



**UNIVERSIDAD DE INVESTIGACIÓN DE TECNOLOGÍA
EXPERIMENTAL YACHAY**

Escuela de Ciencias Matemáticas y Computacionales

**TITULO: IMPLEMENTING YOLO ALGORITHM FOR REAL TIME
OBJECT DETECTION ON EMBEDDED SYSTEM**

Trabajo de integración curricular presentado como requisito para
la obtención del título de Ingeniero en Tecnologías de la
Información

Autor:

Silva Pincay Paul Andre

Tutor:

PhD Guachi Guachi, Lorena de los Angeles

Cotutor:

PhD Ortega-Zamorano, Francisco

Urcuquí, agosto 2019

Urucuquí, 19 de agosto de 2019

SECRETARÍA GENERAL
(Vicerrectorado Académico/Cancillería)
ESCUELA DE CIENCIAS MATEMÁTICAS Y COMPUTACIONALES
CARRERA DE TECNOLOGÍAS DE LA INFORMACIÓN
ACTA DE DEFENSA No. UITEY-ITE-2019-00001-AD

En la ciudad de San Miguel de Urucuquí, Provincia de Imbabura, a los 19 días del mes de agosto de 2019, a las 09:30 horas, en el Aula AI-101 de la Universidad de Investigación de Tecnología Experimental Yachay y ante el Tribunal Calificador, integrado por los docentes:

Presidente Tribunal de Defensa ARMAS ARCINIEGA, JULIO JOAQUIN

Tutor GUACHI GUACHI, LORENA DE LOS ANGELES

Se presenta el(la) señor(ita) estudiante **SILVA PINCAY, PAUL ANDRE**, con cédula de identidad No. 0925542821, de la **ESCUELA DE CIENCIAS MATEMÁTICAS Y COMPUTACIONALES**, de la Carrera de **TECNOLOGÍAS DE LA INFORMACIÓN**, aprobada por el Consejo de Educación Superior (CES), mediante Resolución RPC-SO-43-No.496-2014, con el objeto de rendir la sustentación de su trabajo de titulación denominado: **Implementing YOLO Algorithm for Real Time Object Detection on Embedded System**, previa a la obtención del título de **INGENIERO/A EN TECNOLOGÍAS DE LA INFORMACIÓN**.

El citado trabajo de titulación, fue debidamente aprobado por el(los) docente(s):

Tutor GUACHI GUACHI, LORENA DE LOS ANGELES

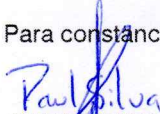
Y recibió las observaciones de los otros miembros del Tribunal Calificador, las mismas que han sido incorporadas por el(la) estudiante.

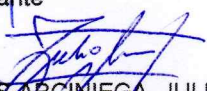
Previamente cumplidos los requisitos legales y reglamentarios, el trabajo de titulación fue sustentado por el(la) estudiante y examinado por los miembros del Tribunal Calificador. Escuchada la sustentación del trabajo de titulación, que integró la exposición de el(la) estudiante sobre el contenido de la misma y las preguntas formuladas por los miembros del Tribunal, se califica la sustentación del trabajo de titulación con las siguientes calificaciones:


Tipo	Docente	Calificación
Tutor	GUACHI GUACHI, LORENA DE LOS ANGELES	10,0
Presidente Tribunal De Defensa	ARMAS ARCINIEGA, JULIO JOAQUIN	9,0
Miembro Tribunal De Defensa	IZA PAREDES, CRISTHIAN	10,0

Lo que da un promedio de: **9.6 (Nueve punto Seis)**, sobre 10 (diez), equivalente a: **APROBADO**

Para constancia de lo actuado, firman los miembros del Tribunal Calificador, el/la estudiante y el/la secretario ad-hoc.


SILVA PINCAY, PAUL ANDRE
Estudiante


ARMAS ARCINIEGA, JULIO JOAQUIN
Presidente Tribunal de Defensa


GUACHI GUACHI, LORENA DE LOS ANGELES
Tutor

Torres Montalván

TORRES MONTALVÁN, TATIANA BEATRIZ
Secretario Ad-hoc

SECRETARÍA GENERAL

(Vicecoordinador Académico/Coordinador)
ESCUELA DE CIENCIAS MATEMÁTICAS Y COMPUTACIONALES
CARRERA DE TECNOLOGÍAS DE LA INFORMACIÓN
ACTA DE DEFENSA No. UITEY-TE-2019-0001-AD

En la ciudad de San Miguel de Urcuquí, Provincia de Imbabura, a los 19 días del mes de agosto de 2019, a las 09:30 horas en el Aula A1-01 de la Universidad de Investigación de Tecnología Experimental Yachay y ante el Tribunal Calificador integrado por los docentes:

Christian JFA
CRISTIAN JFA

Presidente Tribunal de Defensa: ARMAS ARCINIEGA JULIO JOAQUIN
Tutor: GUACHI GUACHI LORENA DE LOS ANGELES

Se presenta el(los) señor(ías) estudiante SILVA PINOY, PAUL ANDRE, con cédula de identidad No. 0823442821, de la ESCUELA DE CIENCIAS MATEMÁTICAS Y COMPUTACIONALES, de la Carrera de TECNOLOGÍAS DE LA INFORMACIÓN, aprobada por el Consejo de Educación Superior (CES), mediante Resolución RRC-SC-43-NO-488-2014, con el objeto de rendir la sustentación de su trabajo de titulación denominada: Implementing YOLO Algorithm for Real Time Object Detection on Embedded System, previa a la obtención del título de INGENIERÍA EN TECNOLOGÍAS DE LA INFORMACIÓN.

El citado trabajo de titulación, fue debidamente aprobado por el(los) docente(s):

Tutor: GUACHI GUACHI LORENA DE LOS ANGELES

Y recibió las observaciones de los otros miembros del Tribunal Calificador, las mismas que han sido incorporadas por el(los) estudiantes.

Previamente cumplidos los requisitos legales y reglamentarios, el trabajo de titulación fue sustentado por el(los) estudiante y examinado por los miembros del Tribunal Calificador. Escuchada la sustentación del trabajo de titulación, que integró la exposición de el(los) estudiante sobre el contenido de la misma y las preguntas formuladas por los miembros del Tribunal, se califica la sustentación del trabajo de titulación con las siguientes calificaciones:

Tipo	Docente	Calificación
Tutor	GUACHI GUACHI LORENA DE LOS ANGELES	10.0
Presidente Tribunal De Defensa	ARMAS ARCINIEGA JULIO JOAQUIN	9.0
Miembro Tribunal De Defensa	CA PAREDES CRISTIAN	10.0

Lo que da un promedio de: 9.8 (nueve punto seis), sobre 10 (diez), equivalente a: APROBADO

Para constancia de lo actuado, firman los miembros del Tribunal Calificador, el(los) estudiante y el(los) secretario ad-hoc:

Silva Pinoy
SILVA PINOY, PAUL ANDRE
Estudiante

Armas Arciniega
ARMAS ARCINIEGA JULIO JOAQUIN
Presidente Tribunal de Defensa

Guachi Guachi
GUACHI GUACHI LORENA DE LOS ANGELES
Tutor

AUTORÍA

Yo, **PAUL ANDRE SILVA PINCAY**, con cédula de identidad 0925542821, declaro que las ideas, juicios, valoraciones, interpretaciones, consultas bibliográficas, definiciones y conceptualizaciones expuestas en el presente trabajo; así como, los procedimientos y herramientas utilizadas en la investigación, son de absoluta responsabilidad de el/la autor(a) del trabajo de integración curricular. Así mismo, me acojo a los reglamentos internos de la Universidad de Investigación de Tecnología Experimental Yachay.

Urcuquí, Agosto 2019.



PAUL ANDRE SILVA PINCAY
CI: 0925542821

AUTORIZACIÓN DE PUBLICACIÓN

Yo, **PAUL ANDRE SILVA PINCAY**, con cédula de identidad 0925542821, cedo a la Universidad de Investigación de Tecnología Experimental Yachay, los derechos de publicación de la presente obra, sin que deba haber un reconocimiento económico por este concepto. Declaro además que el texto del presente trabajo de titulación no podrá ser cedido a ninguna empresa editorial para su publicación u otros fines, sin contar previamente con la autorización escrita de la Universidad.

Asimismo, autorizo a la Universidad que realice la digitalización y publicación de este trabajo de integración curricular en el repositorio virtual, de conformidad a lo dispuesto en el Art. 144 de la Ley Orgánica de Educación Superior

Urcuquí, Agosto 2019.



PAUL ANDRE SILVA PINCAY
CI: 0925542821

Resumen

La detección de objetos se ocupa de la clasificación y localización de múltiples objetos en imágenes y videos al usar un tipo particular de redes neuronales artificiales conocidas como Redes Neuronales Convolutivas. Una de las redes que ha ganado reconocimiento en el área de visión computacional es YOLO, un algoritmo que es un conjunto de múltiples técnicas usadas para mejorar el rendimiento en términos de velocidad y precisión en detectar objetos en imágenes. Mayormente, YOLO es conocido por ser rápido en comparación a otros detectores de objetos modernos que también usan GPUs para entrenamiento y detección. Desafortunadamente, las GPUs son caras y no accesibles a todos de manera que limita el número de aplicaciones potenciales que pueden ser implementadas. En ese sentido, este trabajo se enmarca en implementar YOLO para detección en sistemas embebidos como el Raspberry Pi. A pesar de que este dispositivo es computacionalmente limitado, esto fue una gran oportunidad de ver si esta tarea era posible debido a que el Raspberry Pi es una alternativa de bajo costo más accesible. El trabajo abarca una pequeña descripción sobre detección de objetos y redes neuronales convolutivas con una revisión más detallada de las técnicas, aplicaciones y limitaciones de YOLO. También presenta detalles sobre los parámetros de entrenamiento usados para mejorar la precisión de YOLO y las pruebas realizadas en el Raspberry Pi usando videos.

Palabras claves: detección de objetos, redes neuronales convolutivas, YOLO, parámetros de entrenamiento, Raspberri Pi.

Abstract

Object detection deals with classifying and locating multiples objects in images and videos by using a specific type of artificial neural networks known as Convolutional Neural Networks. One of the networks that have gained recognition in this area of computer vision is YOLO, an algorithm that is a collection of multiple techniques used to improve performance both in terms of speed and precision to detect objects in images. Mostly, YOLO is known for being fast in comparison to other state-of-the-art object detectors which all use GPUs for training and detection. Unfortunately, GPUs are expensive and not accessible to everyone thus limiting the number of potential applications that can be implemented. In that sense, this work is focused on implementing YOLO for detection on an embedded system such as the Raspberry Pi. Although this device is computationally constrained, it represented a great chance to see if this feat was possible as the Raspberry Pi is a more accessible low cost alternative. The work covers a small description on object detection and convolutional neural networks with a more detailed overview of the techniques, applications and challenges of YOLO. It also presents details on the training parameters considered to improved YOLO's precision and the tests performed on the Raspberry Pi using videos.

Keywords: object detection, convolutional neural networks, YOLO, training parameters, Raspberry Pi.

Acknowledgements

First of all, I would like to express my gratitude to my thesis advisors Lorena and Francisco who gave me enough encouragement to develop this project. Their constant support and insights helped me in the research and I got the chance to learn about so many interesting things. Without their guidance, patience and persistence, this thesis project would have not been possible. I am also grateful to all the members of the School of Mathematical and Computational Sciences from the staff, professors, classmates to my dean Andreas Griewank. Thanks for all the great teachings and experiences during my time at Yachay Tech. I immensely thank my parents Jose and Trinidad and my siblings Bryan, Karina and Maria Jose for their unwavering patience, support and love during the preparation of this project. They have always helped me to accomplish too many things in my life.

Contents

List of Figures	iii
List of Tables	v
Abbreviations	viii
1 Introduction	1
1.1 Problem statement	1
1.2 Scope of the Project	2
1.3 General Overview	2
2 Theoretical Framework	3
2.1 Concepts	3
2.2 Object Detection	4
2.2.1 Applications	5
2.2.2 Challenges	6
2.2.3 Features	7
2.3 Convolutional Neural Networks for Object Detection	8
2.3.1 CNN Architecture	8
2.3.2 CNNs Detectors	9
2.3.3 CNNs on Embedded Systems	10
2.3.4 CNNs challenges for Object Detection on Embedded Systems	11
2.4 YOLO	12
2.4.1 Description	13
2.4.2 General Architecture	18
2.4.3 Versions	19

2.4.4	Applications	23
2.4.5	YOLO's Challenges	24
2.4.6	YOLO on Embedded Systems	25
3	YOLO on Raspberry Pi	27
3.1	General Thesis Structure	27
3.1.1	Literature Review	28
3.1.2	Experimental Design	28
3.1.3	Thesis Writing	29
3.2	Materials and Methods	29
3.2.1	Dataset	29
3.2.2	Architecture of Tiny YOLOv3	31
3.2.3	Training Parameters	32
3.2.4	Metrics	33
3.2.5	Training	38
3.2.6	Software and Hardware	39
3.2.7	Deployment	40
3.3	Experimental Design	41
3.3.1	Darknet Simplification for YOLO	41
3.3.2	Training Parameter Tuning	42
3.3.3	Video Performance Evaluation	43
4	Results	45
4.1	Original YOLO vs Simplified YOLO	45
4.2	Tuning the parameters	48
4.3	Video	51
5	Conclusions	55
5.1	Literature Review	55
5.2	Experiments	56
5.3	Results	56
5.4	Future Work	57
	Bibliography	59

List of Figures

2.1	Convolution example.	9
2.2	Basic Convolutional Neural Networks (CNN) Architecture	9
2.3	Max-pooling example.	10
2.4	AP _{.50} vs Inference time of common detection methods.	12
2.5	Bounding Box Prediction	15
2.6	You Only Look Once (YOLO) detection network	19
3.1	Example of types of detections	34
3.2	Input image vs Output Detection.	41
4.1	Precision-Recall Curve for 10 first categories	46
4.2	AP _{.50} growth for original and simplified version	47
4.3	AP _{.50} growth for the first subset of experiments.	49

List of Tables

2.1	Techniques used in the different versions of YOLO.	13
2.2	YOLO and Darknet-19	20
2.3	Darknet-53	21
2.4	YOLO classifier vs YOLO detector.	22
2.5	YOLOv2 classifier vs YOLOv2 detector.	22
2.6	YOLOv3 classifier vs YOLOv3 detector.	23
3.1	ImageNet high-level categories.	30
3.2	Common Objects in Context (COCO) 80 object categories	31
3.3	YOLOv3-Tiny Architecture.	32
3.4	Hardware for testing	40
3.5	Training summary for parameter tuning	42
3.6	Video characteristics.	43
4.1	mAP with different test set.	47
4.2	Original vs Simplified.	47
4.3	Initial Training - 100000 Iterations.	49
4.4	Intermediate Training - 100000 Iterations.	50
4.5	Final Training.	50
4.6	Frames per Second (FPS) on computer vs Raspberry Pi.	52
4.7	Total detections on computer vs Raspberry Pi.	53

Abbreviations

B Batch Size.

CNN Convolutional Neural Networks.

COCO Common Objects in Context.

FN False Negative.

FP False Positive.

FPN Feature Pyramid Network.

FPS Frames per Second.

GPU Graphics Processing Unit.

IoU Intersection over Union.

IR Input Resolution.

LR Learning Rate.

mAP mean Average Precision.

RPN Region Proposal Network.

S Step.

SSD Single Shot Multibox Detector.

Subd Subdivision.

TN True Negative.

TP True Positive.

YOLO You Only Look Once.

Chapter 1

Introduction

The Internet of Things has brought a wide range of applications that improve the lifestyle of people. Small sensors and devices can be found almost everywhere now and they are connected and sending information between each other and external operators that take decisions over the data received. These applications have become critical and are prevalent areas of research nowadays. One of them is the use of computer vision for object detection in applications such as autonomous driving, video surveillance, navigation aid, and robotics by using a special type of artificial neural networks known as CNN. These networks are very powerful and have demonstrated great precision at the moment of detecting objects but there still a lot of work to do, particularly for real-time applications due to their expected great performance, and thus, their high computational requirements.

1.1 Problem statement

The ability of understanding image and video sequences to automatically recognize and detect objects is one of the most challenging problems in computer vision. Modern high-speed technologies and its progressive increase in memory and processing capacity have allowed the use of PCs to perform object detection, which in the past had been restricted to few fields due to the computational cost. The problem is that in many cases these systems are not suitable for embedded solutions, since PCs limit the portability due to high power consumption, size and weight. Although there are many algorithms developed for object detection and that work pretty well [1], there is a barrier that stops this technology in being ubiquitous which is the use of a Graphics Processing Unit (GPU). Granted, GPUs are imperative in the training and

validation of computer vision algorithms but are an expensive piece of hardware that is not accessible to everyone for execution in real environments. Actually, Raspberry Pi is an inexpensive alternative to implement techniques suitable for low-cost embedded systems. By the other hand, YOLO algorithm [2], [3], [4] has gained a wide application as a CNN based object detection approach, which has demonstrated to achieve a remarkable object detection precision alongside a good performance in comparison to other approaches like Faster R-CNN [5] and Single Shot Multibox Detector (SSD) [6]. Therefore, our main purpose is to implement YOLO in a constrained environment finding a way to do only the detection task outside the scope of GPUs in order to achieve precision and speed of YOLO in execution task within the Raspberry Pi platform, which could facilitate the development of more applications. The main metrics to evaluate these two characteristics are mean Average Precision (mAP) and FPS respectively.

