SUMMARY REPORT

DRAFT PoliTo Target Recognition Report

Here's a report about the work I did in the Computer Vision and Deep Learning division in the DRAFT PoliTo student team.

Doc. Ref.	SR-DLCV-002
Tribe:	Deep Learning and Computer Vision
Squad:	Computer Vision
Date of issue	26/10/2020
Prepared by	Francesco Dibitonto
Student ID	S265421
Date of admission of the student work	10/02/2020
Estimated working hours	175
Checked by	Francesco Marino (Tribe Leader)
Approved by	Prof. Giorgio Guglieri

1 Table of Contents

2 Figures

3 Tables

4 Definitions and abbreviations

4.1 Definitions

Scrum meeting	Meeting to stay updated on each other’s activities and for setting the context for the next one.

4.2 Abbreviations

AI	Artificial Intelligence
SSD	Single Shot Detector
IDE	Integrated Development Environment
Pi	Short form for Raspberry Pi
CNN	Convolutional Neural Network

5 Abstract

ENGLISH

Nowadays, AI has become a very hot topic, and its applications become broader and broader in many and various fields. In particular, researches and development of innovative solutions based on AI are heavily applied in the robotics field. Solutions aimed to increase the autonomy of the current drone technology are continuously explored, yet highly demanded. Those include autonomous flight and landing, visual navigation and obstacle avoidance, with multiple applications, like delivery-by-drones, flight refuelling and many more. However, these technologies face many challenges, such as the localization of the target.
In this work, I face up the problem of target recognition and localization, I set up a pipeline starting from scratch up to the objective, and I adapt the solution to be fully working into an embedded system environment. This includes the employment of Computer Vision algorithms for handling and manipulating video-frame images, the creation and labelling of target datasets and the training of a Single Shot Detector (SSD), for reliably recognizing the targets and their position, in the context of Deep Learning.
The results are stable and satisfactory, achieving good performances for the targets, tested and fully working in an embedded system device like a simple Raspberry Pi.

ITALIANO

Oggigiorno si parla sempre piu’ spesso di AI, le cui applicazioni diventano sempre piu’ vaste in molti e nei piu’ disparati ambiti. In particolare, ricerca e sviluppo di soluzioni innovative basate sull’AI vengono ampliamente applicate nel campo della robotica. Soluzioni incentrate sul miglioramento dell’autonomia dell’attuale tecnologia dei droni vengono continuamente esplorate, cio’ nonostante esse sono sempre e ancora altamente richieste. Queste includono volo ed atterraggio autonomo, navigazione visiva e l’elusione degli ostacoli, con molteplici applicazioni, quali consegne effettuate dai droni, rifornimenti in volo ed ancora molte altre. D’altro canto, queste tecnologie sono tuttora soggette a molte sfide, quali ad esempio la localizzazione dell’obiettivo.
In questo lavoro, ho affrontato il problema del riconoscimento e della localizzazione dell’obiettivo, ho impostato una metodologia partendo da zero fino ad arrivare all’obiettivo, e ho adattato la soluzione affinchè essa fosse completamente compatibile e funzionanante nell’ambiente dei sistemi integrati. Ciò include l’utilizzo di algoritmi di Computer Vision per la gestione e la manipolazione dei singoli fotogrammi in quanto immagini, la creazione e l’etichettatura di dataset-obiettivo e l’allenamento di un Single Shot detector (SSD), per riconoscere in modo affidabile gli obiettivi e le loro rispettive posizioni, nel contesto del Deep Learning.
I risultati sono coerenti e soddisfacenti, ottenendo buone performance per gli obiettivi, il tutto testato e pienamente funzionante in un sistema integrato quale puo’ essere un semplice Raspberry Pi.

6 Scope and purpose

This document describes the activity executed by **Francesco Dibitonto** as member of student team DRAFT PoliTO within the cv-squad of Computer Vision and Deep Learning division. This document is in partial fulfilment of the requirements for obtaining the recognition of 6 CFU for the activity conducted in the team.

7 Motivations

Our team's aim was to suit a flying drone with programs and modules in order to participate to the Leonardo's drones competition held on the 18th of September, in Turin. Here you can find for reference an article of the LaStampa newspaper which briefly describes the contest: https://www.lastampa.it/torino/2020/09/18/news/al-via-il-drone-contest-la-gara-in-cui-si-sfidano-i-giovani-ingegneri-volanti-1.39322104.
My work, in particular, concerned the Target Recognition programming part, to be loaded into the drone to recognize, in fact, the targets. The main targets were QR codes displaced all around the competition field, and the competition's objective was to let the drone to find, identify and land on the most possible amount of targets in the least amount of possible time. The secondary target was an 'H' Landing Pad (i.e. an ordinary landing pad with an H depicted onto it) that the drone should have landed onto, once finished with the main target recognition part.

