FACULTY OF SCIENCE AND TECHNOLOGY

MASTER'S THESIS

Study programme/specialisation: Master of Science in Computer Science / Data Science	Spring semester 2021 Confidential
Author: Christoffer René Haaland Holmesland	
Programme coordinator: Trygve Christian Eftestøl External supervisor: Geir Vevle from RFID Solutions AS	
Title of master's thesis: Digital Waste Management - detection technology	
Credits: 30	
Keywords: Object detection, PyTorch, IoT, time of flight, waste management transfer learning, CNN	Number of pages: 54 Stavanger 15. june 2021

Contents

Abstract

In this thesis, the goal has been to improve the data flow from the garbage trucks in Halden municipality. We begin by designing, building, and installing a data capture unit to collect images of the collected waste. Machine learning models were trained to analyze the images and detect whether customers use green bags to dispose of their organic waste. Models were also trained on data from a time of flight camera to measure the volume of the waste.

We discover that high accuracy object detection is possible on organic waste. Limitations on the use of time of flight technology are found when it is used in a garbage collection environment. The result is that volume measurement is not possible unless the environment changes.

Chapter 1

Introduction

Every waste container in Halden municipality are tagged with an RFID label and a QR code as part of their digitalization process of the waste management system. In total there are about 24 000 tagged containers at the moment. RFID readers have been installed on the waste collection trucks to enable detection of when and where the containers are emptied. There are five trucks in operation five days every week.

For the waste management industry, and in particular Halden municipality, there is a great benefit in knowing the amount of waste collected, as well as the purity of it. The purity of the waste is a measure of how much is of the expected type compared to the amount of other waste. There is currently no method to easily measure this. The way it is done now is a manual time consuming and costly process that is done twice every year at best.

The goal of the project is to automatically and in real-time measure the amount of collected waste and detect any anomalies during the collection process. The information about the anomalies can be used to figure out why they are happening and how they can be prevented. Waste amount can be used to optimize the truck routes to minimize the cost of waste collection services and reduce the environmental impact.

1.1 Problem statement

Explore the possibilities of using AI and image recognition to measure the amount of waste, and detect anomalies to enhance the data stream from the garbage trucks. In particular:

1. Use sensor technology to detect whether people use the supplied green bags for their food waste.
2. Use sensor technology to measure the volume of the waste.

1.2 Company description

RFID Solutions AS specialises in designing and implementing systems utilising RFID and IoT technology. The main contact for this project has been CIO Geir Vevle.

Halden municipality is located close to the southern border between Norway and Sweden. It has a population of 31 177 people [19]. The main contact for this projet has been Kåre Edvardsen, who is the technical manager for the waste management in Halden.

1.3 Previous work

The author is not aware of any previous work where camera technology has been installed on garbage trucks to measure the waste that is being collected. No previous work where ToF cameras were used to measure the waste was found either.

Modern waste processing plants use sensor technology to inspect incoming waste. In Norway there are facilities where lasers are used to classify plastic waste. The lasers can be used to know whether a piece of plastic is recyclable or not. Robotic arms are used to automatically retrieve the plastic from the rest of the waste. The main difference compared to this work is that the data from this project can be directly connected to the customer as it is not mixed with other waste before being inspected. In addition, this project uses camera technology instead of lasers.

Waste type classification from images has been done many times previously. This is commonly done by taking a picture of a single piece of waste and then classifying it into one of four classes: paper, glass, plastic, metal. Olugboja Adedeji and Zenghui Wang used a convolutional neural network to do this and achieved an accuracy of 87% [1]. M, Aghilan et al. suggests using cameras inside waste containers, and achieved an accuracy of 79% using a CNN [15]. The main difference from this type of work to the work presented in this thesis is the possibility of classifying multiple objects in the same noisy image.

Chapter 2

Background

2.1 RFID

Radio-frequency identification (RFID) is technology that uses radio waves to communicate [6]. A RFID tag can be attached to any asset to give it an ID number. When the tag is moved inside the range of a RFID reader the ID number is transmitted to the reader so it can be used in external systems. In this way any asset can be tracked on an individual basis through its lifecycle. The RFID tags used on the containers in Halden are shown in figure 2.1.

2.2 ToF camera

A time of flight (ToF) camera is a device that can be used to capture 3D models of objects [12]. It works by emitting light and then measuring the time it takes before the light is reflected back to the camera. The output from the camera is a series of points that form a point cloud.

Figure 2.1: RFID Tag used by Halden on the waste containers

2.3 Neural network

A neural network[9] is a machine learning model constructed using neurons assembled in layers. As a minimum the network needs one input layer and one output layer. It is common to have layers between them called hidden layers. If a network has many hidden layers it is called a deep neural network. The number of neurons in the input layer is equal to the number of features each sample has. The size of the output layer depends on what the network is predicting. If it is used for classification the number of neurons should be equal to the number of classes. When the network is used for regression it is sufficient to have one output neuron. There are multiple types of layers. The most common are fully connected layers where the output from a neuron is used as an input to each neuron in the next layer.

A neuron is a unit with one or more input values and an output value. For each input value the neuron has an associated weight value w_{ij}. In addition to the input value x_j that comes from the neurons in the previous layer each neuron usually has a bias value b_i. Before outputting the neuron value, the calculated value is passed through an activation function α. There are many different activation functions. A frequently used function is the

rectified linear unit (ReLU) which is $f(x) = max(0, x)$. A mathematical expression for a neuron is

$$n_i = \alpha(w_i x + b_i)$$

where x is a vector containing the output from the previous layer. Using this representation of a neuron it is possible to express the neural network as a series of matrix multiplications.

A visual representation of a neural network is shown in figure 2.2. The two green circles represent the input layer. The three gray circles are the hidden layer. The red circle is the only neuron in the output layer indicating that this network is used for single class classification or regression.

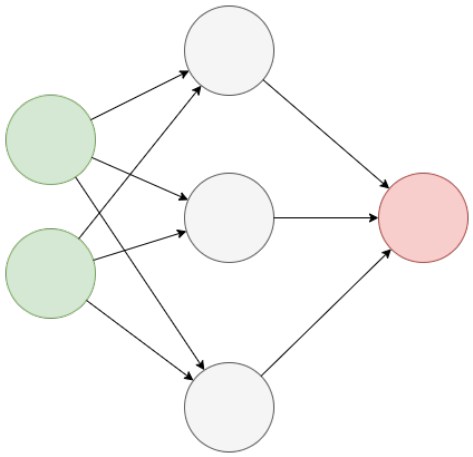

Figure 2.2: Neural network example

To train the network the neuron weights can be adjusted until the desired output value is achieved. To figure out how much the weights should change require a series of operations. The first step is to take the input features for a sample and do a feed forward pass, i.e. calculate the output value, given the current weights. The next step is to calculate a measure of how incorrect the output is. This is called the loss and there are many ways to calculate it. For networks used for regression the mean squared error (MSE) can be used. When we have the loss and the known correct output value we can perform backpropagation to find how much the weights in each layer need to change to produce the correct output value. This is done by starting at

the output layer and then looking at how much the previous layer needs to change to get the correct output value. It is then possible to look back another layer to find how much the layer before that again needs to change to get the correct values in the layer behind the output layer. This process repeats until the input layer. The result of the backpropagation process is that we get the gradient (partial derivative) of the loss. The final step is using the gradient with a learning algorithm to change the weights. There are many learning algorithms to pick between. Each of them have different properties and parameters that can be selected. The simplest example is the iterative stochastic gradient descent algorithm that takes the gradient from one training sample, multiples it by a learning rate and then changes the weights by that.

2.3.1 Convolutional neural network

When the input data for a neural network are images it is common to use a convolutional neural network. A convolutional neural network works like the standard neural network, except the first fully connected layers are replaced by convolutional layers and pooling layers.

