CHALMERS
UNIVERSITY OF TECHNOLOGY

Drone safe to launch system using machine learning

Master's thesis in Complex adaptive systems

Elina Sahlberg, Björn Krook Willén

DEPARTMENT OF PHYSICS

CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2023
www.chalmers.se

Master's thesis 2023

Drone safe to launch system using machine learning

Elina Sahlberg, Björn Krook Willén

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Physics
Chalmers University of Technology
Gothenburg, Sweden 2023

Drone safe to launch system using machine learning
Elina Sahlberg
Björn Krook Willén

Supervisors: Victor Nilsson, Infotiv, Hamid Ebadi, Infotiv
Examiner: Giovanni Volpe, Professor, Department of Physics, Chalmers University of Technology

Master's Thesis 2023
Department of Physics
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Drone and drone launch platform located at Långedrag

Typeset in LaTeX
Printed by Chalmers Reproservice
Gothenburg, Sweden 2023

Drone safe to launch detection using machine learning
Elina Sahlberg, Björn Krook Willén
Department of Physics
Chalmers University of Technology

Abstract

Svenska Sjöräddningssällskapet has partnered with Infotiv to develop a prototype drone and launcher system for search-and-rescue missions at sea. In this thesis project, we investigate the possibility of automating parts of the launch sequence of a seaborne surveillance drone. The main goal is to train a neural network model to use a camera feed to determine if it is safe or not to launch the drone in a given direction, and then integrate this solution with the graphic user interface through an application programming interface. Monocular depth estimation using transfer learning and the KITTI data set is evaluated. The KITTI data set does not contain maritime scenery, leading to an unsatisfactory monocular depth estimation model. U-net and convolutional neural network models are trained on the MaSTr1325 data set, which contains semantically segmented maritime imagery. We collect additional data for the semantic segmentation models and create a post-processing step that evaluates if it is safe to launch or not. These models yielded satisfactory results, and the convolutional neural network will be used by the drone operator as an extra safety measure during launch.

Keywords: Deep learning, monocular depth estimation, semantic segmentation, maritime environments, quantization, neural network, drone.

Acknowledgements

We wish to thank our supervisors Hamid Ebadi and Victior Nilsson for guiding us through the SSRS project and for all the technical guidance. Daniel Petersson and Jacob Lundkvist for all your initiatives and helping us throughout this project.

Elina Sahlberg, Björn Krook Willén, Gothenburg, June 2023

List of Acronyms

Below is the list of acronyms that have been used throughout this thesis listed in alphabetical order:

AI	Artificial intelligence
ANN	Artificial Neural Network
API	Application Programming Interface
ARM	Advanced RISC Machines
CNN	Convolutional Neural Network
CPU	Central Processing Unit
FCN	Fully Convolutional Neural Network
FLOPS	Floating Point Operations per Second
GEDet	Gradient Edge Detection
GPS	Global Positioning System
KITTI	Karlsruhe Institute of Technology and Toyota Technological Institute
Lidar	Light Detection and Ranging
MAE	Mean Absolute Error
MAPE	Mean Absolute Percentage Error
ML	Machine Learning
MSE	Mean Squared Error
RADAR	Radio Detection and Ranging
RAM	andom access memory
ReLU	Rectified Linear Unit
ResNet	Residual Neural Network
RMSE	Root Mean Square Error
RMSP	Root Mean Square Propagation
SSIM	Structural Similarity Index
URL	Uniform Resource Locator
USV	Unmanned Surface Vehicles

Nomenclature

Below is the nomenclature of indices, sets, parameters, and variables that have been used throughout this thesis.

Indices

i	Indices for pixels in neural network

Parameters

n	Number of pixels

Variables

y_i	Pixel value i for predicted output of neural network
x_i	Pixel value i of target for neural network
μ_x	pixel sample mean of x
μ_y	pixel sample mean of y
σ_x^2	variance of x
σ_y^2	variance of y
σ_{xy}^2	covariance of x and y
c_1, c_2	2 variables that stabilize the division with weak denominator
TP	True positive
FP	False positive
FN	False negative

Contents

List of Figures

List of Tables

1
Introduction

1.1 Backgruond

The use of drones for search-and-rescue missions at sea is an exciting prospect. For this reason, Sjöräddningssällskapet has partnered with Infotiv to develop a prototype drone and launcher system [5]. These drones can be made relatively cheaply, and fly autonomously using a global positioning system (GPS) for navigation. They provide the rescuers with a view of the scene before they arrive, giving them valuable information. The goal for Sjöräddningssällskapet is to have numerous drones situated along the Swedish coastline ready to help the rescuers. These drones will be launched from specially constructed launch towers, and because response time is key, ideally one would like to automate as much of the operation as possible. Since the launch towers are stationed in a dynamic environment where weather conditions may change and vessels like sailboats might obstruct the flight path, software is needed that can determine if launching the vehicle in a given direction is safe or not. The intended outcome is for the drone to eventually be fully automated, with the only input being the coordinates the drone will fly to. Another thing to take into account is that the drones need to be cheaply manufactured.

1.2 Purpose

We will develop a deep learning model that can help a drone avoid collisions at launch with an image input. This model is then to be integrated with the existing code base, where the model will output whether it is safe to launch or not. Preferably, there should also be an output that says if the model is unsure of its prediction. The main research questions to be investigated are the following:

- Can artificial neural networks be used to determine if it is safe to launch maritime drones?
- What hardware is required for such a system?
- Under what conditions will this system be able to operate?

Apart from these research questions, there were some additional requests from Infotiv:

- Integrate an Application Programming Interface (API) that communicates with the safe-to-launch system.
- Develop automated tests for all functionality of the safe-to-launch system.

1.3 Delimitations

We will only implement a model that helps the drone avoid collisions at launch and integrate this model into the existing code base. We will not automate the process of finding a suitable launch angle but simply evaluate a pre-selected one. We will limit ourselves to building a collision avoidance model that can mainly detect obstacles in good weather conditions (no rain or fog) and during the day. The furthest distance obstacles are to be detected is around 75 metres. The model will be implemented on a raspberry pi and have an inference time of 1 second or less. These delimitations were discussed and agreed upon by Infotiv and the authors of this thesis.

1.4 Societal, Ethical and Ecological Aspects

The use of artificial intelligence (AI) and artificial neural networks (ANN) in autonomous systems is a phenomenon on the rise globally. There is much to gain by automating various repetitive tasks, but there are also potential pitfalls to look out for. When handing over control to an AI, robustness is absolutely paramount, and every possible failure case must be considered and the risk minimised. In this particular case failure of the ANN model could result in the drone being lost or, in the worst case personal injury. To minimise such risks, the model should be very conservative with its decisions and hand over control to a human operator if there is any uncertainty at all about the ability to launch safely.

Another aspect to consider is the energy cost of training large neural network models. Studies have shown that training large ANN models can generate a considerable amount of carbon emissions [6], and although the model proposed for this project is not particularly large, it is possible to mitigate its environmental impact even further through the use of transfer learning.

2
Theory

2.1 Machine learning

Machine learning (ML) is a subfield of artificial intelligence that involves training machines to recognize patterns and make predictions or decisions from data. ANN are a type of machine learning model that mimics the biological neural network in brains. They consist of multiple nodes called artificial neurons that are connected to each other. Just like synapses in the brain they can receive, process and transmit signals to each other. The signals are real values and the processing step consists of a multiplication "weight" and an addition "bias". ANN typically have an input and an output layer, with multiple hidden layers in between: Signals are sent from layer to layer towards the output. Neural networks are used in a wide range of applications, such as natural language processing, image recognition, speech recognition, autonomous driving, fraud detection, and prediction of disease outcomes. They can handle large and complex data sets, and rapidly learn from new data, making them a powerful tool for providing more accurate predictions and decisions.

2.2 Convulotional neural network

Convolutional neural networks (CNN) are a type of artificial neural network designed to capture spatial information and patterns in images. While work on modern CNNs began in the 1990s, they have gained significant popularity in recent years, particularly in the field of computer vision [7]. CNNs consist of a few to hundreds of layers, each of which learns to detect a particular feature of an image. The first layers often detect simple features such as light or contrast, while deeper layers detect increasingly complex features such as faces, text, or wrinkled fabric. Unlike feed forward artificial neural networks, CNN Uses kernels, filters that are used to extract features from images. A kernel is a matrix that moves over input data performing a dot product. The use of kernels reduces the size of an input by creating new features that summarize the input.

Typically, the last layer or last few layers in a CNN are not convolutional layers but fully connected layers, also called dense layers. These layers classify the input picture based on the detected features and output the final result, such as identifying that the image contains a cat.

Convolutional neural networks without fully connected layers are called fully convo-

lutional networks (FCN). These types of networks are often used to classify pixels, as they avoid the significant number of trainable parameters in dense layers, reducing both training and running time. FCNs are commonly used in tasks such as semantic segmentation and depth estimation[8].

2.2.1 Pooling layers

In addition to convolutional layers and fully connected layers, another important component of a CNN is the pooling layer. The purpose of the pooling layer is to downsample the feature maps produced by the convolutional layers, which makes the network more computationally efficient. The most common type of pooling is max pooling, which takes the maximum value within a window of pixels and outputs that value as the new value for that region. This helps to preserve the important features in the image while reducing the dimension of the data [9].

2.2.2 ReLU activation function

Another important component of ANN is the activation function, which introduces non-linearity into the network. The rectified linear unit (ReLU) activation function is one of the most commonly used activation functions in ANNs. ReLU is defined as max(0, x), which means that any negative values are set to zero and any positive values are unchanged. ReLU has a number of advantages over other activation functions, including its simplicity and computational efficiency. It also helps to prevent the vanishing gradient problem, which can occur when gradients become very small and make it difficult to update the weights of the network during training[10].

2.3 Encoder-decoder neural network

The architecture of encoder-decoder neural networks consists of 2 main parts, an encoder that contract the image and a decoder that acts as an inverse encoder and expands the image representation back to an image. When using images as an input the encoder is constructed of a normal FCN With multiple convolutional and pooling layers. Compression of data occurs leading to a bottleneck layer containing data representations known as feature maps of the input. The decoder decompresses data with transposed-convolutions and pooling layers. Training is done by comparing the output of the decoder and the desired output. Encoder-decoder networks are common to analyze visual imagery as they are faster to train than fully connected networks.

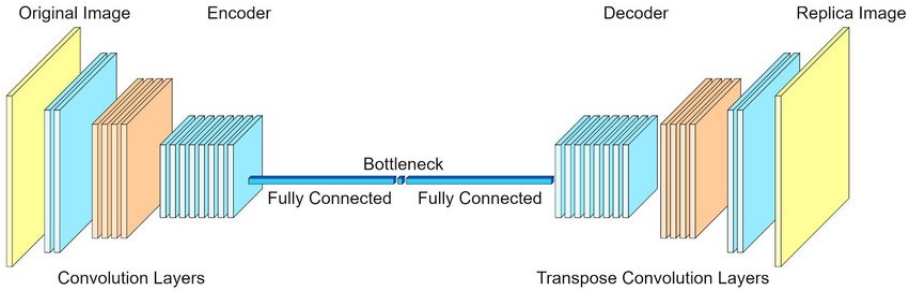

Figure 2.1: The structure of an encoder-decoder neural network with a bottleneck in between [1].

2.4 Transfer learning

To make training processes more efficient one can make use of publicly available pre-trained models. Most models that work on image data will learn to detect similar low-level features, and this knowledge can be reused for other tasks. To do this one copies the first few layers of a model and then adds on them to suit the task at hand, then more training is preformed to "fine tune" the model to the particular use-case at hand. There are a plethora of such models online, and many of them are under a public license and can be used however one wishes.

