NTNU

NORWEGIAN UNIVERSITY OF SCIENCE AND TECHNOLOGY

PROJECT THESIS

Perception system for pose estimation of autonomous quadcopter

Author:
Peter Bull HOVE

Supervisor:
Anastasios LEKKAS
Tom Arne PEDERSEN

A thesis submitted in fulfillment of the requirements for the degree of Masters in Cybernetics and Robotics

in the

Institute for Technical Cybernetics
Faculty of Information Technology and Electrical Engineering

December 22, 2020

NORWEGIAN UNIVERSITY OF SCIENCE AND TECHNOLOGY

Abstract

Faculty of Information Technology and Electrical Engineering

Masters in Cybernetics and Robotics

Perception system for pose estimation of autonomous quadcopter

by Peter Bull HOVE

A quadcopter is a flying vehicle with four propellers. Quadcopters are easy to use, cheap to produce and versatile. They have gained a lot of popularity in the recent years. Due to improvements with battery technology the use-cases for them have increased together with longer possible air-time. In order to reduce labor cost of human pilots controlling quadcopters for missions a lot of work is research is put into creating autonomous quadcopters. The use-cases for autonomous quadcopters are many as the cost of such operations can be minimal. In order for autonomous quadcopters to make decisions and control themselves they require a well performing perception system.

This thesis builds upon the work performed by Thomas Sundvoll in his master thesis [1].

In this thesis a pose estimation perception system is proposed. The pose estimator estimates the position and orientation of the quadcopter in relation to a specific landing platform. The pose estimator utilizes traditional computer vision techniques in a combination with deep learning-based computer vision in order to estimate quadcopter pose.

The pose estimator is tested in a simulated environment, where the quadcopter is able to perform automated landing and stable hovering using the proposed pose estimator. The pose estimator based on deep learning computer vision is tested independently, and the quadcopter is able to perform autonomous missions using this pose estimator as well.

Acknowledgements

I would like to thank the Norwegian University of Science and Technology for providing me with the opportunity to take my master's degree in Cybernetics and Robotics. I would also like to thank my supervisor, Anastasios Lekkas and my co-supervisor Tom Arne Pedersen for providing me with excellent guidance and advice, but also patience and support while I was writing this thesis. I would not have made it as far in this project without the support from my friend Thomas Sundvoll whom also provided me with his time and advice.
I want to thank DNV-GL who provided a 3D model of the ReVolt vessel for use in computer simulations, and thank Sindre Benjamin Remman for providing me with PC with GPU for running experiments.

Contents

List of Figures

List of Tables

List of Abbreviations

UAV	**U**nmanned **A**erial **V**ehicle
mAP	**m**ean **A**verage **P**recision
PID	**P**roportional-**I**ntegrative-**D**erivative
MMA	**M**oving **M**edian **A**veraging
IMU	**I**nertial **M**easurement **U**nit
DNN	**D**eep **N**eural **N**etwork
CNN	**C**onvolutional **N**eural **N**etwork
DSL	**D**eep **S**upervised **L**earning
FCNN	**F**ully **C**onnected **N**eural **N**etwork
YOLO	**Y**ou **O**nly **L**ook **O**nce
CPU	**C**entral **P**rocessing **U**nit
GPU	**G**raphics **P**rocessing **U**nit
FPS	**F**rames **P**er **S**econd
TCV	**T**raditional **C**omputer **V**ision
DL	**D**eep **L**earning

Preface

This thesis is written in the fall of 2020 in Trondheim, at Norwegian University of Science and Technology. It is written as a part of my master's degree in Cybernetics and Robotics. The work presented in this thesis builds upon the work of Thomas Sundvoll, who in his master thesis proposed a pose estimator for a quadcopter using traditional computer vision algorithms.

Automation has been on the mind of human beings for thousands of years. From starting by using animal power for transport and agriculture, to using water power for milling wheat into flour, and later inventing the steam engine. In the last 200 years electric machines have revolutionized the way humans work. Computers have brought another another era of automation, and now we are in the automation era of artificial intelligence. Automation frees up time and capacity for humans to do other things, to continue innovation, to have a higher level overview, or to have more free time. Automation enables scaling of work that was previously restricted by the limit of human capacity and time.

The goal of this project thesis is create a robust position- and orientation (pose) estimator for a autonomous quadcopter using computer vision. The pose estimate is estimated pose in relation to a specific landing platform. This position estimate is to be used for an autonomous landing mission, where the quadcopter will land on the helipad autonomously. This work in this thesis will lay the foundation for my master's project which is to be written in the spring of 2021. The end goal of this thesis and the subsequent master thesis is a fully autonomous takeoff-and-land mission on the DNV GL ReVolt vessel in Trondheimsfjorden.

There are several contributions assisting in the development of this project thesis. My supervisor for this project is Anastasios Lekkas, and my co-supervisor is Tom Arne Pedersen from DNV GL, and they have been providing me with much-needed insight and guidance during the project.

Thomas Sundvoll provided me with with material form his work as well as some guidance in the beginning of the project. He also provided me with help in setting up the ROS and Gazebo runtime environments.

The Department of Engineering Cybernetics has generously provided a Parrot AR.Drone 2.0 as well as a Playstation 4 controller for controlling the Parrot AR. They also provided an OptiPlex 7040 computer with an Intel Core i7-6700 CPU at 3.40GHz x 8 for running computer simulation experiments. DNV GL has provided a 3D model of the ReVolt vessel for use in computer simulation environments.

The work in this thesis has been performed utilizing some open-source software, as well as some free software tools:

- CVAT computer vision annotation tool
- Roboflow software for dataset handling and data augmentation
- Roboflow tutorials, as well as open-source deep learning scripts from Roboflow
- ROS and Gazebo by Open Robotics
- Google Colaboratory training GPUs.
- The open-source ROS packages ardrone_autonomy, tum_simulator and uuv_simulator
- The open-source ROS package "Darknet For Ros" by Leggedrobotics.

Chapter 1

Introduction

1.0.1 Background and Motivation

Autonomous robots and vehicles are currently a major area of research, with new milestones being reached every year. Autonomous vehicles reduce human labour cost, and may become safer and more reliable than their human-controlled counterparts.

Quadcopters are a type of aerial vehicles that has become popular in the recent years. They are small and simple in design, but are reliable and maneuverable. Quadcopters are currently being used for inspection of hard-to-reach locations, cleaning of hard-to-reach locations, exploration of dangerous sites, as well as picture-taking for photographers and film-production. Most of these applications are done with a human pilot controlling these drones remotely. [2] Autonomous control of these quadcopters greatly increases the number of use-cases for them. This is because labor cost is reduced significantly by not requiring a human drone pilot to control them.

The motivation for this project and subsequent thesis is the development of a quadcopter capable of autonomous take-off and landing on a boat. Potential applications of such a drone include rudimentary search and rescue missions. Each year many people are lost at sea, are taken by avalanches and are lost in the mountains. Current search-and-rescue operations consist of many people, helicopters searching with infrared cameras and search-and-rescue dogs in order to find the people who are lost. Such operations are costly, and are therefore delayed until proven necessary. Many lives could be saved if such searches were performed earlier. Because of the high costs associated with such searches, they are only done when other options are exhausted. Utilizing autonomous quadcopters that can perform such searches would permit a cheaper operations, and therefore such operations may be used less sparingly. Having multiple drones perform an organized search would also make such a search more efficient.

Other operations unmanned quadcopters may perform are to deliver life-saving medicine to people in remote areas quickly, autonomous inspection missions, autonomous transportation of goods, and numerous other applications. Autonomous drones are cutting edge technology, and a large amount of research is being put into autonomous drones because of their usefulness.

For such autonomous drones to function well they require a good perception system. Correct decisions require a good understanding of the world the autonomous agent inhabits. A camera view is a good way to gather information about the surrounding world because it collects much information in a single shot. A camera is a general sensor that may be useful in all stages of the missions. Most aerial drones are equipped with cameras. A well performing perception system is required to extract useful information from the camera images.

1.0.2 Previous Work

There is much work that is previously done on computer vision perception systems for use for pose estimation and automatic control. Such computer vision techniques use either a Deep Learning-based approach or a traditional computer vision approach. Multiple approaches use computer vision techniques for autonomous quadcopter control. Most computer vision systems for use with autonomous quadcopters in literature are based on traditional computer vision techniques, although some propose Deep Learning-based methods.

Thomas Sundvoll proposed a pose-estimation system for an autonomous drone in an automated landing scenario with the AR.Drone 2.0 [1]. His work was based on a traditional computer vision techniques for locating the landing platform in the camera frame, and uses that information to calculate quadcopter pose. Experiments with Sundvoll's pose estimator show that the pose estimator estimates pose accurately and consistently in the Gazebo simulation environment, with small error margins when compared to ground truth. The quadcopter is able to perform a successful autonomous landing in the Gazebo simulation environment. Sundvoll's pose estimator requires tuning for the specific light and visual conditions, and when tested with the physical AR.Drone 2.0 in a real-world experiment it was seen that it lacked the robustness to be able to function well and reliably in visual conditions that were different and more challenging than the simulated environment.

A computer vision based pose estimation algorithm is presented in [3] and used for autonomous landing of quadcopter. This pose estimator detects prespecified ArUCO markers, which are similar to camera QR-codes. The ArUCO markers are distinct, and have known location and orientation. The algorithm detects the ArUCO markers in the camera frame, and calculates pose by mapping the ArUCO markers in the camera frame to the known ArUCO marker pose. An automated landing mission is performed by landing the quadcopter directly over a ArUCO marker.

A similar approach is proposed in [4]. The experiments tested with this approach were performed in the Gazebo simulation environment. Automatic landing is achieved with the pose estimates.

Another approach [5] for pose estimation using traditional computer vision is by combining pose estimates from object tracking with inertial measurement unit data in an Extended Kalman Filter (EKF).

A pose estimation for automated landing of a quadcopter is proposed in [6] where the quadcopter detects a known feature on the landing platform using the openCV function findContours(). A white H is used as a distinct feature on the landing platform. A pose estimate is calculated using position and orientation of the detected H in camera frame. The quadcopter is able to use this pose esimate for automated landing on the landing platform, as long as the H is visible, and the size of the H is known.

J. Blom proposes a traditional computer vision based perception system for autonomous landing on an unknown landing platform [7]. This computer vision approach assumes that the landing platform is square, with a textured surface, and used a combination of traditional computer vision methods, such as Canny edge detection as well as feature detection of the textured landing platform to locate the corners of the landing platform and perform autonomous landing.

Visual Simultaneous Localization and Mapping (vSLAM) [8] is a method for tracking features and matching features between frames. This is done to estimate

changes in pose, while at the same time creating a map of the surrounding environment. The approach is based on visual point cloud methods, and requires no prior knowledge about the world. This is used for real-time navigation of robots.

A real-time indoor navigation system for quadcopters located in hallways is proposed in [9]. This approach uses Canny edge detectors and Hough line transforms. Navigation is achieved by steering towards a vanishing point, and performs well in in hallways with good visual conditions.

Deep Learning-based computer vision techniques are also used extensively in literature, and are also used for autonomous control of quadcopters.

YOLO (You Only Look Once) [10] is a bounding box prediction algorithm used for real-time object detection. In [11] YOLOv3 and YOLOv4 bounding box prediction algorithms are tested on KITTI image dataset, to recognize Car, Truck, Person and Two-wheeler in a traffic scene, and is able to achieve high mAP. Bounding box prediction algorithms are used in combination with quadcopters, and [12] presents a system which performs real-time inference on images from camera located on a quadcopter.

Imanberdiyev et al. proposes a Reinforcement Learning scheme for autonomous quadcopter control [13] using TEXPLORE reinforcement learning framework.

In [14] autonomous navigation of quadcopter is achieved using a deep learning pose estimator from camera input. The autonomous navigation is used for following forest trails. The estimated pose from the DNN is relative pose when compared to the trail, and this relative pose is used as feedback for control.

Bounding box prediction algorithms are used for real-time tracking of golf balls in [15]. Golf balls are detected and the position of the golf balls are estimated using bounding boxes around the golf balls. The position estimates are filtered using a Kalman Filter in order to estimate a more accurate ball position. The algorithm is tested using multiple bounding box classification models, and Faster R-CNN produced the best results.

A 6 dimensional object pose estimation scheme is proposed in [16], where the position and orientation of objects are estimated. It works by using locating bounding boxes enclosing the objects of interest, as well as using a separate network to estimate the object orientation. The bounding box, together with the estimated orientation, is used to estimate the object's position.

1.0.3 Objectives

The objective of this thesis is to create a robust position and orientation (pose) estimation system for a quadcopter using a combination of traditional computer vision techniques and deep learning computer vision techniques. Many pose estimation algorithms, using computer vision for autonomous control of quadcopters, are made, some using TCV and some using a DNN approach. A pose estimation system which combines DNN and TCV methods is proposed in this thesis and tested for use in autonomous control of a quadcopter.

The estimates from the two computer vision techniques are to be combined to create a single estimate. The estimate is to be robust, meaning that the estimator is able to perform reliably and well in different lighting and challenging visual conditions. The computer vision techniques are combined in order to get a strong and robust computer vision estimate. Both methods have strengths and weaknesses. DN methods can are fast and general, and the TCV methods can be accurate and precise. By combining the techniques we are able to combine the strengths of the methods, while compensating for the weaknesses of the methods.

This pose estimate is used for control during autonomous missions with the quadcopter. The autonomous missions to be performed are *stable hovering* and *automated landing*.

The contributions in this thesis are specifically

- Creating a labeled dataset of a landing platform, and use that to train YOLO object detectors to detect specific features of the landing platform.
- Creating an algorithm for calculating relative quadcopter pose in relation to the landing platform by using results from the YOLO object detection algorithm. The algorithm uses object detection bounding boxes size and location in order to estimate quadcopter pose.
- Merging this estimated pose with pose estimates generated by a traditional computer vision-based approach created by Thomas Sundvoll in his master thesis [1].
- Testing the robustness of the combined pose estimates in a Gazebo simulated environment by using the combined pose estimate for autonomous quadcopter landing on a landing platform situated on a boat.

1.0.4 Outline

Chapter 2 presents the theory behind the computer vision techniques utilized in this project. Chapter 3 presents the experimental setup and how to replicate this experiment for future testing. Chapter 4 presents the methodology for this project and how the computer vision pose estimators were implemented and combined in order to create a combined robust estimate. Chapter 5 presents the results from the experiments performed with the pose estimator, and discusses the results in light of the objective. Chapter 6 concludes the work and results presented in this thesis, and discusses possible future work on this topic.

Chapter 2

Theory

2.1 Traditional Computer Vision

Computer vision is a field of study where there has been much improvement in the later years. Computer vision facilitates for autonomous robots and autonomous vehicles by being a perception system which robots and autonomous vehicles can use to perceive the world. Perception is key for well functioning autonomous systems, and computer vision is therefore important in the development of robots and autonomous vehicles.

The domain of digital image processing has consisted of mostly traditional computer vision algorithms, however in the later years it has been dominated by Deep Learning (DL). There are benefits and drawbacks of both methods of computer vision. Even though DL computer vision has made numerous breakthroughs, it has not yet made traditional computer vision techniques obsolete. [17]

Traditional computer vision techniques mainly consists of three parts. The first part is feature detection, where the algorithm locates points of interest in the image. Points of interest are corners, edges, and areas with high color contrast in relation to the surrounding pixels. These are points that contain the most information in the image. An image reduced to the points of interest will contain the main structures and features of the image while being represented with much less information.

The next step is feature description, in which the algorithm creates a descriptor for each of the located points of interest in the image. A descriptor is a descriptive representation of a point and it's surrounding pixels. Descriptors are used to locate the same points in other images of the same scene. This is done by comparing descriptors and matching points for which the descriptors match. The third and final step is classification where the points of interest and descriptors are used for the intended purpose of the algorithm.

Thus, traditional CV works by finds low-level features in an image, and performs classification by matching these features together.

2.2 Deep Neural Networks

In the recent years Deep Neural Networks (DNNs) have made much progress within image classification and detection [18] [19]. Real-Time object detection and classification has become possible using algorithms such as YOLO [10] and Faster R-CNN [20]. Deep Neural Networks consist of neurons. Neurons are nodes in a network, where each neuron contains a single value. In most neural networks the value contained in a neuron is a number between 0 and 1. The value in a neuron is calculated as a function of its inputs. The value in a neuron is passed on to other neurons,

where that value will be one of the inputs to the other neurons. Numerous neurons are connected in a network of neurons.

Some of the neurons are input-neurons. Input data is passed in to a neural network by setting the value in these input neurons equal to the input data. For example, when passing image data into a neural network each pixel value of the image is passed into a separate neuron, or multiple neurons if the image is in colors.

Some neurons in a neural network are output-neurons. These neurons do not pass their values on to other neurons, but instead output the result of the neural network. Thus, a neural network is a mapping from input values in the input neurons to output values in the output neurons.

2.2.1 Fully Connected Neural Networks

A Fully Connected Neural Network (FCNN) is a type of DNN where neurons are connected together in organized structures called layers. The structure of a FCNN is illustrated in in Figure 2.1. Input data is passed into the first layer of the FCNN, and the output is read from the last layer in the FCNN. A FCNN, as the name suggest, is fully connected. This means that every single neuron of a layer in connected to all the neurons of the previous and subsequent layers.

Neurons are connected with connections that are called weights. The value of a neuron that is passed onto the next neuron is multiplied with the weight that connects the two neurons. The resulting value of a neuron is therefore a weighted sum of all neurons in the previous layer. An individual bias is also added to each neuron. This result is put through an activation function $f : \mathbb{R} \to \mathbb{R}$ in order to get the value of a neuron in the desired range.

The output value a_j^k of a neuron j in layer k is

$$z_j^k = \sum_{i=0}^{n} w_{ij}^k a_i^{k-1} + b_j^k \tag{2.1}$$

$$a_j^k = f(z_j^k) \tag{2.2}$$

where $w_{ij}^k \in \mathbb{R}$ is the weight connecting neuron i in layer $k-1$ with neuron j in layer k. $b_j^k \in \mathbb{R}$ is the bias for neuron j in layer k. z_j^k is the input value for the neuron and $a_j^k \in \mathbb{R}$ is neuron output value. $f : \mathbb{R} \to \mathbb{R}$ is an activation function, used to get the neuron output value in the desired range.

The output values of all the neurons in layer k is therefore

$$z^k = w^k a^{k-1} + b^k \tag{2.3}$$

$$a^k = f(z^k) \tag{2.4}$$

where $w^k \in \mathbb{R}^{n \times m}$ is a matrix of weight connections from neurons in layer $k-1$ to layer k. $a^{k-1} \in \mathbb{R}^m$ is a vector of the output values from the previous layer, $b^k \in \mathbb{R}^n$ is a vector of bias values for the neurons in layer k.

$z^k \in \mathbb{R}^n$ is a vector of neuron values in layer k before being passed through an activation function, and $a^k \in$ is a vector of neuron values in layer k after being passed through an activation function.

There are different activation functions that are commonly used in DNNs. The most frequently used activation functions are; *Sigmoid*, *ReLU* and *Softmax*, as these are known to produce well trained models.

The Sigmoid function is a function that squishes the output between 0 and 1. The Sigmoid function is defined as

$$f(x) = \frac{1}{1 + e^{-x}} \tag{2.5}$$

where x is the input value. Thus, f is defined: $f : \forall\, x\, f(x) \in (0, 1)$. A graphical representation of the Sigmoid function can be seen in Figure 2.2.

The ReLU function a function which is equal to the input value for all positive input, but is zero for all negative input. Thus, ReLU is written as

$$f(x) = \begin{cases} x, & \text{if } x \geq 0 \\ 0, & \text{otherwise} \end{cases} \tag{2.6}$$

and an illustration of this can be seen in Figure 2.2

The Softmax function is an activation function that squishes the output between 0 and 1, and in a way that the sum of all output equals to one. The softmax is written as

$$f(x_j) = \frac{e^{x_j}}{\sum_{i=0}^{n} e^{x_i}} \tag{2.7}$$

An example of the softmax activation function can be seen in Figure 2.2. The Softmax function is commonly used for the last layer of DNNs that perform classification. This is because each output neuron is generally associated with detection of one class. The value of the neuron associated with one detection is then a measurement for the network's confidence in that classification compared to the other possible classifications. Thus, by forcing the sum of all output confidence to be 1 the output confidence can be read as a confidence percentage.

2.2.2 Training a Deep Neural Network

In order for a DNN to learn, it has to be trained. In what is called supervised learning, a DNN model is presented with training data. The training data is labeled with ground truth labels. These labels are what the trainer wants the DNN to be able to predict.

Training works by three steps, a forward pass, calculating loss, and backpropagating. A forward pass is by providing the DNN with an input in order to get the DNNs predicted output. The input data is then propagated through the hidden layers, and the output value is read from the output neurons. A forward pass is how the network is used for inference on data after the network is trained.

Loss is the error that the model makes when predicting. Loss is calculated as the difference between the expected output and the actual output of the forward pass. Different methods of calculating loss exists. These methods are called loss functions. The most commonly used loss functions are Mean-Squared Error loss and Logarithmic loss. The loss function is denoted by C.

