Deep Learning Garbage sorting line using WLKATA Robot

A dissertation submitted in partial fulfilment of

the requirements for the degree of

BACHELOR OF SCIENCE in Computer Science

in

The Queen's University of Belfast

by

Scott Lam-McGonnell

8th of May 2022

SCHOOL OF ELECTRONICS, ELECTRICAL ENGINEERING and COMPUTER SCIENCE

CSC3002 – COMPUTER SCIENCE PROJECT

Dissertation Cover Sheet

A signed and completed cover sheet must accompany the submission of the Software Engineering dissertation submitted for assessment.

Work submitted without a cover sheet will **NOT** be marked.

Student Name:	Scott Lam-McGonnell	Student Number:	40227918
Project Title:	MV01: Deep Learning Garbage Sorting Line using WLKATA Robot		
Supervisor:	Dr. Mien Van		

Declaration of Academic Integrity

Before submitting your dissertation please check that the submission:

1. Has a full bibliography attached laid out according to the guidelines specified in the Student Project Handbook
2. Contains full acknowledgement of all secondary sources used (paper-based and electronic)
3. Does not exceed the specified page limit
4. Is clearly presented and proof-read
5. Is submitted on, or before, the specified or agreed due date. Late submissions will only be accepted in exceptional circumstances or where a deferment has been granted in advance.

By submitting your dissertation you declare that you have completed the tutorial on plagiarism at http://www.qub.ac.uk/cite2write/introduction5.html and are aware that it is an academic offence to plagiarise. You declare that the submission is your own original work. No part of it has been submitted for any other assignment and you have acknowledged all written and electronic sources used.

6. If selected as an exemplar, I agree to allow my dissertation to be used as a sample for future students. (Please delete this if you do not agree.)

Student's signature	Scott Lam-McGonnell	*Date of submission* 8th of May 2022

Acknowledgements

I would like to offer a huge thank Dr. Mien Van for the opportunity to undertake this task, for the extensive support he's provided, for the resources he's trusted me with and for the time he's taken out of his day, time and time again, to answer my questions and point me in the right direction. A special thank you to my friends and family, whose emotional support and constant reassurance that I was capable of completing and working through the task was essential in keeping me going and persevering through challenges faced.

Abstract

Over the past few years, object detection has helped play a crucial role in many technologies we see today. From mass surveillance to detecting cancer cells, their application is endless. This paper overviews the possibility of using this technology in tandem with robotics, to implement an application capable of detecting and sorting certain types of garbage. The link to the project is listed here: https://gitlab2.eeecs.qub.ac.uk/40227918/garbage-classifcation [1]

CONTENTS

1 INTRODUCTION TO PROBLEM

Garbage classification has always been an import issue in environmental protection, and as the worldwide economy and therefore peoples living standards are rapidly improving, the amount of garbage produced increases. This is due to the fact that societies today are heavily consumer-based, and the more money people make, the more they spend. According to the World's Bank *What a Waste 2.0: A Global Snapshot of Solid Waste Management to 2050* report, global waste is expected to increase by 70% by 2050. It also states that as countries rise in income levels, the quantity of recyclables in the waste stream increases [2].

This project is aimed at implementing a garbage sorting solution that can act as a proof-of-concept for recycling centres and help provide the basis for a real-world, low-cost solution that could be deployed. To achieve this, a deep neural network algorithm will be used for object detection, being fed a live video stream from a camera. From there, the information will be used to manoeuvre a robotic arm to the object, and sort it based on the object's classification.

1.1 PROBLEM AREA

The main area this project intends to target Material Recovery Centres. At MRC's, materials are separated into different types of materials by machinery or hand. One of the biggest issues recycling centres face is having to manually remove types of garbage that aren't supposed to be there, slowing down the entire process. These items can sometimes be missed and can result in damage being caused to machinery [3].

There are also problems that the workers face in these MRCs, with the major issue being health and safety. They are constantly at risk of chemical and dust exposure, as recyclable materials can generate a lot of dust containing micro-particles of plastics, glass, biohazards, and other respiratory irritants. Workers are also at risk of injury as they must often twist, reach, jump and stoop to sort materials. These repetitive motions, coupled with long shifts contribute to stress and injury [4].

This project aims to tackle those issues by implementing a deep learning algorithm to automatically detect and using a robotic arm to separate recyclables, the margin for human error is removed. This would also help reduce the reliance and strain on the current workers at MRCs.

1.2 ROBOTICS

Robotics have been used is applications such as manufacturing, for processes such as material handling and processing operations. The first industrial robot used in production was installed in 1961, to unload parts from a die-casting operation. Since then, industrial robots have improved dramatically and can be seen in use in almost all manufacturing facilities around the globe. An industrial robot is defined as "a reprogrammable, multifunctional manipulator designed to move materials, parts, tools, or specialised devices through variable programmed motions for the performance of a variety of tasks [5]".

Industrial robots are made up of a sequence of link and joint connections. These joints are the movable components of the robots, enabling relative motion between the links.

Robotics used in manufacturing take on repetitive tasks, helping to streamline the overall workflow of the process at hand. Many jobs include a high volume of materials, which when given these tasks to humans, can result in fatigue, stress, and even physical harm. Furthermore, humas can become distracted or lose focus given the repetitive nature of some of the work at hand, leading them to make errors. Robots can help fix this, as they are not prone to fatigue or stress. Combined with high level machine learning, robots can also perform their tasks with high accuracy [6].

Over the past few years, robotics has been introduced into recycling centres to help aid the process of sorting these materials alongside humans. While most MRCs implement bulk processing methods to help sort the different types of recyclables, e.g., magnets can be used to separate metals from the rest of the waste, these options come with disadvantages, for example, magnets do not attract aluminium. There are many more materials which bulk sortation is not feasible [7].

Introducing waste specific sorting robots help combat these issues faced at MRCs, and with improvements in artificial intelligence aided computer vision and robotic arms, it is possible to use robotics to sort and separate the materials. These AI-vision system detect different types of materials, and a robotic arm is manoeuvred to pick up the item and sort them into different chutes based on the type of material. Robotic waste sorting systems use multiple gripping strategies, such as using 'finger-like' grippers or suction cups, based on the material they're responsible for sorting.

"The main components of a robotic waste sorting system are the following:

- Robotic arm
- Gripper
- AI-vision system
- Conveyor system [7]”

1.3 Deep Learning for Image Classification and Object Detection

Deep learning refers to subset of machine learning, where neural networks with three or more layers are used. “Deep learning neural networks attempt to mimic the human brain through a combination of data inputs, weights, and bias. These elements work together to accurately recognise, classify, and describe objects within the data [8].”

Image classification has been a relatively straightforward task for deep neural networks for some time, with the use of Convolution Neural Networks becoming the standard for image classification tasks. However, a model capable of object detection must go beyond the ability of traditional CNNs and implement new architecture allowing it to localise the classified objects within the image itself. These models must also be capable of detecting multiple objects per image and classify them accordingly, as there are very few scenarios where a particular frame would only contain one object.

Furthermore, it is imperative that these models can perform this in real-time, to enable their use in real-world applications.

There have been several different types of algorithms developed over the years that have made this possible, and as of today, developers have a range of algorithms they can choose from to implement based on their needs.

Object detection models can be split into two categories: one-stage and two-stage object detection algorithms. Two-stage detectors work by first finding a region of interest where they predict a high likelihood of it containing an object, and then using that cropped region for image classification. Popular variations of these include Faster R-CNN and Mask R-CNN networks. One-stage detectors predict bounding boxes over the image without the region proposal step, allowing them to perform object detection at a much faster rate [9].

As real-time object detection was imperative for the scope of this task, one-stage detectors were the optimal choice for their much faster computational time.

2 SOLUTION DESCRIPTION AND SYSTEM REQUIREMENTS

2.1 SOLUTION DESCRIPTION

The solution to this task will be deemed successful if the system is able to correctly identify the different types of recyclable materials, localise the object within the frame, and move the robotic arm to sort the detected object respective to its classification.

The first step to figuring out the solution to this task, is to first figure out what Object Detection model would be most suitable

2.2 OBJECT DETECTION MODELS

In terms of the one-stage detectors to consider, the three most popular and widely used algorithms are:

- YOLO (You Only Look Once)
- SSD (Single Shot Multi-Box Detector)
- RetinaNet

The main means of comparison for these models are mean average precision (MAP) and frames per second (FPS). For the task at hand, it would be reasonable to take a lower MAP for a higher number of FPS, as real-time object detection requires the data to be processed from the video stream quickly, as to not incur long delays between the current frame of the video input and the current output of the model.

2.2.1 RetinaNet

RetinaNet is an algorithm that utilises Residual Network (ResNet) and Feature Pyramid Network (FPN) structures as its backbone. ResNet is a way to structure DNNs to help solve the ‘vanishing gradient’ problem, encountered when training artificial neural networks with gradient-based learning methods and backpropagation. When this happens, as more layers are added to ANNs, the gradients of the loss functions approach zero, leading to an inability to properly update the weights of the network and therefore train it. This is not an issue when using networks with only a few layers, however for state-of-the art image classification, these networks are many layers deep.

ResNet solves this issue by using residual connections, that skip one or more layers within the deep neural network

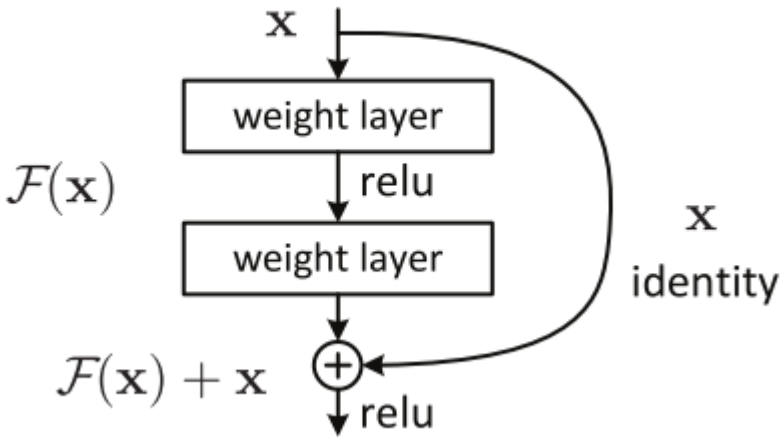

Figure 1: A Residual Block

"The residual connections provide a way to add a value to a layer without it having gone through an activation function, resulting in a higher overall derivative of the block [10]."

Feature Pyramid Networks are feature extractors designed with accuracy and speed in mind. They are composed of a bottom-up and a top-down pathway. Spatial resolution decreases during the bottom-up process, and with more high-level structures detected, the semantic value for each layer increases [11]. They provide a "top-down pathway to construct higher resolution layers from a semantic rich layer [12]."

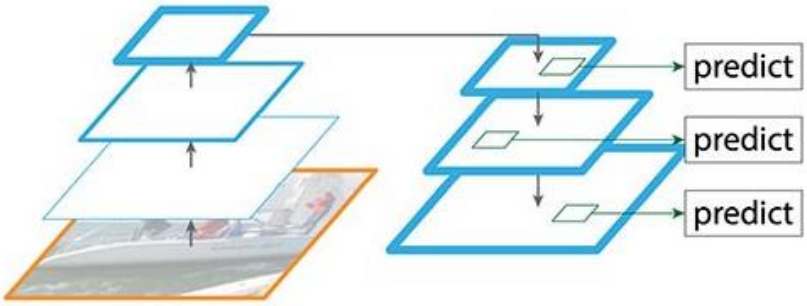

Figure 2: Feature Pyramid Network

The RetinaNet architecture is made up of four major components:

- The bottom-up pathway – ResNet. This calculates the feature maps at different scales.

- The top-down pathway (FPN) and lateral connections. The top-down pathway upsamples the feature maps from higher pyramid levels, and the lateral connections merge the top-down layers and the bottom-up layers.
- Classification network – this predicts the probability of an object being present at each spatial location for each anchor box.
- Regression subnetwork – this regresses the offset for the bounding boxes from the anchor boxes for each detected object. [13]

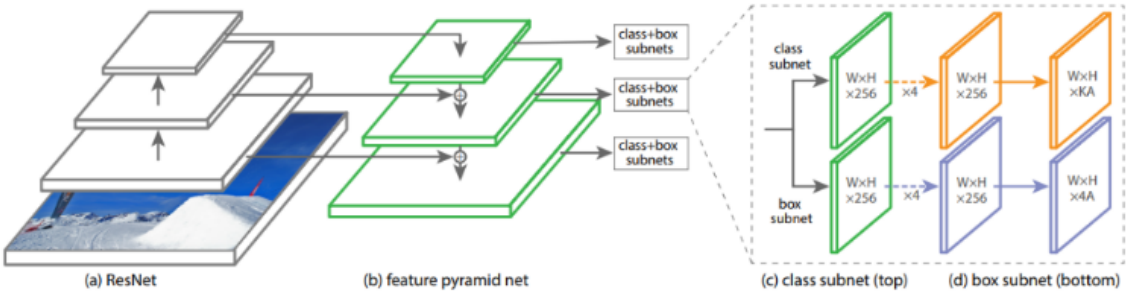

Figure 3: RetinaNet Architecture

RetinaNet also makes use of Focal Loss, which is a loss function to handle the imbalance problem that one-stage object detection models face. This is due to an extreme foreground-background class imbalance problem due to dense sampling on anchor boxes. In RetinaNet, each pyramid layer can produce thousands of anchor boxes, only a few of them will be assigned to a ground-truth object while the vast majority will be background. These can collectively overwhelm the model, so Focal Loss reduces the loss contribution from easy examples and increases the importance of correcting misclassified examples [14].

