Smartphone-based object recognition with embedded machine learning intelligence for unmanned aerial vehicles

Martinez-Alpiste, Ignacio; Casaseca-de-la-Higuera, Pablo; Alcaraz-Calero, Jose M.; Grecos, Christos; Wang, Qi

Published in:
Journal of Field Robotics

DOI:
10.1002/rob.21921

Published: 30/04/2020

Document Version
Peer reviewed version

Smartphone-Based Object Recognition with Embedded Machine Learning Intelligence for Unmanned Aerial Vehicles

Ignacio Martinez-Alpiste* , Pablo Casaseca-de-la-Higuera** , Jose M. Alcaraz-Calero*, Christos Grecos***, Qi Wang*

*University of the West of Scotland, UK
*Universidad de Valladolid, Spain
*National College of Ireland, Ireland

Abstract

Existing artificial intelligence solutions typically operate in powerful platforms with high computational resources availability. However, a growing number of emerging use cases such as those based on Unmanned Aerial Systems (UAS) require new solutions with embedded artificial intelligence on a highly mobile platform. This paper proposes an innovative UAS that explores machine learning (ML) capabilities in a smartphone-based mobile platform for object detection and recognition applications. A new system framework tailored to this challenging use case is designed with a customised workflow specified. Furthermore, the design of the embedded ML leverages TensorFlow, a cutting-edge open source ML framework. The prototype of the system integrates all the architectural components in a fully functional system, and it is suitable for real-world operational environments such as seek and rescue use cases. Experimental results validate the design and prototyping of the system and demonstrate an overall improved performance compared with the state of the art in terms of a wide range of metrics.

Keywords:

Machine learning; object detection and recognition; Unmanned Aerial Vehicle (UAV); image processing; smartphone

1 Introduction

Unmanned Aerial Systems (UAS) have gained increasing global popularity in recent years in a wide range of use cases such as emergency response, disaster relief and humanitarian aid, health-care, agriculture etc. Moreover, there is a growing demand for smart Unmanned Aerial Vehicles (UAV) to fulfil challenging, mission-critical tasks with improved performance. Artificial intelligence is expected to play an important role in such smart UAS with significantly enhanced capabilities beyond conventional systems. However, machine learning (ML) based solutions, especially those for real-time computation-demanding processing tasks, typically require high-performance platforms such as servers or high-end desktop computers with capable graphics processing units (GPUs). This approach is clearly not suitable for UAS use cases with high mobility requirements. Therefore, new solutions are entailed to enable embedded ML capabilities in mobile platforms as an integral part of smart UAS.

There are two main approaches to exploring the porting of machine learning intelligence to an UAS. The first approach is to build the intelligence directly onboard the aircraft in the UAV, whilst the other approach is to deploy the intelligence off-board. There are advantages and disadvantages in either approach. The former approach yields a more powerful aircraft with strengthened onboard intelligence to localise and thus accelerate the processing of the captured data; however, such onboard intelligence is only applicable to a limited number of UAV models with an onboard software development kit (SDK). In addition, this solution has higher energy consumption, leading to fast battery drainage if the UAV is not fuel-powered which is one of the main limiting factors for operations nowadays. In contrast, the latter benefits from a more flexible and much more widely applicable solution at the cost of requiring a ground

programmable mobile platform. The selection of the approach largely depends on the specific deployment requirements taking into account the existing UAV models.

Furthermore, to implement and deploy ML algorithms on an UAV, the selection of an enabling framework combining both ease of development and computational efficiency is of paramount importance.

A number of existing ML frameworks allow easy deployment of ML (particularly deep learning) models to incorporate the needed artificial intelligence to the UAV. Among them, Tensorflow, Caffe, Keras, and OpenCV are becoming increasingly popular. Each of the frameworks has its own strengths and weaknesses, which should be considered for potential adoption to an UAV. Thus, a comparison to ensure an informed appropriate decision might need to be done for some specific applications. More importantly, once an ML framework is selected, the different underpinning ML algorithms should be evaluated in the target system so that the best overall performer can then be employed by the system. In addition, further research and development may be necessary to adapt the selected ML techniques to the target system or the operation environment, or optimise the system-level performance in such a complex system that integrates ML, computer vision, video/imagery communications, onboard or off-board processing for particular tasks and other cutting-edge technologies.

The contribution of this paper is multifold:

- A new integrated, practical UAS architecture is designed and prototyped, achieving off-board ML-based object recognition embedded in a smartphone.

- Three different ML object detection algorithms (SSD, Yolov3, Tiny-Yolo3) have been integrated in the UAS architecture optimised for improved performance in the UAS for object detection/recognition use cases.
- Two Different ML frameworks (OpenCV and TensorFlow) has been integrated in the UAS architecture in order to allow a benchmarking testbed, thereby producing comparison performance evaluation of different neural networks in deep learning and platforms.
- A leading-edge neural network architecture has been adapted and optimised for improved performance in the UAS for object detection/ recognition use cases.
- A comprehensive benchmark testbed has been carried out considering several dimensions such as detection speed, model size, pre-processing time, ram consumption, battery usage and accuracy in object detection.
- In addition, the paper demonstrates a methodology in selecting the right technologies from a large number of available machine learning platforms and algorithms.

The rest of this paper is organised as follows. Section 2 reviews related work. Section 3 presents the design of the proposed ML-enabled UAS. Section 4 describes the implementation details. Sections 5 and 6 respectively present empirical results and a comparison analysis based on experiments in performance evaluation, together with a discussion of those results. Section 7 concludes the paper.

2 Related Work

2.1 Object Detection and Convolutional Neural Networks

Over the last years, research and development in object detection have resorted to proposals based on Convolutional Neural Networks (CNNs), with other techniques based on hand-crafted features feeding Support Vector Machines (SVM) or other type of classifiers phasing out. CNNs have taken advantage over other approaches due to their effective ability to learn informative features from the data. These improvements lead to an enhancement of the accuracy performance, which is a decisive factor in use cases such as surveillance systems (Ko, 2018), cattle control (Kellenberger, 2017) and search and rescue operations (Bejiga, 2016).

The use of CNN entails the creation of a complex neuronal structured. The more complex the structure is in terms of numbers of neurons in each layer and in terms of the number of layers, the more accuracy will be achieved by the algorithms, but in contrast, the more computational complexity is required, this in turn, has a direct effect in both processing speed and power consumption.

The way that these neurons are structured is traditionally defined in a configuration file. This file also establishes the assignment for each layer, how the input layer must accept data and how it is formatted in the output layer. A weights file defining the configuration of the different convolutional layers usually accompanies such other configuration file.

Weights are defined after a (time- and resource-consuming) training process. Depending on the convolutional layer that processes the data, different types of features are automatically extracted. At the first layers, simple low-level features such as dots, lines and edges are detected.

In the following layers, more complex features are extracted resulting from the combination of the simpler, previous ones.

This research deploys three CNN-based object detectors including Single Shot Detector, YOLOv3 and Tiny-YOLOv3. These constitute state-of-the-art detectors with an acceptable trade-off between an accuracy and efficiency that enables their deployment on an UAV.

Single Shot Detector (SSD) (Liu, 2016) features a feed-forward convolutional neural network that provides as a result a selection of bounding boxes with a confidence coefficient of the objects detected in an image. At the beginning of the process VGG16 (Chung, 2018) is used in the backbone providing a set of likely objects. Once the initial detection is performed and after going through the convolutional layers, a non-maximum suppression technique is applied to discard most bounding boxes. Experiments with SSD over the COCO Dataset (see Subsection 2.3) yielded 41.2 mAP (mean Average Precision) of accuracy at 46 FPS (Frames Per Second) with an NVIDIA Titan X GPU (Redmon, YOLOv3: An Incremental Improvement, 2018).

YOLOv3 (Redmon, YOLOv3: An Incremental Improvement, 2018) continues its previous implementation of YOLO9000 (Redmon, YOLO9000: Better, Faster, Stronger, 2016) based on bounding boxes predictions. In addition, the authors have removed the softmax function, just leaving independent logistic classifiers. This adjustment has led to the management of overlapping labels. From previous versions, a valuable improvement implemented is the ability to detect objects at three different scales, allowing the developer to choose the range of size of the objects to detect depending on the output layer that he/she selects. By adjusting the size of an image's strides, YOLOv3 detects large, medium or small objects correspondingly. In this paper,

these sizes correspond to the output layers, which are enumerated as 0, 1 and 2, respectively. These sizes are directly connected to the anchor boxes values, which have been previously calculated before training the model. The sizes are given by downscaling the dimensions of the input accordingly to the stride. The smaller the stride, the tinier the objects that can be detected and, thus, the smaller the anchor boxes. In addition, as a feature extractor, YOLOv3 uses a modified version of Darknet named Darknet-53. YOLOv3 showed a performance of 55.3 mAP and a speed of 35 FPS with an NVIDIA Titan X (Redmon, YOLOv3: An Incremental Improvement, 2018)

Tiny-YOLOv3 is a simplified configuration of YOLOv3. This simplification is based on a reduction of the number of hidden layers. Tiny-YOLOv3 is suitable for constrained environments where its low resources consumption leads to an increase in speed. Nevertheless, this reduction of consumption also leads to a decrease in the accuracy of the neural network which is just able to predict objects at two scales. Tiny-YOLOv3 has a performance of 33.1 mAP and 220 FPS in an NVIDIA Titan X (Redmon, YOLOv3: An Incremental Improvement, 2018).

The above-described object detectors are deployed, executed and evaluated over this work in order to obtain the best results in a constrained mobile platform for UAS use cases.

2.2 Machine Learning Platforms

There are two implementation options for these CNN-based object detectors, either in native code or by using a specific platform. Usually, the challenging task of implementing the object detector in native code could lead to good performance although it could also result in laborious maintenance and give raise to difficulties in adaptations with other software. Therefore, the utilisation of enabling platforms seems a plausible option. There is a number of available

platforms such as Keras (Francois, n.d.), Caffe (Karayev, 2014), OpenCV (Bradski, 2000) or TensorFlow (Martin Abadi). All of them have their advantages and counterparts. Keras for instance, is preferable for prototyping due to its ease of use and the fact that it can run over closer-to-hardware libraries such as TensorFlow. However, it can be overloading for real-time applications. TensorFlow is a mathematical framework that allows an efficient use of the GPU and is increasingly being used for deep learning developments. Caffe provides similar functionality to TensorFlow, and Caffe2 is well-suited to mobile deployment. However, the higher popularity of Tensorflow among the research community makes it a better candidate for fast deployment. OpenCV is a well-established computer vision library that incorporates machine learning algorithms for object tracking and other visualisation features. We have chosen TensorFlow and OpenCV due to their popularity and maturity in addition to the fact that the trade-off between performance and ease to deploy is positive.

TensorFlow is an open-source platform that allows a developer to deploy algorithms with high-level APIs. This is probably one of the key factors to select TensorFlow over other machine learning platforms. These APIs are written in several programming languages, making easier the development of artificial intelligence algorithms. Meanwhile, TensorFlow is not a mature platform yet although it is being consolidated over the time.

Focusing on our use case with convolutional neural networks, TensorFlow just accepts as input both neural network configuration and its weights supported in different programming languages such as Java, Python and C++.