Backpropagation is a technique for changing the model in order to reduce loss. This is how the model is improved during training. The model parameter θ is defined as the weights and biases of the model, s.t. $\theta = [W, b]$. Backpropagation works by calculating the partial derivative of loss with respect to the model parameters:

$$\frac{\delta C(\theta, z_k)}{\delta \theta}$$

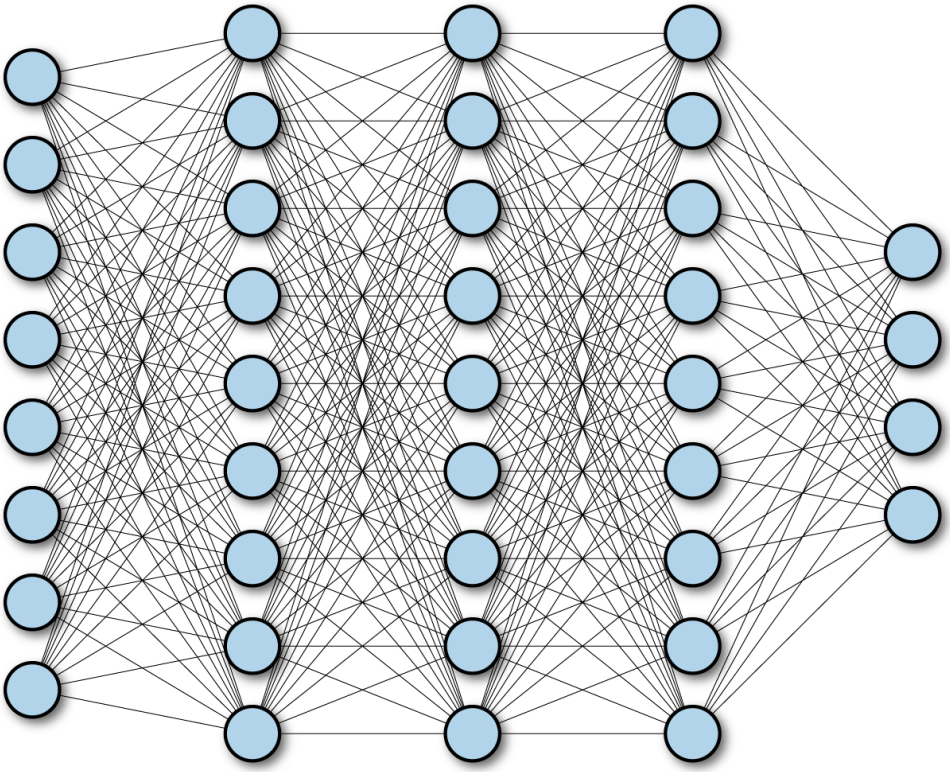


FIGURE 2.1: The structure of a FCNN. The first and leftmost layer is for input data. The last and rightmost layer is output layer. The layers in between are the hidden layers. the lines between neurons are weights. Together these neurons and weights compose a Deep Neural Network

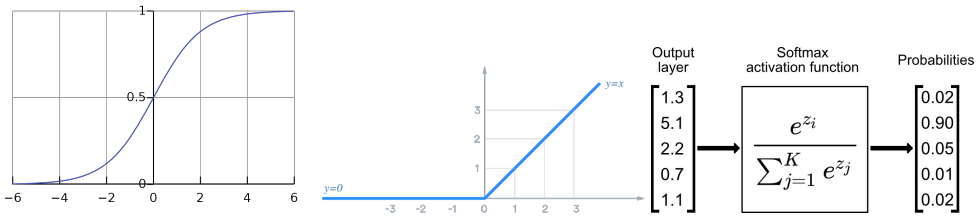


FIGURE 2.2: Sigmoid, ReLU and Softmax activation functions.

This partial derivative is a numerical representation of how changing the model parameters changes loss, given the output of the network on the current data, z_k. The model is then updated according to an update rule. The update rule states how to change the weights given the partial derivative of loss with respect to the model parameter.

One update rule is called gradient descent. Using gradient descent the update rule is such that the model parameter θ is reduced by the derivative of the loss function with respect to the the model parameters, scaled by a learning rate α. Thus, gradient descent is the update law

$$\theta_{\text{new}} = \theta - \alpha \frac{\delta C(\theta, z_k)}{\delta \theta} \tag{2.8}$$

Learning consists of numerous forward passes and backpropagrations, repeated on the training data. Generally input data is presented in batches, and the model performs forward passes on all of the data. Total loss is calculated for the whole batch, and the model is updated such that the total loss over the whole batch is reduced. Training a model with batches makes training less prone to oscillate between trained values, and the model is less influenced by outliers in the training data.

There exists many techniques for improving the training of a DNN. *Dropout* is a technique where the model will temporarily remove randomly chosen weights during the forward passes when training. This makes it so that the network will not use the same neurons for every classification, but will force the model to train all the neurons in the network. From experiments this has been seen to make neural network models more general and robust. *Momentum* is a technique for use with the model update law, in which the model updates will be a moving average of parameter updates. This results in parameter updates which are more smooth and in the general direction of loss decrease. *Batch normalization* is a a technique that consists of normalizing the input data to the model. This makes the model become faster to train, and more stable. Weight Decay is a technique in which all the model parameters are reduced by a small factor each iteration according to a decay parameter v. This prevents some neurons from having very large weight values, and thus dominating the complete network. The result is a more general and well performing network. Early stopping is a technique for stopping the model from becoming overfitted during training. This is done by stopping training if loss decrease has halted during training.

2.2.3 Convolutional Neural Networks

When working with image data, a FCNN is not able to capture all the information in the data. In an FCNN every neuron is connected to every neuron in the next layer. For an image, where the input values are individual pixels, this makes it so that all spacial information is lost. A Convolutional Neural Network (CNN) provides a solution to this [21].

A CNN consists of three layer types; convolutional layers, pooling layers, and fully connected layers. A convolutional layer is a layer which the values of neurons are passed on to neurons in the next layer using a convolutional map. Such a map is a moving local map which combines nearby neurons in a spatial grid to combine the values into one neuron. Such maps extricate local features and qualities in the data. These convolutional maps are trained in the same way that weights are trained in FCNNs. An illustration of a convolutional map can be seen in Figure 2.3, where K is

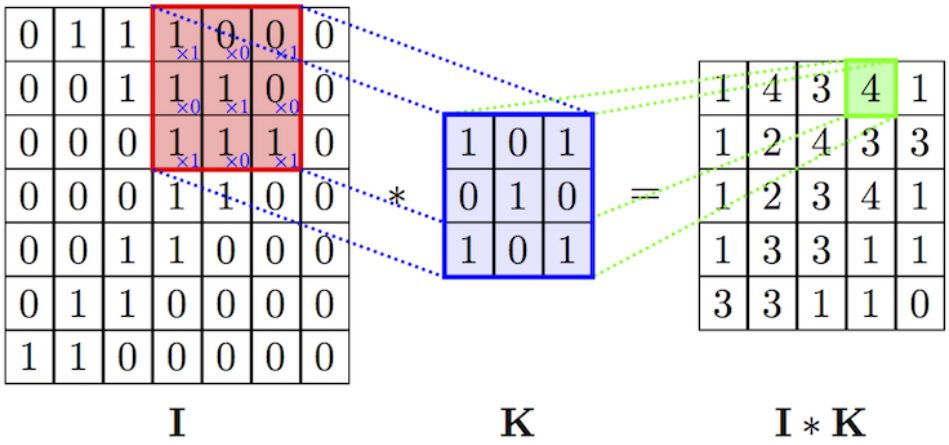


FIGURE 2.3: An illustration of a convolutional map in a convolutional layer in a CNN.

the convolutional map, I is the image data, and I*K is the resulting output from the convolutional layer.

A pooling layer is a layer where multiple neurons in the previous layer are combined into one. Different pooling techniques exist, such as median-pooling, average-pooling, and max-pooling. Max-pooling is the most commonly used pooling technique in CNNs. Pooling layers are used to reduce the dimension of the data, while preserving the useful information. Pooling layers are usually situated in between convolutional layers.

The fundamental structure of a CNN is that input data is passed through multiple layers of both convolution and pooling. The result from this is then passed to Fully Connected Neural Network (FCNN) in the end. This structure can be seen in Figure 2.4. All spatial information is lost in the FCNN, however the FCNN connects the large features extracted from the previous layers and puts them together in order to produce the output values of the network.

CNNs have had huge success in image classification algorithms and all of the leading image classification methods are CNNs.

2.2.4 Traditional Computer Vision Methods compared with Deep Learning Methods

Deep Learning Computer Vision methods have dominated the space of computer vision in the last years, but there are still domains where Traditional Computer Vision (TCV) methods are favored. Both DL methods and traditional methods have advantages and disadvantages. This means that one can choose what methods to use depending what is best for the specific application [17].

TCV methods provide full transparency of implementation and function. This is because TCV methods are based on known methods and can easily be analysed in order to determine why the result was what it was.

DL methods on the other hand are often perceived as black-box functions. A black-box function is a function which one provides input and receives output, but one does not know how the model works. DL methods may utilize a neural network consisting of millions of neurons, weights, biases and convolutional mappings. This makes it very hard to analyse the decision process of the neural network, as there are so many parameters. This black-box behaviour can be a safety risk, because it is hard

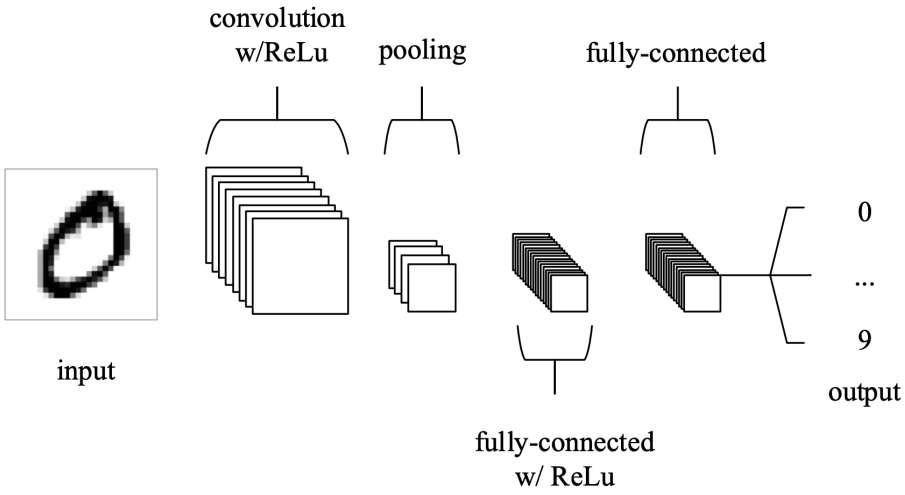


FIGURE 2.4: The general structure of a CNN, containing convolutional layers, pooling layers and fully connected layers. Image collected from [22]

to determine whether it may fail or not. Such methods are therefore not acceptable in all applications.

TCV methods may be easier to tune for specific applications. TCV methods have fewer tuning parameters than the possibly millions of neurons in a DNN. TCV methods can be easier to improve iteratively. This is because in order to improve a DN method one has to provide the model with more or better training data. An alternative to this is to test training of the DNN model with different parameters. Training a DNN is time consuming, and iterative testing is therefore harder to do. Another problem with DNN methods are that they require large datasets of training data which needs to be labeled and specialized for each application. For a DNN model to achieve high accuracy it requires that the dataset is of high quality. A high quality dataset is a dataset which is properly- and precisely labeled, and contains data that is sufficiently varied for the model to become general and robust. TCV methods do not require datasets. Instead TCV methods require knowledge about the specific application. Such methods must be programmed to fit the specific application, and the reusability of such methods may be low.

DL methods may also achieve higher accuracy that TCV methods, especially when considering of complex detection problems such as recognizing humans or other complex structures in data. DNNs can be trained to be very general if provided with a large and varied dataset, and can thus be trained to be functional in a large variety of visual conditions and situations. Varying visual conditions may be challenging for TCV methods, as they need to be programmed for specific situations and may require different tuning for different visual conditions.

2.3 Darknet

Darknet [23] is an Open-Source neural network framework written in C by Joseph Redmon et. al.

The Darknet framework makes it easy to switch between DNN methods. One specifies the configuration file and the corresponding weight file when running classifications with the framework. A model structure is defined by a configuration file, where the number of layers, what kinds of layers, and activation functions are specified. The trained model parameters are specified in a weight file. One can change what model one wishes to run by changing the weight-file and the configuration-file, which makes comparing different methods very efficient.

Multiple image classification algorithms builds upon the Darknet framework. YOLO (You Only Look Once) is the most known and used, but there is also ResNet and AlexNet. Other DNN algorithms builds upon Darknet, for example DarkGo, which in a DNN algorithm for predicting the best moves in the board game Go.

2.4 YOLO

YOLO (You Only Look Once) is an image classification and bounding box prediction algorithm. [24] That means that the YOLO algorithm will predict the existence of certain classes of objects in an image and draw bounding boxes around the identified objects. The algorithm also predicts a confidence score for each prediction. The confidence score is the confidence of the algorithm of it being correct in it's prediction, with 0 being not confident at all and 1 being completely confident. An image with YOLO bounding boxes can be seen in Figure 2.5.

YOLO performs one inference on an image in order to predict the classes, bounding boxes, and confidence scores. The name You Only Look Once comes from the idea that YOLO only performs one forward pass through the network when presented with input data. Other prediction algorithms perform multiple inferences on images in order to detect bounding boxes as well as class predictions. This is one of the reasons that YOLO is a very efficient bounding box prediction network.

YOLO divides the picture into a grid of S x S cells, where S is a predetermined scale. Each cell will then predict B number of bounding boxes, where B is a tunable parameter. Each bounding box is represented by the coordinates of the center of the box (x,y) as well as the width and height of the box (w,h). Each bounding box predicted by each cell is centered in that cell. Each cell will then predict bounding boxes to fit the image data, and predict how confident the cell is that that bounding box is a correct class prediction. YOLO predicts class probabilities for each cell. As some classes are more likely to appear in certain cells in the image than others YOLO is able to use this information when performing predictions. The cell confidence is multiplied with the class probability for that cell. The result after this is very many bounding boxes, each which has a probability score associated with it. YOLO removes all bounding boxes below a certain threshold specified by the user.

YOLO also applies Non-Max Suppression to the bounding boxes. This is a technique that is indented to suppresses multiple detections of the same object. YOLO will predict an object for each cell in the grid, and if there is an object that spans multiple cells it may detect that same object multiple times with overlapping bounding boxes. Non-Max Suppression chooses the box from the set of overlapping bounding boxes with the highest probability score. The other boxes that predicted the same object in the same section of the image will be suppressed, and not output as predictions.

YOLOv3 [10] is the third iteration of YOLO, and was released in 2018. YOLOv3 is built on Darknet-53, meaning that it consists of 53 layers. 52 of the layers are convolutional layers, while the last layer is a fully connected layer. The structure of

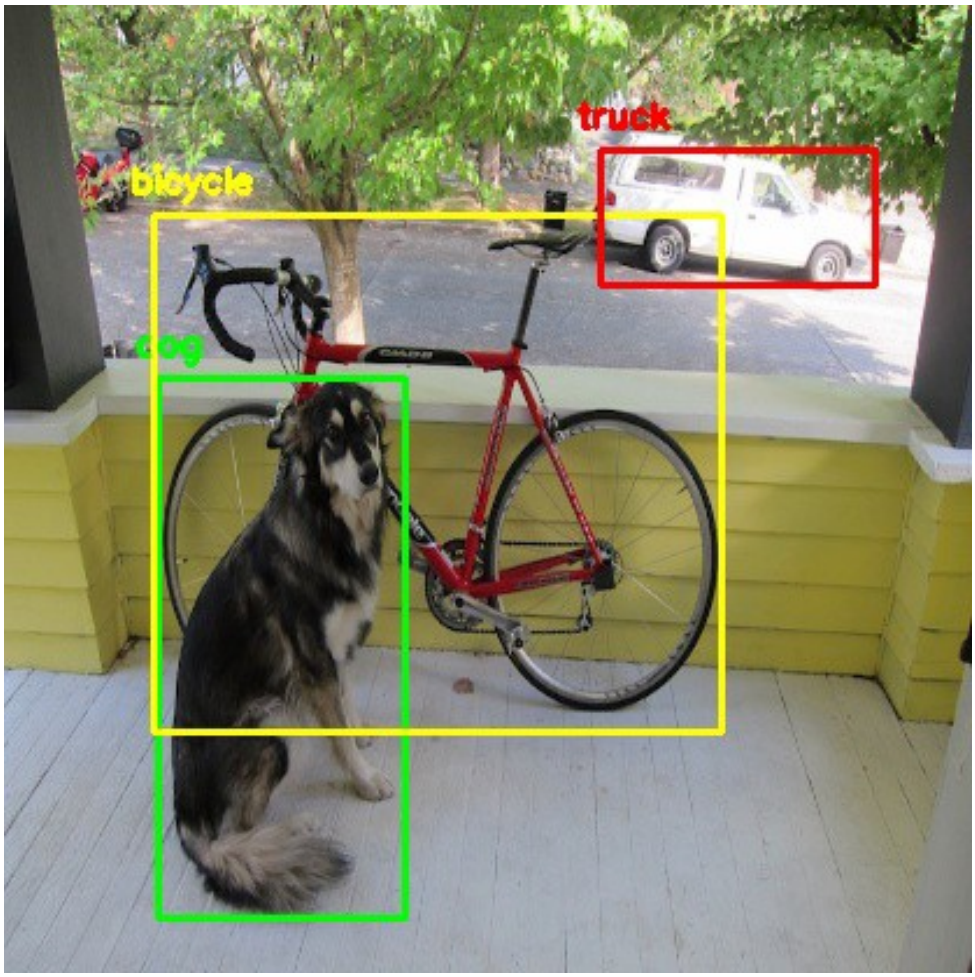


FIGURE 2.5: Image with YOLO bounding box predictions. Image collected from [23]

Darknet-53 is shown in Figure 2.8. YOLOv3 is quite accurate, achieving mAP scores that are on par with the best object classification algorithms. YOLOv3 detections are faster than R-CNN and RetinaNet, as can be seen inFigure 2.7 where YOLOv3 was was tested against other image classification algorithms on the COCO dataset [25].

YOLOv4 [26] is the fourth iteration of YOLO and was released in 2020. It is not developed by the same people who developed YOLO, YOLOv2 and YOLOv3, but has become the new standard for YOLO. This is because it is faster and more precise that YOLOv3, and builds upon the same Darknet framework. YOLOv4 is fast and can perform real-time inference at speeds up to 50FPS on a GPU2.6. This is very fast, and is therefore very useful for real-time classification systems. YOLOv4 is built on CSPDarknet-53. CSP stands for Cross-Stage-Partial and means that the layers contains skip connections where some of the data will not be passed through all the convolutional layers.

There exists tiny versions of the YOLO network as well. These are modeled to be very efficient real-time detectors, for use in mobile- or embedded systems. YOLOv4-tiny [27] is a tiny version of the YOLOv4 network, and YOLOv3-tiny is a tiny version of the YOLOv3 network. The tiny versions have significantly smaller network sizes. This makes for faster training- and inference times. As can be seen in Figure 2.6 there is a significant increase in FPS from YOLOv4 to YOLOv4-tiny, with YOLOv4-tiny reaching up to 375 FPS on a GPU compared to YOLOv4 reaching 50FPS. The smaller network size comes at a cost. The YOLO-tiny networks are less accurate and precise, with a lower mAP on average. They especially struggle to classify smaller objects, or with low resolution images. They are convenient to use when the processing power for inference is low and the need for faster inference trumps the need for higher mAP.

YOLO output is a list of bounding boxes found in the input image. Each bounding box is defined by 6 values:

- probability
- xmin
- ymin
- xmax
- ymax
- class

The bounding box encapsulating the classification is defined by the points (xmin, ymin) and (xmax, ymax), where the bounding box is the square where these points are opposite corners. The probability is the confidence score for the classification, and the class specifies what object class the classification belongs to.

The main advantage of YOLO is that it is very fast and efficient, while achieving a high mAP at the same time. This makes it one of the most used real-time object detection algorithms. Despite this, YOLO is known to have some weaknesses. YOLO has a lower detection rate for small objects in the images. Another problem with YOLO is that the predicted bounding boxes are not stationary around an object when performing real-time detection. The bounding boxes tend to have small translational shifts about the center of the object, making the bounding boxes 'jitter' around the objects [28]. These shifts happen because of multiple reasons, notably the non-max suppression in YOLO. In YOLO, each cell in the image predicts bounding boxes, as

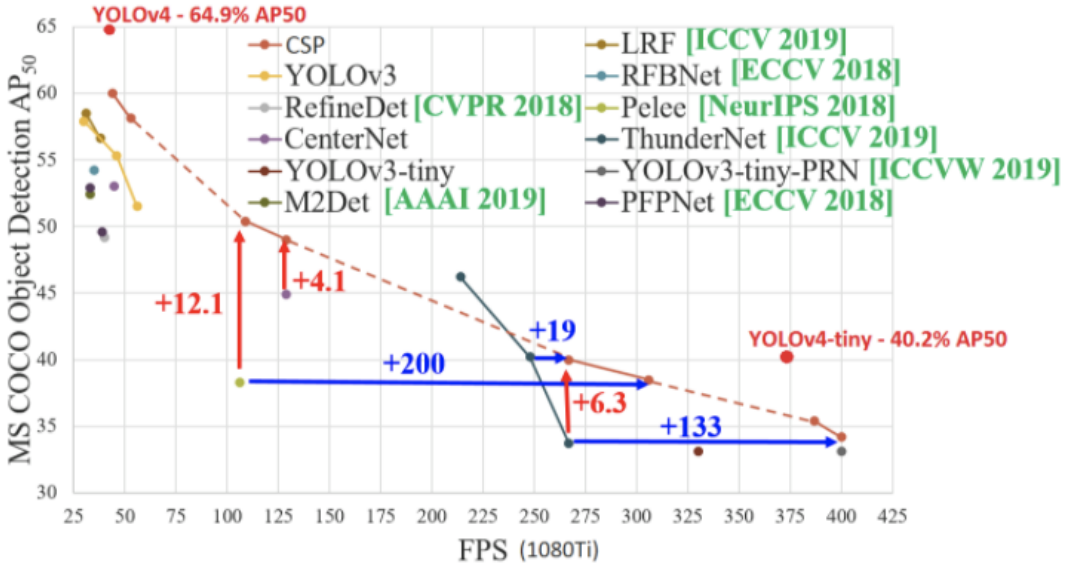


FIGURE 2.6: YOLOv4 and YOLOv4-tiny inference time and mAP, compared other detection algorithms.

