Imperial College London

Department of Aeronautics
Group Design Project

Autonomous Drone Delivery

Project RAST

[Computer Vision Team (ADD02)]

Supervisors: Ferri Aliabadi, Thulasi Mylvaganam
Department: Department of Aeronautics
Course: H401 MEng Aeronautical Engineering
Student: Iaroslav Kazakov
CID: 01334502

Date: August 2, 2020

Abstract

This paper describes ideas behind the Computer Vision (CV) interpretation of objects seen by a drone's camera. For the selected mission profile of the drone, it was crucial to demonstrate how both innovative and more conservative techniques can be used to detect humans and windows in living premises. The algorithm detecting humans entails neural networks and machine learning, whereas the method capturing the window frames and holes included coding in Python 3 with an accent on OpenCV module. The boolean results of these methods were then used by the authorised teams to create flight trajectories and execute specific commands. The OpenCV method, as it is called in this paper, has multiple drawbacks and inaccuracies but still works well in the environment where specific stability criteria of the drone can be satisfied. It is worth mentioning that most of the detection techniques in the modern industry explicitly use neural networks and computationally demanding algorithms which can be unfeasible to run with the capacity restricted hardware. However, the idea behind this paper is also to demonstrate engineering students and drone enthusiasts that even trivial linear 'if' statements can do the job of detecting simple geometries and output features required for pathfinding algorithms.

A Word From Author

Before moving into more technical details, it is essential to mention that the global situation of the year 2020 has impacted the educational process and the key takeaways of this project. One of the critical challenges that the flight control team and the computer vision sub-team encountered was that the software had to be run in the virtual environment Gazebo. The Simulation Team put a lot of effort into ensuring the code integration between all the teams and all the hardware could occur. However, given the time frame and unusual circumstances, most of the CV codes were unfeasible to test in the Gazebo environment. As a result, the majority (not all) of the CV codes were tested in the real world and yielded remarkable results. Hence, the quality and the relevancy of the methods did not suffer. This being said, it could be that the CV algorithms would have improved in terms of CPU processing and contours per frame if the CV team had a more natural working environment. There are a lot more 'ifs' and 'what could have been done' to mention, but the reader is expected to be competent in this topic, so it is needless to say what could have been improved. There is nobody to blame, and as a participant of this project, I feel incredibly proud of achieving high-quality results in such unusual circumstances.

Acknowledgements

I feel obligated to express my acknowledgement to colleagues as well as the scientific advisors who had been of tremendous help throughout this project. In current realms, this project would not have been achievable without the perseverance and scrutiny of all the stakeholders. Specifically, I would like to thank Vlad Marascu, Geoffrey Sheir, Vishnu Nair, Bazil Saiq, Stavros Sikkis, Peijie Wu, Zhi Foong, Elizabeth Y Oon, Ethan Hy, Patryk Kulik, Dr Thulasi Mylvaganam, Dr Vito Tagarielli and Dr Basaran Kocer. I would also like to express my gratitude to Dr Mirko Kovac, who created this unique project and gave us a lot of freedom in terms of making pivoting decisions. This freedom enabled the teams to think from a perspective of a customer buying this drone, hence adjusting the design decisions accordingly. This project was the most hands-on application of all the knowledge myself as a student collected throughout these years, and I am grateful for the opportunity to participate in it.

Contents

A Word From The Author i

Acknowledgements i

1 Introduction 1
1.1 Preamble to the world of Computer Vision . . . 1
1.2 Main Tasks . . . 2

2 Literature Review 2
2.1 Humans . . . 2
2.1.1 Histogram Of Gradients . . . 2
2.1.2 OpenCV . . . 3
2.1.3 Generalised Haar Wavelets . . . 4
2.1.4 Human Detection With Deep Learning . . . 4
2.1.5 Summary Of The Methods . . . 5
2.1.6 Choosing Neural Network . . . 5
2.2 Windows And Holes Recognition . . . 6
2.2.1 OpenCV and Neural Network Comparison . . . 6

3 Code Implementation And Results 7
3.1 Human Real-time Detection . . . 7
3.2 Human Recognition Results . . . 7
3.3 Windows Detection . . . 8
3.3.1 YOLOV3 Approach . . . 8
3.3.2 OpenCV Approach . . . 9
3.3.3 Area Calibration And Common Reference Method . . . 9
3.3.4 Code And Results . . . 10

4 Conclusion And Potential Improvements 12

References 13

Appendix 14

A Appendix 14

B Appendix 15

C Appendix 18

D Appendix 20

List of Figures

1 Autopilot implemented by Tesla . . . 1
2 Comparison between RGB (left) and HSV (right) . . . 1
3 Computer Vision tasks . . . 2
4 HOG feature extraction [1] . . . 3
5 Improved Data Set [1] . . . 3
6 Principle behind Haar Wavelets [2] . . . 4
7 Example of where the Haar method would not work [2] . . . 4
8 Visual Representation Of The Deep Learning . . . 5
9 Defining Classes In Real Time . . . 6
10 Defining Classes From Images [3] . . . 6
11 Recognition Of A Human (Myself) . . . 8
12 Multiple Humans Can Also Be Classified . . . 8
13 Recognition Of A Simulated Human . . . 8
14 Recognition Of A Simulated Human . . . 8
15 Windows Detection . . . 9
16 Windows Detection From A Different Angle . . . 9
17 Window From 3 Meters Away (63cm x 60cm) . . . 9
18 Window From 2 Meters Away (63cm x 60cm) . . . 9
19 One out of 10 Calibrating Pictures . . . 10
20 Test Picture 4x4 cm . . . 10
21 Test Picture 4x4 cm . . . 10
22 Percentage Error Vs Calibrating Images . . . 11
23 Percentage Error Vs Number Of 'Mission' Images . . . 11
24 Window Type 1 (63cm x 60cm) . . . 11
25 Window Type 2 (48.8cm x 45cm) . . . 11
26 RPN Principle Of Working . . . 14
27 RPN Architecture . . . 14
28 First Part Of The Code . . . 15
29 Second Part Of The Code . . . 16
30 Third Part Of The Code . . . 17
31 First Part Of The Window Detection Code Live . . . 18
32 Second Part Of The Window Detection Code Live . . . 19
33 First Part Of The Window Detection Code Image . . . 20
34 Second Part Of The Window Detection Code Image . . . 21

List of Tables

1 Methods summary 5
2 Neural Network Summary 6

1 Introduction

1.1 Preamble to the world of Computer Vision

In the world where visual technologies such as car autopilots, face ID recognisers and disabilities robot become vastly necessary, an understanding of computer vision algorithms plays a detrimental role in many aspects of our routine life. Various literature and scientific papers are being published day by day, and as a result, the CV world becomes more niche and less accessible for a wider audience. Nonetheless, the machine learning complexity of worldwide projects, such as OpenAI and Neuralink [4], develop at a rapid pace and lead humanity towards the digital and more predictable future.

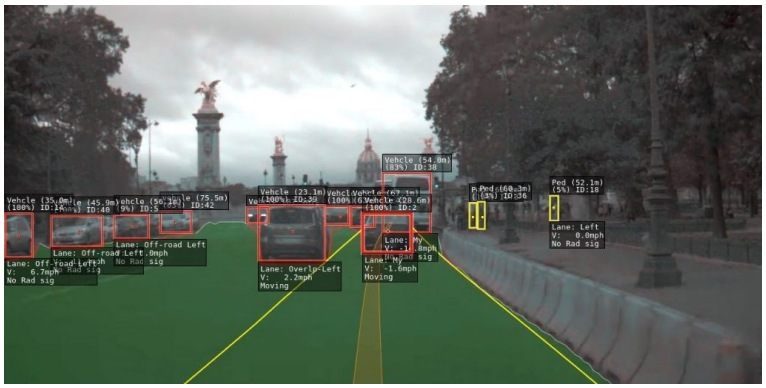


Figure 1: Autopilot implemented by Tesla

Due to this complexity, it was utterly crucial to delegate dealing with the visual perception of the environment to a particular team. In the modern world computer vision is tightly linked with the neural networks and machine learning algorithms. The ability of the hardware to 'learn' and make judgements based on its experience is undoubtedly a reliable tool. Neural networks with either pre-trained or untrained models can be used to detect humans, and that is how pedestrians get recognised by autopiloting Tesla cars, for instance [5]. Machine learning and neural object classifiers are the only reliable tools capable of recognising moving geometries/shapes; other methods tend to fall under a great many assumptions and 'ifs' that engineers cannot afford. However, it can be seen below that simple techniques can also work reasonably well when it comes to simple geometries.

