Università degli Studi di Padova

Dipartimento di Ingegneria dell'Informazione

Master Thesis in Control Systems Engineering

Computer Vision Based Person Detection on Embedded Systems

Master Candidate

Francesco Pasti

Student ID 2024291

Supervisor

Professor Nicola Bellotto

University of Padova

Academic Year
2021/2022

To my family
and friends

Abstract

In industrial settings, safety plays a crucial role to prevent incidents between man and machines.
This thesis, has been developed, in collaboration with a company, with the purpose of implementing an aid safety system to decrease the danger of workers in contact with dangerous machines. The system has to be able to detect persons based on computer vision techniques applied on a camera feed.
The state of the art approaches for human detection in computer vision are often based on machine learning and deep learning. However, this algorithms require a huge computational effort that demands considerable inference times, depending on the hardware, to produce the results.
Furthermore, the device in which the safety system needs to be implemented is a wireless control device, hence an embedded system that has limited hardware capabilities.
For this reasons, this thesis will firstly explore state of the art approaches for person detection and their impact on an embedded device. The work will then cover the original development of a deep neural network able to solve this problem.

Sommario

I sistemi di sicurezza in ambienti industriali hanno un ruolo importante e necessario per prevenire incidenti tra macchinari e lavoratori.
Questo elaborato è stato sviluppato in collaborazione con un'azienda, con lo scopo di studiare un sistema di sicurezza che mira a prevenire i rischi del lavoratore a contatto con pericolosi macchinari. Il sistema in studio, richiede di essere capace di identificare le persone grazie a tecniche di visione artificiale applicate su feed di telecamere.
Nello stato dell'arte, gli algoritmi relativi alla funzione di rilevamento persone si basano principalmente su machine learning e deep learning. Queste tecniche, tuttavia, richiedono una grande potenza di calcolo che, in base all'hardware impiegato, implica considerevoli tempi di inferenza per elaborare il risultato. Inoltre, il dispositivo sul quale è richiesto implementare il sistema di sicurezza è un radio comando con risorse limitate, ovvero un sistema embedded.
L'elaborato si pone quindi da una parte l'obiettivo di esplorare i sistemi per identificare persone allo stato dell'arte e di valutare il loro impatto su un dispositivo embedded, dall'altra si concentra sullo sviluppo originale di una rete neurale capace di risolvere il problema sopracitato.

Contents

List of Figures

List of Tables

List of Code Snippets

List of Acronyms

CNN	Convolutional Neural Network
COCO	Common Objects in Context
CPU	Central Processing Unit
DNN	Deep Neural Network
FN	False Negative
FP	False Positive
FPS	Frame Per Seconds
HOG	Histogram of oriented gradients
IOU	Interesction over Union
JSON	JavaScript Object Notation
L-CNN	Lightweigth CNN
NAS	Neural Architecture Search
OS	Operating System
RAM	Random Access Memory
RGB	Red Green Blue
R-CNN	Region proposal Convolutional Neural Network
ROI	Region of Interest
RPN	Region Proposal Network

SSD Single Shot MultiBox Detector

SVM Support Vector Machines

TP True Positive

VGA Video Graphics Array

VOC Visual Object Classes

YOLO You Only Look Once

1 Introduction

In industrial settings, it's mandatory to consider safety.
Safety has many declination and can be applied in different ways depending on the situation. One is preventing industrial machinery from working if certain conditions, such as the presence of an human, happen.

The work of this thesis has been developed in collaboration with a company based in Vicenza, specialized in producing wireless control devices with available safety functions, as a project for its Research and Development department. The aim was to develop a computer vision system able to detect persons using their remote control devices and camera feeds.
There are many existing systems capable of accomplishing this task, however not all of them are employable on the remote control devices as they have limited resources and are thus considered as embedded devices.
As the remote control device has really similar characteristics as the Raspberry Pi 4, this embedded platform, in addition to the older Raspberry Pi 3, is going to be considered for running the tests.
The main disadvantage of this devices, with respect to other edge platforms such as the Nvidia Jetson family, is that they are not equipped with a GPU. They are are thus not capable of performing efficiently the high number of parallel computations required by modern Convolutional Neural Networks. However it must be noted, that the main advantage of the Raspberry Pi 3 and 4 with respect to the Nvidia Jetson is their sustainability. They are indeed more power efficient and up to 10 times less expensive.