1.2 Scope of the Project

Starting with the hypothesis that a simplification of code allows to implement and execute YOLO within the Raspberry Pi, the aims of this project are:

1. Evaluate the efficiency and performance of YOLO for real-time object detection.
2. Explore ways to increase YOLO's precision.
3. Evaluate YOLO performance on a low cost and accessible embedded system like the Raspberry Pi.

This project consists of the following phases: a literature review, experimental design and thesis writing. Each of these phases contributes to fulfilling the objectives of this research.

1.3 General Overview

The rest of this work is organized in the following way: Chapter 2 covers essential concepts and theory related to object detection, CNNs and YOLO. In Chapter 3, a detailed explanation of the methodology followed in this work, which includes the materials, methods and experimental design, is discussed. Chapter 4 goes through a comprehensive analysis of the results obtained following the methodology. Finally, Chapter 5 concludes this thesis with a summary and directions for future work.

Chapter 2

Theoretical Framework

This chapter contains a look on concepts and published work related to the scope of this project. The first part introduces some basic concepts. Afterwards, a little revision of object detection in terms of applications, challenges and features is presented. Then, a review covering in general some aspects of CNNs, the general architecture, the work done on embedded systems and some additional considerations is shown. Finally, the last section gives details related to YOLO like important features, challenges, versions and how it does on embedded systems so far.

2.1 Concepts

- **Activation Function**

The activation function defines the output of an artificial neural network given one input or a group of inputs. There are many different types of activation functions that work depending on the application such as sigmoid, hyperbolic tangent, ReLu, leaky ReLU, among others.

- **Embedded System**

An embedded system [7] is a type of computer hardware with integrated software designed to be used for a specific application. It is often limited on processing power, memory and storage; and is intended for working on real-time.

- **Feature Map**

A feature map is a matrix that contains the information of the image. If the image has three channels (like a RGB image) then there is a feature map for each of the three channels.

- **Frames per Second**

FPS is the rate that measures the number of frames, each frame represents a single image, that can be processed in a second by the model. This usually is a metric for detection speed.

- **Object Detection**

The object detection task allows for the identification of real-world objects in images or videos. In recent years, most of the algorithms in this area of research use CNNs due to great success in computer vision applications, such as image and video recognition and classification.

- **Overfitting**

Overfitting occurs when a network does a pretty good job learning data on the training dataset but has a bad performance when using a different dataset.

- **Raspberry Pi**

The Raspberry Pi is a micro-controller that can be both an embedded system or general purpose computer depending on the application. Some characteristics [7] are: it is a low cost device, model 3B has a 1.2 GHz ARM Cortex-A53 processor (still outperformed by a personal computer), consumes little power with model 3B consuming up to 5.5 W under load and runs Linux.

- **Real Time Detection**

Up to now, it's not clear what real-time object detection is in the literature. Usually, real-time refers to the ability of a system or solution to respond with certain time constrains. In that sense, the ideal time will vary depending on the characteristics of the application. In the case of object detection, this time constraint is given by the FPS.

2.2 Object Detection

In Computer Vision, one of the tasks that have received a lot of attention lately is Object Detection due to the success given by the use of CNNs. In that sense, multiple methods are being developed every year to tackle this particular problem. Object detection [8] consists of two main tasks: classification and localization. Thus, an object detector will try to figure out if an object belonging to a specific class is in the image and where this object is located in the image. To showcase this location, the most commonly used representation is a bounding box.

2.2.1 Applications

There are many situations in which object detection is being applied both academically or in the industry. Applications such as autonomous driving, surveillance, robotics, rescue operations, image analysis and visual aid are among the many that have object detection as a core challenge. In the following paragraphs, we will review some of these applications and the role performed by the object detector.

Autonomous Driving

For a car to have the ability to drive itself, it needs to identify different objects on the road and it must be fast. Street signs, pedestrians, cyclists and vehicles are among the things that a car needs to accurately see in real-time. Now, this is just a small part of the things the vehicle has to do. A safe and robust autonomous driving system [9] depends on perception, monitoring, decision making and control to provide the vehicle with visual information that allows for path planning, lane detection and vehicle tracking. Some requirements [10] for an object detector in this research area include: accuracy, speed, small model size and energy efficiency. Moreover, this object detector should operate on embedded processors in order to be accessible and promote wide deployment.

Video Surveillance

Video surveillance is useful to monitor security-sensitive areas such as banks, department stores, highways, crowded public places and borders [11]. Traditionally, video output is processed by human operators and saved for later use only in case of an event. An increasing number of cameras have overloaded both the operators and the storage devices with high volumes of data and is very difficult to monitor sensitive areas for long periods of times. Therefore, the use of object detection to filter out redundant information and increase response time has become necessary and requires the development of fast, reliable and robust algorithms. One specific sub field is moving object detection as it handles the segmentation of moving objects from stationary background objects. Dynamic environmental conditions such as illumination changes, shadows and waving tree branches need to be taken into account for object segmentation as it is important to distinguish objects from each other to track and analyze their actions reliably. Detecting natural phenomena such as fire and smoke could help operators take precautions in a shorter time to prevent dangerous situations.

Robotics

An important skill for robots is also object detection, where, it can be used in home, office and factory settings. Detecting classes of objects remains a challenging problem in robotics as it requires high accuracy for a large variety of object classes [12] in order to perform even simple tasks such as moving freely or taking inventory around that requires a robust detection of the surrounding objects. On the other hand, detection of moving objects is a key component in mobile robotic perception and understanding of the environment [13] which is useful for human-robot interaction and collision avoidance. Since robots usually operate in real-time with limited processing power, the computational resources available for different tasks are scarce making it an interesting problem to tackle. Robots, however, have a number of advantages like being able to view a real scene from multiple angles or viewing interesting objects up close when necessary, yet most algorithms cannot directly leverage from these scenarios and still lack in performance.

Visual Aid

Object detection can be used to develop applications to help people with severe vision impairment to independently access, understand, and explore unfamiliar environments. In an indoor setting [14], finding doors, rooms, elevators, stairs, bathrooms, and other building amenities can be challenging when there is no standard design. Consider also enabling visually impaired people to scan and find products with ease on their grocery list while at the supermarket [15]. In an outdoor setting, detecting obstacles [16], [17] in video streams could ensure the safety of the user if the proposed solution takes into account that the camera will change viewpoints constantly. Algorithms that can detect text in city scenes [18] are of great importance so that text can be enhanced or read aloud depending on the user. All these visual aid examples should be able to handle object occlusion as well to give users an accurate description of the surrounding environment.

2.2.2 Challenges

Even with the advances done in recent years, there are some shortcomings for the object detection task. Hereby, there is a list of challenges to be overcome in order to achieve more accurate and robust object detection. These were taken from [19]:

- **Scale Variance:** Object instances vary from image to image. In one picture, an instance may represent a little portion of the pixels while in another one it may take almost all the

picture. A single feature map will have problems trying to predict objects with this kind of variance.

- **Rotational Variance:** Objects could be found with an angle, inverted or even upside down. Flexible objects, like animals and people, bring in even more challenges because it is hard to define a single rotation in comparison to objects with rigid shapes, like text.
- **Domain Adaptation:** Sometimes, an object detector trained on a domain A need to be reused in a different domain B. This is done to take advantage of the training done in domain A as it may be the case that the training was done with a lot of examples and categories while training in domain B was very specific.
- **Object Localization:** Finding where an object is located is still a source of error for object detectors. The main reasons are the size of the object and more strict evaluation protocols proposed on newer datasets.
- **Occlusion:** An object might not be completely present in a frame or might be occluded by a different object. Therefore, there is missing information necessary to identify it. Occlusion is a recurrent phenomenon in real-life images and could lead to incorrect detection or no detection at all.
- **Small Objects:** Detecting medium and large-sized objects is easier than detecting small objects. The detection is affected by limited information, confusion with background, precision requirements for localization and large image size.

More factors that can impact object detection are lighting, contrast, low resolution, specularly, noise and perspective distortion.

2.2.3 Features

In computer vision, the simplest definition of feature or descriptor is any piece of information that can be extracted from an image. The main idea is to find information that is similar, or that represents a characteristic of an object. As the instances of an object can have various representations, features may account for different properties of images like:

- **Color:** Almost every image nowadays is represented with color. Therefore, it's an obvious approach to identify different objects using this characteristic. One way to this is using

color histograms [20], as it helps to get the color distribution in an image. Other descriptors are dominant color and color layout.

- **Texture:** Just like color, texture is a very useful low-level feature used to identify objects. Some attributes of texture are directionality, regularity and coarseness used for the texture browsing descriptor [20]. Other examples of features are the homogeneous texture and the local edge histogram.
- **Shape:** Features dealing with shape [21] provide a powerful clue to detect an object in an image. Humans can recognize objects solely from their shapes meaning that shapes usually carry semantic information. This is not possible using only color or texture. Some descriptors that have been considered are 3-D shape, region-based shape, contour-based shape and 2-D/3-D shape.

A good approach to follow when working with features is to combine two or more of them to then improve robustness in the object detector.

2.3 Convolutional Neural Networks for Object Detection

Convolutional Neural Networks [22] are a specialized kind of deep neural networks with a grid-like topology that are mainly used to process images. The main characteristic of the network is the use of a mathematical operation called convolution instead of the more common matrix multiplication. Yann LeCun is widely recognized as the father of CNNs, taking inspiration from the Neo-Cognitron model designed by Kuniyuki Fukushima. A convolution is a linear operation that filters information from an image by multiplying the feature map by an $N \times N$ filter where N is most commonly 3. Figure 2.1 gives an example of a convolution with a filter used to detect edges. By multiplying each element in the purple region with the elements in the filter and then adding them we obtain -14. The filter moves from left to right, and from top to bottom.

2.3.1 CNN Architecture

Figure 2.2 shows a diagram of the more basic structure of a CNN:

The most common layers in a CNN are the convolutional, the pooling and the fully connected layers:

- The convolutional layer performs a convolution in order to extract features from the input given. Each layer can have a different number of filters meant to extract different features.

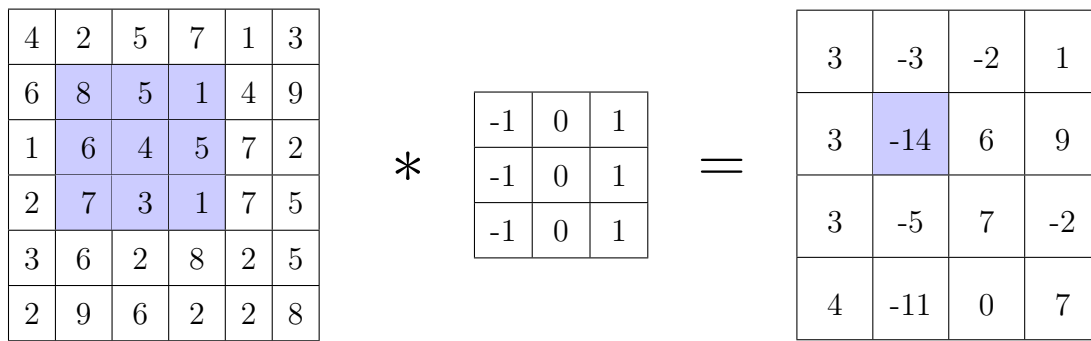


Figure 2.1: Convolution example. Source: Created by authors.

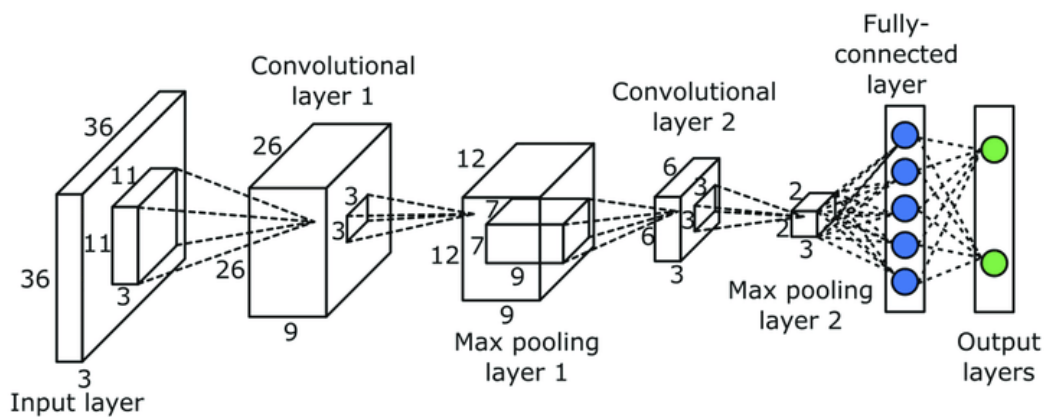


Figure 2.2: Basic CNN Architecture. Source: [23].

- The pooling layer tries to condense the information from the features by reducing the size of the feature map. The most common operations are getting the maximum or the average of a region of the map. Figure 2.3 shows an example of max-pooling in which we reduce the size of a 4x4 map to a 2x2 map by taking the maximum number from the colored regions.
- The fully connected layer is a layer with connected nodes that perform the classification of the data that was filtered on the previous layers. Some new models do not longer use a fully connected layer for classification with YOLO serving as an example. We will review this later on.

2.3.2 CNNs Detectors

The most common use for CNNs is in visual applications. Several models have been developed to tackle the object detection problem and are known as object detectors. Following there is a description of the most common models:

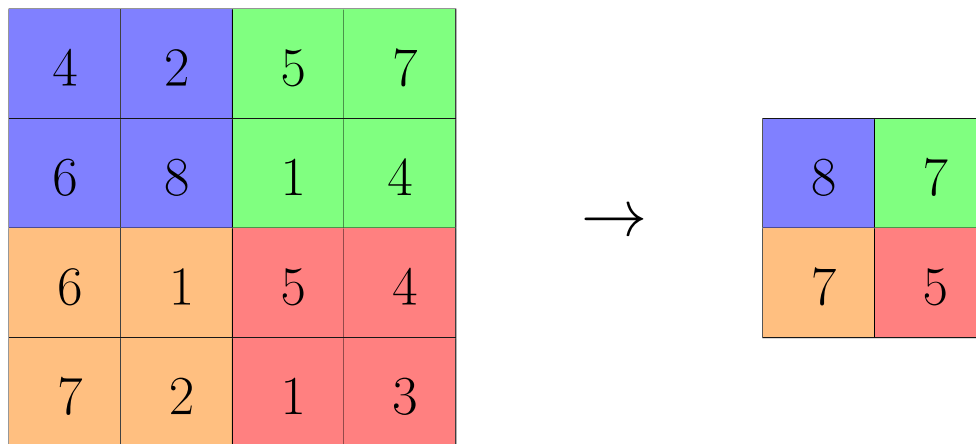


Figure 2.3: Max-pooling example. Source: Created by authors.

- **Faster R-CNN:** This method [5] is a two-stage detector where there exists a dedicated Region Proposal Network (RPN) that shares convolutional layers with a detection network. A region proposal network takes an image and gives a set of rectangular object proposals used later for detection.
- **SSD:** This method [6] is a single-stage detector that uses multi-scale convolutional bounding box outputs attached to multiple feature maps at the top of the network. This network is based on the VGG-16 network from the Visual Geometry Group [24].
- **RetinaNet:** This method [25] is a single-stage detector that borrows techniques such as anchors (RPN [25]) and feature pyramids (SSD [6] and Feature Pyramid Network (FPN) [26]) with the addition of a novel loss known as Focal Loss. This loss addresses the imbalance between foreground and background found in training single-stage detectors and therefore increases performance.

2.3.3 CNNs on Embedded Systems

Given the different applications where CNNs can be used when detecting objects, the need to implement small and efficient models have become of extreme importance in areas such as autonomous driving or augmented reality that are commonly deployed on constraint environments such as embedded system. There have been several attempts to bring the power of CNNs to embedded systems. Efforts such as MobileNets and SqueezeNet are among the top examples. Let's review them now:

- **MobileNets:** [27] It's a CNN model that uses depthwise separable convolutions. By re-

placing normal convolution by depthwise convolution and pointwise convolution the model reduces the number of parameters, therefore, obtaining a lightweight neural network. A normal convolution filters and combines features in a single step while a depthwise convolution first filters information for each channel of the input and then this information is combined by a pointwise convolution by using a 1×1 convolution filter. A depthwise separable convolution can be seen therefore as a factorization of the operations done by a normal convolution.

- **SqueezeNet:** [28] It's a model that introduced a new building block known as a Fire module. By using this new module, SqueezeNet intended to achieve three things: to change most of the 3×3 convolutional filters into 1×1 convolutional filters, to decrease the number of channels per input and to delay the use of pooling to the end of the architecture. The Fire module then is divided into two layers: a squeeze layer composed only by 1×1 filters and an expand layer that is a mix of 1×1 and 3×3 layers. The complete architecture of SqueezeNet is then formed by one common convolutional layer followed by eight Fire modules and a final convolutional layer.

There have been efforts in combining these architectures, given their efficiency and small size, with some of the architectures described in 2.3.2. Combining SSD and Faster-RCNN with MobileNets is one of several examples [27].