2.2.2 Single Shot Multi-Box Detector

SSD is based on a "feed-forward convolutional network that produces a fixed-size collection of bounding boxes and scores for the presence of object class instances in those boxes [15]." The early network layers are based on VGG-16, which was a convolution neural network that is still considered to be one of the best vision model architectures today.

SSD discards the fully connected layers and replaces them with a set of auxiliary convolution layers to enable the model to "extract features at multiple scales and progressively decrease the size of the input to each subsequent layer [16]."

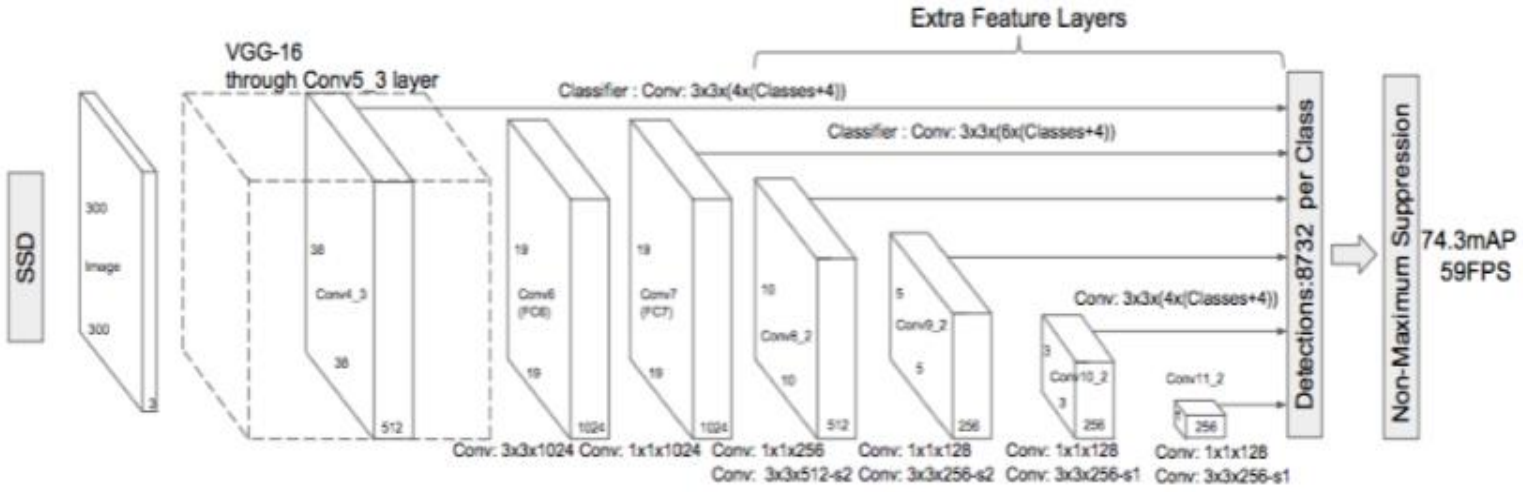

Figure 4: Single Shot MultiBox Detector Architecture

"The bounding box regression technique is a modified version of MultiBox, a method for fast class-agnostic bounding box coordinate proposals. It combined two loss functions:

- Confidence Loss, which measures how confident the network is of the object of the computed bounding box
- Location Loss, which measures how far away the networks predicted bounding boxes are from the ground truth [16]."

Multibox starts with priors – pre-computed, fixed size bounding boxes that closely match the distribution of ground truth boxes. These priors are selected so that their intersection over union ratio is greater than 0.5, which helps provide a strong starting point for the bounding box regression algorithm, rather than starting the predictions from random coordinates [17]. In SSD however, "every feature map cell is associated with bounding boxes of different dimensions and aspect ratios, where these priors are manually chosen. [16]"

The feature maps are representative of the dominant image features at different scales, so running MultiBox on multiple feature maps increases the likelihood of any object to be detected and classified, regardless of size. As the backbone of SSD is the VGG-16 architecture, the feature maps generated are from the VGG-16 network, as shown below [16].

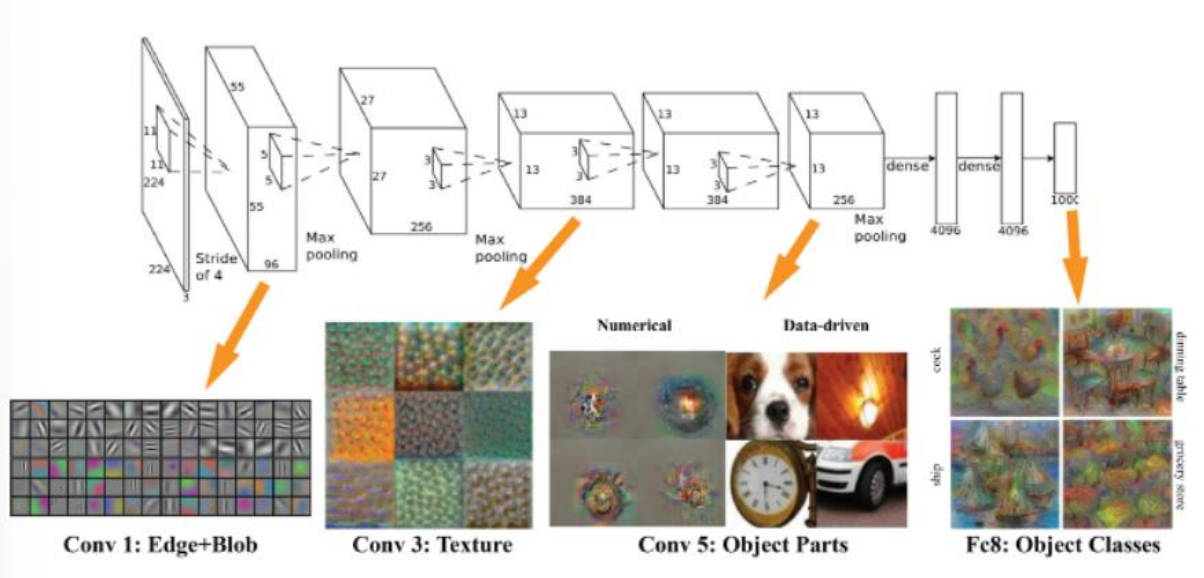

Figure 5: VGG Feature Map Visualisation

2.2.3 You Only Look Once

YOLO treats its object detection as a regression problem, meaning they don't need a complex pipeline to detect objects and predict their respective bounding boxes. YOLO's neural network is fed an image, and predictions are made.

YOLO is a convolutional neural network, consisting of 24 convolutional layers followed by 2 fully connected layers. The first 20 convolutional layers are followed by an average pooling layer and a fully connected layer, are pre-trained on the ImageNet dataset. The initial convolutional layers extract features from the images, while the fully connected layers the output probabilities and coordinates.

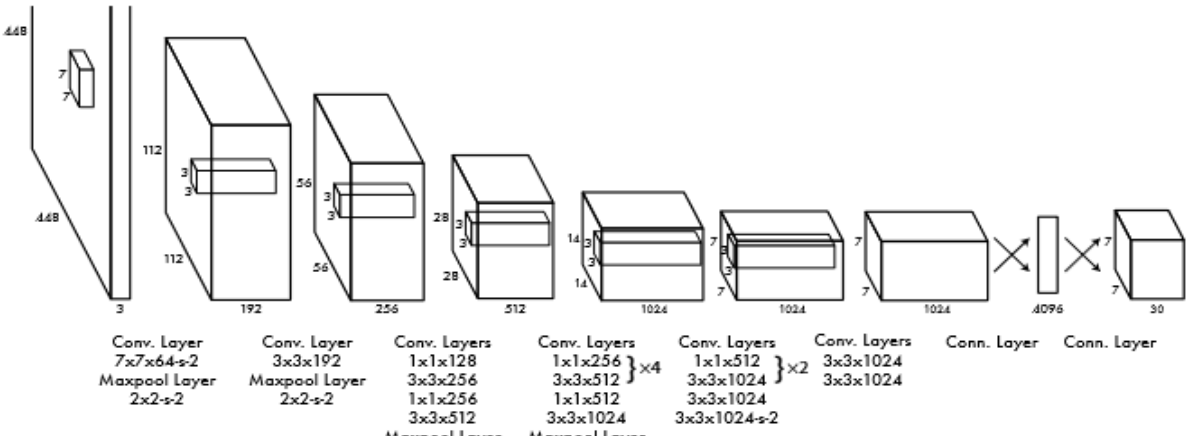

Figure 6: You Only Look Once Architecture

The model divides the input image into an *S x S grid*. “If the centre of an object falls into a grid cell, its that grid cell responsible for detecting the object.” Each cell predicts bounding boxes and confidence scores for those boxes, reflecting how confident the model is that an object is within the predicted bounding box. The confidence for YOLO is defined as the probability score of the object, multiplied by the intersection over union between the predicted box and the ground truth. Each grid cell also predicts class probabilities, given the grid cell has some confidence it has detected an object [18].

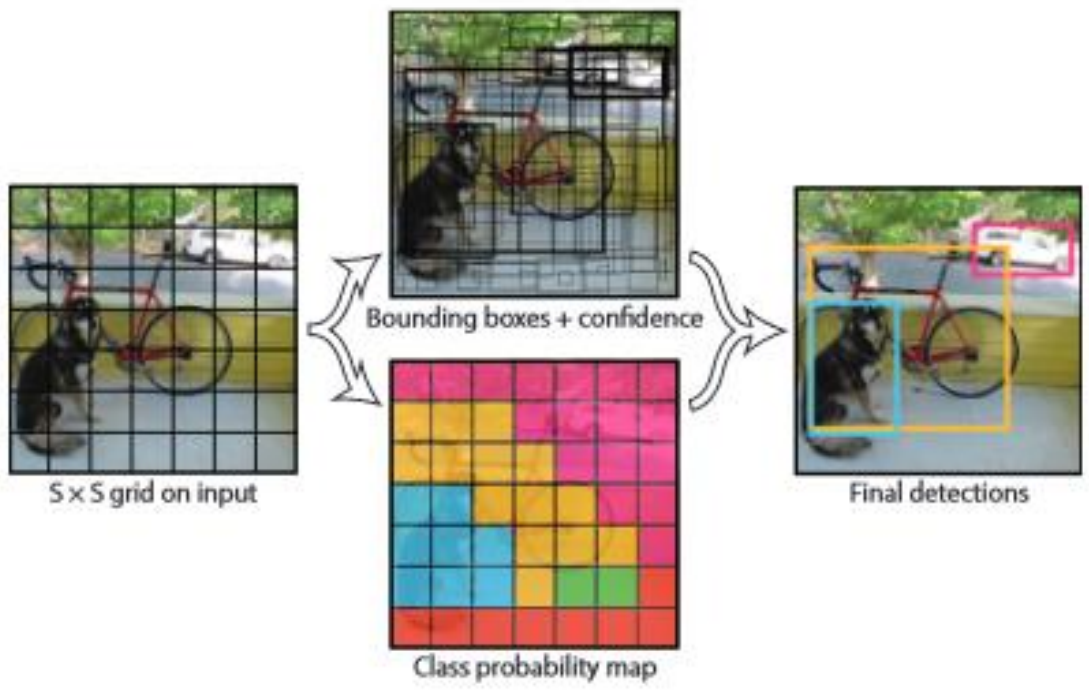

Figure 7: YOLO Model Detection

As each grid cell can predict multiple bounding boxes, a predictor is assigned to be “responsible” for predicting an object based on which prediction has the highest IOU with respect to the ground truth. This leads to “specialisation between the bounding box predictors, with each predictor getting better at specific tasks, such as predicting at certain sizes, aspect ratios or classes of objects, improving overall recall. [18]”

The loss used was an optimised version of sum-squared error, however it weights localisation errors equally with classification errors. As many grid cells of the image will not contain objects, this pushes the confidence scores of those cells towards zero, overpowering the gradient of the cells that do contain objects, leading to model instability. To combat this, it

increases in the loss from the bounding box coordinates and a decrease in the loss from confidence predictions for boxes that don't contain objects was made, helping to reflect that small deviations in large boxes matter less than in small boxes [18].