A convolutional layer is represented by a matrix of weights that moves over every data point in the sample to generate the output values of the layer. The matrix is square and the size of it is called the kernel size. Some convolutional layers have a stride value above 1. The stride value is how many steps the matrix moves before calculating a new output value. For example, a stride value of 2 means that the the layer only generates a output value for every other input value. To generate the output value the weight matrix is centered on a point. The value of that point and the points around it depending on the kernel size are multiplied with the weights to get the output value. Activation functions are usually used with convolutional layers.

To decrease the size of a layer before passing the values to the next layer a pooling layer can be put between them. The pooling layer takes multiple values as its input and outputs a single value. A window is moved across the input similarly to the convolutional layer, but instead of multiplying by weights the values are aggregated. Common aggregations are the minimum, maximum and the average.

Chapter 3

Design and implementation

3.1 Data capture

A data collection unit (DCU) was constructed to enable the data needs for this project. Its main components are two cameras that are used to capture images of the waste that is collected. The first camera is a Raspberry Pi Camera Module V2[7]. It is a normal camera that can be attached to a Raspberry Pi[**?**] using a ribbon cable. The maximum resolution of the images is 8 megapixels (MP), but to reduce the size of the data and the processing time it was configured to output images at 2.1 MP. The other camera is a Pieye Nimbus 3D ToF camera[8] module for the Raspberry Pi that outputs point clouds at a resolution of 352 by 288 pixels. The unit also contains two Raspberry Pi 4 devices. One of them is used to get the image and point cloud from the camera. The other is used to interact with the RFID reader and an external server.

Figure 3.1 shows how the DCU was installed on the garbage truck. The box in the middle of the image is the DCU. The device attached to the left side of the box is the camera. The ToF camera is attached to the right side of the box. The RFID reader and RFID antennas were not installed as part of this project. The antennas can be seen on the right and left edges of the image as white panels. The reader is inside the box at the top of the truck.

Figure 3.1: DCU installed on garbage truck

When a waste container is emptied by one of the trucks the RFID tag inside it gets close enough to the RFID reader that the ID number is read. The ID number and a GPS position is transmitted to an external server to track where and when the container was emptied. At the same time the ID number are transmitted to the first Raspberry Pi inside the DCU. It retrieves the camera data from the other Raspberry Pi and transmits it back to the external system. After training the models only the result of them is returned to the server to reduce the network bandwidth usage.

The initial DCU had the cameras on the outside of the box to make it easier to adjust camera orientation if that was needed. After two weeks water was able to get through the water-proofing. Both cameras were damaged beyond repair and had to be replaced, shown in figure 3.3. A new design was made where the cameras were on the inside instead. The new design is shown in figure 3.2, 3.4, and 3.5.

Figure 3.2: New DCU installed on truck

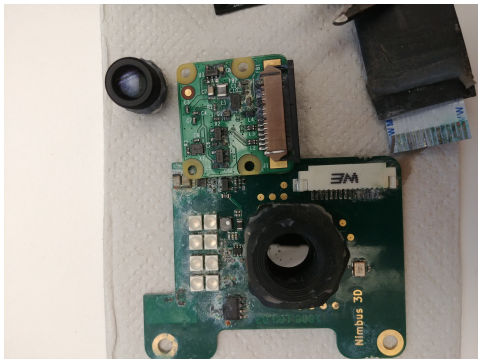

Figure 3.3: Damaged camera

Figure 3.4: The outside of the new DCU

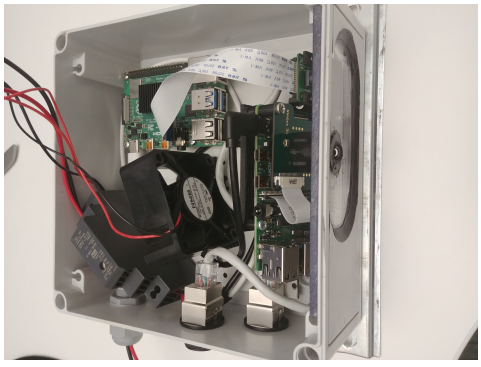

Figure 3.5: The inside of the new DCU

3.2 Green bag detection

In Halden the customers, i.e the people who live there, are supposed to throw organic waste and other non-recyclable waste in the same container. They are given specific green bags that should be used for the organic waste such that the systems are able to automatically sort it at the processing plants. The task is to detect how many green bags are in each container. An example of what the images from the garbage trucks look like is shown in figure 3.6. The same image with some of the green bags marked by red rectangles is shown in figure 3.7.

Figure 3.6: Example image from the garbage truck camera

There are some factors that complicates this. The first is that two containers are emptied at the same time into the same section on the truck. There is no easy way to determine which container the waste came from. The position the containers are lifted to are somewhat offset from the middle so in general one can say that the waste on the left side came from one container and the waste on the right side came from the other, but in practice we see that this is often not necessarily true. Depending on the size, weight and type of waste it can move towards the other side as it is emptied. The second problem is that multiple containers can be emptied into the same section before it is pulled inside the truck and compressed. This means that the images and point clouds will include waste that was not part of any of the two containers that were just emptied. There are techniques that can be

Figure 3.7: Some of the green bags marked on the example image

used to figure out the difference, e.g. by subtracting the previous image from the new one, but as waste is often moved around by new waste falling on top of it, it can be challenging to get good results. The third challenge is that a popular grocery store in Norway uses green grocery bags that are similar in color to the bags issued by Halden.

3.2.1 Method 1: Faster R-CNN

Transfer learning is the process of taking a model that has been trained on a data set and changing the weights by training it on new data. In this case it is assumed that if the model that is being retrained is performing well on the original data then it has learnt some important aspects of object detection. This often requires large data sets that can be very time consuming to collect. Method 1 uses transfer learning to retrain a model that was originally trained on the COCO 2017 data[14]. The pre-trained model is the `fasterrcnn_resnet50_fpn` model[5] from the `torchvision` library by the PyTorch project. The COCO data contains 91 different objects. Because the original model performs well on this data we can assume that it can be retrained to perform well on a new object. The model uses the Faster R-CNN[17] method for object detection.

The network consists of three main parts. In the sections below only the most important layers for that part are shown. The complete network architecture is included in appendix A. When a layer is listed as Conv2d(X, Y, ...) that means it takes X input channels and outputs Y channels.

Feature pyramid network (FPN)

The first step is to use a feature pyramid network[13] to generate feature maps for the input image. As the name of the model suggests this is based on the RestNet-50 architecture[10]. The FPN does two things. First it takes the input image and generates multiple maps from it. The maps are generated by applying a series of convolutional layers to it. The layer stride starts at 2 and then doubles each time a new map is generated. When these maps are put on top of each other it looks like a pyramid which is were the name comes from. Four maps are created in the first part of the network. The second part of the pyramid network is taking these maps and combining them to create a pathway from the top of the pyramid to the bottom, i.e. the input image. First, only the top map is considered. A convolution is applied to it and that is the output pathway. Then the top map is upsampled to match the map below it and they are added together using element-wise addition. The convolution is again applied to it to get another pathway. This process continues until the lowest feature map is reached.

The architecture of the network layers that are used to generate the pathways are:

1. Pathway 1: Conv2d(256, 256, kernel size = 1) followed by Conv2d(256, 256, kernel size = 3).
2. Pathway 2: Conv2d(512, 256, kernel size = 1) followed by Conv2d(256, 256, kernel size = 3).
3. Pathway 3: Conv2d(1024, 256, kernel size = 1) followed by Conv2d(256, 256, kernel size = 3).
4. Pathway 4: Conv2d(2048, 256, kernel size = 1) followed by Conv2d(256, 256, kernel size = 3).