2.3.4 CNNs challenges for Object Detection on Embedded Systems

When working with object detection in an embedded system, we should consider the limited resources that we have at hand. Memory and processing power have an impact on the speed and precision of object detectors, and, in that sense, there exists a trade-off between these two. There have been some attempts to identify this relation as in the comprehensive study performed in [29] where the authors evaluated some state of the art detectors under the same rules. Working with videos on the other hand, brings special attention to the processing speed of the frames belonging to the videos as it needs to be done in real-time. An example in which this aspect takes great importance is autonomous driving, as decisions need to be taken in mere seconds. But here is what happens: most systems show that increasing speed usually means that precision will be affected. Therefore, recent work is focused on either improving the accuracy of fast detectors or improving the speed of accurate detectors. All this work is mostly done on GPU's powered computers. But not every industrial setup can have powerful GPUs, so for most

applications the detectors have to run on CPUs or on different low power embedded devices like Raspberry-Pi.

2.4 YOLO

YOLO [2] is a state of the art algorithm as it uses techniques that have been developed in recent years. It implements most of the current ideas on computer vision such as batch normalization, residual networks, data augmentation, among others and does a good job in terms of precision and speed. YOLO is a CNN and belongs to the single-stage family detectors, which predicts the bounding boxes and the class probabilities for those boxes at the same time. YOLO has also a very small model known as Tiny YOLO, that is intended for constrained environments with the main difference is based on the reduction of the number of layers and the number of filters used while keeping the same training parameters. This is the model that is going to be used on the Raspberry Pi since it requires less computing power. In 2.4.1 we will review the different concepts applied in YOLO and its evolution from the first version to YOLOv3. Figure 2.4 shows a comparison between YOLO and the most common object detectors. YOLO is very fast compared with the rest of methods and at the same time have a good mAP only surpassed by the combined model between FPN and Faster R-CNN in [26].

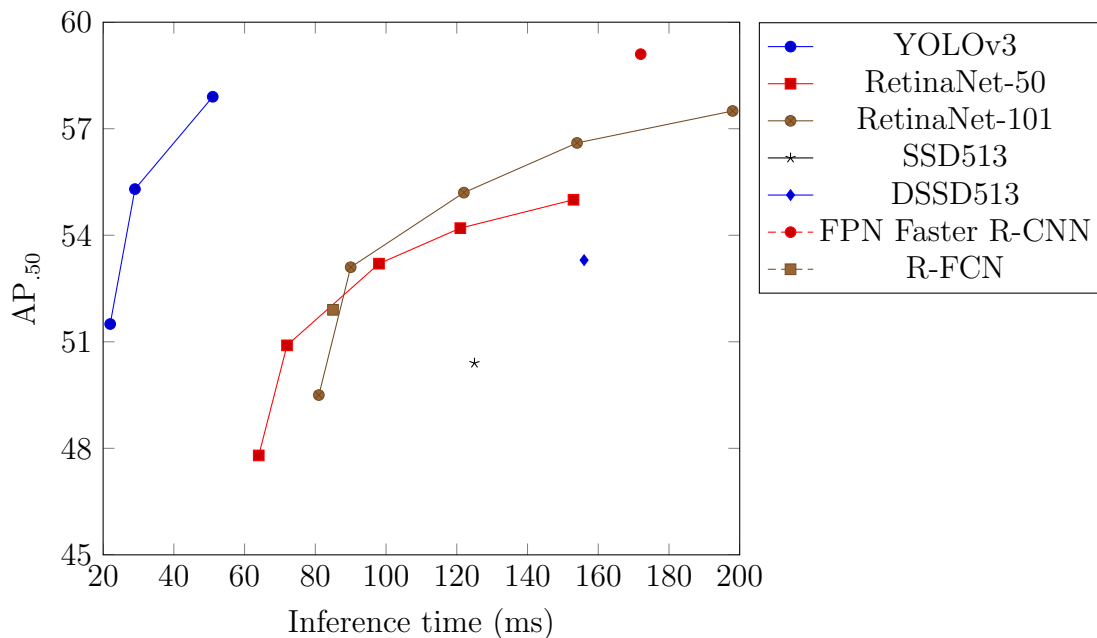


Figure 2.4: AP.50 vs Inference time of common detection methods. Source: Adapted from [4], [25], [30].

2.4.1 Description

YOLO makes use of different concepts that are useful to work with images in Computer Vision. Table 2.1 mentions most of the techniques used in the different versions. Some of these techniques were useful in the original model but were later left out as more recent ones were proved to be more helpful. The techniques will be presented according to the order of appearance in the different versions:

Versions (Year)	YOLO (2016)	YOLOv2 (2017)	YOLOv3 (2018)
Non-maximal suppression	✓	✓	✓
Contextual Reasoning	✓	✓	✓
Data Augmentation	✓	✓	✓
Dropout	✓		
Bounding box prediction	✓	✓	✓
Batch Normalization		✓	✓
High resolution classifier		✓	✓
Convolutional Prediction		✓	✓
Anchor Boxes		✓	✓
Dimension Clusters		✓	✓
Location Prediction		✓	✓
Fine-Grained Features		✓	✓
Multi-Scale Training		✓	✓
Prediction Accross Scales			✓
Residual Network			✓

Table 2.1: Techniques used in the different versions of YOLO. Source: Created by authors.

Non-maximal suppression

YOLO grid design allows for spatial diversity in the predictions done for the bounding boxes. Most of the time is quite clear to which grid an object belongs to and it is easy for the detector to make this detection. However, there are cases where large objects or objects near the border of multiple cells can be detected multiple times by more than one cell thus decreasing the accuracy of the detector. To avoid this, non-maximal suppression is commonly used to fix these multiple detections as it picks the best bounding box prediction for any specific object.

Contextual Reasoning

Context [31] is a statistical property that provides critical information of a phenomena to help solve perceptual inference tasks faster and more accurately. When YOLO performs a prediction, it “reasons” in a global way about the image. In that sense, YOLO sees the entire image during the training and testing phases so it encodes contextual information about classes as well as their appearance.

Data Augmentation

Data augmentation [32] is used to increase the amount of training data based only on information from the training data at our hands in order to avoid overfitting. In YOLO [2], random scaling and translations of up to 20% of the original image size were used. YOLO also randomly adjusted the exposure and saturation parameters of the image by up to a factor of 1.5 in the HSV color space. YOLOv2 [3] used random crops, rotations, hue, saturation when training for classification. When training for detection it used similar techniques as in the first version and in SSD [6] such as random crops and color shifting. YOLOv3 [4] also makes use of data augmentation using the same parameters as YOLOv2.

Dropout

Dropout [33] helps with overfitting by randomly omitting some of the feature detectors on training to prevent cases in which a feature detector might only be helpful in the context in which it was obtained. In the first version, YOLO used a dropout layer with a rate of 0.5 after the first connected layer to prevent complex co-adaptations between layers. In the following versions of YOLO, dropout was left behind, and batch normalization is used instead.

Bounding Box Prediction

YOLO predicts bounding boxes using a similar technique based on the MultiGrasp detection system [34] that consists on dividing an image into a grid in which each cell is in charge of detecting grasps coordinates by using a single CNN. Unlike MultiGrasp, YOLO estimates the size, location and boundaries of the object and predicts its class and it does this for multiple objects in an image. YOLO divides the input image into a 7 x 7 grid. Then, each grid cell is responsible for detecting any object when the center of that object falls within it. Also, each grid cell predicts 2 bounding boxes where each consists of 5 predictions: x, y, w, h, and

confidence. The (x, y) coordinates show where is the center of the box in relation to the grid cell, w and h represent the width and height relative to the whole image, and the confidence is defined as $Pr(Object) * IoU_{pred}^{truth}$. When there is no object in the cell, the confidence should be zero. Otherwise the confidence is equal to the Intersection over Union (IoU) between the prediction and the ground truth. Each grid cell also predicts 20 conditional class probabilities, $Pr(Class_i|Object)$, one for each of the object classes belonging to PASCAL VOC [35]. Only one set of class probabilities is predicted per grid cell, regardless of the 2 predicted boxes. At testing, the conditional class probabilities are multiplied by the individual box confidence as in equation 2.1:

$$Pr(Class_i|Object) * Pr(Object) * IoU_{pred}^{truth} = Pr(Class_i) * IoU_{pred}^{truth} \quad (2.1)$$

which gives class-specific confidence scores for each box and encode both the probability of that class appearing in the box and if the predicted box fits the object. Figure 2.5 shows how the prediction of the bounding boxes is performed.

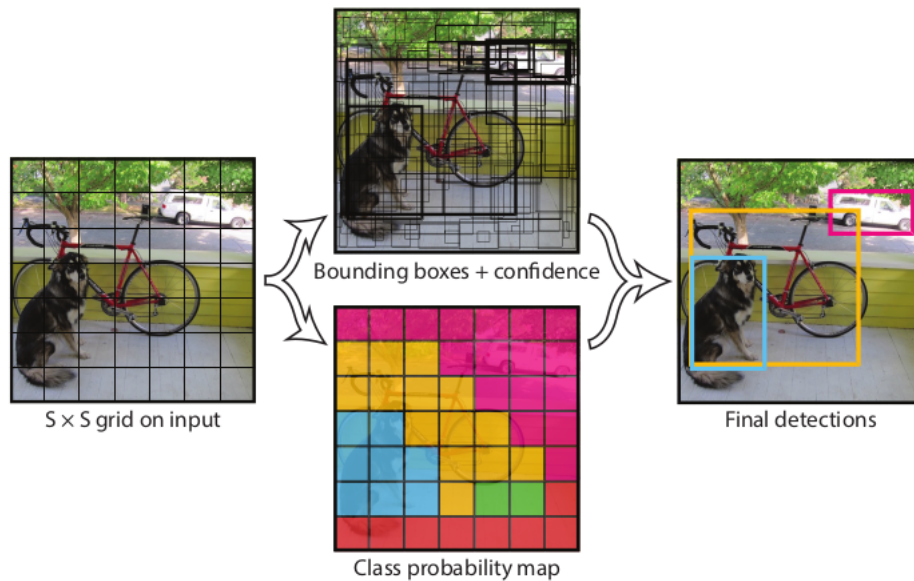


Figure 2.5: Bounding Box Prediction. Source: [2].

Batch Normalization

Batch normalization [36] leads to significant improvements in convergence while eliminating the need for other forms of regularization. This method makes normalization part of the model

architecture and performs it for each training mini-batch. Batch normalization also helps to regularize the model and allows to use much higher learning rate while worrying less about initialization. Since batch normalization acts as a regularizer, it allows to remove dropout from the model and to control overfitting at the same time.

High Resolution Classifier

The first version of YOLO trained the classification network at a resolution of 224×224 and increased to a 448×448 resolution for detection. This resulted in the network working double by switching to learning object detection and adjusting to a different input resolution. Instead, YOLOv2 [3] first fine tuned the classifier at a 448×448 resolution for 10 epochs on ImageNet and then adapted the network for detection. This change in training allows for the network filters to perform better on higher resolution images.

Convolutional Prediction With Anchor Boxes

YOLO detector predicted bounding boxes coordinates using fully connected layers but this changed in YOLOv2 [3] as a convolutional layer was used for prediction instead. In [5], instead of getting the coordinates of the bounding boxes it was easier to predict an offset by using hand-picked anchor boxes. And by using a convolutional layer, the offsets could be predicted at every location in the final feature map. Now, an anchor box is just a preset box of a certain size and weight-height ratio but predicting only offsets and not coordinates simplifies detection and learning. YOLOv2 [3] follows this example and predicts bounding boxes using anchor boxes. Originally, the input of the YOLO detector was 448×448 but it was reduced to 416×416 to have an odd number of locations for the final feature map. The rationale behind this is that there is a tendency that objects, especially large ones, occupy the center of the image and instead of having to predict an object in 4 different locations is better to predict it only at the center of an odd feature map. Taking into account that YOLO also works with input resolutions that are multiple of 32, using a resolution of 416 will at the end give a feature map of 13×13 . By using anchor boxes, YOLO faces two issues: first, the dimensions of the boxes are chosen arbitrarily and second, the model became unstable. Dimension clusters and direct location predictions were implemented in YOLO to reduce the impact of the issues just mentioned. The two techniques are going to be described next.

Dimension Clusters

Up to now, the work of the network is to adjust the anchor boxes to the correct size of the bounding box that corresponds to the object that is being detected. But, it would be better to pick more optimal dimensions in order to relax the work of the network and allow for better learning. For this purpose, YOLOv2 [3] uses k-means clustering to obtain good anchor boxes.

Direct Location Prediction

Anchor boxes make the model unstable, especially at early iterations. When a network with anchor boxes tries to predict the location of the center of the anchor box, the model takes a lot of time to stabilize the prediction of these offsets and thus presents instability. Instead, YOLO [2] predicts these coordinates relative to the location of the grid cell previously defined instead of doing it for the whole image. In the end, YOLO will predict a total of 5 bounding boxes for each grid cell in the output feature map.

Fine-Grained Features

YOLOv2 [3] make detections on a 13 x 13 feature map which usually is enough for large objects. Using fine-grained features could help instead with smaller objects. To perform this, YOLOv2 added a passthrough layer which brings features from early in the network. This layer performs two different operations. In the first case, if it is only pointed to a single layer it is just an identity mapping the feature map on that layer. In the second case, if it points to two layers then it concatenates the high-resolution features to the low-resolution features to get a new feature map. This expanded feature map is what provides the fine-grained features.

Multi-Scale Training

In YOLOv2 [3], instead of using a single input image size for training, the network was changed every few iterations. Every 10 batches the network randomly chooses a new image dimension size. Since YOLOv2 downsamples by a factor of 32, multiples of 32 were used for the image size as well, with the smallest option being 320 x 320 and the largest 608 x 608. The network was then resized according to any dimension in that range and continued training. This allows the network to predict well across a variety of input dimensions and at different resolutions. The network runs faster at smaller image sizes up to a resolution of 128 x 128 (lower resolutions do not show any detection and even 128 x 128 only works with certain images). That means

YOLOv2 is suitable to either provide speed or accuracy according to the characteristics of the input image.

Predictions Across Scales

YOLOv3 [4] predicts boxes at 3 different scales and extracts the features from those scales using a similar concept to FPN [26]. By this method, the last convolution layer makes a prediction of a tensor that holds the information of the bounding box, the probability of this box being an object and the prediction of the object category. For each scale used, 3 boxes were predicted therefore the tensor looks like this:

$$(NxN)x[3 * (4 + 1 + 80)] \quad (2.2)$$

N is the scale, 3 represent each of the 3 boxes, 4 is the information of the bounding box (x,y,width,height), 1 is the probability of being an object and 80 is the total number of object categories. Finally, YOLOv3 randomly picks 3 scales and 9 clusters: (10 x 13), (16 x 30), (33 x 23), (30 x 61), (62 x 45), (59 x 119), (116 x 90), (156 x 198), (373 x 326). The 9 clusters are evenly divided for the three scales.

Residual Network

YOLOv3 [4] takes advantage of the properties of residual networks such as the additive merging of signals both for image recognition and object detection. When deep networks are about to converge, accuracy gets saturated and then degrades if going deeper in the network [37]. This degradation is not caused by overfitting because the training error also gets worse. Therefore, to tackle the degradation of training accuracy, in [38], residual connections were proposed [39]. Although there is not definite prove of all the benefits, using residual connections greatly improves training speed. So, why are residual networks being used in YOLO?. The reason might be that the deep of the model went from 19 layers in YOLOv2 [3] to 53 layers in YOLOv3 [4] which obviously increases training time and by using residual networks this time is decreased.

2.4.2 General Architecture

As mentioned earlier, YOLO is a CNN and is mostly a combination of convolutional layers, maxpool layers and connected layers. Figure 2.6 shows the general structure of the first version of YOLO's detector. This model has changed with time as YOLO introduced the different techniques described previously.

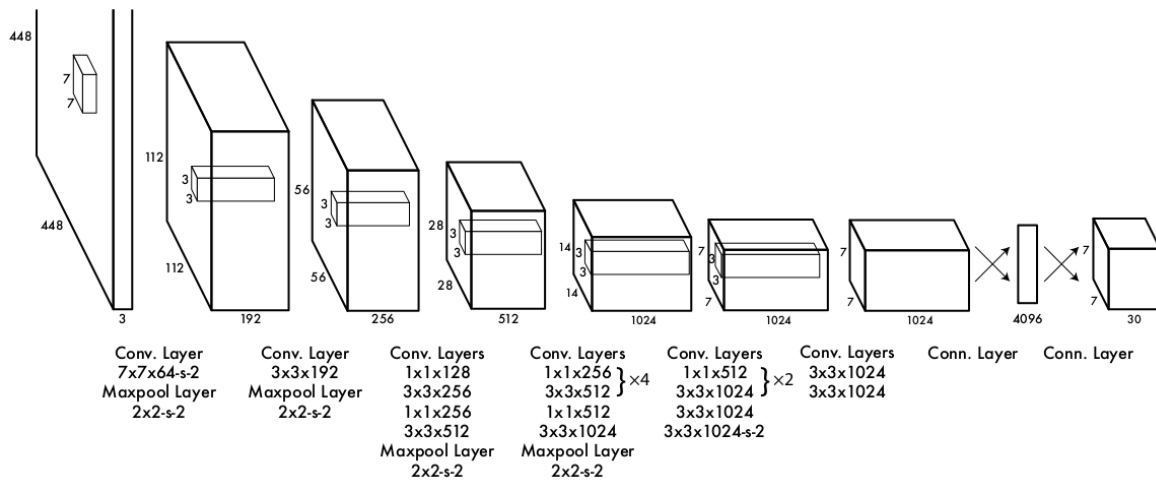


Figure 2.6: YOLO detection network. Source: [2].

