Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Engineering
Department of Mechanical and Industrial Engineering

Martin Mjølhus Helle

Pose Estimation of Shipping Container with PnP and Deep Learning

Master's thesis in Mechanical Engineering
Supervisor: Olav Egeland
June 2021

NTNU
Norwegian University of
Science and Technology

Martin Mjølhus Helle

Pose Estimation of Shipping Container with PnP and Deep Learning

Master's thesis in Mechanical Engineering
Supervisor: Olav Egeland
June 2021

Norwegian University of Science and Technology
Faculty of Engineering
Department of Mechanical and Industrial Engineering

NTNU
Norwegian University of
Science and Technology

Preface

This thesis completes my master's degree in Mechanical Engineering at NTNU.

In my final year of study, I selected subjects involving computer vision and machine learning due to curiosity. In my specialization project, I focused on solving computer vision problems related to autonomous, offshore crane lifts. During this time, It was discovered several methods on solving it as there was not a clear solution to the problem. With artificial intelligence on the rise, it seemed feasible to implement such a technology to this subject. Working with AI and computer vision throughout the specialization project, lots of knowledge was gained on how to combine AI- and classical computer vision technologies to track objects.

This master thesis is about computer vision and artificial intelligence, and will try to help the reader understand the basics before solutions are presented. There are some expectations that the reader has knowledge about linear algebra, basic transformation matrices and 3D geometry, and programming. It is advantageous to have knowledge about deep learning and projective transformations prior to reading this.

Acknowledgements

I would like to express gratitude to my supervisor in my master thesis and specialization project, Olav Egeland. He has been a great help by giving me guidance. I would like to give say thank you to my classmates, and a special thanks to the people I shared office and lunch with for giving me both technical advice and motivation to keep working with this project. Finally, to my family, my flatmates and my friends from home, I feel truly fortunate to have such a network around me during the pandemic to keep my spirits up.

Summary

The report focuses on tracking cargo in offshore environments to develop an offshore autonomous crane lift system. The report discusses the suitability of different sensor solutions, such as implementing a 3D camera with structured light, 2D camera and laser technology. Finally, it was proposed to use a 2D digital camera with Perspective-n-points using deep learning as a feature extractor in conjunction with a corner detector for cargo tracking.

The experiment of tracking and calculating pose automatically was implemented on a small-scale model of a shipping container. The experiment performed with a translational error between 8 mm and up to 15 mm during this experiment and error of 0.14 degrees, describe in Euler angles. Along with the potential of the system, some problems with noisy features were addressed.

The instance segmentation and corner detector combination were prone to noise if the instance segmentation model did not return a precise mask prediction. A new overfitted Mask R-CNN model was trained to test the system in a circumstance where the mask prediction was precise. During the video test, it was able to find image point correspondences in most cases, with exceptions in some frames.

Further, different methods of improving the system was proposed. The propose methods for further work entails solutions to make the system more accurate, faster and more robust against noise.

Upgrading the instance segmentation network. Combining a faster instance segmentation model YOLACT++ (33.5 fps) with higher image resolution was proved through testing to make the system more accurate and lower time delay. Methods of filtering out the noisy features were proposed to make the current system more robust. Solutions such as optical flow or quadrilateral fitting were mentioned. It seems like this solution is promising for tracking planar, rectangle surfaces are promising, and with upgrades it have the potential to become a real-time tracking system with error $\leq$ 10 mm, with noise filters.

Sammendrag

Denne rapporten fokuserer på hyppig positur- og avstandsmåling av last i offshore miljø for å utvikle et offshore autonomt kranløftesystem. Rapporten diskuterer egnetheten for bruk av ulike type sensorer for denne problemstillingen, deriblant 3D kamera som baseres på strukturert lys, 2D camera og laser teknologi. Tilslutt, så ble det foreslått bruk av 2D digital kamera og Perspective-n-points ved hjelp av dyp læring og hjørne detektor for å gjenkjenne karakteristiske trekk for å kunne måle avstand til last.

Det ble utført et eksperiment ved å hyppig regne ut orientering og avstand på en liten modell av en shipping container. Systemet ble utført med en translasjonsfeil mellom 8 mm og opptil 15 mm under dette eksperimentet og en orienteringsfeil på 0,14 grader, beskrevet ut ifra Euler-vinkler. I tillegg til systemets potensial ble det også løst noen problemer med støy.

Eksempler på at dyp læring- og hjørnedetektorkombinasjon var utsatt for støy var når dyp læringssmodellen ikke returnerte en presis segmentering av objektet. En overtilpasset Mask R-CNN-modell ble trent til å teste systemet i omstendighet der segmenteringen var godt trent på. I løpet av videotesten var systemet i de fleste tilfeller i stand til å finne punkt korrespondanser mellom 3D punkter og pixel koordinatene i bildet.

Videre ble det foreslått forskjellige metoder for å forbedre systemet. Metodene for videre arbeid innebærer løsninger for å gjøre systemet mer nøyaktig, raskere og mer robust mot støy.

Å bruke høyere bildeoppløsning ble bevist gjennom testing for å gjøre systemet mer nøyaktig. Det er foreslått å kombinere dette med en ny sanntids segmenteringsmodell YOLACT++ (33.5fps) for å gjøre modellen mer nøyaktig, men også raskere. Metoder for å filtrere ut støy ble foreslått for å gjøre det nåværende systemet mer robust. Løsninger som optisk strømning eller firkantet montering ble nevnt. Det virker som om systemet i denne rapporten er lovende for hyppig måling av orientering og posisjon av rektangeloverflater, og med oppgraderinger har den potensialet til å bli et sanntids sporingssystem med error $\leq$ 10mm, inkludert støyfiltre.

Table of Contents

List of Figures

List of Tables

1 Introduction

1.1 Background

In the world we live in today, it can be observed that technology across various industries focuses on becoming more automated. Robots are replacing processes previously conducted by humans. The offshore industry in Norway is no different.

A powerful tool used in robotics to interact with its environment is computer vision. Computer vision aims to make computers be able to see the world, similar to humans. In order to make robots interact with the world, the movements of robots consist of calculated trajectories. It makes them able to move around. In order for robots to efficiently interact with their environment in space, three-dimensional coordinates of the world are required. 3D coordinates can be used to describe the orientation and position of the robot and environment. Then the robot can compute trajectories to complete the tasks it was designed to do. In order to achieve data for the environment, computer vision can be used. The world geometry can be described with different sensors, such as laser distance measurements, ultrasonic sound, monocular camera and stereo vision, to mention some. To decide the sensors to utilize, one has to consider their advantages and disadvantages.

A big part of computer vision is exploiting real-world features, which entails identifying corners, contrast, shapes, and more.

The maritime sector identifies a need to have a system that loads on- and off ships autonomously, with cranes from land, other ships, or offshore platforms. When picking up cargo from a ship, problems that can occur are the ship's movement due to unpredictable wave motions. In order to automate the crane lift operations, it is necessary to compensate for ship displacement. This can be done by tracking the cargo. The tracking data may consist of the 6 degrees of freedom (DOF) pitch, yaw, roll for rotation, surge sway, and heave for translation. The 6 degrees of freedom need to be accurate in order to be able to pick up cargo. In other robotic applications where the object is still standing, the metric for accuracy is dependant on accurate measurements from sensors. In this application, the cargo will continuously move around due to waves. If the readings from sensors are accurate but with considerable time delay, the cargo may be subjected to significant displacements and effectively means inaccurate readings with respect to time. This means that time is of the essence while at the same time the system is dependant on accurate sensor readings.

There is relatively few autonomous crane control system implemented in today's market. However, other industries are conducting real-time pose estimation to function as an inspiration for this case study. Examples include the video games/film industry that is working on similar projects for other use cases.

1.2 Related Work

This subsection presents related work. It consists of related projects for solving the same or similar problems. This subsection has been an inspiration for the solutions in this report.

1.2.1 Optilift

Optilift has developed several solutions, and one of them calculates relative heave movement [31]. This company also offers other solutions related to offshore crane control [32] such as soft lifting, people detectors, to mention some. This company is solves some of the same problems as this report. It states that it uses AI, and by the appearances in the human detector system, it seems to be utilizing object detection AI to classify humans in the operation area.

1.2.2 Autonomous Crane lifts

The company Intsite develops autonomous construction sites using AI and computer vision [33]. Their focus is indicated to be on land-based construction sites, but their technology's transferability to the offshore sector seems to be significant.

1.2.3 Tracking of a Ship Deck Using Vanishing Points and Factor Graph

In the Paper [34], a new way to track a ship deck by using IMU data integrated into a factor graph fused with vision measurements. Vision measurements found vanishing points from a set of parallel lines to calculate the ship's rotation and translation.

1.2.4 Drone Landing

Unmanned aerial vehicles (UAV) are a popular field that has many potential applications. Due to this being a popular research area, it is interesting to see solutions for landing drones. This is because it has been observed lots of similarities between landing cargo on moving ships and landing drones autonomously. The main difference observed is that the drones have a control system that is more reactive than hydraulic cranes, which may require a high-frequency sensor input to react to new inputs smoothly. A crane can have high-frequency sensor data describing the ship's Pose, but the system itself is slow, so a less frequent sensor input may be tolerated. Drones can be trained to land on standstill platforms or moving land- and water vehicles. The same is for crane operations as the ship can be relatively standstill or moving due to wave motion. The paper [35] reviews different methods on how to land UAVs that has worked as a great inspiration regarding analyzing the problems that may occur and how it has been solved. There are parallels between landing drones and landing cargo onto moving ships, such as one needs to identify a landing zone and handle conditions such as moving landing pads in an outside environment with different weather- and lighting conditions.

1.3 Report Outline

This report will be going through the basics of the maths used in this report. It will include camera models, homographies and deep learning with instance segmentation in focus. Further, it will discuss the problem and its complications with offshore computer vision. It will be discussed different methods of solving the problem and finally introduce a seemingly feasible method. After this method is presented, experiments will follow to provide a proof of concept of this technique and discuss its pros and cons. Evaluation and further work follow before the conclusion of this report in the end.

1.4 Objective

The project itself can be large. For one person to complete a computer vision system in a semester, the scope needs to be narrowed down to something that matches the time and resources spent on this project. This project will focus on finding a solution to track object related to the offshore crane lift operations. By tracking, it is meant to find 6D pose estimation so it can be able to pick up an object. The project will focus on calculating pose of one object, but at the same time keeping in mind that the solution can be further developed to pick up cargo and land it from ship to offshore platform and vice versa.

The solution will break down into following sub-goals

- Identify the requirements of the system.
- Analyze different approaches to solve for tracking of objects and find a suitable solution.
- Use experiments to evaluation the suitability of the solution.
- Discuss optimization techniques for further improvements of the current system.

2 Preliminaries

This section presents preliminaries that are necessary to understand this report. The main topics are homographies, Perspective-n-points (PnP), camera calibration and deep learning.

It will be important for the reader to understand different types of homograhies and PnP to understand how pose (the orientation and translation) will be calculated in this report.

It is also important for the reader to understand some deep learning and how it works. Some of the parameters used in the calculation of the pose includes the use of pixel coordinates and deep learning is used to help the extraction of these image points automatically.

2.1 Pinhole Camera model

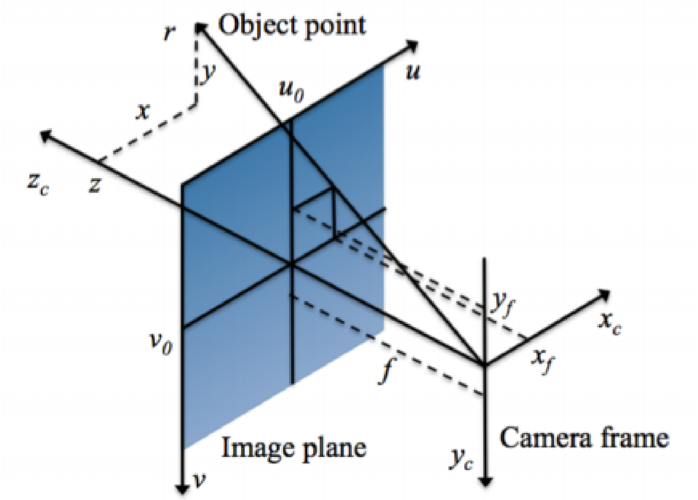


Figure 1: Illustration is taken from [1]

The pinhole camera model mathematically describes the relationship between the 3D world point and the projection onto an ideal pinhole camera's image plane. Properties of the ideal camera model:

$$\tilde{\boldsymbol{p}} = \boldsymbol{K}\tilde{\boldsymbol{s}} \tag{1}$$

where $\tilde{\boldsymbol{p}}$ are the pixel coordinates in the pixel frame, $\boldsymbol{K}$ is the intrinsic camera parameters, and $\tilde{\boldsymbol{s}}$ is the normalized image coordinates. It should be noted that a perspective projection line intersects the camera frame, image point and object point.

Intrinsic camera parameter matrix:

$$\boldsymbol{K} = \begin{bmatrix} \frac{f}{p_w} & k & u_0 \\ 0 & \frac{f}{p_h} & v_0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2}$$

where p_w and p_h is the width and height of one pixel, f is the focal length, k is the skew parameter which can be assumed to be 0 in certain circumstances. u_0 and v_o are the pixel coordinates for the optical center.

The extrinsic camera parameters are the transformation from the camera frame to an object frame. It can be described as the 4x4 matrix:

$$\boldsymbol{T}_o^c = \begin{bmatrix} \boldsymbol{R}_o^c & \boldsymbol{t}_{co}^c \\ \boldsymbol{0}^{\boldsymbol{T}} & 1 \end{bmatrix} \tag{3}$$

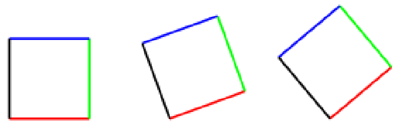
where $\boldsymbol{R}_o^c =$ is the 3x3 rotation matrix from the camera frame to object frame, and $\boldsymbol{t}_{co}^c =$ is the 3x1 translation vector from the origin of the camera frame to the origin of the object frame normally noted as $\begin{bmatrix} x & y & z \end{bmatrix}^T$.

2.2 Homographies in 2D

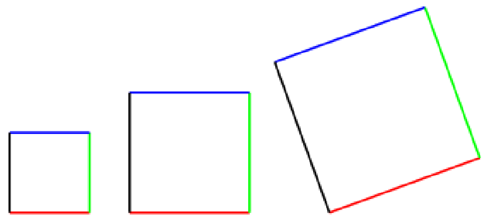
Homographies can be described as a mathematical description of geometry. A homography can typically be used to describe 3D Euclidean space through projected space. Knowing this will help the reader understand how 3D data can be obtained through 2D data in an image.

First, basics of the projective transformations will be explained, before the explanation of the important mathematical formula perspective-n-points or PnP for short. PnP is used for amongst other things, pose estimation, that will be explained further in this report.

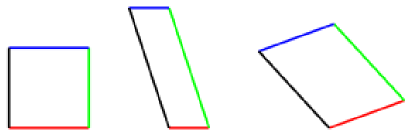
Homographies in 2D shall be explained with the notation given Olav Egeland's *Robot Vision* [1].



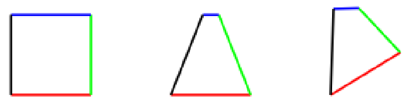
(a) Euclidean transformations: Rotation and translation of a rigid body



(b) Similarity transformations: Rotation, translation and scaling of a rigid body



(c) Affine transformations: Translation, rotation, stretching. Parallel lines remain parallel



(d) Projective transformations.

Figure 2: Brief introduction of different transformations. Illustration is taken from [1].

2.2.1 Euclidean

Further in [1], an euclidean transformation is described as:

$$\boldsymbol{x}' = \boldsymbol{H}_e \boldsymbol{x} = \begin{bmatrix} \boldsymbol{R} & \boldsymbol{t} \\ \boldsymbol{0^T} & \boldsymbol{1} \end{bmatrix} \boldsymbol{x} \tag{4}$$

where $\boldsymbol{t}$ is 2D translation vector and $\boldsymbol{R} \in \mathrm{O}(2)$, where

$$O(2) = \{\boldsymbol{R} \in \mathbb{R}^{2x2} | \boldsymbol{RR^T} = \boldsymbol{I} \text{ and det } \boldsymbol{R} = \pm 1\} \tag{5}$$

is the second order orthogonal group.

$$\boldsymbol{R} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \in SO(2) \tag{6}$$

is a valid 2x2 matrix when $\boldsymbol{RR^T}$ and det $\boldsymbol{R} = -1$. Given this, $\boldsymbol{H_e} \in \mathrm{SE}(2)$ is a 3x3 homogeneous transformation matrix.

The transformation is a rigid reflection when

$$\boldsymbol{R} = \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix} \in SO(2) \tag{7}$$

which is a 2 x 2 reflection matrix where $\boldsymbol{RR^T} = I$ and det $\boldsymbol{R} = -1$.

A Euclidean transformation will have length and area as invariants, and in addition, all the invariants of a similarity transformation.

2.2.2 Similarity

The second transformations is similarity, described as

$$\boldsymbol{x'} = \boldsymbol{H_s x} = \begin{bmatrix} \boldsymbol{sR} & \boldsymbol{t} \\ \boldsymbol{0^T} & \boldsymbol{1} \end{bmatrix} \boldsymbol{x} \tag{8}$$

s is the scaling factor and $\mathbb{R} \in \mathrm{O}(2)$ is a rotation matrix or a reflection matrix. A similarity transformation reduces to a Euclidean transformation when s = 1. The inverse is

$$\boldsymbol{H_s^{-1}} = \begin{bmatrix} (1/s)\boldsymbol{R^T} & -(1/s)\boldsymbol{R^T} \\ \boldsymbol{0^T} & \boldsymbol{1} \end{bmatrix} \tag{9}$$

Similarity transformations will have a ratio of lengths and angles as invariants, and in addition, all the invariants of an affine transformation.

2.2.3 Affine

$$\boldsymbol{x'} = \boldsymbol{H_a x} = \begin{bmatrix} \boldsymbol{A} & \boldsymbol{t} \\ \boldsymbol{0^T} & \boldsymbol{1} \end{bmatrix} \boldsymbol{x} \tag{10}$$

where $\boldsymbol{A}$ is any nonsingular 2 x 2 matrix. There exists circumstances where $\boldsymbol{A} = sR$ where $\boldsymbol{R} \in \mathrm{O}(2)$, which makes an affine transformation equal to an similarity transformation.

Inverse transposed affine transformations used in the transformation of lines is described as

$$\boldsymbol{H_a^{-T}} = \begin{bmatrix} \boldsymbol{A} & \boldsymbol{t} \\ \boldsymbol{0^T} & \boldsymbol{1} \end{bmatrix} \boldsymbol{x} \tag{11}$$

It further states in the compendium that Affine transformations has the following invariants:

- 1. Collinear points, which are three or more points on the same line, are transformed to collinear points.
- 2. Parallel lines will be transformed two parallel lines.
- 3. The ratio of lengths for parallel lines is invariant
- 4. Convex sets are transformed to convex sets.
- 5. Centroids of vectors are invariant.

2.2.4 Projectivity

Projective transformation is written as

$$\boldsymbol{x'} = \boldsymbol{H_p x} = \begin{bmatrix} \boldsymbol{A} & \boldsymbol{t} \\ \boldsymbol{v}^T & v_3 \end{bmatrix} \boldsymbol{x} \tag{12}$$

Projective transformations includes invariants of collinearity of points, intersection of lines, tangency, tangent discontinuities and cross ratios.

The projective transformation can be decomposed into

$$\boldsymbol{H} = \boldsymbol{H}_s \boldsymbol{H}_{as} \boldsymbol{H}_{ps} = \begin{bmatrix} s\boldsymbol{R} & \boldsymbol{r} \\ 0^T & 1 \end{bmatrix} \begin{bmatrix} \boldsymbol{K} & 0 \\ \boldsymbol{0}^T & 1 \end{bmatrix} \begin{bmatrix} \boldsymbol{I} & 0 \\ \boldsymbol{v}^T & v_3 \end{bmatrix} = \begin{bmatrix} s\boldsymbol{RK} + \boldsymbol{r}\boldsymbol{v}^T & 0 \\ \boldsymbol{v}^T & v \end{bmatrix} \tag{13}$$

$\boldsymbol{H_s}$ is similarity transformation, $\boldsymbol{H_a s}$ is affine transformation, $\boldsymbol{H_p s}$ the projective transformation and $\boldsymbol{K}$ is the upper triangular with a det $\boldsymbol{K} = 1$.

2.3 PnP

Fischler and Bolles first introduced the Perspective-n-Point in 1981 [36] to establish the camera pose with respect to an object. The method uses known 3D points with respect to the world and corresponding 2D normalized image coordinates that are projected in the image plane to calculate the transformation between the camera frame and the world frame. PnP will be explained up to P4P because this report uses $n = 4$ number of points to calculate pose.

$$\lambda \boldsymbol{p}_c = \boldsymbol{K}[\boldsymbol{R} \mid \boldsymbol{t}]\boldsymbol{x} \tag{14}$$

where λ is a scaling factor for image point, $\boldsymbol{x}$ is the homogeneous 3D world coordinates and $\boldsymbol{p}_c$ is the corresponding 2D projected image points located in the image plane Figure (5). $\boldsymbol{K}$ is the intrinsic camera parameters (2) and R and t is cameras 3D rotation and 3D translation respectively. Also known as the the extrinsic parameters (3).

$$\lambda \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{f}{p_w} & k & u_0 \\ 0 & \frac{f}{p_h} & v_0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{15}$$

In an instance where the PnP solution utilized 0 image points, the solution has 6 degrees of freedom (DOFs), where 3 of them are describing rotation, and the other 3 are for translation (x, y, z). This would commonly be written as P0P and would not have enough data to estimate orientation nor translation.

For P1P, one point is fixed for an object in the image frame. It makes it so that the one point can rotate in all 3 directions, and it can move along the perspective projection line as it would not change the perception if one is looking through the image plane. What is constrained is that the point can no longer move in u or v direction in the image plane. This means that there are 2 DOFs are now constrained, and 4 DOFs are still free.

For P2P, two points in the image frame are fixed. This will result in both points are constrained along the perspective projection line, and it will consequently mean that two rotations are constrained for the object. It can still rotate about a line formed by the two points and translate along the perspective projection lines. This leaves it such that 4 DOFs are constrained, and two are free.

For P3P, one can imagine a triangle. This leaves 0 DOFs free, and all are constrained. It seems like it is solved now, but it does have 8 possible solutions. It show be noted that 4 of the solutions are in front of the camera and 4 behind the camera. The 4 solution presents the same translation, but the rotation is ambiguous as illustrated in Figure (3) and Figure (4).

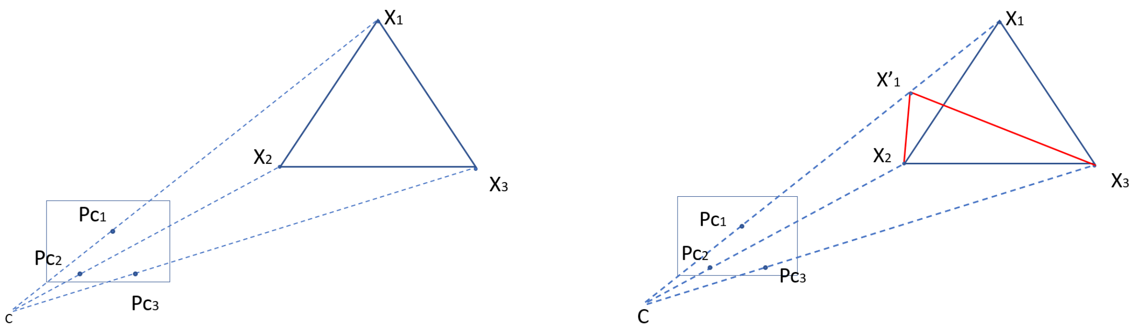


Figure 3: Illustrating two of the potential solutions

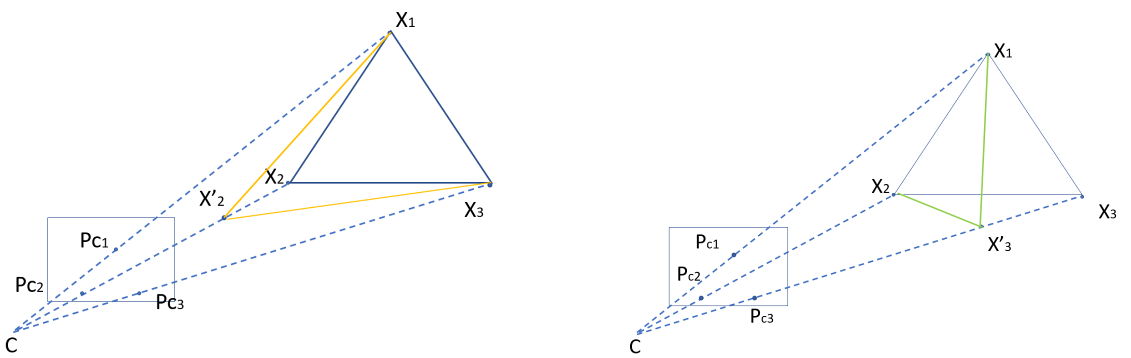


Figure 4: Illustrating the last 2 possible solutions that P3P can have in front of camera

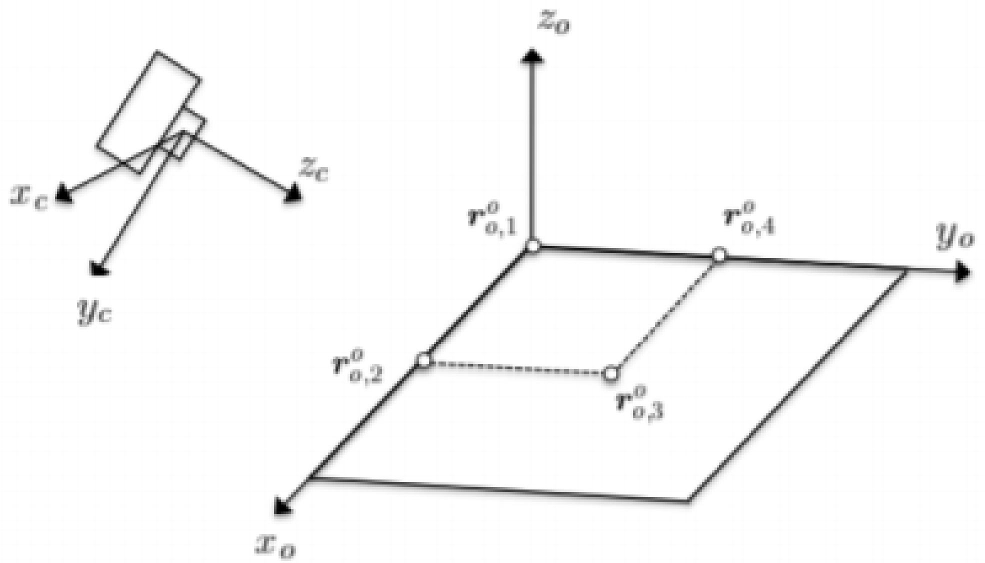


Figure 5: Figure showing setup of P4P. Illustration is taken from p. 74 [1].

Now, an example of P4P with points in a plane from [1] is presented where the rotation and translation between camera frame c and object frame o is presented, as illustrated in Figure (5). The technique uses 4 points in a plane $\boldsymbol{\pi}$. The transformation or pose is

$$\boldsymbol{T}_o^c = \begin{bmatrix} \boldsymbol{R} & \boldsymbol{t} \\ \boldsymbol{0^T} & 1 \end{bmatrix} \tag{16}$$

where $\boldsymbol{R} = \boldsymbol{R}_o^c$ and $\boldsymbol{t} = \boldsymbol{t}_{co}^c$. The four world points $r_{0,1}^o, r_{0,2}^o, r_{0,3}^o, r_{0,4}^o$ are fixed in the plane $\boldsymbol{\pi}$ with homogeneous coordinates $r_{0,i}^o = (x_i, y_i, 0, 1)^T$, $i = 1, 2, 3, 4$ and all points are observed in the image frame.

The normalized image coordinates $\tilde{\boldsymbol{s}}_{\boldsymbol{i}}$ are

$$\lambda_i \tilde{\boldsymbol{s}}_{\boldsymbol{i}} = \boldsymbol{\Pi} \tilde{\boldsymbol{r}}_{\boldsymbol{c,i}}^{\boldsymbol{c}} = \boldsymbol{\Pi} \begin{bmatrix} \boldsymbol{R} & \boldsymbol{t} \\ \boldsymbol{0^T} & 1 \end{bmatrix} \tilde{\boldsymbol{r}}_{\boldsymbol{o,i}}^{\boldsymbol{o}} \tag{17}$$

where λ_i is the depth coordinate set to unity as one can freely select scaling and

$$\boldsymbol{\Pi} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{18}$$

This can be rewritten as

$$\lambda_i \tilde{\boldsymbol{s}}_{\boldsymbol{i}} = \begin{bmatrix} \boldsymbol{R} & \boldsymbol{t} \end{bmatrix} \tilde{\boldsymbol{r}}_{\boldsymbol{o,i}}^{\boldsymbol{o}} \tag{19}$$

and further in the compendium it is stated that since z is set to zero for every $r^o_{0,i}$ the can be written as a homography

$$\tilde{\boldsymbol{s}}_{\boldsymbol{i}} = \boldsymbol{H}\tilde{\boldsymbol{x}}_{\boldsymbol{i}} \tag{20}$$

where

$$\boldsymbol{H} = \begin{bmatrix} \boldsymbol{r_1} & \boldsymbol{r_2} & \boldsymbol{t} \end{bmatrix} \tag{21}$$

and

$$\tilde{\boldsymbol{x}}_{\boldsymbol{i}} = \begin{bmatrix} \boldsymbol{x_i} \\ \boldsymbol{y_i} \\ \boldsymbol{1} \end{bmatrix} \tag{22}$$

The planar homography $\boldsymbol{H}$ can now be found. With $\boldsymbol{H}$ established, it can be used to calculate $\boldsymbol{T}^c_o$. With columns of $\boldsymbol{H}_j = \begin{bmatrix} \boldsymbol{h}_1 & \boldsymbol{h}_2 & \boldsymbol{h}_3 \end{bmatrix}$ and $\boldsymbol{h} = \boldsymbol{H}^T_j$. with

$$\boldsymbol{Ah} = \boldsymbol{0} \tag{23}$$

and

$$\boldsymbol{A} = \begin{bmatrix} \boldsymbol{A_1} \\ \boldsymbol{A_2} \\ \boldsymbol{A_3} \\ \boldsymbol{A_4} \end{bmatrix} \tag{24}$$

$\boldsymbol{A_i}$ is found by point mapping i and $\boldsymbol{h}$ with singular value decomposition (SVD) which is defined as

$$\boldsymbol{A} = \sum_{i=1}^{9} \sigma_i \boldsymbol{u}_i \boldsymbol{v}_i^T, \;\; \boldsymbol{u}_i \in \mathbb{R}^{12}, \boldsymbol{v}_i \in \mathbb{R}^9 \tag{25}$$

The example from the compendium then explains that the column vector of $\boldsymbol{H}$ is obtained with

$$\boldsymbol{r}_1 = k\boldsymbol{h}_1 \tag{26}$$

$$\boldsymbol{r}_2 = k\boldsymbol{h}_2 \tag{27}$$

$$\boldsymbol{t} = k\boldsymbol{h}_3 \tag{28}$$

with scaling being

$$k = \frac{sgn(\boldsymbol{v}_9(9))}{||h_1||} \tag{29}$$

with sign selected for a positive z value in the translation $\boldsymbol{t}$.

The last column vector in the rotation vector is found with

$$\boldsymbol{r_3} = \boldsymbol{r_1} \times \boldsymbol{r_2} \tag{30}$$

2.4 Intrinsic Camera Calibration

The purpose of calibrating a camera is to find the intrinsic camera matrix $\boldsymbol{K}$ and its distortion coefficients, which are used in to calculate the normalized image coordinates $\tilde{s}$ with (2.1) from the pixel coordinates $\tilde{p}$.

The intrinsic camera matrix can be represented as $\boldsymbol{K}$, as done in (2.1)

The 5 intrinsic parameters that have been estimated entails data on the focal length, principal point, and image sensor format. In addition to this, the 5 non-linear lens distortion coefficients are found but cannot be represented in the linear camera matrix.

The lens distortion coefficients account for radial, tangential- and Thin prism lens distortions.

Types of distortion in images can be barrel distortion, pincushion distortion, and mustache distortion. It is important to account for this when calculating correlations between 2D projective planes and the 3D world in photogrammetry. Illustrations of distortion is shown in Figure (6).

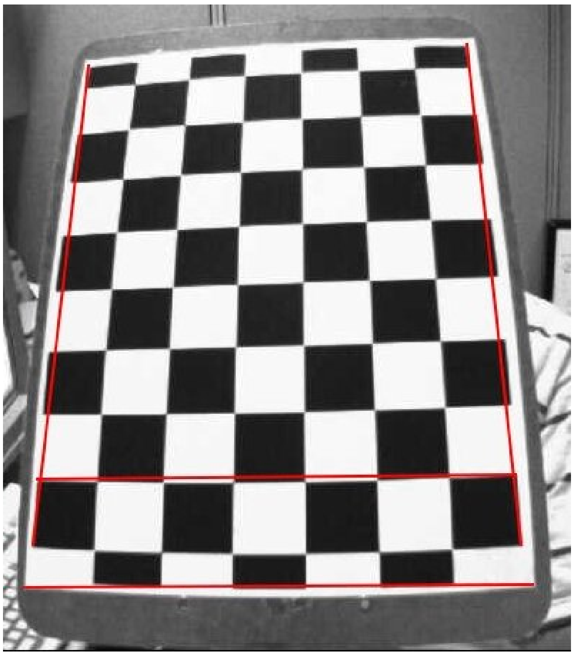


Figure 6: Example of radial distortion in a camera. One knows that in 3D world the lines are straight, but in the image the lines are being radially distorted which can be a problem when calculation the homographies. Illustration is taken from [2].

Non-linear intrinsic parameters such as lens distortion are also important, although they cannot be included in the linear camera model described by the intrinsic parameter matrix. Many modern camera calibration algorithms estimate these intrinsic parameters as well in the form of non-linear optimization techniques. This is done to optimize the camera and distortion parameters in what is generally known as bundle adjustment.