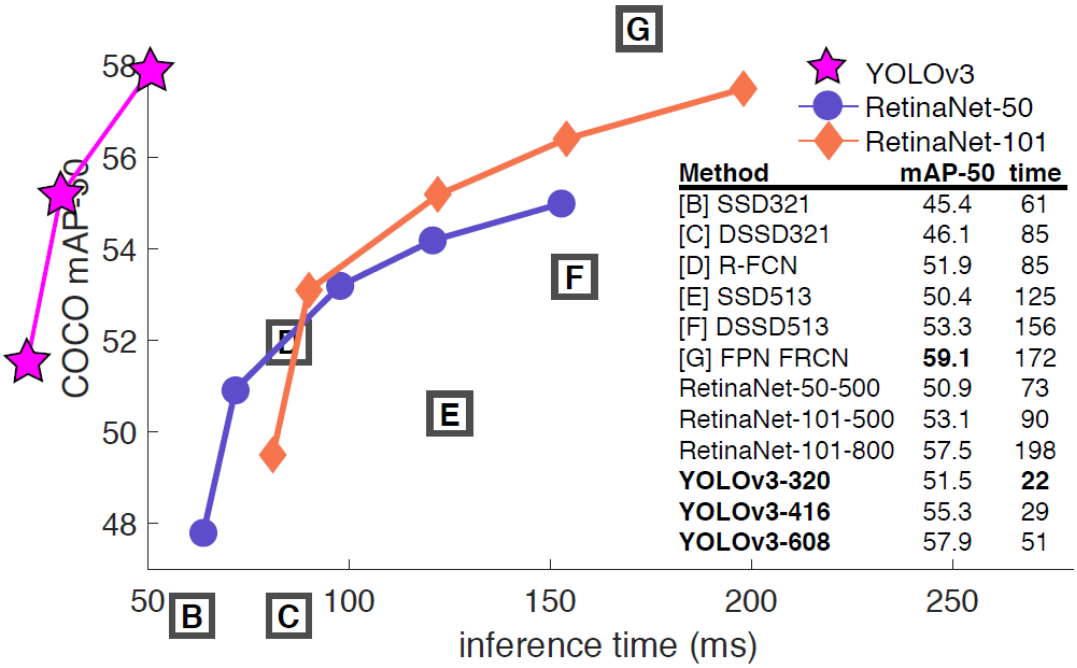
well as a confidence score. Adjacent cells that predict an object overlapping multiple cells may have very similar confidence, and may switch between having the highest confidence between frames. Each cell predicts a bounding box with center in that cell. Non-max suppression suppresses all but the most confident cell. If the most confident cell is not the same cell between frames, then the output bounding boxes may shift translationally between frames, and result in a bounding box 'jitter'.

2.5 Data Augmentation

In Deep Supervised Learning a Deep Neural Network is trained to recognize patterns in data. The model is trained by being exposed to a large number of data samples that is to be recognized. The data contains labels for the data pattern that is to be recognized. The network tries to predict the labels given the data. The model learns by backpropogating the errors that it makes and changing the model parameters in order to become a more accurate model.

A Deep Neural Network only as good as the data it has been provided with. A DNN that has been trained on more and better training data will yield a superior model compared to one that has seen less training data. Generally, a model that is trained on a balanced dataset gives a more accurate model. A balanced dataset is a dataset that contains about the same number of labels for each class of patterns that is to be recognized. Thus one of the challenges of training a good DNN model is to gather a large enough dataset that is balanced, and of high quality.

Data Augmentation is a technique for generating more training data by augmenting the data that is already gathered. [29] Data augmentation is performed by creating copies of the data and changing the data using pre-determined methods. Thus, one can take a single data piece, copy it and change the copies. Applying this to a dataset therefore results in a larger dataset. This makes it so that the initial dataset required to train a DNN model is less than what would otherwise be required.



Method	mAP-50	time
[B] SSD321	45.4	61
[C] DSSD321	46.1	85
[D] R-FCN	51.9	85
[E] SSD513	50.4	125
[F] DSSD513	53.3	156
[G] FPN FRCN	**59.1**	172
RetinaNet-50-500	50.9	73
RetinaNet-101-500	53.1	90
RetinaNet-101-800	57.5	198
YOLOv3-320	51.5	**22**
YOLOv3-416	55.3	29
YOLOv3-608	57.9	51

FIGURE 2.7: YOLOv3 object detection time on COCO dataset. Taken from [10]

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

FIGURE 2.8: Darknet-53 structure used in YOLOv3. Figure taken from [10]

One example of data augmentation for image data is *random rotation*. This augmentation technique creates new training images by rotating copies of the original images a random number of degrees, within a specified range. This makes it so that the DNN will learn to recognize the patterns in the images while being robust to rotation. In many cases, the orientation of the pattern does not change what the pattern is labeled as. A DNN that is being trained to differentiate between cats and dogs should be able to differentiate the animals regardless of the rotation of the images.

Another augmentation technique for images is flipping the images along the horizontal- or the vertical axis, called *horizontal flip* and *vertical flip*. This makes it so the training image seems like a new instance for the model, and the trained model will be more general with regards to flipped instances of the real class.

Other kinds of augmentations for image data are:

- Random skew
- Random noise
- Gaussian blur
- Color space augmentations, by transforming the colors in the image
- Randomly cropping parts of the image
- Translation, i.e. to shift the image in left, right, up or down.
- Combining multiple images into one
- Randomly erasing parts of the images

One wants to use augmentation techniques that are relevant to the specific application for which the model is to be used. If the DNN is to be used in a situation where it will be subject to much noise, then adding a *random noise* augmentation is preferable. If the DNN is to be used in an image classification situation with variable lighting conditions, then adding color space augmentations will be helpful.

2.6 Robot Operating System

Robot Operating System (ROS) is a free and open-source software framework for programming robots, created by Open Robotics [30]. ROS is a collection of tools, libraries and modules that simplifies the process of writing software for robotics. One of the main advantages of ROS is that there is a huge ecosystem surrounding ROS. There exists a community of active developers publishing open-source ROS packages that are publicly available for free.

ROS is based on a modular system, with modules that are called *nodes*. Nodes communicate by utilizing the messaging system of ROS, which is based on communication through messaging channels called *topics*. Nodes send messages over topics by *publishing*. Publishing is a broadcasting way of communication. Other nodes may subscribe to such topics, and by doing so they receive the messages when they are published. Multiple nodes may subscribe to the same topics, and multiple nodes may publish to the same topics.

A synchronous communication system also exists in ROS. This is called a service. A node may utilize a service by requesting a message from another node.

Messages in ROS are standardised between programming languages. Since messages are standardized it means that different nodes may be written in different

programming languages. Each node translates the subscribed message to a format which fits the programming language of the node. This simplifies reuse of code between projects and publicly available modules.

ROS is well integrated with a simulation environment called Gazebo, which is a simulation environment specialized for development of robotics. ROS and Gazebo has been used in extensively in previous work of development of autonomous quadcopter systems [13] [7] [4].

2.7 CPU- and GPU processors

A central processing unit (CPU) is the main processor in a computer. It is able to perform operations and calculations. These calculations is what makes a computer program run. Modern-day CPUs are able to perform billions of operations per second. The CPU usually has a small number of cores. A core is a separate processor able to perform operations. Multiple cores means that there are separate units which are able to operate in parallel. This makes operations more efficient. CPUs are fast and versatile, and are able to perform many different complex operations.

A graphics processing unit (GPU) is a processing unit which consists of a large number of cores. The number of cores in a GPUs is in the range between a couple hundred to a couple thousand, depending on the GPU. These cores are less complex than the cores in a CPU, and are only capable for performing simple operations. The vast number of cores makes a GPU excel at parallel computing. For tasks which there are many operations which are to be calculated in parallel GPUs perform better than CPUs. Most notably among such tasks are graphical operations, hence the name graphics processing unit. This is because when working with image data a large number of pixels are to be processed at the same time, and may be processed in parallel. The operations for processing these pixels are simple operations, so the GPU is able to handle this faster than a CPU which has to process the operation sequentially instead of in parallel.

When it comes to machine learning a GPU is much faster than a CPU. This is because training of- and inference using a neural network consists of millions of simple operations. These operations consist of calculating values for neurons in a neural networks, backpropagating, and updating network parameter values. These operations may be processed in parallel because of the linear structure of a neural network. The modern day GPU have ignited the worldwide AI boom due to it facilitating for larger AI models than the CPU does. Modern GPUs is able to perform neural network operations 50-500 times faster than modern CPUs. [31]

Chapter 3

Experimental Setup

3.1 Gazebo and ROS

The experiments in this thesis are performed in a simulation environment called Gazebo. Gazebo is an open source 3D simulator for robotics development. Gazebo provides a physics engine Gazebo 7.0.0 is used for these experiments.

Gazebo is well integrated with ROS. ROS Kinetic is used for communication between modules, as well as communication with the simulated environment in Gazebo. Camera output feed is published to a ROS topic which the control modules subscribe to. Control signals are published to other ROS topics in order to control the drone.

The experiments are run on Ubuntu 16.04.

3.2 Experimental Components

The quadcopter the experiment with performed on is the Parrot AR Drone 2.0. It is produced by Parrot and was released in 2012. It is released as a programmable drone for use with software development. [32] The Parrot AR. 2.0 is a quadcopter, consisting of four propellers. Each propeller has a separate motor which can be controlled independently. The drone can maneuver by applying larger force to some of the propellers than others. Varying left and right motor force will create a torque in roll direction, and the drone will roll. Varying the front and back motor force will create a torque in pitch movement, and the drone will pitch. Varying diagonal rotor speeds will cause a torque in yaw direction, and cause the drone to turn left or right. An image of the Parrot AR. 2.0 can be seen in Figure 3.1

A simulated version of the AR.Drone 2.0 is available through the ROS package /tum_simulator. This package contains an implementation of a gazebo simulation model of the AR.Drone 2.0. The experiments in this thesis is performed using the

FIGURE 3.1: Parrot AR.Drone 2.0. Image collected from [32]

FIGURE 3.2: A model of the helipad that is used in the experiments. Image collected from [1].

simulated version of the AR. Drone 2.0 in Gazebo. The drone has a bottom facing camera in order to perceive it's surroundings.

The AR. Drone 2.0 comes with a low-level autopilot. This autopilot controls the propellers, and follows reference points that it is fed, but it is also possible to feed control actuation commands directly. Parrot has made available a Software Development Kit which makes it easy for developers to develop their own applications for controlling the drone. The open-source ROS package /ardrone_autonomy is a ROS driver for the AR.Drone 2.0, and is used for communication with the AR. Drone 2.0. [33]

In the experiments the AR. Drone 2.0 performs a landing mission on a landing platform. The drone uses the bottom facing camera to locate the helipad in order to perform the landing mission. A simulated version of the landing platform is included in the gazebo simulator, and an illustration of the landing platform can be seen in Figure 3.2.

The helipad is located on a simulated version of the DNV GL ReVolt vessel. ReVolt is an innovative ship designed by DNV GL, as a project to create an unmanned, zero-emission surface vessel. DNV GL has manufactured a 1:20 scale model of the ReVolt in order to test the autonomous capabilities. The landing platform is mounted on top of the ReVolt model for this experiment. A simulated 3D model of the ReVolt with the helipad is modeled in the gazebo simulation. The 3D model of the ReVolt is provided by DNV GL. An image of the simulated environment with the quadcopter, landing platform and the model of the ReVolt vessel can be seen in Figure 3.3.

The simulations were performed on different computers, one with a CPU processor and another which also had a a GPU processor. The CPU experiments were performed on an OptiPlex 7040 with an Intel® Core™ i7-6700 CPU @ 3.40GHz 8. The simulations performed on a GPU were performed on the GTX Ti GPU.

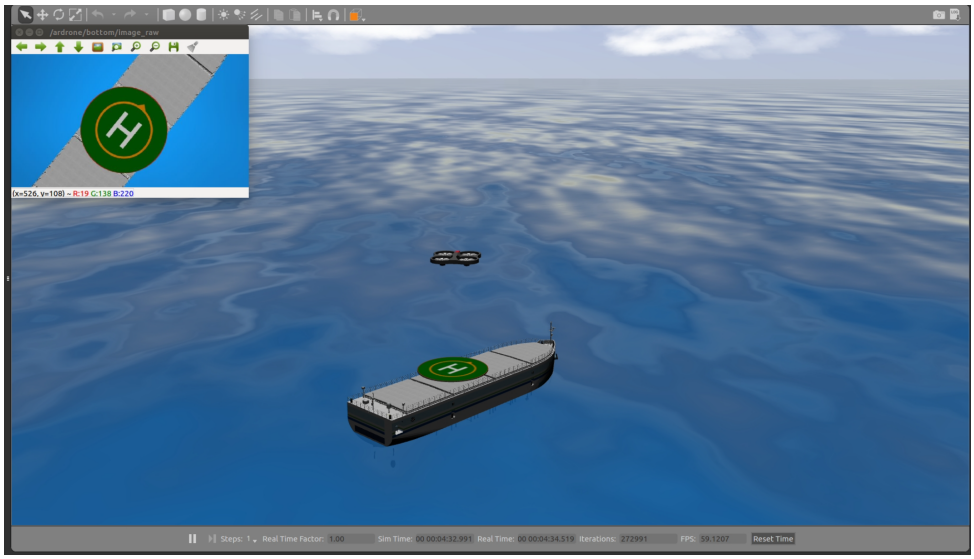


FIGURE 3.3: The Gazebo simulated environment with the quadcopter and the ReVolt vesssel. Image collected from [1]

Chapter 4

Methodology

4.1 Installation and setup of the Simulated Environment and ROS

Ubuntu 16.04 was required in order to install the correct version of ROS and Kinetic, and was therefore downloaded and installed on an OptiPlex 7040 computer. Ros Kinetic was installed, together with Gazebo 7.0.0. In order to set up the 3D simulation environment in Gazebo, 3D models of the simulated objects were collected and added to the simulation environment. DNV GL provided a 3D model of the ReVolt model ship, and Thomas Sundvoll provided a 3D model of the landing platform. These were added to the simulated environment, as well as a simulated ocean from uuv_simulator. Thus, the simulation environment contained the relevant models and objects.

The TCV pose estimation module and other modules from [34] was run in the simulated environment as ROS nodes, and thus the system collected from [1] were able to be run.

In order to develop this system further, multiple new ROS nodes, topics and launch files were created.

Migration of the system to a newer version of ROS, Gazebo, Ubuntu and Python 3 was attempted. Python 2 stopped being maintained at 1.1.2020, and therefore it is desirable to develop the platform on the more recent and updated version of Python. Package conflicts made this attempt unsuccessful because the ROS package /ardrone_autonomy is not updated since 2014, and is not integrated with versions of ROS more recent than ROS Kinetic.

4.2 Creating dataset

In order for a Darknet object detector to be able to locate the landing platform on an image it has to be trained. Supervised learning of a Darknet object detector model requires a dataset which of relevant images with proper labels. Thus, a dataset of of labeled images of the landing scenario is required. The dataset needs to be of sufficient size and quality for the pose estimation algorithm to perform well.

The images in the dataset were gathered from the simulator Gazebo. The images were collected using the output from the simulated AR.Drone's bottom facing camera. 1083 still images were taken and saved from different altitudes, angles and positions. The images were taken by manually flying the drone using keyboard commands and saving the current camera output as an image when specified using keyboard commands.

The images were labeled using CVAT [35]. CVAT is a free, online computer vision annotation tool for labeling of images required for training DNNs. Labeling consists

of drawing a bounding box around each object that is to be recognized. Each box is then given a specified label of which class it belongs to. The images in this dataset were labeled with bounding boxes of three classes; Helipad, H, and Arrow. The Helipad label is for the complete landing platform, encompassing the whole green circle. The H label is for the white H on the helipad. The Arrow label is for the orange triangle in the circle outside the H. A labeled image of the landing platform with the three labels can be seen in Figure 4.1.

The images in the dataset auto-oriented and resized to a proper YOLO format of 416x416 pixels, by scaling down the image and adding black edges to the top and bottom. This was preferable to stretching the image to fit the square format, because with black edged the original shapes of the circles and the H are preserved. Darknet scales input images this way in before inference, and it is desirable that the model is trained on data which is as close to the real data as possible.

The images were then subject to data augmentation, using data augmentation software from Roboflow. [36]. The augmentatinos that were applied were:

- 90 degree rotation
- $\pm$ 10 degree rotation
- $\pm$ 2 degrees of shear

These are augmentations that well reflect the visual conditions that the quadcopter will encounter during the landing mission. It is desirable that the computer vision system is invariant to rotation, so that the quadcopter is able to recognize the landing platform regardless of yaw rotation. If the quadcopter rolls or pitches, then the image of the landing platform will be slightly distorted, and the landing platform will not appear as a perfect circle. Therefore the images were subject to random shear to train the model to be invariant to this.

By augmentation these 1083 images were increased to 2599 images. These images were then split into three sets, one training set, one validation set, and one testing set. The training set consisted of 2274 images, while the validation and testing set consisted of 217 and 108 images respectively. The training set is for training the model. The validation set is for testing the model during training, and logging training results. The version of the model that performed best on the validation is chosen to be the final model. The test set is for testing the final model. The test set is data which the model has never seen during training. Thus the results from inference on the test set is a good score for a model's classification accuracy.

The dataset is a quite balanced. Every landing platform contains one element of each class. Each image contains only one landing platform. Some of the images do not contain the complete landing platform, so in some cases the Arrow or the H is not in the image. The vast majority of the images contains all three classes. The dataset is therefore well balanced, although contains come fewer elements of the class Arrow.

4.3 Darknet object detector

In order to estimate the pose of the quadcopter relative to the landing platform the system needs to locate the landing platform in the camera image. The quadcopter uses the bottom facing camera for capturing images. Locating the landing platform in the images is done by a Darknet object detection model. The Darknet models YOLOv4 and YOLOv4-tiny are used. The YOLO models outputs bounding boxes

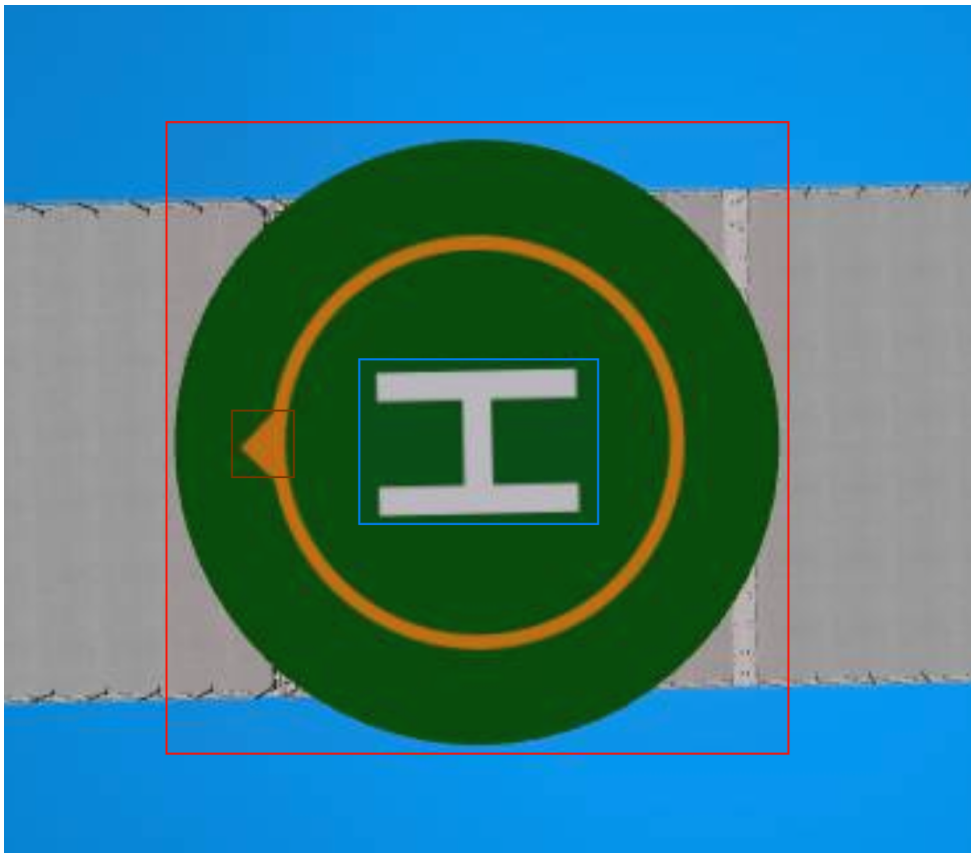


FIGURE 4.1: Labeled image of the landing platform with bounding box labels of the three classes; Helipad, H, and Arrow.

surrounding specific parts of the landing platform. The size and location of these bounding boxes can be used to estimate the center and radius of the landing platform in camera pixels.

4.3.1 Training Darknet

The Darknet models are trained in order to recognize parts of the landing platform. The models are trained by using supervised training. This works by providing the Darknet models with a labeled dataset. The model will be trained to classify the images according to the provided labels. The dataset the models are trained on contains 3 classes of objects. The classes are Helipad, Arrow, and H.

Two Darknet models are trained; YOLOv4 and YOLOv4-tiny. The YOLOv4 model and YOLOv4-tiny model are trained by transfer learning with using pre-trained weights as initial weights. The initial weights are weights released for YOLOv4 and YOLOv4-tiny by AlexeyAB [37], creator of YOLOv4 and YOLOv4-tiny. The initial weights are trained on the COCO dataset [25].