2.4.3 Versions

The architectures described in this section correspond to the three versions of YOLO and encompass both the networks used for training and the ones used for testing. Both networks are similar when comparing the first layers but they differ when reaching the end of the network. The networks used for training are mostly classifier architectures while for testing and deployment these classifiers are adjusted as detectors to perform detections.

YOLO9000 [3], which was introduced alongside YOLOv2, was left out from this description as it uses the same base YOLOv2 architecture. The main two differences between YOLO9000 and YOLOv2 are that it uses only 3 clusters instead of 5 to reduce the output size and it is trained using WordTree, a hierarchical tree constructed with the synsets from WordNet used in ImageNet [40].

Classifiers

All of YOLO classifiers are a mix of convolutional and maxpool layers with varying order depending on the version and are pretrained using the 1000-class dataset from ImageNet [40]. Following is a description of each version classifier.

- **YOLO:** The original classifier architecture of YOLO [2] was composed of 20 convolutional layers and 4 maxpool layers plus one average pooling layer and one fully connected layer at the end. The input resolution used for training on this architecture is 224×224 . See Table 2.2a for more details on how the classifier is constructed.

- **YOLOv2:** YOLOv2 classifier [3] is composed of 19 convolutional layers and 5 maxpool layers thus it is known as Darknet-19. It works with the same input resolution of the first version but with a different number of starting filters. See Table 2.2b for more details.

Type (Filter) → Output		Size/Stride
Conv (64) → 112 x 112		7 x 7/2
Maxpool → 56 x 56		2 x 2/2
Conv (192) → 56 x 56		3 x 3
Maxpool → 28 x 28		2 x 2/2
Conv (128) → 28 x 28		1 x 1
Conv (256) → 28 x 28		3 x 3
Conv (256) → 28 x 28		1 x 1
Conv (512) → 28 x 28		3 x 3
Maxpool → 14 x 14		2 x 2/2
4x	Conv (256)	1 x 1
	Conv (512)	3 x 3
Conv (512) → 14 x 14		1 x 1
Conv (1024) → 14 x 14		3 x 3
Maxpool → 7 x 7		2 x 2/2
2x	Conv (512)	1 x 1
	Conv (1024)	3 x 3
Avgpool		Global
Conn (1000) → 7 x 7		1 x 1
Softmax		

(a) Original Classifier

Type (Filter) → Output		Size/Stride
Conv (32) → 224 x 224		3 x 3
Maxpool → 112 x 112		2 x 2/2
Conv (64) → 112 x 112		3 x 3
Maxpool → 56 x 56		2 x 2/2
Conv (128) → 56 x 56		3 x 3
Conv (64) → 56 x 56		1 x 1
Conv (128) → 56 x 56		3 x 3
Maxpool → 28 x 28		2 x 2/2
Conv (256) → 28 x 28		3 x 3
Conv (128) → 28 x 28		1 x 1
Conv (256) → 28 x 28		3 x 3
Maxpool → 14 x 14		2 x 2/2
2x	Conv (512)	3 x 3
	Conv (256)	1 x 1
Conv (512) → 14 x 14		1 x 1
Maxpool → 7 x 7		2 x 2/2
2x	Conv (1024)	3 x 3
	Conv (512)	1 x 1
Conv (1024) → 7 x 7		3 x 3
Conv (1000) → 7 x 7		1 x 1
Avgpool		Global
Softmax		

(b) Darknet-19

Table 2.2: YOLO and Darknet-19. Source: Adapted from [2], [3].

- **YOLOv3:** This network [4] is a hybrid approach between Darknet-19 and a residual network. The network uses successive 3 x 3 and 1 x 1 convolutional layers but now has some shortcut connections as well which come as part of the residual network implementation. The total number of convolutional layers is 53 thus is called Darknet-53 and the input resolution is 256 x 256. Although in [4] the classifier shows a connected layer at the end this is in reality a convolutional layer. See Table 2.3 for more details.

	Type (Filter) → Output	Size/Stride
	Conv (32) → 256 x 256	3 x 3
	Conv (64) → 128 x 128	3 x 3/2
1x	Conv (32)	1 x 1
	Conv (64)	3 x 3
	Res → 128 x 128	
	Conv (128) → 64 x 64	3 x 3/2
1x	Conv (64)	1 x 1
	Conv (128)	3 x 3
	Res → 64 x 64	
	Conv (256) → 32 x 32	3 x 3/2
1x	Conv (128)	1 x 1
	Conv (256)	3 x 3
	Res → 32 x 32	
	Conv (512) → 16 x 16	3 x 3/2
1x	Conv (256)	1 x 1
	Conv (512)	3 x 3
	Res → 16 x 16	
	Conv (1024) → 8 x 8	3 x 3/2
1x	Conv (512)	1 x 1
	Conv (1024)	3 x 3
	Res → 8 x 8	
	Avgpool	Global
	Conv (1000) → 8 x 8	1 x 1
	Softmax	

Table 2.3: Darknet-53. Source: Adapted from [4].

Detectors

Here we are going to review the bits that change from the classifier architecture to the detector architecture as most of the architecture remains the same for every version. We used the configuration files in the darknet github repository to get the whole picture of the detectors.

- **YOLO:** The first YOLO detection network had 4 additional convolutional layers followed by 2 fully connected layers in comparison to the classifier. Before doing this, the last three layers of the classifier were dropped which are the average-pooling layer, the fully connected layer and the softmax layer. This trend is seen on the next versions too. The input resolution of this network also changed and is 448 x 448. Table 2.4 shows both the layers of the classifier that are changed alongside the layers added for detection.

Type (Filter) → Output	Size/Stride	Type (Filter) → Output	Size/Stride
Avgpool	Global	Conv (1024) → 14 x 14	3 x 3
Conn (1000) → 7 x 7	1 x 1	Conv (1024) → 7 x 7	3 x 3/2
Softmax		Conv (1024) → 7 x 7	3 x 3
		Conv (1024) → 7 x 7	3 x 3
		Local (256) → 7 x 7	3 x 3
		Conn (12544) → 1715	

(a) Classifier

(b) Detector

Table 2.4: YOLO classifier vs YOLO detector. Source: Created by authors.

- **YOLOv2:** YOLOv2 detection network included 5 convolutional layers, left behind the connected layers used in the first version and added two new types which are route and reorg. The route layer is another word for the passthrough layer explained earlier. The first route layer only points to the layer 16 in the network so it brings a feature map of size 26 x 26 considering an input resolution of 416 x 416. The second route layer concatenate both a 13 x 13 map with a 26 x 26 map so it can be used to obtain the fine-grained features. The reorg layer instead decreases the size of the feature maps by 2 and increases the number of filters by a factor of 4. Table 2.5 shows the changes performed to the classifier in this version.

Type (Filter) → Output	Size/Stride	Type (Filter) → Output	Size/Stride
Conv (1000) → 7 x 7	1 x 1	Conv (1024) → 13 x 13	3 x 3
Avgpool	Global	Conv (1024) → 13 x 13	3 x 3
Softmax		Route	
		Conv (64) → 26 x 26	1 x 1
		Reorg → 13 x 13	/2
		Route	
		Conv (1024) → 13 x 13	3 x 3
		Conv (125) → 13 x 13	1 x 1

(a) Classifier

(b) Detector

Table 2.5: YOLOv2 classifier vs YOLOv2 detector. Source: Created by authors.

- **YOLOv3:** The last version of the detector presents instead a more marked difference from the classifier than previous versions. It adds a total of 23 new convolutional layers which are divided in three groups each handling three different feature map sizes: 19 x 19, 38 x 38 and 76 x 76 considering an input resolution of 608 x 608. The route layers are

used in the exact same way as in the second version and it no longer uses the reorg layer but instead upsamples the feature map by a factor of 2. The other outstanding difference is the addition of a layer named as yolo. This layer acts only as a logistic activation and takes care of the 9 clusters in groups of three to perform the predictions across scales. Table 2.6 shows these changes in the network.

Type (Filter) → Output	Size/Stride	
Avgpool	Global	3x
Conv (1000) → 8 x 8	1 x 1	
Softmax		
(a) Classifier		
		3x
Type (Filter) → Output	Size/Stride	
Conv (512) → 19 x 19	1 x 1	
Conv (1024) → 19 x 19	3 x 3	
Conv (255) → 19 x 19	1 x 1	
Yolo		
Route		
Conv (256) → 19 x 19	1 x 1	
Upsample → 38 x 38		
Route		
Conv (256) → 38 x 38	1 x 1	
Conv (512) → 38 x 38	3 x 3	
Conv (255) → 38 x 38	1 x 1	
Yolo		
Route		
Conv (128) → 38 x 38	1 x 1	
Upsample → 76 x 76		
Route		
Conv (128) → 76 x 76	1 x 1	
Conv (256) → 76 x 76	3 x 3	
Conv (255) → 76 x 76	1 x 1	
Yolo		
(b) Detector		

Table 2.6: YOLOv3 classifier vs YOLOv3 detector. Source: Created by authors.

2.4.4 Applications

Since 2016, YOLO has been applied in an array of visual applications. Some recent examples can be found in:

- **Transportation:** by counting the number of vehicles to help in traffic management [41] or identifying any type of objects in the road [42], [43] as vehicles, bikes, pedestrians [44], traffic signs [45], etc .

- **Pets and wild animals care:** by developing a cat recognition system based on nose features [46], underwater fish detection to measure the impact of water power solutions on the environment [47], counting the population of zebras in Nairobi National Park [48] and monitoring gorillas in the wild [49] to conduct ecological surveys.
- **Agriculture and Farming:** by identifying and counting insects for pest control measures [50], by efficiently controlling livestock such as pigs [51], measuring the growth of strawberries [52] and apples [53].
- **Satellite and Aerial Imagery:** by detecting objects in images captured by unmanned aerial vehicles (UAV) [54] and satellites [55]. Most often the images collected in this scenarios are of very high quality and consist of objects in small size representations as the pictures cover more area. Examples of objects that are being detected include cars [56], ships and airplanes.
- **Miscellaneous:** Other less common applications worth mentioning here are the detection of logos in vehicles to measure brand exposure [57] and the detection of building footprints for urban planning [58].

2.4.5 YOLO's Challenges

YOLO has suffered from different challenges since its development and each new version tries to address them with some success. Let's review some of them:

- **Detecting small objects:** In the first version of YOLO [2], given that there were spatial constraints on the prediction of the bounding boxes because each grid cell only predicted two boxes and was allowed to have only one class, YOLO couldn't detect all the small objects, specially the ones that appear in groups like for example a flock of birds. With the use of different scales for predictions on YOLOv3 [4], the network improved on detecting small objects.
- **Scale and rotational variance:** Although YOLO [2] used data augmentation to increase the amount of training data, it struggled to detect objects in new or unusual aspect ratios or settings.
- **Localization errors and low recall:** Given the loss function used for training, YOLO [2] suffered from localization errors as these were treated the same way in both small and

large bounding boxes. Even more, YOLO had relatively low recall when compared to region proposal-based methods. So, basically the improvements introduced in YOLOv2 [3] were meant to tackle these problems by using fine grained features and convolution with anchor boxes.

- **Different metrics:** To evaluate precision in object detectors the main metric used is mAP described in section 3.2.4. The thing is, different competitions have slightly different ways to measure it and therefore the results change. YOLOv3 [4] focused more on making the network stronger to improve precision by adding small improvements to the network which increased PASCAL VOC's mAP [35]. But even with these improvements, YOLOv3 suffered on COCO's mAP [8] given that it calculates precision as an average of different levels of IoU. Basically, this difference in results means that YOLOv3 struggles to align bounding box predictions with the ground truth boxes.
- **Detecting medium and large objects:** With multi-scale predictions, YOLOv3 was better in detecting small objects but affected performance on detecting medium and large objects.

The remaining challenges for object detectors and not only YOLO is to increase precision without sacrificing speed or even better to improve it.

2.4.6 YOLO on Embedded Systems

Some attempts have been made to move object detection to embedded systems but work on raspberry pi seems to be limited due to the increased difficulty of working with scarce resources due to the high computational complexity, power consumption, and store capacity that a network often requires. In this sense, [59] shows that YOLO has been implemented on both the Raspberry Pi and the Nvidia Jetson TX2 [60] for vehicle counting and classification, but a speed metric was not considered in order to determine whether the approach is suitable for real-time object detection. On the other hand, a new approach presented in [61] implemented a distributed system in which the inference task is divided between a server and the Raspberry Pi.

Chapter 3

YOLO on Raspberry Pi

This chapter presents a walk-through on the different activities performed to achieve this work main objective: to use YOLO on embedded systems. First, section 3.1 presents the basic structure of the work done to achieve this goal, going from the literature review and the different experimental phases to finish on this thesis writing. Then, section 3.2 covers the data and the architecture used for training, followed by an explanation on the metrics used to evaluate YOLO's performance on detecting objects. Finally, section 3.3 describes each of the experimental design in more detail: darknet simplification for YOLO, training parameter tuning and video performance evaluation. Darknet simplification for YOLO involves the work done to simplify and organize the framework to only use YOLO, training parameter tuning describes the different combinations for parameters such as learning rate, steps, batch size and subdivision that were considered to improve YOLO's precision, and video performance evaluation deals with finding a good configuration to work with videos on the Raspberry Pi.

3.1 General Thesis Structure

Diagram 3.1 shows the main phases of this work which are: literature review, experimental design and thesis writing. Even more, experimental design is split into three more phases, each containing different experiments carried out with the objective of simplifying, improving and evaluating YOLO:

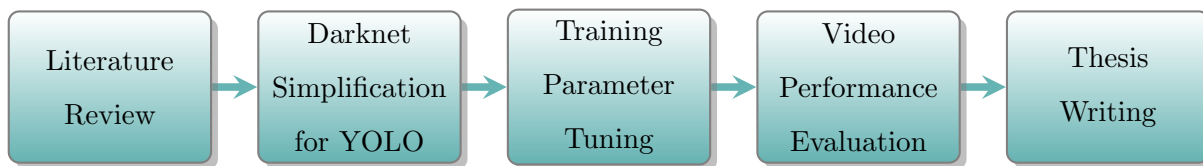


Diagram 3.1: Thesis Process. Source: Created by authors.

3.1.1 Literature Review

The main purpose of this activity was to collect information and ideas related to object detection, CNNs, YOLO and the Raspberry Pi. This activity also provided information on how YOLO performed object detection and all of the techniques that make YOLO a state of the art detector taking into account the different changes that have been implemented up to now. It provided as well information on the metrics currently used to verify the performance of object detectors in terms of accuracy and speed of detection. Finally, it helped figure out how to perform the training and testing in YOLO and things to be considered to obtain better results.

3.1.2 Experimental Design

The experimental part of this work was divided into three phases considering activities that helped obtain a better understanding of object detection and the use of YOLO in embedded systems. In general, the tasks performed in each phase are enumerated next:

- Darknet Simplification for YOLO:
 1. To simplify the code found on Darknet and leave only the parts that correspond to YOLO.
 2. To test that the above changes did not affect YOLO in any way by training with the same parameters as the original.
 3. To run the network on the Raspberry PI.
- Training Parameter Tuning:
 1. To identify training parameters that are likely to improve the overall accuracy of the network.
 2. To train the network with these parameters.
 3. To evaluate network performance and compare it to the simplified version.

- Video Performance Evaluation:
 1. To compare the performance of the weights obtained in the previous phase using videos both in a computer and in the Raspberry Pi.

Each stage will be further explained in the following sections.

3.1.3 Thesis Writing

Thesis writing consisted on organizing the information found on the literature review with the aim on making it easy to read and follow. Certainly, there are ideas that are more complex to explain than others and that can be spotted on certain passages of this work. It also consisted in planning and writing the general structure of the whole text: introduction, state of the art, methodology, results and conclusion. Without a doubt, the parts that required the most work were the state of the art and the results given that the first provides the foundations and the second shows the achievements.

3.2 Materials and Methods

The materials and methods section provides with information necessary to later understand the experimental design. First, it covers the datasets used to pretrain and train YOLO. Later, it shows the architecture used for both training and performance evaluation followed by instructions on how to perform this training. Finally, a description of the metrics used for evaluation is presented.

3.2.1 Dataset

Now, let's discuss the data used for training. Most of the current state of the art detectors pretrain their networks using the ImageNet dataset which is also the case for YOLO [4]. After this initial training, YOLO can be further trained using the COCO dataset. Next is a description for both datasets:

ImageNet

ImageNet [40] is an image dataset constructed with the WordNet hierarchy in mind. WordNet is a lexical database of English where nouns, verbs, adjectives and adverbs are put together into sets of cognitive synonyms known as synsets. Currently, only nouns are being taken into

account in ImageNet, with hundreds and even thousands of images belonging to each node in the hierarchy. Table 3.1 shows the high-level categories of ImageNet where each one has more than 50 synsets and well over 50000 images. In total ImageNet has 14197122 images and 21841 synsets indexed. For the ImageNet competition, there is a subset of 1000 synsets which is the one used to pretrain YOLO. Some examples of synsets are: kit fox, siberian husky, grey whale, accordion, revolver, laptop, strawberry, ballon, canoe, pirate, etc. It's important to note too that ImageNet does not own the copyright of the images and only provides thumbnails and urls of the images like any other search engine. The images are mostly collected and validated from the following search engines: Yahoo, Live Search, Picsearch, Flickr and Google [62].

amphibian	animal	appliance
bird	covering	device
fabric	fish	flower
food	fruit	fungus
furniture	geological formation	invertebrate
mammal	musical instrument	plant
reptile	sport	structure
tool	tree	utensil
vegetable	vehicle	person

Table 3.1: ImageNet high-level categories. Source: Created by authors.