Lenses usually have radial distortion and a small tangential distortion. To account for this, first the normalized image coordinates are calculated in Equation (2.3), then afterwards according to openCV documentation under section Pinhole camera Model the distortion coefficients are accounted for with the formulas written as [37]:

$$\begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} f_x x'' + x_0 \\ f_y y'' + y_0 \end{bmatrix} \tag{31}$$

where

$$\begin{bmatrix} x'' \\ y'' \end{bmatrix} = \begin{bmatrix} x' \frac{1+k_1 r^2 + k_2 r^4 + k_3 r^6}{1+k_4 r^2 + k_5 r^4 + k_6 r^6} + 2p_1 x' y' + p_2(r^2 + 2x'^2) + s_1 r^2 + s_2 r^4 \\ y' \frac{1+k_1 r^2 + k_2 r^4 + k_3 r^6}{1+k_4 r^2 + k_5 r^4 + k_6 r^6} + p_1(r^2 + 2y'^2) + 2p_2 x' y' + s_3 r^2 + s_4 r^4 \end{bmatrix} \tag{32}$$

with

$$r^2 = x'^2 + y'^2 \tag{33}$$

and

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} X_c/Z_c \\ Y_c/Z_c \end{bmatrix} \tag{34}$$

if $Z_c \neq 0$

The radial distortion coefficients are $k_1, k_2, k_3, k_4, k_5, k_6$

The tangential distortion coefficients are p_1, p_2

Thin prism distortion coefficients are: s_1, s_2, s_3, s_4

Two well-known methods of intrinsic camera calibration are Zhang's method [38] and Bouguet [39].

2.4.1 Spatial resolution

Spatial resolution describes the relationship between pixel resolution and 3D Euclidean space. It will affect how accurately a digital camera may measure objects. I.e., in the more extreme circumstance in Figure (8), when a satellite is capturing an image of a house, if the spatial resolution is so that one pixel captures 30 square meters in euclidean space, the output image will not be able to differentiate the house and its surroundings, and the pixel will output one color.

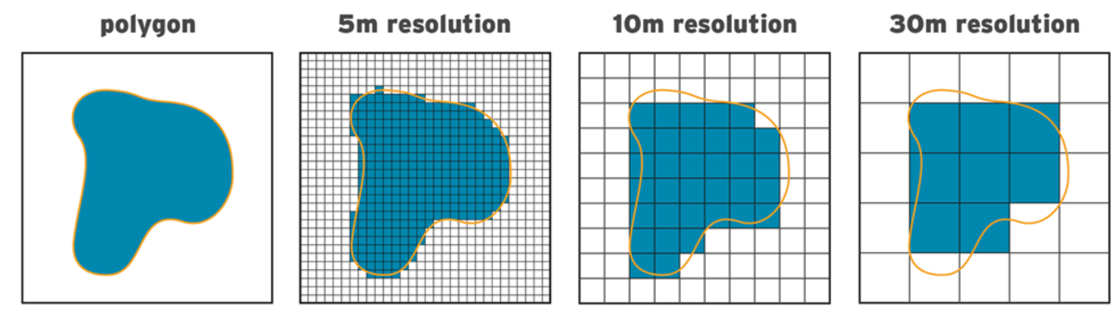


Figure 7: How different resolution affects the output image of a polygon. Illustration is taken from [3]

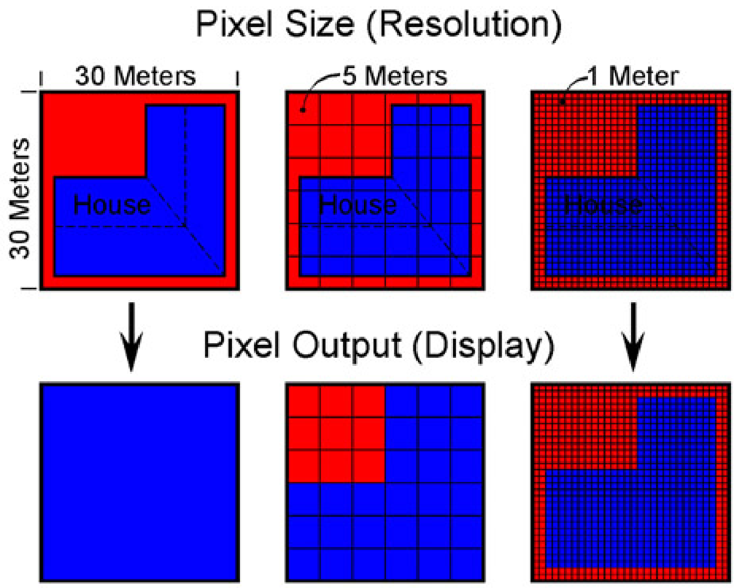


Figure 8: How different resolution affects the output image's precision. This is illustrated for spatial resolution affects for satellites on houses. Illustration is taken from [4]

2.5 Deep Learning

P4P has been explained in previous sections as a way to calculate the pose with the Equation (2.3). What is missing now is a method to find the image points (u, v). Deep learning will be part of the solution for extracting these image points and therefore it is important to understand how deep learning works in the context in object detection in images.

Artificial intelligence or AI for short, is a broad concept based on making intelligent computers to act similarly to how humans do. A branch in the field of AI is machine learning (ML). ML is is way to learn a computer based on data without explicitly programming what it should do. Deep learning is a subgroup of machine learning that process data in multiple layers call artificial neural networks (ANN).

First, basic theory on deep learning with the focus on applications within image data. Following comes an introduction of important definitions. Finally, the deep learning model used in this report will be introduced, named Mask R-CNN.

Here is a list of different terminology used in deep learning and will be used throughout this report.

Table 1: Deep learning Acronyms and full word

Acronym	**Full word**
AI	Artifical Intelligence
ML	Machine learning
DL	deep learning
NN	Neural network
CNN	Convolutional neural network
TL	Transfer learning
Bbox	bounding box
KP	Keypoint
GT	Ground truth
IoU	Intersection over union
Non-maximum suppression	NMS
AP	Average precision
mAP	mean Average precision
FPN	Feature Pyramid Network

In recent years increased use of Computer vision (CV) has been observed. This is much due to stronger and cheaper computer processing and industry 4.0 [40]. Lack of computational power was a limiting factor before the mid-2000s [41]. Within AI, a deep learning architecture named Convolutional Neural Networks (CNNs) has been developed.

As described in [12], CNN's are commonly applied to work with problems with a grid-like topology. Examples of this are images or time series. CNN's in recent years are seeing rapid development and is peaking now in the CV field [42].

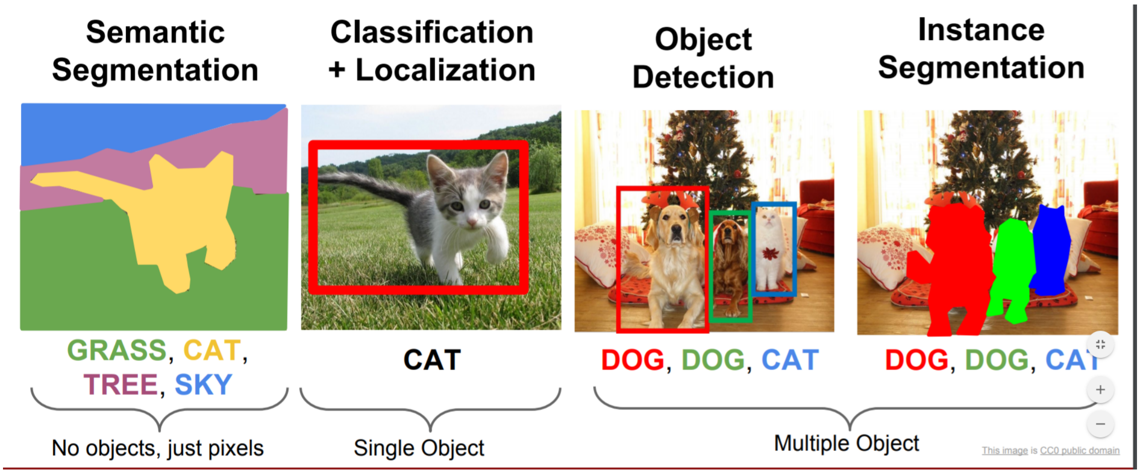


Figure 9: Examples of how a CNN can be used with image datasets. Illustration is taken from [5]

A common representation of images is a 3-dimensional matrix, where the depth dimension is 3 layered and consists of representations of red, green, and blue color intensity. Applying this data form to a CNN can help train an AI model for object detection, image classifications, semantic segmentation, scene understating, image generation and more [43]. Examples of these can be seen in Figure (9).

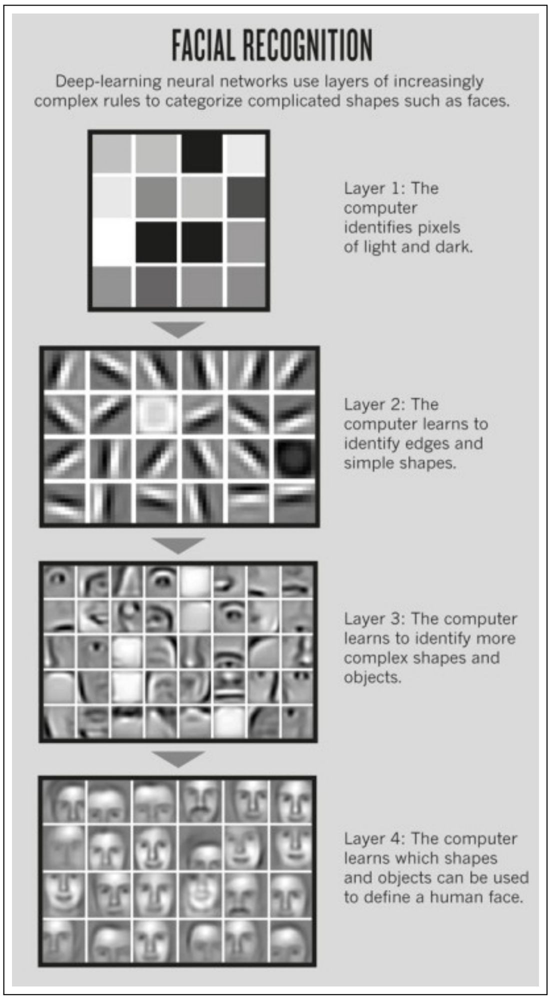


Figure 10: How different layers can operate to recognize faces. Illustration is taken from [6]

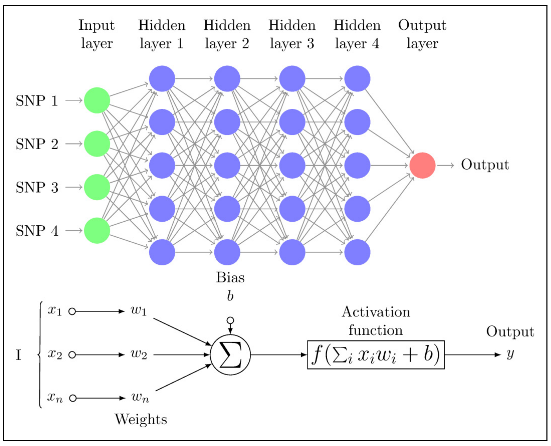


Figure 11: Internal workings of a DL NN. Illustration is taken from [7]

Deep learning consists of deep neural networks (NN) with a potential large amount of neurons. Based on the calculations of these neurons, the deep learning model will be able to, i.e. predict objects. A hidden layers is illustrated in Figure (11). These layers will have different tasks and inputs. An example is illustrated in Figure (10) to detect human faces.

2.5.1 Current Applications of Deep learning

There are several use cases for Deep learning. Object detection (9) is an important use case. Object detection in computer vision is about introducing the program to an image, and from this, the program will classify and localize the object represented in pixel coordinates. One can observe this use case in three major industries; autonomous driving, the medical field, and the gaming industry.

2.5.1.1 Autonomous Driving In autonomous driving, the end goal is to achieve level 5 autonomous driving, which entails that it is in no need of a human operator to survey the driving operation. Most notably has Tesla's autonomous driving system received lots of attention in recent years. It utilizes several input devices, including cameras with object detection to classify and localize different traffic components such as traffic lights, other cars, and road surface marking.

Figure 12: This is an example of how Object detection works in cars. Illustration is taken from [8]

2.5.1.2 Medical Field In the medical field, it is used to analyze image data from various types of images. It can be looking for Glaucoma in the eye, analyze X-ray images and more [9].

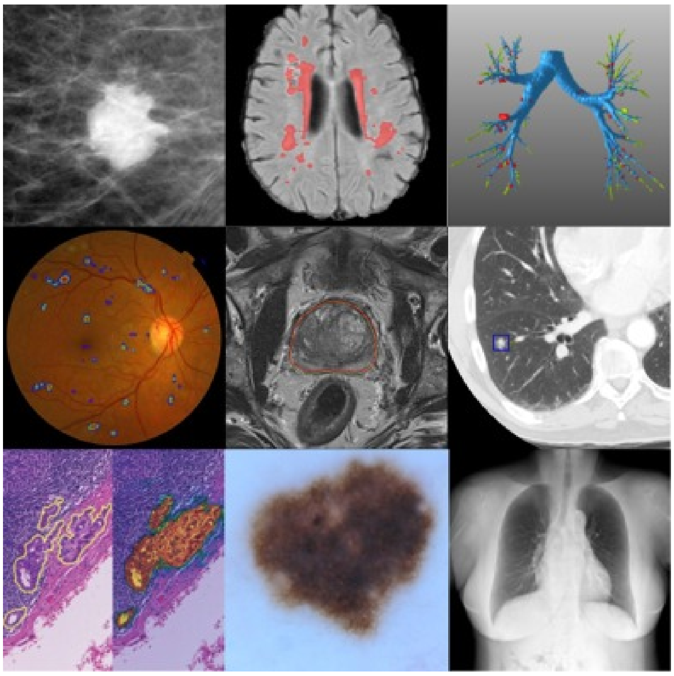


Figure 13: Illustration is taken from [9], illustration application areas of deep learning

2.5.1.3 Human Pose estimation Human pose estimation is a researched field that utilized deep learning to identify joints in the human body. What is very similar to this project is that it uses object detection on different joins and then assigns them a point. This type of human pose estimation can analyze the athlete's movement patterns, create realistic movements in videogames and more [44, 45].

2.5.2 CNN

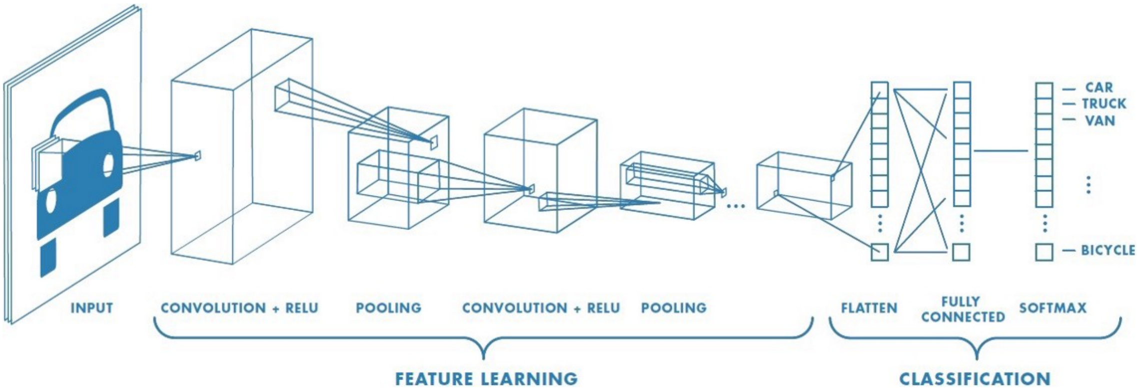


Figure 14: An example for a type of CNN architecture. It uses convolution + ReLU and pooling for learning features. Illustration is taken from [10], accessed 25.11.20.

Origin of the name Convolutional Neural Network is derived from convolution, which is an algorithm that weighs data based on a set of given values that can be time-series or adjacent data in matrices in images. An example for this in time series is heavily inspired by the story presented in [12]:

A scientist uses a laser to measure the position of a moving vehicle. The laser reading of position is only valid for a short amount of time before the vehicle has been displaced to a new position. To solve this, the scientist will use the laser to read positions with a higher frequency. The AI time series's role in this part is that you can tell the AI to prefer using the newest readings and use the older readings to attempt to predict new measurements. In mathematical terms, the distance measured is given by $s(t)$, where t is time. The measurements are noisy, and therefore several measurements are conducted with this high frequency. If one uses these measurements' average, one knows that the older measurements are less relevant than the newer ones. To make the newer measurements more relevant or in other words, weigh heavier, the following formula is used:

$$s(t) = \int x(a) \cdot w(t-a)da \tag{35}$$

where $x(a)$ is the measurement with respect to the age a of the scan, w is the weight/kernel that varies with time/relevance of input. w is true for $w \in \mathbb{R} \geq 0$, because the negative weighted function in this example would indicate that the measurement came from the future.

For images, the weighted function would be a filter, often referred to as a kernel, to look over the matrix representing an image to identify features such as edges. This filter tends to be a significantly smaller matrix than the image matrix. Depending on the kernel size and image size, the kernel uses weighted functions to extract the features in the image.

$$(i, j) = (K * I)(i, j) = \sum_m \sum_n I(i - m, j - n)K(m, n) \tag{36}$$

This data is sent to different neurons that train individually, as illustrated in Figure (11). Some neurons will train on recognizing certain features, and combined will these neurons be capable of identifying complex features.

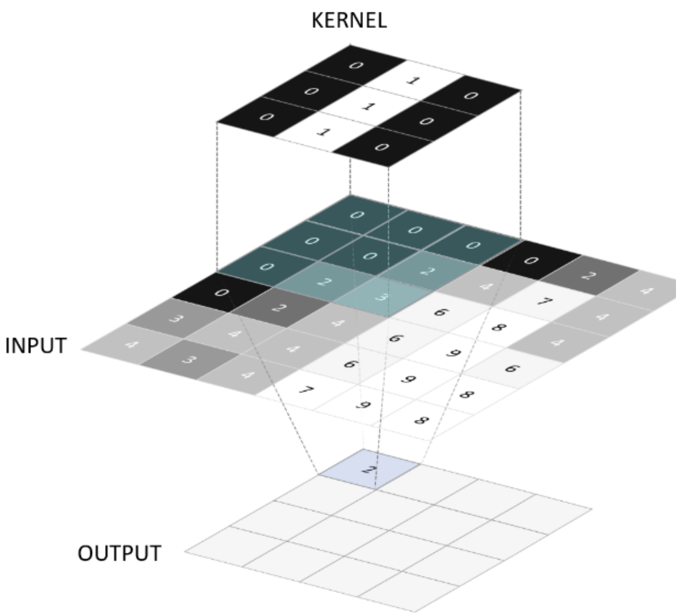


Figure 15: Example of a 3x3 kernel used on an image matrix. an output matrix smaller than the input is generated from this. One output is from by 9 inputs. Illustration is taken from [11]

2.5.2.1 Activation Functions "A neural network without an activation function is essentially just a linear regression model." [46]

"Definition of activation function:- Activation function decides whether a neuron should be activated or not by calculating weighted sum and further adding bias with it. The purpose of the activation function is to introduce non-linearity into the output of a neuron." [47]

These neurons will take the summed weights + biases as input and use them in an activation function to check whether to activate the neuron or not depending on the value of the activation function (11). If the output is True, the neuron will send its weights to the next layers. This method is inspired by how the brain works to process data as briefly discussed in section "Brief overview of neural networks" in the article [46]. The most commonly used activation function is called Rectified Linear Unit, also known as ReLU [46].

To demonstrate how ReLU works for a neuron:

$$f(x) = \max(0, x) \tag{37}$$

$x = xw + b$, where b are biases. These biases are constants that are added to the summation of weights before it is used as input the the activation function. The output with ReLU will either return 0 or x, depending on what the value of x is. If $x \geq 0$, then output x. If $x < 0$, then return 0 If the function returns zero the neuron will not be activated (return False) and if x is the output, then the neuron will be activated (return True). If a bias is applied with the ReLU activation function (37), it guarantees that it will activate the neuron to some degree.

2.5.2.2 Connectivity In Neural networks (NN), the neurons can be connected in various ways. One of these is fully connected layers. Fully connected layers process more information and lead to greater accuracy, but it is more computationally expensive than sparse connectivity. The Figure (16) from [12] below illustrates two different connections.

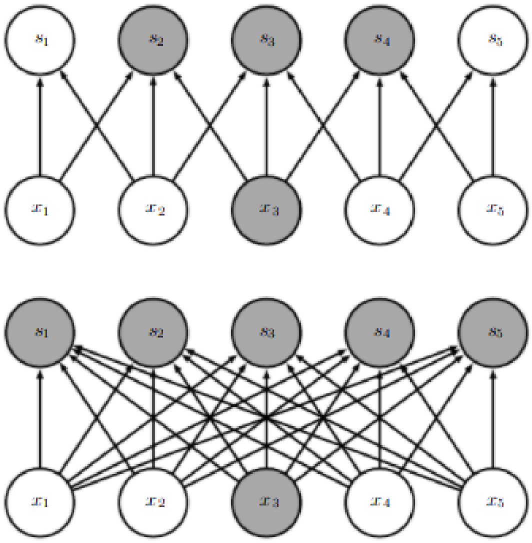


Figure 16: "Sparse connectivity, viewed from below. We highlight one input unit, x_3, and highlight the output units in $\boldsymbol{s}$ that are affected by this unit. *(Top)*When $\boldsymbol{s}$ is formed by convolution with a kernel of width 3, only three outputs are affected by $\boldsymbol{x}$. *(Bottom)*When $\boldsymbol{s}$ is formed by matrix multiplication, connectivity is no longer sparse, so all the outputs are affected by x_3" [12]

2.5.2.3 Pooling In the example Figure (14), pooling layers comes after the convolution layer and activation function.

"In all cases, pooling helps to make the representation become approximately invariant to small translations of the input. Invariance to translation means that if we translate the input by a small amount, the values of most of the pooled outputs do not change." -Page 342, [12]

How does pooling work to achieve this? Underneath is a Figure (17) illustrating the calculation of the two most common pooling techniques, max pooling and average. It should be explained that a stride length of n is how many units the kernel moves in between calculating one pooling feature.

In a max pooling configuration, the pooling layer uses a kernel over the input image, searching for the largest pixel value and collecting the output into what is known as a feature map.

Similar to max pooling, the average pooling technique calculates the average of these numbers and adds this number to the feature map.

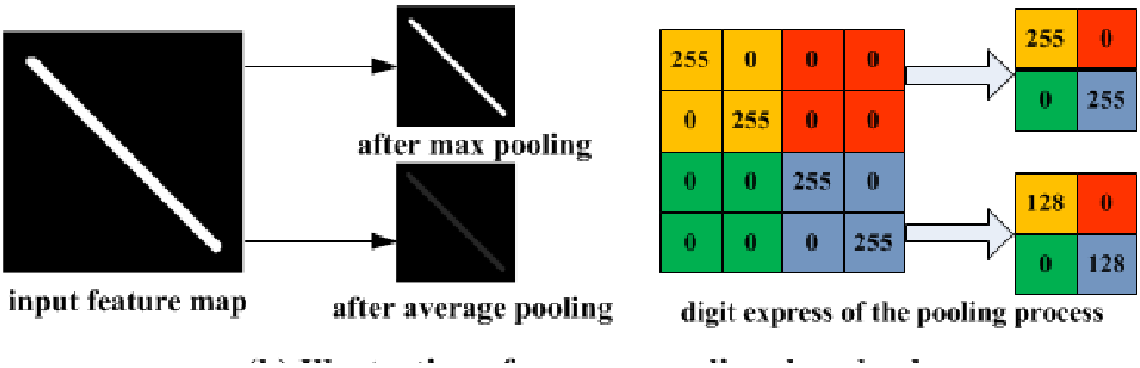


Figure 17: Illustration of how different pooling may affect the feature extraction. Kernel size is 2x2 and moves with a stride length of 2. An example of how max pooling operates. Colors in input and output correspond to values being weighted. Illustration is taken from [13]

2.5.3 Classification

"Classification is the process of predicting the class of given data points. Classes are sometimes called targets/ labels or categories. Classification predictive modeling is the task of approximating a mapping function (f) from input variables (X) to discrete output variables (y)."- Sidath Asiri[48]

"Object detection is a computer vision task that involves both localizing one or more objects within an image and classifying each object in the image.

It is a challenging computer vision task that requires both successful object localization in order to locate and draw a bounding box around each object in an image, and object classification to predict the correct class of object that was localized."- Jason Brownlee [49]

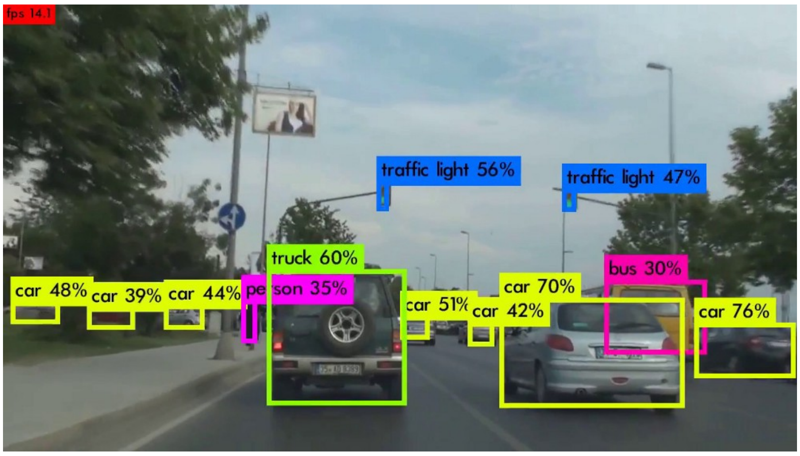


Figure 18: This image illustrates a CNN model predict objects in an image. It has been trained to identify certain objects and return a probability for correct prediction. It needs to identify the right class and decide coordinates for bounding boxes. Illustration is taken from [14]

Softmax is an extension that allows multiple classes in a model. See Figure (14) for how Softmax is used in the classification part of a CNN.

2.5.4 Shortcomings

There are multiple reasons where the deep learning approach is not always the best. Some of the reasons as to why it is not according to Donges [50] follows:

Black boxes - There are so many parameters in a NN that it can be very tough to troubleshoot an NN. Therefore, a NN can be reffered to as an black box. in Figure (19) is a good example. It feeds an image of a cat into the NN. It its output is clear in this figure, the model has predicted that the image is a cat with a 0.97 in probability, where 1.00 is absolute certainty. One cannot be sure of all the calculations made underway, so it can be tough to troubleshoot when the model does not return the expected outcome.

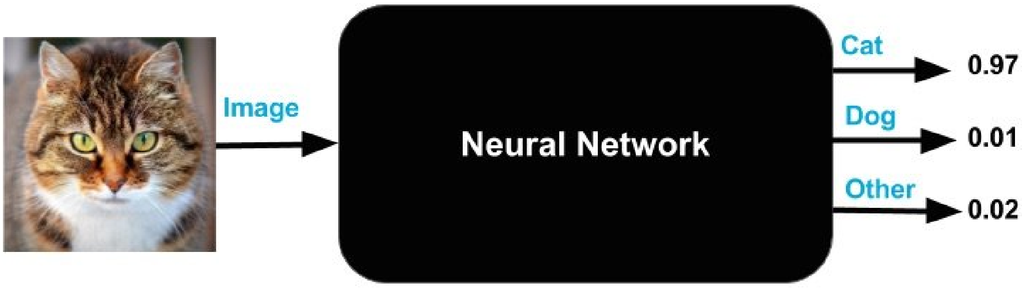


Figure 19: It can be difficult to troubleshoot the NN if the the output was not as expected. Illustration is taken from [15]

Computationally expensive - It requires lots of processing power to train the model. A model requiring a Powerful GPU is not uncommon, depending on the settings chosen and dataset.

Data hungry - Deep learning models require lots of data to achieve good results. The amount of data can vary from project to project, but say thousands to millions of image data with human-made annotations to all images may be required to develop this model. This can be a very time-consuming part. Labelimg is a annotation tool used with its interfaces illustrated in Figure (20).

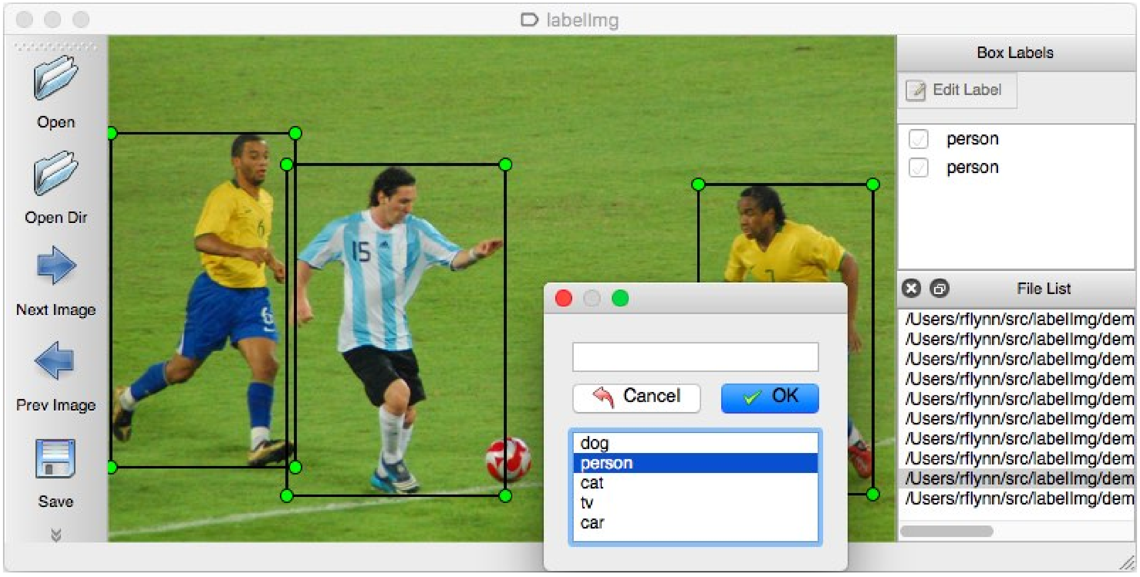


Figure 20: Program called labelImg [16] can be used to annotate each image with a bounding box. These are the ground truths (GT) used in training.

Duration of development - The time preparing dataset and training can be hours, days, weeks, even months depending on the size of data and computational power at your disposal.

Position and Orientations

CNN uses multiple neurons in the NN with different filters to present features to identify objects in a given image. The problem that can occur is that CNN can identify different features and conclude, but the object can have wrong positions and orientations relative to each other. This is illustrated in Figure (21).

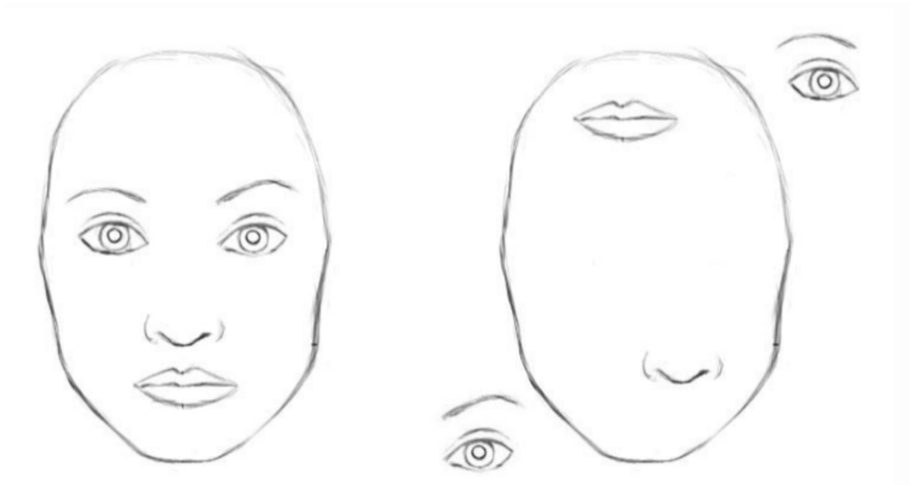


Figure 21: A difficult scenario for a CNN, as both contain the required details of a human face. Illustration is taken from [17], accessed: 12.09.20.

A NN computes to extract features. For example, in some layers, extracting features for identifying the face's contours, the model will highlight this but simultaneously overshadow the eyes, nose, and mouth. This effectively means that it identifies a face without looking at eyes, nose, and mouth in the face's context. When the model extracts features for the eyes, nose, and mouth, it will not look in the context of the rest of the face, just individually these features. This means that CNN can struggle with larger contexts. This can, in some circumstances, lead to the model returning a false positive.

Underfitting - "A model is said to be underfitting when it's not able to classify the data it was trained on." [51] For example: The model has been trained to classify dogs and cats, but when tested on the training image data with a cat, it fails to identify the cat.

In context, a model is trained on a dataset. If it cannot predict well on a test image it previously has possessed the solution to, it will likely struggle when tested on a never before seen image data.

Workarounds for this is among other things:

- Increase the number of layers.

- Increase number of neurons in the layers.
- Change type and location of layers.
- Increase the amount of data. A powerful tool for this is data augmentation.

Increasing the model's complexity requires more computational power, so it is a trade-off that has to be done.

Overfitting - "Overfitting occurs when our model becomes really good at being able to classify or predict on data that was included in the training set, but is not as good at classifying data that it wasn't trained on. So essentially, the model has overfit the data in the training set." [52]

During training, one can analyze the metrics in the training set and validation set. If the training set is considerably better than the validation, it indicates that it has been overfitted. It has been very well-adjusted to the training, and there fails to generalize objects. That is why it struggles to classify the objects in the validation set because it has been too good to classify the data as presented in the training set.

Iou can be defined as:

$$IoU = \frac{|A \cap B|}{|A \cup B|} = \frac{|I|}{|U|} \tag{38}$$

Where A and B are bbox's.