2.5 Semantic segmentation

Semantic segmentation or Image segmentation is a technique commonly used in computer vision to partition an image into multiple segments, typically separating different object or object types from each other. Each pixel is given one preexisting labels, ex the pixels of a cat is given the label "cat" and the rest of the pixels are given the label "not cat" as can be seen in figure 2.2. There can exist a number of labels in one picture but each pixel can only have one label. Semantic segmentation is useful in plenty of fields with the most important ones being self driving cars and medical imagery. Self driving cars uses semantic segmentation to e.g. distinguish the road from sidewalk, as well as to detect humans and traffic signs [11].

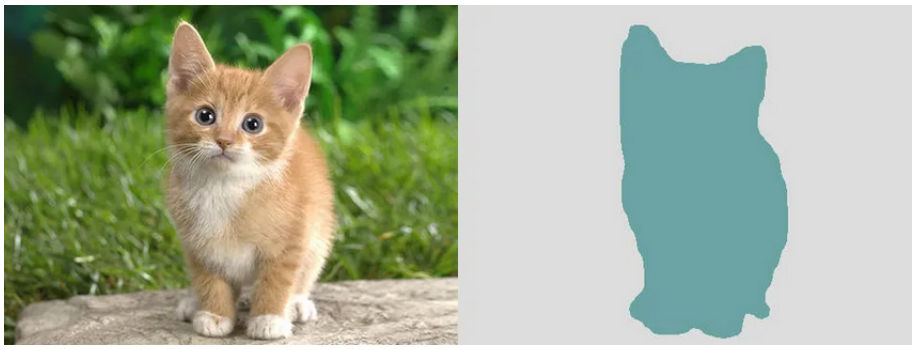

Figure 2.2: Semantic segmentation separating a cat from the background [2].

2.6 U-net

U-net is a commonly used neural network for semantic segmentation tasks. It was introduced by Olaf Ronneberger, Philipp Fischer and Thomas Brox in 2015, initially for biomedical image segmentation [12]. Like the encoder-decoder neural network it contains 2 parts, an encoder similar to a typical FCN, followed by a decoder part. The key feature of U-net is that it contains skip connections between the corresponding layers of the encoder and decoder part. These connections sends information from specific layers of the encoder part directly to corresponding layers in the decoder part. These skip connections help preserve spatial information and aid image reconstruction for small objects and details. A representation of the U-net architecture can be seen in figure 2.3, where the grey arrows represents skip connections.

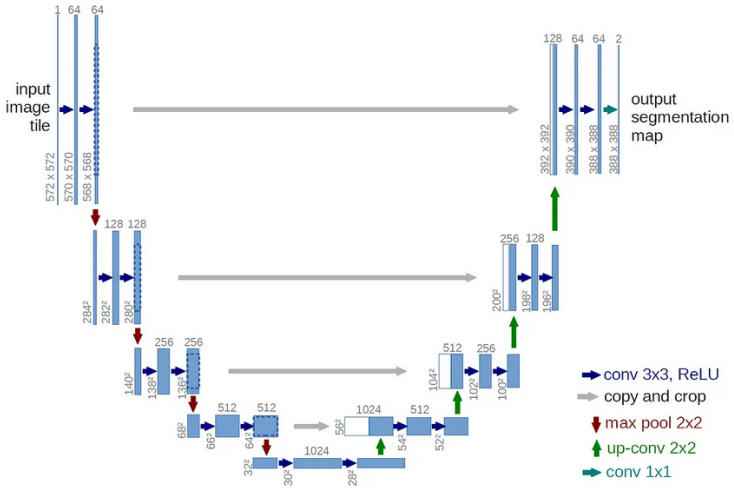

Figure 2.3: Architecture of U-net [2].

2.7 Data augmentation

Data augmentation is the process of creating new data by modifying already existing data. This is commonly used in machine learning to increase the size of the data set, which lead to more robust models and help avoiding over fitting. This improves the overall performance of the neural network and can be done in multiple ways. Flipping an image horizontally or vertically creates a mirrored image, useful in image analysis and creates an variation of the original image. Rotating and scaling an image creates different results compared to flipping an image and can be used in parallel or together. Noise and blurring can also be added to create variations of images.

2.8 Loss and evaluation functions

Loss functions are a essential part of neural networks that are used to measure and quantify the difference between the predicted output and the actual output. Loss

functions quantify how well a ANN is performing and by minimizing the loss during training the ANN will improve.

2.8.1 Mean absolute error

The mean absolute error (MAE) is one of the most commonly used loss functions for machine learning. MAE represents the average absolute deviation of the predictions from the actual values. It is useful if the training data has outliers as MAE does not penalize high errors caused by outliers it also provides an even measure of how well the model is performing. A lower value of MAE indicates better model performance, as it represents smaller errors in the predictions. It is calculated by taking the mean of the pixel wise absolute value of targets (x_i) and predicted output (y_i) in a depth map with n number of pixels.

$$\text{MAE} = \frac{\sum_{i=0}^{n} |y_i - x_i|}{n} \tag{2.1}$$

2.8.2 Mean Squared Error

Mean square error (MSE) is a measure of the average distance between the predicted and actual values, where the differences are squared and averaged. MSE is a popular metric because it weeds out outliers with large errors from the model by putting more weight on them, making it a good choice for models that need to minimize the impact of outliers. The formula for calculating MSE is [13]:

$$\text{MSE} = \frac{\sum_{i=0}^{n} (y_i - x_i)^2}{n} \tag{2.2}$$

also root mean squared error (RMSE)

$$\text{RMSE} = \sqrt{\frac{\sum_{i=0}^{n} (y_i - x_i)^2}{n}} \tag{2.3}$$

2.8.3 Structural similarity index loss function

The structural similarity (SSIM) index is used to predict the perceived quality of 2 digital images. It is done by extracting 3 key features from images; luminescence, contrast and structure. The SSIM index is calculated on various small windows of an image. The measure between 2 images x and y is:

$$\text{SSIM} = \frac{(2\mu_x\mu_y + c_1)(2\sigma_{xy} + c_2)}{(\mu_x^2 + \mu_y^2 + c_1)(\sigma_x^2 + \sigma_y^2 + c_2)} \tag{2.4}$$

μ is the pixel sample mean σ is the variance and c_1 ans c_2 are 2 variables that stabilizes the division with weak denominator[14].

2.8.4 Gradient edge detection

Gradient edge detection (GEDet) is used to find and highlight edges, in the case of depth estimation an edge is a change of depth and in semantic segmentation an edge is the change from a predicted class to another. Gradient edge detection is calculated by first finding the pixel wise gradient difference of y_i and x_i in the 2 directions g_x and g_y and than applying the following equation[15].

$$\text{Grad} = \frac{\sum_i^n |g_x| + |g_y|}{n} \tag{2.5}$$

2.9 Depth Estimation

Deep learning models have been shown to effectively learn and utilize monocular cues to perform depth estimation. Previously, encoder-decoder networks have been used for this task [15], as well as generative adversarial networks [16]. Depth estimation is the process of obtaining a visual representation of the distance of objects in a scene or image, known as a depth map. There are several methods for depth estimation, with active range sensors such as lidar being the most reliable. However, the high cost of lidar and its limitations with various textures, occlusions, and non-lambertian surfaces make it suboptimal in some cases. Another reliable methods is to use one or multiple cameras and simulate the function of the human eyes [17].

Figure 2.4: Pictures of before and after a depth estimation is applied, yellow indicates close object and darker colors objects further away [3].

2.9.1 Stereo depth estimation

Stereo depth estimation involves using two cameras situated within a small distance from each other. Depth is perceived using the small differences in the images captured by the two cameras. This phenomenon is known as retinal disparity, where objects close to the cameras seem to be at different places and objects further away less so. To calculate the actual distance of an object from the cameras, the first step is to find the object in both images and calculate the size of the disparity in the

images. The depth can then be calculated using the size of the disparity and the known distance between the two cameras. This process of depth estimation is also done automatically by the human eye and brain.

The drawback with Stereo depth estimation is that it becomes harder and almost impossible to determine depth at greater distances as the disparity shrinks and eventually disappears [17]. The human eye for example can only detect a disparity up to 10 meters [18], distances further than that and monocular depth estimation is used.

2.9.2 Monocular depth estimation

Monocular depth estimation uses one camera and one or several different monocular cues to perceive depth. One such cue is monocular movement parallax, which occurs when the observer moves their head from side to side, causing nearby objects to move at a higher relative velocity than distant objects. Another cue is relative size, where objects that appear smaller are assumed to be further away, given that the actual size of the object is known. Interposition is another cue, where objects that overlap with each other give the impression that the object in front is closer. Aerial perspective is a cue where distant objects appear less contrasted and may take on a bluish hue due to the atmosphere. Lastly, light and shade cues provide information about the shape and position of objects in space through the shadows they cast [19].

2.10 Raspberry pi

A raspberry pi are small single-board computers, that are popular due to their low cost, modularity, and open design. The raspberry pi uses a different type of central processing unit (CPU): advanced reduced instruction set computer machines (ARM). Is has 2GB of random access memory (RAM) ,4 cores, and can compute 13.5 giga floating point operations per second (GFLOPS). A raspberry pi is integrated in the drone launcher and will run the ANN model, therefore it is important that all dependencies and code work on this architecture. Some versions of torch for example will attempt to execute illegal instructions on arm, so selecting the right versions is important[20].The benefits of using a raspberry pi for embedded applications is that it is small, energy efficient and can run the Linux operative system. This means that programs do not have to be compiled to run on bare metal, instead we can use a high level interpreted programming language like python for all our code. With this setup prototyping code is much faster, and pythons garbage collector also ensures there are no memory leaks that could potentially crash our program.

2.11 Quantization

Quantization of a ANN model is the process of converting all weights and activations to use unsigned 8-bit integers instead of 32-bit floating points during inference, improving performance and memory efficiency. There are three main types

of quantization schemes: post-training static quantization, post-training dynamic quantization, and quantization aware training. Post training static quantization - the method we will be using for this thesis - works by inserting "observers" in each layer of the model. Now when data is passed through the model, the observers keep track of the activation values of each layer, allowing it to compute their distributions. After this calibration step is done, all the weights and biases are converted to integer format and the computed activation distributions are used to determine how activations should be quantized during inference[21].
The main benefits of quantization is a four time increase in memory efficiency and a potential twofold of even threefold inference time speedup[21]. This is because integer addition and multiplication is significantly faster than the corresponding floating point operations on some hardware, and since the models in question are intended to run on a single core ARM CPU, this could potentially have a big impact on final performance.

3
Method

The project has been divided into different phases. Firstly, the selection of the visual sensor, the models to be used, and the programming tools to be employed were decided through literature study and experimentation. Secondly, the necessary data to train and test the models was collected. Finally, the model was iteratively refined while being integrated into the existing codebase for the drone launch project. These phases were not performed sequentially but concurrently.

3.1 Visual sensor

Each launch platform needs a visual sensor for the ANN and a camera for the drone operator. Camera options and some additional sensors were evaluated.

Two camera options evaluated were a monocular and a binocular setup. One camera is cheap and can easily be integrated with the raspberry pi and as a camera will be used by the drone operator, there will be no added cost. But cameras for obstacle detection heavily rely on a good neural network and post-processing to either perceive depth or obstacles.

Multiple cameras can be used to perceive depth, similar to how the eyes work. Using two camera inputs and post-processing, one can create a depth map and collision avoidance software. Efficient collision avoidance methods using depth maps currently exist in cars. A disadvantage of using multiple cameras is that they need to be placed at large distances from each other to perceive longer distances. The safe-to-launch detection needs to be functional up to 75 metres; this leads to both cameras not being able to fit on the launch platform. This could be solved by installing the second camera on its own, but for each new installation, the cameras would need to be recalibrated as the camera distances have to be exact and the camera output must line up exactly.

Instead of using a ML model to perceive depth light detection and ranging (lidar) or radio detection and ranging (RADAR) could be used. Unfortunately, these are expensive and thus can not be installed on all launch platforms. But a laser rangefinder sensor could be a viable option, as this could validate the prediction of the ML model.