Common Objects in Context

COCO [8] is an image dataset used for large-scale object detection, segmentation, and captioning. COCO contains about 350000 images from which more than 200000 are labeled. It also contains 80 objects categories that are listed in Table 3.2. Objects in COCO are currently annotated with a segmentation mask but information on bounding boxes is also available for researchers. In another point, most images in COCO are non-iconic, meaning that there are more than a single large object present in the image. Similar to the case of ImageNet, COCO does not own the copyright of the images which belong to Flickr, an online application for photo management and sharing. Also, the images do not have a standard size. YOLOv3 was trained with the 2014 train/valid dataset that is composed of 117264 images.

In addition to the training set, COCO provides three more sets for testing related to the competition:

- Test-Val: This set has around 5000 images and is used to verify that a submission to the

person	bicycle	car	motorbike	aeroplane
bus	train	truck	boat	traffic light
fire hydrant	stop sign	parking meter	bench	bird
cat	dog	horse	sheep	cow
elephant	bear	zebra	giraffe	backpack
umbrella	handbag	tie	suitcase	frisbee
skis	snowboard	sports ball	kite	baseball bat
baseball glove	skateboard	surfboard	tennis racket	bottle
wine glass	cup	fork	knife	spoon
bowl	banana	apple	sandwich	orange
broccoli	carrot	hot dog	pizza	donut
cake	chair	sofa	pottedplant	bed
diningtable	toilet	tvmonitor	laptop	mouse
remote	keyboard	cell phone	microwave	oven
toaster	sink	refrigerator	book	clock
vase	scissors	teddy bear	hair drier	toothbrush

Table 3.2: COCO 80 object categories. Source: Created by authors.

evaluation server is being done correctly.

- Test-Dev: This set has around 20000 images and is used for testing in general circumstances. It's recommended that the results for publications must be obtained using this set.
- Test-Challenge: This set has around 20000 images and is used to test the results for the COCO challenge.

3.2.2 Architecture of Tiny YOLOv3

As the raspberry pi is a device that is constrained both in terms of computational power and memory, we considered the use of the smaller version of YOLO for this work. The main characteristic of this network is that although it sacrifices some precision it is really fast allowing for detection in constrained environments. Tiny YOLOv3 is remarkably different from the full version. First, the number of convolutional layers is reduced from 75 to 13 which decreases the size of the network considerably. It also has maxpool layers which are not present on the full version and it looks like that the residual layers and the shortcut connections are not being used in this model. Another modification, is that this version works with only 6 clusters instead of 9 which are: 10x14, 23x27, 37x58, 81x82, 135x169 and 344x319 and an input resolution of

416x416. For more details check Table 3.3. As an additional note, here there is no distinction between the network used in training and the network used for detection in contrast to the full YOLO versions that use slightly different architectures for each case.

Type	Filters	Size/Stride	Activation	Output
Convolutional	16	3x3	Leaky	416x416
Maxpool		2x2/2		208x208
Convolutional	32	3x3	Leaky	208x208
Maxpool		2x2/2		104x104
Convolutional	64	3x3	Leaky	104x104
Maxpool		2x2/2		52x52
Convolutional	128	3x3	Leaky	52x52
Maxpool		2x2/2		26x26
Convolutional	256	3x3	Leaky	26x26
Maxpool		2x2/2		13x13
Convolutional	512	3x3	Leaky	13x13
Maxpool		2x2		13x13
Convolutional	1024	3x3	Leaky	13x13
Convolutional	256	1x1	Leaky	13x13
Convolutional	512	3x3	Leaky	13x13
Convolutional	255	1x1	Linear	13x13
Yolo				
Route				
Convolutional	128	1x1	Leaky	13x13
Upsample				26x26
Route				
Convolutional	256	3x3	Leaky	26x26
Convolutional	255	1x1	Linear	26x26
Yolo				

Table 3.3: YOLOv3-Tiny Architecture. Source: Created by authors.

3.2.3 Training Parameters

YOLO's training is done following rules from a configuration file that encompasses the training parameters and the structure of the architecture, For this work, we only considered the following parameters:

- **Learning Rate (LR):** The learning rate is the parameter that controls the speed in which the network is going to learn to detect objects. YOLOv3-tiny uses a standard training

rate of 0.001.

- **Step (S):** The step is a parameter that modifies the LR during training by decreasing it by a given factor. The S specify in which iterations the learning will be decreased. Full training of YOLOv3-tiny consists of 500200 iterations in which the S are enforced at 400000 and 450000 with a factor of 10.
- **Batch Size (B):** Batch Size determines the number of images that will be processed by the network before updating the weights which is also the number of images per iteration in YOLO. The standard B in YOLOv3-tiny is 64 and by decreasing or increasing this number the network might need more/less number of iterations to obtain a higher mAP. For the second part of training in this section, from each LR we selected the weights that gave the highest mAP and trained each with a B of 16 and 128.
- **Subdivision (Subd):** Subdivision is closely related to the Batch Size. This parameter determines the number of images that are processed in parallel which have the potential of decreasing training time. It is also useful in cases in which there is not enough memory for training. Consider a B of 64 and a Subd of 2. This configuration establishes that the batch is going to be divided by 2 and therefore 32 images will be processed at the same time.

More time would be needed if we were to try combinations of all the parameters that are used in training YOLO.

3.2.4 Metrics

The detections done by the trained network described in 3.2.2 and with the different combination of parameters described in 3.2.3 will be evaluated with the following metrics: IoU, precision, recall, accuracy, the precision-recall curve, mAP and FPS. Let's start checking IoU as it is the starting point to get the remaining metrics except for the FPS.

Intersection over Union

The IoU is a metric that determines how well a predicted bounding box (B_p) encloses a ground truth bounding box (B_{gt}) considering the following ratio:

$$IoU = \frac{area(B_p \cap B_{gt})}{area(B_p \cup B_{gt})} \quad (3.1)$$

In short, the formula is comparing the area that is intersected by both boxes over the area of the union. Most competitions accept an IoU of 0.50 to consider that an object has been correctly detected. And in that sense, the IoU will be used to determine if the detections fall into one of the following categories:

- **True Positive (TP):** An object in an image is correctly detected.
- **True Negative (TN):** This one is not considered for object detection as it would mean that there is not an object present in the image thus is not useful whatsoever.
- **False Positive (FP):** An object in an image is not detected given that the detection does not fulfill the IoU threshold.
- **False Negative (FN):** An object in an image is incorrectly detected.

Figure 3.1 shows an example with an image taken from the validation set. In 3.1a, the ground truth object is a cat and the detection is correct as it is clear that the IoU is higher than 0.5, therefore, is considered a TP. In 3.1b, although the object is well classified as a cat, the IoU is lower than 0.5 therefore is an example of a FP. Finally, in 3.1c, although the IoU is higher than 0.5, the object is wrongly classified as a dog therefore is an example of a FN.

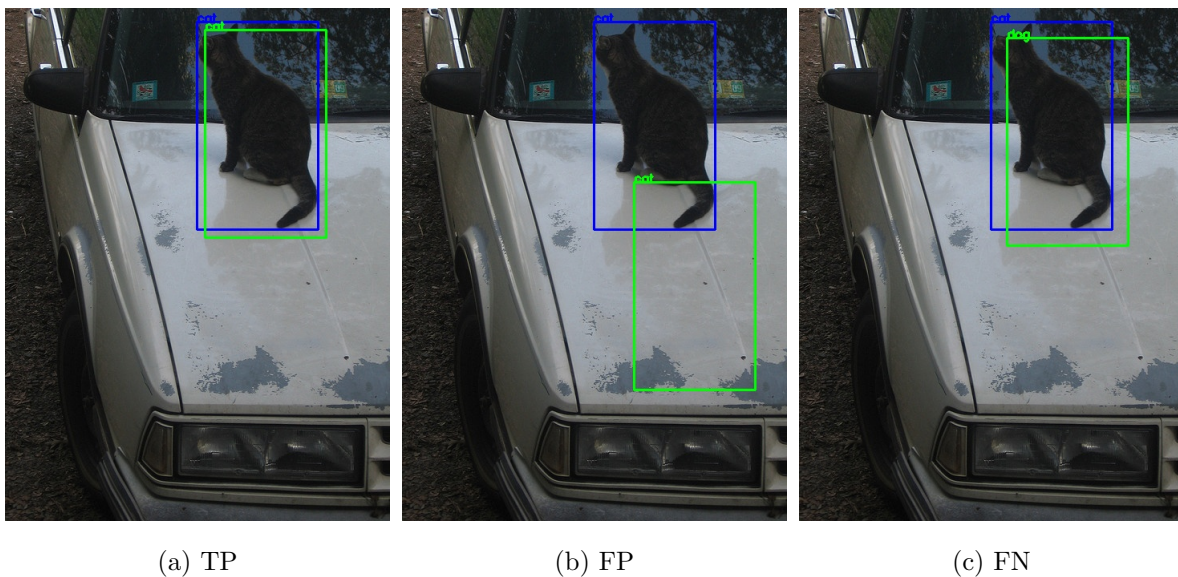


Figure 3.1: Example of types of detections. Source of image used in example: [63].

Precision-Recall Curve and Area under the Curve

Now that IoU is a little bit more clear, it is time to see the precision-recall curve. This curve is necessary to get the mAP as the mAP is not other than the area under it. Let's review first the concept of precision and recall independently:

- **Precision:** also known as the positive predicted value, measures how many correct detections were made from the total number of detections made by the detector.

$$\frac{TP}{TP + FP} \quad (3.2)$$

- **Recall:** also known as the true positive rate or sensitivity, measures the number of correct detections made over the total number of ground truth objects.

$$\frac{TP}{TP + FN} \quad (3.3)$$

Once the precision and recall are obtained for each category, the precision-recall curve can be plotted. In the x axis we found the recall while in the y axis we found the precision. To plot this curve, we take all the detections of a single category and list them in descending order with respect to the class probability given by the detector. As the first element have the highest probability, then there is a high chance that this could be a TP (this is not always the case as we need to also consider the IoU) thus the precision, in this case, would be 1, while the recall would be smaller as it is computed against the number of ground truth objects. As we go through all the detections made, the precision will decrease while the recall will increase.

mean Average Precision

The mAP is a metric that measures the precision of object detectors and is quite popular. Different competitions have different ways to calculate this metric and for this work we only considered the AP₅₀ as the way to measure precision, where AP represents the Average Precision. Now, COCO has 6 different metrics for mAP [64] which are:

- **AP:** AP with 10 different IoU thresholds that go from 0.5 to 0.95 in intervals of 0.05. This is the metric used in the primary COCO challenge.
- **AP₅₀:** AP with an IoU threshold of 0.5. This was the AP traditionally computed before the introduction of the average AP by COCO.

- **AP₇₅**: AP with an IoU threshold of 0.75
- **AP_{small}**: AP for objects with an area less than 32^2
- **AP_{medium}**: AP for objects with an area between 32^2 and 96^2
- **AP_{large}**: AP for objects with an area greater than 96^2

The mAP results, found in the different publications that measure precision with the COCO metric as standard, were obtained using the COCO test-dev dataset [65] which can only be evaluated on COCO evaluation servers [66]. COCO also provides an API ¹ to locally evaluate the results from any object detector and obtain the mAP and some other metrics related to recall. For our case, we used this API and obtained the mAP using the test-val set. Furthermore, some modifications were made to the API in order to obtain the values for TP, FP, FN, and for precision and recall (needed to plot the Precision-Recall curve). From now on, only AP₅₀ would be used to talk about mAP as this was the one used for comparisons. To use COCO's API the results file must have the following structure:

```
[{
  "image_id"      : int,
  "category_id"   : int,
  "bbox"          : [x,y,width, height],
  "score"         : float,
}]
```

Where x, y give the location of the top left position of the bounding box. Lucky for us, darknet provides the following command that gives the results in the format required by COCO:

```
$ ./darknet detector valid cfg/coco.data cfg/yolov3-tiny.cfg
  ↪ yolov3-tiny.weights
```

As mentioned before, the COCO's API was modified to obtain the remaining metrics. To obtain the number of TP, FP and FN we print the following variables:

```
if len(dtm[0])==0:
    print(f'Category: {k}, TD: 0, TP: 0, FP: 0, GT: {npig}')
```

```
else:
```

¹<https://github.com/cocodataset/cocoapi>

```
print(f'Category: {k}, TD: {len(dtm[0])}, TP: {tp_sum}
      ↪ [0][-1]}, FP: {fp_sum[0][-1]}, GT: {npig}')
```

Where dtm is number of total detections, tp_sum is the number of TP, fp_sum is the number of FP and npig is the number of ground truth.

To obtain the precision and recall this was the bit of code written:

```
def save(self, file):
    '''
        Save precision-recall data per class.
    '''
    f=open(file,"w+")
    for k, (i,j) in enumerate(zip(self.eval['precision'],self
    ↪ .eval['recallgraph'])):
        for n in range(0,len(i[0,:,0,0])):
            f.writelines("{0} {1}\n".format("NaN", "NaN")
            ↪ if round(x,3) == 0 and round(y,3) == 0
            ↪ else "{0} {1}\n".format(round(x,3),
            ↪ round(y,3)) for x,y in zip(i[:,n,0,0],j
            ↪[:,n,0,0]))
```

Where the self.eval dictionary contained the data corresponding to precision and recall in both the y and x axis to be then saved to a file in disk.

Accuracy

Given that the number of TP, FP and FN were obtained, the accuracy metric was also considered. This metric measures the number of correct detections over the total number of detections made regardless of this being right or wrong.

$$\frac{TP + TN}{TP + TN + FP + FN} \quad (3.4)$$

Frames per Second

FPS determines the number of frames that are processed by an object detector. A high FPS is better for having real-time object detection. Now, this metric is dependant on the system used for detection meaning that running the detector on a GPU will be faster than running on

the CPU and way more than in the Raspberry Pi. Darknet provides the following command to process video:

```
$ ./darknet detector demo cfg/coco.data cfg/yolov3-tiny.cfg  
↪ yolov3-tiny.weights movie.mp4
```

which gives an amount for FPS as a new detection is made. In order to present a result for FPS, we got an average of all the values for FPS obtained while processing of the video.

3.2.5 Training

To perform the training we used 4 GPUs NVIDIA Tesla K80 that are part of the hardware infrastructure of the supercomputador QUINDE 1 [67] and ran the following script:

```
#!/bin/bash  
  
#BSUB -e err.log  
#BSUB -o out.log  
#BSUB -cwd /home/paulsilvap/orig  
#BSUB -J orig.job  
#BSUB -q normal  
#BSUB -n 4  
  
module load cuda/8.0.61  
module load cudnn/8.0.0  
  
cd /home/paulsilvap/orig  
  
./../darknet/darknet detector train coco.data yolov3-tiny.cfg ../  
↪ yolov3-tiny.conv.13 -gpus 0,1,2,3
```

The important bit here is the line that calls darknet. Let's break it down:

- **detector:** it is the file that contain all the instructions that can be done in darknet in relation to YOLO. We already used valid and demo in some of the commands shown earlier.

- **train:** it is the instruction that performs the training of YOLO with the information found in the .data and .cfg files.
- **coco.data:** it is the file that points to the files containing the location of the images in the train/val and test-val datasets.
- **yolov3-tiny.cfg:** it is the file that contains all the information of the parameters and the architecture for training/detection.
- **yolov3-tiny.conv.13:** it is the weight file used for training. These weights were obtained from the yolov3-tiny.weights that correspond to Tiny YOLO and can be downloaded from [2](#)
- **gpus:** it is an option that indicates how many GPUs are going to be used for training.

For the training to work, we also need to download the images belonging to the dataset. Darknet's author provides a script that does all the job. The instructions can be found in [3](#).

3.2.6 Software and Hardware

The code of the simplified version is available in a github repository⁴. It maintains the base structure of the original version of darknet and keeps C as the main coding language. The framework uses the following libraries and APIs:

- **OpenCV:** The Open Source Computer Vision Library [68] (OpenCV) is an open source library that provides implementations of algorithms focused on computer vision and machine learning applications. It supports Windows, Linux, Android and Mac OS.
- **OpenMP:** OpenMP [69] is an API that facilitates parallel programming in languages such as C/C++ and Fortrand. OpenMP is also multiplatform enabling applications from desktop computers to supercomputers.
- **CUDA:** CUDA [70] is an API that enables parallel computing and programming on NVIDIA GPUs.

²<https://pjreddie.com/darknet/yolo/>

³<https://pjreddie.com/darknet/yolo/>

⁴<https://github.com/paulsilvap/darknet/>

- **Cudnn:** The NVIDIA CUDA Deep Neural Network [71] (cuDNN) provides deep neural networks primitives to be used in GPUs. These primitives include forward and backward convolution, pooling, normalization, and activation layers.