The models are trained on free GPUs provided by Google using their Google Colab Notebooks [38]. Open-source notebooks for training YOLOv4 and YOLOv4-tiny on Google Colab, provided by Roboflow [39] [36], are modified and used for training of the models. The labeled dataset is uploaded to the respective notebook. Thus training the models are done using the Google Colab cloud service.

After training the resulting weight files are downloaded and used with the Darknet algorithm.

4.3.2 Darknet for Ros

In order to incorporate Darknet into the quadcopter system *Darknet for ROS* is used. Darknet for ROS is a ROS package by Legged Robotics [40]. This package is a package for performing inference on images using Darknet classification models. Model configuration files for YOLOv4 and YOLOv4-tiny are added to the Darknet for ROS module, as well as the trained weights for the models into the Darknet for ROS module. By specifying which model configuration file and weight file is to be read in a ROS launch file, one can easily switch between Darknet models, image resolutions and model weights.

Darknet for ROS is set to subscribe to the ROS topic /ardrone/bottom/image_raw. Darknet for ROS automatically does inference on images as they are published, based on what model and weight that are specified. Darknet for ROS then publishes the detected bounding boxes to the topic /darknet_ros/bounding_boxes. The bounding boxes are published as a custom ROS message type; *Bounding_Boxes*. This message contains a list of the ROS message type *Bounding_Box*. Bounding_Box is a custom ROS message type which contains the following information

- confidence
- xmin
- xmax
- ymin
- ymax
- class id

- class_name

The ROS topic /ardrone/bottom/image_raw has a queue size of 1. This ensures that the current image on the topic is overwritten when a new image is published. Thus, Darknet will always start inference on the most recently published image.

4.4 Darknet pose estimator

In order to estimate pose of the quadcopter the algorithm first need to locate the landing platform in the camera frame. YOLO outputs bounding boxes surrounding the classified Helipad, H and Arrow in the image. This bounding box output is used to estimate the center position and radius of the landing platform in camera pixels.

Using a camera pixels to world coordinate transform, adapted from Thomas Sundvoll's coordinate transform in his master thesis [34], pixel values for landing platform center and radius can be transformed into quadcopter pose. This transformation utilizes known dimensions of the landing platform, camera intrisincs and camera geometry. This transformation is based on the assumptions that the quadcopter is hovering flat in the air. This is an assumption that the rotation in pitch and roll are zero. The coordinate transform is also is based on the assumptions that the landing platform is oriented flat on the ground, and thereby parallel with the hovering quadcopter. Thus, by locating the landing platform center and radius in the camera frame a pose estimate for the quadcopter can be calculated. The code for the coordinate transform can be seen in Listing 13.

```
def transform_pixel_position_to_world_coordinates(center_px, r_px):
    focal_length = 374.67
    real_radius = 390
    x_0 = IMG_HEIGHT/2.0
    y_0 = IMG_WIDTH/2.0
    d_x = x_0 - center_px[0]
    d_y = y_0 - center_px[1]
    est_z = real_radius*focal_length / r_px
    est_x = -((est_z * d_x / focal_length) + camera_offset_x)
    est_y = -(est_z * d_y / focal_length)
    est_z += camera_offset_z
    return (est_x,est_y,est_z)
```

LISTING 4.1: Coordinate transform from camera frame coordinates to world coordinates. Adapted from [34].

The assumption that the quadcopter is hovering flat in the air is an assumption that is never perfectly satisfied. In order for the quadcopter to move in x- and y-axis it has to roll and pitch. The quadcopter is constantly experiencing small perturbations in roll and pitch. The control signals sent to the drone are sufficiently slow such that the roll and pitch movements for the quadcopter to move are small. This makes it so that the assumptions are not violated too much. If there was wind while running the tests the error margin from the assumptions would be greater. The wind would apply a force, pushing the quadcopter in a direction, and the quadcopter would have to roll or pitch in order to hover still in the air. The assumption that the landing platform is parallel to the quadcopter is an assumption that holds well in the simulated environment. In the simulated environment the DNV-GL ReVolt vessel is not moving at all. When testing with the real ship there might be waves and wind which causes the ReVolt vessel-, and therefore the landing platform, to roll and pitch.

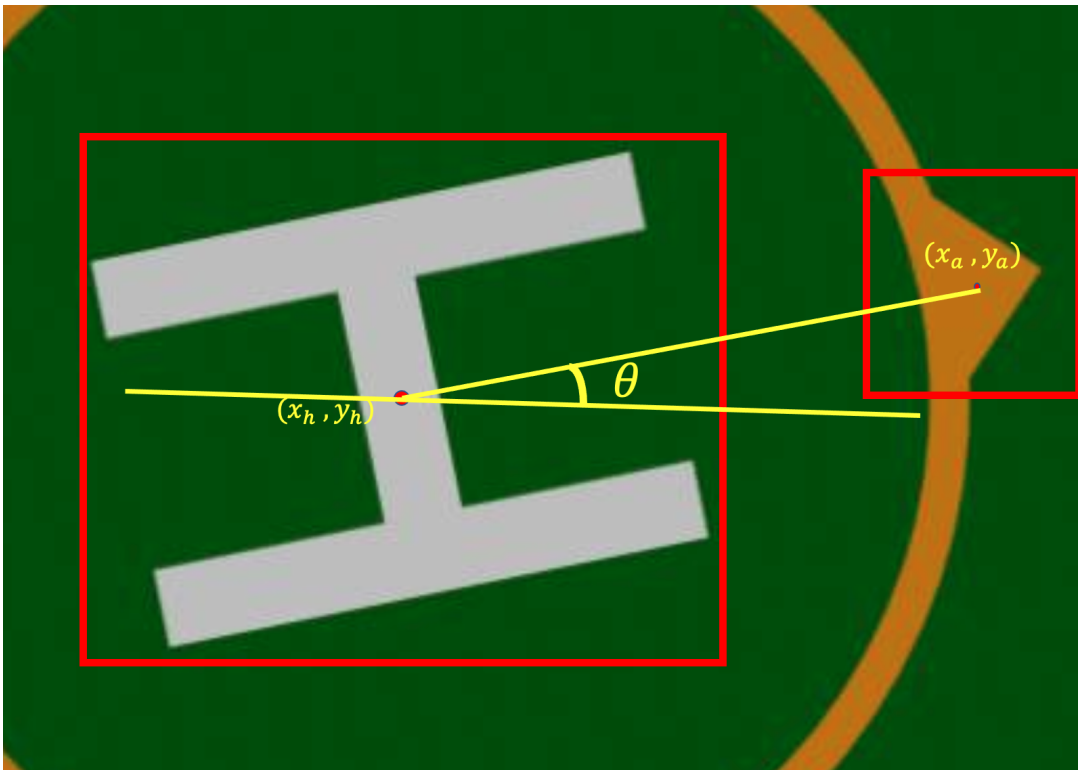


FIGURE 4.2: Calculating theta given center coordinates of bounding box for H (x_h, y_h) and Arrow (x_a, y_a).

The YOLO object detection algorithm may predict that there is more that one instance of a single class in the image. The pose estimator will apply non-max suppression on the bounding boxes, by removing all but the most probable bounding box, such that only one bounding box is used for every class.

4.4.1 Estimation of landing platform rotation

The rotation of the lading platform, θ, is defined as can be seen in Figure 4.2. Zero rotation is defined as when the orange arrow points directly to the right in the camera frame, and a line from the arrow to the center of the landing platform is parallel to the x-axis in the camera frame. Yaw can be estimated by calculating the center of an Arrow bounding box as well as the center of the landing platform. The rotation is found as the atan2 of the difference between the centers of the bounding boxes in x and y direction. Thus, theta is estimated as

$$\theta = atan2(y_a - y_c, x_a - x_c) \tag{4.1}$$

where (x_a, y_a) are center coordinates of bounding box encompassing the Arrow, and (x_c, y_c) are estimated center coordinates of the landing platform

4.4.2 Estimation of landing platform center

The center of the landing platform in located in the center of a perfect bounding box encompassing the Helipad class, as well as in the center of a perfect bounding box encompassing the H. Thus the center of the landing platform can be estimated by finding the center of a bounding box which is drawn around the Helipad or the

H. The center of a bounding box $(\bar{x}_{bb}, \bar{y}_{bb})$ is found in pixel coordinates, given a bounding box coordinates, is found by

$$(w_{bb}, h_{bb}) = (x_{\max} - x_{\min}, y_{\max} - y_{\min})$$
$$(\bar{x}_{bb}, \bar{y}_{bb}) = (x_{\min} + 0.5w, y_{\min} + 0.5h)$$

where $x_{\max}, x_{\min}, y_{\max}, y_{\min}$ are pixel coordinates for the bounding box. w_{bb} and h_{bb} are width and height of the bounding box.

When running the Darknet object classifier the bounding boxes will not be perfectly aligned with the ground truth object. Nor is there a guarantee that the whole of the H or the Helipad are in the image, as can be seen in Figure 4.4. Thus the center of the bounding boxes will be an estimate of the center and may not align with the ground truth center.

If the camera view is taken from an altitude that is so close that is has does not contain the full Helipad, then it is preferable to use the center of a H bounding box to estimate the center, as long as such a bounding box is found. This is because the H is smaller than the Helipad while being located in the center of the helipad, such that the center of the bounding box encompassing the H will be closer to the ground truth center than a Helipad bounding box. If the camera view contains both the full H and the full Helipad, both the centers of the bounding boxes will approximate the ground truth center. If the quadcopter is far away from the landing platform the center of the landing platform is estimated as the center of the bounding box of the Helipad. This is because from far away the H and the Helipad looks very similar, and the Darknet algorithm may wrongly classify the Helipad as the H. When the quadcopter is close to the landing platform, then the Arrow is most likely detected. This is not the case when the quadcopter is far away, as YOLO is known to struggle with detecting small objects. Therefore if the Arrow and the H is detected by the algorithm, then the center is estimated as the center of the H bounding box. Otherwise the center is estimated as the center of the Helipad bounding box.

4.4.3 Estimation of landing platform radius

Bounding boxes of the Helipad and H can be used in order to estimate the radius of the landing platform in the camera frame. The side-lengths of a bounding box surrounding the Helipad is used as an an estimate of the diameter of the landing platform. In the case that the bounding box is not square, the longest side of the bounding box is used as the best estimate for the diameter. This may happen when the quadcopter is so close to the helipad that the camera view does not encompass the whole helipad, or if the YOLO bounding box is not perfect over ground truth. Thus the radius of the landing platform, given a Helipad bounding box is calculated as

$$(w_{\text{bb, Helipad}}, h_{\text{bb, Helipad}}) = (x_{\text{Helipad}_{\max}} - x_{\text{Helipad}_{\min}}, y_{\text{Helipad}_{\max}} - y_{\text{Helipad}_{\min}}) \quad (4.2)$$
$$r_{LP} = \frac{1}{2} \cdot \max(w_{\text{bb, Helipad}}, h_{\text{bb, Helipad}}) \quad (4.3)$$

It is preferable to use a bounding box of the H for estimating the radius of the Helipad given that the quadcopter is close to the landing platform. This is because if the quadcopter is very close to the landing platform, then the camera view will be completely filled with an image of the lading platform. A Helipad bounding box will in that instance cover the entire camera view, and the radius of the landing platform,

estimated using that bounding box, will be estimated to be the complete camera view, and may not predict it to be larger. Thus, when the landing platform fills the camera view the quadcopter would not be able to predict the radius accurately. The H is smaller than the Helipad, and is located in the center of the landing platform. Therefore, an estimate of the radius of the landing platform given a H bounding box will be more accurate than an estimate given the Helipad when the camera view is very close. This is because the H is smaller and is more likely to be contained completely in the camera view than the Helipad. An example of this can be seen in Figure 4.4.

When the quadcopter is far away from the landing platform, the Darknet detector may detect the whole landing platform as the H, so an estimate of the radius given the bounding box encompassing the Helipad will give a more accurate estimate. Thus the same logic is used as for estimating center; that if both the Arrow and the H is detected by the algorithm, then the radius is estimated using the H bounding box. Otherwise the radius is estimated using the Helipad bounding box.

The diameter of the helipad is known to be $0.80m$, and the width and length of the H, (w_H, h_H), are known to be $0.266m$ and $0.177m$ [34]. Thus the relationship between the radius of the landing platform and the height of the H is $r_{LP} = \frac{3}{2}h_H$. From experiments it is seen that the YOLO algorithm draws H bounding boxes that are slightly larger than the H. From experiments it is found that the proper relationship between the radius of the landing platform and the maximum of H bounding box side lengths is 2.65. Thus, the radius of the landing platform, given a H bounding box, is found by

$$(w_{\text{bb, H}}, h_{\text{bb, H}}) = (x_{\text{H}_{\text{max}}} - x_{\text{H}_{\text{min}}}, y_{\text{H}_{\text{max}}} - y_{\text{H}_{\text{min}}}) \tag{4.4}$$

$$r_{LP} = \frac{2.65}{2} \cdot \max(w_{\text{bb, H}}, h_{\text{bb, H}}) \tag{4.5}$$

If the quadcopter is not aligned with the landing platform, the side image of the H will be rotated, meaning that the sides of the H are not parallel to the sides of the camera view. In that case the H bounding box will be larger than the H. An illustration of this can be seen in Figure 4.3, where the height and width of the bounding box are larger than the height and width of the H. By knowing the angle of rotation θ, the known aspect ratio of the H, and the bounding box size (w, h), the height and width of the H in the image (w_H, h_H) can be calculated.

The relationship between the width and height of the bounding box (w_{bb}, h_{bb}) and the width and height of the H (w, h) is found to be

$$\begin{bmatrix} w_{bb} \\ h_{bb} \end{bmatrix} = \begin{bmatrix} sin(\theta) & cos(\theta) \\ cos(\theta) & sin(\theta) \end{bmatrix} \begin{bmatrix} w \\ h \end{bmatrix} \tag{4.6}$$

$$\tag{4.7}$$

Solving for (w, h) , by inverting the matrix, this becomes

$$\begin{bmatrix} w \\ h \end{bmatrix} = \frac{1}{sin^2(\theta) - cos^2(\theta)} \begin{bmatrix} sin(\theta) & -cos(\theta) \\ -cos(\theta) & sin(\theta) \end{bmatrix} \begin{bmatrix} w_{bb} \\ h_{bb} \end{bmatrix} \tag{4.8}$$

This matrix inverse does not exists for certain values of θ. It does not exist when $\theta = k\frac{\pi}{2}$ for any odd integer k, as the determinant of the matrix becomes zero. This makes it impossible to calculate (w, h) explicitly for all values of θ.

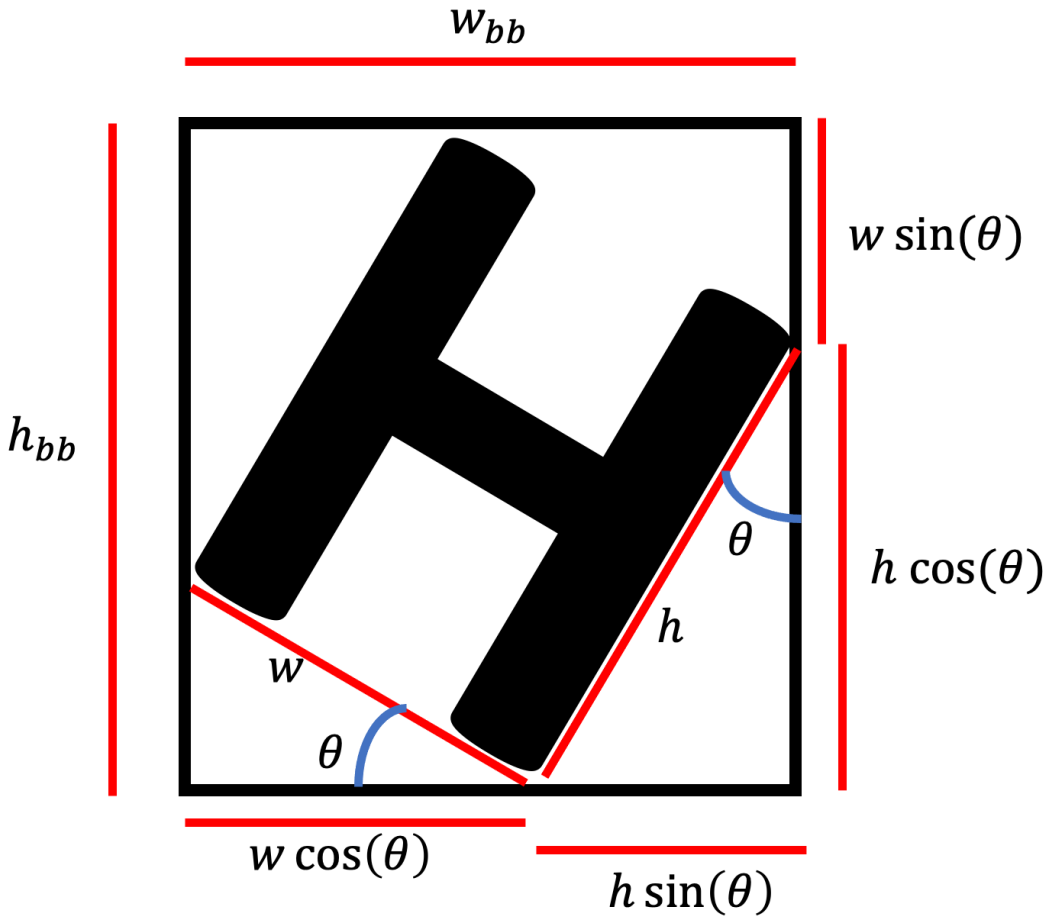


FIGURE 4.3: Calculating original size of H given angle of rotation θ and bounding box size (w_{bb}, h_{bb}).

The dimensions of the H are known. The height and length of the H are $h_H = 0.226m$ and $w_H = 0.177m$ respectively. Thus, the relationship between the side-lengths of the H is such that $h_H = \frac{3}{2} \cdot w_H$. Therefore one can calculate the expected bounding box size (w_{bb}, h_{bb}) for the given rotation θ and the known aspect ratio by

$$\begin{bmatrix} w_{bb} \\ h_{bb} \end{bmatrix} = \begin{bmatrix} cos(\theta) & sin(\theta) \\ sin(\theta) & cos(\theta) \end{bmatrix} \begin{bmatrix} 2 \\ 3 \end{bmatrix} \tag{4.9}$$

Dividing the results, w_{bb} and h_{bb} , by $w = 2$ and $h = 3$ respectively one gets the corresponding factors (k_w, k_h) for scaling the H bounding box to the size of the H. Table with notable scaling values can be found in Appendix A. Thus the radius of the landing platform given a H bounding box and angle of rotation θ is found by

$$(w_{\text{bb, H}}, h_{\text{bb, H}}) = (x_{\text{H}_{\max}} - x_{\text{H}_{\min}}, y_{\text{H}_{\max}} - y_{\text{H}_{\min}}) \tag{4.10}$$

$$k_w = \frac{1}{2}(2cos(\theta) + 3sin(\theta)) \tag{4.11}$$

$$k_h = \frac{1}{3}(3sin(\theta) + 2cos(\theta)) \tag{4.12}$$

$$(w_{\text{bb, H}}, h_{\text{bb, H}}) = (w_{\text{bb, H}} \cdot k_w, h_{\text{bb, H}} \cdot k_h) \tag{4.13}$$

$$r_{LP} = \frac{2.65}{2} \cdot \max(w_{\text{bb, H}}, h_{\text{bb, H}}) \tag{4.14}$$

The complete algorithm for estimating center and radius of the landing platform given bounding boxes is given in algorithm 1

Algorithm 1: Estimating center and radius of the helipad in camera pixel coordinates from YOLO bounding box output

Input: list of bounding boxes for matches
Output: estimated center (x, y) and radius r of Helipad in camera pixel coordinates.
if *matches contains H and matches contains Arrow* **then**
  $\theta \leftarrow$ estimate_rotation(H_{bb}, $Arrow_{bb}$);
  $(w_H, h_h) \leftarrow downscale\_H\_by\_rotation(w_H, h_h, \theta)$;
  $r \leftarrow 0.5 \cdot 2.65 \cdot max(w_H, h_H)$
  $x, y \leftarrow (x_H, y_H)$;
else if *matches contains Helipad* **then**
  $x, y \leftarrow (x_{Helipad}, y_{Helipad})$;
  $r \leftarrow 0.5 \cdot max(w_{Helipad}, h_{Helipad})$;
else
  return *None*
end

In the case that the model struggles to recognize the arrow at all, then an alternative algorithm is proposed in order to make automated landing possible. This alternative method is relevant for when running the results with YOLOv4-tiny. This YOLO model is known to struggle to recognize small objects, and may therefore struggle to classify the arrow. This alternative algorithm will estimate the center and radius of the helipad using the bounding box of the H as long as it is available. This is because when the quadcopter is close to the landing platform an estimate using the bounding box of the Helipad may be very inaccurate. Since YOLOv4-tiny may struggle the recognize the arrow at any heights, the algorithm is chosen to be

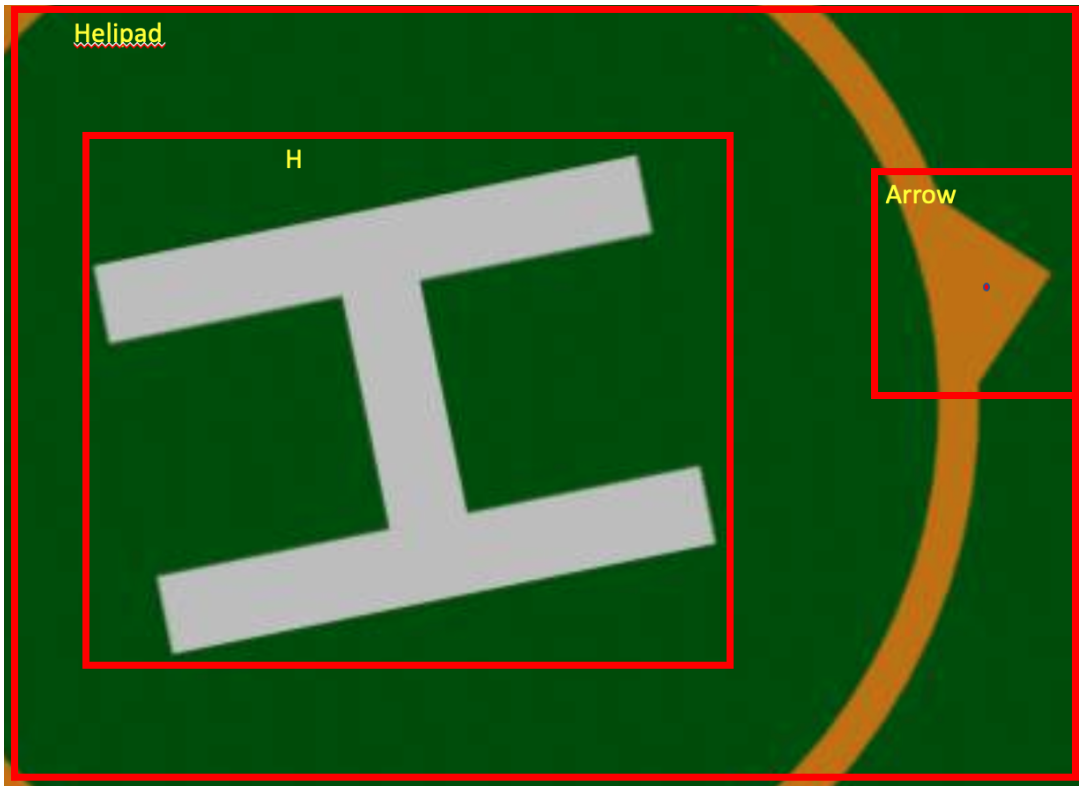


FIGURE 4.4: When the drone is close to the helipad, the whole helipad is not in the picture. The bounding box of the Helipad does not enclose the whole Helipad. Therefore an estimate of the size of the landing platform using the bounding box of the H will give a more accurate estimate.

algorithm 2 when running YOLOv4-tiny.

