FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Tracking and Representing Objects in a Collaborative Interchangeable Reality Platform

Joana Whiteman Catarino

MESTRADO INTEGRADO EM ENGENHARIA ELETROTÉCNICA E DE COMPUTADORES

Supervisor: Luís Filipe Teixeira

Co-supervisor: Rui Nóbrega

External Co-supervisor: Teresa Matos

June 8, 2020

Abstract

Over the last few years, with the constant technological evolution and the emergence of new devices, the interest in topics such as remote collaboration has been increasing and, consequently, several studies are being developed in this area. The desire to interact with someone on the other side of the world as if it was on our side, both in a professional and in a playful context, led to the idea of this work. It emerged as part of the PAINTER project. It consisted of developing a collaborative application where several users can interact in real-time with the same virtual content placed in the exact location where real objects were previously detected through a smartphone.

Due to the real-time goal of this work, the proposed implementation required avid research to integrate all components, as they should interact in the best possible way. We studied different Hardware Acceleration (HA) Software Development Kits (SDKs), and we chose the ArmNN to achieve better performance on running the object detection using a Convolutional Neural Network (CNN) model in our smartphone. It establishes a connection between the chosen Artificial Intelligence (AI) framework and our device's Arm-based CPU/GPU. We analyzed the different possibilities of this type of frameworks and chose Tensorflow, which is compatible with the ArmNN SDK. The choice of the CNN model was related to these two components and to the research conducted to understand which are the most efficient models, that led us to the Tiny-YOLOv2 model.

We also did a tracking analysis and concluded that the best tracker for our application was KCF. As we opted for Android OS, we chose ARCore as our AR framework, which was mainly responsible for the collaboration implementation. The integration of these components was considered the best approach for the development of this system. However, each of them conditioned each other's performance.

We made an analysis of the frame's processing time on different devices and obtained a frame rate value of approximately 18 fps on the Samsung Galaxy S9 Plus. We also verified that its object detection's frame rate was 4 fps. The model achieved a mean Average Precision (mAP) of 13.6%, considering a small sample of images collected and annotated. The KCF tracker showed a frame rate of 10.2 fps and a mAP of 4.7%, under the same conditions used in the previous analysis. Moreover, results show that the lack of GPU acceleration in object detection had a considerable impact on our application's performance. We demonstrated the fundamental goal of this work by performing real object detection and tracking through a smartphone in real-time and allowing users interaction with the corresponding virtual objects.

Resumo

A constante evolução tecnológica e o aparecimento de novos dispositivos ao longo dos últimos anos, estimulou o interesse em temas como a colaboração remota, levando ao desenvolvimento de vários estudos nesta área. A ideia deste trabalho surgiu da necessidade e do desejo de poder interagir com alguém no outro lado do planeta como se estivesse mesmo ao nosso lado, tanto num âmbito profissional como lúdico. Este integra o projeto PAINTER e consiste no desenvolvimento de uma aplicação colaborativa onde múltiplos utilizadores podem interagir em tempo real com objetos virtuais posicionados na localização exata dos objetos reais previamente detetados através de um smartphone.

De acordo com o principal objetivo deste trabalho, a proposta de implementação envolveu uma pesquisa direcionada à integração da melhor forma possível de todos os componentes. Investigámos diferentes SDKs de Aceleração de Hardware e optámos pelo ArmNN de forma a obter uma melhor performance na deteção de objetos através de um modelo CNN num smartphone. Este SDK estabelece uma ligação entre a ferramenta de AI escolhida e o CPU/GPU do dispositivo, desde que estes componentes de hardware sejam da Arm. Analisámos as diferentes opções existentes deste tipo de ferramenta e a sua compatibilidade com o SDK ArmNN que nos levou a escolher o Tensorflow. A escolha do modelo CNN esteve diretamente relacionada com estes dois componentes e, após a pesquisa que nos permitiu perceber quais os modelos mais eficientes, optámos pelo Tiny-YOLOv2.

Realizámos ainda uma análise de forma a perceber qual o melhor tracker a utilizar na nossa aplicação, na qual salientamos o KCF. Dada a decisão de utilizar smartphones com o Sistema Operativo Android, escolhemos o ARCore como nossa ferramenta de AR, utilizada maioritariamente na implementação da parte colaborativa. Considerámos que a integração destes componentes é a melhor abordagem para o desenvolvimento deste sistema. No entanto, cada um deles condicionou a performance dos restantes.

Fizemos uma análise do tempo de processamento de cada frame em diferentes dispositivos e obtivemos aproximadamente 18 fps no Samsung Galaxy S9 Plus. Também verificámos que a deteção de objetos apresentou um frame rate de 4 fps. O nosso modelo alcançou uma mAP de 13.6% quando submetido a uma análise de uma pequena amostra de imagens recolhida e anotada por nós. O tracker KCF apresentou um frame rate de 10.2 fps e uma mAP de 4.7%, sob as condições mencionadas anteriormente. Para além disso, os resultados mostraram que a falta de aceleração do GPU na deteção de objetos teve um impacto considerável na performance da nossa aplicação. Demonstrámos o objetivo principal deste trabalho implementando a deteção e o tracking de objetos reais em tempo real através de um smartphone e permitindo a interação dos utilizadores com os objetos virtuais correspondentes.

Acknowledgements

I would like to thank my supervisors, Professor Rui Nóbrega and Teresa Matos, with a special thanks to Professor Luís Teixeira, for all the guidance and the support in this work. Moreover, I am grateful for the proposal of such interesting work to develop, explore, and learn.

I thank my colleague Isabel Oliveira for the technical support on the neural network training part. I also thank my family, especially my parents and my sister, for all the moral support and for being always available to help me with everything.

And last but not least, I special thanks to Diogo not only for the technical support but also for encouraging the development of this work and for being always on my side no matter what.

Joana Catarino

Contents

List of Figures

List of Tables

Acronyms

AI	Artificial Intelligence
API	Application Programming Interface
APU	AI Processing Unit
AR	Augmented Reality
CNN	Convolutional Neural Network
CV	Computer Vision
DL	Deep Learning
DSP	Digital Signal Processor
FLOPS	Floating-point Operations Per Second
FPN	Feature Pyramid Network
HA	Hardware Acceleration
HAL	Hardware Abstraction Layer
HMD	Head-Mounted Display
HRI	Human-Robot Interaction
IDE	Integrated Development Environment
IOU	Intersection Over Union
KCF	Kernelized Correlation Filter
LAMR	Log-Average Miss Rate
mAP	Mean Average Precision
ML	Machine Learning
MR	Mixed Reality
NN	Neural Network
NNAPI	Neural Network API
NPU	Neural Processing Unit
OCR	Optical Character Recognition
OS	Operating System
R-CNN	Regions with CNN features
RFCN	Region-based Fully Convolutional Networks
RPN	Region Proposal Network
SDK	Software Development Kit
SLAM	Simultaneous Localization and Mapping
SNPE	Snapdragon Neural Processing Engine
SSD	Single Shot Detector
SoC	System on Chip
UI	User Interface
VR	Virtual Reality
YOLO	You Only Look Once

Chapter 1

Introduction

Throughout the years, technology has become present everywhere and, with that, several devices have undergone constant evolution. Smartphones are nowadays almost essential in our lives. We can stay in touch with our friends and family, play a game, watch a video, and listen to music, and we can do some tasks related to our work, simplifying it. We have access to our personal and professional email, and we can schedule our appointments. So, we have almost everything we could need with us, making our daily lives much more comfortable.

However, there are still many things we cannot do with our smartphone or computer. When we want to be with someone, for instance, to play a board game like in old times, we need to be with the person physically. Similarly, in a professional scenario, if we belong to an interior design team, we must be together to discuss specific changes to be made. These are just two examples of many things that we cannot do apart.

Consequently, the desire to interact with someone in another place, as represented in science fiction movies, for instance, with holograms and other approaches, is continuously increasing. Regarding the two cases mentioned above, if we are far from each other, it would be interesting to play together as if we were in the same physical space. That is, each player has its pieces and can position and move them according to the rules and can see the other player's pieces and interactions in real-time. Likewise, the interior design team can work simultaneously on the same project even though they are in different locations if there was a virtual 3D model visible to all where each one can add a separate piece of furniture.

That is the idea of a collaborative platform: several users may view and simultaneously interact with the same virtual environment, cooperating, using Augmented Reality (AR), or Virtual Reality (VR) devices. Our work is part of the PAINTER project that is being developed at FEUP and emerged in this idea's scope.

The creation of a bridge between VR and AR has been encouraged to represent virtual objects in a real environment and allow users to interact with them. This bridge is called Mixed Reality (MR). It joins both concepts mentioned above, i.e., the integration of VR, where the scene is artificial, and AR, where virtual objects overlay over the real environment. It has been an area of avid research because it can be applied in several professional settings, such as architecture,

medicine, education, construction, and leisure activities, such as entertainment and games. This work differs from others that will be presented in Chapter 2 since it aggregates the recognition of particular objects, their tracking and consequently virtual representation in the real environment, and the collaboration between users.

Objectives and Approach This work aims to represent real-world objects in a virtual or partially virtual environment so that several users can interact with each other in real-time as if they were present in the same physical space. The platform to be developed must detect an object and track its position and orientation in the real environment. The tracking data should be used to recreate the object's movements in the virtual environment, based on the object's position in the real world. This position ensures that the virtual object is initially placed in the correct location in the virtual environment. Finally, both users can share an interactive experience, since the virtual objects can change their position and orientation based on real objects. This approach requires several steps, such as recognition, detection, and virtual visualization of the scene. The goal is to create a virtual collaboration tool that works in real-time, using simple cameras and VR headsets.

In this work, a simple collaborative interchangeable platform is proposed. A smartphone application that detects and tracks multiple objects in real-time is created by taking advantage of different available tools.

1.1 Document Structure

After this brief introduction, in which some basic concepts and topics that serve as motivation for this work were presented, a deeper analysis of object detection, tracking, collaboration environment, and related background work is presented in Chapter 2. In Chapter 3, the problem characterization is presented, and the proposed solution and the system architecture are described. In Chapter 4, all implementation details are explained. In Chapter 5, a performance evaluation is presented as well as some considerations about the system's implementation. Finally, conclusions and future work are presented in Chapter 6.

Chapter 2

Background and Related Work

In this chapter, we present several works carried out in multiple relevant areas to the development of this system. These are: (a) objects' classification and detection; (b) object tracking in real-time; (c) 3D models' representation of real objects in a virtual environment; and (d) users' collaboration and interaction. The coalescence of these areas results in the birth of complex systems that can complement the detection and tracking of certain elements with their representation in a virtual context so that multiple users can interact in real-time.

In this context, Gao et al. [8] developed a mobile remote collaboration system based on the mixed reality that allows a specialist to offer real-time assistance in physical tasks to an individual who so desires, independently of the distance between them. Google ARCore[1], a tracking system that can calculate the position of particular objects using an external depth sensor connected to a mobile phone, is used to represent the physical environment in a virtual world. The scene where the subject is present is captured and transmitted wirelessly to the expert that visualizes it on a mobile VR headset (in this case, HTC VIVE Focus). In this way, the remote expert can immerse himself in the virtual scenario and thus offer assistance as if they are sharing the same physical location, simulating face-to-face cooperation. Also, the remote assistance is transmitted back in the form of augmented reality that overlais the video in the subject's display.

Some works such as Room2Room [1] and Mini-Me [2] attempt to represent a person in a virtual environment. The former designed a telepresence system intending to create an image of the person, while the latter creates a Mini-Me and a real-size virtual avatar. Through projected augmented reality, Pejsa et al. [1] allow interaction between two remote participants, recreating a face-to-face conversation experience. Users are captured with color and depth cameras to project their size and features into the remote space. This system creates an illusion of the physical presence of remote users in a given space, as well as the look and movements performed by them. Figure 2.1, shows a representation of the remote interaction between users, wherein in (a) and (d), it is possible to visualize the real people communicating with the projection of each other and, in (b) and (c), a frontal plane shows the respective projections.

[1] more details about this AR framework at https://developers.google.com/ar/ [accessed 24 September 2019]

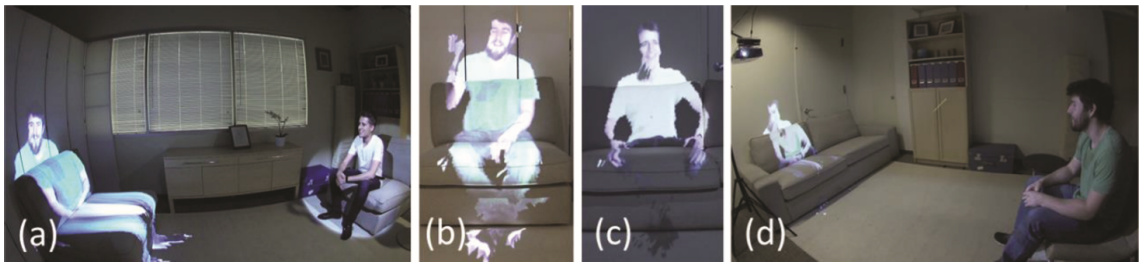


Figure 2.1: Remote interaction between users (taken from [1])

Piumsomboon et al. [2] take a slightly different approach by developing an adaptive avatar to enhance remote MR collaboration between a local AR user and a remote VR user. The Mini-Me avatar represents the gaze direction and body gestures of the VR user while transforms their size and orientation to remain within the AR user's field of vision. These authors also perform a user study that evaluates their system in two possible collaborative scenarios: i) an asymmetric remote expert in VR assisting a local worker in AR; ii) a symmetric collaboration in urban planning. Piumsomboon et al. conclude that the presence of the Mini-Me significantly improves the social presence and the entire collaborative experience in MR. In Figure 2.2, it is possible to visualize some captures of the system in operation. (a) shows the real-size avatar projection and (e) shows the Mini-Me avatar projection. (b) presents both avatars in the virtual environment and (d) shows a user experience.

Figure 2.2: People avatar representation (taken from [2])

Recently, Rünz and Agapito [3] developed multiple moving objects' real-time detection, tracking, and reconstruction system called MaskFusion. Since this is a Simultaneous Localization and Mapping (SLAM) [9, 10] system capable of analyzing a real-time object-level scenario, a new approach was used to deal with the two significant limitations present in this type of systems. Most SLAM methods assume that the environment is predominantly static and that moving objects are, at best, detected as outliers and therefore ignored. Besides, the result of most of these SLAM systems is a purely geometric map of the environment without the inclusion of any semantic information.

To overcome these limitations and to increase the accuracy of the object mask limits, the authors combine the results of two algorithms:

1. Mask-RCNN model [11], a powerful image-based instance-level segmentation algorithm that can predict object category labels for up to 80 object classes;

2. a geometry-based segmentation algorithm capable of creating an object edge map from depth and surface cues.

Through an RGBD camera, a rather crowded and cluttered scenario is captured, followed by an image-based instance-level semantic segmentation capable of creating semantic object masks that allow the detection of real-time objects and their subsequent representation. In the following image, a sequence of three frames illustrating MaskFusion recognition, tracking, and mapping capabilities are presented. The upper half of Figure 2.3 shows a rebuilt scenario with a white background where each element was colored differently. Between frames (b) and (c), a user changed the teddy bear and the bottle's position. The bottom half of the image shows the overlap of the frames captured by the RGBD camera with the semantic masks produced by the segmentation neural network.

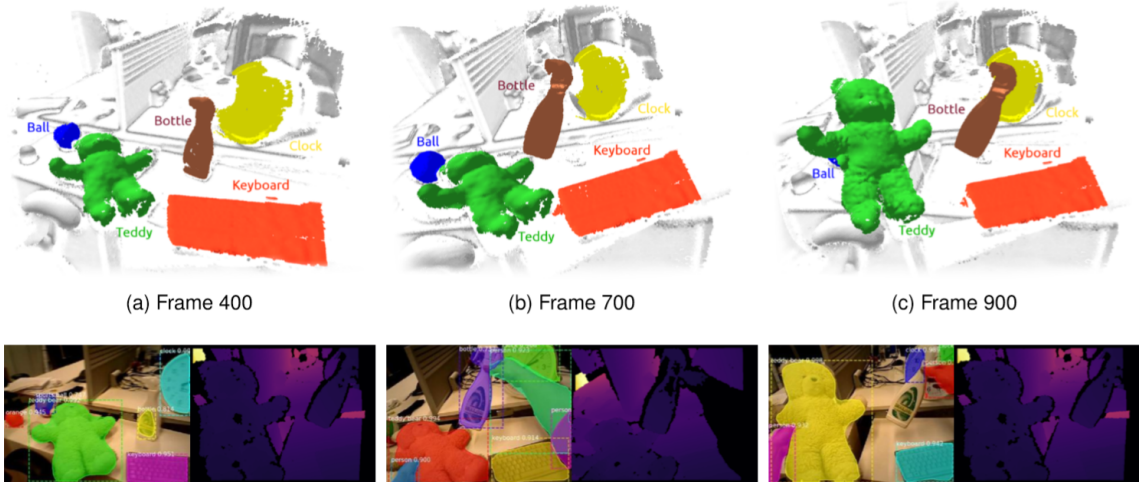


Figure 2.3: Representation of three frames recorded at distinct moments (taken from [3])

2.1 Real-Time Object Detection and Classification

To identify different objects in images, one should start by extracting their features and then proceed to the corresponding matching in the environment under study. For this purpose, several feature extractors such as SIFT [12], SURF [13], and, later, ORB [14], were developed for reducing the time spent on real-time detection. On the next step, it is necessary to train the model for object classification, and it is within this context that machine learning appears. Features of several analyzed images should be grouped by the corresponding label to identify the object.

Deep Learning (DL) emerges as a consequence of integrating Machine Learning (ML) in the image feature extraction, representing an alternative approach, capable of acquiring new information through feature extraction. In the last years, several algorithms based on this approach were developed and used to detect, classify and locate objects in images. [15]

2.1.1 Methods' Description

The performance of object detection was in a standstill stage, and the systems with higher performance combined a sophisticated multiple low-level image features with high-level context from object detectors and scene classifiers, when Girshick et al. [16] developed a scalable and straightforward object detection algorithm, R-CNN: Regions with CNN features. It improved by 30% the highest results obtained until then with the PASCAL VOC 2012 dataset[2]. High capacity convolutional neural networks were used for object location and segmentation, as well as a more effective training approach since it is challenging to train considerable big CNNs when the labeled training data is scarce. In these situations, the common practice involves a non-supervised pre-training followed by a supervised fine-tuning process. However, these authors showed that it is highly useful to perform a supervised pre-training of the network for an auxiliary task using abundant data from classification datasets (in this case, ILSVR2012 [17]), and then adjust the network to the target task, for which there is not so much data, from detection data sets (in this case, PASCAL VOC). Girshick et al. also stated that possibly this "supervised pre-training / domain-specific fine-tuning" paradigm would also be beneficial for a considerable diversity of problems concerning vision where data it is not abundant.

Girshick improved the developed model, creating the Fast R-CNN [18], an object detection method with several innovative features when compared with the former one, to improve both the training speed and test as well as the detection precision. Existing object detection networks like Fast R-CNN, depend on region proposal algorithms for predicting the object locations. The reduction of the running time for these networks highlighted the problem related to the region proposal calculations. Having this adversity in mind, Ren and Girshick, together with other authors, developed a Region Proposal Network (RPN), capable of sharing full-image convolutional features with the detection network, resulting in region proposals with a very reduced cost. RPN is a convolutional network that, simultaneously, scores and predicts object limits for each position. Furthermore, its training was end-to-end to generate region proposals of high quality, used by Fast R-CNN on object detection. In this way, the same authors combined RPN and Fast R-CNN in one single network, sharing their convolutional features, emerging the Faster R-CNN model [19].

Dai et al. [20] presented convolutional networks based on regions for precise and accurate object detection. By opposition to the previous detectors based on regions that apply a subnetwork hundreds of times per region with high costs, this detector named R-FCN: Region-based Fully Convolutional Networks, shares almost all calculations made on the entire image. The authors

[2] more details about this dataset at http://host.robots.ox.ac.uk/pascal/VOC/ [accessed 15 November 2019]

proposed score maps that are sensitive to a position to solve the translation invariance in image classification and the translation variance in object detection.

Liu et al. [21], presented a method for detecting objects in images using a single deep neuronal network, named as Single Shot Detector (SSD). When compared to other methods that make use of object proposals, this SSD is rather pure, since it eliminates the proposal generation and subsequent pixel or feature resampling stages, and performs all the computation with a single network, becoming easy to train and simple to be integrated with systems requiring a detection component.

On the other hand, Lin et al. [22] decided to study the feature pyramids, which stands for a primary component of the object detection systems in different scales. This approach was avoided by the present deep learning object detectors due to the required amount of memory and the required computation to improve object detection systems. The developed architecture, named as Feature Pyramid Network (FPN), showed a significant improvement as general feature extraction in several applications.

In the scope of Single Shot Detectors, Redmon et al. [23] developed YOLO: You Only Look Once. This neural network architecture is based on the GoogLeNet model [24], trained with the ImageNet classification dataset[3] and evaluated with PASCAL VOC detection dataset. It is capable of processing real-time images, with a frame rate of 45 fps, and representing a rectangle around the selected object. Later, the same authors developed a second version of this model, YOLOv2, and a different model named YOLO9000 [25], capable of detecting more than 9000 object categories and presenting several improvements when compared with the previous version. For detecting such a high number of objects, Redmon et Farhadi proposed a new training approach: an algorithm capable of conjugating detection and classification datasets.

As stated by Girshick et al., the present object detection datasets are limited when compared to the datasets directed for classification or tagging. The most common detection datasets contain thousands or even hundreds of thousands of images, with tenths or hundreds of tags. On the other hand, the classification datasets possess millions of images with tenths or hundreds of thousands of categories. The image tagging for detection is more difficult to achieve than the image tagging for classification. For that reason, the authors proposed a new method capable of taking advantage of a large amount of data already available to expand the scope of current detection systems. Thus, this pethood is capable of learning to locate an object with precision while it is using the classification images to increase its robustness and vocabulary. For this purpose, were used, simultaneously, the COCO detection dataset[4] and the ImageNet classification dataset [3].

More recently, the same authors developed the third version of YOLO, named YOLOv3 [26], with some improvements like the processing speed, the increase of correct detection precision, and the modification of the bounding boxes predicting mode. Around the same time, He and Girshick, along with others [11], developed the Mask R-CNN, an enhancement of Faster R-CNN, which predicts an object mask in parallel with the bounding box recognition.

[3] more details about this dataset at http://www.image-net.org/ [accessed 15 November 2019]
[4] more details about this dataset at http://cocodataset.org/#home [accessed 15 November 2019]

2.1.2 Methods' Analysis

According to the previously presented works, except for R-CNN and Mask-RCNN, in this section, we briefly compare some of the mentioned models. However, we should consider that the results came from different sources, meaning that the tests were probably not conducted precisely with the same settings.

We analyzed the models' results when tested with the following datasets: Pascal VOC 2007, Pascal VOC 2012, and COCO.

- In Table 2.1, we present the models' mAP and frame rate on Pascal VOC 2007 test set. All models were pre-trained with Pascal VOC 07+12, which means that their train was on the 2007 trainval and 2012 trainval.
- In Table 2.2, we present only the models' mAP on Pascal VOC 2012 test set, since there is no information about the frame rate. All models were pre-trained with Pascal VOC 07++12, which means that their train was on the 2007 trainval, 2007 test, and 2012 trainval.
- In Table 2.3 we also present only the models' mAP on COCO test-dev set. All models were pre-trained with COCO trainval dataset.

Mean Average Precision (mAP) is a metric used in measuring the accuracy of object detectors like Faster R-CNN, SSD, YOLO, etc. It computes the average precision value for recall value over 0 to 1. We mention the papers that presented the exhibited values in the last column of all tables. In all tables, we present two models of Faster R-CNN that used different feature extractors. The original one used VGG-16, and the model presented by Dai et al. [20] used ResNet-101. In table 2.1 we also presented other YOLOv2 versions. However, we do not show them in Figure 2.4.

Table 2.1: Model's mAP and frame rate comparison on PASCAL VOC 2007 dataset

Model	**mAP (%)**	**fps**	**Cited by**
Fast R-CNN	70.0	0.5	[18]
Faster R-CNN (VGG-16)	73.2	5	[19]
Faster R-CNN (ResNet-101)	76.4	2.4	[20]
R-FCN	80.5	5.8	
SSD 300	74.3	46	[21]
SSD 512	76.8	19	
YOLO	63.4	45	[23]
YOLOv2 288	69.0	91	[25]
YOLOv2 351	73.7	81	
YOLOv2 416	76.8	67	
YOLOv2 480	77.8	59	
YOLOv2 544	78.6	40	

Table 2.2: Model's mAP comparison on PASCAL VOC 2012 dataset

Model	mAP (%)	Cited by
Fast R-CNN	68.4	[18]
Faster R-CNN (VGG-16)	70.4	[19]
Faster R-CNN (ResNet-101)	73.8	[20]
R-FCN	77.6	
SSD 300	72.4	[21]
SSD 512	74.9	
YOLO	57.9	[23]
YOLOv2 544	73.4	[25]

Finally, in table 2.3, we present two columns for mAP, since the MS COCO challenge introduced a new evaluation metric. The mAP @.5 corresponds to the primary challenge metric, i.e., a threshold of 0.5, and it is equivalent to the measurement performed on the Pascal VOC dataset. The mAP @[.5,.95] averages over different Intersection Over Union (IOU) thresholds, from 0.5 to 0.95, step 0.05 (0.5, 0.55, 0.6, ... 0.9, 0.95). In this table, we did not present YOLOv1's results because the authors did not analyze this version on the COCO dataset.

Table 2.3: Model's mAP comparison on COCO dataset

Model	mAP (%) @.5	mAP (%) @[.5,.95]	Cited by
Fast R-CNN	39.3	19.3	[19]
Faster R-CNN (VGG-16)	42.7	21.9	
Faster R-CNN (ResNet-101)	48.4	27.2	[20]
R-FCN	51.9	31.5	
SSD 300	41.2	23.2	[21]
SSD 512	46.5	26.8	
YOLOv2 544	44.0	21.6	[25]

Figure 2.4 presents the corresponding mAP's plots of the previous three tables together. We did not include the mAP @[.5,.95] since these values cannot be compared with the remaining, as we previously explained. In Figure 2.5 we present the models' frame rate comparison on the Pascal VOC 2007 dataset, as shown in Table 2.1.