From this, only opencv and openmp are available in the Raspberry Pi. CUDA and cuDNN require a NVIDIA GPU powered device to be able to compile.

Next, Table 3.4 shows the characteristics of the hardware used to obtain the different metrics and to perform the testing for the video performance evaluation.

	Computer	Raspberry Pi
Processor	Core i7 7th Gen 2.8 GHz	ARM Cortex-A53 Quad Core 1.2 GHz
RAM	16GB	1GB
Architecture	64 bits	64 bits
GPU	GTX 1050 Ti	-
Drive	HDD	SD
Drive Size	1 TB	128 GB

Table 3.4: Hardware for testing. Source: Created by authors.

3.2.7 Deployment

To use darknet to perform detection from scratch, we need to follow the subsequent steps:

1. Download darknet from the github repository.
2. Download the weight file or files.
3. Compile darknet.
4. Run detector with the following command:

```
$ ./darknet detect cfg/yolov3-tiny.cfg yolov3-tiny.weights
↪ image.jpg
```

If we were to use darknet with videos, first we need to install OpenCV and compile darknet again with the correct flag.

3.3 Experimental Design

In the experimental design section, it is described the different experiments performed to obtain the metrics explained in 3.3.2. The three phases are darknet simplification, parameter tuning and performance evaluation.

3.3.1 Darknet Simplification for YOLO

In order to take advantage of all the properties of YOLO, a revision of the code found in [72] was necessary. In general, darknet is a nice framework to train different models of artificial neural networks but for our case, we only needed to work with YOLO. For this reason, the parts that did not belong to the general implementation of the three versions of YOLO were eliminated from the original codebase. Reducing code complexity and improving code organization were among the main tasks for this stage. The other consideration taken was to check if the code was able to run on the Raspberry Pi and modify it in the case the code did not run in this device. After performing the previous action, the next step was to check that the modifications did not affect the general performance of YOLO. For that, we performed a full training on both the original codebase and the simplified version without modifying any training parameter.

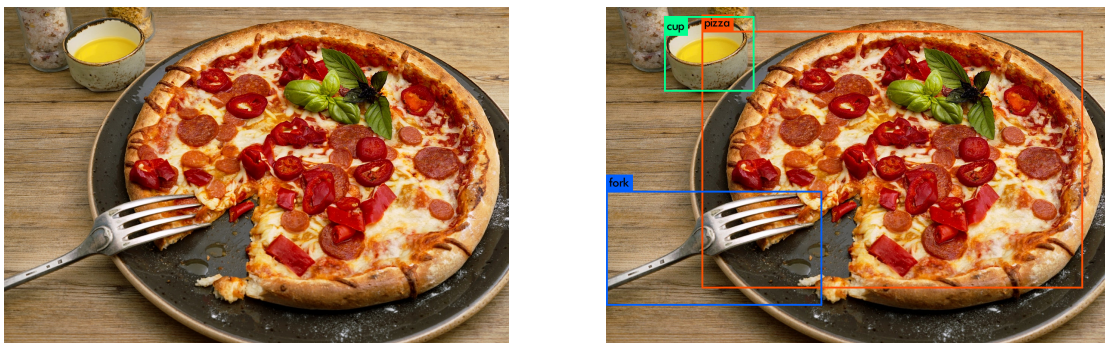


Figure 3.2: Input image vs Output Detection. Source of image used in example: [73]

Figure 3.2 shows, as a way of example, an image that was passed to YOLO as input and then the obtained output. The detection consists of the bounding boxes and the object categories from where the objects in the image are from. This result should be very similar if not the same for both versions. The metrics that are going to be evaluated in this phase are the precision-recall curve, the accuracy, recall and AP_{50} .

3.3.2 Training Parameter Tuning

Once it was confirmed that the changes done to the code did not affect YOLO's the precision negatively, the next step was to find ways to improve it. The approach taken consisted in making combinations of the training parameters explained in 3.2.3. LR and S were used in the first set of tests while B and Subd were used to obtain the final training candidate. Table 3.5 shows a summary of the tests performed.

LR	S		
0.1	-		
0.1	40000, 80000		
0.005	-	B	Subd
0.005	40000, 80000	16	2
0.002	-	16	8
0.002	40000, 80000	128	4
0.0005	-	128	64
0.0005	40000, 80000	(b) Second Set	
0.0005	-		
0.0005	40000, 80000		

(a) First Set

Table 3.5: **Training summary for Parameter Tuning.** 10 experiments for the first set of training. One (successful) candidate per LR is kept and later each is trained with the four combinations of the second set. Source: Created by authors.

- For the first set, we considered the following rates: 0.1, 0.005, 0.002, 0.0005, 0.0001. Initially, we started training only with 0.1, 0.005 and 0.0001 to evaluate how drastic would be the change in learning. Then, we considered to use 0.002 and 0.0005 which are an increase/decrease by a factor of 2 of the standard training rate. As initially we only trained for 100000 iterations, we considered two scenarios for the 5 LR we chose: the first one did not consider S and the second one enforced steps at the 40000 and 80000 iterations. By multiplying the 2 scenarios with the 5 LR we ended up with a set of 10 tests. From each pair of tests (same LR), a candidate will be selected to be used next.
- For the second set, we considered two B: 16 and 128. For 16 a Subd of 2 and 8 was selected while for 128 a Subd of 4 and 64 was used instead. A Subd smaller than 4 required more than 12 GB of ram memory and therefore was not feasible given the memory available.

The B and Subd will be tested for each selected candidate from the first set. At the end of this set, only one candidate is selected.

- Once a final candidate was chosen, this was trained for 100000 and 200000 iterations with S values that account for the 80 and 90 percent of the total number of iterations. For 100000, a S of 80000 and 90000 were used while for 200000 the S was at 160000 and 180000 respectively.

To select the candidates for each set, we used $AP_{.50}$ as the main criteria. The remaining metrics were also calculated except for the precision-recall curve and the FPS.

3.3.3 Video Performance Evaluation

The idea of improving the mAP of YOLO is for using the detector on real-time for videos. Obtaining a better mAP could allow working with lower input size resolutions and therefore decrease detection time which is what we need to work on real-time on the Raspberry Pi. Given the results from subsection 3.3.2, it was time to prove if the previous affirmation held true also in the case of using videos as input.

Video Testing

For evaluating the new weight file obtained from 3.3.2, first we needed to choose videos that contain some of the objects from the 80 categories in COCO. In that sense, we took two videos from Pexels [74], a portal that offers free videos, with the characteristics that can be seen in Table 3.6. The object categories considered this time were person and car.

	Video 1	Video 2
Resolution	1920px x 1080px	1920px x 1080px
Duration	0:15	0:34
FPS	25	29
Aspect Ratio	16:9	16:9
Format	.mp4	.mp4
Codec	H.264 - MPEG-4 AVC	H.264 - MPEG-4 AVC

Table 3.6: Video characteristics. Source: Created by authors.

With this in mind, we ran two tests that took into account the following considerations:

- **Video Resolution:** This is the resolution of the video that enters the network. We considered three different scenarios: the original resolution of the video (1920x1080), half

of this resolution (960x540) and the lowest resolution in which all the object categories could be detected and YOLO could still draw bounding boxes (304x171). The video resolution change is performed using the ffmpeg command in linux. FFmpeg [75] is a very fast multimedia framework, able to change between random sample rates and resize video using a high quality filter.

- **Input Resolution (IR):** This is the input resolution of the network. YOLOv3-tiny uses a standard input resolution of 416x416. We considered three different scenarios: the standard input resolution(416x416), a resolution that is a multiple of 32 and is near the half of the standard input resolution (224x224, being 224 equal to 7x32, 416 is thirteen times 32) and the lowest input resolution in which the detector could still detect all the object categories considered ($192 \times 192 = 6 \times 32$).

We calculated and compared the FPS and the total number of detections (TP + FP) for all the 9 scenarios (3 video resolutions by 3 input resolutions) described above with the aim of finding the best configuration to have object detection on the Raspberry Pi.

Chapter 4

Results

In this chapter, we will show the results from the different experiments described on chapter 3. First, we will present general results of training YOLOv3-tiny with the original darknet code vs a simplified version of the code. Then, we will discuss the results from training the simplified version with the combination of parameters and values described on 3.3.2. Finally, we will evaluate the performance in videos of the winning combination. The different comparisons in each of the following sections consider the metrics discussed in section 3.2.

4.1 Original YOLO vs Simplified YOLO

Figure 4.1 shows the precision-recall curves of the original and the simplified version detections on COCO's validation dataset with an IoU = 0.5. It shows only the curves for the first 10 object categories of COCO. Although the behaviour of the curves does not change dramatically between each version, the original version performed better on the test-val dataset. One example of similar behaviour can be seen on the person object category in which the curves are almost identical and really smooth (blue lines) while the rest have some degree of variance with the most notable case for the train object category (green line). From the categories seen in Figure 4.1, bus and train were better detected than boat and traffic light that did not even reach a recall higher than 0.5 meaning that the number of TP was less than half of the total ground truths. To determine the general performance achieved, Figure 4.2 illustrates the $AP_{.50}$ obtained from training the original and the simplified version a total number of 500000 iterations. The evolution of the $AP_{.50}$ in both cases follows a similar trend, a point that reinforces that the modifications done on the original version, in order to simplify the code, have been performed

appropriately so far. Evaluating the behaviour of the curves, it can be seen that the $AP_{.50}$ increases rapidly in the beginning up to the iteration 100000. Then, the $AP_{.50}$ maintains a semi steady growth for the next 300000 iterations. Once it reached the iteration 400000, the $AP_{.50}$ jumps almost 0.05 points due to the change on the LR from 0.001 to 0.0001. After that, the $AP_{.50}$ changes very little given the fact that the LR is quite small and does not have a huge impact in the training anymore. At this point, the LR is equal to 0.00001.

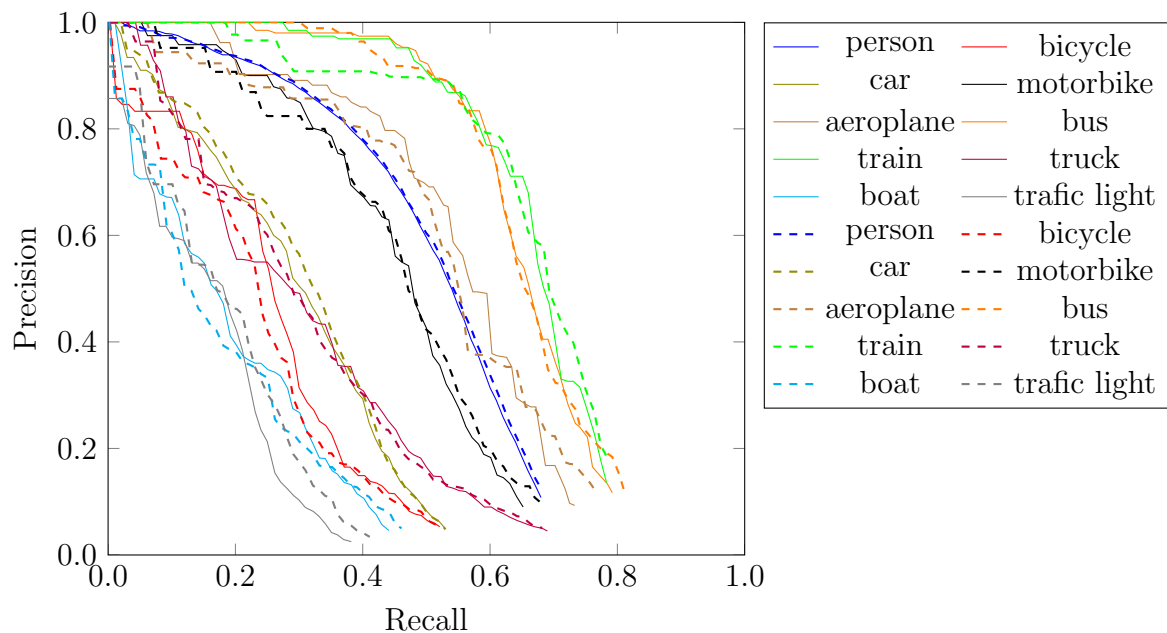


Figure 4.1: **Precision-Recall Curves for 10 first categories.** This graph shows the curves for the original version (continuous) and the simplified version (dashed). Source: Created by authors.

Table 4.1 shows the 6 metrics for $AP_{.50}$ proposed by COCO which gives a general idea of the precision for the four versions evaluated. The results from evaluating YOLOv3 with both the test-dev set and the test-val set are very similar, considering that a change smaller than one point is not yet significant enough. As seen here, YOLOv3 suffers more when detecting small objects than detecting large ones, a trend that gets worse for YOLOv3-tiny. Obtaining similar results for both datasets is a good sign that the test-val set is good enough to evaluate the $AP_{.50}$ for YOLOv3-tiny and the set of tests in Table 3.5. The difference in $AP_{.50}$ is so small for the original and the simplified version that is not even greater than 0.1.

Table 4.2 presents even more evidence to verify that the changes were done appropriately. In

¹Taken from [4]

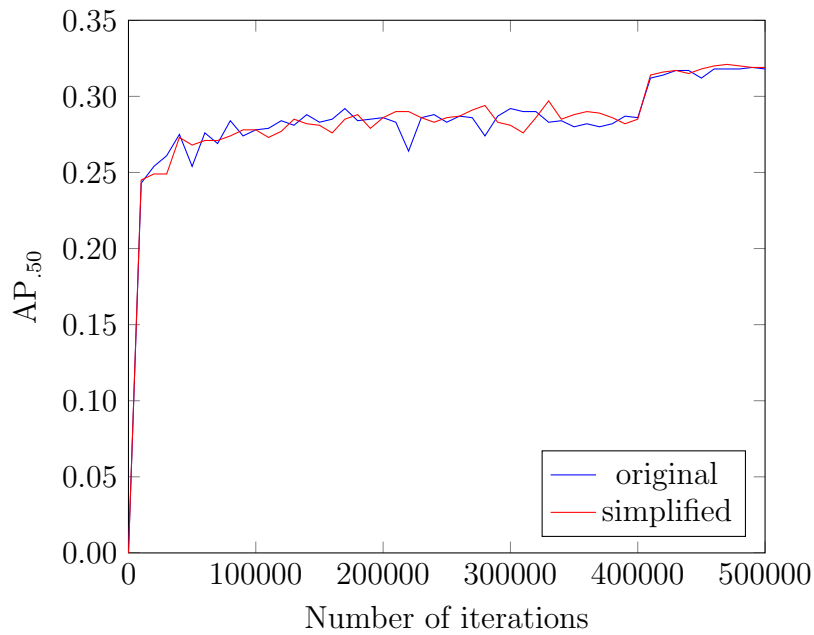


Figure 4.2: AP_{.50} growth for original and simplified version in 500000 iterations. Source: Created by authors.

Detector (Test Set)	AP	AP _{.50}	AP _{.75}	AP _{small}	AP _{medium}	AP _{large}
YOLOv3 608 x 608 (Test-dev) ¹	33.0	57.9	34.4	18.3	35.4	41.9
YOLOv3 608 x 608 (Test-val)	33.4	58.5	34.5	19.4	36.4	43.8
YOLOv3-tiny original (Test-val)	14.5	31.9	11.6	3.2	13.7	24.8
YOLOv3-tiny simplified (Test-val)	14.4	31.8	11.2	3.1	13.6	24.5

Table 4.1: mAP with different test set. Source: Created by authors.

terms of TP, both versions were not that far apart meaning that recall will be very similar as well. On the other hand, the original model made more incorrect predictions that ultimately translates into a lower value for accuracy.

Detector	AP _{.50}	TP	FP	FN	Acc	R
YOLOv3-tiny original	31.9	20651	372064	14443	5.7	57.7
YOLOv3-tiny simplified	31.8	20554	321056	14540	6.6	57.0

Table 4.2: Original vs Simplified. Source: Created by authors.

From obtained results in this first section, it can be seen that the simplified version achieves almost identical performance that the original version which ultimately is not a bad thing. Our aim was to prove that the changes did not affect negatively darknet and YOLO. In that

sense, maintaining the same level of performance is satisfactory. For example, although there is a difference in $AP_{.50}$ and recall between both models, this difference is only 0.1 and 0.7 respectively. The difference is explained by the fact that the model randomly changes resolutions during training, so it is possible that in a new training the $AP_{.50}$ could be higher than the one obtained at this time. One significant improvement though, is that by simplifying the code the overall size of the framework decreased by 1.4 times going from approximately 500 kb to 350 kb in the main library file.

4.2 Tuning the parameters

The results from the first set of training are presented in Table 4.3. Let's now breakdown the results. The first thing to notice is that the training belonging to the LR 0.1 failed even before reaching the 1000 iterations. It seems that this rate is too high for training and therefore the learning is unstable. In the case of an LR of 0.005, we obtained $AP_{.50}$ that were too apart from each other. In fact, the test that did not perform any S obtained the worst $AP_{.50} = 24.7$. On the other hand, when applying the S, the $AP_{.50}$ reached 30.6 and the recall was the highest of all. Meaning that the the number of objects correctly detected was good. Almost the same trend follows for the 0.002 and 0.0005 LR. For 0.0001, the situation was different. The highest $AP_{.50}$ was given for the case in which no S was done. A rate of this size is already small enough for the training to be stable, decreasing it two more times makes the learning too small that it does not learn much anymore. In terms of the time taken, all tests took similar times to complete therefore the LR as well as the S had not great impact on training time. In fact, the small difference in time is more likely given by the parameter "random" in the configuration file, for which the images change size randomly for training. Given all previous considerations, the $AP_{.50}$ s in bold are the candidates chosen for the second set of training.