Algorithm 2: Estimating center and radius of the helipad in camera pixel coordinates from YOLO bounding box output

Input: list of bounding boxes for matches
Output: estimated center (x, y) and radius r of Helipad in camera pixel coordinates.

if *matches contains H* **then**
  if *matches contains Arrow* **then**
    $x, y \leftarrow (x_H, y_H)$;
    $\theta \leftarrow$ estimate_rotation$(H_{bb}, Arrow_{bb})$;
    $(w_H, h_h) \leftarrow downscale\_H\_by\_rotation(w_H, h_h, \theta)$;
    $r \leftarrow 0.5 \cdot 2.65 \cdot max(w_H, h_h)$
  else
    $x, y \leftarrow (x_H, y_H)$;
    $r \leftarrow 0.5 \cdot 2.65 \cdot max(w_H, h_H)$
  end
else if *matches contains Helipad* **then**
  $x, y \leftarrow (x_{Helipad}, y_{Helipad})$;
  $r \leftarrow 0.5 \cdot max(w_{Helipad}, h_{Helipad})$;
else
  return *None*
end

4.5 Traditional Computer Vision Method for pose estimation

A pose estimator using traditional computer vision methods is also used for pose estimation in this thesis. This pose estimator is taken from Thomas Sundvoll's master thesis about pose estimation for autonomous quadcopter flight [1]. Sundvoll's pose estimator is based on locating certain elements of the landing platform using traditional computer vision methods.

The input image is first subject to multiple color segmentations, isolating out green pixels, orange pixels and white pixels as three different images. These colors are colors that are present on the landing platform, and therefore segmenting out these colors reduces unnecessary information in the image. Harris Corner detection and Canny Edge detection are used to isolate aspects of the landing platform, such as the round circles, the corners of the H, and the orange triangle which is the arrow.

By utilizing known aspects of the landing platform, these isolated aspects that are found can be used to locate the center and radius of the landing platform in pixel values. The rotation of the landing platform can also be determined using the location of the arrow compared to the center of the landing platform. The pixel values of the center and radius of the landing platform is transformed into world coordinate position using a coordinate transform using camera geometry, in order to get an estimation of quadcopter position. The estimated rotation is combined with this position estimate, thus producing an estimate of quadcopter pose.

4.6 Combining computer vision methods

The results from [1] show that the traditional computer vision pose estimator is quite reliable in the simulated environment. It is efficient on CPU, and is able to detect pose from multiple altitudes. This technique requires tuning by finding proper color segmentation thresholds that match the colors of the landing platform as they appear to the camera. However, with the proper tuning the estimator is quite good. The need for proper tuning makes this computer vision system less robust to changes in the visual environment, and may fail if the conditions are different than what it is tuned for.

The Darknet pose estimator can be trained to be general and quite invariant to changes in visual conditions if it is trained on a dataset which contains data from multiple- and challenging visual conditions.

A combination of the two computer vision techniques may prove to be a more robust pose estimator than either of the techniques independently. The two pose estimators are based on different computer vision principles, and might handle challenging visual conditions differently. In the scenario where one method fails to detect the landing platform and therefore fails to estimate the current quadcopter pose the other algorithm might detect the platform and be able to estimate the current pose.

Both method provide an estimates for pose, on the form $(x, y, z, 0, 0, \theta)$. Roll and pitch are estimated to be zero at all times. This is an underlying assumption for both estimators, and is required for the current pixel-to-world coordinate transform.

The estimates from the two methods are not synchronized in time. The estimates are output from the methods as soon as they are estimated. The estimates put into and filtered by a filter for fusing the estimates. The estimates from both methods are passed through the same filter. This filter is a moving median averaging filter (MMA-filter). This filter is a combination of a moving median filter and a moving

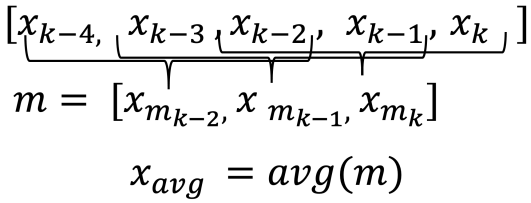


FIGURE 4.5: Moving median average filter, with a median filter size of 3 and average filter size of 3. x_{avg} is the filtered output estimate that is passed onto the Dead Reckoning Module

average filter. The moving median filter outputs the median of the last M estimated values. Such a filter filters out value spikes of the estimates. The output from this moving median filter is put into a moving average filter. This moving average filter averages the last N outputs from the moving median filter. This filter suppresses high frequency noise by smoothing out the estimates. An illustration of a MMA filter with size M = N = 3 can be seen in Figure 4.5.

The estimates from both computer vision pose estimator methods are generated concurrently and passed through this filter. Thus the output will be based on a filtered combination of the two pose estimation methods.

4.7 Experiment design

In order for the combined pose estimator to be a robust pose estimator then both the TCV pose estimator and the Darknet pose estimator need to be able to estimate pose reliably. The TCV pose estimator presented by Sundvoll [1] works reliably and well in the simulated environment. Therefore the experiments need to be designed such that the following is tested

- Whether the DL pose estimator is sufficiently robust, fast and accurate for use in autonomous missions
- Whether the combined pose estimator is sufficiently robust, fast and accurate for use in autonomous missions.
- Whether the combined pose estimator is sufficiently robust to perform well in challenging visual conditions.

In order to test this the quadcopter will perform autonomous missions utilizing the pose estimator algorithms presented in this thesis. The pose estimates will be connected as feedback to the controller which controls the quadcopter motors.

Autonomous missions are implemented as a series of reference points for a controller controlling the quadcopter pose. The quadcopter controller is a PID controller, which works by comparing the estimated pose of the quadcopter to a reference point. The PID controller seeks to control the quadcopter towards the reference point. The PID controlled calculates an error between the current pose and the reference point, called proportional error e_p. It also calculates the integral of the error, e_I as well as the derivative of the error e_d. Control actuation is calculated as a sum of these errors, scaled by respective scaling factors found through experimentation.

When estimated pose is within an error margin of the setpoint, the next setpoint in the mission is applied as reference to the controller. The error margin used in the experiments is $\pm 0.01m$.

The estimated pose from the computer vision pose estimator can be compared to ground truth quadcopter pose which is available from the gazebo simulator. By comparing the estimated pose to ground truth the accuracy of the pose estimates can be assessed.

The mission experiments that are tested with the pose estimator are

- Stationary hovering 1m above the platform
- Stationary hovering 5m above the platform
- Autonomous landing on the platform

4.7.1 Stationary hovering

A hovering mission is performed by applying one setpoint to the controller. The controller will attempt to control the quadcopter towards this point, and stay there. The result is that the quadcopter will attempt to hover stationary in the air at the given setpoint.

It is important that the hovering point is a point where the landing platform is within the camera view of the quadcopter. Otherwise the quadcopter will not be able to estimate the current pose using the proposed computer vision pose estimator.

This experiment test whether the pose estimator is sufficiently accurate for real-time control of the quadcopter. The stationary hovering test will be performed from an altitude of 1m and 5m.

Performing the stationary hovering mission tests whether the quadcopter pose estimator is accurate and reliable for use in real-time control of the quadcopter. Performing the stationary hovering test at 5m challenges the robustness of the pose estimator in a more challenging scenario. The landing platform shrinks in the camera view as the quadcopter gains altitude. Detecting features in a smaller object may be more challenging for the computer vision algorithm, and therefore this is a test of the robustness of the pose estimator in a more challenging visual environment.

4.7.2 Autonomous Landing

In order to perform automated landing, the quadcopter controller is fed a series of reference points.

The quadcopter can initiate the landing mission at different initial poses, as long as the pose estimator is able to estimate the current pose. When the automated landing mission is initiated the controller will control the quadcopter to the first setpoint, which is $(x = 0, y = 0, z = 2.0)$. This entails stationary hovering 2m above the platform.

When the pose estimates reaches an estimate which is within the error margin of this setpoint, the controller switches the next setpoint, which is $(x = 0, y = 0, z = 0.20)$ which means hovering 0.20m above the platform. When this setpoint is reached the drone will turn off the motors, and drop down to the landing platform in order to complete the landing mission.

An autonomous landing mission is a test of whether the pose estimator is sufficiently robust in order to be used for the quadcopter performing autonomous missions. Pose estimates during a landing mission need to be frequent and accurate for the landing mission to be successful. This mission also tests whether the pose estimator is able to accurately predict the quadcopter pose when the quadcopter is close to the landing platform.

Chapter 5

Results

5.1 Training of Darknet models

The YOLOv4 model and YOLOv4-tiny model are trained on the dataset. The models are trained by transfer learning, using pre-trained weights. The initial weights that are used for transfer learning are the official YOLOv4 and YOLOv4tiny weights trained on the COCO dataset [37]. The model is trained to recognize the three classes, *Helipad*, *H*, and *Arrow*, using the dataset collected from the Gazebo simulator. The dataset consists of 1083 images, that are augmented into 2599 images.

These images are resized to a proper YOLO format of 416x416, by squaring the images with black borders on the top and bottom, to make the image square, and scaling down the resolution. Darknet performs this resizing and scaling before inference on all images. This is therefore done on the dataset used for training.

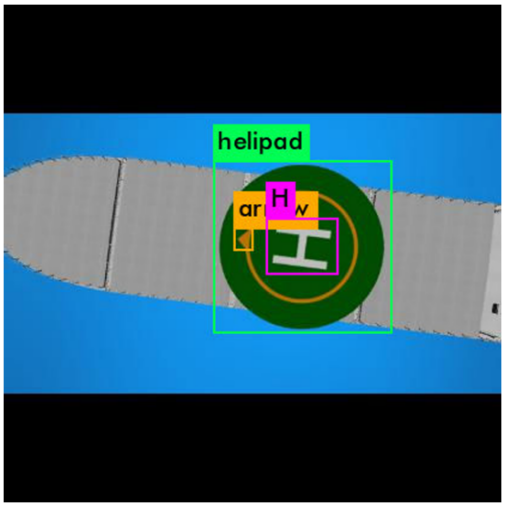
5.1.1 YOLOv4-tiny

The YOLOv4-tiny model is trained on the dataset with the following parameters:

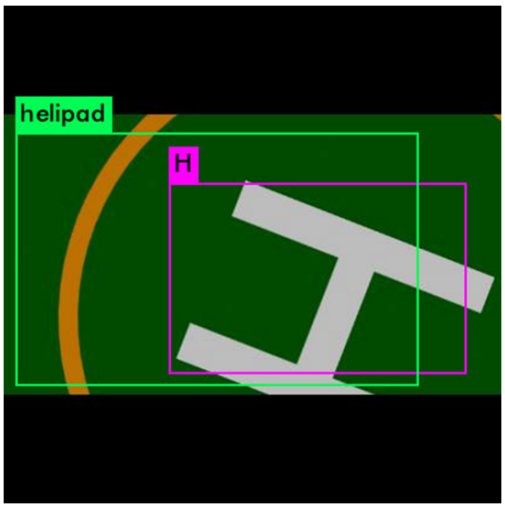
- Decay: 0.005
- Momentum: 0.9
- Batch size: 64
- Learning rate: 0.00261
- No. of Batches in training: 6000
- Image resolution: 416x416

After being trained using these parameters the YOLOv4-tiny model achieved a mAP of 78.5 % on the test set. This is quite low, and does not point towards a robust model. YOLOv4-tiny is a very small DNN model, and is not created for being the most accurate. The main advantage of YOLOv4-tiny is quick inference times. In **??** it is seen that YOLOv4-tiny has significantly lower mAP-, but has higher FPS than the other models.

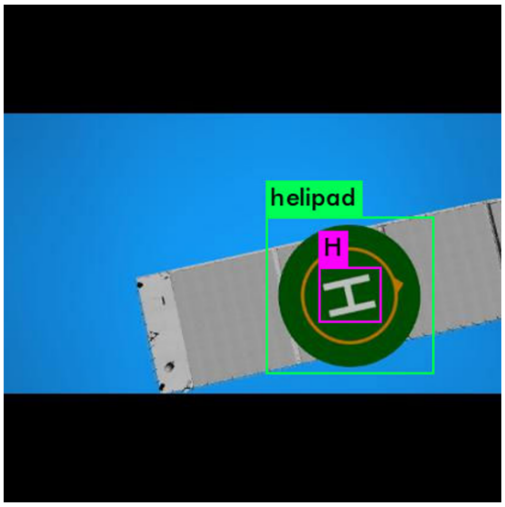
The simulator environment is an unchanging environment. The images in the training set is therefore quite similar to the images in the test set. Because of this the model is expected to perform well on the test set, and be proficient at identifying the classes. The dataset the model was trained with is quite small however. A small model such as YOLOv4-tiny requires less data when training than a larger model such as YOLOv4. This is because it has fewer neurons and layers that need to be trained. Despite this, the dataset is small, and this may be one of the reasons for the low mAP.



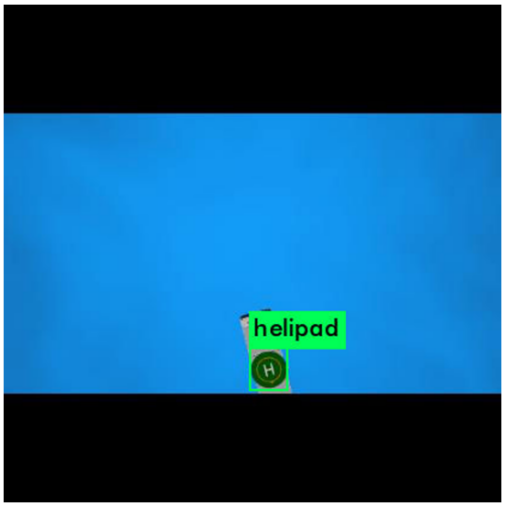
(A) A successful inference



(B) Close up inference. Bounding boxes misaligned.



(C) Does not recognize the arrow



(D) Does not recognize the H, nor the Arrow

FIGURE 5.1: YOLOv4-tiny inference on test images

Figure 5.1 shows output images after inference using a trained YOLOv4-tiny model on images in the test set. A successful inference is shown in Figure 5.1a where the model is able to classify the different classes well. Figure 5.1b is an example of inference where the quadcopter is close to the landing platform. The bounding boxes are not well aligned with ground truth. Figure 5.1c the model does not recognize the Arrow. YOLOv4-tiny is known to struggle with classifying small objects, and thus it struggles to classify the Arrow. From a high altitude the model is not able to recognize either the Arrow, nor the H, as can be seen in Figure 5.1d.

5.1.2 YOLOv4

The YOLOv4 model was trained on the dataset with the following parameters:

- Decay: 0.005
- Momentum: 0.95

- Batch size: 64
- Learning rate: 0.001
- No. of Batches in training: 6000
- Image resolution: 416x416

The resulting mAP of the YOLOv4 model on the test data is 86.9 %. This is not very high, and a robust model is expected to achieve a higher mAP. The simulator setting is a visual environment that is quite stable, and the images in the test set is very similar to the images the model is trained on. The dataset is quite small, and YOLOv4 is a large model, consisting of 52 convolutional layers and one fully connected layer. A large model requires a proportionally large dataset for training. Although transfer learning using pre-trained weights decreases the required number of images in a dataset when training, the size of the dataset on lower end. Increasing the number of images in the dataset would most likely have increased the mAP of the trained model.

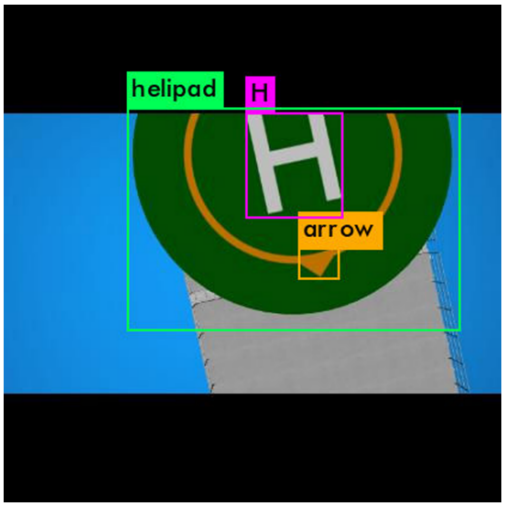
Figure 5.2 shows examples of inference from the trained YOLOv4 model on test images. The results are quite a lot better than the inference results using YOLOv4-tiny. YOLOv4 is significantly better at detecting small objects than YOLOv4-tiny and is able to recognize the Arrow and the H from high above, where YOLOv4-tiny is not (5.1d, 5.2b). This is significant, as the Arrow is quite a small object, and the Helipad and the H becomes small when the quadcopter is high above the landing platform. Thus the YOLOv4 model is more robust than YOLOv4-tiny, as the larger model is able to classify the objects of interest from further away than the smaller model. Classifying the Arrow is required in order to estimate the rotation, and YOLOv4 is therefore more capable of estimating rotation when far away from the landing platform than YOLOv4-tiny.

YOLOv4 detected some of the classes multiple times (5.2c). This is not desirable, and may be one of the reasons for the low mAP. The model is trained to recognize the Helipad and the H from many different altitudes. Therefore the model is trained to detect the classes at varying sizes. A problem with this is that the white H is part of the Helipad class and the parts of the Helipad class is the H. The model struggles to identify whether the landing platform is a large H or the H is a small landing Helipad. Training the model on a larger dataset may to reduce such errors. The model was generally more confident in the correct predictions than the incorrectly classified bounding boxes when there was double detections. The most confident bounding box for each class is chosen as the proper prediction, so in most cases if YOLOv4 predicts one class multiple times the most accurate prediction is selected.

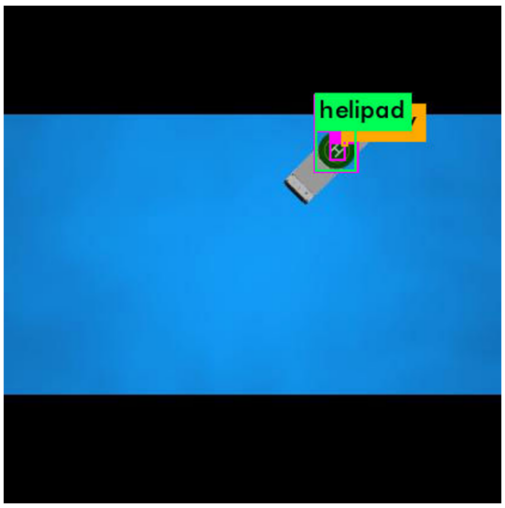
YOLOv4 is more precise with the bounding box placements than YOLOv4-tiny. In Figure 5.1b the bounding boxes are misaligned with the objects in the image, while in Figure 5.2d and 5.2a the placement of the bounding boxes is more precise.

