JOUN TECHNOLOGIES

IoT Application in Agriculture: A Semi-Autonomous Drone

Hanna Saade

Abstract:

Improvements in communications and compute power have fueled the recent growth in the Internet of Things (IoT) network. This in turn has helped to enable the area of Artificial Intelligence (AI), as the increasing number of connected devices provide the needed data for learning and closed-loop deployment of feedback. New types of smart, connected devices are emerging, one such device is the Unmanned Aerial Vehicle (UAV) or drone. Drones have several interesting applications including delivery and surveillance, which can be utilized in several industries, including agriculture. Agricultural drones have become very useful for applications that vary from crop disease detection to irrigation, spraying, and reforestation. In this paper, we will look at the use of an agricultural drone as a component of an IoT system. When integrated with data provided from agricultural sensors in the network, a system of actuators coupled with the use of UAVs can greatly improve the farming yield by providing support to farmers and specialists while reducing human error.

Table of Contents:

Introduction and Objective **4**

A Suitable Drone **4**

The Old Drone 5
Market-Ready Drone 6

The Spraying System **7**

Hardware 7
Software 8

Machine Learning **9**

Introduction 9
Data Collection 11
Labeling 13
Model Training 15

Evaluation of Model Metrics **19**

Accuracy 19
Precision 20
Recall 20

Deployment and Inference **22**

Deployment 22
Data Communication 23
The Final Step: Inference 24

Conclusion **24**

References **25**

About the Author **27**

Introduction and Objective:

The global agriculture drone market size was over $ 1B in 2019 and is projected to reach almost $ 4B by 2027. [1]. UAVs can help to improve efficiency and yield in areas where the terrain is rugged and/or areas where access to advanced technology may have been lacking in the past. In this paper, we will describe an autonomous spraying quadcopter with capabilities of live video streaming, autonomous spraying using object detection, and manual drone flight control. We will describe the research, testing, design, practical coding, and hardware connections that were conducted to achieve a final prototype.

The original project goal was to construct a drone from scratch and then add in additional features such as autonomous olive tree detection. As the work commenced, it became clear very quickly that suitable drones existed with the necessary base functionality. This meant we could skip the complex details of flight control and communication and focus on our specific differentiated features, such as being able to successfully distinguish olive trees and then utilize a spraying system. Our initial focus was to find a drone that could lift heavy payloads and combine it with a Raspberry Pi board to control a water pump and valves. The programming on the board along with machine learning principles were utilized to autonomously spray the needed trees to save time and resources. [2]

A Suitable Drone:

Since market-ready UAVs can be expensive, we chose an older, pre-constructed drone to work with temporarily until we could verify successful results.

The Old Drone:

To build a reliable and fully functional system, getting familiar with the old drone was crucial to maintain and use it. The first step was to learn about its connections and construction. The drone was made up of four brushless motors (MT2213), each with 860 grams of maximum thrust. Their speeds were controlled by four Electronic Speed Controllers (ESCs), and four 8-inch propellers were installed on top of the motors to lift it off the ground. A 12-volt 5.8 GHz First Person View (FPV) transmitter and camera were used to transmit live video to an external 12-volt screen. The drone chassis was made up of four plastic arms held together by two power distribution boards where the ESCs were soldered to acquire power from the 12-volt, 2200-mAh Lithium Polymer **(**LIPO) battery. The flight controller board controlled the ESCs and was itself controlled by the Pulse Position Modulation (PPM) receiver which received 2.4 GHz signals from the Spektrum JR9503 9-channel transmitter.

As we inspected the drone, several problems were found. One of the four arms was broken, the receiver did not work, and there were a few faulty wires that needed fixing and replacement to prevent burning out the flight controller board. In addition, the board needed to be configured and the ESCs needed calibration. A new receiver that matched the old receiver's specifications was ordered, an arm was bought and replaced, wires were replaced, soldered, and sheathed, the ESCs were calibrated [3], and the drone was set up using "Betaflight" software. When the initial repairs were made, other problems surfaced. The 2200 mAh battery only lasted for a couple of minutes, and flying the drone was almost impossible because of the flight controller board's poor quality and the absence of stabilization features. This caused damage to one of the motors and all the

propellers. In addition, it was found that the motors could barely carry the weight of the drone, so it was going to be difficult to carry additional loads. [4] A picture showing the old drone is shown below.

Figure 1: Old Drone

Due to the numerous problems shown above, it was decided to split the hardware into two parts. First, design and implement a spraying system and second, choose a UAV from the market that matched the weight requirements for an integrated system.

Market-Ready Drone:

After conducting thorough research, it was decided to use a drone by Dronevolt called the Hercules20. The Hercules20 had the ability to accept our spraying system, was ruggedized and was easy to control using Raspberry Pi. The drone also met key specifications such as:

1. Able to spray vineyards at 1 to 3 hectares per hour and fields at 4 to 8 hectares per hour
2. Cover more area and reduce wasted resources by 30-50% with its 12L tank and adjustable nozzle size
3. A maximum take-off weight of 33kg
4. A maximum flight speed of 25 m/s which lasted up to 40 minutes

5. A remote control/video transmission range of 2 km.

The Spraying System:

This section describes the spraying system that was operating on our drone. It explains the materials used to build this system (hardware), and how they were programmed to work together simply and effectively (software).

