CNN makes use only of the CNN feature map instead of using the selective search algorithm for regions proposal [69].

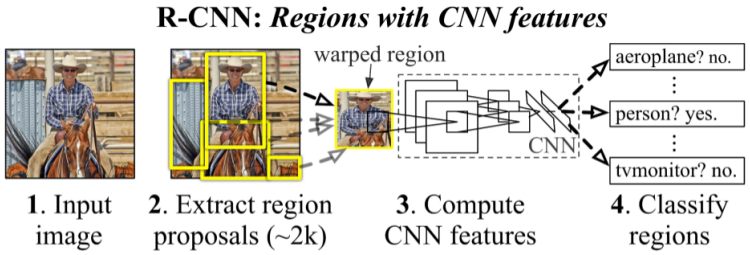

Figure 14: Object detection system overview with RCNN [39].

(ii) **Single Shot Detector (SSD)**

SSD is a one-stage detector for multiple classes, which means that unlike R-CNN it does not rely on region proposals, making it considerably faster and with similar accuracy than R-CNN and its variations. SSD builds a grid with different scales, from which several boxes that are proportional to the cell size are generated per object as seen in **Figure** [15]. In addition, a process called non-maximum suppression is applied to eliminate repeated boxes. Finally, the boxes run through a classifier that compares it to the ground truth bounding box. Considering that the intersection over union has to be at least 50%, the box with the highest confidence score is

selected [53]. The SSD detector is easy to train in comparison to other single-stage methods, SSD produces more accurate results, even with small input size images.

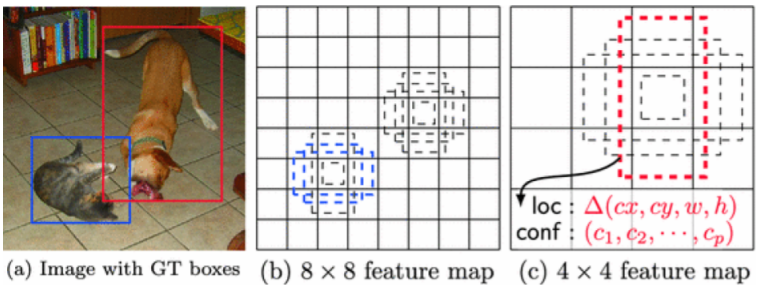

Figure 15: Object detection system overview with SSD [53]

(iii) **You Only Look Once (YOLO)**

YOLO object detection is a single-stage detector that performs classification and bounding box regression in one step, making it a much faster approach compared to the others. For example, YOLO object detection is 1000 faster than R-CNN and 100 faster than Fast R-CNN. As its name suggests, YOLO only sees the image once. It starts by dividing it in a SxS grid where several bounding boxes are created in each cell and a confidence score is given to each of them depending on how well the model thinks the bounding box fits the object that must be detected inside the cell. This process can be seen in **Figure** [**16**]. Each cell also predicts "n" classes which are conditional to the cell containing the object. So, to calculate the confidence score for each class bounding box, the following probabilities are multiplied: i) Probability of an object being classified as one of the "n" classes in a cell, ii) Probability of an object appearing inside the cell, iii) Intersection over union between the predicted box and the ground truth. Depending on the selected threshold the bounding boxes which have higher scores will be the resulting prediction [68].

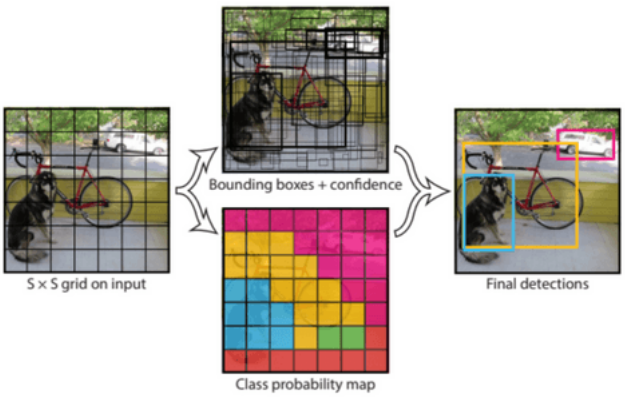

Figure 16: Object detection system overview with YOLO [68]

2.5.3 Image Segmentation

Image segmentation, also known as semantic segmentation, is the process of clustering image pixels together which belong to the same object class as seen in **Figure** [17]. Different from the bounding box output of object detection approaches, in image segmentation the output is an image of the same dimensions of the input image but with pixel intensity values equal to the class it has been assigned. Image Segmentation plays an essential role in Artificial Intelligence systems used in autonomous driving, medical image diagnosis, and precision agriculture, among others. Models are usually evaluated with pixel accuracy metrics, intersection over union and dice coefficient (f1-score).

Figure 17: Example of image segmentation with three different classes [51]

2.5.4 U-Net

U-Net is one of the state of the art deep learning architectures for image segmentation [71]. Originally developed in 2015 for biomedical image segmentation, it gained popularity in different domains for its speed and precision. The neural network model shown in **Figure** [18] integrates two clear sections based on convolutional layers: encoder and decoder.

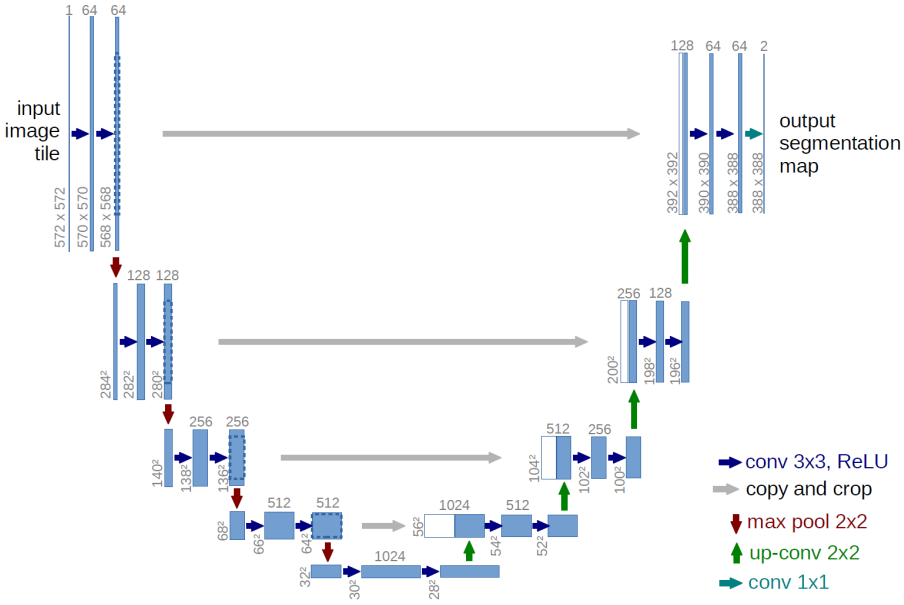

Figure 18: U-Net architecture [71].

The **encoder** follows a typical contraction path where in each block the input image is passed through two 3x3 kernel size unpadded convolutions, each followed by a rectified linear unit (ReLU). In addition, the spatial information is reduced through a 2x2 Max Pooling operation with stride 2, which is represented in **Figure** [18] by the red arrows pointing down. Notice that the feature channel dimension doubles in each block. **Figure** [18] shows how the filter size increases in every block, from 64 in the initial block to 512 in the last one, while the dimensions of the image decreases. In **Figure** [18], the encoding process is repeated 3 times until it reaches the bottom where two more convolutional layers are applied without any Max Pooling layer.

The **decoder** is an extensive pathway that sequentially takes as input the concatenation between the up-convolutional features and the skip-connection as high-resolution features from the contracting path. Every block in the decoder consists of a transposed convolution for upsampling the feature map followed by a 2x2 convolution that divides by two the dimension of the feature channels. In addition, the transposed convolution is concatenated with the correspondingly cropped feature map from the decoder. This is done to add original information from the contracting path in order to gain more precision. Finally, each block ends with two 3x3 convolutions and ReLU activations functions. At the output layer a 1x1 convolution is applied to map the feature channel information to the output classes.

2.5.5 Transfer Learning

Transfer learning is a powerful and popular method in machine learning. It is based on the idea of re-using a pre-trained model for a different application within a similar domain. For example, a ResNet-50 neural network which has been trained on a large dataset such as ImageNet to classify images between 1000 different classes, can be re-used for a personalized application that requires classification between 38 completely new classes [59]. Training a model such as the mentioned above from scratch, without transfer learning, usually requires several hours of training, a substantial amount of data and the use of an appropriate hardware accelerator.

The intuition behind the concept of transfer learning is that a model trained with a very large dataset has already learned several general feature maps that are common to information in a similar domain. For instance, transfer learning is very popular in building convolutional based models, where a pre-trained model can be used as the "backbone" to recognize general features such as: texture, corners, shapes, among others, which will be the input to a last multi-layer perceptron for classification.

Figure [19] shows how a section of a pre-trained model can be exported and used into a new model. Depending on the approach, it is possible to retrain the whole model including the imported pre-trained weights with a personalized dataset or it is possible to “freeze” the pre-trained weights and only train the rest of the model, such as the classification layers.

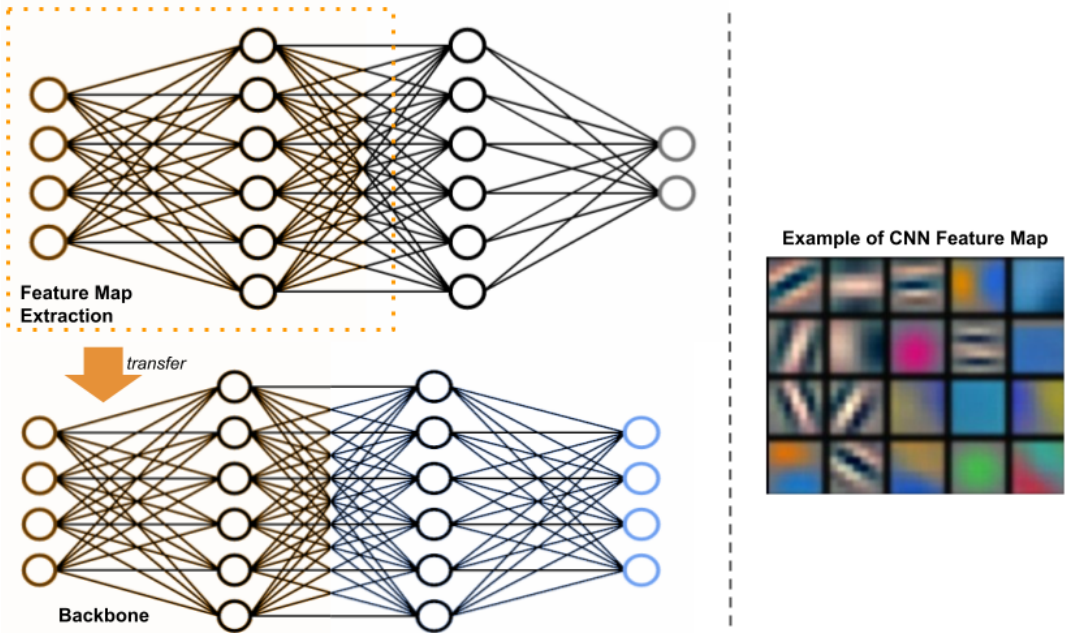

Figure 19: Transfer learning of feature extraction layers and example of CNN feature map.

2.5.6 Post-Training Quantization

The building block of any neural network is called a Perceptron, and is a mathematical equation based on a number of learning parameters such as weights and the bias, as shown in **Figure** [20]. During training, the value of each learning parameter is tuned until convergence has been reached or the last epoch has been executed. After the learning process, the model can be used for inference as a set of interconnected static parameters.

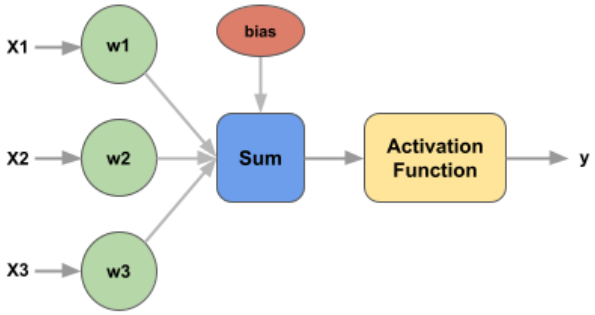

Figure 20: Perceptron model.

The learning parameters are stored and called from memory every time a process needs to run inference on some input. The standard representation of the values is 32-bit floating-point, which allows for a high level of accuracy. However, a deep learning model usually has millions of parameters and could end up taking a considerable amount of space in memory or may not even fit depending on the hardware. For instance, a well known computer vision network, the ResNet-50 contains around 26 million weights and 15 million activations, adding up to around 150 MB of required storage space. So, a model that runs effectively on a development environment, such as a server or a desktop computer, might not be able to run effectively on a lower-end device with hardware constraints. This is where quantization comes into the picture to further optimize a model to better fit a target device. In addition to reducing the required memory space, it also has an important impact on reducing latency and power consumption with little degradation of the model accuracy. It has been demonstrated that using weights and activations with 8-bit integer precision values does not result in significant loss in accuracy [67][57]. This is especially important for low-end devices that do not carry floating point functional units for arithmetic operations.

There are several post-training quantization approaches which are further explained in [70]. However the conservative approach is called “dynamic range quantization”. This quantization process statically quantized the weights from 32-bit floating point to 8-bit precision integer. This is the simplest form of post-training quantization and it is usually the first one to be contemplated. The general idea behind this is to map the min-max values of the floating point tensor to the min-max values of the 8 bit integer. Further work has to be done in order to avoid considering outliers as min or max values, and for this, some statistical clipping techniques are useful [27]. However, some neural networks such as MobilNet, require quantization-aware training to preserve accuracy as much as possible [44].

2.6 Deep Learning Inference at the Edge

Several systems have data intensive tasks that depend on computers with enough resources to process the information and draw conclusions from it. Deep learning is one of these tasks that until recently was mainly deployed via cloud computing [66]. This requires data to be moved from the source location, such as censoring stations or other Internet of Things devices [81], to a centralized location in the cloud. However, this approach comes with several challenges: i) Latency, ii) Privacy, iii) Power Consumption, iv) Scalability [33].

For instance, autonomous vehicles which constantly gather high frequency information from sensors such as LIDARs, radars and cameras, generate hundreds of bytes per second which need to be processed in real time to make a safety-critical decision. This type of system can not afford the delay between transmitting data packages to a processing server and waiting to receive an answer back from it. In addition, the exchange of information opens a potential privacy issue for the system to be hacked. Besides the delay and privacy issues, data transmission comes with a high power consumption cost, especially when it is done continuously and with high data bandwidth demands. Finally, even if the application can tolerate a delay, a fixed response time from the server is not entirely guaranteed due to the fact that accessing the cloud services can become a bottleneck depending on the number of connected devices.

Edge computing is a viable solution to meet the previously mentioned challenges by using optimized software for processing the data and running it on specialized hardware in-situ. This means that in most cases, data transmission is no longer required if the entire deep learning task can be executed in the hardware which is on the same location as the data acquisition system. In the case of the autonomous vehicle, this specialized hardware will be part of the electronic components of the car. Nevertheless, different challenges emerge with this new paradigm. The most important one to address is the limited resources available at the edge, such as: memory, power consumption and computing speed. It is important to take this into account to evaluate if the deep learning model can run properly on the selected edge device.

Companies such as *NVIDIA* are developing specialized hardware for highly demanding computation at the edge, which includes accelerator modules for Artificial Intelligence (AI) functionality built into them. Two of the most popular edge devices for AI acceleration will be mentioned above.

2.6.1 Graphics Processing Unit

The Graphics Processing Unit (GPU), has become one of the most important computing components due to its massive parallel processing capabilities. GPUs were initially designed to accelerate the rendering of 3D graphics in video games, but soon after becoming more flexible and programmable (GPGPUs [45]) they gained popularity among a

wide range of applications, including Deep Learning. It is important to note that a GPU has a more narrow purpose compared to a Central Processing Unit (CPU), which is fast parallel arithmetic operations. So, in a computing system both the CPU and GPU will be working together, with the CPU running orchestrating tasks such as the ones of an Operative System, and the GPU doing the heavy arithmetic operations.

Regarding its computing architecture, GPUs have a Single Instruction Multiple Data (SIMD) architecture which allows them to effectively do data parallelism among multiple processors, with only one instruction. A more specific type of SIMD architecture is called SIMT which adds multithreading to SIMD. It is important to mention that GPUs essentially handle matrix arithmetic operations in a very efficient way. Moreover, these operations are in floating-point format. This is why one of the main performance indicators for GPUs is the number of "Floating Point Operations per Seconds (FLOPS)". A GPU block diagram from NVIDIA can be seen in **Figure** [21].

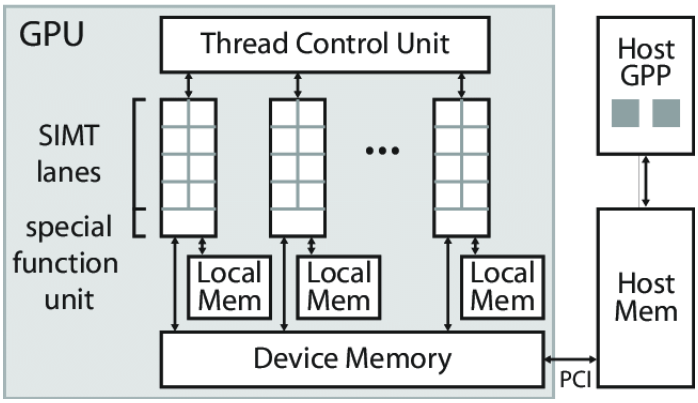

Figure 21: NVIDIA GPU block diagram showing internal architecture and system connectivity [52].

NVIDIA is a major global player in GPU development, and as part of its strategy to diversify the use of GPUs they created the Compute Unified Device Architecture (CUDA) as a parallel computing platform. This allows developers to further optimize their softwares running in NVIDIA GPUs [37]. NVIDIA has a series of GPUs especially designed for embedded AI applications, called the Jetson modules. These modules can be embedded into anything from robots, autonomous cars, drones, sensor arrays and computer vision systems that require AI capabilities with low to medium power consumption. Although NVIDIA provides this System-on-Module (SoM) to third parties device manufacturers, they also sell development kits for anyone who wants to try advanced machine learning deployment. One of these Jetson modules is the Jetson Xavier NX [8], which delivers around 21 Trillions Operations per Second (TOPS) while using only 10 to 15 watts of power depending on its configuration. As mentioned, a GPU comes accompanied with a CPU, which in the case of the Xavier is a 6-core NVIDIA Carmel ARM and a memory of 8 GB. It is important to mention the level of flexibility a developer can have to decide how to run its program in terms of number of cores and clock frequency in the CPU and GPU, which implies a design trade-off between throughput and power consumption.

Figure 22: Jetson Xavier NX developer kit

2.6.2 Tensor Processing Unit

The Tensor Processing Unit (TPU) is a special Application Specific Integrated Circuit (ASIC) design for AI acceleration by *Google* [5]. As its name describes, a TPU highly specializes in tensor processing. Although both TPUs and GPUs are capable of tensor operations, TPUs can take larger tensor operations which are more common in deep learning applications compared to 3D graphics rendering. The arithmetic architecture is made up of a Matrix Multiplication Unit and a Vector Processing Unit. One of the main approaches a TPU uses to speed up tensor operation is called "systolic array", which performs dot products of tensors on a single TPU instead of spreading the computation into multiple parallel cores like in the GPU. This is accomplished using multiply-accumulator units. A TPU block diagram can be seen in **Figure** [**23**].

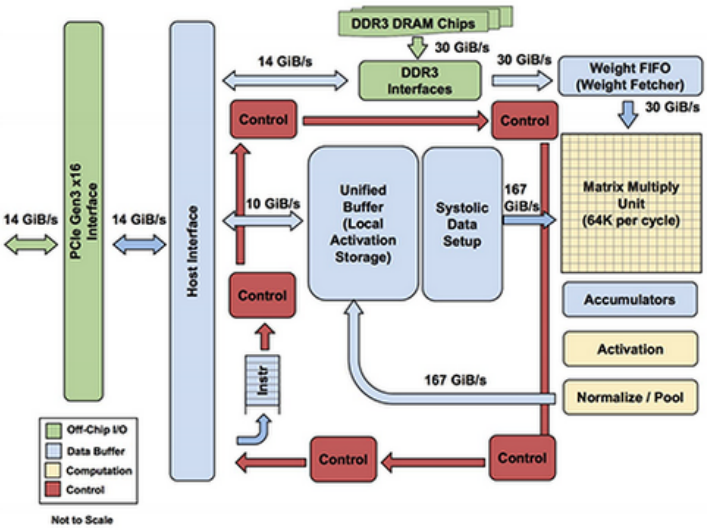

Figure 23: TPU Block Diagram showing functional units such Matrix Multiplier and Accumulators [5].

