ACIT5900

MASTER THESIS

in

Applied Computer and Information Technology (ACIT)

May 2022

Applied Artificial Intelligence

Deep Learning for Crop Instance Segmentation

Patrick Ellefsen

Department of Computer Science

Faculty of Technology, Art and Design

OSLOMET

PREFACE

My motivation for choosing a project topic related to agriculture production derives from its primary function to our society. Though it is a subjective thought, developing efficient ways to bring food to the table seems meaningful.

The work put behind the thesis has been a journey with hours of learning and challenging bumps to get past. There is an ocean of methods and details in the field of computer vision. To be able to navigate through it demands an understanding at a deeper level and a good portion of trial and error with agonizing code.

I want to thank Sidney Pontes-Filho and Gustavo Mello for their insight and support under the writing of the thesis and for letting me join my fascination for growing plants with object detection.

ABSTRACT

This thesis explores object detection with instance segmentation in relation to agriculture. For the purpose of discovering a detection model that could potentially boost robotic greenhouse harvesters with newer and improved detection accuracy.

The project set out to validate a RGBD dataset of sweet pepper crops, train three instance segmentation models and compare the model performances. The RGBD dataset of sweet pepper crops was found to have good quality RGB and annotation files but was missing pixel values in the depth files. The models of Mask R-CNN, YOLACT and QueryInst was trained from scratch and with pretrained weights. Tuning the learning rate was initiated to improve model performance. The models were evaluated on the mean average precision (mAP) metric.

QueryInst failed to produce a mAP higher than zero. Mask R-CNN and YOLACT produced mAP scores of 45% and 30.1% for mask predictions, and 42.4% and 33.3% for box predictions respectively. Mask R-CNN had a slightly better mAP score in both cases. Visualizing the models revealed that Mask R-CNN had several correct predictions, while YOLACT predicted fewer correct and failed to recognize smaller instances. The project aimed to utilize RGBD data and its depth values to produce results in 3D representation. This was realised with the depth information.

CONTENTS

CONTENTS 3
1 INTRODUCTION 6
1.1 Problem statement 6
1.2 Proposed solution 6
1.3 Project plan 7
2 BACKGROUND THEORY 8
2.1 Artificial intelligence 8
2.1.1 Machine learning 8
2.1.2 Artificial neural networks 10
2.1.3 Deep learning 12
2.1.4 Deep learning workflow 13
2.1.5 Transfer learning 15
2.2 Computer vision 16
2.2.1 Object detection 16
2.2.2 Image segmentation 17
2.2.3 Evaluation metrics 17
2.2.4 Instance segmentation models 18
2.2.5 Three-dimensional data 19
2.3 Agriculture 22
2.3.1 Practices in agriculture 23
2.3.2 Harvester robots 23
3 LITERATURE REVIEW 24
3.1 Mask R-CNN for sweet pepper detection 24
3.2 State-of-the-art in 2021 crop monitoring robot 24
3.3 Transfer learning for Mask R-CNN and YOLOv5 25
3.4 Tomato detection with Mask R-CNN and YOLACT 25
3.5 Sweet pepper harvester with shape- and colour detection 25
3.6 3D crop localization in an apple orchard 26
3.7 3D crop detection based on colour, depth and shape 26
3.8 Point cloud instance segmentation for multiple species 27
4 METHODOLOGY 28

4.1 The programming framework 28
4.2 The datasets 29
4.2.1 Dataset selection 29
4.2.2 Dataset file inspection 30
4.2.3 Inspecting the dataset images with visualization tools 31
4.2.4 Other datasets for pretraining purposes 33
4.3 Models 33
4.3.1 Model selection 33
4.3.2 Model training 35
4.3.3 Mask R-CNN 37
4.3.4 YOLACT 40
4.3.5 QueryInst 43
4.4 3D representation 44
5 RESULTS 45
5.1 Dataset visualization 45
5.2 Model training and evaluation 47
5.2.1 Mask R-CNN 47
5.2.2 YOLACT 51
5.3 Model comparison 53
5.4 3D representation 54
6 DISCUSSION 55
7 CONCLUSION 58
8 REFERENCE LIST 59
9 APPENDIX 63

LIST OF FIGURES

Figure 1: How a simplified detection model works. (Images: Generated from the RGBD dataset)..... 9
Figure 2: Artificial neural network (Elgendy, 2020, p. 9). 10
Figure 3: Convolutional neural network (Elgendy, 2020, p. 103). 12
Figure 4: RGB image (left), depth image (right). (Images: Generated from the RGBD dataset). 21
Figure 5: Representing depth in 3D, original RGB (left), Open3D attempt (middle) and Matplotlib attempt (right). 32
Figure 6: RGB and depth visualized. 45
Figure 7: Depth equalized and histogram plotted. 46
Figure 8: Annotations visualized. 47
Figure 9: Model M01 results. 48
Figure 10: Model M02 results. 49
Figure 11: Model M03 results. 49
Figure 12: Model M04PRE results. 50
Figure 13: Model M05PRE results. 50
Figure 14: Model M06PRE results. 51
Figure 15: YOLACT visualization. 52
Figure 16: Mask R-CNN and YOLACT visual comparison. 53
Figure 17: 3D representation of models. 54

LIST OF TABLES

Table 1: Mask R-CNN results, Appendix (8,9) 48
Table 2: YOLACT results, Appendix (10). 52
Table 3: Mask R-CNN and YOLACT score comparison 53

1 INTRODUCTION

The introduction describes the problem statement, the proposed thesis' solution and defines the project plan. Recent research claims that agriculture is facing increased food insecurities. Developing accurate crop detection systems for greenhouse robots could be a key component to secure a stable food supply.

1.1 Problem statement

This section considers food insecurities at a global scale and narrows the problem down to crop detection challenges in greenhouses. The focus of the thesis is on object detection algorithms, though it emphasizes its research decisions in the scenario of harvesting robots in greenhouse environments. On the notion that robotization in greenhouses could prove to be beneficial at ensuring a steady food supply. Food insecurities are increasing because of climate change (IPCC, 2021) and the need to feed a growing population (UN, 2017). High-tech greenhouses are less impacted by weather and support food growing in places where there are limited arable land (Baudoin, et al., 2013, pp. 23-26).

Emerging in greenhouse cultivation are robots that monitors and harvest crops to aid farmers with management. Robotic greenhouse harvesters have the potential to support greenhouse management but needs further development to be a viable option for farmers. The robots operate with computer vision to identify and locate crops from the rest of the plant. How well this task is performed depends on the accuracy of the detection system it uses to locate the crops. There are multiple systems based on different approaches, but the most prominent technique uses deep neural networks (DNN) to learn where the objects are (Kootstra, Wang, Blok, Hemming, & Henten, 2020). The performance of a DNN detection system is essentially based on the dataset it learns from and which model architecture it uses to detect the objects (Zaidi, et al., 2021).

1.2 Proposed solution

To improve the accuracy of a robotic harvester's detection system, the thesis proposes to investigate a newly developed detection model with higher performance. To ensure equal footing the newly developed architecture is compared to two established detection architectures. Preferably two detection architectures that has shown promise in agriculture crop detection tasks. To fit the greenhouse criteria, the three models should learn from a dataset with crops that are cultivatable in a greenhouse environment. To heighten the accuracy of the detection, the models should output results as instance segmentation and

three-dimensional (3D) information. Instance segmentation allows the model to distinguish multiple crops from each other with detailed pixel-level localization (Gu, Bai, & Kong, 2022). 3D represented crops helps the harvester pick fruits in the 3D world and differentiate crops based on the additional depth information (Li, Feng, Qiu, Xie, & Zhao, 2022). The proposed solution hopes that the development of accurate detection systems can tilt the technology to be a profitable and efficient option for farmers.

1.3 Project plan

The project explores the topics of DNN, instance segmentation, RGBD (red, green, blue, depth) and 3D data in relations to agriculture practices. A RGBD dataset of sweet pepper crops was selected for training the models. The established models will be the Mask R-CNN and You Only Look at Coefficients (YOLACT) architectures. These were selected on the background of Mask R-CNN's regularly use in agriculture detection research and YOLACT's fast-based architecture, which stems from the popular You Only Look Once (YOLO) lineage. QueryInst was selected as the newer model for comparison due to its recent conception. The project's main experiments seek to follow the deep learning workflow and utilize transfer learning and learning rate tuning for optimal model performance. Where the models detect and mask the pixel values of multiple object instances in images and elevates the output to 3D information. The thesis will try to explain the technologies while justifying the choices that were made on the way.

To summarize, the goal of the thesis is to explore object detection with instance segmentation in relation to agriculture. This is aimed to be realised by training and comparing the Mask R-CNN, YOLACT and QueryInst architectures. The thesis will investigate these main tasks:

- Validate the RGBD dataset of sweet pepper crops
- Train Mask R-CNN, YOLACT and QueryInst
- Evaluate and compare the model performances

2 BACKGROUND THEORY

The following chapter will clarify the technologies and methods that the thesis is based upon. The chapter describes the task of object detection; how a machine learns to detect objects from images, and expand on the ideas of artificial intelligence, machine learning, computer vision and agriculture practices.

2.1 Artificial intelligence

For a greenhouse harvester to recognize sweet peppers, it needs to know what it should look for. This is learned by processing a dataset with numerous images that contain sweet peppers. The idea of learning has emerged with the objective of artificial intelligence (AI). The field of AI sets to build machines that replicates the human nervous system. This has been realized to some degree with mathematics and modern computer systems, but only for a limited fraction of what the human brain is capable of (Aggarwal, 2018, p. 1). A section of AI called machine learning has researched and developed methods that allow machines to learn from experiencing data. This has been the initiator for several modern learning techniques, where DNNs are at the point for what the thesis is investigating.

2.1.1 Machine learning

Machine learning (ML) learns patterns from a set of data samples, which can be used to predict patterns in new unfamiliar data. This has surfaced the prominent techniques of artificial neural networks (ANN) and deep learning (DL). The recent developments in computer power and the availability of data have been the springboard for their current success. Though ANNs are more complex and powerful than the simpler ML algorithms, they share similar processing principles. There are three prevailing ways of approaching learning in ML: Supervised, unsupervised and reinforced learning (Aggarwal, 2018, p. 2). The next paragraphs focus on supervised learning when performing object detection and explains how learning algorithm works.

Supervised learning

Detection models in supervised learning look at numerous labelled images to learn how to detect objects in unseen images. To set the context, supervised learning has applications for analysing financial markets, finding statistical patterns or speech recognition tasks to mention a few. This project focuses on image data and detection tasks. Object detection is a computer vision task that sets out to locate and identify certain objects from the surrounding environment in images. Supervised learning is the traditional strategy to do this, because of

its accurate predictions and its potential to automate manual work. The method relies on manually labelled datasets to pinpoint out what and where in the image the model should look. The image has tiny pixels in different colours and when combined can form structures and observable things in the image. These structures can be narrowed down to geometrical lines and shapes, which are called image features. A model can learn when these features are observable in an image, and therefore learn to recognize certain objects. This is accomplished by training the model on a labelled dataset (Elgendy, 2020, p. 6).

A simple case of training a detection model

To get a better overview of how the model learns and which components it contains, here is a short and compact explanation of the process. It starts by loading an image into a learning algorithm. This is termed training. For the model to learn which image features it should look at when predicting the location of the crops, it needs guidance. This is learned by predicting an outcome that gets evaluated and then the model's inner statistics are updated in an iterative manner to improve on its performance. This is denoted as an optimization problem, and the model essentially seeks to minimize the error between the predicted localization of crops and the actual labelled crops for each loop as seen from the *Fig.1* example.

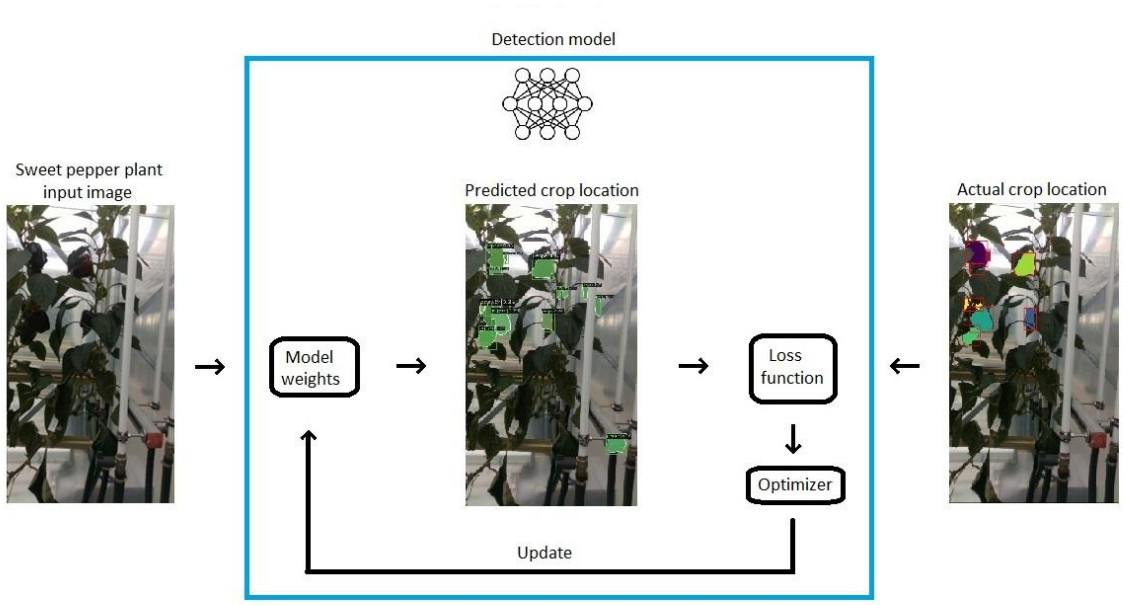

Figure 1: How a simplified detection model works. (Images: Generated from the RGBD dataset).

As shown in *Fig.1*, the prediction is compared with the ground truth in a component called the loss function. The prediction guessed on areas without any crops. This generates an error rate in the loss function, which the optimizer component seeks to minimize by sending an update that changes the network weights which is called backpropagation. The weights decide which features the model will focus on when looking at the input image, which in turn affects the

predicted outcome. These components repeat their process until a high model performance is achieved. This happens when the error rate is small, and it has reached a point called a converging model minimum (Goodfellow, Bengio, & Courville, 2019, pp. 99-105). This simplified training example serves to reveal the basic components of a detection model and their purposes. Details for each component will be explained further into the chapter. Extending on the methods of supervised learning, ANNs have in the last decade been on the forefront for this task.

2.1.2 Artificial neural networks

From the classic machine learning algorithms, the ANNs are superior at abstracting semantic information out of complex data structures (Aggarwal, 2018, p. 1).

How the artificial neural network learns

Inspired by the animal brain, an ANN, replicates the neurons in a biological brain with a network of connecting nodes. The biological brain is structured with nerve cells called neurons that are connected through a web of synapses. The synaptic channels send nervous signals between the neurons. The connections have varying strength, which gets stronger or weaker when the brain's organism is exposed to stimuli. This is how the brain learns. An ANN, *Fig.2* mimics the synaptic connections by designing layers of nodes with weighted connections to learn from input data (Aggarwal, 2018, pp. 1-5).

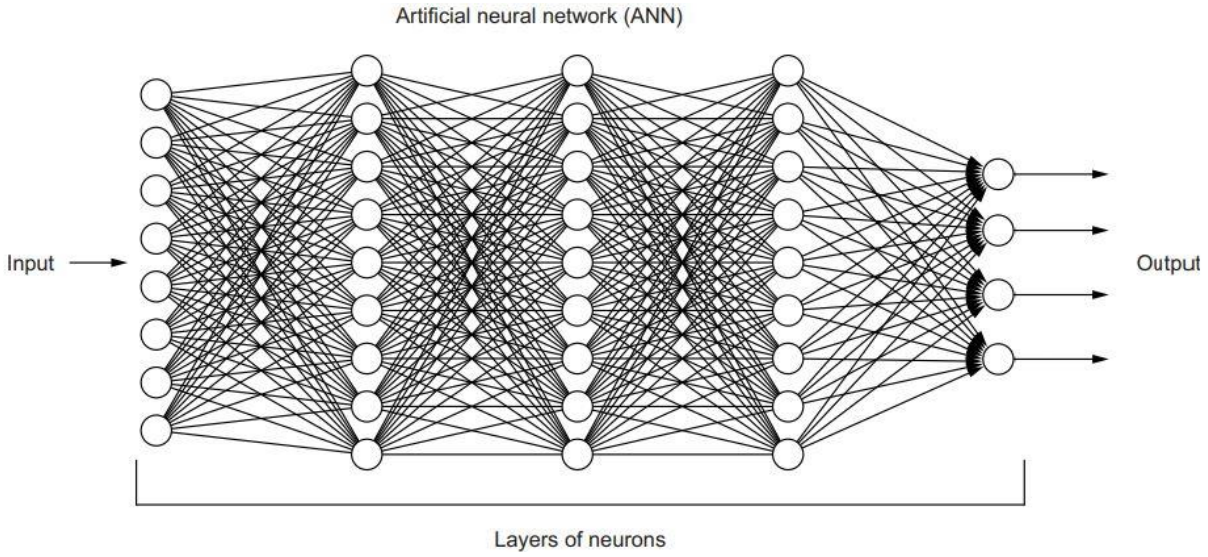

Figure 2: Artificial neural network (Elgendy, 2020, p. 9).

Network structure

For a basic ANN, the nodes contain activation function and the internal components of input,

bias, weights and output. Following the stream of the input data, the data is multiplied with the weights, summed and passed forward to the activation function. These functions take the shapes of linear or nonlinear models that activates at certain threshold values (regarding the bias) to pass the data to the next layer. A multi-layered ANN has several hidden layers that does these computations to the data before sending it forward. Flowing the data forward in one direction between layers is called a feedforward network (Aggarwal, 2018, pp. 11-14).

To be said, there is a difference between training and running the model. Running the model, stops the backpropagation process and allows the data to flow one way. This stops the loss function and optimizer from updating the weights.

Loss functions

The loss function calculates the error between the prediction and the ground truth by setting a number on how wrong the prediction is. A smaller loss equals a higher model accuracy. The two most common loss functions are the mean squared error (MSE) and cross-entropy. The MSE is best suited for regression and the cross-entropy exceeds on classification problems (Elgendy, 2020, pp. 68-73).