First of all this thesis is going to give a quick report of the state of the art for object detection and its declination on embedded devices analyzing studies and algorithms such as MobileNet, R-CNN and its family, SSD and Lightweight CNN. This, is going to be covered in chapter 2.

The most important and original parts of the thesis will then be reported:

- Popular algorithms are going to be studied and implemented on the test devices. This are Yolo, Viola and Jones and the Hog based person detector. Considering some old classic methods (i.e. Viola and Hog) for person detection, which were developed at a time where computers were much less powerful, could be a suitable solution for modern-day GPU-less embedded systems. These algorithms will be compared to YOLO v3, a representative solution of CNN-based architectures. The computer vision systems are going to be considered as an aid to other existing safety systems, for this reason they should be able to give a good detection rate and analyze a number of frames each second on the embedded platforms. This is going to be covered in chapter 3.
- After the study of the off-the shelf methods, this thesis is going to report the implementation of a customized Convolutional Neural Network. This novel architecture is strongly inspired by the original version of Yolo, it adopts some techniques and ideas to squeeze it as much as possible in order to reduce its inference time. As the network will be constructed from scratch, it will require to develop important components such as the loss function and the batch generator. This is going to be covered in chapter 4.
- Another solution is then going to be reported based on the custom training of the more recent Yolo v6, in its nano declination. This is going to be covered in chapter 5.

Finally chapter 5 will report a summary of the whole thesis and a discussion over the result obtained as well as the future works.

The code developed and used for the tests, the implementation and the deployment of the algorithms reported in this thesis is available on the Github page at this link[1].

[1]https://github.com/frapasti/Embedded-Person-detection

2 Related Work

In this chapter a quick overview of the theory for Computer Vision Systems based on Machine Learning and Convolutional Neural Network is going to be given.
This theory, provides the building blocks to understand the techniques and algorithms used through the thesis.

Then, some of the most popular State of the Art algorithms for Object detection and efficient Convolutional Neural Networks are going to be reported. The State of the Art gives useful insights on the current condition of the research and on the available systems that can be adopted for the scope of this thesis.

2.1 Computer Vision

Computer Vision is the collection of all the processes that aim to reproduce key aspects of human vision starting from 2D images. In this sub-chapter some of the fundamentals of this field needed to better comprehend this thesis are reported.

2.1.1 Images

Computer Vision systems start by taking images as inputs, for this reason it is necessary to study how they are represented.

An image is a set of pixels in a matrix form, each pixel can be represented in different ways, the most popular are :

- Unsigned char, assuming values in range $[0, 255]$ hence having 8 bits.
- Int, assuming values in range $[0, 655360]$ hence having 16 bits.
- Float, assuming values in range $[0, 1]$ and having 32 bits.

This representations refers to a pixel in gray-scale level, the number that it has assigned represents the intensity of the gray-level with 0 being black and the maximum value being white.

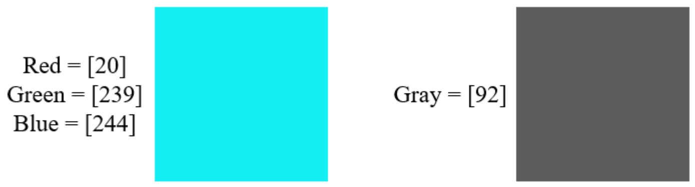


Figure 2.1: Visualization of a RGB pixel versus a gray-scale one

However, images can encode more information, a really important one is color. In this case, the most popular representation is RGB where each pixel is represented by three different values that encode the intensity of the red, green and blue level. The color, in the RGB representation is additive and each color is called channel.
In figure 2.1 the difference between RGB and gray-scale representation can be visualized, both representation in this case use channels with a depth of 8 bits (unsigned char).

Given that images are represented as a set of pixels matrices are their natural shape and they can be represented with their coordinate system with $(0, 0, 0)$ being the top left pixel of channel 1 and $(m, n, 2)$ being the bottom right pixel of channel 3.