3 Design and Implementation

This section presents the design and implementation of the obstacle detection and proximity alert application. Including the selection of the required hardware, the training process of the U-Net model, its integration with stereo-depth estimations, and the deployment of the final application on the Jetson Xavier-NX.

3.1 Hardware Selection

There are three main functional requirements for the application: i) recognize nearby obstacles in front of the FishOtter, ii) estimate the distance to the obstacles, and iii) communicate the proximity distance of the obstacle to the user.

In addition, the following technical requirements should be considered in the design: i) Time response: taking into account the worst case scenario where the obstacle is 1 meter from the USV and considering the USV moves at 0.5m/s, the application should be able to process the stereo image and output an alert in less than 1 second so that appropriate collision avoidance strategies can be executed on time, this translates to a minimum processing speed of 1 frame per second, ii) Depth range: stereo estimations with less than 15% relative error and a sensing range of 1 to 15 meters. The application does not require high accuracy as the estimations will be given as interval distances of 1 meter. For example: "Obstacle detected between 2 to 3 meters". iii) Low number of false positives and false negatives during obstacle detection, justified by a high f1-score value. iii) Low power consumption: 10 Watts (twice the consumption of the Raspberry Pi 4).

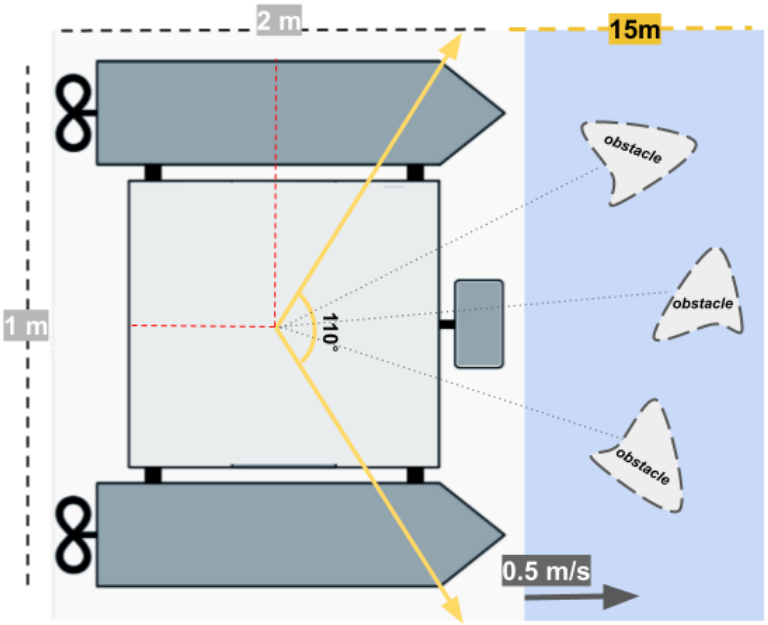

Figure 24: Diagram of the FishOtter dimensions to understand requirements.

3.1.1 Stereo Camera

The selection of the stereo camera is based on the following criteria:

1. **Small size:** suitable dimensions for mounting it on a small USV.
2. **Wide-angle field of view (>100°):** considering the FishOtter is 2 meters long by 1 meter broad, positioning the stereo-camera in the middle as shown in **Figure [24]** will cover the entire front view of the USV, leaving no hidden angles.
3. **Water resistant enclosure (>IP65):** considering the FishOtter is a maritime vehicle constantly exposed to water.
4. **Range of depth estimation:** considering this is a near-field application, it should be capable of at least 15m depth detection range.
5. **SDK - Library:** software to control the camera and compute the depth estimates for rapid prototyping.

The *Stereolabs ZED 2i* industrial stereo camera [16], presented in **Figure [25]** was selected due to its characteristics: i) Dimensions: 175.25x30.25x43.10 mm, ii) Baseline: 12 cm, iii) Several resolutions: 4416x1242, 3840x1080, etc, iv) Depth Range: 0.2-20 m, v) Focal length lens: 4 mm for increased resolution and depth accuracy at longer range, vi) Field of View: 110°(H)x70°(V)x120°(D)max, vii) Polarizing filter to avoid reflections, viii) flexible camera control: resolution, brightness, contrast, saturation, exposure.

Figure 25: ZED2i stereo-camera [16].

3.1.2 Edge Computing Device

Considering the nature of the stereo-depth calculations, the Stereolabs-SDK uses CUDA to accelerate its computations through GPU parallel processing. In addition, the motivation of using deep learning for this application reinforces the selection of an appropriate GPU. For experimental analysis, two developer kits from the Jetson series were chosen: Jetson Nano [7] and Jetson Xavier-NX [8]. The first one shows benchmark performance of 0.5 TFLOPS (FP16) with power consumption between 5 to 10 Watts. While the second one has more than 3 times higher throughput (1.7 TFLOPS) with less than twice the power consumption (10 - 15 Watts). In addition, the difference in price is around a factor of x3.

So, with the objective of using in the future only one device for the whole autonomous navigation system (including DUNE and the present obstacle detection application), the Jetson Xavier-NX was selected for its high performance and efficiency in order to meet the functional and technical requirements mentioned above. In addition, Stereolabs offers the Jetson Xavier as part of a robust water resistant package (IP66) called the ZED Box which is suitable for deployment in the USV. **Figure** [**26**] presents the acquired hardware.

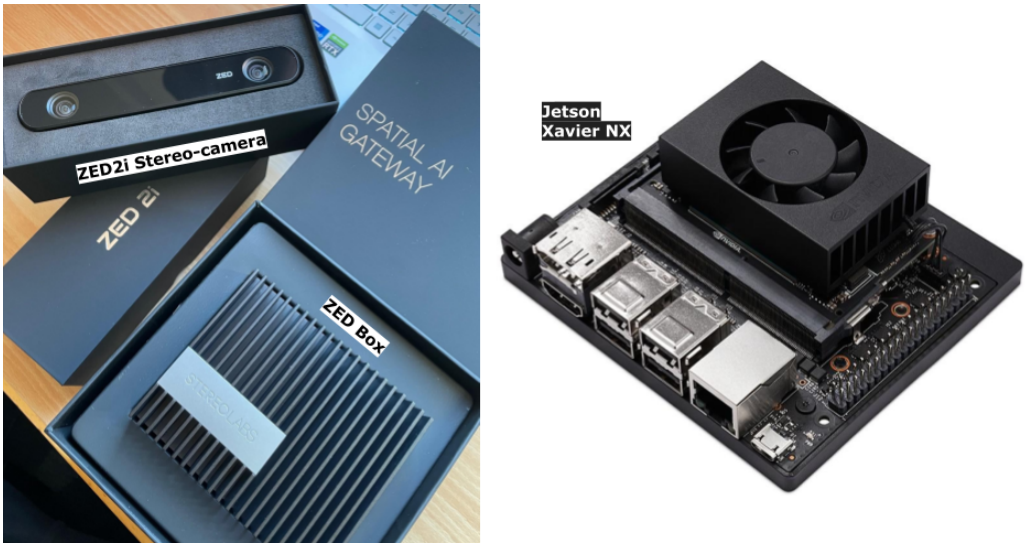

Figure 26: Stereo Labs ZED 2i Camera and ZED Box

3.1.3 Hardware Integration to FishOtter-USV

In order to mount the stereo-camera and the Jetson device on the FishOtter, some modifications to the USV control box are required. **Figure** [28] shows the integration of the Jetson Xavier-NX in the control box and the stereo-camera on the FishOtter. The Xavier-NX is powered using the 24V output from the batteries, and the stereo-camera is connected with an especially modified USB cable to fit the water proof standards of the control box. In addition, the Jetson Xavier-NX is connected to the internet router, and configured with the university VPN for remote connection in headless mode. The integration between the FishOtter and the stereo-system is tested during a field excursion to the Børsa fjords, Norway, as seen in **Figure** [27]. The objective is to verify the integration of the system at sea and record relevant data in the form of videos to assess real conditions.

Figure 27: Field trip to Børsa Fjord with FishOtter ready to record stereo videos.

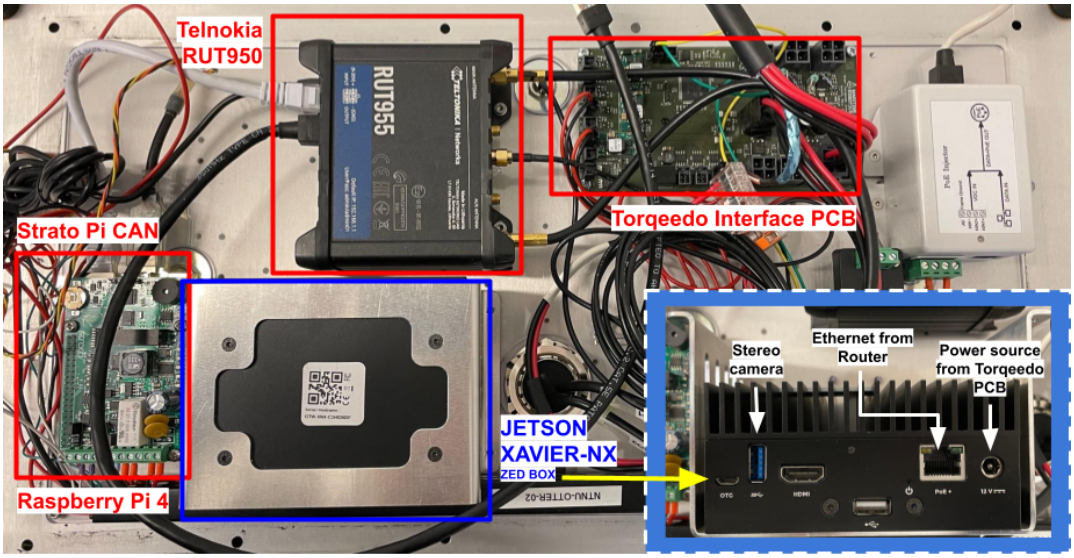

Figure 28: Control Box of FishOtter with the Jetson Xavier-NX

3.2 Obstacle Detection - Design and Implementation

Deep learning approaches have shown to be more robust than traditional computer vision techniques for obstacle detection at sea, especially against adverse environmental conditions such as: light exposure, fog attenuation and varying camera angles due to waves constantly rocking the USV. The current solution contemplates doing semantic segmentation with a U-Net architecture for the following main reasons:

1. U-Net has a lightweight structure compared to other networks, therefore associated with fast predictions.
2. U-Net has very good performance in segmentation tasks, especially with few output classes.
3. U-Net allows for transfer learning by embedding a pre-trained CNN for feature extraction such as MobilNet or ResNet on the encoder section.
4. U-Net architecture implemented with skip-connections requires less amount of data for achieving a good performance.
5. U-Net will provide as output all the pixels classified as objects from which depth must be estimated using the stereo-camera, rather than just four points from an object detection model such as YOLO.
6. Semantic segmentation gives more flexibility to design a neural network, considering the other popular option is YOLO, which is mostly fixed in structure and number of parameters.
7. Semantic segmentation is State of the Art for autonomous systems, which could represent a first step for this USV on its journey to become an entirely autonomous navigation system.

3.2.1 U-Net Model Overview

The U-Net architecture is selected due to being known for fast and precise segmentation of images. In addition, its encoder-decoder structure allows for meaningful training with less amount of data compared to other approaches. On top of that, it is considered flexible, allowing for quick iteration while searching for the best tradeoff between complexity (number of parameters) and performance. In addition, a very relevant technique called transfer learning is applied to the encoder side of the model. As already mentioned, transfer learning uses a pre-trained model and uses its learned weights for feature extraction. MobilNetV2, designed for applications running on limited computing resources, is selected as the backbone due to its small size compared to other known models such as Resnet18 (which is approximately 3 times bigger). After several iterations, where configurations such as the depth of the network and size of the filters are assessed, the best architecture that yields good performance while remaining limited in the amount of parameters is shown in **Figure** [29]. The implemented model has a total number of 1,811,841 trainable parameters. As a reference, YOLOv3 has 8,861,918 trainable parameters. This U-Net working model will be from now on addressed as the OtterNet.

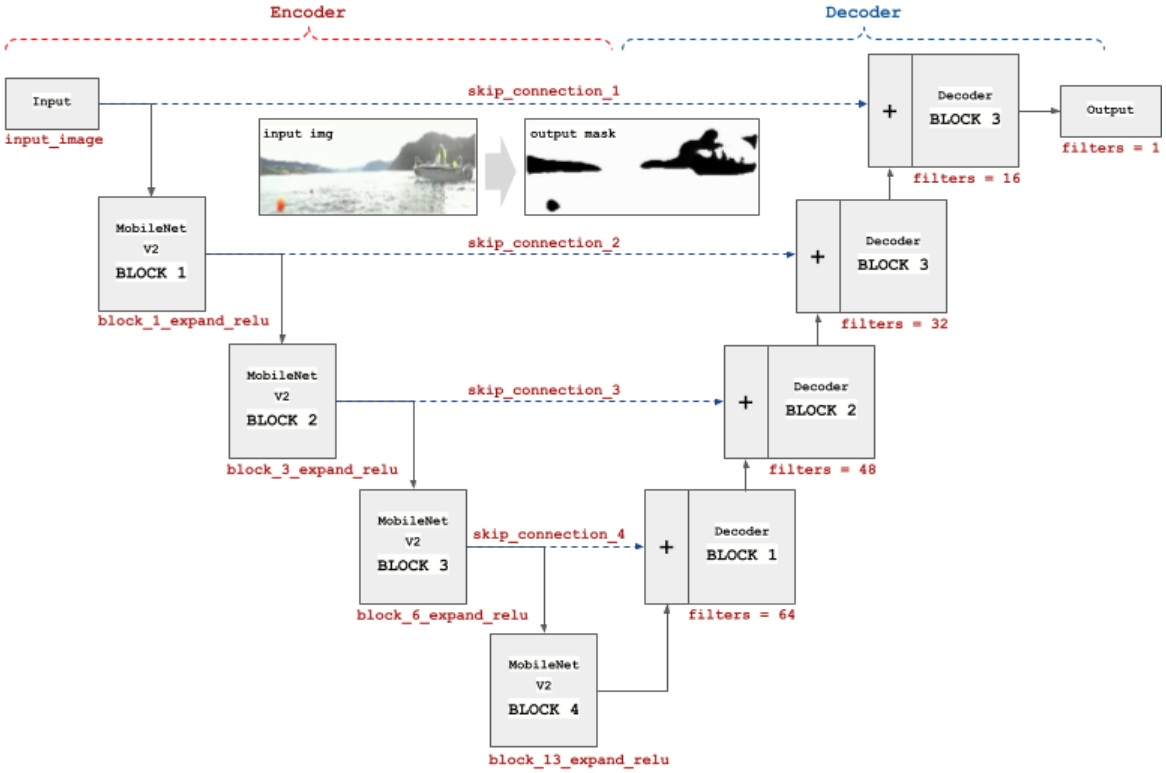

Figure 29: U-Net model - the OtterNet.

3.2.2 Dataset

One of the most important ingredients to any neural network implementation is the dataset. There are only a few marine obstacle detection datasets available online, the most relevant to this project are presented in [12]. For this implementation, the "Marine Semantic Segmentation Training Dataset" (MaSTr1325 [30]) is selected due to the fact that it is tailored for development of obstacle detection methods in small-sized coastal USVs, which resembles our application needs and conditions. The MaSTr1325 dataset includes a total of 1325 images and was built from recordings on the gulf of Koper, Slovenia. Besides considering different weather conditions and times of day to enrich the variety of the data, the images were handpicked to include a wide diversity of marine obstacles such as: cargo ships, sail boats, portable piers, swimmers, rowers, buoys, seagulls, among others. The per-pixel labeling of the images was carried out by human annotators and verified and corrected by an expert to ensure quality. The following four labels are: i) Obstacles and environment (value 0), ii) Water (value 1), ii) Sky (value 2), iv) Ignore region/unknown category (value 4). **Figure [30]** shows samples from the dataset. Due to the fact that the model will work only with two classes, the four different labels are reduced to two: i) obstacle (value 0) which is the same as the obstacles and environment category of the original labeling, and ii) not obstacle (value 1) which is every other category from the original labeling criteria. The script that is used to transform the MaSTr1325 labels to the format required by the OtterNet training script is also included in the GitHub repository of this project.

Figure 30: Samples from MaSTr1325 dataset [12].

3.2.3 Building the Otter Dataset

Although the public dataset contains an important variety of images that are relevant to our application, the neural network requires more images from similar conditions to the ones the FishOtter is exposed to. So, with the idea of building a dataset, 300 images are handpicked from 60 videos (with length of approximately 2 minutes each at 15 frames/sec) recorded at the fjords (Børsa, Norway). Following the MaSTr1325 dataset methodology, special care is taken to account for the different lighting and environmental conditions such as rain and fog, plus different angles the camera might be at depending on the waves rocking the USV. To label the images, MATLAB's application called "Image Labeler" is used. Approximately 8 minutes per image are needed, and two classes are considered: i) obstacle and environment and ii) not obstacle (sea and sky). **Figure** [31] shows the result of such activity. From now on, the output binary classification which can be seen as the black and white image will be referred to as the output mask. The script that is used to transform the MATLAB labels to the format required by the OtterNet training script is also included in the GitHub repository of this project. It is important to mention that to plot the black and white output masks, a pixel intensity conversion is needed in order to assign a 255 value to every pixel that is not classified as an obstacle.

Figure 31: Example of images and annotations from the Otter Dataset.

3.2.4 Data Augmentation

Considering the limited amount of data that is available to train the model, performing data augmentation during training is beneficial so the model can get more variety of data to learn from. The artificially generated images from data augmentation have to be coherent to what the FishOtter could encounter under real navigation conditions. For instance, due to the waves rocking the USV, several pictures of the same object could be taken with slightly rotated angles in the horizontal plane. In addition, light exposure may change quite drastically depending on the hour of the day and season, which could be artificially induced by attenuating the images. Overall, data augmentation techniques are performed equally to both the images and output masks in a random manner through the following tasks:

- **Horizontal Flipping:**

Figure 32: Examples of Horizontal Flipping using samples from the Otter Dataset.

- **Random Rotation:** 5% to 15%

Figure 33: Examples of Rotation using samples from the Otter Dataset.

- **Random Light Attenuation:** 20%

Figure 34: Examples of Light Attenuation using samples from the Otter Dataset.

3.2.5 Neural Network Training

This section dives into the technical side of the deep learning implementation. Although the most relevant sections of the code are presented here, the complete script with detailed comments and instructions on how to run it can be found in the GitHub repository of this project.

Google Colaboratory is used to build and train the proposed U-Net neural network, due to the fact that it allows for accelerated training using GPU or TPU processing units. Furthermore, it can be used by any developer independently of their own development environment configuration. In addition, TensorFlow is the Deep Learning framework of choice due to previous experiences that the author has with it. **Figure** [35] shows an overview of the designed training pipeline.

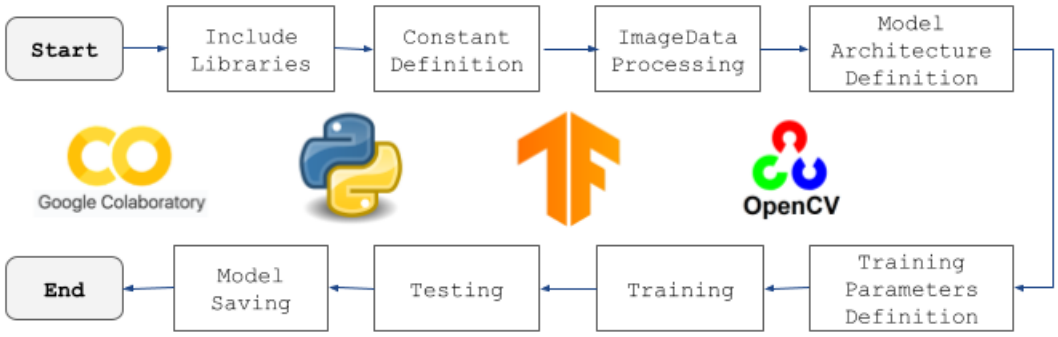

Figure 35: Overview of U-Net training pipeline.

Include Libraries
In this section, the following libraries are included: i) Tensorflow: Open source library created by Google for end to end development of neural networks, ii) Scikit-learn: Open source library based on python for machine learning applications, iii) OpenCV: Open source Computer Vision library, iv) Numpy: Open source library based on python, highly optimized to work with large multidimensional vectors and matrices, along with math functions to operate on them, v) Glob: File management python library used to return all files from a specific path.