To start off briefly, it is vital to understand the underlying theory behind the CV logic described below. A computer can interpret images in terms of pixels, which in essence get converted into ones and zeros. A digital colour image pixel is a number representing an RGB data value (Red, Green, Blue). These are numerical RGB components that represent the colour of the pixel area.

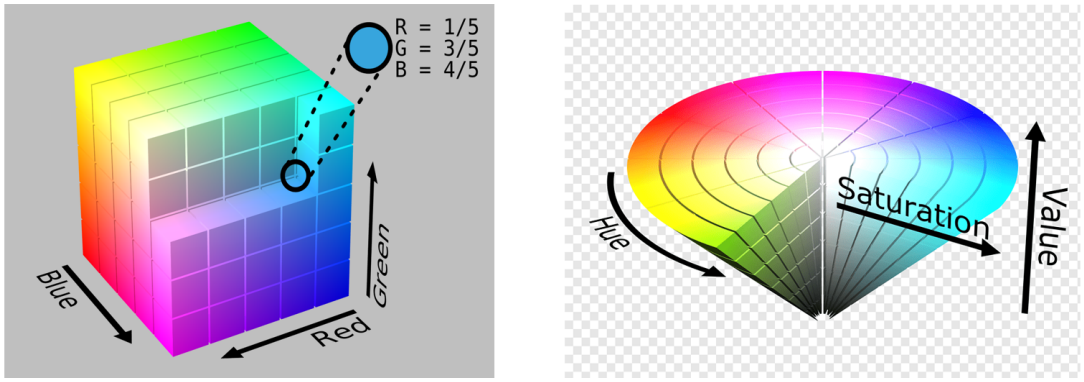


Figure 2: Comparison between RGB (left) and HSV (right)

Each colour component is an 8-bit number for each pixel; hence each pixel has 24-bit colour. For example, a pixel that stores (255, 0, 0) array has a red colour etc. This relatively high variability of colour shades allows for the implementation of conservative detection mechanisms (e.g. Histogram

of Gradients (2.1.1)). There are indeed other colour modules that can include: CIE, YUV, HSL, CMYK (2). For instance, HSL/HSV is a more natural way of processing colours in terms of their hue, saturation and value. Different engineering industries exploit different formats, but broadly RGB is the most common colour space used by most computer vision engineers.

1.2 Main Tasks

Initially computer Vision team took care of three key major tasks (3): human detection, window detection, hole detection.

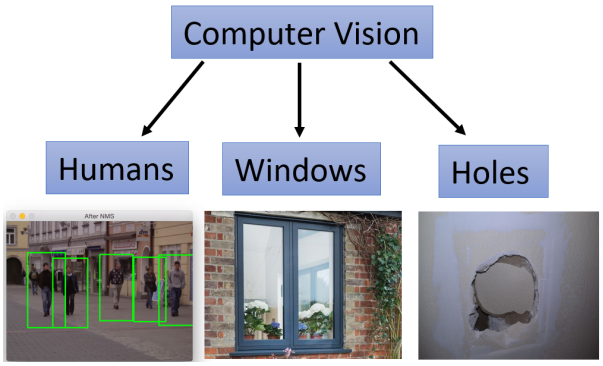


Figure 3: Computer Vision tasks

Then the holes were removed from the simulation due to computational heaviness of the initially created environment. These objects are critical to be recognised during the mission operation. Once windows are identified, specific data is sent to the pathfinding team such that the trajectory can be calculated and the drones can get inside the house. Once inside the house, the humans could be recognised using neural network algorithms and pre-trained data. After humans get detected and classified, the necessary payload can be dropped, and the signal can be transmitted to the rescue team.

2 Literature Review

2.1 Humans

2.1.1 Histogram Of Gradients

During the first week of the project, it was critical to identify the tools available for computer vision technologies in the context of humans. Now, what signifies humans is that they move, and as a result, methods that can be adopted for static objects recognition, such as window frames, furniture etc. would fail under dynamical conditions. The starting point of human recognition became the HOG method.

Histogram of Oriented Gradients (HOG) adopts linear support vector machine (SVM) based methods. SVMs are the supervised learning models that can be employed to analyse data used for classification. The understanding of the underlying theory became evident by reading [1]. The development of HOG in the early '90s and a relatively simple algorithm (compared to those of neural network origins) grasped the CV team's attention. The idea there was to apply a gamma mask to the colours of the image, calculate colour change per pixel (a.k.a colour gradients). Then divide the image into spatial regions or cells, whereby for each cell, a histogram of gradients can be estimated. To mitigate blurring, illumination inconsistencies etc. a contrast of the image can be normalised. The process can be visualised as follows:

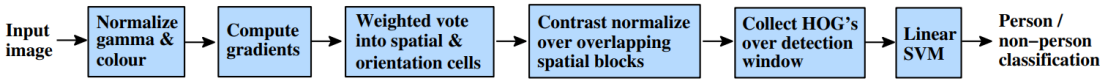


Figure 4: HOG feature extraction [1]

A user then can output a boolean classifier of whether an object is a person or not. The issue with HOG is many assumptions need to be made before the implementation. All humans are different, and there is a wide range of poses humans can take. HOG assumes that humans are in the upright position most of the time, there is a study from MIT students performed on a pedestrian data set, showing that HOG fails to detect humans in a pose other than the upright one [6]. As per [1], the data set used to validate a standard HOG method was taken from [6], containing 200 images of pedestrians in city scenes, limited by the only front and back views. Even with the improvements described below and using a more challenging data set of 1805 images [7], the limitation of this method can be noticed:

Figure 5: Improved Data Set [1]

It could be further noticed that the first four images have something in common - distinctive colour gradients between humans and the environment. That is why, on average HOG would not work adequately for the cases when, e.g. colourful human clothes match the interior of the living hood. Neither it would work for the cases when two or three people stand close to each other; the algorithm will fail to output the right boolean variable. HOG method could be improved by local normalisation of the gradients and through the extensive usage of large data sets. However, if large training data sets are ought to be used then why not using neural networks and modern classification tools? Even though this would be possible, the major underlying problem is that the training classifier is SVM, which cannot perform well on skewed and imbalanced data sets. SVM fails to perform well in circumstances where people physically interact with each other, and as a result, different forms of occlusion can show up [8] (could be the case during flooding). Lastly, the kernels of the SVM based algorithms can be challenging to tune. The kernel tuning used to be a research area of multiple universities and given the time constraint for this project this would not be reasonable [9]. As the project managers (PMs) advised, this option had to be withdrawn at an early stage due to its limitations and the unsuitability for the defined mission profile (people cannot be assumed to stand upright when flooding occurs).

2.1.2 OpenCV

OpenCV is a Python library of programming functions designated for real-time computer vision tasks. This library has profound interest from CV communities across the globe and is constantly gets updated and optimised for the power restricted hardware [10]. However, the critical issue with OpenCV in the context of humans recognition is the linearity and incapability of classifier training. The library on its own has multiple functions that can output contours, bounded boxes as well as draw all kinds of geometries etc. The functions can output certain classes based on the geometrical properties of an object, such as the number of points, thickness of edges, area of the closed contour etc. The issue with this tool was that at the beginning of the project, authorised teams were not sure about the human's appearance in the simulated environment. So the computer vision team could not take the risk of creating a code with linear 'if' statements for an 'average' person that could end up not working in the simulation. Lastly, describing humans in terms of basic geometries, amount

of contours, contour thickness, edges, points etc. could be, say the least, challenging. This method, however, was deemed to be applicable for the static objects which have defined properties, such as windows.

2.1.3 Generalised Haar Wavelets

The idea this method follows is first to output the constituent component and then combine the elements to classify an object. In the context of human detection, the components, for example, could be head, legs, left arm and right arm [2]. Based on the constituents, the 'person' or 'nonperson' can be outputted. The first issue that is linked to the selected mission profile is that when humans' houses get flooded there is a possibility of obscuration limbs. Since the CV team was ambitious about making the code general and equally applicable for different scenarios, this became a vital issue.