Different backbones can be used for the YOLO model, such as ResNet, VGG-16 and Darknet

2.2.4 Comparison of Object Detection Models

Improvements to the above-mentioned models have been made throughout the years, with the current YOLO model being on its 5th version. Below is a comparison of results comparing RetinaNet, SSD and YOLOv3 [19]:

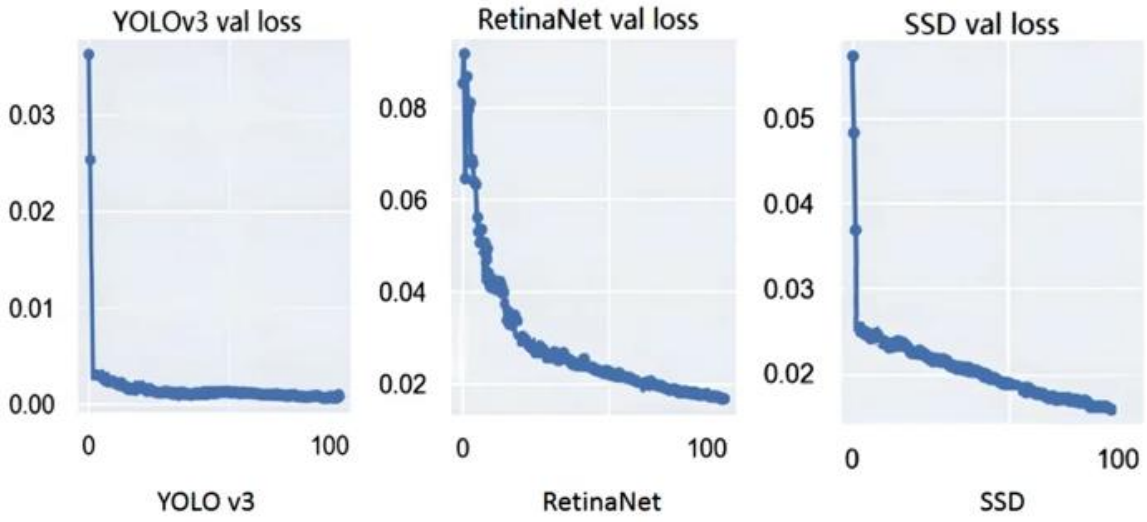

Figure 8: YOLOv3 VS RetinaNet VS SSD - Loss

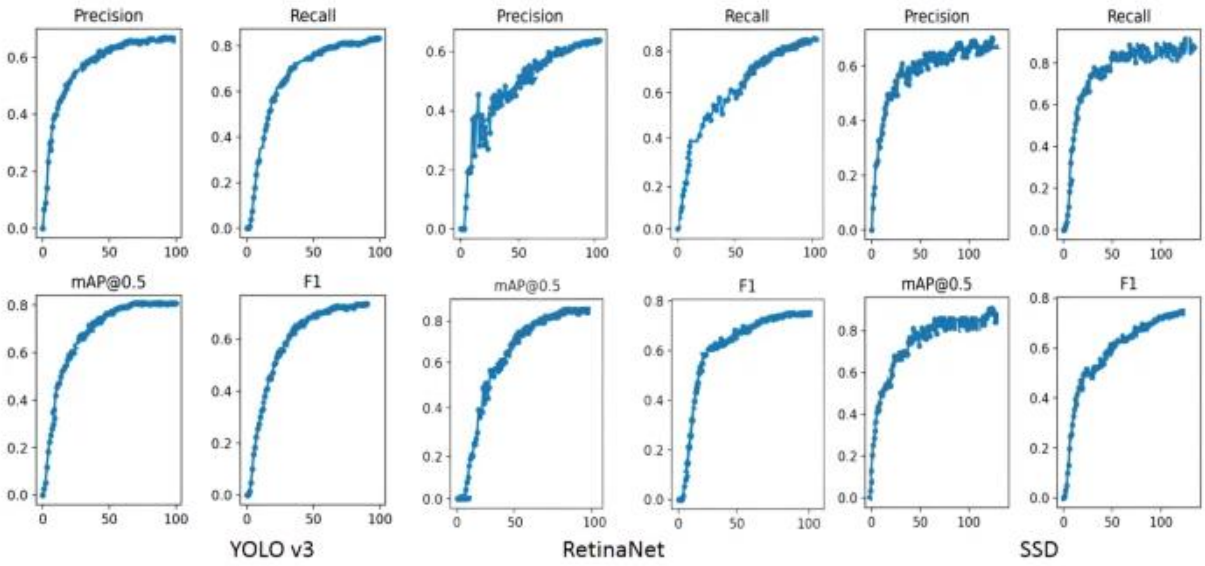

Figure 9: YOLOv3 vs RetinaNet vs SSD - Metrics

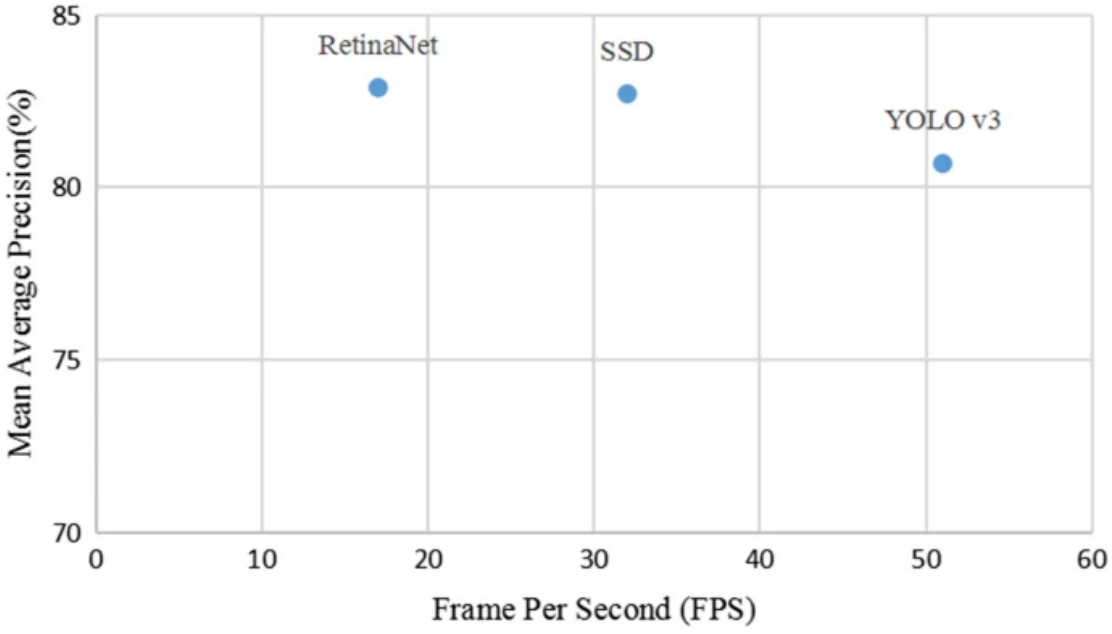

Figure 10: YOLOv3 VS RetinaNet VS SSD - MAP and FPS

Mean Average Precision is a metric used to evaluate object detection models. It is based on a Confusion Matrix, Intersection over Union, Recall, and Precision. It is calculated by finding the average precision for each class and taking an average over the number of classes [20].

$$\text{mAP} = \frac{1}{N} \sum_{i=1}^{N} \text{AP}_i$$

Figure 11: Mean-Average Precision Formula

As we can see from the above graphs, SSD and RetinaNet have a higher MAP by 2.2% and 1.8% respectively. However, the FPS of the YOLOv3 model greatly outperforms the other two, making it an ideal candidate for real-time object detection. As the difference in MAP between YOLOv3 and the other two models shown is only a few percent, coupled with the fact that YOLO is now on version 5, offering improvements in both MAP and FPS, the YOLO model was the idea choice.

YOLOv5 is a PyTorch implementation of the YOLOv4 model, offering a lightweight and easily transferable model. While YOLOv4 does slightly outperform YOLOv5 on accuracy, the image classification task at hand of identifying different types of recycling materials is not too demanding compared with other computer vision classification tasks. YOLOv5 seemed like the ideal choice due to the balance between FPS, MAP and the ability to have different implementations of the model to choose from based on the target machines hardware. For these reasons, YOLOv5 was chosen as the object detection model to be used.

2.3 DATASETS

To begin training the model on image classification and object detection, first there must be suitable datasets that we can use to train the model. Many datasets can be found on sites like Kaggle.com, a community for machine learning and data science where users are able to upload their own datasets.

For this task, a dataset containing images for the appropriate classes must first be found. When dealing with recycling materials, the main ones are aluminium, plastic, cardboard, and glass. However, as this is both an image classification and object detection task, the images

must have the appropriate labels to ensure the model is able to train itself on localisation of the image. The corresponding labels for each image to train a YOLO model must be in the *c, x, y, w, h* format, where:

- C is the class label of the object, as an integer
- X is the starting point of the bounding box, located in the upper left corner of the box
- Y is the y-axis starting point of the bounding box, located in the upper left corner
- W is the width of the bounding box, in pixels relative to the image resolution
- H is the height of the bounding box, in pixels relative to the image resolution

There are websites out there, such as Roboflow [21] that enable users to draw bounding boxes around the objects within an image and then output the appropriate label .txt for each respective image. Thankfully, however, Kaggle has datasets that were suitable for object detection that has the images labelled in the appropriate format.

The data must then be pre-processed, and split into train, test, and validation directories. There are many ways to do this, such as writing scripts in python to pre-process the images and split them into different directories. However, Roboflow provides resources to do this, with many types of pre-processing and image augmentation options available for object detection datasets, generating the new dataset and updated images for you. This seemed like the most logical choice, as when augmenting an image, in certain cases the corresponding labels would also have to be updated to ensure they still matched where the object was.

Once the dataset has been generated via Roboflow, a zipped version can be downloaded locally and used to train the model. The dataset contains images, and its label .txt file counterpart for each image.

2.4 IMAGE CLASSIFICATION AND OBJECT DETECTION

Once the appropriate dataset has been generated, training of the model for image classification and object detection can begin. Training on the model was done via cloning the YOLOv5 repository [22]. A *.yaml* file must be created for the model and placed in its *data* directory, where the file contains the path to the dataset relative to the *data* directory, as well as the location of the *train, test, and validation* directories relative to the path, followed by the number of classes and the names of the classes.

When this is completed, the *train.py* file of the repo can be run, via the following command:

python path/to/train.py --data path/to/data.yaml --weights yolov5s.pt --img 640

The usage of weights here is optional, by selecting a pre-defined weights file provided with the repo will train the model from a pre-trained state, or there is the option to leave that parameter blank and train the model from scratch. The default number of epochs for the model to train on is 300, but training will automatically stop earlier if it reaches an optimal state, with the best weights found being saved in *runs/train/name_of_model/weights/best.pt*

The model was trained from its pre-trained state, as the assumption is it will be considerably more accurate than training it from scratch, as its pre-trained state was trained on the *COCO 2017* dataset, with 80 different classes.

2.5 STEREOSCOPIC CAMERAS

After the model has been appropriately trained, a camera of some description will be used to capture a live video feed and in turn, pass each frame of the video feed to the model. The camera must be stereoscopic, i.e., a camera with at least two lenses. This allows the camera to capture 3D information of the video/frame, allowing information such as depth to be calculated. This is imperative as the robotic arm in use must be able to navigate and manoeuvre through a 3D space. Using a camera of this calibre will allow the object detection to take place, giving the location of the object detected in reference to the camera as well as the depth of the object from the camera.

Stereoscopic cameras work by imitating how eyes work, achieving this by using "two sensors a set distance apart to triangulate similar pixels from both 2D planes [23]."

"Each pixel in a digital camera image collects light that reaches the camera along a 3D ray. If a feature in the world can be identified as a pixel location in an image, we know that this feature lies on the 3D ray associated with that pixel. If we use multiple cameras, we can obtain multiple rays. Finding where these rays intersect tells us the 3D location of an object and its feature [23]."

2.6 SYSTEM REQUIREMENTS

- A functioning model based on the YOLOv5 architecture, updated for the provided classes
- The model must correctly be able to identify and localise the required objects
- Identify the pose of the object to ensure it can be grasped correctly

- Manoeuvring the WLKATA Mirobot in 3D space
- The arm must be able to navigate to the detected object and sort it based on its classification

3 DESIGN

3.1 YOLOV5 ARCHITECTURE

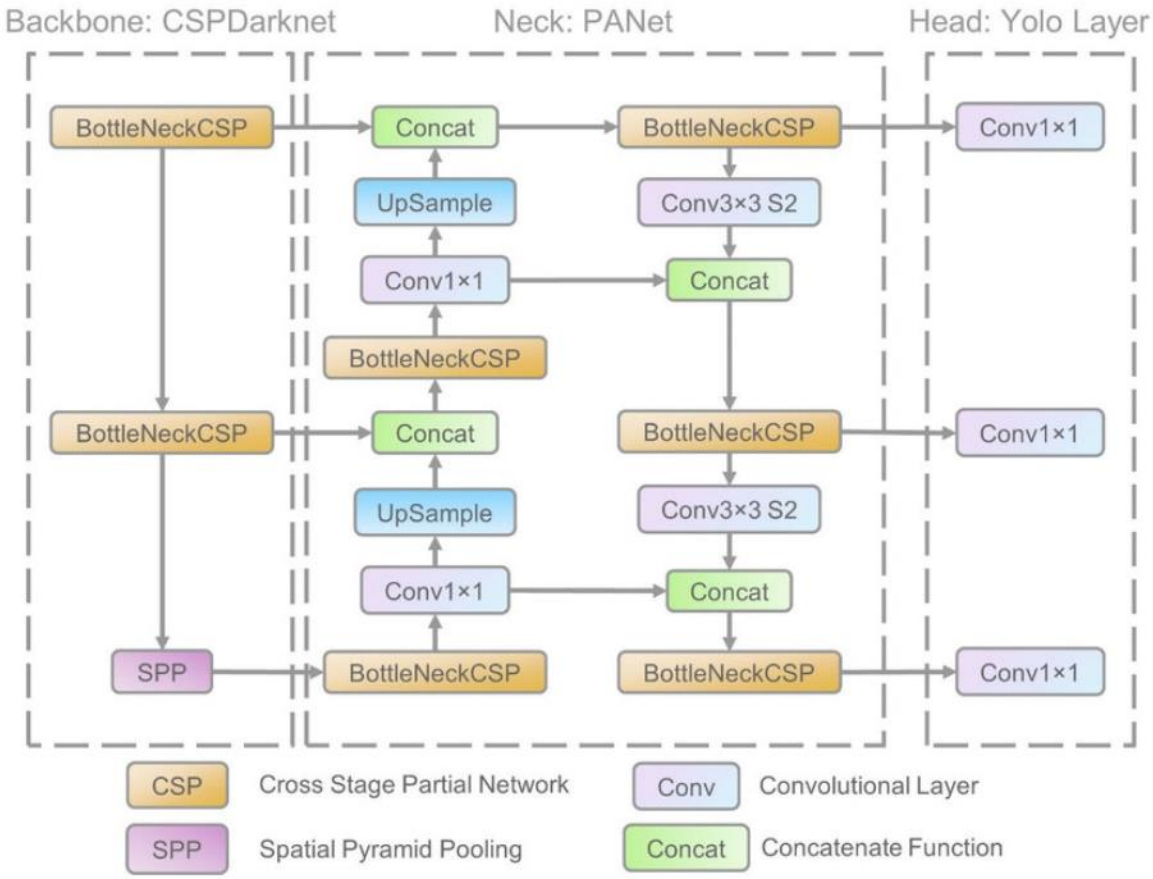

Figure 12: YOLOv5 Architecture

The above diagram is an overview of the YOLOv5 model architecture [24]. It is made up of three main stages: the backbone, the neck, and the head.