OpenCV has also been considered in this work. OpenCV is a computer vision library free for academic and commercial use. Similar to TensorFlow, it is supported in different operating systems and by different programming languages. In the beginning, it was created for computer vision although it has also been developed through machine learning fields. As a difference compared with TensorFlow, this platform is more consolidated. Its strengths are based on how the library works close to real-time with images and videos.

2.3 COCO Dataset

Microsoft COCO Dataset (Lin, 2014) has a total of 2.5 million labelled instances of 91 common objects easily recognisable by every human. In addition, it is a very popular training and testing set for object recognition in order to obtain the best results possible. In (Lin, 2014), it is compared against other datasets such as PASCAL, which has twenty categories that coincide with some of COCO categories. Nevertheless, COCO has more instances per category than PASCAL. As an example, while PASCAL has more than 30,000 instances for the “person” category, COCO has more than 500,000 instances. This difference in the number of categories allows a more complete dataset with different features to enable a better learning process for the object recognition algorithm.

The different object detection algorithms deployed in this work have been trained, tested and validated with the COCO dataset. The training set has 80 different objects to detect, more than 200K images with 3 instances per image on average. The training dataset is formed by 50% of the total dataset and the testing and validating datasets are formed by 25% of the images each.

2.4 Comparison analysis of use cases in the literature

AV expect to serve a wide range of use cases of considerable impact for the society such as autonomous 5G vehicles (Panwar, 2016) including its assistance in congestion (Wang X. a., 2019), and healing 5G networks (Sharma, 2018) by a correct positioning in order to increase coverage. This subsection analyses different use cases found in the literature for two different angles. Firstly, object detection for UAV use cases and secondly, neural networks for object detection in android mobile devices. This division of the study is motivated by the few previous studies specifically related to object detection in UAV linked to portable devices.

2.4.1 UAV Analysis

A literature review has been realised in order to create a comparison table of published results related to the topic that this paper addresses. Table 1 shows a comparison among various papers where object detection algorithms have been applied to the UAS' field. As shown in the table, a significant number of objects to detect contributions rely on execution environments that are not portable (computers and powerful graphics cards are usually used). Also, those that decided to run the experiment in a portable resource constrained execution environment rely on classical computer vision methods and do not use CNN-based object detectors. Typically, Haar-like (Aguilar, 2017), (Rudol, 2018) and SVM techniques (Bejiga, 2016), (Zhou, 2016) are utilised due to their fast performance and easy implementation. Nevertheless, the higher accuracy of CNN-based object detectors comes at the expense of higher computational resources compared to classical methods.

Table 1: Comparison analysis of previous UAV use cases founded in the literature

Aspects							
Ref	**Objective**	**Algorithm**	**Exec Environment**	**Platform**	**Accuracy**	**FPS**	**Model Size**
(Xu, 2017)	Car & Roads	Viola-Jones	PC	OpenCV	82.17%	0.94	NG
(Aguilar, 2017)	People	Haar-LBP & HOG	NG	NG	73%	NG	NG
(Kellenberger, 2017)	Wild Animals	AlexNet & Proposed	PC & GTX980 TI	NG	66% (F1 score)	72.65	NG
(Wang L. a., 2016)	Traffic	Optical Flow	PC	MATLAB	99.8%	NG	NN
(Bejiga, 2016)	Victims Avalanches	GoogleNet & SVM	PC	NG	94.29%	0.9	NG
(Zhou, 2016)	Power Line	Canny &Edge Detection	iPad Air	DJI	NG	"Real Time"	NN
(Rudol, 2018)	Human	Haar-Like	PC Pentium III	OpenCV	NG	NG	NG
(Martinez-de Dios, 2001)	Fire	ANFIS	PC	NG	98.00%	NG	NG
(Yong, 2018)	Human	SSD	NG	TensorFlow	73.00%	NG	NG
(Chiu, 2014)	Obstacles	Lucas-Kanade	Embedded Backfin ADSP	uClinux	NG	16.6	NG
TP	**Common Objects**	**SSD/YOLOv3 /Tiny-YOLOv3**	**Smartphone**	**DJI+OpenCV DJI+TensorFlow**	**41.2%/55.3% /33.1% mAp**	**[0.05,1]**	**[22,237]MB**

NG= Not Given; NN = Not Necessary; TP = This Paper; PC = Computer; Red = Lack of information; Green = uses CNN or portable device

The most popular platform so far is OpenCV (Xu, 2017) (Aguilar, 2017) (Rudol, 2018). Accuracy is also specified in most of the cases although some of the algorithms are evaluated in their own dataset. Speed is detailed in frames per second although just two publications achieve a real-time detection speed thanks to the help of a powerful graphics card or the low computational load realised by the algorithms. Finally, the model size is not given in any paper although it is an important fact for portable platforms such as smartphones.

In contrast, our paper is focused on object detection from UAV images with high portability and high reliability. Therefore, different CNN-based detectors are implemented in different platforms in order to compare results. More importantly, the evaluation tests are carried out in a smartphone to achieve those objectives.

2.4.2 Android CNN Analysis

The analysis in Table 2 depicts a comparison among different publications that utilise neural networks in smartphones. Notice that there is not any single publication about the utilisation of neural networks with UAV in constrained devices such as smartphones and tablets, and to the best of our understanding this is the first publication to focus on this execution environment.

Table 2: Comparison analysis of previous CNN deployed in Smartphones use cases

	Aspects							
Ref	**Objective**	**CNN**	**Exec Environment**	**Platform**	**Trained**	**Accuracy**	**Speed**	**Model Size**
(Tobias, 2016)	Sculpture	AlexNet GoogleNet	Ipad 1.3GHz	Caffee	ImageNet	[57.4,59.3]%	[289,992]ms	NG

(Li, 2017)	Traffic	SqueezeNet	Android	TensorFlow	KITTI Vision	76.7%	NG	8MB
(Stoimenov, 2016)	Faces	Own CNN	Android	Google API	Own DB	"Small error"	NG	NG
(Swastika, 2018)	Words	Own CNN	External server	Matlab	Alphabet	[18,100]%	NG	NG
(Idris, 2016)	Gestures	3L MLP	Android	Matlab	Own DB	95%	NG	NG
TP	**Common Objects**	**SSD/YOLOv3 /Tiny-YOLOv3**	**Android**	**OpenCV TensorFlow**	**COCO**	**41.2%/55.3% /33.1% mAP**	**[1,16.757]s**	**[24,237]MB**

NG = Not Given; TP = This Paper; Red = Lack of information

Table 2 compares publication using CNNs in smartphones. However, the architecture proposed by (Swastika, 2018) uses the help of an external server in order to realise the most consuming task, which is the execution of the CNN. Although the accuracy is well explained in every publication, but surprisingly speed in terms of frames per seconds (fps) is not specified, which is a key factor in every constrained machine learning environment. In addition, the model size is just specified in (Li, 2017) as being an influential aspect in the system. Authors in (Idris, 2016) and in (Stoimenov, 2016) use their own training dataset, leading to a high accuracy although it may not be representative enough to be validated properly. Other authors make use of new machine learning platforms such as Caffe and Google Vision API being used in (Tobias, 2016) and (Stoimenov, 2016), respectively. However, these studies either indicate something completely suggestive such as "a small error" or do not provide the size of the model.

Our paper intends to offer a comprehensive benchmarking in object detection systems with neural networks executed in constrained and portable devices with high accuracy in UAV images.

This will make a difference in the speed aspect although not in the accuracy in comparison to other publications referenced in Table 2.

3 Design of the proposed smartphone-based object recognition in the UAV System

This section is divided into two subsections. Firstly, the object detection workflow is expounded in order to clarify how the UAV's frames are finally detected by a smartphone and, secondly, how our approach for YOLOv3 is executed in TensorFlow for Android devices.

3.1 Workflow for object detection using mobile devices

The object detection system presented in this paper is based on two platforms: OpenCV and Tensorflow. In order to obtain comparison data, both platforms are deployed in a similar scheme. Figure 1 shows the three elements used in our system. First of all, a UAV is able to transmit video at high quality such as 720p/1080p with low latency due to the transmission protocols such as LightBridge and OcuSync. Secondly, the controller, which is wirelessly connected to the UAV and wired to the Android device forwards the video stream from the UAV to the smartphone. Finally, the whole computational load will be executed in the Android smartphone.

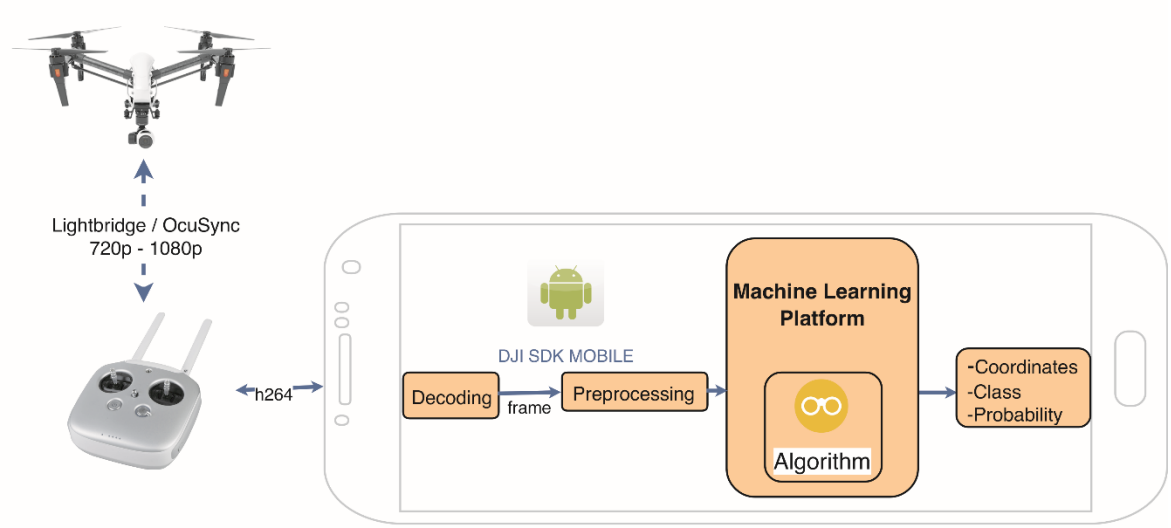

Figure1: System diagram of the proposed UAV system composed by UAV, controller and smartphone, where the smartphone deploys and executes the machine learning platform

When a frame is sent from the UAV until it is shown on the screen, a complex process occurs. Figure 2 represents the whole process in order to clarify the design of our system.