Hardware:

The first step was constructing the spraying system structure and testing it to ensure it was functional. A board with sufficient processing power was needed to apply the highly sophisticated object detection and give it control over our system. Raspberry Pi turned out to be the most optimal in terms of speed and ease of use. [5] Since object detection was to be done after testing the spraying system, the detected olive tree was represented by a push button, and the needed parts such as a mechanical float switch, a buzzer, a pushbutton, Light Emitting-Diode (LEDs), resistors, a breadboard, a switch, a relay, and a twelve to five-volt converter were also acquired.

For testing purposes, the drone battery was used to power this system. It was connected to a switch that turned on or off three parallel circuits: the pump, the Raspberry Pi, and a circuit to power the external screen when needed to test the camera. Since the Pi needs 5 volts to function properly and may consume up to 2.5 A, a 3A twelve-to-five-volt buck converter was purchased to supply 5 volts from a 12-volt battery. Three signal output pins were used to power the LED, the buzzer, and the relay which controlled the pump contact. Two signal input pins were used to acknowledge the press of the pushbutton and the presence of water in the container by the mechanical switch. The following two figures illustrate the hardware implementation of the spraying system.

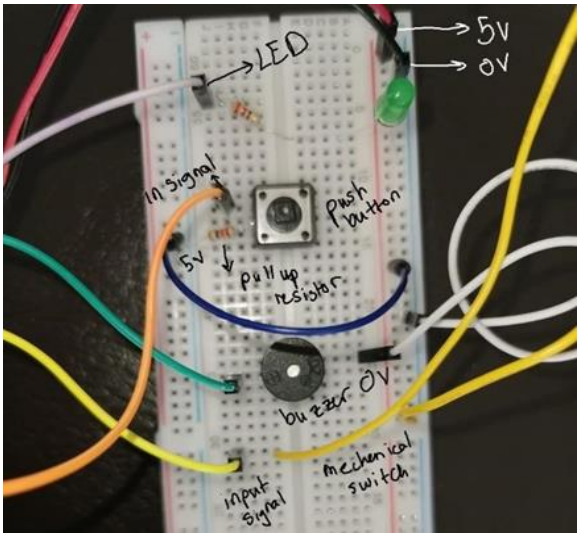

Figure 2: Breadboard

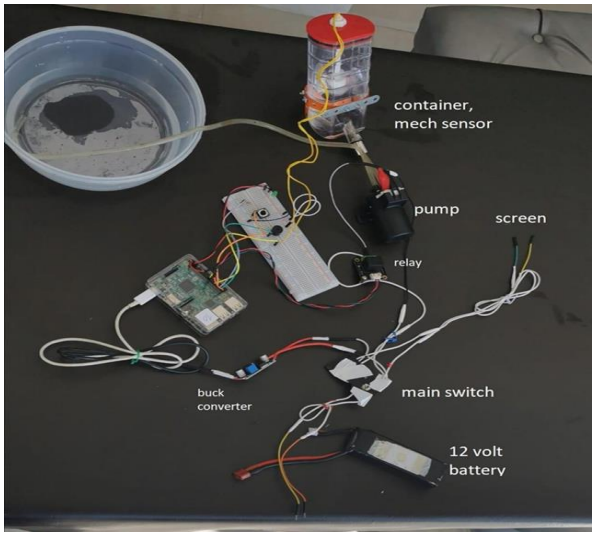

Figure 3: Spraying System

Software:

Programming General Purpose Input Output (GPIO) pins was done using Python 3 integrated development environment (IDE) on Raspberry Pi. By default, when it boots, the Pi does not run the Python code by itself like other boards present in the market. To run the code at startup, the "rc.local" file needed editing to show the Pi which file to run when it turns on. Some of the code used to run the spraying system is shown below.

```
while(1):                       #main code loop

  buttonpress=GPIO.input(pb)            #tske input from pushbutton
  #print(buttonpress)
  if(buttonpress==True):                #if button is pressed... representation for tree recognized in image detection

    waterlvl=GPIO.input(lvl_sens)
    if(waterlvl == 1):                  #if the cintainer is not empty
        GPIO.output(led, False)
        GPIO.output(pump, True)
        GPIO.output(buzzer, False)

    elif(waterlvl == 0):                #if the cintainer is empty
        GPIO.output(led, True)          #turn LED on
        GPIO.output(pump, False)
        if not 'event' in locals():
            event= GPIO.add_event_detect(lvl_sens, GPIO.FALLING, callback=level_low_callback,
                                         bouncetime=100)      #call the interrupt function which causes buzzing

  elif(buttonpress==False):             #if button not pressed hence not tree detected then turn everything off
    GPIO.output(buzzer, False)
    GPIO.output(led, False)
    GPIO.output(pump, False)
```

Figure 4: System Program Loop

While the fluid level was high, if an olive tree was detected, or the push button was pressed in this phase, the pump turns on to spray the olive tree. If the fluid level decreases to the minimum level, the pump stops spraying and a LED light with a buzzer turns on to indicate to the user that the tank needs refilling (the buzzer was only used for simulation; it was not implemented on the drone). After the spraying system was physically implemented, further research needed to be done in the field of machine learning to apply olive tree detection and make it control this system.