Optimizers

The optimizer takes the loss function error and strives to minimize it by updating the weights. The ANNs can contain millions of weights, which would take a supercomputer ages to find the perfect combination of weight values. The purpose of the optimizer is therefore to find a "good enough" combination of weights that generates practical predictions. There are various optimizers to choose from depending on the task and simplicity, where the technique of gradient descent stands as the prevailing approach for ANNs. Three versions have emerged from gradient descent (GD), batch (BGD), stochastic (SGD) and mini-batch (MB-GD) (Elgendy, 2020, pp. 74-83)

The challenges with batch gradient descent are that it may get stuck in local minima and that a too large batch size exhausts computer memory. At the start of training, the weights are randomly selected. This can make the optimizer minimize the error to local minima and miss out on a global minimum which provides lower error rates. The batch size is how many samples the computer takes into memory to process at a time. With larger datasets it is impractical to process all the samples at the same time. SGD mitigates local minima and is the go-to optimizer for ANNs. This is because it tries to improve one sample at a time. Though it may get stuck in periods of the training, its random initialization allows it to jump out of local minima and to potential global minima. It is fast but will only reach close to the global minimum. MB-GD is in-between BGD and SGD by allowing a set batch size to be processed (Elgendy, 2020, pp. 83-85).

For the ANN to be effective at learning it has been discovered that with more layers the network can have a higher performance. This has brought forth the design of DNNs and the term DL.

2.1.3 Deep learning

DL and DNNs have achieved performances that exceed human perception (Elgendy, 2020, p. 10). Building on previous sections, the greenhouse harvester now has the means to process the data and learns by adjusting its weights for better crop detection performance. For it to recognize shapes and colours in an image it needs a method to make sense of the pixel values (a pixel is one small square in the image that contains a colour). This will help it to define geometrical shapes by combining pixel values and set the premise to learn to identify and locate a sweet pepper in an image. The method of convolutional neural network (CNN) stands in the base of most modern object detection algorithms, because of its high detection accuracy and fast computing speed (Elgendy, 2020, p. 11).

Convolutional neural networks

CNNs rely on multiple layers that build distinct shapes which increase in complexity for each successive layer. The first layer will focus on simple shapes as lines and edges. The next layer will combine previous layers and form more complex shapes as squares and circles. After several sequences recognizable features as leaf-like structures, bending stems and parts of fruits will appear. The architecture behind the CNN follows a similar design as the DNN. Multiple layers have neurons that activate at certain patterns. The weights between the layers are changed to improve the learning of features. This is calculating by the error in the prediction and backpropagated to update the weights. The difference is that it does feature extraction with convolutional layers as visualized in *Fig.3* (Elgendy, 2020, pp. 102-107).

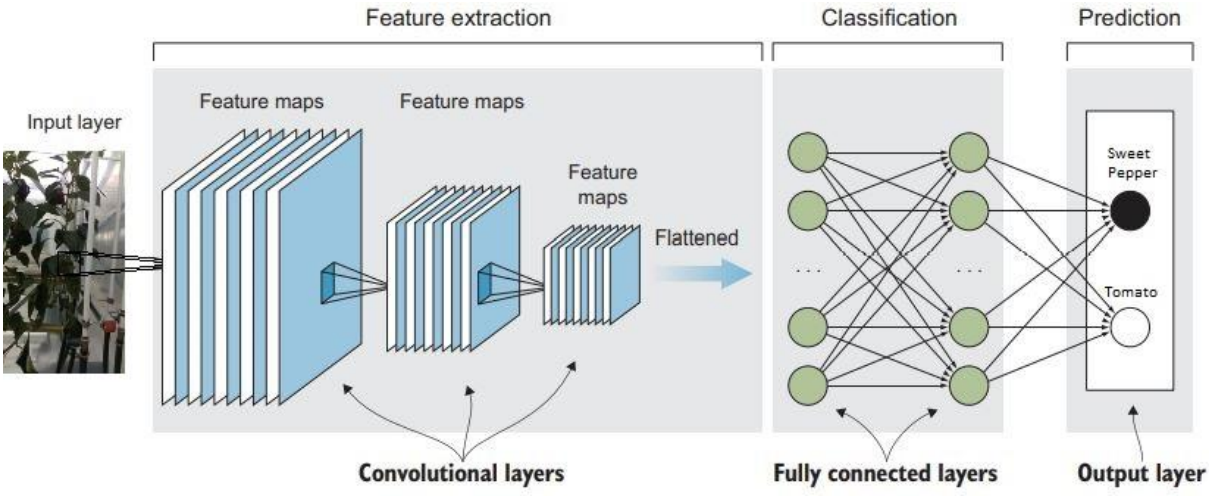

Figure 3: Convolutional neural network (Elgendy, 2020, p. 103).

Convolutional layers look at small parts of the image and its pixel values to create a new modified image that serves as a feature map. They are not fully connected to each pixel in the input image but locally connected to a few pixels. This allows the algorithm to focus on specific features in the image by applying a filter to the locally connected pixel fields. The filter covers a given area by a set matrix size and slides over the image while doing calculations. These filters are the weights of the network, so they will be given a random value at initialization and be changed as the network learns. The filters have the purpose of amplifying features in the image and take out important information and put them into feature maps that the classification layer uses to make a prediction (Elgendy, 2020, pp. 108-120). This is the short explanation of how CNN functions. State-of-the-art object detection algorithms are built on the principles of the CNN architecture.

2.1.4 Deep learning workflow

The overall flow of a DL project is to train a model, analyse its results, change model parameters and repeat until an appropriate performance is met. DL models have many components that assemble into a system of possible outcomes. This makes it difficult to pinpoint out which exact component that generates a certain behaviour. Although, under model development there are established decisions that have a larger impact on the result. The decisions relate to the model's architecture, pre-processing of the dataset, performance metrics to analyse the results and hyperparameter tuning to achieve higher model performance (Elgendy, 2020, p. 145).

Selecting the architecture

The model's architecture derives from the purpose of the project and the desired model output, while the data format is based on the model architecture. The purpose of the thesis is to do object detection on a dataset of sweet pepper crops, where the object outputs are set to be of instance segmentation. These criteria determine which models that can be selected.

Data formats and pre-processing

DL models need specific data formats to be able to input the data and do processing on it. The data in a dataset would often need to be transformed into the data format the model understands. This could be to change the colour channels, resize the image shapes, normalize the pixel values to a range between one and zero or transform how the data is stored as a data type. The data formatting ensures that the input data is of the same base and is necessary to produce consistent results and make the model work properly. Getting the right data format, is usually settled by thoroughly inspecting the current data and then transforming it to the necessary format. There is a chance of human errors or faults from previous data handling, which can cause defective information in the dataset. Ensuring that the dataset shows what it is supposed to is a safety measure to ensure that the model works as intended (Elgendy, 2020, pp. 153-155).

Dataset splits

For learner algorithms it is a routine to split the dataset into either training and testing or training, validation and testing. The rule of thumb is that the test data is apart from the sets used under training, hence, to ensure that the model gets evaluated on data that it has not seen before. The training set is for the model to extract features and learn weights. The validation set is for evaluating the model under training and is the initiator of changing the model parameters to enhance performance. The testing set is the final evaluation of the model performance. For a small dataset, the split itself is normally at 80/20 or 70/30 percent between training and testing (Elgendy, 2020, pp. 151-153).

Performance metrics and behaviour

The performance metric varies in line with the output goal of the model and is a comparison between the predicted outcome and the ground truth values. The ground truth values are the annotated datafiles that are linked to the image files in the dataset. For object detection the performance is calculated by an accuracy metric that compares the predicted area with the actual area where the object is in the image. How this metric works will be further specified later in the chapter. What makes a good or bad prediction is decided from the performance metric (Elgendy, 2020, p. 156).

For DNNs, the performance can be affected by defects in the dataset, bottlenecks in the components or poor results because of under- or overfitting. The defects and bottlenecks could be exposed by unusual performance results and a long processing time. Problems like this could be mitigated by searching the code for errors and analyse each component and their behavioural response. Over- and underfitting is a common problem for bad performing models. Underfitting issues occurs when the model struggles to learn from the training set, while overfitting occurs when the model learns the data too good and is therefore inferior when presented with new images. The occurrence of these two states is acknowledged when comparing the training and validation error. If the training error is low and the validation error high, it can be presumed that the model is overfitting the data. Changing the hyperparameters may resolve to better performance. If the training and validation error is high, the problem could be that the model is too simple. Adding more layers to the network or training for a longer time lets the model go deeper into abstraction and thus detect with greater detail. Plotting a learning curve is a helpful tool to visualize the potential under- or overfitting over a set time period (Elgendy, 2020, pp. 167-169).

Hyperparameters and tuning

Hyperparameters and parameters are variables that change inside the DNNs, where parameters are modified by the network itself, hyperparameters are used to tune the settings of the network. Hyperparameters are defined before training is started and can be roughly grouped into hyperparameters that relates to network architecture, learning rate and

regularization techniques (Elgendy, 2020, p. 163).

For the network architecture, the hyperparameters are the number of hidden layers, the number of neurons and the type of activation functions. The hidden layers define the depth of the network and is heavily coupled to the networks ability to learn features. If enough data is available, it is proven performance-wise favourable with a deeper network. Adding more neurons tend to mitigate problems with underfitting. For activation functions there are many available options, though when replicating an existing model, the functions are generally already selected to fit the model's design (Elgendy, 2020, pp. 163-164).

The learning hyperparameters address how the network learns by approaching a minimal error and optimize the weights. The learning rate decides how many steps the optimizer takes when it is searching to find the lowest error rate. A large learning rate makes the optimizer go fast but may jump over points with low error rates. On the other hand, a small learning rate makes the model learn slowly and use time to achieve results (Elgendy, 2020, p. 164).

Regularization is a technique that seeks to counteract problems with overfitting while training. This can be by reducing the importance of random weights, dropping out neurons from the training or augment the data so the model has a bigger pool of samples to learn from. The purpose of this is to make the network simpler and robust (Elgendy, 2020, pp. 177-180).

2.1.5 Transfer learning

Transfer learning is the idea of training a learner algorithm on a dataset with certain features and then transfer these learned weights over to training on a new dataset. Training takes time, starting from a point with already optimized weights saves training time. Larger datasets are directly connected to better results because it allows the learner algorithm to have a broader set of features to learn from. Transfer learning from a larger generalized dataset helps capture the basic geometrical shapes and figures that are often shared between objects. In cases where the dataset of interest contains few samples, transfer learning from a basic featured dataset raises the learner algorithm's results. This is the general idea of transfer learning, but it can be implemented in different parts of the model. Either as a pretrained feature extractor referred to as a backbone, a pretrained detector referred to as a head or to finetune a few of the already trained layers of the model (Elgendy, 2020, pp. 240-244).

For CNNs that do features extraction and detection, the feature extraction part can potentially be pretrained and the detection part can be trained from scratch and vice-versa. A pretrained feature extractor has optimized its weights to find feature maps. How transferable these weights are to datasets with different objects depends on the geometrical similarities in the datasets. The generated feature maps increase in complexity as the model delves deeper into the network layers. In the model's early stages, the feature maps are from basic geometrical

shapes, while later stages recognize complex features specific for that dataset. Optimal results are produced when the pretrained dataset and the custom dataset share similarities. It is possible to freeze certain layers of the pretrained dataset, to keep the generalized early layers if the two datasets do not share similar complex features (Elgendy, 2020, pp. 244-250).

Finetuning takes generalized weights from the feature extractor and trains the rest of the network from scratch. This is useful if the targeted dataset has few similarities to the pretrained dataset and if the targeted dataset is small. For a small dataset that is not like the pretrained dataset, the best approach is to freeze half of the early stages of the feature extractor and train the rest of the model from scratch. Because of the dataset's small size, finetuning the whole network could make it overfit the data. (Elgendy, 2020, pp. 250-261).

2.2 Computer vision

Computer vision (CV) interweaves with DL by being a considerable part of AI development. CV sets out to imitate the human visual system by using sensing devices that captures the world around it. This visual information is saved as data that can be processed by a DNN (Elgendy, 2020, pp. 4-8).

2.2.1 Object detection

Object detection uses computer vision and DNNs to recognize and locate multiple objects in an image. Earlier classification methods could only set one label to each image. Object detection improves on this approach by breaking down the original image into smaller regions of interest and label each region. The located objects are usually framed in with a bounding box (Elgendy, 2020, pp. 283-284).

The basis of object detection

The object detection algorithms of region-based convolutional neural network (R-CNN), single-shot detector (SSD) and YOLO, all share a general detection structure. They are based on DNNs and have four architectural practices they follow: region proposal, feature extraction, non-maximum suppression and evaluation metrics. Region proposal has its own DL algorithm that generates bounding boxes for each region that is considered interesting. What regions the network decides as interesting is dependent on an objectness score. The top objectness scoring regions are passed forward to the feature extraction layers. In this part, the network extract features from the image. These features are then used to determine if the image contains any recognizable objects. To mitigate a chaos of boxes, the non-maximum suppression layers find and combine repeating and overlapping boxes into one bounding box for each object. The final piece in the architectural structure is the evaluation metric, this sets

a score to the predictions and evaluate the model's performance (Elgendy, 2020, p. 285). For the harvester to locate sweet pepper boundaries in greater detail it needs a method to focus on pixel extraction.

2.2.2 Image segmentation

Image segmentation seeks to label and locate objects by highlighting their pixels. It improves on object detection's localization capabilities; instead of using inaccurate bounding boxes, image segmentation masks the exact pixels inside each object. Resulting in higher detection accuracy (He, et al., 2021, p. 1).

Types of segmentation

There are three types of segmentation tasks for images: Semantic, instance and part segmentation. Semantic segmentation targets to label all the pixels in the image to a class. The classes can be of certain objects in the image or the background. It does not differentiate between two entities of the same class. Instance segmentation aims to have multiple instances of the same class to differentiate between entities. Part segmentation takes it one step further and targets to classify the different parts of the objects (He, et al., 2021, pp. 1-2).

Why is it useful?

For a harvester robot, detecting every nuance in a crop's shape could be a practical addition for a robotic arm when it stretches out to pick a ripe fruit. A pixelwise representation of the crops is therefore a favourable approach as to a bounding box, which provides little information to support detailed localization. For crop detection, instance segmentation has the necessary attributes to distinguish clusters of fruits and with further development can be beneficial for counting the total yield. This makes it the preferred alternative (Xu, et al., 2022, pp. 1-2).

2.2.3 Evaluation metrics

The most used method for evaluating image segmentation models is the mean average precision (mAP). This is used to evaluate both the bounding box and the mask predictions. The mAP metric is generated from measuring the intersection over union (IoU) and a precision-recall curve (PR curve) (Elgendy, 2020, p. 289).

Intersection over union

IoU measures the overlap between the prediction and the ground truth by a 0-100 percentage score. Where a higher score means a closer overlap. In research and for model performance charts, IoU scores over threshold values of 50 and 75% are usually selected to sort out the

predictions that are closer to being correct. The IoU score sets the base for measuring the accuracy for each prediction, but this does not count for predictions that are incorrect (Elgendy, 2020, pp. 289-291).

Precision and recall

Higher precision minimizes the amount of wrongly labelled predictions, and a higher recall minimizes the amount of missed labels predictions. Explaining precision and recall is easier with an example. From an image with tomatoes and other crops, the model has a goal to predict the location of tomatoes. If it locates a tomato correctly, the prediction is a true positive. If the location is wrong, it is a false positive. If the model locates another crop and classifies it as something else then a tomato, this is a true negative. The tomatoes the model missed to locate are false negatives. These terms are then used to calculate the precision and recall (Elgendy, 2020, pp. 147-148).

Mean average precision

The mAP metric builds on the earlier mentioned calculations and outputs a score in the range of 0-100. From calculating the PR curve for all the model classes, the mAP can be determined by measuring the area under the curve (AUC). The mAP metric is often referred to as just AP. It is commonly used with COCO evaluation (Elgendy, 2020, pp. 289-292).

2.2.4 Instance segmentation models

The instance segmentation models build on ideas unravelled in the previous chapters and add techniques that improve shortcomings or fixes known challenges. It is a maturing field that has still room for improvements when it comes to detection accuracy and speed.

Model architectures

There are three types of instance segmentation architectures: Single-stage, two-stage and multi-stage. The two-stage architecture is the most used configuration because of its high accuracy and adequate processing time. The multi-stage and two-stage architectures can be divided into a sequence of two parts: Object detection and object segmentation. The single-stage deviates from this by being able to do both parts at the same time (Gu, Bai, & Kong, 2022, p. 8).

A two-stage detector uses region proposal in the first stage to feed regions of interest to the second stage of the model, where the model predicts the class and localization of potential objects. The single-stage detectors do the classification and localization at the same time by applying a network that suggests masks for image regions. The method is more effective, because it finds correlation between the detection and segmentation tasks, where the two-

stage execute these steps separately. This makes the single-stage detector a lot faster but on behalf of accuracy (Gu, Bai, & Kong, 2022, p. 17). Although, research states that the trade-off is in the single-stage's favour by gaining more speed than it loses on accuracy (Bolya, Zhou, Xiao, & Lee, YOLACT, 2019).

Mask R-CNN

Mask R-CNN (He, Gkioxari, Dollar, & Girshick, 2017) builds on the previous object detection models from the R-CNN family. The R-CNN family relies on a region proposal network (RPN) to suggest regions of interest (ROI) in the feature maps generated by the CNN backbone. Mask R-CNN brings in the novel approach of RoIAlign, which uses small feature maps from the input data to align the ROIs with higher precision. This allows for the generation of accurate object masks.

YOLACT

YOLACT, (Bolya, Zhou, Xiao, & Lee, YOLACT, 2019) adds a mask section to the one-stage-based object detector. A fully convolutional network (FCN) is used to propose masks for the entire image, in addition to a prediction head that produce coefficients for each mask instance from the FCN in parallel. This allows the network to localize instances. In contrast to Mask R-CNN, the network drops the RPN layer which makes it a lot faster. A consequence of this, is that the predictions lack quality and makes therefore YOLACT performs worse at detecting smaller objects.

QueryInst

QueryInst's (Fang, et al., 2021) novel approach is to treat every instance as a learnable query. This query is shared by the bounding box detection and mask segmentation layers in parallel to achieve state-of-the-art performance in 2021. QueryInst can be built upon existing query-based detectors as Sparce R-CNN. It runs six query stages in parallel that query features from mask RoIs with dynamic mask heads that do convolutions.

2.2.5 Three-dimensional data

Visualizing a 3D scene or object can be realized with datatypes as RGBD, volumetric, multi-view, point clouds or mesh structures.

Data representation

From a 2D image with width and height, the 3D representation adds the third dimension of depth. 3D data structures can be sorted into Euclidean space or non-Euclidean space. Euclidean space defines the data graphically with (x, y, z) coordinates resembling (width,

height, depth) and is supported by the methods of RGBD, volumetric and multi-view. This approach is more suitable to work with 2D DL architectures because of their computable coordinate representations. Unlike the non-Euclidean methods as point clouds and mesh without set representations (Ahmed, et al., 2019, p. 3).

Euclidean methods

RGBD provides 2,5D information by storing a 2D colour image and a 3D depth map. It has had a rise in interest because of its inexpensive equipment costs, simplicity and versatile use. Volumetric data uses voxels to represent 3D shapes. Voxels are squares of a set size that can be both visible parts and internal non-visible parts. This makes it quite memory heavy to process as well as it lacks high-resolution data. Multi-view captures the 3D object by looking at 2D images from various angles circling around the object. This is less computational heavy but suffers from selecting the optimal number of views for each object. Both volumetric and multi-view is therefore most useful when analysing stationary 3D objects (Ahmed, et al., 2019, pp. 4-5).