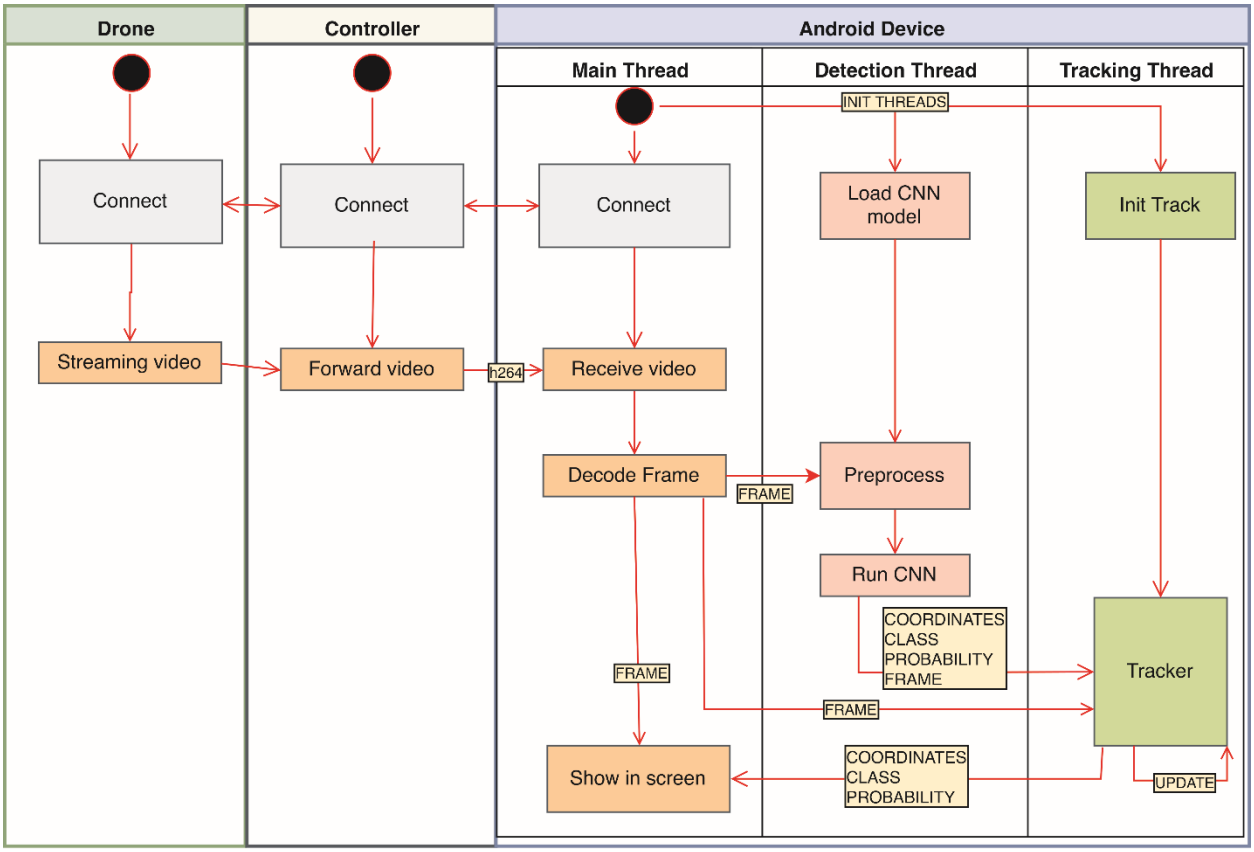

Figure 2: Diagram of the UAV object recognition system where three different threads on the smartphone execute the different object detection and tracking process.

First of all, the three components must connect and synchronise to each other. During this process, the controller recognises the type of UAV and notifies the Android device. In addition, cryptography keys will be exchanged in order to secure the channel. This is possible since the DJI mobile SDK (DJI, n.d.) deployed in an Android device provides such security capabilities to establish a communication to the UAV.

Currently, two threads will be initiate inside the Android device. A **Detection Thread** that will detect the objects from the frames and a **Tracking Thread** that will track each object. At the

moment, the video stream starts from the UAV to the controller and it is forwarded to the smartphone. The video, which is encoded in H.264, is decoded using an internal process inside the device in order to obtain the YUV frames. These YUV frames that are stored in a First In First Out (FIFO) queue and shown on the device screen, while they are available to the Detection and Tracker threads.

In addition to decoding the YUV frames from the UAS, the main thread also has the task of showing the video on the mobile device screen. While this functionality is being realised, the main thread receives the results from the localisation of the detected objects in addition to the class type and the probability coefficient.

The **Detection Thread**, which is in charge of the CNN management, loads the detector model including the CNN configuration and the weights in memory. From this point on, a continuous loop obtains YUV frames from the FIFO queue in order to run the CNN to detect the objects. This loop is highly computationally demanding and is divided in three steps.

- A preprocessing step that transforms each YUV frame in order to prepare it as an input to the detector. This step includes a conversion from YUV to a color format (RGB or BGR), a downscale to the proper input size for the CNN and an image normalisation.
- Secondly, the image is loaded to the CNN, which detects, classifies and recognises the objects it was trained for. Depending on the algorithm, the CNN supplies different output formats. In addition, in some cases such as YOLOv3 and Tiny-YOLOv3 that perform detection at different scales, it is required to indicate the size of the object where the

detection is performed (small-, mid- and big-size), reference henceforth as layers 2, 1 and 0, respectively.

- Finally, the output, usually encoded as an array, shows three basic types of data in an object detector: coordinates to indicate where the object is located, object class and its probability to be that object. Once all objects detected are filtered accordingly to a threshold probability, these data and the frame processed are sent to the tracker in order to update the values stored on it and start to follow the new objects detected.

The **Tracking Thread** tracks the objects detected in the Detection Thread in each of the frames obtained from the FIFO queue. The purpose of this thread is to follow each object during each execution of the detection loop due to the low computationally-demanding task that it runs. The tracker takes less time than the detector and yields a better experience for the final user because it usually runs more frames in less time. Therefore, a fluent track of the objects over the screen is shown. This thread is modular, thus, different tracking algorithms may be used in this system with different performance.

The tracker receives information from the detector with coordinates, object class, probability and the frame where the detection has been performed. When the Detector Thread is performing a new detection, the tracker is obtaining frames from the queue and tracking the objects with the information that it has stored and updating with the new coordinates. Once the detector is finished, new values are updated in the tracker.

As we detail in Section 5.2, the detector and tracker processes are not in real-time, therefore, in order to obtain from the FIFO queue the most recently YUV frames recorded, many frames will

be ignored during the simultaneous running of the 2 processes. In order to make this process clearer, Table 3 presents an example of 15 frames and the actions carried out by the parallel thread running. Based on different time-consuming tasks for each thread, this example counts each obtained frame as a unit of time instead of milliseconds.

- One execution of the detection process lasts for three frames. During the last frame-unit this thread will update the tracker data.
- One execution of the tracking process lasts for two frames. During the last frame-unit this thread will update its own data. This update has lower priority than the update from the detection process.
- Ignoring frames is executed when the other threads are running their tasks. This is the lowest priority option.

Table 3 presents this loop, which starts obtaining frame from the queue and running the object detector on it. At frame 4 the tracker and detector start at the same time, the tracker with the previous data and the detector working to obtain new data values. The tracker will update its values at frame five in order to start again the algorithm at frame six. A critical decision occurs at frame nine, when both (tracker and detection) can update the tracker database. In this situation, the detection values have a higher priority. Notice how the images are always rendered to the final user and then the result of the detection is also rendered as soon as the first tracker finish its work at frame 5. Obviously there will be some framework that will render with the tracking information of previous framework, e.g. in frame 5, we will have in the screen rendered frame 5 and the detection results of frame 4.

Table 3: An example of detection and tracking loop when dealing in non-real-time scenarios

Frame	Detection Thread	Tracking Thread	Ignored	Image Rendering	Detection Rendering
1	Detect_Frame1		NO	YES	NO
2			YES	YES	NO
3	Detector Finish		YES	YES	NO
4	Detect_Frame4	Tracker_Frame4	NO	YES	NO
5		Tracker Finish	YES	YES	YES_Frame4
6	Detector Finish	Tracker_Frame6	NO	YES	YES_Frame4
7	Detect_Frame7	Tracker Finish	NO	YES	YES_Frame6
8		Tracker_Frame8	NO	YES	YES_Frame6
9	Detector Finish	Tracker Finish	YES	YES	YES_Frame8
10	Detect_Frame10	Tracker_Frame10	NO	YES	YES_Frame8
11		Tracker Finish	YES	YES	YES_Frame10
12	Detector Finish	Tracker_Frame12	NO	YES	YES_Frame10
13	Detect_Frame13	Tracker Finish	NO	YES	YES_Frame12
14		Tracker_Frame14	NO	YES	YES_Frame12
15	Detector Finish	Tracker Finish	YES	YES	YES_Frame14

The whole design describes a generic scenario where different tools and algorithms can be developed as it is explained in the following section. This paper is focused on moving the computational load to the Android device edge. Therefore, the implementation completely developed in the mobile device in order to warranty mobility of the platform.

3.2 Object detection algorithm

This subsection focuses on our approach to implement the object detection algorithm, through the adaption of YOLOv3 to TensorFlow models as the selected platform for the workflow defined in the previous subsection. This design attempts to compete with the state-of-the-art YOLOv3 implementation available in OpenCV framework in a similar system. Algorithm 1 highlights the data extraction process for each detected object through pseudocode.

The output array that stores all the detection results is named detection kernel. Its length (see , line 6 and Equation 1) depends on stride (defined as space between pixels of interest that the filter shifts), detection layer, number of anchor detection (defined as number of detection attempts in a zone of interest) and feature map size. The feature map corresponds to the array that stores the information of every detected object for each block. The first four values corresponds to the bounding box (x,y,w,h), the next one to the confidence coefficient and rest corresponds values each class of the object of the trained dataset.

$$blocksize_w = \frac{width}{stride}$$

$$blocksize_h = \frac{height}{stride}$$

$$featMapSize = bbox + conf + numClasses$$

$$kernelSize = blocksize_w \times blocksize_h \times boxesXBlock \times featMapSize$$

Equation 1: Describes the calculation of the kernel size for YOLOv3

The detection kernel iteration to obtain the result of the detection data should be realised by processing the feature maps over each block and over the maximum boxes per block. For this

reason, an offset (line 11).

```
1 pixels ← normalized bitmap
2 feed_Inference (inLayer, pixels, 416, 416)
3 run_Inference (outLayer)
4 blocksizeW ← bitmap.getWidth() / stride
5 blocksizeH ← bitmap.getHeight() / stride
6 output ← new foat[DETECTION_KERNEL_SIZE]
7 fetch_Inference (outLayer, output)
8 for y=0; y<blocksizeH; y++ do
9         for x=0; x<blocksizeW; x++ do
10                for b=0; b<BOXES PER BLOCK; b++ do
11                        offset ← FEATURE_MAP_SIZE * var
12                        xPos ← x + sigm(output[offset + 0]) * stride
13                        yPos ← y + sigm(output[offset + 1]) * stride
14                        w ← output[offset + 2] * ANCHORS[2b + 0]
15                        h ← output[offset + 3] * ANCHORS[2b + 1]
16                        conf ← sigmoid(output[offset+4])
17                        rect ← createRectangle(xPos,yPos,w,h)
18                        class ← mostProbabilityClass()
19                        detections ← addDetection(class, rect, conf)
20 return detections
```

Algorithm 1: Defines the process to obtain the detected objects from the kernel output of YOLOv3 using the kernel size described in Equation 1.