The backbone of YOLOv5 a PyTorch implementation based off of Darknet, an open source neural network framework written in C and Cuda [25]. Darknet53 is used as the backbone for YOLOv4, however YOLOv5 differs from previous releases. It utilises PyTorch instead of Darknet but keeps CSPDarknet53 as its backbone.

Cross Stage Partial Network (CSPNet) was integrated into Darknet, creating CSPDarknet as its backbone. CSPNet solves the problems of repeated gradient information in large-scale backbones and integrates the gradient changes into the feature map. This decreases the parameters and floating-point operations per second of the model, ensuring inference speed and accuracy, as well as reducing the model size [26].

The neck of the model is mainly used to generate feature pyramids. YOLOv5 uses PANet as its neck, which adopts a new FPN structure with enhanced bottom-up path, improving the propagation of low-level features. Adaptive feature pooling, which links the feature grid and all feature levels, is used to make useful information in each feature level propagate to the following subnetwork. PANet improves the use of accurate localisation in lower levels, enhancing the location accuracy of the object [24].

The head of the network, the YOLO layer [Figure 6] is used for the output of classification and object localisation.

The output layer of the YOLO model used had the number of classes overwritten, from its default 80 classes to 5 classes that will be used for this project – metal, glass, PET plastic, HDPE plastic, and cardboard.

3.2 CAMERA – ZED 2

The camera to be used to capture a live video feed and stream each frame into the YOLOv5 model is the ZED 2. A camera with depth sensing was a must, as in order to be able to navigate the robot through 3D space it is imperative to have a camera capable of producing values for an X, Y and Z axis.

The ZED 2 is a camera capable of neural depth sensing, spatial object detection, positional tracking and much more. It also enables users to directly integrate different types of models, such as PyTorch, YOLO, and Tensorflow directly into the cameras on board AI-chip. However, as during the first few months of development it was unclear which camera was going to be used in the system, none of these advanced features were used. Instead, the only feature used from the camera was its depth sensing ability.

An SDK is available for the camera, enabling users to use its API to unlock its advanced features for development. The ZED uses a 3D cartesian coordinate system to specify positions and orientations., with its default coordinate system being “right-handed up” [27].

Figure 13: ZED 2 Axis Orientation, Right-Handed Up

The camera uses depth maps to store a distance value (Z) for each corresponding pixel (X,Y) in the image. The distance is expressed in metric units of the user's choice from millimetres to metres and is calculated from the back of the left eye of the camera to the object. These depth maps can be displayed by normalising the 32-bit values into 8-bit representation in monochrome, where 255 represents the closest possible depth value and 0 the most distant [28].

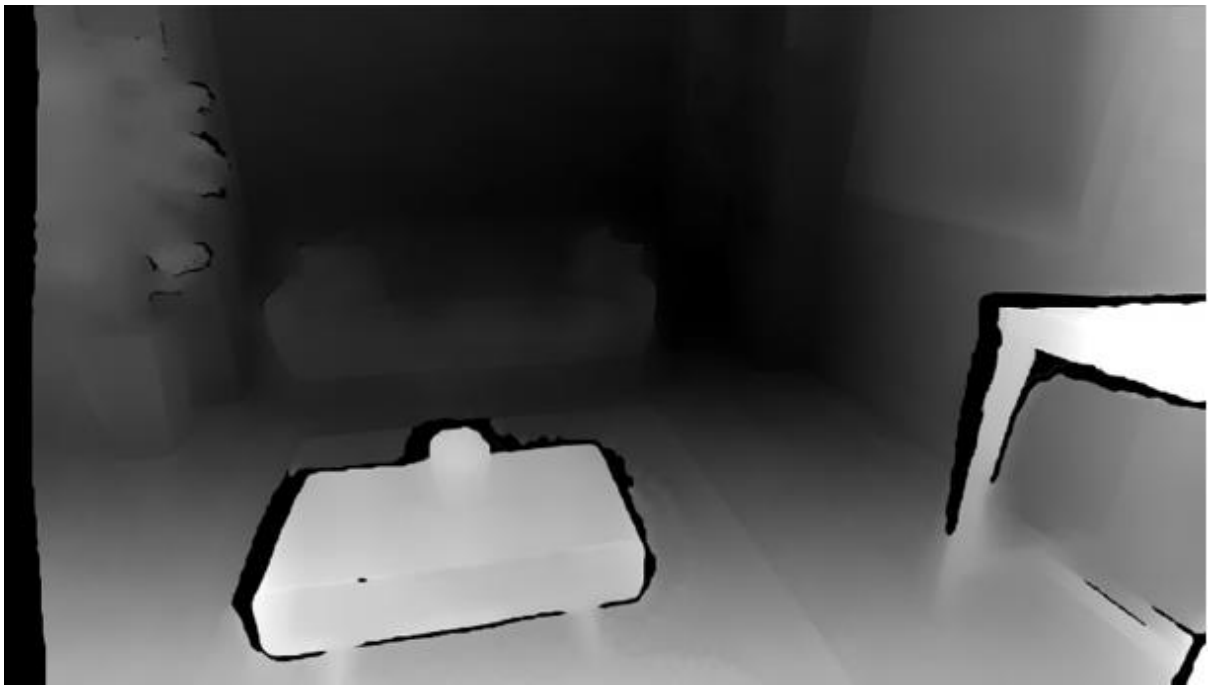

Figure 14: ZED Depth Map

3.3 WLKATA MIROBOT

The WLKATA Mirobot is a 6 degree-of-freedom robotic arm, mimicking a typical industrial robot. The 6Dof allows high manoeuvrability and accuracy to enable the arm to be navigated to a desired position in 3D space. It is a cartesian robot, allowing the user to manipulate its

position either via inputting degrees to the 6 joints, or giving it coordinates in its cartesian workspace [29].

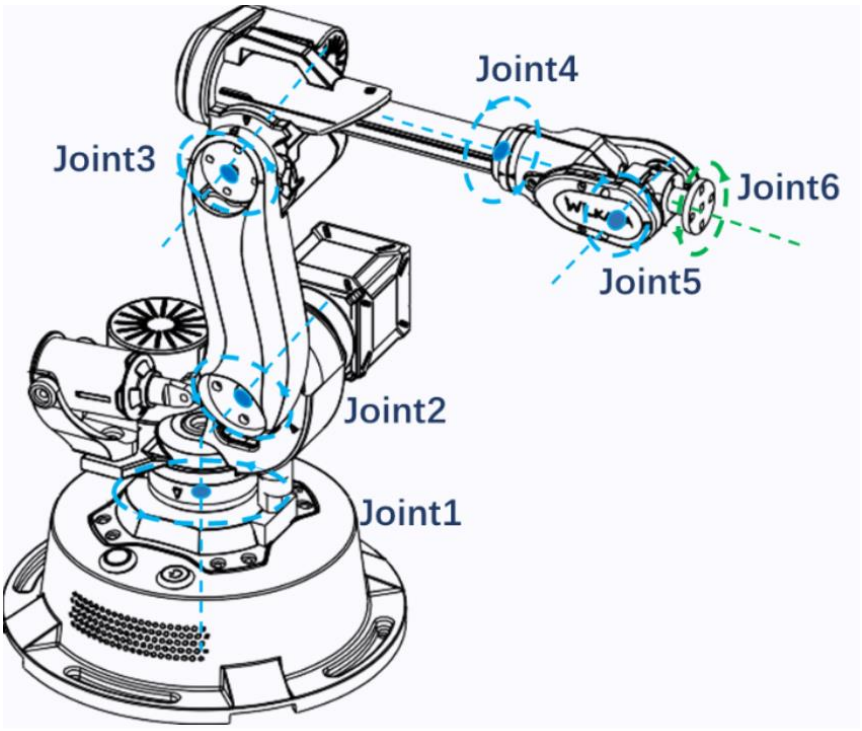

Figure 15: 6 Joint Coordinate System of WLAKTA Mirobot

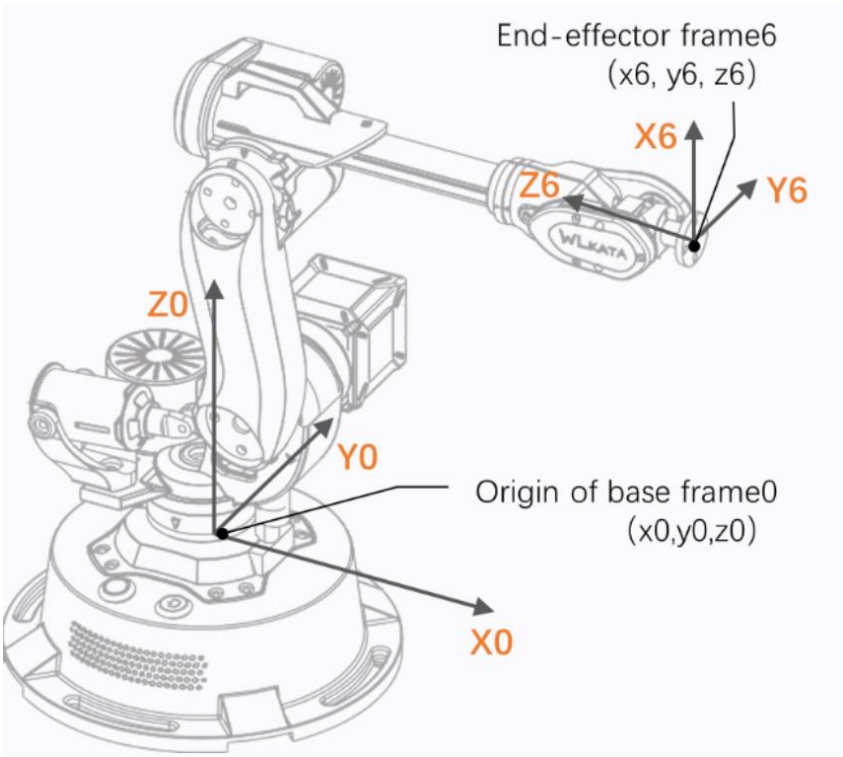

Figure 16: Cartesian Space Coordinate System of WLKATA Mirobot

The Mirobot comes with a few end-effector tools to choose from. The choices are a ‘gripper’, ‘claw’ or ‘suction cup’ attachment. The suction cup attachment was chosen, as it seemed to provide the easiest interface for the robot to pick up the objects. The suction cup enables the object to be picked up from placing the head of this tool on it almost anywhere, whereas the gripper and claw tools imply a much larger emphasis on where and how the object is picked up. The suction cup can be placed onto the object at any point and turned on to activate the suction between the tool and the object, but if using the gripper or claw it is much more important where you try grasp the object, as not all points will be equal.

There is a python API available to allow programming and manipulation of the robot via this API, enabling it to easily be added to programs and projects.

3.3.1 Inverse Kinematics

Normally, to program the movement of robotic arms, a process called *inverse kinematics* is required to map the motions of the joints to a cartesian coordinate system.

“Inverse kinematics is the use of kinematic equations to determine the motion of a robot to reach a desired position. For example, to perform automated bin picking, a robotic arm used in a manufacturing line needs precise motion from an initial position to a desired position between bins and manufacturing machines. The grasping end of a robot arm is designated as the end-effector. The robot configuration is a list of joint positions that are within the position limits of the robot model and do not violate any constraints the robot has [30].”

Inverse kinematics can determine the appropriate joint configuration which enable the end-effector to move to the target position. There are two main methods of inverse kinematic solutions: numerical and analytical. The preferred method is usually the analytical approach, as once the equations are derived computation is much faster compared to the iterative, numerical approach.

“The steps to calculating the inverse kinematics of a 6 degree-of-freedom robotic arm are:

- **Step 1:** Draw the kinematic diagram of just the first three joints, and perform inverse kinematics using the graphical approach.
- **Step 2:** Compute the forward kinematics on the first three joints to get the rotation of joint 3 relative to the global (i.e. base) coordinate frame. The outcome of this step will yield a matrix **rot_mat_0_3**, which means the rotation of frame 3 relative to frame 0 (i.e. the global coordinate frame).