Non-Euclidean methods

Examining the non-Euclidean methods, point clouds are unstructured formations of points that creates a 3D scene. Though it has local Euclidean coordinates, the global lack of structure and because of connectivity issues between points, the point cloud is challenging to compute. Mesh data is represented by a set of polygons which is formed by multiple 2D bounded geometrical figures. The irregular mesh structure has limited compatibilities with current DNNs. Mainly because of the availability of datasets and its transferability with current DNNs. RGBD data representation is the current favourable choice (Ahmed, et al., 2019, pp. 3-6).

RGBD data

RGBD is the combination of the colour channels RGB (red, green, blue) and the depth values. RGB is the colour values for each pixel in an image and can have a value between 0-255. Mixing the colour channels together creates a distinct colour for one specific image pixel. The depth values can be combined with RGB to give each pixel a distance from the vision device. This can be used to 3D-represent the scene or capture depth information in relation to the image pixels, thus allowing for object localization with depth information (Ward, Laga, & Bennamoun, 2019, pp. 1-2).

Normally, RGB and depth values are two separate image files similar to *Fig. X*. This makes it possible to process the RGB image with well-established 2D detection models, before the depth image is used to find the object's placement in 3D. Both image files are in matrix formats, with three dimensions that decide the width, height and multiple channels for each image. The two first dimensions set the coordinated position for each pixel in the image, starting with (0, 0) position at the top left pixel. The channels can have RGB data, depth data

or other signal values (Elgendy, 2020, pp. 20-22).

Figure 4: RGB image (left), depth image (right). (Images: Generated from the RGBD dataset).

There are a few challenges related to RGBD data. In the case of a sweet pepper crop, the plant has leaves and stems that often blocks the robotic vision from seeing the whole crop. This makes object detection algorithms prone to accuracy loss when one crop can be mistaken for two if separated by occlusion or crops hidden behind plant material will not be recognized at all. Related to the RGBD devices, they are affected by changes in illumination and have a limited sensor range, these issues can cause blurry and missing data. For 3D information compared to 2D, there are computational drawbacks because of the additional layers that needs processing. The equipment is also expensive and to get the optimal data extraction it requires the ability to handle sensors with more complexity. This makes quality RGBD datasets sparse compared to pure RGB datasets (Ward, Laga, & Bennamoun, 2019, pp. 3-4).

3D data extraction

3D datasets are captured with sensory devices that extract 3D information. These devices range from everyday cameras to devices that use frequencies as sonar, infrared and laser (Martinez-Guanter, et al., 2019, pp. 2-4). There are three prevailing methods of acquiring 3D data in agriculture: Lidar, stereovision and RGBD.

Lidar measures the distance from the device to the surrounding environment by calculating the time it takes for a laser light to bounce back to the device. This generates a point cloud. Lidar is very accurate, but it lacks colour registration and is rather complex to work with. Therefore, it is best suited for mapping and navigation rather than crop detection tasks (Martinez-Guanter, et al., 2019, p. 3). Stereovision uses two RGB cameras and structure-of-motion (SfM) photogrammetry to generate 3D data. The two cameras calculate

correspondence between pixels in the captured scene with the SfM algorithm and the camera parameters. Although it is the currently most accurate method, the high processing time makes it unsuitable for real time appliances (Gené-Mola, et al., 2019, pp. 1,6). RGBD data is acquired with RGBD cameras that have both an RGB camera and a depth sensor attached to it. The depth sensor relies on the time-of-flight principle, which measures the time between an emitting infrared light and the reflection back to the device (Lin, Tang, Zou, Xiong, & Fang, 2019, p. 3). RGBD devices have so far, the upper hand regarding colour texture availability, low user complexity and a reasonable pricing. The RGBD qualities gives a greenhouse harvesting robot narrow vision between rows of crops and fast processing speeds when doing crop detection while moving.

Methods for processing RGBD

Processing RGBD data for image segmentation tasks can follow various pipelines to when the 3D information should be included to which part of the network structure. For DL there are three current approaches to processing RGBD data with multiple objects.

The first method processes the RGB data and does detection on it in 2D before it elevates the findings into 3D coordinates. This allows for the use of well-established 2D object detection algorithms. In addition, the depth values often have less resolution which leads to poorer quality when the data is 3D represented. Though, the method does not benefit from the additional information the depth data gives. The second method tries to correct this by processing the depth data directly. There are various proposed approaches to handle the data, most treat the depth data as another image to be processed by algorithms based on CNN. The third method transforms the RGBD into 3D volumetric data. This data format can then be processed by fully 3D CNNs. Though this sets a high pressure on computational capabilities, and it is estimates that it takes up to 30 minutes longer to process data compared to the 2D approaches (Ward, Laga, & Bennamoun, 2019, pp. 11-13).

2.3 Agriculture

To build on the thesis' scenario, this section will unfold important terms in agriculture. Agriculture has from early humans been the key to a growing civilization. Throughout history, technological breakthroughs have created better tools to optimize the farm work. Heavy machinery improved the field work under the industrial revolution. In the last decades, chemicals have made farming more profitable by protecting the crops. Biotechnology is improving the crops genetic capabilities to withstand hazards and maximize yield (Johns Hopkins, 2022). Current research uses big data and devices to gather heaps of data to analyse and monitor farm management. Closing in are the use of fully automized harvesting robots (Kootstra, Wang, Blok, Hemming, & Henten, 2020).

2.3.1 Practices in agriculture

Agriculture has various practices depending on the crop and the surrounding climate. The thesis has defined set criteria it pursues to follow when it comes to selecting a dataset and the purpose of the detection models. This section unravels why the set criteria are considered.

Traditional farming is commonly performed on open fields. This is a very efficient way of farming, because of convenient mass-harvesting machinery (Kootstra, Wang, Blok, Hemming, & Henten, 2020, p. 99). Though, practical for only certain crops. High-value crops like tomatoes, peppers and apples cannot be mass-harvested by machinery and requires manual work to be collected. Normally, these crops are cultivated in orchards or inside greenhouses.

A high-tech greenhouse has the benefit of allowing environmental control and enables a facility to be located (in theory) anywhere. Depending on the location, a drawback is the potential sky-high energy expenditures from regulating the grow-lights and temperature if placed in a disadvantageous regional climate (Baudoin, et al., 2013, p. 35). In a controlled environment, fresh-water and fertilizers can be carefully optimized. This helps reduce the depletion of critical and limited regional resources. A greenhouse environment will be able to stabilize the outer weather extremities and work as a cover for crop damage (Baudoin, et al., 2013, pp. 23-25). This sets a base for predictability where research can develop harvester robots. To make it as profitable as to open-field farming, the technology must be of the highest quality.

2.3.2 Harvester robots

A chain of practices works together to deliver a pack of tomatoes on the table. Seeds must be sown, the tomato plants need continuous nurture and optimal surrounding conditions to grow, and when the time is right the tomatoes needs to be picked and packed ready for delivery. In agriculture there are both outdoor and indoor production and emerging robots are specialized at different environments and fruits. Today there are few robots that can accomplish these manual tasks. Tasks as moving around without damaging the crops and having the ability to remove occlusions to get a better view of the surroundings. Inside a greenhouse environment, the plants get bigger in an unique and uncontrollable manner as they grow. Developing automation in this type of environment has shown to be a challenge, because of all the variations in the environment. In addition, over time the vision and electronics will also be affected by humidity, temperature and changes in illumination (Kootstra, Wang, Blok, Hemming, & Henten, 2020). Although these obstacles, there are some systems starting to take form both in research and in the industry. In the past decades as stated in this review paper (Bac, Henten, Hemming, & Edan, 2014) there have been over 50 attempts to commercialize a fully automated robotic system without success. The cause being that the systems have too poor performance to be a useful replacement for manual work.

3 LITERATURE REVIEW

This chapter analyses current literature in agriculture on instance segmentation models. The literature review sets a bar for where the thesis aspires to contribute.

3.1 Mask R-CNN for sweet pepper detection

Halstead et al. developed three crop-based datasets which was tested with Mask R-CNN and Faster R-CNN (Halstead, Denman, Fookes, & McCool, 2020). The research pursues to generalize sweet pepper crop detection with a cross-domain approach by capturing datasets based on different species and environments. Their results show that Mask R-CNN has increased performance on cross-domain datasets compared to previous R-CNN frameworks, while it offers more accurate localization with object masks.

The conducted research demonstrates Mask R-CNN capabilities and transferability to agricultural detection environments. The thesis takes inspiration from their work and implements their RGBD sweet pepper dataset as the main data source.

3.2 State-of-the-art in 2021 crop monitoring robot

Smitt et al. presents a robot capable of crop surveying a greenhouse environment (Smitt & Mccool, 2021). Their solution takes RGB and depth data to map the facility and count potential yield, being one of the first to show results of accomplishing these tasks. For their detection system they use Mask R-CNN with the depth data to delimit the range of which crops that would be detected and counted. The detection model was trained on the three datasets from the research of Halstead et al.

Mask R-CNN is also here selected for detection, which adds on its adoption for crop detection tasks. Their research utilises the depth values after the mask segmentation has been completed. The report sets the bar for where the technology is today by being the state-of-the-art in greenhouse monitoring.

3.3 Transfer learning for Mask R-CNN and YOLOv5

Autz et al. compares pretrained models to models trained from scratch on agriculture datasets (Autz, Mishra, Herrmann, & Hertzberg, 2022). The experiments were conducted with Mask R-CNN and YOLOv5, a two-stage detector versus a one-stage. The results for Mask R-CNN producing a mAP of 75% when trained from scratch and a mAP of 74.55% when pretrained on COCO. The authors conclude that performance is higher if the network is trained from scratch, then if trained on large pretrained networks but large agriculture datasets are not yet available.

The report states the shortage of quality datasets in agriculture. Their research is based on just one dataset of sugar beets and the models do not share the attribute of instance segmentation. This produces results based on a small foundation. In contrast to this report, the thesis trains the instance segmentation version of YOLO, YOLACT and compares it with Mask R-CNN and QueryInst.

3.4 Tomato detection with Mask R-CNN and YOLACT

Xu et al. researched accurate recognition of fruits and stems for cherry tomato crops for the purpose of automatic picking (Xu, et al., 2022). The authors developed an improved Mask R-CNN that achieved a mAP of 93.76% and had a higher accuracy then the standard Mask R-CNN and YOLACT architectures. Processing an image takes 0.04 seconds. Their novel contribution implemented RGBD data to enhance the feature extraction.

The report concludes RGBD has the potential to boost model accuracy, which was a key investigation criterion when starting the thesis but was in the end not implemented. YOLACT is used to compare the results, which adds on its acceptance for crop detection tasks. The thesis wants to compare the Mask R-CNN with the newer QueryInst to see if it outperforms Mask R-CNN and YOLACT.

3.5 Sweet pepper harvester with shape- and colour detection

Arad et al. developed a harvester robot called SWEEPER which uses RGBD cameras and a shape and colour sensitive algorithm to detect crops (Arad, et al., 2020). The algorithm is only capable of semantic segmentation. Under testing, SWEEPER achieved a crop detection rate of 69% accuracy. In total a 18% of the ripe crops were harvested in a commercial greenhouse layout.

The research was produced in January 2020, though the field is having a rapid development this research represents the state of current harvester on the market. The act of harvesting a crop requires a stepwise execution from detection, reaching, cutting, handling and storing. The vision system is the first layer that starts this stepwise process. A better vision system would improve the rest of the chain of harvesting.

3.6 3D crop localization in an apple orchard

Li et al. present a one-stage detector to better tackle occlusion and noise in the vision system of a robotic harvester (Li, Feng, Qiu, Xie, & Zhao, 2022). Their model uses YOLACT++ to process 2D information and detect crops with instance segmentation and bounding boxes. The point cloud was generated from the segmented masks and the depth data from the robot's RGBD vision system. This produced better 3D localization results for partial occluded crops.

The article also conducts a comparison between Mask R-CNN and YOLACT architectures with ResNet-50 and ResNet-100 backbones, in addition to measure the FPS for each model. This serves as a good benchmark for the model's performances on an agriculture dataset that has similarities to the thesis' sweet pepper dataset. Relative to the article, the thesis trains the newer detection model of QueryInst to analyse its performance against the established detectors.

3.7 3D crop detection based on colour, depth and shape

Lin et al. researched the detection of spherical and cylindrical crops based on colour, shape and depth features for a robotic harvester (Lin, Tang, Zou, Xiong, & Fang, 2019). The model recognises the crops by colour which creates a mask that is used to filter the depth data to focus on the crops alone. The data is processed by an image segmentation method that sorts the crops into clusters. This data is then transformed to a point cloud which a 3D shape detector uses to detect crops. Their results show a mAP of 87%, 74% and 81% for pepper, eggplant and guava detection, but the detection time can take up to 4.70 second to finish.

The report presents results for an approach unlike the DL architectural basis of the thesis. DL demands large quantities of data to excel and takes therefore time to establish, algorithms as in the article shows that alternative approaches are possible with workable performance.

3.8 Point cloud instance segmentation for multiple species

Li et al. (Li, et al., 2022) present a 3D based network that recognise tobacco, tomato and sorghum, focusing on the instance segmentation of leaves on the purpose of detecting plant species and plant parts. Their research shows a mAP of 83.30% for detecting the species and instance segmenting out the leaves. They state there is a shortage for well labelled 3D datasets.

Though it is not directly related to the 2D model approaches nor the crop detection as in the thesis, the research shows alternative ways to get high accuracy detection from fully 3D based networks. There is little research to be found on 3D instance segmentation for agriculture and the shortage of datasets maybe the main cause. Approaching 3D data with 2D detectors seems to produce the best performing models for the time being.

4 METHODOLOGY

The methodology explains in detail how the testing of the dataset and the selected models was conducted and which decision that were made.

4.1 The programming framework

The framework consists of multiple programming components that are used as tools to develop the project.

Python programming language

Detection models rely on a programming language, code extensions and hardware technologies that process large chunks of data to work efficiently. To build the project, the programming language of Python (Rossum, Guido, Drake, & Fred, 2009) was selected because of its wide acceptance in computer science and ML communities. Its broad acknowledgement avail examples and helpful tips online, which reduces the time stuck on error handling. The models used in the project are quite complex; in the way they handle the data and which libraries that is needed for them to run. Libraries are extended code that simplifies programming. For ML there are libraries that make building DNNs simpler with existing functions for DNN components. These libraries often have DNN models ready for deployment. Building the detection models from scratch would take time, the thesis seeks to use existing code and edit it to fit the purpose of the project.

Compute unified device architecture

Certain aspects regarding the hardware and software (devices and versions are further detailed in the Appendix) were dependant on the right build to make the code work as intended. To train the models effectively, using a graphic processing unit (GPU) is essential. Compared to the central processing unit (CPU), the GPU process data at higher speeds. Accomplishing this on a computer can be utilized with compute unified device architecture (CUDA) compatible graphic cards. CUDA joins the processing capabilities of the GPU with a software like Python, making it available for use when programming (NVIDIA, Vingelmann, P., Fitzek, & P., 2020). This serves as the bridge between the GPU and the Python libraries used in the coming detections models.

Google Colaboratory

The project was first initiated on a local computer, but it was early discovered that the setup lacked computational resources to handle the task of instance segmentation. The model

training was therefore moved to Google Colab. This is a free online GPU service which facilitates for training ML algorithms (Google Colab, 2022). The platform was used for training the model at a much higher speed than locally.

The local computer was still used for visualizing the dataset and the model results, because of the simplicity of working locally with files and programs, and because Colab had a few restrictions to its usage. For good reasons, Colab is restricted to a few hours of connections to the GPUs, therefore it was best to keep this valuable time for training purposes. The local computer was set up with Anaconda (Anaconda Software Distribution, 2020), a package management tool to easier install and remove Python packages. Packages communicate with one another, and it is therefore crucial to have the right versions that are compatible for smooth code production.

4.2 The datasets

The datasets are crucial to the task of object detection, because they directly relate to how the DNNs learn to locate and recognize objects.

4.2.1 Dataset selection

On the premise of a greenhouse harvester certain criteria were sought for. The object of detection should be a crop cultivatable in a greenhouse environment, the annotations should preferably be the type of instance segmentation and there should be depth data for the possibility to 3D localize the crops.

From these dataset specifications, several public datasets from an agriculture dataset survey (Lu & Young, 2020) were investigated and a few private datasets were inquired access to through e-mail. Conclusively, a dataset from Agrobotics was selected because it matched the criteria to the letter by having images of sweet pepper crops in the format of RGBD. The dataset was shared by Michael Halstead in Agrobotics and was developed in the conducted research from “Fruit Detection in the Wild” (Halstead, Denman, Fookes, & McCool, 2020). To be said, there was a limited number of viable datasets for 3D localization tasks, few were available to the public and finding a practical sample for the set criteria required a comprehensive search through individual websites.

Other public depth-based datasets that were considered: Fuji apple dataset (Gené-Mola, et al., 2019), an RGBD-based dataset of apples captured with a Kinect V2 camera with annotated bounding boxes distinguishing out the fruits. A broccoli dataset (Kusumam, Krajnik, Pearson, Duckett, & Cielniak, 2017) captured with a Kinect V2 and transformed into a point cloud,

where the annotations were the centroids for each broccoli flower. A sugar beets dataset (Chebrolu, et al., 2017) from multiple sensors capturing LiDAR, RGBD and GPS coordinates, with annotated segmentation of the crops and weeds. Comparing the datasets, the RGBD dataset of sweet pepper crops exceeded the others by having indoor crops with instance segmented annotations.

4.2.2 Dataset file inspection

The dataset contained RGB images, depth images, raw annotation files of each crop and instance annotations based on crop colours.

The RGB and depth files

The RGB images are in the PNG format. The depth data were in the tagged image file format (TIFF), which stores and compresses large image files. The annotations were also in the PNG format. The PNG format permits image visualization, while also containing data with extra attributes. This allows for labelling areas of the image and saving this as attached annotated data to the PNG file. A great deal of time was put into understanding the data, especially the PNG, which was very difficult to extract information from when there was no overview of what it contained. PNGs can contain various data and there was no Python code to inspect it all at once. There was also an uncertainty if the TIFF file was opened the right way and that it conserved the data due to hard-to-find information on how to perform the data extraction correctly.

The annotation files

The raw annotation files are of the type PNG and had all the annotated instances with each instance part as its own file. The raw annotations were first used as the main source for labelling. A program was made to combine all the instances and all the separate parts into a distinct variable for image sample. This is detailed in the *annotation_to_segmentation* function from the *visualize_annotation.py (21)*. The program also gave each instance its own colour on the premise that this could distinguish the instances from each other in coming processing tasks. When starting on the model pre-processing it was though discovered that it was easier to use the already coded labels from the supplementary annotations of instances based on colour and crop ripeness instead. The annotation files based on crop colour had each instance of fruit separated into a ripeness colour ranging from black, green, mixed and red. Therefore, four annotated image files are available per image sample. Each instance had its own placement in the image as pixel coordinates and could be identified by a designated label. These annotation files were therefore selected for further processing.