4 Implementation

The described system is a generic enough design able to run on TensorFlow and OpenCV platforms, which accept YUV frames as inputs and obtain data formatted in a similar way as outputs. In addition, this system is widely applicable to different types of CNN-based object detectors such as SSD, YOLOv3 and Tiny-YOLOv3. Additionally, due to its generic implementation, different types of trackers can be used for a better user experience.

These three algorithms are already implemented in OpenCV, therefore, we will deploy them in our scenario. Nevertheless, TensorFlow for Android just implements SSD. In this section, we present our approach for the implementation of YOLOv3 and Tiny-YOLOv3 for TensorFlow in

order to explain how this project has achieved the results produced in Section 5.2 and be able to compare both platforms with the same algorithms. The deployment of YOLOv3 is coded for TensorFlow on Android in order to be executed on mobile platforms.

4.1 Generation of Protobuf models

In order to load YOLOv3 models in the TensorFlow platform using the proper format (protobuf), we have implemented a new strategy. The protobuf format joins the network configuration file and the weight file into one protobuf (.pb) file. For YOLOv2, an existing project supported by Google, called DarkFlow (Trieu, n.d.) is able to transform both YOLOv2 files into a TensorFlow YOLOv2.pb file.

Nevertheless, DarkFlow is currently obsolete and does not support YOLO v3 conversion into a protobuf file. For this reason, this work has managed to achieve the same objective through a different way. For this purpose, we resorted to Keras due to its capability to work with TensorFlow [1].

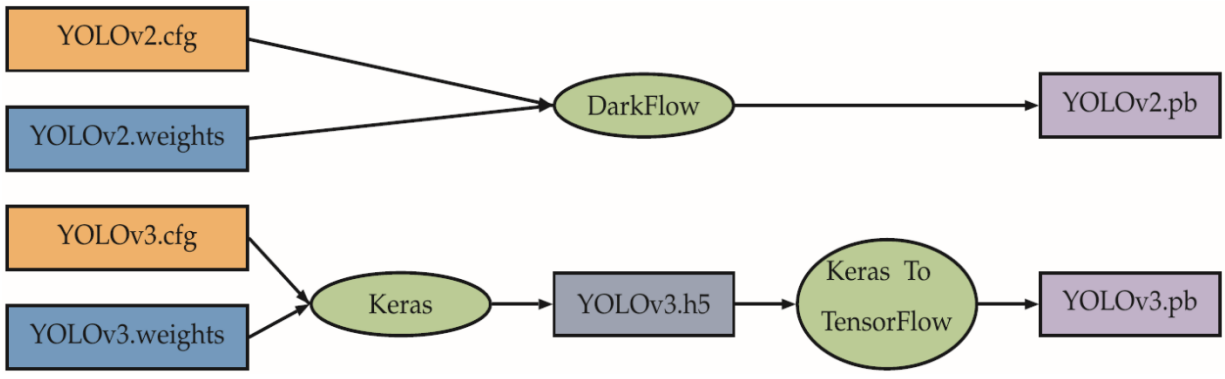

Figure 3: How to generate TensorFlow models from YOLO configuration and weights file to protobuf format

[1] This was part of the preparation for implementation. Keras was not used in the final implementation.

Whilst YOLOv2 just needs one step to obtain its protobuf model, YOLO v3 needs two steps as it is represented in Figure 3. First of all, the network configuration file and the weight file from YOLOv3 is processed by Keras, which will generate an intermediate file with a *h5* extension. In this process, Keras creates a help file, which defines the neural network in JSON format. After *h5* file is created, a keras\to\tensorflow (Abdi, n.d.) script is applied where the input and outputs layers are indicated.

In this last step, the number of output layers indicated depends on the type of YOLO networks. In the YOLOv3 case, three output layers need to be defined; however, for Tiny-YOLOv3, just two should be indicated.

4.2 Integration of Tensorflow to Android

In order to integrate TensorFlow platform to Android, we have explored a library provided by Tensorflow named Android Inference Library, which is an interface that permits this integration.

This library is prepared to be built by Bazel (Group, n.d.), which is a speed and scalable tool to compile projects. Although TensorFlow recommends this technology to build its library, DJI SDK needs directives that Bazel does not support. For this reason, *cmake* is utilized as a substitute in order to build TensorFlow and DJI SDK in the same application. This built also includes the tracker compilation in C++.

For our algorithm, we have employed four main methods provided by TensorFlow Inference Interface in order to run YOLOv3:

- *TensorFlowInferenceInterface* constructor is dedicated to load the protobuf model into memory. This method must be called before any treatment with the CNN.

- *Feed* method (line 2) copies the input data into TensorFlow before running it. Some parameters as the input name, the input size of the frames and dimensions are necessary to be specified.
- In order to run the inference call, *run* method (line 3}) performs this action whilst it is necessary to specify the output name, in YOLOv3 is the number of output layer.
- *Fetch* method (line 7) copies the output Tensor into a output array which will be post-processed in order to obtain the detection results.

4.3 Definition of YOLOv3 anchor boxes

In order to implement YOLOv3 and Tiny-YOLOv3, we have utilised the anchor boxes concept, which predefines some initial sizes (width, height) for the coordinates of the objects detected. The anchor will be resized to the closest size of the detected object obtained from the Tensor output of the neural network.

Specifically, we have defined three anchor boxes per output layer with a format “width, height”; therefore, for Tiny-YOLOv3 six anchor boxes are defined and for YOLOv3 nine. In lines 14 and 15 of the anchors are extracted and calculated in order to obtain the width and height of the detected object. These anchor boxes were calculated after clustering studies on ground truth labels on COCO dataset as YOLOv3 authors recommend.

5 Performance Evaluation

This section presents the empirical performance evaluation of the proposed approach based on TensorFlow and compared with the alternative approach based on OpenCV. For brevity, we use “TensorFlow and OpenCV” to represent the two approaches. All the results are based on realistic

tests applied to both approaches. Over the section, a description of the deployment of the scenario and the benchmarking testbed is detailed. In addition, several tests highlight different features of this design, leading to a comprehensive comparison analysis.

5.1 Deployment of the testbed

5.1.1 Prototype equipment

The UAV is the main entity of the scenario and it has the main responsibility of sending the video stream to the controller. In this paper a DJI-Inspire v1 (SZ DJI Technology Co., Ltd., see Figure 4) is used. It supports a ZENMUSE X3 camera, which has a wide angle lens and can record 4K (@25fps) and 1920x1080 (@60fps) videos and take 12MP photographs.

The video recorded from the UAV is transmitted in real-time to the controller by DJI Lightbrige technology, which is able to transfer the video stream up to 5 kilometres far away.

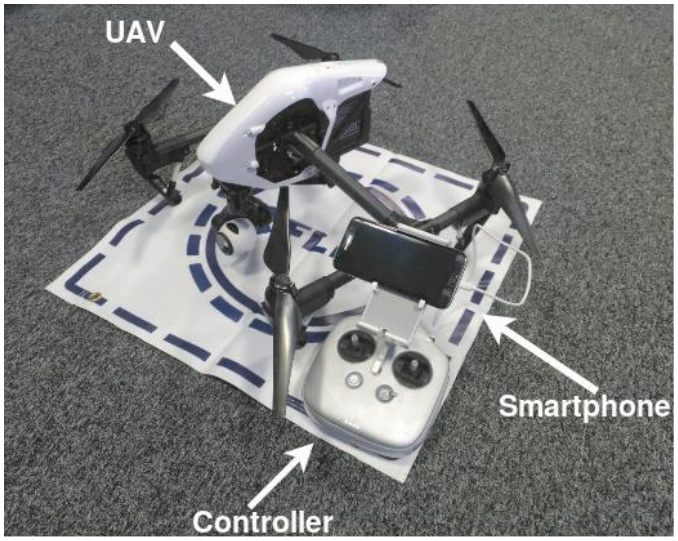

Figure 4: Real testbed Scenario: UAV, controller and smartphone

The Controller is the second entity in the scenario and it is responsible for establishing a direct communication to the UAV. Every command relevant to the UAV flight and also the video

transmission are sent over a dedicated radio channel (operating frequency: 5.725-5.825 GHz and 2.400-2.483 GHz). It also has a USB port that links the controller to the mobile device.

The third entity is the smartphone which is connected to the controller by wire. It runs the Mobile SDK provided by DJI and it allows the user to interact with the UAV and visualise the video stream. The mobile device employed is a Samsung Galaxy S7 Edge (SM-G935F) running Android 7.0, equipped with a Samsung Exynos 8 Octa 8890. (2.3Ghz Quad-Core and 1.6Ghz Quad-Core), 4G RAM and a battery capacity of 3600mAh. The final decision has been to deploy the system in the smartphone instead of the on-board UAV. The main reason is that our system is power-hungry since it needs to continuously execute the CNN and it would significantly affect the performance of the on-board battery of the UAV which are, a valuable resource of the system. It is worth mentioning that a fully charge battery only last for 15 minutes of operation without any system running on top. It can lead to a time frame that is out of the acceptance boundaries for operations.

Furthermore, we have analysed other potential host platforms. Firstly, the Nvidia Jetson AGX Xavier performs better than the smartphone, and a Google TPU based approach is also promising. However, they both require a complete power supply, which compromises the portability of the solution, and they do not come with a screen, which is mandatory to perform the flying operations of the UAV. Secondly, Raspberry Pi 4 comes with a “Broadcom VideoCore V” elemental GPU without support for GPU processing acceleration required in our system. Thirdly, the Neural Compute Stick is an interesting recent add-on; however, the main constraint associated with it is that the USB stick still requires a host platform to be connected. And thus, it

will require the smartphone anyway. Therefore, we have concluded to deploy the system on a smartphone platform.

5.1.2 Machine learning platforms

Two Machine Learning Platforms have been deployed inside the smartphone: TensorFlow and OpenCV. The generic system designed and implemented in Section 3 allows the integration of both platforms without restructuring the scheme.

TensorFlow and OpenCV provide an Android SDK able to implement object detection algorithms among others. This SDK is written with Android JNI allowing a better performance in the device. In addition, different trackers are supported by each of them. In this scenario, OpenCV library is version 4.0.0 which includes YOLOv3 and TensorFlow is deployed (git tag v1.12.0-rc0).