Machine Learning:

Introduction:

The building blocks of computer vision are CNNs (Convolutional Neural Networks). A CNN is a multi-layer neural network that is used in the field of machine learning, specifically deep learning and object detection. These networks are fed sub-images from many regions of interest in an image to pick certain patterns and identify certain objects. What makes a CNN special is that it has a convolutional layer which contains filters. You can think of a filter as a matrix whose output value tells you how likely a certain part of an image matches the same part of a candidate object. Rectified Linear Unit (ReLU) is the second layer in a CNN. It has a basic yet important task of normalizing the matrix values or limiting them between zero and one. The third layer is the pooling layer. It functions by taking a large image and shrinking it down while keeping its important information intact. For example, if we choose a 2 by 2 pooling, the pixel values of an image are organized into 2 by 2 matrices, and the largest pixel value of the four is chosen while the others are discarded. These three layers are stacked on top of each other, and each image is fed to those layers, along with other classification layers, to come up with an output which is the decision of what the object might be [6]. Real-time object detection feeds frames of a live video feed to such

a neural network and the objects in each frame are detected. CNNs have had many developments over the years. CNNs became R-CNNs (Region-based CNNs), Fast R-CNNs and Faster R-CNNs, and then the YOLO (You Only Look Once) algorithm emerged which uses the previous networks along with other layers to provide even better object detection.

A R-CNN is better than a conventional CNN because instead of selecting a huge number of regions, we select only two thousand regions. Each region is fed into the CNN algorithm to extract features from them. The main drawback of R-CNN is that it takes approximately 47 seconds to identify each image, making it not very suitable for real-time applications. To fix this problem, fast R-CNN feeds the whole image to the CNN and comes up with a convolutional feature map. After that, we identify the regions of interest from this feature map. Instead of selecting regions and feeding them to the neural network (a time-consuming process), a faster R-CNN provides the image as an input to the CNN which comes up with the convolutional feature map. Instead of selecting regions, a separate network is used to predict the regions of interest [5].

All the above algorithms use regions to identify an object within an image. In YOLO, a 53-layer convolutional network is used to predict bounding boxes of an image. A YOLO algorithm takes the input image and splits it into an “SxS” grid. Within each grid, it predicts “m” bounding boxes, and the network outputs a class probability for every bounding box. When a bounding box has a class probability that is above a certain value, it is selected to locate the object of interest within the image. The drawback of YOLO is that it faces minor difficulties in detecting small objects, which was not an issue for our project. [7].

There are three types of machine learning: supervised learning, unsupervised learning, and reinforcement learning. For our project, we were interested in supervised machine learning where

the machine learning model was taught algorithms through multiple iterations. It was given images to map to specific output names, then checked its results based on the labels present for each image. After thousands of iterations, it fixed its judgment based on the true labels vs. the prediction and learned the map of mathematical formulas that relate the input image to the output name. Initially, we must collect images to feed them to the neural network and the images need to be labeled. Next, we need to train our algorithm based on this labeled data, check the training results, and deploy our trained model to the Raspberry Pi to make inferences. The following figure illustrates the supervised learning pipeline.

Figure 5: Supervised Learning Pipeline

After getting familiar with the manner of operation of some image detection algorithms, it was time to start the machine learning process. The next section goes through data collection and processing.

Data Collection:

A dataset of 509 images was created. These images were mostly of olive trees (400 images) since this was the object of interest for detection. The remaining photos were those of the many kinds of different trees that may be also found in olive tree fields. This relatively small sample of different trees is essential for the training process because it gives the model some data to compare

to the olive trees. This comparison ensures that the output of the training will be accurate, resulting in accurate image classification.

To be certain that the image detection will be exact, olive tree data collection was done in a way to include all kinds of conditions and situations that might occur. To begin with, images were taken of different kinds of olive trees. We made sure that we captured images of trees that differed in size. The images needed to be taken aerially from different heights (since the flying drone's camera can only photograph what is under it). This was a challenge when collecting images of large trees, so some of the photos were taken from balconies and for others we used ladders. Olive trees do not only vary in size, but they also vary in the thickness of branches and the number of leaves on each branch. This was also taken into consideration when collecting data by taking images of trees with almost no, few, and many leaves. Even the size and kind of leaves might differ between trees. They can differ slightly in color or have larger/smaller sizes. This difference usually occurs across different seasons. For that reason, images were collected in different seasons (mainly fall and winter), and the shots were taken from various random angles to ensure the complexity of our dataset.

Significant attention was paid to the fact that every single tree might have a different background. Some trees stand directly on soil and others have grass beneath them. Also, since some of the photos were taken in the winter, a few trees had snow below them. In addition to that, pictures were taken during different times of the day, where some of the trees were either hit by direct sunlight or shaded. Images were also taken under a variety of weather conditions.