The dataset split

In total, 150 samples and their belonging depth and annotation data were taken out from the dataset to be used to train and test the models. Following the previously covered in section *2.1.4 Deep learning workflow*, the samples were divided into a 1/3 separation, where training was given 100 samples and testing 50 samples. In addition, a separate test image was used to visualize the model performances. The split was decided to be train/test because that fitted the input structure of the Mask R-CNN architecture. An attempt was made to figure out a way to make the Mask R-CNN code follow a train/validation/test structure, but it turned out cumbersome to investigate the lines of code that was behind the evaluation metric.

4.2.3 Inspecting the dataset images with visualization tools

To quality assure that the dataset contained what it was supposed to have, the dataset was visualized and inspected with various Python tools as further showed with figures in the results chapter. Getting to know the data was also necessary to understand its transmission with the coming detection models. A set of helper functions were made with NumPy, OpenCV, Matplotlib and Pillow. Matplotlib and OpenCV was used to upload the data to NumPy arrays for further data manipulation. These libraries are generally used Python computing tools for array computing, image processing and data visualization.

Inspecting the RGB and depth files

Examining the RGB and depth images revealed image areas without representable values, though on the positive side RGB and depth values looked to share the same pixel placements. Given a dataset sample, *read_data (1)* loaded the RGB, depth and annotation masks into variables. In *subplot_rgb_depth_2D (2)* the RGB and depth were visualized and inspected with Matplotlib as showed in *Fig.6*. The depth data was concatenated into the RGB with a NumPy stack function that added the depth as an extra channel. Matplotlib treated the depth channel as a binary mask, visualizing every value over one. The visualization revealed that when the two images were overlayed, there was indications of missing data where the image had areas that were completely white. This was seen in the depth image as well, where the image looked almost black and lacked contrast. The depth pixels were also checked by measuring the frequency of the values with a histogram representation in *histogram_depth (3).* An OpenCV equalizer function was also added to the depth image, to get a better representation of the data as seen in *Fig.7.* The missing data could be from the sensor's laser pulse not bouncing back. When manually looking at the overlayed image there was areas with missing data around leaves and crops that had valuable depth information. From known issues with this sensor, this could be caused by moisture on the lens or bad lightning conditions.

An effort was made to visualize the depth data without any functional results. The library of Open3D (Open3D, 2022) was tested due to its echoing recommendations through online

forums. Their visualization function was reliant on camera intrinsic parameters to visualize the scenes properly. This was not acquirable from the RGBD dataset of sweet pepper crops, but an attempt to use a generalized approach was tried which produced the obscured results as shown i Fig.5. Matplotlib was also tested for 3D visualization, but it turned out difficult to make it show colour, while it also was unpractically slow to work with.

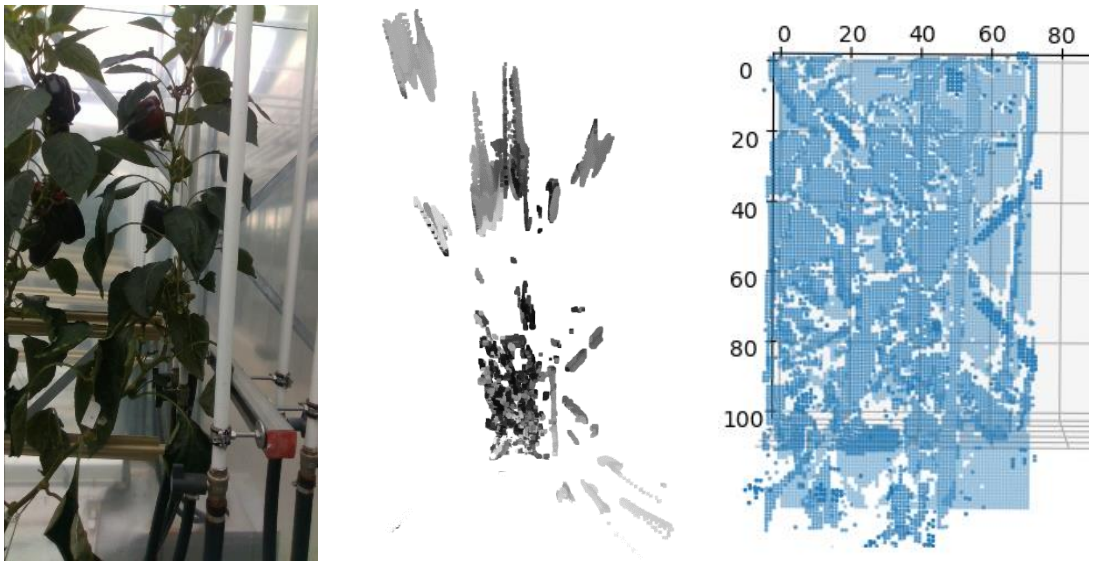

Figure 5: Representing depth in 3D, original RGB (left), Open3D attempt (middle) and Matplotlib attempt (right).

Inspecting the annotation files

The annotation files were manually examined to ensure that the labels were placed correctly in the samples. Parts of the *read_data (1)* function combined the four annotation files of crop instances into one variable and sorted the labels in the right order for easier processing. An offset measured the length of the number of labels for each annotated image to mitigate that crop instances from the images ended up sharing the same label. The output was then changed so that each annotated crop had its own label even if the instance had multiple unconnected parts. Developing this function was achieved through tips from the Stackoverflow (Stackoverflow, 2022) community and by printing out important data transformations to detect if the code changed to something out of order.

subplot_mask_2D (4) visualized the crop instances with distinct colours per labelled crop. In addition, a bounding box framed the area which the crop's pixels were annotated by calculating their minimum and maximum positions in the image as seen in *Fig.8*. The masked images were then confirmed to be true by manually counting and comparing the annotations to the generated masked image. This was completed for ten randomly picked samples from the dataset. Stating an assumption of consistency in the dataset with correctly placed annotations for the crop instances.

4.2.4 Other datasets for pretraining purposes

Selecting the most suited dataset for transfer learning could improve model performance. The most prominent current general datasets for instance segmentation tasks are the Microsoft common objects in context (COCO), Pascal visual object classes (VOC), Large vocabulary instance segmentation (LVIS) and Open image. COCO contains 2.5 million annotations for 328 thousand images with 80 classes available for instance segmentation. The images vary in resolution and captures objects from everyday scenes. It is the most implemented dataset because of its large image base of general objects (Gu, Bai, & Kong, 2022, p. 5).

For pretraining the models, the COCO dataset was selected over the others because of its large general database of instance annotated images and its available pairing with established model frameworks. The sweet pepper dataset maybe too small to achieve desirable results. Therefore, utilizing pretrained weights from a COCO dataset might be an important function to boost the performance of model.

4.3 Models

Building the detection models ready for predictions requires hardware and software with capacities to train the models. The detection models were selected on behalf of the Literature review chapter, current agriculture approaches and Papers with Code's most implemented instance segmentation models (PaperswithCode, 2022). This indicated that Mask R-CNN and YOLACT were two reliable model architectures for agriculture tasks. The thesis wanted to rival the two established algorithms with a newer addition. Available from the MMDetection library, QueryInst was released in 2021 and was therefore selected as the third model.

4.3.1 Model selection

For selecting the models, there were a few criteria that should be met to have a functioning solution and a smooth project development. The models needed to be capable of processing RGBD images and output instance segmentation results. Preferably the models should have high performances and a renowned reputation in the field to ensure that the implementation is working and there is available information for error handling.

Ideas that were dropped

Initially, the final model was going to use ensemble methods in a layer over the three detection models. For clearer communication between the detection models and the ensemble methods, an emphasis was made to run all the models on the same GPU-accessing library. Two top GPU-accessing libraries were Tensorflow and PyTorch. Using both, would

most likely cascade in dependency problems with other libraries and CUDA. Dependency problems happen when libraries of different versions not compatible with each other is used to build a program. A popular science report has often many suggested code approaches for the report's solution besides official code examples. As seen from Papers with Code, PyTorch is a commonly used tool in instance segmentation papers and was therefore selected for this project.

Concerning the ensemble method, it is a technique that can be used to train separate models and combine their results into an average score to reduce generalization errors. One model's poor performance will have less impact on the final ensemble score, when the score is the average result for all models (Goodfellow, Bengio, & Courville, 2019, pp. 256-258). The project had already started implementing two models when it was decided to drop this feature. Even though the models were built on PyTorch, their outputs were quite different. It would take time to implement an ensemble that could understand the model outputs and configurate this data together. This seemed unrealistic to achieve with the time left before thesis deadline.

Before it was decided to be dropped, 3D-based models were searched for. Two point cloud-based alternatives were found, SSTNet (Liang, Li, Xu, Tan, & Jia, Instance Segmentation in 3D Scenes using Semantic Superpoint Tree Networks, 2021) and OccuSeg (Han, Zheng, Xu, & Fang, 2020). They offered instance segmentation output, but were specialized in scene reconstruction, could be hard to implement because of less popularity and needed the data to be of the point cloud format. From further reading it was also discovered that the 2D-based models could potentially process a third depth layer, therefore the fully 3D-based models were dropped.

It was decided to use 2D detection models that only handled the RGB data from the RGBD dataset of sweet pepper crops and instead elevate the final output to be represented in 3D. This was not optimal nor the desired outcome when first starting on this project. Though, because of lack of available tutorials, few models to choose from and a still blooming 3D processing community, priorities were made to be able to finish the project on time. Feeding the 2D detection model with the extra depth dimension could be possible without too much alteration, but still required throughout understanding of the three models for it to function properly. It is also uncertain how the networks would handle the 2D annotations. Based on the large addition of data the depth information adds to the networks, the project could have encountered shortcomings on computer resources. The reasonable path regarding time was then decided to be by applying 2D detection models to the dataset.

How the models were selected

From the literature review, the dataset search and reading research related to agriculture, an overview started to form. Mask R-CNN was often referred to in agriculture detection task and had the highest popularity on Papers with Code for instance segmentation (Papers with Code, 2022). This made it a safe bet. Since Mask R-CNN was based on the two-stage architecture it

seemed natural to seek out a one-stage architecture for comparison. The YOLO lineage got high appraise on Papers with Code, though it only handled detection through bounding boxes its instance segmentation based relative YOLACT seemed like a good fit. YOLACT also appeared in agriculture detection research as in the apple orchard detection article by Li et al. (Li, Feng, Qiu, Xie, & Zhao, 2022) and from the cherry tomato detection article by Xu et al. (Xu, et al., 2022). In addition to the detection model survey by Zaidi et al. (Zaidi, et al., 2021) which highlights recent development in the field. YOLACT was on these premises selected as the second model.

Both Mask R-CNN and YOLACT had newer improved versions, as Mask Scoring R-CNN (Huang, Huang, Gong, Huang, & Wang, 2019) and YOLACT++ (Bolya, Zhou, Xiao, & Lee, YOLACT++: Better Real-time Instance Segmentation, 2019), these were though not engaged, because newer could potentially mean less robust code and fewer tutorials to learn from, which could in worst case make the project get stuck at error handling. Considering the two first models were safe bets; the idea was that the third model could be more of a wild card. The QueryInst model available in the MMDetection library was selected because of implementation convenience, differing architecture and that it was newer and scored better at benchmark tests (Fang, et al., 2021, p. 6).

4.3.2 Model training

The three models shared a few similar architectural configurations. These configurations work as the base for the forthcoming model training sections.

Backbone

The backbone of ResNet-50 was selected because the deeper ResNets are marginally improving the accuracy in exchange for complexity and thus longer training time (He, Gkioxari, Dollar, & Girshick, 2017, pp. 6-7). ResNet-50 is a feature extractor that has shown a high performance (Elgendy, 2020, p. 230) and is therefore commonly included in ML libraries like the thesis utilizes. This makes it a convenient choice and the preferred pick under model training.

Optimizers

For the optimizer, a mini-batch SGD with Momentum or Adam is the recommended choice for DL (Elgendy, 2020, p. 174). These generalize well on multiple types of DL architectures by adding learning decay to make the learning rate more dynamic under training. Momentum helps the gradient descent move in a relevant direction by adding a portion from the previous step vector. Often set as gamma equals 0.9 (Ruder, 2017, p. 4). Adam shares similarities with momentum's push in the right direction by multiplying with an average of past gradients. It also adds an average of past squared gradients to mitigate shrinking learning rates (Ruder,

2017, p. 7). This is the short and simplified explanation. If obtainable in the model repository, these optimizers will be favoured.

The batch size will preferably be set to 4 with 2 number of workers to speed up the training. 4 was selected to ensure that the training did not exhaust the limited Google Colab resources, and over 2 workers made a warning message appear in the Colab terminal.

Learning rate and epochs

After initiating a test run on Google Colab, running the model took around 30 minutes per 10th epoch with default settings. Google Colab has a limited resource capacity within a 12-hour window with diminishing GPU availability (Google Colab, 2022). A maximum of 50 epochs was therefore favoured. For DL learning rate (LR) tuning, the LR is recommended to start from 0.01, and go down a tenth per training regime (Elgendy, 2020, p. 169). The thesis aspires to train the models with varying training rates to identify if the performance will hit a minimum faster because of the limited training time of 50 epochs.

Visualizing the results

The models were visualized on just one image excluded from the training sets. The image showed 24 instances of sweet peppers. Predictions over a IoU threshold of 50 was considered as true positives to show the instances the models predicted were over a 50% certainty for that class. Mask visualization was selected for pixels that model predicted was over 95% certain belonged to the instance.

Evaluation

The evaluation was produced by the Pycocotools for Mask R-CNN and from an internally function for the MMDetection models. They both followed the COCO evaluation standard and produces the similar output metrics. An attempt was initiated to better evaluate the training itself with graphs showing the loss rate over time, but it did not match with the Pycocotools evaluation. So, it turned out be one or the other. Pycocotools were the preferred pick. As stated in the section *2.1.4 Deep learning workflow*, a learning curve would be a practical addition to analyse how the training loss and validation loss behaved over time. From PyTorch's issues page (PyTorch, 2022), having a way to acquire the loss seems like an open case. Figuring out a solution to this would take lot of time and requires a deep understanding on how PyTorch process code under training and evaluation. Since it was not available for Mask R-CNN it was discarded for the two other models as well.

In Chapter 5, the evaluation metrics showing the mAP scores with a IoU threshold of 50 (IoU=50) was selected because it is regularly used in literature. The scores for IoU on an average over 10 IoU thresholds between 50 and 95 (IoU=50:95) was also included because it is stated as COCO's primary challenge metric and it sets a measure for how certain the model

was when compared with IoU=50 (COCO, 2022).

4.3.3 Mask R-CNN

As previously reviewed literature states, the mask R-CNN is an established algorithm used in many crop-detection models for agriculture applications.

Implementation code

From Papers with Code (PaperswithCode, 2022) one of the topmost implemented papers for instance segmentation was the Matterport mask R-CNN (Matterport, 2019). This was first selected on the notion that its popularity makes it clear to implement. Parts of the code was though outdated, and Anaconda did not support the requirements for the Tensorflow GPU version. After some time trying to fix these problems, it was decided instead that the project should use PyTorch for all its code implementation to round up the detection models under the same conditions. Broadening the search, there were many tutorials and a few older ones. To lower the chance of dependency problems an up-to-date tutorial from PyTorch was selected (PyTorch, 2022). This had the option to use a pre-trained model on the COCO dataset and use transfer learning to expand the training to the RGBD dataset of sweet pepper crops.

PyTorch model requirements

The requirements for the tutorial were a dataset containing images and annotations of object instances, a tool called Pycocotool for evaluation, reference scripts and a few libraries detailed in Appendix (6). The RGBD dataset of sweet pepper crops contained all the data information but needed to be altered to fit the model. For the model to process the data correctly, the input should be a Python dictionary containing: An image in PIL format, bounding boxes with two X and Y coordinates, distinct labels for each instance, an image identifier, an area variable defining the bounding box area, an instance mask for each object and their pixel placement in the image. The Pycocotools (GitHub, 2022) was downloaded to be used as the evaluation metric that produced mAP scores. Helper functions were downloaded from the PyTorch GitHub page to simplify the processing as further detailed in the Appendix (6).

Pre-processing

For the Mask R-CNN model based on the PyTorch tutorial, the dataset needed alterations for the model to understand what kind of data it was getting tasked to process. The RGB image files did not need any configuration. After searching for a tutorial, it was discovered that the common practice of storing the annotations was in a JSON or TXT file. It was decided not to copy this, but rather make a code snippet that structured the data in the required format.

To sort the annotation images into one processable variable, the *read_data (1)* loop was again

used for that purpose and was included in the Python training file, further detailed in *mrcnn_train.py (21).* Running the training file caused an error with the generated bounding boxes. Two of the bounding box coordinates shared the same value in the same axis, which would form a straight line in-between two instances instead of a box. This was barely visible when zooming in on the image area. The annotations were then counted manually by looking at the image and compared with the *np.unique* function. This function stated how many unique instances the program counted. It was then discovered that the program counted for extra instances. To get a better view of the problem, the image was visualized with the object masks. This revealed that two instances had overlapping pixels, which originated from how *np.add* adds the pixel values together and therefore create a new instance. An if statement was added to ensure that when there were labels exceeding the expected number calculated from the offset, the extra labels were given the value zero. Though, this made the in-between label areas exclude data, it was just a few pixels that were deleted and had little impact on the overall model. This required a thoroughly search in the code by printing out every important data transformation to detect where the code changed.

Error handling

Then CUDA out of memory errors occurred. The earlier error handling was solved on a local computer. To solve the new CUDA error, the batch size was reduced from 2 to 1 to ease the memory reserve, but without any improvements. It was decided to conduct further training of the model on Google Colab for better performance with their available GPUs.

The RGBD dataset of sweet pepper crops and the reference scripts were uploaded to a Google Drive and further testing was continued there. It was then discovered that multiple errors spawned from the reference scripts. The latest stable PyTorch version was 1.11.0, but the reference scripts from the tutorial was from an earlier PyTorch version. An attempt was made to use the latest reference scripts and error handle the outcomes, but this ended up with the need to change code snippets in the main PyTorch library. This approach was dropped, because in turn it could potentially disturb the programming base for the next models. The reference scripts supporting the latest stable PyTorch version and Torchvision version of 0.12 was selected conclusively.

Training the program on Google Colab with CUDA, made the program jump processes. The program was though training fine with the CPU on the local computer. Training on the CPU would take closer to 15 hours if the same pace was kept, while training on the GPU still gave CUDA memory errors. So, it was not a practical solution. From the errors in Colab it was guessed that something in the way the data were processed was the fault of the jumping. After trying a lot of different approaches, it was discovered that Colab required the listed files to be sorted before it could process them.