2.1.2 High Level Image Processing

Given an image different types of operations and algorithms can be applied to it, they are grouped into :

- Low level image processing; which is concerned in changing pixels based on their value, the value of their neighbours, statistics of the image such as histograms, geometric transformations etc.
- Mid level image processing; which uses higher, more abstract level concepts to drive the algorithms and is often employed to highlight details such as edges or clusters or extract significant features such as corners and blobs. As an example in figure 2.2 it can be noticed that the popular Sift feature extractor is able to detect salient key-points as well as their orientation.

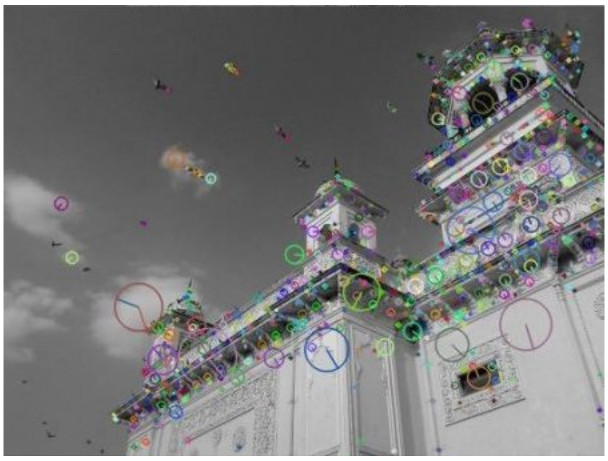


Figure 2.2: Visualization of the popular Sift (Scale Invariant Feature Transform) features

- High level image processing; it's concerned with even more abstract concepts present in images such as identifying what is present in an image (Image Classification) and detecting object as well as identifying them by giving their coordinates (Object localization). The types of detection strongly depend on the problem at end

For the scope of this thesis mainly high level concepts are employed hence a quick overview of some of them is going to be reported.
High level vision research in recent times concentrate itself on Machine learning and Deep learning with a particular focus on the latter.

Machine learning is a field of Computer Science that investigates how it is

possible to learn from data producing models capable of making predictions.
The learning framework usually has :

- Input; a dataset of inputs and their targets in case of Supervised learning or just inputs in case of Unsupervised learning.
- Goal; find a function f^* that best approximates the unknown function f.

The learning process usually optimizes the model parameters based on how different the output of the model is from the actual target.
In Computer Vision, Machine learning models are applied to domain specific representation of images. So, the features that best represent the application are extracted by means of low and mid level image processing algorithm and then models learn on such transformed data.
The representations that are fed to the model should be easy as learning is usually not effective when the input space has too many parameters.

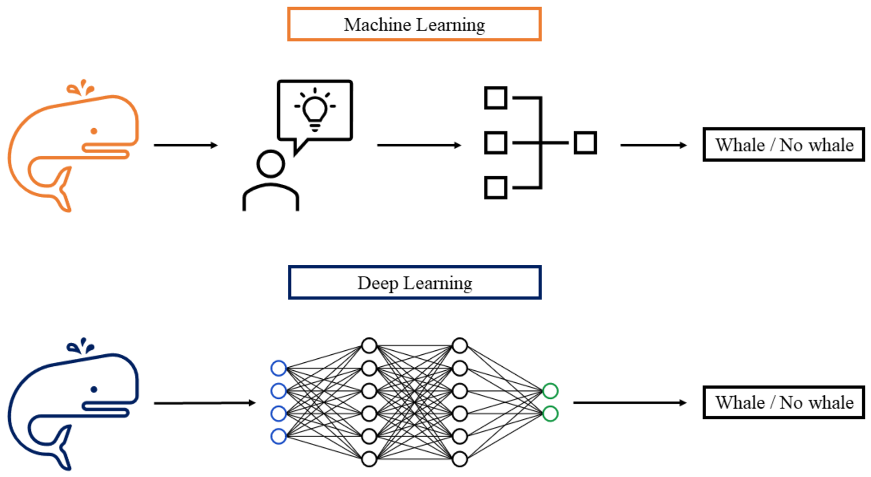


Figure 2.3: Machine learning approach to Computer Vision versus the Deep Learning one

Deep learning is a field of Machine learning based on Neural Networks. It is composed of multiple level of abstraction of the input data based on layers.
As it can be noticed in figure 2.3, Deep learning offers an end-to-end approach as the features are learned and adapted from the input data directly from the network and there's no need to extract the best ones like in Machine learning.