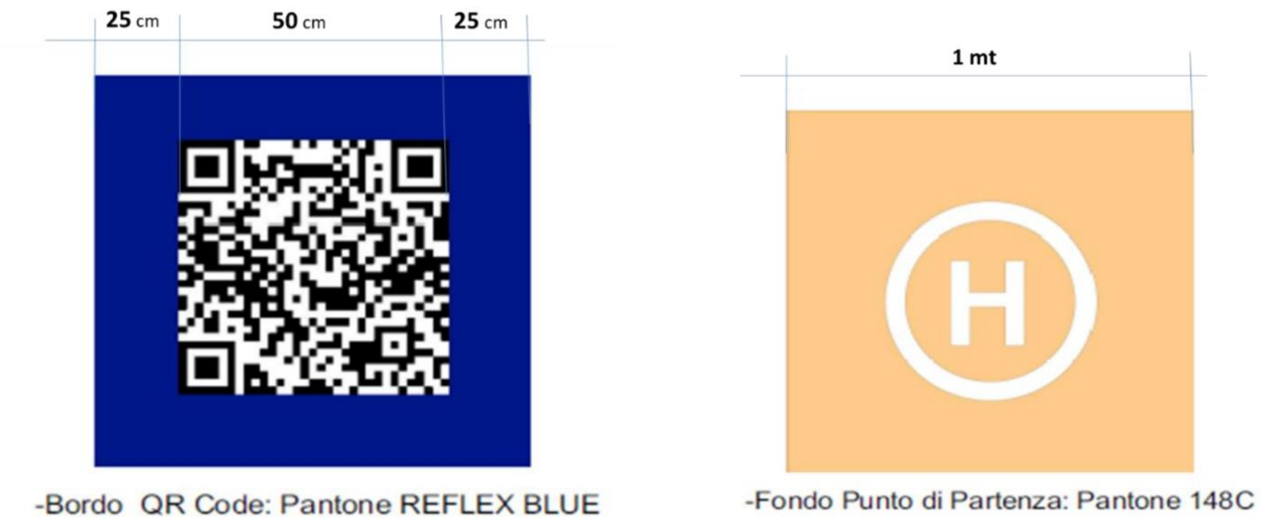
25 cm
50 cm
25 cm
-Bordo QR Code: Pantone REFLEX BLUE
1 mt
H
-Fondo Punto di Partenza: Pantone 148C

Figure 1: Target examples. On the left, a QR code target example. On the right, an H landing pad target example.

Hence, my work included the recognition of the QR code, including its scanning and reading, either to be passed to the Trajectory Planning squad to adjust the pathing of the drone, in order to reach, and land on, the targets, and also for reading the target QR codes once near enough to them. It also included the recognition of the final landing pad.

The logical pipeline I planned yet the ordered workflow I applied can be resumed by the following steps, subdivided into module groups according to the chronological order I worked on them:

G1 (Group1):

- Creation of a programming module (i.e. a script) able to look for and read the value a QR code (our main type of target) on a single image.
- Creation of a script able to read each video-frame captured from the installed camera as a sequence of single images.
- Creation of a script able to perform a 2D straightening operation on an already identified, but not yer read, QR code, in a way the QR gets correctly read even if found on a frame with some 2D inclination (e.g. 90 degrees).
- Integration of all the programming modules previously listed into a single one able to capture, read, and display the reading of each QR code found in the video-frames, regardless of their 2D inclination.

G2 (Group2):

- Creation of a dataset containing images of QR codes 'in the wild', i.e. QR codes found on walls, sold products, boxes etc.
- Manual labelling of each image within the dataset, including bounding boxes (i.e. squares enclosing each target object) and class labels (QR codes and Landing Pads).
- Translation of the labelling annotations format (.xml) into a simpler (.csv) format.

G3 (Group3):

- Selection and implementation of a Deep Architecture, both light enough to be mounted on the desired (e.g. Raspberry Pi) embedded system and able to perform object detection.
- Creation of an online Data Augmentation script able to augment (i.e. perturb and/or modify) the samples available during the training phase, taken from the previously created dataset.
- Training and Evaluation of the Deep model on a common PC.
- Testing the Deep model on the desired embedded system device.

G4 (Group4):

- Integration of the modules belonging to G1 with G3, obtaining a single overall module able to detect a target within a frame, and to limit the scanning and the reading of the QR code to the zone enclosed by the bounding box(es).
- Testing this new module both on PC and on the embedded system device.

G5 (Group5):

- Creation of a script able to create 3D rotated versions of a flat landing pad image.
- Creation of an 'artificial' dataset consisting of the 3D rotated landing pad images superimposed on other images coming from aerial and scene datasets.
- Creation of a sub-script able to automatically label the artificial images on the fly (i.e. while the dataset is being created).
- Translation of the labelling annotations format (.xml) into a simpler (.csv) format.

G6 (Group6):

- Re-training on both the two types of target (QR and landing pad) end Evaluation.
- Final Testing of the Deep model on the desired embedded system device.

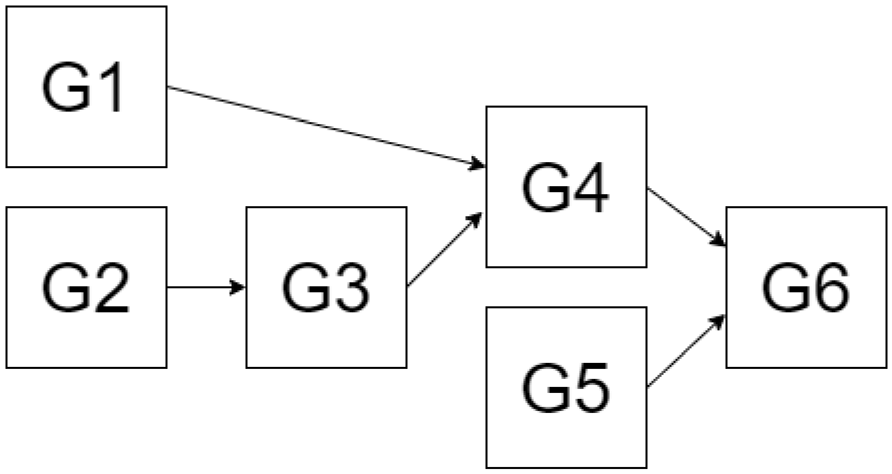

Figure 2: Working blocks. The module groups of the pipeline.