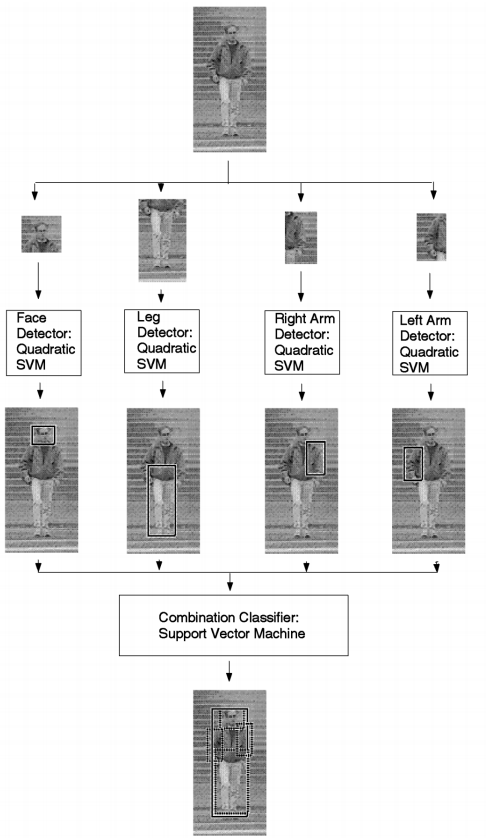


Figure 6: Principle behind Haar Wavelets [2]

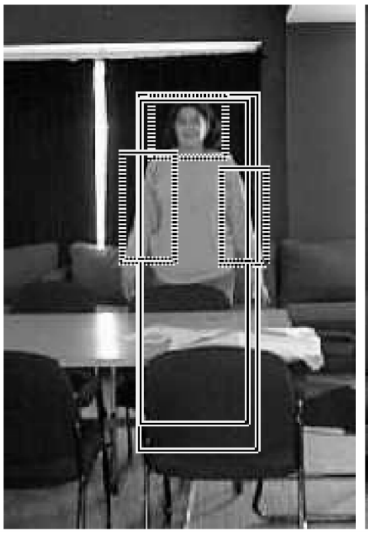


Figure 7: Example of where the Haar method would not work [2]

Another complexity with this method was that the Haar Wavelets 'frames' usually are applicable for the images, not video live stream. Indeed, this problem could have been overcome by storing frames or snapshots of humans during the mission operation and performing analysis on each frame. However, the implementation of such algorithm given the timescale and complexity of the simulation would have made this task almost impossible. Each frame (or one frame out of the thirty available per second) can be saved through the Robot Operating System ROS middleware interface command to Arduino controller and then given the stored location is known the algorithm scanning the image could be implemented. Timely comments from the simulation team and obstacle avoidance team were received, explaining that input/output ports of the Arduino board can be overloaded with the commands required for the respective actions throughout the mission.

2.1.4 Human Detection With Deep Learning

This method is relatively new and is still an ongoing research topic in multiple scientific institutions. The idea behind is to apply real-time object detection using deep learning and OpenCV to output

live video streams. This algorithm both includes deep learning and the object detection code. Deep learning is part of machine learning methods based on artificial neural networks (available on the web), and depending on the application, the learning can be supervised or non supervised. To put it into the context:

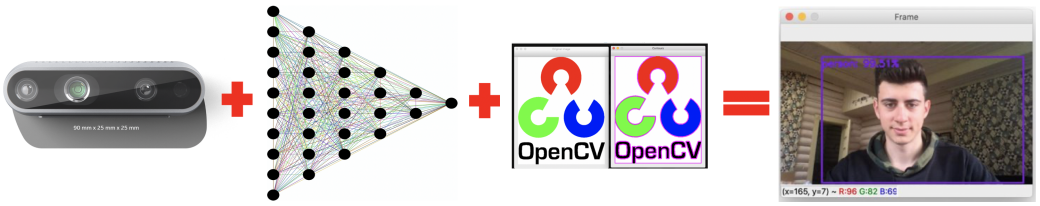


Figure 8: Visual Representation Of The Deep Learning

This method was chosen as the most optimal one in terms of computational costs, implementation and the number of online sources available on the internet. Most importantly, this method granted the members of the CV team learning experience that otherwise would not be possible.

2.1.5 Summary Of The Methods

The key criteria CV team had to look for can be categorised in the table below, where Comp. Cost is the computational cost, Open Resources means how easy it was to find literature/codes/examples associated with a particular method, and Environment Versatility means how agile the method is. Agility implies how well can one specific method work in different environments, such as real-time video, simulated environment etc.

Table 1: Methods summary

-	Comp. Cost	Open Resources	Industry Interest	Simplicity	Env. Versatility
HOG	Moderate	Low	Low	Moderate	Moderate
OpenCV	Cheap	High	Moderate	Simple	Low
Haar	Moderate	Low	Low	Moderate	Low
Deep Learning	Expensive	High	High	Hard	High

As it can be seen from (1) the only downside when choosing the Deep Learning (DL) Method was the relatively expensive computational cost. However, this was not the issue for this project since the high computational costs are inherited when one trains a classifier through a neural network himself/herself. But if a pre-trained classifier can be used, then the computational costs of running the DL do not arise.

2.1.6 Choosing Neural Network

A lot of training networks were developed in the past decade, such as GoogLeNet, VGG-16, Tensorflow et al. When it comes to deep learning-based objection detection with the pre-trained network there are three primary methods available online 1. R-CNN [11] 2. You Only Look Once (YOLO) [12] 3. Single Shot Detectors [13].
R-CNN uses a Region Proposal Network (RPN) that shares full-image convolutional features with the detection network, thus enabling nearly cost-free region proposals. An RPN is a fully convolutional network that simultaneously predicts object bounds and objectness scores at each position [11]. To generate these so called “proposals” for the region where the object lies, a small network is slide over a convolutional feature map that is the output by the last convolutional layer [14]. RPN generate the proposal for the objects (26). RPN has a specialized and unique architecture in itself

(27). This is a subject of ongoing research papers on its own, and it could take months for the CV team to understand the profound idea behind proposals network [14]. The problem that was encountered with this network is that the method was too complicated to grasp within a few weeks. The implementation would take even longer since it required a deep understanding of the pyramid of Anchors, which is challenging for beginners in DL. Lastly, according to [15], the algorithm can be quite slow, on the order of 7 FPS. YOLO is much faster, capable of processing 40-90 FPS on a Titan X GPU [15]. YOLO developers framed object detection as a regression problem to spatially separated bounding boxes and associated class probabilities. A single neural network (NN) predicts bounding boxes and class probabilities directly from full images in one evaluation [12]. SSDs, originally developed by Google, are a balance between the two. The algorithm is more straightforward (and better explained in the original seminal paper) than Faster R-CNNs [15]. The findings can be summarised in the table:

Table 2: Neural Network Summary

-	Comp. Cost	Availability	Difficulty	FPS	Users Friendly	Pre-trained
GoogLeNet etc.	Moderate	Moderate	Hard	Moderate	No	No
R-CNN	Light	Low	Very Hard	Low	No	No
YOLO	Heavy	High	Moderate	High	Yes	Yes
SSD	Light	High	Easy	Moderate	Yes	Yes

The priority in this project was given to the computational cost as some neural networks can be impossible to run with the limited capacity hardware. As a result, a user-friendly and computationally light SSD was chosen as an NN.