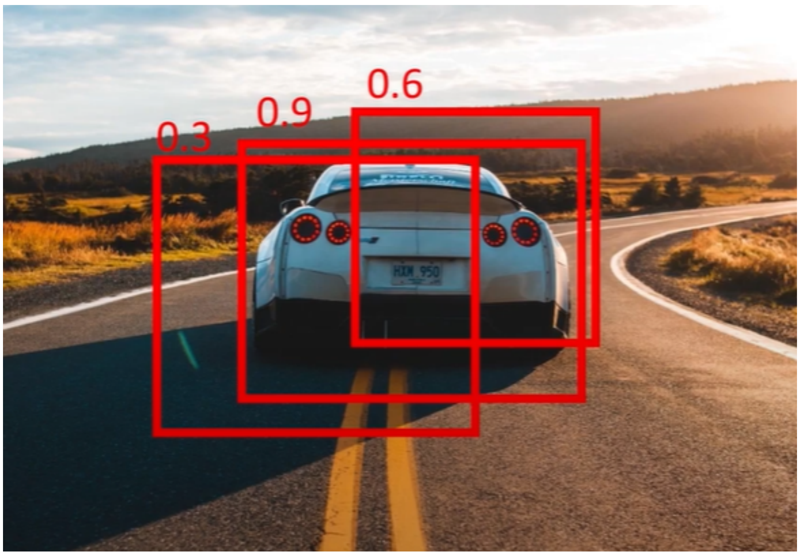


Figure 22: Multiple Predictions are made. A threshold IoU is set. If the IoU is higher than threshold, store the bbox with the highest probability score. The other bbox is assumed to be duplicates. In this image, Only one bbox is needed for this image. Illustration is taken from [18]

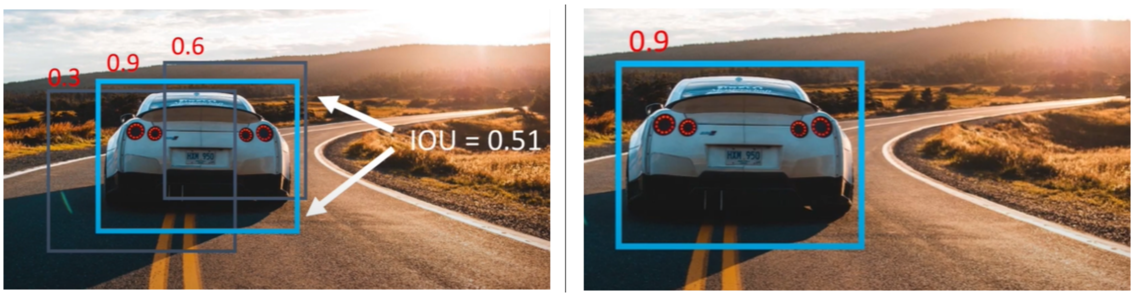


Figure 23: Illustration is taken from Youtube video made by [18]

This method of removing multiple boxes is a method called non-maximum suppression that filters out bboxs with lower confidence score.

2.5.5 Important Definitions in Machine Learning

Here will important definition regarding Deep learning be introduced. It will be important when interpreting the test results later in Section (5).

2.5.5.1 Inference The inference is using a trained model to make a prediction.

True Positive (TP): • Reality: A wolf threatened. • Shepherd said: "Wolf." • Outcome: Shepherd is a hero.	**False Positive (FP):** • Reality: No wolf threatened. • Shepherd said: "Wolf." • Outcome: Villagers are angry at shepherd for waking them up.
False Negative (FN): • Reality: A wolf threatened. • Shepherd said: "No wolf." • Outcome: The wolf ate all the sheep.	**True Negative (TN):** • Reality: No wolf threatened. • Shepherd said: "No wolf." • Outcome: Everyone is fine.

Figure 24: Example of how these definitions work in practice. Illustration from [19]

2.5.5.2 True/False Positives/Negative: To translate the example in Figure (24) to a CNN model, let's say that we have two classes. One is wolf (Positive class) and the other one is background (Negative class). When an image containing a wolf is fed into the CNN, it will analyze the image. (How it analyzes depends on the model chosen.) It will (hopefully) return a wolf (TP) and classify everything else as background, also known as a negative class (TN).

2.5.5.3 Precision

$$Precision = \frac{TP}{TP + FP} \tag{39}$$

Precision is looking at all the predicted classes, then calculates how often does the model predict correctly of these. Every time there is an object corresponding to a class, how often does it guess correct.

2.5.5.4 Recall

$$Recall = \frac{TP}{TP + FN} \tag{40}$$

Out of all Classes in an image, how often does the model find the class? It does not consider how often it is wrong but just considers if it finds all the objects in the image. the model compares its predictions to the GT made in the annotations, shown in Figure (20). I.e., If all GT's in an image are predicted, then the recall is 1, independently of how many times the model makes FP predictions.

2.5.5.5 Training, validation, test When training a model, 3 directories of images are created

- Training dataset
- Validation dataset
- Test dataset

The training dataset consists of images of the class that it is supposed to be trained on, and metadata about the annotated ground truth in each image. This is information about the class and pixel coordinates of bbox, masks in instance segmentation and more. This directory is what the model tries to learn

The validation dataset contains images and annotated ground truths for each image in the folder. The difference is that the model tries to predict on the validation, usually mid training or after the weights has been adjusted. The purpose is to calculate metrics such as precision, recall, AP, mAP to mention some.

The test dataset consists of images only. The purpose is to test the trained model to different inputs to see how it performs.

2.5.5.6 Batch size Say you have 80 images in a dataset with a batch size of 4. What this means is that the model collects 4 images at a time to make predictions. After it has completed the prediction in those 4 images, it will adjust its weights. This effectively means it considers 4 training data, i.e., images before it adjusts its weights. This process requires lots of GPU memory, and the memory may well be the limiting factor for not increasing the batch size. However, the higher the batch size may not always be better.

2.5.5.7 Iterations

When the model has trained through an entire dataset, it has completed 1 iteration. So using iterations larger than 1 will train the model on the same data multiple times.

2.5.5.8 Backbone

"A convolutional neural network that aggregates and forms image features at different granularities." [53]

2.5.5.9 Neck

"The portion of an object detection model that forms features from the base convolutional neural network backbone." [53]

2.5.5.10 Head

"The portion of an object detector where prediction is made. The head consumes features produced in the neck of the object detector." [53]

2.5.5.11 Transfer Learning

Transfer learning is a method of utilizing weights from other projects that can be applied in this project. I.e., another project may have trained and become good at extracting features, so using these pre-trained weights will save our model from relearning all this from scratch. It means that the model has been trained to extract features similar to other projects, such as extracting edges, curves, and more.

2.5.5.12 Freezing layers

A CNN usually consists of several layers. During training, these layers will adjust its weights. Freezing layers entails making layers of neurons immutable, which means they will not adjust during training. This is usually done in a context of utilizing transfer learning as the first few layers are usually well-trained from a previously trained model. So, the layers that has well-adjusted weights should not further adjust itself. A method of freezing these layers in Python is to change the datatype of these layers into tuples.

2.5.5.13 Data Augmentation

Data augmentation is a process of manipulating existing data to create more data. I.e., one can take a dataset containing 500 images, use mirror data augmentation with all these images. It means to mirror all the data images and use them for training your model. Effectively, one now has 1000 images to train instead of 500 if it mirrors each image data once. It exists several data augmentation technique in ML.

2.5.5.14 Hyper parameter

Hyperparameters are parameters that control the learning process of a model.

2.5.6 Mask R-CNN - Mask Region-based CNN

A well-known method used for instance segmentation predictions is Mask R-CNN. It is an extension of Faster R-CNN where an additional branch is added for predicting object masks in parallel with bounding box predictions [20]. According to the original paper [20] it is able to run at 5 fps in 2018 with their hardware.

Short about Faster R-CNN is the history starts with R-CNN came first, then fast R-CNN, and so came faster R-CNN in 2015 [54]. It was at the time the State-of-the-art model for object prediction striving to achieve real-time with Region proposal networks (RPN). In the paper it achieved 5 fps by using a deep layered VGG-16 model.

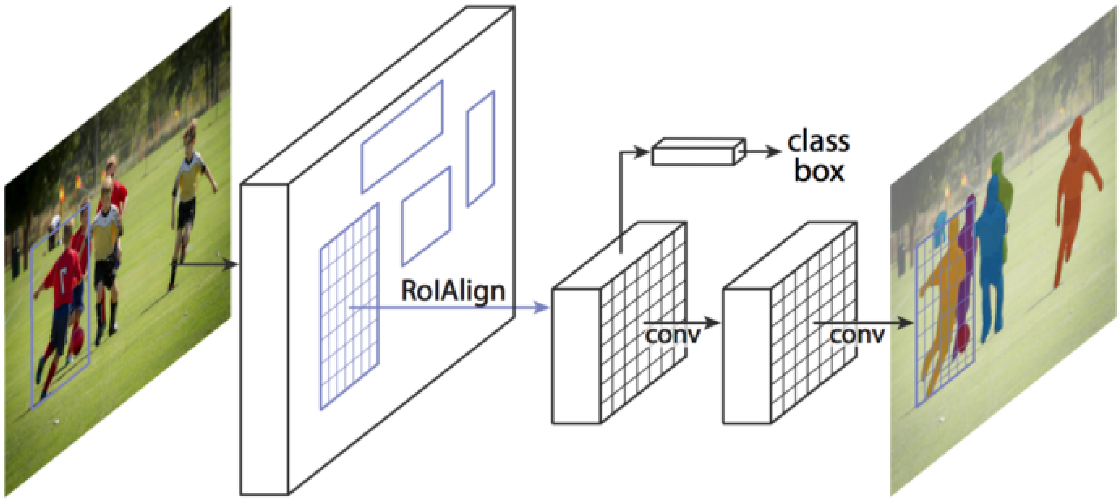


Figure 25: Framework of Mask R-CNN from input image to output. illustration from [20]

At a high level, the framework can be separated into these modules:

2.5.6.1 Backbone Consists of standard CNN, with options of ResNet50 or ResNet101, where 50 and 101 represents numbers of layers.

In addition to this, Feature Pyramid Network (FPN) is used as backbone. FPN was introduced by the authors of Mask R-CNN as a tool for representing different objects at various scales. Normally, feature maps are passes from lower to higher level, but FPN passes high level features to low level. This way, features at every level can be accessed.

2.5.6.2 RPN - Regional Proposal Network "RPN is a The RPN is a lightweight neural network that scans the image in a sliding-window fashion and finds areas that contain objects." [21]

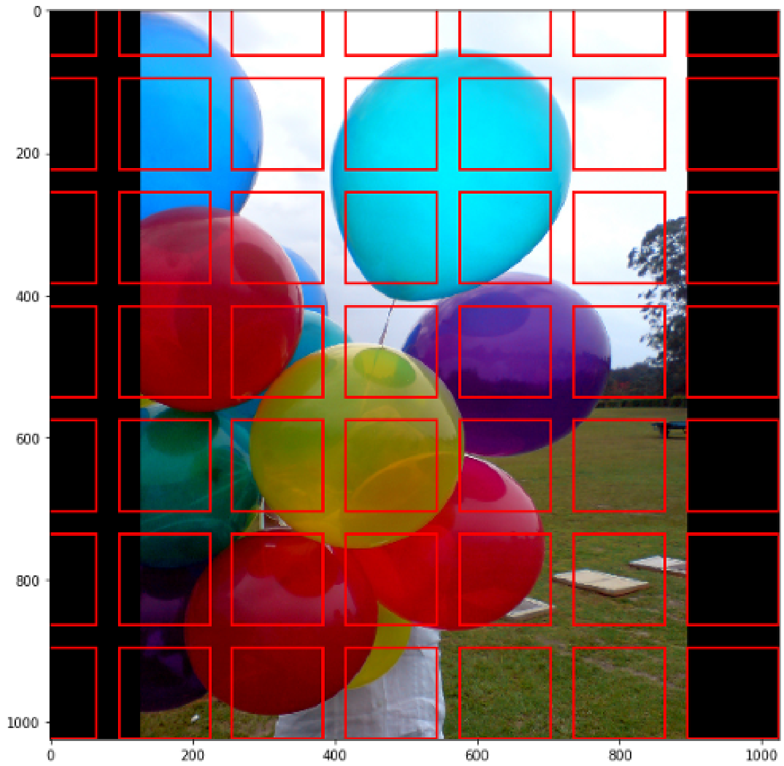


Figure 26: 49 anchors from RPN. Illustration from [21]

In reality it scans the backbone feature map and not the input image itself. It is not uncommon to use 200 000 anchors with various size, aspect ratios and overlapping anchors in the feature map. It can run in about 10 ms due to parallel computing with GPU according to original Faster R-CNN paper [22].

For each anchor in the RPN, it will generate anchor class and bounding box refinement. Anchor class is either foreground (FG) or background (BG). FG implies that the region contains an object of a class. Background implies no object in the anchor. Bounding box refinement is also called a positive anchor which implies that the anchor contains an object. The RPN predicts which anchor is most likely to contain an object and uses NMS to filter out other anchors that has lower foreground score.

2.5.6.3 Region of Interest Classifier & Bounding Box Regressor Region of interest (ROI) runs based on the input of RPN. It will output similar as the RPN, but the difference is that the ROI network is deeper and can classify regions and connect it to specific classes given by the user. (Car, boat, Person,...etc.) This output is called Class.

The other bounding box refinement is further work to refine the location of the bbox to predict an object. Following comes ROI pooling.

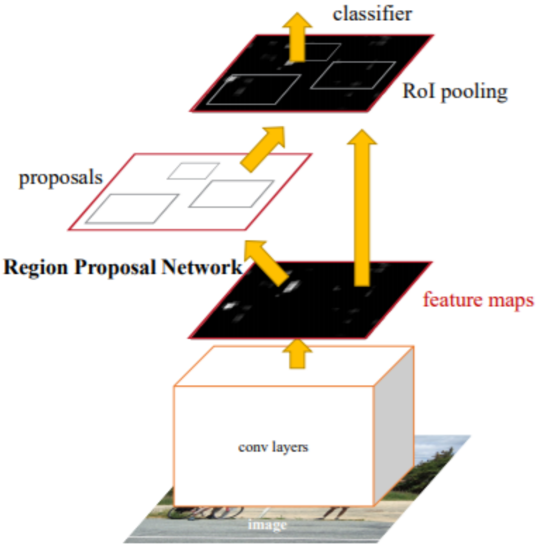


Figure 27: Pipeline illustrating the connections of Faster R-CNN. The same implementation used in Mask R-CNN. Image from [22].

2.5.6.4 Segmentation Masks At this stage, object detection has been conducted from previous stages. From now, masks in instance segmentation prediction is being predicted. The segmentation mask is a Convolutional network that uses the positive regions of the ROI classifier. The masks are at default 28x28 pixels in float representations, so it contains more information than other formats such as binary or integers.

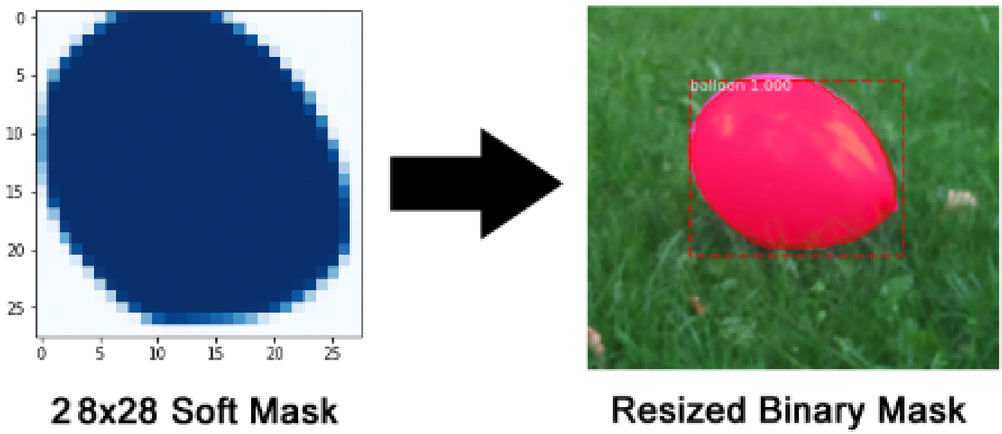


Figure 28: Illustration from [21].

3 Methodology

The goal is to make crane lifts offshore autonomously. As mentioned in the Introduction (1), this report focus is on the robot vision aspect of the system. It is deemed necessary to be able to track the object of interest.

Figure 29: It is assumed that the crane operation may look something like in these illustrations. Illustrations are taken from [23] and [24] respectively.

It is assumed in this report that a lifting operation may look something similar to what is shown in the videos: Crane Operation 1[1] and Crane Operation 2[2].

In robotic applications, 6D pose estimation is utilized to pick up objects, including rotation and translation. The rotation can be described with yaw, pitch, roll angles and translation with surge, sway and heave as shown in Figure (30).

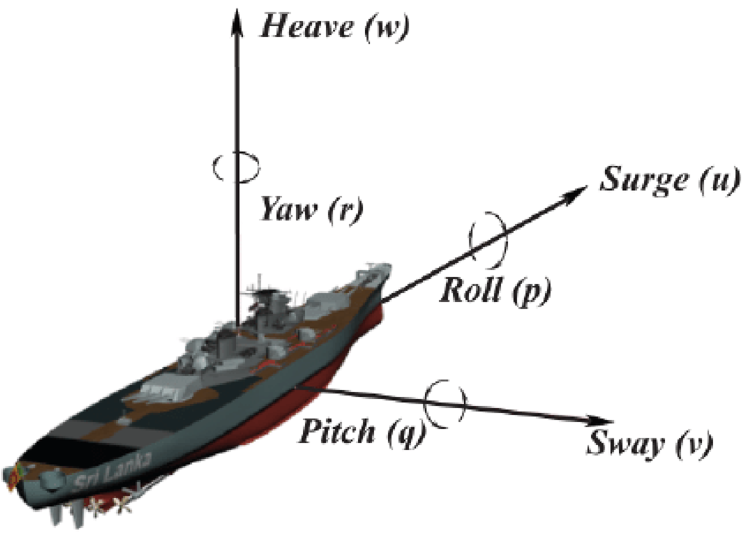


Figure 30: A change in one or more of these dimensions over time can create inaccurate pose estimations due to time delay. Illustration is taken from [25]

[1] https://www.youtube.com/watch?v=abFD-8BGx1I
[2] https://www.youtube.com/watch?v=lNaPZFwcoZY&ab_channel=RunGun

In this project, it is assumed that for a control system to pick up cargo with a hydraulic crane autonomously, it needs a 6D pose estimation of the object. This is so that a hydraulic crane or robotic arm can pick up the object due to its known geometric data. The term hydraulic gripper may now be referred to as a robotic arm throughout this report.

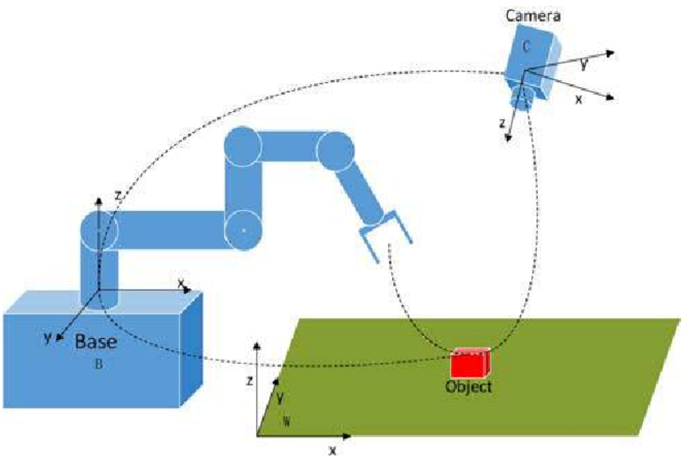


Figure 31: The robotic gripper needs 3D data of object and the surface it is supposed to land on. The illustration is taken from [26]

If the robotic gripper has frequent 6D pose estimation readings of the object, it can pick up the object. If the robotic gripper has 3D data of the surface it is loading its cargo onto, let us image a point cloud, then the robotic gripper can land the object based on 3D data in Euclidean space. If the crane were to load cargo onto the oil platform, a predefined 3D point cloud could be generated so that the robotic gripper knows where it can land the cargo.

Given that information is acquired about the pose of cargo and oil platform, it can pick up the cargo from the ship and land onto the oil platform. This solves for de-loading the ship. What about loading onto a ship? A critical difference between the ships' surface and the surface on the oil platform is that the ship is continuously moving due to wave motion. Each ship will dock differently, so a predefined point cloud of the ship is not feasible. Instead, the computer vision system requires 6D pose estimation tracking of the ship. The method for tracking objects with frequent 6D pose estimations is assumed to work for the ship's surface and cargo.

3.1 Selecting Sensor

Different types of sensors were studied for the use of calculating pose of ships, cargo or both.

A requirement is that it is accurate at ranges of roughly 3-15 meters, and it should be high frequency. The range is a rough estimate of the expected distance from the camera to an object.

Can there be a sensor that can be generalized to track ship deck, oil platform and cargo? It was researched to use 3D scanning with calibrated stereo vision, 3D point cloud generation with a camera and structured light, digital 2D camera, or laser technology. A breakdown of the methods follows.

3.1.1 Laser Technology

A system consisting of 1D laser distance measurement units could be used to calculate the distance with time of flight. Each laser could provide information about a point in Euclidean space. If the number of lasers is three or more, 3 points can be used to calculate the normal vector and translation. More lasers can be used to make the point cloud more dense and robust against noisy points.

According to the datasheet of class 1 laser LIDAR-Lite v3HP [55], the operating range is up to 40 meters and has an update frequency of 1 kHz.

Using this laser technology can provide the possibility to calculate the normal vector of a plane, along with its translation. In a scenario of using 3 laser point to calculate the normal vector of the plane and an origin, it may similar to what is shown in Figure (32):

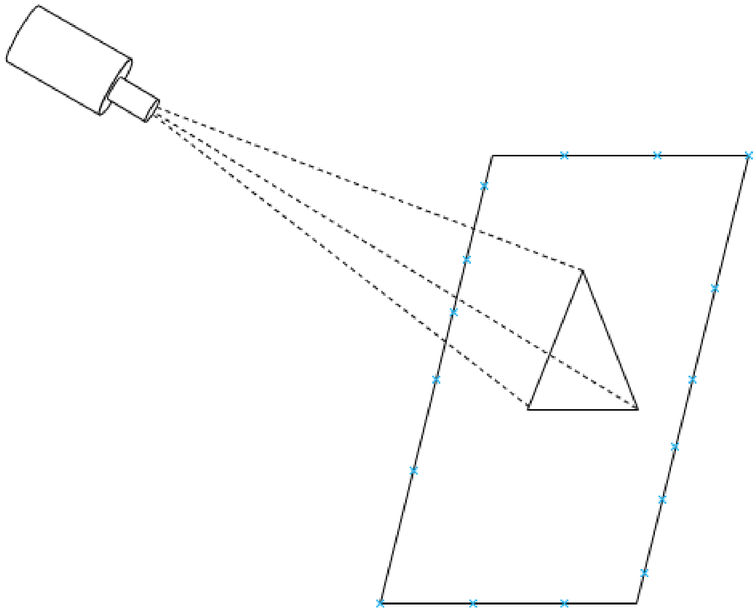


Figure 32: 3 laser scanners could calculate the normal vector of a plane

A significant problem here is that it will have 3 DOFs free and only pitch, roll and heave constrained. An example is that if the ship rotates only about the yaw angle, laser technology may not observe the change due to the nature of laser technology. The same applies to sway and surge along a plane.

It would need additional sensor methods in order to observe these changes. If the system does not have these DOFs established, it may be difficult for the robot arm to know where to place the load as yaw, surge, and sway can not be established.

During the process of loading on and off cargo, the sensor system needs to measure different areas as the ship, cargo and area of interest change over time. A system able to detect and steer the lasers to the area of interest is needed. It may require motor control to steer the laser sensor and an object detection system to point towards the area of interest.

Another complication is that the sensor system only sees the laser points, so if there are obstacles between the euclidean points generated by the lasers, it cannot observe this. A laser pattern that is sufficiently dense and a large enough FOV is needed in order to clear all space in landing procedures. A circumstance where this is illustrated can be seen in Figure (33).

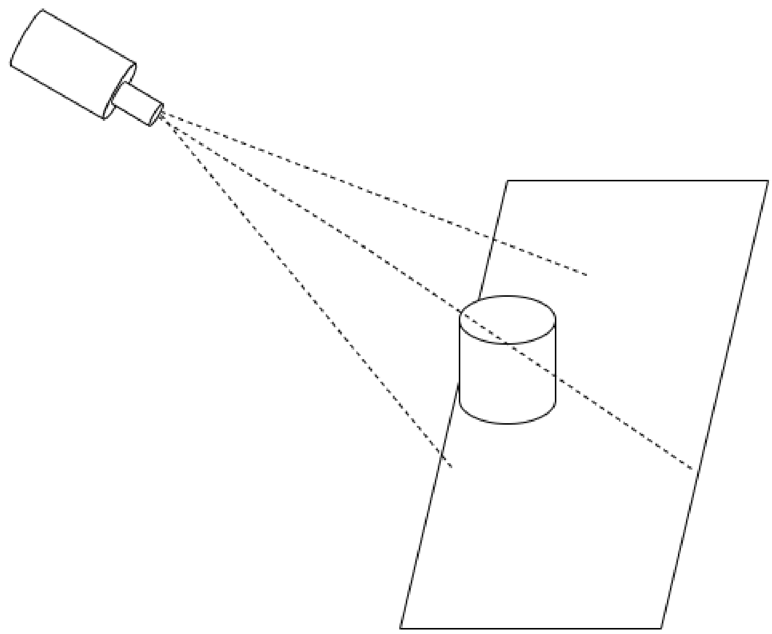


Figure 33: Low density laser system may not detect obstacles like the cylinder on the plane. More points may help solving it, but is more expensive and requires more precision

3.1.2 3D Camera

3D camera entails studying sensors that produce RGB-D data like Stereo Vision, 2D camera with structured light.

Generating a 3D point cloud with calibrated stereo vision or camera with structured light would provide the opportunity to use RGB-D data, where D is for depth. It could give measurements of 3D coordinates of both the ship and cargo. The requirement for the sensor system should be precise at an operating range of 3-15 meters.

The Intel 415's datasheet states the hardware has $\leq 2\%$ precision with a range of up to 2 meters at Table 4-9, page 66 in datasheet [56]. Its measurement frequency is z.

Consider a high-end 3D camera. Zivid has a documented precision of roughly 5000 μm at a range of 5 meters with an error of precision exponentially increasing by distance. Its measurement frequency is z with this hardware.

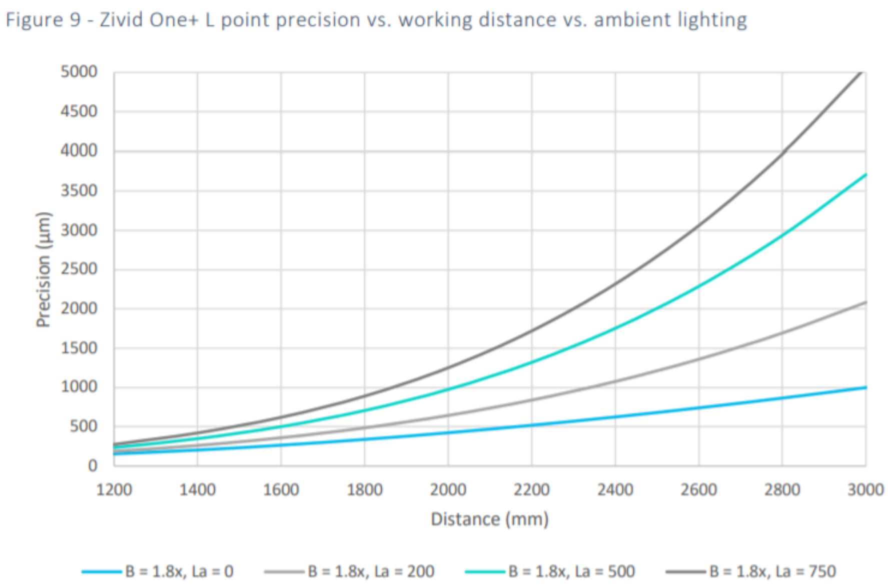


Figure 34: Zivid Large 3D camera's datasheet [27] describing accuracy over distance

It seems that the 3D cameras may be unsuited to this task primarily due to operating range. It is suspected that the bandwidth for a continuous live feed of 3D data may be a limiting factor unless an industrial GPU is used and the processing time might be too slow. The 3D camera may also face challenges in outdoor environments with shiny surfaces in various degrees, depending on several factors: lighting, surface material, weather conditions, and more. It is assumed that these conditions will lead to a decrease in the precision of the sensor.

A downside of this sensor, not unlike other options, is that it still requires a method to distinguish between what cargo is and what the ship is in terms of tracking.

Suppose a 3D point cloud could be established accurately at a range of roughly 3-15 meters, and the data per measurement would be a lot less (for example, a low-density cloud point). In that case, one still needs to track and distinguish the cargo for deloading operations and track the ship when loading onto the ship. So, an algorithm to track the object of interest is still needed.

3.1.3 Digital 2D Camera

The 2D digital camera provides information described in the pinhole camera model. The data will output data as an image plane in 3 layers, each describing the color intensity of RGB colors. It will not have sensor capabilities of measuring 3D data, but methods such as PnP can be utilized in conjunction with RGB input from a 2D camera to calculate 3D data. This type of sensor depends on feature extraction to calculate 3D data, which can be done numerous ways.

The Spatial resolution must also be high enough for its purpose. This project aims at distances of up to 15 meters. A standard camera is widescreen and is commonly comes with an aspect ratio of 16:9, 16:10, 4:3. An important notation is that the coarsest spatial resolution in these common aspect ratios is the vertical axis. This axis is assumed to be the least precise axis in terms of spatial resolution.

A critical notation is that one can decide to change a camera with the pixel resolution of 640 x 480 (4:3 ratio) to, for example, 3840 x 2160 (16:9) will significantly increase the computational cost. For the first camera, 640 x 480 corresponds to 0.3 Megapixels, and the latter is 8.3 Megapixels. The matrix size has become 27.67 larger, but the spatial resolution is 2160/480 = 4.5 times larger on the vertical axis. The bottleneck may become hardware that cannot process that much information due to RAM shortage or computational power.

3.2 Solution

The problem requires a fast system that can target the object of interest and track it, preferably by constraining all 6DOFs, pitch, yaw, roll, surge, sway, and heave at an operating range from 3-15 meters. This report has been focusing on a low computational cost system that can calculate 6DOFs frequently. Based on the requirements of operation range and speed, using PnP with a relatively low number of points, low image resolution, and a lightweight feature extractor is deemed a feasible solution to track objects. Using a lightweight deep learning algorithm in computational cost can be used for feature extraction in conjunction with a PnP solver with few matching points to solve 6D pose estimation for cargo. Using deep learning for feature extraction can also help to increase the robustness of the feature extraction problem in various settings, such as time of day and weather conditions based on its training dataset. Varying the dataset to generalize localization and objectification in different settings is assumed to increase the system's robustness. Using a PnP solver for calculation of pose estimation can be lightweight in terms of the designer can choose $n \geq 6$ to solve it. Increasing the number n will increase its robustness at the expense of computational cost, but having relatively few points decreases the time delay. The system's precision is assumed to depend significantly on the precision of the feature extractor of the deep learning model for image point localization, pixel resolution in the camera and the time delay.

The following Section (4) will introduce a method to track the container. The experiment will use deep learning to predict a planar surface on the standardized container. It is intended that the camera shall be placed somewhere on the crane or platform, similar to what is illustrated in Figure (31), above the container to track so it will have clear visibility of the ceiling of the container. This makes the environment more controlled as the perspective will be relatively similar from each operation. Using deep learning to identify the rectangle that is the container ceiling will solve the problem of targeting the object of interest and track it. A keypoint detection algorithm is made off a more classic computer vision corner detector after the AI object feature extraction has been applied to increase the accuracy of the image points matching with the 3D object points model. The pipeline(35) can be seen in the next section (4) .

4 Experiments

To order to achieve pose estimation $\boldsymbol{T}_{\boldsymbol{o}}^{\boldsymbol{c}}$ between the camera frame and object frame with a PnP solution, 2 parameters are required, as explained in Section (2.3).

- The normalized image coordinates $\tilde{\boldsymbol{s}}_{\boldsymbol{i}}$
- 3D object points $\tilde{\boldsymbol{r}}_{\boldsymbol{o},\boldsymbol{i}}^{\boldsymbol{o}}$

The normalized image coordinates $\tilde{\boldsymbol{s}}_{\boldsymbol{i}}$ are obtained by using with the use of the intrinsic camera parameters and pixel coordinates by using the relationship $\tilde{\boldsymbol{p}} = \boldsymbol{K}\tilde{\boldsymbol{s}}$ that is explained in Equation (2.1). So, now it is required to obtain 2D image point and the intrinsic camera parameters $\boldsymbol{K}$ + distortion coefficients. In the following sections, it will be explained how all parameters was obtained.

The Flowchart (35) illustrates the pipeline of the system and the Figures (36) and (37) illustrates the image output of all steps of the system.