- **Step 3:** Calculate the inverse of **rot_mat_0_3.**
- **Step 4:** Compute the forward kinematics on the last three joints, and extract the part that governs the rotation. This rotation matrix will be denoted as **rot_mat_3_6.**
- **Step 5:** Determine the rotation matrix from frame 0 to 6 (i.e. **rot_mat_0_6**).
- **Step 6:** Taking our desired x, y, and z coordinates as input, use the inverse kinematics equations from Step 1 to calculate the angles for the first three joints.
- **Step 7:** Given the joint angles from Step 6, use the rotation matrix to calculate the values for the last three joints of the robotic arm. [31]"

Fortunately, the WLKATA Mirobot's provided API intrinsically handles all forms of forwards and inverse kinematics, so manually computing these kinematic matrices isn't necessary.

3.3.2 Motion Settings

The WLKATA Mirobot provides different motion settings via its API, which change how the robot arm moves given a particular coordinate.

There are two main motion modes, cartesian and joint as shown in Figure 16 and Figure 15 respectively. The cartesian motion modes controls the position of the end-effector tool, and as this is the part of the Mirobot that needs to be moved to the object, only cartesian motion modes will be considered.

The main ones to consider are *linear interpolation,* which moves the arm in a linear fashion to the desired point, and *door interpolation,* which moves the arm in a 'door-shaped' rectangular motion.

Figure 17: Linear Interpolation Motion

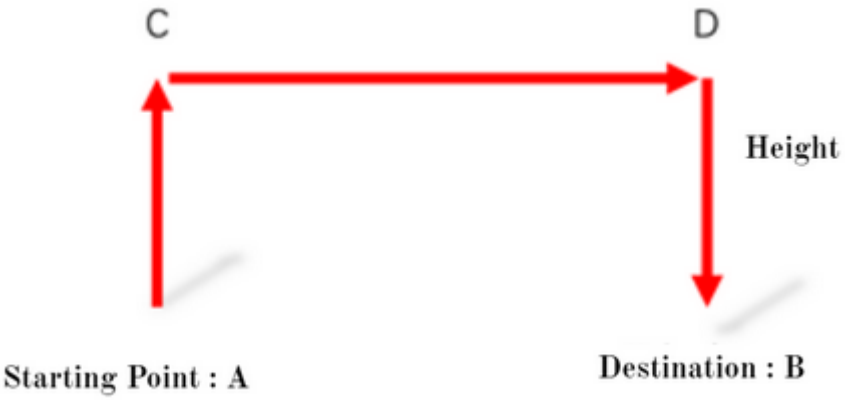

Figure 18: Door-shaped Interpolation

The linear interpolation motion setting moves the arm in such a way that its trajectory is a straight line, providing the most direct and quickest route to the object.

The door interpolation motion setting first raises the arm up, before moving it to its desired (X,Y) position, and the descending to its desired Z location. The advantage over this motion setting is that if there are multiple objects near each other, the height of an object might interfere with the route of its linear path. Using this setting would minimise the risk of an object being in trajectory of the path the arm takes, as it descends upon the object from above instead of moving there in one smooth motion.

3.4 System Setup

The system will be configured in such a way that the ZED 2 camera is above the robotic arm, with its lenses facing downwards towards the ground. It will be placed opposite the robotic arm. Every frame captured by the cameras live video feed will be fed into the YOLOv5 model, enabling it to make predictions on the type of object and location relative to the camera frame.

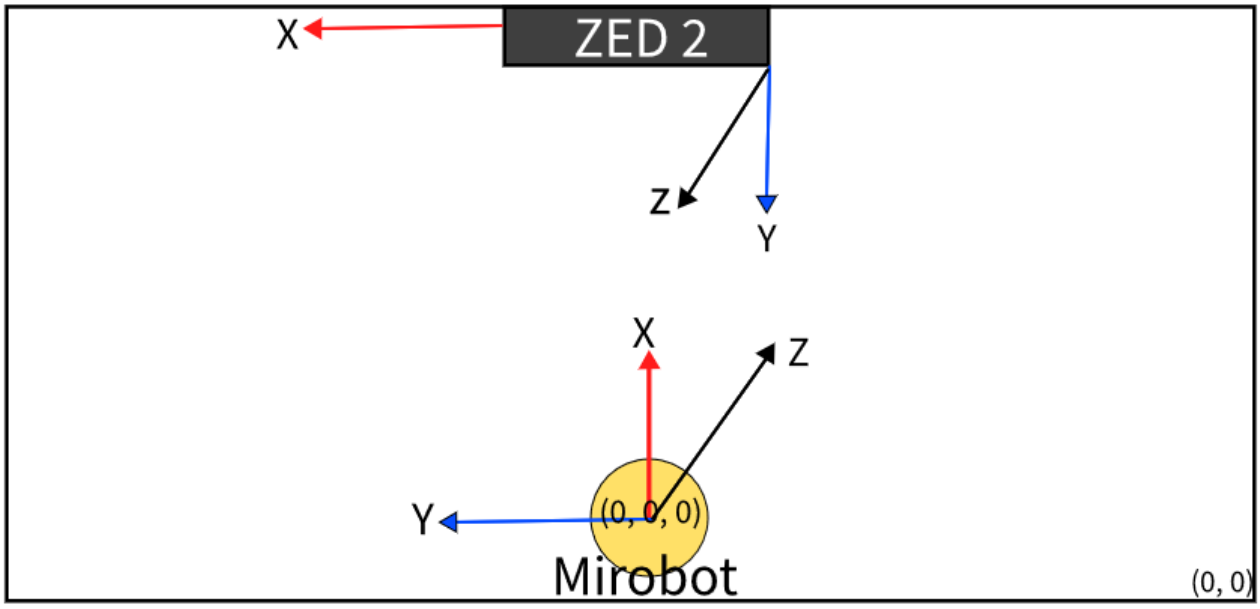

Figure 19: Mechanical System Setup

The 0 frame of the robotic arm is located at the centre of its base, while the 0 frame of the camera is located at the bottom left-hand corner of the image captured, relative to the camera. The Z axis of the Mirobot points upwards, towards the camera. The Z axis of the camera points downwards, towards the ground. Using this information, it is possible to map the robotic frame to the camera frame with the use of a rotation matrix and a displacement vector. Once this is done, the coordinates of the object relative to the camera can be transformed into coordinates relative to the robot, enabling the movement of the WLKATA Mirobot to the objects' location.

3.5 COORDINATE MAPPING

To successfully achieve the manoeuvring of the Mirobot's end-effector tool to the objects position, coordinate mapping between the 0-frame of the camera and the 0-frame of the Mirobot must be implemented.

The first stage to coordinate mapping is to describe the position of the manipulation system in a known coordinate system. As the WLKATA Mirobot can be referenced using a cartesian coordinate system, the position of the Mirobot is known. This coordinate system can be annotated as {A}. Once established, it is possible to locate any point within that coordinate universe using a *3x1* position vector $\begin{bmatrix} x \\ y \\ z \end{bmatrix}$

The next stage is to find the coordinate system of the system in which we will be mapping {A} too, in this case it is the coordinate system of the camera. As the camera also operates in a cartesian coordinate system, this system can be annotated as {B}.

To find the coordinates of system {A} relative to {B}, a homogeneous transformation matrix must be computed. This is made up of two parts:

- A displacement vector from {A} to {B}
- A rotation matrix mapping the axes of {A} relative to {B}

As shown in Figure 19, the axes of the two coordinate systems are not aligned in both orientation and location. Finding the displacement vector from {A} to {B} is the first step. This is done by physically measuring the displacement from the origin of the Mirobot's base zero position to the origin of the zero position of the camera. A measuring tape was used, to measure the distance from the X, Y, and Z axes of the Mirobot to the corresponding axes of the camera.

Rotation matrices act as a way to 'map' the axes of one coordinate system to another. This is achieved by using the 'right-hand rule' and rotating the axes of {A} a certain number of degrees to ensure the axes of both coordinate systems now match up.

The standard formula for implementing rotation matrices can be seen below:

$$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix}$$

$$R_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix}$$

$$R_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Figure 20: Rotation Matrices Formula

If rotating about more than one axis, the final rotation matrix can be found by computing the *dot product* between the matrices to yield the rotation matrix.

After this has been calculated, the displacement vector is appended to the matrix, as well as an extra row at the bottom of [0 0 0 1], resulting in the homogenous transformation matrix.

Using this matrix, the coordinates of an object relative to {B} can be found in {A} by computing the dot product of the homogeneous transformation matrix, and the unit position vector of the object relative to {B}. The result of this operation is a position vector of the objects location relative to {A}.

3.6 SOFTWARE DESIGN

Using the various APIs provided with the different software and hardware components enables development of the system as a whole possible. All components of this system have python APIs, allowing the sharing of information between the various components ensuring the ability to get the desired output from their received input.

A basic flow of the information will be as follows: ZED 2 → YOLO MODEL → MIROBOT

Where the frames captured by the ZED 2 camera will be used as an input image for the object detection model, enabling the model to predict the objects location and class.

Using the objects predicted location, the depth of the object will be retrieved from the depth map as shown in Figure 14, in most cases by using the centre of the predicted bounding box as the X,Y location to find the corresponding depth for that pixel. As the depth is in millimetres, and the Mirobot coordinates are in millimetres, these XY values are converted to millimetres and appended with the depth to create a position vector of the object relative to the camera.

Once the depth is retrieved, coordinate mapping can be completed to get the relative position of the object to the Mirobot. This position vector will then be used as the coordinates to move the Mirobot to the objects' location.

4 IMPLEMENTATION

4.1 LANGUAGES, SOFTWARE AND ENVIRONMENT

This system was implemented using Python, as this language is used and favoured heavily by data scientists and machine learning researchers across the globe. It supports a multitude of packages and libraries, helping to make implementation of these topics as simple as possible.

Python is also a very straightforward language to use, with its programming style more akin to pseudocode than the likes of other languages, such as Java or C++.

While the YOLOv5 model was trained and implemented in PyTorch, it was exported as a Tensorflow model and Tensorflow and Keras were used for the rest of the implementation. Tensorflow and Keras are "open-source machine learning libraries, with a comprehensive, flexible ecosystem of tools, libraries, and community support [32]."

Tensorflow benefits from the use of GPUs, allowing models to be trained quicker than if it were using just a CPU. To enable support for the use of a GPU, a CUDA supported NVIDIA graphics card must be present on the machine. The CUDA Framework must be installed, as well as the drivers for the graphics card.

CUDA support was also a requirement for the ZED 2 camera to be compatible with the machine, and to enable use of the cameras advanced features.

The IDE used was VSCode, while it is common practice to use Anaconda, a data science platform capable of creating environments and simplifying package management, there were issues faced with certain versions of Tensorflow compatibility, hence VSCode was used with the machines Python 3.9 used as the environment on which to install the required tools and packages.

4.2 SOFTWARE LIBRARIES

Packages and Libraries	Description
Numpy	Numpy is a mathematically library for Python, allowing the creation and manipulation of vectors and mulit-dimensional arrays.
Tensorflow	An open source machine learning library. Allows importing of models and inference to be computed via tensors.

Wlkata_Mirobot	The Python API for manipulating and moving the WLKATA Mirobot.
Pyzed.sl	The API for interacting with the ZED 2 camera and accessing its functions.
OpenCV (CV2)	A computer vision library for python, enabling the manipulation of images, as well as the display of images. Used in pair with the ZED 2 API to enable displaying of the video stream.
PyTorch	Used in the YOLOv5 Repository for implementation and training of the Object Detection Model.

4.3 GENERATING DATASETS

The datasets used for this project were a combination off two sourced on Kaggle

- Drinking Waste Classification [33]
    - Contains 4 classes – metal drinking cans, PET plastic bottles, HDPE plastic bottles and glass
- Cardboard Boxes [34]
    - Contains one class, cardboard

As these are two different datasets, the class labels of the cardboard dataset align with one of the class labels from the drinking waste classification dataset, metal. To solve this, a small python script was written to iterate through the label files of the cardboard dataset, and change the first 0 in the file, the class, to a 4.

Once this was done, the images and labels from the cardboard dataset were manually moved into the drinking waste classification dataset, forming the full dataset to be used to train the model.

Pre-processing and augmentations to all the images were done via Roboflow, as this made applying many different augmentations and pre-processing to bounding box labelled images very simple.

The following settings were used when generating the augmented dataset:

- Grayscale – apply to 10% of images
- Saturation – between +/-25%
- Exposure – between +/-25%

- Noise – up to 5% of pixels
- Bounding Box: Rotation – between +/-15°
- Bounding Box: Shear – between +/-15° horizontal/vertical
- Bounding Box: Brightness – between +/-25%
- Bounding Box: Blur – up to 10px

This turned the original dataset from 5294 images and labels to 12,704 images and labels to used. The images are split into training, validation, and testing folders at a 70-20-10 split ratio with respect to the original image set. The newly augmented images are only applicable to the training set. The final image folder sizes are:

- Training: 11,000
- Validation: 1, 059
- Test: 530

The original, non-augmented images are found in this projects repository. Training of the model will be done on both the augmented dataset, and the original dataset, to see which version of the model produces better real-world results.