2.2 Windows And Holes Recognition

2.2.1 OpenCV and Neural Network Comparison

OpenCV as was mentioned in the sections above is a brilliant tool that can define simple geometries and by using linear 'if' statements in a code one can create a robust algorithm that can distinguish different objects. The CV team would have gone for the DL approach for in the context of stationary objects as well, but the lack of an appropriate data set of wall holes made this task unfeasible. Note that to train a classifier using, for instance, Darknet neural network [16], the maximum layer per image [17] that a CV team member could go through was twenty-two (out of forty required). OpenCV, in contrast, does not need any classifier to be trained, it only takes the basic geometrical constraints and spits out contours as well as the bounding boxes with the class name defined by a user. Examples can include:

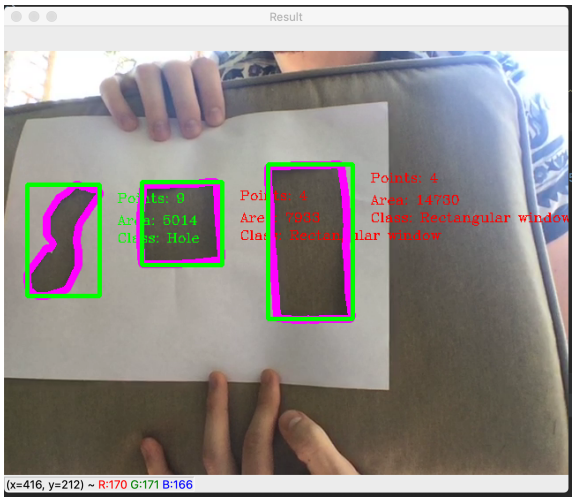


Figure 9: Defining Classes In Real Time

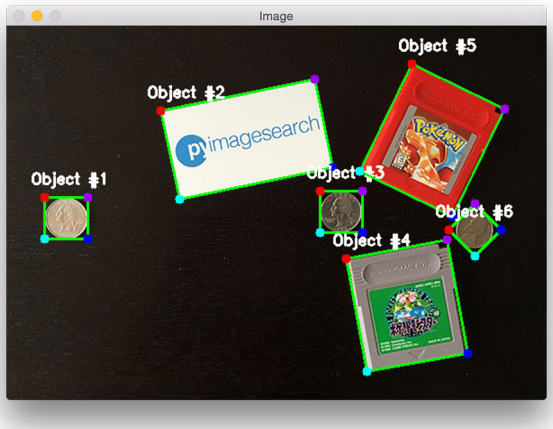


Figure 10: Defining Classes From Images [3]

3 Code Implementation And Results

3.1 Human Real-time Detection

The full code describing Human Recognition can be found in (B). Following closely the approach from Adrien Rosenbrock [18] the code returned highly accurate results. To follow the line enumeration, please refer to the `https://github.com/mirkokovac/GDP2020/blob/master/Computer_Vision/human_detection_custom.py`. It is crucial to explain the reader MobileNet-SSD classifier before diving into technical aspects of the code. MobileNet-SSD extracts feature maps from a single shot and applies convolution filter (a.k.a kernel) to detect the objects [19]. More information about the convolution filters can be found in [20]. The data set on which the classifier was trained COCO (Common Objects In Context), it is the largest free database of labelled images [21]. The MobileNet-SSD couple, as developers say, is capable of classifying object with mean average precision (mAP) of 72%, which is high given that the convolution matrix is relatively small. Computationally-wise this couple does not require new generation video cards to detect classes.

The code was written in Python language, so the basic understanding of Python is assumed in this chapter:

1. Lines 1 to 2 download the necessary modules.
2. Lines 6 to 9 initialize the list of class labels. These labels were taken from the MobileNet SSD, a neural pre-trained platform.
3. Lines 11 to 16 tell the program which classes need to be ignored (i.e. all except for "person"). COLOURS define the random bounding box colour for each object class.
4. Lines 19 to 27 load the serialized model, providing the references to a prototxt and model files. The prototxt is the protocol of the trained model that is required by the cv2.dnn.readNetFromCaffe() command. Line 27 connects to the default camera and outputs the live video stream.
5. Line 30 triggers the infinite loop over which each frame is stored.
6. Line 31 to 41 save each frame and convert the image into the blob, passing it to the network. Blob conversion is essential for the Deep Neural Networks as it performs the two necessary actions: mean subtraction and factor scaling [22].
7. Lines 42 to 82 iterate over each blob detection. The bounding boxes can only be displayed if the confidence level is higher than 20% (convention) and if the class= "person". In line 61, the box variable and np.array command output the (x,y) coordinates of a top left corner of a bounding box as well as its width and height (w,h). Text and labels can be outputted next to the bounding boxes as can be seen at lines 76 to 82.
8. Lines 85 to 97 output the frames and terminate the code if a 'q' on a keyboard is pressed.

Note that to fully understand the idea behind this code, experience of working with Neural Networks is favourable. There is a lot more underlying theory behind each line of code; however, that is not the scope of this project. What the CV team was after were the reliable results that would work in the simulated environment.

3.2 Human Recognition Results

The results of running the code can be seen below:

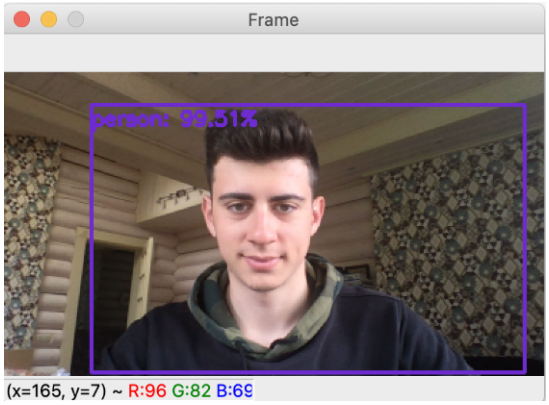


Figure 11: Recognition Of A Human (Myself)

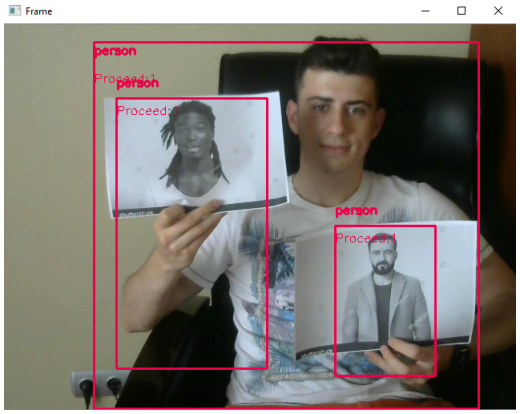


Figure 12: Multiple Humans Can Also Be Classified

The results were astounding! The model was even capable of detecting several humans within a single frame. The snapshot on the right (12) outputs the class without the probability value; probability visualisation was decided to be redundant as only one out of twenty Caffe classes was not ignored. Besides, the proceed command is displayed, depending on the class probability. If '1' is shown, then the authorised team (e.g. pathfinders) can proceed with the necessary commands, i.e. drop the payload etc. The simulation results are shown below::

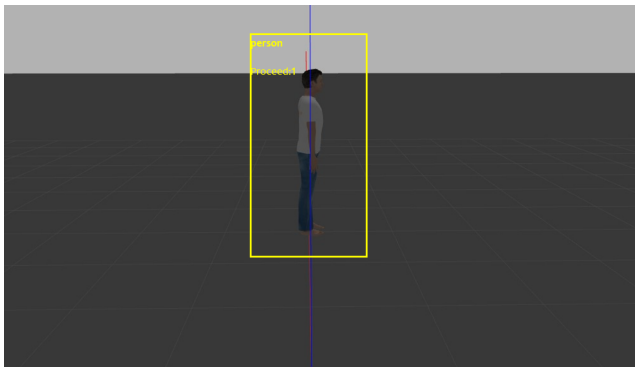


Figure 13: Recognition Of A Simulated Human

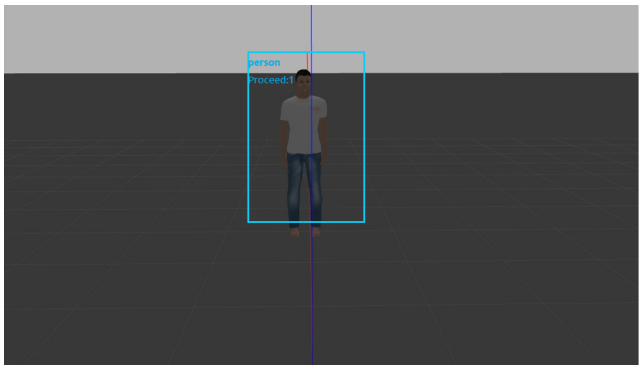


Figure 14: Recognition Of A Simulated Human

What is astonishing about this Caffe model code is that the program identifies humans with high accuracy (confidence probabilities were always higher than 95%). Issues would arise when occluded objects or objects from unusual angles would have to be detected. Caffe model cannot recognise uncommon positions, such as those that were not included in the COCO training data set.