Typically Deep learning needs a huge amount of data in order to produce satisfying results.

The main problem of Machine and Deep learning is overfitting, that is when there's a nearly perfect fitting of the training data with no generalization leading to high errors on the test data. This typically happens when the model is too complex.
To avoid overfitting some techniques can be applied :

- Regularization; it can be of multiple types. The most popular one is dropout that randomly drops units during training, each unit is retained with a fixed probability
- Pre-processing; it consists in adding variance to the input data like noise and rotation. This typically leads to better, more robust models.

2.1.3 Deep Learning

As Deep learning is the most used and relevant technique for the scope of this thesis a quick introduction to some of its elements is going to be reported.

A Deep Neural Network is a composition of several simple functions called layers.
Each layer is composed of a set of neurons, the inputs of each neuron are some or all of the outputs of the preceding layer. In figure 2.4 it can be noticed how the layers are organized and how the outputs of the neurons are connected to the next layer. Each neuron then computes its output based on an activation function that has as input a weighted average of the inputs of the neuron plus a bias.

Several activation functions are used, the most popular ones are :

- Sigmoid function; its outputs are in range $[0, 1]$, it has a smooth output but saturates and kills gradients leading to poor learning close to 0 or 1.
- Tanh function; its outputs are in range $[-1, 1]$, it's simply a scaled version of the sigmoid and thus has the same problems.

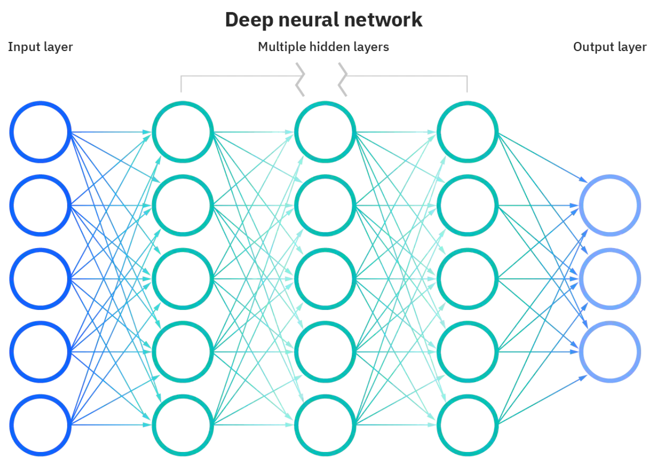


Figure 2.4: Visualization of the architecture of a Deep Neural Network with several layers and their neurons

- ReLU function; its outputs are in range [0, + inf], its a simpler function that allows faster training thanks to its non saturating nature and lighter computation

In figure 2.5 the different functions can be visualized.

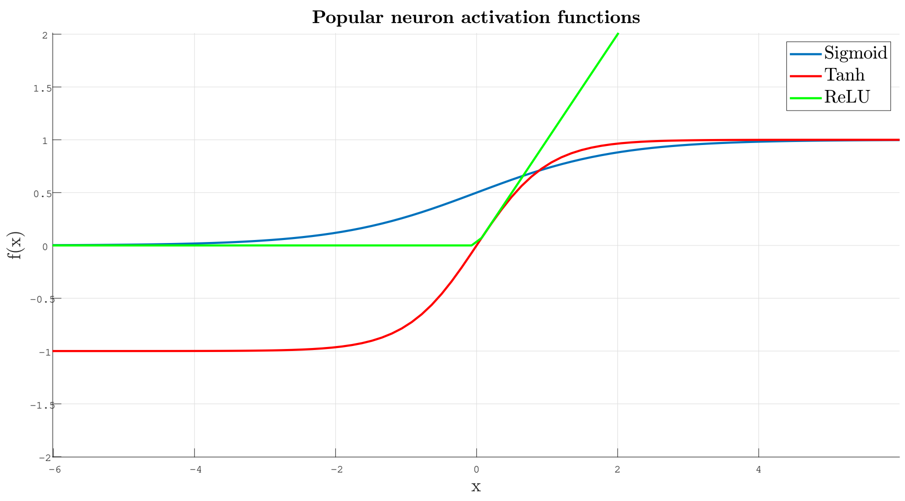


Figure 2.5: Plot comparison of the most popular activation functions for neurons in Neural Networks

Each neuron then has as parameters the weights applied to the input and the bias for its outputs. This parameters are usually randomly initialized and learned during training.