A fault in the annotations were found for *frame_2019_10_1_10_0_59_656095,* where the labels 7 and 8 were jumped and thus making the algorithm get an error when generating

bounding boxes. The sample was removed from the folder. Training went smoothly after this.

Hyperparameter selection

For the tuning of hyperparameters it seemed right to take inspiration from the agriculture papers and then vary some of the parameters to see if the model improves on the RGBD dataset of sweet pepper crops.

The original Mask R-CNN paper, Halstead et al. sweet pepper experiments and the improved Mask R-CNN on cherry tomatoes were the basis for the hyperparameter selection:

- Learning rate: 0.02, batch size: 16, momentum: 0.9, weight decay: 0.0001, epochs: 160000, backbone: ResNet-50, hidden layers: unknown (He, et al., 2021).
- Learning rate: 0.001, batch size: 4, momentum: 0.9, weight decay: 0.0005, epochs 250, backbone: unknown, hidden layers: 256 (Halstead, Denman, Fookes, & McCool, 2020).
- Learning rate: 0.001, batch size: 8, momentum: 0.9, weight decay: 0.0001, max epochs: 19500, backbone: unknown, hidden layers: unknown (Xu, et al., 2022).

Since the research shared similar hyperparameters, Halstead et al. was the followed example because their research also handled the RGBD dataset of sweet pepper crops, and their model was prepared for fewer parameters. Although, in the report, it was stated that their hyperparameters were mostly based on the PyTorch library's defaults (Halstead, Denman, Fookes, & McCool, 2020, p. 4).

How the training was conducted

Training was completed from scratch and with pretrained COCO weights. They both shared the pretrained backbone of ResNet-50, had 256 hidden layers and were trained for 50 epochs on a batch size of 4. The optimizer was set to stochastic gradient descent (SGD) with a momentum of 0.9 and a weight decay of 0.0005, which in PyTorch is a mini-batch SGD. A learning scheduler was set to the tutorial's default value with a step size of 3 and a gamma of 0.1. This made the model multiply the learning rate with 0.1 every 3rd epoch. *get_transform* in *mrcnn_train.py (21)* randomly flipped the images to heighten variation in the training set.

Training from scratch

Initiating training with a learning rate of 0.001 and training for 25 epochs made the learning rate drop drastically and settled the box and mask predictions at a mAP of 1.1% and 1.4% (IoU=50). Since the learning scheduler made the learning rate drop remarkably, it was decided that further learning rate testing was without the learning scheduler.

Training was initiated with a learning rate of 0.01, 0.001 and 0.0001 for 50 epochs to analyse

the performance of the model. From the testing it was discovered that the learning rate of 0.01 and 0.001 had close performance results. Comparing the IoU=50 with IoU=50:95, the model with a learning rate of 0.001 was more certain with its predictions.

Training with pretrained weights

The model was trained with pretrained Mask R-CNN model (PyTorch, 2022) weights from COCO train2017 (COCO, 2022). Few custom configurations were available for the pretrained model. Training was initiated with a learning rate of 0.01, 0.001 and 0.0001 for 50 epochs to analyse the performance of the model.

Evaluation

The evaluation metrics was generated from the PyTorch library and the Pycocotools. The evaluation is not changing any hyperparameters or influencing the training. The metric tests the performance of the training each epoch on a separate test set. This prints out a table of the predicted boxes and masks with mAP scores for common IoU thresholds for both precision and recall. There were few options to acquiring the loss, it was decided to include the last loss rate for epoch 50 in each run to give an impression on what number the training loss reached.

Visualization

The script *mrcnn_test.py (21)* was used to load the trained models. The function borrowed code snippets from the training file to initiate model configurations. This was tested on one test sample. The box and mask predictions and ground truths were visualized in four Matplotlib subplots. A score threshold was implemented to only show instances over the threshold of 50% certainty. This accumulated from the classification probability measure. For the mask visualization, a threshold of 95% certainty was implemented to show the pixel values that the model was most certain belonged to the respective instance. The visualization would have been better if the image predictions were combined, but because of limited time a solution to the problem was not considered.

4.3.4 YOLACT

In object detection, YOLO has had a streak of noteworthy performance over the past years with a series of improving models. YOLACT derives from the YOLO architecture and adds to it instance segmentation capabilities.

Deciding the implementation code

There were multiple versions of YOLACT when searching the internet, on Papers with Code one of the highest scoring libraries to implement the algorithm was the OpenMMlab library (OpenMMLab, 2022). Their MMDetection toolbox had pretrained models for instance segmentation purposes, and from here the YOLACT was selected as the second detection model.

Pre-processing

For the model to work properly, it was decided to transform the dataset into the COCO JSON annotation format. The complete transformation took inspiration from MMDetection's own tutorial (MMDetection, 2022) and from the earlier Mask R-CNN implementation structure and was joined in the *convert_pepper_to_coco* function from the *annotation_to_coco.py (21)* file. The function gathered the sample's height, width, filename, annotated area and its bounding boxes into a Python dictionary, which was saved as an uploadable JSON file for further processing. This was completed for two subsets, training and testing.

Each MMDetection model had its own configuration file, where preparations were made before training the model. Parts of *yolact_train.py (21)* inherited attributes from the configuration file and allowed for model parameters and hyperparameters to be customized.

Error handling

A problem occurred with the creation of the JSON file. The model was not capable of handling the mask data in the same manner as the previous Mask R-CNN architecture did. Instead of an array with the mask's pixel positions it required the data to be as a polygon with just the coordinates for the mask contour. This was realised with an image processing library called Scikit-Image and its *find_contours* function (Walt, et al., 2014).

The tutorial was based on a Mask R-CNN configuration file, a few of the variables had to be changed to run training smoothly. Understanding how the configuration file of *yolact_default_config.py (21)* was structured and how the data was processed took some research. Few online examples were directly related to the YOLACT configuration file structure. The script *yolact_train.py (21)* was used to change certain values in the configuration file.

Model configurations

Model attributes mostly followed the default settings from the standard configuration file as detailed in the *yolact_config_default.py (21)*, which had similar values as established in the Model training section. Dataset pathways and number of classes was changed to fit the RGBD dataset of sweet pepper crops. A warmup for the learning rate was turned off, to have a more

consistent learning rate through the training. This could have been tuned on for better performance in the early and late stages of training but was decided not to be tampered with for simplicity and saving time.

Evaluation metric

The evaluation metric was mean average precision (mAP). This was built into to the model and shared a similar output structure as the COCO evaluation standard. The loss was not included, because it was uncertain how it was calculated.

How the training was conducted

Like the Mask R-CNN approach, YOLACT was trained from scratch and with pretrained COCO weights. The backbone was based on ResNet-50 and the model was trained for 50 epochs on a batch size of 4. The optimizer was set to stochastic gradient descent (SGD) with a momentum of 0.9 and a weight decay of 0.0005. Different learning rates were tested for optimal performance on the dataset.

Training from scratch

The training was planned for a learning rate of 0.01, 0.001 and 0.0001 to generate the best performance in the short 50 epochs time window. The training behaved a bit differently compared to Mask R-CNN. Starting training generated an error message stating that the classification score reached infinite or NaN. Changing the batch size, lowering the loss weights or removing the learning rate warmup layer still produced the same error message. Lowering the learning rate to 0.0001 made the training start. Running through 50 epochs, the model converged slowly.

Training with pretrained weights

Training was initialized with pretrained weights and the learning rates of 0.01, 0.001 and 0.0001. Using a static training rate of 0.01, without the warmup parameters, resulted in the previous infinite classification score error. Though, when initiating training with the warmup the model trained with a 0.01 learning rate. The warmup parameter increases the learning rate at the first epoch before it decays down. To keep the training fair to Mask R-CNN's hyperparameter settings, the static learning rate was favoured. The learning scheduler was also turned off. The model was trained with a learning rate of 0.001 and 0.0001 for 50 epochs.

Visualization

The visualization file of *yolact_test.py (21)* used parts of the model training configuration of *yolact_train.py (21)* to initiate a model instance that was based on the newly trained model. The generation of figures was handled by the MMDetection function, *show_result_pyplot*. The

test image from the test dataset was used to visualize the model predictions.

4.3.5 QueryInst

QueryInst based its architecture on queries for each instance, which made it produce higher mAP results on the COCO dataset compared to the Mask R-CNN and YOLACT.

Implementation

The model architecture was loaded from the MMDetection library, and the implementation followed the same structure as with YOLACT. The earlier configurated JSON annotation files for training and validation were used as input. *queryinst_train.py (21)* initiated the training by preparing and inheriting attributes from a configuration file relevant for the QueryInst model. This was similar to the *queryinst_default_config.py (21)* file. The configuration file deviated from the YOLACT configuration file, but defined classes and optimizer related variables in the same manner. The file itself was selected from four possible model architectures that was based on different backbones. Requirements are stated in Appendix (7).

Evaluation metric

Evaluation was as before produced from MMDetection's replication of the COCO evaluation table showing the scores in mAP.

How the training was conducted

QueryInst was from default based on the ResNet-50 backbone and used the optimizer of Adam with a learning rate of 0.0001 and a learning rate warmup configurator. The optimizer was changed to SGD and the learning rate configurator was turned off, to keep a static learning rate. It was planned to train the model for the learning rates of 0.01, 0.001 and 0.0001 from scratch and with pretrained weights.

Training

The model was trained with the three learning rates for 50 epochs each. Though, without showing any improvement on the evaluation score and staying at zero mAP through the full run. Training was finalised for 50 epochs for each configuration to see if the models just needed more time to converge to a minimum. An attempt was made to change the number of model stages from 6 to 1 and 3, without any difference to the score. The number of query proposals was changed from 100 to 50, 10 and 1 but without any success either. The model configurations were changed back to its default settings and a new run was initiated. As before the model showed no progress on the score. Number of stages and number of query

proposals was changed again to see if it impacted the score. The different approaches were tried with and without pretrained weights.

It was also discovered that the line that previously for YOLACT saved the model checkpoint was not functioning as intended. Being able to visualize the results could have revealed if the problems derived from the evaluation metric, but without a saved model that was not possible. The files config file and the pretrained model was checked again to ensure there was not a mix up. The MMDetection version was 2.24.1 and the latest MMCV version of 1.15.1 was checked and the versions should be compatible. Github was searched for all known issues related to QueryInst on the MMDetection page. No issue matched the zero evaluation score. From QueryInst's report it was stated that the training took 36 to 50 epochs so the training time should not be the issue. Although it was announced from a user in the issues page on MMDetection Github that the QueryInst used a long time to train. Therefore, a last attempt was initiated to run the model on default settings for 100 epochs to see if it made any difference. After 100 epochs the evaluation was still on zero as detailed in Appendix (20). QueryInst was ultimately dropped from the comparison.

4.4 3D representation

In lack of a better ways of representation, the depth was visualized besides the model predictions. This was realized with *visualize_annotation.py (21).* Although the visualization does not present too much information. The image values from the annotations and the depth can easily be combined to allow each object in the image be available as (x, y, z) image coordinates. As shown in Fig.6 with the RGBD visualization. A NumPy *dstack* function was used to add the depth value as the fourth channel in the image array.

5 RESULTS

This chapter seeks to concisely present the results that were detailed in the methodology chapter.

5.1 Dataset visualization

The dataset files are inspected with various Python tools to ensure that they have RGB, depth and annotation files that are correct.

RGB and depth visualization

The RGB and depth data were visualized with Matplotlib in *Fig.1* to ensure that their pixel placements matched, and that the depth data was correct.

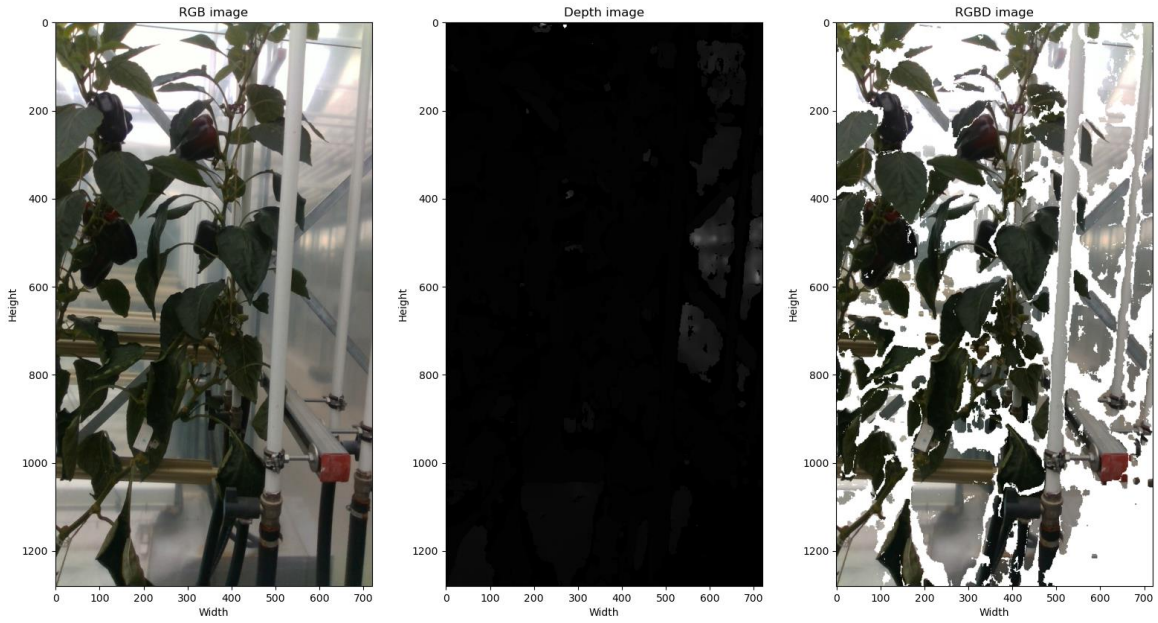

Figure 6: RGB and depth visualized.

Their pixel placements look to match in *Fig.6, RGBD (right)* otherwise the overlay would have been shifted. A *print(RGBD.shape)* command declares the image height as 1280 pixels and width as 720 pixels. The depth image and the RGBD image gave indications of missing data by exposing areas that where completly white. The presumed missing data were further investigated in *Fig.7* by shaping the frequency of the pixel values into a histogram representation.

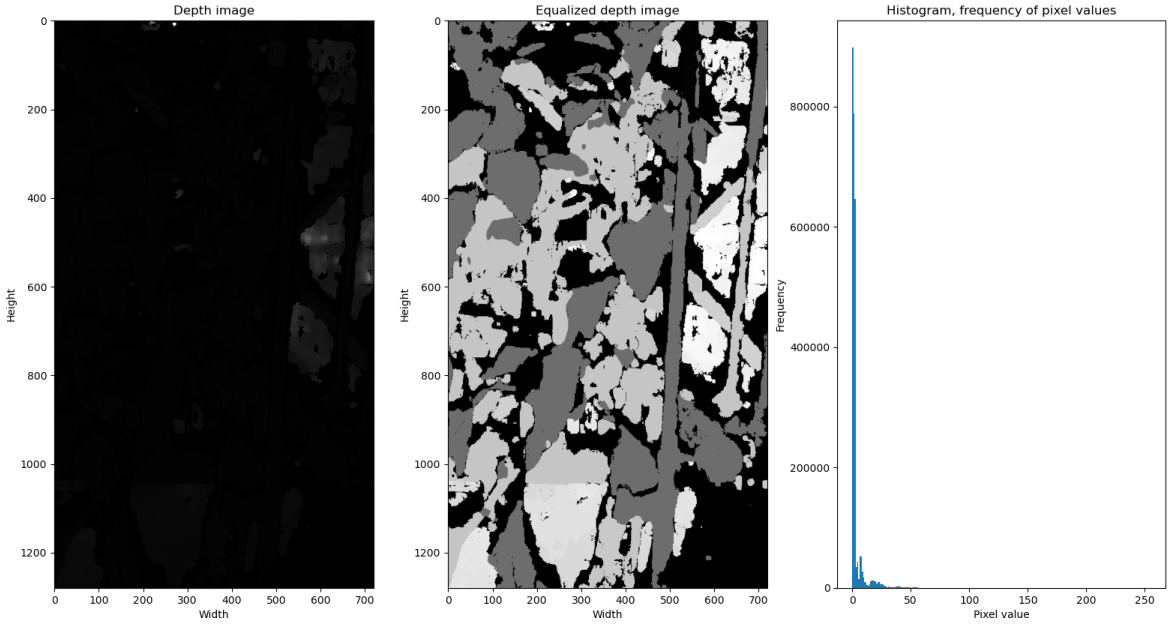

Figure 7: Depth equalized and histogram plotted.

The histogram representation shows a high number of pixels having the value of zero. This confirms that the dataset samples have areas where the data lacks visual representation. Multiple samples were tested, and they also showed the same tendency. By manually looking at the samples, the missing data did occur in areas around the leaves and crops, which impacts the quality of the depth information. From known issues with the RGBD sensors, the missing data could be caused by out-of-range object parts, disadvantageous illumination conditions and moisture on the lens.

Mask and bounding box visualization

The dataset annotations were manually inspected by generating masked RGB images with the crop instances from *subplot_mask_2D (4)* and manually comparing these with the raw annotation files.

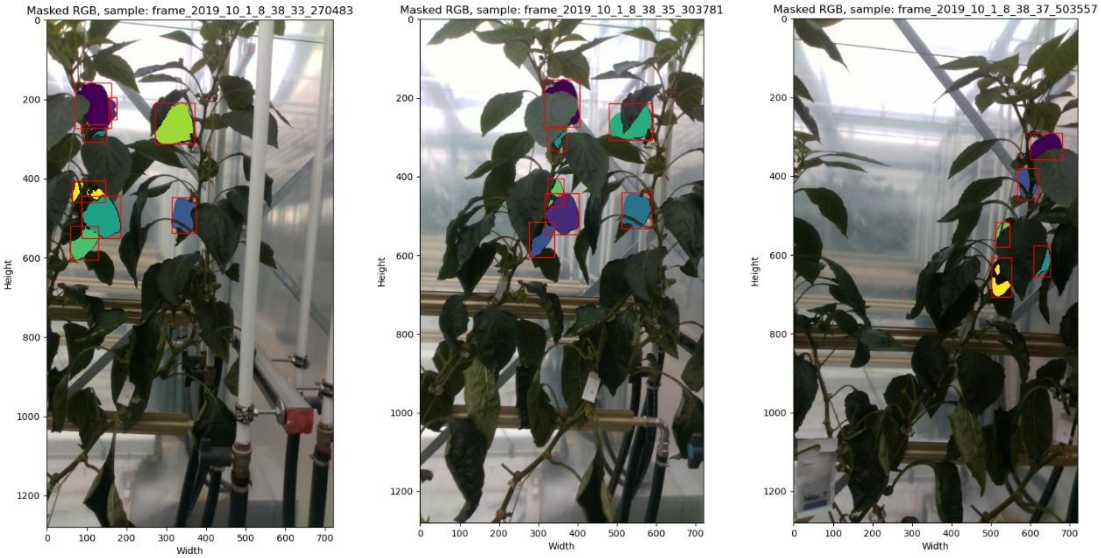

Figure 8: Annotations visualized.