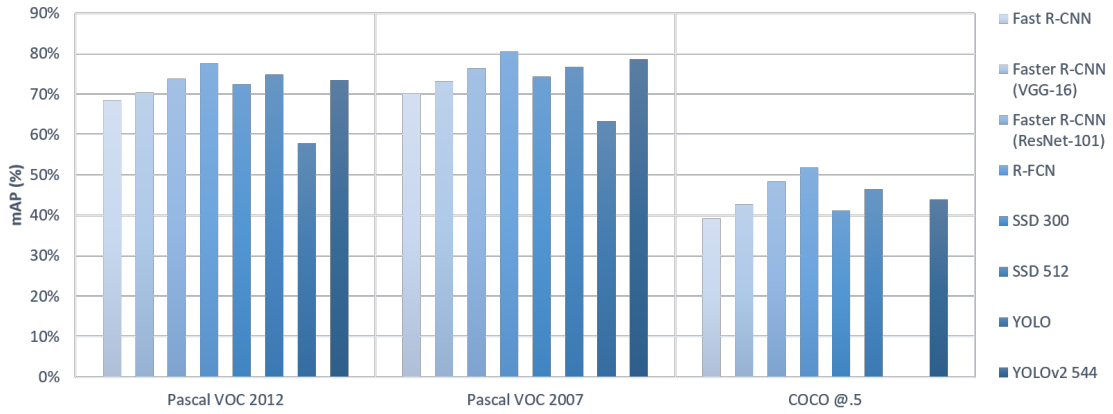


Figure 2.4: Model's mAP comparison

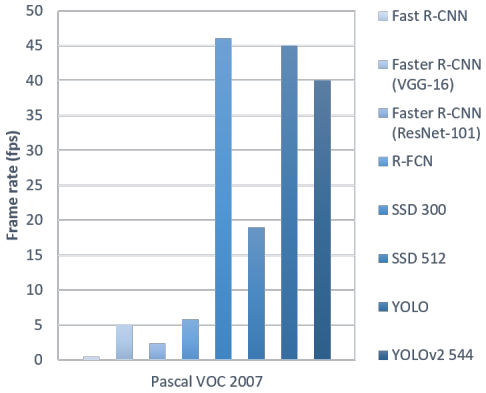


Figure 2.5: Model's frame rate comparison on PASCAL VOC 2007 dataset

The Pascal VOC 2007 dataset contains approximately 10k images and the corresponding annotations of 20 classes. As the Pascal VOC 2012 dataset is more recent than the previous one, it contains the same number of images and annotations and an additional 23k, approximately. Thus, the models' mAP tested with the oldest dataset are slightly higher than the latest. With similar reasoning, as the COCO dataset contains approximately 100k images and the corresponding annotations about 80 classes, the results are lower than the results with both Pascal VOC datasets. The mAP @ [.5,.95] averaged on the COCO dataset places a significantly larger emphasis on localization compared to the PASCAL VOC metric, which only requires an IOU of 0.5, the standard one. In other words, for the Pascal VOC 2007 dataset, the mAP was averaged over all 20 object classes and over 1 IOU threshold; and for the COCO dataset, it was averaged over all 80 object categories and all 10 IOU thresholds.

As we said before, the results came from different sources, which means that the tests were not conducted with the same settings and parameters. Nevertheless, with these results, we verify that R-FCN, followed by SSD 512, Faster R-CNN with Resnet-101, and YOLOv2 544, are probably

the best models. However, when we compare these models regarding the processing time, as shown in Figure 2.5, the SSD 300 and both YOLO models are considerably better than the R-FCN, the SSD 512 and the Faster R-CNN with Resnet-101.

We also considered the analysis made by Redmon on his website[5], that is constantly updated, as illustrated in Figure 2.6. The presented models were pre-trained with the COCO trainval dataset and tested on its test-dev. The plot (a) presents models' mAP, and the plot (b) shows the corresponding frame rate.

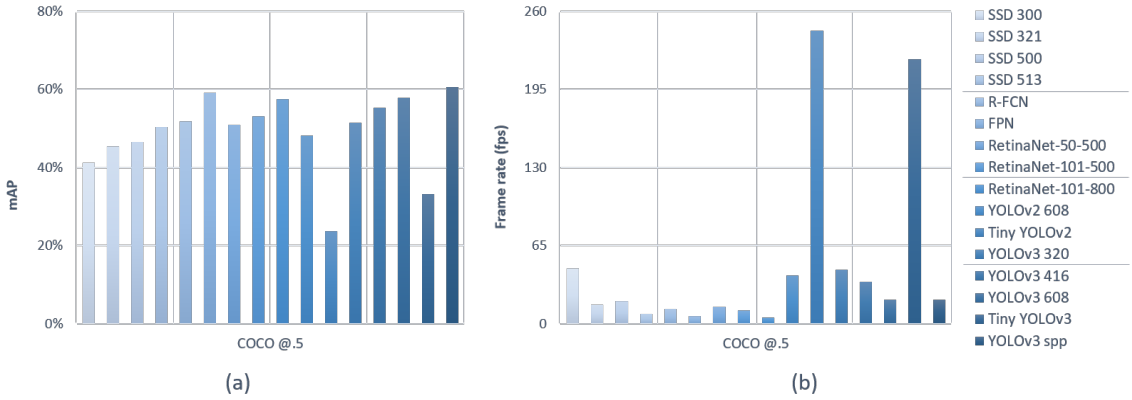


Figure 2.6: Model's mAP and frame rate comparison on COCO dataset [5]

In this case, all the models were trained and tested under the same conditions, which allows a valid conclusion. As we can see, the top 3 models with a higher mAP are YOLOv3 spp, FPN, and YOLOv3 608, followed by RetinaNet-101-800. On the other hand, the worst models regarding the mAP are Tiny YOLOv2 and Tiny YOLOv3. We compared the best models of this analysis with the best models of the previous one, and present the results in Figure 2.7. We verified that all the best models of Figure 2.6 are better than the remaining models of Figure 2.4.

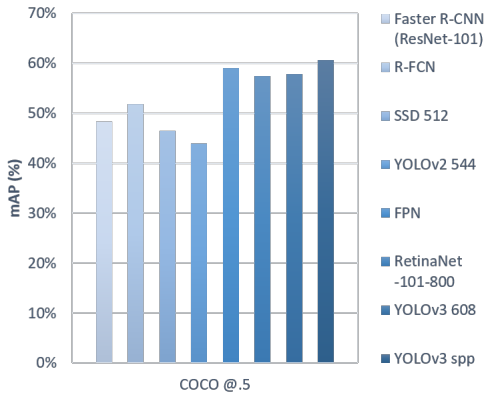


Figure 2.7: Best model's mAP comparison on COCO dataset

[5] more information about these models at https://pjreddie.com/darknet/yolo/ [accessed 27 November 2019]

This time, we also have information about the processing time, and, in contrast to what happened with mAP, the only two models that stand out considerably from the rest are Tiny YOLOv2 and Tiny YOLOv3. A system with a processing time very high is necessarily worse at the object detection part. Similarly, FPN and RetinaNet-101-800 have the worst processing time values.

2.1.3 Related Works

By further analyzing Redmon et al.'s developed models, one can find some practical examples of YOLO, such as a study performed by Mallick et al. [15] on detection and location of objects in overfilled environments using a robot. In this study, it was concluded that better results were achieved by a model with ResNet as the backbone instead of YOLO. Duggal et al. [27] also made use of YOLO for video analysis, as well as Zhou et al. [28] which have used it in the object detection and tracking in Human-Robot Interaction (HRI).

Kharchenko et al. [29] decided to use the YOLOv3 model for aircraft detection on the ground instead of SSD or RetinaNet. The model with less precision was SSD, YOLOv3 had a precision to a two-stage detector, and RetinaNet stood for the most effective detector in this group. However, YOLOv3 was the chosen one, due to the conjugated processed speed and acceptable precision results.

In the textile industry, namely on fault detection on a textile factory [30], three models based on the second version of YOLO were used and compared: YOLO9000, Tiny-YOLO, and YOLO-VOC. According to the analysis made, the first two are not adequate for this kind of fault detection due to the values obtained for the parameters under analysis. YOLO-VOC obtained the best result inclusively on the IOU parameter, although with a not very high mark for this parameter, probably due to the superposition of the fault's small size and the reduced frame selection area. However, when the IOU mark is approximately 50%, faults can successfully be detected and classified.

Wong et al. [31] developed a new model, inspired on the fire microarchitecture efficiency introduced by SqueezeNet [32] and on the single-shot microarchitecture object detection performance introduced by SSD. They create the Tiny-SSD model, a single shot detection deep convolutional neural network for real-time embedded object detection similar to Tiny-YOLO. The authors concluded that the size of their model is 2.3 MB, which is 26 times less than the size of Tiny-YOLO. Furthermore, Wong et al. verify that Tiny-SSD presented a mAP of 61.3% using Pascal VOC 2007 dataset, representing a 4.2% increase when compared with Tiny-YOLO.

2.2 Real-Time Object Tracking

As seen in Section 2.1, in the last years, the use of deep learning together with computer vision, improved object detection, and object classification. Consequently, visual tracking also using deep learning considerably increased its performance.

Visual tracking stands for a way of detecting, extracting, identifying, and locating objects in images or video sequences. This process starts with object detection in the video's first frame. Then, in the following frames, its position and attributes are followed. This technique is thus an

aggregation of different areas, such as image processing, pattern recognition, artificial intelligence, among others, meaning that it does not only have object detection, extraction, recognition, and location, but also a data organizing plan and decision support. The procedures performed in visual tracking are interdependent, since the way the object features are detected and extracted establish the method's efficiency and robustness.

Most of the best algorithms for image classification and object detection use deep learning. However, as a traditional computer vision task, visual tracking with deep learning started later and developed slower than other tasks. Even with the advantages of deep learning methods on feature extraction, complex video processing with deep networks makes it challenging to perform real-time tracking. Moreover, the lack of tracking samples limits supervised training since it requires a considerable number of them.

2.2.1 How Does A Tracking Algorithm Work?

In a video or a real-time transmission, while one frame is being processed, the tracking must be performed on the remaining received frames. In contrast to detection algorithms that are trained with many examples of an object, having more knowledge about it, tracking algorithms know more about the specific instance of the class they are tracking. So, first, an object must be detected to get its location and its dimensions. Then, with its speed and direction, the location of the object in the next frame can be predicted. Hence, while a detection algorithm always has to start from scratch, a good tracking algorithm uses all the provided information about the object to predict the new location.

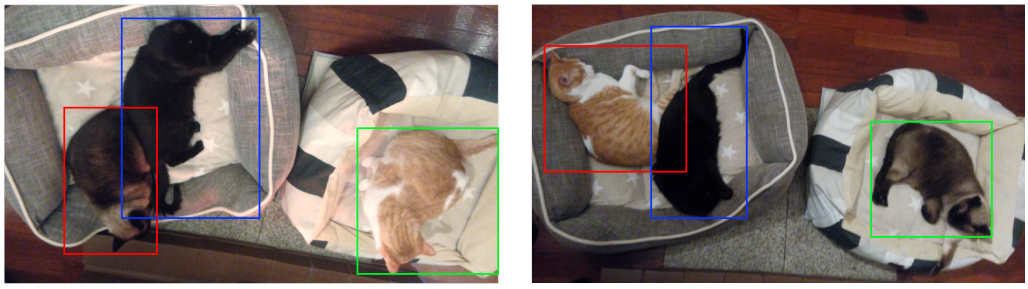
We cannot detect an object just once because an obstacle can overlap it for an extended period, or it can move so fast that the tracking algorithm cannot follow it. Hence, tracking algorithms usually accumulate errors, making the object's bounding box to drift away from the object that is being tracked slowly. Running a detection algorithm every so often can avoid these errors. Therefore, while designing an efficient system, usually object detection runs on every Nth frame while the N-1 frames in between employ the tracking algorithm. That is why, usually tracking algorithms are faster than detector algorithms.

With this processing, the tracker understands how the object moves. It means that the motion model is known. So, the new location of the object could be predicted based on the acquired information about the object speed and its direction of motion. Besides, the object's position and size gave the object's appearance in the previous frames, which is useful to search in a small neighborhood of the location predicted by the motion model to predict the location of the object more accurately.

As the appearance of an object can change dramatically, many modern trackers use a classifier model trained online to predict the new appearance of the object. The classifier should take a rectangular region of an image and classify it as an object or background. Thus, the classifier takes an image patch as input and verifies if it contains the object. If the object is there, it classifies the image patch with a 1; if it is guaranteed background, the assigned value is 0. The training of this model is online because it learns on the fly at run time. On the other hand, the training

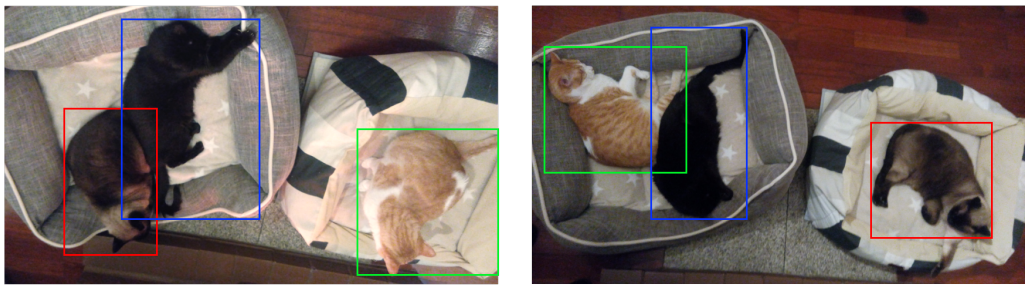
of a detection model is offline because it needs thousands of examples to learn and only a few examples are not enough.[6]

Another useful characteristic of tracking algorithms is that they can preserve the identity of the object. The algorithm responsible for drawing the bounding boxes in the images, uses the same color sequence regardless of the detected object, as shown in Figure 2.8. However, in (b), the Siamese cat should continue to show the red bounding box instead of the green one. On the other hand, with tracking, we obtain a similar result to the one presented in Figure 2.9, i.e., in (b) all cats still have the expected bounding box.



(a) (b)

Figure 2.8: Detection demonstration



(a) (b)

Figure 2.9: Tracking demonstration

2.2.2 What Are The Different Approaches Used By Trackers?

There are many types of trackers, but in this work, we decided to focus our research on the trackers available in the OpenCV library. Next, we briefly describe how they work, based on the work of Lehtola et al. [7].

The Boosting tracker is based on an online version of AdaBoost, and it is trained with the object's positive and negative examples at runtime. It considers the initial bounding box as a positive example and regions in the outside as the background. Next, the classifier runs on every

[6] more information about tracking at https://www.learnopencv.com/object-tracking-using-opencv-cpp-python/

pixel near the previous location, applying the binary classifier. The maximum value gives the new location of the object. As more frames come in, the classifier must be updated with the additional data.

The MIL tracker is like the previous one. But for classification, it uses the "Multiple Instance Learning" approach. Instead of considering only the object's current location as a positive example, it looks in a small neighborhood around it to generate several potential positive examples. MIL calls them "bags". It groups similar samples into bags, where it considers an overall positive sample if at least one of the individual samples it contains is positive, and negative otherwise. This approach is useful when the current location of the tracked object is not accurate because it is possible to have a positive neighborhood sample in the bag, which means that the object is in that location.

The Kernelized Correlation Filter (KCF) tracker, with a similar approach to the previous methods, takes advantage of the large overlapping regions caused by the multiple positive samples. The KCF employs the shift-invariance property of Fourier transform to simplify the correlation computation created by the overlapping data to make it extremely fast. The KCF avoids the inability to use a large enough number of training data available from each input frame due to a high computational load through its lightweight implementation.

The TLD tracker stands for Tracking, Learning, and Detection. The tracker must follow the object, while the detector localizes all appearances to correct the tracker if necessary. The learning part is responsible for estimating the detector's errors and updates the tracker to avoid them posteriorly. It avoids learning from misinformation, disabling online learning if the object is out of frame or wholly occluded by other objects. With this approach, this method is better than the previous ones when these situations occur. However, it has lots of false positives, which makes it not reliable.

The MedianFlow tracker's approach is based on tracking error measures. It tracks the points of each object both forward and backward in time and compares the resulting trajectories. Minimizing this error enables us to reliably detect tracking failures and select reliable trajectories in video sequences. The MedianFlow tracks and measures the points inside a bounding box for error, and then classifies them as inliers and outliers by the result. Outliers are filtered out, and the bounding box motion is estimated based on the inliers.

OpenCV library also has three more trackers available. GOTURN is the only one based on CNNs, and it uses a Caffe model for tracking. Minimum Output Sum of Squared Error (MOSSE) uses an adaptive correlation for object tracking. When initialized, it produces stable correlation filters using a single frame. Finally, the Discriminative Correlation Filter with Channel and Spatial Reliability (DCF-CSR) uses the spatial reliability map to adjust the filter support to the frame's selected region for tracking, improving the non-rectangular regions or objects' tracking and enlarging the selected region.

2.2.3 Tracker Comparison

Lehtola et al. [7] demonstrated that in terms of processed frames per second, MedianFlow is typically the fastest and KCF the second fastest, although the last one reaches very high performance on a few sequences. MIL and TLD are the slowest ones, and the Boosting algorithm sits in the middle. They concluded that only MedianFlow and KCF could perform at speeds that could be considered as real-time. The other ones can be used in a practical application reducing the video's resolution. MedianFlow and KCF allow the most significant resolution while maintaining real-time performance.

The same authors used the Jaccard Index as a measure of accuracy when comparing bounded boxes of the tracked object with the corresponding ground-truth. In this case, the MedianFlow has notably low accuracy. The rest share similar average accuracies, but KCF has the best average and can reach better accuracy than the others in the best cases. They also analyzed the Jaccard index as a function of a video frame. They concluded that MedianFlow presents a high frame rate and is the fastest one, but it shows a low accuracy, and it is not able to detect the target again once lost. Therefore, KCF has a more robust tracking accuracy while keeping the second-highest frame rate. At the time the paper was published, GOTURN had recently been introduced to the OpenCV library, so Lehtola et al. only referred to the results shown by its authors. They showed that the tracker could run at 165 fps on a GPU, but only 2.7 fps on CPU, a much lower value.

In terms of reliability, when tracking fails, Boosting and MIL are the worst. KCF has shown some improvements in this field, and MedianFlow is much better than the previous ones, especially if there was no occlusion. MIL and KCF trackers can handle partial occlusion, but they cannot recover from full occlusion, unlike TLD, which deals better with this.

GOTURN is robust to variations in lighting, scale, pose, and non-rigid deformations, as well as MOSSE. However, the first one does not handle occlusion very well. On the other hand, MOSSE tracker detects occlusion based upon the peak to sidelobe ratio (SLR), enabling the tracker to pause and resume where it left off when the object reappears. It also can operate at a high frame per second values, like 450 fps, and is accurate and fast, but on a performance scale, it lags behind trackers based on deep learning. Lastly, DCF-CSR can operate at a comparatively lower frame per second value (25 fps) but gives higher accuracy for object tracking.[6]

In Table 2.4, we present the approximate median value of the frame rate obtained by the analyzed trackers in work presented by Lehtola et al. And in Table 2.5, we show the accuracy results.

Table 2.4: Tracking frame rate (taken from [7])

	Trackers				
Tracking	Boosting	MIL	KCF	TLD	MedianFlow
	3.5 fps	1.5 fps	7.5 fps	1 fps	16.5 fps

Table 2.5: Tracking accuracy based on Jaccard Index (taken from [7])

	Trackers				
Jaccard Index	Boosting	MIL	KCF	TLD	MedianFlow
	0.35	0.36	0.40	0.30	0.20

2.2.4 Related Works

Due to the superiority shown concerning image recognition, CNNs became the mainstream model in computer vision and, consequently, in visual tracking. Feng et al. [33], mentioned two visual tracking algorithms based on deep learning: FCNT, a fully convolutional network tracker [34] and MD Net, a multi-domain convolutional neural network [35].

Ren et al. [36] proposed a collaborative deep reinforcement learning (C-DRL) model for multiple object tracking. The authors also developed a deep prediction-decision network with the purpose of simultaneously detect and preview objects under a unified network, since the most of the multi-object tracking methods make use of the strategy of tracking by detection. They start to detect objects in every frame and associate them to different frames, resulting in a limitation on the performance obtained in the detection phase, which can be affected by occlusions or overcrowded environments. So, the authors considered each object as an agent and performed its tracking through the prevision network, looking for the optimal tracked results found through the exploration of collaborative interactions of different agents and environments.

Still regarding object location prediction, Ehsani, Bagherinezhad, and Mottaghi, together with the leading authors from the three YOLO versions [37], developed DECADE a dataset of egocentric videos from a dog's perspective as well as its corresponding movements. The developed model gathers the existent visual information and directly predicts the agent's actions, i.e., how the dog is going to act and how to predict its movements. Moreover, the representation that is learned by this model codifies distinct information when compared with representations trained in image classification so that it can be generalized to other domains.

Scheidegger et al. [38] developed a multiple object tracking algorithm that receives an image and determines the trajectories of the detected objects in a world coordinate system. For that purpose, the authors used a deep neural network trained to detect and estimate the distance to the objects from a given image. The detection of a sequence of images is inserted in a state-of-the-art Poisson multi-Bernoulli mixture tracking filter, which in combination with the detector, results in an algorithm capable of performing 3D tracking just using images from a monocular camera as input. A mono-camera exclusively captures the image's bidimensional plan and thus, with no information regarding the object distances. The KITTI object tracking dataset [39] evaluates the model's performance in 3D world coordinates and 2D image coordinates. The authors obtained impressive results that proved the algorithm's efficiency, processing an average of 20 fps and the capability of performing precise object tracking, correctly manipulating data associations, even in the presence of image occlusion.

The previously mentioned dataset developed by Gaidon et al. [39] had as primary objective to aid modern computer vision algorithms to perform data acquisition and labeling. They took advantage of the growing computer graphics technology to create wholly labeled proxy virtual worlds, photorealistic, and dynamic. The purpose was to develop an efficient real-to-world cloning method and validate it through a new set of video data, named "Virtual KITTI", automatically labeled with precise information of the ground for object detection, tracking, scene segmentation, depth, and optical flux.

The same authors also presented quantitative experimental evidence suggesting that: i) modern deep learning algorithms pre-trained with real data behave similarly in real and virtual environments and ii) the pre-training made with virtual data improves the model's performance. Since the differences between virtual and real worlds are relatively small, the first allows measuring the impact of different meteorological and image conditions in the recognition performance. They also showed that these factors drastically affect high-performing deep models for tracking.

Held at al. [40] showed that, contrary to most of the object tracker that exist and perform their training online, it is possible to do real-time generic object tracking through visualizing offline videos of moving objects. To accomplish this goal, they created GOTURN, Generic Object Tracking Using Regression Networks, which, through offline training, learn to perform the tracking of new objects in a rapid, robust, and precise way.

Computer vision frequently uses machine learning techniques due to their capability of dealing with large quantities of data to improve systems performance. However, most of the generic object trackers are trained online from scratch, and thus they do not benefit from the number of offline available's videos for training purposes. The method developed by Held at al. is significantly faster than the previous methods that used neural networks for tracking. These networks are generally very slow and present small viability for real-time applications. GOTURN uses a simple feed-forward network without the need for online training. It learns a generic relationship between object movement and appearance that can be used for tracking new objects that are not present in the dataset. When in test and new objects are tracked, the network weights are freeze, and it is not necessary an online tuning. The network was tested in a standard tracking benchmark to score the method's performance and is capable of tracking objects at 100 fps.

Recently, Wang et al. [41] presented a new approach of efficient regression-based object tracking, which is related to the GOTURN tool. Inspired on the IC-LK tool, they created Deep-LK and proved that their tool substantially overcomes the tool created by Held at al., showing a comparable tracking performance to the current state-of-the-art deep trackers on high frame-rate sequences.

2.3 Virtual Environment Development

The development of a virtual environment based on a real-world scenario may be useful in several areas like architecture, construction, decoration, games, a collaboration between users, among others. The automatic creation of 3D virtual content in the context of a real-world scenario requires

image analysis where relevant features such as leading lines, point features, vanishing points or 3D structures can be detected. [42]

Some works developed in this area, use different features detection techniques than the previously discussed. Arnaud et al. [43] developed a mobile platform that uses a new tablet equipped with a depth sensor to rebuild a 3D real-time environment. This platform builds a model by identifying only four structural elements: floor, ceiling, walls, and windows/doors. Generally, the three-dimensional model of buildings is drawn manually by professionals using CAD software, a very exhaustive process, especially in large buildings. The main objective is to simplify the evaluation of the geometric characteristics of buildings and their energy performance. In this way, the use of depth sensors is considered a promising solution to accelerate and simplify the 3D modeling process, while improving the efficiency of the professionals in the design of these models.

The most popular sensors in 3D reconstruction are stereoscopic and Time of Flight cameras. These guarantee high resolution but require lots of processing power to extract the depth data, which makes it challenging to have a system that is powerful enough to operate in real-time without disrupting the user's mobility. Recently, a new generation of intelligent depth sensors for mobile devices, inspired by famous RGBD sensors such as Kinect, has emerged. Sensors like StructureSensor can be plugged into standard tablets, or Google Project Tango[7], a new generation of tablets with a built-in depth sensor allows for the development of several augmented reality applications and 3D modeling of real objects while providing low-cost mobile platforms. However, most RGBD sensors face some shortcomings, such as occlusion of regions by opaque objects, transparent surfaces, harmful reflection properties, and an inadequate distance to the sensor. For this reason, the authors of this platform, simplified the detection of windows/doors, considering that windows are transparent, and doors are always open, facilitating the selection of previously detected regions of interest when detecting the walls.

A few years earlier, when RGBD sensors were still starting to appear, Izadi et al. [44] used a standard Kinect camera to perform detailed reconstruction of existing interior space objects, creating KinectFusion. They only used the depth data obtained by the camera for tracking the 3D pose of the sensor and for a geometrically precise real-time reconstruction of 3D models of the physical environment. Besides, they developed a GPU based pipeline that demonstrated that object segmentation and user interaction in front of the camera did not interfere with the reconstruction, nor did degrade the tracking performed by the camera. Extensions added to the core GPU pipeline allow multi-touch interactions anywhere and in real-time.