Images are 2 dimensional matrices and the spatial neighborhood between pixels is important, important features are local, shift and deformation invariant, they could be present at any position size and position of the image matrix. For this reason Computer Vision uses Convolutional Neural Networks.
This type of network is composed by Convolutional layers where filters of specified sized are applied to the image and their weights are learned. So the filters are applied through the whole image via a convolution leading to a spatially invariant response.
Local connectivity means that only neighboring nodes are connected so there's a lower number of weights to train.
The convolutional layer usually applies a series of filter that lead to multiple feature maps that are then sub-sampled via special operations called pooling.

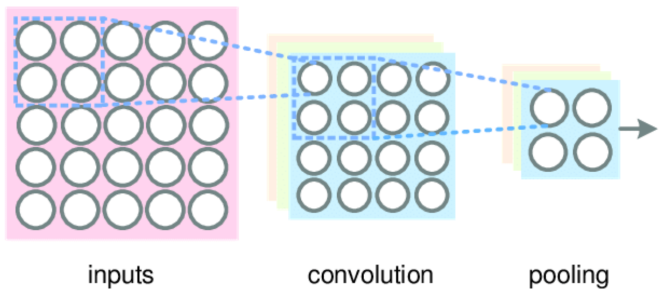


Figure 2.6: Visualization of the convolution and pooling operation performed by a Convolutional Neural Network

Pooling usually chooses between a number of units their maximum or average, this reduces the resolution of the feature map hence reducing the computational complexity of the next layers. In figure 2.6 the convolution and pooling operation can be visualized.

2.2 MobileNet

The work by A. G. Howard, M. Zhu et al. in [7] introduced an efficient class of models for mobile and embedded vision applications.
Convolutional Neural Networks have been made deeper and deeper with more

convolutions and filters in order to increase the accuracy of the predictions. This trend however does not make them more efficient in terms of size and speed making them un-compatible for embedded applications.

Prior works for embedded applications consisted in shrinking the architectures but this often led to significantly lower performances in terms of accuracy. Mobile Net instead allows the developer to choose a small network that specifically matches the resource constraints considering size as well as speed.

Mobile Net is based on Depthwise separable convolutions that factorize a standard convolution into a depthwise convolution and a pointwise $1x1$ one.

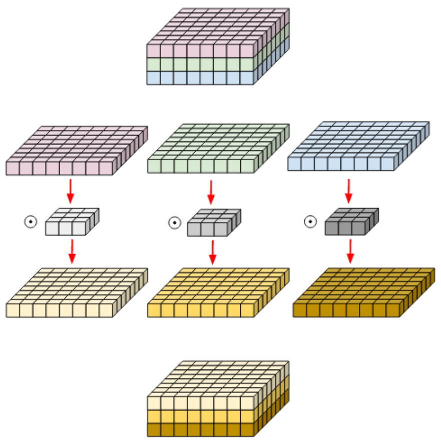


Figure 2.7: Visualization of the operations performed by a Depthwise Convolution

Each channel of the filter of the layer is applied to a single channel of the image separately, the results are then stacked together with the pointwise convolution. As it can be visualized in figure 2.7 a 3 channel image and a 3 channel filter. Both the filter and the image are broken down into separate channels and then the convolution operations is executed separately for each channel.
Finally the results are stacked together producing the output of the convolution.