In *Fig.8* three generated masked samples were visualized to be compared and counted. Examples 1, 2 and 3 had respectively 8, 7 and 5 instances of crop rightly accounted for. In the ten samples inspected no unaccounted-for crops could be found. It was on these discoveries presumed that the dataset annotations were correctly labelled and placed in the images. After initiating training, a mistake in the annotation was found in sample *frame_2019_10_1_10_0_59_656095*, which was therefore removed from the training set.

5.2 Model training and evaluation

The training was realised with and without transfer learning from a pretrained model. Further details on how the results were accomplished can be found in Chapter 4.

5.2.1 Mask R-CNN

Mask R-CNN was trained, evaluated and visualized from scratch and with pretrained COCO weights.

Training Mask R-CNN

Training Mask R-CNN produced the results in Tab.1.

Model NR:	Learning rate:	Loss:	Box mAP (IoU=50):	Box mAP (IoU=50:95):	Mask mAP (IoU=50):	Mask mAP (IoU=50:95):
M01	0.01	0.52	64.7%	41.7%	69.4%	42.7%
M02	0.001	0.92	64.3%	38.5%	69.0%	38.8%
M03	0.0001	1.69	23.4%	7.6%	27%	11.8%
M04PRE	0.01	0.25	61.7%	42.4%	68.1%	45%
M05PRE	0.001	0.40	61.9%	40.9%	66.6%	42.5%
M06PRE	0.0001	1.23	64.3%	38.5%	69.0%	38.8%

Table 1: Mask R-CNN results, Appendix (8,9)

Model M01 and M02 had similar results. The training loss was marginally lower on the model M01. For the pretrained models, M04PRE had the highest score, though the two other models were closely after. M06PRE showed a higher score for the IoU=50, but a slightly higher loss as well. Comparing the models with and without weights, they are close in score. M04PRE going out as the overall best performing.

Visualizing the models generated the results in the figures below.

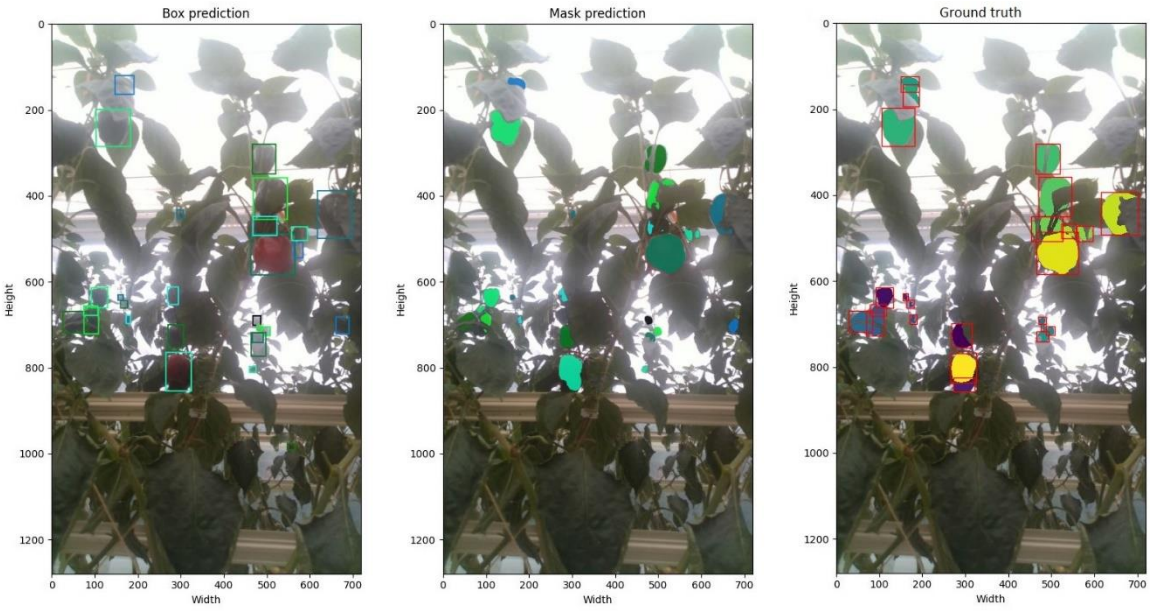

Figure 9: Model M01 results.

Model M01 predicted in all 28 instances. The model wrongfully placed a few masks, coupled two instances together and was unsuccessful in localizing 4 out of the total 24 instances.

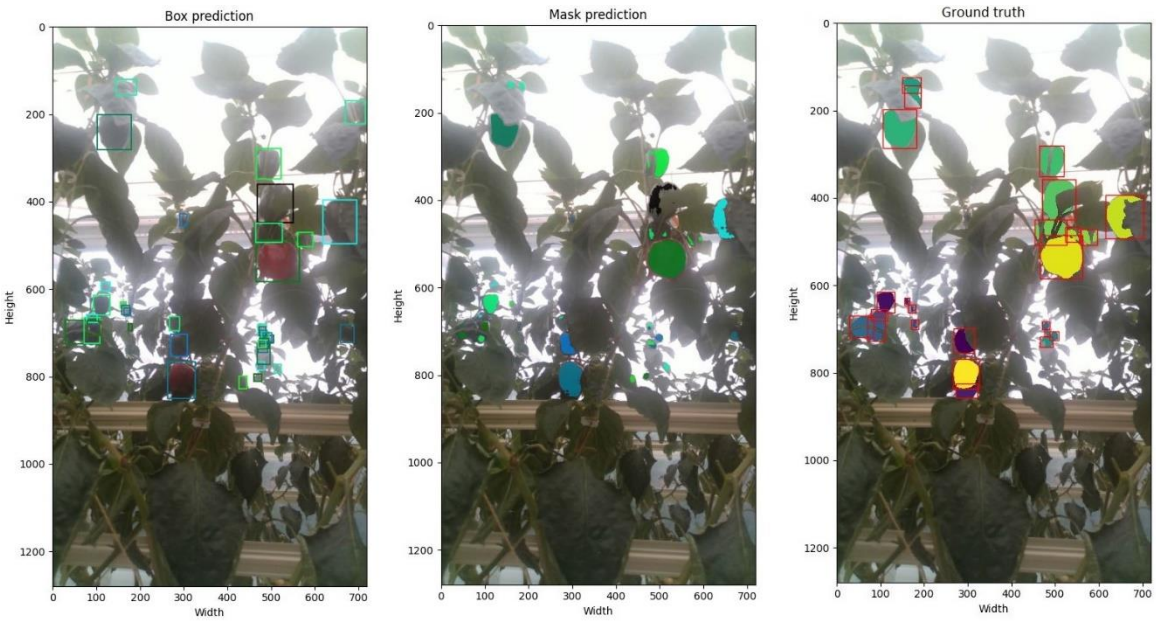

Figure 10: Model M02 results.

Model M02 predicted in all 38 instances. The model wrongfully placed 7 instances, separated 2 instances into multiple objects and was unsuccessful at localizing 5 instances. Compared to model M01, model M02 had slightly poorer performance.

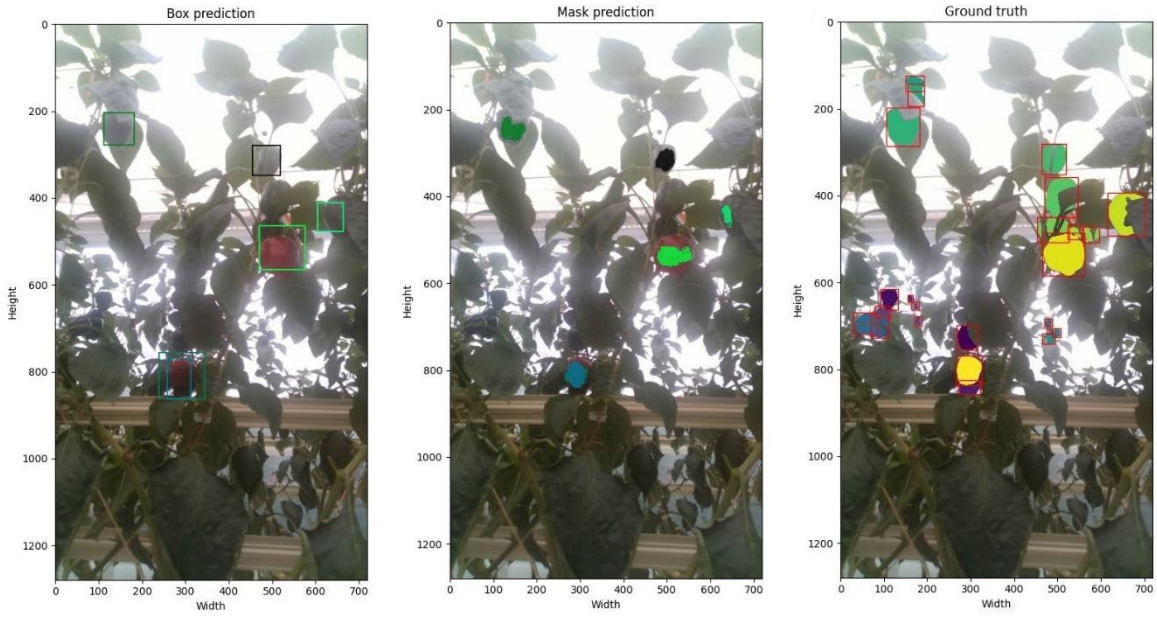

Figure 11: Model M03 results.

Model M03 predicted in all 6 instances. The model wrongfully predicted 0 instances but missed out on 18 instances and the mask predictions were poorly matched.

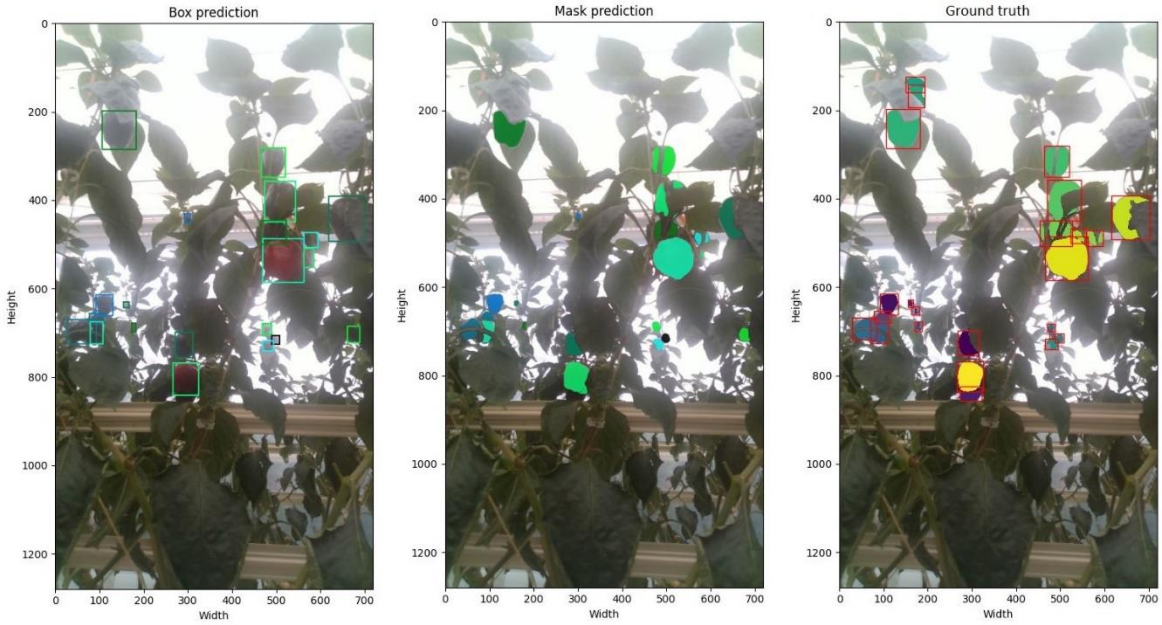

Figure 12: Model M04PRE results.

Model M04PRE predicted in all 21 instances. The model wrongfully predicted 3 instances and missed out on 5.

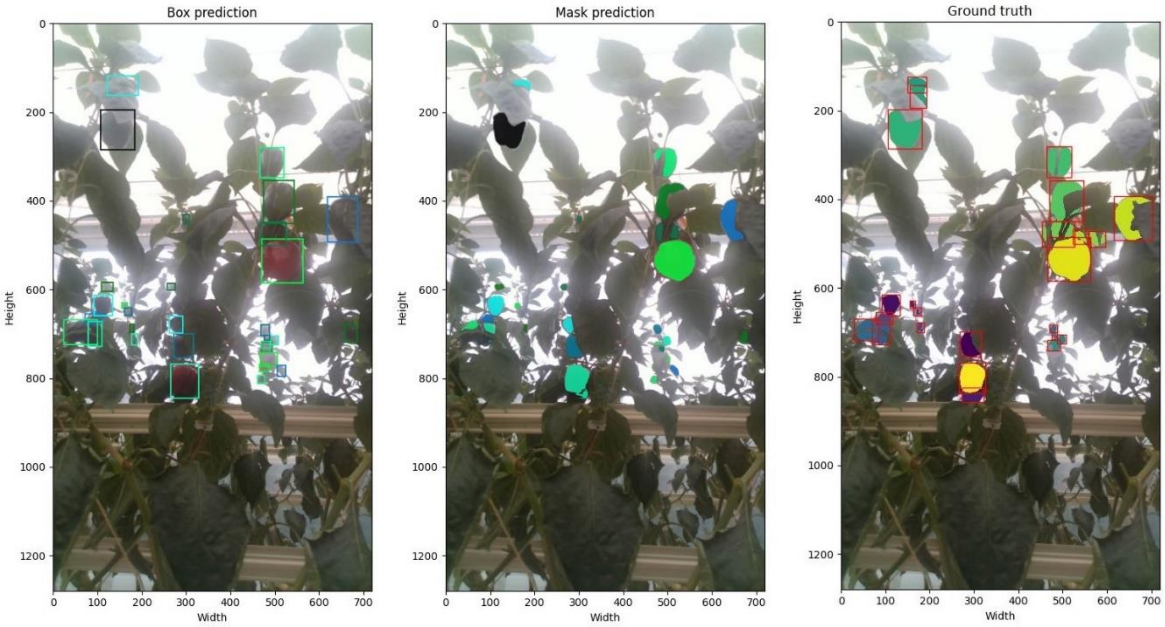

Figure 13: Model M05PRE results.

Model M05PRE predicted a total of 31 instances. The model wrongfully placed 8 instances and missed out on 3. Similar performance as the previous M04PRE.

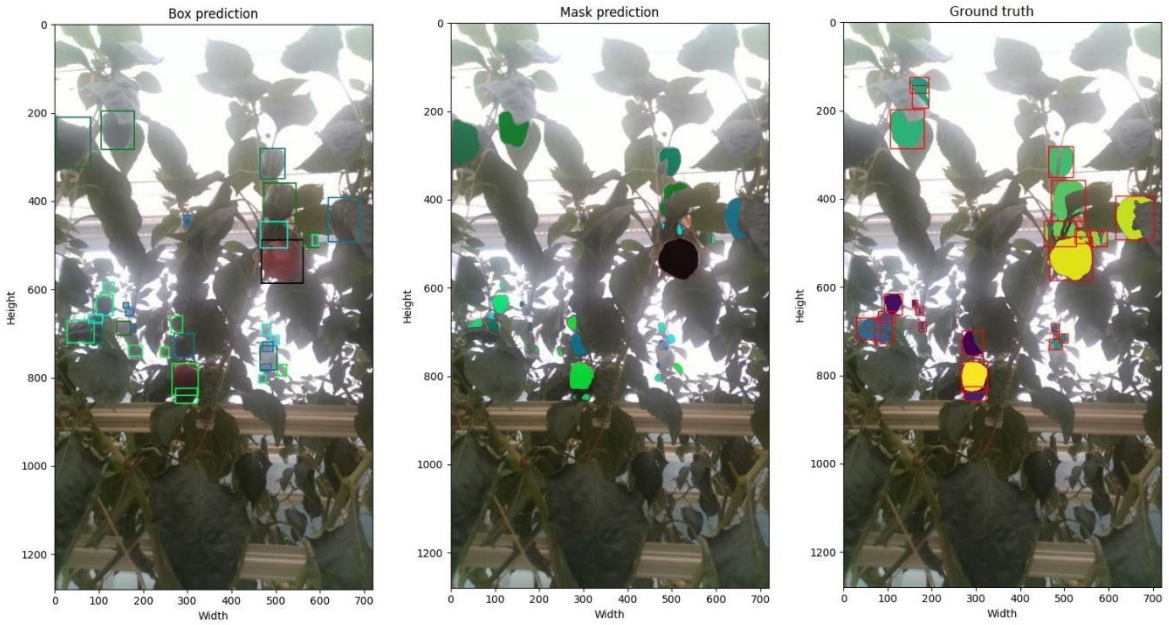

Figure 14: Model M06PRE results.

Model M06PRE predicted in total 32 instances. The model wrongfully placed 8 instances and missed out on 4. Again, the performance rivals the two previous model implementations, but with slightly lower mAP score.

5.2.2 YOLACT

YOLACT was trained, evaluated and visualized partly from scratch and with pretrained COCO weights.

Evaluation

Training YOLACT from scratch was only possible for a learning rate of 0.0001 and lower, while with pretrained weights it was possible for a learning rate of 0.001 and 0.0001. The produced results are shown in *Tab.2,* where the model number differs the pretrained with the letters PRE.

Model NR:	Learning rate:	Box mAP (IoU=50):	Box mAP (IoU=50:95):	Mask mAP (IoU=50):	Mask mAP (IoU=50:95):
Y01	0.0001	0.1%	0%	0%	0%
Y01PRE	0.001	55.3%	33.3%	57.0%	30.1%
Y02PRE	0.0001	49.7%	30.0%	55.1%	31.9%

Table 2: YOLACT results, Appendix (10).

Model Y01 had very poor mAP score. Could be that it would have achieved better performance if it was trained for longer. Y01PRE and Y02PRE had closely related scores. The original pretrained model scored a 29.0% mAP on the COCO dataset (Bolya, Zhou, Xiao, & Lee, Github, 2022). Though, the scores are not directly comparable it gives the impression that Y01PRE scores high.

Visualization

Model Y01 was dropped from visualization due to its poor performance. The pretrained models generated the results in Fig.15.