Region proposal network (RPN)

Next, the pathway values are passed to a region proposal network[17] to find regions of interest. It does this by sliding another series of convolutions over the pathway values to determine areas where it is likely to be an object.

The network layers look like this

1. Conv2d(256, 256, kernel size = 3)
2. Conv2d(256, 12, kernel size = 1)

This part of the network is also responsible for generating anchor boxes which is the main reason for why the Faster R-CNN model is fast. The anchor boxes are boxes that have matching aspect ratio to the objects that should be detected. Because the aspect ratio is the same they can by scaled up and down and still work correctly. This means that they work well with the downsampled values from the feature pyramid network.

Faster R-CNN

The regions of interest are finally passed to a faster R-CNN predictor to determine if there is an object there and which object type it is. The network first passes the input image through a series of convolutional layers and then uses the output from that in combination with the regions of interest from the RPN to make a prediction on the bounding boxes of objects in the image. The pre-trained model is capable of detecting all of the 91 objects that are labelled in the COCO dataset. For this detection task the output of the faster R-CNN predictor is replaced with only two classes: green bag and background.

The first two layers of this part are fully connected layers, both of them have 1024 neurons. The output from the second layer is passed to two new layers. One of them has two neurons and is responsible for classifying the object. In this case there are only two classes so there are only two neurons. The other layer that gets the same input values has 8 neurons which predicts the position of the box.

Training

In addition to the pre-trained model, the torchvision team provides recommendations[3] on which learning algorithm to use to update the network weights. The suggested algorithm is a stochastic gradient descent with an initial learning rate of 0.005, momentum of 0.9, and a weight decay of 0.0005. It is also recommended to gradually decrease the learning rate by multiplying the learning rate by 10^{-1} every 3 epochs. This was done when retraining the model. The model was trained for 10 epochs. The loss for the object bounding boxes is calculated using the intersection over union[18] method which is the ratio of how much of the area of the boxes overlap.

3.2.2 Method 2: YOLO

Method 2 uses transfer learning on a You Only Look Once (YOLO)[16] model. The YOLOv5s[11] is used because it is the smallest which means it has better performance, measured in execution time, when it is used on a Raspberry Pi. The model is trained on the same data as Faster R-CNN model, and it is also trained for 10 epochs. The complete network architecture is listed in appendix A. The main difference when it is compared to the faster R-CNN network is that instead of having three parts where each of them is responsible for a specific task, the YOLO network as a whole does everything.

3.3 Waste volume measurement

To measure the waste volume (in litres) the point clouds from the ToF camera are used. Before installing the equipment on the garbage trucks 103 images were taken with the camera in a testing area. The objects in the images are a combination of differently sized cardboard boxes. The plan was to use these images to explore what kind of data it is possible to get from the camera. In addition, a simple baseline model was created to measure the volume of the boxes. The Pieye Nimbus ToF camera is one of the simplest and cheapest ToF cameras on the market, so this also served as a test of whether a more sophisticated camera is necessary for the task.

When looking at the first group of images from the garbage truck it was clear that there was a problem with the data collection process. Every point cloud had a hole in the middle where no data was recorded. This can be seen in figure 3.8. Factors like dirty lens, bad positioning and light conditions were checked to see if they could explain the issue but the missing points were still a problem. The more advanced Basler Blaze ToF camera[2] was tested to see if the Pieye camera was the issue. When recreating the images with the new camera it became apparent that the problem was because of reflections from the metallic surfaces on the truck. This turned out to be a inherit problem with the ToF technology that cannot be fixed by using a better camera. For this reason the presented methods are used on the test images of the cardboard boxes.

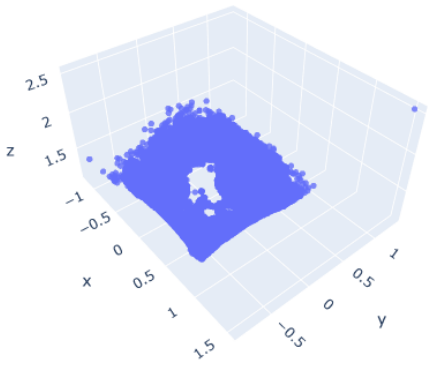

Figure 3.8: ToF point cloud with missing data

3.3.1 Method 1: Baseline

The baseline method uses the z-coordinate of the points in the point cloud to calculate the volume. When the camera is in a fixed position with a static

background the z-coordinates can be used as the distance from the camera to the object. In this way it is possible to find the volume if the area inside the camera view is measured. The test floor area was measured to be 325 by 275 centimeters. Using the camera resolution it was found that every pixel in the x-direction is 1.0831 cm, and every pixel in the y-direction is 0.9549 cm.

The points are processed in the following way:

1. Take a background image of the empty area. This is used to remove background elements like the floor or any walls from the image. Because the camera has to be in a fixed position with a static background for this to work the same background image can be used for measurement.
2. Take another image when the objects are in the area. The z-coordinates in this image show how close the waste is to the camera.
3. Extract the objects from the area by calculating the element-wise difference between each z-coordinate.
4. Remove some of the values that were consistently incorrect. The points along the outer border of the images are almost always just noise. To fix this the 15 pixels closest to the edge are discarded. In addition, the camera is sometimes unable to calculate the distance to a point. When that happens the z-coordinate is below zero. This is fixed by setting any point with a negative distance to 0.
5. The points are very noisy. To make the shapes smoother so they are more similar to the real world objects a filter is applied. For the boxes the percentile filter[4] from SciPy was found to produce good results. The selected percentile is 75, and the size is 30. The percentile filter works by taking the closest 30 points for every point in the image. The values at the 30 points are sorted in ascending order and then the value of the point is set to the 75th percentile. The result is thus something between an average and a maximum filter.
6. Calculate the sum of every point. This is the area under the curve which should be a good representation of the volume of the object.

The points at each step of the process are illustrated in figures 3.9 - 3.13.