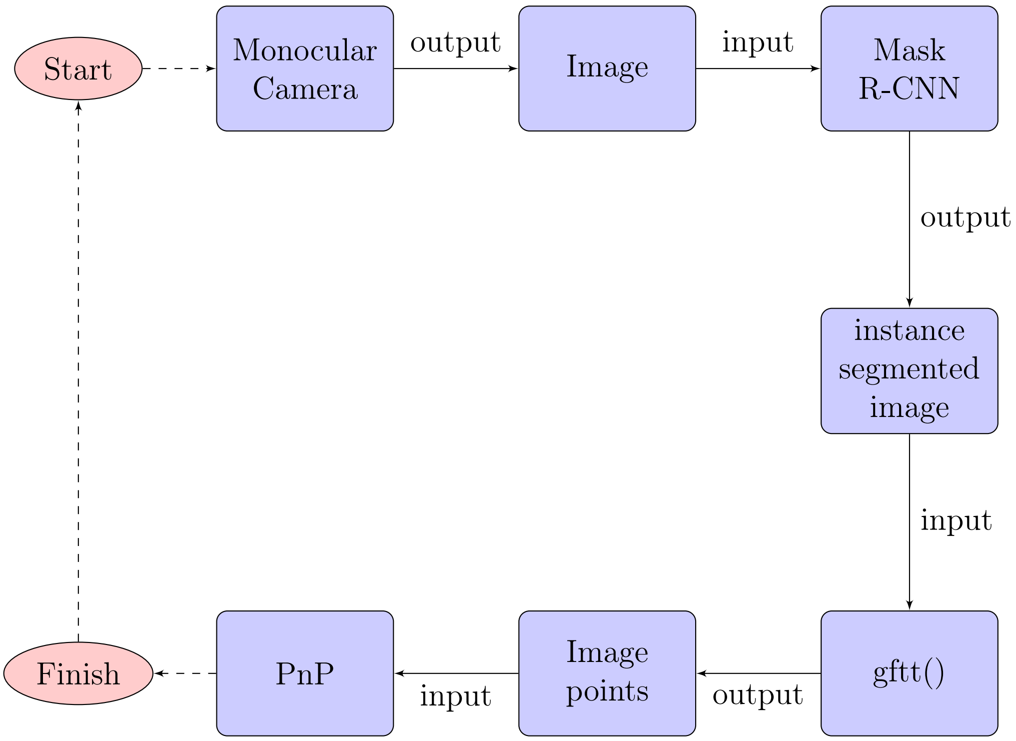
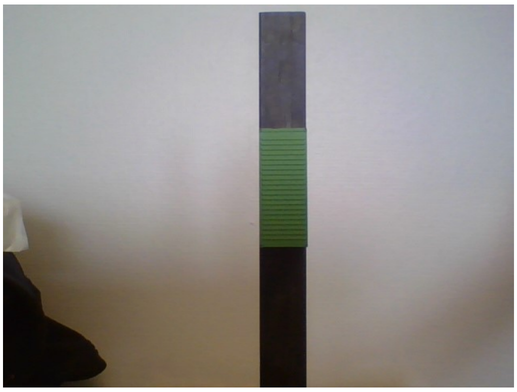


Figure 35: Pipeline of the system



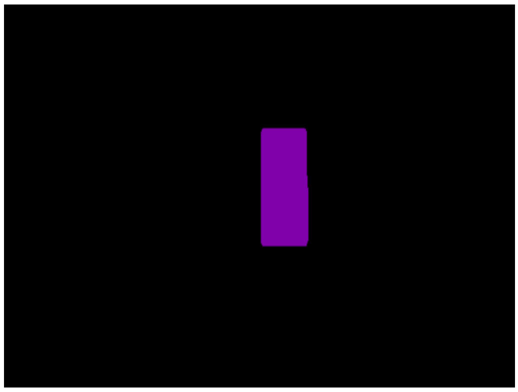
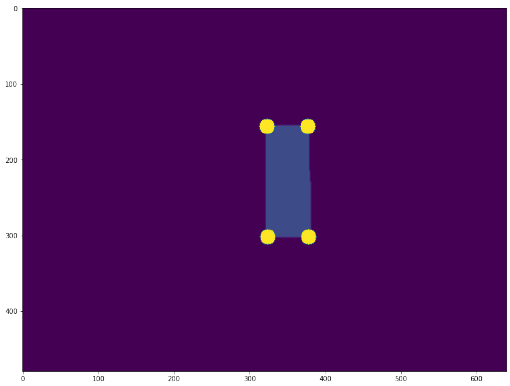


Figure 36: Original image with 640x480 (left) used as input to Mask R-CNN +gftt(). Image after Mask R-CNN with custom post processing filter (right).



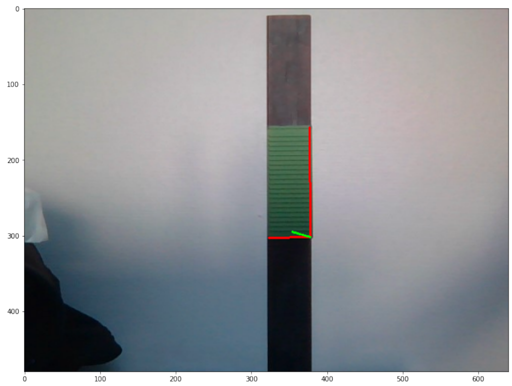


Figure 37: Left image is after gftt() is used to find corners. Right image is of Orthogonal axis drawn onto object based on data from calculated rotation matrix.

4.1 2D image point Extraction

The normalized image coordinates $\tilde{\boldsymbol{s}}_{\boldsymbol{i}}$ is used in order to solve the extrinsic camera parameters between the camera frame and the object frame is the 2D image points projected onto the image plane that corresponds to the 3D object points.

The 3D generated model was based on 4 key points in each corner of the ceiling. To solve the transformation, normalized image coordinates $\tilde{\boldsymbol{s}}_{\boldsymbol{i}}$ correspondences must be found and matched as shown in Figure (54). An AI approach in conjunction with a more classical CV corner detector was used. The machine learning algorithm is the well-known Mask RCNN, and its function is feature extraction. Following is the corner detector goodFeaturesToTrack() or gftt() for short. It was compared to other algorithms such as Harris Corner Detector, but gftt() outperformed it in precision in almost every experimental trial in this report. After classifying and localizing the object of interest in the image with Mask RCNN to alter the image, the altered image was used as input for the corner detector. From there, the corner detector from OpenCV goodFeaturesToTrack() is used. If 4 corners are found, an array containing these image points in a random order will be passed as arguments into pixelSorting(), which is an algorithm created to sort the image points so it will correspond to the order of objpoints() array that contains all 3D world coordinates in the object frame, explained in Section (4.3). After all the parameters are sorted and found, it is used in the P4P algorithm. The P4P algorithm returns rotation matrix and translation vector with respect to camera frame relative to object frame.

The plane may have multiple solutions, depending on the pixel sorting. In order to correct this, the function correctsRmatrix() adjusts for this, explained in Section (4.1.4). First, it asserts that the rotation matrix is a valid rotation matrix by $\boldsymbol{R^T} \times \boldsymbol{R} = \boldsymbol{I}$ with a precision of 1E-6. Then, the algorithm checks the diagonals positives and negatives and rotates the matrix into a positive orientation.

4.1.1 Detectron2

"Detectron2 is Facebook AI Research's next generation software system that implements state-of-the-art object detection algorithms. It is a ground-up rewrite of the previous version, Detectron, and it originates from maskrcnn-benchmark."[57]

In Detectron2, one has the opportunity to implement different types of detection algorithms and compose it as one sees fit. One can see in the Appendix (A) on how Detectron2 was implemented in this report. It is suggested to follow the installation manual from Detectron2's Github [57], but the installation guide might not always work for everyone since some of the middleware is hardware dependant and one may not have the same hardware as the authors.

4.1.2 Mask R-CNN

Assuming Detectron2 was successfully installed, a training script is created in order to start training and testing AI models.

In this instance, Google Colab (an overlay of Jupyter notebook) is primarily utilized due to its interfaces with Tensorboard, a toolkit for surveying different plots of metrics of the trained model and it's used for fast testing of scripts.

It is programmed in python language with .ipynb file format. Following is a series of the code utilized to train the instance segmentation model with Mask R-CNN. The initial setup is heavily inspired by the work of Detectron2's "getting started" [57] and the work of gilbert Tanner [58].

Modelzoo is a script in detectron2 that makes it easier to load in initial weights from state-of-the-art models for object detection, instance segmentation, panoptic segmentation and more.

```python
import torch, torchvision
import detectron2
from detectron2.utils.logger import setup_logger
setup_logger()

# import some common libraries
import numpy as np
import cv2
import matplotlib.pyplot as plt
import os
import json
import random
from matplotlib import pyplot as plt

# import some common detectron2 utilities
from detectron2 import model_zoo
from detectron2.engine import DefaultPredictor
from detectron2.config import get_cfg
from detectron2.utils.visualizer import Visualizer #For drawing
    ↪ prediction onto images
from detectron2.data import MetadataCatalog, DatasetCatalog
from detectron2.structures import BoxMode
from detectron2.engine import DefaultTrainer
from detectron2.utils.visualizer import ColorMode
from detectron2.utils.visualizer import GenericMask
from google.colab.patches import cv2_imshow # replaced from
    ↪ cv2.imshow() when using google colab
#import detectron2.utils.visualizer #suppressed but untouched. It was
    ↪ to check whether the dictionary was loaded properly. After training
    ↪ it has been replaced by another custom visualizer class, but not
    ↪ overwritten.
```

The train/valid/test data was annotated using the annotation software labelme, then it was structured inside a folder like this:

dir:train
file: *.jpg
file: *.json

dir:valid
file: *.jpg
file: *.json

dir:test
file: *.jpg

From there, a python script Labelme2coco[3] was used to convert the data structure of the .json files into the coco format. This is done because one can utilize function associated to coco, including data registration and evaluation. The function reg-

[3] https://pypi.org/project/labelme2coco/

ister_coco_instances(name, metadata, json_file, image_root)): registers data for training as shown below.

```python
from detectron2.data.datasets import register_coco_instances
register_coco_instances("containerCeiling_train", {}, "/content/testrig1v2Annetvalid/train/train.json", "/content/testrig1v2Annetvalid/train/")
register_coco_instances("containerCeiling_valid", {}, "/content/testrig1v2Annetvalid/valid/valid.json", "/content/testrig1v2Annetvalid/valid")
register_coco_instances("containerCeiling_test", {}, "/content/testrig1v2Annetvalid/valid/valid.json", "/content/testrig1v2Annetvalid/test")

containerCeiling_metadata = MetadataCatalog.get("containerCeiling_train")
dataset_dicts = DatasetCatalog.get("containerCeiling_train")
```

After the metadata has been created in a dictionary, a new cell in .ipynb will test if the data has been loaded correctly. This is done by using the Visualizer class to print out the image with its corresponding annotation. It takes 3 randomly sampled images and prints the output with annotations. It is a verification step to see if the data was properly loaded. It is not necessary to train the model itself.

```python
dataset_dicts = get_containerCeiling_dicts("containerCeilingV3/train")
for d in random.sample(dataset_dicts, 3):
    img = cv2.imread(d["file_name"])
    v = Visualizer(img[:, :, ::-1], metadata=containerCeiling_metadata, scale=0.5)
    v = v.draw_dataset_dict(d)
    plt.figure(figsize = (14, 10))
    plt.imshow(cv2.cvtColor(v.get_image()[:, :, ::-1], cv2.COLOR_BGR2RGB))
    plt.show()
```

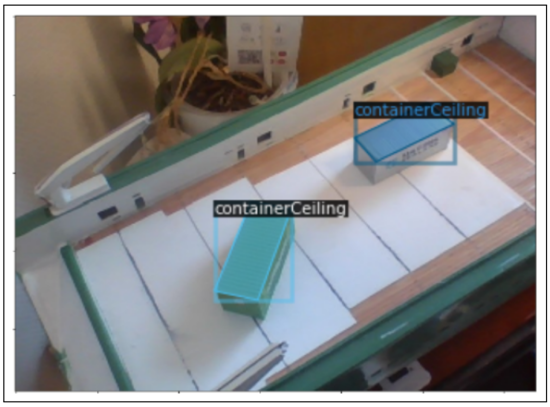


Figure 38: example of output of cell above. This image is loaded correctly

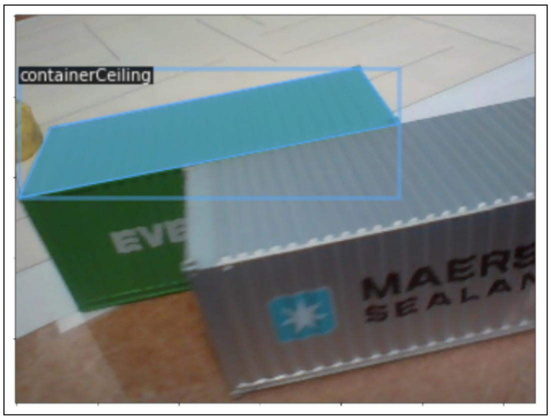


Figure 39: Only objects with full visibility of all corners should be accepted.

When the data has been verified to been loaded into the dictionary properly, then the configuration class get_cfg() shall set the settings for this training. The get_cfg() has its default settings, and it's up to the user to overwrite these configurations with its own parameters.

Used mask rcnn R 50 FPN as the configuration file. It contains information about which configurations shall be used in training, including hyperparameters and more.

Transfer learning was used. The weights from COCO-InstanceSegmentation/ mask_rcnn_R_50_FPN_3x.yaml was utilized here. The first two layers then frozen, so the weights will not adjust during training. One class was registered, which is containerCeiling. The model will train to learn to predict this class. After trial and error, 1500 training iterations seem fine with the given dataset based on the metrics and testing.

```
cfg = get_cfg() #create an object from class get_cfg()
cfg.merge_from_file(model_zoo.get_config_file(
"COCO-InstanceSegmentation/mask_rcnn_R_50_FPN_3x.yaml"))
cfg.DATASETS.TRAIN = ("containerCeilingV3_train",)
cfg.DATASETS.TEST = ()
cfg.DATALOADER.NUM_WORKERS = 2
cfg.MODEL.WEIGHTS = model_zoo.get_checkpoint_url(
    "COCO-InstanceSegmentation/mask_rcnn_R_50_FPN_3x.yaml")
cfg.SOLVER.IMS_PER_BATCH = 2
cfg.SOLVER.BASE_LR = 0.00025
cfg.SOLVER.MAX_ITER = 1500
cfg.MODEL.ROI_HEADS.NUM_CLASSES = 1

os.makedirs(cfg.OUTPUT_DIR, exist_ok=True)
trainer = DefaultTrainer(cfg)
trainer.resume_or_load(resume=False)
trainer.train()
```

The full config object is listed in Appendix (D)

The model has been trained with the configurations mentioned above. From here, more parameters from the default configurations are being overwritten by new parameters that are used in inference. For example, the weights from training are used, and the confidence score must be 0.9 in order to show prediction. In order to focus on only one object at once, a restriction of one detection per image is enforced. The prediction with the highest confidence score is shown. The inference is to be tested on images from cfg.DATASETS.TEST. DefaultPredictor is chosen with the updated instance of cfg.

```
cfg.MODEL.WEIGHTS = os.path.join(cfg.OUTPUT_DIR, "model_final.pth")
cfg.MODEL.ROI_HEADS.SCORE_THRESH_TEST = 0.9
cfg.TEST.DETECTIONS_PER_IMAGE = 1 #maximum number of predictions in
↪  each image
cfg.DATASETS.TEST = ("containerCeiling_test", )
predictor = DefaultPredictor(cfg)
```

Up to now, the class Visualizer that was imported in the code cell was used for asserting that the data was loaded correctly in the dictionary "...". From now on, the Visualizer class has been modified, and the full customized Visualizer class can be found in Appendix (B). The changes that have been made are all pixel values that are not a part of the prediction are set to a pixel value in RGB scale (0,0,0). The instance of the prediction is assigned a random color that is not black (0,0,0). Also, the opacity of the prediction has been set to 100%.

The following cell reads an image, uses the predictor from DefaultPredictor(cfg) with the object cfg.

```
dataset_dicts = get_containerCeiling_dicts('containerCeilingV3/test')

im = cv2.imread( "containerCeilingV3/test/640x480_attempt1_300mm.jpg")
im2 = im # copy original

outputs = predictor(im)
v = Visualizer(im[:, :, ::-1],
                metadata=containerCeiling_metadata,
                scale=1,
                instance_mode=ColorMode.IMAGE_BW
                )
v = v.draw_instance_predictions(outputs["instances"].to("cpu"))
plt.figure(figsize = (14, 10))
plt.imshow(cv2.cvtColor(v.get_image()[:, :, ::-1], cv2.COLOR_BGR2RGB))
plt.show()
```

Now it should end up having a prediction with a custom filter looking like what is shown in Figure (41):

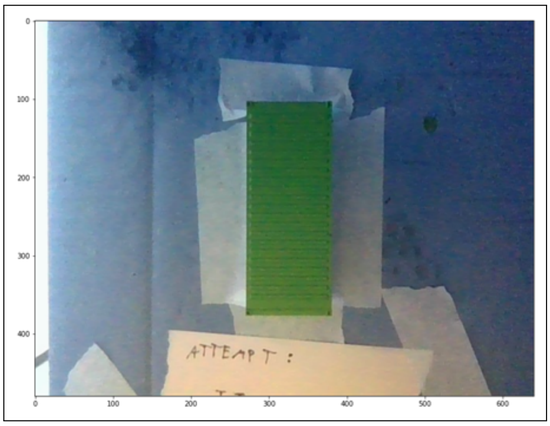


Figure 40: Original image

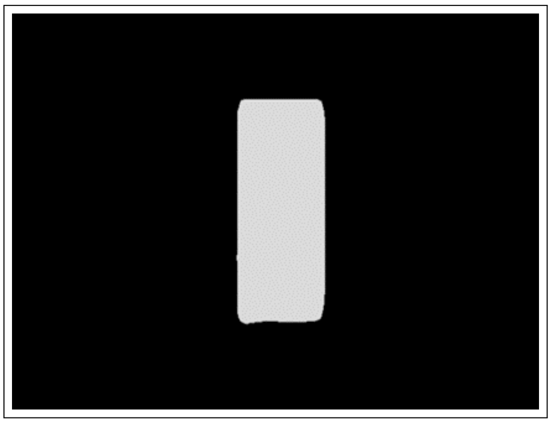


Figure 41: Image after instance prediction and custom post processing

4.1.3 goodFeaturesToTrack()

This new post-processed image is used as an input to a more classical CV approach to finding key points, which in this circumstance is the four corners in the planar surface. A corner detector from OpenCV named goodFeaturesToTrack() or gftt() for short was used. The upper bound for allowable corners detected in an image was set to 4, and a minimum distance between two corner detections is set to 30 pixels. The images that are passed into gftt() need to be in a grayscale format, so a conversion is used. After a maximum of four corners is detected, the image points are stored as integers and also printed and drawn onto the image. This way, the test results can be analyzed more closely.

In 1994, J. Shi and C. Tomasi modified the Harris Corner Detector named Good Features To Track. According to the OpenCV documentation [28], the Harris Corner Detector has a scoring function:

$$R = \lambda_1 \lambda_2 - k(\lambda_1 + \lambda_2)^2 \tag{41}$$

Shi-Tomasi's alteration:

$$R = min(\lambda_1, \lambda_2) \tag{42}$$

This leads to that both λ_1 and λ_2 must surpass a certain threshold in order for the algorithm to acknowledge it as a corner. The algorithm from open CV passes 4 parameters:

$$R = \lambda_1 \lambda_2 - k(\lambda_1 + \lambda_2)^2 \tag{43}$$

Shi-Tomasi's alteration:

$$R = min(\lambda_1, \lambda_2) \tag{44}$$

This leads to that both λ_1 and λ_2 must surpass a certain threshold in order for the algorithm to acknowledge it as a corner. The algorithm from open CV passes 4 parameters:

- Grayscale image (matrix)
- number of detectable corners (integer)
- Quality level of corner detected (value between 0-1)
- minimum distance between each pixel

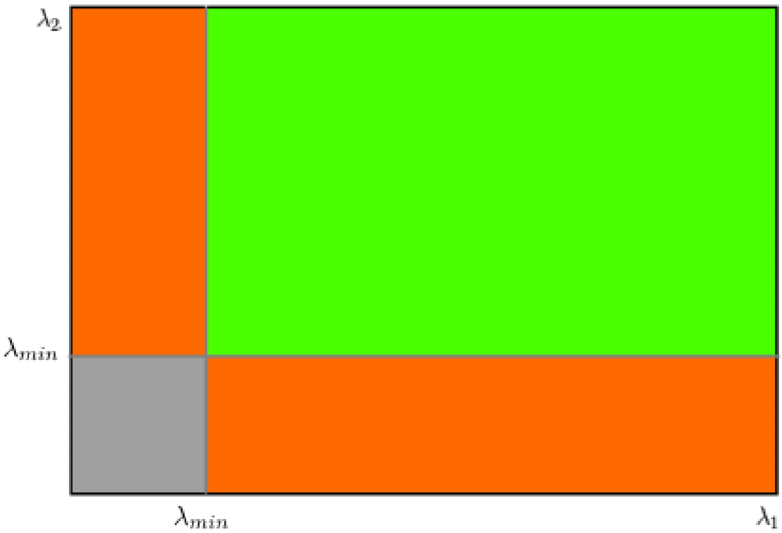


Figure 42: When λ_1 and λ_2 is greater that λ_{min}, then it is considered a corner. The green rectangle represents a detected corner under a given threshold. Figure from [28]

```
pred_im = cv2.cvtColor(v.get_image()[:, :, ::-1], cv2.COLOR_BGR2RGB)
gray = cv2.cvtColor(pred_im,cv2.COLOR_BGR2GRAY)

#cv2.goodFeaturesToTrack(matrix, FEATURE_DETECT_MAX_CORNERS,
↪   FEATURE_DETECT_QUALITY_LEVEL, FEATURE_DETECT_MIN_DISTANCE)
corners = cv2.goodFeaturesToTrack(gray ,4 ,0.3 ,30,  )
corners = np.array(corners, dtype= int) #convert into integers for
↪   image plane
goodCorners = corners

#Draw circles around the detected corners.
for i in corners:
    x,y = i.ravel()
    cv2.circle(gray,(x,y),10,255,-1)

plt.figure(figsize = (14, 10))
plt.imshow(gray)#,plt.show()
cv2_imshow(im)
```

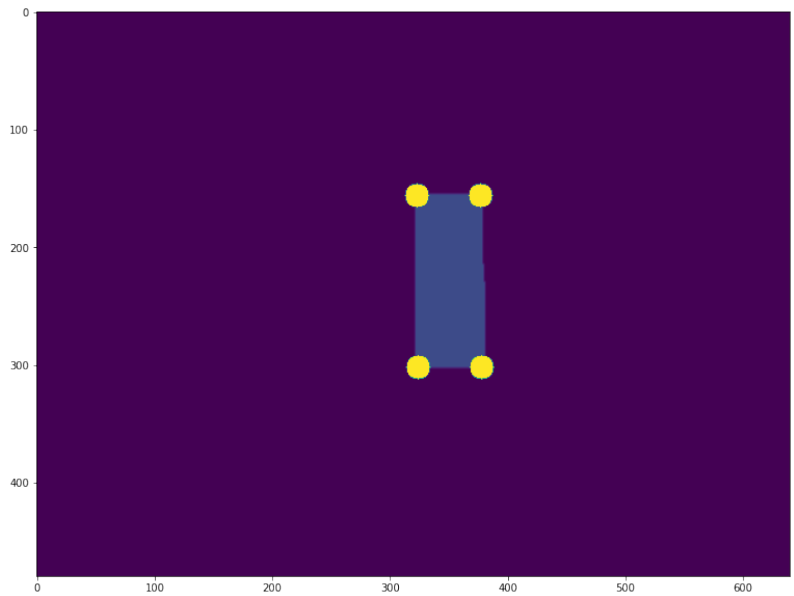


Figure 43: Post gftt(), the output image is expected to look like this

4.1.4 pixelSorting()

Now there is an array list of pixel coordinates from gftt(). An essential part is that the order of the list of image points and 3D obj points corresponds. The reality is that the 3D obj points are constant since it is created in an array list, described in Section (4.3), but the order of 2D image points are random in the gftt()-algorithm. Therefore, an algorithm to sort these image points in the correct order is conducted. The algorithm named pixelSorting() accepts an array list consisting of 4 image points created by gftt(). The intention of the pixelSorting() algorithm is to sort the image points in a clockwise manner in the image plane.

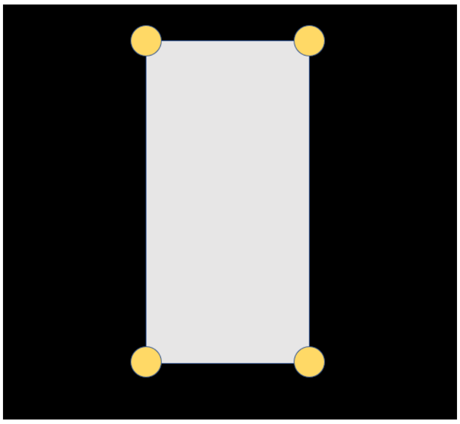


Figure 44: Example of 4 image points that are stored in variable pixelarray

An arbitrary point may be selected, which in this instance, the reference point (RP) is the first element in the pixel array. For the example here, let's state that pt1 is top left in the Figure (44), pt2 is top right, pt3 is at the bottom right, and pt4 is bottom left. The task now is to sort a randomly sorted array of pixel points into the order $numpy.array([pt1, pt2, pt3, pt4])$.

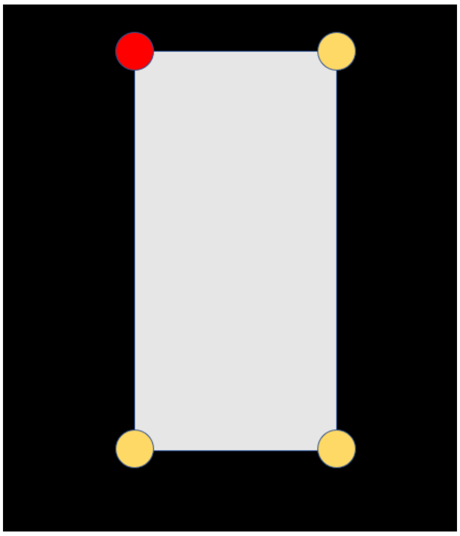


Figure 45: Red circle is illustrated as the arbitrary reference point

For this example, let us state the first image point in the pixel array is pt1. This makes the $RP = pt1$ in Figure (45). The algorithm calculates vec12, vec13, vec23 and finds the shortest vector from RP. It is assumed that the point closest to the reference point in the image plane is the same point that is closest in the Euclidean space that will be tested in this experiment. The point found is stored as pt2 as the second element in the pixel array.

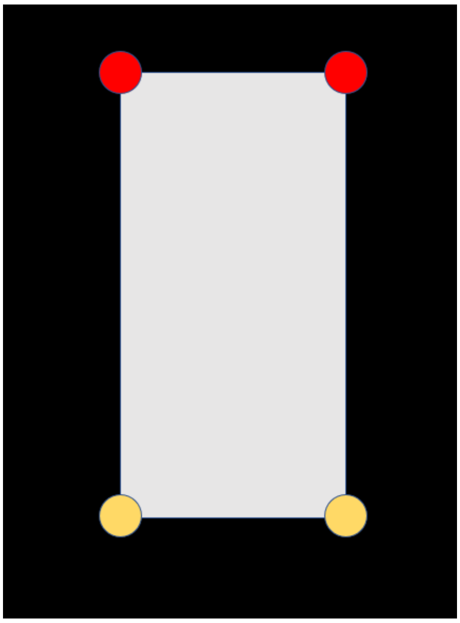


Figure 46: The point with the shortest vector RP is assumed to be alongside the short edge of RP. It means relative position with respect to RP has been established for pt2

The next step is to identify the third image point, which would be the diagonal of RP. Initially, when studying the figures above (44)(45), one could identify the pt3 by calculating the vector furthest away from RP and identify that point as the diagonal point in Euclidean space. However, due to affine transformations in the image plane, there are circumstances where this won't necessarily work.

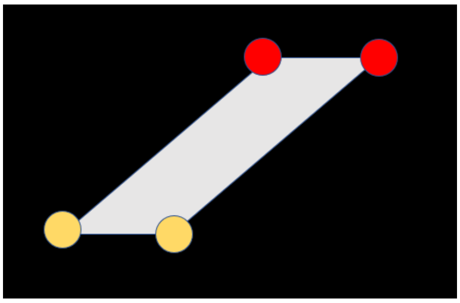


Figure 47: An instance where the point diagonal to RP is not furthest away

The circumstance here is that vec14 is the longest. As a result of this, a different approach was needed to counter this. The next step of the pixelSorting algorithm uses the established points pt1 and pt2 to find the length of vec13, vec14, vec23, vec24 is calculated. It is assumed here that the longest vector of all these vectors is a part of the diagonal. This method will identify a point diagonally to another.

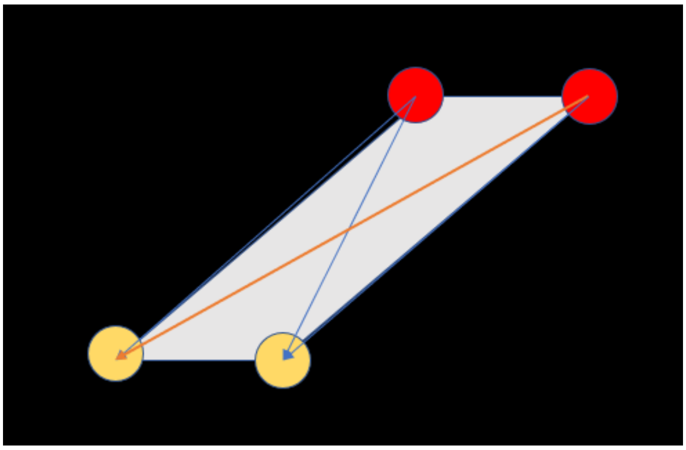


Figure 48: Calculates 4 vectors. The longest vector in this instance is the orange line going from pt2 to pt4

This method compensates for affine transformations and most projective transformations.

Now, two points have been located and sorted. Since the longest vector is between two points being diagonal to each other, a relative position has been acquired. I.e., in Figure (48), the longest vector is vec24. Given that pt2 and pt4 are located diagonally to each other, it means that pt1 and pt3 are diagonal to each other. Another instance is shown in the figure (49), where pt1 and pt3 is the longest vector and therefore diagonally to each other.

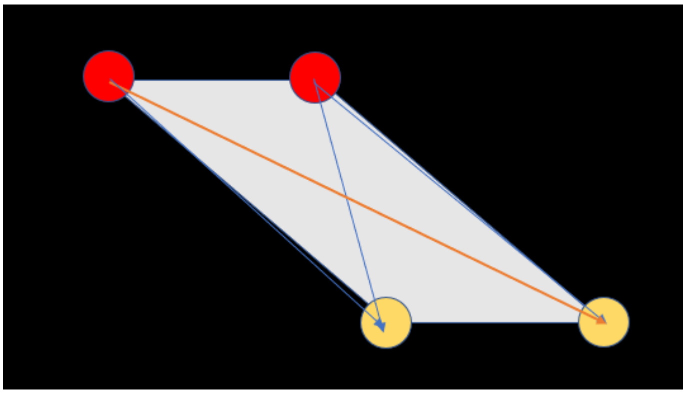


Figure 49: The longest vector in this instance is the orange line going from pt1 to pt3

Now, relative positioning between pt1, pt2, pt3 and pt4 has been established. The pixel array is being sorted into pt1, pt2, pt3, pt4, respectively. The code associated to pixelSorting()-algorithm follows

```
#Initializing with an arbitrary image point, gftt[0]. Finding image
↪  point with shortest distance

def getLengthOfVector(vec):
  assert len(vec) == 2, "The vector needs to be in length of 2. Ex:
  ↪  [20,21]"
  return np.sqrt(vec[0]**2 + vec[1]**2)

def getPointDiagonallyInProjectedRectangle(pt1, pt2, pt3, pt4,
↪  pixelarray):
  #returns the point placed diagonally to reference point pt1.
  dist13 = getLengthOfVector(abs(pt3 - pt1))
  dist14 = getLengthOfVector(abs(pt4 - pt1))
  dist23 = getLengthOfVector(abs(pt3 - pt2))
  dist24 = getLengthOfVector(abs(pt4 - pt2))

  largestDiagonal = max(dist13, dist14, dist23, dist24)
  if dist13 == largestDiagonal:
    return pt3
  elif dist14 == largestDiagonal:
    return pt4
  elif dist23 == largestDiagonal:
    return pt4
  elif dist24 == largestDiagonal:
    return pt3
  else:
    print("Can't find largest diagonal. Lets return None")
    return None

def swap(pt3 , pt4):
  c = pt3
```

```
    pt3 = pt4
    pt4 = c
    return pt3, pt4

def pixelSorting(pixelarray):
    assert len(pixelarray) == 4, "The pixel array needs to have a length of 4 with this format-> Ex:  [[370  88], [254 100], [413 270], [225 286]] "

    vec12 = abs(pixelarray[1] - pixelarray[0])
    vec13 = abs(pixelarray[2] - pixelarray[0])
    vec14 = abs(pixelarray[3] - pixelarray[0])

    pt1 = pixelarray[0] #Reference point
    pt2 = 0
    pt3 = 0
    pt4 = 0

    if getLengthOfVector(vec14) > getLengthOfVector(vec12) < getLengthOfVector(vec13):
        print("vec12 is smaller than vec13 and vec14. This corresponds to imagepoint 2 in lst is closest to point 1")
        pt2 = pixelarray[1]

        #initially start of variables pt3 and pt4
        pt3 = pixelarray[2]
        pt4 = pixelarray[3]

        if getPointDiagonallyInProjectedRectangle(pt1, pt2, pt3, pt4, pixelarray)[0] == pt3[0] and getPointDiagonallyInProjectedRectangle(pt1, pt2, pt3, pt4, pixelarray)[1] == pt3[1]:
        elif getPointDiagonallyInProjectedRectangle(pt1, pt2, pt3, pt4, pixelarray)[0] == pt4[0] and getPointDiagonallyInProjectedRectangle(pt1, pt2, pt3, pt4, pixelarray)[1] == pt4[1]:
            pt3, pt4 = swap(pt3, pt4)

    elif getLengthOfVector(vec14) > getLengthOfVector(vec13) < getLengthOfVector(vec12):
        print("vec13 is smaller than vec12 and vec14. This corresponds to imagepoint 3 in lst is closest to point 1")
        pt2 = pixelarray[2]
        #initially setting these variables here
        pt3 = pixelarray[1]
        pt4 = pixelarray[3]

```

```
      if getPointDiagonallyInProjectedRectangle(pt1, pt2, pt3, pt4, pixelarray)[0] == pt3[0] and getPointDiagonallyInProjectedRectangle(pt1, pt2, pt3, pt4, pixelarray)[1] == pt3[1]:

      elif getPointDiagonallyInProjectedRectangle(pt1, pt2, pt3, pt4, pixelarray)[0] == pt4[0] and getPointDiagonallyInProjectedRectangle(pt1, pt2, pt3, pt4, pixelarray)[1] == pt4[1]:
        pt3, pt4 = swap(pt3, pt4)

    elif getLengthOfVector(vec13) > getLengthOfVector(vec14) < getLengthOfVector(vec12):
      print("vec14 is smaller than vec12 and vec13. This corresponds to imagepoint 4 in lst is closest to point 1")
      pt2 = pixelarray[3]
      #initially setting these variables here
      pt3 = pixelarray[1]
      pt4 = pixelarray[2]


      if getPointDiagonallyInProjectedRectangle(pt1, pt2, pt3, pt4, pixelarray)[0] == pt3[0] and getPointDiagonallyInProjectedRectangle(pt1, pt2, pt3, pt4, pixelarray)[1] == pt3[1]:
      elif getPointDiagonallyInProjectedRectangle(pt1, pt2, pt3, pt4, pixelarray)[0] == pt4[0] and getPointDiagonallyInProjectedRectangle(pt1, pt2, pt3, pt4, pixelarray)[1] == pt4[1]:
        pt3, pt4 = swap(pt3, pt4)


    pixelarray = np.array([pt1, pt2, pt3, pt4])

    return pixelarray
```

4.2 Camera Calibration

The intrinsic camera parameters, including its distortion coefficients, were established by using a camera calibration script, found in Appendix (E) and programmed by Tiziano Fiorenzani [59] from a template in OpenCV that is based on Zhang's method [38]. The camera shall be calibrated before using the other algorithms. The script itself can be found in Appendix (E)

The webcam of the laptop Lenovo IdeaPad L340 Gaming offers various resolutions, amongst 640x480 pixel resolution in width and height respectively for image- and video capture. In order to reproduce the results of the system in this report, it is advised to use the same camera for training images in the instance segmentation model, camera calibration and feature extraction for the PnP solver. The Instance segmentation model will be trained at predicting classes at images with a resolution of 640x480, which means it will be best suited for this resolution. Therefore, the intrinsic camera calibration and pixel point feature extractor will use the same resolution and camera.