4.4 FUNCTIONS AND ALGORITHMS

4.4.1 Train.py

This file is found within the YOLOv5 repo folder and is responsible for training the model. By running the command

python path/to/train.py --data path/to/data.yaml --weights pretrainedWeightsFile --img 640

The model begins training. A *.yaml* file was created for use of the recycling dataset, giving it the paths to the training, testing, and validation directories for the images and labels. The model was trained with the following settings:

- Image size = 640x640
- Epochs = 150
- Batch size = -1
- Weights = yolov5s.pt

YOLOv5 recommends using the largest batch size possible that the GPU allows, so passing in -1 as the batch size allows YOLOv5 to automatically set the best batch size for the GPU.

It should be noted that while the number of epochs is set to 150, training may be stopped earlier if optimal values are reached earlier, and subsequent training starts to produce worse results.

The results are visualised on wandb.ai, a central dashboard that keeps track of system hyperparameters, metrics, and predictions. An account must be made with this site to login and see the results and graphs produced by training model.

Issues were encountered when trying to train the model on a local machine, as the machine in use was equipped with a GTX 1060, which lacked the GPU memory to enable proper training. On multiple occasions, the training of the model was failing around epoch 30. Due to this, training was moved to Google Colab [35], a service provided by Google that enables users to host Jupyter Notebooks and use their resources. The GPU provided with Colab was a TESLA P100.

A pre-trained model was used, using the *yolov5s.pt* weights file. This file corresponds to the 'small' version of the YOLOv5 model, which is only one third as deep as the full YOLOv5 model. This was chosen as the rest of the application would be run on a local machine, using a GTX 1060. This graphics card lacked the memory resources to handle running of a larger model, which would have produced better results across the board.

After training was complete, using the YOLOv5 *export.py* file, the trained model was exported in Tensorflow format for use in the application.

4.4.2 ObjectDetection.py

This file contains the methods responsible for extracting, scaling, and parsing the predictions returned by the Object Detection model. As YOLOv5 splits the image into an *SxS* grid cell [Figure 7], the coordinates for the objects are relative to each respective grid cell, not the image itself. Processing must be done to these coordinates to map them respective to the image.

Furthermore, filtering, and non-maxima-suppression must be done to the predictions returned, as predictions are made for each grid cell. Confidence filtering is applied to remove any predictions with a confidence score of below 0.5, as well as non-maxima-suppression to

remove potentially overlapping bounding boxes using the Intersection-Over-Union method, with a threshold of 0.5.

These functions reduce the number of total outputs and only return the ones with a high probability of being an object and returning only the highest scoring bounding box per object.

4.4.3 Camera.py

This file is responsible for setting up the ZED 2 camera object, and its corresponding image matrices and depth maps. Each frame captured is returned by the *videoStream* function, which can then be passed to the *getPredictions* function. This returns the processed scores, boxes, and classes, which can further be passed as parameters to the *drawPredictions* function, returning the frame used to generate the predictions with the bounding boxes, confidence scores, and labels drawn onto the frame.

An important function is the *getCameraCoords* function, that takes in the bounding boxes and the depth map of the image. This is responsible for returning the coordinates of the object within the camera frame, so coordinate mapping can be achieved between the camera frame and the Mirobot base frame. To achieve this, however, the camera coordinates of the objects' location must be converted into metric units. In this system they are converted to millimetres for reasons mentioned above. This is done by finding out how many pixels there are per millimetre of the cameras frame. To achieve this, a measuring tape was placed across the cameras frame, and reading the value where the cameras frame ends. This is a static value that only needs to be measured once when the system is setup. With this value measured, dividing it by the number of pixels based on the axis measured gives the ratio of millimetres per pixel. In this case, the measurement was taken along the cameras X-axis, or the width of the frame. Since the resolution the camera is set to is HD1080, the resolution of the image is *1080x1920.* As the measurement was taken along the frames width, dividing the value measured by 1920 gives the ratio of millimetres per pixel. This value is later used in the function *bbox_to_mm.*

The *getCameraCoords* function calls *getDepth*, which gets the depth of the image from the centre of the bounding box in millimetres. On some occasions the depth returned of the specified location can be *nan*, due to possible issues with shadows and lighting conditions. To help combat this problem, if a *nan* value is encountered, the *getDepth* function iterates

through the area of the bounding box of the object and returns the smallest depth value found as this is most likely to be the one representative of the object. After this, *bbox_to*_mm is called, which converts the corners of the bounding box values from its pixel representation to millimetres by multiplying the bounding box coordinates with the ratio of millimetres per pixel found above. These values are then appended into an array, which is passed on to certain functions within the *Mirobot.py* file to allow for coordinate mapping.

4.4.4 BinaryPoseEstimation.py

This was a rather simple file, as the pose estimation task at hand was not complex. Unlike pose estimation for humans or animals, objects of this calibre are only in one of two poses: horizontal or vertical. This file implements a very basic pose estimation check, simply by checking if the width of the object detected is greater than its height. If so, the object is horizontal, else, its vertical.

To achieve this, the width of the object is calculated using values obtained from *getCameraCoords,* using the bounding box corners of the image. As the bounding box corners of the image are converted into millimetres, a simple subtraction is done from the *xmax* and *xmin* values, resulting in the width of the object.

The height is calculated in a similar way. As the displacement from the ground and the camera is a known, static value, this is used and the value retrieved from the depth map, corresponding to the distance of the object from the camera, are subtracted from one another. This gives us the height of the object.

Comparing these two values enables a very basic form of pose estimation, allowing the program to know whether the object is horizontal or vertical. This value can then be used to determine how to manipulate the end-effector of the Mirobot, ensuring it can grab the object in an appropriate way.

4.4.5 Mirobot.py

This file is responsible for manoeuvring the WLKATA Mirobot arm. Before any movements can be made, the axes of the robotic arm need to be unlocked, which is achieved via its *homing* function available by its API.

Appropriate frame mapping must also be completed, so it is possible to map the coordinates of the object obtained via the camera to coordinates of the robotic arm base frame. To do this, a function called *transformationMatrix* has been created. This function sets up the

homogeneous transformation matrix, which is a combination of the rotation matrix from the Mirobot base frame to the base frame of the camera, and a displacement vector of the displacement of the axes of the Mirobot base frame to the axes of the camera frame. This homogeneous transformation matrix has been declared *global* so it can be used by the rest of the functions within that file, as returning the matrix and passing it to the functions as a parameter seemed needless.

As explained in the Coordinate Mapping section, the homogeneous transformation matrix from the Mirobot base frame (annotated as '0' within this file) to the camera base frame (annotated as 'c' within this file) must be found. The system is set up as depicted in Figure 19, so using the 'right-hand rule', if the axes of the Mirobot are rotated 180° about the Y-axis, and -90° about the Z-axis, the orientation of the axes of the Mirobot and the ZED 2 will now be in the same direction. For this rotation, it was assumed that a clockwise rotation was positive, and an anti-clockwise rotation was negative. Using these angle values and the matrix formulas depicted in [Figure 20], the dot product of the rotation matrix about the Y-axis and Z-axis was calculated, yielding the final rotation matrix. This rotation matrix will remain true in all instances where the mechanical setup is the same as depicted in Figure 19.

The next step to completing the homogenous transformation matrix is to find the displacement of the axes of the robotic frame to the camera frame. This was done with the use of a measuring tape, measuring the displacement from the centre of the Mirobot base (the zero position) to the cameras zero position, located in the bottom left-hand corner relative to the image produced by the camera. This was measured in millimetres, as the cartesian coordinate system of the Mirobot is in millimetres.

The displacement vector and rotation matrix are combined, with an added extra row along the bottom with the values [0, 0, 0, 1] to create a *4 x 4* matrix. This is done so the dot product between the homogenous transformation matrix and the camera coordinate vector can be calculated.

With the homogenous transformation matrix calculated, it is now possible to map the coordinates of the object relative to the camera to coordinates relative to the robotic base frame.

Using this matrix in tandem with the values returned by *getCameraCoords*, the millimetre representation of the bounding boxes is passed as a parameter to the *moveRoboticArm* function. Within this function, the centre value of the X and Y location of the object in

millimetres is found by adding the two X points of the bounding box together and dividing them by two, giving us the mid-point X value of the bounding box. The same process is repeated with the Y values of the bounding box. These new values are combined into a vector, along with the Z depth value, and the value 1 to make this a *4x1* vector capable of matrix multiplication with the transformation matrix.

This vector is then passed as a parameter *getRobotCoords,* that returns the dot product between the homogeneous transformation matrix and the camera coordinates vector, producing the cartesian coordinates of the object relative to the robotic base frame. These cartesian coordinates can then simply be passed to a *motion setting* of the WLKATA Mirobots API, which in turn moves the arm to the specified position.

All of this processing is done automatically, the only function that has to be called is *setUpMirobot*, which returns the robotic arm object created by initialising the WLKATA Mirobot API, and *moveRoboticArm,* which takes in the arm object returned from *setUpMirobot*, the coordinates of the object which is returned from the *getCameraCoords* from the *camera.py* file, the classes associated with each object so the Mirobot can sort the objects based on their classification, and the binary pose associated with each object so the Mirobot can grab the object appropriately.

Within the *moveRoboticArm* function, two functions are called to enable to appropriate manoeuvring of the arm. The first function called, *moveArmToObject*, moves the arm to the objects position based on its pose. As a simple binary pose was all that was needed, the movement is simple. If the pose of the object equals 0, that means the object is horizontal. If this is the case, all the Mirobot has to do is perform a linear movement to the coordinates returned by the *getRobotCoords* function. If the pose equals 1, this indicates that the pose of the object is vertical. If this is the case, the roll of the end effector is set to either positive or negative 90, which rotates the end effector tool 90°. A simple check is done to figure out if the tool has to be rotated in the positive or negative direction. If the Y coordinate of the object relative to the arm is negative, then the tool must be rotated negatively, else, rotated positively.

The next function called is *moveArmToObjectBin,* which takes in the class associated with each object. The positions that the objects are sorted into based on their classification will be static, as ‘bins’ of some description will be placed near the Mirobot so it can move the object

into its respective bin. The coordinates of these bins will be pre-defined, as they are static, so the Mirobot will be able to move each object into its bin.

5 TESTING AND EVALUATION

5.1 EVALUATING YOLOV5 MODELS

During the training stage of the model, graphs are plotted for different metrics and saved via Wandb.ai. This allows users to see the progress and performance of the model during different stages of training.

Each model was trained using the settings described in section 4.4.1.

5.1.1 Model Trained on Augmented Dataset

The dataset used for this model was the augmented dataset described in section 4.3. The total training time for this model was around 18 hours.

5.1.1.1 Metrics

From training the model, graphs are outputted to show the models progress over the number of epochs. The final results of the model after training are as follows:

- Precision: 0.8877
- Mean Average Precision 0.5: 0.8783
- Mean Average Precision 0.5-0.95: 0.573
- Recall: 0.813

The graphs for each metric are shown below:

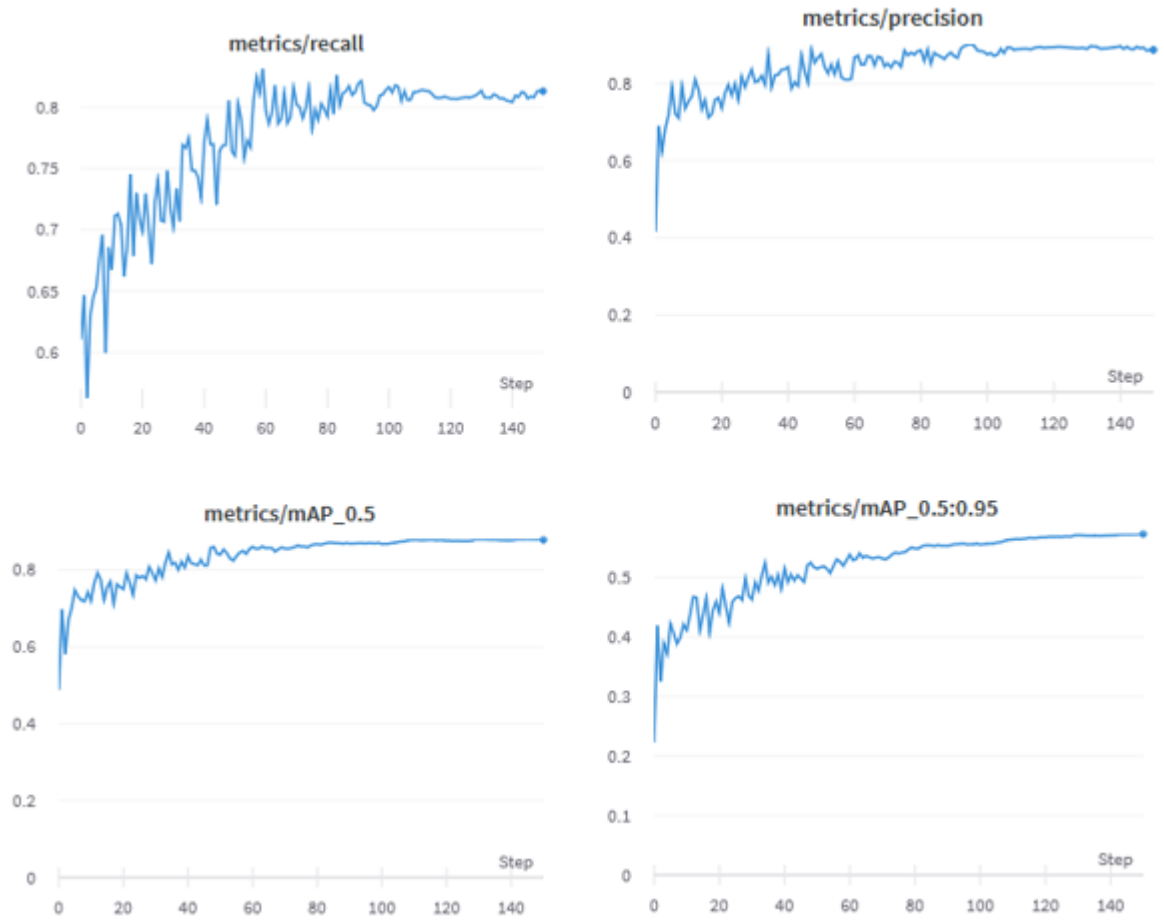

Figure 21: YOLOv5 Training Metrics on Augmented Dataset

While training the model for a larger number of epochs would have produced better results, it is clear that a plateau in improvements is reached around the 80-epoch mark. From here, the improvements made minuscule, so training for a larger number of epochs would have only achieved a negligible improvement.