Recently, Nóbrega et al. [42] developed an interactive mixed-reality application where virtual objects interact with images of real-world scenarios. The user only needs to capture images of the real environment with a simple camera or use pre-existing images. The introduction of virtual objects into photographs presents some challenges, such as pose estimation and creation of a visually correct interaction between virtual objects and the boundaries of the scene. The proposed system detects the scene automatically from an image, with additional features obtained through

[7] Tango Project was an AR computing platform, developed and authored by the Advanced Technology and Projects (ATAP), a skunkworks division of Google. In 2018, ARCore replaced it.

simple annotations to facilitate its usage by non-experienced users. Through the analysis of one or more images, the system detects the most relevant features, such as those described in the first paragraph of this subsection, to allow the creation of mixed and augmented reality applications in which the user can introduce fully integrated virtual objects in the environment's image in real-time. The principal motivation of this work was to create a system capable of mixing virtual content with real-world scenarios without using additional sensors, such as depth-sensing cameras, relying only on image recognition and automatic reconstruction of the scene.

Sra et al. [45] presented a new system capable of automatically generating immersive and interactive virtual reality environments using the real world only as a model. This system starts by capturing 3D interior scenarios, detecting obstacles such as furniture or walls, and mapping walkable areas. The main objective is, through the recognition of physical space, generate a virtual world with the same dimensions and characteristics, while representing a different scenario, to allow the user to walk through the real space with a completely different perception of reality.

The recognition and object tracking during the virtual reality experiment use additional depth data. The detected objects are paired with their virtual counterparts to make the virtual experience even more real, allowing a tactile experience with real objects with a different meaning in the virtual context. Sra et al. implemented this fully functional system in a Google Project Tango tablet [7].

Further work was developed to solve problems related to the occlusion of virtual objects integrated into physical spaces. It is critical to deal dynamically with occlusion to ensure realistic and immersive experiences, and to have correct depth perception in augmented reality applications. An existing solution is data storage; however, this requires the assumption of a static scene or high computational complexity. Thus, Du et al. [46] proposed an algorithm to improve depth maps for dynamic manipulation of occlusion in augmented reality applications. This algorithm uses an edge snapping approach, formulated as discrete optimization, capable of correctly defining the boundaries of objects according to RGB and depth data. This optimization problem runs efficiently on a tablet in near real-time.

Kasperi et al. [47] proposed an alternative method that uses smartphones without depth sensors to dynamically construct virtual buildings in the exterior without interfering with existing buildings. To do so, it uses geospatial data to construct geometric models of real buildings around virtual buildings. This method removes target regions from virtual buildings through masks constructed from real buildings. Although this method is not pixel perfect, which means that the simulated occlusion is not entirely realistic, the results indicate that the authors achieved their goal.

Another study by Walton and Steed [4] proposes a method that uses the color and depth data provided by an RGBD camera to provide better occlusion. It is possible to be executed in real-time and can manipulate the occlusion of virtual objects by dynamically moving objects, such as the hands of the user. This method is applied to individual RGBD frames and, therefore, can work in unfamiliar environments and respond appropriately to sudden changes. Figure 2.10 represents the occlusion performed by virtual objects integrated into a real environment. In (a), a virtual laptop

overlapping the real world is presented. In (b), it is possible to see some occlusion improvements provoked by the virtual object, and in (c), the virtual laptop and the whole scenario agree in terms of virtual and real objects.

Figure 2.10: Occlusion caused by virtual objects integrated in a real environment (taken from [4])

2.4 User Interaction and Collaboration

The rapid advances of AR technologies have accompanied the idea that the interaction between people located in different environments is possible in a space shared among all. Recently, telepresence based on the creation of a virtual avatar in which a user that is wearing a Head-Mounted Display (HMD) communicates with the image of another remote user superimposed on their physical space, has been increasingly explored.

In this context, Kim et al. [5] presented a new method capable of representing the movements of a human associated with a particular object in an avatar that is posteriorly associated with an object similar to the real one, as can be visualized in Figure 2.11. To achieve this goal, they developed a spatial map that defines the correspondences between any points in three-dimensional space around their objects. This spatial map can identify the desired locations of the avatar's body parts for any detected movement of the person, and, as its creation is offline, the redirection of the movements can be performed in real-time. The movement's redirection preserves essential features of the original movement, such as the human's position and the spatial relation to the object.

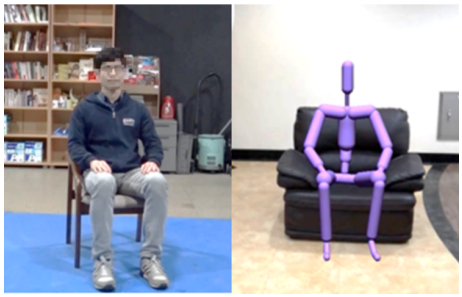


Figure 2.11: Human and its avatar counterpart (taken from [5])

In a completely different theme, Bonfert et al. [48] implemented a multiplayer mixed-reality game prototype in which they use a smartphone and an HMD in outdoor environments. Many virtual and mixed-reality games focus on single-player experiences. As such, these authors chose to develop a game that allows multiplayer interaction that forms teams to combat alien drones. Their implementation obtains the players' position relative to each other using GPS and relies on the rear camera of the smartphone to augment the environment and arm teammates with virtual objects. The combination of many factors such as multiplayer, mixed reality, geographic location, and group action outdoors using accessible mobile equipment allow for a new strategic and social gaming experience.

Casarin et al. [49] verified that the rapid evolution of VR, AR, and MR devices has significantly affected the maintenance and portability of applications. As such, they developed a model for designing AR, VR, or MR applications utterly agnostic of the device type used for its realization. To do so, they used freedom degrees to define the interaction limits between the tasks performed and the device.

Chapter 3

Proposed System Architecture

This chapter begins by presenting the problem and the corresponding solution proposal. Next, it describes the proposed system architecture, including the description of the different decisions taken concerning each of its constituent parts. Thus, the first section presents a characterization of the problems addressed by this work. The subsequent section exposes an overview of the solutions proposed to address these problems. Finally, the last section of this chapter describes all the constituents of this architecture and the reasons that motivated the decisions made.

3.1 Problem Characterization

Especially in a professional context, the possibility to collaborate remotely is an area that has been increasingly explored. International corporations require extreme communication between teams separated by thousands of kilometers; therefore, remote collaboration tools are essential. Moreover, at times like this, when a pandemic has been declared, prophylactic isolation as a preventive measure is indispensable. Being able to maintain social and professional activities through remote collaboration is a viable solution.

Countless companies allow their employees to work remotely, providing greater schedule freedom and comfort, but also a more significant responsibility in fulfilling and monitoring the proposed tasks.

Considering that a coworker is not physically present in the same place as the other team elements induce some drawbacks, such as the inability to interact beyond speech, sight, and hearing. Physically interacting with particular objects as if they were present in the same environment is an ambitious challenge that is increasingly desirable and can be easily extended to other areas.

This work focuses on developing a system capable of interacting with the real world and a virtual environment in real time. It emerged under the PAINTER project with the necessity of establishing a connection between both sides. The principal purpose is to perceive some objects from the real world and represent them in a specific location in the virtual environment. It requires the detection of some real objects in the real world to define a three-dimensional axis that relates

the two worlds. Afterward, the detected objects must be tracked so that the virtual environment stays up to date with the real world.

After the completion of the first two stages, i.e., (1) object detection and (2) object tracking, the (3) collaboration between users should be developed. One of the users must have a smartphone where he sees the real world and, through AR, perceives what the other user sees in the virtual environment. These three phases have very distinct goals and requirements and, thus, are handled in different ways.

3.2 Proposed Solution

To detect objects in the real world using a smartphone, we start by collecting frames from the device's camera and then apply preprocessing methods so that the images can be processed through a CNN. Due to their effectiveness in the acquisition of new information through feature extraction, as we mentioned in Section 2.1, we opted for this approach in the development of our work. The CNN model needs to be previously trained with a suitable dataset. Then, the output of CNN is analyzed to understand which and how many objects were found and where.

After an object has been detected and classified, its position coordinates are used to overlay some information in the device's screen, such as bounding boxes and a corresponding virtual representation of the object. From this moment on, the detected objects are efficiently followed with a tracking algorithm, ensuring that the users' perception is always as close as possible to reality.

For collaboration development, the shared three-dimensional axis is essential. So, it is required to determine the position of the detected objects in the real world relative to the origin coordinate of the defined axis. Hence, it is possible to find its position in the virtual environment. Then, it is required to establish communication between users, so that all relevant information in the real world gets translated to the virtual environment in real-time.

At an early stage, the detected object is treated as a mark, i.e., only the coordinates of its position relative to the environment and the camera are calculated to represent it in the parallel environment or the virtual space. After the conclusion of this step, the object's orientation should be computed to feature this data in its representation. However, this is out of the scope of this work. This extra information would allow the user to hold the object, move it, and rotate it so that the remote user can have a better perception of the interaction that is occurring. Finally, both users would be able to, in different environments, view and interact with the object and with each other.

The biggest challenge of this work and what differentiates it from other work already done is the complex integration of all the elements that take part in this cooperation environment. Like in a puzzle, we must find out the best way to fit the pieces correctly, as shown in Figure 3.1. Firstly, one of the main requirements is the development of an application that works in real-time. As such, one option is to use a tool that can speed up all the processing required as a Hardware Acceleration SDK, which implies a selection of a compatible AI framework for object detection and, consequently, a CNN model based on it. Also, not every smartphone is compatible with

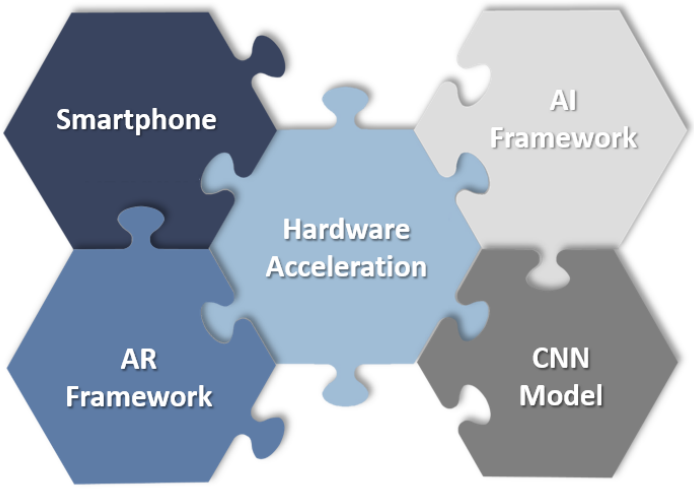


Figure 3.1: Integration of the required components

the chosen HA SDK and with the AR framework. Furthermore, as this work should run on a smartphone, it is also required to obtain all the necessary shared files that allow these frameworks and SDKs to work together.

3.3 Architecture

We chose to develop this work for Android devices due to the market share dominance of its Operating System (OS). Our application splits into two parts: Computer Vision (CV) and Artificial Intelligence (AI), on the right side and left side of Figure 3.2, respectively. In the CV component, we capture the camera image of the chosen Android device using ARCore. Next, we implement image processing, applying the OpenCV library.

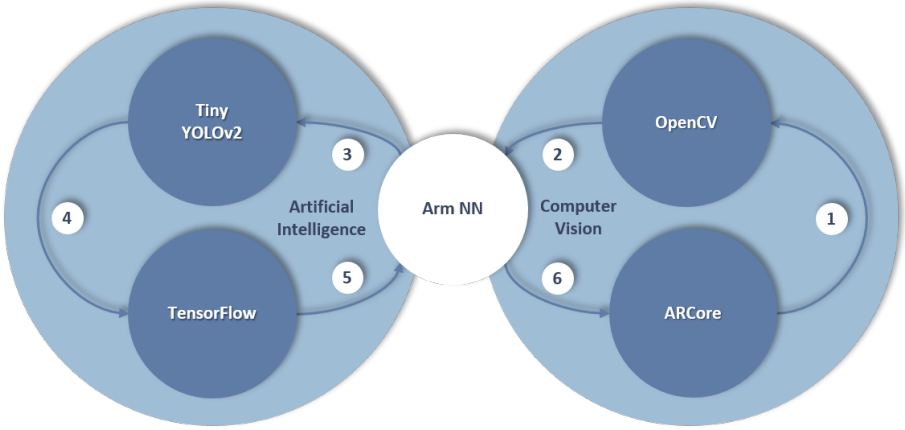


Figure 3.2: System architecture

On the left side of the image, we prepare the image data to agree with the chosen CNN model requirements, Tiny YOLOv2. We accomplish all the processes associated with CNN using Tensorflow, the selected AI framework. Finally, we use ArmNN SDK as an attempt to accelerate the machine learning computations on the mobile device, connecting both sides. In the following subsections, we describe all these components and explain the reasons why they were selected.

3.3.1 ARCore (AR Frameworks)

Nowadays, AR technologies are continually changing and evolving. To keep with the pace, developers rely on AR SDKs or dedicated frameworks. They provide a coding environment where users can create and implement all functionalities of their AR applications. These frameworks are helpful because they also offer a set of tools, libraries, relevant documentation, code samples, and guides to develop applications on specific platforms, known as Integrated Development Environment (IDE). For the development of this work, we opted to use Unity, which is the most popular AR/VR/MR platform.

There are many SDKs and AR frameworks available, but we decided to focus our research only on a few of them that are supported by Unity. The best known are ARCore, ARKit, and Vuforia, which facilitate some components that can be useful in the development of the application, for instance: recognition, tracking, and content rendering.

ARCore[1], is Google's platform for building AR experiences. It was designed to work on a wide variety of qualified Android smartphones running Android 7.0 (Nougat) and later. ARCore can also work on iOS but requires an ARKit compatible device running iOS 11.0 or later. It enables the chosen gadget to understand the surrounding world and interact with the information in it. ARCore has some fundamental concepts to integrate virtual content with the real world, as seen through the phone's camera, which we briefly described in Figure 3.3.

In the same way, ARKit[2] is a platform for building AR experiences developed by Apple. It is a direct competitor of ARCore because both do almost the same things, but ARKit can only work with the iOS operating system. It combines device motion tracking, camera scene capture, advanced scene processing, and display conveniences to simplify the task of building an AR experience. In Figure 3.4, we show the main concepts of this platform.

[1] more information about this framework at https://developers.google.com/ar/discover/concepts [accessed 12 October 2019]

[2] more information about this framework at https://developer.apple.com/augmented-reality/ [accessed 12 October 2019]

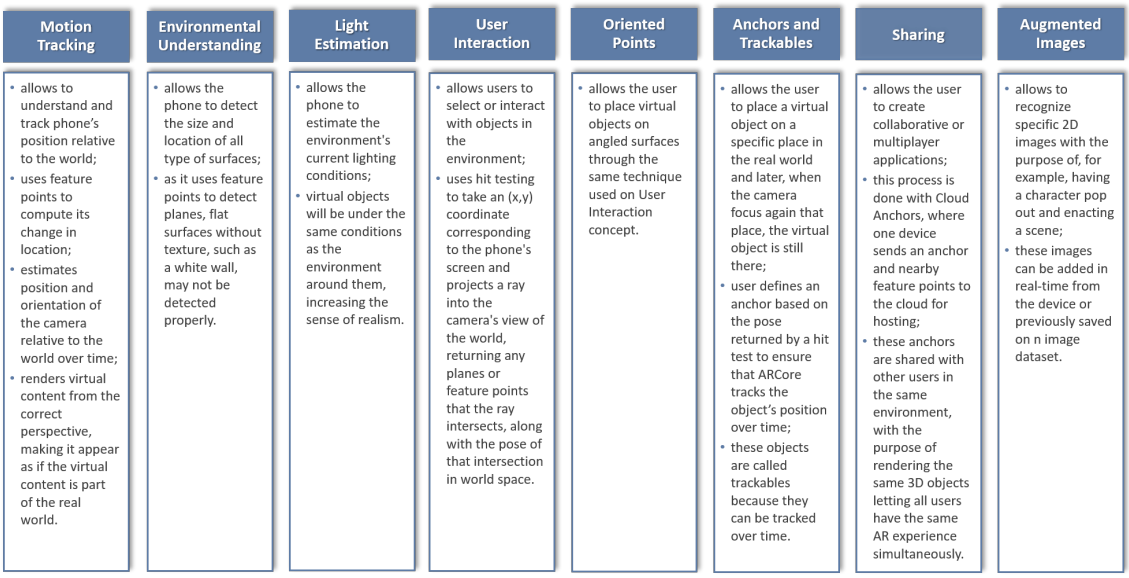


Figure 3.3: ARCore main concepts description

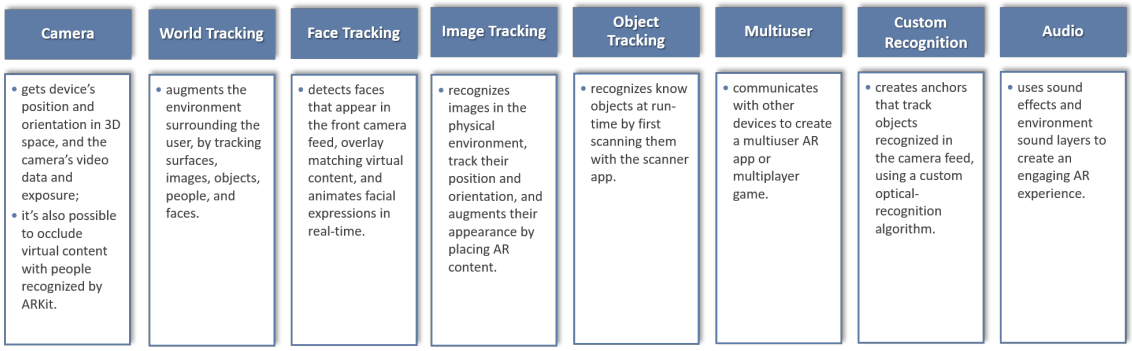


Figure 3.4: ARKit main concepts description

Vuforia Engine[3] is the most widely AR development platform used to create Android, iOS or Windows apps with advanced computer vision functionalities to obtain AR experiences that realistically interact with objects and the environment. Currently, the main advantage of Vuforia when compared with ARCore or ARKit is that it can run on any smartphone with a rear camera. This SDK does not depend on the smartphone's OS or CPU. Despite having several features similar to the other frameworks in terms of detection and tracking, Vuforia is based on image targets, i.e., simple images with non-uniform patterns to facilitate feature point detection. This capability

[3] more information about this framework at https://developer.vuforia.com/ [accessed 12 October 2019]

requires less computing power. It can also recognize elementary 3D objects, but it is limited when compared with ARCore and ARKit.

For all these reasons, and since we decided to use an Android smartphone, our choice for the development of our application was the ARCore framework integrated with Unity.

3.3.2 OpenCV

OpenCV is an open-source computer vision and machine learning software library, built to accelerate the use of machine perception in commercial products and to provide a common infrastructure for computer vision applications.[4] This library is helpful in image processing and, in this work, we used it in the following parts:

- Converting camera image from YUV to RGB: the camera image from ARCore has its data accessible from the CPU in YUV-420-888 format. However, the image input required by the CNN model is RGB.
- Resizing camera image: another requirement from the CNN model is specific image size.
- Drawing bounding boxes: to test our application and analyze the obtained results, we draw a rectangle over the image. When the app is running, we do not use this tool because we want to represent an augmented bounding box on the smartphone screen.
- Tracking: to implement a tracking system. In the most recent versions of this library, there are available the following trackers: Boosting, MIL, KCF, TLD, MedianFlow, GOTURN, MOSSE, and CSRT. The details of these trackers were explored in Section 2.2. For our work, we chose the KCF tracker based on the results presented in Section 5.2.

3.3.3 ArmNN (Hardware Acceleration SDKs)

Considering acceleration solutions [6] is fundamental to achieve a real-time result in object detection and tracking. Over the last years, some innovations in computer processing capabilities have emerged. Initially, computers were mostly equipped with a single, stand-alone, CPU, but it was soon evident that the computational performance was too limited for running multiple applications. Special co-processors were created, with optimized architecture for many signal processing tasks, to work in parallel with the main CPU, the Digital Signal Processors (DSPs).

DSPs were advantageous with, for instance, applications related to computer graphics, sound, video decoding, and even for running the first deep learning Optical Character Recognition (OCR) models. However, at the end of the millennium, their popularity started to decrease. They began to be replaced by CPUs with integrated DSPs instructions, GPUs for efficient parallel computations, and FPGAs.

[4] more details about this library at https://opencv.org/

Unlike what happened with desktop computers, at the beginning of the 1990s, DSPs started to appear in mobile phones, and they were not replaced by CPUs and GPUs, because they often offered superior performance at lower power consumption, which is useful for portable devices.

In recent years, the computational power of mobile DSPs and other System on Chip (SoC) components has grown drastically. Consequently, complemented by GPUs, Neural Processing Units (NPUs), and dedicated AI cores, they enable AI and deep learning-based computations. Following this, some companies developed mobile SoCs with potential acceleration support for third-party AI applications. The four most prominent companies that we are going to talk about are Qualcomm, Huawei, MediaTek, and Samsung, as shown in Figure 3.5.

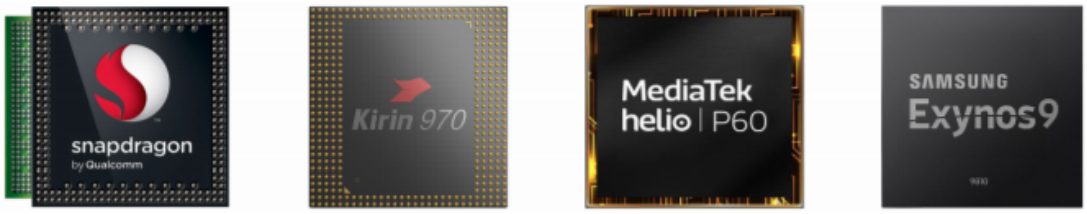


Figure 3.5: Mobile SoCs with potencial acceleration support for AI applications (taken from [6])

Qualcomm One of the leading companies dedicated to developing mobile SoCs is Qualcomm. Currently, about 55% of the smartphone SoC market uses Qualcomm chipsets, known as Snapdragon. There are different SoC components integrated into Snapdragon chipsets, like CPU and GPU. The primary CPU cores are based on the Arm architecture and usually have a custom design by Qualcomm itself, also based on Arm Cortex cores. They additionally develop their GPU, named Adreno, instead of using Arm Mali, like other companies.

Qualcomm implemented HA for AI computations for the first time in Snapdragon 820 in 2015. One year later, Qualcomm announced its proprietary Snapdragon Neural Processing Engine (SNPE) SDK, which offers runtime acceleration across all Snapdragon's processing components. The SDK supports common deep learning model frameworks, such as Caffe, Tensorflow, among others, and it was designed to enable developers to run their own custom neural network models on various Qualcomm-powered devices.

Huawei Another big company in this area is HiSilicon, a Chinese semiconductor company founded in 2004 as a subsidiary of Huawei. Unlike Qualcomm, HiSilicon does not create customized CPU and GPU designs, so all Kirin chipsets are based on off-the-shelf Arm Cortex CPU cores and various versions of Arm Mali GPUs.

For accelerating AI computations, HiSilicon has developed a different approach. They introduced a specialized NPU aimed at fast vector and matrix-based computations widely used in AI and deep learning algorithms, instead of relying on GPUs and DSPs. To give external access to Kirin's NPU, Huawei released in 2017 the HiAI Mobile Computing Platform SDK, which also

supports Caffe, and TensorFlow Mobile and Lite frameworks. However, these chipsets are only available for Huawei devices.

MediaTek MediaTek's approach is like HiSilicon in terms of CPU and GPU design but sometimes replaces Mali GPUs with PowerVR graphics. In 2018, MediaTek launched its Helio P60 platform, an embedded AI Processing Unit (APU). Their approach lies in between the solutions from Huawei and Qualcomm: a dedicated chip for quantized computations (as in Kirin's SoC) and CPU/GPU for float ones (as in Snapdragon chipsets).

At this time, MediaTek also introduced NeuroPilot SDK constructed around Tensorflow Lite and Android Neural Network Application Programming Interface (NNAPI). Nonetheless, the SDK only supports purely MediaTek NeuroPilot-compatible chipsets (currently on their SoCs only), and, beyond this, only one smartphone in the whole current market uses Helio P60.

Samsung There is still one more big company dedicated to mobile SoCs development, Samsung. Initially, they chose to use Arm Cortex CPU cores and Arm Mali GPU, but in the eighth generation of Exynos chipsets, their in-house developed Mongoose Arm-based CPU cores were integrated into high-end SoCs. Unfortunately, no drivers, SDKs, or additional details were released by Samsung, making it inaccessible by third-party applications.

Arm Currently, all CPU cores integrated into mobile SoCs are based on the Arm architecture, and they are responsible for running all AI algorithms when there is no support for HA. Considering that not all devices support HA for machine learning applications, in the last few years, Arm has developed different approaches to deal with this issue. One of them was ArmNN[5], an open-source SDK to speed-up the computations.

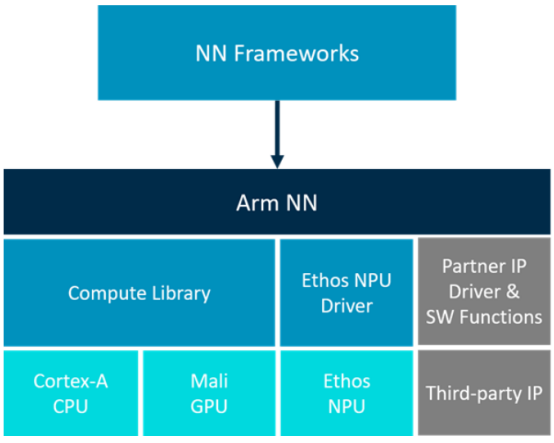


Figure 3.6: ArmNN connection with NN framework and the existing hardware [5]

[5] more details about this SDK at https://developer.arm.com/ip-products/processors/machine-learning/arm-nn [accessed 8 October 2019]

ArmNN SDK provides a bridge between existing neural network (NN) frameworks, such as Tensorflow and Caffe, and power-efficient Arm Cortex-A CPUs, Arm Mali GPUs, and Arm Ethos NPUs, as shown in Figures 3.6 and 3.7.