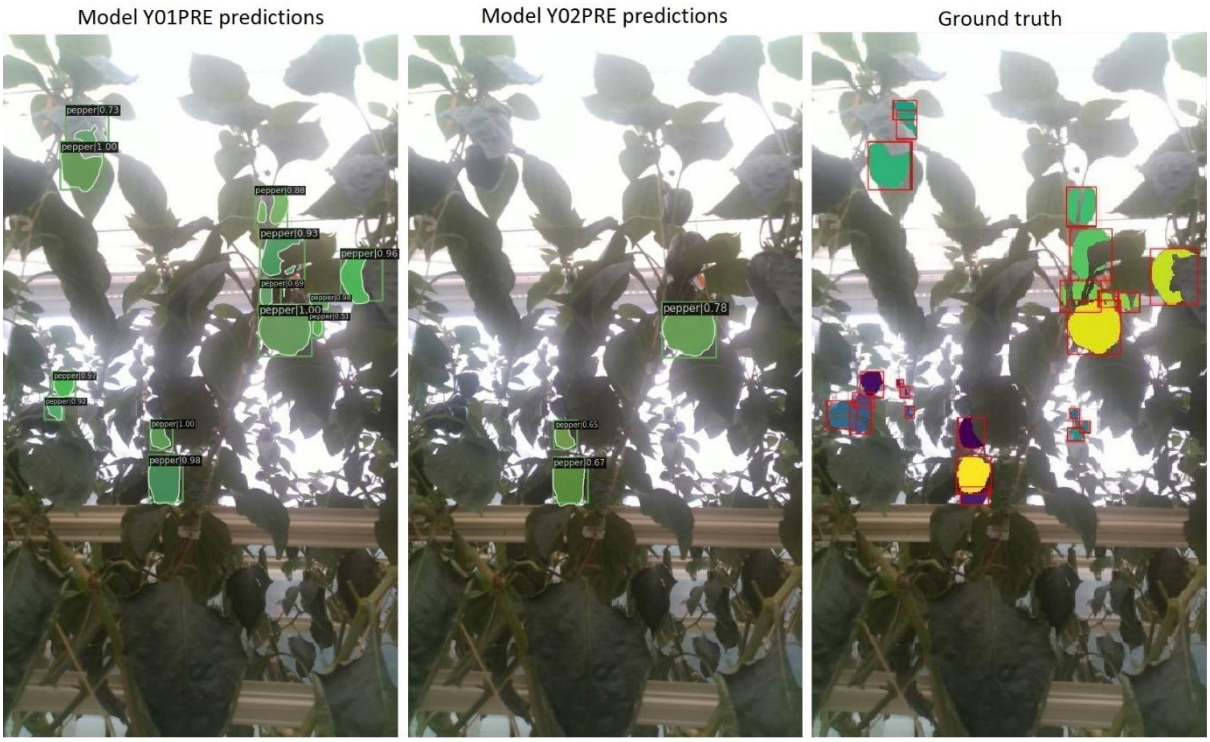

Figure 15: YOLACT visualization.

Model Y01PRE predicted in all 13 instances. The model wrongfully placed two instances and was unsuccessful in localizing 13 out of the total 24 instance. Model Y02PRE predicted in all 3

instances. It wrongfully places zero but combined two instances into one. Even though their mAP scores were closely related, the visualization of Y02PRE was considerably poorer. Comparing Y01PRE to the Mask R-CNN results reveal a much pickier model with fewer wrong instance predictions but failed to recognize smaller crops.

5.3 Model comparison

The best performing models from each architecture in section 5.2 are compared for a closer analysis.

Model NR:	Learning rate	Loss	Box mAP (IoU=50)	Box mAP (IoU=50:95)	Mask mAP (IoU=50)	Mask mAP (IoU=50:95)
M04PRE	0.01	0.25	61.7%	42.4%	68.1%	45%
Y01PRE	0.001	N.A.	55.3%	33.3%	57.0%	30.1%

Table 3: Mask R-CNN and YOLACT score comparison.

From the mAP scores Mask R-CNN's model M04PRE scored higher than the YOLACT model. In addition to converging at a lower learning rate. Google Colab varies the available resources when training depending on peak hours. Therefore, comparing the training times are just mere estimations. YOLACT trained usually under 15 minutes and Mask R-CNN from 1 hour and more.

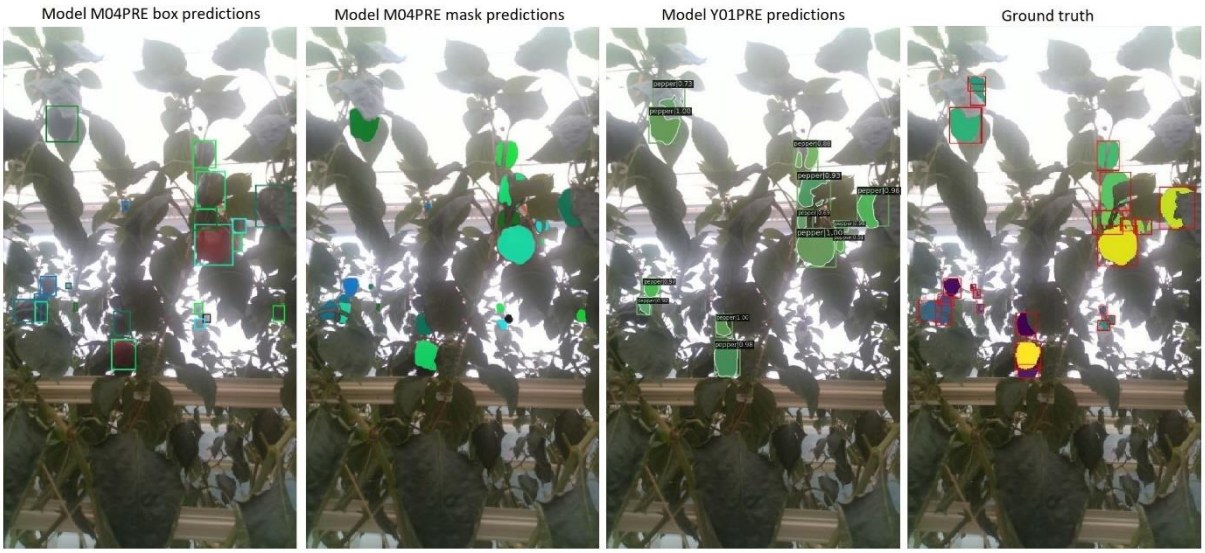

Figure 16: Mask R-CNN and YOLACT visual comparison.

Comparing the visualized predictions, Mask R-CNN matched all the instances that YOLACT found, plus the smaller instances, which YOLACT totally missed.

5.4 3D representation

The predictions are visualized together with the depth data.

Figure 17: 3D representation of models.

In Fig. 17, the depth data is shown as a grayscale image where the black colour equals zero distance from the capture device and lighter colours resemblance pixels farther away. As seen, the depth data is lacking information in all the black areas of the image. This looks to be valuable depth information that would have been practical for a robotic harvester.

6 DISCUSSION

Chapter 1 introduced food insecurities as a rising problem and claimed that improving agriculture detection technology could help mitigate the problem. It was discovered that current crop detection systems for robotic greenhouse harvesters are not a profitable option for farmers. A solution was proposed, improving the detection system of the robotic harvester could give the technology the edge it needed to be an efficient alternative for farmers. The project plan was to train Mask R-CNN, YOLACT and QueryInst on a RGBD dataset of sweet pepper crops and compare their performance. As a final, the results were going to be represented in 3D space.

Project criteria

A few project criteria were set from the proposed solution. A dataset with crops cultivatable in a greenhouse environment was to be preferred. This dataset should also include annotations that could be used for instance segmentation and RGBD data for 3D representation. The models being DNNs with instance segmentation capabilities. All set in the scenario of a robotic greenhouse harvester.

On the project's way, decisions were made to select the RGBD dataset with sweet pepper crops. This was a befitting dataset to the criteria, because it had RGBD information, annotations for instance segmentation and greenhouse cultivatable crops. There were few other options as the public agriculture dataset survey proved (Lu & Young, 2020). The dataset had few samples to train on, 99 to be precise, after removing a bad annotation. Initially it was thought that this would generate poor results, but as seen in Tab.1, 2, and 3 the models reached functional scores for both architectures with pretrained weights. Where Mask R-CNN also scored high when training from scratch. The two model architectures had the data augmented before processing, these happen at random (Stackoverflow, 2022). It is therefore hard to calculate how much this impacted the model, but at least it multiplied the amount of training samples. The dataset had just one crop type to locate, which is very specific when grading its practicality for farmers. From Chapter 3, the reports also based their research on one crop: sweet peppers (Halstead, Denman, Fookes, & McCool, 2020), tomatoes (Xu, et al., 2022), while one report detected peppers, eggplants and guavas at the same time (Lin, Tang, Zou, Xiong, & Fang, 2019). Developing detection systems that could generalize over several crop species would be optimal.

About the decision of using DNNs for detection tasks. Chapter 3 revealed that most object detection research are currently utilizing supervised DL to achieve the highest performance. Papers with Code instance segmentation benchmarks are heavily influenced by DL models (PaperswithCode, 2022). Current agriculture trends states that DL is boosting robotic

perception and the detection algorithms (Kootstra, Wang, Blok, Hemming, & Henten, 2020, p. 100). Lin et al. (Lin, Tang, Zou, Xiong, & Fang, 2019) based their detection on simpler algorithms that scored high on mAP but had a processing time of 4.70 second per detection. This approach would still need improving to rival DL.

Initially, the project wanted to have a higher focus on utilizing the 3D data. Finding research that handled instance detection in 3D was hard to come by as discovered in section 4.3.1 Model selection. The limited research on the field conveys to less available information online, which makes program development potentially more difficult to work out. The different approaches for processing 3D information are found to be by: Joining a 2D prediction with 3D information (Li, Feng, Qiu, Xie, & Zhao, 2022), process point clouds with fully 3D-based detection models (Li, et al., 2022) and process the depth information as an extra layer for 2D detectors (Xu, et al., 2022). Processing RGB and depth information in a 2D-based detector suggests being a possible middle way for current technology. This way, the optimized 2D-based detectors have a lead on regarding speed and accuracy. Fully 3D-based networks are increasing with cheaper devices and a bigger acquirable 3D dataset (Ahmed, et al., 2019). 3D representation brings an opportunity to distinguish crops apart and heighten the model's ability to localize crops behind occlusion. This is an ongoing challenge in agriculture detection (Kootstra, Wang, Blok, Hemming, & Henten, 2020, p. 96). 3D representation could potentially support a robotic harvester and its gripper to reach target crops (Li, Feng, Qiu, Xie, & Zhao, 2022). Processing the depth training as an extra channel would have been the optimal approach for the project.

Instance segmentation was a favoured trait, because it would produce detailed pixel masks for multiple object instances (Gu, Bai, & Kong, 2022, p. 3). There was not found any comparable alternative that offered the same detailed information for multiple instances. The discovered literature was sticking to this approach as well.

Results

The dataset was visualized to validate if it passed the project criteria. This was achieved by inspecting and visualizing the different files and their data. The RGB and annotation files were found to be of functional value, but the depth data lacked data points. This made it inaccurate as discovered in sections 5.1 and 5.4. The original idea of 3D representing the data, preferably with the annotations was not fulfilled. Matplotlib had few options for 3D visualization, the large set of datapoints made the program slow to work with. The RGB data array and depth was though combined as visualized in the RGBD image in Fig.6.

The model plan was to train and compare two established detection architectures against a newer architecture. This was partly achieved. Mask R-CNN and YOLACT were two detection models commonly implemented both for general detection tasks and in agriculture research (Zaidi, et al., 2021). QueryInst was developed in 2021 and showed promise on benchmarks (Fang, et al., 2021). The Mask R-CNN trained with and without pretrained weights and could

show to practical detection capabilities in Tab.1 and 2. YOLACT could not be trained with learning rates higher then 0.0001 without, and 0.001 with pretrained weights. This was caused by an error in the initialization of training and could stem from the default model settings pushing a parameter related to classification too high. QueryInst was trained with numerous hyperparameter combinations but could not reach a score over 0 mAP. Comparing the results for Mask R-CNN and YOLACT, revealed that Mask R-CNN had a slight better performance in Tab.4. It being a two-stage detector, it was favoured from the start to perform better (Xu, et al., 2022, p. 5). YOLACT was close in score, while its training time was down to a fourth of Mask R-CNN. The visualization showed that YOLACT had fewer wrong predictions and failed at recognizing small objects. This is a known shortcoming stated in YOLACT (Bolya, Zhou, Xiao, & Lee, YOLACT, 2019). Since there already has been minor flaws in the dataset, it is possible that the Mask R-CNN predictions captured unannotated instances.

The results show Mask R-CNN's and YOLACT's potential to produce results from a small dataset based on sweet pepper crops. The results failed to bring forth an eventual better performing model and the comparison between established and newer architectures flopped.

Research

After the idea of agriculture crop detection, a research plan was organized. Related papers to object detection in agriculture was gathered. From the literature review an impression of the field and where the project could contribute was formed. Topics like DL, instance segmentation and 3D data processing takes time to understand all the directions and methods available. To produce the results and the approach to inspecting the dataset, was influenced by the practices described in section *2.1.4 Deep learning workflow* (Elgendy, 2020). The programs and the tools that was utilized under the project development was inspired and based off tutorials related to the model architectures. GitHub (GitHub, 2022) and Stackoverflow (Stackoverflow, 2022) was visited for problem solving program errors.

In hindsight, the project could have been shorted down to just Mask R-CNN training with added weight on investigating the depth values as an extra channel fed into the detection model. A downfall to this project was too many complex topics that took hours to navigate.

7 CONCLUSION

The main goal of the thesis was to explore object detection with instance segmentation in relation to agriculture. The project set out to train and compare the architectures of Mask R-CNN, YOLACT and QueryInst. Three main tasks where set to be investigated.

Validate the RGBD dataset of sweet pepper crops

The dataset files were first inspected with various Python tools to acquire their data specifications. This revealed data and was necessary to visualize the depth and annotation files. The RGB files were a intact and functional. The annotations files showed the masked instances and their belonging bounding boxes when visualized. There was a flaw in one of the samples, which was removed from the dataset. The depth data was missing valuable pixel information in image areas, which could produce inaccurate depth coordinates.

Train Mask R-CNN, YOLACT and QueryInst

Training was completed for Mask R-CNN, YOLACT and QueryInst, but the latter failed to produce a mAP score over zero. Hyperparameter tuning was initiated by experimenting with learning rates in the range of 0.01, 0.001 and 0.0001 for the intention of improving performance. Major model architecture components shared by the models were preserved to create an equal footing. The training was initiated from scratch and from pretrained weights to investigate if it improved the performance.

Evaluate and compare the model performances

The training results were evaluated with the mAP metric and visualized. The evaluation metric showed that Mask R-CNN's mAP of 45% passed YOLACT's mAP of 30.1% for the mask predictions. For the box predictions Mask R-CNN mAP of 42.4% did better than YOLACT's mAP of 33.3%. From visualizing the models, Mask R-CNN had several correct predictions, while YOLACT predicted correct it missed a few instances, especially the smaller ones.

Mask R-CNN and YOLACT show promising results on a small dataset of sweet pepper crops.

8 REFERENCE LIST

Aggarwal, C. C. (2018). *Neural Networks and Deep Learning.*

Agrobotics. (2022). Retrieved from https://www.agrobot.com/e-series

Ahmed, E., Saint, A., Shabayek, A., Cherenkova, K., Das, R., Gusev, G., & Aouada, D. (2019). *A survey on Deep Learning Advances on Different 3D Data Representations.*

Anaconda Software Distribution. (2020). *Anaconda Documentation.*

Arad, B., Balendonck, J., Barth, R., Ben-Shahar, O., Edan, Y., Hellström, T., . . . Tuijl, B. v. (2020). *Development of a sweet pepper harvesting robot.*

Autz, J., Mishra, S. K., Herrmann, L., & Hertzberg, J. (2022). *The pitfalls of transfer learning in computer vision for agriculture.*

Bac, C. W., Henten, E. J., Hemming, J., & Edan, Y. (2014). *Harvesting Robots for High-value Crops: State-of-the-art Review and Challenges Ahead.*

Baudoin, W., Nono-Womdim, R., Lutaladio, N., Hodder, A., Nicolás Castilla, C. L., Pascale, S. D., . . . Duffy, R. (2013). *Good Agricultural Practices for greenhouse vegetable crops.*

Bolya, D., Zhou, C., Xiao, F., & Lee, Y. J. (2019). *YOLACT.*

Bolya, D., Zhou, C., Xiao, F., & Lee, Y. J. (2019). *YOLACT++: Better Real-time Instance Segmentation.*

Bolya, D., Zhou, C., Xiao, F., & Lee, Y. J. (2022). *Github*. Retrieved from YOLACT benchmark: https://github.com/open-mmlab/mmdetection/tree/master/configs/yolact

Chebrolu, N., Lottes, P., Schaefer, A., Winterhalter, W., Burgard, W., & Stachniss, C. (2017). *Agricultural robot dataset for plant classification, localization and mapping on sugar beet fields.*

COCO. (2022). *Dataset 2017*. Retrieved from https://cocodataset.org/#home

COCO. (2022). *Evaluation metric*. Retrieved from https://cocodataset.org/#detection-eval

Elgendy, M. (2020). *Deep Learning for Vision Systems.*

Fang, Y., Yang, S., Wang, X., Li, Y., Fang, C., Shan, Y., . . . Liu, W. (2021). *Instances as Queries.*

Gené-Mola, J., Vilaplana, V., Rosell-Polo, J. R., Morros, J.-R., Ruiz-Hidalgo, J., & Gregorio, E. (2019). *Multi-modal deep learning for Fuji apple detection using RGB-D cameras and their radiometric capabilities.* Retrieved from http://www.grap.udl.cat/en/publications/KFuji_RGBDS_database.html

GitHub. (2022). *tylin*. Retrieved from https://github.com/cocodataset/cocoapi#subdirectory=PythonAPI

Gongal, A., Amatya, S., Karkee, M., Zhang, Q., & Lewis, K. (2015). *Sensors and systems for fruit detection and localization: A review.*

Goodfellow, I., Bengio, Y., & Courville, A. (2019). *Deep learning.*

Google Colab. (2022). *Colaboratory basics*. Retrieved from https://research.google.com/colaboratory/faq.html

Gu, W., Bai, S., & Kong, L. (2022). *A review on 2D instance segmentation based on deep neural networks.*

Halstead, M., Ahmadi, A., Smitt, C., Schmittmann, O., & McCool, C. (2021). *Crop Agnostic Monitoring Driven by Deep Learning.*

Halstead, M., Denman, S., Fookes, C., & McCool, C. (2020). *Fruit Detection in the Wild: The Impact of Varying Conditions and Cultivar.*

Han, L., Zheng, T., Xu, L., & Fang, L. (2020). *OccuSeg: Occupancy-aware 3D Instance Segmentation.*

He, K., Gkioxari, G., Dollar, P., & Girshick, R. (2017). *Mask R-CNN.*

He, Y., YU, H., LIU, X., YANG, Z., SUN, W., WANG, Y., . . . MIAN, A. (2021). *Deep Learning based 3D Segmentation: A Survey.*

Huang, Z., Huang, L., Gong, Y., Huang, C., & Wang, X. (2019). *Mask Scoring R-CNN.*