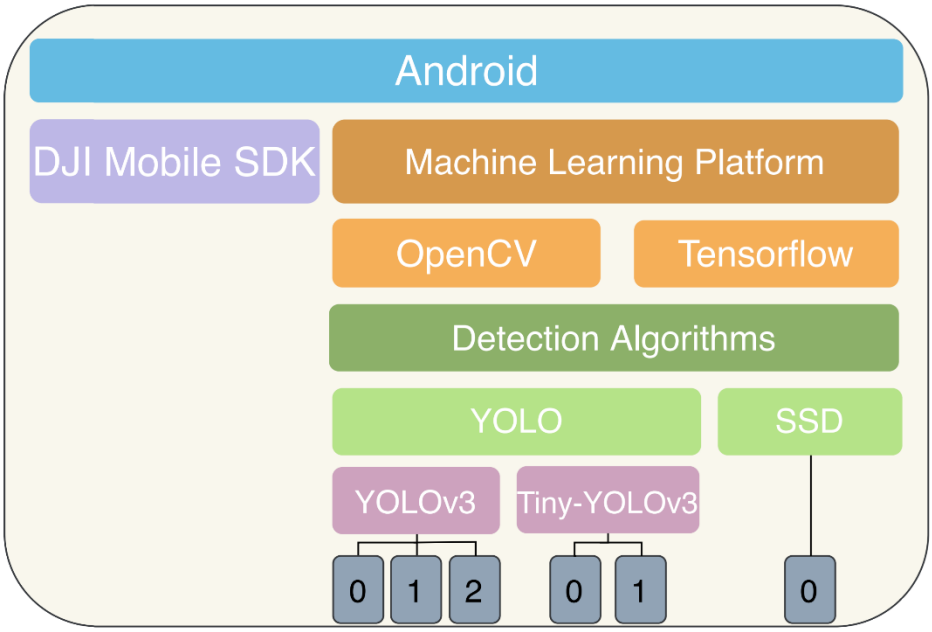

Figure 5: Structure of the technologies deployed in the smartphone for the proposed system

5.1.3 Object detection algorithms

The scenario contemplates two different up-to-day algorithms and a simplification of one of them. SSD, YOLOv3 and Tiny-YOLOv3 will be executed by both platforms in a similar way due to the generic system implemented.

YOLOv3 and Tiny-YOLOv3 can run at different output layers according to the size of the object that they want to detect (prediction across scales). While YOLOv3 can run at three different layers and Tiny-YOLOv3 at two different ones, SSD just can run at one final layer.

Figure 5 shows an architectural overview of every technology deployed in the smartphone. Android OS runs the DJI Mobile SDK in order to communicate with the UAV and the Machine Learning Platforms: TensorFlow and OpenCV. Each of them runs in a lower layer three detection algorithms: YOLOv3, Tiny-YOLOv3 and SSD. Finally, depending on the object detection algorithm, different output layers could be activated.

5.2 Experiments and Results

Several tests have been run in order to analyse different features. Evaluations for accuracy, speed (frames per second), RAM, battery consumption, and temperature are explained in the following subsections. Most of those experiments are performed on both machine learning platforms and on each object detection algorithm available in such platforms and on each output layer available in such algorithm, ending up in more than 60 different datasets.

5.2.1 Tracker

This scenario considers the possibility of neural networks not achieving real-time object detection in a smartphone, therefore, external help is needed such as a tracker algorithm. For this reason. a tracker per machine learning platform is proposed.

For both Machine Learning libraries, optical flow trackers have been employed. Each library implements, using different techniques, such algorithms. For instance, OpenCV library implements their trackers in JNI interface and TensorFlow compiles their C implementation.

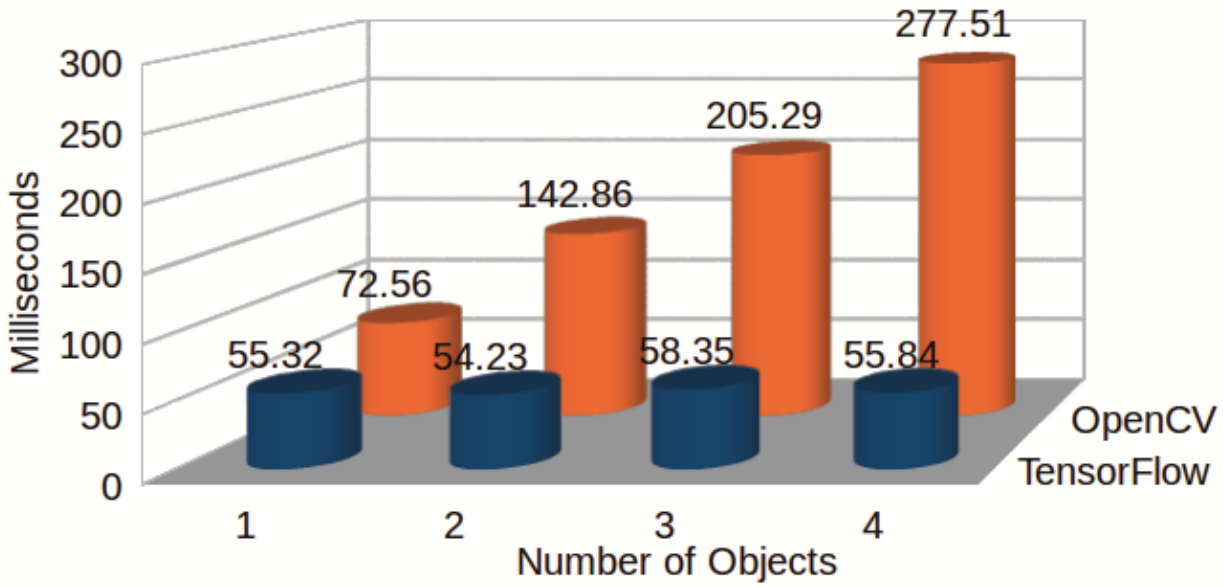

Figure 6: Performance comparison on the tracking process of our system when ranging different number of objects to be followed using both OpenCV and TensorFlow platforms

In order to obtain results from the trackers for each platform, a speed test is performed in this section. We tracked 1--4 objects over 300 frames, extracting performance results for both platforms. As shown in Figure 6, TensorFlow obtains the same times for one, two, three or four objects per frame, however, the performance of OpenCV is decreased, almost proportionally to

the number of objects tracked. This test shows that TensorFlow is able to scale properly when the number of objects to track increases per frame.

Nevertheless, in order to realise a real-time tracking, it is necessary to perform these algorithms under 41.6 milliseconds (with a video stream of 24 fps). TensorFlow is not able to achieve this performance although it just ignores six frames per second being the rest of them correctly tracked.

5.2.2 Models

Three different models has been tested in this scenario. Each one presents variations in comparison to others and within machine learning platforms. As previously defined in section 2, these models are composed by a configuration file which represents the neural network structure and the weights file which stores previously trained values.

Table 4: Performance comparison on the detection process of our system when ranging the size of the CNN configuration using both OpenCV and TensorFlow platforms

CNN	Size		Loading Time	
	OpenCV	**TensorFlow**	**OpenCV**	**TensorFlow**
SSD	23.2MB	29.1MB	404.2ms	1626.2ms
Tiny-YOLOv3	33.8MB	33.8MB	156.6ms	840.4ms
YOLOv3	237.08MB	236.66MB	1474.8ms	6091.6ms

Two tests have been realised in order to measure the size of the models loaded in memory and the time taken to complete the task (Table 4). According to the size, all YOLO models have an equal size in both platforms, although, for SSD, TensorFlow uses its own API leading to 6MB larger size.

On the other hand, as the size of the model increases, the time taken for OpenCV to load it, also increases. Nevertheless, TensorFlow suffers deficiencies in this factor, for instance, the time consumed loading SSD model, which has the lowest size, is five times slower than loading it with OpenCV.

5.2.3 Speed

Nowadays, mobile devices are not able to perform a highly demanding computational task at great speed. They do not possess enough capability power to run complete CNN-based object detectors at real time. This subsection evaluates the time that takes each algorithm to process each frame. In addition, TensorFlow and OpenCV are compared in order to show which one leads to a faster execution.

These tests are executed in a real scenario where the video is received from the UAV and processed by the smartphone. The times are measured from the instant the frame enters the input layers of the CNN until it receives the output from it. Each frame has been scaled down to a resolution of 416x416 pixels, which is an appropriate size for detection. In order to obtain reliable data, we examined a thousand frames for each output layer produced by the algorithms run on each platform. The data is presented as graphs, which illustrate the cumulative average of each frame in milliseconds over the thousand frames tested.

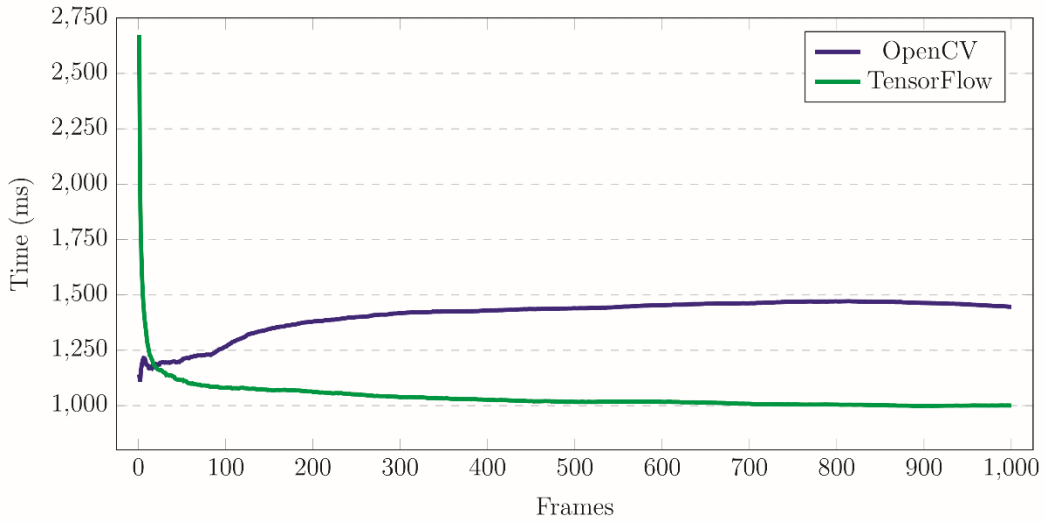

Figure 7: SSD – Accumulative average in milliseconds per frame

Figure 7 shows the performance obtained from the execution of SSD for Tensorflow and for OpenCV. As it can be observed, TensorFlow consumes more than two seconds in the first frames due to a need to access primary memory. Afterwards, it reduces the processing time to one second per frame. On the other hand, OpenCV increases the processing time as the number of frames increments.

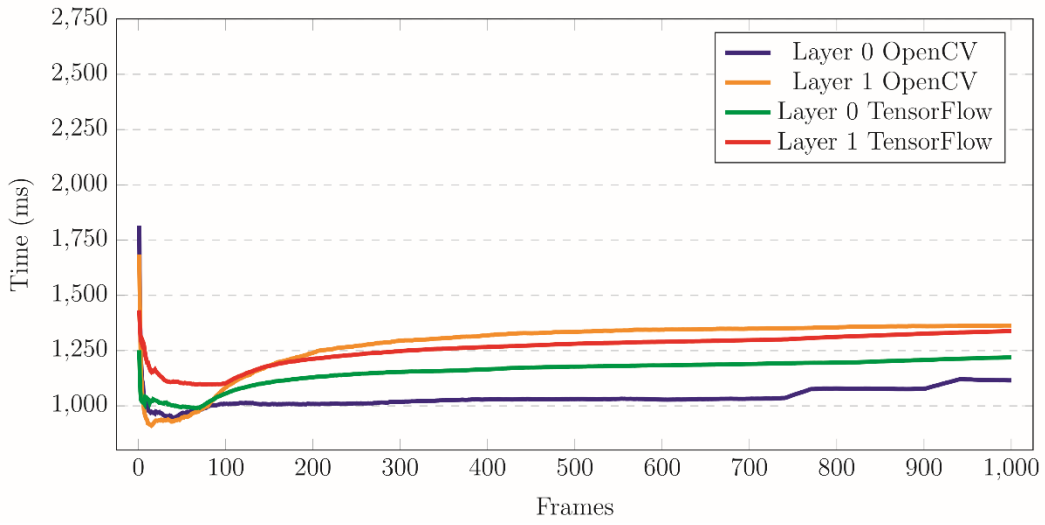

Figure 8: Tiny-YOLOv3 – Accumulative average in milliseconds per frame

In order to evaluate the performance of Tiny-YOLOv3, both output layers have been tested per machine learning platform, and results are depicted in Figure 8. In this analysis, all cases need more time at the first frames to access primary memory. For layer 0, TensorFlow works slower than OpenCV, although in layer 1 TensorFlow processes each frame faster.