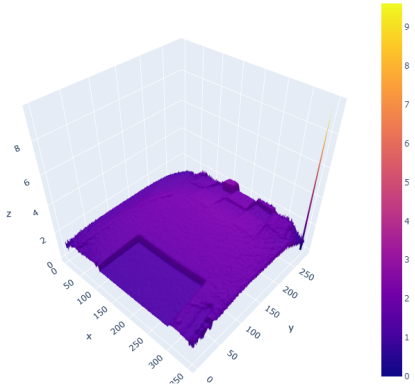

Figure 3.9: Step 1: Background image

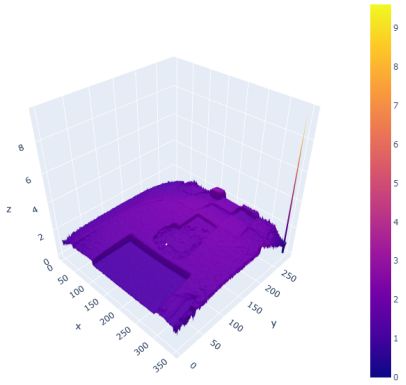

Figure 3.10: Step 2: Image with object and background

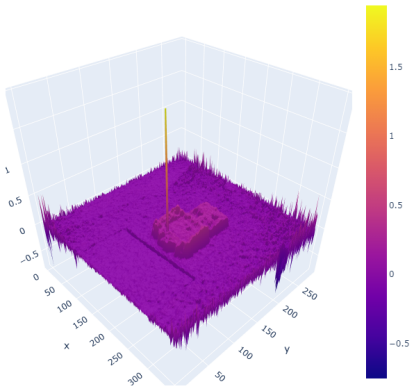

Figure 3.11: Step 3: Object image without the background

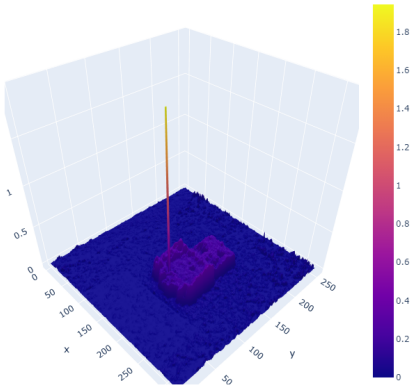

Figure 3.12: Step 4: Edge removal

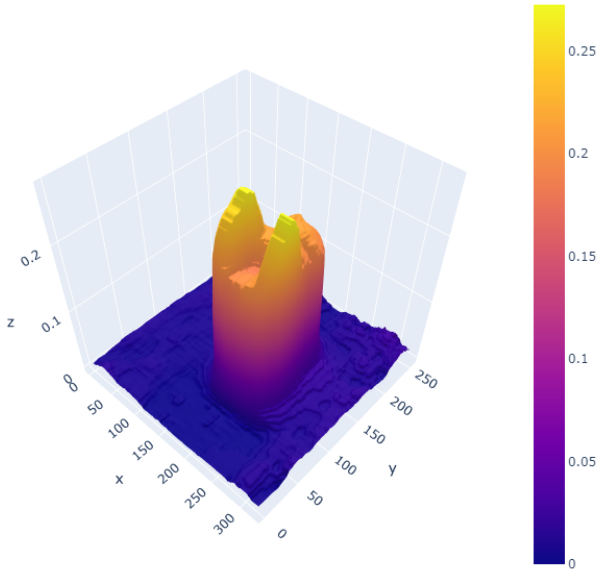

Figure 3.13: Step 5: Object smoothing

If we assume that the area under the curve represents the volume then the addition of another object to the image should always increase the sum by the same amount, meaning that there is a linear function that maps the area to the volume.

$$V(x) = \beta_0 + \beta_1 \cdot x + \epsilon$$

The ϵ represents the irreducible error that occurs because of the noise from the camera, and the error that occurs from objects being hidden behind each other. To find the function a linear regression model is used, because if the irreducible error is significant then the result benefits from the high bias associated with the model. The ordinary least squares method is used to find the coefficients. Lasso or ridge regression is not necessary as there

is only two coefficients that need to be estimated.

The linear regression model outputs the following parameters $\hat{\beta}_0 = 0.60518140$ and $\hat{\beta}_1 = 0.02247492$ giving the estimator:

$$\hat{V}(x) = 0.60518140 + 0.02247492x$$

3.3.2 Method 2: Deep learning

The deep learning approach uses a series of convolutional layers and fully connected layers to find the volume. The input to the network follows the same process as in the baseline method, except the output from step 5 is used as the input to the network. The output of the network is a single number representing the volume in litres.

The network layers are:

1. Convolutional layer with one input channel, one output channel, and kernel size 3.
2. Max-pooling layer with kernel size 2, and stride 2.
3. Convolutional layer with one input channel, one output channel, and kernel size 5.
4. Max-pooling layer with kernel size 2, and stride 2.
5. Fully connected layer with 120 nodes.
6. Fully connected layer with 32 nodes.
7. Output layer with 1 node.

The convolutional and fully connected layers use the rectified linear unit (ReLU) activation function. The output layer does not use an activation function.

During the training process the mean squared error of the output is the criterion that is attempted to minimize. This is achieved using an optimizer

performing stochastic gradient descent with a learning rate of 0.001. The model is trained for 100 epochs with a batch size of 25. As shown in figure 3.14 the model converges around epoch 18.

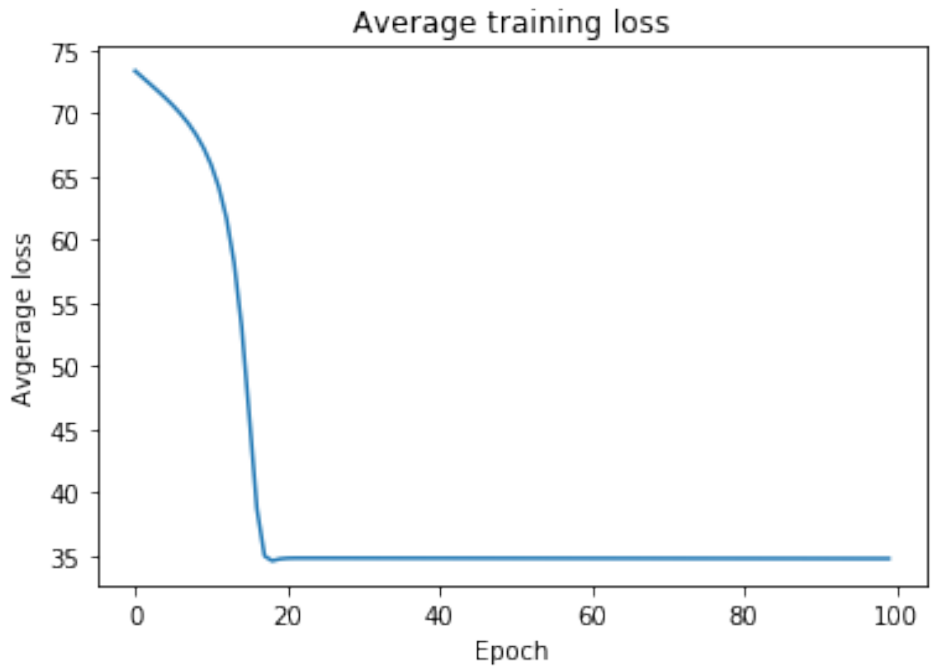

Figure 3.14: Average loss during training

Chapter 4

Results

The two models in each section are trained on the same data and tested on the same data. For each of them the available dataset is split using the standard 70/30 train/test split.

4.1 Green bag detection

The models were trained on 500 images, and test performance is reported from 211 images. There are two evaluation metrics, the first is the mean squared error (MSE). This is calculated based on the number of green bags that are detected. The other metric is the intersection over union (IoU) which is calculated from the predicted bounding boxes. The performance of the models at the confidence threshold that minimises the MSE is shown in table 4.1.

Model	Threshold	MSE (Test)	Average IoU (Test)
Faster R-CNN	0.75	1.8626	0.6477
YOLO	0.37	1.2095	0.1552

Table 4.1: Green bag model performance

The MSE and average IoU at different confidence thresholds for the Faster R-CNN model are shown in figure 4.1 and figure 4.2. The threshold that

minimises the MSE is 0.75 and it is highlighted in the figures by a yellow diamond shape. The values for the YOLO model are shown in figure 4.3 and figure 4.4. For this model the highlighted threshold is 0.37. The YOLO model does not produce any boxes with a confidence above 0.79 which is why the IoU is 0.

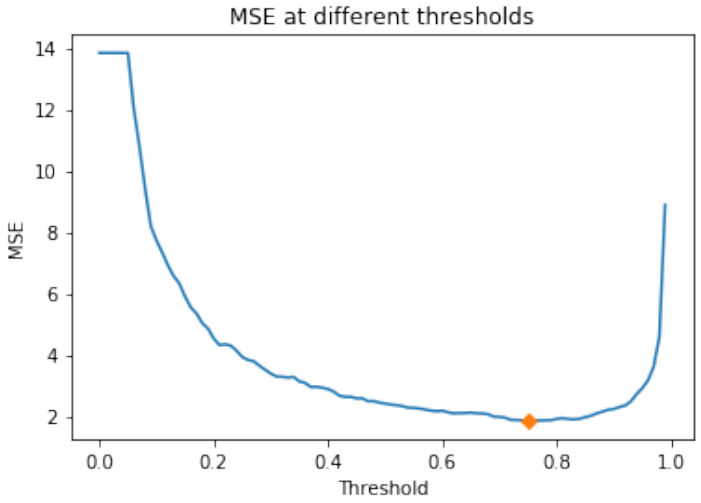

Figure 4.1: The MSE at different confidence thresholds for the Faster R-CNN model

4.2 Waste volume measurement

The models were trained on 72 images, and tested on 32 images. The model performance is listed in table 4.2.