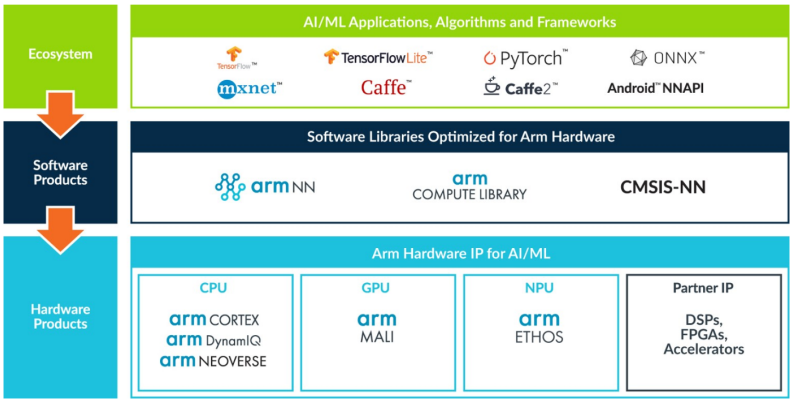


Figure 3.7: ArmNN SDK [5]

ArmNN takes networks from these frameworks, translates them to the internal ArmNN format, and then, through the Compute Library, deploys them efficiently on Cortex-A CPUs, or, if present, Mali GPUs, such as the Mali-G71 and Mali-G72. Although the ArmNN is compatible with Cortex-A CPUs, it does not provide support for Cortex-M CPUs. Figure 3.8 (a) shows the integration of ArmNN on our smartphone at high-level.

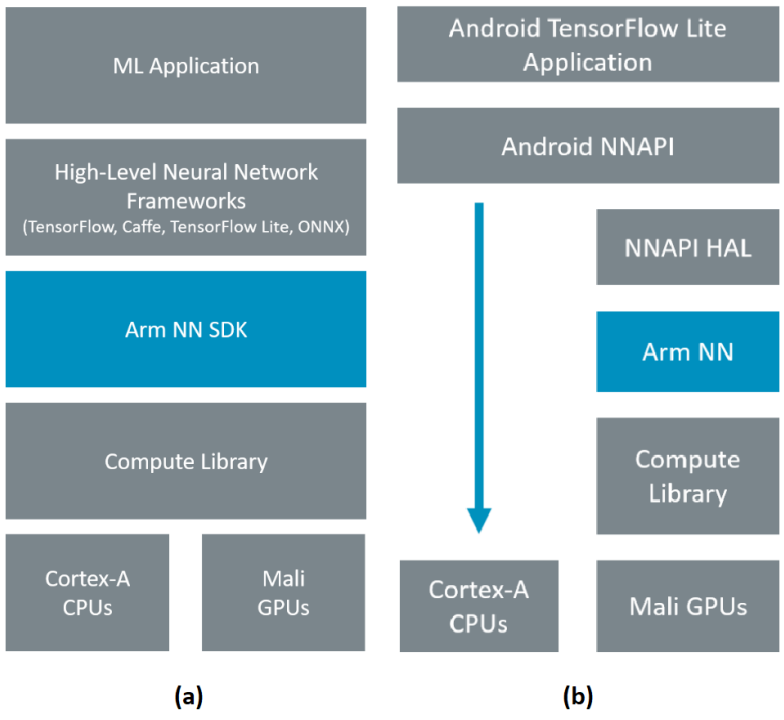


Figure 3.8: ArmNN approach [5]

ArmNN is also available for Android NNAPI, a Google's interface for accelerating neural networks on Android devices only available for the Android O version. Google has introduced Android NNAPI as an Android C Application Programming Interface (API) designed for running computationally intensive machine and deep learning operations on mobile devices. They want to solve a common problem present in different Hardware Acceleration SDKs, which only provide access to specific chipsets, and are additionally incompatible with each other.

By default, NNAPI runs neural network workloads on the device's CPU cores but also provides a Hardware Abstraction Layer (HAL) that can target other processor types, such as GPUs. ArmNN for Android NNAPI can provide this HAL for Mali GPUs. Figure 3.8 (b) shows the integration of ArmNN on our smartphone at low-level. A further release adds support for the Arm Machine Learning processor. This support for Android NNAPI gives more than a 4x performance boost.

After this analysis, we chose ArmNN SDK to provide Hardware Acceleration to our application. In Table 3.1, we present the selected mobile devices for this work, their chipsets, CPU, and GPU, to compare their performance and the obtained results with different specifications. One of the reasons that made us choose these devices was because all of them are compatible with ARCore.

Table 3.1: Devices specifications

Name	**Chipset**	**CPU**	**GPU**
OnePlus 6	Snapdragon 845	Kryo 385	Adreno 630
Samsung Galaxy S9 Plus	Exynos 9810	Mongoose + Cortex-A55	Mali-G72
Huawei Mate 20 Lite	HiSilicon Kirin 710	Cortex-A73 + Cortex-A53	Mali-G51
OnePlus 7 Pro	Snapdragon 855	Kryo 495	Adreno 640
Samsung Galaxy S10e	Exynos 9820	Mongoose M4 + Cortex-A75 + Cortex-A55	Mali-G76

As we chose smartphones from diverse brands or, at least, distinct models, the chipset present in each of them belongs to different companies, as mentioned above. We selected two different generations of smartphones from OnePlus and from Samsung to understand if there was any generational improvement.

Both OnePlus devices have a Qualcomm chipset, which has a CPU and a GPU developed by them. However, only the Adreno series GPUs are fully developed by Qualcomm, both Kryo 3xx and Kryo 4xx series CPUs have an Arm-based architecture. The first one features semi-custom Gold and Silver cores derivative of Arm's Cortex-A75 and Cortex-A55, respectively, and the second one features Gold Prime/Gold and Silver cores derivative of Arm's Cortex-A76 and

Cortex-A55, respectively as well. For these two options, it's possible to use Qualcomm's SNPE SDK or ArmNN SDK.

In the case of Samsung Galaxy S9 Plus and Samsung Galaxy S10e, both CPU and GPU belong to Arm. Samsung designs the Mongoose CPU part, but it is also based on the Arm architecture. So, both components of Exynos chipset can work with ArmNN SDK. Similarly, Huawei Mate 20 Lite chipset is composed of an Arm CPU and GPU, so it's also possible to run ArmNN or their SDK, HiAI.

As all the chosen mobile devices are compatible with ArmNN SDK, and not all of them can run SNPE SDK or HiAI SDK, we assumed that using ArmNN SDK and their libraries was the best option for the development of this work.

3.3.4 Tiny-YOLOv2

Following the analysis in Section 2.1 on existing detection and classification methods, we decided to focus on YOLO for this work. Contrary to prior detection systems that repurpose classifiers or localizers to perform detection, YOLO uses an entirely different approach. These methods apply the model to an image at multiple locations and scales to detect the highest scoring regions. On the other hand, YOLO applies a single neural network to the full picture, which divides it into parts and predicts bounding boxes and probabilities for each one. The predicted probabilities weight these bounding boxes to find out which ones are relevant detections.

This approach has several advantages over classifier-based systems, such as looking at the whole image at test time to ensure that predictions are made based on the global context of the image and making predictions with a single network evaluation. For instance, R-CNN requires thousands of networks for evaluating a single figure, which makes it 1000x slower than YOLO. Even Fast R-CNN is 100x slower than YOLO, which is an essential feature for our goals on real-time detection and tracking.

YOLOv2 was built as an improved version of YOLOv1 with better training and increased performance. Like SSD, this new version uses a fully convolutional model. Similar to Faster R-CNN, it adjusts priors on bounding boxes, obtained from the dataset annotations, instead of predicting the width and height outright, both presented in Section 2.1. However, YOLOv2 still predicts the x and y coordinates directly. To increase detection speed, Tiny-YOLOv2 was created, but as expected, accuracy decreased. Recently, it was developed a new version. YOLOv3 brought training improvements, increased performance, multi-scale predictions and a better backbone classifier[6].

We made an analysis of which models could be used on our application. Through a tool provided by TensorFlow, we find out which operations are carried out in each of the networks. Then, we realized that only the Tiny-YOLOv2 model was compatible with the ArmNN SDK since all the others required some functions not currently supported by it. Based on YOLO's official website, we present in Table 3.2 the available models and their expected mAP.

[6] more information about this model at https://pjreddie.com/darknet/yolo/ [accessed 23 October 2019]

Table 3.2: YOLO versions' specifications

Name	Training Dataset	Test Dataset	mAP (%)	fps
YOLOv1.0	-	-	-	-
Tiny-YOLOv1.0	-	-	-	-
YOLOv1.1	Pascal VOC 07+12	-	-	45
Tiny-YOLOv1.1	Pascal VOC 07+12	-	-	155
YOLOv2	Pascal VOC 07+12	Pascal VOC 07	76.8	67
Tiny-YOLOv2	Pascal VOC 07+12	Pascal VOC 07	57.1	207
YOLOv2	COCO trainval	COCO test-dev	48.1	40
Tiny-YOLOv2	COCO trainval	COCO test-dev	23.7	244
YOLOv3	COCO trainval	COCO test-dev	55.3	35
Tiny-YOLOv3	COCO trainval	COCO test-dev	33.1	220

Unfortunately, we were forced to choose the model with the worst displayed value, although it has the highest frame rate. The blank cells mean that there was no information about that model's performance. However, we had access to their configuration file and their weights, so we analyzed the operations performed by them as well. In Appendix A, we present the analysis of each model and the respective network design.

This version of YOLO [50] requires an input image divided into an *SxS* grid of cells, where *S* is the size, which in this model is 13. Each grid cell predicts *B* bounding boxes. And each bounding box predicts *N* class probabilities, i.e., the probability of each object exists in that bounding box. For this model, *B* is 5 and *N* is 20, as there are 20 classes to be detected. The correct way to calculate the confidence scores of each class ($cs(object)$) is to multiply the probability of being that class by the IOU, that represents a fraction between 0 and 1, as shown by Equation 3.1:

$$CS(class) = P(class) * IOU^{truthpred} \tag{3.1}$$

The intersection, represented in Figure 3.9 (a), is the overlapping area between the ground-truth (A) and the predicted bounding box (B). The union, illustrated in Figure 3.9 (b), is the total area between both. Preferably, the IOU must be close to 1, to indicate that the predicted bounding box is near to the ground-truth.

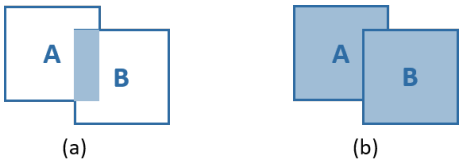


Figure 3.9: IOU - (a) Intersection; (b) Union

The bounding box prediction has 5 *Components*: x, y, width, height, and confidence. Therefore, the output size is 21125, according to Equation 3.2:

$$S * S * (B * (5 * Components + N)) \tag{3.2}$$

The (x, y) coordinates represent the center of the bounding box, relative to the grid cell location, and are normalized to be between 0 and 1. The bounding box dimensions, width and height, are also normalized to the same interval. In this context, the loss function is used to correct the center and the bounding box of each prediction, and YOLO uses Equation 3.3 to calculate loss and ultimately optimize confidence:

$$\begin{aligned}
&Loss = \\
&\lambda_{coord} \sum_{i=0}^{s^2} \sum_{j=0}^{A} 1_{ij}^{obj} [(b_{x_i} - b_{\hat{x}_i})^2 + (b_{y_i} - b_{\hat{y}_i})^2] \\
&+ \lambda_{coord} \sum_{i=0}^{s^2} \sum_{j=0}^{A} 1_{ij}^{obj} [(\sqrt{b_{w_i}} - \sqrt{b_{\hat{w}_i}})^2 + (\sqrt{b_{h_i}} - \sqrt{b_{\hat{h}_i}})^2] \\
&+ \sum_{i=0}^{s^2} \sum_{j=0}^{A} 1_{ij}^{obj} (C_i - \hat{C}_i)^2 \\
&+ \lambda_{noobj} \sum_{i=0}^{s^2} \sum_{j=0}^{A} 1_{ij}^{noobj} (C_i - \hat{C}_i)^2 \\
&+ \sum_{i=0}^{s^2} 1_{i}^{obj} \sum_{c \in classes} (CS_i(c) - \hat{CS}_i(c))^2
\end{aligned} \tag{3.3}$$

In this equations, the *bw* and *bh* refer to the bounding box dimensions, while *bx* and *by* variables refer to the center of each prediction. The $\lambda coord$ variable emphasizes boxes with objects, and $\lambda noobj$ variable depreciates boxes with no objects. $CS(c)$ refers to the classification prediction of each object, and C refers to the confidence. The 1_{ij}^{obj} is equal to 1 if the j^{th} bounding box in the i^{th} cell is responsible for the object's prediction, and 0 otherwise. 1_{i}^{obj} is equal to 0 if the object is not in cell i, and 1 if it is present. The loss indicates the model's performance. Therefore, a lower loss indicates higher performance.

While the loss is used to train a model, the accuracy of predictions in object detection is calculated through the average precision equation shown below:

$$avgPrecision = \sum_{k=1}^{n} p(k) * \Delta r(k) \tag{3.4}$$

Here, $\Delta r(k)$ refers to the change in a recall, while $p(k)$ refers to the precision at threshold k. The precision measures how accurate are predictions and the recall measures how accurate are the positive results.

The YOLO's NN architecture contains twenty-four convolutional layers and two fully connected layers. Versions 2 and 3 of YOLO can minimize localization errors and increase mAP. As

seen in Figure 2.6, Tiny-YOLOv2 has a mAP of 23.7% and the lowest Floating-point Operations Per Second (FLOPS) of 5.41 billion. However, according to Rachel Huang et al., when Tiny-YOLOv2 runs on a non-GPU laptop (in that case, Dell XPS 13), the model speed decreases from 244 fps to about 2.4 fps. With this constrain, real-time object detection is not easily accessible on many devices without a GPU, such as most cellphones or laptops.

3.3.5 TensorFlow (or AI/DL Frameworks)

Currently, there are many machine learning frameworks available for developing applications in this area. In Figure 3.10, we will describe some of them to explain why and how we decide which one was the best AI framework for this work[7].

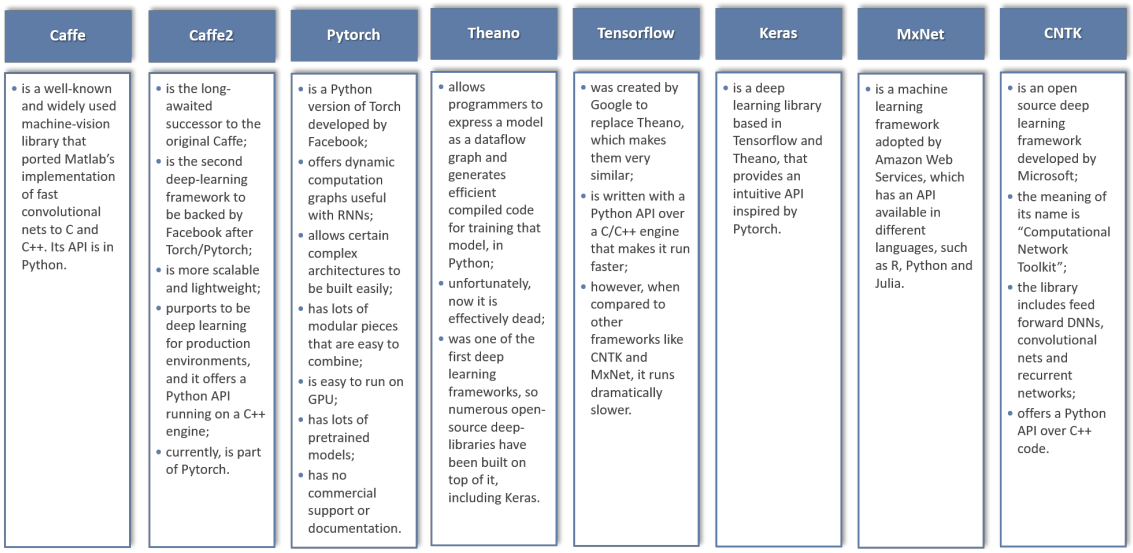


Figure 3.10: AR Frameworks characteristics

As shown in Figure 3.1, the choice of the AI framework was directly related to the chosen detection model and the Hardware Acceleration SDK. Since we decided to use Tiny-YOLOv2, we had to analyze which AI frameworks can work with it. All YOLO versions are based on Darknet, an open-source neural network framework written in C and CUDA. However, this framework is incompatible with the chosen Hardware Acceleration SDK. For instance, there are implementations of YOLO in Caffe, but the scarce community support made the cross-compile part for the smartphone complex. Although Pytorch and MxNet are compatible with both requirements, there is even less documentation and support for their use. As mentioned in Figure 3.10, Theano is deprecated. The HA SDK does not support Keras and CNTK. Therefore, we decided to use

[7] more information about these frameworks at https://pathmind.com/wiki/comparison-frameworks-dl4j-tensorflow-pytorch [accessed 25 October 2019]

TensorFlow[8], despite it being considerably slower than other frameworks. However, TensorFlow does not work directly with YOLO models. So, we needed to implement this integration using Darkflow[9], which acts as a middleware between Darknet[10] and TensorFlow.

[8] more information about this framework at https://www.tensorflow.org/ [accessed 26 October 2019]

[9] more details about this framework at this Github repository: https://github.com/thtrieu/darkflow [accessed 26 October 2019]

[10] more details about this framework at this Github repository: https://github.com/pjreddie/darknet [accessed 26 October 2019]

Chapter 4

Implementation

In this chapter, we explain how this work was implemented. It is divided into four sections to clarify all the steps performed. In the first section, we explain how the Unity Scene was developed for the integration of every part, as we had shown in Figure 3.1 in the previous chapter. The second section is about CNN input preparation, which includes image processing and network creation. In the third section, we explain how the frames pass through CNN and how we parse the output to represent the obtained bounding boxes on the smartphone screen. Finally, the fourth section addresses the user's collaboration, including object tracking and the constant sharing between both devices to maintain the system synchronized in real-time.

4.1 Unity Scene Development

As previously mentioned, we opted to use the Unity Engine to develop our application. The first step was to create a scene to build the entirety of the work. This main scene is composed of six elements, known as Game Objects, as shown in Figure 4.1 and described below.

- ARCore Device - manages the ARCore session and holds the main camera, responsible for capturing real-world images through the back camera of the Android device and making them available as the AR scene's background.

- Environmental Light - is responsible for adjusting the lighting in the AR scene.

- Point Cloud - manages the feature points detected in the current frame captured by the main camera.

- Session Controller - manages the background session, responsible for the basic actions, such as quit and sleep.

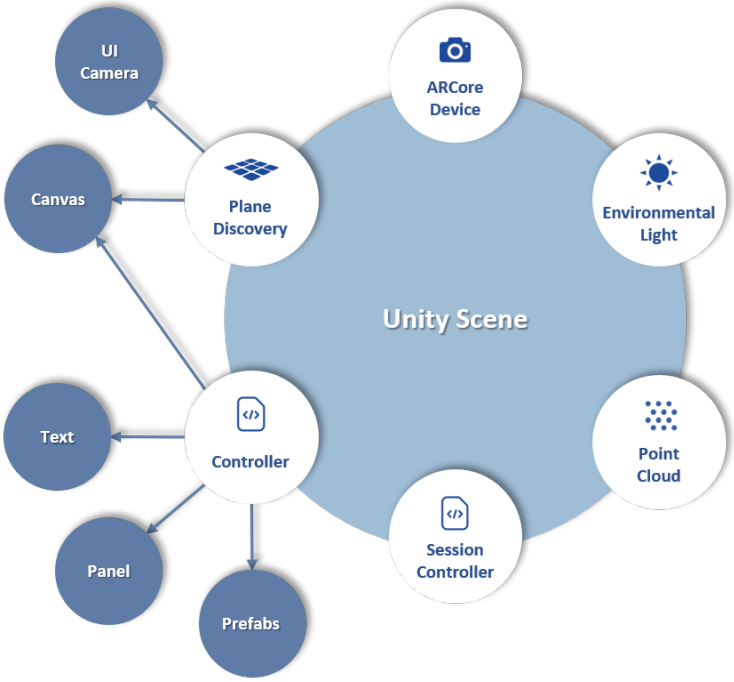


Figure 4.1: Unity scene

- Plane Discovery - provides plane discovery visuals that guide the user to scan surroundings and discover planes, representing their feature points.
  - Canvas - is the User Interface (UI) part associated with this Game Object. Here is where all the panels corresponding to the bounding boxes and virtual objects that interact with the user are created.
  - UI Camera - is responsible for all UI elements, unlike the main camera that can see everything except these elements. This approach is required because, otherwise, real-world overlaps UI elements. It is also necessary to set the depth parameter of this second camera to a higher value than the depth parameter of the main camera to get that result.
- Controller - has the controller script that manages the AR scene. This script has references to the following Game Objects:
  - Canvas - to establish a connection to the UI part;
  - Panel - to visualize the bounding boxes on the screen;
  - Text - to show the label and the confidence value of the detected object;
  - Prefab - a list of virtual objects to interact with the user. [1]

[1] all virtual objects were obtained through the Unity Asset Store. They are available at https://assetstore.unity.com/packages/3d/props/furniture/bedroom-architect-series-85476 and https://veg3d.com/ [accessed 2 December 2019]

All the Game Objects and their children belong to the Default layer. Only Canvas and their children, including the UI camera, belong to the UI layer. Thus, each of the cameras knows what elements should render.

With the Unity Scene set up, it was possible to develop the next three parts of this work. Figure 4.2 shows a simple diagram of the implementation with a brief description of the different steps of the process. In the next sections, we explain all these steps in detail.

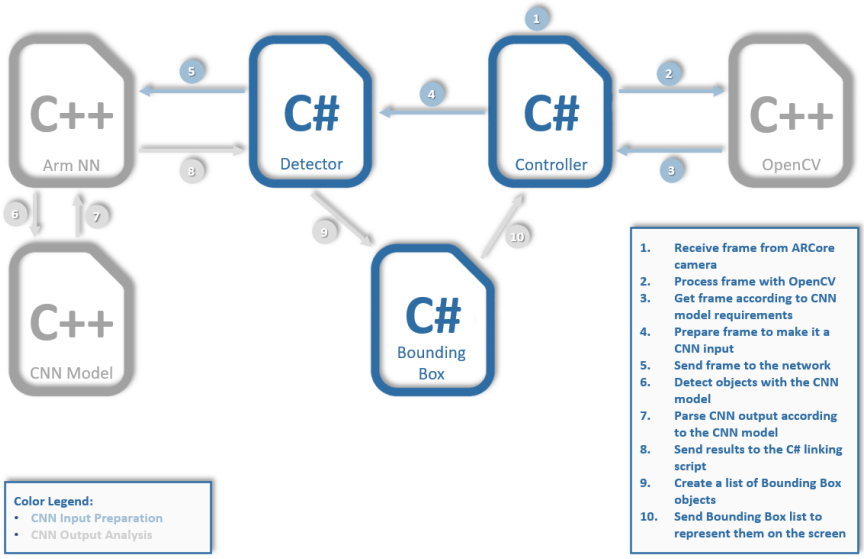


Figure 4.2: Implementation diagram

4.2 CNN Input Preparation

This part consists of preparing the network's input, creating the necessary structure for processing the images received by the camera in real-time, transforming them into the input of the network, and building the network itself.

The Controller module mentioned in the previous section is also responsible for establishing the link between the computer vision part and the artificial intelligence part, helped by another module, named Detector. The first one receives frames from the camera through the ARCore library to render them in a new thread created to ensure that frame processing coincides with camera operation. Thus, all new frames are ignored until the previous frame is processed. The Detector module makes that frame the CNN input, creating a connection with the network. All the work performed by CNN is done separately, i.e., outside the Unity framework. That code is written in C++ language, and, as Unity works with C#, it's necessary to import shared object files to create the connection between them and the Unity scene.

4.2.1 Image Processing

The image processing part was mostly done in the Controller module. The application is continually receiving camera frames. If an image is available, its bytes are acquired by ARCore in YUV420p format, which is briefly explained in Appendix B. As the network was trained to receive the image bytes in RGB format, it was necessary to make a conversion.

Furthermore, the camera frame dimensions are higher than the expected values, and the image is inverted in the x-axis. To solve these issues, we decided to use the OpenCV library. We created a shared object because this part of the code was written in C++ instead of C#, as shown in Figure 4.2. At this time, the image is ready to be sent to the network.

If an image is available, but another frame is being rendered, this new frame is ignored. So, a new thread handles the image processing and its passage through the network, to obtain the detection results in the end.

4.2.2 CNN Construction

In the Controller module, memory is allocated for the network input and output. Then, the network is created in the ArmNN module. The parameters used in the network's creation depend on the chosen model and mainly on the AI framework. The first parameter of this function is the model name. As we chose Tensorflow, the second parameter is the name of the input tensor, followed by the image batch size, image height, image width, and channels' number. The third parameter corresponds to the name of the output tensor. These two names can be found in one of the first and last lines of the model, respectively, as shown in Appendix A.

The ArmNN library has three different options for network creation. We opted by protobuf binary file, and we explain its concept in Appendix C. After the network creation, it is necessary to get the network input and output binding info, to optimize the network and to load it onto the runtime device.

The interconnection between CV and AI is explained in detail in Appendix D. At this stage, we send the image to the network to obtain the output for further analysis.

4.3 CNN Output Analysis

This part consists of analyzing the network's output, understanding how to save the data after passing through CNN, and organizing the results to represent them on the device's screen.

4.3.1 CNN Output Processing

The CNN output depends on the chosen model. In Subsection 3.3.4, we explained why we decided to use Tiny-YOLOv2 and how it works. Since the network output is a pointer with all data sequentially stored, we created the CNN Model module to organize it.

First, we iterate through all the cells, traversing all rows and columns of the grid and getting the 5 components of each bounding box, as shown in the first 5 lines of Algorithm 1. Then, we

calculate the correct value of x, y, width, height and confidence, as presented in `BoundingBox`, using the Sigmoid and the Exponential functions, described in Equations 4.1 an 4.2, respectively. Furthermore, we store all class confidence scores.