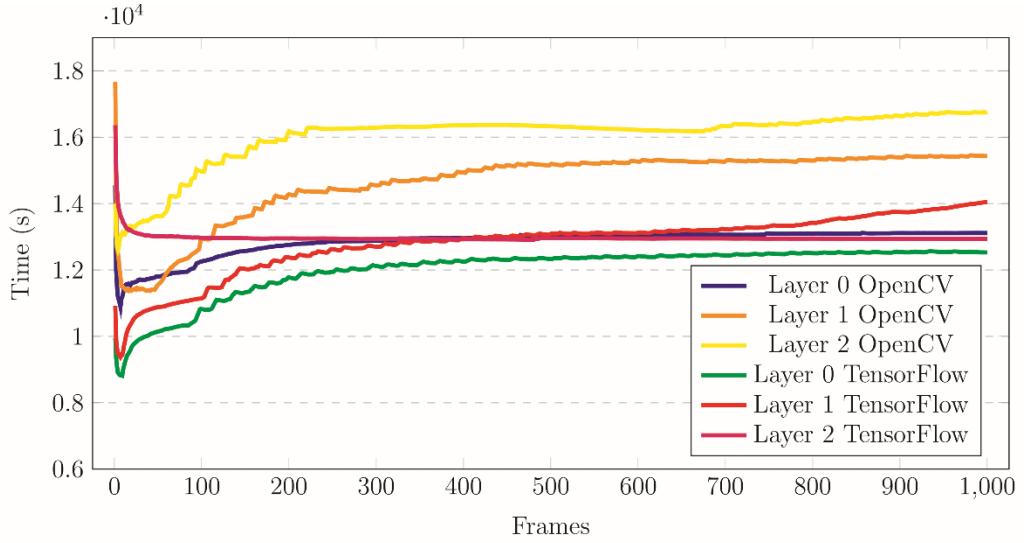

Figure 9: YOLOv3 – Accumulative average in milliseconds per frame

Figure 9 presents six different tests for YOLOv3, where the three layers are evaluated for both platforms. Due to the considerable size and complexity of the model, the speed decreases significantly compared to SSD and Tiny-YOLO. Overall, TensorFlow optimises the processing for all layers obtaining better results than OpenCV.

Table 5: Average of milliseconds per frame for different approaches and CNNs

	SSD	Tiny-YOLOv3		YOLOv3		
Layer/ML	**0**	**0**	**1**	**0**	**1**	**2**
TF	1000.4	1219.2	1338.8	12527.9	14042.4	12939.6

OCV	1446.3	1115.6	1363.1	13111.8	15429.7	16757.3

Table 5 summarises the average time that each frame takes to be processed for each algorithm and each platform. Overall, TensorFlow (TF) is faster in five of out of six cases, surpassed only by OpenCV (OCV) in layer 0 of Tiny-YOLO.

5.2.4 Battery Consumption

The different processes being run on the system could lead to high battery consumption. Running neural networks, which is a demanding computational task, has a direct impact on the mobile device battery.

As explained before, the UAV's controller is connected by wire to the smartphone, therefore, it supplies energy to the mobile device leading to a slower drain of the battery. This charge allows a better performance of the test, with an input voltage of 5V. Although with the battery supply, the consumption is heavy and the battery drains faster than it is charged.

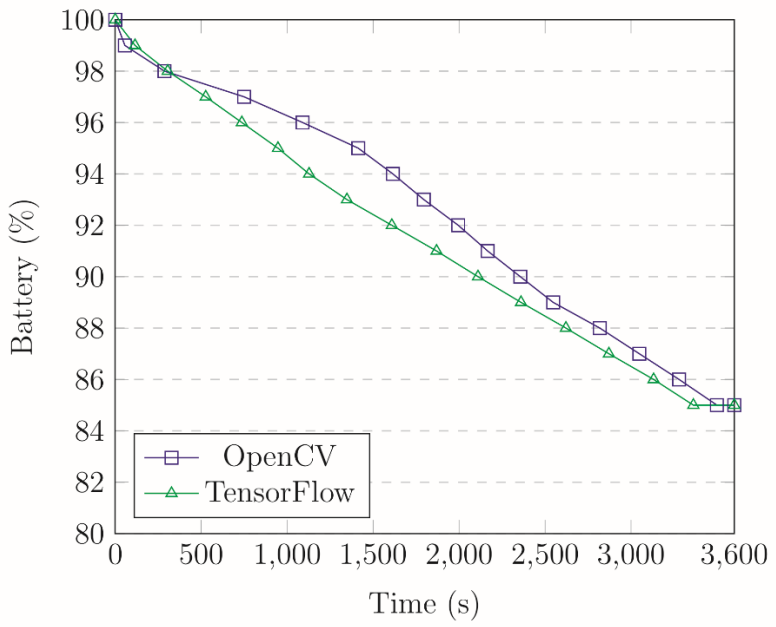

Figure 10: SSD – Battery consumption over an hour of time

In this subsection, the battery is evaluated and compared between platforms for each algorithm. Over these tests, the smartphone battery (3582 mAh) is evaluated during an hour, receiving video streams from the UAV and applying the object detection algorithms. The figures represent the battery percentage (Y-axis) over the time in seconds (X-axis). All the experiments carried out in this subsection are executed for a similar time duration, 3600 seconds which is a decisive factor for battery consumption.

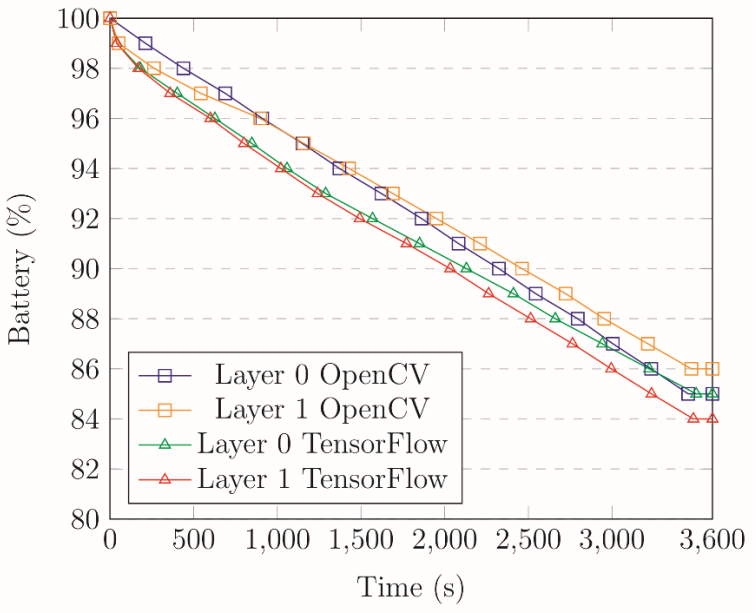

Figure 11: Tiny-YOLOv3 – Battery consumption over an hour of time

The battery consumption presented in Figure 10 shows that TensorFlow and OpenCV perform similarly due to both of them still having an 85% remaining battery after the test.

For the Tiny-YOLOv3 (Figure 11) case there is also a draw in its layer 0 between OpenCV and TensorFlow. There is a difference in the second layer of 2 percent, in this case, OpenCV is more efficient with an 86% of the battery remaining.

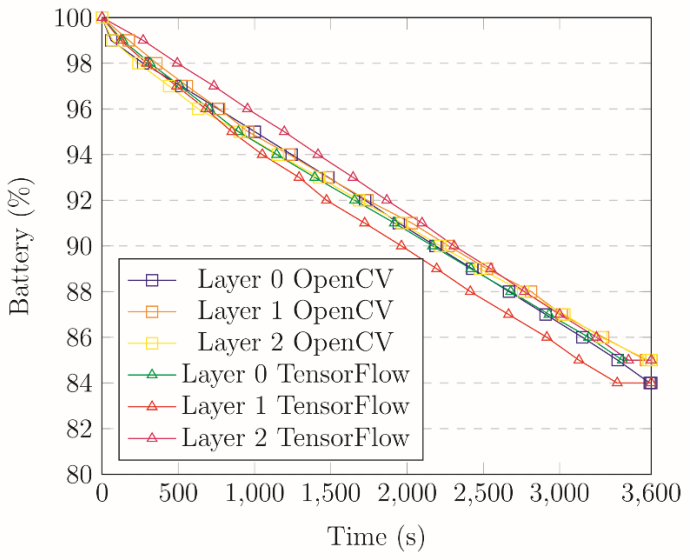

Figure 12: YOLOv3 – Battery consumption over an hour of time

Figure 12 shows YOLOv3 battery. There are two differences, for layer 0, OpenCV performs a higher drain losing one percent of battery, however, for layer 1 TensorFlow shows less efficient and completes the task with an 84% of the total battery.

In summary, both platforms have similar battery consumption with the three algorithms, in all cases being around 85%. In addition, not all the drainage is related to the CNN execution, it is also related to the screen usage in order to show the video and the streaming reception from the UAV.

5.2.5 Temperature

A high battery consumption in addition of a CNN execution leads to a temperature increase in the mobile device. This increase could affect the performance of the smartphone, therefore, it is a factor that needs to be evaluated. The Samsung S7 edge utilised in this experiment has a constant temperature in normal environments of around 30ºC. However, the smartphone should

not be damaged from overheating, which is achievable through shutting down when it reaches a peak of 80-90ºC.

The following figures present the temperature increment over an hour of running the object detection algorithms. In the Y-axis degrees temperature are presented and the X-axis shows time in seconds over and hour. Figures for each object detection algorithm performing in both platforms are presented.

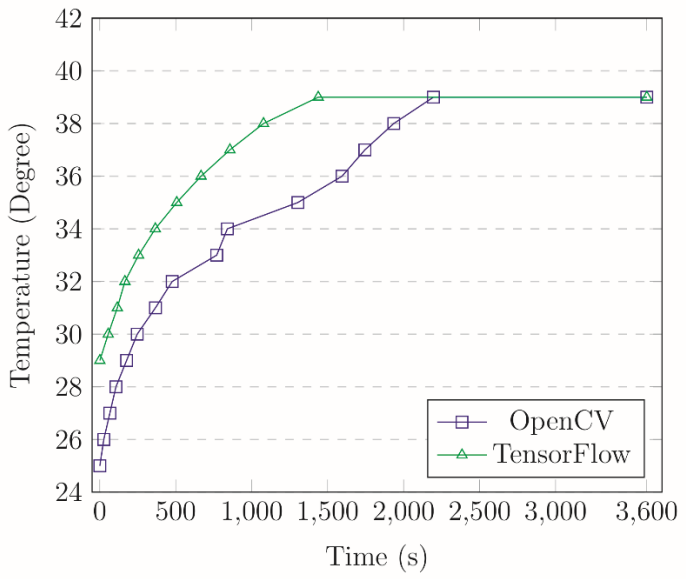

Figure 13: YOLOv3 – Temperature performance over an hour of time

The data collected from the test of the SSD algorithm is presented in Figure 13, which shows how the temperature of the device increases until it stabilises at 39ºC. At the end, both platforms achieve the same temperature although TensorFlow has a sharper increase in the temperature reaching 39ºC 758 seconds earlier than OpenCV.