When running the YOLOv4 model on the simulator in real-time, hovering 1m above the helipad, the model predicts the bounding boxes with confidences in a range between

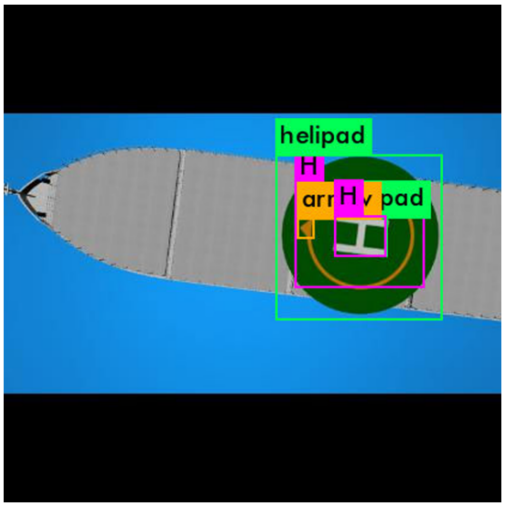
- Helipad: 99-100% confidence
- H: 85-95% confidence
- Arrow: 40-65% confidence



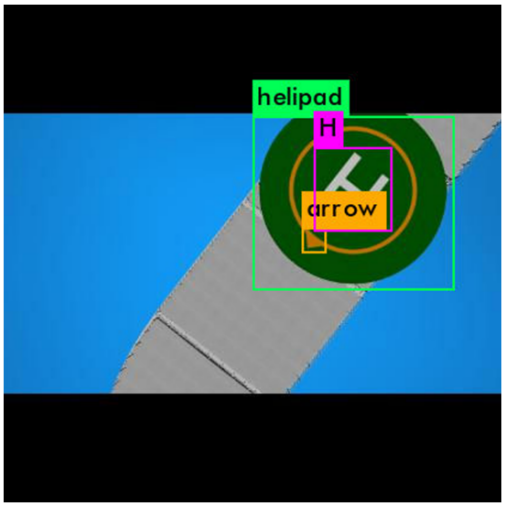
(A) Successful detection of part of Helipad.



(B) YOLOv4 detecting all classes from far away. Double detection of H.



(C) Double detection of H and Helipad.



(D) Detecting all classes well.

FIGURE 5.2: YOLOv4 inference on test images.

The model is less confident in the p-redictions of the Arrow than the H, and less confident in the predictions of the H than the Helipad. YOLO is better at classifying larger objects than small ones, and this can be seen in the results as well. The Arrow may also be hard for the model to classify as it has to be distinguished from the orange circle surrounding the H. The orange circle is the same color as the Arrow, and this may therefore make classification of the Arrow difficult. The other two classes are distinctive in the image, being of a distinctive color and having distinctive shapes, and may therefore be easier to classify than the Arrow.

5.1.3 YOLO inference rate

The inference rate for the Darknet pose estimator while running the experiments can be seen in Table 5.1. These inference rates were achieved on a OptiPlex 7040 with an Intel® Core™ i7-6700 CPU @ 3.40GHz 8, while the Gazebo simulator environment is running.

The YOLOv4 model has a very low inference rate when running on a CPU. The inference frequencies, at different resolutions, are all below 0.3 Hz. This is too slow to be used as pose estimate for real-time control of the quadcopter. With an inference rate of 0.3 Hz the pose estimates would be published less frequently than once every 3 seconds, which is not enough for controlling the quadcopter.

YOLOv4-tiny is significantly faster than it's larger counterpart, but is still slow on the CPU. Using images at a resolution of 256x256 pixels as input YOLOv4-tiny produces estimates at 0.5 Hz. This is very slow for a system like the quadcopter. The estimates are delayed as well. That means that the pose estimates are an estimate of where the quadcopter was when the inference started. With an inference time of $\frac{1}{0.5Hz} = 2s$ that means that the pose estimate is an estimate of where the quadcopter was 2 seconds earlier. This may cause the system to be unstable. A control command that is based on a pose estimate that is 2 seconds old may not be the correct command for the current environment. The quadcopter has fast dynamics, and may move a lot in a second or two. Using a pose estimates with such a delay may cause the quadcopter to become unstable.

The YOLOv4-tiny model at a very low resolution of 128x128 manages to produce pose estimates at a frequency of about 2.3 Hz. This is still quite low, but is significantly faster than the other results on CPU.

The problem with reducing the resolution in order to get a higher inference rate is that the model will lose predictive capabilities. If the resolution is very low, such as at 256x256 or 128x128, the model may struggle to recognize the smallest features. This is because the will be blurry and small features will be hard to detect. There is therefore a trade-off when reducing the resolution, that the inference speed goes up, while the predictive capabilities goes down. The mAP results from training the models was results based on a resolution of 416x416. That mAP was in the lower end of what is desired for the model to have, so it is not desirable to reduce the resolution. However, when running on CPU the inference rate of a resolution of 416x416 is way too low for any real-time application of quadcopter control, as the frequency is about 0.2 Hz.

Li et al. presents in [41] an optimized version of YOLOv4 containing 74% less parameters. Fewer parameters in a DNN model results in faster inference times, as less calculations have to be performed during inference. This reduction in parameters did not decrease the precision of the optimized model, which achieved a precision increase of 2.6% compared to the original YOLOv4 model. Therefore, using this proposed model, instead of the original YOLOv4 model, may be an option if GPU is not available.

Running the DNN on a GPU the inference rate is much higher than when running on a CPU. This can be seen in Table 5.1. These inference rates were achieved when running the experiment on a GTX Ti GPU, while the Gazebo simulator is running at the same time. These inference rates are very high. The model was able to produce estimates at about 190 Hz using the fastest model at the lowest resolution. The most accurate model, YOLOv4, on the highest resolution 416x416, achieved over 50FPS. This is faster than the estimation rate of the TCV pose estimator, which produces pose estimates at a rate of 5 - 10Hz. Autonomous missions using the TCV pose estimator [1], were completed successfully. Therefore 50 Hz is more than enough for real-time control of the quadcopter.

Because inference times were so good with the GPU images with a higher resolution may be tested in order to gain better predictive capabilities for the model.

It is preferable to run YOLO inference on a GPU when performing real-time control of a quadcopter because of the increase in inference rate. When using a CPU

Model	Processor	Resolution	Hz
YOLOv4	CPU	416x416	< 0.1 Hz
YOLOv4-tiny	CPU	416x416	∼0.2 Hz
YOLOv4	CPU	256x256	< 0.1 Hz
YOLOv4-tiny	CPU	256x256	∼ 0.5 Hz
YOLOv4	CPU	128x128	∼ 0.3 Hz
YOLOv4-tiny	CPU	128x128	∼ 2.3 Hz
YOLOv4	GPU	416x416	∼ 50 Hz
YOLOv4-tiny	GPU	416x416	∼ 100 Hz
YOLOv4	GPU	256x256	∼ 140 Hz
YOLOv4-tiny	GPU	256x256	∼ 170 Hz
YOLOv4	GPU	128x128	∼ 150 Hz
YOLOv4-tiny	GPU	128x128	∼ 190 Hz

TABLE 5.1: Inference time results for different Darknet models while running the Gazebo simulator

the worst performing model, YOLOv4-tiny, on the lowest resolution, was the only model which was able to perform inference at a rate that was above 2 Hz. On a GPU it is possible to use the best model with higher resolution images.

5.2 Simulation results on CPU

It is interesting to analyze the results from running the experiments on a CPU because a GPU may not be available while experimenting with the physical quadcopter. It is key to understand the limits of a CPU for operations like this, and whether a GPU is strictly necessary for real-time detection of the landing platform, or whether it is feasible to perform such missions with CPU processors.

The alternative algorithm for estimating center and radius is used, the algorithm 2, when running the results on CPU, as the YOLOv4-tiny model at the lowest resolutions struggles to classify the arrow. YOLOv4-tiny is the model that were able to produce estimates at a rate that is t to run the experiments. Therefore the method of estimating center and radius was chosen to be algorithm 2 when running on the CPU.

5.2.1 Stationary hovering test with Darknet pose estimator on CPU.

The stationary hovering test are performed by setting a reference point for the quadcopter controller, and letting that be constant during the test. The hover tests are performed with at reference point $(x = -0.10m, y = 0.0m, z = z_r)$, where $z_r = 1m$ or $z_r = 5m$. The stationary hovering experiments are tested with different pose estimator systems at different resolutions.

The Darknet pose estimator is used for pose estimation, with the model YOLOv4-tiny, with image resolution of 256x256. This model achieves a pose estimation frequency of 0.5 Hz on the CPU. This is very low. Because of the low pose estimation frequency the median filter size is set to be 1 and the averaging filter size is set to be 1. This is because filtering creates a delay in the system, as the newest estimates are filtered together with older estimates. This is not desirable when the estimate frequency is so low.

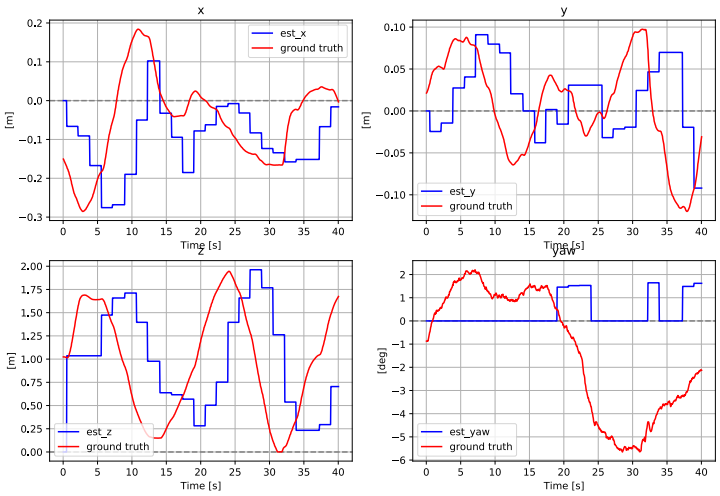


FIGURE 5.3: 1m Hover test on CPU, using YOLOv4-tiny at 256x256 resolution. Darknet used for pose estimation. Figure shows position estimates vs ground truth.

The estimated variables x, y, z, θ for this are plotted versus ground truth values in Figure 5.3. The quadcopter experienced large oscillations in altitude, which can be seen as the ground truth line in the z-axis. The oscillations were increasing in amplitude, and the quadcopter crashed into the landing platform 30 seconds into the test. Thus, the system did not manage to hover in a stable position using this pose estimator. The pose estimates are not sufficiently accurate in order for the controller to stabilize the quadcopter. The estimates do approximate the ground truth values quite well, however they are delayed about 2 seconds in time. A system may be able to handle a feedback delay as long as the feedback delay is significantly shorter than the response time for the system. The feedback delay is significant in relation to how quickly the state of the system changes. Therefore the quadcopter was not able to use this pose feedback to achieve stability.

Performing the stationary hovering test with $z_r = 5m$ the quadcopter performed slightly better than when hovering closer to the platform. The pose estimates and ground truth during this test can be seen in Figure 5.4. The quadcopter started oscillating in altitude. The oscillations had an amplitude of about 1.5m, which is quite significant. The quadcopter controller was not able to stop the oscillations because of the feedback delay in the pose estimates. The oscillations in altitude are slightly increasing in size, and if the hover test was run for longer amount of time the oscillations might get large enough to crash in the water or the landing platform, although this has not been tested. The

While there is a feedback delay, the estimates x- and y-axis the were able to track the ground truth sufficiently in order to achieve stability in x- and y-axis. The quadcopter does not move far away from the reference points of $x = -0.10m$ and $y = 0.0m$. The controller gains for errors in x- and y-axis are low, and therefore the quadcopter reacts slowly to position errors in x- and y-axis. This makes it so that the feedback delay is less significant, and the quadcopter controller is able to stabilize the quadcopter close to the reference points.

The model was not able to recognize the Arrow from that distance, and thus did not estimate the yaw.

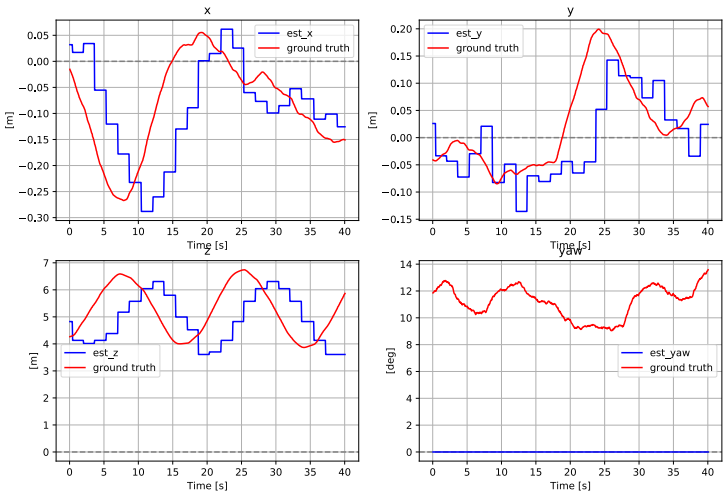


FIGURE 5.4: 5m Hover test on CPU, using YOLOv4-tiny at 256x256 resolution. Darknet used for pose estimation. Figure shows position estimates vs ground truth.

The long time delay seems to matter less when the quadcopter is further away from the landing platform. This may be because the landing platform moves more slowly in the camera frame when the quadcopter is further away. When hovering at 5m there is more room for error than with hovering at 1m, because the quadcopter can more further while the landing platform stays in the camera frame. When hovering at 1m the quadcopter does not have to move far away before the helipad is out of the frame, while at 5m it has to move a lot further away. This makes it so that the quadcopter has more time to recover from a bad trajectory. This might be the reason that the quadcopter performed slightly better while hovering at $z_r = 5m$ crashing sooner when hovering at $z_r = 1m$.

By reducing image resolution to 128x128 YOLOv4-tiny achieves significantly shorter inference time, producing pose estimates at about 2.3Hz. The stationary hovering test is $z_r = 1m$ using YOLOv4-tiny with a resolution of 128x128. The estimated pose and ground truth during test can be seen in Figure 5.5.

The increased inference frequency with the lower image resolutions results in pose estimates that track the ground truth closer than compared with the previous tests. Using these estimates as feedback the quadcopter is able to stabilize itself around the reference point. Deviations from the reference points in x- and y- axis are at most 10cm, and the estimated position for x and y tracks the ground truth with maximum errors of 2cm during the test. The estimated position in z tracks the ground truth sufficiently well in order for the quadcopter to hover at the reference point.

The problem with reducing the resolution to such a low resolution is that the model is less accurate in the classifications. YOLOv4-tiny is already a small object classification network that struggles with recognizing small objects. By reducing the resolution the model struggles even more. At 1m, YOLOv4-tiny at 128x128 was not able to recognize the Arrow, and thus did not estimate the yaw. It had no problem with recognizing the Helipad and the H from that height, and was able to achieve stable hovering.

The stationary hovering test was performed with $z_r = 5m$ with YOLOv4-tiny

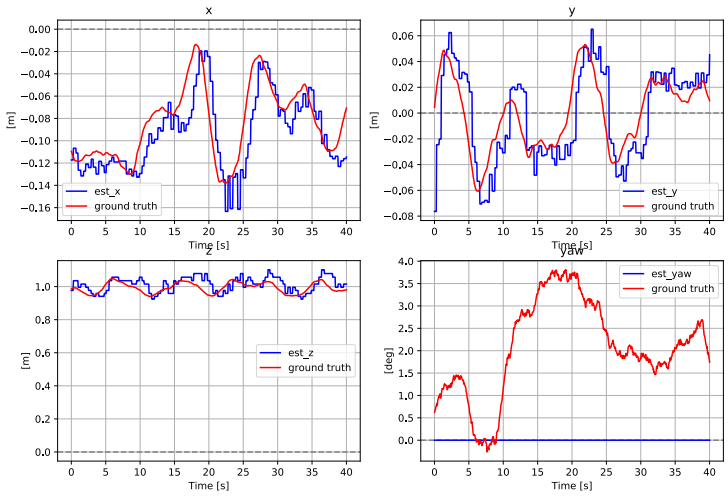


FIGURE 5.5: 1m Hover test on CPU, using YOLOv4-tiny at 128x128 resolution. Darknet used for pose estimation. Figure shows pose estimates vs ground truth.

with 128x128 resolution. The results were not impressive. The classification model was not able to recognize the Helipad, nor any of the other classes, and thus was not able to produce any pose estimates.

The model did not recognize the landing platform from that altitude because it is a very small model, and the input images were of very low resolution. Thus, the model on such a resolution is not a very robust model, as it was not able to handle pose estimation from an altitude of 5m.

The hovering test was not performed using the YOLOv4 algorithm with a CPU processor This is because the inference frequency for that model is too slow for real-time control of the quadcopter, even at the lowest resolution 128x128 (5.1). YOLOv4 is better than YOLOv4-tiny at recognizing small and medium-sized objects, and has a higher mAP overall. Therefore it would be preferable to be able to use that model of the tiny model. YOLOv4 would be able to recognize the landing platform from a greater height than it's smaller counterpart, as well as locating the Arrow which YOLOv4-tiny struggles with. Locating the arrow would facilitate for a more accurate pose estimate as well as making it possible to estimate yaw. On a CPU that was not feasible when the inference was so slow.

5.2.2 Stationary hovering test with combined methods estimate on CPU

Both the DL pose estimator and the TCV pose estimator estimate the quadcopter pose concurrently. A combined estimate is created by feeding estimates from both estimation methods into a moving median averaging filter, using a median filter size of 3 and averaging filter size of 3. The output from this filter is the combined estimate. This combined estimate is used as pose estimate for the quadcopter. Using the combined methods as pose feedback the stationary hovering test is performed with $z_r = 1m$. For this test the YOLOv4-tiny with a resolution of 128x128 was used. The estimated pose can be seen in Figure 5.7 where the combined estimate is plotted against the quadcopter's ground truth pose during the test. The filtered output

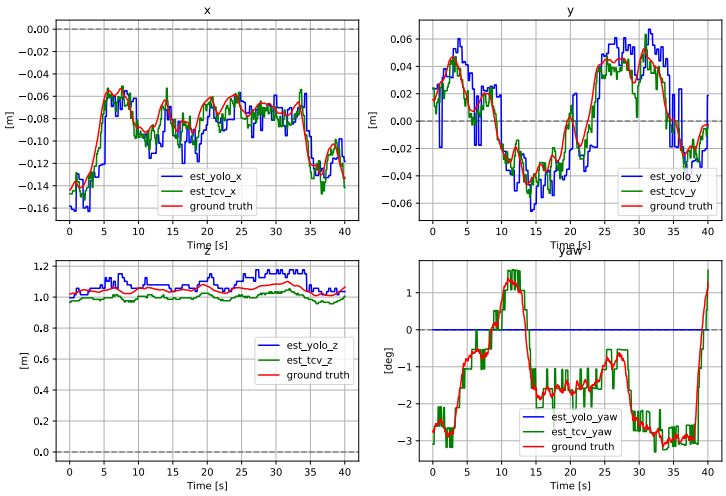


FIGURE 5.6: 1m Hover test on CPU, using YOLOv4-tiny at 128x128 resolution. Filtered combination of methods used for pose estimation. Figure shows position estimates from both methods vs ground truth.

tracks ground truth quite well, and the quadcopter is able to utilize this pose estimate in order to hover in a stable manner around the reference point.

In Figure 5.6 the estimates from the different methods are plotted versus ground truth. The TCV estimates are published at a frequency that is higher than the estimates from the DL method. The TCV method publishes estimates at a frequency of about 5-10 Hz, depending on whether the YOLO method is running concurrently or not. Using the moving median averaging filter the Darknet estimates will therefore mostly filtered out by the median filter. This happens because the TCV method outputs many similar estimates at a high frequency, and thus the median filter will filter out the DL estimates. By comparing Figure 5.7 and Figure 5.6 one can see that the filtered output is similar to the TCV estimates, although more smooth. That is a disadvantage of using a median filter, that the fastest method is prioritized. When running on a GPU the DL estimates are published at a frequency which is much higher than the TCV methods. (5.1). Thus if running on a GPU the Darknet pose estimator would be significantly faster that the TCV estimator, which is estimating pose at about 10 Hz, and the median filter would filter out the TCV estimates. A solution for this is to restrict the fastest method to a maximum pose estimation rate. This would make the filtered estimate become a mix of the two results. This is not a feasible solution when the update frequency of the slowest method is so low.

5.2.3 Automated Landing Using Darknet pose estimator on CPU

An automated landing mission is performed using the Darknet pose estimator on a CPU. The initial pose of the quadcopter mission is hovering at $z_r = 1m$. The quadcopter is fed setpoints of $z_r = 2.0m$ and $z_r = 0.20m$, and a signal to turn motors off when it reaches the final setpoint. $x_r = 0.0m$ and $y_r = 0.0m$ for the complete mission. This test is performed using a moving median filter size of 1 and averaging filter size of 1. Thus, filtering is turned off for this test. This is done because filtering over multiple measurements introduces a delay in the estimate. When the update frequency of the estimates is low, more accurate estimates are achieved by removing

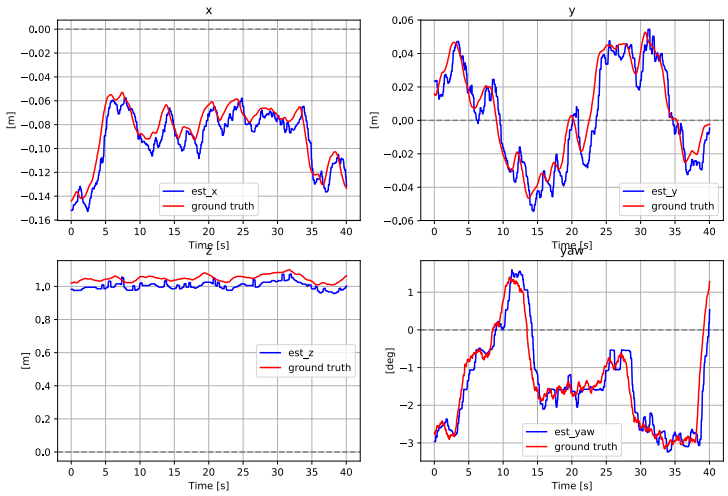


FIGURE 5.7: 1m Hover test on CPU, using YOLOv4-tiny at 128x128 resolution. Filtered combination of methods used for pose estimation. Figure shows filtered position estimates vs ground truth.

the filter. The mission was performed on the YOLOv4-tiny on a 128x128 resolution, which was seen to be the best pose estimation model for real-time control on the CPU.