A total of 50 images was captured from various distances and orientations. The transformation $\boldsymbol{T}_{co}^{c}$ between the checkerboard and camera were also varied in order to make different distortions more prominent in different images.

The checkerboard used was 9x6 rows and columns, respectively, with a 15 mm length of each square.

Each image was supervised and either accepted or discarded as input to the camera calibration script.

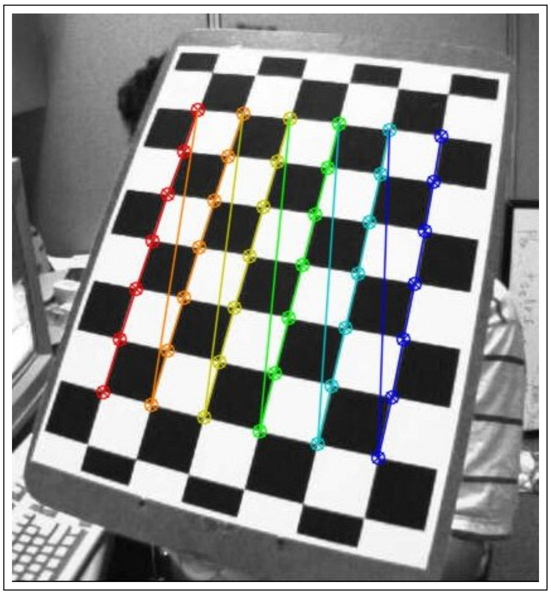


Figure 50: Acceptable image. Figure from [29]

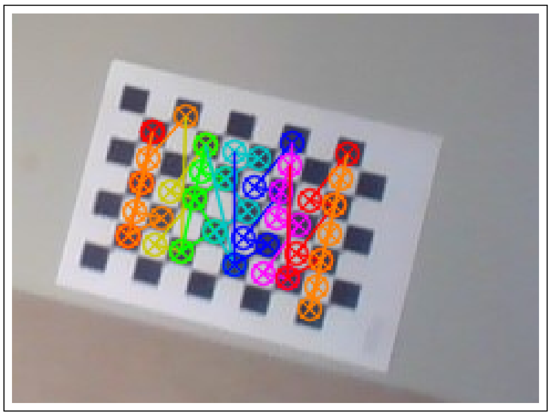


Figure 51: This image was discarded

After the images were controlled, the algorithm started calculating the camera intrinsic parameters. The parameters was be stored in .txt-files, named cameraMatrix.txt and cameraDistortion.txt in an output folder.

4.3 3D world Point Model

The 3D object points are the 3D generated model of the planar surface. Measurements of the containers were conducted with a digital caliper to find relative positioning in Euclidean space.

Figure 52: Length of container measured to be 69.50 mm

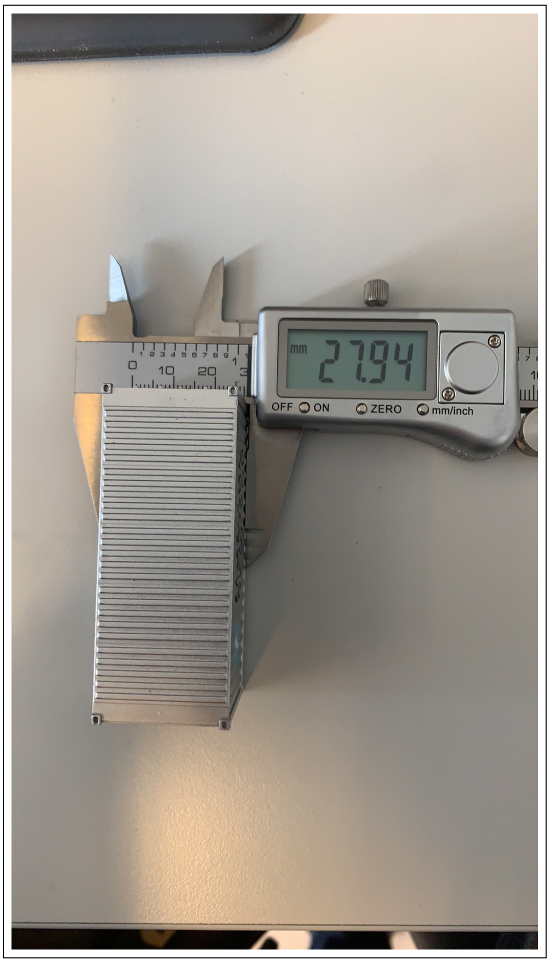


Figure 53: Width of container measured to be 27.94 mm

Furthermore, the object frame's origin is located in the center of the plane. The coordinates of the edges are described from this origin in Euclidean space.

Figure 54: an array is created from these points, with an order starting from top left and moving horizontally towards the right. Similar to a rolling shutter movement. The origin of object frame is marked with a red cross

```
#Creating 3D array of Object points in mm
↪ -------------------------------------

l = 69.50
w = 27.94
h = 0

objpoints = np.array([[-w/2, -l/2, 0],
                      [w/2, -l/2, 0],
                      [w/2, l/2, 0],
                      [-w/2, l/2, 0]])
```

4.4 P4P Solver

All of the parameters needed for P4P is now assumed to be collected, given one has followed the setup and executed the program described in Sections (4.1) (4.2)(4.3).This data can now be used as input to calculate the pose of the object with respect to the camera frame with P4P.

The image points are sorted relatively to each other and will now be used in P4P solver along with the rest of the parameters. The image points are converted into normalized image coordinates with Equation (2.1) and distortion is accounted. The distortion in Equation (2.4) is accounting for 12 elements of distortion. The camera calibration script shown in Section (4.2) accounts for radial- and tangential distortion as these are the usually the most significant. According to openCV's documentation [29], radial distortion can be represented as:

$$
\begin{aligned}
x_{distorted} &= x(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) \\
y_{distorted} &= y(1 + k_1 r^2 + k_2 r^4 + k_3 r^6)
\end{aligned}
$$

and tangential distortion as

$$
\begin{aligned}
x_{distorted} &= x + [2p_1 xy + p_2(r^2 + 2x^2)] \\
y_{distorted} &= y + [p_1(r^2 + 2y^2) + 2p_2 xy]
\end{aligned}
$$

The distortion coefficent found in Section (4.2) script are

$$
\text{Distortion coefficients} = \begin{bmatrix} k_1 & k_2 & p_1 & p_2 & k_3 \end{bmatrix} \tag{45}
$$

After accounting for distortion, the pose is now calculated by using the method presented in Section (2.3).

4.4.1 Multiple solutions

A problem that needs to be addressed is that the pixelSorting() algorithm will choose one out of 4 points as a reference point. This is expected to be random each time. The relative position to RP is set, but let's make some examples here to illustrate what is happening. It is assumed that True rotation matrix is identity matrix in the image. It may look something similar to this:

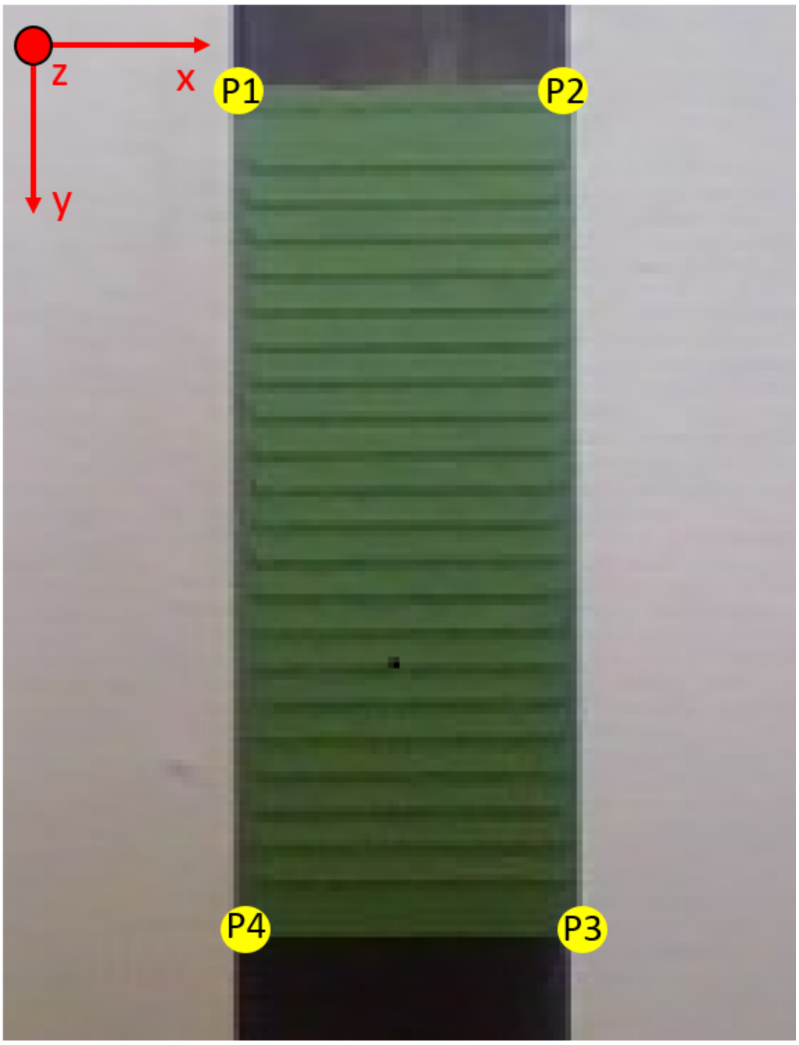


Figure 55: expected rotation matrix of container in this image with respect to camera is identity matrix. Z axis would be equivalent to heave and is positive when pointing towards the object.

Further, the four different possible solutions will be presented.

Case 1: It chooses the RP as $\boldsymbol{pt1}$ as illustrated in Figure (44), $\boldsymbol{pt2}$ as its closest, $\boldsymbol{pt3}$ as its diagonal and $\boldsymbol{pt4}$ as it last point, as illustrated in the Section (4.1.3). Since it corresponds with the object points given in Section (4.3), the PnP will return $\boldsymbol{R} = \boldsymbol{I}$.

Case 2: RP in this case is $\boldsymbol{pt2}$. It will choose $\boldsymbol{pt1}$ as it closest point. Diagonal will be $\boldsymbol{pt4}$ and the last would be $\boldsymbol{pt3}$. Since the algorithm expects the RP to be where $\boldsymbol{pt1}$ is located, it means that the algorithm will calculate that the plane is flipped π radians about the y-axis. This means that

$$\boldsymbol{R} = \begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{bmatrix} \tag{46}$$

which equivalently means that the container is flipped on its head, which seems highly unlikely. Another way to look at it is a rotation of π angles about roll angle, illustrated in Figure (30), seen from above the ship where the Z-axis is heave.

Case 3: RP in this case is $\boldsymbol{pt3}$. It will choose $\boldsymbol{pt4}$ as it closest point. Diagonal will be $\boldsymbol{pt1}$ and the last would be $\boldsymbol{pt2}$. Since the algorithm expects the RP to be where $\boldsymbol{pt1}$ is located, it means that the algorithm will calculate that the plane is flipped π radians about the roll axis. This means that

$$\boldsymbol{R} = \begin{bmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{47}$$

which equivalently means that it is rotated about the yaw axis π radians.

Case 4: RP in this case is $\boldsymbol{pt4}$. It will choose $\boldsymbol{pt3}$ as it closest point. Diagonal will be $\boldsymbol{pt2}$ and the last would be $\boldsymbol{pt1}$. Since the algorithm expects the RP to be where $\boldsymbol{pt1}$ is located, it means that the algorithm will calculate that the plane is flipped π radians about the y-axis. This means that

$$\boldsymbol{R} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{bmatrix} \tag{48}$$

which equivalently means that the object is rotation π radians about the pitch angle.

Due to this, the function correctsRmatrix() is created to compensate for this. If the rotation matrix $\boldsymbol{R}$ returns a matrix with the signs on the diagonal described in case 2, 3 or 4, it will flip $\boldsymbol{R}$ π radians towards a solution that is positively oriented along the diagonal of $\boldsymbol{R}$ so it will be closer to identity matrix for each frame. A tool to analyze the test results quickly frame by frame, AR is utilized to project the orthogonal axis on the plane with origin in the reference point RP. It is worth noting that the red line is projecting normal to the plane, away from the object. The AR solves the PnP equation with respect to image points with the cv2.projectPoints() function. From this, vectors based on the image points are drawn onto the image.

4.5 Test Rig

Figure 56: Setup of the produced Rig after being designed in CAD.

In order to test the accuracy of the system, a test rig was created to test the results. The test rig is designed to fully define the 6 DOFs of the camera- and object frame. The camera's optical line will be perpendicular to the vertical plate it is leaning against. For the object, the plate can rotate about an axis of the pin. This axis that this pin creates intersects the center of the container. The reason for this is that the pinhole's position relative to the camera frame is known. With this information, when testing different angles, the translation is expected to remain the same, independently of how the container is rotated. In this experiment, it is assumed that the geometry is ideal and can represent the True pose with no errors. This may not be realistic, but for result comparisons, this is assumed.

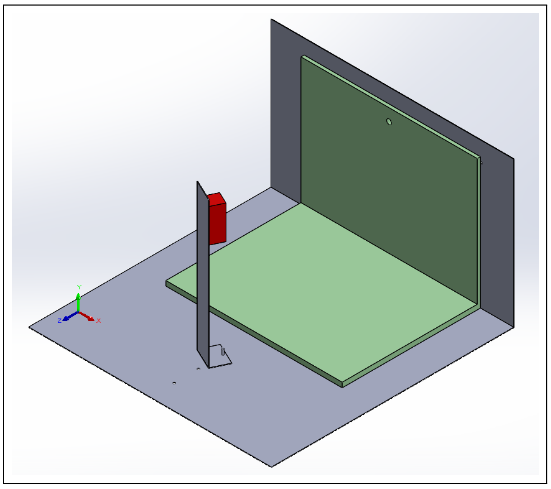


Figure 57: CAD of test rig made in Solidworks. Green is laptop with integrated webcam. Red is container

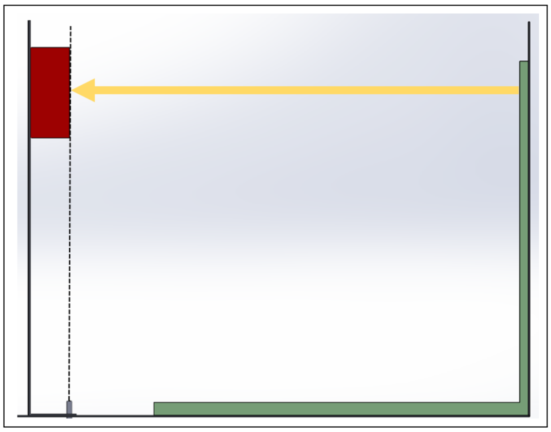


Figure 58: CAD of test rig made in Solidworks. Green is laptop with integrated webcam. Red is container

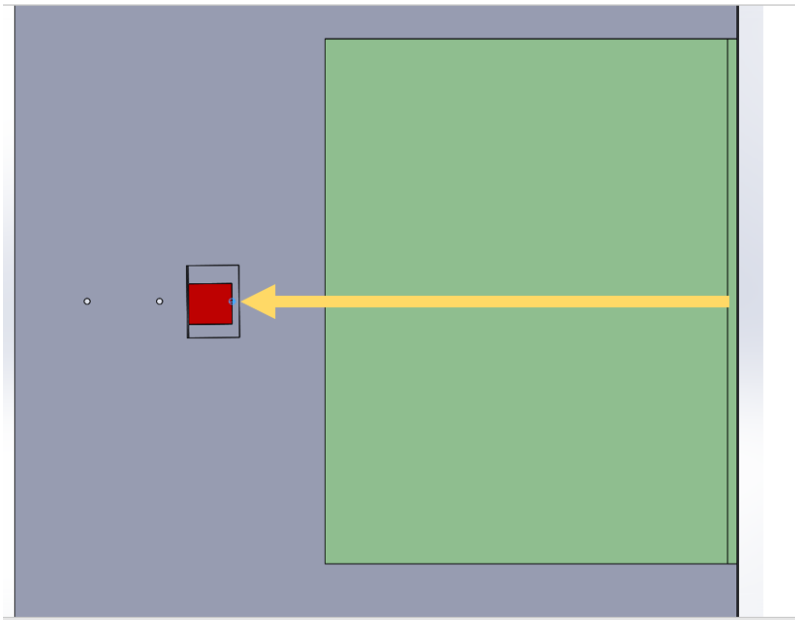


Figure 59: Yellow arrow illustrates the optical axis aiming at the centre of the ceiling of the container

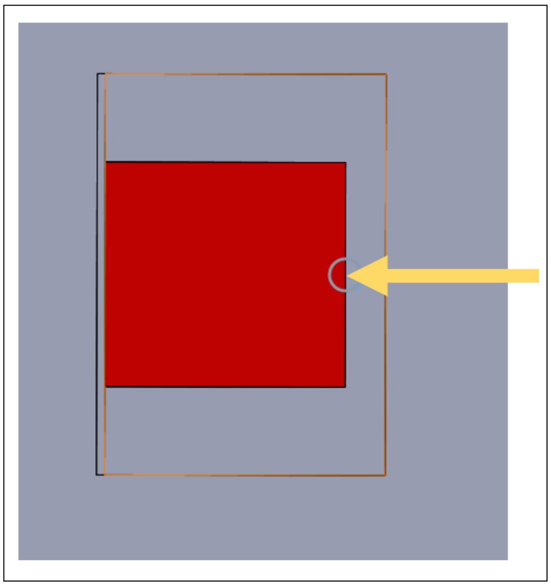


Figure 60: yellow arrow is pointing at the rotation axis

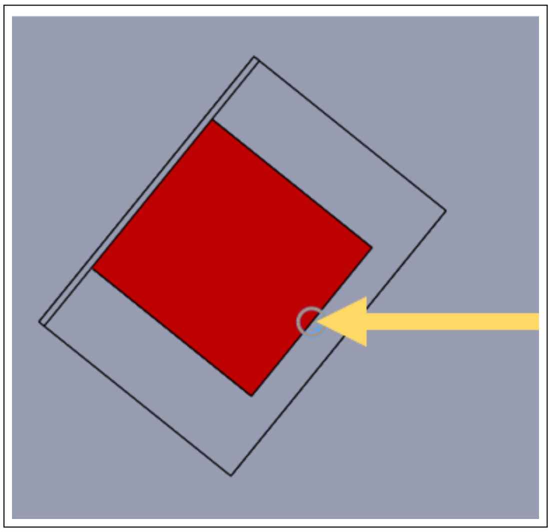
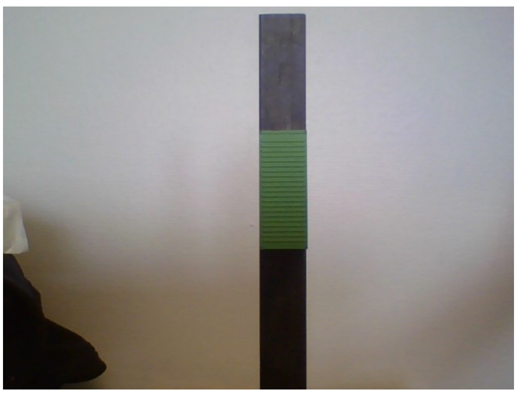


Figure 61: translation of the object is invariant of rotation since the object frame does not move

4.6 Ground Truth image point extraction

A method to test how well the feature extractor in Section (4.1) is performing, it will be tested against a benchmark. This benchmark consists of using ground truth (GT) image points. These will be handpicked in the test image and used as input for a P4P solver that is explained in Section (2.3). In the software paint, on can hover over pixels and the coordinates will be printed. The image points were manually selected and used as input.



(a) A test image used in the experiments

(b) Same image, but zoomed in. The image coordinates were manually extracted

4.7 Video experiment

In the previous experiment, accuracy was tested. Since the purpose of this test is tracking, video testing was utilized as a way to evaluate speed performance and noise. A test featuring accuracy, speed and noise evaluations simultaneously is the optimal circumstance to evaluate the system. For testing live video, while at the same time knowing the true pose for each frame can be done but is challenging. Instead, testing with video input for evaluating the feature extractor and potential noise is evaluated. This can be done by analyzing the corner detector in each frame.

The test video used can be seen here: Original Test Video[4]

[4] https://www.youtube.com/watch?v=cHk5R2saTi4

5 Results

This section will introduce various results of the system.

5.1 Instance Segmentation Model

Some metrics of the train DL model is being explained before presented. The total loss function, FP and FP is calculated during training. The precision and recall are calculated by using the validation set.

5.1.1 Deep Learning Metrics

Total loss function:

The multi-task loss function of Mask R-CNN combines the loss of classification, localization and segmentation mask:

$$\mathcal{L}_{total} = \mathcal{L}_{cls} + \mathcal{L}_{box} + \mathcal{L}_{mask} \tag{49}$$

more details on how each loss functions is calculated, see Appendix (F).

false negative:

If a container ceiling is present in the image and the model predicts no object of the class in the image, it will return a false negative since it wrongfully claimed there was no object.

false positive:

If the model predicts an object in the image, but the $IoU < 0.5$ between GT and prediction, then it classified as a FP. It is invariant whether there exists an object in the image, it only checks if it passes the IoU threshold compared to GT set by the user.

Precision @IoU:

The precision and recall metrics are calculated by running the trained model in the validation set.

Precision is explained in Section (2.5.5.3) as:

$$Precision = \frac{TP}{TP + FP} \tag{50}$$

In the false positive explanation above, it was explained that it returns TP or FP based on a threshold value for IoU. In this experiment, the average precision (AP) is calculated with different IoU values from all the images in a validation dataset. If the IoU notation is $@IoU = 0.50 : 0 : 95$ it means that it calculates AP for all IoU with and incremental step of 0.05, starting from 0.5 and up to 0.95.

Recall @IoU:

Recall is explained in (2.5.5.4) as:

$$Recall = \frac{TP}{TP + FN} \tag{51}$$

The recall in the results includes the average recall (AR) from different IoU thresholds.

5.1.2 Deep Learning Results

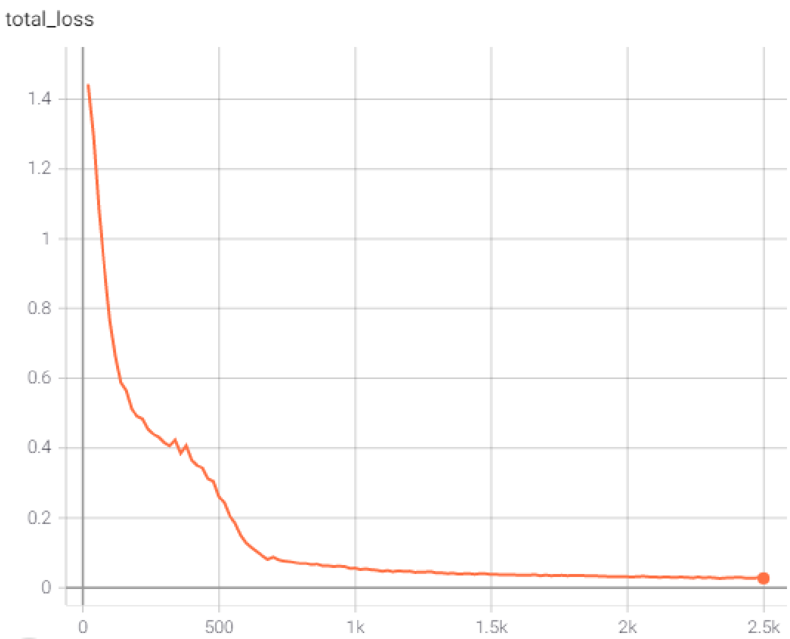


Figure 63: $\mathcal{L}_{total}$ over i iterations. $\mathcal{L}_{total} = 0.02679$ at 2500 iterations.

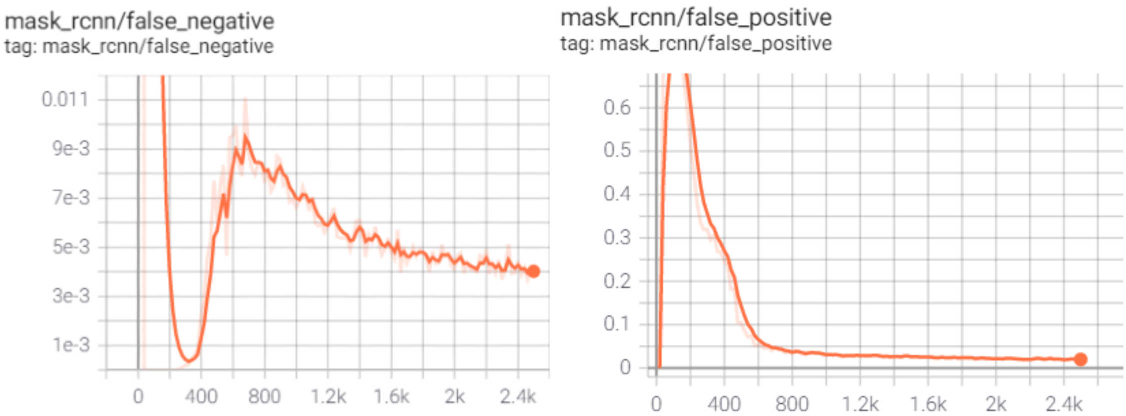


Figure 64: Horizontal axis represents number of iterations during training.

Precision table	
Average Precision (AP) @ IoU=0.50:0.95	= 42.1%
Average Precision (AP) @ IoU=0.50	= 62.1%
Average Precision (AP) @ IoU=0.75	= 56.6%

Recall table	
Average Recall (AR) @ IoU=0.50:0.95	= 56.6%

5.2 Accuracy Test P4P

Test with estimated True pose of identity matrix and translation vector $(x, y, z) = (0, 0, 300)$ in mm.

5.2.1 With instance segmentation and gftt

Achieved results on test image:

rotate π radians about the z-axis

$$\boldsymbol{T_{co}^{c}} = \begin{bmatrix} 0.999999 & 0.006836 & 0.000119 & 13.410 \\ 0.000007 & 0.999974 & 0.002432 & 11.440 \\ -0.000119 & -0.002432 & 0.999974 & 314.801 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{52}$$

Yaw, Pitch, Roll respectively in degrees:

$$\begin{bmatrix} -0.1394 & 0.0004 & 0.0068 \end{bmatrix} \tag{53}$$

5.2.2 With GT image points

Comparing these results to an instance where GT image points are used as input instead of instance segmentation + gftt().

$$\boldsymbol{T_{co}^{c}} = \begin{bmatrix} 0.999999 & 0.000000 & 0.000121 & 13.365 \\ 0.000006 & 0.999997 & 0.002436 & 11.640 \\ -0.000121 & -0.002436 & 0.999997 & 313.758 \\ 0.000000 & 0.000000 & 0.000000 & 1.000000 \end{bmatrix} \tag{54}$$

Yaw, Pitch, Roll respectively in degrees:

$$\begin{bmatrix} -0.1396 & 0.0004 & 0.0070 \end{bmatrix} \tag{55}$$

5.2.3 Error between instance segmentation + gftt and GT image points

If there is not error, then

$$\boldsymbol{I} = \boldsymbol{R}\boldsymbol{R}^T \tag{56}$$

The error between $\boldsymbol{R}_{AI}\boldsymbol{R}_{GT}$ is calculated by expecting an identity matrix $\boldsymbol{I}$

$$\boldsymbol{R}_{error} = \boldsymbol{R}_{AI}\boldsymbol{R}_{GT}^T = \begin{bmatrix} 0.99999 & 0.00684 & -0.00002 \\ 0.00000 & 0.99998 & -0.00000 \\ 0.00000 & 0.00000 & 0.99997 \end{bmatrix} \tag{57}$$

Yaw, Pitch, Roll respectively in degrees:

$$\begin{bmatrix} 0.00023 & 0.00042 & -0.0001 \end{bmatrix} \tag{58}$$

Translation vector

$$\boldsymbol{t}_{AI}\boldsymbol{t}_{GT} = 0.044 - 0.21.043 \tag{59}$$

5.2.4 High resolution 1280x720 GT

An instance where GT image points was used with a higher resolution camera. Expected True pose is identity matrix for rotation and translation vector $(x, y, z) = (0, 0, 300)$ in mm.

$$\boldsymbol{T}_{\boldsymbol{co}}^{\boldsymbol{c}} = \begin{bmatrix} 1.00000 & -0.000000 & 0.000000 & -0.9110 \\ -0.000000 & 1.00000 & 0.000000 & 20.3160 \\ -0.000000 & -0.000000 & 1.000000 & 308.2378 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{60}$$

Yaw, Pitch, Roll respectively in degrees:

$$\begin{bmatrix} 0.0000 & -0.0000 & 0.0000 \end{bmatrix} \tag{61}$$

5.3 Feature extraction

These results include the instance segmentation and gftt(). The results are comparing the GT image points from the test image and the image points extracted from instance segmentation model and gftt().

$$\text{GT image points} = \begin{bmatrix} 321 & 155 \\ 380 & 155 \\ 380 & 304 \\ 322 & 304 \end{bmatrix} \tag{62}$$

$$
\text{Mask R-CNN} + \text{gftt()} = \begin{bmatrix} 323 & 156 \\ 377 & 156 \\ 378 & 302 \\ 324 & 302 \end{bmatrix} \tag{63}
$$

$$
P_{error} = P_{AI+gftt} - P_{GT} = \begin{bmatrix} 2 & 1 \\ -3 & 1 \\ -2 & -2 \\ 2 & -2 \end{bmatrix} \tag{64}
$$

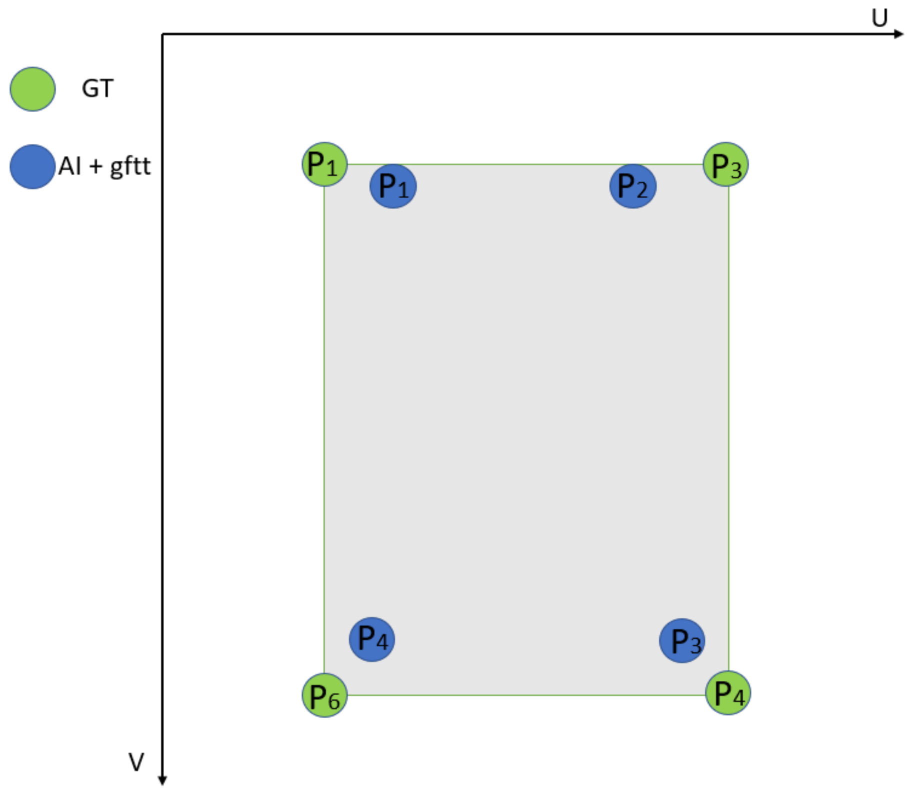


Figure 65: Illustrating relative error between GT and AI + gftt in image plane. Scale is not precise with respect to image plane of 640x480.

5.4 Speed Test

Full Pose Estimation[5] Speed testing this 120 frames video on local computer with RTX 3070, including pose estimation and visualizations. To calculate pose of 120 frames took 13.0 seconds. This corresponds to 9.23 FPS.

Instance segmentation + gftt()[6] Same video input, but with gftt() and top of instance predictions. Managed to perform at 13 seconds or 9.23 fps.

Instance segmentation[7] Time spent instance segmentation on video input with 120 frames was 13.0 seconds when run locally with a single RTX 3070 GPU. This equals to an inference speed of 9.23 FPS.

Default Visualizer implementation[8] This video shows the standard visualizer class with the trained model. Used 14.0 seconds to predict 120 frames.

5.5 Camera Calibration

Intrinsic camera matrix $\boldsymbol{K}$:

$$\boldsymbol{K} = \begin{bmatrix} f_x & s & x_0 \\ 0 & f_y & y_0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 662.61758 & 0.00000 & 322.27689 \\ 0.00000 & 661.39105 & 204.96692 \\ 0.00000 & 0.00000 & 1.00000 \end{bmatrix} \tag{65}$$

Distortion coefficients $= \begin{bmatrix} k_1 & k_2 & p_1 & p_2 & k_3 \end{bmatrix}$ with variables described in Section (2.4) that accounts for radial- and tangential distortion.

$$\begin{bmatrix} k_1 \\ k_2 \\ p_1 \\ p_2 \\ k_3 \end{bmatrix} = \begin{bmatrix} 0.02586 \\ -0.06599 \\ 0.00118 \\ 0.00001 \\ -0.21194 \end{bmatrix} \tag{66}$$

[5] https://youtu.be/kPFWiagKGG8
[6] https://youtu.be/L5DCWHwRRVU
[7] https://youtu.be/LKBR3pX3BrY
[8] https://youtu.be/ID4tRdz48vY

6 Discussion

6.1 Accuracy

This subsection will present factors that may have affected the accuracy of the system.