In the end, one camera as input to a ML model was decided.

3.2 Monocular depth estimation

The first ML models tested were convolusional neural networks trained to create monocular depth estimation. Models utilised an image as input and created a depth map. Some experimentation was also done with U-net-based models as well as pretrained resnet-based encoder-decoders. Models were evaluated using the metrics described in section 3.2.3, as well as ocular evaluation.

3.2.1 Data Set

The KITTI data set was utilised for monocular depth estimation, which is widely used in the self-driving car industry. The data set consists of stereo images and their corresponding depth maps of cities, highways, and roads. The data was captured using two high-resolution color and gray scale video cameras and a Velodyne laser scanner, which were mounted on a vehicle driven around the city of Karlsruhe. The sparse depth maps created by lidar were preprocessed to full density and sorted into training, evaluation, and test images. The monocular depth estimation data set, which includes 93,000 training and 1,000 evaluation images, as well as 500 test images, was utilised for our work [22]. The KITTI data set does not contain maritime environments, which makes it a suboptimal data set, but it was tested to see if knowledge of depth estimation in maritime environments could be transferable from road imagery.

3.2.2 Model Architecture

Several monocular depth estimation architectures were implemented, beginning with simple convolutional neural networks, which evolved into transfer learning models utilising an already pre-trained Residual Neural Network (ResNet) model. The final architecture was an U-Net model with 42.6 million parameters, which was based on Ibraheem Alhashim's existing architecture [15]. The model consists of a pre-trained resnet model followed by four decoder blocks.

Each decoder block consists of up sampling, concatenating with the skip connections from resnet , a convolutional layer, a leaky ReLU layer, followed by another convolutional layer and a leaky ReLU layer. Ibraheem Alhashim's already trained model was used to see how well the kitti data set transferred to maritime environments.

Hyperparameters	Value
Batch size	32
Epochs	20
Learning rate	0.0002
Decoder depth	4

Table 3.1: Hyperparameters used during training

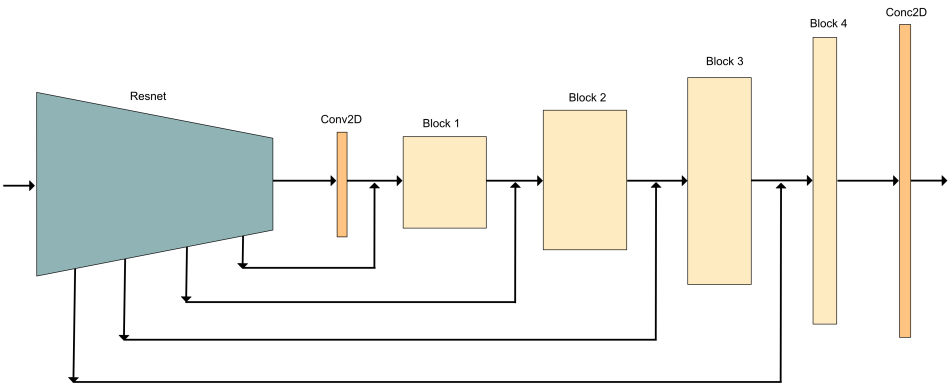

Figure 3.1: Architecture of machine learning model used for depth estimation.

3.2.3 Loss functions

The U-net model was optimised using three different loss functions: mean absolute error, structural similarity index, and gradient edge detection. To create the final loss function, the results from these three loss functions were concatenated in a 1:1:1 ratio. The three loss functions were used as they focus on different image qualities. SSIM measures the similarity between two images. The SSIM index can be viewed as a quality measure. MSE will calculate the mean square error between each pixel for the two images we are comparing, and GED will find and highlight edges.

3.3 Semantic segmentation

Semantic segmentation is a popular object detection method and is widely used in the self-driving car industry by, e.g. Tesla [23] Semantic segmented pictures are able to be annotated by hand. An existing maritime semantically segmented data set existed, which made using a semantic segmentation ANN model an obvious next choice.

3.3.1 Data set

The Marine Semantic Segmentation Training Data set (MaSTr1325) was utilised for our work, which is a large-scale marine semantic segmentation training data set containing 1325 diverse images captured over a two-year period to ensure realistic conditions. The images were handpicked to represent a broad range of marine environments, including different weather conditions and times of day. All images were per-pixel semantically labelled by humans with four categories (sky, sea, obstacle, unknown category), and all annotations were verified and corrected by an expert [4].

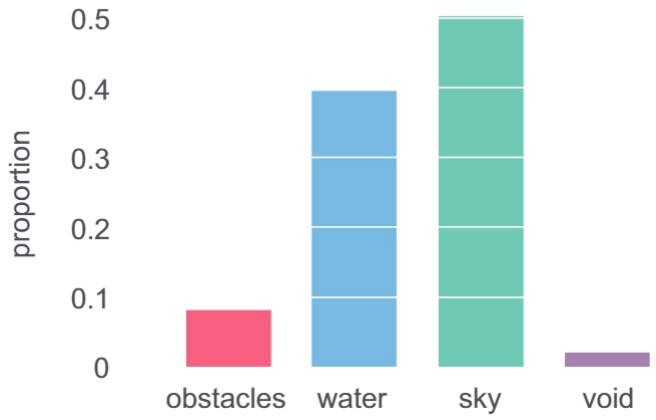

Figure 3.2: Average proportion of annotated pixels for each category [4].

3.3.1.1 Data Collection

The MaSTr1325 data set includes images captured by unmanned surface vehicles (USV) at sea, which varies slightly from our intended use-case where images will be taken from the shore. To enhance the robustness of our model and prevent data drift, we decided to supplement this data set with site-specific images.
These images were captured at the launch site in Långedrag from the shore or pier, facing the sea. The images were taken by two cellphone cameras and were then manually annotated to the standard of MaSTr1325.

3.3.1.2 Data Augmentation

The MaSTr1325 data set is relatively small, containing only 1325 images. To further enhance robustness, we applied data augmentation by randomly applying color-space transformations and rotations to the images, as well as horizontally flipping them. Further details on this process are provided in the appendix A.2.

3.3.1.3 Division of data set

The data set containing MaSTr1325 and the collected data was divided in a 5 to 95, validation-training ratio. The data set was divided before augmentation to avoid similar images existing in the training and validation sets. Both data sets contain images from the MaSTr1325 data set and from the launch site.

3.3.2 Training and evaluation

During the training process, the chosen optimizer was Adam, which utilises a blend of two gradient descent techniques known as "gradient descent with momentum" and the Root Mean Square Propagation (RMSP) algorithms. Adam is recognised for its swift convergence and effectiveness in dealing with noisy and sparse data sets.

Multiple learning rates were tested to get a good result for the model. The loss function used for semantic segmentation was mean square error loss. Both cross-

entropy loss and MSE loss were tested on a U-net model with a learning rate of 0.0001.

3.3.2.1 Evaluation

The performance of the models was evaluated using multiple evaluation functions. MSE loss, mean absolute percentage error (MAPE) loss, and accuracy. Accuracy is calculated by dividing all correctly classified pixels by all pixels in the validation set. The result is expressed as a fraction, where 1 indicates that all pixels are correctly classified.

$$\text{Accuracy} = \frac{\text{Correctly classified pixels}}{\text{Number of pixels}} \tag{3.1}$$

3.3.3 U-net models

Multiple semantic segmentation models were evaluated; these can be divided into two categories: u-net models and convolutional neural networks. U-net was chosen because its skip connections enable detailed spatial information to be used during the upsampling sampling 2.3. Allowing U-Net to effectively capture both local and global information. These properties make U-net a popular CNN that is widely used in segmentation tasks.

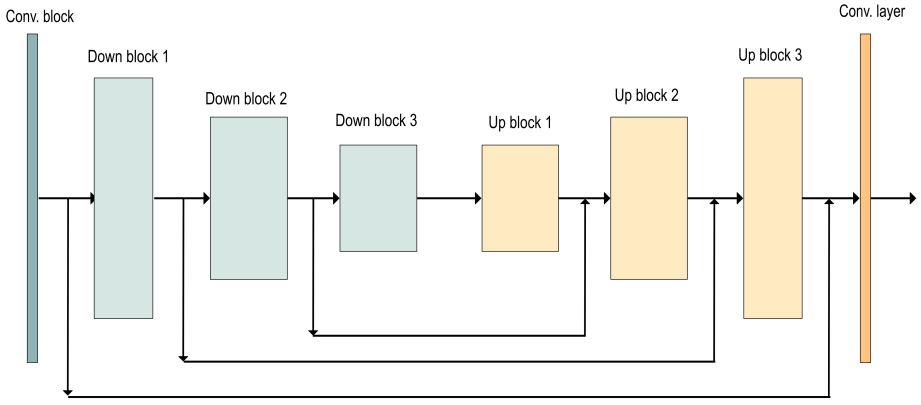

Figure 3.3: Architecture of U-net model used for semantic segmentation.

Each encoder block consists of a max pooling layer, a convolutional layer, and a ReLU function. Each decoder block consists of a bilinear upsampling convolutional layer and a ReLU function.

3.3.4 CNN

Convolutional neural networks were implemented and tested as it is known that they excel in semantic segmentation tasks, their smaller size also enables them to be run on a raspberry pi.

Hyperparameters	Value
Batch size	8
Epochs	6
Learning rate	0.0001
Encoder/Decoder depth	3

Table 3.2: Hyperparameters used during training

Hyperparameters	Value
Batch size	8
Epochs	
Learning rate	0.0001
Encoder/Decoder depth	3

Table 3.3: Hyperparameters used during training

3.3.5 Post processing

The segmented output image from the machine learning model is processed to evaluate if there are any obstacles. This is done in multiple steps. The model segments the input image into four categories: sea, sky, obstacle, and unknown. Pixels categorised as sea and sky do not pose a threat of collision, and they are reclassified as the same category. Pixels classified as unknown are classified as obstacles, creating post-processing that will not launch for pixels classified as unknown. The post processing step is thus more hesitant to launch. Only a small launch window located in the lower centre of the image is evaluated to see if launch is possible. Each pixel classified as an obstacle in the launch window is counted. If the number of pixels exceeds an arbitrary threshold, the post-processing output will be "not safe-to-launch". This approach does, however, introduce another delimitation: we may only use the model on images taken when the camera is pointing towards the horizon. This way, if an obstacle is in contact with the water, its y-position in image space will have a direct relationship to its distance from the camera. We may then say that any obstacles detected in the "launch window" are likely to be too close to the launcher platform.

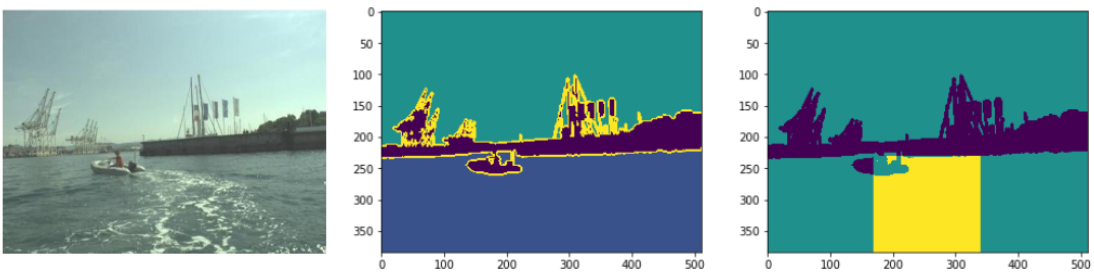

Figure 3.4: Processing steps of the safe-to-launch sequence, yellow in the middle image, are pixels classified as unknown.

We can see in figure 3.4 the input picture, semantic segmentation, and in the third figure the applied post processing. The yellow square is the evaluated "launch window".