The estimated pose versus ground truth for this test is plotted in Figure 5.8. The quadcopter successfully managed to complete the autonomous landing mission using the Darknet pose estimator. The pose estimation algorithm struggles to estimate pose correctly when the quadcopter is very close to the landing platform. This happens when the altitude is below 0.5m. When the quadcopter is so close, the landing platform covers the complete camera view. If the camera image consists of only the green part of the platform, then the algorithm will struggle to estimate the correct center of the landing platform. The Darknet network will classify the complete camera view as Helipad. Predicting the center of the landing platform to be in the center of this bounding box results in the center of the landing platform to be estimated to be in the center of the camera view. This may not be the correct center. Therefore the model will struggle when the quadcopter is below a certain altitude. This will be less of a problem if the H is located in the camera view, as the H will then be used to predict the center. It did mange to track the y and z estimates well until the end, but struggled to estimate x well. It is not unexpected that the algorithm struggles to estimate pose when hovering so close to the platform given the implementation. A possible solution for this is to train the model to recognize smaller parts of the landing platform, such that it may locate those when the quadcopter is close, and thereby be able to estimate the correct center and radius.

A 3D plot of estimated pose is plotted versus ground truth during the automated landing mission and can be seen in Figure 5.9. The largest errors in the pose estimate appear when the quadcopter is very close to the landing platform as well as after it has landed.

The algorithm was not able to identify the Arrow and therefore did not estimate yaw rotation.

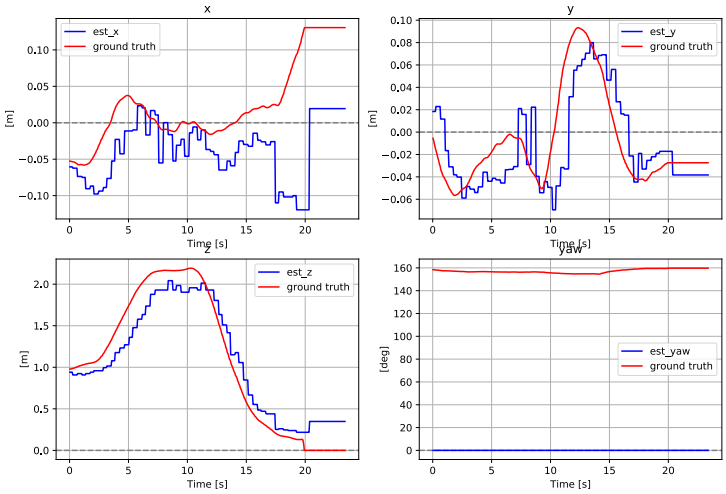


FIGURE 5.8: Landing test on CPU, using YOLOv4-tiny at 128x128 resolution. Darknet used for pose estimation. Figure shows pose estimate vs ground truth.

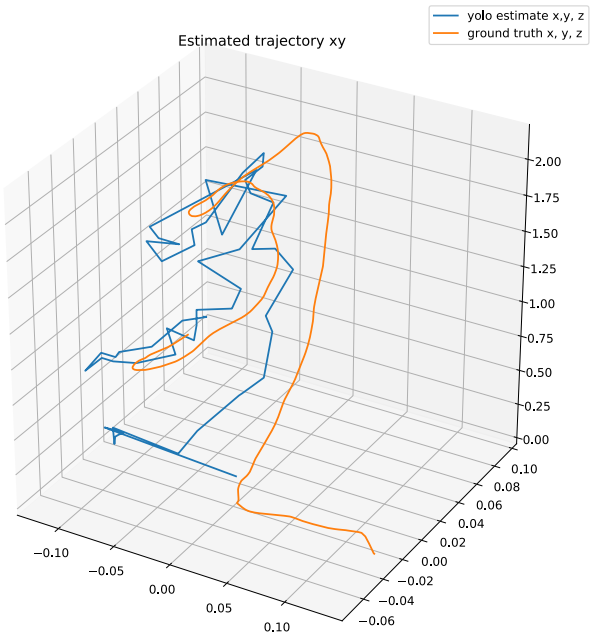


FIGURE 5.9: Landing test on CPU, using YOLOv4-tiny at 128x128 resolution. Darknet used for pose estimation. Figure shows position estimates vs ground truth.

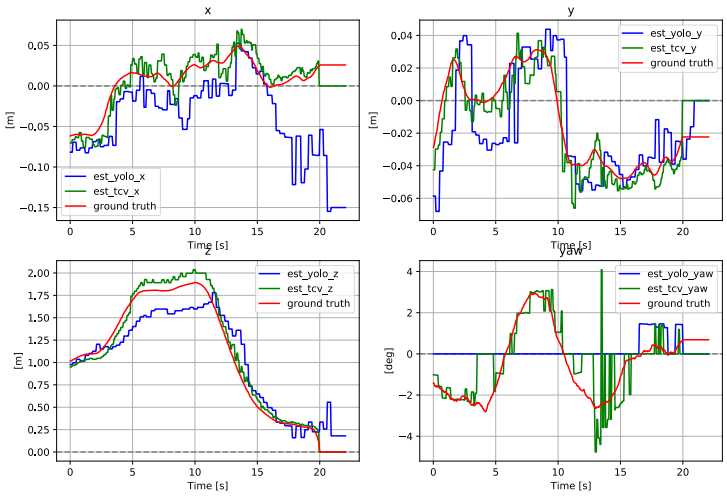


FIGURE 5.10: Landing test on CPU, using YOLOv4-tiny at 128x128 resolution. Filtered combination of methods used for pose estimation. Figure shows plot of position estimate vs ground truth for both pose estimation methods.

5.2.4 Automated landing with combined methods estimate on CPU

The automated landing test is performed using the combined methods on CPU. The Darknet pose estimator model that was used was YOLOv4-tiny with image resolution of 128x128. The filtered combined pose estimate during the test can be seen in Figure 5.11 and a 3D plot of estimated pose versus ground truth can be seen in Figure 5.12. The pose estimates from the two estimators during the landing scenario are presented in Figure 5.10.

The combined estimates are closer to the ground truth than the estimates from the Darknet estimator alone, as can be seen when comparing 3D plots for the two landing missions; Figure 5.12 and Figure 5.9. The Darknet estimator struggled with accurate pose estimates when the quadcopter was close to the landing platform. The TCV estimator does not do this, and is accurate even when very close. The TCV module compensates for the estimation errors from Darknet, and the resulting estimate is closer to the ground truth.

5.2.5 CPU results discussion

The combined estimate is sufficiently accurate in order to perform an automated landing mission in the simulated environment. Despite this, the estimation errors are quite significant for the Darknet pose estimator. This could make the landing mission fail in a more challenging landing scenario. The Darknet model YOLOv4-tiny on a resolution of 128x128 is not robust, will therefore not be good at detecting the landing platform in a challenging visual environment. Because of large estimation errors when close to the platform, and the low detection capabilities of the Darknet module when run on CPU, the combined model is less robust than the TCV pose estimator. The TCV module produces more accurate estimates than the combined module, and adding the YOLOv4-tiny detector does not add robustness to the model.

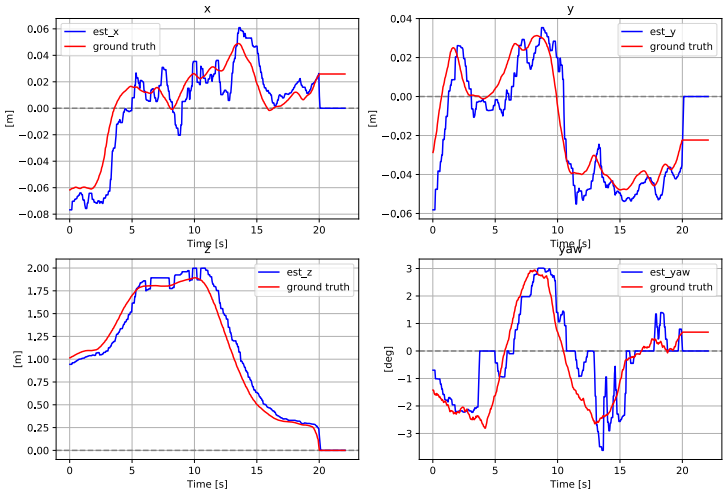


FIGURE 5.11: Landing test on CPU, using YOLOv4-tiny at 128x128 resolution. Filtered combination of methods used for pose estimation. Figure shows filtered pose estimates vs ground truth.

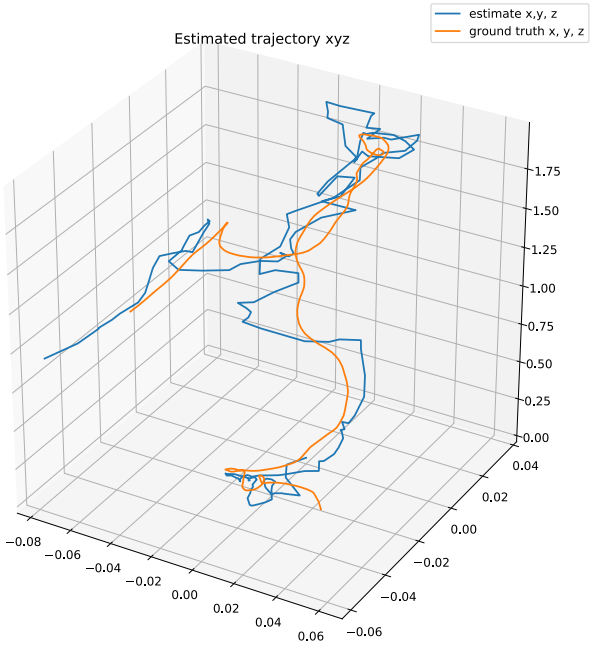


FIGURE 5.12: Landing test on CPU, using YOLOv4-tiny at 128x128 resolution. Filtered combination of methods used for pose estimation. Figure shows filtered position estimate vs ground truth.

5.3 Simulation results with GPU

When running the experiments on a GPU the Darknet pose estimator achieved an inference rate that was much faster than on the CPU. This is important, as delay in the estimates have been shown to be quite detrimental for estimation accuracy and quadcopter control. The only model that was able to produce estimates at a sufficient frequency was YOLOv4-tiny at the lowest resolution, which struggled to classify the Arrow, as well as not classifying the Helipad when hovering at 5m. Therefore it is preferable to use a GPU when running these experiments. When using a GPU the pose estimator achieves sufficient pose estimation frequency with all the models that were tested. The model with the highest mAP, YOLOv4 at the resolution 416x416, is used for all the tests processed on a GPU.

5.3.1 Stationary hover test with Darknet pose estimator on GPU

The stationary hover test is performed using the Darknet pose estimator on GPU. The altitude reference point is $z_r = 1.0m$, and reference points for x and y are $x_r = -0.10m$ and $y_r = 0.0m$. The initial altitude for this test was 1.4m, as can be seen in the plot of pose estimates during the hover test Figure 5.13. The high inference rate, as well as the more accurate YOLO model, results in estimates that align with the ground truth quite well. The estimates for x and y are very close to the ground truth.

There is no notable delay in the estimates. The estimates are quite oscillatory. This might be because of bounding box jitter, where the sizes and locations of the bounding boxes are not constant between frames, instead jumping back and forth slightly. One solution for this might be to increase the filter sizes in order to smooth out the estimates. These estimates are filtered using a median filter size of 3 and an averaging filter size of 3. The deviations from the ground truth in the x- and y-axis are at most a 3 cm when performing the test.

The model struggles to estimate yaw correctly. This is most likely because of an error in the code, where the yaw estimate was predicted to be zero if the Arrow was not detected. Therefore the filter averages these results, making the estimated yaw to be quite oscillatory and lower than the ground truth yaw. A way to fix this is to estimate yaw to be the previously estimated yaw if a current estimate of yaw is not available. A fix for this was not tested due to lacking GPU resources for testing. Further investigation may need to be done of why the yaw estimate is incorrect.

There estimates in the z-axis have a varying deviation from the ground truth. A possible explanation for this is that the model failed to estimate yaw correctly, and therefore did not scale down the bounding box of the H as it was supposed to. If the H is rotated the bounding box surrounding the H will be larger than the H. When the Arrow is detected, the bounding box of the H is used for radius estimation. If the Arrow is detected, but yaw is not estimated correctly, then the H will be scaled improperly. The radius estimate of the landing platform is based on the size of the H if the Arrow is detected. Estimating the radius of the landing platform using an improperly scaled bounding box of the H will give an incorrect estimate of the landing platform radius, and therefore the subsequent z estimate will be incorrect.

When using the same model and performing the stationary hover test with $z_r = 5m$ the quadcopter managed to stay quite stable in the air. The estimated pose versus ground truth can be seen in Figure 5.14. The estimates in x- and y-axis are quite oscillatory, which may be due to bounding box jitter. A way to fix this could be to increase the filter sizes in order to smooth the filtered signal further. Having a large filter size increases estimation lag, and may therefore be undesirable if the estimation

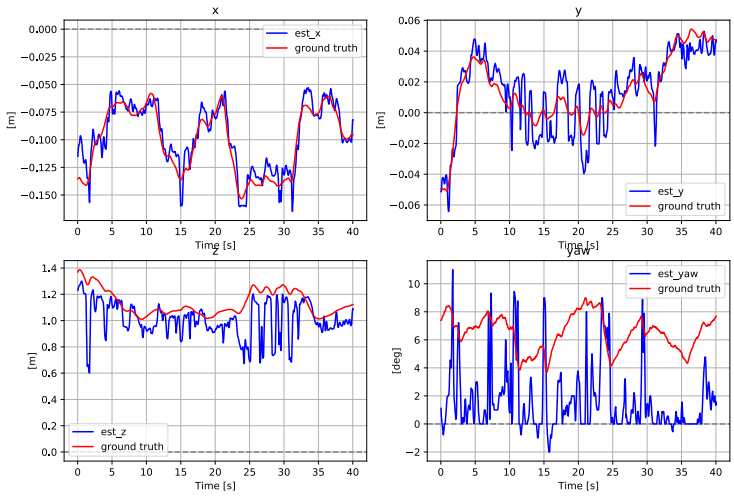


FIGURE 5.13: Hover test, $z_r = 1m$, using Darknet pose estimator on GPU. Filtered output with filter sizes of 3

frequency is low. The estimator error is quite low in x- and y-axis. While hovering at 5m the estimates are within $\pm$ 20 cm of the ground truth within the whole test duration. This is sufficiently low given the large distance from the quadcopter to the landing platform, and the quadcopter is able to perform stable hovering at 5m.

The estimates in the z-axis are quite oscillatory as well, and the estimates oscillate with an amplitude of about 80cm while hovering at 5m. This is quite significant. The oscillations in the estimates are of a high frequency compared to the quadcopter dynamics, and it does not affect control too much. These oscillations may happen because the bounding box drawn around the helipad varies slightly in size. A fix for this could be to increase the filter sizes. The quadcopter managed to hover at an altitude of around 5m, although having some low frequency oscillations in altitude. The estimates of quadcopter altitude are slightly below the ground truth line. This may happen if the bounding box surrounding the platform is larger than the platform. This can happen if the YOLO detection are not perfectly precise. If that happens, the resulting estimate of helipad radius in pixels will be too large, resulting in the algorithm estimating that the quadcopter is closer to the landing platform than it actually is. A solution for this is to find, through experimentation, how much larger the bounding boxes are than the landing platform, and compensate for this when estimating the radius.

The algorithm was not able to detect the Arrow from an altitude of 5m, and therefore did not estimate yaw.

5.3.2 Stationary hover test with combined methods estimate on GPU

When using combined estimates the update frequency of the Darknet pose estimator is capped at 10 Hz. This is done in order to make the filtered output be the best combination of both methods. If the Darknet estimator produce estimates at a frequency that is much higher than the TCV based estimator then the moving median averaging filter will remove most of the TCV estimates in favor of the majority of estimates, which come from the Darknet estimator.

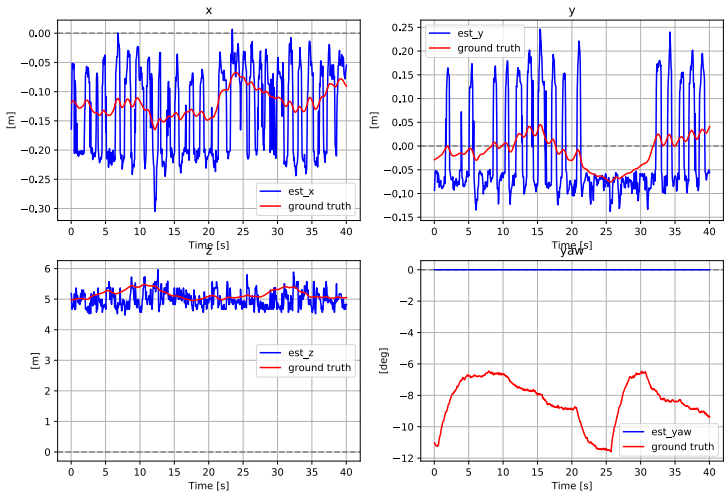


FIGURE 5.14: Hover test, $z_r = 5m$, with Darknet pose estimator on GPU. Filtered output with filter sizes of 3.

The hover test is performed at $z_r = 1m$ using the combined pose estimator, with Darknet on GPU. The estimated pose from both methods can be seen in Figure 5.15 and the combined estimate can be seen in Figure 5.16. The Darknet pose estimator struggles to estimate rotation correctly, however the TCV estimator is able to track rotation well. Further investigation is required to find the cause of this problem. Positional estimates are close to the ground truth. The resulting filtered estimate tracks the ground truth well.

There is a constant deviation in the estimate of quadcopter altitude, making the quadcopter hover above the reference point of 1m. Both the TCV estimator and the Darknet estimator estimates the altitude to be slightly lower than the ground truth altitude, and thus the resulting combined estimate is too low as well. The Darknet pose estimator may estimate the altitude to be lower than the ground truth altitude if the bounding boxes used for estimating altitude is too large. Similar problems were experienced by Xu et. al. [42] when using YOLOv4 for detection of objects from a high altitude using UAV based aerial images, where the bounding boxes were detected at wrong size and shape.

The quadcopter is able to perform the 1m hover test well using combined estimates as feedback on GPU.

The stationary hover test at $z_r = 5m$ is performed using the combined estimate processed on GPU. The filtered pose estimate for this test can be seen in Figure 5.18 and the estimates from both methods can be seen in Figure 5.17. Neither of the estimates are able to estimate the rotation reliably from that altitude. The TCV estimates track the ground truth close in x- and y- axis, while the Darknet estimates are quite oscillatory around the ground truth, and offset by a couple centimeters in the x-axis. Both methods are able to estimate the altitude quite well, although the Darknet estimates are more oscillatory.

From an altitude of 5m the estimates from both methods are quite good, and the combined filtered estimates track the ground truth quite well. The quadcopter manages to perform the 5m hover test successfully using the filtered combines estimates as pose estimates. Thus the combined filter is sufficiently robust to perform reliable pose estimates from an altitude of 5m in the simulated environment.

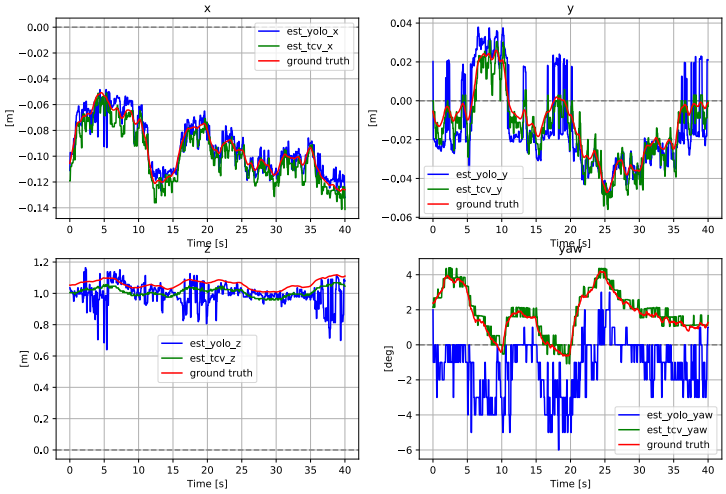


FIGURE 5.15: 1m Hover test on GPU. Filtered combination of methods used for pose estimation. Figure shows position estimates from both methods vs ground truth.

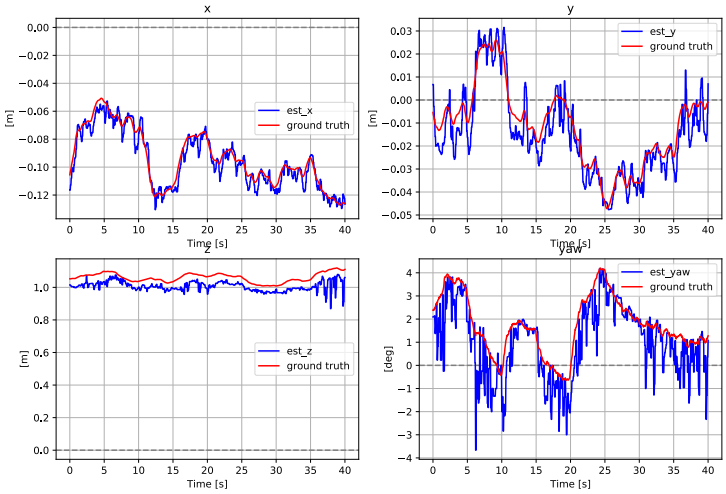


FIGURE 5.16: 1m Hover test on GPU. Filtered combination of methods used for pose estimation. Figure shows filtered estimates vs ground truth.