5.1.1.2 Loss

The loss of the model was computed using the Binary Cross-Entropy with Logits Loss function from PyTorch. Below are the graphs obtained from making inference on the validation dataset while testing. There are different loss graphs based on what the loss is measuring in reference to. There is a loss graph for the loss computed from the classification of the objects, another graph of the loss relating to the probability an object is contained within a region of the image, and a final graph showing the loss in relation to the bounding boxes predicted compared with their ground truth

The final results of the loss computed on the validation set are as follows:

- Box Loss: 0.02458
- Object Loss: 0.01711
- Class Loss: 0.001015

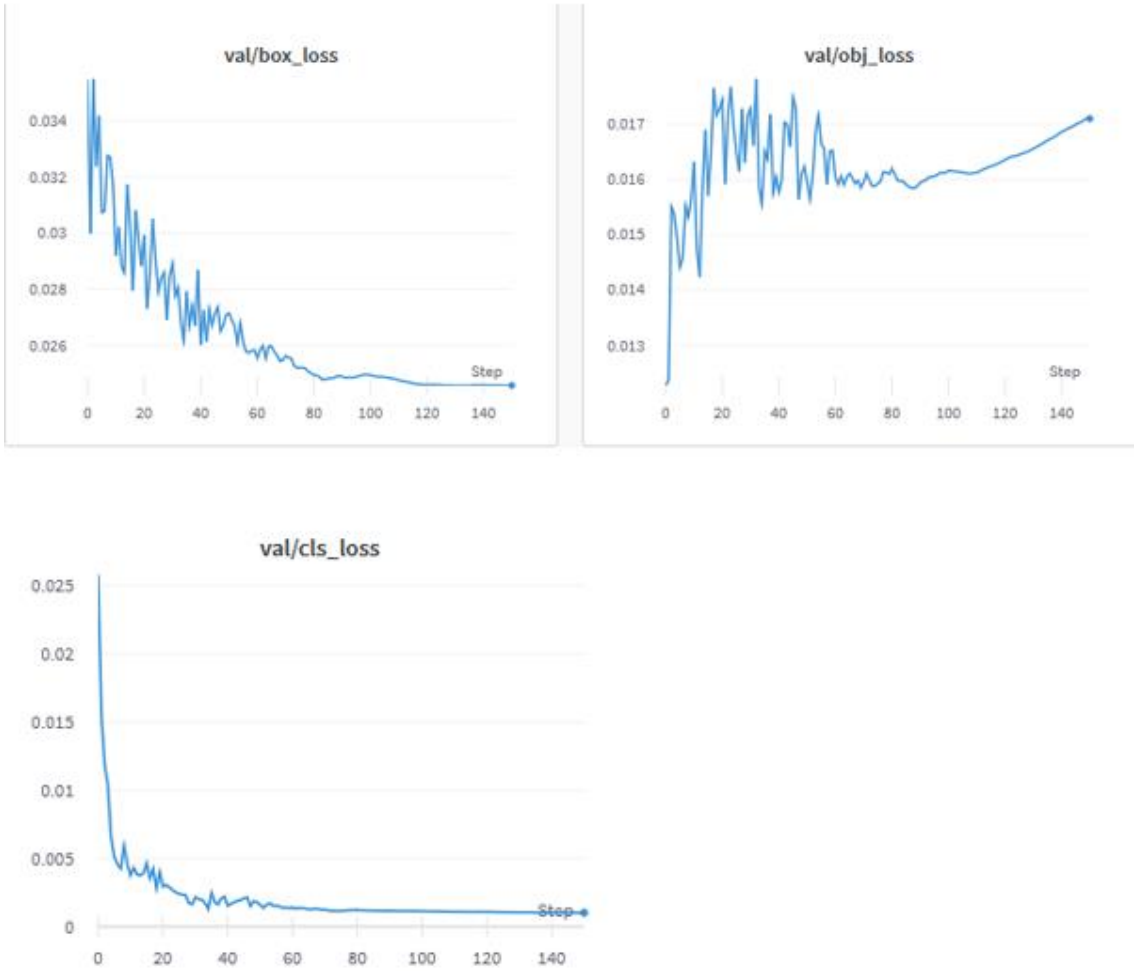

Figure 22: YOLOv5 Validation Loss on Augmented Dataset

The increase in object loss shown above could be indicative of overfitting on the training dataset.

5.1.2 Model Trained on Non-Augmented Dataset

The training set used for this model was the non-augmented dataset described in section 4.3. As this dataset was significantly smaller than the augmented dataset produced, training time for this model was only around 7 hours.

5.1.2.1 Metrics

The final values for the metrics are as follows:

- Recall: 0.9476
- Precision: 0.9729
- Mean Average Precision 0.5: 0.9666
- Mean Average Precision 0.5-0.95: 0.8187

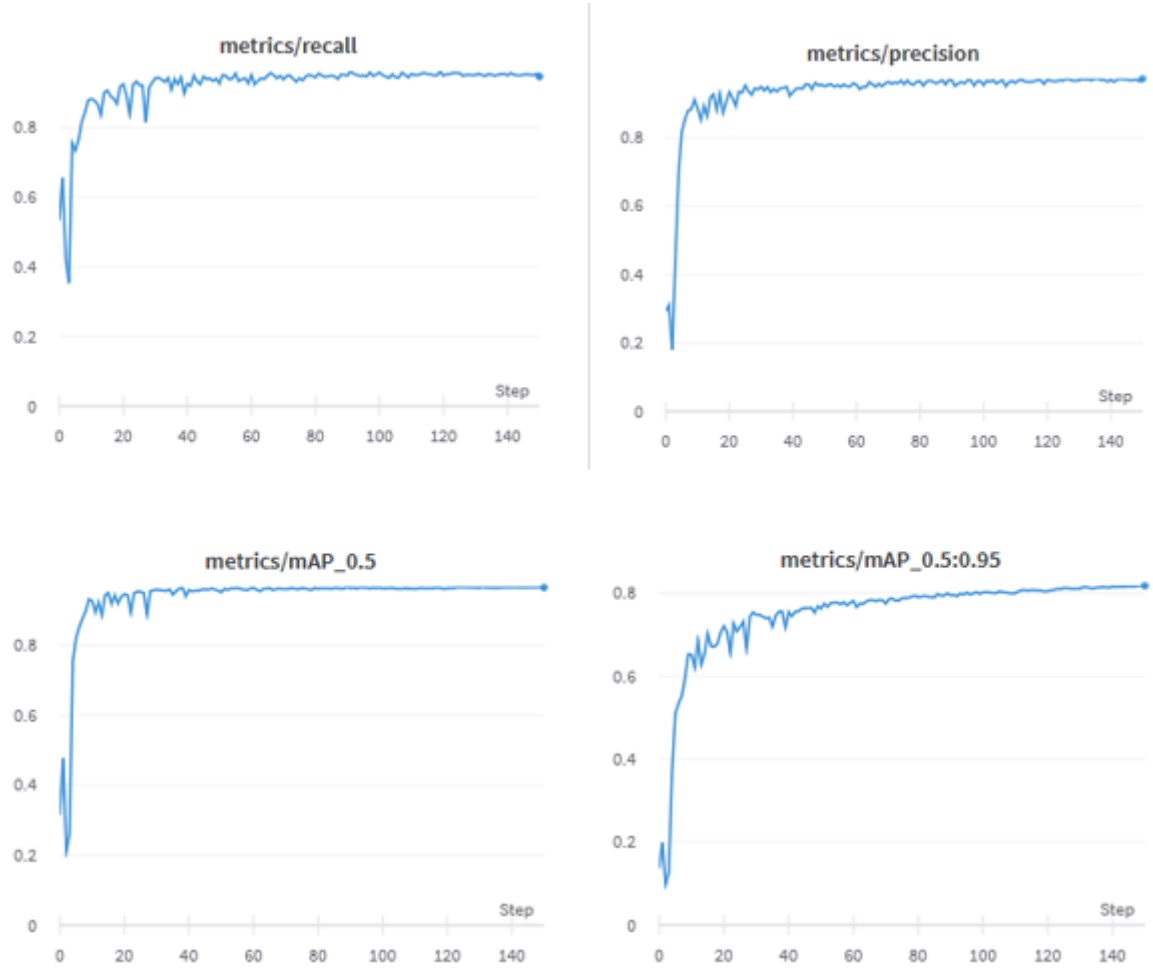

Figure 23: YOLOv5 Metrics on Raw Dataset

5.1.2.2 Loss

The final values for the validation loss are as follows:

- Box Loss: 0.01678
- Object Loss: 0.01062
- Class Loss: 0.0004499

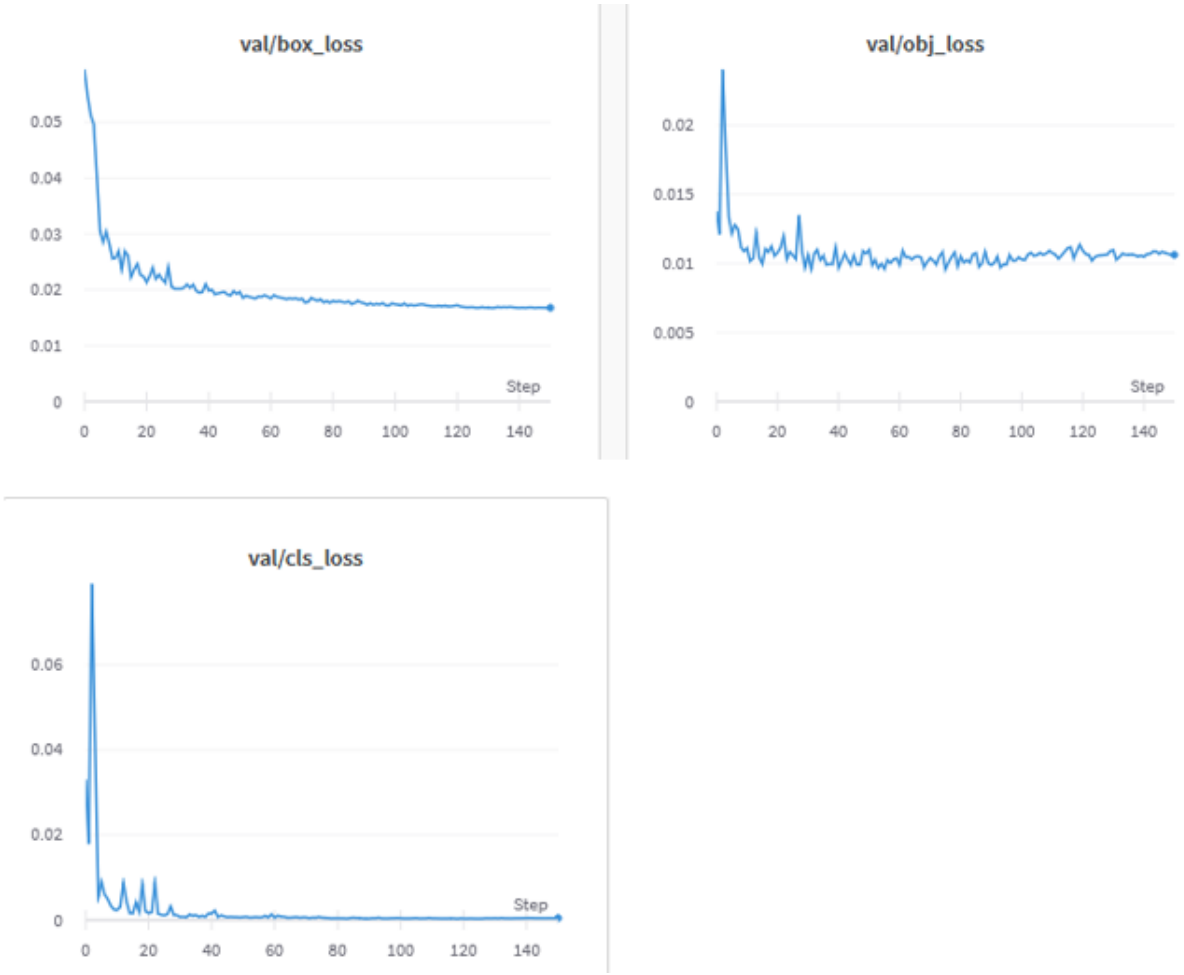

Figure 24: YOLOv5 Loss on Raw Dataset

5.1.3 Comparing Models

As shown in the graphs above, training completed on the non-augmented set of images produced significantly better results. However, this is not surprising given the large difference in size between the two sets. While this was expected, the idea was that even given worse metrics at the end of training, the model trained on the augmented dataset would perform better when deployed in the system as it was thought that augmenting the images would make the model less likely to overfit and enable better detection when deployed in the system. However, this was not at all the case. When using the model trained on the augmented dataset in the system, it would fail to detect anything within the frames captured via the ZED 2 camera.