Listing 1: Include Libraries

```
import os
import numpy as np
import cv2
from glob import glob
import tensorflow as tf
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
```

Constant Definition

This section defines several constants, such as: Input Image Size, Image File Path, Target (Mask) File Path and Model Save Path.

Listing 2: Constant Definition

```
IMG_SIZE = 256
path_img = "...PATH/Dataset_Images/"
path_target = "PATH/Dataset_Masks/"
```

Image Data Processing

In this section the input images and output masks are processed before they are fed into the neural network. The following steps are executed for the input image processing: i) Read image from path using OpenCV, ii) Resize image to a standard size of 255x255 pixels as required by the MobileNetV2 network as input, iii) Normalize the pixel intensity by dividing the image matrix by its maximum value (255), resulting in 255x255x3 matrix of floating point values between 0 and 1. On the other hand, the output masks are processed in a similar way: i) Read image from path as grayscale using OpenCV, ii) Resize image to a standard size of 255x255 pixels, iii) No need to normalize, the pixel intensity represent the two classes: zero (not object) and one (object), iv) Expand one dimension of the array to result in 255x255x1 for training purposes. Finally, the train, validation and test dataset are selected from the whole MaSTr1325 dataset in a random way following a 80%, 10% and 10% ratio respectively. The train test split function from sklearn is used for this purpose. Overall, 1061 instances are used for training, 132 for validation, and 132 for testing.

Listing 3: Image Data Processing

```
def load_img(path_img, path_target, split=0.1):
    #Obtain all input and target file paths.
    images = sorted(glob(os.path.join(path_img,"*")))
    target = sorted(glob(os.path.join(path_target,"*")))
    #Randomly select 10% as validation data.
    train_x, valid_x = train_test_split(images,
                        test_size=int(split*len(images)),
                        random_state=42)
    train_y, valid_y = train_test_split(target,
                        test_size=int(split*len(images)),
                        random_state=42)
    #Randomly select 10% as testing data.
    train_x, test_x = train_test_split(train_x,
                       test_size=int(split*len(images)),
                       random_state=42)
    train_y, test_y = train_test_split(train_y,
                       test_size=int(split*len(images)),
                       random_state=42)
    return (train_x,train_y),
           (valid_x,valid_y),
           (test_x,test_y)
```

```
def read_image(path):
    #Get path from file input image.
    path = path.decode()
    #Read image from path using OpenCV.
    img = cv2.imread(path)
    #Resize image to 255x255x3.
    img = cv2.resize(x,(IMG_SIZE,IMG_SIZE))
    #Normalize image.
    img = img/255.0
    return img

def read_target(path):
    #Get path from file target mask.
    path = path.decode()
    #Read image from path as greyscale using OpenCV.
    msk = cv2.imread(path,cv2.IMREAD_GRAYSCALE)
    #Resize target to 255x255.
    msk = cv2.resize(msk,(IMG_SIZE,IMG_SIZE))
    #Define target at floating point.
    msk = msk/1.0
    #Add one dimension to image array.
    msk = np.expand_dims(msk,axis=-1)
    return msk

def call_convert(img,msk):
    def _convert(img,msk):
        #Call data processing functions.
        img = read_image(img)
        msk = read_target(msk)
        return img, msk
    #Define image and target mask formats.
    img,msk = tf.numpy_function(_convert,[img,msk],
            [tf.float64,tf.float64])
            img.set_shape([IMG_SIZE,IMG_SIZE,3])
            msk.set_shape([IMG_SIZE,IMG_SIZE,1])
    return img,msk

def parse_dataset(img,msk,BATCH):
    #Process input image and target mask data through map function.
    data_set = tf.data.Dataset.from_tensor_slices((img,msk))
    data_set = data_set.map(call_convert)
    data_set = (data_set
    .shuffle(BATCH*100)
    .batch(BATCH)
    .map(lambda i,j:(trainAug(i),j),
    num_parallel_calls=tf.data.AUTOTUNE)
    .prefetch(tf.data.AUTOTUNE))
    data_set = data_set.repeat()
```

```
    return data_set

#Do data augmentation on the training dataset.
train_augmentation = Sequential([
        preprocessing.Rescaling(scale=1.0 / 255),
        preprocessing.RandomFlip("horizontal"),
        preprocessing.RandomZoom(
        height_factor=(-0.05, -0.15),
        width_factor=(-0.05, -0.15)),
        preprocessing.RandomRotation(0.2)
])
```

Model Architecture Definition

After several try and error iterations to find the most suitable architecture and hyper-parameters, the proposed U-Net model is implemented in this section using Tensorflow. To this end, the following steps are followed: i) Build the encoder using the first 13 main blocks of the MobileNetV2, ii) Select the start of the skip-connection from the MobileNetV2 blocks, iii) Build first input section of the decoder by concatenating the output of the encoder and the last skip-connection, along with two 2D convolutional layers with the biggest filter size, iv) Keep building the decoder by concatenating skip-connections and the output of the previous decoder block along with the two 2D convolutional layers with smaller filter size, until there are not skip-connections left to do, v) Build the last layer as a 2D Convolutional layer with 1 size filter and (1,1) size kernel in order to have at the output a 255x255x1 dimension, which should be the same as the output mask data. The structure has: i) 2D Convolution Filter size from = [16, 32, 48, 64], ii) 2D Convolution Kernel Size = (3,3), iii) Number of Skip-Connections = 4, and iv) Activation Layers = Relu (Decoder Block) and Sigmoid (Output Layer).

Listing 4: Model and Training Parameters Definition

```
def OtterNet():
  #Define input layer with size and name.
  inputs = Input(shape=(IMG_SIZE, IMG_SIZE, 3),
                 name="input_image")
  #Load MobileNetV2:
  #input layer is assigned to input of model.
  #weights are pre-training on ImageNet
  #do not include fully-connected layer at the top.
  #alpha > 1 proportionally increases number of filters per layer.
  encoder = MobileNetV2(input_tensor=inputs, w
                        eights="imagenet",
                        include_top=False,
                        alpha=1.3)
  #Get MobileNetV2 specific layer output.
  encoder_output = encoder.get_layer("block_13_expand_relu").output
  #Get MobileNetV2 specific layer output as skip connection.
  x_skip = encoder.get_layer("block_6_expand_relu").output
  x = UpSampling2D((2, 2))(encoder_output)
```

```
    x = Concatenate()([x, x_skip])
    x = Conv2D(64, (3, 3), padding="same")(x)
    x = BatchNormalization()(x)
    x = Activation("relu")(x)
    x = Conv2D(64, (3, 3), padding="same")(x)
    x = BatchNormalization()(x)
    x = Activation("relu")(x)
    #Get MobileNetV2 specific layer output as skip connection.
    x_skip = encoder.get_layer("block_3_expand_relu").output
    x = UpSampling2D((2, 2))(x)
    x = Concatenate()([x, x_skip])
    x = Conv2D(48, (3, 3), padding="same")(x)
    x = BatchNormalization()(x)
    x = Activation("relu")(x)
    x = Conv2D(48, (3, 3), padding="same")(x)
    x = BatchNormalization()(x)
    x = Activation("relu")(x)
    #Get MobileNetV2 specific layer output as skip connection.
    x_skip = encoder.get_layer("block_1_expand_relu").output
    x = UpSampling2D((2, 2))(x)
    x = Concatenate()([x, x_skip])
    x = Conv2D(32, (3, 3), padding="same")(x)
    x = BatchNormalization()(x)
    x = Activation("relu")(x)
    x = Conv2D(32, (3, 3), padding="same")(x)
    x = BatchNormalization()(x)
    x = Activation("relu")(x)
    #Get MobileNetV2 specific layer output as skip connection.
    x_skip = encoder.get_layer("input_image").output
    x = UpSampling2D((2, 2))(x)
    x = Concatenate()([x, x_skip])
    x = Conv2D(16, (3, 3), padding="same")(x)
    x = BatchNormalization()(x)
    x = Activation("relu")(x)
    x = Conv2D(16, (3, 3), padding="same")(x)
    x = BatchNormalization()(x)
    x = Activation("relu")(x)
    #Last layer.
    x = Conv2D(1, (1, 1), padding="same")(x)
    x = Activation("sigmoid")(x)
    OtterNet = Model(inputs, x)
return OtterNet
```

Training Parameters Definition
In this section, the following hyper-parameters are defined: i) Epochs = 30, ii) Batch size = 32, iii) Learning rate = 1e-4. In addition, the f1-score based loss function is defined as shown in **Equation (5)**. On the other hand, Nadam (Nesterov Momentum into Adam) is selected as the optimizer.

$$loss\_coefficient = 1 - \frac{2 * Precision * Recall}{Precision + Recall} \tag{5}$$

Listing 5: Training Parameters Definition

```
#Hyper-parameters definition:
epochs = 30
bach_size = 32
learning_rate = 1e-4

#F1-score coefficient definition.
def f1_score_coef(y_true, y_pred):
    #Constant to avoid division by zero.
    smooth = 1e-7
    #Flatten real target data into 1D array.
    y_true = tf.keras.layers.Flatten()(y_true)
    #Flatten prediction target data into 1D array.
    y_pred = tf.keras.layers.Flatten()(y_pred)
    #Apply F1_Score formula:
    intersection = tf.reduce_sum(y_true * y_pred)
    f1_score_coef = ((2. * intersection + smooth) /
                     (tf.reduce_sum(y_true) +
                     tf.reduce_sum(y_pred) +
                     smooth))
    return f1_score_coef

#F1-score loss definition.
def f1_score_loss(y_true, y_pred):
    #Calculate F1_Score Loss from coefficient.
    return 1.0 - f1_score_coef(y_true, y_pred)

#Compile model with: loss, optimizer with learning rate, and metric.
OtterNet.compile(loss=f1_score_loss,
                 optimizer=tf.keras.optimizers.Nadam(learning_rate),
                 metrics=[f1_score_coef])
```

Training

In this section, two callback functions are instantiated: i) Early Stopping: form of regularization used to avoid over-fitting during training if the monitored metric does not improve after a number of epochs called the "patience", and ii) Reduce Learning Rate: reduces the learning rate during training by a given "factor" after no improvement is seen for a "patience" number of epochs.

Listing 6: Training

```
#Callbacks definition for early stopping and reducing learning rate.
callbacks = [EarlyStopping(monitor='val_loss', patience=10)
    ReduceLROnPlateau(monitor='val_loss',factor=0.1, patience=10)]

#Process and load train and validation data with utility functions.
train_dataset = parse_dataset(train_x, train_y, batch=batch)
valid_dataset = parse_dataset(valid_x, valid_y, batch=batch)

#Number of train and validation steps are needed to point out how
#many batches (integer number) will be executed during an epoch.
train_steps = len(train_x)//batch_size
if len(train_x) % batch_size != 0: train_steps += 1
val_steps = len(valid_x)//batch_size
if len(valid_x) % batch_size != 0: val_steps += 1

#Model starts training.
history = OtterNet.fit(train_dataset, validation_data=valid_dataset,
            epochs=epochs, steps_per_epoch=train_steps,
            validation_steps=val_steps, callbacks=callbacks)
```

Figure [36] shows how the validation loss is closely decreasing at a similar ratio as the training loss, which is a good indicator of how well the model is learning. In addition, it is also an indicator that the model is not over-fitting the training dataset based on the gap it still exists between both curves.

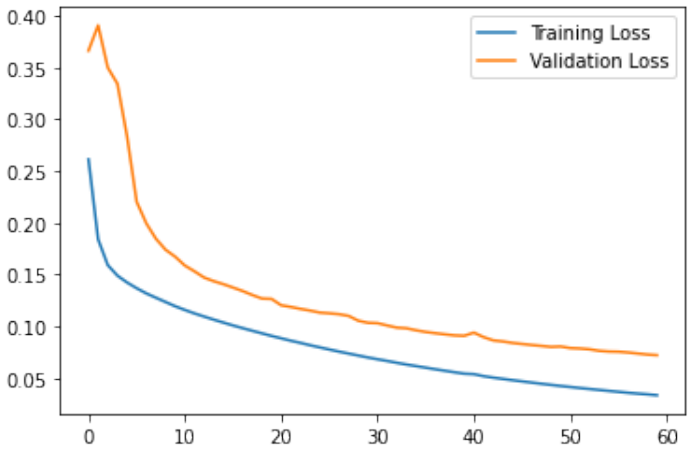

Figure 36: Learning curve of best U-Net model.

Testing

In this section, the test dataset is used to see how well the neural network is capable of performing with data it has never seen before during training.

Listing 7: Testing

```
#Load OtterNet trained model.
test_model = tf.keras.models.load_model("PATH.../otternetvx.h5",
                 custom_objects={'dice_loss':f1_score_loss,
                 'dice_coef':f1_score_coef})

#Process and load test dataset using the utility function.
test_dataset = parse_dataset(test_x, test_y, batch=batch)

#Number of test steps are needed to point out how
#many batches (integer number) will be executed during an epoch.
test_steps = (len(test_data)//batch_size)
if n_data_test % batch_size != 0: test_steps += 1

#Evaluate model will give back the metric scores.
OtterNet.evaluate(test_dataset, steps=test_steps)
```

Model Saving

In this section, the model is saved for two purposes: i) to keep track of the different model versions, ii) to save the model and its dependencies (architecture, weights, optimizer, training configurations) to re-create the model even without the code that originally created it.

Listing 8: Model Saving

```
model.save("PATH/otternet.h5")
```

3.3 Obstacle Detection and Proximity Alert Application - Design and Implementation

To integrate the different components of this project, such as: camera control, image processing, deep learning inference, and proximity alert generation, an application is built using C++. As it is known, C++ is the preferred choice when it comes to performance-critical applications that require efficient memory management while maintaining high speed rates.

3.3.1 Application Overview

As required, the application should run in real-time and at the edge. Which means it's important not to over-dimension the functionalities and make good use of the available resources. Further details on how to optimize the application based on performance analysis can be found in the next chapter. The behavior of this application and how the different components and tasks interact with each other are presented in a general view on **Figure** [37]. As it can be seen, the application will receive left and right frames from the stereo-camera. The semantic segmentation UNET model, called OtterNet, will take the left frame and run inference on it, resulting in the binary classification of each pixel (output mask). Although the **Figure** [37] shows both the inference and depth map extraction occurring in parallel, this is not the case on the application, as they are executed sequentially. Future work can be done if a parallel execution is needed in order to increase throughput. Nevertheless, once the pixel class and the depth map are obtained, the next step is to identify the proximity of each pixel in 5 distance intervals, to finally notify if an object is found on one of these intervals for more than N number of frames.

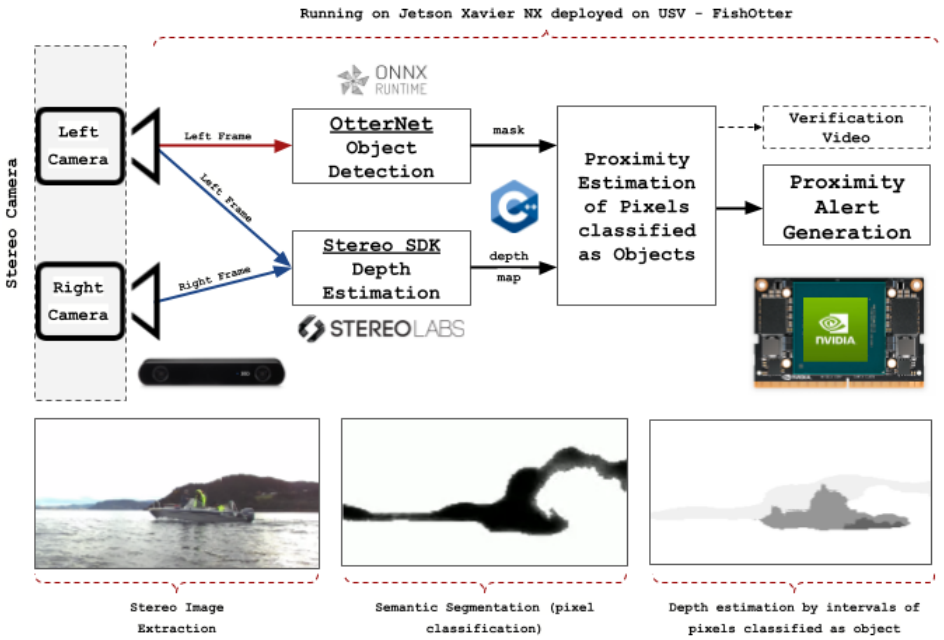

Figure 37: Obstacle detection and proximity alert application overview.

3.3.2 Application - Development

This section dives into the technical side of the application development. The most relevant sections of the code are presented here. Nevertheless, the complete source code with detailed comments and instructions on how to build and run it can be found in the GitHub repository of this project. It is important to mention that the code also generates a "verification video" from all the processed frames in real time which is not necessarily needed in a fully verified production environment. This functionality is important when verifying that the application is behaving as expected but it can be removed to increase performance. *Visual Studio 2022* is used to develop the proposed application.

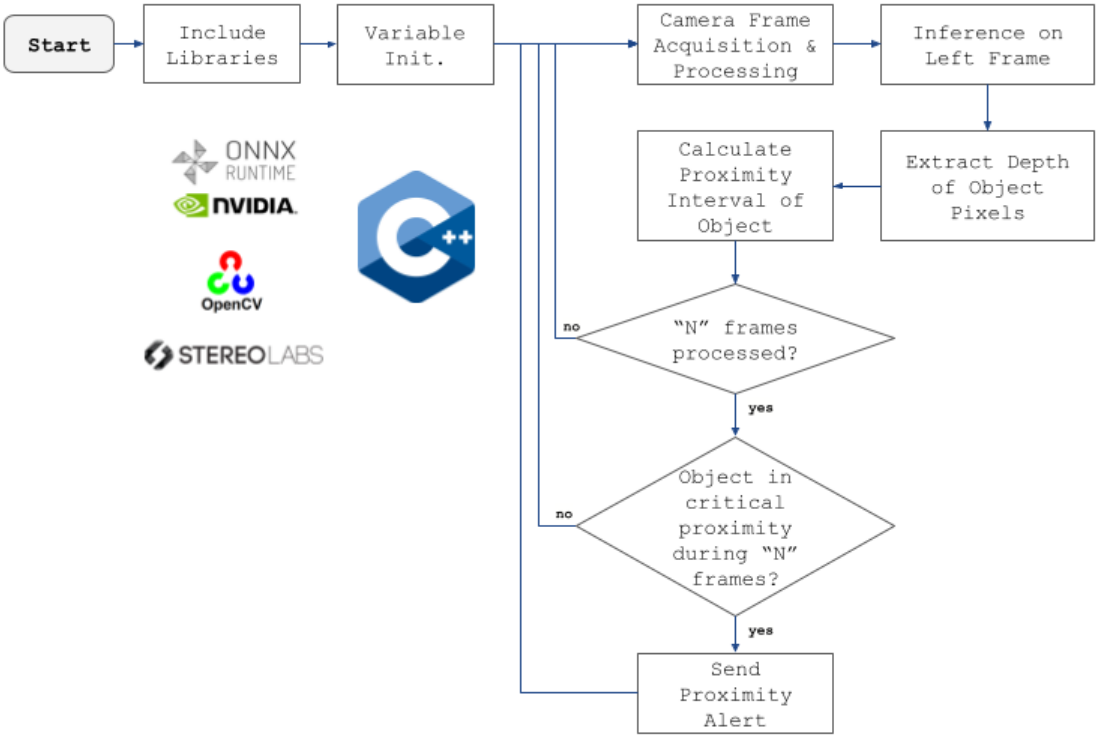

Figure 38: Block diagram of obstacle detection and proximity alert application

Include Libraries

In this section, the following libraries are included: i) OpenCV: used for image data manipulation and processing, ii) Onnx Runtime: used for run inference on the object detection neural network model, iii) Stereo Labs SDK: used for stereo camera data acquisition and depth estimation.

Listing 9: Include Libraries

```
#include <sl/Camera.hpp>
#include <iostream>
#include <onnxruntime_cxx_api.h>
#include <opencv2/opencv.hpp>
#include "utils.hpp"
#include "Helpers.cpp"
```

Variable Initialization

In this section, several constant values and variables are defined and initialized. There are two clear groups of variables that correspond to the ones needed by Stereolabs and ONNX Runtime libraries. To run inference, an environment must be first instantiated. This environment is the main entry point of the ONNX Runtime to create a so-called session for inference. In addition, the memory space for the input and output tensors following the ONNX format are generated based on the input image and output mask dimensions used to train the network. The model will run inference using the address space of the input and output tensors as reference, expecting to find the input image on the input tensor so it could leave the resulting output mask on the output tensor. Finally, the inference session is instantiated in the environment and the ONNX extension model is loaded through a reference to the file path.