3.4 Integration of model

On its own, the neural network model is not immediately useful. To utilise it efficiently, it should be fully integrated into the drone launcher software and be made available to make predictions through an API. Software is needed that communicates with the hardware of the launcher, specifically the camera, and then puts that data through the model and finally routes it to the end user.
The operator communicates with the launcher using a browser application. Through this application, you make calls to the safe-to-launch model API, and you get back necessary information such as the image from the camera, a segmented image, and a predicted safe-to-launch command.
Besides having the desired functionality, the software should adhere to the same linting and testing standards as the rest of the project code.

3.4.1 CameraMLmodel class

Following the object oriented coding paradigm used within the project, it was decided that the neural network model and all its external utilities should be part of a self-contained class. This class should contain methods to connect to the camera hardware, make predictions, create segmented images, and so on.
This class is then a part of a larger hierarchical structure, visualised in figure 3.5.

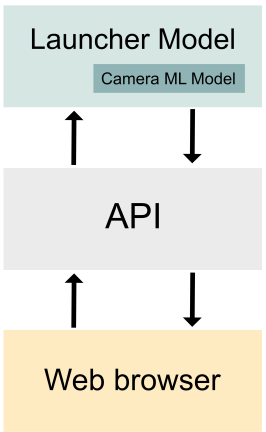

Figure 3.5: Hierarchical structure of drone launcher.

3.4.2 API

The API allows the end user to send and receive data from CameraMLModel; it uses a Python library called Flask to send this data over a local network, and this is how the drone launcher software communicates with the web application [24]. The way this works is that you define a function using the def statement like you normally would in Python, then you add the decorator @app.route(path), where path is the uniform resource locator (URL) you want to map your function to. Now whenever your browser sends a request to the URL defined by path, the decorated function is executed and its return value is sent through a socket to the user.

3.4.3 Automatic testing

In accordance with standard devops protocol, every piece of code submitted to the repository must adhere to a certain standard when it comes to linting and test coverage. For automatic testing, Infotiv uses Jenkins to run all tests and create reports every time code is pushed to GitLab. This is very useful to make sure that new commits do not break the software or cause it to behave in unintended ways, and eliminating the need for manual testing can significantly speed up the development process. There are several different types of tests, but in this project, mainly the standard unit tests were considered.

3.4.3.1 Unittests

Using the Python package unittest, one can define tests for any piece of code using assert statements [25]. For CameraMLModel, every method is covered by a test case, ensuring that everything works as intended.

3.4.3.2 Prediction test

This particular test was designed to have Jenkins automatically evaluate the performance of a new ANN model in case it needs to be swapped out, retrained, etc. A small set of test images is fed to the model, and the outputs are compared to the labels to make sure the new model does not perform worse than the old one. To make this test as valuable as possible, the test images are all from the launch site in Långedrag where this system might be deployed in the future.

3.4.4 Training notebook

To quickly iterate on solutions and try out different model architectures, a training "pipline" was setup. This pipline is also intended to be used by possible future thesis project groups at Infotiv. This consists of a Jupyter notebook containing everything needed to define, train, evaluate, quantize, and save ANN models. This notebook provided the basis for most of the experimentation done during this project. This entire notebook can be found in appendix A.3.

3.4.5 Augmentation notebook

Similar to the training notebook, this file allows the user to quickly try out different types of data augmentation, as described in Section 3.3.1.2. This notebook can be found in appendix A.2.

4

Results

4.1 Evaluation of sensors

For the scope of this project, it was decided that a single camera would be the best sensor option because it is cheap and also very flexible: many different machine learning models for detecting obstacles can work with only an image as input. This will also allow the functionality of the detection system to be extended in the future simply by adding more software. Other options evaluated but decided against were, lidar, radar, laser rangefinders, and ultrasound rangefinders. More elaborate motivations for why these sensors were not used follow below.

4.1.1 Lidar and Radar

Both of these options are prohibitively expensive and come with some additional issues. Radar has a limited spatial resolution imposed on it by the wavelength of the EM-waves it uses [26]. Because of this, smaller obstacles like buoys and ropes would not be detected reliably. Lidar, on the other hand, can potentially be more accurate, but the point-cloud data it produces would be harder to analyse and likely demand more computing power than the raspberry pi can offer.

4.1.2 Laser range finder

A laser range finder could potentially be useful for determining the distance to an obstacle detected by the proposed segmentation algorithm. It could also be used to verify a prediction from a depth estimation model. The downside, however, is that it can only sample one point at a time, which is not good enough to detect all possible obstacles in the flight path of the drone, and by the time enough points have been sampled, obstacles may have moved and the produced point cloud might be an inaccurate representation of reality.

4.1.3 Ultrasound range finder

Ultrasound rangefinders do not provide sufficient range to meet the demands of this project. The cheap ultrasound rangefinders have a range of 1 to 10 metres. Ranges of 75 metres exist, but they are increasingly expensive and not affordable for this project [27].

4.2 Monocular depth estimation

As no data existed for maritime environments for depth estimation, all evaluations had to be done by comparing the model output with an estimated truth. This was done by estemating the true depth of the input picture with the model output by the authors of this thesis.

Monocular depth estimation using Ibraheem Alhashim's model yields semi-satisfactory results; the figure 4.1 shows the depth field of the monocular depth estimation model and how well it distinguishes details. The model shows no difficulty understanding water surfaces. However, it has difficulties understanding reflective surfaces, as shown in the picture with the boat and the fence, where the model believes that the reflection is further away than the actual object.

There are also difficulties distinguishing near objects from the water surface; this can be seen in the image of the boat, where the bow is nearly indistinguishable from the water surface. This is not true for the model when it processes images with cars, as we can see in the first image, where the car in the lower right corner is well distinguishable.

The sky is coloured a dark blue in 5 out of 6 images, indicating that the model believes that the sky is one of the closest objects. Only in the image over open water without obstacles does the model indicate that the sky is far away, but it still believes that the top part of the sky is a close obstacle.

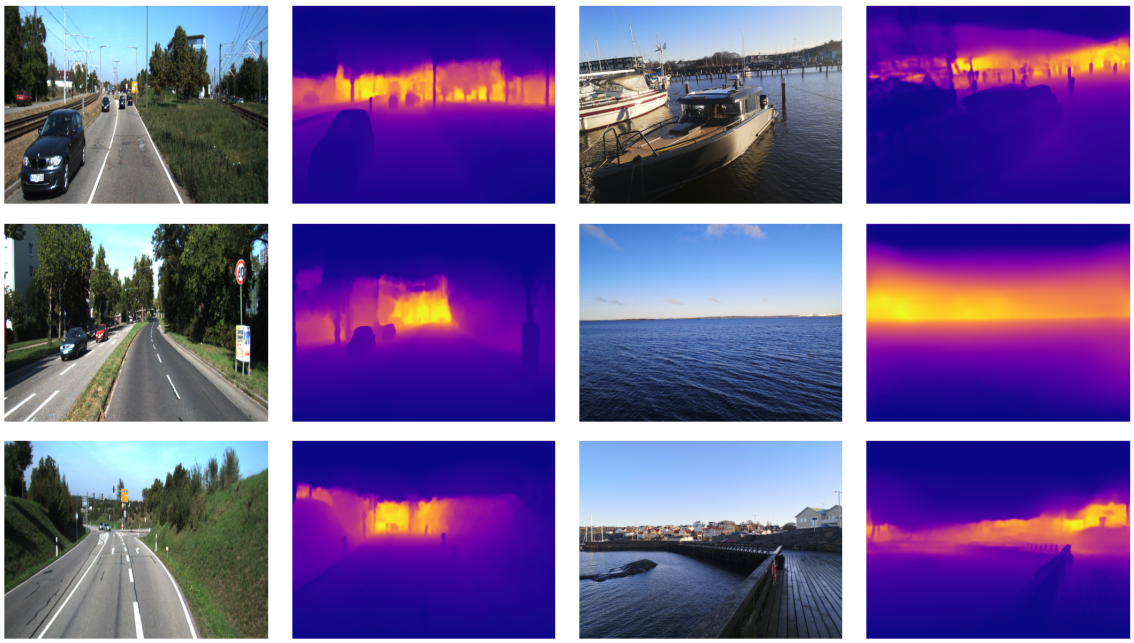

Figure 4.1: Images and their corresponding predicted depth estimation using Ibraheem Alhashim's existing model. Purple indicates close objects and yellow, objects far away.

Table 4.1 shows multiple evaluation parameters and their results. As we can see from the evaluation parameters the model is very accurate in recreating a depth map of road imagery.

Data set	MAPE	RMSE	squared MAPE	RMSE log
Kitti	5.8%	2.360	8.8%	0.968

Table 4.1: Evaluation parameters for Ibraheem Alhashim's existing model.

A smaller monocular depth estimation model containing a resnet 50 encoder with 1 upscale and 2 transposed convulsional layers results in models that return gibberish. In figure 4.2 one can see the top 4 pictures that are the model input the 4 middle pictures the desired output and the bottom 4 the actual output of the model.

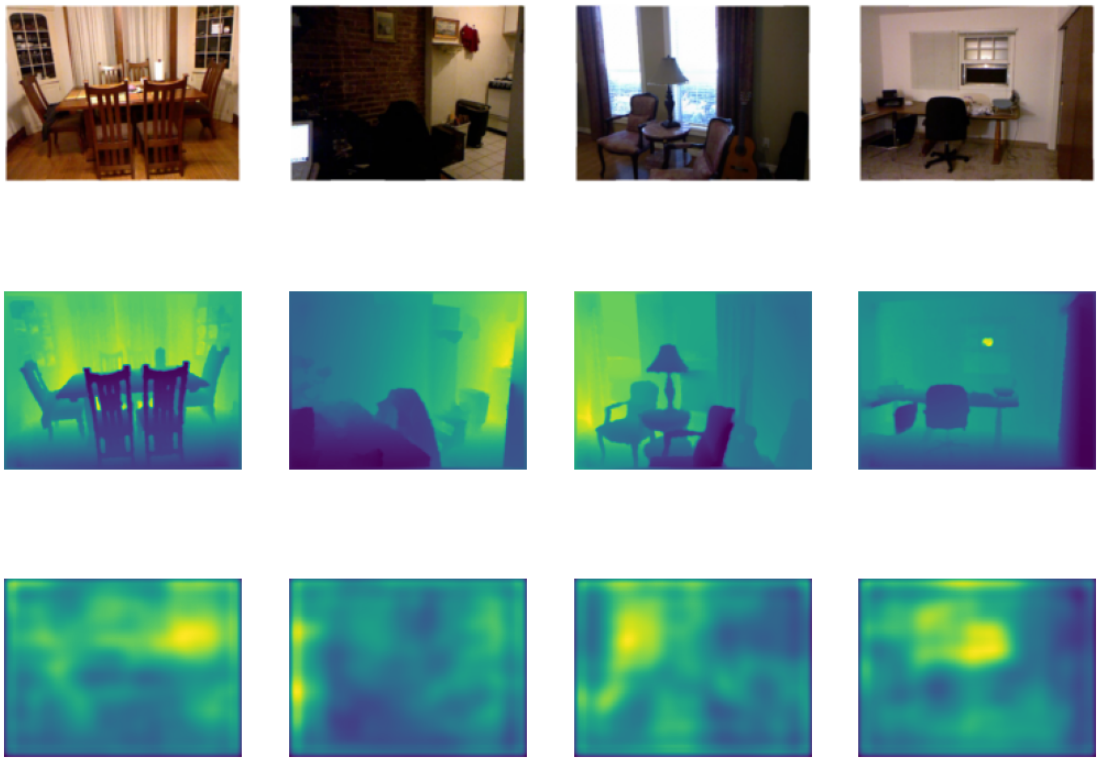

Figure 4.2: Images and their corresponding predicted depth estimation using small monocular depth estimation model. Dark blue indicates close objects and yellow, objects far away.