Figure 4.3 shows the evolution of the $AP_{.50}$ from the first iteration until iteration 100000. Note that in some cases, the training seems that goes up and down randomly. Changing resolution from time to time seems to be responsible for this behavior. Conversely, the LR that seems to have the most stable training is 0.0001. Training with steps and without steps also impacts the resulting $AP_{.50}$. In all of the cases, applying the first S at 40000 makes that the $AP_{.50}$ increases considerably with 0.005 being the one with the highest jump from 23 to 29. Instead, the second step at 80000 fulfills the role of stabilizing the training. It is important to point out that this behaviour is given by the action of performing the S rather than when the S is performed as it

LR	S (000)	B	Subd	AP _{.50}	T (d.)	TP	FP	FN	Acc.	R.
0.1	-	64	2	-	-	-	-	-	-	-
0.1	40 - 80	64	2	-	-	-	-	-	-	-
0.005	-	64	2	24.7	0.81	18871	353284	16223	6.1	50.1
0.005	40 - 80	64	2	30.6	0.82	20332	346912	14762	6.2	55.7
0.002	-	64	2	26.3	0.82	18852	270762	16242	7.0	50.3
0.002	40 - 80	64	2	30.8	0.82	20143	338228	14951	6.4	55.1
0.0005	-	64	2	27.7	0.80	19454	334865	15640	6.3	53.1
0.0005	40 - 80	64	2	29.6	0.81	19965	359209	15129	5.8	54.5
0.0001	-	64	2	27.9	0.80	19418	328067	15676	6.3	52.2
0.0001	40 - 80	64	2	26.0	0.80	19125	376721	15969	5.5	51.2

Table 4.3: Initial Training - 100000 Iterations. Source: Created by authors.

seems that it helps the training to leave a position of a local minimum.

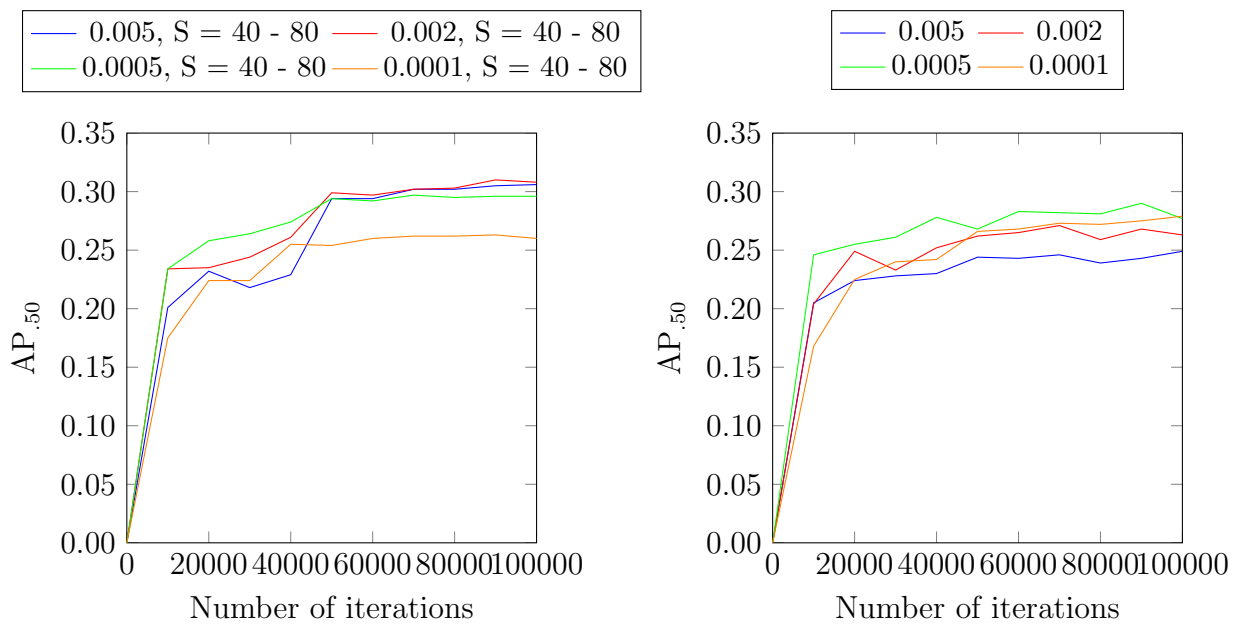


Figure 4.3: AP_{.50} growth for the first subset of experiments. Source: Created by authors.

Table 4.4 shows the second part of the training in which the chosen LR were the ones that gave a higher AP_{.50} with regard to being trained with or without S. The candidates from the initial training were 0.005, 0.002 and 0.0005 with S and 0.0001 without steps. Now, for this new set of tests, the parameters considered were the size of the B and Subd. The results in the second set were more alike given that the LR with higher AP_{.50} are the ones with a B equal to 128 and Subd of 4. Also, the B and Subd did affect the training time, getting smaller times for the

configuration with small B. Conversely, the Subd tends to increase training size if being higher. Since we put emphasis on the $AP_{.50}$, the candidate chosen for the final training was the one with an **LR equal to 0.005 and steps in 40000 and 80000, with a B equal to 128 and Subd of 4**, as it obtains a $AP_{.50}$ equal to **31.4**. Even more, this is the one that also obtained the highest value for recall being equal to **56.4**.

LR	S (000)	B	Subd	$AP_{.50}$	T (d.)	TP	FP	FN	Acc.	R.
0.005	40 - 80	16	2	-	-	-	-	-	-	-
0.005	40 - 80	16	8	-	-	-	-	-	-	-
0.005	40 - 80	128	4	31.4	1.55	20422	321737	14672	6.8	56.4
0.005	40 - 80	128	64	29.7	1.89	19434	255012	15660	8.0	52.0
0.002	40 - 80	16	2	27.6	0.27	19804	379454	15290	5.5	53.4
0.002	40 - 80	16	8	26.1	0.31	18874	299324	16220	6.4	50.0
0.002	40 - 80	128	4	31.1	1.52	20254	325005	14840	6.6	55.6
0.002	40 - 80	128	64	29.8	2.06	19407	257442	15687	7.6	52.1
0.0005	40 - 80	16	2	28.0	0.26	19650	379963	15444	5.5	53.3
0.0005	40 - 80	16	8	26.8	0.31	18981	308349	16113	6.2	50.5
0.0005	40 - 80	128	4	29.6	1.50	19986	337172	15108	6.2	54.6
0.0005	40 - 80	128	64	28.9	2.05	19274	290050	15820	6.7	51.8
0.0001	-	16	2	26.6	0.27	19258	365004	15836	5.4	51.9
0.0001	-	16	8	25.0	0.31	18771	348523	16323	5.7	49.9
0.0001	-	128	4	28.0	1.46	19430	328091	15664	6.2	52.7
0.0001	-	128	64	26.4	2.07	18894	305170	16200	6.3	50.3

Table 4.4: Intermediate Training - 100000 Iterations. Source: Created by authors.

Finally, Table 4.5 shows the result of the last set of training. The $AP_{.50}$ did not change considerably within the candidate from the second set and the two tests carried out in this one. The winning combination that will be used in video testing is the one with a step policy done in the iterations 160000 and 180000. The $AP_{.50}$ and recall for this candidate are 31.5 and 56.4 respectively.

LR	S (000)	B	Subd	$AP_{.50}$	T (d.)	TP	FP	FN	Acc.	R.
0.005	40 - 80	128	4	31.4	1.55	20422	321737	15089	6.8	56.1
0.005	80 - 90	128	4	31.1	1.51	20342	318642	14752	6.6	55.8
0.005	160 - 180	128	4	31.5	3.02	20584	314085	14510	6.8	56.4

Table 4.5: Final Training. Source: Created by authors.

Something to point out here is that although this one obtained the highest $AP_{.50}$, it would

probably be fine to use the selected candidate from the second set as the difference in $AP_{.50}$ is not higher than 0.1. In cases in which time is a premium, this should be the route of parameter combinations for training to be taken as it only takes half of the time to train in comparison to the finally selected combination.

4.3 Video

In the last section of results, we evaluate how well the winning combination performs both in a computer and in the Raspberry Pi against the original and simplified version in terms of the FPS. Table 4.6 shows the FPS difference obtained from the two platforms. In a high resolution video, there is no much difference in FPS with respect to the input resolution used. In other cases, the change is more marked. The FPS increase when decreasing both the video resolution and the input resolution, being the combination of a video resolution of 304 x 171 and input resolution of 192 x 192 the fastest with more than 230 FPS on the computer. On the other hand, the results in the raspberry pi were too slow when compared to the computer. The FPS changes in the same fashion as in the computer but in none of the cases the FPS was higher than 0.40. Processing at 0.1 FPS, a 15-seconds video with 25 FPS takes a lot of time. The total number of frames to process is 375 given by 25 FPS x 15 seconds, at a rate of 0.1 is equal to 3750 seconds = 62.5 minutes = 1.04 hours. Other alternatives should be considered otherwise the time taken in the different tests make detection to be really slow. A note aside, FFmpeg does play a role on FPS as the FPS was increased, in comparison to the the original video resolution, by more than two times for both 960x540 and 304x171 resolutions.

Table 4.7 shows on the other hand the total detections. In general, the higher the input resolution the higher the total number of detections. As soon as the input resolution decreases the number of detections decreases as well. However, changing the video resolution does not seem to affect that much the number of detections and there are cases in which the number of detections is higher in a lower video resolution as in 960 x 540, but only for the person category. It seems that the detectors are better in detecting cars in higher video resolutions than in smaller ones. There are also some outcasts, like the one marked with an asterisk, that do not follow the general trend. Now, the difference in detections between the computer and the Raspberry Pi is not significant and varies from video 1 to video 2. For video 1, the difference does not go higher than 10 detections in most cases. The difference is more marked for video 2. The reason behind it might be that video 2 has a sequence of frames where there is a crowd of people thus it misses

Detector	Video R	Input R	Computer		Raspberry Pi	
			Video 1	Video 2	Video 1	Video 2
original	1920x1080	416x416	34.9	34.6	<0.1	<0.1
		224x224	36.2	35.9	<0.1	<0.1
		192x192	36.1	36.7	<0.1	<0.1
	960x540	416x416	86.8	83.3	<0.1	<0.1
		224x224	109.1	107.9	0.20	0.20
		192x192	131.6	130.1	0.31	0.31
	304x171	416x416	96.8	96.8	<0.1	<0.1
		224x224	212.6	212.7	0.21	0.21
		192x192	233.3	237.1	0.34	0.37
simplified	1920x1080	416x416	36.4	34.6	<0.1	<0.1
		224x224	35.8	36.7	<0.1	<0.1
		192x192	36.5	36.8	<0.1	<0.1
	960x540	416x416	84.3	83.2	<0.1	<0.1
		224x224	108.2	105.9	0.21	0.21
		192x192	131.1	130.1	0.33	0.30
	304x171	416x416	96.0	96.8	<0.1	<0.1
		224x224	211.4	212.0	0.22	0.20
		192x192	238.5	235.2	0.31	0.31
0.005	1920x1080	416x416	34.4	35.1	<0.1	<0.1
		224x224	35.5	36.8	<0.1	<0.1
		192x192	36.6	36.2	<0.1	<0.1
	960x540	416x416	87.0	82.1	<0.1	<0.1
		224x224	111.9	106.4	0.25	0.20
		192x192	130.4	129.5	0.31	0.32
	304x171	416x416	96.4	96.9	<0.1	<0.1
		224x224	207.9	215.2	0.25	0.29
		192x192	234.4	236.4	0.31	0.30

Table 4.6: FPS on computer vs Raspberry Pi. Source: Created by authors.

to detect objects in this category from time to time.

Detector	Video R	I.R.	Computer				Raspberry Pi			
			Video 1		Video 2		Video 1		Video 2	
			person	car	person	car	person	car	person	car
orig	1920x1080	416	982	622	3589	544	-	-	-	-
		224	770	169	2822	249	-	-	-	-
		192	601	136	2251	159	-	-	-	-
	960x540	416	975	563	3617	540	-	-	-	-
		224	795	164	2764	290	801	158	2911	292
		192	598	141	2403	183	597	139	2426	180
	304x171	416	905	412	3199	534	-	-	-	-
		224	737	99*	2639	325	737	94	2683	327
		192	593	135	2120	204	592	128	2211	216
simp	1920x1080	416	989	542	3592	544	-	-	-	-
		224	793	71	2509	232	-	-	-	-
		192	641	70	2159	162	-	-	-	-
	960x540	416	998	476	3616	530	-	-	-	-
		224	818	67	2698	276	821	62	2742	278
		192	640	55	2094	222	639	50	2174	231
	304x171	416	964	362	3204	527	-	-	-	-
		224	816	48	2400	285	815	46	2521	290
		192	630	40	1821	215	630	36	1960	227
0.005	1920x1080	416	966	554	3566	521	966	556	3585	541
		224	743	56	2157	202	744	55	2243	214
		192	561	39	1828	166	567	37	1881	170
	960x540	416	968	533	3581	549	970	532	3631	551
		224	793	72	2298	269	795	69	2297	268
		192	577	45	1783	223	575	40	1912	235
	304x171	416	922	368	3209	536	921	356	3231	538
		224	766	42	2037	311	767	42	2156	328
		192	574	25	1715	274	573	25	1792	291

Table 4.7: Total detections on computer vs Raspberry Pi. Source: Created by authors.

Chapter 5

Conclusions

The final chapter covers a review of the work done in this research project starting from the literature review, deciding what to test and improve to finally evaluate how it went. This work ends with some concluding remarks and future work in this research topic.

5.1 Literature Review

This work started by reviewing the most important concepts related to object detection going from what is this about into the most important applications, challenges and features. We showed that object detection is a hot topic in research at the moment and that it is being used strongly in many visual applications. But even though the high popularity, there were some areas in which object detection could be improved therefore boosting even more interest from researchers. Next, we focused on explaining a little bit about CNNs and how these look in general. More specifically, we also took interest in implementations for embedded systems. There were some good models that have been proposed already for constrained environments but we explored the idea of using a model that is considered fast when compared to what could be called state-of-the-art detectors such as SSD and Faster R-CNN. This model is YOLO. Most of the success of YOLO is given by the use of multiple borrowed techniques from different models. We showed how these combinations have improved YOLO making it more robust and powerful while still maintaining its main characteristic: speed. And in terms of speed, YOLO has a smaller version called Tiny YOLO which runs even faster. Nevertheless, YOLO suffered from the same challenges showcased for object detection plus some extra consideration on localization errors and bounding box alignment.

5.2 Experiments

Having darknet as the main framework to train and use YOLO was helpful but there is some room for improvement. In that sense, we took away the little parts that were not related to YOLO and reorganized some bits of code from one place to another one. The main darknet code is not able to run on the Raspberry Pi due to a segmentation fault so it was time to do some debugging as well. Once this was ready, learning how to use darknet was not that much complicated although a complete guide could be helpful. Then, it came to the training part. Training on a consumer-level computer would have taken time that was already in short supply. For that reason, the training was done using 4 K80 GPUs which allowed us to consider multiple parameter options for training: learning rate, steps, batch size and subdivision. Another important aspect was choosing the right pretrained weight file. Using weights from a different architecture or starting weights randomly increased the training time needed to achieve a better $AP_{.50}$.

5.3 Results

YOLO is able to run on the Raspberry Pi, albeit not at the speed we were looking for. Using CNNs in this type of devices put a heavy load on the already constrained hardware resources. Even smaller models have still a lot of work to do to be competitive with their GPU based counterparts. However, we were only able to run Tiny YOLO on the Raspberry Pi, the full version is too big for this type of architectures. In terms of average precision, the results gave some new perspectives on how to perform training. By tuning the parameters, we explored several routes where some were quite successful while others fell short in terms of precision. Probably the most remarkable aspect of the training was that by increasing the LR and the B a smaller training time was required to achieve a very similar $AP_{.50}$ when compared to the conditions used for full training. Finally, it is time to talk a bit about performance on video. As expected, the detection speed was really slow on the Raspberry Pi compared to using a consumer-level computer. Both the video resolution and the input resolution do affect detection speed with the most remarkable case being represented by a video resolution of 304x171 and input resolution of 192x192 when the detector reaches an average speed of 234.4 FPS. An important factor to account for, it is where the video conversion is taking place, either inside or outside of darknet. The total detections are also affected by the changes in video resolution and input

resolution, but it's more apparent when changing the input resolution.

5.4 Future Work

Achieving a higher precision and speed is among the most important problems researchers are working on regard object detection. And for that there are several alternatives: tuning the parameters, modifying the network architecture, and improving computing power. In this work, we took care of only one of these activities: tuning the parameters. A more detailed exploration on the parameters is needed in order to determine which ones have the more impact and tune them accordingly. Adjusting parameters related to data augmentation such as saturation, exposure and hue could may have an impact on increasing the $AP_{.50}$. Modifying YOLO's architecture by increasing or decreasing the number of convolutional layers or by adding new methods and techniques could also be fruitful.