All those details were taken into consideration to ensure a robust data set for image detection. Since the training was done on such a dataset, we had confidence that any kind of olive

tree, in any season, at any time of the day, and from whatever position it is seen from, would be detected accurately by the Raspberry Pi. The data collection procedure was by far the most time-consuming part of the project.

Labeling:

The object detection model cannot learn without proper labels for the olive tree dataset since, in each training iteration, it needs image labels to compare them to its own predictions in order to adjust for the error in prediction. For this purpose, a widely known software tool called "LabelImg" was used. "LabelImg" is a free, open-source tool for graphically labeling images. It is an easy, free way to label a few hundred images, which is just what we needed for our project.

Labeled images are saved as file formats. The tool we used backs labeling in either YOLO text file format or VOC XML. VOC XML is usually the default format that is widely recommended, but in this case, the YOLO text file format was used since YOLO network image detection was being used. Each image was labeled manually by placing a rectangular box around the object of interest in the image and giving it a name. In our case, olive trees were named as "olive tree" and others were only named as "tree". The following figure demonstrates how labeling an olive tree in an image is done.

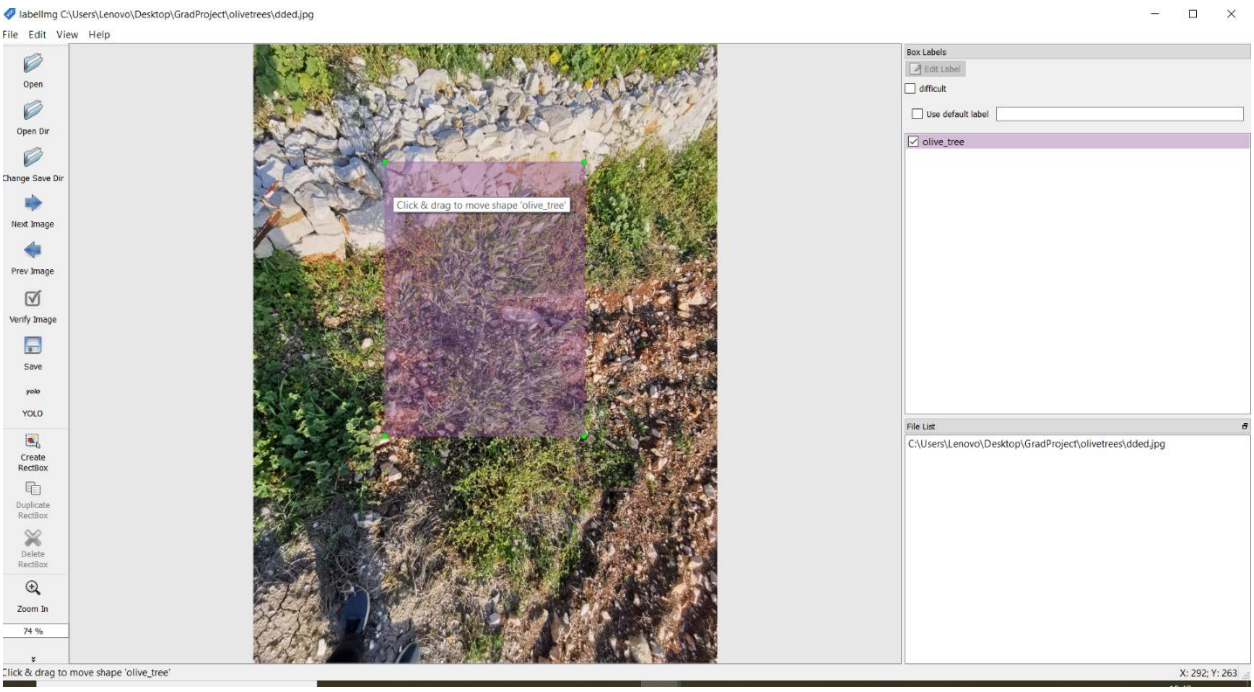

Figure 6: Labeling an Image

After saving each labeled image, a “txt” file was created next to the image in the same directory (after changing the save directory). The text file contained five numbers of significance. The numbers in order (left to right) represented the class, center x (of the object), center y, the width (of the object), and the height. Center x, center y, width, and height are all decimal numbers since the image was considered a two-dimensional array of pixels, with each pixel value ranging from 0 to 1. What follows is an example of one of the text files:

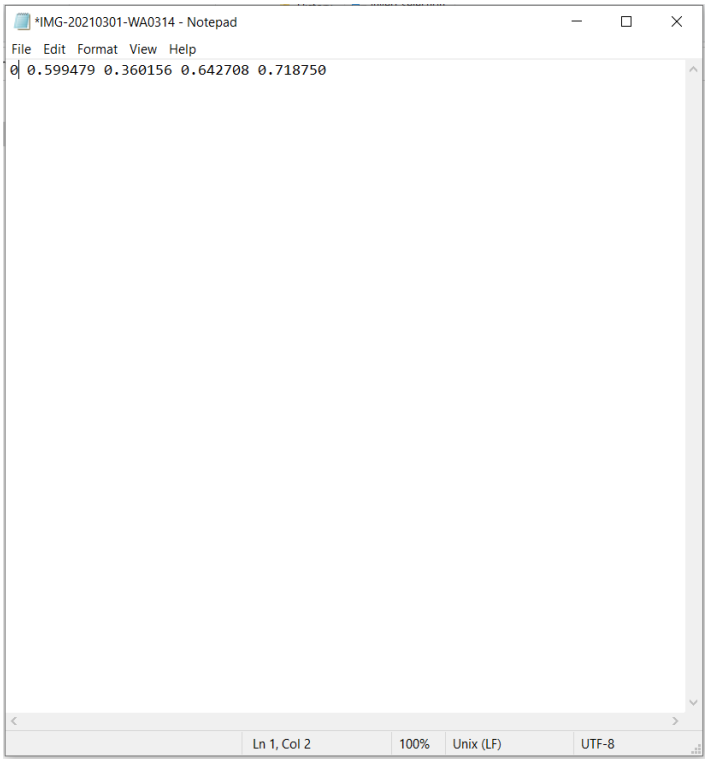

Figure 7: Txt File

After the labeling process was done, the machine learning model needed to be trained based on the data. All the images and corresponding txt files were exported to a GitHub online repository. You can see this repository at https://github.com/ZaZ-PaZaZ/GradImages/tree/master/MyImages.

Model Training:

A small portion of the collected data was used for validation. The validation dataset was used to check that the model had learned up to a certain point in the training process. At each iteration, the model was exposed to previously unseen data. This provided validation accuracy which was more reliable.

Training a model requires large amounts of processing power. For this reason, we chose to perform the training step on Google Collaboratory, which includes a notebook for Python code writing and a free Graphics Processing Unit (GPU) to build the algorithm upon. Training a new model from scratch also requires a huge dataset and a huge number of back-propagation iterations to reach sufficient accuracy. Since we did not have such a dataset, and to save collection and training time, it was decided to use a variant of the YOLOV3 object detection model. It was pre-trained in the Darknet-19 framework and used transfer learning to retrain only its top layers and make the algorithm fit for olive tree detection. YOLOV3-tiny was the choice because the deployment was on a Raspberry Pi which is very limited in terms of Random-Access Memory (RAM), flash memory, processing power, and latency. It is a smaller version of its YOLO parent, made up of a small 19-layer neural network instead of a 53-layer one. It was created to make it easier for the YOLO algorithm to run on devices with small computing power. However, the tiny YOLO is less accurate than YOLO since it has a fewer number of layers and utilizes Quantization Aware Training (QAT), post-training quantization, and neuron pruning. These techniques change more accurate floating-point numbers to less accurate integers, which require less computational power and less flash memory. The quantization technique is illustrated next.

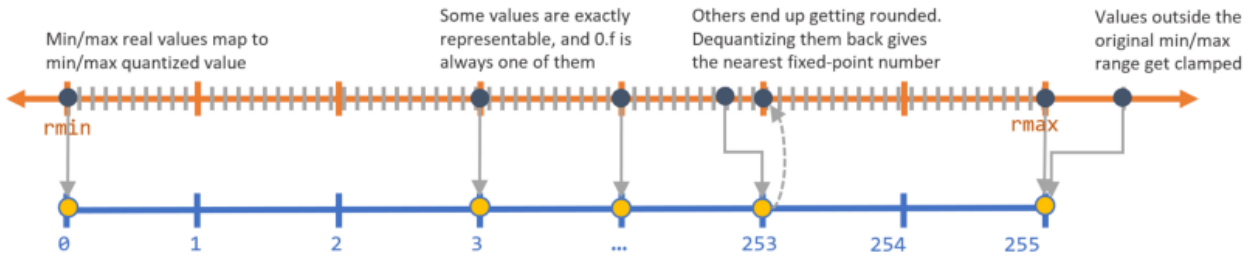

Figure 8: Floating-point to 8-Bit Integer Quantization [12]

By sacrificing a small amount of accuracy, relatively acceptable Frames Per Second (FPS) rates may be reached despite smaller memory and RAM availability using state-of-the-art techniques. For example, the YOLOv3 model size is 243 megabytes and the number of parameters it uses for computation is approximately 64 million. In contrast, the YOLOv3-tiny model size is only 33 megabytes and uses only about 9 million parameters [8]. The following figure presents the small network architecture with some convolutional, pooling, and classification layers we talked about earlier.