6.1.1 True Pose inaccuracy

(u,v) coordinates had errors. The optical axis is expected to intersect the object frame in the center of the container ceiling. By checking the test image, it showed that the object frame was projected in image coordinates (429, 266) instead of the image center, which is located at the coordinates half of the image resolution $(u, v) = (320, 240)$. By calculating the spatial resolution regarding the deviation, one can correct the translation in the (x,y) direction.

$$\boldsymbol{T^o_{co}} = \begin{bmatrix} 0.999999 & 0.006836 & 0.000110 & 13.410 \\ 0.000007 & 0.999974 & 0.002430 & 11.440 \\ -0.000119 & -0.002432 & 0.999974 & 314.801 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{67}$$

The expected true pose from the test was

$$\boldsymbol{T^c_{co}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 300 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{68}$$

After studying the test image, the object frame is located in 350x232. This indicates that the True pose has some offset in the XY plane. In the image plane it has an offset from image centre by $(u, v) = (30, -8)$. The optical projection line almost intersects the long edge. Since the distance from the longest edge to the centre of the object is $\frac{w}{2} = \frac{27.94}{2} = 13.97$. This can also be calculated with the spatial resolution of a pixel and calculate how many pixels are between the projective center line and object frame. The distance from where the optical centerline intersects the object and the object frame was $y = 5.80mm$.

The correction of this data is added to the previous True pose transformation matrix:

$$\boldsymbol{T_{True}} = \begin{bmatrix} 1 & 0 & 0 & 13.97 \\ 0 & 1 & 0 & 5.80 \\ 0 & 0 & 1 & 300 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{69}$$

Calculating the error of True pose and pose estimation from the AI + gftt() with

$$\boldsymbol{R}_{error} = \boldsymbol{R}_{AI+gftt}\boldsymbol{R}^T_{True} \tag{70}$$

, and expecting Identity matrix $\boldsymbol{I}$, and error in translation with

$$\boldsymbol{t}_{error} = \boldsymbol{t}_{AI+gftt} - \boldsymbol{t}_{True} \tag{71}$$

would result in a error:

$$\boldsymbol{T_{error}} = \boldsymbol{T}_{AI+gftt} - \boldsymbol{T}_{True} = \begin{bmatrix} 0.99999 & 0.00684 & 0.00012 & 0.56 \\ 0.00000 & 0.99997 & 0.00243 & 5.64 \\ -0.00012 & -0.00243200 & 0.999974 & 14.801 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{72}$$

The rotation matrix $\boldsymbol{R}_{error}$ was close to an identity matrix. The most significant error would be the depth translation of 14.801 mm. It should be noted that using a higher resolution (1280x720) resulted in an error of 8.238 mm, described in Section (5.2.4).

6.1.2 Feature Extractor

When considering the pose estimation, the Z-axis measured 315 mm while the true pose was 300 mm, as shown in Figure (65). This metric showed the most significant deviance of the 3 translation axis. The image points projected from the AI + gftt were relatively closer to adjacent image points than the GT image points. By looking at the Figure (65), it shows that the planar surface has a smaller projection in the image plane when using AI + gftt than GT. Consequently, this will make it seem like the 3D object is further away than it is. In the accuracy-test, shown in Section (5.2.3), the results demonstrated that the object was estimated to be further away from it than it did with the GT results with its 315 mm instead of 308.6 mm.

In terms of feature extraction, it achieved a mean error in the image plane compared to GT

$$Error_{avg} = \frac{\sum vec_{GT}^{AI+gftt}}{n_p} = 2.76 \tag{73}$$

where $vec_{GT}^{AI+gftt}$ is the length of a vector in the image plane defined by prediction image point generated from AI + gftt and to the GT image point. The number of image points used in this calculation is the 4 from the image point list in Equation (5.3).

Another method of evaluating the accuracy of the feature extractor, a comparison between the image point extracted and the GT image points, is made by calculating the error in pose estimations Section (5.2.3). The error was at most 1.043 mm along the Z-axis. So, just by isolating the instance segmentation and gftt() as a feature extractor, this has a relatively good accuracy.

6.1.3 pixelsorter

The pixelSorting()-method has been explained in Section (4.1.4). The algorithm is based on the length of vectors in the image plane. It exists circumstances where this algorithm will fail.

This youtube video Container Projections[9] illustrates how different perspectives on the container ceiling is subdued to different projections in the image plane by looking at it from different angles.

To make an algorithm that sorts and assigns image points corresponding to 3D object points, one must be aware that it may fail under different projective transformations if the algorithm is based on vectors and lengths between edges in the image plane.

Ex: Let us assume that the pixel coordinates in all four corners on the container's ceiling are already established in an image correctly. So, if one wants to define the two corners that make up the short edge based on finding the closest points, this fails in angles such as shown in Figure (66). The pixelSorting()-function calculates the shortest vector in the image plane, defined by two image points. In this specific image, the shortest vector projected in the image plane will be two points that form the longest edge.

Figure 66: In this image, the longest edges are projected as shorter compared to the short edges in the image plane.

The problem that may occur and is illustrated in the Figure (66), is that in certain angles between the camera and the planar surface is that the projection in the image plane may change the shortest distance for the points. At the end of the Video (6.1.3), the longest edge will actually be projected as shorter than the shortest edge in the image plane. When designing an algorithm to sort this, this may need to be solved, depending on what the relative perspective one is encountering. If the camera is held above, as suggested in Section (3.2), then the camera is assumed to avoid this problem.

[9] https://youtu.be/UdprbdvsJL8

6.1.4 Multiple solutions

In Section (4.1.4), the problem with multiple solutions were discussed. This system calculates the pose of a planar surface and will be able to return multiple solutions. This includes

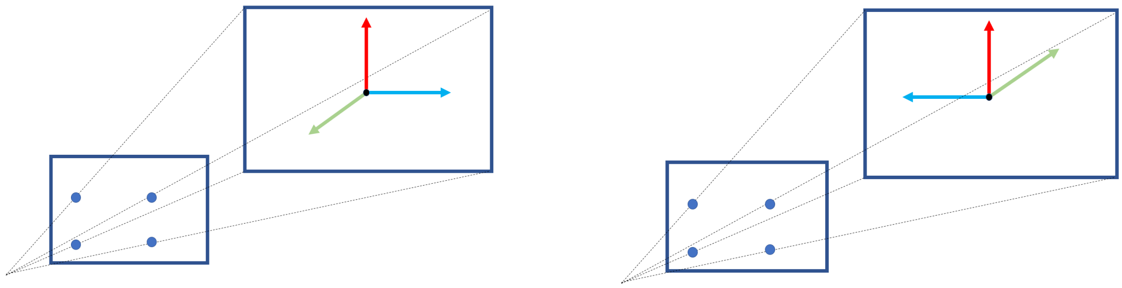


Figure 67: The PnP solver may return inverted Z-orientation due to randomized reference point in pixelSorting(). Left is the expected orientation in the object frame with respect to camera. Right image is rotated about red axis.

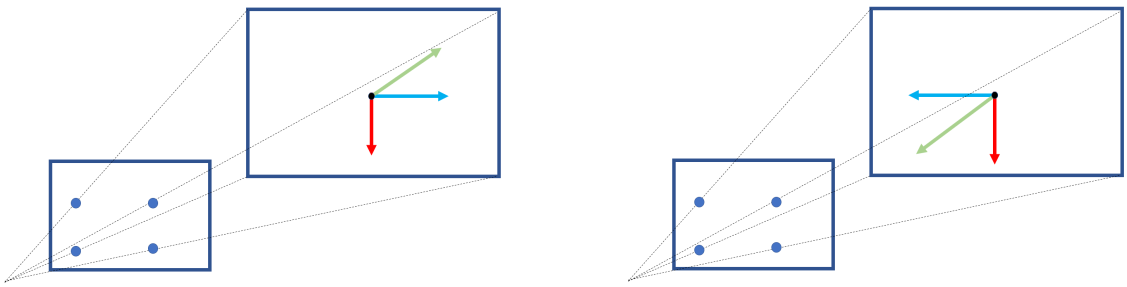


Figure 68: Rotated 180 degrees about blue axis (left image) and rotated 180 degrees about green axis (right image)

There are four possible solutions, illustrated in Figures (67)(68). The P4P algorithm will return one of these solutions. The correctsRmatrix() will flip the plane so that it has a positive z axis. If the z axis is negative, it would indicate that the camera would see the plane for underneath. This is an unrealistic circumstance since it would mean that either the container is flipped upside down or that the camera can see the image points from underneath, which seems unrealistic for the scenario where the camera is located with a top plane view.

The function correctsRmatrix() will pass the output rotation matrix and it will check which orientation the X,Y and Z axis is returning. If the x and z is inverted (negative sign) and y is positive, the algorithm will rotate π radians about the y axis. The same applies to circumstances for x as the only positive and a circumstance where z is the only positive. So, the correctsRmatrix() would correct the matrix so it outputs an expected orientation where the depth translation value Z is positive and the camera is looking at the object from above.

6.1.5 Image resolution

Image input in this experiment is 640x480 with 3 layered channel consisting of RGB color intensities. This is represented in a 3 Dimensional matrix with a array size of $640x480x3 = 921600$. The deviation from the True pose was suspected to partially be represented by low spatial resolution. If one chooses to increase it to full HD or 1920x1080, it would mean an image input of array size of $1920x1080x3 = 6220800$ which is $\frac{6220800}{921600} = 6.75$ larger than the initial image input with 640 x 480 resolution. The computational cost would increase, but the relative spatial resolution would also increase by $\frac{1920}{640} = 3$ along x-axis and $\frac{1080}{480} = 2.25$ in the y-direction.

Increasing pixel resolution further to 4k Will increase the image matrices 27 times, but precision may increase with 4.5 spatial resolution in the v-direction and 6 times in the u-direction, seen from the image plane.

The effect of adjusting the same webcam to a resolution of 1280x720 was noticeable. The difference in rotation of the ground truth images between Equation (5.2.2) and (5.2.4) was considered neglectable, but the translation improved from 313 mm to 308 mm in depth, where the actual depth was 300 mm. It is indicated through the test results in Section (5.2.4) that increasing the spatial resolution may help the accuracy of the system at the expense of increased computational cost. The computational cost can make the system slower.

6.2 Video Performance

The performance of the video was promising but showed some noisy frames. The DL model could predict an object in all 120 frames with a confidence threshold of 0.9 or higher. Studying the video frame by frame, the system is somewhat prone to noise as the lines predicted tends to be curvy in certain areas and affect the gftt()-algorithm. It could make it so that the corner detector returned false positives in terms of image points that would result in a significant error as illustrated with the Figure (69).

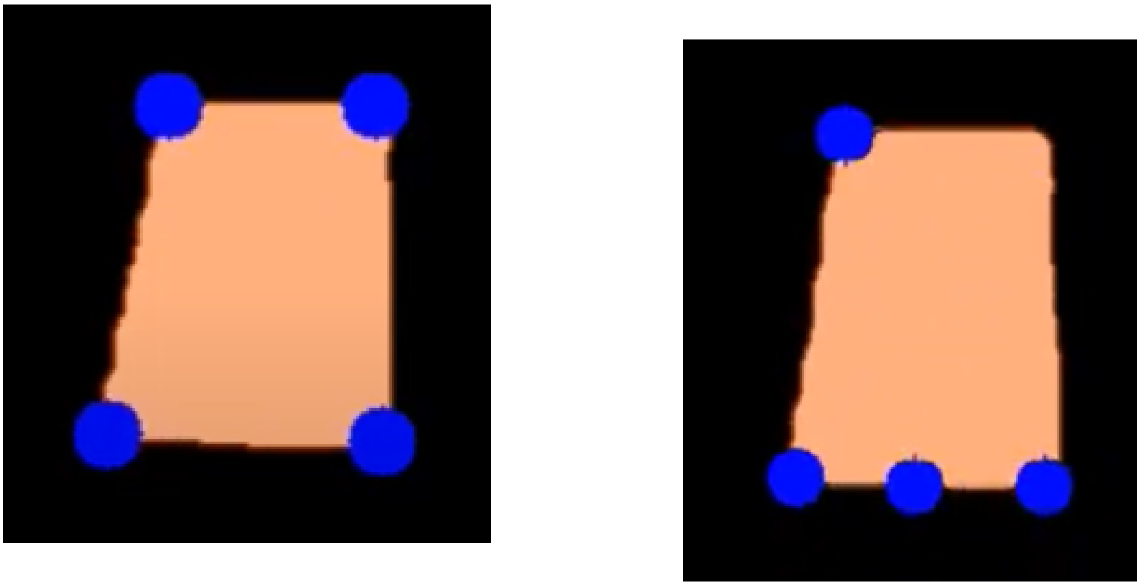


Figure 69: Example of how the corner detector find inliers (left) and outliers (right). The pixel extraction is a cooperation between DL-model and the gftt()-algorithm and the desired outcome is to find the 4 corners of the rectangle.

6.2.1 Overfitting

The model performed well in trained environments but was over-fitted due to the problem it had to generalize the containerCeiling-class in other circumstances. It was attempted to create a more generalized model, but the result was highly curved lines, see Figure (69). The generalized model managed to be better when detecting a container in an arbitrary image outside the training dataset. However, it did return more curved images, which resulted in corner detection, as seen in Figure (69).

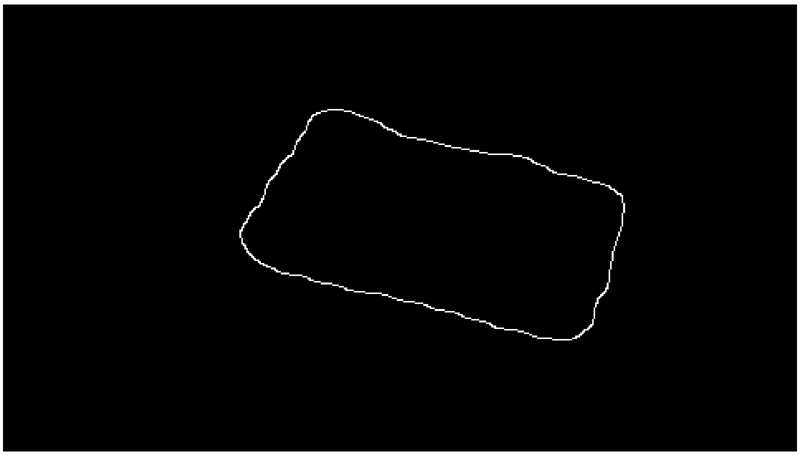


Figure 70: A canny image for highlighting of the edges. It shower that a more generalized model resulted in curvy edges. The corner detector had difficulty finding the 4 corners of the rectangle with this prediction.

It was suspected that it needed more training to remove the uncertainty it returned around the edges in the output image, shown in Figure (70). To solve this, an over-fitted model was trained to provide a proof of concept where the circumstance is a well-trained model for an image. It was decided that it should be trained on the same images it was to be tested on, see Figure (36). The purpose of this is to provide a model that is optimally trained so one can observe an instance where an ideal model is trained and can output a high-quality output for the rest of the system. Surprisingly, the metrics found by examining the model with the validation set in Section (5.1.2) was achieving a relatively high score with Average precision and average recall. However, this experiment requires very high precision from the model in order to extract image points that are close to the ground truth. In some frames, the image points extracted could look like what is illustrated here in Figure (69), due to uneven predictions.

6.3 Evaluation

A total evaluation of the system is presented.

This report aimed to develop a computer vision system that can help a robotic gripper, most likely a hydraulic crane of some sort, to pick up cargo from a ship and land onto an offshore platform, and vice versa. This system has proven to calculate the pose of small-scale shipping containers with an accuracy of approximately 15 mm at 9.23 FPS with 640x480 image resolution. Its principle can be used to further develop a pose estimation for other planar surfaces, including but not limited to barrels and ship decks.

The translation vector was more accurate when using higher image resolution in Equation (6.1.5), so it is indicated that in order to increase the accuracy of the translation vector, higher resolution can be a promising solution.

During testing with video, some frames were noisy. The robustness could increase by optimizing the AI model or image point extractor as explained in Section (4.1). The speed of 9.23 fps is assumed to be sufficient for a robotic arm that is assumed to be a slow hydraulic crane.

It remains a solution for generating the 3D geometry of the platform drop zone. For now, the computer vision system should be able to pick up containers with the 3D data, but it does not currently have 3D data about where to land the object. It still needs an algorithm to solve for pose estimation of the ship to land cargo onto ships. It requires 3D data of the landing zone for landing cargo onto the platform as well. The platform 3D data may potentially consist of a predefined 3D point cloud under the assumption that the orientation and translation are assumed to be fixed for the platform. Therefore, tracking might not be necessary. The computer vision system should include safety systems and anti-collision control. However, this is out of scope for this report.

6.4 Further Work

3 things will be suggested for improving this system. The first is to upgrade to the speed of the instance segmentation, as discussed in Section (6.4.1). It will allow an increase of image resolution, which in return indicated an increase in accuracy in Section (6.1.5). Second, it is suggested in Section (6.4.3) to improve the stability of point correspondences with quadrilateral fitting using contours, to filter out noise. Also, using optical flow for pixel sorting and filtering outliers with the current P4P solution or with the corners of the quadrilateral-fitted rectangle may stabilize the rotation matrix, as discussed in Section (6.4.2).

These methods is believed to make the system more accurate and stable for rectangle planar pose estimation with less than 10 mm error in translation, with the possibility of doing so in real-time or at least close to.

6.4.1 YOLACT++

Consider upgrading to a new model, such as YOLACT++. [30].

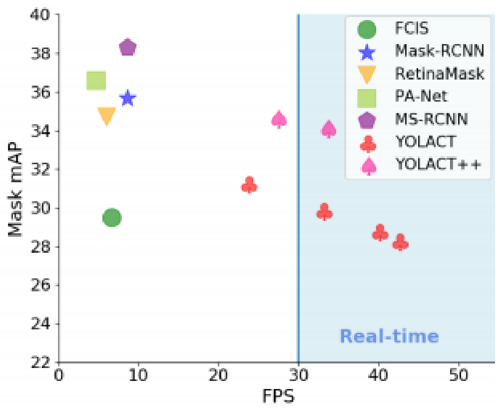


Figure 71: "Speed-performance trade-off for various instance segmentation methods on COCO. To our knowledge, ours is the first real-time (above 30 FPS) approach with over 30 mask mAP on COCO test-dev" Figure and quote from [30].

The paper of YOLACT++ [30] documented that it could have a precision close to state-of-the-art models while running in real-time (>30fps). There are still imperfections in mask generation in both YOLACT++ and Mask R-CNN. However, the speed on YOLACT++ in conjunction with a more robust image point extractor could make this system significantly more accurate and possibly more than 3x faster. This faster instance segmentation opens up the opportunity of increasing the image resolution that was indicated to improve the accuracy of the pose estimation, according to the results in Section (6.1.5). It is suggested to set the resolution to 1920 x 1080 initially, and from there, do a parameter study of optimal image resolution where accuracy and speed are measured.

6.4.2 Optical Flow

It can be observed in the videos containing gftt() in Section (5.4), recognized by the circles drawn in the image, that the corner detector is prone to noise in certain frames. For the first step, instance segmentation is applied. The output of this is used as input to gftt(). In certain frames, such as in Figure (69), one can observe the predicted corners is placed along the more curvy edges, created by the DL-model. The system can benefit from using a system that consider the previous frames when predicting the pixel coordinates in the next one in order to filter out these sudden jumps in the image point location. This can be solved by using optical flow to the drawn on circles. According to the documentation in OpenCV [60], optical flow works on assumptions that:

- The pixel intensities of an object do not change between consecutive frames.
- Neighbouring pixels have similar motion.

When using the output of the predicted image with the custom post processed filter, the pixel intensities remain the same for the pixels that are to be tracked and this also is valid for its neighbouring pixels. For example, the circles drawn onto the image here will have the same color intensities for each frame, illustrated in Figure (37). After the first image frame, this method can also replace the pixelSorting()-algorithm. The intention of pixelSorting() allocates a pixel coordinate to its corresponding 3D object point, but optical flow can allocate the consecutive frames' image point to the corresponding object point. For the first frame it needs to be sorted as it has no previous reference but afterwards, the optical flow can control the noise and image point sorting. Since optical flow works under the assumption that the color intensity remains the same between frames, then all the corner pixels can have its own unique color for more lenient pixel Sorting. I.e. the first pixel can be red (I.e. RGB= (255,0,0)), the next image point orange, then yellow and so on. Then the red pixel can correspond to a given object point as illustrated underneath in Figure (72).

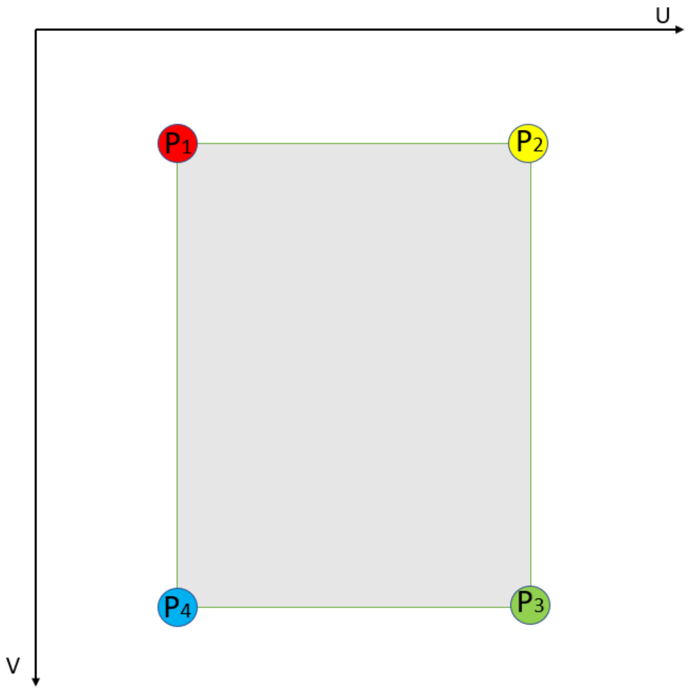


Figure 72: Optical flow is color based. Using post processing of image input can give constant color assignments

6.4.3 Quadrilateral fitting

Based on the experiments, increasing the image resolution improved the accuracy of the translation vector. A supplement to make the solution more robust would be to increase the number of points used as well, as the translation tended to show some volatility during testing due to outliers.

Instead of detecting 4 corners, one can use contours to fit a rectangle, also known as quadrilateral fitting. This method considers more points into the pose estimation, so the model is assumed to become more stable and more computationally expensive. The general idea is to include more points so the noise will be reduced if some image points have a deviation from the ground truth, but with the current model with fps < 10, it will become a trade-off with speed for stability.

Figure 73: The custom post processing can be converted using contours generation. This rectangle can then be used for quadrilateral fitting

7 Conclusion

This report sought to solve the computer vision aspect of an autonomous offshore crane lift system. Different requirements were addressed. To solve it, one needs 3D data about the ship, platform and cargo. The scope of this report was decided to be narrowed down to cargo tracking.

To track cargo, it was attempted to use deep learning for feature extraction in conjunction with goodFeaturesToTrack-algorithm from OpenCV, then solve pose with PnP where $n = 4$.

To test this system, a small-scale shipping container was used as a test object. It delivered some promising results with an accuracy of 0.14 degrees and 15 mm, with a speed of 9.23 fps, given a very well-trained instance segmentation model was used. Increasing the image resolution from 640x480 to 1280x720 further improved the accuracy to 8 mm, and 0 degrees error up to the 5th decimal. It was strongly indicated that this system can benefit for higher image resolution and it should be implemented.

During testing, it was clear that some noisy image points needed to be filtered out to make the system more robust. It was discussed different methods for increasing the robustness such as optical flow and quadrilateral fitting. Further work also suggests exploring a different instance segmentation model named YOLACT++ due to its documented accuracy and speed. Combining this new, fast 30 fps instance segmentation model with higher image resolution, quadrilateral fitting for feature extracton, and cross-examine the image points with optical flow seems promising to make the system more accurate, stable and faster.

Bibliography

[1] O. Egeland, *Robot vision*. London: Department of Mechanical and Industrial Engineering NTNU, 2020.

[2] [Online]. Available: https://learnopencv.com/understanding-lens-distortion/

[3] R. ALEXANDRE, "Dem spatial resolution – what does this mean for flood modellers?" https://www.jbarisk.com/news-blogs/dem-spatial-resolution-what-does-this-mean-for-flood-modellers/ [Accessed: 08.06.21].

[4] "Swir resolution+ unlocks potential," https://www.geoimage.com.au/SWIR%20Series/resolution.

[5] "Image segmentation with machine learning." [Online]. Available: https://data-flair.training/blogs/image-segmentation-machine-learning/

[6] 2016. [Online]. Available: https://medium.com/@ivanliljeqvist/the-essence-of-artificial-neural-networks-5de300c995d6

[7] 2019. [Online]. Available: https://www.mdpi.com/2073-4425/10/7/553/htm

[8] "Tesla autopilot recognition of roadside structures," 2018, https://www.youtube.com/watch?v=7ztK5AhShqU&ab_channel=greentheonly[Accessed: 30.11.20].

[9] G. Litjens, T. Kooi, B. E. Bejnordi, A. A. A. Setio, F. Ciompi, M. Ghafoorian, J. A. Van Der Laak, B. Van Ginneken, and C. I. Sánchez, "A survey on deep learning in medical image analysis," *Medical image analysis*, vol. 42, pp. 60–88, 2017.

[10] S. Saha, "A comprehensive guide to convolutional neural networks — the eli5 way." [Online]. Available: https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53

[11] M. Jay, "Deep learning concepts - part 1," 2017. [Online]. Available: https://github.com/markjay4k/Deep-Learning-Concepts/blob/master/part1%20-%20Convolution.pdf

[12] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[13] D. Yu, H. Wang, P. Chen, and Z. Wei, "Mixed pooling for convolutional neural networks," in *RSKT*, 2014.

[14] J. Shubham, "What exactly does cnn see?" 2018. [Online]. Available: https://becominghuman.ai/what-exactly-does-cnn-see-4d436d8e6e52

[15] S. Mallick, "Neural networks : A 30,000 feet view for beginners," https://learnopencv.com/neural-networks-a-30000-feet-view-for-beginners/ [Accessed: 06.06.20].

[16] tzutalin, "Labelimg." [Online]. Available: "https://github.com/tzutalin/labelImg"

[17] S. H. Sumit, "Drawbacks of convolutional neural networks." [Online]. Available: https://sakhawathsumit.github.io/sumit.log/2018/07/21/drawbacks-of-convolutional-neural-networks.html

[18] A. Persson. [Online]. Available: https://www.youtube.com/watch?v=YDkjWEN8jNA&ab_channel=AladdinPersson

[19] G. D. Team. [Online]. Available: https://developers.google.com/machine-learning/crash-course/classification/true-false-positive-negative

[20] K. He, G. Gkioxari, P. Dollár, and R. Girshick, "Mask r-cnn," 2018.

[21] W. Abdulla, "Splash of color: Instance segmentation with mask r-cnn and tensorflow." [Online]. Available: https://engineering.matterport.com/splash-of-color-instance-segmentation-with-mask-r-cnn-and-tensorflow-7c761e238b46

[22] S. Ren, K. He, R. Girshick, and J. Sun, "Faster r-cnn: Towards real-time object detection with region proposal networks," 2016.

[23] Sparrows group secures five-year deal with adma-opco. [Online]. Available: https://www.worldoil.com/news/2015/9/15/sparrows-group-secures-five-year-deal-with-adma-opco

[24] Offshore crane operator – stage 3 training. [Online]. Available: https://coreqhse.com/calendar/lifting/offshore-crane-operator-stage-3-training-6/

[25] D. Sandaruwan, N. Kodikara, C. Ceppitiyagama, R. Rosa, K. Dias, P. Samarasinghe, U. Soysa, and C. Perera, "Virtual learning and training environment for maritime education (vlteme)," 12 2009.

[26] H. Pan, N. Wang, and Y. Qin, "A closed-form solution to eye-to-hand calibration towards visual grasping," *Ind. Robot*, vol. 41, pp. 567–574, 2014.

[27] (2019) Zivid one+ technical specification. [Online]. Available: https://www.zivid.com/hubfs/files/SPEC/Zivid%20One%20Plus%20Datasheet.pdf?hsCtaTracking=f6b119a4-7444-47ac-83a3-44d4840aa429%7Ccabb4557-32f2-4851-9612-99fe7552235b

[28] Shi-tomasi corner detector & good features to track. [Online]. Available: https://docs.opencv.org/master/d4/d8c/tutorial_py_shi_tomasi.html

[29] Camera calibration. [Online]. Available: https://docs.opencv.org/master/dc/dbb/tutorial_py_calibration.html

[30] D. Bolya, C. Zhou, F. Xiao, and Y. J. Lee, "Yolact++: Better real-time instance segmentation," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, p. 1–1, 2020. [Online]. Available: http://dx.doi.org/10.1109/TPAMI.2020.3014297

[31] T. Engedal, "Optilift motion reporter as seen on the crane monitor," https://vimeo.com/258928996 [Accessed: 03.12.20].

[32] "Optilift assistance systems," https://optilift.no/#assistance-systems [Accessed: 03.12.20].

[33] "http://intsite.ai/".

[34] O. E. Hans Kristian Holen, "Tracking of a ship deck using vanishing points and factor graph," 2019-2020.

[35] A. Gautam, S. P.B, and S. Saripalli, "A survey of autonomous landing techniques for uavs," 05 2014, pp. 1210–1218.

[36] M. A. Fischler and R. C. Bolles, "Random sample consensus: A paradigm for model fitting with applications to image analysis and automated cartography," *Commun. ACM*, vol. 24, no. 6, p. 381–395, Jun. 1981. [Online]. Available: https://doi.org/10.1145/358669.358692

[37] 2020. [Online]. Available: https://docs.opencv.org/master/d9/d0c/group__calib3d.html#ga549c2075fac14829ff4a58bc931c033d

[38] Z. Zhang, "A flexible new technique for camera calibration," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 22, no. 11, pp. 1330–1334, 2000.

[39] J.-Y. BOUGUET, "Camera calibration toolbox for matlab," *http://www.vision.caltech.edu/bouguetj/calib_doc/index.html*, 2004. [Online]. Available: https://ci.nii.ac.jp/naid/10019641632/en/

[40] R. Y. Zhong, X. Xu, E. Klotz, and S. T. Newman, "Intelligent manufacturing in the context of industry 4.0: A review," *Engineering*, vol. 3, no. 5, pp. 616 – 630, 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S2095809917307130

[41] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2012, pp. 1097–1105. [Online]. Available: http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf

[42] X. Zhou, W. Gong, W. Fu, and F. Du, "Application of deep learning in object detection," in *2017 IEEE/ACIS 16th International Conference on Computer and Information Science (ICIS)*, 2017, pp. 631–634.

[43] S. Khan, H. Rahmani, S. A. A. Shah, and M. Bennamoun, "A guide to convolutional neural networks for computer vision," *Synthesis Lectures on Computer Vision*, vol. 8, no. 1, pp. 1–207, 2018.

[44] "Behind the scenes - detroit: Become human (motion capture)," 2018, https://www.youtube.com/watch?v=8136CxSh5Ps&ab_channel=TeddyKGaming [Accessed: 30.11.20].

[45] S. C. Babu, “A 2019 guide to human pose estimation with deep learning,” 2019. [Online]. Available: https://nanonets.com/blog/human-pose-estimation-2d-guide/

[46] D. GUPTA, “Fundamentals of deep learning – activation functions and when to use them?” 2020. [Online]. Available: https://www.analyticsvidhya.com/blog/2020/01/fundamentals-deep-learning-activation-functions-when-to-use-them/

[47] S. Tiwari, “Activation functions in neural networks.” [Online]. Available: https://www.geeksforgeeks.org/activation-functions-neural-networks/

[48] S. Asiri, “Machine learning classifiers,” 2018. [Online]. Available: https://towardsdatascience.com/machine-learning-classifiers-a5cc4e1b0623

[49] J. Brownlee, “How to perform object detection with yolov3 in keras,” 2019. [Online]. Available: https://machinelearningmastery.com/how-to-perform-object-detection-with-yolov3-in-keras/

[50] N. Donges, “4 reasons why deep learning and neural networks aren’t always the right choice,” 2020. [Online]. Available: https://builtin.com/data-science/disadvantages-neural-networks

[51] “Underfitting in a neural network explained.” [Online]. Available: https://deeplizard.com/learn/video/0h8lAm5Ki5g

[52] “Overfitting in a neural network explained.” [Online]. Available: https://deeplizard.com/learn/video/DEMmkFC6IGM

[53] J. S. Brad Dwyer, “Glossary of common computer vision terms,” 2020. [Online]. Available: https://blog.roboflow.com/yolov5-improvements-and-evaluation/

[54] S. Ren, K. He, R. B. Girshick, and J. Sun, “Faster R-CNN: towards real-time object detection with region proposal networks,” *CoRR*, vol. abs/1506.01497, 2015. [Online]. Available: http://arxiv.org/abs/1506.01497

[55] Datasheet to lidar-lite v3hp. [Online]. Available: https://www.elfadistrelec.no/Web/Downloads/_m/an/SEN-14599_eng_man.pdf

[56] Intel® realsensetm product family d400 series. [Online]. Available: https://www.intelrealsense.com/wp-content/uploads/2020/06/Intel-RealSense-D400-Series-Datasheet-June-2020.pdf

[57] Y. Wu, A. Kirillov, F. Massa, W.-Y. Lo, and R. Girshick, “Detectron2,” https://github.com/facebookresearch/detectron2, 2019.

[58] G. Tanner, “Detectron2 train a custom instance segmentation model,” https://github.com/TannerGilbert/Detectron2-Train-a-Instance-Segmentation-Model.

[59] T. Fiorenzani, “how_do_drones_work/opencv,” https://github.com/tizianofiorenzani/how_do_drones_work/blob/master/opencv/cameracalib.py, 2019.

[60] Optical flow. [Online]. Available: https://docs.opencv.org/3.4/d4/dee/tutorial_optical_flow.html

Appendix

A Detectron2 Installation

For this project, Detectron2 was used as a platform to train the Mask R-CNN.

"Detectron2 is Facebook AI Research's next generation software system that implements state-of-the-art object detection algorithms. It is a ground-up rewrite of the previous version, Detectron, and it originates from maskrcnn-benchmark."[57]

In order to start out with detectron2 on your local computer, it is highly recommended to start by creating a virtual environment.

In this project, anaconda was used. It enables one to create a virtual environment where different packages/dependencies can be installed without affecting the global environment on your personal computer. This way, in case of the installation of your packages crashes, your computer will not fail on a global- level, such as one is required to reboot your computer. Worst case scenario with conda installment, your virtual environment is ruined, and you can create a new environment with the terminal command "conda create [insert name of venv]". It should be noted that for my personal computer, some packages were unable to install with the <conda install> command in the anaconda terminal. A workaround was to use <pip install>, but it should be noted that pip installing is a python installation and will affect other python tasks. For the most part, this will be harmless, but in some instances, packages may be broken due to other dependencies etc. Keeping them in isolated environments should be the preferred method.