From figure 4.3 we can see that the small depth estimation model is fully trained and thus wont improve any more.

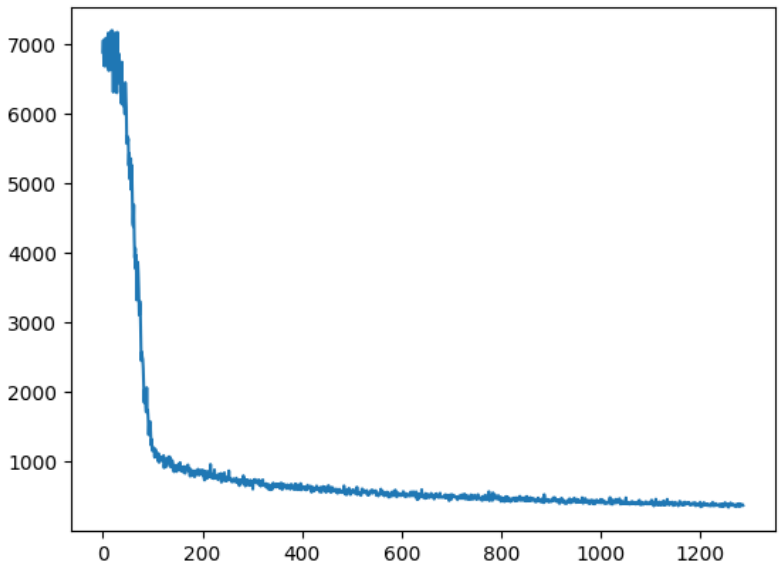

Figure 4.3: Training loss of small monocular depth estimation model.

4.3 Semantic segmentation

4.3.1 UNet

In the end, U-net models were not able to be quantified due to their layer types and skip connections. This, combined with all u-net models being too large to run on the raspberry pi with an acceptable inference time, made them unusable for safe-to-launch detection. The U-net model has 7.7 million trainable parameters.

Model	MSE	MAE	Accuracy
u-net	7.234	2.455	0.93

Table 4.2: U-net evaluation metrics.

4.3.2 Training and evaluation

Results of training with different learning rates are shown in figure 4.4 the graph visualises loss for each batch during training of a u-net model. Multiple experiments with different learning rates are shown, each with its own color. Eight batches at a time were normalised and plotted together to simplify the figure. We can see that the learning rates 10^{-6} and lower take longer to converge and also converge to a higher loss. For learning rates above 0.0001, the loss for batches oscillated and could not converge to a small minimum. The optimal learning rate for the u-net model was found to be around 0.0001.

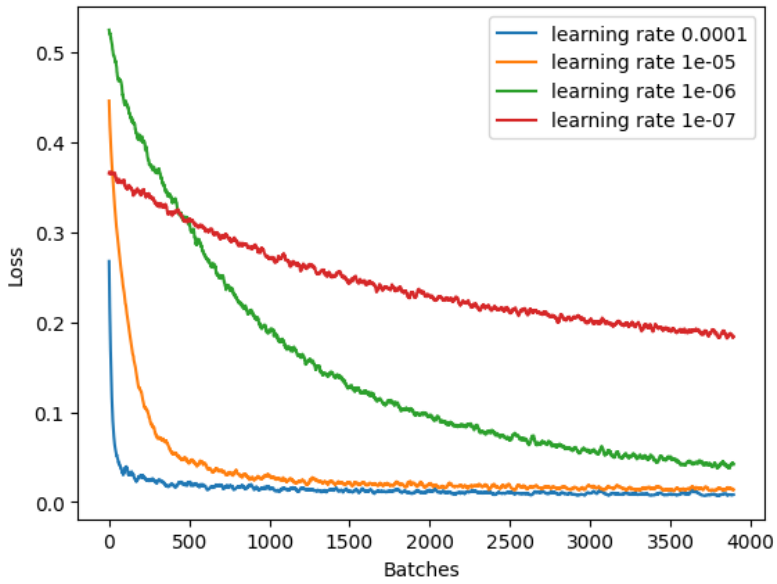

Figure 4.4: Training of U-net model with different learning rate.

Comparing cross-entropy loss with MSE loss we find that MSE loss yields better results as it converges faster, leading to faster training of an ANN model. This can be seen in figure 4.5.

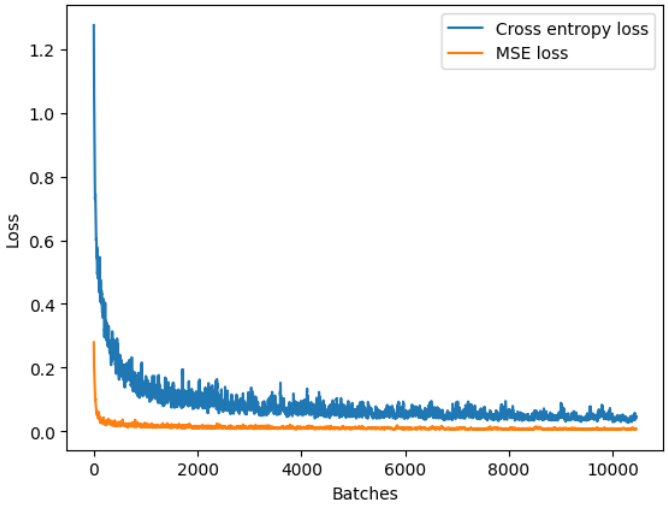

Figure 4.5: Training of U-net model with Cross entropy loss and MSE loss.

4.3.3 Autoencoder models

Having concluded that a conventional autoencoder model might be a good choice, some experimentation was done to determine how depth and kernel size might impact performance. In figure 4.8 the training and validation losses for six different models are compared. These models differ only in depth and kernel size; their exact specifications are shown in table 4.6. Apart from MSE-loss, a selection of some other performance metrics were used, intersection over union as well as L1 distance, visible in table 4.4. The hypothesis was that a large receptive field would be required to accurately classify any given pixel, and thus a larger kernel size might increase performance. Making the network deeper might also help it store more valuable information, but it comes at a resolution cost, more down sampling in the encoder will result in a loss of detail in the segments produced.

block	kernels
down	2*n
down	4*n
down	6*n
down	8*n
up	8*n
up	6*n
up	4*n
up	2*n

down
maxpool2d
conv2d
ReLU

up
upsample2d
conv2d
ReLU

Figure 4.6: Autoencoder architectuere.

Another similar experiment was done to compare validation loss to the number of

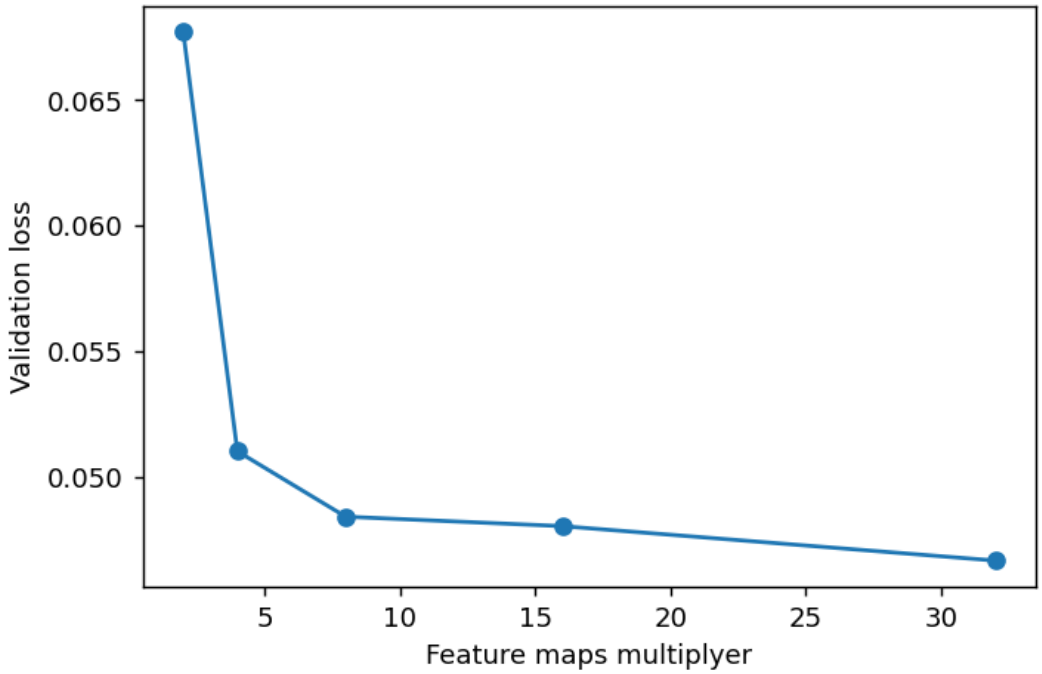

Figure 4.7: How the number of feature maps impacts performance.

kernels used in each convolutional layer: first a model with a very small number of kernels was created, then this number was multiplied by a range of values, and the model was trained and evaluated for each. The results of this experiment, shown in figure 4.7, tell us that increasing the number of kernels decreases validation loss as expected, but that a point of diminishing returns is quickly reached at around a multiplier of 16.

For the sake of convenience, both experiments described above use the same basic autoencoder architecture, described in table 4.6. Here n is the multiplier for the number of kernels in the convolution layer of each block. Maxpool2d and upsample2d-layers halves and doubles image resolution, respectively. For the experiments with depth 3, the last layer of the encoder and the first layer of the decoder were removed.

Model	Depth	Kernel size
a	3	3
b	3	5
c	3	7
d	4	3
e	4	5
f	4	7

Table 4.3: Autoencoder specifications.

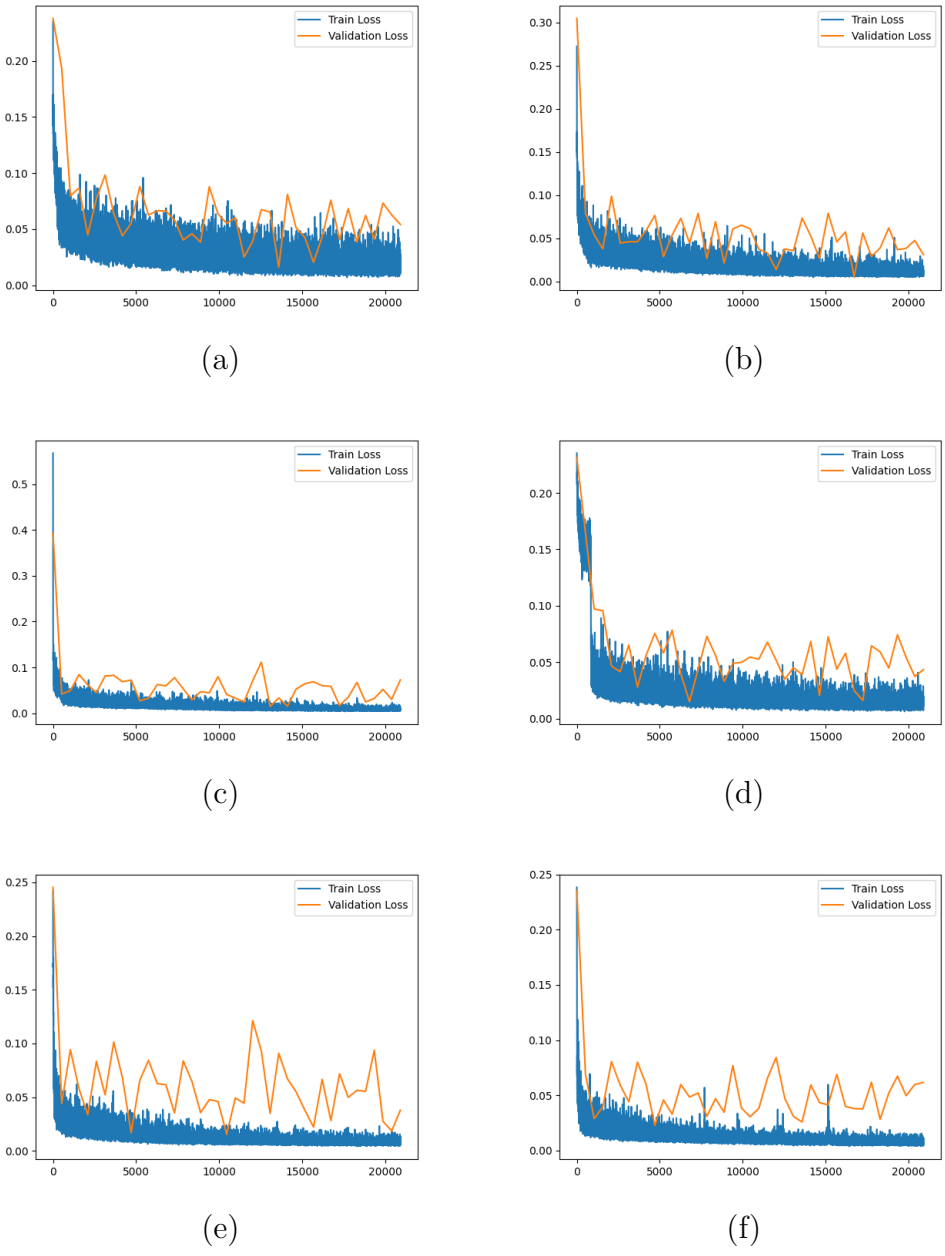

Figure 4.8: Comparison of different depths and kernel sizes.