3.3 Windows Detection

3.3.1 YOLOV3 Approach

It was decided by the Simulation Team at a later stage that the holes would be removed; hence the CV team had to focus on identifying the windows. As mentioned in the previous sections, the procedure chosen for this task was set to be the OpenCV module. The alternative approach using YOLOV3 Neural Network (NN) was also implemented, but the team encountered issues with Epochs (training stages). Using the COCO database [21], which has a massive dataset of windows photos, the YOLO NN was used to train the classifier. Still, the computational power did not perform an adequate amount of layers per image. The results the team obtained using a twenty-two out of hundred required layers can be seen:

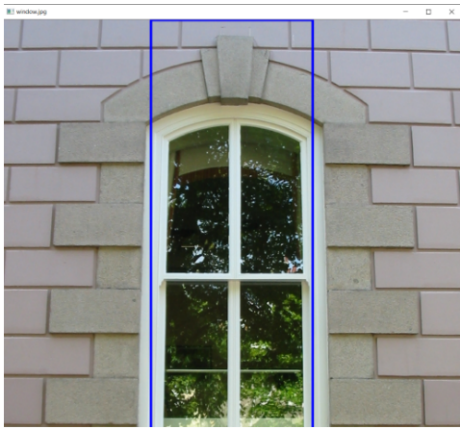


Figure 15: Windows Detection

Figure 16: Windows Detection From A Different Angle

The results are reasonable, but there is one issue with the window on the right-hand side (16). The outputted class probabilities are quite low that is due to two reasons: the picture is taken slightly offset from the perpendicular line, and the training stages limitations decreased the confidence level. As only twenty-two epochs were used, the YOLO NN was less certain about that object being a window. The perpendicularity issue was even more severe when it came to OpenCV approach.

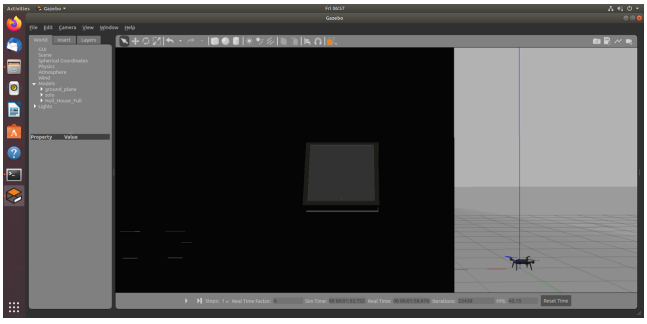
3.3.2 OpenCV Approach

The idea here was to perform simple OpenCV functions without NNs to complete a simple geometry detection. The goal was to output the area in meters squared and the central point of each window such that the necessary trajectory estimations can be performed.

First, the calibration of the object area had to be carried out. OpenCV can only output the area of a particular contour in pixels, not meters squared. Throughout the mission, the drone can move and as it moves the pixel area changes respectively, if the drone is further away from the wall then the pixel area decreases if it is closer to the wall the area pixel increases. This method is unreliable, thus a reliable conversion had to be performed.

3.3.3 Area Calibration And Common Reference Method

The CV team assumed that some parameters before the flight are known, i.e. a snapshot of a dummy object with a known area can be made. This is known as a common reference method (CRM), i.e. a pixel area is converted and scaled based on the parameters known prior to the start of the mission. In reality, if the camera's pixel resolution does not change, the procedure of referencing an image of a known area does not need to be repeated every time before the mission starts. At the time of writing this report, the camera is not installed in the simulation, so only the screenshots could be processed for CRM. One out of multiple window calibration photos is shown below:



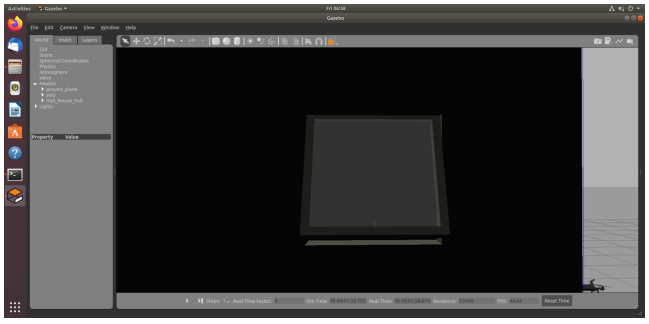


Figure 17: Window From 3 Meters Away (63cm x 60cm) **Figure 18:** Window From 2 Meters Away (63cm x 60cm)

First, the pixel area of the window had to be determined. The screenshot of the window above was taken from the monitor of 1920 x 935 pixels, hence the pixel area of a window from 3 and 2

meters away are 208 x 180 and 461 x 380 respectively. One can assume that the area in meters squared is directly proportional to the product of the pixel area and distance from which that pixel area was taken. Mathematically:

$$A_m = k * A_p * d \tag{1}$$

where k, A_p, d are proportionality coefficient, pixel area and distance respectively. Note, the more pictures are used for calibration, the more accurate results can be obtained. Hence, given that the A_m is defined and does not change the unknown area of an object can be found as follows:

$$A_{m2} = \frac{A_{m1} * A_{p2} * d_2}{A_{p1} * d_1} \tag{2}$$

where indices 1 and 2 stand for fully and partially known parameters respectively. Note that d_1 has to be outputted by the sonar ultrasound sensor. Alternatively, the pathfinding team could have ensured that before the window detection, the drone always has to be a certain distance (e.g. always 2 meters) away from the house. This option was withdrawn as it would severely narrow down the types of windows that can be observed, e.g. panoramic windows. N.B. to test the common reference idea at an early stage of the project, the real-time tests were performed instead of the simulated ones.

3.3.4 Code And Results

The full code including the calibration can be found from `https://imperialcollegelondon.app.box.com/folder/113412233234` or the Appendix (C). Download the file *windowsdetectionfile* from the link above to follow the correct enumeration.

1. Lines 1-15 download the necessary modules, set up pixel resolution, connect to the default camera and define a dummy function.
2. Lines 17-22 create track bars that the user can manipulate to set up filters and define threshold values of the minimum area that is to be detected.
3. Lines 26-30 had to be altered manually by the user to define the calibration parameters.
4. Lines 37-65 define a function that does the following. It takes two parameters the image itself. i.e. each video frame and the dilation filter. The filter is required to find the contours, N.B. any other filter can be used. Within this function an iteration occurs over each detected contour and based on the restrictions, such as the area of the contour and the number of points the contour has, the bounding box (green rectangle) is drawn outside of this contour. Then the string variables, such as area in cm, area in px are displayed.
5. Lines 67-85 create a while loop over which each frame is stored, and variable filters get applied to it. Then once the imgDil filter is created, the function getContours() is called and the second output is then shown as a final result.

The calibration picture of a window (a cut-off square from an A4) of 8 x 8 cm, taken 15 cm away from the camera as well as the testing window of 4x4 cm output the following:

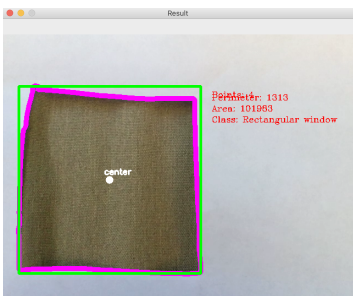


Figure 19: One out of 10 Calibrating Pictures

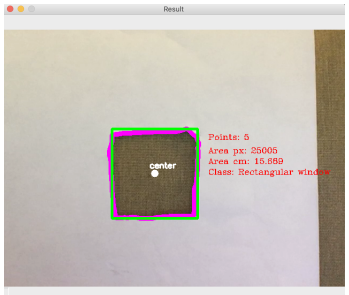


Figure 20: Test Picture 4x4 cm

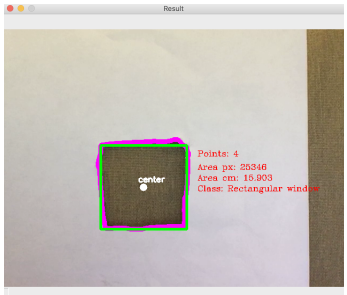


Figure 21: Test Picture 4x4 cm

All the figures had to be taken a constant distance away from the camera (e.g. 15 cm) because a home laptop does not have a sensor measuring the distance in life (the drone indeed has a sonar). The results turned out to be surprisingly accurate, given that the measurement of the distance from the camera was taken by the ruler. For ten different calibrating images, the measured areas yielded between 15.600 cm^2 and 16.300 cm^2, i.e. approximately 2.5% different to the 'real' area. However, the real area was measured using the ruler, so the 'true' area is slightly different to 16 cm^2. To reduce the area uncertainty, multiple cutoffs were made by different humans, and the respective areas were recorded. The CV team investigated how the percentage error would change with the number of calibration pictures used:

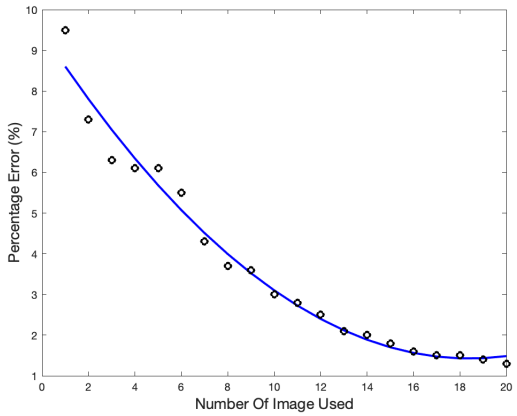


Figure 22: Percentage Error Vs Calibrating Images

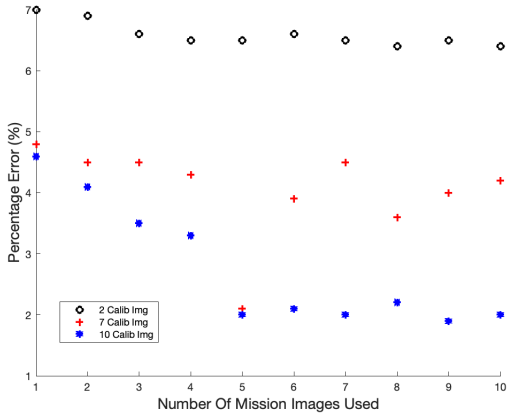


Figure 23: Percentage Error Vs Number Of 'Mission' Images

It can be seen from Figure (22) that the trend somewhat resembles a hyperbola whereby the percentage error approaches 1.1%. The most significant drop occurred between using 1 and 2 calibrating pictures and between 6 and 7. So it is strongly advised to the rescue team to feed at least two images to ensure the percentage error stays around 7%. It was then investigated how would the percentage error between the real and the measured area behave given that multiple pixel area observations can be taken throughout the mission. Assuming the rescue team could make a reasonable number of calibrating pictures, the percentage error versus the 'mission' images can be seen in Figure (23). It can be observed how the error changes for a given number of pictures taken during the mission operation (i.e. when the drone is hovering in front of a window). There is no significant change in error when only two calibration pictures are used, the error oscillates about 6.5%, but the error decreases significantly for the case when ten calibrations were used during pre-flight. So, it is crucial not only to use more than seven calibrations but also to use a reasonable amount of in-flight observations to minimise the area discrepancy.

The outputs from the simulated environment can be seen below:

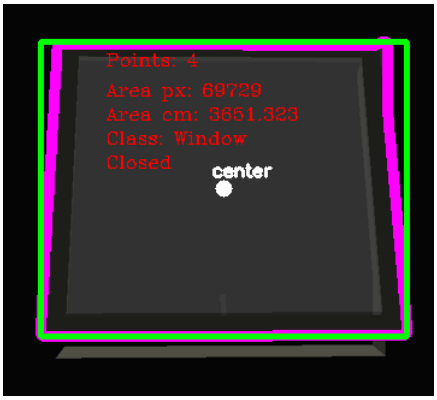


Figure 24: Window Type 1 (63cm x 60cm)

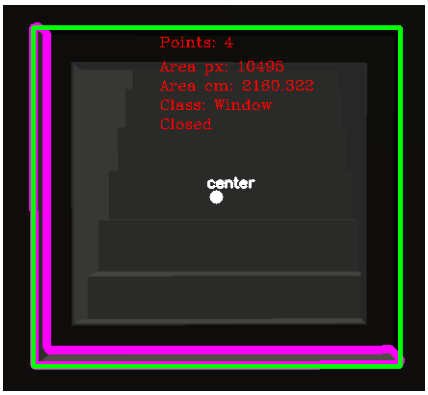


Figure 25: Window Type 2 (48.8cm x 45cm)

Based on the calibration of windows from Figures (17) and (32) it can be seen that the area discrepancy is minimal, 3.4% and 1.6% for Figure (24) and (25) respectively. The percentage error is small in both cases because as shown from in Figure (22), the more calibration pictures are used the better the results would be. The code reading the images (not live stream) can be found in Appendix (33), (34).

4 Conclusion And Potential Improvements

Most importantly, the results were achieved in accordance with the tasks set by the project managers. Both algorithms were capable of detecting the humans and the window frames. With the codes provided on box, the reader can run the scripts on his/her personal machine and verify that the human detection works remarkably well, but there is indeed a room for improvement. Having tested the human code on various people, one issue became apparent, the script failed to recognise the humans from above their heads, or from the feet level. This is due to the limitations of the COCO dataset that was used as the training basis of the Caffe MobileNet-SSD model. Indeed this dataset could have expanded if the CV team added more pictures of humans taken from the perspectives mentioned above. However, this would require more time as making 500-600 images is not a quick procedure, and then using classifier training platforms, a new model could be implemented. Note that classifier training with 50-100 stages can take several weeks on a standard PC. This is a much longer procedure, but it gives more agility and generality to the code as well as to the cases which the rescue team (a.k.a. customers) can encounter.

Apropos the windows detection, given that certain aspects are satisfied; this method can yield accurate results as well. The picture has to be taken perpendicular to the window otherwise, the window gets skewed, and the points restriction set by the code (31) restrains the classification. For the circular windows or any other geometry that does not have 4 to 6 points, the code would have to be adjusted accordingly. Besides, tunning the filter thresholds can be inconvenient, especially for operators new to the CV world. Additional instructions provided from the CV team outlining the threshold boundaries depending on the lighting conditions would also improve the quality of the rescue operations. Indeed the NN method would be advantageous and more innovative. The training dataset available on COCO was used, but the computational power of the home-installed machines was not sufficient to exploit the full power of this ML algorithm.

Last, great news to the potential buyers of this drone would be that the Caffe model implemented earlier was not restricted by a single class. For instance, the Caffe model had the 'IGNORE' function not been used (28), can identify pets. So if the rescue team knows that there are cats or dogs in the flooded house, then vets should also come to rescue, and even more lives can be saved. Another example of improving the quality of the rescuing operation will be to see if there are TV monitors in the house during flooding. The MobileNet Caffe model can identify TV monitors, the drone prior to the rescue team arrival can output the danger sound to the humans inside the house and warn them to disable the electricity or unplug the TV. This action can diminish the risks associated with an electric shock.

References

[1] Navneet Dalal and Bill Triggs. Histograms of oriented gradients for human detection. URL https://lear.inrialpes.fr/people/triggs/pubs/Dalal-cvpr05.pdf.

[2] Constantine Papageorgiou Anuj Mohan and IEEE Tomaso Poggio, Member. Example-based object detection in images by components. URL http://luthuli.cs.uiuc.edu/~daf/courses/AppCV/Papers-2/00917571.pdf.

[3] Adrian Rosenbrock. Ordering coordinates clockwise with python and opencv, . URL https://www.pyimagesearch.com/2016/03/21/ordering-coordinates-clockwise-with-python-and-opencv/.

[4] Karen Hao. The messy, secretive reality behind openai's bid to save the world. URL https://www.technologyreview.com/2020/02/17/844721/ai-openai-moonshot-elon-musk-sam-altman-greg-brockman-messy-secretive-reality/.

[5] Andrej Karpathy. Scaled machine learning conference. URL https://electrek.co/2020/04/21/tesla-videos-autopilot-avoid-pedestrian-crashes/.