Model	MSE (Train)	MSE (Test)	R^2
Baseline	73.7899	78.5538	0.8650
Deep learning	34.7859	584.4843	-

Table 4.2: Volume model performance

The error made by each model on every sample in the test set is shown in figure 4.5

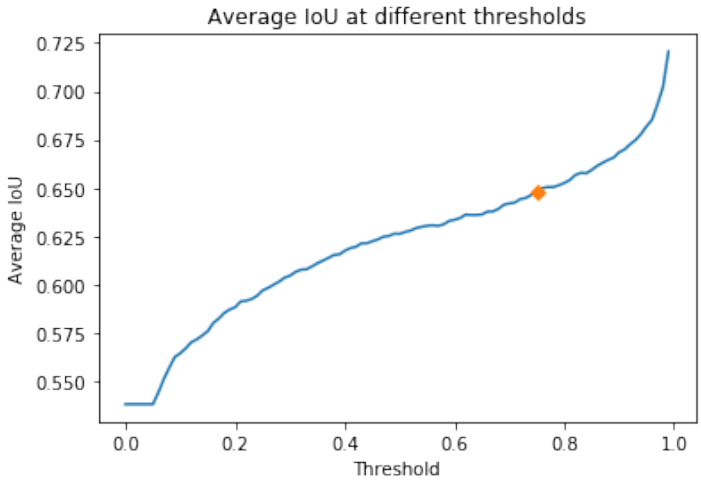

Figure 4.2: The average IoU at different confidence thresholds for the Faster R-CNN model

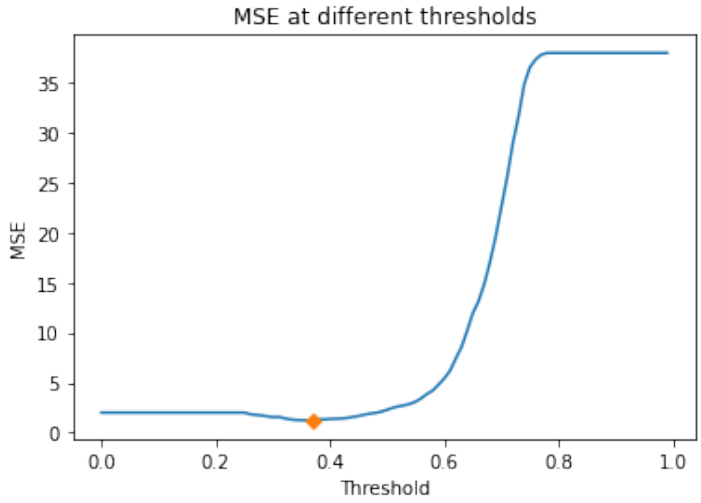

Figure 4.3: The MSE at different confidence thresholds for the YOLO model

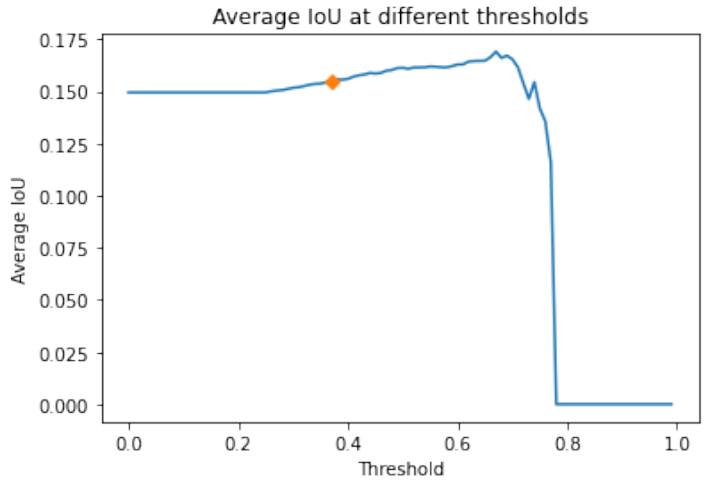

Figure 4.4: The average IoU at different confidence thresholds for the YOLO model

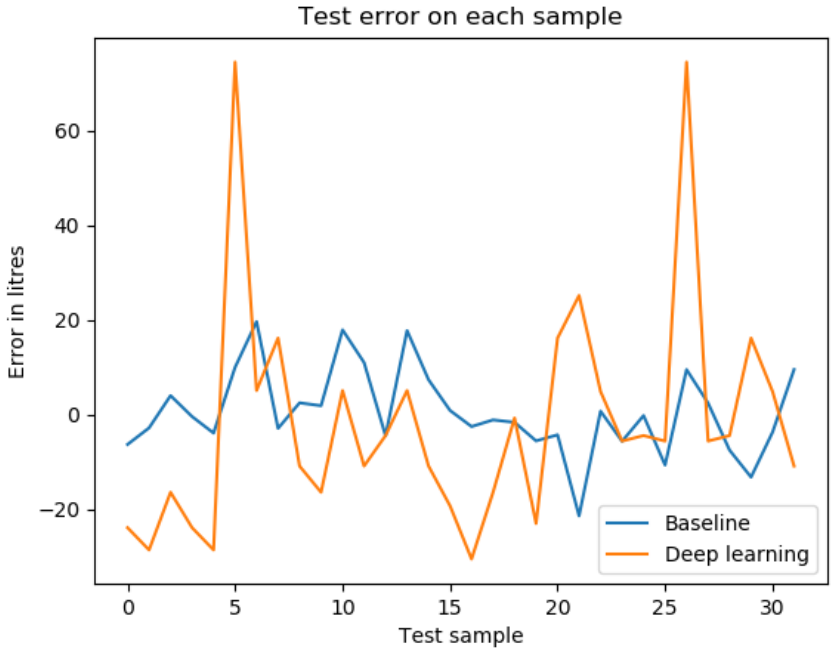

Figure 4.5: Error on each test sample

Chapter 5

Evaluation and discussion

5.1 Green bag detection

Both of the models have good enough performance to give useful information to Halden municipality. The ideal situation would be if the number of green bags could be associated with each customer. This could be achieved if it was possible to know which container is on what side of the truck. The bags could be counted based on the position of their bounding box. In that case the Faster R-CNN model should be used because it is significantly better at accurately predicting the position of the bags. In the current RFID system design this information is not available as it is possible for one of the antennas to pick up the RFID tag of both containers. The next best option is to count all of the bags from the two containers that are empited and then split the number between them. Even though the average IoU of the YOLO model isn't as good as the Faster R-CNN model, the YOLO model is a better choice for this use case since the MSE is lower.

In both situations the data cannot be used on a day by day basis to make judgements. It is possible that the customer did not have any food waste for that particular period and so even though no bags are detected it cannot be said that the customer does not use the green bags for food waste. Instead, it is necessary to collect the count over a longer period and then analyze it to find trends. This enables information gain on multiple levels. Data from

multiple customers in the same area can be compared to see if there are any particular neighbourhoods that does not use the bags. The municipality has access to additional information about the customers like their age. This can be used to see if there are any specific age groups that do not use the bags. In this way marketing campaigns can be targeted towards the customers who are not sorting their waste properly.

5.2 Volume measurement

If the ToF camera is placed in a position of the waste collection process that does not have reflective surfaces in the measurement area then the presented methods can be used to measure the volume. Another solution is to paint the garbage trucks in a non-reflective paint, however that is not a good long-term solution as the paint wears off.