Listing 10: Variable Initialization - ONNX

```
Ort::Env env;
Ort::RunOptions runOptions;
Ort::Session session(nullptr);
Ort::AllocatorWithDefaultOptions ort_alloc;
auto memory_info =
Ort::MemoryInfo::CreateCpu(OrtDeviceAllocator,OrtMemTypeCPU);

//Define path from where onnx model will be loaded.
const char*  modelPath = L"PATH\\otternet.onnx";

//Define Input and Output dimension (W,H,C).
constexpr int64_t width = 256;
constexpr int64_t height = 256;
constexpr int64_t numChannels = 3;
constexpr int64_t numClasses = 1;

//Define and initialize the Input and Output Tensors.
const array<int64_t,4> inputShape = {1,height,width, numChannels};
const array<int64_t,4> outputShape = {1,height,width, numClasses};

//Define input and output vector-flatten size.
constexpr int64_t numOutputClasses = numClasses*height*width;
constexpr int64_t numInputElements = numChannels*height*width;
float* input = new float[numInputElements];
float* results = new float[numOutputClasses];
auto inputTensor =
Ort::Value::CreateTensor<float>(memory_info,input,
numInputElements,inputShape.data(),inputShape.size());
auto outputTensor =
Ort::Value::CreateTensor<float>(memory_info,results,
numOutputClasses,outputShape.data(),outputShape.size());

//Create ONNX Session with OtterNet model.
session = Ort::Session(env, modelPath, Ort::SessionOptions{nullptr});
```

```
//Define Input and Output names from the OtterNet.
char* inputName = session.GetInputName(0,ort_alloc);
char* outputName = session.GetOutputName(0,ort_alloc);
const array<const char*,1> inputNames = {inputName};
const array<const char*,1> outputNames = {outputName};
```

The next group corresponds to Stereolabs-SDK variable initialization, from which the most relevant is the initialization of the camera's object which will be used for camera parameter configuration, such as sample frequency (FPS), and image extraction.

Listing 11: Variable Initialization - Stereo Camera

```
//Create zed camera object and parameters.
Camera zed;
InitParameters init_parameters;

//Set camera resolution to HD2K = 2208x1242x3.
init_parameters.camera_resolution = RESOLUTION::HD2K;

//Set 10 frames per second.
init_parameters.camera_fps = 10;

//Set depth measurements in millimeters.
init_parameters.depth_mode = sl::DEPTH_MODE::PERFORMANCE;
init_parameters.coordinate_units = UNIT::MILLIMETER;

//Configure camera object with defined parameters.
auto returned_state = zed.open(init_parameters);
auto resolution =
zed.getCameraInformation().camera_configuration.resolution;
```

Camera Frame Acquisition and Image Processing

In this section, first the camera object previously initialized is used to extract the left frame under specific configuration conditions shown at the code. Next, with the help of OpenCV the following steps are executed to fulfill the input data shape and pixel value required by MobilNetV2: i) resizing to 256x256, ii) flattening the image as required by ONNX input tensor, iii) normalizing the pixel values between 0 and 1.

Listing 12: Camera Frame Acquisition and Image Processing

```
//Fetch Left frame from stereo-camera.
zed.retrieveImage(svo_image, VIEW::LEFT, MEM::CPU, low_resolution);
cv::Mat image = svo_image;

//Resize the image to the input dimension of the OtterNet.
resize(image, image, Size(256, 256));

//Reshape the image to 1D vector.
image = image.reshape(1, 1);

//Normailze number to between 0 and 1 and convert to vector<float>.
vector<float> imageVec;
image.convertTo(imageVec, CV_32FC1, 1. / 255);
```

Run Inference on Left Frame

In this section, the processed image frame is copied to the input tensor space in memory and then the inference session runs with the following relevant inputs: i) input layer name of the model, ii) address pointing to the input tensor memory space, iii) output layer name of the model, and iv) address pointing to the output tensor memory space where the inference result will be stored.

Listing 13: Run Inference

```
//Copy image data to tensor input array.
copy(imageVec.begin(), imageVec.end(), input);

//Run Inference.
session.Run(runOptions, inputNames.data(), &inputTensor,
            1, outputNames.data(), &outputTensor, 1);
```

Depth Map Extraction

In this section, the depth map of the stereo images is computed by calling a function from Stereolabs-SDK. This function will calculate the depth of each pixel in the image.

Listing 14: Extract Depth Map

```
zed.retrieveMeasure(depth_map, MEASURE::DEPTH);
```

Calculate Proximity Interval of Object

In this section, the output of the neural network in addition to the depth map are used to estimate how far an object is. To this end, the resulting binary output (*target_mask*) is evaluated pixel by pixel through its entire dimension (which after resizing is the same as the camera frame dimensions). If the pixel is classified as an object, then its depth is extracted from the depth map. Next, each pixel is classified according to its pixel depth in one of the 5 proximity distance intervals (less than 1m, 1-2m, 2-3m, 3-4m, and 4-5m) and a counter (*pixel_count_Xmtr*) takes note of how many pixels are classified in each interval.

Listing 15: Calculate Proximity Interval of Object

```
//Convert the output mask from flatten output to 2D image dimensions.
img_result = cv::Mat(256, 256, CV_32FC1, results);
//Output mask is 256x256, so it must be resized back to the
//original stereo-camera frame resolution because the depth map is
//calculated for that frame size.
cv::resize(img_result, img_result, Size(frameWidth, frameHeigth));

for (size_t y = 0; y < target_mask.rows; y++){
    for (size_t x = 0; x < target_mask.cols; x++){
        //If pixel is classified as obstacle, then get its depth.
        if (target_mask.at<uchar>(Point(x, y)) == 0){
            //Calculate depth map.
            depth_map.getValue(x, y, &depth_value);
            //Proximity Interval: 1m.
            if (depth_value <= 1000)
            pixel_count_1mtr++;
            //Proximity Interval: 1m - 2m.
            else if (depth_value > 1000 && depth_value <= 2000)
            pixel_count_2mtr++;
            //Proximity Interval: 2m - 3m.
            else if (depth_value > 2000 && depth_value <= 3000)
            pixel_count_3mtr++;
            //Proximity Interval: 3m - 4m.
            else if (depth_value > 3000 && depth_value <= 4000)
            pixel_count_4mtr++;
            //Proximity Interval: 4m - 5m.
            else if (depth_value > 4000 && depth_value <= 5000)
            pixel_count_5mtr++;
        }
    }
}
```

Send Proximity Alert

This section is in charge of communicating a proximity alert message if an obstacle has been detected in the nearby vicinity and in what interval of distance. To this purpose, another counter (*alert_Xmtr*) is used to keep track of the number of obstacles detected at each proximity interval per frame. The counter will consider an obstacle, if the amount of pixels for a given proximity interval is more than a fixed threshold (*threshold_tobe_object*) defined by the user. If after "N" frames, the obstacle has been detected "N" times at a certain distance, then it is considered as a potential collision threat and the alarm for the respective distance will be communicated. The reason for using a threshold to consider a group of pixels as a potential threat is that it reduces false positive results, making the solution more robust to noise. Similarly, the alarm evaluation being done every "N" frames helps in having more data to validate an alarm decision. The downside is that it increases the number of frames needed for a proximity alarm decision, which translates to a lower alarm throughput, given by **Equation (6)**. So, if the response time needs to be improved, this frequency could be increased by optimizing the application further to increase FPS or by limiting the number of evaluation frames.

$$Frequency\_Proximity\_Alarm = \frac{FPS(application\_throughput)}{N\_evaluation\_frames} \tag{6}$$

Listing 16: Send Proximity Alert

```
if (n_iter == n_frames) {
    if (alert_1mtr >= n_frames-1){
        cout << "Proximity Alert 1mtr" << endl;
        }
    if (alert_3mtr >= n_frames-1){
        cout << "Proximity Alert 2mtr" << endl;
        }
    if (alert_5mtr >= n_frames-1){
        cout << "Proximity Alert 3mtr" << endl;
        }
    if (alert_10mtr >= n_frames-1){
        cout << "Proximity Alert 4mtr" << endl;
        }
    if (alert_20mtr >= n_frames-1){
        cout << "Proximity Alert 5mtr" << endl;
        }
    n_iter = 0;}
else {
    if (pixel_count_1mtr >= threshold_tobe_object) alert_1mtr++;
    if (pixel_count_3mtr >= threshold_tobe_object) alert_3mtr++;
    if (pixel_count_5mtr >= threshold_tobe_object) alert_5mtr++;
    if (pixel_count_10mtr >= threshold_tobe_object) alert_10mtr++;
    if (pixel_count_20mtr >= threshold_tobe_object) alert_20mtr++;
    n_iter++;
}
```

3.4 Obstacle Detection and Proximity Alert Application - Deployment

As required, the application must run in real-time as part of the FishOtter USV system. To accomplish this, a series of steps must be first performed to integrate different components of the application. The next sections will further detail important aspects to consider.

3.4.1 Neural Network Model Conversion

In order to conclude the Deep Learning related activities in *Google Colab*, a python script is developed to handle the conversion from TensorFlow to ONNX extension. "Open Neural Network Exchange" (ONNX [14]), is key to integrating the solution in C++. It is an open source ecosystem that allows you to represent machine learning models with a set of standard operators across multiple frameworks, as well as access hardware optimizations to maximize performance. For example, ONNX requires CUDA to run its optimization on the Jetson GPU. To this end, it is important to consider that the complete model representation has to be loaded as a ".h5" extension, which has been previously explained.

Listing 17: Neural Network Model Conversion

```
#Read Keras model from path.
model = tf.keras.models.load_model('PATH/otternet.h5',
        custom_objects={'f1_score_loss':f1_score_loss,
                        'f1_score_coef':f1_score_coef})

#Define input tensor type and name.
format = (tf.TensorSpec((None, 256, 256, 3), tf.float32,
            name="input_image"),)

#Convert Keras model to ONNX. Optset depends on Jetpack version,
#this case it has to be 12 because 13 is not supported.
onnx_model, _ = tf2onnx.convert.from_keras(model,
                input_signature = format,
                opset=12)

#Define path to save ONNX model.
onnx_path = "PATH/otternet.onnx"
#Save ONNX model.
onnx.save(onnx_model, onnx_path)
```

To verify that the model has been properly converted and that it is running correctly in ONNX format with the same accuracy, it is recommended to do a small verification activity which is also included in the Python script.

3.4.2 Post-Training Quantization

Post-training quantization of the trained neural network is also evaluated to assess the tradeoff between inference speed and prediction accuracy. This is accomplished by reducing arithmetic precision to 8-bit integer rather than full precision of 32-bit floating point.

Listing 18: Post-Training Quantization

```
import onnx
from onnxruntime.quantization import quantize_dynamic, QuantType
#Define path of original ONNX 32-bit fp model.
model_fp32 = "PATH/otternet.onnx"
#Define quantized model saving path.
model_quant = "PATH/otternet.quant.onnx"

#Model Quantization to INT8.
quantized_model = quantize_dynamic(model_fp32, model_quant,
                  weight_type=QuantType.QUInt8)
```

To verify that the model has been properly converted and that it is running correctly in ONNX format, it is recommended to do a small verification activity which is also included in the Python script.

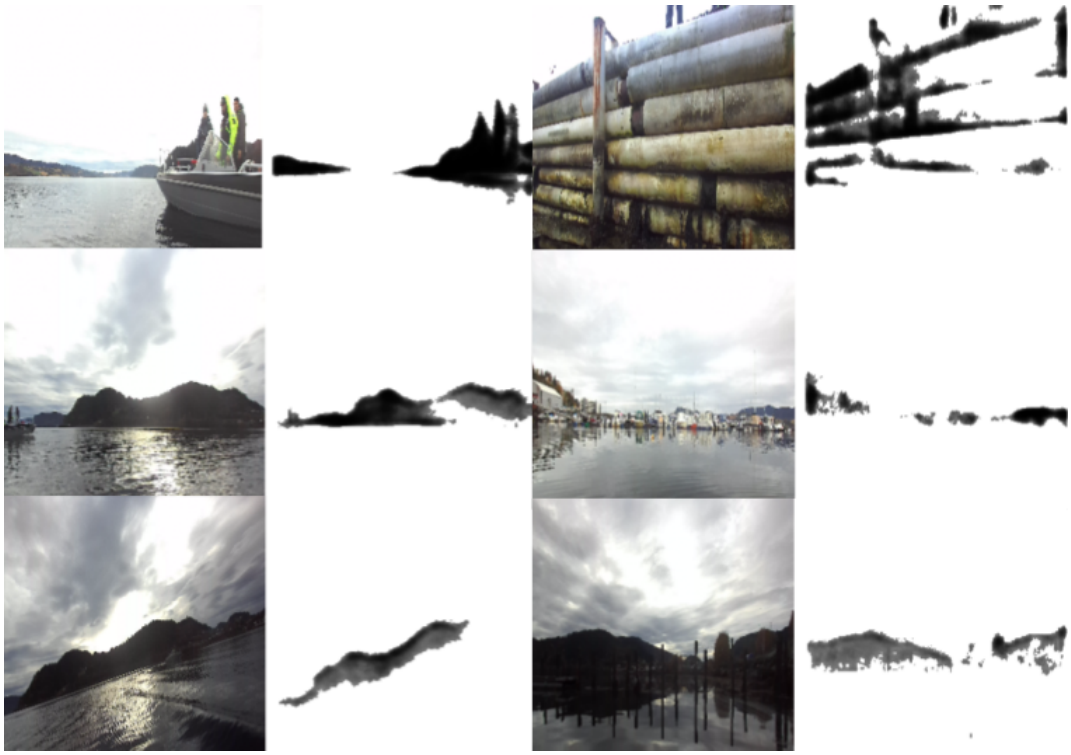

Figure 39: Example of predictions with quantized U-Net model.

3.4.3 Dual Platform Development

The C++ application was developed using *Windows Visual Studio 2022*, which makes it easy to import and include libraries through the user interface. However the Edge devices run a Linux based operating system called Jetpack from NVIDIA and due to its resource constraint nature, these devices pack a different computing architecture which in some cases requires building the libraries, such as ONNX, from source.

The development environment selected for this project is *Windows Visual Studio 2022*, due to past experience with it and for being a rapid development IDE platform. The required libraries were included into the platform following the steps shown below.

OpenCV and Onnxruntime:

- **go to:** *Project → Manage NuGet*
- **search and install:** *Microsoft.ML.OnnxRuntime and opencv4.2*

Stereolabs:

- **go to:** *Property Pages → C/C++ → General → Additional Include Libraries*
- **add:** *ZED SDK include; NVIDIA GPU Computing Toolkit CUDA vXX.X include*
- **go to:** *Property Pages → Linker → General → Additional Libraries Directory*
- **add:** *ZED SDK lib; NVIDIA GPU Computing Toolkit CUDA vXX.X libx64*

To deploy the application into the Jetson devices, CMAKE is used to build the application in order to have a compact solution that can make it easier to search, include and link the required libraries and compile the program for future developments. The entire folder required to build the application, which includes the CMakeList along with a cmake configuration for onnxruntime based on [13] is part of the GitHub repository of this project.

3.4.4 Configuration and Deployment on Jetson Xavier-NX and Nano

The following configurations are done in headless mode as well as connected to a monitor. *Putty* and *FileZilla* are used for remote connection and friendly SFTP user interface between development PC (running Windows) and both edge devices.

Install Jetpack on Jetson
In case the Jetson model has no pre-installed Jetpack version, install the one that corresponds as stated in [3]. Otherwise, if the Jetson model has a preinstalled Jetpack version, check which one by executing the following command:

Listing 19: Install Jetpack on Jetson

```
sudo apt-cache show nvidia-jetpack
```

The Jetpack versions used for this project are:

- Jetson Nano: 4.5
- Jetson Xavier-NX: 4.6

Install the ZED SDK on Jetson
Follow [4] to install the ZED SDK on the NVIDIA Jetson device.

Install the ONNX Runtime Repository
Depending on the Jetpack version installed, different versions of CUDA, cuDNN and TensorRT will be also installed and available. It is not recommended to uninstall and install any of these modules independently. The procedure will vary depending on the Jetson module and Jetpack version. According to the version of these modules, follow the requirement chart in [19] and [1] to make sure there is compatibility between versions. Following the Jetpack versions used for this project, execute the next commands to install onnxruntime v1.6.0:

Jetson Nano: 4.5

Listing 20: Install the ONNX Runtime Repository on Jetson Nano

```
git clone --single-branch --recursive --branch rel-1.6.0
https://github.com/Microsoft/onnxruntime
```

Jetson Xavier-NX: 4.6

Listing 21: Install the ONNX Runtime Repository on Jetson Xavier-NX

```
git clone --single-branch --recursive --branch rel-1.6.0
https://github.com/Microsoft/onnxruntime
cd onnxruntime
git submodule update --init
```

```
cd cmake/external/onnx-tensorrt
git remote update
git checkout 8.0-GA
```

Specify the CUDA compiler, or add its location to the PATH.
CMAKE can't automatically find the correct nvcc if it's not in the PATH, so assign it using the following command.

Listing 22: Export CUDA path

```
export CUDACXX="/usr/local/cuda/bin/nvcc"
```

Install the ONNX Runtime build dependencies
Install the required dependencies with the following command:

Listing 23: Install the ONNX Runtime build dependencies

```
sudo apt install -y --no-install-recommends build-essential
software-properties-common libopenblas-dev libpython3.6-dev
python3-pip python3-dev python3-setuptools python3-wheel
```

Install Cmake building from source
Cmake is needed to build ONNX Runtime. Because the minimum required version is 3.18, it is necessary to build it from source. Execute the next commands based on [6].

Listing 24: Install Cmake building from source

```
wget https://cmake.org/files/v3.23/cmake-3.23.0.tar.gz
tar xf cmake-3.23.0.tar.gz
cd cmake-3.23.0
sudo apt-get install libssl-dev
./bootstrap
make
sudo make install
cmake --version
```

If after cmake –version, cmake is still not found, add the installed cmake bin directory path in .bashrc:

Listing 25: Export cmake path

```
export PATH=/home/user/cmake-3.23.0/bin
```

Build the ONNX Runtime Python wheel with TensorRT support
Change directory to where the build.sh file is located and run the following command.

Listing 26: Build the ONNX Runtime Python wheel with TensorRT support

```
./build.sh --config Release --update
--build --build_wheel --skip_submodule_sync
--build_shared_lib --use_tensorrt
--cuda_home /usr/local/cuda
--cudnn_home /usr/lib/aarch64-linux-gnu
--tensorrt_home /usr/lib/aarch64-linux-gnu
```

Copy the required library and include files
Cmake looks for the libraries and includes files on predefined paths. In order to run properly the next cmake file and CMakeList, copy the "Release" directory located inside */build/Linux* into */home/<username>/.local/lib* and copy the "include" directory into */home/<USERNAME>/.local/*.

Include the cmake file needed to locate Onnxruntime library
Create */cmake/onnxruntime* directories inside */home/<username>/.local/* and copy onnxruntime -config.cmake that can be found in the GitHub repository into */home/ <username>/.local/share/cmake/onnxruntime*.

Transfer the Application into Jetson
Copy the Application folder into a directory. The folder includes: i) CMakeList.txt needed to build the application, ii) Deep Learning Model directory from which the application will read the ONNX model, iii) source directory with the main.cpp program, and iv) include directory with the helpers.cpp, helpers.h and utils.h files.

Build Application
Create the build directory and build the application as follows.

Listing 27: Build Application

```
mkdir build && cd build
cmake ..
```

Compile Application
When the build is complete, go to the build directory and compile.

Listing 28: Compile Application

```
make
```

4 Experiments, Results and Discussion

After successfully implementing the obstacle detection and proximity alert application on both edge devices, several videos recorded from the field trip are used to assess the performance. Having limited computing resources at the edge makes it imperative for the developer to be thoroughly aware of the following metrics: i) latency, ii) power consumption and iii) memory consumption. To this end, a total of 40 recordings are used on the experiments. The corresponding sections present the results for both edge devices: Jetson Nano and Xavier-NX. The original C++ application is slightly modified to take recordings as input instead of direct feed from the camera. This modified code can also be found at the GitHub repository of this project.

On the other hand, the i) obstacle detection and ii) depth estimation accuracy are tested on individual experiments and independently from the edge devices. The reason is that no modification (e.g. quantization or pruning) is done to the model and that the computer architecture will not change the prediction results, so it is assumed that the model will perform similarly as in the development environment. In the same way, the resulting depth map estimates should be the same independent of the device, considering that the Stereolab-SDK is properly used. Nevertheless, the results are also visually inspected by comparing the output verification video from the edge devices to the output verification videos obtained from the development environment, showing no perceivable difference. Some of these verification videos are attached to the GitHub repository as complementary material.