Model	MSE	MAE	Accuracy
a	0.0168	0.1004	0.925
b	0.0091	0.1009	0.9269
c	0.0066	0.0576	0.9282
d	0.0113	0.0995	0.9345
e	0.0084	0.0995	0.9330
f	0.0077	0.0899	0.9302

Table 4.4: Evaluation metrics.

Given the results described, it was decided that model type f with a kernel multiplier of 16 was the best compromise between speed and performance. This was the model selected to be integrated with the launcher software. The code for this autoencoder class can be found in Appendix A.1.

5
Conclusion

Monocular depth estimation was not successfully implemented as the main method for safe-to-launch prediction due to multiple problems. Firstly, the available data sets mainly consist of images from roads, which transfer poorly to maritime environments. Secondly, for the large depth estimation model objects close to the camera, the depth estimations are nearly indistinguishable from the foreground, and lastly, the monocular depth estimation network with its 42.6 million parameters can't be run on the raspberry pi in under 1 second. Using smaller models for monocular depth estimation was not possible; smaller models did not learn the monocular cues to perceive depth and interpreted light or dark areas and edges as depth.

In light of these insights, it was concluded that using monocular depth estimation on a raspberry pi as a safe-to-launch system was not feasible. On the other hand, semantic segmentation seems to be viable even with smaller neural networks. Even though semantic segmentation can only be used to launch towards the sea, it is reliable as an extra safety measure for the drone operator.

The final conclusion of this project is that depth estimation models are too complicated and cumbersome to train for this particular use case, and semantic segmentation is the better option.

It was also concluded that the semantic segmentation system in its current state of only being able to launch towards the sea is not advanced enough to operate completely autonomously, but it is useful for the drone operator as an extra layer of safety. Ideally, this system should be expanded upon in the future for more reliability and redundancy.

5.1 Future work

A functional safe-to-launch system has successfully been implemented and integrated into the existing codebase. A lot of time has been spent integrating the machine learning model, but there are several areas that can be improved and worked upon.

5.1.1 Further data collection

More data is almost always better. The data collection done in this project was on a very small scale and mostly intended to supplement the data found online. Because

of time constraints, only a small number of photos were annotated and perhaps in the future one might want to collect and annotate more to further increase the accuracy of models. Especially expand the data collected in different weather conditions.

5.1.2 Further model optimizations

A lot of time was spent finding a good model type (Resnet, Unet, autoencoders, etc. were considered) before an autoencoder was decided on. Some optimisation of this model with respect to depth, kernel size, and number of kernels was done; however, some questions remain unanswered, for instance: Does having different kernel sizes in different layers make a difference? Do different optimisation algorithms converge to different solutions? Does pruning or weight decay make the model better at generalising? Are there better methods of augmenting the training data?

5.1.3 Calibration of post processing

The size of the "launch window" was decided arbitrarily. In the future, it should be determined more carefully what the optimal parameters are, especially the dimensions of the launch window.

5.1.4 Ensemble of models

Although the solution using semantic segmentation was concluded to be viable, it has its limitations. For example, the model's inability to perceive depth puts restrictions on its use. The camera angle at inference needs to be parallel to the ground so that the distance of an object from the camera can be inferred geometrically, as explained in section 3.3.5 of this thesis. Modern self driving systems often use an ensemble of different machine-learning models so that they may cover each other's weaknesses. We propose that the safe-to-launch system be supplemented with either an object detection or a crude depth estimation model for extra redundancy.

5.1.5 Additional sensors

The only sensor currently used by the drone launcher is a surveillance camera. Some other types of sensors were considered, but it was ultimately decided that adding more at this stage would either complicate the project too much or be of insignificant use. But in the future, one might want to add more sensors to the platform to extend its functionality. A laser range finder for example could have some limited use.

5.1.6 Launch towards land

Our model can only launch towards the sea or sky (it will not go if launched towards land). The reason for this is again the limitations of the semantic segmentation approach, and the only way to remedy this shortcoming is to somehow allow the system to perceive depth, either by adding more software or sensors, as discussed above.

Bibliography

[1] Hollenstein Lukas et al. *Unsupervised Learning and Simulation for Complexity Management in Business Operations*, pages 313–331. 06 2019.

[2] Heet Sankesara. *UNet Introducing Symmetry in Segmentation.* 2019. https://towardsdatascience.com/u-net-b229b32b4a71.

[3] Miangoleh S. Mahdi et al. *Boosting Monocular Depth Estimation Models to High-Resolution via Content-Adaptive Multi-Resolution Merging.* 2021. https://arxiv.org/abs/2105.14021.

[4] Bovcon Borja et al. The mastr1325 dataset for training deep usv obstacle detection models. In *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2019.

[5] sjöräddningssällskapet. *Drönare*, 2023. https://www.sjoraddning.se/forskning-och-utveckling /dronare.

[6] Ganesh A. McCallum A Strubell, E. *Energy and Policy Considerations for Deep Learning in NLP.* 2019. http://arxiv.org/abs/1906.02243.

[7] Rachel Draelos. *The History of Convolutional Neural Networks.* April 2019. https://glassboxmedicine.com/2019/04/13/a-short-history-of-convolutional-neural-networks/.

[8] Jian Ji and et al Lu. Parallel fully convolutional network for semantic segmentation. *IEEE Access*, 9:673–682, 2021.

[9] Hossein Gholamalinezhad and Hossein Khosravi. Pooling methods in deep neural networks, a review. *CoRR*, abs/2009.07485, 2020.

[10] Abien Fred Agarap. Deep learning using rectified linear units (relu). *CoRR*, abs/1803.08375, 2018.

[11] Tuan Pham. Semantic road segmentation using deep learning. In *2020 Applying New Technology in Green Buildings (ATiGB)*, pages 45–48, 2021. doi: 10.1109/ATiGB50996.2021.9423307.

[12] Olaf et al Ronneberger. U-net: Convolutional networks for biomedical image segmentation. In *Medical Image Computing and Computer-Assisted Intervention – MICCAI 2015*, pages 234–241, Cham, 2015. Springer International Publishing.

[13] Padhma M. *End-to-End Introduction to Evaluating Regression Models.* 2018. https://www.analyticsvidhya.com/blog/2021/10 /evaluation-metric-for-regression-models/.

[14] Pranjal Datta. *All about Structural Similarity Index (SSIM): Theory + Code in PyTorchg.* 2020. https://medium.com/srm-mic/all-about-structural-similarity-index-ssim-theory-code-in-pytorch-6551b455541e.

[15] Ibraheem Alhashim and Peter Wonka. High quality monocular depth estimation via transfer learning. *arXiv e-prints*, abs/1812.11941, 2018.

[16] Lore Kin Gwn et al. Generative adversarial networks for depth map estimation from rgb video. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, pages 1258–1288, 2018.

[17] Daryl Tan. *Depth Estimation: Basics and Intuition, Towards data science*, 2023. https://towardsdatascience.com/depth-estimation-1-basics-and-intuition-86f2c9538cd1.

[18] Takanori Okoshi. *Three-dimensional imaging techniques*. Academic Press, 2012. p. 387.

[19] Luu C Kalloniatis M. *The Perception of Depth, The Organization of the Retina and Visual System*. 2005. https://www.ncbi.nlm.nih.gov/books/NBK11512/.

[20] Raspberry pi. https://www.raspberrypi.com/products/raspberry-pi-3-model-b-plus/.

[21] Schmid philipp, *Static Quantization with Hugging Face 'optimum' for 3x latency improvements*, June, 2022. https://www.philschmid.de/static-quantization-optimum.

[22] Andreas Geiger, Philip Lenz, Christoph Stiller, and Raquel Urtasun. Vision meets robotics: The kitti dataset. *International Journal of Robotics Research (IJRR)*, 2013.

[23] tesla. *AI Robotics*. https://www.tesla.com/AI.

[24] pansaripulkit13. *Flask App Routing*. https://www.geeksforgeeks.org/flask-app-routing/.

[25] python. *unittest documentation*. https://docs.python.org/ 3/ library/ unittest.html.

[26] Nationalencyklopedin, *Radar*. http://www.ne.se/uppslagsverk/encyklopedi/lång/radar.

[27] Massa Donald P. *Choosing an Ultrasonic Sensor for Proximity or Distance Measurement Part 1: Acoustic Considerations*, 2023. https://www. fierceelectronics.com/ components/ choosing-ultrasonic-sensor-for- proximity-or-distance-measurement-part-1-acoustic.

A

Appendix 1

Here all code developed during the project is listed.

A.1 Autoencoder template

This is the template for the autoencoder architecture used for the final model. There are two main versions, one with four blocks of convolution for the encoder and decoder respectively, and one with three blocks. Kernel size can be set when initializing the object, but to change number of kernels (or feature maps) one must edit the class definition.

```
class Down(nn.Module):
    """
    Halves input resolution and preforms convolution and relu-operation
    """
    def __init__(self, in_channels, out_channels, kernel_size):
        super(Down, self).__init__()
        self.pool = nn.MaxPool2d(2)
        self.conv = nn.Conv2d(in_channels, out_channels, kernel_size, padding=(kernel_size-1)//2)
        self.relu = nn.ReLU()

    def forward(self, x):
        return self.relu(self.conv(self.pool(x)))

class Up(nn.Module):
    """
    Doubles input resolution and preforms convolution and relu-operation
    """
    def __init__(self, in_channels, out_channels, kernel_size):
        super(Up, self).__init__()
        self.upsample = nn.UpsamplingBilinear2d(scale_factor=2)
        self.conv = nn.Conv2d(in_channels, out_channels, kernel_size, padding=(kernel_size-1)//2)
        self.relu = nn.ReLU()

    def forward(self, x):
        return self.relu(self.conv(self.upsample(x)))

class Autoencoder_d3(nn.Module):
    """
    Autoencoder model consisting of two modules: encoder and decoder,
    each of wich is made up of 3 blocks of either down- or upsampling followed by conv2d and relu.
    """
    def __init__(self, kernel_size):
        super(Autoencoder_d3, self).__init__()
        self.quant = torch.quantization.QuantStub()
        self.encoder = nn.Sequential(
            Down(3, 32, kernel_size),
            Down(32, 64, kernel_size),
            Down(64, 96, kernel_size),
        )
        self.decoder = nn.Sequential(
            Up(96, 64, kernel_size),
            Up(64, 32, kernel_size),
            Up(32, 4, kernel_size)
        )
        self.dequant = torch.quantization.DeQuantStub()


    def forward(self, x):
        x = self.quant(x)
        x = self.encoder(x)
        x = self.decoder(x)
        x = self.dequant(x)
        return x

class Autoencoder_d4(nn.Module):
    """
    Autoencoder model consisting of two modules: encoder and decoder,
    each of wich is made up of 4 blocks of either down- or upsampling followed by conv2d and relu.
    """
    def __init__(self, kernel_size):
        super(Autoencoder_d4, self).__init__()
        self.quant = torch.quantization.QuantStub()
        self.encoder = nn.Sequential(
            Down(3, 32, kernel_size),
            Down(32, 48, kernel_size),
            Down(48, 64, kernel_size),
            Down(64, 128, kernel_size)
        )
        self.decoder = nn.Sequential(
            Up(128, 64, kernel_size),
            Up(64, 48, kernel_size),
            Up(48, 32, kernel_size),
            Up(32, 4, kernel_size)
        )
        self.dequant = torch.quantization.DeQuantStub()
```

```
def forward(self, x):
    x = self.quant(x)
    x = self.encoder(x)
    x = self.decoder(x)
    x = self.dequant(x)
    return x
```

A.2 Data augmentation notebook

INFO

The purpose of this notebook is to take two datasets (mastr1325 and our own ssrs_dataset), augment and combine them and save the images and targets in a new folder.