After a conda environment is created, the installation procedure could start. Some of the packages and other software have dependencies between each other, so if a package's version is installed, one needs to install a package that is compatible with this specific version. Other packages are compatible with certain hardware. So, the first types of software that were installed were the ones that are hardware dependant due to the difficulty of working around these.

First, identify what hardware and OS one is using. In this project, OMEN 25L Desktop GT11-0829no was used. Software must be compatible with the Graphics card and OS. The reason for this is that the python scripts for training are using CUDA devices for multiprocessing data. For instance, this computer uses Windows 10 with x86_64 architecture with Nvidia RTX3070. One needs to install a CUDA driver compatible with this on the NVIDIA homepage for CUDA software. Along with this, Visual Studio has a C++ compiler that supports CUDA, so the corresponding needs to be downloaded with it. Please follow the instruction found at NVIDIA.

Next step was to install the Pytorch + dependencies that corresponded with the CUDA toolkit that was previously installed.

The start was initially promising, but some dependencies that were listed in the gettingstarted.md at Detectron2 GitHub [57] may fail for you since because of OS and/or hardware incompatibility. It was solved with a trial and error approach. The final list over the virtual environment can be found in Appendix (C).

It was attempted to use Virtual Machine (VM), but the problem is that the VM will not have 100% access to the GPU since it shares hardware with the host operating system(OS). If one wants to use a different OS, dual booting is advised. Dual boot with RTX3000 series GPU ran into errors as of February 2021, and it did not work. Therefore, against the advice in the GitHub [57], Windows 10 as OS was utilized with no dual booting setup.

B Visualizer

```python
# Copyright (c) Facebook, Inc. and its affiliates. All Rights Reserved
#Detectron2 is released under Apache2.0 license.
import colorsys
import logging
import math
import numpy as np
from enum import Enum, unique
import cv2
import matplotlib as mpl
import matplotlib.colors as mplc
import matplotlib.figure as mplfigure
import pycocotools.mask as mask_util
import torch
from fvcore.common.file_io import PathManager
from matplotlib.backends.backend_agg import FigureCanvasAgg
from PIL import Image

from detectron2.data import MetadataCatalog
from detectron2.structures import BitMasks, Boxes, BoxMode, Keypoints, PolygonMasks, RotatedBoxes


from detectron2.utils.colormap import random_color

logger = logging.getLogger(__name__)

__all__ = ["ColorMode", "VisImage", "Visualizer"]


_SMALL_OBJECT_AREA_THRESH = 1000
_LARGE_MASK_AREA_THRESH = 120000
_OFF_WHITE = (1.0, 1.0, 240.0 / 255)
_BLACK = (0, 0, 0)
_RED = (1.0, 0, 0)

_KEYPOINT_THRESHOLD = 0.05


@unique
class ColorMode(Enum):
    """
    Enum of different color modes to use for instance visualizations.
    """

    IMAGE = 0
    """
```

```
    Picks a random color for every instance and overlay segmentations with low opacity.
    """
    SEGMENTATION = 1
    """
    Let instances of the same category have similar colors
    (from metadata.thing_colors), and overlay them with
    high opacity. This provides more attention on the quality of segmentation.
    """
    IMAGE_BW = 2
    """
    Same as IMAGE, but convert all areas without masks to gray-scale.
    Only available for drawing per-instance mask predictions.
    """


class GenericMask:
    """
    Attribute:
        polygons (list[ndarray]): list[ndarray]: polygons for this mask.
            Each ndarray has format [x, y, x, y, ...]
        mask (ndarray): a binary mask
    """

    def __init__(self, mask_or_polygons, height, width):
        self._mask = self._polygons = self._has_holes = None
        self.height = height
        self.width = width

        m = mask_or_polygons
        if isinstance(m, dict):
            # RLEs
            assert "counts" in m and "size" in m
            if isinstance(m["counts"], list):  # uncompressed RLEs
                h, w = m["size"]
                assert h == height and w == width
                m = mask_util.frPyObjects(m, h, w)
            self._mask = mask_util.decode(m)[:, :]
            return

        if isinstance(m, list):  # list[ndarray]
            self._polygons = [np.asarray(x).reshape(-1) for x in m]
            return

        if isinstance(m, np.ndarray):  # assumed to be a binary mask
            assert m.shape[1] != 2, m.shape
            assert m.shape == (height, width), m.shape
            self._mask = m.astype("uint8")
```

```python
            return

        raise ValueError("GenericMask cannot handle object {} of type '{}'".format(m, type(m)))

    @property
    def mask(self):
        if self._mask is None:
            self._mask = self.polygons_to_mask(self._polygons)
        return self._mask

    @property
    def polygons(self):
        if self._polygons is None:
            self._polygons, self._has_holes = self.mask_to_polygons(self._mask)
        return self._polygons

    @property
    def has_holes(self):
        if self._has_holes is None:
            if self._mask is not None:
                self._polygons, self._has_holes = self.mask_to_polygons(self._mask)
            else:
                self._has_holes = False  # if original format is polygon, does not have holes
        return self._has_holes

    def mask_to_polygons(self, mask):
        # cv2.RETR_CCOMP flag retrieves all the contours and arranges them to a 2-level
        # hierarchy. External contours (boundary) of the object are placed in hierarchy-1.
        # Internal contours (holes) are placed in hierarchy-2.
        # cv2.CHAIN_APPROX_NONE flag gets vertices of polygons from contours.
        mask = np.ascontiguousarray(mask)  # some versions of cv2 does not support incontiguous arr
        res = cv2.findContours(mask.astype("uint8"), cv2.RETR_CCOMP, cv2.CHAIN_APPROX_NONE)
        hierarchy = res[-1]
        if hierarchy is None:  # empty mask
            return [], False
        has_holes = (hierarchy.reshape(-1, 4)[:, 3] >= 0).sum() > 0
        res = res[-2]
        res = [x.flatten() for x in res]
        res = [x for x in res if len(x) >= 6]
        return res, has_holes

```

```python
    def polygons_to_mask(self, polygons):
        rle = mask_util.frPyObjects(polygons, self.height, self.width)
        rle = mask_util.merge(rle)
        return mask_util.decode(rle)[:, :]

    def area(self):
        return self.mask.sum()

    def bbox(self):
        p = mask_util.frPyObjects(self.polygons, self.height,
        ↪  self.width)
        p = mask_util.merge(p)
        bbox = mask_util.toBbox(p)
        bbox[2] += bbox[0]
        bbox[3] += bbox[1]
        return bbox


class _PanopticPrediction:
    def __init__(self, panoptic_seg, segments_info):
        self._seg = panoptic_seg

        self._sinfo = {s["id"]: s for s in segments_info}  # seg id ->
        ↪  seg info
        segment_ids, areas = torch.unique(panoptic_seg, sorted=True,
        ↪  return_counts=True)
        areas = areas.numpy()
        sorted_idxs = np.argsort(-areas)
        self._seg_ids, self._seg_areas = segment_ids[sorted_idxs],
        ↪  areas[sorted_idxs]
        self._seg_ids = self._seg_ids.tolist()
        for sid, area in zip(self._seg_ids, self._seg_areas):
            if sid in self._sinfo:
                self._sinfo[sid]["area"] = float(area)

    def non_empty_mask(self):
        """
        Returns:
            (H, W) array, a mask for all pixels that have a prediction
        """
        empty_ids = []
        for id in self._seg_ids:
            if id not in self._sinfo:
                empty_ids.append(id)
        if len(empty_ids) == 0:
            return np.zeros(self._seg.shape, dtype=np.uint8)
        assert (
            len(empty_ids) == 1
        ), ">1 ids corresponds to no labels. This is currently not
        ↪  supported"
```

```
        return (self._seg != empty_ids[0]).numpy().astype(np.bool)

    def semantic_masks(self):
        for sid in self._seg_ids:
            sinfo = self._sinfo.get(sid)
            if sinfo is None or sinfo["isthing"]:
                # Some pixels (e.g. id 0 in PanopticFPN) have no
                ↪ instance or semantic predictions.
                continue
            yield (self._seg == sid).numpy().astype(np.bool), sinfo

    def instance_masks(self):
        for sid in self._seg_ids:
            sinfo = self._sinfo.get(sid)
            if sinfo is None or not sinfo["isthing"]:
                continue
            mask = (self._seg == sid).numpy().astype(np.bool)
            if mask.sum() > 0:
                yield mask, sinfo


def _create_text_labels(classes, scores, class_names):
    """
    Args:
        classes (list[int] or None):
        scores (list[float] or None):
        class_names (list[str] or None):

    Returns:
        list[str] or None
    """
    labels = None
    if classes is not None and class_names is not None and
    ↪ len(class_names) > 0:
        labels = [class_names[i] for i in classes]
    if scores is not None:
        if labels is None:
            labels = ["{:.0f}%".format(s * 100) for s in scores]
        else:
            labels = ["{} {:.0f}%".format(l, s * 100) for l, s in
            ↪ zip(labels, scores)]
    return labels


class VisImage:
    def __init__(self, img, scale=1.0):
        """
        Args:
            img (ndarray): an RGB image of shape (H, W, 3).
            scale (float): scale the input image
```

```python
        """
        self.img = img
        self.scale = scale
        self.width, self.height = img.shape[1], img.shape[0]
        self._setup_figure(img)

    def _setup_figure(self, img):
        """
        Args:
            Same as in :meth:`__init__()`.

        Returns:
            fig (matplotlib.pyplot.figure): top level container for all the image plot elements.
            ax (matplotlib.pyplot.Axes): contains figure elements and sets the coordinate system.
        """
        fig = mplfigure.Figure(frameon=False)
        self.dpi = fig.get_dpi()
        # add a small 1e-2 to avoid precision lost due to matplotlib's truncation
        # (https://github.com/matplotlib/matplotlib/issues/15363)
        fig.set_size_inches(
            (self.width * self.scale + 1e-2) / self.dpi,
            (self.height * self.scale + 1e-2) / self.dpi,
        )
        self.canvas = FigureCanvasAgg(fig)
        # self.canvas = mpl.backends.backend_cairo.FigureCanvasCairo(fig)
        ax = fig.add_axes([0.0, 0.0, 1.0, 1.0])
        ax.axis("off")
        ax.set_xlim(0.0, self.width)
        ax.set_ylim(self.height)

        self.fig = fig
        self.ax = ax

    def save(self, filepath):
        """
        Args:
            filepath (str): a string that contains the absolute path, including the file name, where
                the visualized image will be saved.
        """
        if filepath.lower().endswith(".jpg") or filepath.lower().endswith(".png"):
            # faster than matplotlib's imshow
            cv2.imwrite(filepath, self.get_image()[:, :, ::-1])
        else:
            # support general formats (e.g. pdf)
```

```
            self.ax.imshow(self.img, interpolation="nearest")
            self.fig.savefig(filepath)

    def get_image(self):
        """
        Returns:
            ndarray:
                the visualized image of shape (H, W, 3) (RGB) in uint8 type.
                The shape is scaled w.r.t the input image using the given `scale` argument.
        """
        canvas = self.canvas
        s, (width, height) = canvas.print_to_buffer()
        if (self.width, self.height) != (width, height):
            img = cv2.resize(self.img, (width, height))
        else:
            img = self.img

        # buf = io.BytesIO()  # works for cairo backend
        # canvas.print_rgba(buf)
        # width, height = self.width, self.height
        # s = buf.getvalue()

        buffer = np.frombuffer(s, dtype="uint8")

        # imshow is slow. blend manually (still quite slow)
        img_rgba = buffer.reshape(height, width, 4)
        rgb, alpha = np.split(img_rgba, [3], axis=2)

        try:
            import numexpr as ne  # fuse them with numexpr

            visualized_image = ne.evaluate("img * (1 - alpha / 255.0) + rgb * (alpha / 255.0)")
        except ImportError:
            alpha = alpha.astype("float32") / 255.0
            visualized_image = img * (1 - alpha) + rgb * alpha

        visualized_image = visualized_image.astype("uint8")

        return visualized_image


class Visualizer:
    """
    Visualizer that draws data about detection/segmentation on images.

    It contains methods like `draw_{text,box,circle,line,binary_mask,polygon}`
```

```
    that draw primitive objects to images, as well as high-level wrappers like
    `draw_{instance_predictions,sem_seg,panoptic_seg_predictions,dataset_dict}`
    that draw composite data in some pre-defined style.

    Note that the exact visualization style for the high-level wrappers are subject to change.
    Style such as color, opacity, label contents, visibility of labels, or even the visibility
    of objects themselves (e.g. when the object is too small) may change according
    to different heuristics, as long as the results still look visually reasonable.
    To obtain a consistent style, implement custom drawing functions with the primitive
    methods instead.

    This visualizer focuses on high rendering quality rather than performance. It is not
    designed to be used for real-time applications.
    """

    def __init__(self, img_rgb, metadata=None, scale=1.0, instance_mode=ColorMode.IMAGE):
        """
        Args:
            img_rgb: a numpy array of shape (H, W, C), where H and W correspond to
                the height and width of the image respectively. C is the number of
                color channels. The image is required to be in RGB format since that
                is a requirement of the Matplotlib library. The image is also expected
                to be in the range [0, 255].
            metadata (MetadataCatalog): image metadata.
            instance_mode (ColorMode): defines one of the pre-defined style for drawing
                instances on an image.
        """
        self.img = np.asarray(img_rgb).clip(0, 255).astype(np.uint8)
        if metadata is None:
            metadata = MetadataCatalog.get("__nonexist__")
        self.metadata = metadata
        self.output = VisImage(self.img, scale=scale)
        self.cpu_device = torch.device("cpu")

        # too small texts are useless, therefore clamp to 9
        self._default_font_size = max(
```

```
            np.sqrt(self.output.height * self.output.width) // 90, 10 // scale
        )
        self._instance_mode = instance_mode

    def draw_instance_predictions(self, predictions):
        """
        Draw instance-level prediction results on an image.

        Args:
            predictions (Instances): the output of an instance detection/segmentation
                model. Following fields will be used to draw:
                "pred_boxes", "pred_classes", "scores", "pred_masks" (or "pred_masks_rle").

        Returns:
            output (VisImage): image object with visualizations.
        """
        boxes = predictions.pred_boxes if predictions.has("pred_boxes") else None
        scores = predictions.scores if predictions.has("scores") else None
        classes = predictions.pred_classes if predictions.has("pred_classes") else None
        labels = _create_text_labels(classes, scores, self.metadata.get("thing_classes", None))
        keypoints = predictions.pred_keypoints if predictions.has("pred_keypoints") else None

        if predictions.has("pred_masks"):
            masks = np.asarray(predictions.pred_masks)
            masks = [GenericMask(x, self.output.height, self.output.width) for x in masks]
        else:
            masks = None

        if self._instance_mode == ColorMode.SEGMENTATION and self.metadata.get("thing_colors"):
            colors = [
                self._jitter([x / 255 for x in self.metadata.thing_colors[c]]) for c in classes
            ]
            alpha = 1.0  # her skriver man opacity til masken og dens innhold. original er alpha = 0.8, men endret den til 1.0
        else:
            colors = None
            alpha = 1

```

```
        if self._instance_mode == ColorMode.IMAGE_BW:
            self.output.img = self._change_color_brightness(color=_BLACK, brightness_factor=0)

            alpha = 1.0  # her skriver man opacity inne i masken. originalt er 0.3

        self.overlay_instances(
            masks=masks,
            #boxes=boxes,
            #labels=labels,
            keypoints=keypoints,
            assigned_colors=colors,
            alpha=alpha,
        )
        return self.output

    def draw_sem_seg(self, sem_seg, area_threshold=None, alpha=0.8):
        """
        Draw semantic segmentation predictions/labels.

        Args:
            sem_seg (Tensor or ndarray): the segmentation of shape (H, W).
                Each value is the integer label of the pixel.
            area_threshold (int): segments with less than `area_threshold` are not drawn.
            alpha (float): the larger it is, the more opaque the segmentations are.

        Returns:
            output (VisImage): image object with visualizations.
        """
        if isinstance(sem_seg, torch.Tensor):
            sem_seg = sem_seg.numpy()
        labels, areas = np.unique(sem_seg, return_counts=True)
        sorted_idxs = np.argsort(-areas).tolist()
        labels = labels[sorted_idxs]
        for label in filter(lambda l: l < len(self.metadata.stuff_classes), labels):
            try:
                mask_color = [x / 255 for x in self.metadata.stuff_colors[label]]
            except (AttributeError, IndexError):
                mask_color = None

            binary_mask = (sem_seg == label).astype(np.uint8)
            text = self.metadata.stuff_classes[label]
            self.draw_binary_mask(
                binary_mask,
```

```
                color=mask_color,
                edge_color=_OFF_WHITE,
                text=text,
                alpha=alpha,
                area_threshold=area_threshold,
            )
        return self.output

    def draw_panoptic_seg_predictions(
        self, panoptic_seg, segments_info, area_threshold=None, alpha=0.7
    ):
        """
        Draw panoptic prediction results on an image.

        Args:
            panoptic_seg (Tensor): of shape (height, width) where the values are ids for each
                segment.
            segments_info (list[dict]): Describe each segment in `panoptic_seg`.
                Each dict contains keys "id", "category_id", "isthing".
            area_threshold (int): stuff segments with less than `area_threshold` are not drawn.

        Returns:
            output (VisImage): image object with visualizations.
        """
        pred = _PanopticPrediction(panoptic_seg, segments_info)

        if self._instance_mode == ColorMode.IMAGE_BW:
            self.output.img = self._create_grayscale_image(pred.non_empty_mask())

        # draw mask for all semantic segments first i.e. "stuff"
        for mask, sinfo in pred.semantic_masks():
            category_idx = sinfo["category_id"]
            try:
                mask_color = [x / 255 for x in self.metadata.stuff_colors[category_idx]]
            except AttributeError:
                mask_color = None

            text = self.metadata.stuff_classes[category_idx]
            self.draw_binary_mask(
                mask,
                color=mask_color,
                edge_color=_OFF_WHITE,
                text=text,
```

```
                alpha=alpha,
                area_threshold=area_threshold,
            )

        # draw mask for all instances second
        all_instances = list(pred.instance_masks())
        if len(all_instances) == 0:
            return self.output
        masks, sinfo = list(zip(*all_instances))
        category_ids = [x["category_id"] for x in sinfo]

        try:
            scores = [x["score"] for x in sinfo]
        except KeyError:
            scores = None
        labels = _create_text_labels(category_ids, scores, self.metadata.thing_classes)

        try:
            colors = [random_color(rgb=True, maximum=1) for k in category_ids]
        except AttributeError:
            colors = None
        self.overlay_instances(masks=masks, labels=labels, assigned_colors=colors, alpha=alpha)

        return self.output

    def draw_dataset_dict(self, dic):
        """
        Draw annotations/segmentaions in Detectron2 Dataset format.

        Args:
            dic (dict): annotation/segmentation data of one image, in Detectron2 Dataset format.

        Returns:
            output (VisImage): image object with visualizations.
        """
        annos = dic.get("annotations", None)
        if annos:
            if "segmentation" in annos[0]:
                masks = [x["segmentation"] for x in annos]
            else:
                masks = None
            if "keypoints" in annos[0]:
                keypts = [x["keypoints"] for x in annos]
                keypts = np.array(keypts).reshape(len(annos), -1, 3)
            else:
                keypts = None
```

```python

            boxes = [BoxMode.convert(x["bbox"], x["bbox_mode"], BoxMode.XYXY_ABS) for x in annos]

            labels = [x["category_id"] for x in annos]
            colors = None
            if self._instance_mode == ColorMode.SEGMENTATION and self.metadata.get("thing_colors"):
                colors = [
                    self._jitter([x / 255 for x in self.metadata.thing_colors[c]]) for c in labels
                ]
            names = self.metadata.get("thing_classes", None)
            if names:
                labels = [names[i] for i in labels]
            labels = [
                "{}".format(i) + ("|crowd" if a.get("iscrowd", 0) else "")
                for i, a in zip(labels, annos)
            ]
            self.overlay_instances(
                labels=labels, boxes=boxes, masks=masks, keypoints=keypts, assigned_colors=colors
            )

        sem_seg = dic.get("sem_seg", None)
        if sem_seg is None and "sem_seg_file_name" in dic:
            with PathManager.open(dic["sem_seg_file_name"], "rb") as f:
                sem_seg = Image.open(f)
                sem_seg = np.asarray(sem_seg, dtype="uint8")
        if sem_seg is not None:
            self.draw_sem_seg(sem_seg, area_threshold=0, alpha=0.5)
        return self.output

    def overlay_instances(
        self,
        *,
        boxes=None,
        labels=None,
        masks=None,
        keypoints=None,
        assigned_colors=None,
        alpha=0.5
    ):
        """
        Args:
            boxes (Boxes, RotatedBoxes or ndarray): either a :class:`Boxes`,
                or an Nx4 numpy array of XYXY_ABS format for the N objects in a single image,
```

```
                or a :class:`RotatedBoxes`,
                or an Nx5 numpy array of (x_center, y_center, width, height, angle_degrees) format
                for the N objects in a single image,
            labels (list[str]): the text to be displayed for each instance.
            masks (masks-like object): Supported types are:

                * :class:`detectron2.structures.PolygonMasks`,
                  :class:`detectron2.structures.BitMasks`.
                * list[list[ndarray]]: contains the segmentation masks for all objects in one image.
                  The first level of the list corresponds to individual instances. The second
                  level to all the polygon that compose the instance, and the third level
                  to the polygon coordinates. The third level should have the format of
                  [x0, y0, x1, y1, ..., xn, yn] (n >= 3).
                * list[ndarray]: each ndarray is a binary mask of shape (H, W).
                * list[dict]: each dict is a COCO-style RLE.
            keypoints (Keypoint or array like): an array-like object of shape (N, K, 3),
                where the N is the number of instances and K is the number of keypoints.
                The last dimension corresponds to (x, y, visibility or score).
            assigned_colors (list[matplotlib.colors]): a list of colors, where each color
                corresponds to each mask or box in the image. Refer to 'matplotlib.colors'
                for full list of formats that the colors are accepted in.

        Returns:
            output (VisImage): image object with visualizations.
        """
        num_instances = None
        if boxes is not None:
            boxes = self._convert_boxes(boxes)
            num_instances = len(boxes)
        if masks is not None:
            masks = self._convert_masks(masks)
            if num_instances:
                assert len(masks) == num_instances
            else:
                num_instances = len(masks)
        if keypoints is not None:
            if num_instances:
```

```
                assert len(keypoints) == num_instances
            else:
                num_instances = len(keypoints)
            keypoints = self._convert_keypoints(keypoints)
        if labels is not None:
            assert len(labels) == num_instances
        if assigned_colors is None:
            assigned_colors = [random_color(rgb=True, maximum=1) for _ in range(num_instances)]
        if num_instances == 0:
            return self.output
        if boxes is not None and boxes.shape[1] == 5:
            return self.overlay_rotated_instances(
                boxes=boxes, labels=labels, assigned_colors=assigned_colors
            )

        # Display in largest to smallest order to reduce occlusion.
        areas = None
        if boxes is not None:
            areas = np.prod(boxes[:, 2:] - boxes[:, :2], axis=1)
        elif masks is not None:
            areas = np.asarray([x.area() for x in masks])

        if areas is not None:
            sorted_idxs = np.argsort(-areas).tolist()
            # Re-order overlapped instances in descending order.
            boxes = boxes[sorted_idxs] if boxes is not None else None
            labels = [labels[k] for k in sorted_idxs] if labels is not None else None
            masks = [masks[idx] for idx in sorted_idxs] if masks is not None else None
            assigned_colors = [assigned_colors[idx] for idx in sorted_idxs]
            keypoints = keypoints[sorted_idxs] if keypoints is not None else None

        for i in range(num_instances):
            color = assigned_colors[i]
            if boxes is not None:
                self.draw_box(boxes[i], edge_color=color)

            if masks is not None:
                for segment in masks[i].polygons:
                    self.draw_polygon(segment.reshape(-1, 2), color, alpha=alpha)

            if labels is not None:
                # first get a box
                if boxes is not None:
```

```
                x0, y0, x1, y1 = boxes[i]
                text_pos = (x0, y0)  # if drawing boxes, put text on the box corner.
                horiz_align = "left"
            elif masks is not None:
                # skip small mask without polygon
                if len(masks[i].polygons) == 0:
                    continue

                x0, y0, x1, y1 = masks[i].bbox()

                # draw text in the center (defined by median) when box is not drawn
                # median is less sensitive to outliers.
                text_pos = np.median(masks[i].mask.nonzero(), axis=1)[::-1]
                horiz_align = "center"
            else:
                continue  # drawing the box confidence for keypoints isn't very useful.
            # for small objects, draw text at the side to avoid occlusion
            instance_area = (y1 - y0) * (x1 - x0)
            if (
                instance_area < _SMALL_OBJECT_AREA_THRESH * self.output.scale
                or y1 - y0 < 40 * self.output.scale
            ):
                if y1 >= self.output.height - 5:
                    text_pos = (x1, y0)
                else:
                    text_pos = (x0, y1)

            height_ratio = (y1 - y0) / np.sqrt(self.output.height * self.output.width)
            lighter_color = self._change_color_brightness(color, brightness_factor=0.7)
            font_size = (
                np.clip((height_ratio - 0.02) / 0.08 + 1, 1.2, 2)
                * 0.5
                * self._default_font_size
            )
            self.draw_text(
                labels[i],
                text_pos,
                color=lighter_color,
                horizontal_alignment=horiz_align,
                font_size=font_size,
            )

```

```python
        # draw keypoints
        if keypoints is not None:
            for keypoints_per_instance in keypoints:
                self.draw_and_connect_keypoints(keypoints_per_instance)

        return self.output

    def overlay_rotated_instances(self, boxes=None, labels=None, assigned_colors=None):
        """
        Args:
            boxes (ndarray): an Nx5 numpy array of
                (x_center, y_center, width, height, angle_degrees) format
                for the N objects in a single image.
            labels (list[str]): the text to be displayed for each instance.
            assigned_colors (list[matplotlib.colors]): a list of colors, where each color
                corresponds to each mask or box in the image. Refer to 'matplotlib.colors'
                for full list of formats that the colors are accepted in.

        Returns:
            output (VisImage): image object with visualizations.
        """
        num_instances = len(boxes)

        if assigned_colors is None:
            assigned_colors = [random_color(rgb=True, maximum=1) for _ in range(num_instances)]
        if num_instances == 0:
            return self.output

        # Display in largest to smallest order to reduce occlusion.
        if boxes is not None:
            areas = boxes[:, 2] * boxes[:, 3]

        sorted_idxs = np.argsort(-areas).tolist()
        # Re-order overlapped instances in descending order.
        boxes = boxes[sorted_idxs]
        labels = [labels[k] for k in sorted_idxs] if labels is not None else None
        colors = [assigned_colors[idx] for idx in sorted_idxs]

        for i in range(num_instances):
            self.draw_rotated_box_with_label(
                boxes[i], edge_color=colors[i], label=labels[i] if labels is not None else None
```

```
            )

        return self.output

    def draw_and_connect_keypoints(self, keypoints):
        """
        Draws keypoints of an instance and follows the rules for keypoint connections
        to draw lines between appropriate keypoints. This follows color heuristics for
        line color.

        Args:
            keypoints (Tensor): a tensor of shape (K, 3), where K is the number of keypoints
                and the last dimension corresponds to (x, y, probability).

        Returns:
            output (VisImage): image object with visualizations.
        """
        visible = {}
        keypoint_names = self.metadata.get("keypoint_names") #Originale. denne er byttet med den under
        #keypoint_names = self.metadata.get("keypoint_names")  # Originale. denne er byttet med den under
        for idx, keypoint in enumerate(keypoints):
            # draw keypoint
            x, y, prob = keypoint
            if prob > _KEYPOINT_THRESHOLD:
                self.draw_circle((x, y), color=_RED)
                if keypoint_names:
                    keypoint_name = keypoint_names[idx]
                    visible[keypoint_name] = (x, y)

        if self.metadata.get("keypoint_connection_rules"):
            for kp0, kp1, color in self.metadata.keypoint_connection_rules:
                if kp0 in visible and kp1 in visible:
                    x0, y0 = visible[kp0]
                    x1, y1 = visible[kp1]
                    color = tuple(x / 255.0 for x in color)
                    self.draw_line([x0, x1], [y0, y1], color=color)

        # draw lines from nose to mid-shoulder and mid-shoulder to mid-hip
        # Note that this strategy is specific to person keypoints.
        # For other keypoints, it should just do nothing
        try:
            ls_x, ls_y = visible["left_shoulder"]
```

```
            rs_x, rs_y = visible["right_shoulder"]
            mid_shoulder_x, mid_shoulder_y = (ls_x + rs_x) / 2, (ls_y + rs_y) / 2
        except KeyError:
            pass
        else:
            # draw line from nose to mid-shoulder
            nose_x, nose_y = visible.get("nose", (None, None))
            if nose_x is not None:
                self.draw_line([nose_x, mid_shoulder_x], [nose_y, mid_shoulder_y], color=_RED)

            try:
                # draw line from mid-shoulder to mid-hip
                lh_x, lh_y = visible["left_hip"]
                rh_x, rh_y = visible["right_hip"]
            except KeyError:
                pass
            else:
                mid_hip_x, mid_hip_y = (lh_x + rh_x) / 2, (lh_y + rh_y) / 2
                self.draw_line([mid_hip_x, mid_shoulder_x], [mid_hip_y, mid_shoulder_y], color=_RED)
        return self.output

    """
    Primitive drawing functions:
    """

    def draw_text(
        self,
        text,
        position,
        *,
        font_size=None,
        color="g",
        horizontal_alignment="center",
        rotation=0
    ):
        """
        Args:
            text (str): class label
            position (tuple): a tuple of the x and y coordinates to place text on image.
            font_size (int, optional): font of the text. If not provided, a font size
                proportional to the image width is calculated and used.
            color: color of the text. Refer to `matplotlib.colors` for full list
```

```
                for full list of formats that are accepted.
            horizontal_alignment (str): see `matplotlib.text.Text`
            rotation: rotation angle in degrees CCW

        Returns:
            output (VisImage): image object with text drawn.
        """
        if not font_size:
            font_size = self._default_font_size

        # since the text background is dark, we don't want the text to be dark
        color = np.maximum(list(mplc.to_rgb(color)), 0.2)
        color[np.argmax(color)] = max(0.8, np.max(color))

        x, y = position
        self.output.ax.text(
            x,
            y,
            text,
            size=font_size * self.output.scale,
            family="sans-serif",
            bbox={"facecolor": "black", "alpha": 0.8, "pad": 0.7, "edgecolor": "none"},
            verticalalignment="top",
            horizontalalignment=horizontal_alignment,
            color=color,
            zorder=10,
            rotation=rotation,
        )
        return self.output

    def draw_box(self, box_coord, alpha=0.5, edge_color="g", line_style="-"):
        """
        Args:
            box_coord (tuple): a tuple containing x0, y0, x1, y1 coordinates, where x0 and y0
                are the coordinates of the image's top left corner. x1 and y1 are the
                coordinates of the image's bottom right corner.
            alpha (float): blending efficient. Smaller values lead to more transparent masks.
            edge_color: color of the outline of the box. Refer to `matplotlib.colors`
                for full list of formats that are accepted.
            line_style (string): the string to use to create the outline of the boxes.

        Returns:
```

```
            output (VisImage): image object with box drawn.
        """
        x0, y0, x1, y1 = box_coord
        width = x1 - x0
        height = y1 - y0

        linewidth = max(self._default_font_size / 4, 1)

        self.output.ax.add_patch(
            mpl.patches.Rectangle(
                (x0, y0),
                width,
                height,
                fill=False,
                edgecolor=edge_color,
                linewidth=linewidth * self.output.scale,
                alpha=alpha,
                linestyle=line_style,
            )
        )
        return self.output

    def draw_rotated_box_with_label(
        self, rotated_box, alpha=0.5, edge_color="g", line_style="-", label=None
    ):
        """
        Draw a rotated box with label on its top-left corner.

        Args:
            rotated_box (tuple): a tuple containing (cnt_x, cnt_y, w, h, angle),
                where cnt_x and cnt_y are the center coordinates of the box.
                w and h are the width and height of the box. angle represents how
                many degrees the box is rotated CCW with regard to the 0-degree box.
            alpha (float): blending efficient. Smaller values lead to more transparent masks.
            edge_color: color of the outline of the box. Refer to `matplotlib.colors`
                for full list of formats that are accepted.
            line_style (string): the string to use to create the outline of the boxes.
            label (string): label for rotated box. It will not be rendered when set to None.

        Returns:
            output (VisImage): image object with box drawn.
```

```
        """
        cnt_x, cnt_y, w, h, angle = rotated_box
        area = w * h
        # use thinner lines when the box is small
        linewidth = self._default_font_size / (
            6 if area < _SMALL_OBJECT_AREA_THRESH * self.output.scale else 3
        )

        theta = angle * math.pi / 180.0
        c = math.cos(theta)
        s = math.sin(theta)
        rect = [(-w / 2, h / 2), (-w / 2, -h / 2), (w / 2, -h / 2), (w / 2, h / 2)]
        # x: left->right ; y: top->down
        rotated_rect = [(s * yy + c * xx + cnt_x, c * yy - s * xx + cnt_y) for (xx, yy) in rect]
        for k in range(4):
            j = (k + 1) % 4
            self.draw_line(
                [rotated_rect[k][0], rotated_rect[j][0]],
                [rotated_rect[k][1], rotated_rect[j][1]],
                color=edge_color,
                linestyle="--" if k == 1 else line_style,
                linewidth=linewidth,
            )

        if label is not None:
            text_pos = rotated_rect[1]  # topleft corner

            height_ratio = h / np.sqrt(self.output.height * self.output.width)
            label_color = self._change_color_brightness(edge_color, brightness_factor=0.7)
            font_size = (
                np.clip((height_ratio - 0.02) / 0.08 + 1, 1.2, 2) * 0.5 * self._default_font_size
            )
            self.draw_text(label, text_pos, color=label_color, font_size=font_size, rotation=angle)

        return self.output

    def draw_circle(self, circle_coord, color, radius=3):
        """
        Args:
            circle_coord (list(int) or tuple(int)): contains the x and y coordinates
                of the center of the circle.
```

```python
            color: color of the polygon. Refer to `matplotlib.colors` for a full list of
                formats that are accepted.
            radius (int): radius of the circle.

        Returns:
            output (VisImage): image object with box drawn.
        """
        x, y = circle_coord
        self.output.ax.add_patch(
            mpl.patches.Circle(circle_coord, radius=radius, fill=True, color=color)
        )
        return self.output

    def draw_line(self, x_data, y_data, color, linestyle="-", linewidth=None):
        """
        Args:
            x_data (list[int]): a list containing x values of all the points being drawn.
                Length of list should match the length of y_data.
            y_data (list[int]): a list containing y values of all the points being drawn.
                Length of list should match the length of x_data.
            color: color of the line. Refer to `matplotlib.colors` for a full list of
                formats that are accepted.
            linestyle: style of the line. Refer to `matplotlib.lines.Line2D`
                for a full list of formats that are accepted.
            linewidth (float or None): width of the line. When it's None,
                a default value will be computed and used.

        Returns:
            output (VisImage): image object with line drawn.
        """
        if linewidth is None:
            linewidth = self._default_font_size / 3
        linewidth = max(linewidth, 1)
        self.output.ax.add_line(
            mpl.lines.Line2D(
                x_data,
                y_data,
                linewidth=linewidth * self.output.scale,
                color=color,
                linestyle=linestyle,
            )
        )
```

```
        return self.output

    def draw_binary_mask(
        self, binary_mask, color=None, *, edge_color=None, text=None, alpha=0.5, area_threshold=0
    ):
        """
        Args:
            binary_mask (ndarray): numpy array of shape (H, W), where H is the image height and
                W is the image width. Each value in the array is either a 0 or 1 value of uint8
                type.
            color: color of the mask. Refer to `matplotlib.colors` for a full list of
                formats that are accepted. If None, will pick a random color.
            edge_color: color of the polygon edges. Refer to `matplotlib.colors` for a
                full list of formats that are accepted.
            text (str): if None, will be drawn in the object's center of mass.
            alpha (float): blending efficient. Smaller values lead to more transparent masks.
            area_threshold (float): a connected component small than this will not be shown.

        Returns:
            output (VisImage): image object with mask drawn.
        """
        if color is None:
            color = random_color(rgb=True, maximum=1)
        color = mplc.to_rgb(color)

        has_valid_segment = False
        binary_mask = binary_mask.astype("uint8")  # opencv needs uint8
        mask = GenericMask(binary_mask, self.output.height, self.output.width)
        shape2d = (binary_mask.shape[0], binary_mask.shape[1])

        if not mask.has_holes:
            # draw polygons for regular masks
            for segment in mask.polygons:
                area = mask_util.area(mask_util.frPyObjects([segment], shape2d[0], shape2d[1]))
                if area < (area_threshold or 0):
                    continue
                has_valid_segment = True
                segment = segment.reshape(-1, 2)
```

```python
                self.draw_polygon(segment, color=color, edge_color=edge_color, alpha=alpha)
        else:
            rgba = np.zeros(shape2d + (4,), dtype="float32")
            rgba[:, :, :3] = color
            rgba[:, :, 3] = (mask.mask == 1).astype("float32") * alpha
            has_valid_segment = True
            self.output.ax.imshow(rgba)

        if text is not None and has_valid_segment:
            # TODO sometimes drawn on wrong objects. the heuristics here can improve.
            lighter_color = self._change_color_brightness(color, brightness_factor=0.7)
            _num_cc, cc_labels, stats, centroids = cv2.connectedComponentsWithStats(binary_mask, 8)
            largest_component_id = np.argmax(stats[1:, -1]) + 1

            # draw text on the largest component, as well as other very large components.
            for cid in range(1, _num_cc):
                if cid == largest_component_id or stats[cid, -1] > _LARGE_MASK_AREA_THRESH:
                    # median is more stable than centroid
                    # center = centroids[largest_component_id]
                    center = np.median((cc_labels == cid).nonzero(), axis=1)[::-1]
                    self.draw_text(text, center, color=lighter_color)
        return self.output

    def draw_polygon(self, segment, color, edge_color=None, alpha=0.5):
        """
        Args:
            segment: numpy array of shape Nx2, containing all the points in the polygon.
            color: color of the polygon. Refer to `matplotlib.colors` for a full list of
                formats that are accepted.
            edge_color: color of the polygon edges. Refer to `matplotlib.colors` for a
                full list of formats that are accepted. If not provided, a darker shade
                of the polygon color will be used instead.
            alpha (float): blending efficient. Smaller values lead to more transparent masks.

        Returns:
            output (VisImage): image object with polygon drawn.
        """
        if edge_color is None:
```

```
            # make edge color darker than the polygon color
            if alpha > 0.8:
                edge_color = self._change_color_brightness(color, brightness_factor=-0.7)
            else:
                edge_color = color
        edge_color = mplc.to_rgb(edge_color) + (1,)

        polygon = mpl.patches.Polygon(
            segment,
            fill=True,
            facecolor=mplc.to_rgb(color) + (alpha,),
            edgecolor=edge_color,
            linewidth=max(self._default_font_size // 15 * self.output.scale, 1),
        )
        self.output.ax.add_patch(polygon)
        return self.output

    """
    Internal methods:
    """

    def _jitter(self, color):
        """
        Randomly modifies given color to produce a slightly different color than the color given.

        Args:
            color (tuple[double]): a tuple of 3 elements, containing the RGB values of the color
                picked. The values in the list are in the [0.0, 1.0] range.

        Returns:
            jittered_color (tuple[double]): a tuple of 3 elements, containing the RGB values of the
                color after being jittered. The values in the list are in the [0.0, 1.0] range.
        """
        color = mplc.to_rgb(color)
        vec = np.random.rand(3)
        # better to do it in another color space
        vec = vec / np.linalg.norm(vec) * 0.5
        res = np.clip(vec + color, 0, 1)
        return tuple(res)

    def _create_grayscale_image(self, mask=None):
        """
        Create a grayscale version of the original image.
```

```python
        The colors in masked area, if given, will be kept.
        """
        img_bw = self.img.astype("f4").mean(axis=2)
        img_bw = np.stack([img_bw] * 3, axis=2)
        if mask is not None:
            img_bw[mask] = self.img[mask]
        return img_bw

    def _change_color_brightness(self, color, brightness_factor=-1):
        """
        Depending on the brightness_factor, gives a lighter or darker color i.e. a color with
        less or more saturation than the original color.

        Args:
            color: color of the polygon. Refer to `matplotlib.colors` for a full list of
                formats that are accepted.
            brightness_factor (float): a value in [-1.0, 1.0] range. A lightness factor of
                0 will correspond to no change, a factor in [-1.0, 0) range will result in
                a darker color and a factor in (0, 1.0] range will result in a lighter color.

        Returns:
            modified_color (tuple[double]): a tuple containing the RGB values of the
                modified color. Each value in the tuple is in the [0.0, 1.0] range.
        """
        assert brightness_factor >= -1.0 and brightness_factor <= 1.0
        color = mplc.to_rgb(color)
        polygon_color = colorsys.rgb_to_hls(*mplc.to_rgb(color))
        modified_lightness = polygon_color[1] + (brightness_factor * polygon_color[1])
        modified_lightness = 0.0 if modified_lightness < 0.0 else modified_lightness
        modified_lightness = 1.0 if modified_lightness > 1.0 else modified_lightness
        modified_color = colorsys.hls_to_rgb(polygon_color[0], modified_lightness, polygon_color[2])
        return modified_color

    def _convert_boxes(self, boxes):
        """
        Convert different format of boxes to an NxB array, where B = 4 or 5 is the box dimension.
        """
        if isinstance(boxes, Boxes) or isinstance(boxes, RotatedBoxes):
```

```python
            return boxes.tensor.numpy()
        else:
            return np.asarray(boxes)

    def _convert_masks(self, masks_or_polygons):
        """
        Convert different format of masks or polygons to a tuple of
 ↪  masks and polygons.

        Returns:
            list[GenericMask]:
        """

        m = masks_or_polygons
        if isinstance(m, PolygonMasks):
            m = m.polygons
        if isinstance(m, BitMasks):
            m = m.tensor.numpy()
        if isinstance(m, torch.Tensor):
            m = m.numpy()
        ret = []
        for x in m:
            if isinstance(x, GenericMask):
                ret.append(x)
            else:
                ret.append(GenericMask(x, self.output.height,
                 ↪  self.output.width))
        return ret

    def _convert_keypoints(self, keypoints):
        if isinstance(keypoints, Keypoints):
            keypoints = keypoints.tensor
        keypoints = np.asarray(keypoints)
        return keypoints

    def get_output(self):
        """
        Returns:
            output (VisImage): the image output containing the
 ↪  visualizations added
            to the image.
        """
        return self.output
```