If the volume is measured over time then the information can be used for two things. The first is charging customers based on how much waste they throw away instead of a flat rate that is the same for everyone. Like the green bag detection this would require a change in the RFID system. The other use case is optimising the truck routes. When a trend is observed the amount of waste from a container can be predicted and the routes can be adjusted such that the time spent emptying waste is reduced when there is still more room left in the truck.

The $R^2 = 0.8650$ value of the baseline model suggests that the model fits the data well which is expected as the volume is linear in nature. It is hard to determine whether or not the MSE of 78.6 is high as no data of actual waste was collected. Compared to the volumes of the cardboard boxes the error does make it hard to extract useful information from the images. It is possible that this would have been lower if a better ToF camera was used. The trained deep learning model only predicts the average of the samples in the training data which is why the training error is so low compared to the test error. It cannot be used for anything useful. It is likely that the network is too complex compared to the amount of training data that was available.

5.3 Machine learning on Raspberry Pi

Running the models on a Raspberry Pi was chosen for a couple of reasons. One Raspberry Pi was required in any case because the ToF camera requires one to work. The garbage trucks use a lot of power for collecting waste and the RFID system so the edge device options are somewhat limited, and using a device with a GPU could use too much power. Adding another Raspberry Pi does not increase the power usage of the system by much. In addition, a device capable of general purpose computation was required to interact with the external server.

The images are only of interest for training purposes. After the models are trained the only data that is interesting is the model output. This is a good reason for running the models on an edge device. The combined size of the image from the camera and the ToF camera is 4MB for each container. There are five trucks and on average each of them empty 750 containers every day. That equals 15GB of data that would have to be transferred over a mobile connection every day. Even when not considering the mobile data cost the time it takes to transfer the images on a unreliable mobile connection would make it hard to ensure all of the data is transferred.

There are two main challenges with running the models. The first is that PyTorch is not officially supported on the Raspberry Pi. The solution to this is building the project from source and excluding the parts that do not work. As only the base functionality and the torchvision package is required for this project it turned out to not be a problem. The other challenge is the limited computation power. This meant that it was only able to run the smallest version of the YOLO model. With more computational power it would be possible to base the model on one of the better YOLO models to increase the performance.

Chapter 6

Conclusion

The main goal of this project has been to improve the data flow coming from the garbage trucks in Halden municipality. This was achieved in the following ways:

1. A data capturing unit was designed and installed on one of the garbage trucks. It is capable of taking images of the waste and capture 3D point clouds of the waste when a container is emptied. The data is collected using a standard camera and a time of flight camera.

2. A model was created that is capable of counting how many green food waste bags there are in each container. This data can be used by Halden municipality to figure out how people are sorting their waste, and it lets them implement measures to improve the sorting behavior of their customers.

3. Another model was created that can use the 3D point cloud data to measure the waste volume. There are some challenges in the current environment which means the point clouds from the garbage trucks cannot be used. If the point clouds came from another point in the waste collection process they could be used.

List of Figures

LIST OF FIGURES

Bibliography

[1] Olugboja Adedeji and Zenghui Wang. Intelligent waste classification system using deep learning convolutional neural network. 2019.

[2] Basler AG. Basler blaze - time-of-flight (tof) camera, May 2021.

[3] Sasank Chilamkurthy and Torch Contributors. Transfer learning for computer vision tutorial, April 2021.

[4] The SciPy community. scipy ndimage percentile_filter, April 2021.

[5] Torch Contributors. Torchvision models, May 2021.

[6] FDA. Radio frequency identification (rfid), September 2018.

[7] The Raspberry Pi Foundation. Camera module v2, May 2021.

[8] Pieye GmbH. Nimbus 3d camera, May 2021.

[9] Bhiksha Raj Haohan Wang. On the origin of deep learning. 2017.

[10] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. 2015.

[11] Glenn Jocher, Alex Stoken, Jirka Borovec, NanoCode012, Ayush Chaurasia, TaoXie, Liu Changyu, Abhiram V, Laughing, tkianai, yxNONG, Adam Hogan, lorenzomammana, AlexWang1900, Jan Hajek, Laurentiu Diaconu, Marc, Yonghye Kwon, oleg, wanghaoyang0106, Yann Defretin, Aditya Lohia, ml5ah, Ben Milanko, Benjamin Fineran, Daniel Khromov, Ding Yiwei, Doug, Durgesh, and Francisco Ingham. ultralytics/yolov5: v5.0 - YOLOv5-P6 1280 models, AWS, Supervise.ly and YouTube integrations, April 2021.

[12] Texas Instruments Larry Li. Time-of-flight camera - an introduction, April 2021.

[13] Tsung-Yi Lin, Piotr Dollár, Ross Girshick, Kaiming He, Bharath Hariharan, and Serge Belongie. Feature pyramid networks for object detection. 2017.

[14] Tsung-Yi Lin, Michael Maire, Serge Belongie, Lubomir Bourdev, Ross Girshick, James Hays, Pietro Perona, Deva Ramanan, C. Lawrence Zitnick, and Piotr Dollár. Microsoft coco: Common objects in context. 2015.

[15] Aghilan M, Arun Kumar M, Mohammed Aafrid TS, Nirmal Kumar A, and Muthulakshmi S. Garbage waste classification using supervised deep learning techniques. 2020.

[16] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. 2016.

[17] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. 2016.

[18] Hamid Rezatofighi, Nathan Tsoi, JunYoung Gwak, Amir Sadeghian, Ian Reid, and Silvio Savarese. Generalized intersection over union: A metric and a loss for bounding box regression. 2019.

[19] Geir Thorsnæs. Halden - store norske leksikon, March 2021.