In order for this to work, the input datasets have to follow certain naming and organization conventions: The mastr1325 dataset should be divided into two folders, one with input images and one with targets, these files should be namend in such a way that when the two folders are sorted, images and corresponding targets are inte the same order (the mastr dataset follows this convention by default, do not rename files after downloading). Download here: https://www.vicos.si/resources/mastr1325/ For the ssrs dataset, the input and target images should be in the same folder and have the same names, only file extension should be different. For inputs the extension should be ".jpg" or ".JPG", and for targets it should be ".png". This dataset can be found here: infotiv sharepoint

The format for the targets are different for mastr1325 and ssrs_dataset: mastr targets are single-channel uint8, where the value of each pixel corresponds to a certain class. In ssrs_dataset, the targets are RGB images where each channel corresponds to a different class. The function "RGB_to_categorical" converts to the desiered format.

This augmentation script will output the targets in a common format similar to that of mastr1325, but wheras mastr1325 targets have 4 classes, our output will have 3: "sky", "water", "obstruction". The forth class "uncertain" is reassigned as "obstruction".

Under the tab "Config" parameters for the augmentation can be changed.

Run the entire notbook to output augmented data into "OUTPUT_PATH".

Config

```
# Paths to images and targets and the path for the images to be output
into
MASTR_IMAGE_PATH =
"/home/lab/Desktop/dataset/mastr1325/MaSTr1325_images_512x384"
MASTR_TARGET_PATH =
"/home/lab/Desktop/dataset/mastr1325/MaSTr1325_masks_512x384"
SSRS_PATH = "/home/lab/Documents/langedrag_dataset"
OUTPUT_PATH = "/home/lab/Documents/augmented_dataset"

# Number of augmented samples from each dataset
MASTR_SAMPLES = 2
SSRS_SAMPLES = 10

# Augmentation settings
FLIP_IMAGE = True
HUE_VAR = 10
SATURATION_VAR = 50
```

```
VALUE_VAR = 60
NOISE = 0.2
```

Imports / Helpers

```
import os
import cv2
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

def make_df_mastr(x_path, t_path):
    """Takes two folders full of images and returns a dataframe with
filenames.
    x_path is the path to the input images and t_path is for
targets"""
    x_names = sorted(os.listdir(x_path))
    y_names = sorted(os.listdir(t_path))
    df = pd.DataFrame({"x": x_names, "t": y_names})
    return df

def make_df_ssrs(path):
    """Takes a folder full of images and returns a dataframe with
filenames.
    Images ending with .jpg or .JPG are assumed to be inputs, and
images
    ending with .png are assumed to be targets"""
    names =  sorted(os.listdir(path))
    x_names = []
    y_names = []
    for n in names:
        if n.split(".")[1]=="jpg" or n.split(".")[1]=="JPG":
            x_names.append(n)
        else:
            y_names.append(n)
    df = pd.DataFrame({"x": x_names, "t": y_names})
    return df

def change_hsv(image, DH, DS, DV):
    """Randomly change hue, saturation and value for an image.
    Input and output is image in RGB format (3d-array)"""
    dh = int((np.random.random()-0.5)*2*DH)
    ds = int((np.random.random()-0.5)*2*DS)
    dv = int((np.random.random()-0.5)*2*DV)
    image = cv2.cvtColor(image, cv2.COLOR_RGB2HSV)
    h, s, v = cv2.split(image)
    h_new = np.mod(h + dh, 179).astype(np.uint8)
    s_new = s.astype(np.int16) + ds
    v_new = v.astype(np.int16) + dv
    s_over = np.where(s_new > 255)
    s_under = np.where(s_new < 0)
    v_over = np.where(v_new > 255)
```

```
    v_under = np.where(v_new < 0)
    s_new[s_over] = 255
    s_new[s_under] = 0
    v_new[v_over] = 255
    v_new[v_under] = 0
    result = cv2.merge([h_new, s_new.astype(np.uint8),
v_new.astype(np.uint8)])
    result = cv2.cvtColor(result, cv2.COLOR_HSV2RGB)
    return result

def apply_noise(img, noise):
    row,col,ch= img.shape
    gauss = np.random.normal(0,noise,(row,col,ch))
    return img + gauss

def RGB_to_categorical(img):
    """Convert segmentation target from RGB to
    one-channel uint categorical.
    Input should be 4-channel uint8."""
    img = img/255
    _, g, b = cv2.split(img)
    b = b.round()
    g = g.round()
    result = b*2 + g
    return result

def crop_image(image, slice):
    """Crops an image to region specified by slice"""
    output_image = image[slice[0]:slice[1], slice[2]:slice[3], :]
    return output_image

def augment_images(df_ssrs, df_mastr, output_path):
    """Augments images specified by df_ssrs and df_mastr
    and puts them in output_path. Requires some global vars to be
defined"""
    try:
        os.mkdir(output_path)
    except:
        pass
    os.chdir(output_path)
    # Mastr images
    for i in range(len(df_mastr)):
        print(f"Augmenting {i}/{len(df_mastr)} mastr images")
        img_raw = cv2.imread(MASTR_IMAGE_PATH+"/"+df_mastr.iloc[i]
["x"], cv2.IMREAD_UNCHANGED)
        img_rgb = cv2.cvtColor(img_raw, cv2.COLOR_BGR2RGB)
        target = cv2.imread(MASTR_TARGET_PATH+"/"+df_mastr.iloc[i]
["t"], cv2.IMREAD_UNCHANGED)
        for j in range(MASTR_SAMPLES):
            augmented_img = change_hsv(img_rgb, HUE_VAR,
SATURATION_VAR, VALUE_VAR)
```

```
            augmented_img = apply_noise(augmented_img,
NOISE*np.random.random())
            cv2.imwrite(f"x_0{(i*j)+j}.jpg", augmented_img)
            cv2.imwrite(f"y_0{(i*j)+j}.png", target)
            if FLIP_IMAGE:
                flipped_img =  cv2.flip(augmented_img, 1)
                flipped_target = cv2.flip(target, 1)
                cv2.imwrite(f"x_1{(i*j)+j}.jpg", flipped_img)
                cv2.imwrite(f"y_1{(i*j)+j}.png", flipped_target)
    # SSRS images
    for i in range(len(df_ssrs)):
        print(f"Augmenting {i}/{len(df_ssrs)} ssrs images")
        img_rgb = cv2.imread(SSRS_PATH+"/"+df_ssrs.iloc[i]["x"],
cv2.IMREAD_UNCHANGED)
        target_raw = cv2.imread(SSRS_PATH+"/"+df_ssrs.iloc[i]["t"],
cv2.IMREAD_UNCHANGED)
        target_rgb = cv2.cvtColor(target_raw, cv2.COLOR_BGR2RGB)
        for j in range(SSRS_SAMPLES):
            img_shape = img_rgb.shape
            dx = int(img_shape[0]/2)
            x = int(dx*np.random.random())
            dy = int(img_shape[1]/2)
            y = int(dx*np.random.random())
            slice = [x, x+dx, y, y+dy]
            augmented_img = change_hsv(img_rgb, HUE_VAR,
SATURATION_VAR, VALUE_VAR)
            augmented_img = apply_noise(augmented_img,
NOISE*np.random.random())
            cropped_img = crop_image(augmented_img, slice)
            cropped_target = crop_image(target_rgb, slice)
            target = RGB_to_categorical(cropped_target)
            cv2.imwrite(f"x_2{(i*j)+j}.jpg", cropped_img)
            cv2.imwrite(f"y_2{(i*j)+j}.png", target)
            if FLIP_IMAGE:
                flipped_img =  cv2.flip(cropped_img, 1)
                flipped_target = cv2.flip(target, 1)
                cv2.imwrite(f"x_3{(i*j)+j}.jpg", flipped_img)
                cv2.imwrite(f"y_3{(i*j)+j}.png", flipped_target)
```

Augmentation

```
mastr_df = make_df_mastr(MASTR_IMAGE_PATH, MASTR_TARGET_PATH)
ssrs_df = make_df_ssrs(SSRS_PATH)
augment_images(ssrs_df, mastr_df, OUTPUT_PATH)
```

A.3 Model training notebook

Train and quantize an autoencoder model

This notebook contains everything needed to create, train, quantize and pickle an autoencoder nn-module.

At the bottom of the page is a section for automating this entire process.

Dependencies

```
import numpy as np
import matplotlib.pyplot as plt
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
from PIL import Image
import os
from tqdm import tqdm
import pandas as pd
import copy
import dill
```