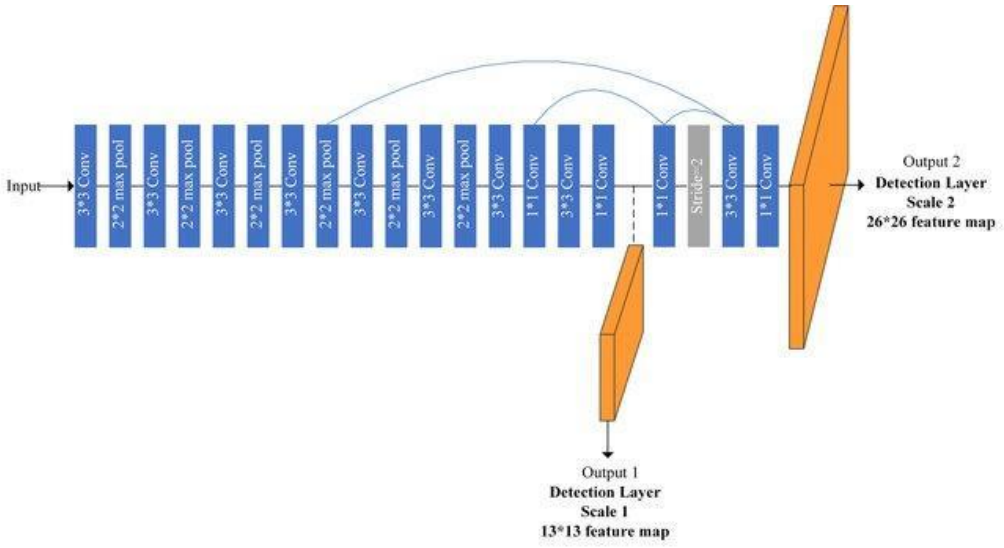

Figure 9: YOLOv3-tiny Network Architecture [13]

YOLOV3-tiny is pre-trained on the COCO dataset, a dataset of 328,000 images with a total of 2.5 million labeled instances of 91 different object classes [9]. Collecting such a huge number of images and labeling them is a hectic process. Alternatively, only 509 images were collected and image augmentations such as flipping, rotating, and brightness modification were applied to the training dataset to train this model to differentiate between an olive tree, an ordinary tree, and all

the other classes. Building upon the pre-trained model makes the training process more lightweight, taking approximately 4-5 hours on Google Collaboratory.

To train an object detection model, frameworks like Tensorflow and Darknet needed to be imported, and a "GitHub" blueprint repository for the pre-trained algorithm needed to be cloned and modified according to criteria specific for the project's application. This included the class names ("olive tree" and "tree") and the number of classes. Then, the "GitHub" repository of labeled images was also copied to the Collaboratory notebook. Next, different parameters required mitigation before training which included the batch size, the image resolution, and the number of iterations of training. The batch size generally remained at sixty-four for YOLOV3-tiny, and it denoted the number of images that the algorithm went over in each iteration. The width and height of the image were chosen as 224 pixels because the resolution of the webcam installed on the drone was 224 pixels by 224 pixels. The number of steps needed to be a minimum of two thousand for each class. However, through testing, it was found that the optimal number of steps for the two classes, in terms of accuracy, was twelve thousand. All these steps were done to make the model ready for training. After that, training commenced.

During training, for each one thousand iterations, a ".weights" file was generated, and the algorithm saved the best file generated in a specific location based on the training and validation accuracy in each iteration. The neural network learns through its neurons and each neuron has a set of mathematical parameters: weights and biases. These were included in the file. After the training was done, a ".cfg" binary file was generated. It contained the model structure for each mathematical parameter and equations to be done between the different weights and biases. Both files together made up the whole algorithm. After going through all these steps, the final validation

accuracy of 96.63% was reached based on a 0.25 confidence threshold. The result was excellent but still needed further testing to acquire a real accuracy score based on the test dataset and set a confidence threshold based on the requirements of our project [10].

The following link contains our Google Collaboratory notebook which illustrates how our training was done. (Olive tree Detector.ipynb - Colaboratory (google.com)).

Evaluation of Model Metrics:

Before reflecting on the results obtained after the training step (and using the test dataset), we will go over some metrics, namely accuracy, precision, and recall.

Accuracy:

Accuracy is the percentage of the correct positive predictions that the model makes. In other words, accuracy is the number of correct positive predictions divided by the total number of predictions.

$$Accuracy = \frac{true\ positives}{total\ number\ of\ predictions}.$$

True positives denote the images which contain olive trees and are correctly predicted as olive trees. Correct predictions include true positives and true negatives, while the total number of predictions are the correct predictions plus the number of false positives and false negatives. True negatives occur when the model does not detect an olive tree when it should not, false negatives occur when the model classifies an olive tree as either a non-olive tree or another object, and false positives take place when the model classifies a non-olive tree or another object as an olive tree [11].

Precision:

Precision is how precise the model is in detecting positive values. We can calculate it using the following formula:

$$Precision = \frac{true\ positives}{true\ positives\ +\ false\ positives}$$

When the cost of false positives is high, the higher the precision value the better [11].

Recall:

Recall refers to the number of actual positives that were identified correctly. The formula is:

$$Recall = \frac{true\ positives}{true\ positives\ +\ false\ negatives}$$

Recall measures how much our model was able to correctly identify true positives. When the cost of false negatives is high, the higher the recall value the better [11].

To correctly evaluate model metrics, it was tested on both olive and non-olive trees. New pictures of a total of 22 olive trees and 22 non-olive trees were taken. Three pictures for each tree were taken at three different heights of 80, 150, and 200 centimeters, with a total of 66 pictures of olive trees and 66 of non-olive trees. A dataset of 132 images was collected. Such a test dataset is large enough to allow inference of reliable results on the trained model.