4.1 Obstacle Detection Accuracy

The model is first evaluated with 132 images, corresponding to 10% of the MaSTr1325 dataset destined for this purpose. The metric used to evaluate the classification accuracy is f1-score, due to the imbalance nature of the dataset and for being a commonly used image segmentation metric that considers recall and precision in one single expression as seen in **Equation (7)**. The evaluation resulted in a f1-score of 0.93, with 1.0 representing a perfect match. Considering the small dataset used for training, it is safe to say that applying transfer learning by using MobileNetv2 as the backbone, in addition to having a well suited encoder-decoder architecture and a simplified binary target, is beneficial.

$$f1 - score = \frac{2 * Precision * Recall}{Precision + Recall} = \frac{2 * TP}{2 * TP + FP + FN} \tag{7}$$

Nevertheless, it is important to consider that the train and test dataset is not balanced due to the limited amount of images, which could lead to a lower score. In addition, different factors unseen in the train and test dataset, such as lighting conditions and completely new obstacles may hinder the overall performance.

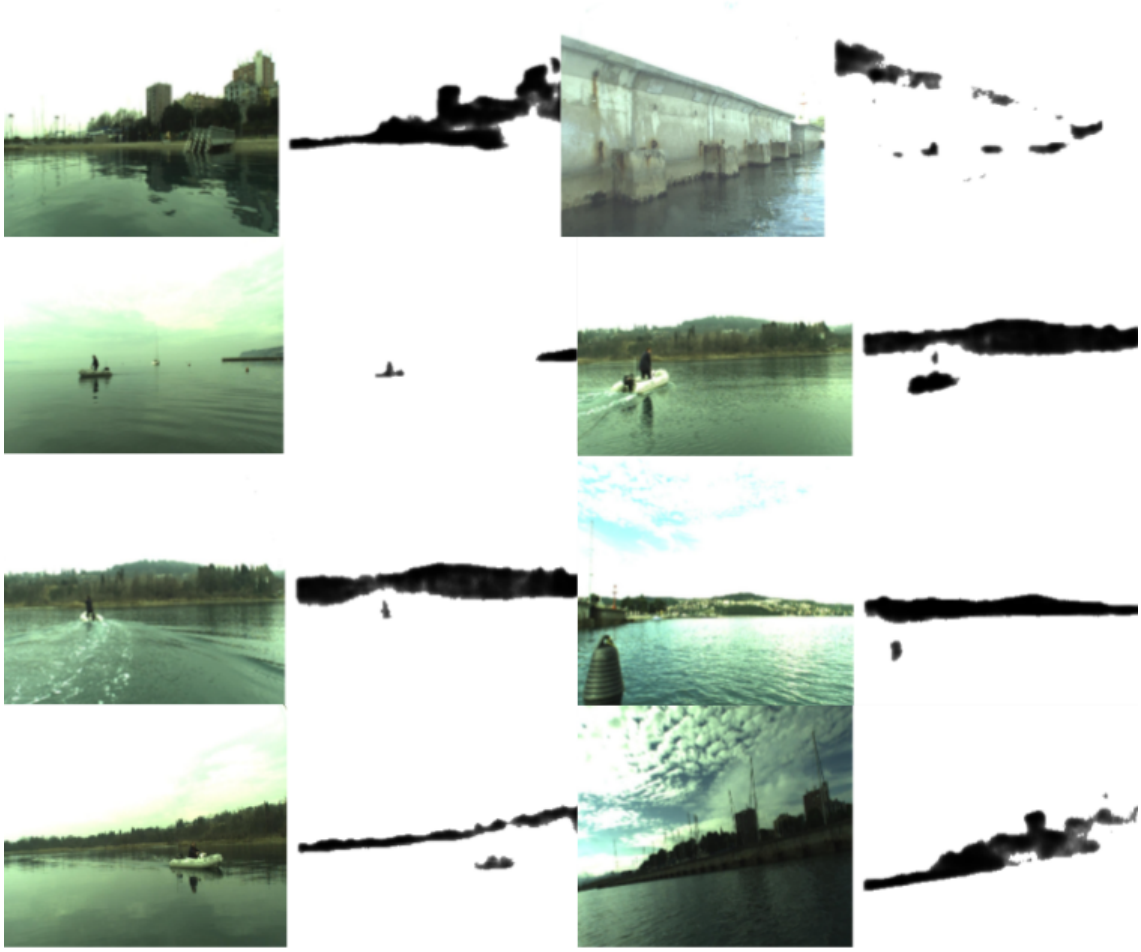

Figure 40: Examples of U-Net predictions using the test images from MaSTr1325 public dataset.

To better assess the quality of the model, a total of 100 images from the Otter Dataset are used. As expected, the results show a lower f1-score of 0.83. Nevertheless, it is still considered a very good first model to develop a first version of an obstacle detection system considering the model was trained with only the MaSTr1325 public dataset and has never seen the Otter Dataset before. Moreover, even though it misses the classification of some obstacle class pixels (False Negatives), it does not classify non obstacle pixels as obstacles (False Positives), which is highly beneficial for this application. By recognizing some of the obstacle pixels, it is still possible to calculate the depth of the obstacle and alert the system correctly. On the other hand, missed classified obstacle pixels will generate false positives alerts to the system.

Some samples of the new test dataset are shown below, it can be seen that the lighting conditions are very different from the ones at the MaSTr1325 public dataset used for training and validation. In addition, it is very clear how the model has been able to almost perfectly never miss classify a pixel as an obstacle outside the real object's boundary (very low False Positives).

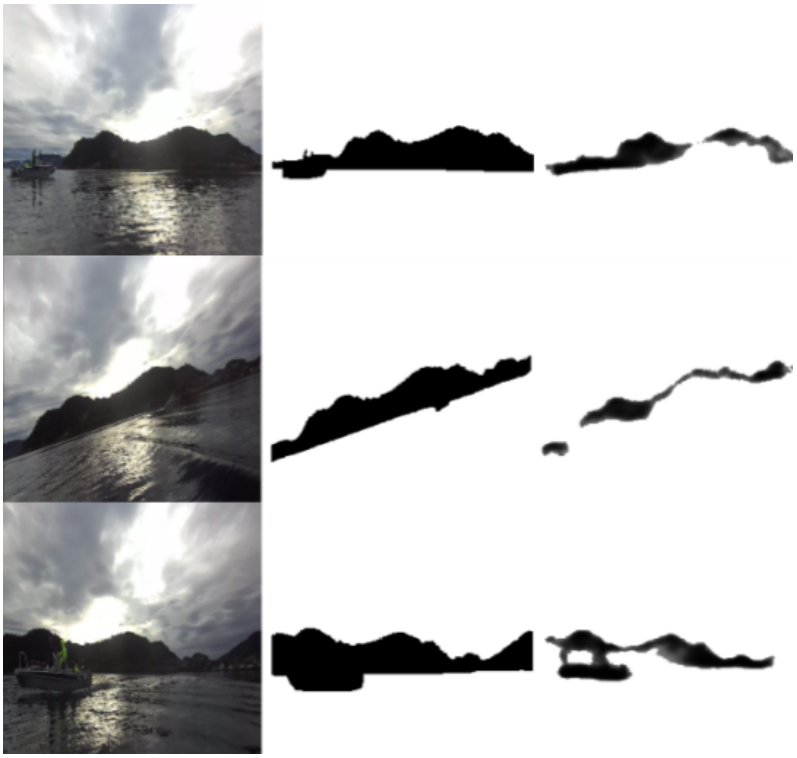

Figure 41: Examples of U-Net predictions using the Otter Dataset.

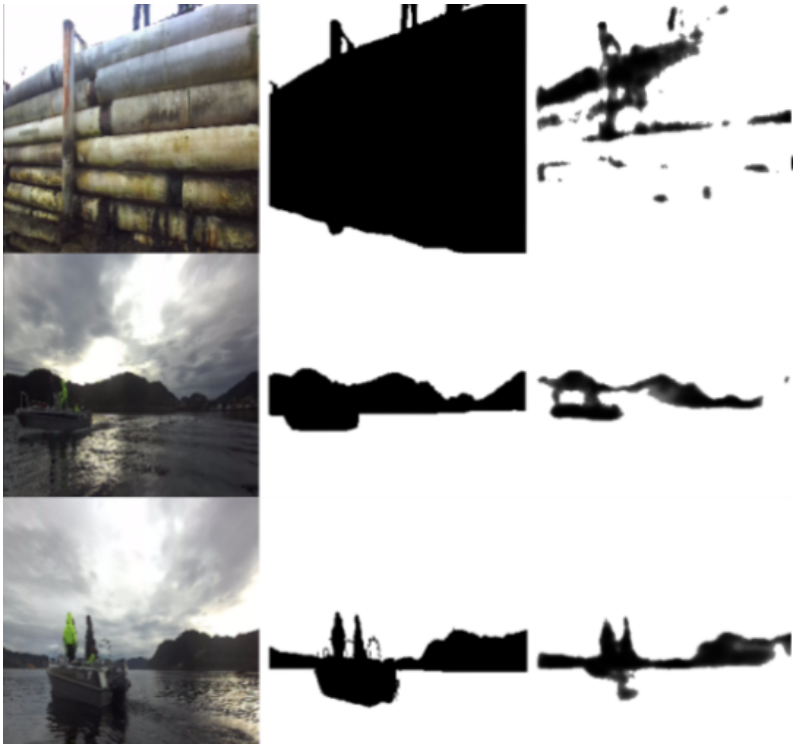

Figure 42: Examples of U-Net predictions using the Otter Dataset.

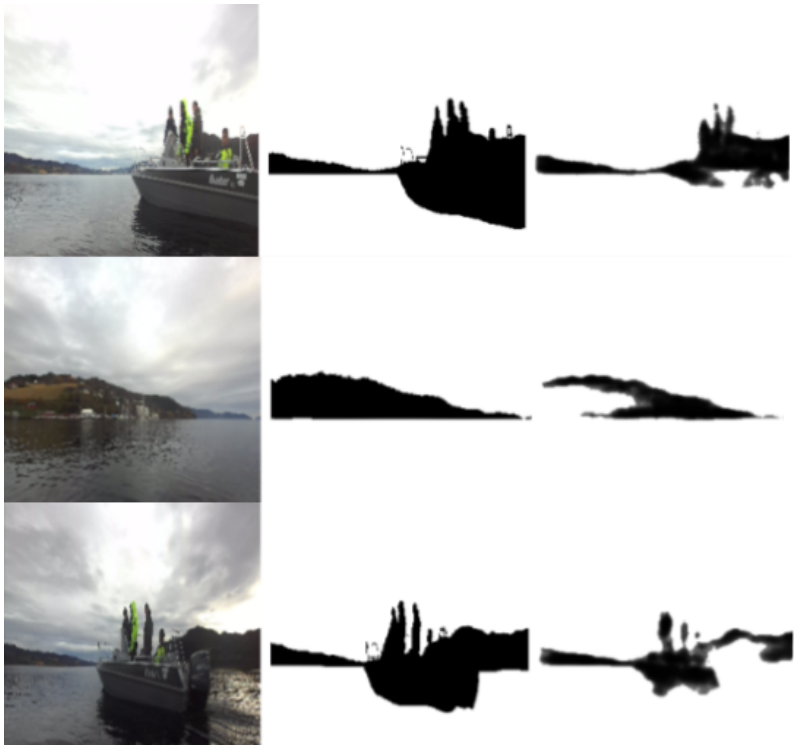

Figure 43: Examples of U-Net predictions using the Otter Dataset.

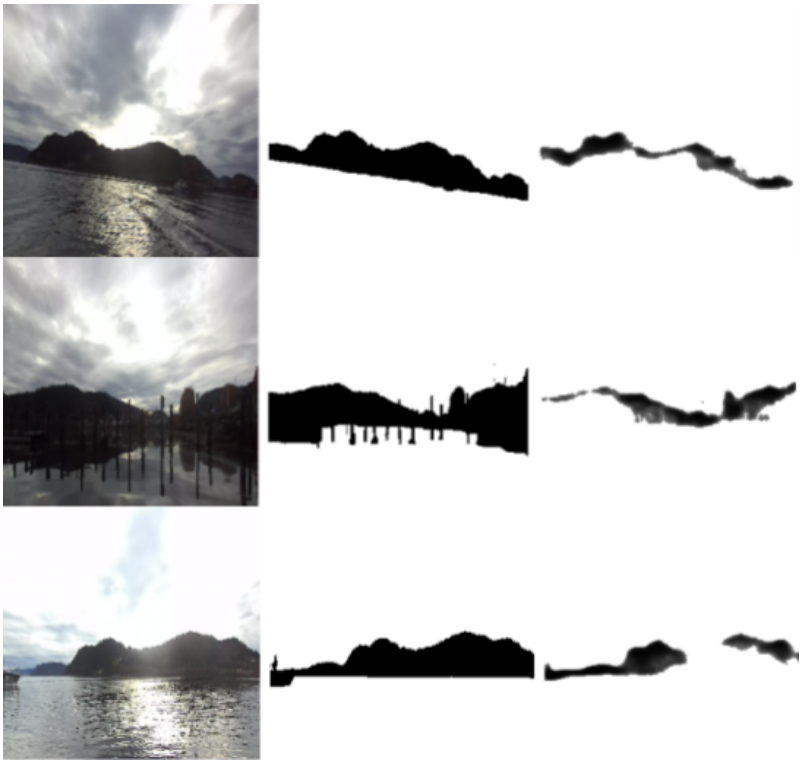

Figure 44: Examples of U-Net predictions using the Otter Dataset.

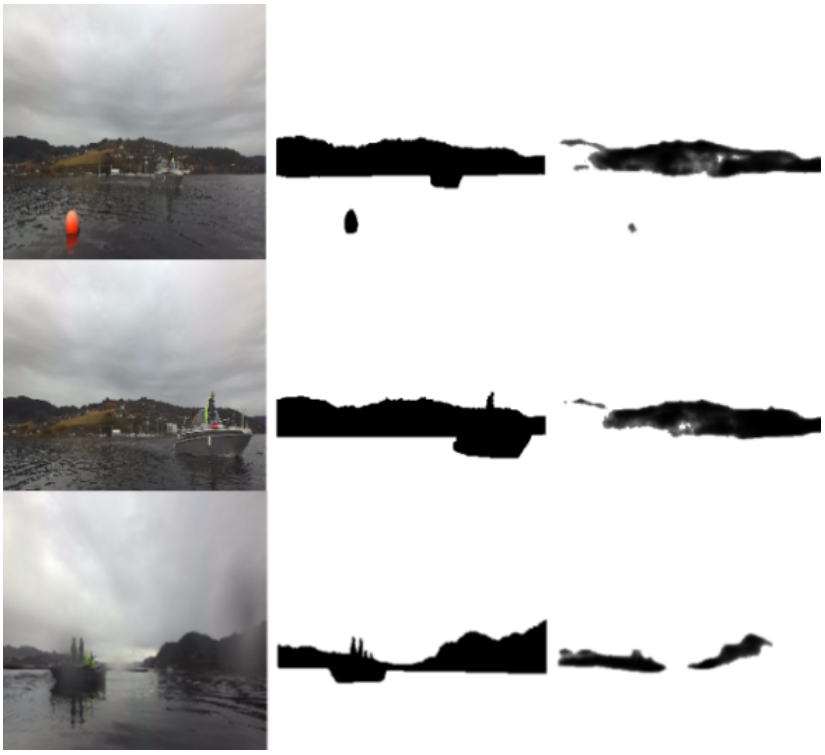

Figure 45: Examples of U-Net predictions using the Otter Dataset.

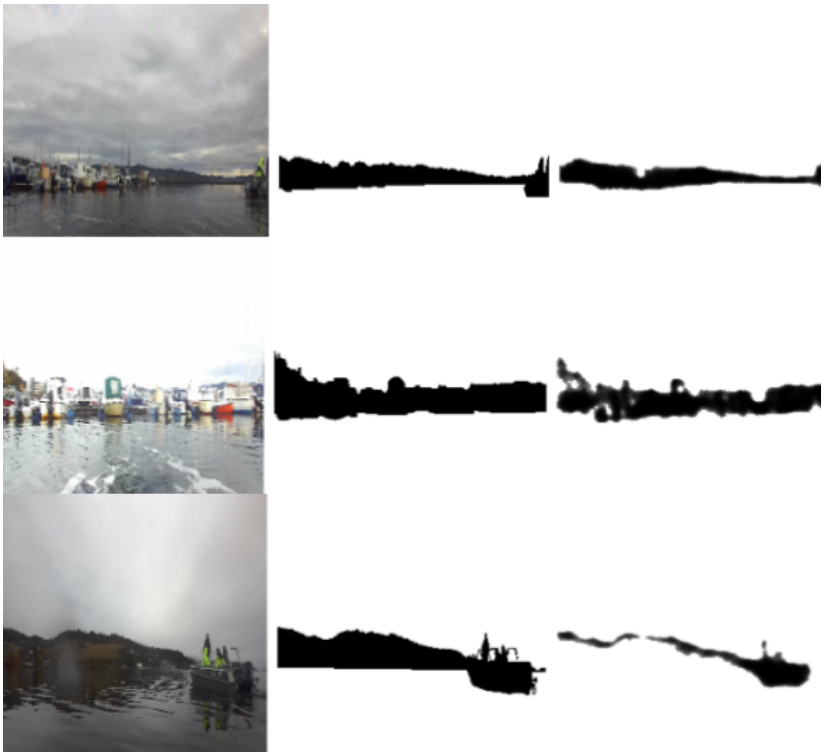

Figure 46: Examples of U-Net predictions using the Otter Dataset.

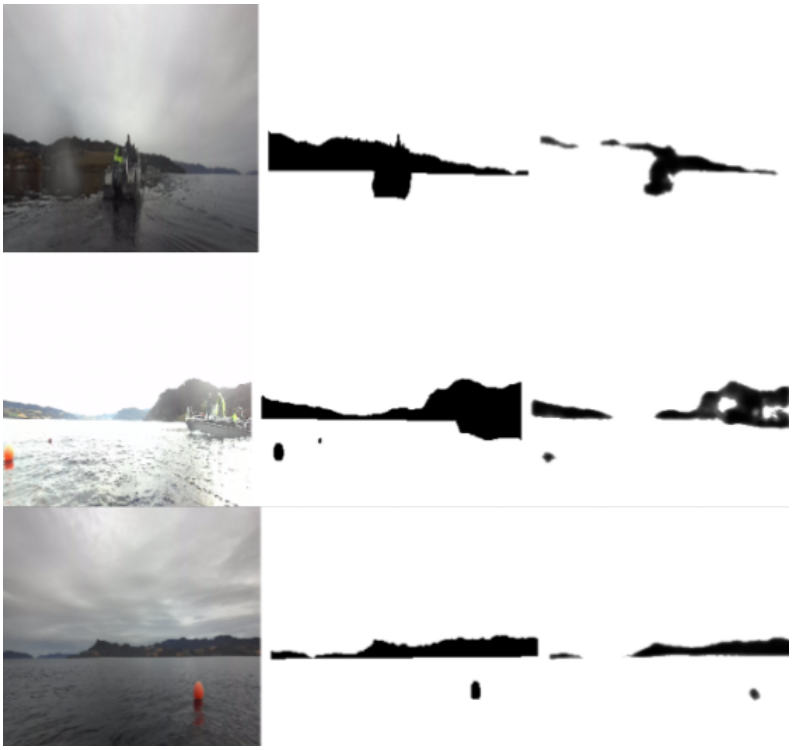

Figure 47: Examples of U-Net predictions using the Otter Dataset.

4.2 Depth Estimation Accuracy

Although depth estimation is part of the Stereolabs-SDK, it is still important to assess the accuracy of its measurements. According to the library provider, it is possible to detect distances in the range of 0.4 to 20 meters. Ideally, a system that can provide a precise ground truth is needed to compare the stereo depth estimates, such as a LIDAR. But due to the fact that it is not possible to acquire one at this stage of the project, a more hand-made approach is considered. In addition, because the application classifies the distance of the obstacles into 5 proximity intervals, a margin of error in accuracy can be tolerated.

To this end, the experimental setup included different obstacles placed at different distances from 1 to 15 meters. For the ground truth, a measuring tape is used to measure the distance between the camera and the objects on a straight line from object to camera. A total of 7 setups are recorded and analyzed, each setup includes 8 objects in random locations chosen to cover the entire distance range.