[6] C. Papageorgiou and T. Poggio. A trainable system for object detection. URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.91.1537&rep=rep1&type=pdf.

[7] Public Users. Humans in differnt poses data set. URL http://lear.inrialpes.fr/data.

[8] The University of Texas at Austin. Support vector machine approach in the context of image processing. URL http://www.yaksis.com/posts/why-use-svm.html.

[9] Huan-Jun Liu Yao-Nan Wang, Xiao-Fen Lu. A method to choose kernel function and its parameters for support vector machines. URL https://www.researchgate.net/publication/4184763_A_method_to_choose_kernel_function_and_its_parameters_for_support_vector_machines.

[10] OpenCV Enthusiasts. Online documentation of opencv functions. URL https://docs.opencv.org/.

[11] Ross Girshick Jian Sun Shaoqing Ren, Kaiming He. Faster r-cnn: Towards real-time object detection with region proposal networks. URL https://arxiv.org/abs/1506.01497.

[12] Ross Girshick Ali Farhadi Joseph Redmon, Santosh Divvala. You only look once: Unified, real-time object detection. URL https://arxiv.org/abs/1506.02640.

[13] Dumitru Erhan Christian Szegedy Scott Reed Cheng-Yang Fu Alexander C. Berg Wei Liu, Dragomir Anguelov. Ssd: Single shot multibox detector. URL https://arxiv.org/abs/1512.02325.

[14] Tanay Karmarkar. Region proposal network (rpn) — backbone of faster r-cnn. URL https://medium.com/egen/region-proposal-network-rpn-backbone-of-faster-r-cnn-4a744a38d7f9.

[15] Adrian Rosenbrock. Object detection with deep learning and opencv, . URL https://www.pyimagesearch.com/2017/09/11/object-detection-with-deep-learning-and-opencv/.

[16] AlexeyAB. Convolution neural networks. URL https://github.com/pjreddie/darknet.

[17] Jason Brownlee. Convolutional layers for deep learning neural networks. URL https://machinelearningmastery.com/convolutional-layers-for-deep-learning-neural-networks/.

[18] Adrian Rosenbrock. Real-time object detection with deep learning and opencv, . URL https://www.pyimagesearch.com/2017/09/18/real-time-object-detection-with-deep-learning-and-opencv/.

[19] Prakhar Ganesh. Types of convolution kernels : Simplified. URL https://towardsdatascience.com/types-of-convolution-kernels-simplified-f040cb307c37.

[20] Hazım Kemal Ekenel Mehmet Aygün, Yusuf Aytar. Exploiting convolution filter patterns for transfer learning. URL https://arxiv.org/abs/1708.06973.

[21] Michael Maire Pietro Perona Larry Zitnick Tsung Yi Lun, Genevieve Patterson. Common objects in context database. URL http://cocodataset.org/#home.

[22] Adrian Rosebrock. Deep learning: How opencv's blobfromimage works. URL https://www.pyimagesearch.com/2017/11/06/deep-learning-opencvs-blobfromimage-works/.

A Appendix

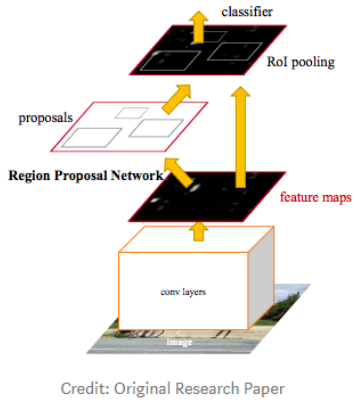


Figure 26: RPN Principle Of Working

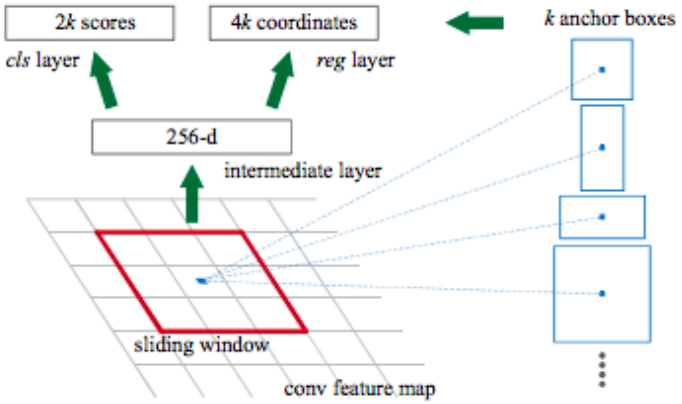


Figure 27: RPN Architecture

B Appendix

```
import
numpy
as np
        import cv2


        # initialize the list of class labels MobileNet SSD was trained to
        # detect, then generate a set of bounding box colors for each class
        CLASSES = ["background", "aeroplane", "bicycle", "bird", "boat",
                   "bottle", "bus", "car", "cat", "chair", "cow", "diningtable",
                   "dog", "horse", "motorbike", "person", "pottedplant", "sheep",
                   "sofa", "train", "tvmonitor"]


        IGNORE = set(["background", "aeroplane", "bicycle", "bird", "boat",
                      "bottle", "bus", "car", "cat", "chair", "cow", "diningtable",
                      "dog", "horse", "motorbike", "pottedplant", "sheep",
                      "sofa", "train", "tvmonitor"])


        COLORS = np.random.uniform(0, 255, size=(len(CLASSES), 3))


        # load our serialized model from disk
        print("[INFO] loading model...")
        net = cv2.dnn.readNetFromCaffe(
            "MobileNetSSD_deploy.prototxt.txt", "MobileNetSSD_deploy.caffemodel")


        # initialize the video stream, allow the cammera sensor to warmup,
        # and initialize the FPS counter
        print("[INFO] starting video stream...")


        cap = cv2.VideoCapture(0)


        # loop over the frames from the video stream
        while True:
            grabbed, frame = cap.read()  # grab the frame dimensions and convert it to a blob
            (h, w) = frame.shape[:2]
            blob = cv2.dnn.blobFromImage(cv2.resize(frame, (300, 300)),
                                         0.007843, (300, 300), 127.5)


            # pass the blob through the network and obtain the detections and
            # predictions
            net.setInput(blob)
            detections = net.forward()
```

Figure 28: First Part Of The Code

```
# loop over the detections
for i in np.arange(0, detections.shape[2]):
    # extract the confidence (i.e., probability) associated with
    # the prediction
    confidence = detections[0, 0, i, 2]


    # filter out weak detections by ensuring the `confidence` is
    # greater than the minimum confidence
    if confidence > 0.2:
        # extract the index of the class label from the
        # `detections`
        idx = int(detections[0, 0, i, 1])


        # if the predicted class label is in the set of classes
        # we want to ignore then skip the detection
        if CLASSES[idx] in IGNORE:
            continue


        # compute the (x, y)-coordinates of the bounding box for
        # the object
        box = detections[0, 0, i, 3:7] * np.array([w, h, w, h])
        (startX, startY, endX, endY) = box.astype("int")


        # draw the prediction on the frame


        # label = "{}: {:.2f}%".format(CLASSES[idx],
        #                              confidence * 100)


        label = "person"
        #varbool = 0
        if label == "person":
            varbool = 1
        else:
            varbool = 0


        cv2.rectangle(frame, (startX, startY), (endX, endY),
                      COLORS[idx], 2)
        y = startY - 15 if startY - 15 > 15 else startY + 15
        cv2.putText(frame, label, (startX, y),
                    cv2.FONT_HERSHEY_SIMPLEX, 0.5, COLORS[idx], 2)
        cv2.putText(frame, "Proceed:"+str(varbool), (startX, y+35),
                    cv2.FONT_HERSHEY_SIMPLEX, 0.5, COLORS[idx], 1)
```

Figure 29: Second Part Of The Code

```
    # show the output frame
    cv2.imshow("Frame", frame)
    #varbool = "0"
    # if label == "person":
    #    varbool == "1"
    print(str(varbool))
    key = cv2.waitKey(1) & 0xFF


    # if the `q` key was pressed, break from the loop
    if key == ord("q"):
        break


cv2.destroyAllWindows()
cap.stop()
```