The dataset was fed to the trained model for testing, and confidence rates for each object in each image were generated. Depending on a certain threshold, the values of accuracy, precision, and recall were obtained. To select an optimal threshold value, it was crucial to experiment with

different thresholds. For this project, three values were chosen: 0.4, 0.55, and 0.7, as shown in the figure below.

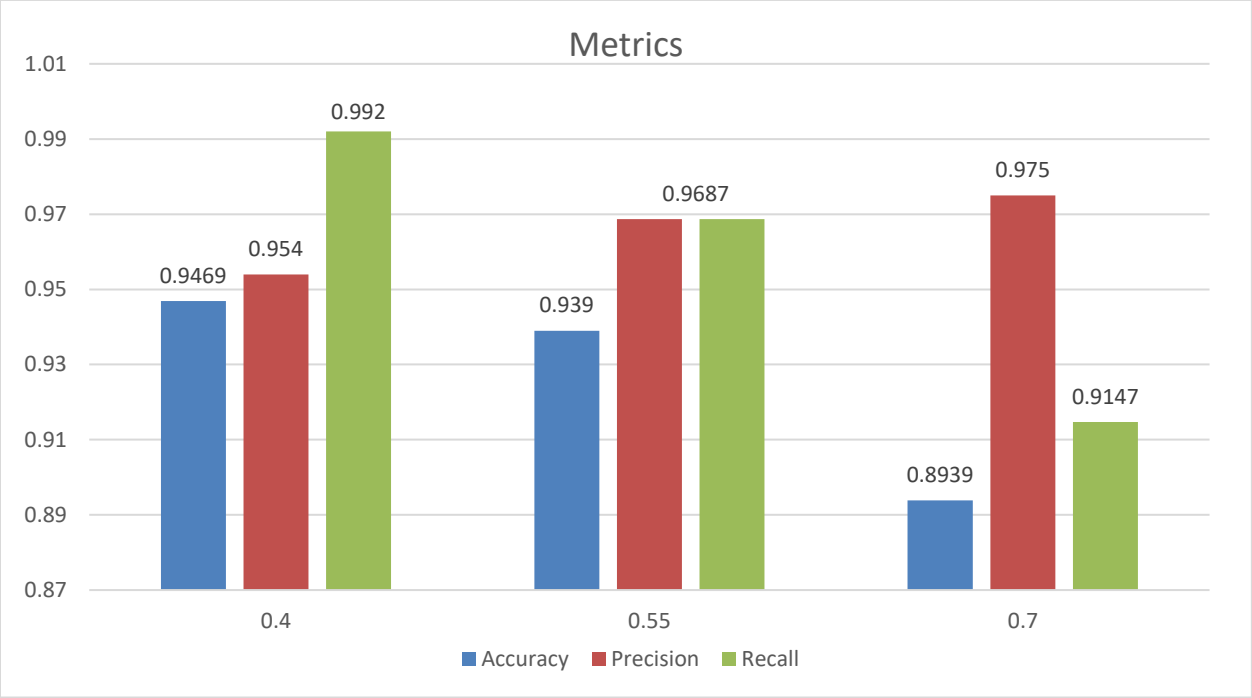

Figure 10: Accuracy, Precision, and Recall for different threshold values

A high precision value is required in order not to waste resources while maintaining a high recall to enhance the user's experience, avoid draining the battery, and skipping trees. Although a threshold of 0.4 resulted in the highest accuracy and recall, it gave the lowest precision. Since the cost of a low precision was higher than the cost of a low recall, and since a low recall is also costly, the threshold of 0.55 was chosen as the best option due to the balance it provides between precision and recall while maintaining high metrics: 93.9% test accuracy and 96.9% test precision and recall.

The credibility of the testing phase is illustrated clearly in the pictures below. As seen, a bush of very similar characteristics to an olive tree was detected as a tree, and an olive tree with a

very green background and many branches covering it in the foreground was accurately detected as an olive tree. The first picture illustrates the superiority of machines over man in detecting patterns; even if an experienced farmer looks at such a picture, they will probably misinterpret it for an olive tree. That is the importance of autonomous spraying! If one had to fly the drone manually and spray manually, many olive trees, such as the one seen in the second picture, would be kept unsprayed, and many other trees of a similar appearance to olive trees would be sprayed. This is a waste of liquids and a decrease in yield and health for the unsprayed ones.

Figure 11: Detection of a non-olive tree

Figure 12: Detection of an olive tree

Deployment and Inference:

Deployment:

After the training was done, deploying the model to work on Raspberry Pi became the next step. First, different modules needed to be installed to the Pi to implement object detection and activate OpenCV. OpenCV (Open Computer Vision) is a library of functions that can be used with Python, which helps in the process of real-time computer vision. Its functions contain reading

images, showing, and writing or saving them. It also provides help with reading video frames and feeding them to neural networks along with drawing shapes and text on each of the frames. This was important for us to display the bounding boxes and confidence rates of each object in each of the frames, and hence generate a real-time annotated video.