The camera records 10 frames (left and right) from which the depth estimation library can calculate 10 depth values for the obstacle pixel. To obtain the depth of the test objects, it is necessary to identify one of its pixel coordinates in the left or right frame image. To this end, some special markers are used to easily identify them in the image. *MATLAB* "Image Viewer" application is used to visualize and inspect one of the frames to retrieve the XY coordinate value. With these XY pixel values, the library can obtain its specific depth value with the following code, which is also part of the GitHub repository.

Listing 29: Stereo-depth Extraction

```
while (i <= 10){
    returned_state = zed.grab();
    if (returned_state == ERROR_CODE::SUCCESS){
    //Get image from svo recording.
    zed.retrieveImage(svo_image, VIEW::LEFT,
    MEM::CPU,low_resolution);
    //Get depth map
    zed.retrieveMeasure(depth_map, MEASURE::DEPTH);
    //Calculate Depth of obstacle pixel
    int x = 834;
    int y = 694;
    depth_map.getValue(x, y, &depth_value);
    printf("Depth to Camera at (%d ,%d): %f mm\n",
    x, y, depth_value);
    imwrite("PATH/Frame_test_.jpg", svo_image_ocv);
    i++;
}
```

Figure [48] shows an example of the results for one of the tests. It presents the inspected XY pixel coordinates where the object is located in the image, the stereo depth estimation, and the ground truth measurements in meters. The stereo estimate results for each object is an average of the 10 depth values given per frame. It can be seen that the difference between the ground-truth and the estimated values is small for our application purposes even at distances close to 15 meters. In addition, the standard deviation indicates that the stereo depth estimates are usually similar for the same pixel over the 10 runs, the differences may be caused by environmental conditions such as lighting and/or in the internal depth estimation process done by the SDK library.

	Test 1				
	Pixel XY Coordinates	Mean Stereo Estimate (m)	STD Stereo Estimate	Ground Truth (m)	Relative Error (%)
Object 1	(2033, 701)	0,95	4,66%	1	5,66%
Object 2	(561, 1075)	1,20	3,82%	1,15	3,82%
Object 3	(285, 422)	2,07	1,89%	2,1	1,45%
Object 4	(1822, 826)	4,20	2,92%	4,3	2,38%
Object 5	(899, 640)	8,21	3,15%	8,4	2,31%
Object 6	(1140, 772)	10,30	3,42%	10,8	4,85%
Object 7	(414, 795)	11,38	3,25%	11,9	4,57%
Object 8	(322, 426)	14,23	3,52%	14,8	4,01%

Figure 48: Shows the results from Test 1 showing mean stereo depth estimates vs ground truth measurements in meters.

Figure [49] shows a summary of the seven setups and recordings results, where the relative error between ground-truth and stereo estimates for all the different obstacles in the setups are averaged. The average standard deviation is calculated in the same way. It can be seen that the relative error is no larger than 5%, which is suitable for this application that works on proximity distance intervals of 1 meter. It is important to consider that using the measuring tape also induces human error. However these tests are meant to prove that the Stereolab camera system provides coherent depth estimates, for a more accurate comparison the stereo-vision 3D point cloud should be compared against a point cloud ground truth such as the one provided by a LIDAR.

	Test 1	Test 2	Test 3	Test 4	Test 5	Test 6	Test 7
Average Relative Error (%)	3,71%	3,57%	3,72%	4,31%	3,76%	4,28%	4,62%
Average STD	4,40%	3,21%	5,11%	4,10%	4,21%	3,84%	5,22%

Figure 49: Shows the summary results from all seven tests showing average relative error (%) and average STD (%).

4.3 Latency Analysis

To better understand the behavior of the application as one way of looking up for optimization potential, it is necessary to calculate how much time it takes for each important section of the code to execute. To do this, the code is divided into 5 sections: i) Init variables, ii) Get image, iii) Get depth map, iv) Model inference, v) Get object proximity. The library "chrono" is used to calculate the time between each operation by inserting "starts" and "stops" timestamps to calculate the duration as the difference between them. The name of these 5 sections can be recognized inside the code at the comment section to make it clear where these timestamps are positioned.

- **Init variables:** Section of the code where all the required parameters, variables and constants are initialized. Especially the ones required by onnxruntime and Stereolabs-SDK for inference and stereo-camera data manipulation respectively.
- **Get image:** Section of the code where a frame is fetched from the stereo-camera and passed for processing (image resizing, normalization, dimension manipulation and image to tensor conversion) which is explained in detail in the previous chapter.
- **Get depth map:** Section of the code where the depth of each pixel is calculated from the stereo images. This is a call to a Stereolabs-SDK library function.
- **Model inference:** Section of the code where the tensor representation of the image is loaded into the onnxruntime session to start inference.
- **Get object proximity:** Section of the code that receives the target mask and classifies each object pixel into 5 intervals of distance proximity based on its individual pixel depth obtained from the depth map.

Environment	Dev (RTX 3060)		Edge (Xavier NX)		Edge (Nano)	
Metrics (milliseconds)	mean	std	mean	std	mean	std
Init variables	1612,2	5,1%	2569,7	8,5%	5824,6	5824,6
Get image	101,7	5,1%	56,6	2,0%	80,7	80,7
Get depth map	9,7	5,8%	10,2	2,6%	20,5	20,5
Model Inference	**21,5**	17,7%	**276,9**	2,4%	**607,8**	607,8
Get object proximity	**28,5**	6,9%	**181,2**	1,2%	**321,7**	321,7
Overall FPS	***6,1***		***1,9***		***1,0***	

Figure 50: Shows the execution time in milliseconds for each of the inspected sections, and the overall FPS for each deployment.

The Development environment is a Laptop (Sword 15 A11UE [17]) with Corei7 and a RTX3060 NVIDIA GPU . This GPU has around 7 times (12.74 TFLOPS) the processing speed of the Xavier-NX, so the execution speed is expected to be substantially higher as seen in **Figure** [**50**]. Although it is not fair to compare it with the edge devices, it serves as a reference if an upgrade to the Jetson system is being considered at some point in the future.

On the other hand, as expected and seen in **Figure** [**50**], the Xavier-NX performs better than the Nano with almost twice the speed. Showing that the application can process almost 2 frames per second (FPS). However it is still important to remember that the proximity alarm frequency depends on the value of the “N” evaluated frames, as shown in **Equation (6)**. Therefore, for the moment, it is recommended not to use a value of "N" greater than 2, to avoid violating the technical requirement of 1 FPS as response time.

Moreover, this analysis spotted potential optimization opportunities that can further reduce the time of execution. For instance, the potential sections that can be further improved and that are by far the most time consuming ones, are the inference and object proximity calculation. The first one would require modification of the neural network (reducing its size and number of parameters) or further optimizations on it (post-training quantization or pruning). Both approaches will potentially decrease the computation time but may hinder the accuracy of the model as a trade-off. The second one is directly related to the evaluation of each binary classified pixel to extract its class and depth if needed. Currently the code polls pixel by pixel from the entire output mask (same dimension as input image) through the use of two nested "for loops". This is a time consuming operation, so two possible approaches can be considered: i) reducing the camera frame input size, which will also reduce the target mask size, in order to reduce the total number of iteration in both nested "for loops" (NxM iterations) and, ii) using a more efficient algorithm to detect obstacles at the output mask, such as a clustering algorithm.

On the other hand, there is not much that can be done with respect to the initialization of variables due to its fixed and one time execution nature. In addition, although the tasks of fetching and processing an image, and getting the depth map are executed per frame, there is not much more that can be done to speed up the computation considering these are calls to third party libraries. However, as it was already mentioned in the previous chapter, it is worth to explore the possibility of executing the inference and depth map in parallel as they are independent from each other.

4.4 Power Consumption Analysis

It is especially important when developing applications for autonomous systems, that have limited amounts of energy available for several activities, to monitor power consumption during run-time in order to estimate the systems' lifetime and spot potential power reductions. In order to do so, NVIDIA counts with an SDK called "tegrastats utility", which helps profiling an application running on Tegra-based devices [18]. To profile the applications performance on both devices, the exact same 40 test videos are executed on the edge devices, while running the "tegrastats" command at a sample frequency of 10 Hz. This procedure is done individually for each of the videos. The command used is the following:

Listing 30: Execute Tegrastats

```
sudo tegrastats --interval 100 --logfile /PATH/log_X.txt
```

After executing all the test videos, the output log of the "tegrastats" command is analyzed using a python script developed for this purpose. The script searches for keywords related to power consumption and temperature information and calculates a basic statistical report along with charts, which will be presented next. This script can be used for automatically summarizing data gathered from the Jetson Xavier-NX and Jetson Nano just by inputting the respective "tegrastats" log file. The script is also part of the GitHub repository of the project.

Figure [51] shows the mean and standard deviation of the power consumption and temperature for the analyzed test videos. The power consumption can be further divided into gpu/cpu and general board consumption, which includes everything else. On the other hand, the temperature is also considered for analysis due to the fact that this application will be running non-stop. This is important to consider because the control box of the FishOtter has limited ventilation, and high operating temperatures can degrade the system performance and harm its lifespan.

As expected, the Xavier-NX shows higher power consumption than the Nano with a difference of approximately 1.5 Watts. Both edge devices show average temperatures below 40°C, which is non problematic. However, temperature should be monitored over long periods of execution at sea, using the "tegrastats" SDK along with the developed python script.

Edge Device	XAVIER		NANO	
	mean	std	mean	std
Average Power IN (mW)	5511,1	3,5%	4193,0	5,4%
Average Power GPU/CPU (mW)	1458,5	5,5%	1050,5	9,4%
Average CPU Temperature (°C)	28,4	1,4%	38,4	1,5%
Average GPU Temperature (°C)	29,6	2,3%	35,5	2,1%

Figure 51: Power consumption and operating temperature for both edge devices.

Figure [52] and **Figure** [53] show the average power (mW) for each time sample during the application execution for the Jetson Nano and Xavier-NX respectively, for only one of the recordings. The data is recorded from approximately 3 minutes before the start of the application execution, until 3 minutes after the end of the execution. This is done with the only purpose of plotting the full power cycle before, during and after execution. From the charts, it can be noticed that the Xavier-NX consumes around 3700 mW during IDLE while the Nano consumes around 1400 mW. In addition, the difference between maximum power consumption from both devices is about 1300 mW, similar to the difference between the average of the 40 videos. Finally, the power consumption values for both edge devices start to decrease slowly, not reaching IDLE values even after 3 minutes.

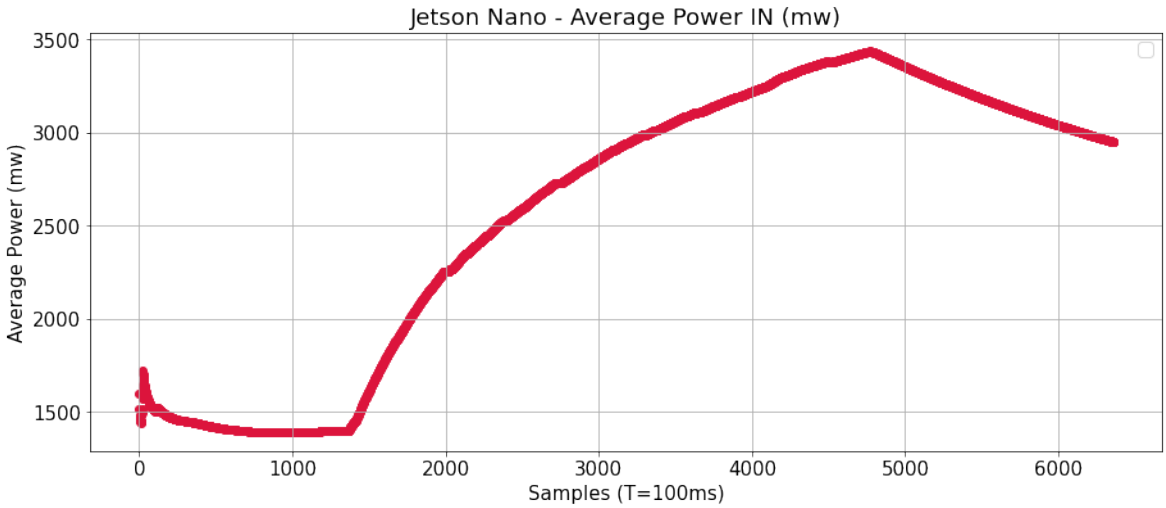

Figure 52: Jetson Nano average power consumption while executing the application.

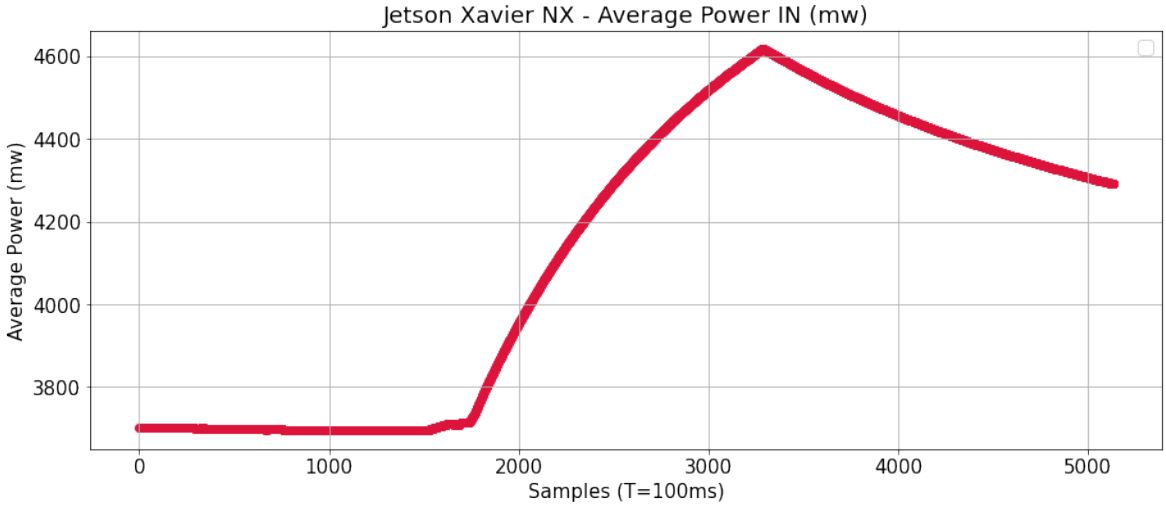

Figure 53: Jetson Xavier-NX average power consumption while executing the application.

4.5 Resource Utilization Analysis

Similar to the previous analysis, the "tegrastats" SDK is used to gather information about GPU and CPU utilization. **Figure** [54] presents the GPU resource utilization results for both edge devices, it can be seen that the Nano runs on a much higher frequency than the Xavier (around 60% higher). This could be explained by the fact that the Nano needs to speed up its computation (overclock) to make up for the limited amount of cores (NVIDIA Maxwell GPU with 128 CUDA cores) compared to the Xavier-NX (NVIDIA Volta architecture with 512 NVIDIA CUDA cores and 64 Tensor cores 22 TOPS (INT8)). This could also explain the higher average temperatures presented at the Nano, as seen in the previous section.

In addition, being the GPU a power hungry device, there are two power saving techniques which are apparently being used. The first one is trying to reduce time of GPU utilization by computing the load as fast as possible to return to idle. This can be seen on the "GPU no Load Ratio (%)", where both devices are only active around 15% of the time, while the rest of the time they have no load to run. The second one is the use of "Dynamic voltage frequency scaling (DVFS)", which also explains why the Nano, with fewer cores, has to run at higher frequencies in order avoid missing system deadlines.

Edge Device	XAVIER		NANO	
	mean	std	mean	std
Average GPU Utilization (%)	31,3	7,9%	46,9	9,4%
GPU no Load Ratio (%)	85,1	0,8%	89,4	1,3%
Average Frequency (Hz)	461,4	6,1%	745,6	10,1%

Figure 54: GPU resource utilization for Jetson Xavier NX and Nano

Figure [69] and **Figure** [56] show the distribution of the different frequencies and utilization percentage of the Jetson Nano GPU. The utilization of zero is not considered for plotting purposes because it is by far the most frequent. In addition, it presents the different scaling frequencies the Nano uses while operating, being 153 MHz and 921 MHz the most common ones. On the other hand, **Figure** [57] shows the multi-core utilization, which is important to acknowledge because most of the functionalities executed at the C++ application will run on CPU, such as: initialization of variables, fetching camera-frame, image processing, getting the object pixel proximity, and communicating the proximity alarm when required.

Similar charts are presented for the Jetson Xavier-NX in **Figure** [58], **Figure** [59], and **Figure** [60]. There, it is possible to see how the GPU utilization percentage is distributed on the lower values, while the maximum typical GPU frequencies only reach around 500 to 600 MHz.

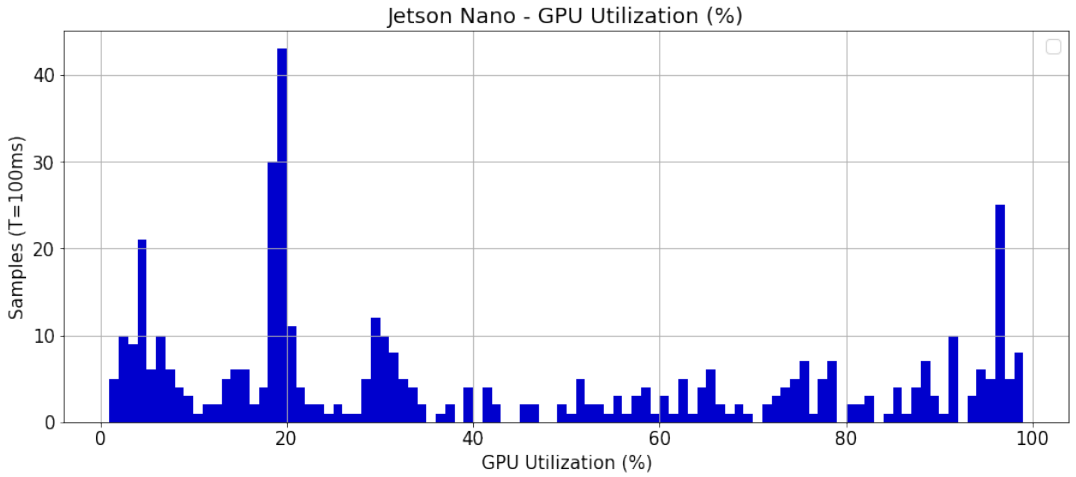

Figure 55: GPU utilization during program execution of Nano.

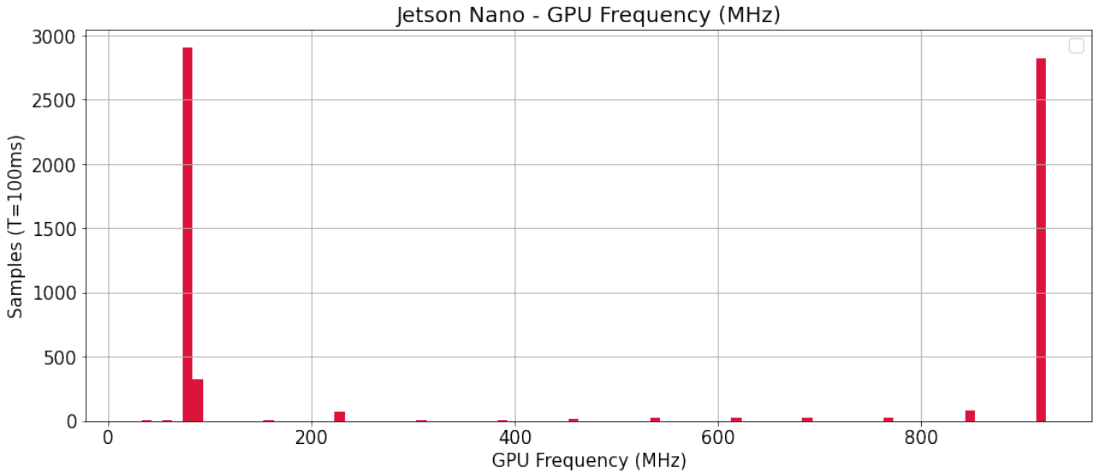

Figure 56: GPU frequency scaling during program execution of Nano.