Algorithm 1 Output Organization Algorithm

```
1:  for cy = 0; cy < 13; cy++ do ← going through image's lines
2:      for cx = 0; cx < 13; cx++ do ← going through image's columns
3:          for b = 0; b < 5; b++ do ← going through each bounding box
4:              BOUNDINGBOX(outputBuffer, outputIndex, cx, cy, b)
5:              outputIndex = outputIndex + 20 + 5 ← 20 classes + 5 components per each bb

6:  function BOUNDINGBOX(*output, index, cx, cy, b)
7:      x = cx + sigmoid(output[index]) * 416/13 ← image's width divided by cell's size
8:      y = cy + sigmoid(output[index + 1]) * 416/13 ← image's height divided by cell's size
9:      w = exponential(output[index + 2]) * anchor[2*index]* 416/13 ← w for width
10:     h = exponential(output[index + 3]) * anchor[2*index+1]* 416/13 ← h for height
11:     c = sigmoid(output[index + 4]) ← c for confidence
12:     for i = 0; i < 20; i++ do
13:         classes[i] = output[i + index + 5] ← to obtain each class confidence score
14:     CLASSES(classes)

15: function CLASSES(*classes)
16:     newClasses = softmax(classes)
17:     bestClass = argmax(newClasses)
```

$$f(x) = \frac{1}{1+e^x} \tag{4.1}$$

$$f(x) = e^x \tag{4.2}$$

The confidence score of each class can be greater or less than zero, it can be a few tens or even hundreds, or it can have no value (nan) at all. These values indicate the likelihood of the bounding box matching with the presented class. To give some statistical meaning to these values, we used the Softmax function, presented in Equation 4.3 and in the first line of `Classes` function in Algorithm 1.

$$\sigma(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}, \text{ for } i = 1, ..., K \tag{4.3}$$

K means the number of classes for each bounding box, in this case, 20. The purpose of this function is to normalize a set of real numbers into a probability distribution. Thus, it is verified that the larger input components will correspond to larger probabilities. In case the sum is less than zero or does not have a valid value (nan), all classes will have the same probabilty. The softmax function is often used in neural networks to map the non-normalized output of a network into a probability distribution over predicted output classes.

After all probabilities are ordered, we apply the Argmax function, represented by Equation 4.4 and the second line in `Classes` function in Algorithm 1.

$$argmax_x\ f(x) := \{x | \forall y : f(y) \leq f(x)\} \tag{4.4}$$

In this case, we find the highest probability and save the corresponding index in the array. Then, this probability is multiplied by the confidence value. If the result is higher than a threshold that has been defined, we assume that this bounding box is valid, and we save all its information in a priority queue.

We checked if any element overlaps with those in front of it. Only the elements completely separated from each other are inserted into a list created to store the successfully detected bounding boxes. Then, we returned the completed priority queue to the ArmNN module. As shown in Figure 4.3, the approach used to detect the existence of overlaps consists of verifying if:

- the left side of the object A is smaller than the right side of the object B (Figure 4.3 (b));
- the right side of the object A is bigger than the left side of the object B (Figure 4.3 (c));
- the top side of the object A is smaller than the bottom side of the object B (Figure 4.3 (d));
- the bottom side of the object A is bigger than the top side of the object B (Figure 4.3 (e)).

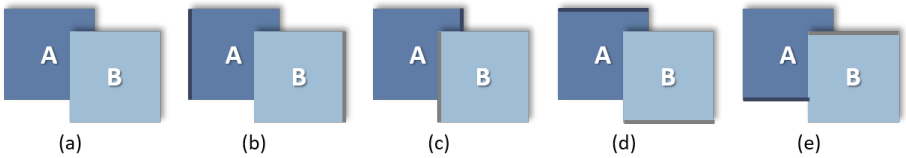


Figure 4.3: Bounding box overlap

4.3.2 Bounding Boxes Representation

After completing the object detection process, it was required to have access to the bounding boxes information in the C# side. In Appendix D, we explained the correct way to communicate between the two sides. With all data correctly organized and stored, the Controller module can represent bounding boxes and the corresponding virtual objects.

For that, we needed the Game Objects associated with the module mentioned in Section 4.1. The panel served to represent the bounding box on the screen. And the text was used for the object name and its confidence. The list of Game Objects was required to represent the corresponding virtual object on the screen overlapping the real object. As the model that we chose only recognizes 20 classes, we only had the corresponding 20 virtual objects on our list.

First, we need to understand the relation between the image that we send to the network and the image seen on the device's screen. In Figure 4.4 we present an example of this relation.

Figure 4.4: Relation between processed images (1)

The image marked with a **1** inside Figure 4.4 is an example of the image seen on the screen, with 992 x 744px, despite the default smartphone resolution being 2280 x 1080px. The image **2** is the corresponding example of the image read by ARCore, and it decreases the original resolution to 640 x 480px. However, the resolution of the frame received by this YOLO's version must be 416 x 416px, so we resized it like image **3**, as mentioned in Subsection 4.2.1.

In Figure 4.5, we present an object detection's example, a cat, one of the objects belonging to the 20 detectable classes.

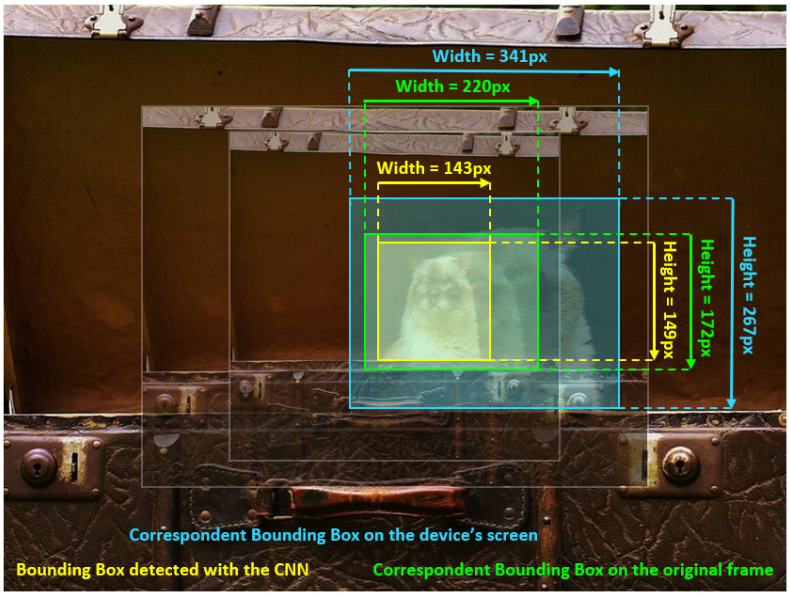


Figure 4.5: Relation between processed images (2)

Considering the resolution of the bounding box is 143 x 149px on the CNN image, the resolution on the device's screen would correspond to 341 x 267px. To get these values, first, we need to obtain the resolution of the bounding box on the original frame, which is 220 x 172px. Equations 4.5, 4.6, 4.7 and 4.8 correspond to these steps.

$$width_{original\,frame} = width_{CNNimage} * \frac{640}{416} \tag{4.5}$$

$$height_{original\,frame} = height_{CNNimage} * \frac{480}{416} \tag{4.6}$$

$$width_{screendevice} = width_{original\,frame} * \frac{992}{640} \tag{4.7}$$

$$height_{screendevice} = height_{original\,frame} * \frac{744}{480} \tag{4.8}$$

Another thing that we need to consider is that the axis of origin of the image represented on the screen differs from the network output. The axis origin of the bounding box (x, y) coordinates obtained with the CNN is in the upper left corner of the image, as we show in Figure 4.6 in yellow. On the other hand, the axis origin of the virtual objects added to the screen is on its center, represented in blue in the figure.

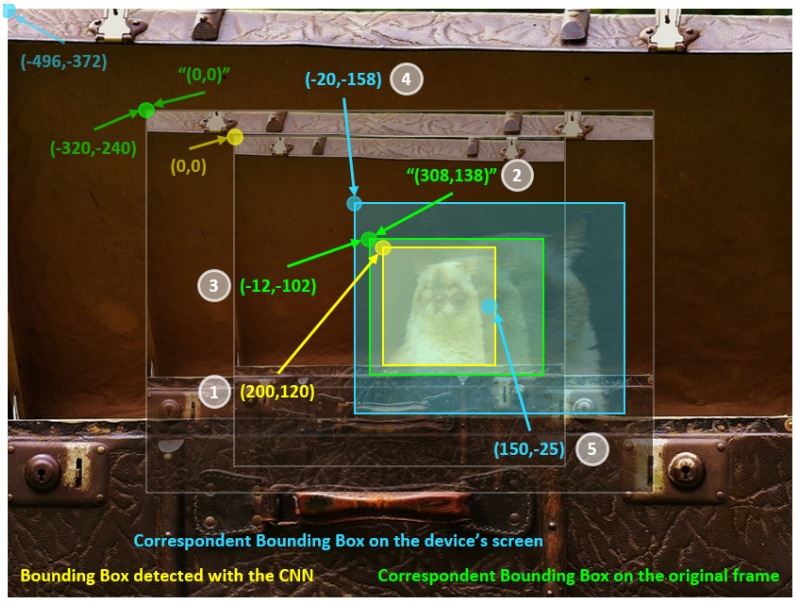


Figure 4.6: Relation between processed images (3)

Supposing that the coordinates of the bounding box obtained by the network are (200, 120), indicated in the figure with a **1**, we obtained the coordinates of the corresponding bounding box in

the original frame marked with a **2** through Equations 4.9 and 4.10.

$$x_{originalframe} = x_{CNNimage} * \frac{640}{416} \tag{4.9}$$

$$y_{originalframe} = y_{CNNimage} * \frac{480}{416} \tag{4.10}$$

Note that for these coordinates, the origin is also in the upper left corner. Next, we need to do a translation to get the origin in the center of the image, signaled with a **3**. To do this, we must apply Equations 4.11 and 4.12.

$$x_{originalframe} = x_{originalframe} - \frac{640}{2} \tag{4.11}$$

$$y_{originalframe} = y_{originalframe} - \frac{480}{2} \tag{4.12}$$

Then, we can calculate the corresponding coordinates in the device's screen represented in Figure 4.6 with a **4** with Equations 4.13 and 4.14.

$$x_{screendevice} = x_{originalframe} * \frac{992}{640} \tag{4.13}$$

$$y_{screendevice} = y_{originalframe} * \frac{744}{480} \tag{4.14}$$

However, to finish, we must do a translation of the coordinates from the upper left corner to the image center, obtaining the corresponding coordinates (150, -25) indicated with a **5**. Equations 4.15 and 4.16 correspond to this final step.

$$x_{screendevice} = x_{originalframe} + \frac{width_{boundingbox}}{2} \tag{4.15}$$

$$y_{screendevice} = y_{originalframe} + \frac{height_{boundingbox}}{2} \tag{4.16}$$

After determining the central coordinates of the detected object, we represent the name of the object and its confidence percentage above the bounding box. We also place the corresponding virtual object in the location where the real object was detected, as shown in Figure 4.7.

Figure 4.7: Relation between processed images (4)

4.4 Tracking and Collaboration Between Users

This section describes the approaches used in the development of tracking and collaboration.

4.4.1 Tracking

We initialize the object's tracking whenever the CNN has detected at least one object. To do so, we first create the chosen tracker using the OpenCV library. Then, we initialize the tracker with the processed frame and the corresponding object detection result, i.e., the bounding box.

As explained in Section 5.2, we decided to use the Kernelized Correlation Filter (KCF) on our application, a tracking framework proposed by Henriques et al. [51]. To initialize this tracker, OpenCV creates a ROI of the provided bounding box and adjusts the corresponding center coordinates (x,y) to its corner. They initialize the Hann Window Filter to select a sample subset and later apply a Fourier Transform. It provides low aliasing with just a small reduction in image resolution. With a calculated output sigma, they apply the Fourier Transform to the Gaussian response and record the compressed and non-compressed descriptors of the Principal Component Analysis (PCA) feature extractor. A valid intersection between the ROI and the image enables the tracking in the following frames.

As mentioned in Section 4.2, when the CNN is processing one frame, the other frames received from the camera are ignored. In these frames, we apply the tracker update function from the OpenCV library. We send this frame to the tracking algorithm and analyze its output: a new bounding box with a prediction of the object location.

The tracking update starts by performing a detection. It extracts, pre-processes the patch, and gets the compressed and the non-compressed descriptors. It compresses the Kernel Regularized Least Squares (KRLS) model to merge all the features, computes the Gaussian kernel and the Fourier transform to the kernel, calculates the filter response and extracts the maximum response to update the bounding box. Lastly, it extracts the patch for learning purposes and gets all the descriptors to update the training data and the KRSL model for the next frame.

4.4.2 Collaboration

We extended our Unity Scene to integrate the new required files for the collaboration's development. We added two new Game Objects, as shown in Figure 4.8 and which we briefly describe below.

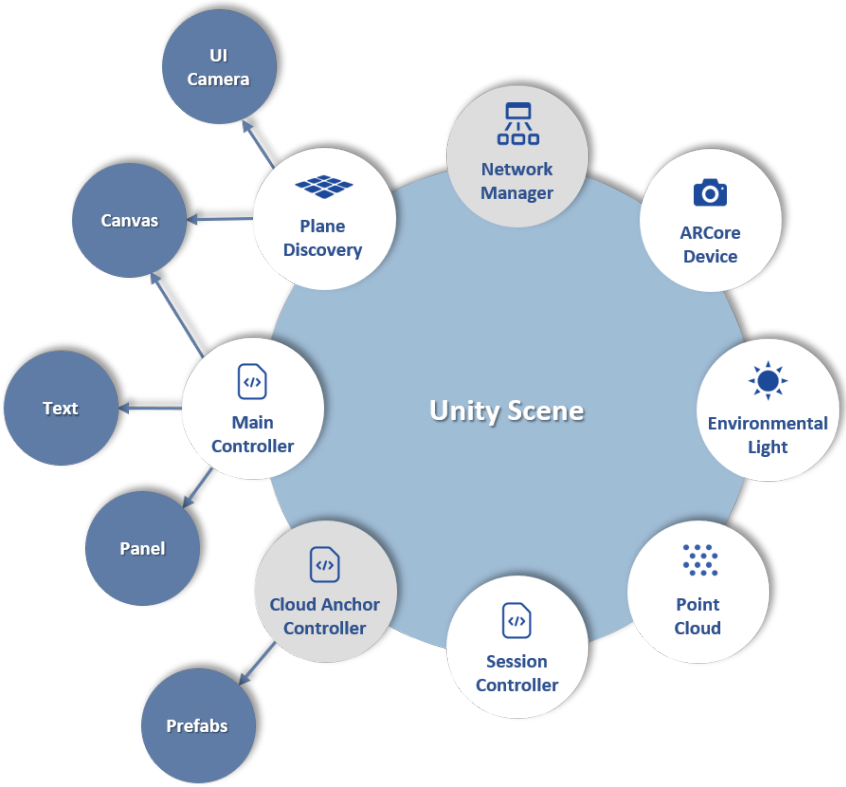


Figure 4.8: Unity new scene

- Network Manager - establishes a connection between the application and the cloud service where the location of the virtual objects that are shared by more than one user is stored.

- Cloud Anchor Controller - has a controller script that manages user's actions related to the collaboration. Before, Main Controller managed prefabs, but now, this script has references to this list of virtual objects to place them on a surface that allows the collaboration between multiple users through a cloud service.

ARCore has a Cloud Anchor[2] service available that allows multiplayer or collaborative AR experiences that Android and iOS users can share. Cloud Anchors let multiple users add virtual objects to the AR scene, view, and interact with these objects simultaneously from different positions in a shared physical space. To enable these shared experiences, ARCore connects to the ARCore Cloud Anchor service to host and resolve anchors, which requires a working Internet connection.

First, to use Cloud Anchors, we need to add a Google ARCore Cloud Anchor API Key to our app. We can generate this key through the Google Cloud Platform and then include it in our app. When the smartphone launches the app, ARCore starts searching for planes in front of the camera. Once planes are found, the user can touch the screen to place an anchor on a plane. The user action launches a host request to the cloud service that includes data representing the anchor's position relative to the visual features near it. If this request is successful, a unique ID is assigned to this anchor. This ID allows other users to share the same room. If another user decides to join this room, a resolve request is sent to the ARCore Cloud Anchor service to recreate the same anchor and render the 3D virtual objects attached to it. Figure 4.9 shows the interaction between two users using this service.

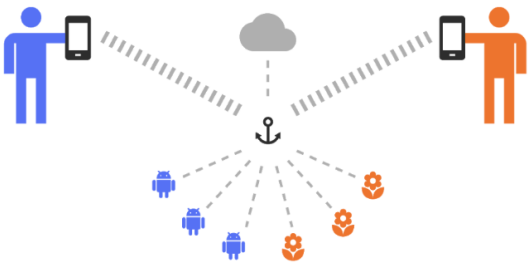


Figure 4.9: Colaboration between users [2]

After placing the anchor, we can start object detection and tracking. When an object is detected, a similar virtual object is placed in front of it in the near plane detected by ARCore. From this moment on, any user who enters the same room can see the virtual object placed by the initial user and can share a collaborative experience.

[2] more information about this service at https://developers.google.com/ar/develop/java/cloud-anchors/overview-android [accessed 4 January 2020]

Chapter 5

Results and Analysis

In this chapter, we present the obtained results in different stages of this work. It is divided into three sections. The first section is about object detection and classification results. In the second section, we present the tracking analysis. Finally, the processing time obtained by different devices, as mentioned in Section 3.3, is addressed in the third section.

5.1 Object Detection and Classification

For object detection and classification analysis, we collected a sample of 148 images with the OnePlus 6's camera to annotate all the objects present in the images. Figure 5.1 presents all the classes annotated in the collected files.

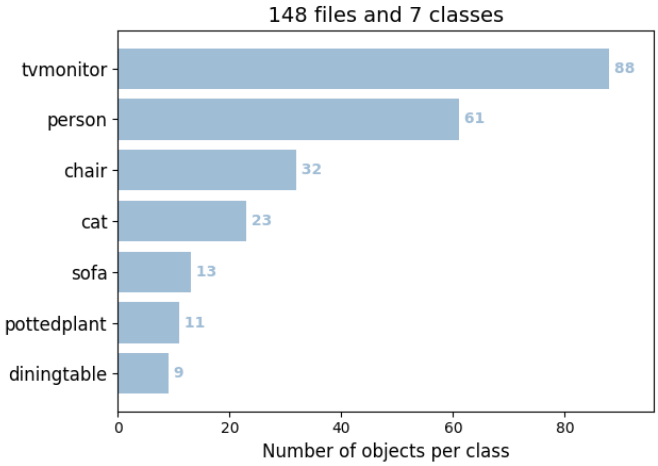


Figure 5.1: Images' ground-truth

We obtained the results presented in Figure 5.2 by applying these images to the model in the application runtime. We found that only 66 of the tv monitors' 88 annotations were detected. Furthermore, 46 were classified as true positives and 20 as false positives, which gave a mAP of 13.59%, as shown in Figure 5.3 (a).

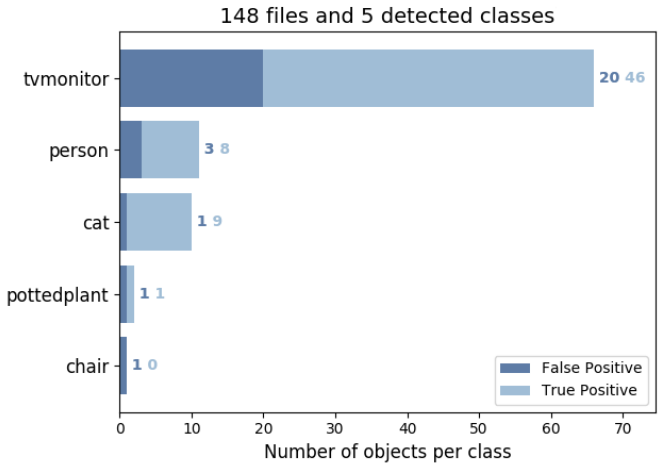


Figure 5.2: Model's detection results

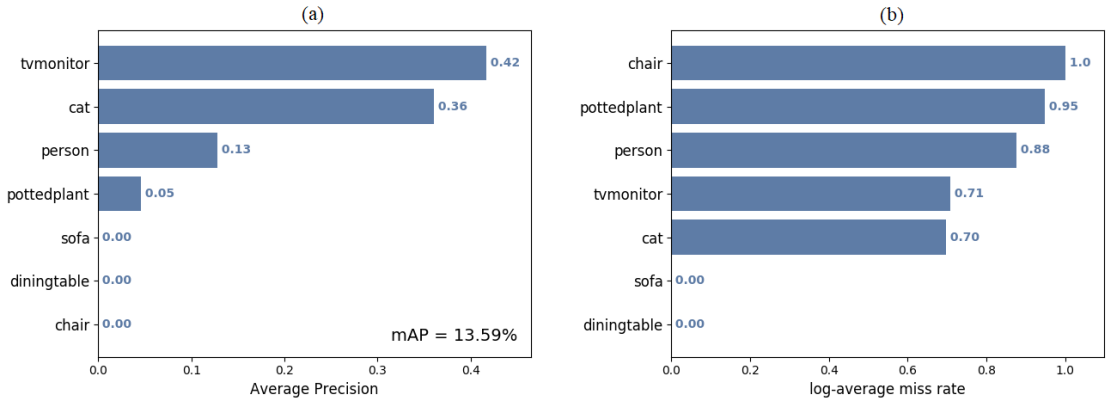


Figure 5.3: Model's (a) mAP and (b) LAMR

The Tiny-YOLOv2 model that we use in our application has a mAP of 23.7% registered in the YOLO official website[1]. We verified a lower mAP, probably due to the significantly smaller sample used for testing.

In Figure 5.3 (a), we present the mAP result and, in Figure 5.3 (b), we show the Log-Average Miss Rate (LAMR). Contrary to the mAP that refers to the detected objects, LAMR refers to the objects that were not detected. For example, the tv monitor class has a mAP of 42%; and the chair class has a LAMR value of 100%, since the only time a chair was detected, that object was not a chair.

To improve our detection and classification results, we tried different training approaches described in Appendix E.

[1] more information about this model at `https://pjreddie.com/darknet/yolo/` [accessed 26 January 2020]

5.2 Tracking Analysis

For tracking analysis, we follow a similar approach. We applied different trackers to the previous sample of images. We registered the mAP of each tracker based on the Intersection Over Union (IOU) or Jaccard Index to understand the tracking precision. As mentioned in Section 2.2, there are some trackers available in the OpenCV library, such as Boosting, MIL, KCF, TLD, MedianFlow, MOSSE, GOTURN, and CSRT. We decided to use them in our application to compare and evaluate their performance. However, we cannot use the last two trackers. GOTURN uses a Caffe model for tracking since it is CNN based. Furthermore, as we decided to work with TensorFlow, all the required shared files for the integration of every component of this work were based in this model, which makes this tracking algorithm's implementation hard. And CSRT is not available in the OpenCV version used.

Figure 5.4 shows a comparison between the trackers concerning the mAP. We verified that KCF and MOSSE are the best options. However, this analysis does not consider the percentage of the analyzed frames of each algorithm. Due to this, in Figure 5.5 we present a comparison between the previous best trackers, whereas MOSSE seems to be closer to KCF, but it only tracked 8.55% of the frames available for that, unlike KCF that did 44.55%.

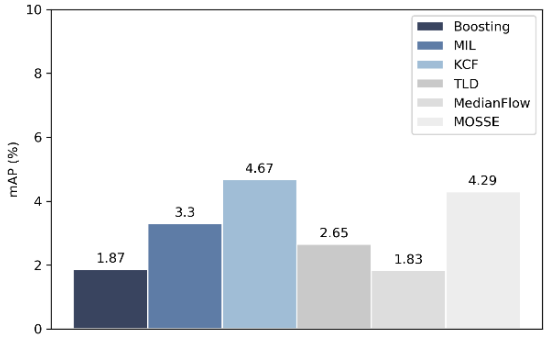


Figure 5.4: Trackers' mAP

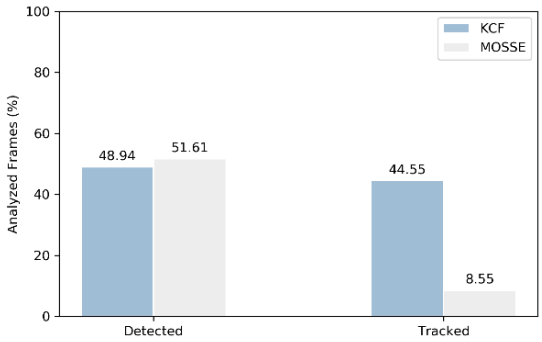


Figure 5.5: KCF and MOSSE's analyzed frames

In Figure 5.6, we present a bar chart of frames' processing time obtained with the KCF tracker. We obtained this plot by applying the same sample of images used in Section 5.1 directly to the neural network. Considering that each bar corresponds to 10ms, the tracking takes between 90ms and 100ms in more than 30% of the samples. Similarly, almost 20% of detections take around 340ms to be processed.

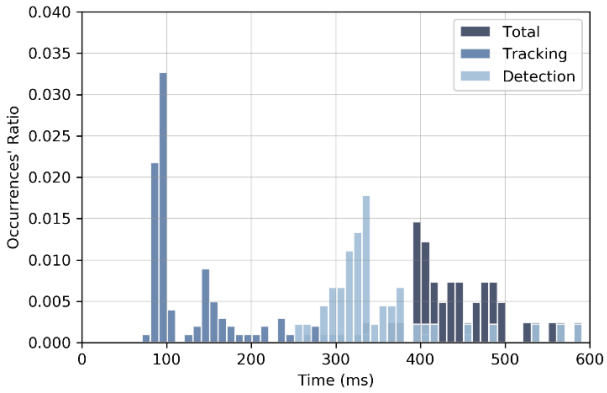


Figure 5.6: KCF's processed frames

Next, we present a comparison between all these trackers regarding the detection duration (Figure 5.7 (a)), the tracking duration (Figure 5.7 (b)), with the scale adjusted for better visibility, and the total duration of each frame (Figure 5.7 (c)) when the app is running.