C Conda Environment

```
(detectron2) C:\Users\Marti>conda list
# packages in environment at C:\Users\Marti\anaconda3\envs\detectron2:
#
# Name                    Version                   Build  Channel
absl-py                   0.11.0                   pypi_0    pypi
anyio                     2.1.0            py37h03978a9_0    conda-forge
argon2-cffi               20.1.0           py37hcc03f2d_2    conda-forge
async_generator           1.10                       py_0    conda-forge
attrs                     20.3.0             pyhd3deb0d_0    conda-forge
babel                     2.9.0              pyhd3deb0d_0    conda-forge
backcall                  0.2.0              pyh9f0ad1d_0    conda-forge
backports                 1.0                        py_2    conda-forge
backports.functools_lru_cache 1.6.1                    py_0    conda-forge
blas                      1.0                         mkl
bleach                    3.3.0              pyh44b312d_0    conda-forge
brotlipy                  0.7.0           py37hcc03f2d_1001    conda-forge
ca-certificates           2020.12.5            h5b45459_0    conda-forge
cachetools                4.2.1                    pypi_0    pypi
certifi                   2020.12.5        py37h03978a9_1    conda-forge
cffi                      1.14.4           py37hd8e9650_1    conda-forge
chardet                   3.0.4                    pypi_0    pypi
cloudpickle               1.6.0                    pypi_0    pypi
colorama                  0.4.4              pyh9f0ad1d_0    conda-forge
cryptography              3.4.4            py37h65266a2_0    conda-forge
cudatoolkit               11.0.221             h74a9793_0
cycler                    0.10.0                   pypi_0    pypi
cython                    0.29.21                  pypi_0    pypi
decorator                 4.4.2                      py_0    conda-forge
defusedxml                0.6.0                      py_0    conda-forge
detectron2                0.2.1                     dev_0    <develop>
entrypoints               0.3             pyhd8ed1ab_1003    conda-forge
freetype                  2.10.4               hd328e21_0
future                    0.18.2                   pypi_0    pypi
fvcore                    0.1.1.post20200716           py37    <unknown>
google-auth               1.4.2                    pypi_0    pypi
google-auth-oauthlib      0.4.2                    pypi_0    pypi
google-colab              1.0.0                    pypi_0    pypi
grpcio                    1.35.0                   pypi_0    pypi
idna                      2.8                      pypi_0    pypi
imgviz                    1.2.5                    pypi_0    pypi
importlib-metadata        3.4.0            py37h03978a9_0    conda-forge
importlib_metadata        3.4.0                hd8ed1ab_0    conda-forge
intel-openmp              2020.2                      254
iopath                    0.1.3                    pypi_0    pypi
ipykernel                 4.6.1                    pypi_0    pypi
```

ipython	5.5.0	pypi_0	pypi
ipython-genutils	0.2.0	pypi_0	pypi
ipython_genutils	0.2.0	py_1	conda-forge
jedi	0.18.0	py37h03978a9_2	conda-forge
jinja2	2.11.3	pyh44b312d_0	conda-forge
jpeg	9b	hb83a4c4_2	
json5	0.9.5	pyh9f0ad1d_0	conda-forge
jsonschema	3.2.0	py_2	conda-forge
jupyter-http-over-ws	0.0.8	pypi_0	pypi
jupyter_client	6.1.11	pyhd8ed1ab_1	conda-forge
jupyter_core	4.7.1	py37h03978a9_0	conda-forge
jupyter_server	1.3.0	py37h03978a9_0	conda-forge
jupyterlab	3.0.7	pyhd8ed1ab_0	conda-forge
jupyterlab_pygments	0.1.2	pyh9f0ad1d_0	conda-forge
jupyterlab_server	2.2.0	pyhd8ed1ab_0	conda-forge
kiwisolver	1.3.1	pypi_0	pypi
labelme	4.5.7	pypi_0	pypi
libpng	1.6.37	h2a8f88b_0	
libsodium	1.0.18	h8d14728_1	conda-forge
libtiff	4.1.0	h56a325e_1	
libuv	1.40.0	he774522_0	
lvis	0.5.3	pypi_0	pypi
lz4-c	1.9.3	h2bbff1b_0	
m2w64-gcc-libgfortran	5.3.0	6	conda-forge
m2w64-gcc-libs	5.3.0	7	conda-forge
m2w64-gcc-libs-core	5.3.0	7	conda-forge
m2w64-gmp	6.1.0	2	conda-forge
m2w64-libwinpthread-git	5.0.0.4634.697f757	2	conda-forge
markdown	3.3.3	pypi_0	pypi
markupsafe	1.1.1	py37hcc03f2d_3	conda-forge
matplotlib	3.2.2	pypi_0	pypi
mistune	0.8.4	py37hcc03f2d_1003	conda-forge
mkl	2020.2	256	
mkl-service	2.3.0	py37h196d8e1_0	
mkl_fft	1.2.0	py37h45dec08_0	
mkl_random	1.1.1	py37h47e9c7a_0	
mock	4.0.3	pypi_0	pypi
msys2-conda-epoch	20160418	1	conda-forge
nbclassic	0.2.6	pyhd8ed1ab_0	conda-forge
nbclient	0.5.1	pypi_0	pypi
nbconvert	6.0.7	py37h03978a9_3	conda-forge
nbformat	5.1.2	pyhd8ed1ab_1	conda-forge
nest-asyncio	1.5.1	pypi_0	pypi
ninja	1.10.2	py37h6d14046_0	
notebook	5.2.2	pypi_0	pypi
numpy	1.19.2	py37hadc3359_0	
numpy-base	1.19.2	py37ha3acd2a_0	
oauthlib	3.1.0	pypi_0	pypi

ocrd-fork-pylsd	0.0.3	pypi_0	pypi
olefile	0.46	py37_0	
opencv-python	4.5.1.48	pypi_0	pypi
openssl	1.1.1i	h8ffe710_0	conda-forge
packaging	20.9	pyh44b312d_0	conda-forge
pandas	0.24.2	pypi_0	pypi
pandoc	2.11.4	h8ffe710_0	conda-forge
pandocfilters	1.4.3	pypi_0	pypi
parso	0.8.1	pyhd8ed1ab_0	conda-forge
pickleshare	0.7.5	py_1003	conda-forge
pillow	8.1.0	py37h4fa10fc_0	
pip	20.3.3	py37haa95532_0	
portalocker	2.2.0	pypi_0	pypi
portpicker	1.2.0	pypi_0	pypi
prometheus_client	0.9.0	pyhd3deb0d_0	conda-forge
prompt-toolkit	1.0.18	pypi_0	pypi
protobuf	3.14.0	pypi_0	pypi
pyasn1	0.4.8	pypi_0	pypi
pyasn1-modules	0.2.8	pypi_0	pypi
pycocotools	2.0.2	pypi_0	pypi
pycparser	2.20	pyh9f0ad1d_2	conda-forge
pydot	1.4.1	pypi_0	pypi
pygments	2.7.4	pyhd8ed1ab_0	conda-forge
pylsd	0.0.2	pypi_0	pypi
pyopenssl	20.0.1	pyhd8ed1ab_0	conda-forge
pyparsing	2.4.7	pyh9f0ad1d_0	conda-forge
pyqt5	5.15.4	pypi_0	pypi
pyqt5-qt5	5.15.2	pypi_0	pypi
pyqt5-sip	12.8.1	pypi_0	pypi
pyrsistent	0.17.3	py37hcc03f2d_2	conda-forge
pysocks	1.7.1	py37h03978a9_3	conda-forge
python	3.7.0	hea74fb7_0	
python-dateutil	2.8.1	py_0	conda-forge
python_abi	3.7	1_cp37m	conda-forge
pytorch	1.7.1	py3.7_cuda110_cudnn8_0	pytorch
pytz	2021.1	pyhd8ed1ab_0	conda-forge
pywin32	300	pypi_0	pypi
pywinpty	0.5.7	py37hc8dfbb8_1	conda-forge
pyyaml	5.4.1	pypi_0	pypi
pyzmq	22.0.2	pypi_0	pypi
qtpy	1.9.0	pypi_0	pypi
requests	2.21.0	pypi_0	pypi
requests-oauthlib	1.3.0	pypi_0	pypi
rsa	4.7	pypi_0	pypi
send2trash	1.5.0	py_0	conda-forge
setuptools	52.0.0	py37haa95532_0	
simplegeneric	0.8.1	pypi_0	pypi
six	1.12.0	pypi_0	pypi

sniffio	1.2.0	py37h03978a9_1	conda-forge
tabulate	0.8.7	py37_0	
tensorboard	2.4.1	pypi_0	pypi
tensorboard-plugin-wit	1.8.0	pypi_0	pypi
termcolor	1.1.0	pypi_0	pypi
terminado	0.9.2	py37h03978a9_0	conda-forge
testpath	0.4.4	py_0	conda-forge
tk	8.6.10	he774522_0	
torchaudio	0.7.2	py37	pytorch
torchvision	0.8.2	py37_cu110	pytorch
tornado	4.5.3	pypi_0	pypi
tqdm	4.56.0	pyhd3eb1b0_0	
traitlets	5.0.5	py_0	conda-forge
typing_extensions	3.7.4.3	pyh06a4308_0	
urllib3	1.24.3	pypi_0	pypi
vc	14.2	h21ff451_1	
vs2015_runtime	14.27.29016	h5e58377_2	
wcwidth	0.2.5	pyh9f0ad1d_2	conda-forge
webencodings	0.5.1	pypi_0	pypi
werkzeug	1.0.1	pypi_0	pypi
wheel	0.36.2	pyhd3eb1b0_0	
win_inet_pton	1.1.0	py37h03978a9_2	conda-forge
wincertstore	0.2	py37_0	
winpty	0.4.3	4	conda-forge
xz	5.2.5	h62dcd97_0	
yacs	0.1.8	pypi_0	pypi
yaml	0.2.5	he774522_0	
zeromq	4.3.3	h0e60522_3	conda-forge
zipp	3.4.0	py_0	conda-forge
zlib	1.2.11	h62dcd97_4	
zstd	1.4.5	h04227a9_0	

D config

CUDNN_BENCHMARK: false
DATALOADER:
ASPECT_RATIO_GROUPING: true
FILTER_EMPTY_ANNOTATIONS: true
NUM_WORKERS: 2
REPEAT_THRESHOLD: 0.0
SAMPLER_TRAIN: TrainingSampler
DATASETS:
PRECOMPUTED_PROPOSAL_TOPK_TEST: 1000
PRECOMPUTED_PROPOSAL_TOPK_TRAIN: 2000
PROPOSAL_FILES_TEST: []
PROPOSAL_FILES_TRAIN: []
TEST:
- container_ceiling_test
TRAIN:
- containerCeilingV3_TestimgOnly_train
GLOBAL:
HACK: 1.0
INPUT:
CROP:
ENABLED: false
SIZE:
- 0.9
- 0.9
TYPE: relative_range
FORMAT: BGR
MASK_FORMAT: polygon
MAX_SIZE_TEST: 1333
MAX_SIZE_TRAIN: 1333
MIN_SIZE_TEST: 800
MIN_SIZE_TRAIN:
- 640
- 672
- 704
- 736
- 768
- 800
MIN_SIZE_TRAIN_SAMPLING: choice
RANDOM_FLIP: horizontal
MODEL:
ANCHOR_GENERATOR:
ANGLES:
- - -90
- 0
- 90

ASPECT_RATIOS:
- - 0.5
- 1.0
- 2.0
NAME: DefaultAnchorGenerator
OFFSET: 0.0
SIZES:
- - 32
- - 64
- - 128
- - 256
- - 512
BACKBONE:
FREEZE_AT: 2
NAME: build_resnet_fpn_backbone
DEVICE: cuda
FPN:
FUSE_TYPE: sum
IN_FEATURES:
- res2
- res3
- res4
- res5
NORM: ”
OUT_CHANNELS: 256
KEYPOINT_ON: false
LOAD_PROPOSALS: false
MASK_ON: true
META_ARCHITECTURE: GeneralizedRCNN
PANOPTIC_FPN:
COMBINE:
ENABLED: true
INSTANCES_CONFIDENCE_THRESH: 0.5
OVERLAP_THRESH: 0.5
STUFF_AREA_LIMIT: 4096
INSTANCE_LOSS_WEIGHT: 1.0
PIXEL_MEAN:
- 103.53
- 116.28
- 123.675
PIXEL_STD:
- 1.0
- 1.0
- 1.0
PROPOSAL_GENERATOR:
MIN_SIZE: 0
NAME: RPN
RESNETS:

DEFORM_MODULATED: false
DEFORM_NUM_GROUPS: 1
DEFORM_ON_PER_STAGE:
- false
- false
- false
- false
DEPTH: 50
NORM: FrozenBN
NUM_GROUPS: 1
OUT_FEATURES:
- res2
- res3
- res4
- res5
RES2_OUT_CHANNELS: 256
RES5_DILATION: 1
STEM_OUT_CHANNELS: 64
STRIDE_IN_1X1: true
WIDTH_PER_GROUP: 64
RETINANET:
BBOX_REG_LOSS_TYPE: smooth_l1
BBOX_REG_WEIGHTS: &id001
- 1.0
- 1.0
- 1.0
- 1.0
FOCAL_LOSS_ALPHA: 0.25
FOCAL_LOSS_GAMMA: 2.0
IN_FEATURES:
- p3
- p4
- p5
- p6
- p7
IOU_LABELS:
- 0
- -1
- 1
IOU_THRESHOLDS:
- 0.4
- 0.5
NMS_THRESH_TEST: 0.5
NORM: ”
NUM_CLASSES: 80
NUM_CONVS: 4
PRIOR_PROB: 0.01
SCORE_THRESH_TEST: 0.05

SMOOTH_L1_LOSS_BETA: 0.1
TOPK_CANDIDATES_TEST: 1000
ROI_BOX_CASCADE_HEAD:
BBOX_REG_WEIGHTS:
- - 10.0
- 10.0
- 5.0
- 5.0
- - 20.0
- 20.0
- 10.0
- 10.0
- - 30.0
- 30.0
- 15.0
- 15.0
IOUS:
- 0.5
- 0.6
- 0.7
ROI_BOX_HEAD:
BBOX_REG_LOSS_TYPE: smooth_l1
BBOX_REG_LOSS_WEIGHT: 1.0
BBOX_REG_WEIGHTS:
- 10.0
- 10.0
- 5.0
- 5.0
CLS_AGNOSTIC_BBOX_REG: false
CONV_DIM: 256
FC_DIM: 1024
NAME: FastRCNNConvFCHead
NORM: ”
NUM_CONV: 0
NUM_FC: 2
POOLER_RESOLUTION: 7
POOLER_SAMPLING_RATIO: 0
POOLER_TYPE: ROIAlignV2
SMOOTH_L1_BETA: 0.0
TRAIN_ON_PRED_BOXES: false
ROI_HEADS:
BATCH_SIZE_PER_IMAGE: 512
IN_FEATURES:
- p2
- p3
- p4
- p5
IOU_LABELS:

- 0
- 1
IOU_THRESHOLDS:
- 0.5
NAME: StandardROIHeads
NMS_THRESH_TEST: 0.5
NUM_CLASSES: 1
POSITIVE_FRACTION: 0.25
PROPOSAL_APPEND_GT: true
SCORE_THRESH_TEST: 0.9
ROI_KEYPOINT_HEAD:
CONV_DIMS:
- 512
- 512
- 512
- 512
- 512
- 512
- 512
- 512
LOSS_WEIGHT: 1.0
MIN_KEYPOINTS_PER_IMAGE: 1
NAME: KRCNNConvDeconvUpsampleHead
NORMALIZE_LOSS_BY_VISIBLE_KEYPOINTS: true
NUM_KEYPOINTS: 17
POOLER_RESOLUTION: 14
POOLER_SAMPLING_RATIO: 0
POOLER_TYPE: ROIAlignV2
ROI_MASK_HEAD:
CLS_AGNOSTIC_MASK: false
CONV_DIM: 256
NAME: MaskRCNNConvUpsampleHead
NORM: ”
NUM_CONV: 4
POOLER_RESOLUTION: 14
POOLER_SAMPLING_RATIO: 0
POOLER_TYPE: ROIAlignV2
RPN:
BATCH_SIZE_PER_IMAGE: 256
BBOX_REG_LOSS_TYPE: smooth_l1
BBOX_REG_LOSS_WEIGHT: 1.0
BBOX_REG_WEIGHTS: *id001
BOUNDARY_THRESH: -1
HEAD_NAME: StandardRPNHead
IN_FEATURES:
- p2
- p3
- p4

- p5
- p6
IOU_LABELS:
- 0
- -1
- 1
IOU_THRESHOLDS:
- 0.3
- 0.7
LOSS_WEIGHT: 1.0
NMS_THRESH: 0.7
POSITIVE_FRACTION: 0.5
POST_NMS_TOPK_TEST: 1000
POST_NMS_TOPK_TRAIN: 1000
PRE_NMS_TOPK_TEST: 1000
PRE_NMS_TOPK_TRAIN: 2000
SMOOTH_L1_BETA: 0.0
SEM_SEG_HEAD:
COMMON_STRIDE: 4
CONVS_DIM: 128
IGNORE_VALUE: 255
IN_FEATURES:
- p2
- p3
- p4
- p5
LOSS_WEIGHT: 1.0
NAME: SemSegFPNHead
NORM: GN
NUM_CLASSES: 54
WEIGHTS: ./output/model_final.pth
OUTPUT_DIR: ./output
SEED: -1
SOLVER:
AMP:
ENABLED: false
BASE_LR: 0.00025
BIAS_LR_FACTOR: 1.0
CHECKPOINT_PERIOD: 5000
CLIP_GRADIENTS:
CLIP_TYPE: value
CLIP_VALUE: 1.0
ENABLED: false
NORM_TYPE: 2.0
GAMMA: 0.1
IMS_PER_BATCH: 2
LR_SCHEDULER_NAME: WarmupMultiStepLR
MAX_ITER: 1000

MOMENTUM: 0.9
NESTEROV: false
REFERENCE_WORLD_SIZE: 0
STEPS:
- 210000
- 250000
WARMUP_FACTOR: 0.001
WARMUP_ITERS: 1000
WARMUP_METHOD: linear
WEIGHT_DECAY: 0.0001
WEIGHT_DECAY_BIAS: 0.0001
WEIGHT_DECAY_NORM: 0.0
TEST:
AUG:
ENABLED: false
FLIP: true
MAX_SIZE: 4000
MIN_SIZES:
- 400
- 500
- 600
- 700
- 800
- 900
- 1000
- 1100
- 1200
DETECTIONS_PER_IMAGE: 1
EVAL_PERIOD: 0
EXPECTED_RESULTS: []
KEYPOINT_OKS_SIGMAS: []
PRECISE_BN:
ENABLED: false
NUM_ITER: 200
VERSION: 2
VIS_PERIOD: 0

E Camera Calibration

```python
import numpy as np
import cv2
import glob
import sys
import argparse

#---------------------- SET THE PARAMETERS
nRows = 6
nCols = 9
dimension = 15 #- mm

workingFolder   = "./Calibration Images/IphoneCalibration" #find path of your images
imageType       = 'JPG' #image filetype
#------------------------------------------

# termination criteria
criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, dimension, 0.001)

# prepare object points, like (0,0,0), (1,0,0), (2,0,0) ....,(6,5,0)
objp = np.zeros((nRows*nCols,3), np.float32)
objp[:,:2] = np.mgrid[0:nCols,0:nRows].T.reshape(-1,2)

# Arrays to store object points and image points from all the images.
objpoints = [] # 3d point in real world space
imgpoints = [] # 2d points in image plane.

if len(sys.argv) < 6:
        print("\n Not enough inputs are provided. Using the default values.\n\n" \
              " type -h for help")
else: # can pass arguments from console to overwrite currentt arguments.
    workingFolder   = sys.argv[1]
    imageType       = sys.argv[2]
    nRows           = int(sys.argv[3])
    nCols           = int(sys.argv[4])
    dimension       = float(sys.argv[5])

if '-h' in sys.argv or '--h' in sys.argv:
    print("\n IMAGE CALIBRATION GIVEN A SET OF IMAGES")
    print(" call: python cameracalib.py <folder> <image type> <num rows (9)> <num cols (6)> <cell dimension (25)>")
    print("\n The script will look for every image in the provided folder and will show the pattern found." \
```

```
          " User can skip the image pressing ESC or accepting the image with RETURN. " \
          " At the end the end the following files are created:" \
          "  - cameraDistortion.txt" \
          "  - cameraMatrix.txt \n\n")

    sys.exit()

# Find the images files
filename    = workingFolder + "/*." + imageType
images      = glob.glob(filename)

print(len(images))
if len(images) < 9:
    print("Not enough images were found: at least 9 shall be provided!!!")
    sys.exit()

else:
    nPatternFound = 0
    imgNotGood = images[1]

    for fname in images:
        if 'calibresult' in fname: continue
        #-- Read the file and convert in greyscale
        img     = cv2.imread(fname)
        gray    = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)

        print("Reading image ", fname)

        # Find the chess board corners
        ret, corners = cv2.findChessboardCorners(gray, (nCols,nRows),None)

        # If found, add object points, image points (after refining them)
        if ret == True:
            print("Pattern found! Press ESC to skip or ENTER to accept")
            #--- Sometimes, Harris cornes fails with crappy pictures, so
            corners2 = cv2.cornerSubPix(gray,corners,(11,11),(-1,-1),criteria)

            # Draw and display the corners
            cv2.drawChessboardCorners(img, (nCols,nRows), corners2,ret)
            cv2.imshow('img',img)
            # cv2.waitKey(0)
            k = cv2.waitKey(0) & 0xFF
            if k == 27: #-- ESC Button
```

```
                print("Image Skipped")
                imgNotGood = fname
                continue

            print("Image accepted")
            nPatternFound += 1
            objpoints.append(objp)
            imgpoints.append(corners2)

            # cv2.waitKey(0)
        else:
            imgNotGood = fname

cv2.destroyAllWindows()

if (nPatternFound > 1):
    print("Found %d good images" % (nPatternFound))
    ret, mtx, dist, rvecs, tvecs = cv2.calibrateCamera(objpoints,
    ↪   imgpoints, gray.shape[::-1],None,None)

    # Undistort an image
    img = cv2.imread(imgNotGood)
    h,  w = img.shape[:2]
    print("Image to undistort: ", imgNotGood)
    newcameramtx,
    ↪   roi=cv2.getOptimalNewCameraMatrix(mtx,dist,(w,h),1,(w,h))

    # undistort
    mapx,mapy =
    ↪   cv2.initUndistortRectifyMap(mtx,dist,None,newcameramtx,(w,h),5)
    dst = cv2.remap(img,mapx,mapy,cv2.INTER_LINEAR)

    # crop the image
    x,y,w,h = roi
    dst = dst[y:y+h, x:x+w]
    print("ROI: ", x, y, w, h)

    cv2.imwrite(workingFolder + "/calibresult.png",dst)
    print("Calibrated picture saved as calibresult.png")
    print("Calibration Matrix: ")
    print(mtx)
    print("Disortion: ", dist)

    #--------- Save result
    filename = workingFolder + "/cameraMatrix.txt"
    np.savetxt(filename, mtx, delimiter=',')
    filename = workingFolder + "/cameraDistortion.txt"
    np.savetxt(filename, dist, delimiter=',')

    mean_error = 0
```

```python
    for i in range(len(objpoints)):
        imgpoints2, _ = cv2.projectPoints(objpoints[i], rvecs[i],
        ↪  tvecs[i], mtx, dist)
        error = cv2.norm(imgpoints[i],imgpoints2,
        ↪  cv2.NORM_L2)/len(imgpoints2)
        mean_error += error

    print("total error: ", mean_error/len(objpoints))

else:
    print("In order to calibrate you need at least 9 good pictures...
    ↪  try again")
```

F Mask Rcnn metrics

The total loss function in Mask R-CNN is calculated as:

$$\mathcal{L}_{total} = \mathcal{L}_{cls} + \mathcal{L}_{box} + \mathcal{L}_{mask} \tag{74}$$

Symbol	**Explanation**
$\mathbf{p}_i$	Predicted probability of anchor i being an object.
$\mathbf{p}_i^*$	Ground truth label (binary) of whether anchor i is an object.
$\mathbf{t}_i$	Predicted four parameterized coordinates.
$\mathbf{t}_i^*$	Ground truth coordinates.
$\mathbf{N}_{cls}$	Normalization term, set to 256
$\mathbf{N}_{box}$	Normalization term, set to 2400
λ	A balancing parameter, set to be 10

$$\mathcal{L} = \mathcal{L}_{\text{cls}} + \mathcal{L}_{\text{box}} \tag{75}$$

$$\mathcal{L}(\{p_i\}, \{t_i\}) = \frac{1}{N_{\text{cls}}} \sum_i \mathcal{L}_{\text{cls}}(p_i, p_i^*) + \frac{\lambda}{N_{\text{box}}} \sum_i p_i^* \cdot L_1^{\text{smooth}}(t_i - t_i^*) \tag{76}$$

The term λ $\mathcal{L}_{\text{cls}} + \mathcal{L}_{\text{box}}$ is set to 10 so (so that both $\mathcal{L}_{cls}$ and $\mathcal{L}_{box}$ terms are roughly equally weighted).
where

$$\mathcal{L}_{\text{cls}}(p_i, p_i^*) = -p_i^* \log p_i - (1 - p_i^*) \log(1 - p_i) \tag{77}$$

and

$$L_1^{\text{smooth}} = 0.1 \tag{78}$$

$\mathcal{L}_{mask}$ is calculated:

$$\mathcal{L}_{mask} = \frac{1}{m^2} \sum_{1 \le i,j \le m} [y_{ij} \log \hat{y}_{ij}^k + (1 - y_{ij}) \log(1 - \hat{y}_{ij}^k] \tag{79}$$

Mask loss function:

"As in Fast R-CNN, an RoI is considered positive if it has IoU with a ground-truth box of at least 0.5 and negative otherwise. The mask loss Lmask is defined only on positive RoIs. The mask target is the intersection between an RoI and its associated ground-truth mask." [20]

NTNU
Norwegian University of
Science and Technology