8 Methodology

8.1 Communication Tools

Before starting with the real work, I was introduced to the communication platform our team would have used for updating and coordinating ourselves and our works. We used Slack (https://app.slack.com/) to share general ideas, articles, papers and news related to the topic, as well as for organizing the next meetings.
Concerning the type of meeting, we used to do a bi-weekly online Scrum meetings of the duration of one hour in which we described our previous work and we were assigned and/or proposed new tasks to do.
For describing, updating, commenting, or monitor the status of our tasks and the ones of the teammates, we used Trello (https://trello.com/), a free online tool including all such functionalities. It was also a useful reminder to track down what was already done in the past for reference, as well as for attaching links and material to facilitate our task development.

8.2 Code development, programming language and libraries

All the code I developed was written in the Python programming language. The IDEs I used were mainly Thonny (https://thonny.org/, the default editor pre-installed on the Raspberry Pi) when working directly on the Pi, and Google Colab (https://colab.research.google.com/) when working on my PC, or when running the computational heavy, deep model related, scripts. Other than providing an IDE for coding in Python, Colab provides free cloud GPUs to be used for Deep Learning, making it especially useful for training deep neural networks.
The main libraries I used all around my work were mainly OpenCV (https://opencv.org/), a real factotum for Computer Vision, and Keras (https://keras.io/), one of the main neural network libraries available on the web.
Yet another honourable mention goes to imgaug (https://imgaug.readthedocs.io/en/latest/), a library supporting many data augmentation techniques.

8.3 QR code reading

The initial task I was assigned to was about developing a script able to read out the value of a QR code. After looking on the web about the already existing solutions, I implemented mine based on the Python library pyzbar (https://pypi.org/project/pyzbar/), a ready-to-go library already implementing the QR reading functions. I tested my pyzbar-based solution onto some static images and it worked well, made an exception for when the QR code in the image was not 'straight', but rather present at some degree of inclination (e.g. 90 degrees). For the moment I neglected the problem and I tested it with the images provided by the Pi camera, by first taking the video sequence and isolating every frame to be passed as an image to the QR reader module. Working still well, but yet presenting the same tilt problem, I solved that thanks to OpenCV, by basically ‘straightening up’ the QR code image (i.e. ~ 0 degrees) before the reading. But still, the reading was too much dependent on the 3D inclination rather than on the 2D one, hence the camera had to point nearly exactly faced to the QR code, otherwise both the recognition and the reading wouldn’t work.