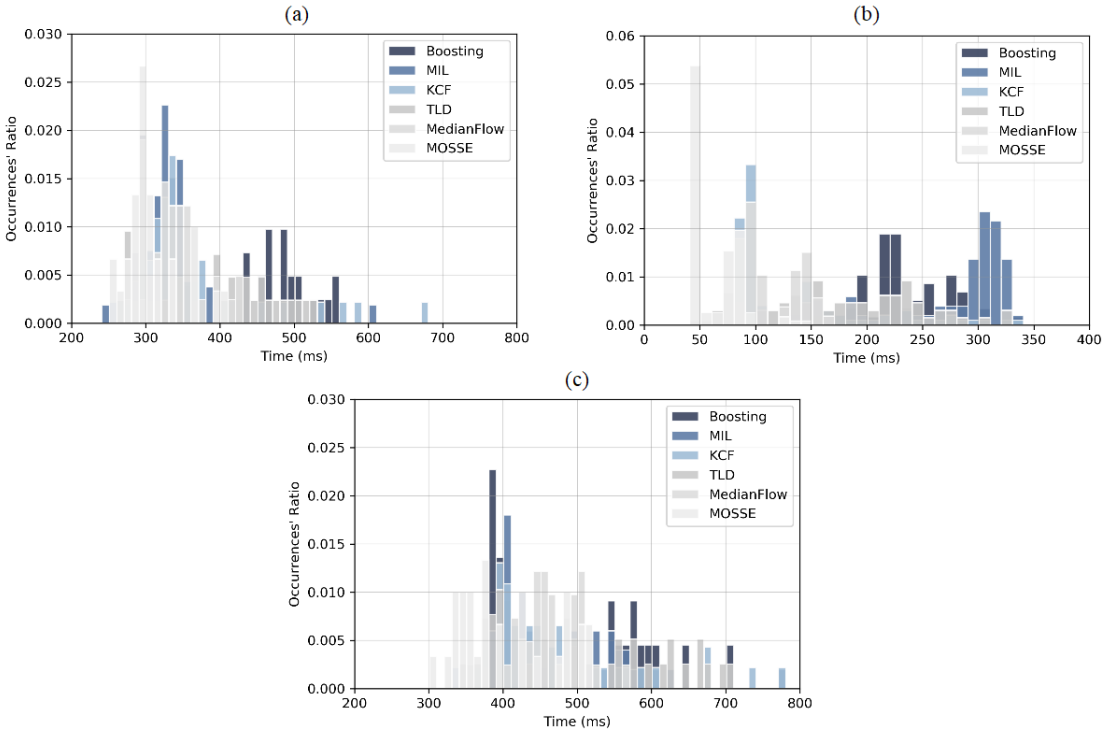


Figure 5.7: Trackers' (a) detection, (b) tracking, and (c) total duration

As we can see, despite the different scales, the MOSSE tracker is the fastest, and MIL is the slowest. KCF is followed by MedianFlow, TLD, and Boosting, respectively. We can also verify that the performance of each tracker has a small effect on the correspondent detection duration, and consequently, in the total processing time of a frame.

We compare each tracker with three different metrics: average, median, and 95th percentile, as shown in Figure 5.8 (a), (b), and (c), respectively.

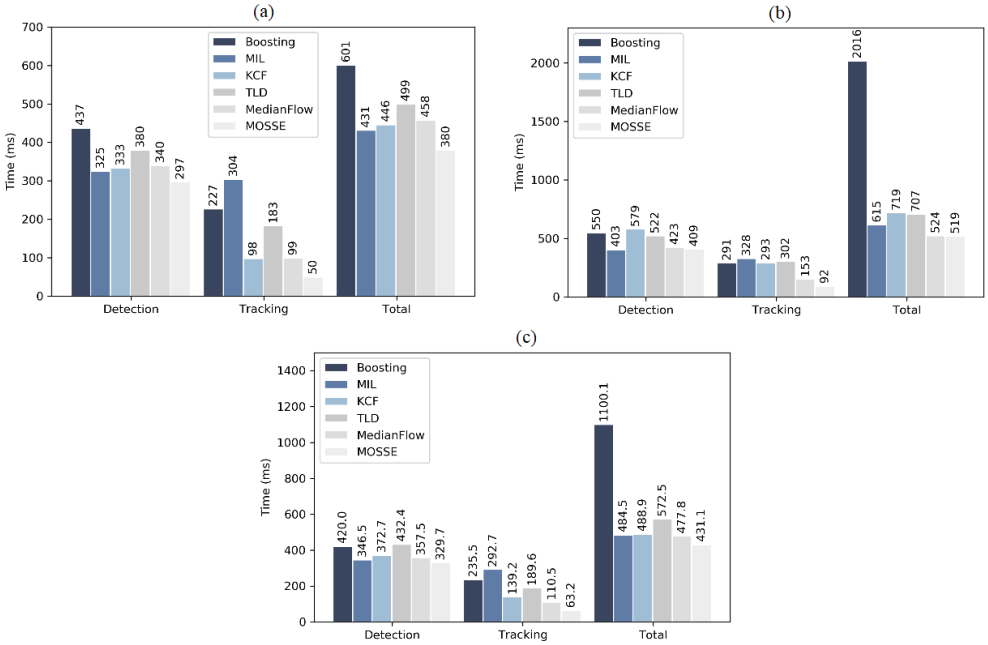


Figure 5.8: Trackers' (a) median, (b) 95th percentile, and (c) average processing time

We can see that the Boosting algorithm has the worst performance. During its execution, the application responsiveness significantly decreased, which is the most plausible assumption to do concerning the 95th percentile value and its average in the total duration. On the other hand, MOSSE is the best, probably due to the number of ignored frames in tracking despite its clear performance superiority. However, considering the previous analysis, the best algorithm option is the KCF, since it has the best mAP and is the second-fastest.

In Figure 5.9, we present the number of frames processed during detection evaluated by the same metrics used above.

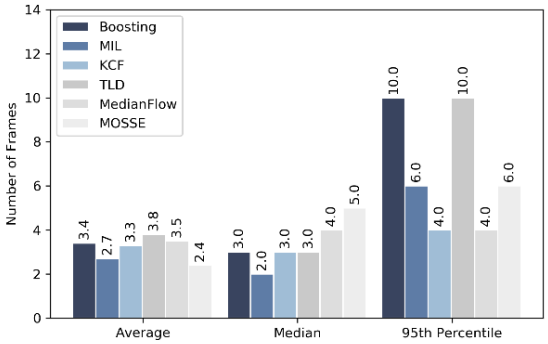


Figure 5.9: Processed frames during detection

As we can see, all the trackers have close values, except for the Boosting and the TLD algorithms regarding the 95th percentile value. In Table 5.1, we present the median value of the frame rate obtained by all trackers concerning the tracking duration.

Table 5.1: Tracking frame rate (taken from [7])

	Trackers					
Tracking	Boosting	MIL	KCF	TLD	MedianFlow	MOSSE
	4.40 fps	3.29 fps	10.20 fps	5.46 fps	10.10 fps	20.00 fps

Next, we present the detailed results of the KCF tracker obtained with a similar approach as the one used in Section 5.1 for detection results. In Figure 5.10, we can see, for instance, that the tv monitor was successfully tracked 21 times, despite the 27 false-positive values.

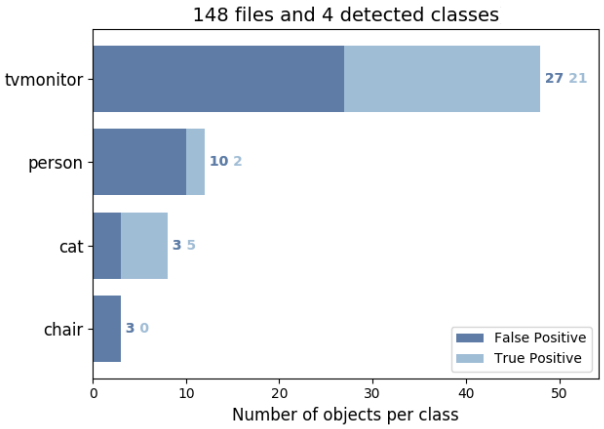


Figure 5.10: KCF's tracking results

Finally, in Figure 5.11 (a), we show the mAP and, in Figure 5.11 (b), we present the LAMR result.

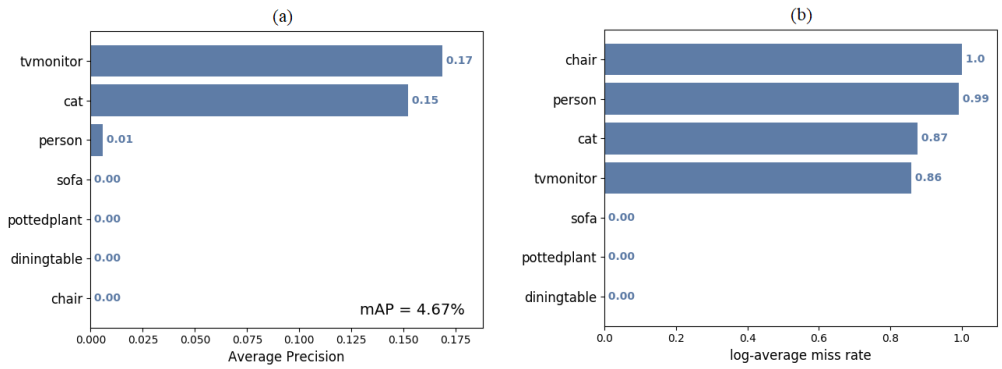


Figure 5.11: KCF's (a) mAP and (b) LAMR

For instance, 15% of the time, cat class was correctly tracked; but 87%, it was not detected when it should be.

We verified that our results were similar to the results obtained by Lehtola et al. [7] presented in Table 2.4 from Section 2.2. Except for MedianFlow, that presented a lower frame rate, 10.1 fps instead of 16.5 fps and TLD, that showed a higher value, 5.46 fps instead of 1 fps. Thus, in our analysis, both are no longer the ones with the best and the worst frame rate, respectively. In our analysis, KCF was the tracker with the best frame rate.

With regards to mAP, all trackers were below expectations when compared to the results obtained by Lehtola et al. presented in Table 2.5 from Section 2.2.

5.3 Processing Time

The execution of our application embraces many stages, as we describe in Section 4.1. However, we focused our analysis on the most important ones, i.e., detection and tracking. We start by presenting a bar chart of the frames' processing time obtained through the OnePlus 6's camera simultaneously with the application execution. In the normalized histogram of Figure 5.12, we compare the time spent in detection, tracking, and in the execution of every step required for each processed frame. Considering that each bar corresponds to 10ms, the tracking takes between 50ms and 60ms in almost 20% of the samples. Similarly, almost 15% of the detections take around 330ms to be processed. Furthermore, comparing the tracking and detection duration with the total time allows us to conclude that the detection takes up the majority of the processing time of a frame.

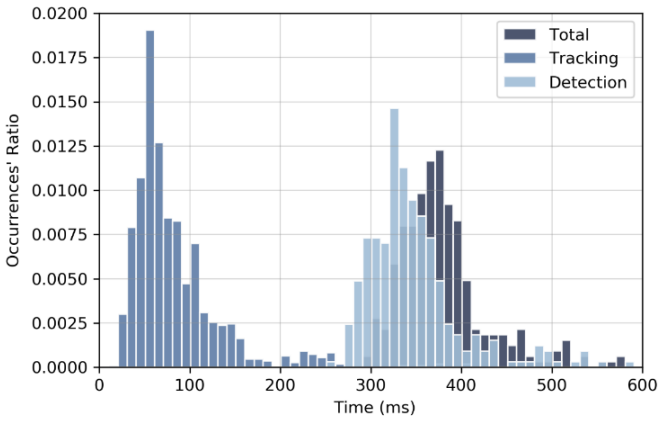


Figure 5.12: OnePlus 6's processed frames

In Section 5.2, we made a similar analysis where we verified a comparable result. Probably, the verified differences between this plot and the plot presented in Figure 5.6 of the previous section are related to the required time for receiving the captured image from the device's camera

and the required time to open an image file from a specific directory in the smartphone and collect all these bytes to obtain the RGB pixels, which takes more time.

Next, we present a comparison between the five devices mentioned in Section 3.3. Figure 5.13 (a) represents the detection duration on each device; Figure 5.13 (b) represents the tracking duration, with the scale adjusted for better visibility, since processing time is considerably shorter than the detection time; and Figure 5.13 (c) represents the total duration of each frame when the app is running. As we can see, both Samsung devices are faster than OnePlus devices in object detection. In tracking, all of them, except the Huawei smartphone, achieve similar processing time results. The Huawei Mate 20 Lite is the slowest one, both in detection and tracking, demonstrating a performance considerably inferior to the others.

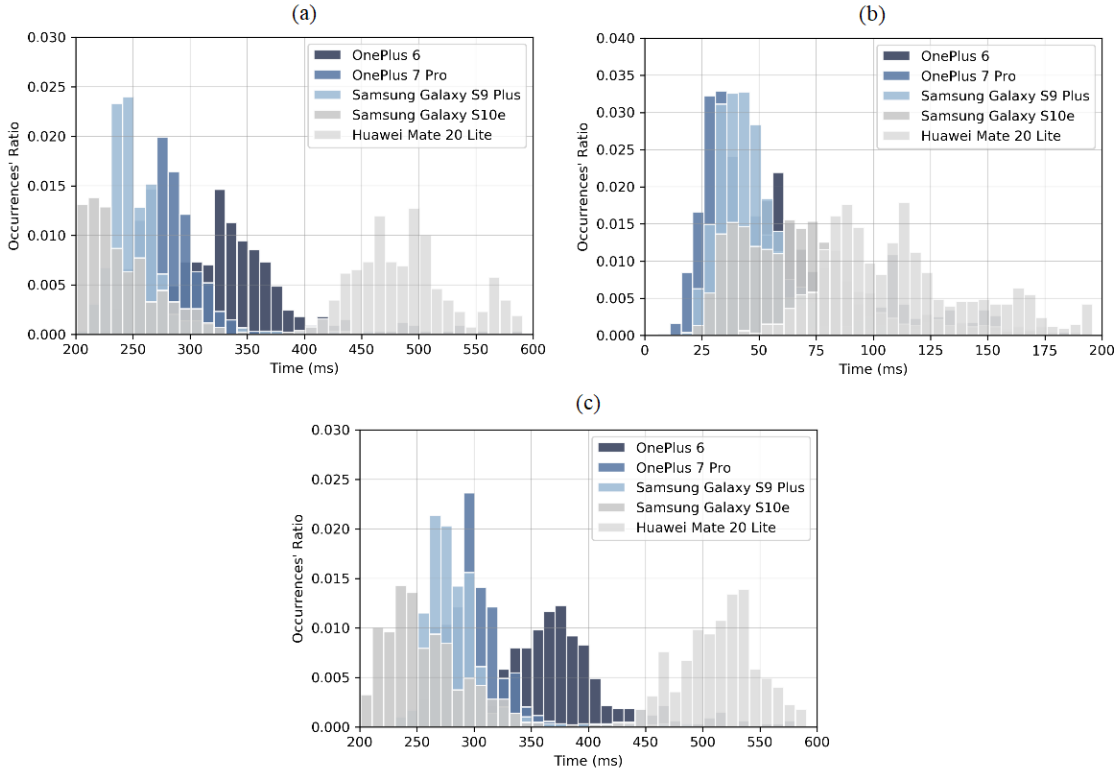


Figure 5.13: Devices' (a) detection, (b) tracking, and (c) total duration

We also made an analysis comparing each device with the same three metrics used before: average, median, and 95th percentile, as shown in Figure 5.14 (a), (b), and (c), respectively. Here, we also accounted for the duration of other processing required for the application's functionality. It includes the time spent in memory allocations, in the preprocessing (YUV to RGB conversion and resizing), in the placement of virtual objects and the tracking initialization. Again, we concluded that the Huawei Mate 20 Lite is the device with the worst performance. Also, the OnePlus 7 Pro and both Samsung devices perform better than the OnePlus 6 as well.

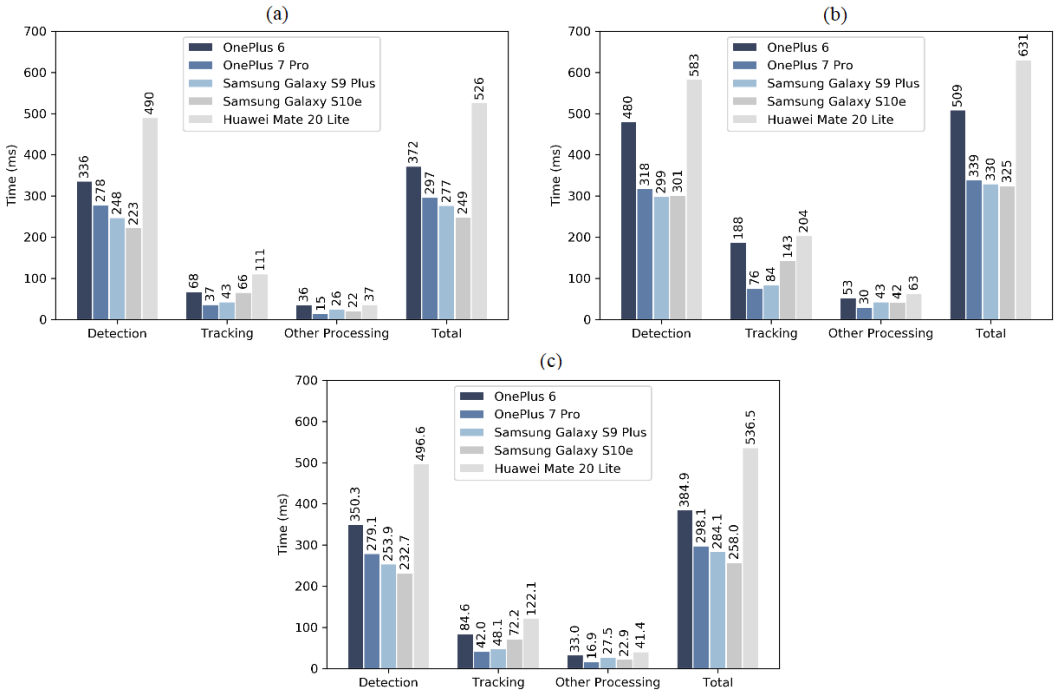


Figure 5.14: Devices' (a) median, (b) 95th percentile, and (c) average processing time

Considering the devices' compatibility with the Hardware Acceleration framework, we expected that both Samsung and Huawei devices could perform the detection on their GPU. Unfortunately, we did not manage to do so, and through Profiler application, we verified that the GPU component was not used in the network's execution. Thus, the detection duration values were lower than expected, especially in these three devices.

In Figure 5.15 (a), we present a circular chart with the distribution of the processing times corresponding to the OnePlus 6 smartphone. The remaining devices showed a similar distribution due to the values shown in the graphs in Figure 5.14. On average, detection occupies 75% of the processing time, tracking 18% and other operations the remaining 7%.

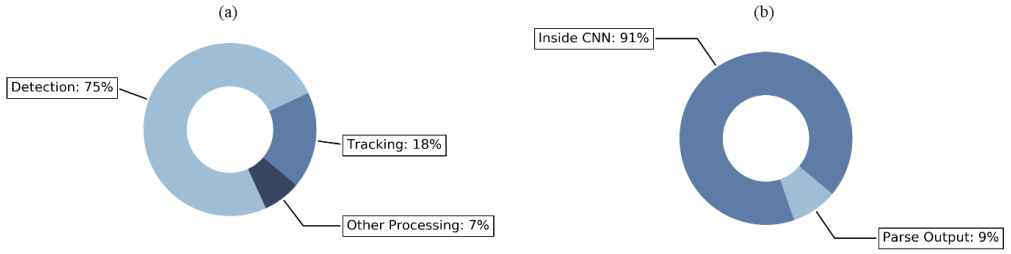


Figure 5.15: OnePlus 6's (a) processing time and (b) detection duration

Figure 5.15 (b) shows the distribution of the time spent inside the neural network and the parsing of its output. That is, 91% of the OnePlus 6's detection time effectively corresponds to the

analysis performed by the CNN model while the output parsing corresponds to the remaining.

As we explained in Section 4.2, during the detection applied to a captured frame, other frames are being processed, for instance, for tracking. In Figure 5.16, we present the same three metrics for the number of frames processed during detection. Considering the variation that can occur in the calculation of the average, as we can see in OnePlus 6 and Samsung Galaxy S10e devices, we focus the frame rate estimate on the median value. For instance, the OnePlus 6 processes a median of 5 frames per 372ms, which corresponds to 13.44 fps. Considering that it processes 1 frame every 336ms of detection, there are 2.98 fps detected. We verified the statement made by the authors Rachel Huang et al. mentioned in Section 3.3. They said that Tiny-YOLOv2 speed is around 244 fps when it runs on a GPU but decreases to about 2.4 fps when there is no GPU compatible. With similar reasoning, the tracking frame-rate is 14.7 fps. However, there is a median of 5 fps tracked per each detection.

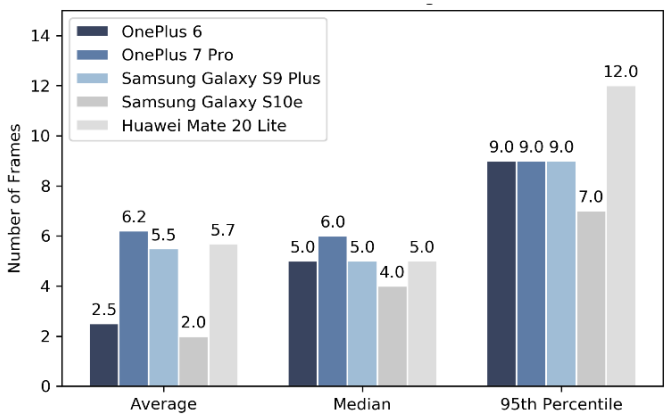


Figure 5.16: Processed frames during detection

The time spent in the collaboration is insignificant compared to the time spent in these two stages of our application's execution. The anchor's placement only occurs once and, in the OnePlus 6, we noted that it takes approximately 7s. Since this happens in parallel with the other processing steps, it does not affect the global execution time.

In Table 5.2, we present the median value of the frame rate obtained in the five devices concerning the total duration of the processing and the duration of the detection.

Table 5.2: Total and detection frame rate

	Devices				
	OnePlus 6	OnePlus 7 Pro	Samsung Galaxy S9 Plus	Samsung Galaxy S10e	Huawei Mate 20 Lite
Total	13.44 fps	20.20 fps	18.05 fps	16.06 fps	9.50 fps
Detection	2.98 fps	3.60 fps	4.03 fps	4.49 fps	2.04 fps

5.4 Collaboration Demo

To analyze the collaboration, we made a demo[2] with two users. They used OnePlus 6 and OnePlus 7 Pro to try the application. The user that enters the room first becomes the host, and the other one becomes the client. The client only renders the virtual content generated by the detections performed on the host. As demonstrated on the demo, the users started to analyze the ground to detect surfaces. Then, the host placed the anchor to create the reference point between the two devices. From that moment on, objects on the scene could be detected (red bounding box) and tracked (green bounding box). The host started to recognize the tv monitor, and a matching virtual object was automatically placed at the nearest position on an identified surface. This virtual object was immediately available to the client in the same location, as shown in Figure 5.17. Taking the potted plant's recognition as an example, we could see that even though the smartphone correctly detected and tracked the object, the placement of the corresponding virtual object was delayed by a few seconds. That happened because, due to the object's height, there was no identified surface near it. After a surface was found, the host correctly placed the virtual object on the scene.

We verified that the detection accuracy and speed were satisfactory. The virtual object's placement and the communication between users were also acceptable, ensuring the real-time requirement. We only used tracking as an interpolation between objects' detections, and we missed the initial goal of updating the virtual objects' position based on it. Thus, even when the person or the bottle moved, the corresponding virtual objects remained in the original location.

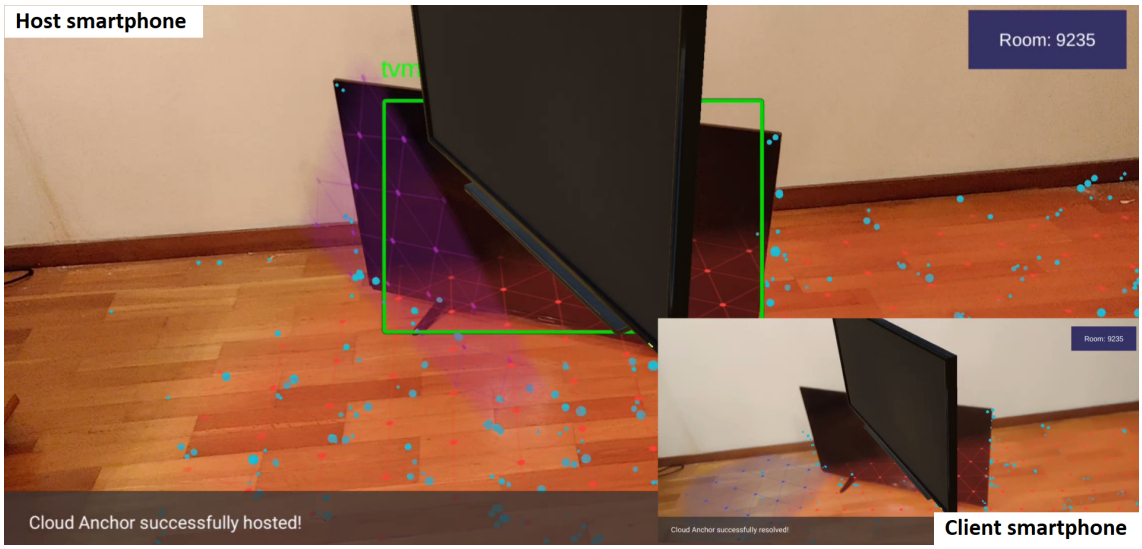


Figure 5.17: TV monitor's detection and virtual rendering

[2] demo available at https://youtu.be/LnjAvKzgf24 [accessed 6 March 2020]

Chapter 6

Conclusion and Future Work

The possibility of collaborating remotely in both a professional and ludic context is an interest that has been growing over the last few years. In this work, we studied and developed a collaborative application where several users can view the same virtual content placed in the exact location where real objects were previously detected in real-time.