Standard convolutions take as input a $D_F \times D_F \times M$ feature map and produce as outputs a $D_F \times D_F \times N$ feature map where D_F is assumed to be a squared spatial

dimension and the model is supposed not to shrink.
The convolution is is parametrized as a kernel of size $D_K \times D_K \times M \times N$ where D_K is the spatial dimension of the kernel while M and N are the size of input and output channels.
The computational cost is hence:

$$D_K \cdot D_K \cdot M \cdot N \cdot D_F \cdot D_F$$

MobileNet uses depthwise separable convolutions that break down the interaction between the number of output channels and the size of the kernel.
First of all a $D_K \times D_K \times M$ kernel is applied to each input channel separately, then the results are combined togheter with the pointwise convolution, a linear operation.
The computational cost is hence:

$$D_K \cdot D_K \cdot M \cdot D_F \cdot D_F + M \cdot N \cdot D_F \cdot D_F$$

By analyzing the two results a significant reduction of computational cost can be noticed, indeed:

$$\frac{D_K \cdot D_K \cdot M \cdot D_F \cdot D_F + M \cdot N \cdot D_F \cdot D_F}{D_K \cdot D_K \cdot M \cdot N \cdot D_F \cdot D_F} = \frac{1}{N} + \frac{1}{D_K^2}$$

In figure 2.8 it can be visualized how a standard convolutional filter is separated

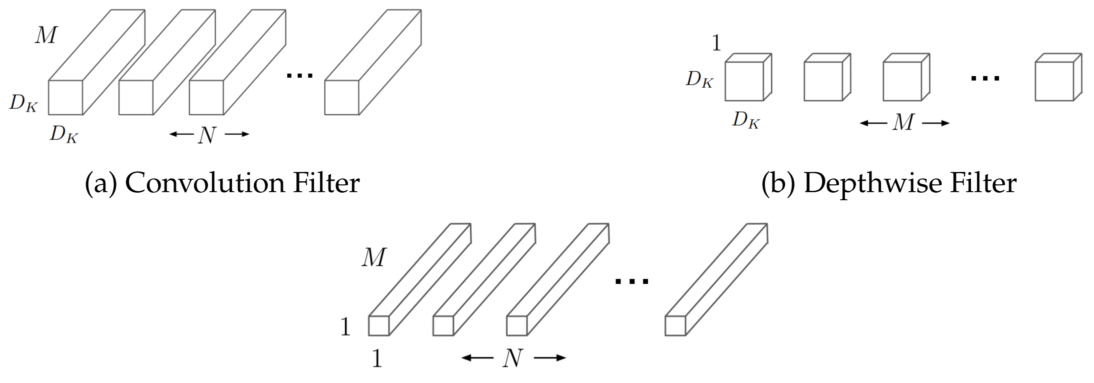


Figure 2.8: Visualization of the size differences between standard convolutions (a) and depthwise separable convolutions in (b) and (c)

into a depthwise and pointwise one.

Furthermore MobileNet introduces two multipliers :

- Width multiplier α; to thin the network uniformly at each layer reducing the channels of the filters (M).
- Resolution Multiplier ρ; to thin the network uniformly at each layer reducing the spatial resolutions (D_F).

The multipliers can further improve the performances of the network by reducing the computational cost of roughly α^2 and ρ^2 respectively.

The authors of MobileNet tested a network using depthwise separable convolutions w.r.t one using full convolutions on the ImageNet dataset for classification.

	Accuracy	Milion Mult./Adds.	Milion Params
MobileNet Conv.	71.7%	4866	29.3
MobileNet	70.6	**569**	**4.2**

Table 2.1: Comparison of fully convolutional MobileNet versus MobileNet with depthwise separable convolutions on the ImageNet dataset

As it can be noticed in table 2.2 the drop in accuracy measured by the author is of only 1% while the number of computations is significantly smaller.

The authors of MobileNet later introduced another architecture, MobileNet v2 in [23].
To further improve the performances of Deep Neural Networks in embedded devices this new architecture uses, in addition to the previously analyzed depthwise separable convolutions, inverted residual and linear bottlenecks.