The model trained on the non-augmented dataset, however, when deployed in the system was able to classify and localise objects within the frames with relative accuracy. While not perfect, it will sometimes misclassify objects

5.2 INFERENCE EXAMPLES

Upon training completion, the YOLOv5 repository produces some image mosaics containing the predictions made on images during the *training* stage of the model.

A python script called *testObjectDetection.py* allows the testing of model inference on a folder of images. For this, the images located within the *test* directory of the images used to train the model was used. This script will iterate through the images within the directory specified and make predictions on the classification of the object as well as its bounding box. It draws the bounding box, image classification, and confidence score onto the image to enable users to view the predictions the model has been making on each respective image. The output of these detection examples will be saved to a folder *detect/test/{run number}*, to enable viewing of all the images it has made inference on. An example from the *test* directory of the images used to train the model are shown below:

Figure 25: Model Inference Example

While the model trained has become very accurate at detecting and classifying objects within the test dataset, these inference examples inherently have a bias. These images within the test dataset are a separated portion of the images from the datasets used to train the model, leading them to be very similar to the training images. When deploying the model in the application with the use of the ZED 2 camera, it struggles more to correctly identify and localise the objects in the given frame. This is because the pictures produced from the ZED 2 camera differ very greatly from the training images used. There is a difference in resolution, lighting, and exposure. Using a multitude of datasets and combining them would help solve these issues, as well as using the ZED 2 camera to take pictures of objects and label them, adding them to the training dataset. Unfortunately, the datasets available to the general public are limited, so gathering a large set of diverse and labelled images was proven to be unachievable.

5.3 COORDINATE MAPPING

A large part of the task was to successfully map the coordinates from the camera to the robotic arm base frame. Trial and error had to be done for the calibration leading up to this part, such as ensuring there was the correct amount of 'pixels per millimetre' from physically measuring the width of the camera frame with a measuring tape. This was easy to test, as all that needed to be done was multiply the central X-axis pixel location of the object by the ratio of pixels per millimetre and checking if it lined up on the measuring tape. Once confident this value is correct, the next stage of coordinate mapping can be tested.

The WLKATA Mirobot came with a configuration mat, which is a mat containing cartesian grid coordinates in centimetres and degree lines, coming out from a circle that represents the base of the arm. Placing the arm in this circle, it is clear what cartesian coordinates the arm needs to move too based on the object. Using this mat, it also made it much easier to measure the displacement vector from the axes of the Mirobot to the axes of the camera. The zero position of the camera frame is the bottom left-hand corner of the image produced on screen, using this as a guide it is possible to mark the zero position in the real world. For marking this position, a pen was used. Once this position was marked, a measuring tape was used to measure the displacement from the zero frame of the robot (the circle representing the base in the Mirobot mat) to the zero frame of the camera just marked. This displacement vector was used in combination with the rotation matrix associated with mapping the axes from the Mirobot to the camera to create the homogenous transformation matrix.

Testing for this was done by placing an object on a known grid position on the mat, and then calculating the dot product of the position vector of the object in millimetres relative to the camera and the homogenous transformation matrix, and the resulting output should be the position vector in millimetres of the object relative to the Mirobot base frame. This should match up where the object is on the cartesian grid map provided. When the value of the object in relation to the Mirobot is correct, the coordinate mapping has been successful.

5.4 MIROBOT

Testing of the Mirobot was very straightforward. There is a WLAKTA Studio application that is available to download. This has a very straightforward UI that can be used to manipulate the Mirobot in many ways, such as giving it cartesian coordinates in coordinate mode or angles for each separate joint. There is also a basic Python script section enabling users to use its API and test how the Mirobot moves with different functions. As this task manipulates the Mirobot in coordinate mode, all the testing was done using its coordinate mode feature or different types of cartesian motion settings. For the most part, simple testing was done with this by entering the coordinates calculated from the coordinate mapping, and then visually checking if the Mirobot was moving to the desired position. Doing this also confirmed that the cartesian coordinates used to manipulate the Mirobot are indeed millimetres in 3D space, based on the location of the end-effector tool. The motion setting used was *linear interpolation*, as using *door interpolation* would sometimes cause issues with certain soft limits of the axes being hit, as the axes were not moving all at once in a smooth motion.

Unfortunately, however, taking into consideration of the pose of the object caused issues that resulted in unsuccessful grasping of the object. As the end effector tool of choice was the suction cup, it lacked enough power to grasp the object if the object was standing vertically. This happens as when attempting to grasp the object from the side, rather than top-down like it would with a horizontal object, the suction cup lacked the power to achieve the necessary contact needed with the object. Instead, the Mirobot simply ended up pushing the object away as it attempted to suction onto it. This is caused by the fact that most of the objects it is attempting to grasp are very light, i.e., empty cans. This would most likely not be an issue with heavier objects as there would be enough weight to prevent the object from being pushed by the arm and allow the end effector tool to make a strong connection with the object.

Due to this, in the final application, the pose estimation function was left unused.

5.5 CONCLUSION

This project has been successful and can act as a base in which to improve upon for further use. The WLKATA Mirobot and ZED 2 camera are relatively low-cost compared to industrial robotic arms and sorting machines, paired with the relative success of the task at hand proves this setup could be implemented and MRCs or other recycling sorting facilities where the implementation of much larger, industrial solutions is not possible. However, to ensure this is feasible for use in real-world applications, object tracking would have to be implemented as its unlikely the recyclables that are being sorted would be stationary. Furthermore, the range of motion of these Mirobots' is not extensive, so multiple would have to be purchased to work together to enable high speed, accurate sorting.

Inference results produced by the model with the ZED 2 camera were not always the best. To improve upon this, training a larger version of the model on a larger dataset would help improve the predictions made by the model. However, given the small size of the dataset used and the small YOLOv5 model used, the results when deploying the model into the application were surprisingly good.

The largest shock was the inability of the model trained on an augmented dataset to infer any predictions in the actual application. This seems to suggest that the YOLOv5 model seems to perform best when trained on a dataset that mimics the objects more realistically, instead of applying multiple augmentations and processing steps to the images. To improve the usability of the model in real-world scenarios where there would be a multitude of tightly packed objects near each other, larger and more diverse datasets would have to be used to retrain the model with.

One of the largest datasets relative to the task of recycling object detection is called WasteNet [36], which contains over 3 million images sourced and labelled from Material Recovery Centres around the globe. Unfortunately access to their image dataset is restricted, and while they can provide access to people for academic and research purposes, during the development of this application access was unable to be obtained. Retraining the model with a large dataset such as this would enable the system to perform much better when deployed.

6 REFERENCES

[1] S. Lam-McGonnell, “garbage-classification,” [Online]. Available: https://gitlab2.eeecs.qub.ac.uk/40227918/garbage-classifcation.

[2] S. Kaza, L. Yao, P. Bhada-Tat and F. Van Woerden, “What a Waste 2.0: A Global Snapshot of Solid Waste Management to 2050,” World Bank, 2018.

[3] “What happens to my recycling?,” Waste and Resources Action Programme, [Online]. Available: https://www.recyclenow.com/how-to-recycle/what-happens-to-my-recycling.

[4] T. Graham, J. Tessler, P. Orris, J. Shimek, M. Wilson and H. Witt, “Sustainable And Safe Recycling:,” 2015.

[5] Britannica, “Industrial Robotics,” [Online]. Available: https://www.britannica.com/technology/automation/Development-of-robotics . [Accessed 29 04 2022].

[6] Matt, “Impact of Robotics in Manufacturing,” Mantec, 28 April 2020. [Online]. Available: https://mantec.org/robotics-manufacturing/ . [Accessed 29 April 2022].

[7] M. Fairchild, “Recycling Robots: How They Work & Are They Worth the Investment?,” 18 January 2022. [Online]. Available: https://www.howtorobot.com/expert-insight/recycling-robots . [Accessed 29 April 2022].

[8] IBM, “Deep Learning,” 1 May 2020. [Online]. Available: https://www.ibm.com/cloud/learn/deep-learning.

[9] G. Boesch, “Object Detection in 2022: The Definitive Guide,” Viso Suite, 2022. [Online]. Available: https://viso.ai/deep-learning/object-detection/.

[10] K. He, X. Zhang, S. Ren and J. Sun, “Deep Residual Learning for Image Recognition,” Cornell University, 2015.

[11] T.-Y. Lin, P. Dollar, R. Girshick, K. He, B. Hariharan and S. Belongie, “Feature Pyramid Networks for Object Detection,” Facebook AI Research , 2017.

[12] J. Hui, “Understanding Feature Pyramid Networks for object detection (FPN),” 27 March 2018. [Online]. [Accessed 15 April 2022].

[13] ArcGIS Developer, “How RetinaNet works?,” [Online]. Available: https://developers.arcgis.com/python/guide/how-retinanet-works/. [Accessed 19 April 2022].

[14] T.-Y. Lin, P. Goyal, R. Girshick, K. He and P. Dollár, “Focal Loss for Dense Object Detection,” arXiv:1708.02002, 2017.

[15] W. Liu, D. Anguelov, D. Erhan, C. Szeged, S. R. C.-Y. Fu and A. C. Berg, “SSD: Single Shot MultiBox Detector,” arXiv:1512.02325, 2015.

[16] E. Forson, “Understanding SSD MultiBox — Real-Time Object Detection In Deep Learning,” Towards Data Science, 18 November 2017. [Online]. Available: https://towardsdatascience.com/understanding-ssd-multibox-real-time-object-detection-in-deep-learning-495ef744fab. [Accessed 19 April 2022].

[17] C. Szegedy, S. Reed, D. Erhan, D. Anguelov and S. Ioffe, “Scalable, High-Quality Object Detection,” arXiv:1412.1441, 2014.

[18] J. Redmon, S. Divvala, R. Girshick and A. Farhadi, “You Only Look Once: Unified, Real-Time Object Detection,” arXiv:1506.02640, 2016.

[19] L. Tan, T. Huangfu, L. Wu and W. Chen, “Comparison of RetinaNet, SSD, and YOLO v3 for real-time pill identification,” BMC Medical Informatics and Decision Making, 2021.

[20] A. Rizzoli, “Mean Average Precision (mAP) Explained: Everything You Need to Know,” 15 March 2022. [Online]. Available: https://www.v7labs.com/blog/mean-average-precision#what-is-map. [Accessed 28 April 2022].

[21] Roboflow, [Online]. Available: https://roboflow.com/.

[22] Ultralytics, “YOLOv5,” [Online]. Available: https://github.com/ultralytics/yolov5.

[23] ClearView Imaging, “Stereo Vision for 3D Machine Vision Applications,” [Online]. Available: https://www.clearview-imaging.com/en/blog/stereo-vision-for-3d-machine-vision-applications. [Accessed 29 April 2022].

[24] R. Xu, H. Lin, K. Lu, L. Cao and Y. Liu, “A Forest Fire Detection System Based on Ensemble Learning,” ResearchGate, 2021.

[25] Darknet, “Darknet: Open Source Neural Networks in C,” [Online]. Available: https://pjreddie.com/darknet/.

[26] C.-Y. Wang, H.-Y. M. Liao, I.-H. Yeh, Y.-H. Wu, P.-Y. Chen and J.-W. Hsieh, “CSPNet: A New Backbone that can Enhance Learning Capability of CNN,” arXiv:1911.11929, 2019.

[27] SteroLabs, “Coordinate Frames,” 2021. [Online]. Available: https://www.stereolabs.com/docs/positional-tracking/coordinate-frames/.

[28] SteroLabs, “Depth Sensing Overview,” [Online]. Available: https://www.stereolabs.com/docs/depth-sensing/.

[29] WLKATA, “3.5 Working principle and specification,” 2021. [Online]. Available: https://document.wlkata.com/?doc=/wlkata-mirobot-user-manual/35-working-principle-and-specification/.

[30] MathWorks, “Inverse kinematics (IK) algorithm design with MATLAB and Simulink,” [Online]. Available: https://www.howtorobot.com/expert-insight/recycling-robots . [Accessed 29 April 2022].

[31] A. Sears-Collins, “The Ultimate Guide to Inverse Kinematics for 6DOF Robot Arms,” 23 October 2020. [Online]. Available: https://automaticaddison.com/the-ultimate-guide-to-inverse-kinematics-for-6dof-robot-arms/. [Accessed 29 April 2022].

[32] TensorFlow, [Online]. Available: https://www.tensorflow.org/. [Accessed 20 April 2022].

[33] A. Serezhkin, “Drinking Waste Classification,” Kaggle, [Online]. Available: https://www.kaggle.com/datasets/arkadiyhacks/drinking-waste-classification .

[34] S. Kulkarni, “Cardboard box images with annotations for YOLOv5,” Kaggle, [Online]. Available: https://www.kaggle.com/datasets/sakshi12345/cardboard-box-images-with-annotations-for-yolov5 .

[35] Google, “Frequently Asked Questions,” [Online]. Available: https://research.google.com/colaboratory/faq.html. [Accessed 30 04 2022].

[36] RecycleEye, [Online]. Available: https://recycleye.com/wastenet/. [Accessed 30 04 2022].