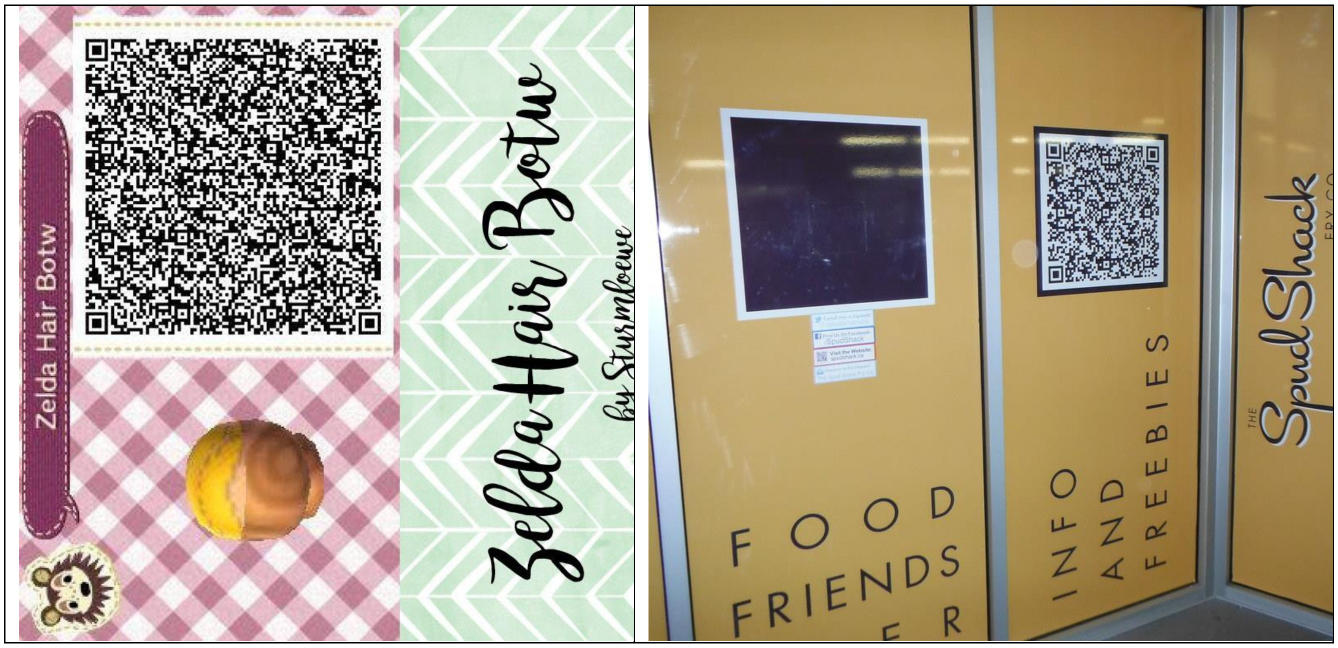

Figure 3: Reading cases. On the left, the recognition/reading algorithm would work after the 2D straightening trick. On the right, the recognition/reading algorithm would not ever work, even if the 3D tilt is minimal.

So I needed some way to still at least be able to identify the QR code in the 3D tilt scenario (without reading it), such that the drone could then move up to it, face it straight, hence read its value and eventually land on it.
After a lookup both at articles found on the web and at papers available from literature, I got the paths I could take were mainly two: one based on a more traditional approach entirely based on Computer Vision methods (hence on OpenCV) for the QR recognition, and one based on AI and Deep Learning based on Deep Neural Networks for Deep Object Detection.
Since I was sceptical about the traditional route, especially about the capability of a pure Computer Vision algorithm at detecting an object in the presence of perturbations (obstructions, light glares etc.), and since I was also more experienced about Deep Learning, I chose the latter approach. So the first two things I would have needed were a Deep model, complex enough to recognize the QR codes but at the same time light enough to be run on an embedded system device like the Pi, and a dataset for training such model.

8.4 Model Selection

After looking around on the web for a good model able to perform object detection, I chose Mobilenet (https://keras.io/api/applications/mobilenet/). Mobilenet is a CNN whose architecture has been built purposely to be lightweight in terms of computational complexity while still achieving good performance. That’s what makes it very desirable for embedded system devices like the Pi. What made it even better for my needs, it’s the fact that it can be even integrated into an SSD framework, hence becoming able to learn to predict bounding box locations while also classifying those locations in one pass (or in ‘one-shot’).

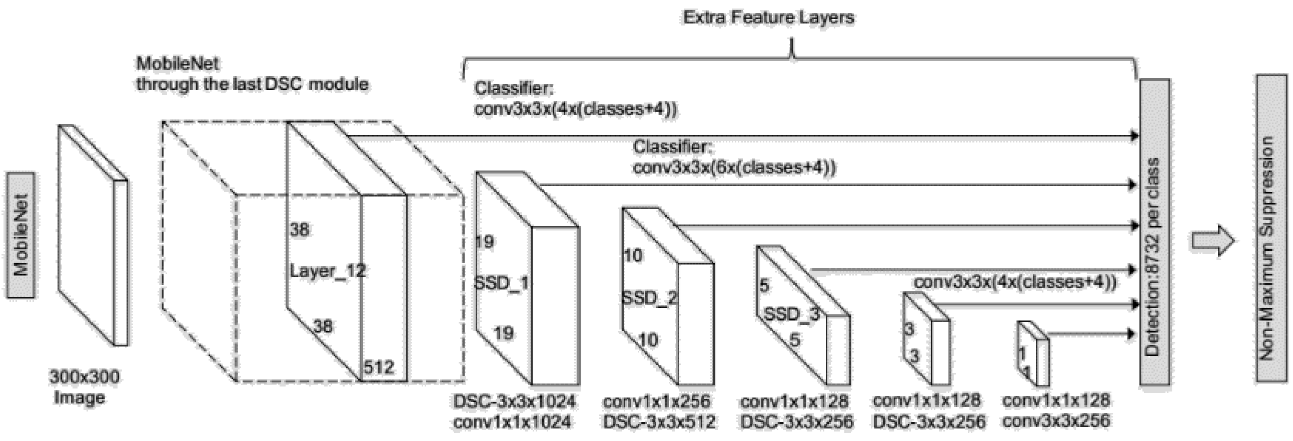

Figure 4: SSD-Mobilenet architecture. The Mobilenet base architecture (on the left) gets connected to the SSD layers (on the right).

Thus, by using a SSD, only a 'single shot' has to be taken in order to detect multiple objects within the same image (e.g. two QR codes, or a QR code and a landing pad next to each other), making it faster yet more desirable with respect to other equivalent (from a functional point of view) solutions. Moreover, the SSD-Mobilenet architecture was expressly thought to be used for real-time detection, i.e. constant object recognition from a video-stream, which it's exactly our case scenario.
All of these points, together with the fact that plenty of documentation is available about the model, as well as plenty of Python code and examples, drove me towards the choice of using the SSD-Mobilenet model.