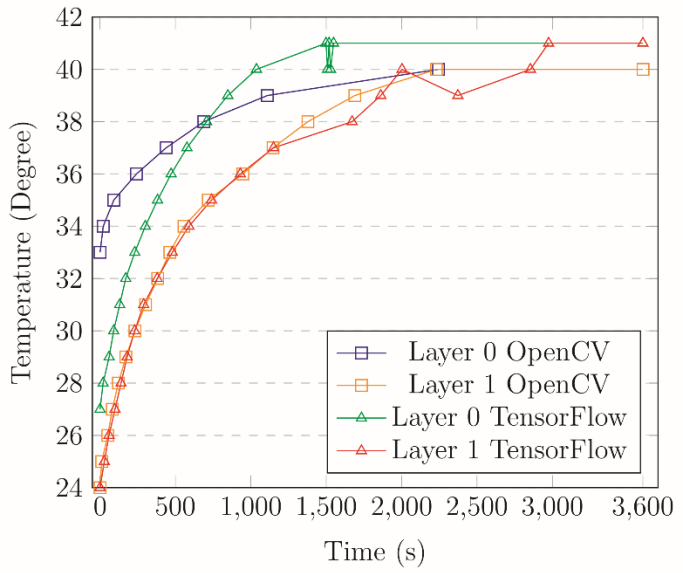

Figure 14: Tiny-YOLOv3 – Temperature performance over an hour of time

In Figure 14 a difference in the final temperature exists between both platforms. While TensorFlow obtains 40ºC in both Tiny-YOLOv3 layers, OpenCV reduces the value to 39ºC being more efficient in this case.

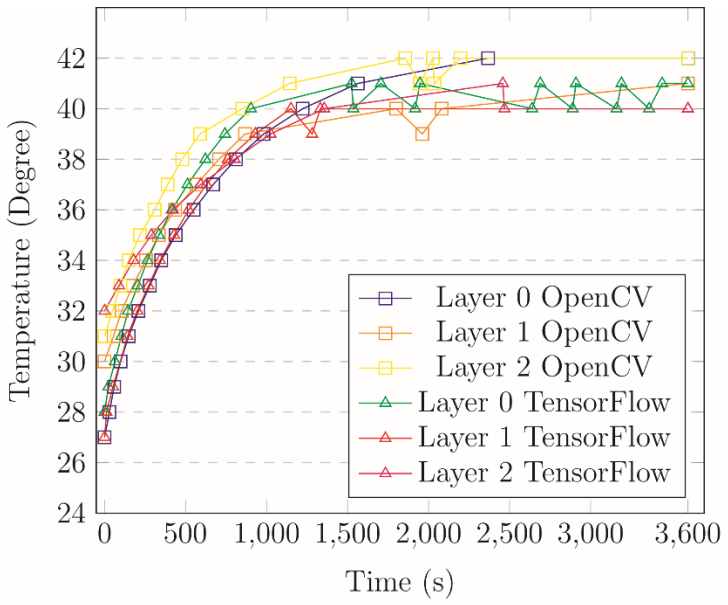

Figure 15: YOLOv3 - Temperature performance over an hour of time

Figure 15 shows the values obtained from the test applied to YOLOv3. On average, the final temperatures are higher than for SSD and Tiny-YOLOv3 due to the larger size of the model. For the three layers, TensorFlow is more efficient with 41ºC, 40ºC and 40ºC (respectively) against OpenCV which reaches a peak of 42ºC in layers 0 and 2, and 41 ºC in layer 1.

5.2.6 Accuracy

Accuracy is a key factor in object detection, particularly, when dealing with UAV-acquired images where the objects are small and the UAV flies at a high altitude.

The three models have been trained equally to evaluate performance in terms of accuracy. This evaluation (Redmon, YOLOv3: An Incremental Improvement, 2018),measured with the COCO Dataset (Lin, 2014), provides the accuracy value as mAP, which stands for mean Average Precision. The Average Precision is calculated as the average of maximum Precision at different Recall levels from the matches of the ground truth (Intersection over Union above 0.5). The Precision measures the accuracy of the CNN predictions and the Recall the proportion of positives detected. mAP is usually employed as a standard measure for comparing algorithms.

After the SSD, YOLOv3 and Tiny-YOLOv3 models were trained, the test dataset, which is detailed in section 2.3, is applied in order to get the mAP. Notice that this test make sure of the dataset rather than the video streamed directly from the UAV to allow fair comparison under equal conditions. SSD provided 41.2 mAP, Tiny-YOLOv3, 33.1 mAP and YOLOv3, 55.3 mAP. Being YOLOv3 the most accurate one and Tiny-YOLOv3 the lowest one. Accuracy achieved by the models is independent of the machine learning platforms, it just depends on the algorithm, in

fact, no meaningful differences between TensorFlow and OpenCV has been obtained, validating such assumption.

Figure 16a: SSD – Accuracy performance for different CNNs in a real environment

Figure 16b: Tiny-YOLOv3 - Accuracy performance for different CNNs in a real environment

Figure 16c: YOLOv3 - Accuracy performance for different CNNs in a real environment

In order to present a visual example with UAV footage, Figure 16a, Figure 16b, Figure 16c show images where the three algorithms have been applied. The sample picture was taken with the UAV described in Section 5.1, therefore, it represents a real use case. In the image, which was taken from an altitude of 20 meters, cattle (colloquially cows) appears resting at different depths. Consequently, they are represented by different number of pixels.

Figure 16a shows the results of executing the SSD algorithm. It just wrongly detects a sheep with 0.285 of confidence while only cows appear in the image. In this example, SSD wrongly detects a tiny sheep without a high confidence.

Tiny-YOLOv3 results are shown in Figure 16b. It correctly detects the bigger cow in the picture with a confidence of 0.312 obtaining a better performance than SSD in this use case.

Finally, Figure 16c displays YOLOv3 results. As it can been observed, YOLOv3 is able to detect ten objects. Four of them are detected wrongly as sheep and the other six are correctly detected as cows. While the sheep gives an average of 0.756 of confidence, the cows are 0.567. Both values can be considered as high confidence values due to the small size of the animals.

In this use case, both YOLO algorithms have been executed in their last output layer which is able to detect smaller objects. This is one of the main characteristics that tips the scale in favour of YOLO algorithms. Although not only cows but also sheep are detected, both animals are similar even for human sight at that high altitude, therefore, it does not signify a great fail for the object detection algorithms. It is also worth nothing that the objects to search for can be limited in the training dataset. The COCO datasets has a wide variety of objects. For particular applications, such as human search for instance, a specific annotation in the dataset could lead to less errors and an accuracy improvement.

5.2.7 RAM Usage

In previous subsections, the size of each model has been studied. This could lead to a hypothesis in which “the bigger the model, the higher the memory consumption”. In order to clarify this assumption, this test evaluates the amount of RAM used to run these algorithms in each platform. The mobile device has 4GB of memory available for applications, therefore, this is the maximum possible to run our scenario.

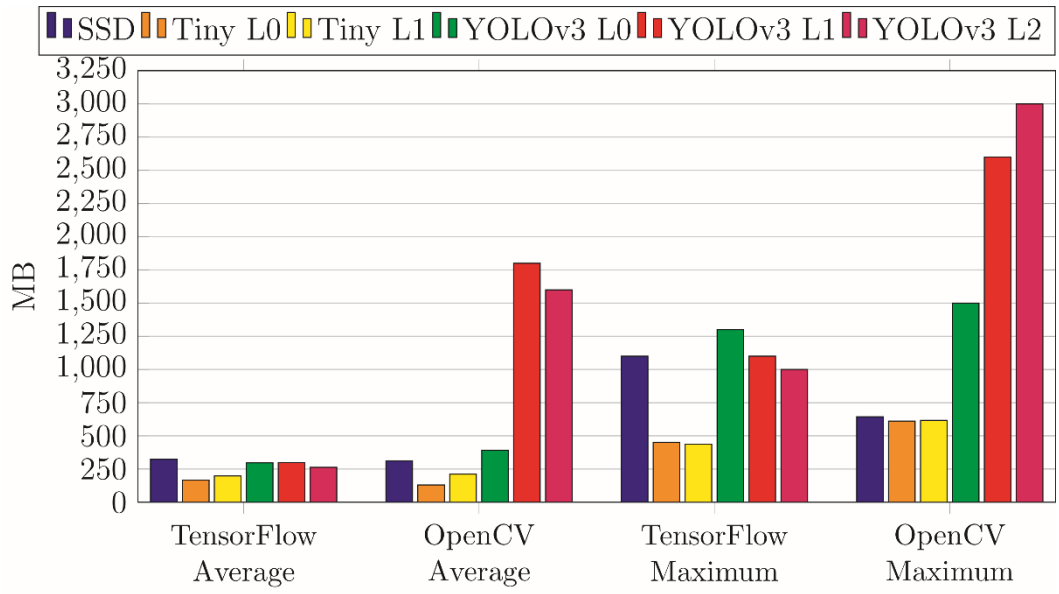

Figure 17: Average and maximum RAM used over an hour of time

Figure 17 represents the tests carried out. Over these tests, the average and the maximum RAM used have been evaluated. Each column represents a value of memory for an algorithm and each group of columns the machine learning platform. The Y-axis represents MB of memory used.

On the one hand, TensorFlow shows a stabilised average of the memory used over the different algorithms. Nevertheless, OpenCV has two peaks when YOLOv3 in layers 1 and 2 are executed.

On the other hand, the maximum RAM used for both platforms are clearly increased when is used in YOLOv3 due to the big size of the model. In comparison, TensorFlow reserves less memory than OpenCV making it a better choice in this category.

After the experimentation, the data obtained can confirm that the hypothesis of “the bigger the model, the higher the RAM consumption” is valid for OpenCV. TensorFlow shows a similar trend in the maximum RAM used, although it scales better the memory used.

5.2.8 Preprocessing

Due to the fact that preparation of the frames as a correct input for the CNNs is a key factor, a time test has been performed to evaluate how much time-consuming this task is.

In OpenCV, the YUV frame received is transformed to a library-specific matrix (*Mat* object). Afterwards, the frame colour will be changed to RGB and the frame will be prepared to enter into the CNN, where it will be resized and set the input.

On the other hand, TensorFlow encodes the YUV frame into ARGB directly in Android without utilising an object class as OpenCV. Afterwards, it transforms the ARGB frame to an Android Bitmap object, and finally, it prepares the input data as a tensor for the CNN.