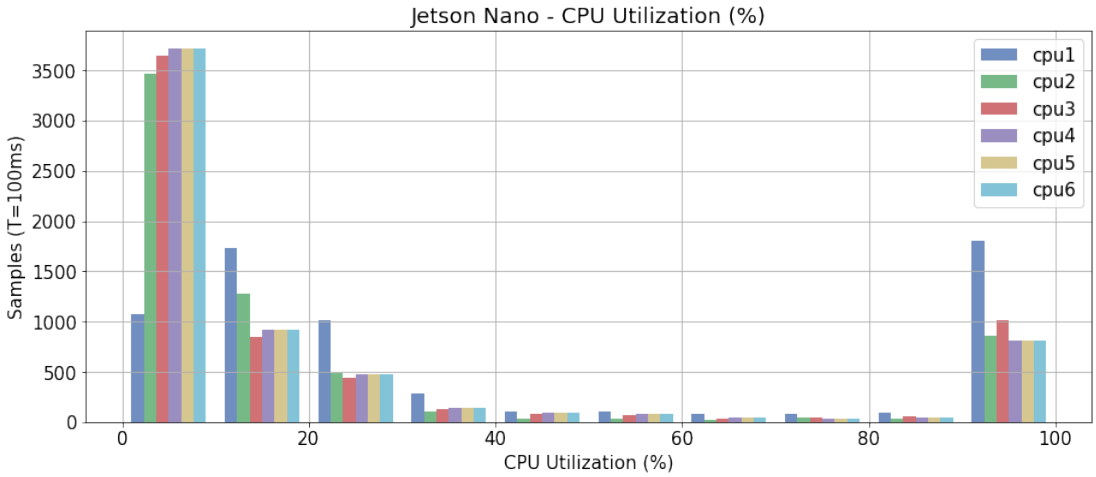

Figure 57: CPU utilization during program execution of Nano.

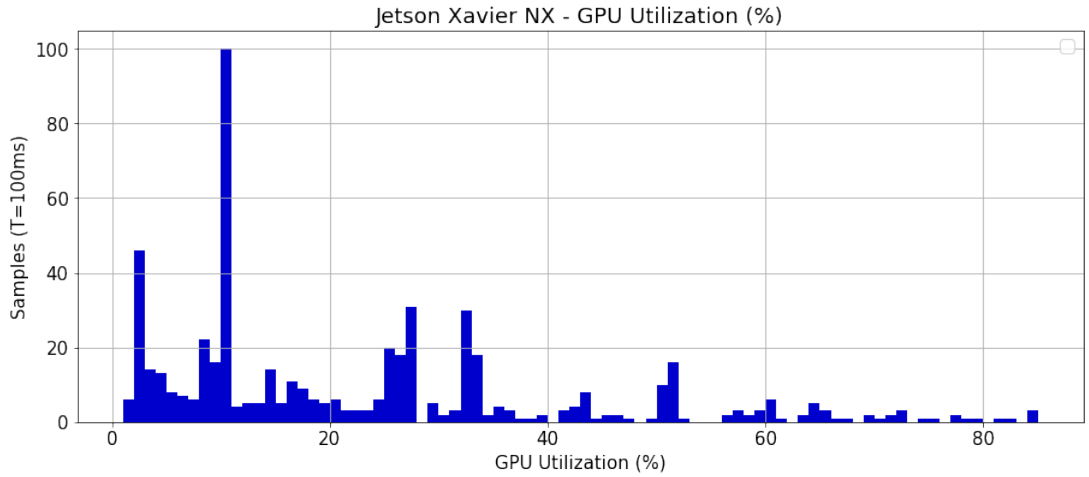

Figure 58: GPU utilization during program execution of Xavier-NX.

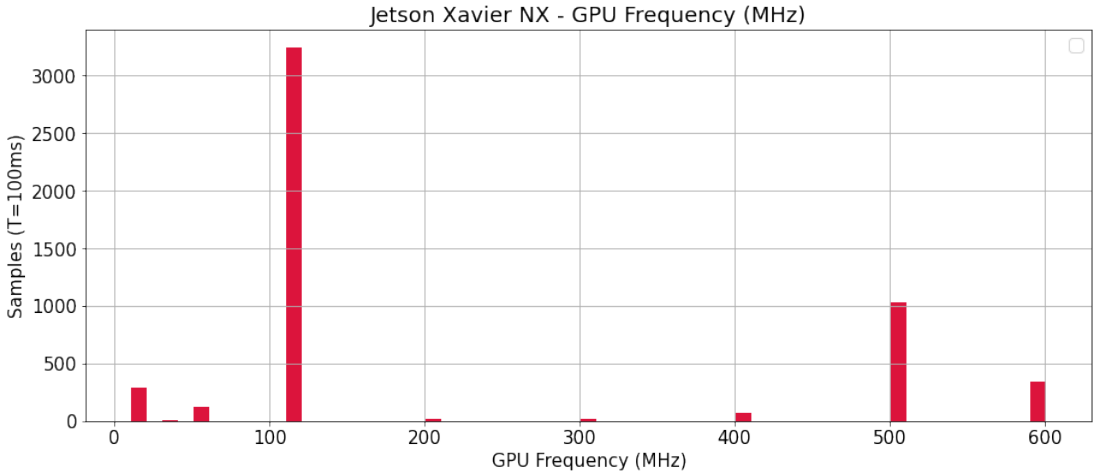

Figure 59: GPU frequency scaling during program execution of Xavier-NX.

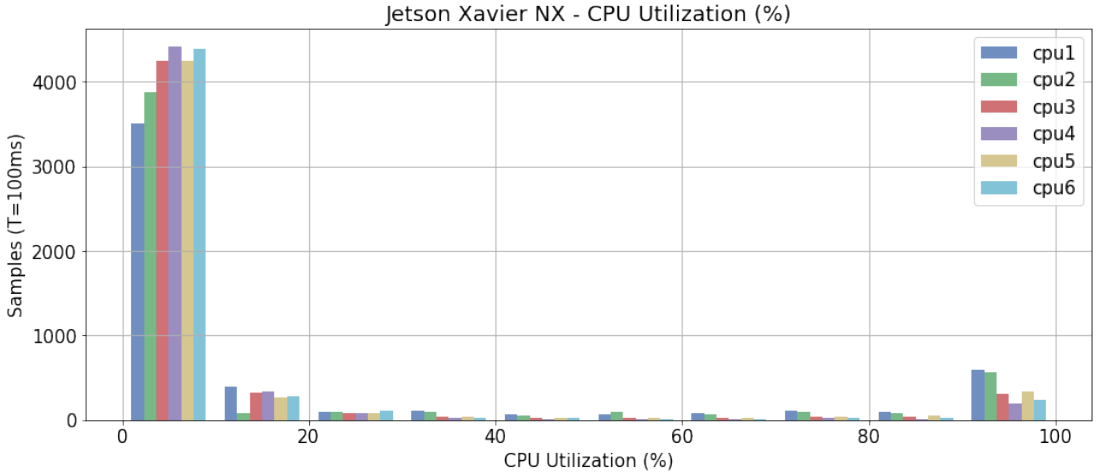

Figure 60: CPU utilization during program execution of Xavier-NX.

4.6 Field Trip - Examples

The following figures present some examples of real data after being processed by the obstacle detection and proximity distance application running on the Jetson Xavier-NX. The upper half of the figures is composed by the left frame from the stereo camera along with the OtterNet output mask prediction. On the bottom, the frame from the verification video generated by the complete C++ application. There, it is possible to observe the proximity alarm messages per frame. In addition, all the obstacles are colored with light gray tone and different gray-scales are used to represent each of the 5 proximity intervals, darker meaning closer. For visualization purposes, the proximity intervals are wider than in the real application.

It is interesting to see in **Figure**[63], **Figure** [64], **Figure**[66], and **Figure** [67] how even when the stereo-camera has some water drops on its lenses from the rain, the application is still capable of generating coherent outputs thanks to the robustness of the deep learning approach. These intervals can be easily modified in the code as required by the application.

The entire verification videos of which some frames are presented here, can be found in the link provided at the GitHub repository.

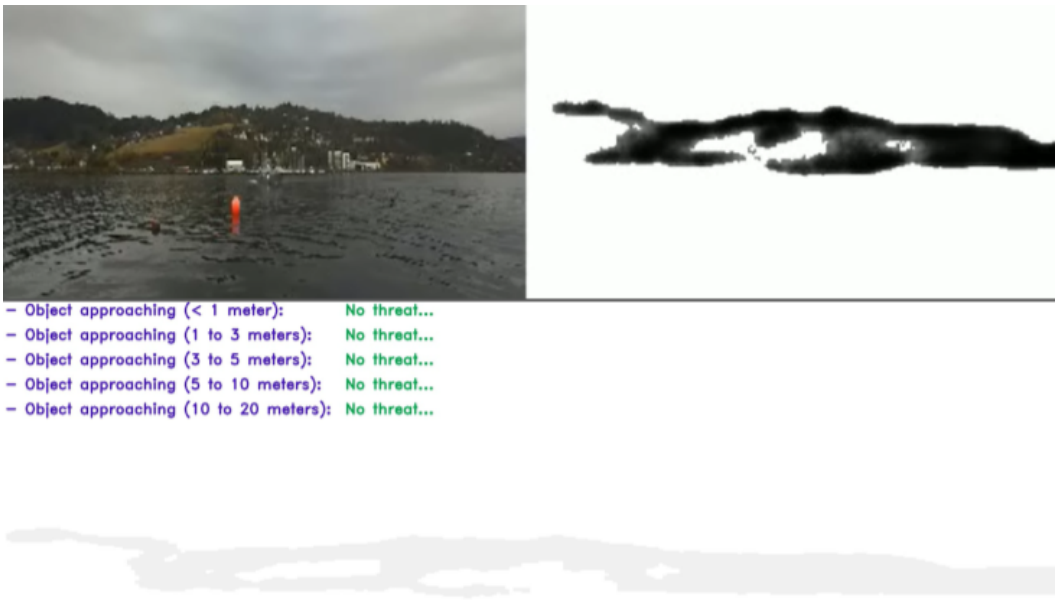

Figure 61: Example of a frame processed by the application running at the edge. Here, the coast is detected by the OtterNet, but it misses the buoy. However, it is able to recognize it in the following frames as it gets closer to it.

Figure 62: A buoy and a boat can be recognized. The buoy is between 1 to 3 meters from the FishOtter and part of the boat is around 5 to 10 meters while the rest is between 10 to 20 meters while it approaches the FishOtter.

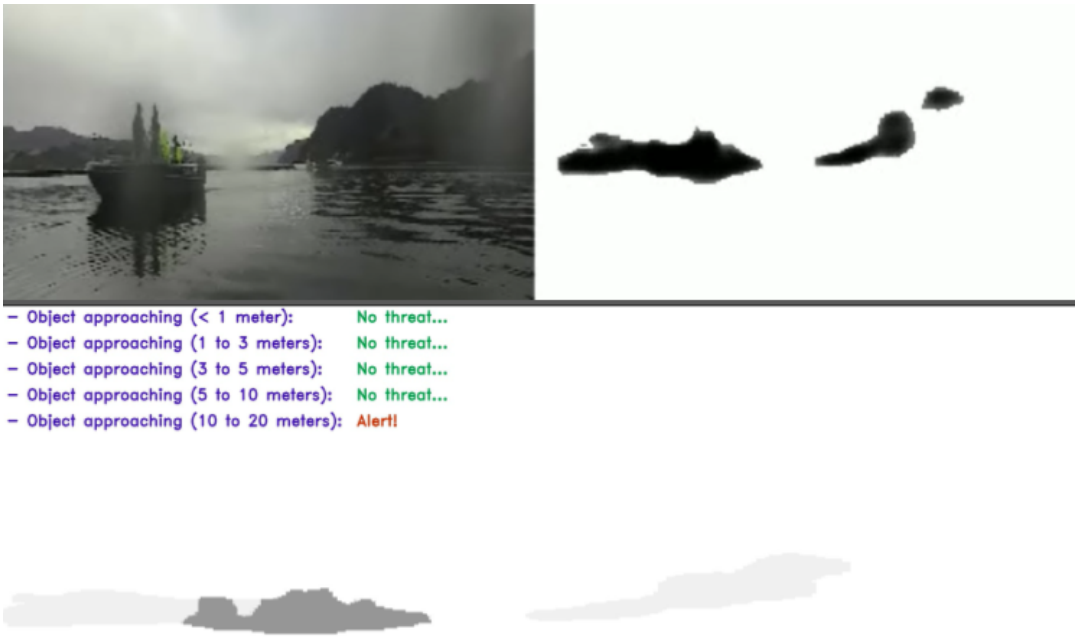

Figure 63: A boat can be recognized and estimated to be around 10 to 20 meters, even during the rain.

Figure 64: The pier and a boat can be recognized and pieces of two boats that are closer are estimated to be between 10 to 20 meters from the FishOtter.

Figure 65: A buoy can be recognized to be around 3 to 5 meters from the FishOtter.

Figure 66: Two sections of a boat that is approaching the FishOtter can be recognized.

Figure 67: A boat can be recognized and estimated to be around 10 to 20 meters, even during the rain.

Figure 68: A boat can be recognized and estimated to be around 10 to 20 meters, even during the rain.

5 Recommendations and Future Work

Although the developed application shows acceptable performance, there is still work to do in several domains in order to further optimize and improve it so that it can be effectively used as part of a robust system to execute anti collision strategies on-board. It is recommended to explore the following points based on the literature review, the experience gained during the project, and the optimization potentials discovered during the tests.

(i) The latency analysis concluded that there are some options that can be taken to increase the speed of the application: a) Properly setup and use the 8-bit integer quantized model, b) Execute the depth map extraction and inference tasks in parallel, and c) Reduce camera-frame dimensions by reducing resolution to decrement number of nested iteration while polling through the target mask pixels as already explained.

(ii) The OtterNet was quantized to an 8-bit integer. However, the resulting model has lower accuracy and a similar to worst execution time in the Jetson Xavier-NX. That is why it was not further evaluated. This may be due to the fact that post-training quantization is not enough for this neural network, and requires quantization-aware training to maintain accuracy [44].

(iii) Currently, the obstacle detection and proximity alert C++ application considers the generation of a verification video that is an output ".avi" file showing the output mask prediction of the OtterNet with the different distance intervals and the alert messages. This video is very important to verify that the application is running accordingly, but is only intended to be used as a debugging tool and can be removed to increase performance. After running some tests without generating the verification video, the increase in processing speed (FPS) was found to be approximately 18%. So, the Jetson Xavier-NX and Jetson Nano could run at a speed of approximately 2.2 FPS and 1.2 FPS respectively if the video generation functionality is removed.

(iv) Although the OtterNet shows promising results, it is important to continue its evaluation with more data. Due to the time consuming task it represents to build such a dataset (8 minutes per image), it was not possible to generate more than 100 instances of data during the period of this project. Having more data will give more statistically valid accuracy results and will be a key stone to increase the accuracy of the model during the next model training iterations. To this end, all the required source files to capture data, create the annotations and train the OtterNet are provided in the GitHub repository of this project. The 100 annotated images are provided along with the 200 images that are missing labels.

(v) Although the OtterNet has currently a MobileNetV2, using a different CNN-based model as backbone for transfer learning could be evaluated to see if there are benefits in terms of accuracy. One option is using a ResNet18 model, which has more

parameters than the MobilNet, but as it has been proven, the Jetson accelerator has resources to support the working load considering that the latency must not increase.

(vi) The ZED2i stereo-camera has an internal IMU sensor that has not been evaluated. Incorporating the USV roll and pitch measurements to the image segmentation model could further reduce false positives and false negatives by having a better horizon reference from the on-board IMU as seen in [31]. In addition, the StereoLabs-SDK allows recording stereo images along with IMU sensor data perfectly synchronized in a format called ".svo". This is critical for building the multi-sensor dataset that would be needed.

(vii) It is important to properly verify the stereo camera depth map at sea while comparing it to an accurate ground truth source such as a LIDAR.

(viii) The present work considers the FishOtter working conditions during daylight. However, the USV also works during night or under very dim light conditions. This is an entirely new challenge that could be addressed by projecting artificial light into the front of the USV, but will require a new neural network and verification of stereo-depth accuracy under those conditions.

(ix) The ZED2i stereo-camera has several flexible control parameters (resolution, brightness, contrast, saturation, exposure) that still have to be evaluated and properly tuned.

(x) The ZED2i stereo-camera comes pre-calibrated and there was no further need to adjust the calibration parameters after evaluating the depth accuracy for close depth estimation. However, a calibration task should be considered in the future.

(xi) During the sea trials, it became evident that the ZED2i stereo-camera had to be carefully mounted on the USV. To this end, it is important to make sure that the camera's field of view is 100% clear (no section of the FishOtter is visible), this is important to avoid having proximity alerts of less than 1 meter that correspond to the USV chassis.

(xii) By executing the obstacle detection and proximity alert application, the user can get alert messages through the command terminal. However, an additional improvement could consider transmitting the alert message to a server running a web application that could serve as a monitoring station for several FishOtters. In addition, it could also send the image or images of the potential obstacles that could be then annotated to enrich the Otter Dataset.

(xiii) The integration of the obstacle detection and proximity alert with the FishOtter main software controller based on DUNE, was out of the scope of this work due to limited time. So, there is work to be done in order to transmit the alert messages from the Jetson Xavier-NX to the Raspberry 4. This process is straightforward and should be addressed by creating a task in DUNE that expects an IMC format message from the Jetson Xavier. In addition, assessing how the depth of the object

changes during time can lead to estimating the object's speed, which along the IMU information provided by stereo-camera can be used to design a navigation maneuver in the same application and transmit it through IMC for the Raspberry Pi to execute.

6 Conclusions

The NTNU FishOtter USV is a mobile platform for autonomous robotic search that aims to facilitate investigation and understanding of fundamental biological processes in the ocean such as the migration and movement ecology of fishes. However, the FishOtter is not yet a fully autonomous vehicle. In addition, several field experiences have shown the threat that unknown small to medium size obstacles represent to the FishOtter. Therefore, it is imperative to make the USV self-aware of its surroundings during runtime so it can execute anti-collision maneuvers accordingly. Few studies and practical work exists around maritime obstacle detection, and several of the existing ones contemplate the use of LIDAR or radar sensors which are not suitable for small size USVs such as the FishOtter.

The present work shows in detail an end-to-end development of an obstacle detection and proximity alert application for collision avoidance at sea using a stereo camera-based solution. From the design, integration, deployment and testing of the solution on real working conditions, this work represents a base to build a more robust system considering all the optimization potentials discovered along the way and mentioned in the previous chapter. In addition, the report condensates in technical detail the important steps from design to deployment of a deep learning model on an edge device such as the Jetson Nano and Xavier NX, and provides a series of source files well detailed to build such an application, in the GitHub repository of this project.

The hardware selected for this project is a stereo-camera ZED2i and a Jetson Xavier-NX for deployment of the solution at the edge, both provided by StereoLabs with an industrial-grade enclosure design (IP66). On the other hand, the deep learning component is a U-Net segmentation neural network called the OtterNet, developed using Python and Tensorflow. In addition, one of the public available datasets for maritime obstacle segmentation is used to train, validate and test the model. However, this project also shows how to start building a customized dataset from real recordings of the FishOtter, which includes left frame images from the stereo-camera and annotations created using MATLAB. This dataset, of a total of 100 images and annotations, is used for testing the OtterNet, and should be used to re-train the network expecting increase in accuracy. Furthermore, the stereo-vision depth estimates and the OtterNet are integrated in an obstacle detection and proximity estimation C++ application. After configuration of the edge device, the application is also deployed on the Jetson Xavier-NX. Besides, the application is also deployed on a Jetson Nano to assess its performance on a lower-cost and lower-performance hardware.

As already mentioned, the U-Net segmentation model is evaluated with 100 images manually annotated, resulting in a f1-score of 0.83. In addition, the accuracy of the stereo-depth estimates is also assessed, showing a relative error no larger than 5% for distances no longer than 15 meters. Finally, considering the resource constraint nature of edge devices such as the Jetson Xavier-NX and Nano, performance tests are executed to evaluate the latency of the application as well as the memory utilization and power consumption during

run-time. These tests are important to discover potential optimization opportunities and calculate the processing speed of the application, which is approximately 1 frame per second and 2 frames per second for the Jetson Nano and Jetson Xavier-NX respectively.

Overall, the developed obstacle detection and proximity alert application meets the functional and technical requirements defined during chapter three, and shows promising results to be potentially used as part of a collision avoidance protocol. It is important to mention that this project sets a technical base line for further optimizations, in a way that it details the pipeline that goes from design to implementation and deployment using the different platforms required to build the application. In addition, this document works both as a technical guide for code understanding and platform configuration, as well as providing an overall explanation of the project design and motivation.

Figure 69: Field excursion for FishOtter sea trials.

7 Project Repository

As it has been mentioned several times over the report, this work includes a GitHub repository where all the source files and configuration scripts are included. This chapter will explain the organization of the repository.