Helpers

```
def one_hot_encoding(tensor):
    """
    Does one-hot encoding in accordance with
    class convention of MasTr1325 dataset.
    Requires gobal var "device" (torch.device)
    """
    encoded = torch.zeros(tensor.expand(-1,4,-1,-1).shape)
    encoded[:,0,:,:] = tensor[:,0,:,:] == 0
    encoded[:,1,:,:] = tensor[:,0,:,:] == 1
    encoded[:,2,:,:] = tensor[:,0,:,:] == 2
    encoded[:,3,:,:] = tensor[:,0,:,:] == 4
    return encoded.to(device)

class DataLoader():
  """
  Fetches batches of data for training or validation
  """
  def __init__(self, dir, batch_size=32, device ='cpu'):
    self.path_imgs = dir
    self.device = torch.device(device)
    self.batch = 0
```

```
    self.batch_size = batch_size
    self.x_names = []
    self.y_names = []
    names = os.listdir(self.path_imgs)
    for name in names:
      if name.split("_")[0]=="x":
        self.x_names.append(name)
      else:
        self.y_names.append(name)
    self.x_names = sorted(self.x_names)
    self.y_names = sorted(self.y_names)
    self.length = len(self.x_names)
    self.idx = np.random.permutation(self.length)
    self.transform = transforms.Compose([transforms.Resize((256,
256)), transforms.PILToTensor()])

  def reset_epoch(self):
    self.idx = np.random.permutation(self.length)
    self.batch = 0

  def __call__(self):
    file_idx = self.idx[(self.batch*self.batch_size):
((self.batch+1)*self.batch_size)]
    x, y = [], []
    for i in file_idx:
      x_path = self.x_names[i]
      y_path = self.y_names[i]
      x_img = Image.open(self.path_imgs+"/"+x_path)
      y_img = Image.open(self.path_imgs+"/"+y_path)
      x_tensor = self.transform(x_img)
      y_tensor = self.transform(y_img)
      x.append(x_tensor)
      y.append(y_tensor)
    self.batch += 1
    if self.batch*self.batch_size > self.length:
      self.reset_epoch()
    return torch.stack(x).to(self.device, dtype=torch.float32),
one_hot_encoding(torch.stack(y).to(self.device, dtype=torch.uint8))

def eval_model_loss(model, loss_fn, data_loader):
    """
    Evaluate <model> on data from <data_loader> with criterion defined
by <loss_fn>
    """
    avg_loss = 0
    for _ in range(25):
        x, t = data_loader()
        y = model.forward(x)
        loss = loss_fn(y, t)
```

```
        avg_loss += loss.item()/25
    return avg_loss

def plt_sample(model, loader):
    """
    Compare model output with target and input image from <loader>
    """
    x, y = loader()
    pred = model(x)
    img = torch.swapaxes(x[0,:,:,:],0,2).cpu().detach().numpy()/255
    img = np.rot90(img, k=1, axes=(1,0))
    img = np.fliplr(img)
    if len(y.shape) == 2:
        print(y.shape)
        target = y
    else:
        _, target = torch.max(y[0,:,:,:], 0)
    _, idx = torch.max(pred[0,:,:,:], 0)
    target = target.cpu().detach().numpy()
    idx = idx.cpu().detach().numpy()
    plt.figure(dpi=200)
    plt.subplot(1,3,1)
    plt.imshow(img)
    plt.subplot(1,3,2)
    plt.imshow(target)
    plt.subplot(1,3,3)
    plt.imshow(idx)

def TrainModel(model, loader, epochs=10):
  """
  Train <model> using <loader> for <epochs> epochs.
  Training is done in-place (disregard return value).
  This function also creates global var "losses" (list) for evaluating
training.
  """
  loader.reset_epoch()
  # glob vars to track losses for printing after training
  global losses
  losses = []
  batches = int(loader.length/loader.batch_size)

  for ep in range(epochs):
    ep_loss = []
    for batch in tqdm (range (batches), desc="Epoch
"+str(ep+1)+"/"+str(epochs)+": "):
      x_train, y_train = loader()
      pred = model(x_train)
      optimizer.zero_grad()
      loss = loss_fn(pred, y_train)
      loss.backward()
      optimizer.step()
```

```
      losses.append(loss.item())
      ep_loss.append(loss.item())
    print(f"Avg loss: {sum(ep_loss)/len(ep_loss)}")
  return 0

def IoU(model, loader):
    """
    Computes intersection over union
    """
    avg_res = 0
    softmax = nn.Softmax(1)
    for _ in range(5):
        x, target = loader()
        target = target.to('cpu')
        y = model.forward(x)
        y = softmax(y)
        _, p = torch.max(y, 1)
        pred = torch.zeros(y.shape)
        pred[p]=1
        pred = pred.to('cpu')
        intersection = sum(sum(sum(sum(target*pred))))
        union = torch.count_nonzero(target+pred)
        avg_res += intersection/union
    avg_res = avg_res/5
    return avg_res.item()

def fuse_autoencoder(m):
    """
    Fuse an autoencoder model as defined by Autoencoder class in
section "Model".
    This functions fuses conv2d to relu in all parts of the model.
    """
    model = copy.deepcopy(m)
    # fuse encoder
    for i in range(len(model.encoder)):
        model.encoder[i] =
torch.quantization.fuse_modules(model.encoder[i], [['conv', 'relu']])
    # fuse decoder
    for i in range(len(model.decoder)):
        model.decoder[i] =
torch.quantization.fuse_modules(model.decoder[i], [['conv', 'relu']])

    return model
```

Model

```
class Down(nn.Module):
    """
    Halves input resolution and preforms convolution and relu-
operation
```

```
    """
    def __init__(self, in_channels, out_channels, kernel_size):
        super(Down, self).__init__()
        self.pool = nn.MaxPool2d(2)
        self.conv = nn.Conv2d(in_channels, out_channels, kernel_size,
padding=(kernel_size-1)//2)
        self.relu = nn.ReLU()

    def forward(self, x):
        return self.relu(self.conv(self.pool(x)))

class Up(nn.Module):
    """
    Doubles input resolution and preforms convolution and relu-
operation
    """
    def __init__(self, in_channels, out_channels, kernel_size):
        super(Up, self).__init__()
        self.upsample = nn.UpsamplingBilinear2d(scale_factor=2)
        self.conv = nn.Conv2d(in_channels, out_channels, kernel_size,
padding=(kernel_size-1)//2)
        self.relu = nn.ReLU()

    def forward(self, x):
        return self.relu(self.conv(self.upsample(x)))

class Autoencoder_d3(nn.Module):
    """
    Autoencoder model consisting of two modules: encoder and decoder,
    each of wich is made up of 3 blocks of either down- or upsampling
followed by conv2d and relu.
    """
    def __init__(self, kernel_size):
        super(Autoencoder_d3, self).__init__()
        self.quant = torch.quantization.QuantStub()
        self.encoder = nn.Sequential(
            Down(3, 32, kernel_size),
            Down(32, 64, kernel_size),
            Down(64, 96, kernel_size),
        )
        self.decoder = nn.Sequential(
            Up(96, 64, kernel_size),
            Up(64, 32, kernel_size),
            Up(32, 4, kernel_size)
        )
        self.dequant = torch.quantization.DeQuantStub()


    def forward(self, x):
        x = self.quant(x)
```

```
        x = self.encoder(x)
        x = self.decoder(x)
        x = self.dequant(x)
        return x

class Autoencoder_d4(nn.Module):
    """
    Autoencoder model consisting of two modules: encoder and decoder,
    each of wich is made up of 4 blocks of either down- or upsampling
followed by conv2d and relu.
    """
    def __init__(self, kernel_size):
        super(Autoencoder_d4, self).__init__()
        self.quant = torch.quantization.QuantStub()
        self.encoder = nn.Sequential(
            Down(3, 32, kernel_size),
            Down(32, 48, kernel_size),
            Down(48, 64, kernel_size),
            Down(64, 128, kernel_size)
        )
        self.decoder = nn.Sequential(
            Up(128, 64, kernel_size),
            Up(64, 48, kernel_size),
            Up(48, 32, kernel_size),
            Up(32, 4, kernel_size)
        )
        self.dequant = torch.quantization.DeQuantStub()


    def forward(self, x):
        x = self.quant(x)
        x = self.encoder(x)
        x = self.decoder(x)
        x = self.dequant(x)
        return x
```

Setup

Setup working directories

```
# Adjust as needed
TRAINING_SET_PATH =
"D:/Skola/Exjobb/agumented_data_set/augmented_train"
VALIDATION_SET_PATH =
"D:/Skola/Exjobb/agumented_data_set/augmented_val"
```

Setup model and device

```
device = torch.device('cuda:0' if torch.cuda.is_available() else
'cpu')
model = Autoencoder_d4(3)
model.to(device)
print(f"Device: {device}")
```

Setup dataloader, loss function and optimizer

```
train_loader = DataLoader(TRAINING_SET_PATH, batch_size = 16,
device=device)
loss_fn = nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr=0.00003)
```

Training

```
TrainModel(model, train_loader, 150)
```

```
plt.plot(range(len(losses)), losses)
```

Post Training Quantization

```
# Configuration and preparation
model_copy = copy.deepcopy(model)
model_copy.to('cpu')
model_copy.qconfig = torch.quantization.get_default_qconfig('qnnpack')
model_fused = fuse_autoencoder(model_copy)
model_prepared = torch.quantization.prepare(model_fused)
loader = DataLoader(TRAINING_SET_PATH)
# Observer calibration (doing inference on prepared model calibrates
it)
batches = 100
for i in range(batches):
    x, _ = loader()
    model_prepared(x)
# Conversion
model_q = torch.quantization.convert(model_prepared.eval(),
inplace=False)
```

Eval

Compute number of model parameters

```
print(str(sum(p.numel() for p in model.parameters() if
p.requires_grad))+" trainable parameters")
```

Compare inference time of regular vs quantized model

```
import time
train_loader = DataLoader(TRAINING_SET_PATH, 1)
```

```
samples = 100

model.to('cpu')
x, _ = train_loader()
start = time.time()
for i in range(samples):
    _ = model(x)
end = time.time()
print(f"Regular model: {(end-start)/samples} seconds/batch")

start = time.time()
for i in range(samples):
    _ = model_q(x)
end = time.time()
print(f"Quantised model: {(end-start)/samples} seconds/batch")
```

A sample of model output

```
plt_sample(model, train_loader)
plt_sample(model_q, train_loader)
```

Save/Load models

Save trained model weights

```
torch.save(model.state_dict(), "autoencoder_3x3.pth")
```

Save trained model as pickle (dill) object

```
dill.settings['recurse'] = True
dill.dump(model, open("autoencoder_3x3.pkl", "wb"))
```

Save quantized model as pickle (dill) object

```
dill.settings['recurse'] = True
dill.dump(model_q, open("autoencoder_3x3_quantized.pkl", "wb"))
```

Automation

This section is used to automatically train, evaluate and save a list of models and then shut down the computer once it is done.

```
import gc
import pandas as pd

dill.settings['recurse'] = True
device = torch.device('cuda:0' if torch.cuda.is_available() else
'cpu')
```

```
# settings
model_list = [Autoencoder_d3, Autoencoder_d3, Autoencoder_d3,
Autoencoder_d4, Autoencoder_d4, Autoencoder_d4]
model_names = ["autoencoder_d33x3", "autoencoder_d35x5",
"autoencoder__d37x7",
               "autoencoder_d43x3", "autoencoder_d45x5",
"autoencoder__d47x7"]
model_params = [3, 5, 7, 3, 5, 7]
learn_rate = [0.00003 for _ in range(6)]
EPOCHS = 150
train_loader = DataLoader(TRAINING_SET_PATH, batch_size = 32,
device=device)
val_loader = DataLoader(VALIDATION_SET_PATH, batch_size = 16,
device=device)
loss_fn = nn.MSELoss()

results = pd.DataFrame({'Name': model_names,
                        'Epochs': [EPOCHS for _ in
range(len(model_list))],
                        'Batch size': [32 for _ in
range(len(model_list))],
                        'Learning rate': learn_rate,
                        'Training loss': [0 for _ in
range(len(model_list))],
                        'Validation loss': [0 for _ in
range(len(model_list))],
                        'L1_loss': [0 for _ in
range(len(model_list))],
                        'IoU': [0 for _ in range(len(model_list))]})

for i in range(len(model_list)):
    print(f'Training {i+1} out of {len(model_list)} models')

    # Train model
    model = model_list[i](model_params[i])
    optimizer = optim.Adam(model.parameters(), lr=learn_rate[i])
    model.to(device)
    TrainModel(model, train_loader, EPOCHS)

    # Save figure
    fig, ax = plt.subplots(nrows=1, ncols=1)
    ax.plot(range(len(losses)), losses)
    fig.savefig(f'{model_names[i]}_tloss.png')
    plt.close(fig)

    # Evaluation of model
    t_loss = eval_model_loss(model, loss_fn, train_loader)
    v_loss = eval_model_loss(model, loss_fn, val_loader)
    L1 = eval_model_loss(model, nn.L1Loss(), val_loader)
    iou = IoU(model, val_loader)
```

```
        results['Validation loss'].iloc[i] = t_loss
        results['Validation loss'].iloc[i] = v_loss
        results['L1_loss'].iloc[i] = L1
        results['IoU'].iloc[i] = iou

        # Save original model
        dill.dump(model, open(model_names[i]+".pkl", "wb"))

        # Manually free memory just to be safe
        del model
        gc.collect()

    # Save training data
    results.to_csv('training_results.csv')

    # shutdown
    os.system("shutdown now")
```

DEPARTMENT OF PHYSICS
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden
www.chalmers.se

CHALMERS
UNIVERSITY OF TECHNOLOGY