Tensorflow consumes more time in the preprocessing step, around 189.54 ms. A bit more than 128 ms of difference between platforms demonstrates that OpenCV shows a better performance due to the use of the primary objects implemented in its library such as *Mat*. This optimisation is a strength for OpenCV, whose main field of use is computer vision, contrary to the more general use of TensorFlow which is not developed exclusively for computer vision applications.

6 Comparison Analysis

The previous sections have described how the system has been deployed in OpenCV and TensorFlow. A comparison analytic table is presented in this section to summarise the previous evaluation in order to identify the most appropriate machine learning platform for UAS use cases.

Table 6: Overall performance comparison of TensorFlow- VS OpenCV-based approaches

CNN	SSD	Tiny-YOLOv3	YOLOv3

Layer	0	0	1	0	1	2
Tracker	TensorFlow					
Model Size	OpenCV	Draw		TensorFlow		
Load Model	OpenCV	OpenCV		OpenCV		
FPS	TensorFlow	OpenCV	TensorFlow	TensorFlow	TensorFlow	TensorFlow
Battery	Draw	Draw	OpenCV	TensorFlow	OpenCV	Draw
Temperature	Draw	OpenCV	OpenCV	TensorFlow	TensorFlow	TensorFlow
Accuracy	-	-	-	-	-	-
RAM Average	Draw	TensorFlow	Draw	TensorFlow	TensorFlow	TensorFlow
RAM Maximum	OpenCV	TensorFlow	TensorFlow	TensorFlow	TensorFlow	TensorFlow
Preprocessing	OpenCV					

This summary is depicted in Table 6. In the table, we have highlighted in yellow when OpenCV is superior to TensorFlow, in red when TensorFlow is superior to OpenCV and in green when the results are similar. For instance, OpenCV wins over TensorFlow when the preprocessing of the image is being realised. This process is continually performed every time before the neural network is executed. Nevertheless, TensorFlow shows a better execution when tracking is currently being executed. Because of tracking process is executed more times per second than the preprocessing process, it is preferable to choose TensorFlow platform for both features.

OpenCV scores better results against TensorFlow related to the models' sizes and the time taken to load them, although this feature is realised just once at the starting of the Android application.

According to the RAM usage and the temperature data, TensorFlow performs better results when the CNN is running than OpenCV. In battery results OpenCV is just a little bit better when some layers of YOLOv3 are being executed, nonetheless, this small difference is negligible.

Finally, the most important factor for resource-constrained mobile devices is the speed performed when the neural networks are being executed. In this case, TensorFlow is the clear winner since it achieves higher frames per second than OpenCV. In addition, our implementation of YOLOv3 in TensorFlow in more efficient than the OpenCV implementation.

7 Conclusion

This paper has proposed and prototyped a new, highly portable and capable UAS for object detection and recognition use cases. The solution is empowered by machine learning run over resource-constrained mobile devices such as smartphones. The workflow has been customised to be suitable for such a challenging operational environment, meanwhile it can be applicable to multiple machine learning frameworks such as TensorFlow and OpenCV. The proposed system supports a range of different CNN algorithms including SSD, YOLOv3 and Tiny-YOLOv3. In particular, the design has enabled a new YOLOv3-based object detection algorithm to run in TensorFlow for high detection accuracy.

The proposed design with TensorFlow as the ML platform has been prototyped and empirically compared with an alternative state-of-the-art solution based on OpenCV. Overall, the proposed TensorFlow based approach outperforms the OpenCV alternative in common instructions for ML despite the fact that OpenCV allows better optimisation of the images administration as its target is computer vision. Even with the strengths of each platform, real-time object detection needs to be enhanced as a consequence of the modest resources that the smartphone owns.

Future work will be focused on exploring techniques to further reduce the computational load on mobile devices. New developments such as hardware implementations will be explored to investigate the increase of the performance of the system. Novel structure of the convolutional neural network focused on the reduction of the inference time required to detect small objects will be explored. The main idea is to explore further techniques to real a point where the tracking process is not required since the detection could be achieved in real-time.

8 References

Abdi, A. (n.d.). *Keras To TensorFlow*. (GitHub Repository) Retrieved July 23, 2019, from https://github.com/amir-abdi/keras_to_tensorflow/

Aguilar, W. G. (2017). Pedestrian Detection for UAVs Using Cascade Classifiers with Meanshift. *Proceedings - IEEE 11th International Conference on Semantic Computing, ICSC 2017*.

Bejiga, M. B. (2016). Convolutional neural networks for near real-time object detection from UAV imagery in avalanche search and rescue operations. *2016 IEEE International Geoscience and Remote Sensing Symposium (IGARSS)*, 693--696.

Bradski, G. (2000). The OpenCV Library. *Dr. Dobb's Journal of Software Tools*.

Chiu, C. C. (2014). Vision-based automatic obstacle avoidance for unmanned aerial vehicles. *17*, 797--805.

Chung, C. a. (2018). Very deep convolutional networks for large-scale image recognition. *American Journal of Health-System Pharmacy, 75*, 398--406.

DJI. (n.d.). *DJI Mobile SDK*. (DJI Company) Retrieved July 23, 2019, from https://developer.dji.com/mobile-sdk/

Francois, C. (n.d.). *Keras*. (GitHub repository) Retrieved July 23, 2019, from https://github.com/fchollet/keras

Group, G. (n.d.). *Bazel*. Retrieved July 23, 2019, from https://bazel.build/

Idris, M. I. (2016). Human posture recognition using android smartphone and artificial neural network. *Proceedings - 2015 6th IEEE Control and System Graduate Research Colloquium, ICSGRC 2015, IEEE*.

Karayev, S. a. (2014). Caffe. 675--678.

Kellenberger, B. a. (2017). Fast animal detection in UAV images using convolutional neural networks. *International Geoscience and Remote Sensing Symposium (IGARSS), 2017-July*, 866--869.

Ko, K. E. (2018). Deep convolutional framework for abnormal behavior detection in a smart surveillance system. *Engineering Applications of Artificial Intelligence*.

Li, Z. a. (2017). Object Detection and Its Implementation on Android Devices.

Lin, T. Y. (2014). Microsoft COCO: Common objects in context. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 8693 LNCS*, 740--755.

Liu, W. a. (2016). SSD: Single shot multibox detector. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 9905 LNCS*, 21--37.

Martin Abadi, A. P. (n.d.). TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. *Software available from tensorflow.org*.

Martinez-de Dios, J. R. (2001). Fire Detection Using Autonomous Aerial Vehicles With Infrared and Visual Cameras. *Control, 16*, 660--665.

Panwar, N. a. (2016). A survey on 5G: The next generation of mobile communication. *Physical Communication, 18*, 64--84.

Redmon, J. a. (2016). YOLO9000: Better, Faster, Stronger.

Redmon, J. a. (2018). YOLOv3: An Incremental Improvement.

Rudol, P. a. (2018). Human body detection and geolocalization for UAV search and rescue missions using color and thermal imagery. *IEEE Aerospace Conference Proceedings*.

Sharma, V. a. (2018). On the positioning likelihood of UAVs in 5G networks. *Physical Communication, 31*, 1--9.

Stoimenov, S. a. (2016). Face recognition system in Android using neural networks. *2016 13th Symposium on Neural Networks and Applications, NEUREL 2016, IEEE*, 1--4.

Swastika, W. a. (2018). Android based application for recognition of Indonesian restaurant menus using convolution neural network. *Proceedings - 2017 International Conference on Sustainable Information Engineering and Technology, SIET 2017, 2018-Janua*, 30--34.

Tobias, L. a. (2016). Convolutional Neural Networks for object recognition on mobile devices: A case study. *2016 23rd International Conference on Pattern Recognition (ICPR)*.

Trieu. (n.d.). *DarkFlow*. (GitHub Repository) Retrieved July 23, 2019, from https://github.com/thtrieu/darkflow

Wang, L. a. (2016). Detecting and tracking vehicles in traffic by unmanned aerial vehicles. *Automation in Construction*, 294--308.

Wang, X. a. (2019). A UAV-assisted topology-aware data aggregation protocol in WSN. *Physical Communication*.

Xu, Y. a. (2017). An Enhanced Viola-Jones Vehicle Detection Method from Unmanned Aerial Vehicles Imagery. *IEEE Transactions on Intelligent Transportation Systems, 18*, 1845--1856.

Yong, S.-p. a.-c. (2018). Human Object Detection in Forest with Deep Learning based on Drone ' s Vision. *2018 4th International Conference on Computer and Information Sciences (ICCOINS)*, 1--5.

Zhou, G. a.-L. (2016). Robust real-time UAV based power line detection and tracking. *Image Processing (ICIP), 2016 IEEE International Conference on*, 744--748.

9 Biography

Ignacio Martinez-Alpiste is a PhD researcher at the University of the West of Scotland, United Kingdom, where he is actively involved in the "Smart Unmanned Aerial System for Real-Time Object Detection" project. His main research interests include video networking, object detection algorithms, artificial intelligence with applications in Unmanned Aerial Systems, among others. He is also technical program committee member of international scientific conferences. He completed his BSc Computer Engineering degree at Universidad de Murcia.

Pablo Casaseca-de-la-Higuera is an Associate Professor at the Department of Signal Theory & Communications, and Telematics Engineering, Universidad de Valladolid. He has published several book chapters and more than 70 papers in indexed journals and flagship international conferences. His research interests are signal and image processing and artificial intelligence with applications in biomedical engineering, computer vision, telecommunications, and unmanned aerial vehicles. He is a reviewer of several scientific journals and program committee member of main conferences, and acts as an assessor for the evaluation of research proposals for UK Research Councils.

Jose M. Alcaraz Calero SMIEEE, is Full Professor in Network at University of the West of Scotland (UWS). He is the Technical Co-Manager of EU Horizon 2020 5G-PPP projects SELFNET and SliceNet. He has published more than 120 contributions, including 15 patents and IPRs, in international conferences and renowned journals in the fields of network management, video networking and UAV applications. He is member of the NATO IST-118 working group. Prof. Alcaraz-Calero is active in chairman and editorship activities where he has served in more than 20 international conferences and 20 editorial activities for journals like IEEE Communication Magazine.

Professor Christos Grecos (SM IEEE 06, SM SPIE 2008) is the Vice Dean of Postgraduate studies and Research in the National College of Ireland. His research interests include image/video compression standards, image/video processing and analysis, image/video networking and computer vision. He has published over 195 papers in top-tier international publications on these topics. He has been a UK delegate on the NATO panel on Disadvantaged Networks, was involved in the Video Quality Experts Group and is a High End Foreign Expert for the Chinese government. He has obtained significant funding from several international projects funded by UK EPSRC/TSB and EU.

Qi Wang is a Full Professor with the University of the West of Scotland (UWS), UK. He is the Technical Co-Manager of EU Horizon 2020 5G-PPP projects SELFNET and SliceNet, and Principal Investigator of a number of projects funded by UK EPSRC etc. His primary research interests include future-generation mobile networks, video networking and UAV applications. He has published over 120

peer-reviewed papers in these areas, and is a winner of several Best Paper Awards from flagship international conferences such as IEEE ICCE 2012 and 2014. He received his PhD degree in Mobile Networking from the University of Plymouth, UK.