The ".weights" and ".cfg" files were downloaded from the Collaboratory notebook and copied to the Raspberry Pi where inference was run. Next, a detector script from GitHub was cloned. This script was split into two: detector.py and models.py. Models.py contained the paths to those files, and detector.py was the python file that loaded the weights file and the configuration file from the "models.py" file to the network to build a machine learning model. In addition, it took live video from the webcam, configured each frame of this video, fed it to the model, and extracted the output labels and confidence scores to present them on the video in real time.

Data Communication:

After detecting objects from each frame of the live video captured by the camera using the YOLOv3 algorithm, the annotated frames had to be transmitted from the drone to either an external screen or a mobile phone to see the object detection results. For this to be done, the drone was equipped with another 5.8Ghz transmitter, a High-Definition Multimedia Interface (HDMI) transmitter, that was connected to the Raspberry Pi's HDMI output port. When the Raspberry Pi was turned on, it automatically started the detection process where a window showing live video was presented along with other details. The Pi's screen was sent through the transmitter directly, and a compatible receiver received the video stream. This kind of receiver may be connected to any LCD screen, or even to a mobile phone or PC (through an OTG cable). Thus, the person operating the drone was able to view the live annotated video from a first person's view.

The Final Step: Inference:

Finally, after recognizing an olive tree, the spraying system had to be activated, and then turned off when there were no olive trees in the camera's line of sight (under the drone, towards the ground). To achieve this, the code previously presented was integrated in the detector.py script with each olive tree detection. After testing this integration on the hardware, everything (surprisingly!) worked as expected. A video stream of many olive trees, non-olive trees, and other bushes and landmarks was captured, and fed to the detector. When an olive tree was detected, spraying was activated until the fluid reservoir was emptied. When its level reached the minimum, a LED light turned on and the Raspberry displayed a message that indicated it needed refilling. However, when there was no olive tree present in the camera's line of sight, the spraying was turned off, and the model waited for another olive tree to appear.

Conclusion:

Drones are being utilized in ways that were not thought to be possible a decade ago. One of those ways, which was demonstrated with this project, is in agriculture. With the rapid development of technology and specifically machine learning, we look forward to further enhancing our drone. Using quantization and neuron pruning techniques to achieve a higher frame rate and lower inference time, we expect to optimize the detection model even further. Another objective is to make the drones fully autonomous and part of a smart agriculture system. Sensors in a field may provide live data to the drone to do specific functions. The drones may also serve multiple functions, such as plant disease detection, firefighting, harvesting and pruning and more.

References:

[1] *Agriculture Drone Market Size, Share | Industry Analysis, 2027*. (n.d.). Fortine Buisness Insights. Retrieved May 26, 2021, from https://www.fortunebusinessinsights.com/agriculture-drones-market-102589

[2] Vihari, M. M., Nelakuditi, U. R., & Teja, M. P. (2018). IoT based Unmanned Aerial Vehicle system for Agriculture applications. *2018 International Conference on Smart Systems and Inventive Technology (ICSSIT)*. Published. https://doi.org/10.1109/icssit.2018.8748794

[3] ArduPilot Dev Team (2020). Electronic Speed Controller (ESC) Calibration. ArduPilot. https://ardupilot.org/copter/docs/esc-calibration.html

[4] Find the Perfect Motor to fit your drone. Tom's guide. (November, 2014). https://forums.tomsguide.com/faq/find-the-perfect-motor-to-fit-your-drone.26313

[5] Richardson, M., & Wallace, S. (2012). Getting Started with Raspberry Pi. Media, Inc.

[6] Gandhi, R. (2018, December 3). R-CNN, Fast R-CNN, Faster R-CNN, YOLO — Object Detection Algorithms. Medium. https://towardsdatascience.com/r-cnn-fast-r-cnn-faster-r-cnn-yolo-object-detection-algorithms-36d53571365e

[7] Redmon, J., Divalla, S., Girshick, R., Farhadi, A.You Only Look Once, real time object detection. University of Washington https://arxiv.org/pdf/1506.02640v5.pdf

[8] Zhang, P., Zhong, Y., & Li, X. (2019, July 25). SlimYOLOv3: Narrower, Faster and Better for Real-Time UAV Applications. Y. Zhong. https://arxiv.org/abs/1907.11093

[9] Lin, T., Maire, M., Belongie, S., Bourdev, L., Girshick, R., & Hays, J. et al. (2021). Microsoft COCO: Common Objects in Context. arXiv.org. https://arxiv.org/abs/1405.0312

[10] AlexeyAB/darknet. GitHub. (2021). https://github.com/AlexeyAB/darknet#how-to-train-to-detect-your-custom-objects.

[11] Shung, K. P. (2020, April 10). Accuracy, Precision, Recall or F1? - Towards Data Science. Medium. https://towardsdatascience.com/accuracy-precision-recall-or-f1- 331fb37c5cb9

About the Author:

Hanna Saade

Hanna Saade is finishing his third year of Mechatronics Engineering at the University of Balamand – Koura Campus. His interests include Mechatronics, IoT, Deep Learning and various areas of software programming.