Figure 30: Third Part Of The Code

C Appendix

```
#Code created by Iaroslav Kazakov foe ADD GDP Imperial College London.
12/6/20

import cv2
import numpy as np
import math

Width = 720  # define the width pixel resolution
Height = 480  # define the height pixel resolution
# can be altered if the CPU of the drone is overloaded
cap = cv2.VideoCapture(0) # Video capturing from the default camera on your
PC
cap.set(3, Width)
cap.set(4, Height)  # Set the pixel resolution

def dummy(x):  # dummy function required for the trackbars
    pass

cv2.namedWindow("Parameters")  # Set window parameters
cv2.resizeWindow("Parameters", 1280, 720)
cv2.createTrackbar("Threshold1", "Parameters", 150, 225,dummy)
cv2.createTrackbar("Threshold2", "Parameters", 150, 225,dummy)
cv2.createTrackbar("A", "Parameters", 5000, 50000, dummy)  # trackbar
creation for areas and threshold
AMinimum = cv2.getTrackbarPos("A", "Parameters")

####### How to change pixels to meters:

# preflight parameters
Am1 = 8*8  # insert the area of a dummy object here
Ap1 = 102000  # use the code below to calculate the pixel are for a given cam
d1 = 15  # distance from the
d2 = 15  # input distance from the lidar sensor




def getCountors(img, imgContour):  # define a function for contour output

    contours, hierarchy = cv2.findContours(img, cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_NONE)  # contour finding


    for cnt in contours:  # looping over each contour to sketch the overall
contour based on the area size
        A = cv2.contourArea(cnt)
        peri = cv2.arcLength(cnt, True)  # outputs perimeter
        approx = cv2.approxPolyDP(cnt, 0.02 * peri, True)  # 0.02 by peri is
a resolution, closed contour stands for True
        Am2 = Am1 * A * d2/(Ap1*d1)
        np.arr[cnt] = Am2
        if A > AMinimum and (len(approx) >= 4 and len(approx) <= 6) and
(4*math.pi*A/peri**2 <= 0.95):
            cv2.drawContours(imgContour, cnt, -1, (255, 0, 255), 7)  # draw
contours if an area of a contour is large enough
            x, y, w, h = cv2.boundingRect(approx)

            cv2.rectangle(imgContour, (x, y), (x+w, y+h), (0, 255, 0), 3)
            center_x = math.floor(x+w/2)
            center_y = math.floor(y+h/2)
            cv2.circle(imgContour, (center_x, center_y), 7, (255, 255, 255),
-1)
```

Figure 31: First Part Of The Window Detection Code Live

```
            cv2.putText(imgContour, "center", (center_x-10, center_y-10),
cv2.FONT_HERSHEY_SIMPLEX, 0.5, (255, 255, 255), 2)

            cv2.putText(imgContour, "Points: " + str(len(approx)), (x + w +
20, y + 20), cv2.FONT_HERSHEY_COMPLEX, .5,
                        (0, 0, 255), 1)
            cv2.putText(imgContour, "Area px: " + str(int(A)), (x + w + 20, y
+ 45), cv2.FONT_HERSHEY_COMPLEX, 0.5,
                        (0, 0, 255), 1)
            cv2.putText(imgContour, "Area cm: " + str(round(Am2,3)), (x + w +
20, y + 65), cv2.FONT_HERSHEY_COMPLEX, 0.5,
                        (0, 0, 255), 1)
            cv2.putText(imgContour, "Class: " + "Rectangular window", (x + w
+ 20, y + 85), cv2.FONT_HERSHEY_COMPLEX, 0.5,
                        (0, 0, 255), 1)

while True:
    success, img = cap.read()  # reads the frame per second from the default
cam
    imgContour = img.copy()  # copying the img frame per second

    imgBlur = cv2.GaussianBlur(img, (7, 7), 1)  # applying the gaussian blur
    imgGray = cv2.cvtColor(imgBlur, cv2.COLOR_BGR2GRAY)  # changing the bgr
into the gray format

    threshold1 = cv2.getTrackbarPos("Threshold1", "Parameters")  # defining
the threshold
    threshold2 = cv2.getTrackbarPos("Threshold2", "Parameters")
    imgCanny = cv2.Canny(imgGray, threshold1, threshold2)  # applying the
canny filter

    kernel = np.ones((5, 5))#kernel is the 5x5 array
    imgDil = cv2.dilate(imgCanny, kernel, iterations=1)

    getCountors(imgDil, imgContour)# contouring the imgDil image

    cv2.imshow("Result", imgContour)
    if cv2.waitKey(1) & 0xFF == ord('q'):
       break
```

Figure 32: Second Part Of The Window Detection Code Live

D Appendix

```
import numpy as np
import cv2
import math

Am1 = 8*8  # insert the area of a dummy object here
Ap1 = 102000  # use the code below to calculate the pixel are for a given cam
d1 = 15  # distance from the
d2 = 15  # input distance from the lidar sensor
AMinimum = 5000

def getCountors(img, imgContour):  # define a function for contour output

    contours, hierarchy = cv2.findContours(img, cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_NONE) # contour finding


    for cnt in contours:  # looping over each contour to sketch the overall
contour based on the area size
        A = cv2.contourArea(cnt)
        peri = cv2.arcLength(cnt, True)  # outputs perimeter
        approx = cv2.approxPolyDP(cnt, 0.02 * peri, True)  # 0.02 by peri is
a resolution, closed contour stands for True
        Am2 = Am1*A*d2 / (Ap1*d1)
        if A > AMinimum and (len(approx) >= 4 and len(approx) <= 6) and
(4*math.pi*A/peri**2 <= 0.95):
            cv2.drawContours(imgContour, cnt, -1, (255, 0, 255), 7)# draw
contours if an area of a contour is large enough
            x, y, w, h = cv2.boundingRect(approx)

            cv2.rectangle(imgContour, (x, y), (x+w, y+h), (0, 255, 0), 3)
            center_x = math.floor(x+w/2)
            center_y = math.floor(y+h/2)
            cv2.circle(imgContour, (center_x, center_y), 7, (255, 255, 255),
-1)
            cv2.putText(imgContour, "center", (center_x-10, center_y-10),
cv2.FONT_HERSHEY_SIMPLEX, 0.5, (255, 255, 255), 2)

            cv2.putText(imgContour, "Points: " + str(len(approx)), (x + w -
250, y + 20), cv2.FONT_HERSHEY_COMPLEX, .5,
                        (0, 0, 255), 1)
            cv2.putText(imgContour, "Area px: " + str(int(A)), (x + w -250, y
+ 45), cv2.FONT_HERSHEY_COMPLEX, 0.5,
                        (0, 0, 255), 1)
            cv2.putText(imgContour, "Area cm: " + str(round(Am2, 3)), (x + w
-250, y + 65), cv2.FONT_HERSHEY_COMPLEX, 0.5,
                        (0, 0, 255), 1)
            cv2.putText(imgContour, "Class: " + "Window", (x + w -250, y +
85), cv2.FONT_HERSHEY_COMPLEX, 0.5,
                        (0, 0, 255), 1)
            cv2.putText(imgContour, "Closed " , (x + w - 250, y + 105),
cv2.FONT_HERSHEY_COMPLEX,
                        0.5,
                        (0, 0, 255), 1)


img = cv2.imread("Window_contour_sim1.png") #reads images
threshold1 = 68  # Tune these parameters based on the environment lightning
threshold2 = 70  # Same as above
imgContour = img.copy()  # copying the img frame per second
imgBlur = cv2.GaussianBlur(img, (7, 7), 1)  # applying the gaussian blur
imgGray = cv2.cvtColor(imgBlur, cv2.COLOR_BGR2GRAY)
imgCanny = cv2.Canny(imgGray, threshold1, threshold2)  # applying the canny
filter
kernel = np.ones((5, 5))  # kernel is the 5x5 array
```

Figure 33: First Part Of The Window Detection Code Image

```
imgDil = cv2.dilate(imgCanny, kernel, iterations=1)
getCountors(imgDil, imgContour)  # contouring the imgDil image
cv2.imshow("Output", imgContour)  # name of the output

cv2.imwrite('Window_contour_sim11.png', imgContour)
cv2.waitKey(15000)
cv2.destroyAllWindows()
```

Figure 34: Second Part Of The Window Detection Code Image