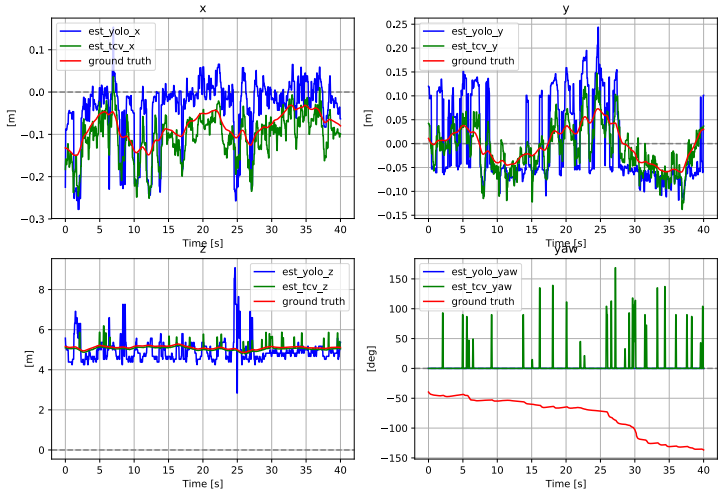


FIGURE 5.17: 5m Hover test on GPU. Filtered combination of methods used for pose estimation. Figure shows position estimates from both methods vs ground truth.

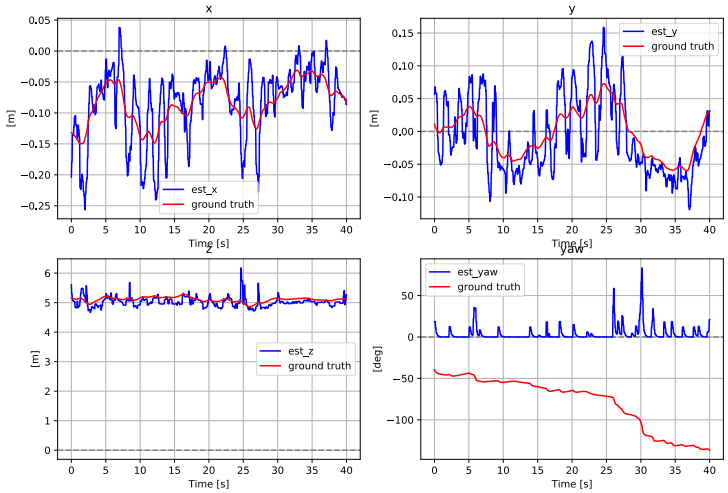


FIGURE 5.18: 5m Hover test on GPU. Filtered combination of methods used for pose estimation. Figure shows filtered estimates vs ground truth.

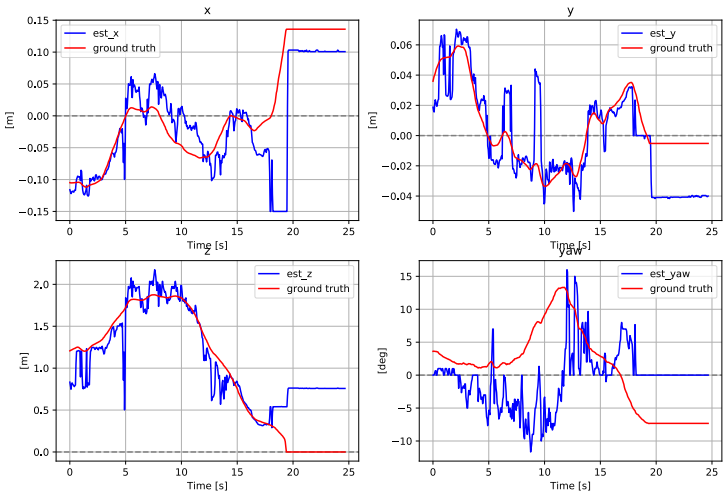


FIGURE 5.19: Automatic landing test on GPU. Darknet used for pose estimation. Figure shows filtered estimates vs ground truth.

5.3.3 Automated Landing using Darknet pose estimator on GPU

The quadcopter is able to perform a successful automated landing mission using the Darknet pose estimator on GPU as can be seen in Figure 5.19. It is still evident that the Darknet estimator struggles to estimate rotation. This does not impact the automated landing mission to a high degree. The increased update frequency, as compared to running the test on CPU, enables the position estimates to track the ground truth closer. As the quadcopter comes closer to the landing platform the estimates deviate from the ground truth.

5.3.4 Automated landing with combined methods estimate on GPU

An automated landing test is performed with combined methods estimate on a GPU processor. The filtered estimate while running this test can be seen in Figure 5.21, while the estimates from the individual methods can be seen in Figure 5.20.

The quadcopter is able to perform a successful landing mission using the combined pose estimate as pose feedback. Both methods are tracking the ground truth quite well, although the Darknet estimates deviate when the quadcopter gets too close to the landing platform. This makes the combined estimate deviates as well. The scale of the deviations in the filtered estimate is less than the deviations in the Darknet estimates. This is because they are combined with the estimates from the TCV module which do not deviate. The TCV estimates track the ground truth quite well during the whole mission, and is able to track the yaw better than the Darknet estimates.

The combined pose estimates are sufficiently close to the ground truth in order successfully perform an automated landing mission, although the deviations from the ground truth when the quadcopter is close to the landing platform could pose a problem in a more challenging landing environment.

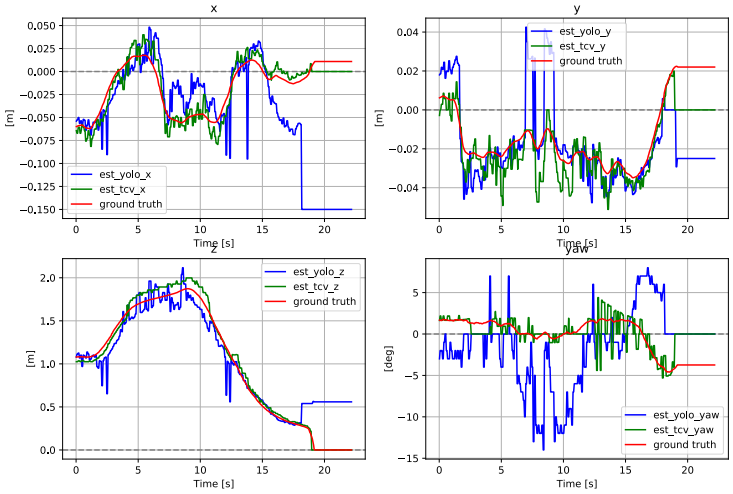


FIGURE 5.20: Automatic landing test on GPU. Combined methods used for pose estimation. Figure shows estimates for both methods vs ground truth.

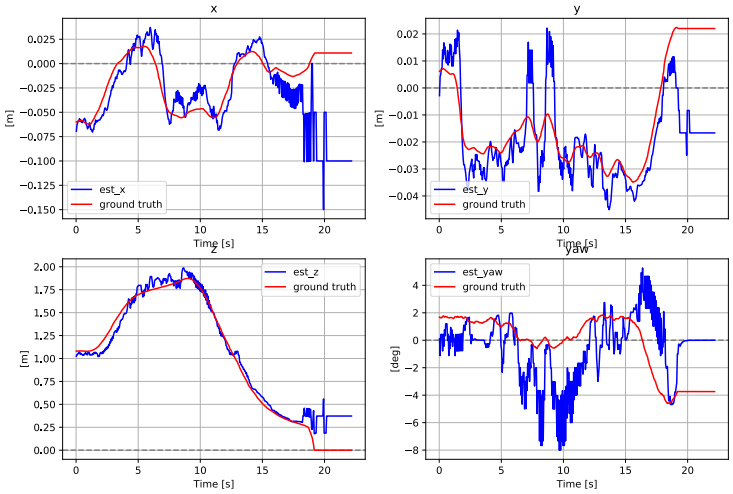


FIGURE 5.21: Automatic landing test on GPU. Combined methods used for pose estimation. Figure shows filtered estimates vs ground truth.

5.3.5 GPU discussion

The combined computer vision pose estimator, when run on GPU, is accurate- and robust enough in order for the quadcopter to successfully perform the autonomous missions.

The TCV pose estimator is a pose estimator which is very accurate and well tuned for the simulation environment. Thus it is able to estimate pose accurately, with low estimation errors. The Darknet pose estimator was seen to be able to estimate pose reliably, although the estimation errors were generally larger than the estimated errors using the TCV module. This was especially evident when the quadcopter was close to the landing platform, and for estimates of yaw.

The TCV module requires tuning for individual lighting conditions. It struggled to estimate pose correctly when tested with the physical quadcopter and landing platform [1]. The robustness of the combined model may therefore be better than the robustness of each model individually. This is because the two computer vision modules use different computer vision techniques for detection of the landing platform, and where one may fail the other may detect the landing platform. Thus the combined pose estimator may be more robust. The Darknet module do not require specific tuning for individual visual conditions, and may therefore supplement the TCV module by being a more general estimator with a higher detection rate. For the Darknet module to be robust in changing visual conditions it must be provided with a Dataset which contains images of the landing platform in changing visual conditions.

The combined pose estimator was able to accurate and reliably estimate the pose of the quadcopter. It was seen to be robust in the simulator environment. Further testing on visual conditions that are more challenging is required to determine the limits of the robustness of the model.

Chapter 6

Conclusion

The objective for this thesis was to create a robust pose estimation computer vision system for an autonomous quadcopter using a combination of traditional- and deep learning computer vision techniques. A pose estimator using deep learning was created and tested. That pose estimator was also tested in combination with an established pose estimator based on traditional computer vision techniques.

The results show that it is possible to create a reliable pose estimation algorithm using a deep learning approach. The pose estimates based on a deep learning approach were sufficient for both stable hovering and automated landing missions. The deep learning pose estimator had both weaknesses and strengths. It was able to produce pose estimates at a higher frequency when run on a GPU than the pose estimator based on traditional computer vision. This is not the case when run on a CPU, as the DL-based pose estimate frequency on a CPU was very low. Although successful autonomous missions achieved, while running the pose estimator on a CPU, these was only successful with the DL-model with the lowest detection accuracy of those that were tested. Results showed therefore that a is GPU is necessary in order to have a robust real-time pose estimator using a deep learning approach. The pose estimates were quite oscillatory using the deep learning approach, and it struggled to estimate the yaw rotation correctly. The oscillations were generally quite small, and the quadcopter was able to complete the autonomous missions using this pose estimator. The deep learning based estimator struggled to estimate the quadcopter pose when the quadcopter was very close to the landing platform and the large features of the landing platform were not distinguishable in the camera frame. This was not a sufficient enough problem to hinder the automated landing from succeeding, but may be a problem in a more challenging landing scenario.

The combined computer vision system is not yet tested in more challenging visual conditions than the computer simulator. The simulator system is an environment in which both the computer vision methods are quite proficient at estimating the quadcopter pose. Therefore the level of robustness for the combined pose estimator is not tested. Further testing is required in order to determine the robustness of the combined computer vision system.

6.0.1 Future work

The results in this thesis are encouraging as a step on the path to creating a quadcopter capable of autonomous takeoff-and-landing missions.

Although it has much potential, the deep learning pose estimator requires some refinement. The pose estimator struggled to estimate yaw rotation correctly. An investigation of this problem is required. The DL algorithm can be trained on a larger data, with images from more challenging visual conditions in order to make the pose estimates more robust. The deep learning pose estimator also struggled to estimate

pose when the quadcopter was very close to the landing platform, so there is an option to train the network to recognize smaller parts of the landing platform in order to be able to produce a decent pose estimate during landing missions. The model can also be trained to recognize the ReVolt vessel and use that for pose estimation. That vessel is quite a large object, and will therefore be easier to locate. The orientation of the ReVolt be utilized for estimation of rotation as well. The ship is larger than the small arrow on the landing platform, so yaw estimates should be easier to estimate using the orientation of the ReVolt vessel.

Further refinement of the filter for combining the methods may be investigated. The DL pose estimator struggled when the quadcopter was close to the landing platform, something the TCV pose estimator did not. Therefore a filter which combines the methods based on their strengths and weaknesses may be implemented. Currently the quadcopter relies solely on visual data for pose estimation. An Extended Kalman filter can be used for a robust pose estimation by fusing together data from the quadcopter IMU sensors together with visual pose estimation in order to predict pose [3]. This can reduce spikes in pose estimates, and make the quadcopter more robust by being less reliant on constant detection of the landing platform. .

The pose estimator should be tested on the physical AR. Drone 2.0 with the real landing platform and the physical ReVolt vessel. It should also be tested in different visual conditions in order to determine the robustness of the method. The DL method should be trained on a dataset consisting of images of the physical landing platform in an outside environment in order to be functional in an outside environment as well.

Other sensors can be added to the quadcopter as well. A GPS sensor can be used and fused together with the other pose estimates in order to create a more robust pose estimate. GPS data can also be used for the quadcopter to localize the ship if the ship is outside the quadcopter's visual range. GPS opens for the possibility that the quadcopter may perform autonomous missions outside the visual range of the landing platform.

Appendix A

Appendix

θ	$(\text{scale}_w, scale_h)$	θ	$(\text{scale}_w, scale_h)$
0	(1.0, 1.0)	45	(0.566, 0.849)
5	(0.887, 0.948)	50	(0.558, 0.867)
10	(0.803, 0.909)	55	(0.555, 0.893)
15	(0.738, 0.878)	60	(0.556, 0.928)
20	(0.688, 0.856)	65	(0.561, 0.974)
25	(0.649, 0.842)	70	(0.571, 1.033)
30	(0.619, 0.834)	75	(0.586, 1.108)
35	(0.595, 0.832)	80	(0.606, 1.205)
40	(0.578, 0.837)	85	(0.632, 1.331)
45	(0.566, 0.849)	90	(0.667, 1.5)

TABLE A.1: Factors for scaling bounding box surrounding H to size of H by rotation θ

Bibliography

[1] Thomas Sundvoll. *A Camera-based Perception System for Autonomous Quadcopter Landing on a Marine Vessel*. 2020.

[2] *Future Of Drones Drone technology uses and applications*. https://www.businessinsider.com/drone-technology-uses-applications?r=US&IR=T. Accessed: 2020-10-20.

[3] Mohammad Fattahi Sani, Maryam Shoaran, and G. Karimian. "Automatic landing of a low-cost quadrotor using monocular vision and Kalman filter in GPS-denied environments". In: *Turkish Journal of Electrical Engineering and Computer Sciences* 27 (2019), pp. 1821–1838.

[4] Tiago Gomes Carreira. "Quadcopter Automatic Landing on a Docking Station". In: 2013.

[5] Mary B. Alatise and Gerhard P. Hancke. "Pose Estimation of a Mobile Robot Based on Fusion of IMU Data and Vision Data Using an Extended Kalman Filter". In: *Sensors* 17.10 (2017). ISSN: 1424-8220. DOI: 10.3390/s17102164. URL: https://www.mdpi.com/1424-8220/17/10/2164.

[6] Malik Demirhan and C. Premachandra. "Development of an Automated Camera-Based Drone Landing System". In: *IEEE Access* 8 (2020), pp. 202111–202121.

[7] J. Blom. "Onboard Visual Control of a Quadcopter MAV Performing a Landing Task on an Unknown Platform". In: 2019.

[8] N. Karlsson et al. "The vSLAM Algorithm for Robust Localization and Mapping". In: *Proceedings of the 2005 IEEE International Conference on Robotics and Automation*. 2005, pp. 24–29. DOI: 10.1109/ROBOT.2005.1570091.

[9] A. Garcia, E. Mattison, and K. Ghose. "High-speed vision-based autonomous indoor navigation of a quadcopter". In: *2015 International Conference on Unmanned Aircraft Systems (ICUAS)*. 2015, pp. 338–347. DOI: 10.1109/ICUAS.2015.7152308.

[10] Joseph Redmon and Ali Farhadi. *YOLOv3: An Incremental Improvement*. 2018. arXiv: 1804.02767 [cs.CV].

[11] C. Kumar B., R. Punitha, and Mohana. "YOLOv3 and YOLOv4: Multiple Object Detection for Surveillance Applications". In: *2020 Third International Conference on Smart Systems and Inventive Technology (ICSSIT)*. 2020, pp. 1316–1321. DOI: 10.1109/ICSSIT48917.2020.9214094.

[12] W. Budiharto et al. "Fast Object Detection for Quadcopter Drone Using Deep Learning". In: *2018 3rd International Conference on Computer and Communication Systems (ICCCS)*. 2018, pp. 192–195. DOI: 10.1109/CCOMS.2018.8463284.

[13] N. Imanberdiyev et al. "Autonomous navigation of UAV by using real-time model-based reinforcement learning". In: *2016 14th International Conference on Control, Automation, Robotics and Vision (ICARCV)*. 2016, pp. 1–6. DOI: 10.1109/ICARCV.2016.7838739.

[14] N. Smolyanskiy et al. "Toward low-flying autonomous MAV trail navigation using deep neural networks for environmental awareness". In: *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2017, pp. 4241–4247. DOI: 10.1109/IROS.2017.8206285.

[15] Tianxiao Zhang et al. *Efficient Golf Ball Detection and Tracking Based on Convolutional Neural Networks and Kalman Filter*. 2020. arXiv: 2012.09393 [cs.CV].

[16] Jin Liu and Sheng He. "6D Object Pose Estimation Based on 2D Bounding Box". In: *CoRR* abs/1901.09366 (2019). arXiv: 1901.09366. URL: http://arxiv.org/abs/1901.09366.

[17] Joseph Walsh et al. "Deep Learning vs. Traditional Computer Vision". In: Apr. 2019. ISBN: 978-981-13-6209-5. DOI: 10.1007/978-3-030-17795-9_10.

[18] Juan Du. "Understanding of Object Detection Based on CNN Family and YOLO". In: (2018).

[19] Michael A. Nielsen. *Neural Networks And Deep Learning*. 2015.

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

[21] Keiron O'Shea and Ryan Nash. *An Introduction to Convolutional Neural Networks*. 2015. arXiv: 1511.08458 [cs.NE].

[22] Keiron O'Shea and Ryan Nash. *An Introduction to Convolutional Neural Networks*. 2015. arXiv: 1511.08458 [cs.NE].

[23] Joseph Redmon el. al. *Darknet, A neural network framework*. https://github.com/pjreddie/darknet. 2018.

[24] Kapoor Mittal Vaidya. "Object Detection and Classification Using Yolo". In: *International Journal of Scientific Research & Engineering Trends* 5 (2019).

[25] Tsung-Yi Lin et al. *Microsoft COCO: Common Objects in Context*. 2015. arXiv: 1405.0312 [cs.CV].

[26] Alexey Bochkovskiy, Chien-Yao Wang, and Hong-Yuan Mark Liao. *YOLOv4: Optimal Speed and Accuracy of Object Detection*. 2020. arXiv: 2004.10934 [cs.CV].

[27] Zicong Jiang et al. *Real-time object detection method based on improved YOLOv4-tiny*. 2020. arXiv: 2011.04244 [cs.CV].

[28] Sanjeev Suresh Donovang Fung Prerna Dhareshwar. *Recurrent CNNs for Bounding Box stability in Object Detection*. http://cs230.stanford.edu/projects_winter_2019/reports/15812427.pdf. 2018.

[29] Khoshgoftaar Shorten C. "A survey on Image Data Augmentation for Deep Learning". In: *Journal Of Big Data* 6 (2019).

[30] Smart Quigley Gerkey. "Programming Robots with ROS. A practical introduction to the robot operating system". In: (2015).

[31] Kevin Krewell. *Difference between CPU and GPU*. https://blogs.nvidia.com/blog/2009/12/16/whats-the-difference-between-a-cpu-and-a-gpu/. 2009.

[32] S. Piskorski et al. "A.R.Drone Deveoper Guide". In: (2012).

[33] *ardrone_autonomy*. https://ardrone-autonomy.readthedocs.io/en/latest/. Accessed: 2020-12-05.

[34] Thomas Sundvoll. *A Camera-based Perception System for Autonomous Quadcopter Landing on a Marine Vessel*. https://github.com/mrSundvoll/master_project. 2019.

[35] *cvat.org: a computer vision annotation tool*. https://cvat.org. Accessed: 2020-10-20.

[36] *A dataset analysis and augmentation tool*. https://roboflow.com/. Accessed: 2020-10-20.

[37] AlexeyAB. *YOLOv4 - Neural Network for Object Detection*. https://github.com/AlexeyAB/darknet. 2020.

[38] google. *Google Colab: Providing free GPUs for online training of*. https://colab.research.google.com/.

[39] Josep Nelson. *Roboflow Model Library: Free open-source Google Colab notebooks for training of DNN models*. https://models.roboflow.com/.

[40] Marko Bjelonic. *YOLO ROS: Real-Time Object Detection for ROS*. https://github.com/leggedrobotics/darknet_ros. 2016–2018.

[41] Y. Li et al. "A Deep Learning-Based Hybrid Framework for Object Detection and Recognition in Autonomous Driving". In: *IEEE Access* 8 (2020), pp. 194228–194239. DOI: 10.1109/ACCESS.2020.3033289.

[42] H. Xu et al. "Performance Comparison of Small Object Detection Algorithms of UAV based Aerial Images". In: *2020 19th International Symposium on Distributed Computing and Applications for Business Engineering and Science (DCABES)*. 2020, pp. 16–19. DOI: 10.1109/DCABES50732.2020.00014.

[43] *The ReVolt Vessel by DNV-GL*. https://www.dnvgl.com/technology-innovation/revolt/index.html. Accessed: 2020-10-20.

[44] Kaiming He et al. *Deep Residual Learning for Image Recognition*. 2015. arXiv: 1512.03385 [cs.CV].

[45] *YOLOv5 by Ultralytics*. https://towardsdatascience.com/how-to-create-an-end-to-end-object-detector-using-yolov5-35fbb1a02810. Accessed: 2020-10-22.

[46] Marko Bjelonic. *Enabling YOLOv4 to run with darknet-ros: Modified version of leggedrobotics: darknet-for-ros*. https://github.com/tom13133/darknet_ros. 2017–2020.