The proposed implementation integrates five distinct components. The biggest challenge was the real-time requirement, which made us study different Hardware Acceleration SDKs (1). We opted for the ArmNN product, with many advantages, namely the compatibility with all Arm-based smartphones (2), allowing us to test our application in different devices, but with constraints. For instance, the incompatibility with some AI frameworks (3) and CNN models (4). We decided to use Android OS, which made us choose ARCore as our AR framework (5). The integration of the selected components was considered the best approach for the development of this system despite conditioning each other, as shown in this work's results.

We analyzed the processing time of a frame on different devices with distinct specifications. We obtained a frame rate of 18.05 fps on the Samsung Galaxy S9 Plus, which was not in line with our expectations, mostly due to the low frame rate of 4.03 fps obtained in the object detection processing. We concluded that these values were affected by the unavailability of the GPU acceleration during processing, which we did not expect in Samsung devices, given the compatibility reported by ArmNN.

For object detection and classification, we obtained a mAP of 13.6%, which is lower than the 23.7% presented by the model's author, probably due to the small sample of images collected for testing, given the exhaustive task of making the corresponding annotations for ground-truth. The Tiny-YOLOv2 model was pre-trained with COCO trainval dataset, and we improved it with the Pascal VOC 2012 dataset, looking for a better mAP. We verified that ArmNN does not support all operations performed by Tensorflow, the chosen AI framework, resulting in the impossibility of using a better version of YOLO.

After the tracking analysis, we opted by using the KCF tracker over the other available options in the OpenCV library. We verified that it has a frame rate of 10.20 fps, similar to the result presented by Lehtola et al. in their paper analyzed in Section 2.2. On the other hand, the mAP of

all trackers was lower than expected. This tracker achieved a mAP of 4.7%, which means that it could not follow an object with high precision.

Regarding the collaboration between users, we verified that through the use of the Google ARCore Cloud Anchor API, we achieved a satisfactory experience. The initial user opens a new session and starts identifying a plane to place the anchor and establish a connection with the server. After that, every user who joins the same session, could detect and track objects in the real environment and view their virtual representation in all devices.

6.1 Future Work

As mentioned before, we verified that the object detection of our application runs entirely on the CPU, even in smartphones with ArmNN compatible GPUs. Unfortunately, we were unable to identify the problem that prevents the network from running on the device's GPU since we implemented everything according to the Arm instructions. We are already using one of the fastest CNN models, so improving the frame rate involves discovering how to overcome this obstacle. Another task consists in enhancing the training of the network. As explained in Appendix E, we tried different training approaches. Besides the pre-training of the model with the COCO dataset, we also trained it with the Pascal VOC 2012 dataset. Bearing in mind that the model knew all the images of these two datasets, we were left with no alternatives to test it apart from the small sample we collected. It would be interesting to test it with another complete dataset. Moreover, if, in the future, the ArmNN SDK implements the remaining operations used by Tensorflow, it would be possible to use other YOLO versions.

In Section 3.3, we explained the complexity of the cross-compiling required for the usage of AI frameworks in smartphones. The implementation in Caffe, for instance, could be explored, instead of Tensorflow. Other referenced models with high frame rates and accuracy values can also be tested. As we are using the tiny version of the YOLOv2 model, we are only able to detect 20 classes. The number of detectable classes could be increased. All these aspects are fundamental to the remaining steps' development that has yet to be completed.

Tracking performance was below expectations concerning their mAP. Understanding their behavior to improve them and trying other trackers, such as GOTURN, could be appropriate since it has impressive accuracy results as it is CNN based. We only used tracking as an interpolation between objects' detections, and we missed the initial goal of updating the virtual objects' position based on it. We did not implement multitracking due to trackers' weak precision, the considerable amount of time spent in object detection, and the model's mAP, even though we can detect multiple objects simultaneously. Multitracking would be a significant improvement in our application since it would be possible to know all the detectable objects in a particular environment and ensuring the placement of only one virtual object in the same location where the real object had previously been detected.

For the collaboration task, it is essential to update the location and the orientation of the virtual objects in real-time, according to the movements that the real objects may have. The collaboration's improvement between users within distinct rooms would enhance the user experience and would allow the use of this application in critical contexts. Finally, the creation of a virtual environment to interact with this augmented scene directly could make this application even more complete and suitable for different purposes.

Appendix A

YOLO Models

A.1 YOLOv1.0

The Tensorflow tool could not parse the `.pb` file because it is bigger than the supported size: 1GB. We still present its model appearance. For the remaining models, we also marked the ArmNN's respective not supported operation, both in the model specifications and in its representation.

Figure A.1: YOLOv1.0 model

A.2 Tiny-YOLOv1.0

```
Found 1 possible inputs: (name=input, type=float(1), shape=[?,448,448,3])
No variables spotted.
Found 1 possible outputs: (name=output, op=Identity)
Found 45089466 (45.09M) const parameters, 0 (0) variable parameters, and 0 control_edges
Op types used: 50 Const, 12 BiasAdd, 11 Mul, 10 Maximum, 9 Conv2D, 9 Pad, 6 MaxPool, 3 MatMul, 1 Transpose, 1 Sub, 1 StridedSlice,
1 Shape, 1 Reshape, 1 Placeholder, 1 Pack, 1 NoOp, 1 Identity
To use with tensorflow/tools/benchmark:benchmark_model try these arguments:
bazel run tensorflow/tools/benchmark:benchmark_model -- --graph=/mnt/c/Users/joana/Downloads/YOLO-models/yolo-tiny.pb --show_flops
--input_layer=input --input_layer_type=float --input_layer_shape=-1,448,448,3 --output_layer=output
```

Figure A.2: Tiny-YOLOv1.0 model's specifications

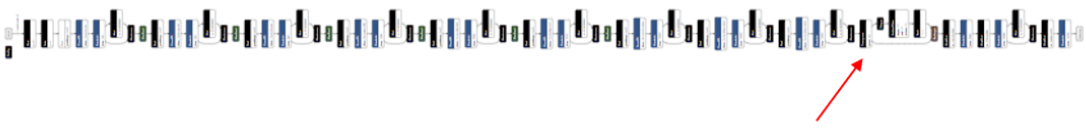


Figure A.3: Tiny-YOLOv1.0 model

A.3 YOLOv1.1

```
Found 1 possible inputs: (name=input, type=float(1), shape=[?,448,448,3])
No variables spotted.
Found 1 possible outputs: (name=output, op=Identity)
2020-02-23 01:41:52.097415: W tensorflow/core/framework/allocator.cc:113] Allocation of 86051840 exceeds 10% of system memory.
Found 197316528 (197.32M) const parameters, 0 (0) variable parameters, and 0 control_edges
Op types used: 381 Const, 73 Conv2D, 50 StridedSlice, 49 Mul, 25 BiasAdd, 25 Maximum, 25 Pad, 24 Sub, 24 RealDiv, 8 ConcatV2, 4
MaxPool, 1 Transpose, 1 Shape, 1 Reshape, 1 Placeholder, 1 Pack, 1 NoOp, 1 MatMul, 1 Identity
To use with tensorflow/tools/benchmark:benchmark_model try these arguments:
bazel run tensorflow/tools/benchmark:benchmark_model -- --graph=/mnt/c/Users/joana/Downloads/YOLO-models/yolov1.pb --show_flops
--input_layer=input --input_layer_type=float --input_layer_shape=-1,448,448,3 --output_layer=output
```

Figure A.4: YOLOv1.1 model's specifications

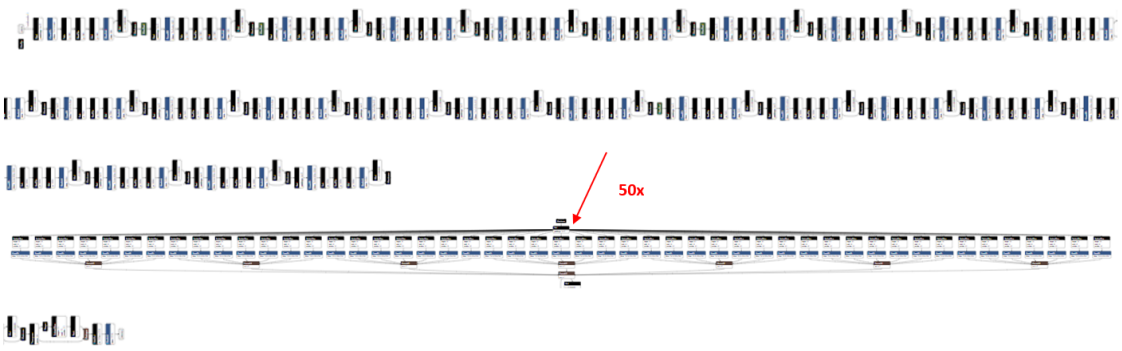


Figure A.5: YOLOv1.1 model

A.4 Tiny-YOLOv1.1

```
Found 1 possible inputs: (name=input, type=float(1), shape=[?,448,448,3])
No variables spotted.
Found 1 possible outputs: (name=output, op=Identity)
Found 27100030 (27.10M) const parameters, 0 (0) variable parameters, and 0 control_edges
Op types used: 63 Const, 16 Mul, 9 BiasAdd, 8 Conv2D, 8 Sub, 8 Maximum, 8 Pad, 8 RealDiv, 6 MaxPool, 1 Transpose, 1 StridedSlice,
1 Shape, 1 Reshape, 1 Placeholder, 1 Pack, 1 NoOp, 1 MatMul, 1 Identity
To use with tensorflow/tools/benchmark:benchmark_model try these arguments:
bazel run tensorflow/tools/benchmark:benchmark_model -- --graph=/mnt/c/Users/joana/Downloads/YOLO-models/tiny-yolov1.pb --show_flo
ps --input_layer=input --input_layer_type=float --input_layer_shape=-1,448,448,3 --output_layer=output
```

Figure A.6: Tiny-YOLOv1.1 model's specifications

Figure A.7: Tiny-YOLOv1.1 model

A.5 YOLOv2 [VOC]

```
Found 1 possible inputs: (name=input, type=float(1), shape=[?,416,416,3])
No variables spotted.
Found 1 possible outputs: (name=output, op=Identity)
Found 50676269 (50.68M) const parameters, 0 (0) variable parameters, and 0 control_edges
Op types used: 159 Const, 44 Mul, 23 BiasAdd, 23 Conv2D, 23 Pad, 22 Maximum, 22 RealDiv, 22 Sub, 5 MaxPool, 2 Identity, 1 ConcatV2,
1 ExtractImagePatches, 1 NoOp, 1 Placeholder
To use with tensorflow/tools/benchmark:benchmark_model try these arguments:
bazel run tensorflow/tools/benchmark:benchmark_model -- --graph=/mnt/c/Users/joana/Downloads/YOLO-models/yolov2-voc.pb --show_flops
--input_layer=input --input_layer_type=float --input_layer_shape=-1,416,416,3 --output_layer=output
```

Figure A.8: YOLOv2 [VOC] model's specifications

Figure A.9: YOLOv2 [VOC] model

A.6 Tiny-YOLOv2 [VOC]

```
Found 1 possible inputs: (name=input, type=float(1), shape=[?,416,416,3])
No variables spotted.
Found 1 possible outputs: (name=output, op=Identity)
Found 15867965 (15.87M) const parameters, 0 (0) variable parameters, and 0 control_edges
Op types used: 59 Const, 16 Mul, 9 BiasAdd, 9 Conv2D, 9 Pad, 8 Maximum, 8 RealDiv, 8 Sub, 6 MaxPool, 1 Identity, 1 NoOp, 1 Placeholder
To use with tensorflow/tools/benchmark:benchmark_model try these arguments:
bazel run tensorflow/tools/benchmark:benchmark_model -- --graph=/mnt/c/Users/joana/Downloads/YOLO-models/tiny-yolo-voc.pb --show_flops
--input_layer=input --input_layer_type=float --input_layer_shape=-1,416,416,3 --output_layer=output
```

Figure A.10: Tiny-YOLOv2 [VOC] model's specifications

Figure A.11: Tiny-YOLOv2 [VOC] model

A.7 YOLOv2 [COCO]

```
Found 1 possible inputs: (name=input, type=float(1), shape=[?,608,608,3])
No variables spotted.
Found 1 possible outputs: (name=output, op=Identity)
2020-02-26 12:43:43.256065: W tensorflow/core/framework/allocator.cc:113] Allocation of 37748736 exceeds 10% of system memory.
2020-02-26 12:43:43.298782: W tensorflow/core/framework/allocator.cc:113] Allocation of 37748736 exceeds 10% of system memory.
2020-02-26 12:43:43.340889: W tensorflow/core/framework/allocator.cc:113] Allocation of 47185920 exceeds 10% of system memory.
Found 50983769 (50.98M) const parameters, 0 (0) variable parameters, and 0 control_edges
Op types used: 159 Const, 44 Mul, 23 BiasAdd, 23 Conv2D, 23 Pad, 22 Maximum, 22 RealDiv, 22 Sub, 5 MaxPool, 2 Identity, 1 ConcatV2,
1 ExtractImagePatches, 1 NoOp, 1 Placeholder
To use with tensorflow/tools/benchmark:benchmark_model try these arguments:
bazel run tensorflow/tools/benchmark:benchmark_model -- --graph=/mnt/c/Users/joana/Downloads/YOLO-models/yolov2.pb --show_flops --in
put_layer=input --input_layer_type=float --input_layer_shape=-1,608,608,3 --output_layer=output
```

Figure A.12: YOLOv2 [COCO] model's specifications

Figure A.13: YOLOv2 [COCO] model

A.8 Tiny-YOLOv2 [VOC]

```
Found 1 possible inputs: (name=input, type=float(1), shape=[?,416,416,3])
No variables spotted.
Found 1 possible outputs: (name=output, op=Identity)
Found 11237225 (11.24M) const parameters, 0 (0) variable parameters, and 0 control_edges
Op types used: 59 Const, 16 Mul, 9 BiasAdd, 9 Conv2D, 9 Pad, 8 Maximum, 8 RealDiv, 8 Sub, 6 MaxPool, 1 Identity, 1 NoOp, 1 Placeholder
To use with tensorflow/tools/benchmark:benchmark_model try these arguments:
bazel run tensorflow/tools/benchmark:benchmark_model -- --graph=/mnt/c/Users/joana/Downloads/YOLO-models/yolov2-tiny.pb --show_flops --
input_layer=input --input_layer_type=float --input_layer_shape=-1,416,416,3 --output_layer=output
```

Figure A.14: Tiny-YOLOv2 [COCO] model's specifications

Figure A.15: Tiny-YOLOv2 [COCO] model

A.9 YOLOv3

```
Found 1 possible inputs: (name=inputs, type=float(1), shape=[?,416,416,3])
No variables spotted.
Found 1 possible outputs: (name=output_boxes, op=ConcatV2)
Found 62002037 (62.00M) const parameters, 0 (0) variable parameters, and 0 control_edges
Op types used: 537 Const, 372 Identity, 87 Mul, 75 Conv2D, 72 FusedBatchNorm, 72 Maximum, 28 Add, 25 Reshape, 14 ConcatV2, 9 Sigmoid,
6 Tile, 6 Range, 5 Pad, 4 SplitV, 3 Pack, 3 RealDiv, 3 Fill, 3 Exp, 3 BiasAdd, 2 ResizeNearestNeighbor, 2 Sub, 1 Placeholder
To use with tensorflow/tools/benchmark:benchmark_model try these arguments:
bazel run tensorflow/tools/benchmark:benchmark_model -- --graph=/mnt/c/Users/joana/Downloads/YoloV3/frozen_yolo_v3.pb --show_flops --i
nput_layer=inputs --input_layer_type=float --input_layer_shape=-1,416,416,3 --output_layer=output_boxes
```

Figure A.16: YOLOv3 model's specifications

Figure A.17: YOLOv3 model

A.10 Tiny-YOLOv3

```
Found 1 possible inputs: (name=inputs, type=float(1), shape=[?,416,416,3])
No variables spotted.
Found 1 possible outputs: (name=output_boxes, op=ConcatV2)
Found 8858858 (8.86M) const parameters, 0 (0) variable parameters, and 0 control_edges
Op types used: 134 Const, 63 Identity, 21 Mul, 16 Reshape, 13 Conv2D, 11 FusedBatchNorm, 11 Maximum, 10 ConcatV2, 6 Sigmoid, 6 MaxPool,
4 Tile, 4 Add, 4 Range, 3 RealDiv, 3 SplitV, 2 Pack, 2 Fill, 2 Exp, 2 Sub, 2 BiasAdd, 1 Placeholder, 1 ResizeNearestNeighbor
To use with tensorflow/tools/benchmark:benchmark_model try these arguments:
bazel run tensorflow/tools/benchmark:benchmark_model -- --graph=/mnt/c/Users/joana/Downloads/tiny-YoloV3/frozen_tiny_yolo_v3.pb --show_f
lops --input_layer=inputs --input_layer_type=float --input_layer_shape=-1,416,416,3 --output_layer=output_boxes
```

Figure A.18: Tiny-YOLOv3 model's specifications

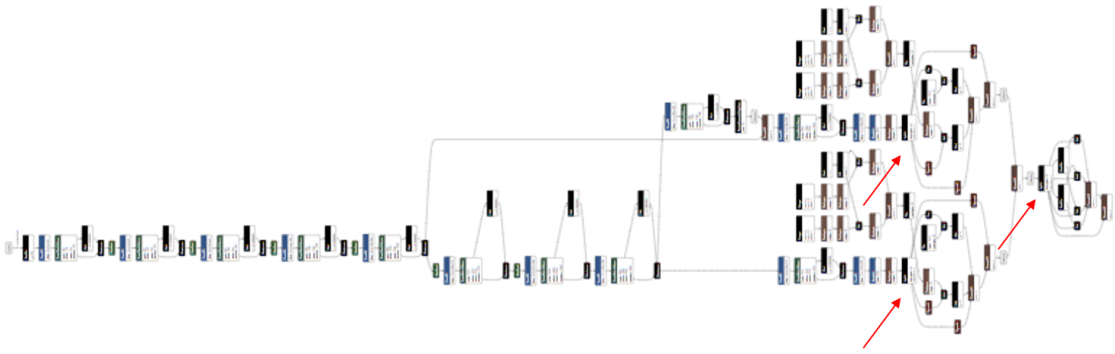


Figure A.19: Tiny-YOLOv3 model

Appendix B

YUV-420-888 Format

YUV-420-888 or YUV420p is a planar format, which means that Y, U, and V are grouped instead of interspersed to make the image more compressible than in other formats, such as RGB. In an array of an image in this format, Y values come first, followed by all the U values and followed finally by all the V values. The number of Y values is the same as image pixels since each pixel has a specific Y value. Each U and V values are assigned to 4 pixels. As shown in Figure B.1, Y, U, and V components are encoded separately in sequential blocks.

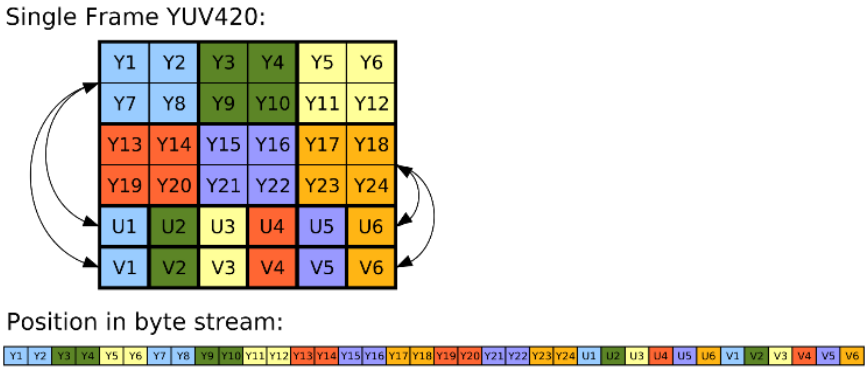


Figure B.1: Frame YUV420p

The Y block could be found at position 0 of the array, the U block at position $x * y$, where x corresponds to the number of columns and y to the number of lines, and the V block at position $x * y + \frac{(x*y)}{4}$. Thus, the image YUV buffer would have $\#Y * \frac{3}{2}$ bytes, and the RGB buffer would have double size.

We tried to convert the YUV buffer on the RGB buffer required by the neural network following the instructions available, but the constants required for conversion were quite variable, and we were never able to get the correct image in RGB format. We finally found that the OpenCV library could make this conversion directly. We used the `COLOR_YUV2RGB_NV12` flag to obtain the expected image.

However, according to the provided information, NV12 format is like YUV420p, but it has the order of U and V values switched, i.e., the Y values are followed by the V values, with the U values last. Moreover, NV21 is the standard picture format on the Android camera preview and is the inverse of NV12. Nonetheless, it made the image more bluish, which we did not understand.[1]

[1] more information about these formats at https://en.wikipedia.org/wiki/YUV [accessed 18 September 2019]

Appendix C

Protobuf

Google developed Protocol Buffers (Protobuf) to serialize structured data. It can be used for programs' development to communicate with each other over a wire or for storing data instead of XML or JSON. Protobuf was designed to be smaller and faster than these two approaches. The method involves an interface description language that describes the structure of some data and a program that generates source code from that description for generating or parsing a stream of bytes that represents the structured data.[1]

Since Google also developed TensorFlow, Protobuf is used to store the graph definition and the model's weights. Thus, the `.pb` file is all we need to be able to run a given trained model. The model's information can be saved on a Protobuf binary file, a Protobuf text file, or a Protobuf text in a string. We always used the first one to load the network on our application.

[1] more information about Protobuf at https://developers.google.com/protocol-buffers/docs/proto [accessed 20 September 2019]

Appendix D

Interconnection Between CV and AI

After the input preparation, we send the image to the network to obtain the output for further analysis. We mentioned in Section 4.2 that we had to create a new thread in the Controller module for the processing image part. Here we also prepared the image data to transform it into a network input. Since the programming language used in the acquisition is C# and in detection is C++, according to the diagram in Figure 4.2 from Section 4.2, we have to access some values in their corresponding memory positions in the C++ code, so we should pass by reference a specific type of variable, to make possible the communication between both sides.

To do this, we convert our buffer into an IntPtr object through the System.Runtime.InteropServices class. This class contains a method, Marshal.AllocHGlobal that allows us to allocate the same number of bytes as the unmanaged variable occupies and returns an IntPtr object that points to the beginning of the unmanaged block of memory. This class also has the Marshal.Copy method, that copies data from a managed array to an unmanaged memory pointer, or from an unmanaged memory pointer to a managed array. With this approach, it was also possible to do the opposite case, i.e., modify some values in their correspondent memory positions in the C++ code and access them in the C# code later.

Appendix E

Tiny-YOLOv2 Training

To improve the Tiny-YOLOv2 model accuracy, we tried different training approaches. There are two models available on the official YOLO website. One was pre-trained with the COCO trainval dataset[1] and the other one with Pascal VOC 2007+2012[2]. Their mAP results are quite different, probably due to the different test's dataset size, as mentioned in Section 2.1. Firstly, we took the version pre-trained with COCO and performed three training trials with the following datasets:

1. Pascal VOC trainval2012 + trainvaltest2007, during more than 150k steps;
2. Pascal VOC trainval2012: during more than 16k steps;
3. Pascal VOC trainval2012: during more than 60k steps

The three of them were tested with the Pascal VOC trainvaltest2007 and have shown a similar loss chart to the presented in Figure E.1.

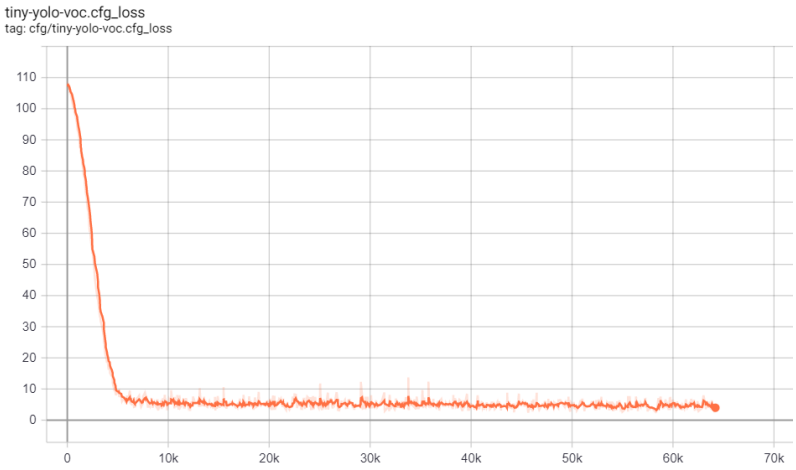


Figure E.1: Loss chart

[1] more information about Tiny-YOLOv2 pre-trained with COCO dataset at https://pjreddie.com/darknet/yolo/ [accessed 18 February 2020]

[2] more information about Tiny-YOLOv2 pre-trained with Pascal VOC dataset at https://pjreddie.com/darknet/yolov2/ [accessed 18 February 2020]

However, the model (2) showed the worst mAP result with 0.4% compared with (1) 24.9% and (3) 5.9%. All results are presented in Figures E.2 and E.3. The mAP difference between (1) and (3) is probably because the training of (1) included the dataset used for testing.