In terms of the training dataset, COCO is pretty good and challenging in its own respect. But here is the deal. Object detectors are better at detecting some categories than others. There could be several reasons for that like the number of ground truth objects for training is not equivalent between categories or that the training images present better information and features for some and not all of them. Trying a different dataset or working with a subset of COCO with the categories that are not well detected could bring more details on the reason behind this behavior and probably allow to prepare measures to improve detection. It will be also interesting to add more categories to the 80 classes used by COCO and thus be able to detect more objects.

Testing YOLO performance with videos was among the things we tried to do. In that account, there are several things we did not explore. Things that could be tested in the future include but are not limited to: using live feed instead of a video, working with videos of night scenes or low light conditions and trying a different video format and compression. Something that could be tested as well but is more related to a configuration in darknet is trying different probability thresholds. We noticed that this configuration does affect detection speed.

Although our main approach was to show if it was feasible to run YOLO solely on the Raspberry Pi, it would be very interesting to try out a combination of the Raspberry Pi with a neural stick like the Intel Movidius [76] or use a different hardware such as the Nvidia Jetson Nano [77] which brings the benefits of full-size GPUs to embedded systems. Another approach will be to consider running YOLO on a cellphone as it has become essential to our lifestyles and therefore

having an object detector on it could help the development of interesting applications in areas such as augmented reality. Finally, at the time of wrapping up this work, the Raspberry Pi 4 [78] was being launched with a more powerful processor and up to 4GB of RAM. These new characteristics could easily help improve detection speed.

Bibliography

- [1] S. Agarwal, J. O. D. Terrail, and F. Jurie, “Recent advances in object detection in the age of deep convolutional neural networks,” *CoRR*, vol. abs/1809.03193, 2018. [Online]. Available: <http://arxiv.org/abs/1809.03193>
- [2] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, “You only look once: Unified, real-time object detection,” *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.
- [3] J. Redmon and A. Farhadi, “Yolo9000: Better, faster, stronger,” *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 6517–6525, 2017.
- [4] —, “Yolov3: An incremental improvement,” *arXiv*, 2018.
- [5] S. Ren, K. He, R. Girshick, and J. Sun, “Faster r-cnn: Towards real-time object detection with region proposal networks,” *IEEE Transactions on Pattern Analysis & Machine Intelligence*, vol. 39, no. 06, pp. 1137–1149, jun 2017.
- [6] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg, “Ssd: Single shot multibox detector,” in *Computer Vision – ECCV 2016*, B. Leibe, J. Matas, N. Sebe, and M. Welling, Eds. Cham: Springer International Publishing, 2016, pp. 21–37.
- [7] D. Molloy, *Exploring Raspberry Pi: interfacing to the real world with embedded Linux*. John Wiley & Sons, 2016.
- [8] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick, “Microsoft coco: Common objects in context,” *Computer Vision – ECCV 2014 Lecture Notes in Computer Science*, p. 740–755, 2014.

- [9] L. JL, T. SM, W. JQ, Z. HB, and W. YK, “A review on object detection based on deep convolutional neural networks for autonomous driving,” in *Chinese Control and Decision Conference*, 2019.
- [10] B. Wu, F. Iandola, P. H. Jin, and K. Keutzer, “Squeezedet: Unified, small, low power fully convolutional neural networks for real-time object detection for autonomous driving,” in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, July 2017.
- [11] S. Selvi and A. I. Chellam, “Smart video surveillance: Object detection, tracking and classification,” *International Journal of Innovations and Advancement in Computer Science*, vol. 7, no. 3, March 2018.
- [12] A. Coates and A. Y. Ng, “Multi-camera object detection for robotics,” in *2010 IEEE International Conference on Robotics and Automation*, May 2010, pp. 412–419.
- [13] A. Kundu, K. M. Krishna, and J. Sivaswamy, “Moving object detection by multi-view geometric techniques from a single camera mounted robot,” in *2009 IEEE/RSJ International Conference on Intelligent Robots and Systems*, Oct 2009, pp. 4306–4312.
- [14] X. Yang, Y. Tian, C. Yi, and A. Arditi, “Context-based indoor object detection as an aid to blind persons accessing unfamiliar environments,” in *Proceedings of the 18th ACM International Conference on Multimedia*, ser. MM ’10. New York, NY, USA: ACM, 2010, pp. 1087–1090. [Online]. Available: <http://doi.acm.org/10.1145/1873951.1874156>
- [15] T. Winlock, E. Christiansen, and S. Belongie, “Toward real-time grocery detection for the visually impaired,” in *2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition - Workshops*, June 2010, pp. 49–56.
- [16] Chia-Hsiang Lee, Yu-Chi Su, and Liang-Gee Chen, “An intelligent depth-based obstacle detection system for visually-impaired aid applications,” in *2012 13th International Workshop on Image Analysis for Multimedia Interactive Services*, May 2012, pp. 1–4.
- [17] R. Tapu, B. Mocanu, A. Bursuc, and T. Zaharia, “A smartphone-based obstacle detection and classification system for assisting visually impaired people,” in *The IEEE International Conference on Computer Vision (ICCV) Workshops*, June 2013.

- [18] Xiangrong Chen and A. L. Yuille, “A time-efficient cascade for real-time object detection: With applications for the visually impaired,” in *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR’05) - Workshops*, Sep. 2005, pp. 28–28.
- [19] S. Agarwal, J. O. D. Terrail, and F. Jurie, “Recent advances in object detection in the age of deep convolutional neural networks,” *CoRR*, vol. abs/1809.03193, 2018. [Online]. Available: <http://arxiv.org/abs/1809.03193>
- [20] B. S. Manjunath, J. . Ohm, V. V. Vasudevan, and A. Yamada, “Color and texture descriptors,” *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 11, no. 6, pp. 703–715, June 2001.
- [21] M. Bober, “Mpeg-7 visual shape descriptors,” *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 11, no. 6, pp. 716–719, June 2001.
- [22] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, <http://www.deeplearningbook.org>.
- [23] R. Xu, C. Li, A. H. Paterson, Y. Jiang, S. Sun, and J. Robertson, “Aerial images and convolutional neural network for cotton bloom detection,” *Frontiers in Plant Science*, vol. 8, 02 2018.
- [24] K. Simonyan and A. Zisserman, “Very deep convolutional networks for large-scale image recognition,” *CoRR*, vol. abs/1409.1556, 2014. [Online]. Available: <http://arxiv.org/abs/1409.1556>
- [25] T. Lin, P. Goyal, R. Girshick, K. He, and P. Dollar, “Focal loss for dense object detection,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pp. 1–1, 2018.
- [26] T. Lin, P. Dollár, R. B. Girshick, K. He, B. Hariharan, and S. J. Belongie, “Feature pyramid networks for object detection,” *CoRR*, vol. abs/1612.03144, 2016. [Online]. Available: <http://arxiv.org/abs/1612.03144>
- [27] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, “Mobilenets: Efficient convolutional neural networks for mobile vision applications,” *CoRR*, vol. abs/1704.04861, 2017. [Online]. Available: <http://arxiv.org/abs/1704.04861>

- [28] F. N. Iandola, M. W. Moskewicz, K. Ashraf, S. Han, W. J. Dally, and K. Keutzer, “Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <1mb model size,” *CoRR*, vol. abs/1602.07360, 2016. [Online]. Available: <http://arxiv.org/abs/1602.07360>
- [29] J. Huang, V. Rathod, C. Sun, M. Zhu, A. Korattikara, A. Fathi, I. Fischer, Z. Wojna, Y. Song, S. Guadarrama, and K. Murphy, “Speed/accuracy trade-offs for modern convolutional object detectors,” in *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, July 2017, pp. 3296–3297.
- [30] J. Dai, Y. C. Li, K. He, and J. Sun, “R-fcn: Object detection via region-based fully convolutional networks,” in *NIPS*, 2016.
- [31] R. Mottaghi, X. Chen, X. Liu, N. Cho, S. Lee, S. Fidler, R. Urtasun, and A. Yuille, “The role of context for object detection and semantic segmentation in the wild,” in *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*. IEEE Computer Society, 1 2014, pp. 891–898.
- [32] J. Wang and L. Perez, “The effectiveness of data augmentation in image classification using deep learning,” *Convolutional Neural Networks Vis. Recognit*, 2017.
- [33] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov, “Improving neural networks by preventing co-adaptation of feature detectors,” *arXiv e-prints*, Jul. 2012.
- [34] J. Redmon and A. Angelova, “Real-time grasp detection using convolutional neural networks,” in *2015 IEEE International Conference on Robotics and Automation (ICRA)*, May 2015, pp. 1316–1322.
- [35] M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman, “The PASCAL Visual Object Classes Challenge 2007 (VOC2007) Results,” <http://www.pascal-network.org/challenges/VOC/voc2007/workshop/index.html>.
- [36] S. Ioffe and C. Szegedy, “Batch normalization: Accelerating deep network training by reducing internal covariate shift,” in *Proceedings of the 32Nd International Conference on International Conference on Machine Learning - Volume 37*, ser. ICML’15. JMLR.org, 2015, pp. 448–456. [Online]. Available: <http://dl.acm.org/citation.cfm?id=3045118.3045167>

- [37] K. He and J. Sun, “Convolutional neural networks at constrained time cost,” *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 5353–5360, 2015.
- [38] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016, pp. 770–778.
- [39] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. Alemi, “Inception-v4, inception-resnet and the impact of residual connections on learning,” *AAAI Conference on Artificial Intelligence*, 2017. [Online]. Available: <https://aaai.org/ocs/index.php/AAAI/AAAI17/paper/view/14806>
- [40] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, “ImageNet Large Scale Visual Recognition Challenge,” *International Journal of Computer Vision (IJCV)*, vol. 115, no. 3, pp. 211–252, 2015.
- [41] J. Lin and M. Sun, “A yolo-based traffic counting system,” in *2018 Conference on Technologies and Applications of Artificial Intelligence (TAAI)*, Nov 2018, pp. 82–85.
- [42] J. Tao, H. Wang, X. Zhang, X. Li, and H. Yang, “An object detection system based on yolo in traffic scene,” in *2017 6th International Conference on Computer Science and Network Technology (ICCSNT)*, Oct 2017, pp. 315–319.
- [43] N. Bhandary, C. MacKay, A. Richards, J. Tong, and D. C. Anastasiu, “Robust classification of city roadway objects for traffic related applications,” in *2017 IEEE SmartWorld, Ubiquitous Intelligence Computing, Advanced Trusted Computed, Scalable Computing Communications, Cloud Big Data Computing, Internet of People and Smart City Innovation (SmartWorld/SCALCOM/UIC/ATC/CBDCOM/IOP/SCI)*, Aug 2017, pp. 1–6.
- [44] Z. Yi, S. Yongliang, and Z. Jun, “An improved tiny-yolov3 pedestrian detection algorithm,” *Optik*, vol. 183, pp. 17 – 23, 2019. [Online]. Available: <http://www.sciencedirect.com/science/article/pii/S003040261930155X>
- [45] Álvaro Arcos-García, J. A. Álvarez García, and L. M. Soria-Morillo, “Evaluation of deep neural networks for traffic sign detection systems,” *Neurocomputing*, vol. 316, pp. 332 – 344, 2018. [Online]. Available: <http://www.sciencedirect.com/science/article/pii/S092523121830924X>

- [46] R. Widyastuti and C.-K. Yang, “Cat’s nose recognition using you only look once (yolo) and scale-invariant feature transform (sift),” *2018 IEEE 7th Global Conference on Consumer Electronics (GCCE)*, pp. 55–56, 2018.
- [47] W. Xu and S. Matzner, “Underwater fish detection using deep learning for water power applications,” *CoRR*, vol. abs/1811.01494, 2018. [Online]. Available: <http://arxiv.org/abs/1811.01494>
- [48] J. R. Parham, J. P. Crall, C. V. Stewart, T. Y. Berger-Wolf, and D. I. Rubenstein, “Animal population censusing at scale with citizen science and photographic identification,” in *AAAI Spring Symposia*, 2017.
- [49] C.-A. Brust, T. Burghardt, M. Groenenberg, C. Kading, H. S. Kuhl, M. L. Manguette, and J. Denzler, “Towards automated visual monitoring of individual gorillas in the wild,” in *The IEEE International Conference on Computer Vision (ICCV) Workshops*, Oct 2017.
- [50] Y. Zhong, J. Gao, Q. Lei, and Y. Zhou, “A vision-based counting and recognition system for flying insects in intelligent agriculture,” *Sensors*, vol. 18, no. 5, p. 1489, 2018.
- [51] J. Seo, J. Sa, Y. Choi, Y. Chung, D. Park, and H. Kim, “A yolo-based separation of touching-pigs for smart pig farm applications,” in *2019 21st International Conference on Advanced Communication Technology (ICACT)*, Feb 2019, pp. 395–401.
- [52] E. G. Solberg, “Deep neural networks for object detection in agricultural robotics,” Master’s thesis, Norwegian University of Life Sciences, 2017.
- [53] Y. Tian, G. Yang, Z. Wang, H. Wang, E. Li, and Z. Liang, “Apple detection during different growth stages in orchards using the improved yolo-v3 model,” *Computers and Electronics in Agriculture*, vol. 157, pp. 417 – 426, 2019. [Online]. Available: <http://www.sciencedirect.com/science/article/pii/S016816991831528X>
- [54] M. Radovic, O. Adarkwa, and Q. Wang, “Object recognition in aerial images using convolutional neural networks,” *Journal of Imaging*, vol. 3, no. 2, 2017. [Online]. Available: <https://www.mdpi.com/2313-433X/3/2/21>
- [55] A. V. Etten, “You only look twice: Rapid multi-scale object detection in satellite imagery,” *ArXiv*, vol. abs/1805.09512, 2018.

- [56] J. Carlet and B. Abayowa, “Fast vehicle detection in aerial imagery,” *CoRR*, vol. abs/1709.08666, 2017. [Online]. Available: <http://arxiv.org/abs/1709.08666>
- [57] S. Yang, J. Zhang, C. Bo, M. Wang, and L. Chen, “Fast vehicle logo detection in complex scenes,” *Optics & Laser Technology*, vol. 110, pp. 196 – 201, 2019, special Issue: Optical Imaging for Extreme Environment. [Online]. Available: <http://www.sciencedirect.com/science/article/pii/S0030399218310715>
- [58] Y. Xie, J. Cai, R. Bhojwani, S. Shekhar, and J. Knight, “A locally-constrained yolo framework for detecting small and densely-distributed building footprints,” *International Journal of Geographical Information Science*, vol. 0, no. 0, pp. 1–25, 2019. [Online]. Available: <https://doi.org/10.1080/13658816.2019.1624761>
- [59] M. S. Chauhan, A. Singh, M. Khemka, A. Prateek, and R. Sen, “Embedded cnn based vehicle classification and counting in non-laned road traffic,” in *Proceedings of the Tenth International Conference on Information and Communication Technologies and Development*, ser. ICTD '19. New York, NY, USA: ACM, 2019, pp. 5:1–5:11. [Online]. Available: <http://doi.acm.org/10.1145/3287098.3287118>
- [60] “Jetson tx2 module,” Dec 2018. [Online]. Available: <https://developer.nvidia.com/embedded/buy/jetson-tx2>
- [61] Z. Zhao, Z. Jiang, N. Ling, X. Shuai, and G. Xing, “Ecrt: An edge computing system for real-time image-based object tracking,” in *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems*, ser. SenSys '18. New York, NY, USA: ACM, 2018, pp. 394–395. [Online]. Available: <http://doi.acm.org/10.1145/3274783.3275199>
- [62] L. Fei-Fei, “Imagenet: crowdsourcing, benchmarking & other cool things,” <http://image-net.org/about-publication>, 2010.
- [63] “Flickr,” https://farm4.staticflickr.com/3160/2901140028_aaed0953c0_z.jpg.
- [64] “Detection evaluation,” <http://cocodataset.org/#detection-eval>.
- [65] “Test guidelines,” <http://cocodataset.org/#guidelines>.
- [66] “Coco detection challenge,” https://competitions.codalab.org/competitions/5181#learn_the_details.

- [67] <https://hpc.yachay.gob.ec/>.
- [68] “OpenCV,” <https://opencv.org/about/>.
- [69] “OpenMP,” <https://www.openmp.org/>.
- [70] “CUDA FAQ,” <https://www.openmp.org/>.
- [71] “NVIDIA cuDNN,” <https://developer.nvidia.com/cudnn>.
- [72] “Darknet,” <https://github.com/pjreddie/darknet>, accessed: 2019-02-10.
- [73] W. Click, “Unsplash,” <https://unsplash.com/photos/jkC1ul95ujQ>, 2018, accessed: 2019-04-15.
- [74] “Pexels,” <https://www.pexels.com/videos/>, accessed: 2018-10-20.
- [75] “FFmpeg documentation,” <https://ffmpeg.org/ffmpeg-all.html>.
- [76] “Intel Neural Compute Stick,” <https://software.intel.com/en-us/neural-compute-stick>.
- [77] “Jetson nano developer kit,” <https://developer.nvidia.com/embedded/jetson-nano-developer-kit>.
- [78] “Raspberry Pi 4,” <https://www.raspberrypi.org/products/raspberry-pi-4-model-b/>.