GitHub Project Repository:
Maritime Obstacle Detection and Proximity Alert - FishOtter USV

Below is an explanation of what can be found in each of the directories. It is important to mention that all of the source files include detailed comments. In addition, all the build procedures for the Jetson devices are executed using CMAKE, so it is important to respect the file structure already given in the directories and that can be directly cloned. In addition, all of the file PATHs have to be changed depending on the system and device. Finally, every Python script was developed using Google Colab, so they can also be easily deployed in that development environment.

Obstacle Detection and Proximity Alert Application

(i) **Jetson Xavier NX and Nano:** Includes two directories with the necessary files to build the C++ application on the Jetson devices. The application needs the name of the verification video as an input argument. In addition, it uses the stereo-camera data stream as an input and runs for 200 frames before it stops. This parameter is hard-coded so the source code has to be modified when the application needs to run continuously. The application also generates a verification video.

(ii) **Windows Visual Studio:** Includes the files needed to build the C++ application in the development environment using Windows Visual Studio 2022. This application uses a ".svo" video as input and is used for development purposes because it allows to check results and make adjustments on the application without the need of actually being on the field with real camera data stream. The application will run until it finishes processing all the ".svo" video frames and will also provide a verification video.

ONNX Configuration CMAKE

(i) **ONNX-cmakeconfig:** Includes the cmake configuration file for ONNX runtime in Jetpack which is provided by [13].

Record Video at the Edge

(i) **Record SVO Video:** Includes the files needed to build the C++ application to record 100 frames in ".svo" format. This is a modified code provided by Stereolabs-SDK examples [22] and is used to record the 60 videos on the field, in order to generate the first instances of the Otter Dataset.

(ii) **Export SVO to AVI:** Includes the files needed to build the C++ application to export a SVO video to AVI format. This is a modified code provided by Stereolabs-SDK examples [21] and is used to get a ".avi" extension video that is the input the Video Generation Python script needs to create the OtterNet output mask video results.

OtterNet Training

(i) **Dataset Modification - Script:** Includes the python script that converts the public dataset to the binarized format as required by the OtterNet training script.

(ii) **OtterNet Training - Script:** Includes the python script to train the U-Net neural network called the OtterNet.

(iii) **Convert Model to ONNX - Script:** Includes the python script to convert a keras tensorflow model to ONNX format. The script also includes the quantization procedure.

(iv) **MaSTr1325 Modified Dataset:** Includes a Google Drive link to all the 1325 images and binarized target masks from the original public dataset.

(v) **Otter Dataset:** Includes a Google Drive to all the 100 images and binarized target masks from the Otter Dataset. In addition, it also includes the remaining 200 images that are still to be labeled.

Experiments

(i) **Tegrastats Analysis:** Includes the python script that analyzes Tegrastats log files for both edge devices: Jetson Xavier-NX or Nano.

(ii) **Depth Extraction:** Includes the files needed to build the C++ application to extract depth measurements from a given pixel coordinate with a ".svo" video format recording as the input in Visual Studio 2022.

Results

(i) **Obstacle Detection and Proximity Alert verification videos:** Includes a Google Drive link to 5 the verification video results from the Jetson Xavier-NX.

(ii) **OtterNet Output videos:** Includes a Google Drive link to 5 of the videos generated from the OtterNet obstacle segmentation task done in Python.

References

[1] Cuda execution provider. https://onnxruntime.ai/docs/execution-providers/CUDA-ExecutionProvider.html. Accessed: 2022-10-03.

[2] Dune unified navigation environment. https://lsts.fe.up.pt/toolchain/dune. Accessed: 2022-10-15.

[3] How to install jetpack. https://docs.nvidia.com/jetson/jetpack/index.html. Accessed: 2022-10-03.

[4] How to install zed sdk on nvidia jetson. https://www.stereolabs.com/docs/installation/jetson/. Accessed: 2022-10-03.

[5] An in-depth look at google's first tensor processing unit (tpu). https://cloud.google.com/blog/products/ai-machine-learning/an-in-depth-look-at-googles-first-tensor-processing-unit-tpu. Accessed: 2022-09-14.

[6] Installing cmake. https://cmake.org/install/. Accessed: 2022-10-10.

[7] Jetson nano developer kit. https://developer.nvidia.com/embedded/jetson-nano-developer-kit. Accessed: 2022-12-04.

[8] Jetson xavier nx series. https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-xavier-nx/. Accessed: 2022-11-12.

[9] Lidar and multibeam survey using an usv at trondheim fjord in norway. https://geo-matching.com/content/lidar-and-multibeam-survey-using-an-usv-at-trondheim-fjord-in-norway. Accessed: 2022-10-12.

[10] Maritime robotics. https://www.maritimerobotics.com/. Accessed: 2022-12-05.

[11] The ntnu fishotter project - a robotic fish tracking system. https://otter.itk.ntnu.no/doku.php?id=start. Accessed: 2022-10-15.

[12] Obstacle detection in a marine environment. https://box.vicos.si/borja/viamaro/index.html. Accessed: 2022-09-06.

[13] Onnxruntimeconfig cmake. https://github.com/microsoft/onnxruntime/issues/3124. Accessed: 2022-10-23.

[14] Open neural network exchange. https://onnx.ai/. Accessed: 2022-10-23.

[15] Rolls-royce and finferries demonstrate world's first fully autonomous ferry. https://www.rolls-royce.com/media/press-releases/2018/03-12-2018-rr-and-finferries-demonstrate-worlds-first-fully-autonomous-ferry.aspx. Accessed: 2022-10-14.

[16] Stereolabs zed2i. https://www.stereolabs.com/zed-2i/. Accessed: 2022-12-04.

[17] Sword 15 a11ue. https://www.msi.com/Laptop/Sword-15-A11UX/Specification. Accessed: 2022-12-12.

[18] tegrastats utility. https://docs.nvidia.com/drive/drive_os_5.1.6.1L/nvvib_docs/index.html#page/DRIVE_OS_Linux_SDK_Development_Guide/Utilities/util_tegrastats.html. Accessed: 2022-11-13.

[19] Tensorrt execution provider. https://onnxruntime.ai/docs/execution-providers/TensorRT-ExecutionProvider.html. Accessed: 2022-10-05.

[20] Unmanned surface vehicle, sounder. https://www.kongsberg.com/maritime/products/marine-robotics/uncrewed-surface-vehicle-sounder/. Accessed: 2022-10-14.

[21] Zed sdk - svo export. https://github.com/stereolabs/zed-examples/tree/master/svo%20recording/export/cpp. Accessed: 2022-11-12.

[22] Zed sdk - svo recording. https://github.com/stereolabs/zed-examples/tree/master/svo%20recording/recording/cpp. Accessed: 2022-11-12.

[23] F.; Martín D.; De La Escalera A.; Armingol J.M. Al-Kaff, A.; García. *Obstacle Detection and Avoidance System Based on Monocular Camera and Size Expansion Algorithm for UAVs*, pages 17,1061, May 2017.

[24] Bashar Alsadik and Nagham Amer Abdulateef. Epipolar geometry between photogrammetry and computer vision–a computational guide. *ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, 5:25–32, 2022.

[25] Alireza Asvadi, Cristiano Premebida, Paulo Peixoto, and Urbano Jose C. Nunes. 3d lidar-based static and moving obstacle detection in driving environments: An approach based on voxels and multi-region ground planes. *Robotics Auton. Syst.*, 83:299–311, 2016.

[26] Samira Badrloo, Masood Varshosaz, Saied Pirasteh, and Jonathan Li. Image-based obstacle detection methods for the safe navigation of unmanned vehicles: A review. *Remote Sensing*, 14(15), 2022.

[27] Ron Banner, Yury Nahshan, Elad Hoffer, and Daniel Soudry. ACIQ: analytical clipping for integer quantization of neural networks. *CoRR*, abs/1810.05723, 2018.

[28] Nicola Bernini, Massimo Bertozzi, Luca Castangia, Marco Patander, and Mario Sabbatelli. Real-time obstacle detection using stereo vision for autonomous ground vehicles: A survey. *17th International IEEE Conference on Intelligent Transportation Systems (ITSC)*, pages 873–878, 2014.

[29] Borja Bovcon, Rok Mandeljc, Janez Pers, and Matej Kristan. Stereo obstacle detection for unmanned surface vehicles by imu-assisted semantic segmentation. *Robotics and Autonomous Systems*, 104, 02 2018.

[30] Borja Bovcon, Jon Muhovič, Janez Perš, and Matej Kristan. The mastr1325 dataset for training deep usv obstacle detection models. In *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 3431–3438, 2019.

[31] Borja Bovcon, Janez Perš, Matej Kristan, et al. Stereo obstacle detection for unmanned surface vehicles by imu-assisted semantic segmentation. *Robotics and Autonomous Systems*, 104:1–13, 2018.

[32] Teng Cao, Zhi-Yu Xiang, and Ji-Lin Liu. Perception in disparity: An efficient navigation framework for autonomous vehicles with stereo cameras. *Trans. Intell. Transport. Syst.*, 16(5):2935–2948, sep 2015.

[33] Jiasi Chen and Xukan Ran. Deep learning with edge computing: A review. *Proceedings of the IEEE*, 107(8):1655–1674, 2019.

[34] Dennis P. Curtin. Understanding the baseline. https://www.shortcourses.com/stereo/stereo3-14.html. Accessed: 2022-10-25.

[35] Alberto Dallolio, Helge B. Bjerck, Henning A. Urke, and Jo A. Alfredsen. A persistent sea-going platform for robotic fish telemetry using a wave-propelled usv: Technical solution and proof-of-concept. *Frontiers in Marine Science*, 9, 2022.

[36] Manuel Dominguez-Morales, Angel Jiménez-Fernandez, Rafael Paz-Vicente, Alejandro Linares-Barranco, and Gabriel Jimenez. *Stereo Matching: From the Basis to Neuromorphic Engineering*. 07 2012.

[37] Michael Garland, Scott Le Grand, John Nickolls, Joshua Anderson, Jim Hardwick, Scott Morton, Everett Phillips, Yao Zhang, and Vasily Volkov. Parallel computing experiences with cuda. *IEEE Micro*, 28(4):13–27, 2008.

[38] Ross B. Girshick. Fast R-CNN. *CoRR*, abs/1504.08083, 2015.

[39] Ross B. Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. *CoRR*, abs/1311.2524, 2013.

[40] Sercan Gul, Javid Shiriyev, Vivek Singhal, Oney Erge, and Cenk Temizel. Chapter three - advanced materials and sensors in well logging, drilling, and completion operations. In Cenk Temizel, Mufrettin Murat Sari, Celal Hakan Canbaz, Luigi A. Saputelli, and Ole Torsæter, editors, *Sustainable Materials for Oil and Gas Applications*, volume 1 of *Advanced Materials and Sensors for the Oil and Gas Industry*, pages 93–123. Gulf Professional Publishing, 2021.

[41] Ryan Halterman and Michael Bruch. Velodyne hdl-64e lidar for unmanned surface vehicle obstacle detection. In *Unmanned Systems Technology XII*, volume 7692, pages 123–130. SPIE, 2010.

[42] Jungwook Han, Yonghoon Cho, Jonghwi Kim, Jinwhan Kim, Nam-sun Son, and Sun Young Kim. Autonomous collision detection and avoidance for aragon usv: Development and field tests. *Journal of Field Robotics*, 37(6):987–1002, 2020.

[43] Jungwook Han, Jeonghong Park, Jinwhan Kim, and Nam sun Son. Gps-less coastal navigation using marine radar for usv operation. *IFAC-PapersOnLine*, 49:598–603, 2016.

[44] Song Han, Jeff Pool, John Tran, and William J. Dally. Learning both weights and connections for efficient neural networks. *CoRR*, abs/1506.02626, 2015.

[45] Tianyi David Han and Tarek S. Abdelrahman. hicuda: High-level gpgpu programming. *IEEE Transactions on Parallel and Distributed Systems*, 22(1):78–90, 2011.

[46] Hordur K. Heidarsson and Gaurav S. Sukhatme. Obstacle detection and avoidance for an autonomous surface vehicle using a profiling sonar. In *2011 IEEE International Conference on Robotics and Automation*, pages 731–736, 2011.

[47] Kunsoo Huh, Jaehak Park, Junyeon Hwang, and Daegun Hong. A stereo vision-based obstacle detection system in vehicles. *Optics and Lasers in Engineering*, 46(2):168–178, 2008.

[48] Antonio RamÓn Jimenez Ruiz and Fernando Seco Granja. A short-range ship navigation system based on ladar imaging and target tracking for improved safety and efficiency. *IEEE Transactions on Intelligent Transportation Systems*, 10(1):186–197, 2009.

[49] Hanguen Kim, Jungmo Koo, Donghoon Kim, Byeolteo Park, Yonggil Jo, Hyun Myung, and Donghwa Lee. Vision-based real-time obstacle segmentation algorithm for autonomous surface vehicle. *IEEE Access*, 7:179420–179428, 2019.

[50] Heesu Kim, Evangelos Boulougouris, and Sang-Hyun Kim. Object detection algorithm for unmanned surface vehicle using faster r-cnn. In *World Maritime Technology Conference 2018*, 2018.

[51] Lingyun Li and Hongbing Ma. Rdctrans u-net: A hybrid variable architecture for liver ct image segmentation. *Sensors*, 22(7), 2022.

[52] Michael Linderman, Robert Bruggner, Vivek Athalye, Teresa Meng, Narges Asadi, and Garry Nolan. High-throughput bayesian network learning using heterogeneous multicore computers. volume 2010, pages 95–104, 06 2010.

[53] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott E. Reed, Cheng-Yang Fu, and Alexander C. Berg. SSD: single shot multibox detector. *CoRR*, abs/1512.02325, 2015.

[54] Yu Han Liu. Feature extraction and image recognition with convolutional neural networks. In *Journal of Physics: Conference Series*, volume 1087, page 062032. IOP Publishing, 2018.

[55] Yuncheng Lu, Zhucun Xue, Gui-Song Xia, and Liangpei Zhang. A survey on vision-based uav navigation. *Geo-spatial Information Science*, 21(1):21–32, 2018.

[56] Yunhong Liu Qingjie. Mashaly, Ahmed Wang. Efficient sky segmentation approach for small uav autonomous obstacles avoidance in cluttered environment. *IGARSS 2016 - 2016 IEEE International Geoscience and Remote Sensing Symposium*, pages 6710–6713, July 2016.

[57] Naveen Mellempudi, Abhisek Kundu, Dipankar Das, Dheevatsa Mudigere, and Bharat Kaul. Mixed low-precision deep learning inference using dynamic fixed point. *CoRR*, abs/1701.08978, 2017.

[58] Jon Muhovic, Rok Mandeljc, Borja Bovcon, Matej Kristan, and Janez Per. Obstacle tracking for unmanned surface vessels using 3-d point cloud. *IEEE Journal of Oceanic Engineering*, 45:786–798, 2020.

[59] Ishrat Zahan Mukti and Dipayan Biswas. Transfer learning based plant diseases detection using resnet50. In *2019 4th International Conference on Electrical Information and Communication Technology (EICT)*, pages 1–6, 2019.

[60] Luca Nobile, Marco Randazzo, Michele Colledanchise, Luca Monorchio, Wilson Villa, Francesco Puja, and Lorenzo Natale. Active exploration for obstacle detection on a mobile humanoid robot. *Actuators*, 10(9), 2021.

[61] Ram Prasad Padhy, Suman Kumar Choudhury, Pankaj Kumar Sa, and Sambit Bakshi. Obstacle avoidance for unmanned aerial vehicles: Using visual features in unknown environments. *IEEE Consumer Electronics Magazine*, 8(3):74–80, 2019.

[62] Adam Paszke, Abhishek Chaurasia, Sangpil Kim, and Eugenio Culurciello. Enet: A deep neural network architecture for real-time semantic segmentation. *CoRR*, abs/1606.02147, 2016.

[63] David Pfeiffer and Uwe Franke. Towards a global optimal multi-layer stixel representation of dense 3d data. 08 2011.

[64] José Pinto, Paulo S Dias, Ricardo Martins, Joao Fortuna, Eduardo Marques, and Joao Sousa. The lsts toolchain for networked vehicle systems. In *2013 MTS/IEEE OCEANS-Bergen*, pages 1–9. IEEE, 2013.

[65] Dilip K. Prasad, Chandrashekar Krishna Prasath, Deepu Rajan, Lily Rachmawati, Eshan Rajabally, and Chai Quek. Object detection in a maritime environment: Performance evaluation of background subtraction methods. *IEEE Transactions on Intelligent Transportation Systems*, 20(5):1787–1802, 2019.

[66] Ling Qian, Zhiguo Luo, Yujian Du, and Leitao Guo. Cloud computing: An overview. In *IEEE international conference on cloud computing*, pages 626–631. Springer, 2009.

[67] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. *CoRR*, abs/1603.05279, 2016.

[68] Joseph Redmon, Santosh Kumar Divvala, Ross B. Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. *CoRR*, abs/1506.02640, 2015.

[69] Shaoqing Ren, Kaiming He, Ross B. Girshick, and Jian Sun. Faster R-CNN: towards real-time object detection with region proposal networks. *CoRR*, abs/1506.01497, 2015.

[70] Babak Rokh, Ali Azarpeyvand, and Alireza Khanteymoori. A comprehensive survey on model quantization for deep neural networks. *arXiv preprint arXiv:2205.07877*, 2022.

[71] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. *CoRR*, abs/1505.04597, 2015.

[72] and Haidi Ibrahim Rostam Affendi Hamzah. Literature survey on stereo vision disparity map algorithms. *Journal of Sensors*, page 23, 2016.

[73] Inwook Shim, Jongwon Choi, Seunghak Shin, Tae-Hyun Oh, Unghui Lee, Byungtae Ahn, Dong-Geol Choi, David Hyunchul Shim, and In-So Kweon. An autonomous driving system for unknown environments using a unified map. *IEEE Transactions on Intelligent Transportation Systems*, 16(4):1999–2013, 2015.

[74] W.; Liu H.; Yan J.; Gao S.; Feng P." "Sun, B.; Li. Obstacle detection of intelligent vehicle based on fusion of lidar and machine vision. *Eng. Lett.*, page 29, 2021.

[75] Anete Vagale, Rachid Oucheikh, Robin T Bye, Ottar L Osen, and Thor I Fossen. Path planning and collision avoidance for autonomous surface vehicles i: a review. *Journal of Marine Science and Technology*, pages 1–15, 2021.

[76] Øystein Volden, Annette Stahl, and Thor I Fossen. Vision-based positioning system for auto-docking of unmanned surface vehicles (usvs). *International Journal of Intelligent Robotics and Applications*, 6(1):86–103, 2022.

[77] Han Wang and Zhuo Wei. Stereovision based obstacle detection system for unmanned surface vehicle. pages 917–921, 12 2013.

[78] Yan Wang, Wei-Lun Chao, Divyansh Garg, Bharath Hariharan, Mark Campbell, and Kilian Q Weinberger. Pseudo-lidar from visual depth estimation: Bridging the gap in 3d object detection for autonomous driving. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 8445–8453, 2019.

[79] Yan Wang, Bin Yang, Rui Hu, Ming Liang, and Raquel Urtasun. Plumenet: Efficient 3d object detection from stereo images. In *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 3383–3390. IEEE, 2021.

[80] Nakwan. Woo, Joohyun Kim. Vision based obstacle detection and collision risk estimation of an unmanned surface vehicle. *2016 13th International Conference on Ubiquitous Robots and Ambient Intelligence (URAI)*, pages 461–465, August 2016.

[81] Feng Xia, Laurence T Yang, Lizhe Wang, and Alexey Vinel. Internet of things. *International journal of communication systems*, 25(9):1101, 2012.

[82] Linghong Yao, Dimitrios Kanoulas, Ze Ji, and Yuanchang Liu. Shorelinenet: An efficient deep learning approach for shoreline semantic segmentation for unmanned surface vehicles. In *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 5403–5409, 2021.

[83] Ryota Yoneyama and Yuichiro Dake. Vision-based maritime object detection covering far and tiny obstacles. *IFAC-PapersOnLine*, 55(31):210–215, 2022. 14th IFAC Conference on Control Applications in Marine Systems, Robotics, and Vehicles CAMS 2022.

[84] Xiaoyan Yu and Marin Marinov. A study on recent developments and issues with obstacle detection systems for automated vehicles. *Sustainability*, 12(8), 2020.

[85] Zutao Zhang, Hong Xu, Zhifeng Chao, Xiaopei Li, and Chunbai Wang. A novel vehicle reversing speed control based on obstacle detection and sparse representation. *IEEE Transactions on Intelligent Transportation Systems*, 16(3):1321–1334, 2015.

[86] Trine Ødegard Olsen. Stereo vision using local methods for autonomous ferry. https://folk.ntnu.no/edmundfo/msc2019-2020/TrineOlsenStereoVision.pdf. Accessed: 2022-10-25.