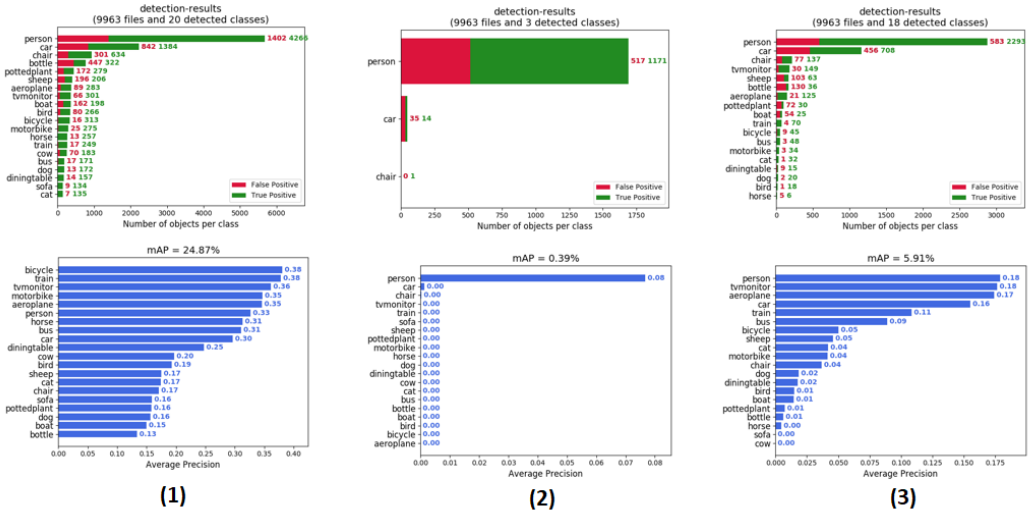


Figure E.2: Detection results and mAP of each model for the 1st trial

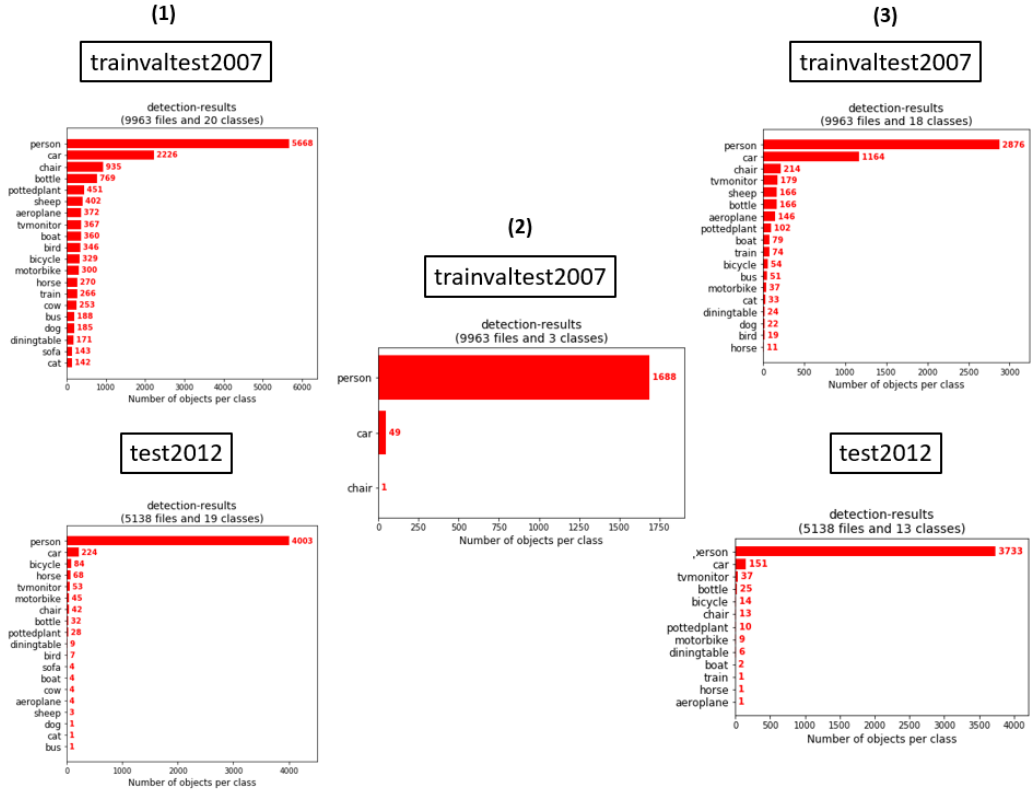


Figure E.3: Detection-results of each model

We ran a new test on (1) and (3) with just the Pascal VOC test2012 dataset and saw that the only detections were of the Person class, as shown in Figure E.4. After further inspection, as presented in Figures E.5 and E.6, we concluded that this was due to a high number of annotations being missing in the dataset, especially from the other classes. We also tried the Tiny-YOLOv2 model pre-trained with the Pascal VOC dataset. However, we did not notice any difference in the running time of our application when compared to (3), which we ended up using.

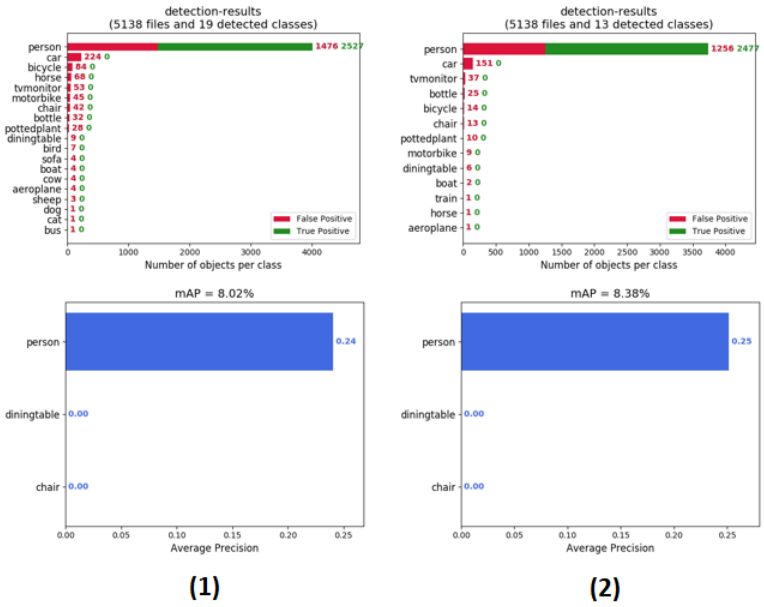


Figure E.4: Detection results and mAP of each model for the 2nd trial

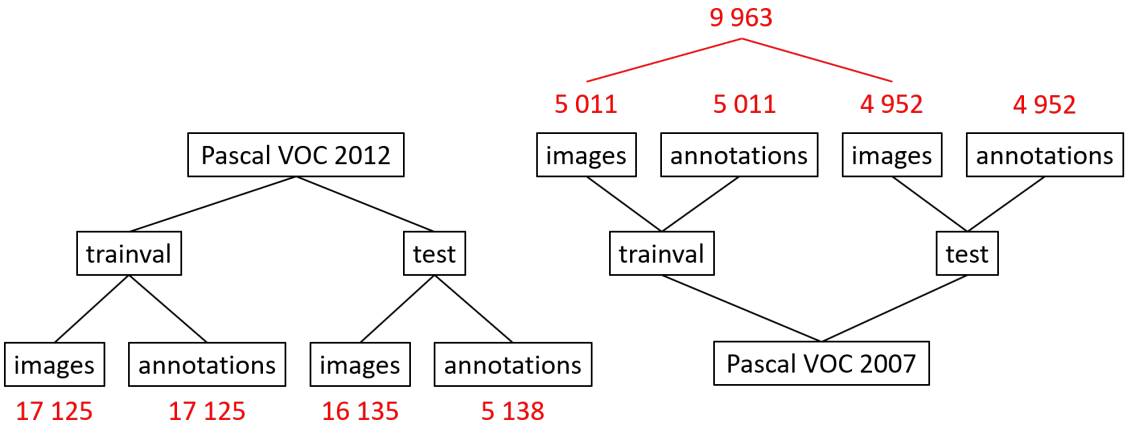


Figure E.5: Number of images and annotations in each dataset

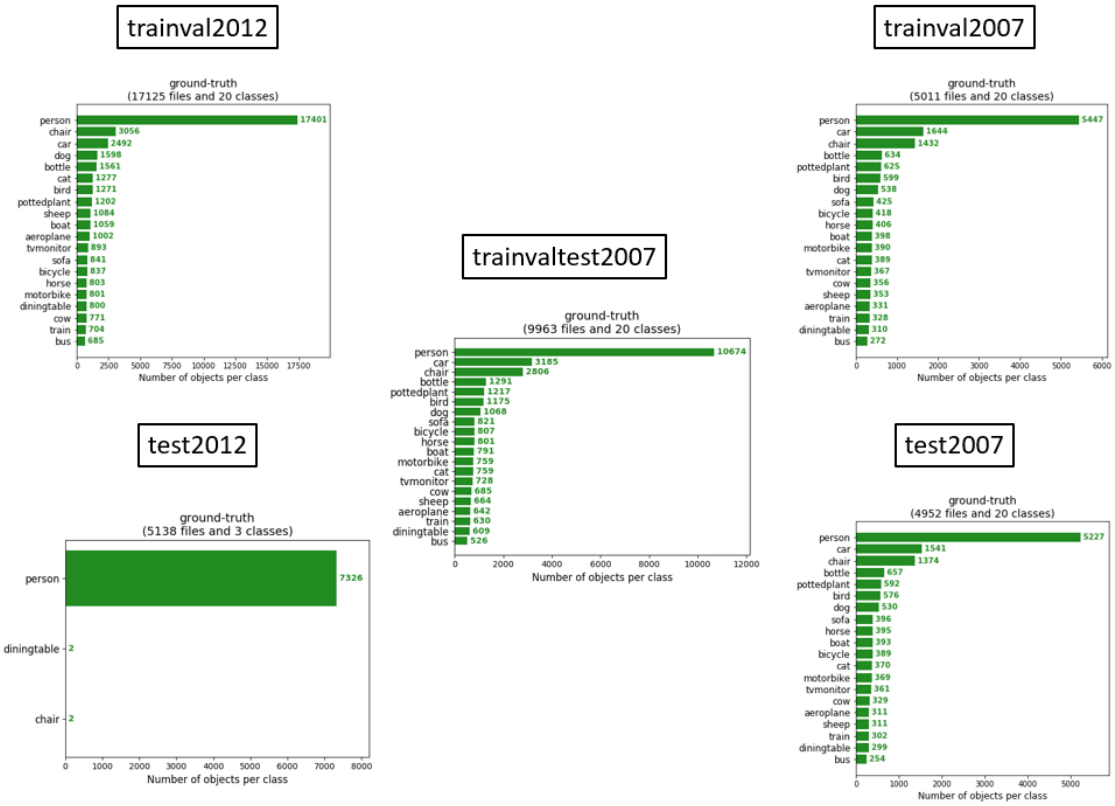


Figure E.6: Ground-truth of each dataset

References

[1] Tomislav Pejsa, Julian Kantor, Hrvoje Benko, Eyal Ofek, and Andrew D Wilson. Room2Room: Enabling Life-Size Telepresence in a Projected Augmented Reality Environment. In *Proceedings of the 19th ACM Conference on Computer-Supported Cooperative Work & Social Computing - CSCW '16*, pages 1714–1723, New York, New York, USA, 2016. ACM Press. URL: http://dl.acm.org/citation.cfm?doid=2818048.2819965, doi:10.1145/2818048.2819965.

[2] Thammathip Piumsomboon, Gun A. Lee, Jonathon D. Hart, Barrett Ens, Robert W. Lindeman, Bruce H. Thomas, and Mark Billinghurst. Mini-Me. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems - CHI '18*, pages 1–13, New York, New York, USA, 2018. ACM Press. URL: http://dl.acm.org/citation.cfm?doid=3173574.3173620, doi:10.1145/3173574.3173620.

[3] Martin Rünz, Maud Buffier, and Lourdes Agapito. MaskFusion: Real-Time Recognition, Tracking and Reconstruction of Multiple Moving Objects. In *2018 IEEE International Symposium on Mixed and Augmented Reality (ISMAR)*, apr 2018. URL: http://arxiv.org/abs/1804.09194, arXiv:1804.09194.

[4] David R. Walton and Anthony Steed. Accurate real-time occlusion for mixed reality. In *Proceedings of the 23rd ACM Symposium on Virtual Reality Software and Technology - VRST '17*, pages 1–10, New York, New York, USA, 2017. ACM Press. URL: http://dl.acm.org/citation.cfm?doid=3139131.3139153, doi:10.1145/3139131.3139153.

[5] Yeonjoon Kim, Hangil Park, Seungbae Bang, and Sung-Hee Lee. Retargeting Human-Object Interaction to Virtual Avatars. *IEEE Transactions on Visualization and Computer Graphics*, 22(11):2405–2412, nov 2016. URL: http://ieeexplore.ieee.org/document/7523447/, doi:10.1109/TVCG.2016.2593780.

[6] Andrey Ignatov, Radu Timofte, William Chou, Ke Wang, Max Wu, Tim Hartley, and Luc Van Gool. AI Benchmark: Running Deep Neural Networks on Android Smartphones. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 11133 LNCS:288–314, oct 2018. URL: http://arxiv.org/abs/1810.01109, arXiv:1810.01109.

[7] Ville Lehtola, Heikki Huttunen, Francois Christophe, and Tommi Mikkonen. Evaluation of visual tracking algorithms for embedded devices. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 10269 LNCS, pages 88–97. Springer Verlag, 2017. URL: http://link.springer.com/10.1007/978-3-319-59126-1{_}8, doi:10.1007/978-3-319-59126-1_8.

[8] Lei Gao, Huidong Bai, Weiping He, Mark Billinghurst, and Robert W. Lindeman. Real-time visual representations for mobile mixed reality remote collaboration. In *SIGGRAPH Asia 2018 Virtual & Augmented Reality on - SA '18*, pages 1–2, New York, New York, USA, 2018. ACM Press. URL: http://dl.acm.org/citation.cfm?doid=3275495.3275515, doi:10.1145/3275495.3275515.

[9] H. Durrant-Whyte and T. Bailey. Simultaneous localization and mapping: part I. *IEEE Robotics & Automation Magazine*, 13(2):99–110, jun 2006. URL: http://ieeexplore.ieee.org/document/1638022/, doi:10.1109/MRA.2006.1638022.

[10] T. Bailey and H. Durrant-Whyte. SLAM: part II. *IEEE Robotics & Automation Magazine*, 13(3):108–117, sep 2006. URL: http://ieeexplore.ieee.org/document/1678144/, doi:10.1109/MRA.2006.1678144.

[11] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross Girshick. Mask R-CNN. mar 2017. URL: http://arxiv.org/abs/1703.06870, arXiv:1703.06870.

[12] David G. Lowe. Distinctive image features from scale-invariant keypoints. *Int. J. Comput. Vision*, 60(2):91–110, November 2004. URL: https://doi.org/10.1023/B:VISI.0000029664.99615.94, doi:10.1023/B:VISI.0000029664.99615.94.

[13] Herbert Bay, Andreas Ess, Tinne Tuytelaars, and Luc Van Gool. Speeded-up robust features (surf). *Comput. Vis. Image Underst.*, 110(3):346–359, June 2008. URL: http://dx.doi.org/10.1016/j.cviu.2007.09.014, doi:10.1016/j.cviu.2007.09.014.

[14] Ethan Rublee, Vincent Rabaud, Kurt Konolige, and Gary Bradski. Orb: An efficient alternative to sift or surf. In *Proceedings of the 2011 International Conference on Computer Vision*, ICCV '11, pages 2564–2571, Washington, DC, USA, 2011. IEEE Computer Society. URL: http://dx.doi.org/10.1109/ICCV.2011.6126544, doi:10.1109/ICCV.2011.6126544.

[15] Arijit Mallick, Angel P. del Pobil, and Enric Cervera. Deep Learning based Object Recognition for Robot picking task. In *Proceedings of the 12th International Conference on Ubiquitous Information Management and Communication - IMCOM '18*, pages 1–9, New York, New York, USA, 2018. ACM Press. URL: http://dl.acm.org/citation.cfm?doid=3164541.3164628, doi:10.1145/3164541.3164628.

[16] Ross B. Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. *CoRR*, abs/1311.2524, 2013. URL: http://arxiv.org/abs/1311.2524, arXiv:1311.2524.

[17] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision*, 115(3):211–252, dec 2015. URL: http://link.springer.com/10.1007/s11263-015-0816-y, doi:10.1007/s11263-015-0816-y.

[18] Ross Girshick. Fast R-CNN. apr 2015. URL: http://arxiv.org/abs/1504.08083, arXiv:1504.08083.

[19] Shaoqing Ren, Kaiming He, Ross B. Girshick, and Jian Sun. Faster R-CNN: towards real-time object detection with region proposal networks. *CoRR*, abs/1506.01497, 2015. URL: http://arxiv.org/abs/1506.01497, arXiv:1506.01497.

[20] Jifeng Dai, Yi Li, Kaiming He, and Jian Sun. R-FCN: object detection via region-based fully convolutional networks. *CoRR*, abs/1605.06409, 2016. URL: http://arxiv.org/abs/1605.06409, arXiv:1605.06409.

[21] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott E. Reed, Cheng-Yang Fu, and Alexander C. Berg. SSD: single shot multibox detector. *CoRR*, abs/1512.02325, 2015. URL: http://arxiv.org/abs/1512.02325, arXiv:1512.02325.

[22] Tsung-Yi Lin, Piotr Dollár, Ross B. Girshick, Kaiming He, Bharath Hariharan, and Serge J. Belongie. Feature pyramid networks for object detection. *CoRR*, abs/1612.03144, 2016. URL: http://arxiv.org/abs/1612.03144, arXiv:1612.03144.

[23] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You Only Look Once: Unified, Real-Time Object Detection. jun 2015. URL: http://arxiv.org/abs/1506.02640, arXiv:1506.02640.

[24] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1–9. IEEE, jun 2015. URL: http://ieeexplore.ieee.org/document/7298594/, doi:10.1109/CVPR.2015.7298594.

[25] Joseph Redmon and Ali Farhadi. YOLO9000: Better, Faster, Stronger. dec 2016. URL: http://arxiv.org/abs/1612.08242, arXiv:1612.08242.

[26] Joseph Redmon and Ali Farhadi. YOLOv3: An Incremental Improvement. apr 2018. URL: http://arxiv.org/abs/1804.02767, arXiv:1804.02767.

[27] Shivam Duggal, Shrey Manik, and Mohan Ghai. Amalgamation of Video Description and Multiple Object Localization using single Deep Learning Model. In *Proceedings of the 9th International Conference on Signal Processing Systems - ICSPS 2017*, pages 109–115, New York, New York, USA, 2017. ACM Press. URL: http://dl.acm.org/citation.cfm?doid=3163080.3163108, doi:10.1145/3163080.3163108.

[28] Jiahuan Zhou, Lihang Feng, Ryad Chellali, and Haonan Zhu. Detecting and tracking objects in HRI: YOLO networks for the NAO "I See You" function *. In *2018 27th IEEE International Symposium on Robot and Human Interactive Communication (RO-MAN)*, pages 479–482. IEEE, aug 2018. URL: https://ieeexplore.ieee.org/document/8525582/, doi:10.1109/ROMAN.2018.8525582.

[29] Volodymyr Kharchenko and Iurii Chyrka. Detection of Airplanes on the Ground Using YOLO Neural Network. In *2018 IEEE 17th International Conference on Mathematical Methods in Electromagnetic Theory (MMET)*, pages 294–297. IEEE, jul 2018. URL: https://ieeexplore.ieee.org/document/8460392/, doi:10.1109/MMET.2018.8460392.

[30] Hong-wei Zhang, Ling-jie Zhang, Peng-fei Li, and De Gu. Yarn-dyed Fabric Defect Detection with YOLOV2 Based on Deep Convolution Neural Networks. In *2018 IEEE 7th Data Driven Control and Learning Systems Conference (DDCLS)*, pages 170–174. IEEE, may 2018. URL: https://ieeexplore.ieee.org/document/8516094/, doi:10.1109/DDCLS.2018.8516094.

[31] Alexander Wong, Mohammad Javad Shafiee, Francis Li, and Brendan Chwyl. Tiny SSD: A Tiny Single-shot Detection Deep Convolutional Neural Network for Real-time Embedded Object Detection. feb 2018. URL: http://arxiv.org/abs/1802.06488, arXiv:1802.06488.

[32] Forrest N. Iandola, Song Han, Matthew W. Moskewicz, Khalid Ashraf, William J. Dally, and Kurt Keutzer. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size. feb 2016. URL: http://arxiv.org/abs/1602.07360, arXiv:1602.07360.

[33] Xiaoyu Feng, Wei Mei, and Dashuai Hu. A review of visual tracking with deep learning. 01 2016. doi:10.2991/aiie-16.2016.54.

[34] Lijun Wang, Wanli Ouyang, Xiaogang Wang, and Huchuan Lu. Visual tracking with fully convolutional networks. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 3119–3127. Institute of Electrical and Electronics Engineers Inc., 2015. URL: https://ieeexplore.ieee.org/document/7410714, doi:10.1109/ICCV.2015.357.

[35] Hyeonseob Nam and Bohyung Han. Learning Multi-Domain Convolutional Neural Networks for Visual Tracking. *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2016-December:4293–4302, oct 2015. URL: http://arxiv.org/abs/1510.07945, arXiv:1510.07945.

[36] Liangliang Ren, Jiwen Lu, Zifeng Wang, Qi Tian, and Jie Zhou. Collaborative Deep Reinforcement Learning for Multi-object Tracking. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 11207 LNCS, pages 605–621. Springer Verlag, sep 2018. doi:10.1007/978-3-030-01219-9_36.

[37] Kiana Ehsani, Hessam Bagherinezhad, Joseph Redmon, Roozbeh Mottaghi, and Ali Farhadi. Who Let The Dogs Out? Modeling Dog Behavior From Visual Data. Technical report. URL: https://pjreddie.com/media/files/papers/1803.10827.pdf, arXiv:1803.10827v1.

[38] Samuel Scheidegger, Joachim Benjaminsson, Emil Rosenberg, Amrit Krishnan, and Karl Granström. Mono-camera 3d multi-object tracking using deep learning detections and PMBM filtering. *CoRR*, abs/1802.09975, 2018. URL: http://arxiv.org/abs/1802.09975, arXiv:1802.09975.

[39] Adrien Gaidon, Qiao Wang, Yohann Cabon, and Eleonora Vig. Virtual Worlds as Proxy for Multi-Object Tracking Analysis. In *Conference on Computer Vision and Pattern Recognition*, 2016. URL: https://arxiv.org/pdf/1605.06457.

[40] David Held, Sebastian Thrun, and Silvio Savarese. Learning to Track at 100 FPS with Deep Regression Networks. pages 749–765. 2016. URL: http://link.springer.com/10.1007/978-3-319-46448-0{_}45, doi:10.1007/978-3-319-46448-0_45.

[41] Chaoyang Wang, Hamed Kiani Galoogahi, Chen-Hsuan Lin, and Simon Lucey. Deep-LK for Efficient Adaptive Object Tracking. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 627–634. IEEE, may 2018. URL: https://ieeexplore.ieee.org/document/8460815/, doi:10.1109/ICRA.2018.8460815.

[42] Rui Nóbrega and Nuno Correia. Interactive 3D content insertion in images for multimedia applications. *Multimedia Tools and Applications*, 76(1):163–197, jan 2017. URL: http://link.springer.com/10.1007/s11042-015-3031-5, doi:10.1007/s11042-015-3031-5.

[43] Adrien Arnaud, Julien Christophe, Michèle Gouiffes, and Mehdi Ammi. 3D reconstruction of indoor building environments with new generation of tablets. In *Proceedings of the 22nd ACM Conference on Virtual Reality Software and Technology - VRST '16*, pages 187–190, New York, New York, USA, 2016. ACM Press. URL: http://dl.acm.org/citation.cfm?doid=2993369.2993403, doi:10.1145/2993369.2993403.

[44] Shahram Izadi, Andrew Davison, Andrew Fitzgibbon, David Kim, Otmar Hilliges, David Molyneaux, Richard Newcombe, Pushmeet Kohli, Jamie Shotton, Steve Hodges, and Dustin Freeman. KinectFusion. In *Proceedings of the 24th annual ACM symposium on User interface software and technology - UIST '11*, page 559, New York, New York, USA, 2011. ACM Press. URL: http://dl.acm.org/citation.cfm?doid=2047196.2047270, doi:10.1145/2047196.2047270.

[45] Misha Sra, Sergio Garrido-Jurado, Chris Schmandt, and Pattie Maes. Procedurally generated virtual reality from 3D reconstructed physical space. In *Proceedings of the 22nd ACM Conference on Virtual Reality Software and Technology - VRST '16*, pages 191–200, New York, New York, USA, 2016. ACM Press. URL: http://dl.acm.org/citation.cfm?doid=2993369.2993372, doi:10.1145/2993369.2993372.

[46] Chao Du, Yen-Lin Chen, Mao Ye, and Liu Ren. Edge Snapping-Based Depth Enhancement for Dynamic Occlusion Handling in Augmented Reality. In *2016 IEEE International Symposium on Mixed and Augmented Reality (ISMAR)*, pages 54–62. IEEE, sep 2016. URL: http://ieeexplore.ieee.org/document/7781766/, doi:10.1109/ISMAR.2016.17.

[47] Johan Kasperi, Malin Picha Edwardsson, and Mario Romero. Occlusion in outdoor augmented reality using geospatial building data. In *Proceedings of the 23rd ACM Symposium on Virtual Reality Software and Technology - VRST '17*, pages 1–10, New York, New York, USA, 2017. ACM Press. URL: http://dl.acm.org/citation.cfm?doid=3139131.3139159, doi:10.1145/3139131.3139159.

[48] Michael Bonfert, Inga Lehne, Ralf Morawe, Melina Cahnbley, Gabriel Zachmann, and Johannes Schöning. Augmented invaders. In *Proceedings of the 23rd ACM Symposium on Virtual Reality Software and Technology - VRST '17*, pages 1–2, New York, New York, USA, 2017. ACM Press. URL: http://dl.acm.org/citation.cfm?doid=3139131.3141208, doi:10.1145/3139131.3141208.

[49] Julien Casarin, Dominique Bechmann, and Marilyn Keller. A unified model for interaction in 3D environment. In *Proceedings of the 23rd ACM Symposium on Virtual Reality Software and Technology - VRST '17*, pages 1–7, New York, New York, USA, 2017. ACM Press. URL: http://dl.acm.org/citation.cfm?doid=3139131.3139140, doi:10.1145/3139131.3139140.

[50] Jonathan Pedoeem and Rachel Huang. YOLO-LITE: A Real-Time Object Detection Algorithm Optimized for Non-GPU Computers. *Proceedings - 2018 IEEE International Conference on Big Data, Big Data 2018*, pages 2503–2510, nov 2018. URL: http://arxiv.org/abs/1811.05588, arXiv:1811.05588.

[51] João F. Henriques, Rui Caseiro, Pedro Martins, and Jorge Batista. High-Speed Tracking with Kernelized Correlation Filters. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 37(3):583–596, apr 2014. URL: http://arxiv.org/abs/1404.7584http://dx.doi.org/10.1109/TPAMI.2014.2345390, arXiv:1404.7584, doi:10.1109/TPAMI.2014.2345390.