8.5 QR dataset creation

In order to let a neural network to 'learn', we have to feed it with many examples such that it can 'digest' the information contained in them. Such samples have to be labelled so, in the object detection scenario, they have both to include the bounding boxes (i.e. square boxes enclosing each target object to be recognized) and the class labels (i.e., for each bounding box, I've to specify the type of target object contained within it, in our case either QR code or H landing pad).
I searched on Google Images (https://www.google.com/imghp?hl=EN) images about QR codes 'in the wild', meaning that those images had to show QR codes on walls, sold products, boxes etc. This had to be done as the network has to learn to identify QR codes found in the real world, hence it needs to learn from real-world examples. I collected ~1400 image samples in this way, and I labelled each of them by using the free labelImg (https://github.com/tzutalin/labelImg) tool.
As the label output format of labelImg is the PASCAL VOC format, i.e. XML format containing many data, I wrote a script for translating the labels into a simpler CSV format while also eliminating the unnedeed data, for the sake of simplicity as it will be easier in this way to load this cleaner data into the network in the next step.

8.6 Data Augmentation

In order to strengthen the detection capabilities of the network in case of perturbation and in case of novel outputs (i.e. unseen with respect to the ones provided as training samples), data augmentation comes very handy. Data augmentation consists in a series of techiques aimed to artificially increase the size of the training dataset, in a way the network 'sees' more diverse types of images when learning, training also on them, with the result of improving the network generalization capabilities. I used the imgaug library which already implements many of such techniques, like grayscaling, shearing, cropping, rotating etc.
As around 1000 train samples are not so many when talking about training a deep neural network, especially when aiming to very good performance scores, data augmentation is beneficial in our setting also for this reason, as it is going to partially compensate the relative scarcity of data I collected. Finally, I implemented it in a way it's performed in an online fashion, so the image samples are 'augmented' on the fly when directly 'seen' by the network at training time.

8.7 Model Training

Before feeding the collected image samples into network to let it learn, I divided the dataset first into a training and a test set, with a ratio of 80% to 20% respectively. This has to be done as the network needs to be tested at the end of the training process on totally unseen images, in order to impartially assess its generalization capabilities when dealing with images it didn't see at training time. Hence we should do it as we want the network to recognize objects from images coming from the real-world, potentially always new and various, so the network should have trained in a way to be flexible enough for detecting and classifying still correctly such new image cases.
But still, a way to assess the performance of the model during the training process is needed, hence I furtherly subdivided the training set into training and validation set respectively, again with a ratio of 80% to 20%. This time this is done to 'test' the model on the fly at each learning iteration, so we can see if the network is gradually improving or not.
When fed into the netwoek, all the images are automatically resized to the network input size (i.e. the image dimension the network 'accepts') through a couple of lines of codes using OpenCV.
Regarding hyperparameters optimization (i.e. the 'settings' that can be changed in order to achieve the best possible results), I left the default-ones proposed in the official implementation, made for exception for the number of epochs. Those are:

- Batch size = 8
- Adam optimizer
- Learning rate = 0.0001
- Weight decay = 0.0001
- Number of Epochs = 300 (versus 100 in the original implementation)

Let's see very briefly what each of such term mean with an high level explanation for each one.
The batch size is the amount of images the network sees at each single training step. The training phase is an iterative process, i.e. it consists of single steps made one after the other, such that the network learns procedurally one step at time.
The optimizer is an algorithm deciding how the network 'improves' when training, so it affects the way it effectively learns.
The learning rate determines how much 'roughly' we want the network learns, with high learning rates meaning the network learns faster but in broad terms, and with low learning rates meaning the network learnes slower but in a more fine-grained and detailed way.
Weight decay is an hyper-parameter that impacts on the complexity of the model. The smaller it is, the more we force the model to be flexible about the possible unseen image samples, rather than stay 'stubborn' and focus more on the samples seen during training only. At the same time, the smaller this parameter is, the longer the learning process will also be.
The Number of Epochs is about the number of times the network process and learns onto the whole training set. The higher this number is, the more it's the 'time' we let the network for learning.
I used Average Precision as a metric in order to assess the performance of the model at each learning epoch. Since I noticed that after 100 epochs the Average Precision score on the validation set was still increasing, hence the network was still improving, I made the training process last longer up to 300 epochs, where the Average Precision stopped around 0.74.

8.8 Model Testing

In order to evaluate the goodness of the trained network at detecting the QR codes, I fed the network with the held-out test images samples and I achieved a test Average Precision Score of 0.34. Not satisfied by this result (being it less than half of the validation accuracy), I wanted to visually assess where the network's detections were wrong, to understand where the error was. Hence, I showed the networks outputs on the test set in the form of bounding boxes together with the associated class labels. Here are shown some of the results:

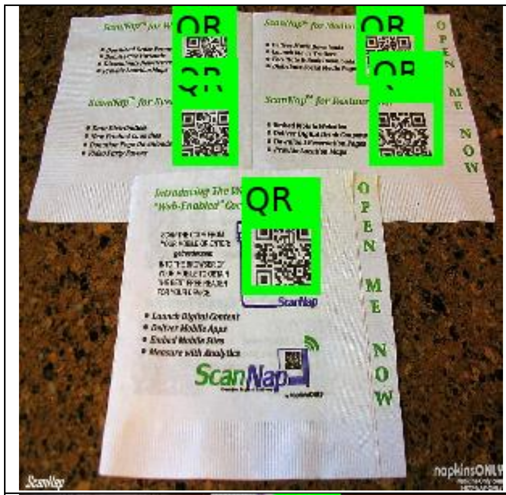	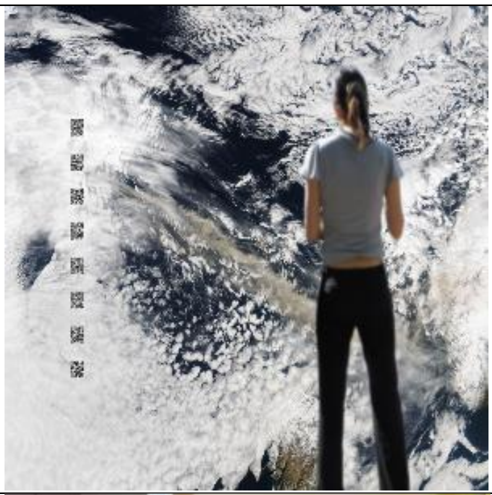
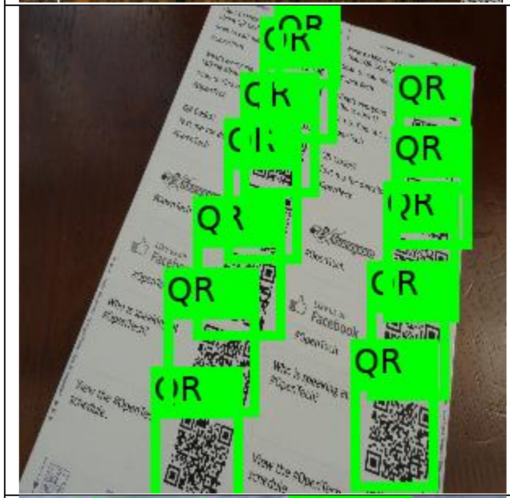	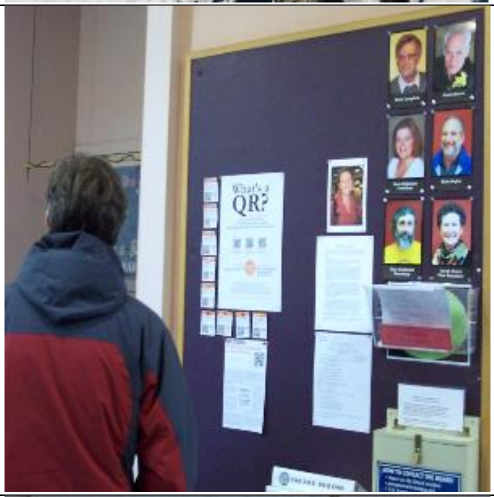
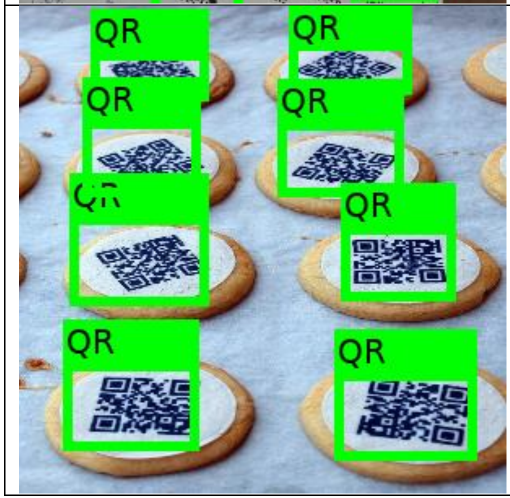	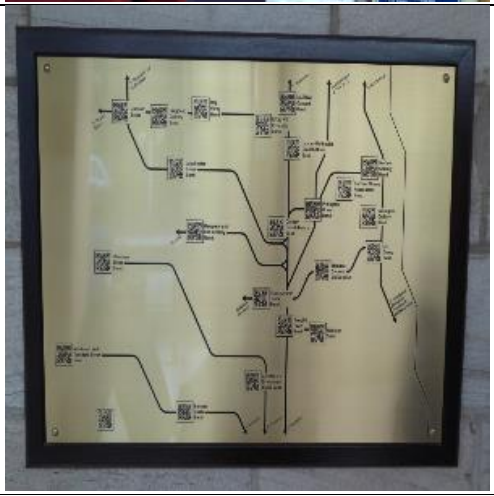

Table 1: recognized test targets (on the left) versus unrecognized test targets (on the right).
We can notice how when the QR codes size become too small the network stops to recognize them.

Fortunately, the network was doing good! The issue was it failed to recognize all the targets that were too small, and that was the reason of the drop on the Average Precision. As it was perfectly fine for me, it was now time to test the network in real life on the Pi.

8.9 Testing on the Raspberry Pi

By providing each video frame captured by the Pi camera to the network, I obtained these results:

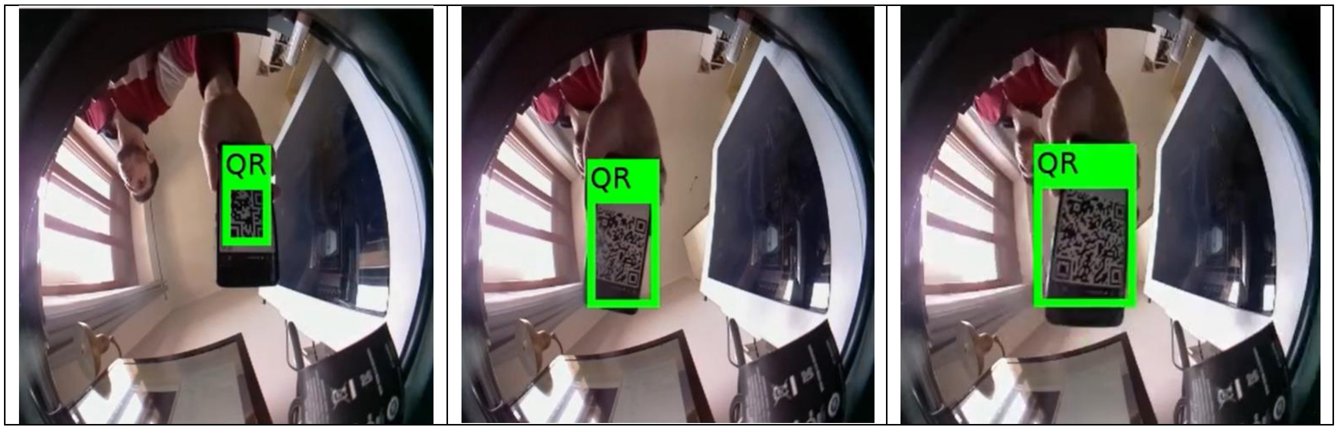

Figure 5: Recognized targets from single video frames.

The network was able to detect every QR code in 99% of the frames! Even with poor light conditions and with the distortion effect due to the fish-eye camera.
So I integrated the network code with the reading code and also the reading worked, but still provided that the QR was not tri-dimensionally tilted more than a few degrees. But now that should have not been a problem anymore, since if the drone would now be able to detect the target, whatever its position and tilt, it would then move to directly face it, hence becoming able to read it.

8.10 Landing Pad dataset creation

Due to the fact the H landing pad used in the competition was simply a 'flat mat', I couldn't find images on the web about such pads 'on the wild' like I did for the QR codes. So the alternative was to create an 'synthetic' dataset, i.e. a dataset created artificially with the intent of approximate and resemble as much as possible a dataset made of real-life images.
After a brief literature review I decided to adopt the same method employed by Yang et Al. (https://www.mdpi.com/2072-4292/10/11/1829), basically consisting of artificial 3D rotations of the flat target, and the superimposition of those onto aerial and scene datasets. Hence, I created a script for the rotation, I downloaded the validation sets of the DOTA (https://captain-whu.github.io/DOTA/index.html) and the COCO (https://cocodataset.org/#home) datasets, and finally created another script to superimpose the rotated pads onto random locations of the collected images. While performing the superimposition operation, I recorded the position and obtained automatically the bounding box (and the class) of each superimposed H landing pad, and I again successively translated such annotations into the CSV format. In this way I created an artificial annotated dataset made of ~2000 images. Here's an example for visual reference:

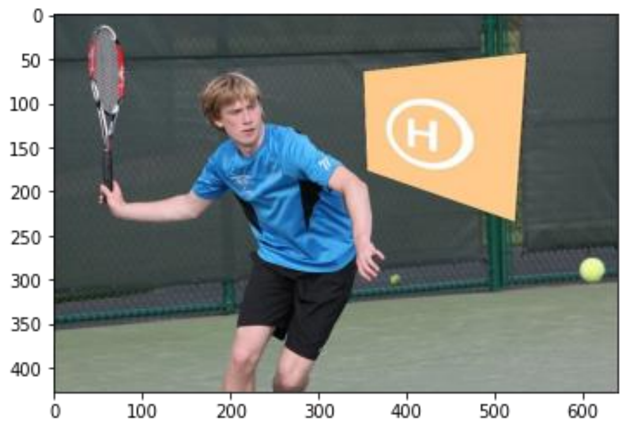

Figure 6: sample from artifical H landing pad dataset.
The 3D rotated flat landing pad is superimposed on a COCO validation set image.

Note that also these samples were augmented in the same way of the QR code ones, again to improve the generalization capabilities of the network.

8.11 Model Re-Training

Now that I had both the QR code and the H landing pad dataset, I re-trained the network from scratch with the two available datasets, so by feeding it with both images of QR codes and landing pads and let it learn to detect them both.
For the rest, the training process was identical with respect to the previous QR codes-only one, including train-validation-test size ratios and hyper-parameters.
At the end of the training procedure the Average Precision scores for the QR codes and the H landing pads validation sets were respectively 0.7 and 0.95.

8.12 Final Testing

I repetead the testing procedure by feeding into the network both the QR code and the H landing pad test sets and achieved a test Average Precision score of 0.36 and of 0.93 respectively. Here are shown some visual results:

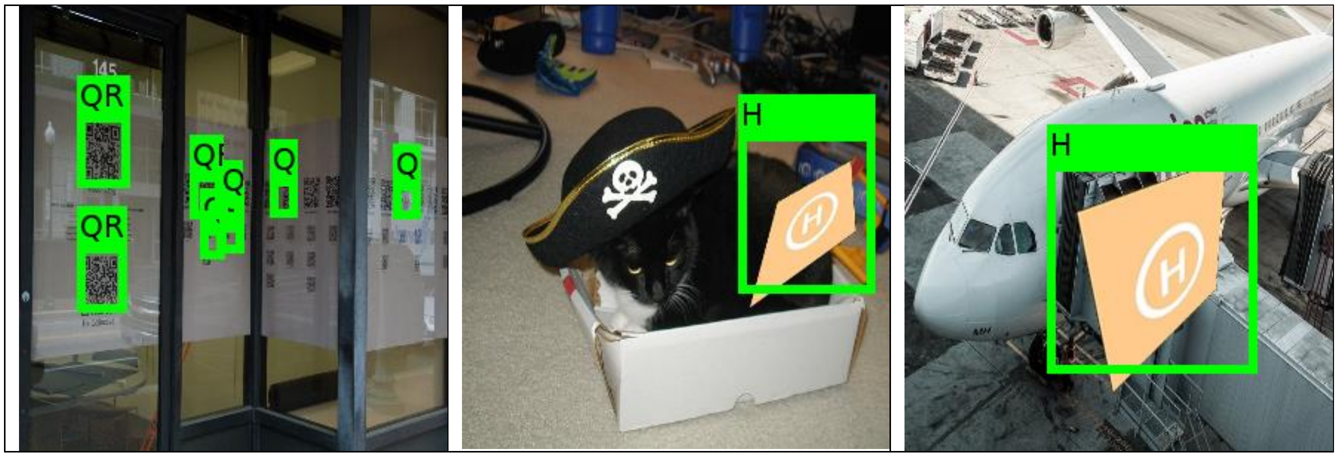

Figure 7: recognized test targets from both the test sets.
Again we can notice the smallest targets are not recognized.

Finally, I tested this final model on the Pi. These final results will be shown and discussed in the next section.

9 Results and recommendations

As a final test, I fed again the network with the video-frame images captured by the Pi and these were the results:

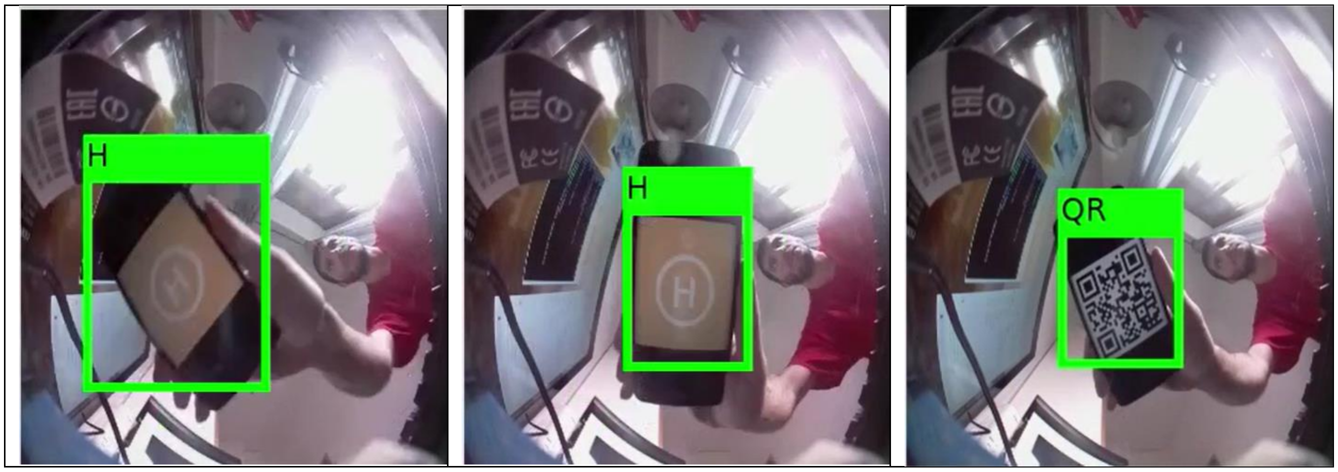

Figure 8: Recognized both types of targets from single video frames.

Once more time, the network was able to detect every QR code in 99% of the frames, but around 80% of the H landing pads. This was due to the fact the H landing pad training was performed on a synthetic dataset, hence the degree of performance by doing so cannot be on par with a training on a real dataset. The artificial dataset acted as a surrogate of a real one, in order to mitigate the lack of real-life images of such flat H landing pads. Yet still a good attempt, as the network still recognized such pads in the vast majority of the frames.
In order to further improve performances for the detection of the H landing pads to reach the goodness achieved on the QR codes, different ways and approaches to create a synthetic dataset could still be explored, including more samples and also more various. Also new and different types of data augmentation techniques could be tried to furtherly improve the flexibility of the network for identifying more different types of H landing pads, or, in our case, the same type but subject to much more degrees of rotation, light conditions and perturbations. In addition to that, different regularization techniques could be also tried, i.e. technquiques aimed to enhance the generalization capabilites of the network by slightly modifying the network architecture and/or its training process. An exhaustive hyper-parameters search could be done as well, but that would require more processing power than the one freely (and rather limited) offered by Google Colab.
Another different point that could be still improved, is still the QR reading part, in a way it would work even if the camera doesn't directly face the QR code. For solving such an open issue, for instance, more classical solutions involving Computer Vision could be employed, like for example 3D straightening, provided that such algorithms would be not too much computational intensive.
Despite of these last points, the approach I developed and implemented worked nicely, recognizing the most of the targets even in presence of various types of perturbations, while staying also fully compatible with embedded system devices due to the contained computational cost.