Appendix A

Neural network architecture

A.1 Faster R-CNN

The complete network architecture of the Faster R-CNN network after applying transfer learning.

```
FasterRCNN(
(transform): GeneralizedRCNNTransform(
  Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
  Resize(min_size=(800,), max_size=1333, mode='bilinear')
)
(backbone): BackboneWithFPN(
(body): IntermediateLayerGetter(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3),
  bias=False)
  (bn1): FrozenBatchNorm2d(64)
  (relu): ReLU(inplace=True)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1,
  ceil_mode=False)
  (layer1): Sequential(
    (0): Bottleneck(
      (conv1): Conv2d(64, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): FrozenBatchNorm2d(64)
```

```
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)
      , bias=False)
      (bn2): FrozenBatchNorm2d(64)
      (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): FrozenBatchNorm2d(256)
      (relu): ReLU(inplace=True)
      (downsample): Sequential(
        (0): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (1): FrozenBatchNorm2d(256)
      )
    )
    (1): Bottleneck(
      (conv1): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): FrozenBatchNorm2d(64)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)
      , bias=False)
      (bn2): FrozenBatchNorm2d(64)
      (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): FrozenBatchNorm2d(256)
      (relu): ReLU(inplace=True)
    )
    (2): Bottleneck(
      (conv1): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): FrozenBatchNorm2d(64)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)
      , bias=False)
      (bn2): FrozenBatchNorm2d(64)
      (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): FrozenBatchNorm2d(256)
      (relu): ReLU(inplace=True)
    )
  )
  (layer2): Sequential(
    (0): Bottleneck(
      (conv1): Conv2d(256, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): FrozenBatchNorm2d(128)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(2, 2)
      , padding=(1, 1), bias=False)
      (bn2): FrozenBatchNorm2d(128)
```

```
      (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): FrozenBatchNorm2d(512)
      (relu): ReLU(inplace=True)
      (downsample): Sequential(
        (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): FrozenBatchNorm2d(512)
      )
    )
    (1): Bottleneck(
      (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): FrozenBatchNorm2d(128)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1)
      , padding=(1, 1), bias=False)
      (bn2): FrozenBatchNorm2d(128)
      (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): FrozenBatchNorm2d(512)
      (relu): ReLU(inplace=True)
    )
    (2): Bottleneck(
      (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): FrozenBatchNorm2d(128)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1)
      , padding=(1, 1), bias=False)
      (bn2): FrozenBatchNorm2d(128)
      (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): FrozenBatchNorm2d(512)
      (relu): ReLU(inplace=True)
    )
    (3): Bottleneck(
      (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): FrozenBatchNorm2d(128)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1)
      , padding=(1, 1), bias=False)
      (bn2): FrozenBatchNorm2d(128)
      (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): FrozenBatchNorm2d(512)
      (relu): ReLU(inplace=True)
    )
  )
```

```
(layer3): Sequential(
  (0): Bottleneck(
    (conv1): Conv2d(512, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): FrozenBatchNorm2d(256)
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(2, 2)
    , padding=(1, 1), bias=False)
    (bn2): FrozenBatchNorm2d(256)
    (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): FrozenBatchNorm2d(1024)
    (relu): ReLU(inplace=True)
    (downsample): Sequential(
      (0): Conv2d(512, 1024, kernel_size=(1, 1), stride=(2, 2), bias=False)
      (1): FrozenBatchNorm2d(1024)
    )
  )
  (1): Bottleneck(
    (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): FrozenBatchNorm2d(256)
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1)
    , padding=(1, 1), bias=False)
    (bn2): FrozenBatchNorm2d(256)
    (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): FrozenBatchNorm2d(1024)
    (relu): ReLU(inplace=True)
  )
  (2): Bottleneck(
    (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): FrozenBatchNorm2d(256)
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1)
    , padding=(1, 1), bias=False)
    (bn2): FrozenBatchNorm2d(256)
    (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): FrozenBatchNorm2d(1024)
    (relu): ReLU(inplace=True)
  )
  (3): Bottleneck(
    (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): FrozenBatchNorm2d(256)
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1)
```

```
    , padding=(1, 1), bias=False)
    (bn2): FrozenBatchNorm2d(256)
    (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): FrozenBatchNorm2d(1024)
    (relu): ReLU(inplace=True)
  )
  (4): Bottleneck(
    (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): FrozenBatchNorm2d(256)
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1)
    , padding=(1, 1), bias=False)
    (bn2): FrozenBatchNorm2d(256)
    (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): FrozenBatchNorm2d(1024)
    (relu): ReLU(inplace=True)
  )
  (5): Bottleneck(
    (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): FrozenBatchNorm2d(256)
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1)
    , padding=(1, 1), bias=False)
    (bn2): FrozenBatchNorm2d(256)
    (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): FrozenBatchNorm2d(1024)
    (relu): ReLU(inplace=True)
  )
)
(layer4): Sequential(
  (0): Bottleneck(
    (conv1): Conv2d(1024, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): FrozenBatchNorm2d(512)
    (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(2, 2)
    , padding=(1, 1), bias=False)
    (bn2): FrozenBatchNorm2d(512)
    (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): FrozenBatchNorm2d(2048)
    (relu): ReLU(inplace=True)
    (downsample): Sequential(
      (0): Conv2d(1024, 2048, kernel_size=(1, 1), stride=(2, 2), bias=False)
```

```
        (1): FrozenBatchNorm2d(2048)
      )
    )
    (1): Bottleneck(
      (conv1): Conv2d(2048, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): FrozenBatchNorm2d(512)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1)
      , padding=(1, 1), bias=False)
      (bn2): FrozenBatchNorm2d(512)
      (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): FrozenBatchNorm2d(2048)
      (relu): ReLU(inplace=True)
    )
    (2): Bottleneck(
      (conv1): Conv2d(2048, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): FrozenBatchNorm2d(512)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1)
      , padding=(1, 1), bias=False)
      (bn2): FrozenBatchNorm2d(512)
      (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): FrozenBatchNorm2d(2048)
      (relu): ReLU(inplace=True)
    )
  )
)
(fpn): FeaturePyramidNetwork(
  (inner_blocks): ModuleList(
    (0): Conv2d(256, 256, kernel_size=(1, 1), stride=(1, 1))
    (1): Conv2d(512, 256, kernel_size=(1, 1), stride=(1, 1))
    (2): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1))
    (3): Conv2d(2048, 256, kernel_size=(1, 1), stride=(1, 1))
  )
  (layer_blocks): ModuleList(
    (0): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  )
  (extra_blocks): LastLevelMaxPool()
```

```
)
)
(rpn): RegionProposalNetwork(
(anchor_generator): AnchorGenerator()
(head): RPNHead(
  (conv): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (cls_logits): Conv2d(256, 3, kernel_size=(1, 1), stride=(1, 1))
  (bbox_pred): Conv2d(256, 12, kernel_size=(1, 1), stride=(1, 1))
)
)
(roi_heads): RoIHeads(
(box_roi_pool): MultiScaleRoIAlign()
(box_head): TwoMLPHead(
  (fc6): Linear(in_features=12544, out_features=1024, bias=True)
  (fc7): Linear(in_features=1024, out_features=1024, bias=True)
)
(box_predictor): FastRCNNPredictor(
  (cls_score): Linear(in_features=1024, out_features=2, bias=True)
  (bbox_pred): Linear(in_features=1024, out_features=8, bias=True)
)
)
)
```