IPCC. (2021). *Sixth assessment Report (AR6)*. Retrieved from https://www.ipcc.ch/2021/08/09/ar6-wg1-20210809-pr/

ISPA. (2022). *International Society of Precision Agriculture*. Retrieved from https://www.ispag.org/

Johns Hopkins. (2022). Retrieved from https://www.foodsystemprimer.org/food-production/history-of-agriculture/

Kang, H., & Chen, C. (2020). *Fruit detection, segmentation and 3D visualisation of environments in apple orchards.*

Kootstra, G., Wang, X., Blok, P. M., Hemming, J., & Henten, E. v. (2020). *Selective Harvesting Robotics: Current Research, Trends, and Future Directions.*

Kusumam, K., Krajnik, T., Pearson, S., Duckett, T., & Cielniak, G. (2017). *3D-Vision Based Detection, Localisation and Sizing of Broccoli Heads in the Field.*

Li, D., Shi, G., Li, J., Chen, Y., Zhang, S., Xiang, S., & Jinde, S. (2022). *PlantNet: A dual-function point cloud segmentation network for multiple plant species.*

Li, T., Feng, Q., Qiu, Q., Xie, F., & Zhao, C. (2022). *Occluded Apple Fruit Detection and Localization with a Frustum-Based Point-Cloud-Processing Approach for Robotic Harvesting.*

Liang, Z., Li, Z., Xu, S., Tan, M., & Jia, K. (2021). *Instance Segmentation in 3D Scenes using Semantic Superpoint Tree Networks.*

Liang, Z., Li, Z., Xu, S., Tan, M., & Jia, K. (2021). *Instance Segmentation in 3D Scenes using Semantic Superpoint Tree Networks.*

Lin, G., Tang, Y., Zou, X., Xiong, J., & Fang, Y. (2019). *Color-, depth-, and shape-based 3D fruit detection.*

Ling, X., Zhao, Y., Gong, L., Liu, C., & Wang, T. (2019). *Dual-arm cooperation and implementing for robotic harvesting tomato using binocular vision.*

Lu, Y., & Young, S. (2020). *A survey of public datasets for computer vision tasks in precision agriculture.*

Martinez-Guanter, J., Ribeiro, Á., Peteinatos, G. G., Pérez-Ruiz, M., Gerhards, R., Bengochea-Guevara, J. M., . . . Andújar, D. (2019). *Low-Cost Three-Dimensional Modeling of Crop Plants.*

Matterport. (2019). *Mask_RCNN*. Retrieved from github: https://github.com/matterport/Mask_RCNN

Mavridou, E., Vrochidou, E., Papakostas, G. A., Pachidis, T., & Kaburlasos, V. G. (2019). *Machine Vision Systems in Precision Agriculture for Crop Farming.*

METOMOTION. (2022). Retrieved from https://www.metomotion.com/

MMDetection. (2022). Retrieved from COCO annotation format: https://github.com/open-mmlab/mmdetection/blob/master/demo/MMDet_InstanceSeg_Tutorial.ipynb

NG. (2019). *National Geographic*. Retrieved from World History: https://www.nationalgeographic.org/article/development-agriculture/

NVIDIA, Vingelmann, P., Fitzek, & P., F. H. (2020). *CUDA.*

Open3D. (2022). *Open3D Homepage*. Retrieved from http://www.open3d.org/

OpenMMLab. (2022). Retrieved from git frontpage: https://github.com/open-mmlab

Papers with Code. (2022). Retrieved from Object detection : https://paperswithcode.com/task/object-detection

PaperswithCode. (2022). *Instance segmentation*. Retrieved from Benchmarks: https://paperswithcode.com/task/instance-segmentation

PyTorch. (2022). Retrieved from Visualization utilities: https://pytorch.org/vision/main/auto_examples/plot_visualization_utils.html

PyTorch. (2022). *Cost eval*. Retrieved from https://github.com/pytorch/vision/issues/1574

PyTorch. (2022). *Mask R-CNN*. Retrieved from https://pytorch.org/tutorials/intermediate/torchvision_tutorial.html

PyTorch. (2022). *Mask R-CNN version*. Retrieved from https://download.pytorch.org/models/maskrcnn_resnet50_fpn_coco-bf2d0c1e.pth

Roberts, L. (1963). *Machine Perception of Three-Dimensional Solids.*

Rossum, V., Guido, Drake, & Fred, L. (2009). *Python 3 Reference Manual.*

Ruder, S. (2017). *An overview of gradient descent optimization algorithms.*

Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., . . . Fei-Fei, L. (2015). *ImageNet Large Scale Visual Recognition Challenge.* Retrieved from https://paperswithcode.com/dataset/imagenet

SAIA. (2022). Retrieved from https://www.saia-agrobotics.com/

Smitt, C. G., & Mccool, C. (2021). *PATHoBot: A Robot for Glasshouse Crop Phenotyping and Intervention.*

Souza, R. d., Peña-Fleitas, M. T., Thompson, R. B., Gallardo, M., Grasso, R., & Padilla, F. M. (2021). *Use of fuorescence indices as predictors of crop N status and yield for greenhouse sweet pepper crops.*

Stackoverflow. (2022). *Data augmentation in PyTorch*. Retrieved from https://stackoverflow.com/questions/51677788/data-augmentation-in-pytorch

Stackoverflow. (2022). *Homepage*. Retrieved from https://stackoverflow.com/

UN. (2017). *World Population Prospects: The 2017 Revision, Key Findings and Advance .*

UN. (2021). *The 17 Goals*. Retrieved from SDGs: https://sdgs.un.org/goals

Walt, V. d., Stefan, Schønberger, L, J., Nunez-Iglesias, Juan, . . . Tony. (2014). *Scikit-Image.*

Ward, I. R., Laga, H., & Bennamoun, M. (2019). RGB-D image-based Object Detection: from Traditional Methods to Deep Learning Techniques. In *RGB-D Image Analysis and Processing.*

Xiong, Y., Ge, Y., Grimstad, L., & From, P. J. (2019). *An autonomous strawberry-harvesting robot: Design, development, integration, and field evaluation.*

Xu, P., Fang, N., Liu, N., Lin, F., Yang, S., & Ning, J. (2022). *Visual recognition of cherry tomatoes in plant factory based on improved deep instance segmentation .*

Zaidi, S. S., Ansari, M. S., Aslam, A., Kanwal, N., Asghar, M., & Lee, B. (2021). *A Survey of Modern Deep Learning based Object Detection Models.*

9 APPENDIX

(1) read_data, function from visualize_annotation.py

```
DATASET_PATH = "C:/Users/Patrick/EM/dataset/train"

# Get the path and read the image file to np array
def read_data(file_example):
    # Get RGB image
    rgb_path = (DATASET_PATH + '/rgb/' + file_example + '.png')
    rgb_read = plt.imread(rgb_path)
    # Get depth image
    depth_path = (DATASET_PATH + '/depth/' + file_example + '.tiff')
    depth_read = plt.imread(depth_path)

    # Get mask by combining the annotation files
    mask_path = (DATASET_PATH + '/mask/' + file_example)
    # List the black, green, mixed and red annotation files
    anno_id_list = list(os.listdir(mask_path))
    # Get the first annotation image and transform to np array
    anno_id_path = os.path.join(mask_path, anno_id_list[0])
    new_mask = Image.open(anno_id_path)
    new_mask = np.array(new_mask)
    # Get an offset to shift label placements
    offset = len(np.unique(new_mask)) # Move to the next label position
    # Relabel the remaining annotations and combine them into one variable
    for num in range(1, len(anno_id_list)):
        # Get the next annotation image and transform to np array
        anno_id_path = os.path.join(mask_path, anno_id_list[num])
        anno_id = Image.open(anno_id_path)
        anno_id = np.array(anno_id)
        # Relabel the instances
        for label in np.unique(anno_id):
            if label > 0: # Skip background label
                anno_id[anno_id == label] = offset
                offset += 1
        new_mask = np.add(new_mask, anno_id)
    print('Unique instances before:', np.unique(new_mask))
    print('Number of instances before:', len(np.unique(new_mask)))
    print('Offset before:', offset)
    # Delete generated overlapping instance labels
    if offset != len(np.unique(new_mask)):
        new_mask[new_mask > offset] = 0
    print('Unique instances after:', np.unique(new_mask))
    print('Number of instances after:', len(np.unique(new_mask)))
    print('Offset after:', offset)
    return rgb_read, depth_read, new_mask
```

(2) subplot_rgb_depth_2D, from visualize_annotation.py

```
53  # Plots the RGB, depth and RGBD in 2D
54  def subplot_rgb_depth_2D(rgb, depth):
55      # Plot RGB image
56      plt.subplot(1,3,1)
57      plt.title('RGB image')
58      plt.xlabel("Width")
59      plt.ylabel("Height")
60      plt.imshow(rgb)
61      # Plot depth image
62      plt.subplot(1,3,2)
63      plt.title('Depth image')
64      plt.xlabel("Width")
65      plt.ylabel("Height")
66      plt.imshow(depth, cmap='gray')
67      # Plot RGB and depth overlayed
68      plt.subplot(1,3,3)
69      plt.title('RGBD image')
70      plt.xlabel("Width")
71      plt.ylabel("Height")
72      rgbd = np.dstack([rgb, depth])
73      plt.imshow(rgbd)
74      plt.plot()
75      plt.show()
```

(3) histogram_depth, from visualize_annotation.py

```
77  def histogram_depth(depth):
78      # Plot depth image
79      plt.subplot(1,3,1)
80      plt.title('Depth image')
81      plt.xlabel("Width")
82      plt.ylabel("Height")
83      plt.imshow(depth,cmap='gray')
84      # Plot equalized depth image
85      plt.subplot(1,3,2)
86      plt.title('Equalized depth image')
87      plt.xlabel("Width")
88      plt.ylabel("Height")
89      depth_eq = cv2.cvtColor(depth, cv2.COLOR_BGR2GRAY)
90      eq = cv2.equalizeHist(depth_eq)
91      plt.imshow(eq, cmap='gray')
92      # Plot histogram with value frequencies
93      plt.subplot(1,3,3)
94      plt.title('Histogram, frequency of pixel values')
95      plt.xlabel("Pixel value")
96      plt.ylabel("Frequency")
97      plt.hist(depth.ravel(), bins=250)
98      plt.show()
```

(4) subplot_mask_2D, from visualize_annotation.py

```
# Plots the instance segmentation and bounding boxes in 2D
def subplot_mask_2D(rgb, mask, file_example):
    # Plot masked RGB image with bounding boxes
    fig, ax = plt.subplots()
    plt.title('Masked RGB, sample: ' + file_example)
    plt.xlabel("Width")
    plt.ylabel("Height")
    ax.imshow(rgb)
    masked = np.ma.masked_where(mask == 0, mask)
    ax.imshow(masked)
    # Get annotation labels and remove background
    obj_ids = np.unique(mask)
    obj_ids = obj_ids[1:]
    # Get the binary map of the annotations
    masks = mask == obj_ids[:, None, None]
    # Get bounding box coordinates for each mask
    num_objs = len(obj_ids)
    boxes = []
    # Calculate the bounding box position
    for i in range(num_objs):
        pos = np.where(masks[i])
        xmin = np.min(pos[1])
        xmax = np.max(pos[1])
        ymin = np.min(pos[0])
        ymax = np.max(pos[0])
        boxes.append([xmin, ymin, xmax, ymax])
    print('Num of boxes:', len(boxes))
    # Plot the bounding boxes
    for i in range(len(boxes)):
        rect = patches.Rectangle((boxes[i][0], boxes[i][1]), boxes[i][2]-boxes[i][0],
            boxes[i][3]-boxes[i][1], linewidth=1, edgecolor='r', facecolor='none')
        ax.add_patch(rect)
    plt.plot()
    plt.show()
```

(5) Local computer, specifications:

Asus PC, Intel i5-4460 3.20 Ghz processors, 8 GB RAM, NVIDIA GeForce GTX 750 Ti, Windows 10 Home build 19042, cudatoolkit 11.5 v.496.76, CUDA 10.1. The latest and stable PyTorch version 1.11.0 compatible with CUDA 11.3.

(6) Mask R-CNN requirements

Packages: Pytorch, torchvision, cuda, cython, numpy, pillow, git.

Github repository: Pycocotools.

PyTorch helper functions: coco_eval.py, coco_utils.py, engine.py, transforms.py, utils.py.

(7) YOLACT and QueryInst requirements

Packages: Pytorch, torchvision, cuda, MMCV, MMDetection.

Pretrained model: yolact_r50_1x8_coco_20200908-f38d58df.pth.

Pretrained model: queryinst_r50_fpn_1x_coco_20210907_084916-5a8f1998.pth

(8) Mask R-CNN mAP scores

```
IoU metric: bbox
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.035
 Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.117
 Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.014
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.002
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.053
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.047
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.018
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.071
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.125
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.070
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.174
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.062
IoU metric: segm
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.052
 Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.154
 Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.025
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.002
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.069
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.133
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.024
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.100
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.161
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.071
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.206
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.133
```

LR of 0.01

```
IoU metric: bbox
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.417
 Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.647
 Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.475
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.252
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.425
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.499
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.073
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.450
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.564
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.471
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.600
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.558
IoU metric: segm
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.427
 Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.694
 Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.427
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.190
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.416
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.628
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.077
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.456
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.557
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.430
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.555
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.653
```

LR of 0.001.

```
IoU metric: bbox
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.383
 Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.639
 Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.405
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.182
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.375
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.483
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.065
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.411
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.518
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.364
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.555
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.557
IoU metric: segm
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.387
 Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.697
 Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.385
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.111
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.349
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.564
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.072
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.407
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.491
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.340
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.491
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.602
```

LR of 0.0001

```
IoU metric: bbox
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.076
 Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.234
 Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.021
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.000
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.085
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.134
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.025
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.134
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.204
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.045
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.214
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.303
IoU metric: segm
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.118
 Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.270
 Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.082
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.000
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.083
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.276
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.044
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.189
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.256
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.044
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.268
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.393
```

(9) Mask R-CNN, pretrained mAP scores

LR of 0.01

```
IoU metric: bbox
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.424
 Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.617
 Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.486
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.213
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.424
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.551
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.075
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.461
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.537
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.358
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.573
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.596
IoU metric: segm
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.450
 Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.681
 Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.480
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.154
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.447
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.671
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.080
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.477
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.557
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.354
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.564
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.692
```

LR of 0.001

```
IoU metric: bbox
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.409
 Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.619
 Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.449
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.208
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.398
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.531
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.071
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.441
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.556
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.388
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.592
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.600
IoU metric: segm
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.425
 Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.666
 Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.448
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.137
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.397
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.659
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.075
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.455
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.555
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.375
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.558
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.682
```

LR of 0.0001

```
IoU metric: bbox
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.385
 Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.643
 Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.407
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.198
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.375
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.493
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.071
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.408
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.525
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.391
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.558
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.559
IoU metric: segm
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.388
 Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.690
 Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.375
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.137
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.339
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.571
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.075
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.410
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.494
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.344
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.498
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.598
```

(19) YOLACT mAP scores

LR of 0.0001, bounding box (upper) & mask (lower) metrics

```
Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.000
Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=1000 ] = 0.001
Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=1000 ] = 0.000
Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=1000 ] = 0.000
Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=1000 ] = 0.000
Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=1000 ] = 0.001
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.021
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=300 ] = 0.021
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=1000 ] = 0.021
Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=1000 ] = 0.000
Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=1000 ] = 0.023
Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=1000 ] = 0.032
```

```
Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.000
Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=1000 ] = 0.000
Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=1000 ] = 0.000
Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=1000 ] = 0.000
Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=1000 ] = 0.000
Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=1000 ] = 0.000
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.000
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=300 ] = 0.000
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=1000 ] = 0.000
Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=1000 ] = 0.000
Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=1000 ] = 0.000
Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=1000 ] = 0.001
```

Pretrained, LR of 0.001

```
Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.333
Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=1000 ] = 0.553
Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=1000 ] = 0.362
Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=1000 ] = 0.155
Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=1000 ] = 0.355
Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=1000 ] = 0.447
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.495
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=300 ] = 0.495
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=1000 ] = 0.495
Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=1000 ] = 0.294
Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=1000 ] = 0.548
Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=1000 ] = 0.536
```

```
Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.301
Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=1000 ] = 0.570
Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=1000 ] = 0.286
Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=1000 ] = 0.025
Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=1000 ] = 0.288
Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=1000 ] = 0.609
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.426
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=300 ] = 0.426
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=1000 ] = 0.426
Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=1000 ] = 0.200
Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=1000 ] = 0.410
Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=1000 ] = 0.628
```

Pretrained, LR of 0.0001

```
Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.300
Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=1000 ] = 0.497
Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=1000 ] = 0.326
Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=1000 ] = 0.089
Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=1000 ] = 0.309
Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=1000 ] = 0.440
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.448
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=300 ] = 0.448
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=1000 ] = 0.448
Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=1000 ] = 0.205
Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=1000 ] = 0.498
Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=1000 ] = 0.530
```

```
Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.319
Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=1000 ] = 0.551
Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=1000 ] = 0.336
Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=1000 ] = 0.024
Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=1000 ] = 0.290
Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=1000 ] = 0.605
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.429
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=300 ] = 0.429
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=1000 ] = 0.429
Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=1000 ] = 0.189
Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=1000 ] = 0.423
Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=1000 ] = 0.622
```

(20) QueryInst

100 epochs run

```
2022-05-21 21:09:28,281 - mmdet - INFO - Epoch [100][50/50]
```

```
2022-05-21 21:10:09,242 - mmdet - INFO -
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.000
 Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=1000 ] = 0.000
 Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=1000 ] = 0.000
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=1000 ] = 0.000
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=1000 ] = 0.000
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=1000 ] = 0.000
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.000
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=300 ] = 0.000
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=1000 ] = 0.000
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=1000 ] = 0.000
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=1000 ] = 0.000
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=1000 ] = 0.000
```

```
2022-05-21 21:10:10,539 - mmdet - INFO -
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.000
 Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=1000 ] = 0.000
 Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=1000 ] = 0.000
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=1000 ] = 0.000
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=1000 ] = 0.000
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=1000 ] = 0.000
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.000
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=300 ] = 0.000
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=1000 ] = 0.000
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=1000 ] = 0.000
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=1000 ] = 0.000
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=1000 ] = 0.000
```

(21) Folder structure for the program code. Available in an external folder.

MODELS AND DATASET WERE REMOVED FROM ZIP-FILE DUE TO SiZE.

dataset

-> test: containing depth, mask and rgb test data.

-> train: containing depth, mask and rgb training data.

-> val: containing depth, mask and rgb validation data.

mrcnn

-> models: containing the Mask R-CNN trained models.

-> coco_eval.py, coco_utils.py, engine.py, transform.py, utils.py.

-> mrcnn_test.py

-> mrcnn_train.py

queryinst

-> queryinst_default_config.py

-> queryinst_r50_fpn_1x_coco_20210907_084916-5a8f1998.pth

-> queryinst_test.py

-> queryinst_train.py

yolact

-> models: containing the YOLACT trained models.

-> annotation_to_coco.py

-> yolact_default_config.py

-> yolact_test.py

-> yolact_train.py

visualize_annotation.py