A.2 YOLOv5s

The complete network architecture of the YOLOv5s network after applying transfer learning.

```
Model(
(model): Sequential(
(0): Focus(
  (conv): Conv(
    (conv): Conv2d(12, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (act): SiLU(inplace=True)
  )
)
(1): Conv(
```

```
  (conv): Conv2d(32, 64, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
  (act): SiLU(inplace=True)
)
(2): C3(
  (cv1): Conv(
    (conv): Conv2d(64, 32, kernel_size=(1, 1), stride=(1, 1))
    (act): SiLU(inplace=True)
  )
  (cv2): Conv(
    (conv): Conv2d(64, 32, kernel_size=(1, 1), stride=(1, 1))
    (act): SiLU(inplace=True)
  )
  (cv3): Conv(
    (conv): Conv2d(64, 64, kernel_size=(1, 1), stride=(1, 1))
    (act): SiLU(inplace=True)
  )
  (m): Sequential(
    (0): Bottleneck(
      (cv1): Conv(
        (conv): Conv2d(32, 32, kernel_size=(1, 1), stride=(1, 1))
        (act): SiLU(inplace=True)
      )
      (cv2): Conv(
        (conv): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (act): SiLU(inplace=True)
      )
    )
  )
)
(3): Conv(
  (conv): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
  (act): SiLU(inplace=True)
)
(4): C3(
  (cv1): Conv(
    (conv): Conv2d(128, 64, kernel_size=(1, 1), stride=(1, 1))
    (act): SiLU(inplace=True)
  )
  (cv2): Conv(
```

```
    (conv): Conv2d(128, 64, kernel_size=(1, 1), stride=(1, 1))
    (act): SiLU(inplace=True)
  )
  (cv3): Conv(
    (conv): Conv2d(128, 128, kernel_size=(1, 1), stride=(1, 1))
    (act): SiLU(inplace=True)
  )
  (m): Sequential(
    (0): Bottleneck(
      (cv1): Conv(
        (conv): Conv2d(64, 64, kernel_size=(1, 1), stride=(1, 1))
        (act): SiLU(inplace=True)
      )
      (cv2): Conv(
        (conv): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (act): SiLU(inplace=True)
      )
    )
    (1): Bottleneck(
      (cv1): Conv(
        (conv): Conv2d(64, 64, kernel_size=(1, 1), stride=(1, 1))
        (act): SiLU(inplace=True)
      )
      (cv2): Conv(
        (conv): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (act): SiLU(inplace=True)
      )
    )
    (2): Bottleneck(
      (cv1): Conv(
        (conv): Conv2d(64, 64, kernel_size=(1, 1), stride=(1, 1))
        (act): SiLU(inplace=True)
      )
      (cv2): Conv(
        (conv): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (act): SiLU(inplace=True)
      )
    )
  )
```

```
)
(5): Conv(
  (conv): Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
  (act): SiLU(inplace=True)
)
(6): C3(
  (cv1): Conv(
    (conv): Conv2d(256, 128, kernel_size=(1, 1), stride=(1, 1))
    (act): SiLU(inplace=True)
  )
  (cv2): Conv(
    (conv): Conv2d(256, 128, kernel_size=(1, 1), stride=(1, 1))
    (act): SiLU(inplace=True)
  )
  (cv3): Conv(
    (conv): Conv2d(256, 256, kernel_size=(1, 1), stride=(1, 1))
    (act): SiLU(inplace=True)
  )
  (m): Sequential(
    (0): Bottleneck(
      (cv1): Conv(
        (conv): Conv2d(128, 128, kernel_size=(1, 1), stride=(1, 1))
        (act): SiLU(inplace=True)
      )
      (cv2): Conv(
        (conv): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (act): SiLU(inplace=True)
      )
    )
    (1): Bottleneck(
      (cv1): Conv(
        (conv): Conv2d(128, 128, kernel_size=(1, 1), stride=(1, 1))
        (act): SiLU(inplace=True)
      )
      (cv2): Conv(
        (conv): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (act): SiLU(inplace=True)
      )
    )
```

```
    (2): Bottleneck(
      (cv1): Conv(
        (conv): Conv2d(128, 128, kernel_size=(1, 1), stride=(1, 1))
        (act): SiLU(inplace=True)
      )
      (cv2): Conv(
        (conv): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (act): SiLU(inplace=True)
      )
    )
  )
)
(7): Conv(
  (conv): Conv2d(256, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
  (act): SiLU(inplace=True)
)
(8): SPP(
  (cv1): Conv(
    (conv): Conv2d(512, 256, kernel_size=(1, 1), stride=(1, 1))
    (act): SiLU(inplace=True)
  )
  (cv2): Conv(
    (conv): Conv2d(1024, 512, kernel_size=(1, 1), stride=(1, 1))
    (act): SiLU(inplace=True)
  )
  (m): ModuleList(
    (0): MaxPool2d(kernel_size=5, stride=1, padding=2, dilation=1, ceil_mode=False)
    (1): MaxPool2d(kernel_size=9, stride=1, padding=4, dilation=1, ceil_mode=False)
    (2): MaxPool2d(kernel_size=13, stride=1, padding=6, dilation=1, ceil_mode=False)
  )
)
(9): C3(
  (cv1): Conv(
    (conv): Conv2d(512, 256, kernel_size=(1, 1), stride=(1, 1))
    (act): SiLU(inplace=True)
  )
  (cv2): Conv(
    (conv): Conv2d(512, 256, kernel_size=(1, 1), stride=(1, 1))
    (act): SiLU(inplace=True)
```

```
  )
  (cv3): Conv(
    (conv): Conv2d(512, 512, kernel_size=(1, 1), stride=(1, 1))
    (act): SiLU(inplace=True)
  )
  (m): Sequential(
    (0): Bottleneck(
      (cv1): Conv(
        (conv): Conv2d(256, 256, kernel_size=(1, 1), stride=(1, 1))
        (act): SiLU(inplace=True)
      )
      (cv2): Conv(
        (conv): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (act): SiLU(inplace=True)
      )
    )
  )
)
(10): Conv(
  (conv): Conv2d(512, 256, kernel_size=(1, 1), stride=(1, 1))
  (act): SiLU(inplace=True)
)
(11): Upsample(scale_factor=2.0, mode=nearest)
(12): Concat()
(13): C3(
  (cv1): Conv(
    (conv): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1))
    (act): SiLU(inplace=True)
  )
  (cv2): Conv(
    (conv): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1))
    (act): SiLU(inplace=True)
  )
  (cv3): Conv(
    (conv): Conv2d(256, 256, kernel_size=(1, 1), stride=(1, 1))
    (act): SiLU(inplace=True)
  )
  (m): Sequential(
    (0): Bottleneck(
```

```
      (cv1): Conv(
        (conv): Conv2d(128, 128, kernel_size=(1, 1), stride=(1, 1))
        (act): SiLU(inplace=True)
      )
      (cv2): Conv(
        (conv): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (act): SiLU(inplace=True)
      )
    )
  )
)
(14): Conv(
  (conv): Conv2d(256, 128, kernel_size=(1, 1), stride=(1, 1))
  (act): SiLU(inplace=True)
)
(15): Upsample(scale_factor=2.0, mode=nearest)
(16): Concat()
(17): C3(
  (cv1): Conv(
    (conv): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1))
    (act): SiLU(inplace=True)
  )
  (cv2): Conv(
    (conv): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1))
    (act): SiLU(inplace=True)
  )
  (cv3): Conv(
    (conv): Conv2d(128, 128, kernel_size=(1, 1), stride=(1, 1))
    (act): SiLU(inplace=True)
  )
  (m): Sequential(
    (0): Bottleneck(
      (cv1): Conv(
        (conv): Conv2d(64, 64, kernel_size=(1, 1), stride=(1, 1))
        (act): SiLU(inplace=True)
      )
      (cv2): Conv(
        (conv): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (act): SiLU(inplace=True)
```

```
      )
    )
  )
)
(18): Conv(
  (conv): Conv2d(128, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
  (act): SiLU(inplace=True)
)
(19): Concat()
(20): C3(
  (cv1): Conv(
    (conv): Conv2d(256, 128, kernel_size=(1, 1), stride=(1, 1))
    (act): SiLU(inplace=True)
  )
  (cv2): Conv(
    (conv): Conv2d(256, 128, kernel_size=(1, 1), stride=(1, 1))
    (act): SiLU(inplace=True)
  )
  (cv3): Conv(
    (conv): Conv2d(256, 256, kernel_size=(1, 1), stride=(1, 1))
    (act): SiLU(inplace=True)
  )
  (m): Sequential(
    (0): Bottleneck(
      (cv1): Conv(
        (conv): Conv2d(128, 128, kernel_size=(1, 1), stride=(1, 1))
        (act): SiLU(inplace=True)
      )
      (cv2): Conv(
        (conv): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (act): SiLU(inplace=True)
      )
    )
  )
)
(21): Conv(
  (conv): Conv2d(256, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
  (act): SiLU(inplace=True)
)
```

```
(22): Concat()
(23): C3(
  (cv1): Conv(
    (conv): Conv2d(512, 256, kernel_size=(1, 1), stride=(1, 1))
    (act): SiLU(inplace=True)
  )
  (cv2): Conv(
    (conv): Conv2d(512, 256, kernel_size=(1, 1), stride=(1, 1))
    (act): SiLU(inplace=True)
  )
  (cv3): Conv(
    (conv): Conv2d(512, 512, kernel_size=(1, 1), stride=(1, 1))
    (act): SiLU(inplace=True)
  )
  (m): Sequential(
    (0): Bottleneck(
      (cv1): Conv(
        (conv): Conv2d(256, 256, kernel_size=(1, 1), stride=(1, 1))
        (act): SiLU(inplace=True)
      )
      (cv2): Conv(
        (conv): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (act): SiLU(inplace=True)
      )
    )
  )
)
(24): Detect(
  (m): ModuleList(
    (0): Conv2d(128, 18, kernel_size=(1, 1), stride=(1, 1))
    (1): Conv2d(256, 18, kernel_size=(1, 1), stride=(1, 1))
    (2): Conv2d(512, 18, kernel_size=(1, 1), stride=(1, 1))
  )
)
)
)
```