The scope of a residual block is connecting the beginning and the end of a convolutional block via a skip connection. Traditional Neural Networks have layers that feed directly into the next one, residual blocks instead, allow the first layer of the block to feed also to layers that are some hops away. In this way fewer layers are used in the initial stages of training as they are skipped while later they are gradually restored. This allows the network to avoid problems of Deep Neural Networks such as vanishing gradient that prevents the network to

effectively learn and degradation with its higher accuracy errors.
The traditional residual blocks typically have a structure that goes through a wide, narrow, wide number of channels. Inverted residual instead have a narrow, wide, narrow number of channels leading to fewer overall parameters.

In [6] the authors introduced further improvements to the network mainly by using the technique of network search to optimize each layer.
Neural Architecture Search is a technique that automates the process of designing the network architecture leading to models that often outperform the hand designed ones.

2.3 Viola and Jones

The task of object detection is a challenging problem. There's a huge number of pixels to process and the object could be in any position at multiple scales.

The approach of Viola and Jones firstly evaluates features; a feature-based system has lot less parameters than all the pixel intensities of the image and should, in theory, operate faster in detecting objects. The algorithm uses Haar based features, they are computed using a black and white rectangular filter that is placed at multiple positions and scales over the original image. The Haar feature is then calculated as the difference between the sum of the pixels within the black and the white regions of the rectangle.

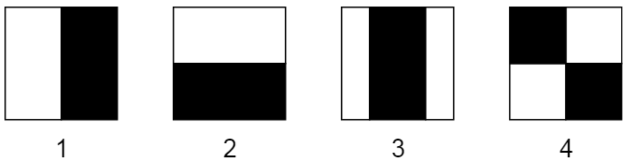


Figure 2.9: Types of rectangles used to evaluate the Haar-based features in the Viola and Jones approach for fast object detection

As it can be noticed in figure 2.9 there are multiple types of rectangles that can produce interesting results when applied to certain parts of the image. As an example, rectangle 3 could be applied on bodies or faces with the white

parts representing the eyes and the black part representing the nose. These features indeed are sensitive to edges, bars and other simple structures the main advantage is that they are computationally efficient, they are however very coarse.
For a fast evaluation of the Haar based features the algorithm uses the integral image that is defined by:

$$ii(x, y) = \sum_{x'<x, y'<y} i(x', y'). \tag{2.1}$$

Where $ii(x, y)$, the integral image contains at every location the sum of all the pixels of and $i(x, y)$, the original one, above and left of such point. Using recurrences, it can be computed in just one pass over the original.

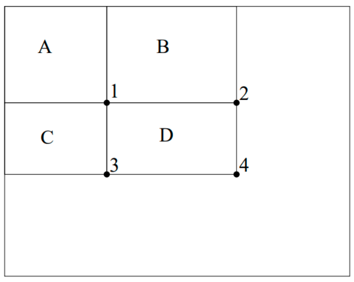


Figure 2.10: Visualization of the integral image approach introduced by Viola and Jones for evaluating rectangular differences of areas of the images

As it can be noticed in figure 2.10 evaluating the Haar features requires a simple operation, indeed the sum of all the pixels in the area A is contained in the pixel at location 1, the one over B is $2 - 1$ and so on.

Given a feature set for each image and a set of positives and negatives examples the algorithm can proceed with the machine learning step. Viola and Jones chose to adopt AdaBoosts, a classification algorithm , to both select a small number of features and train the classifier. The number of rectangles features is indeed huge and, although they are easy to compute this causes an efficiency problem. For this reason the AdaBoosts algorithm selects the single features that best separates the positive and negative examples by setting a threshold on it and produces a series of weak learners. The single weak learner is represented by :

$$h_j(x) = \begin{cases} 1 & \text{if } p_j f_j(x) < p_j \theta_j \\ 0 & \text{otherwise} \end{cases} \tag{2.2}$$

Where f_j is the feature, p_j a parity term and θ_j the threshold selected by the learner. In general, features selected in early rounds have low error rates.

Finally a cascade of classifiers is built to increase the detection performances and reduce the computations. The structure is that of a decision tree, a positive result of the first classifier triggers an evaluation of the second one and so on, while a negative results rejects the sub-window that is being analyzed. So each stage of the cascade acts as a filter and the first one that discards a sample prevents all the subsequent classifiers to work dramatically reducing computations. The first stages should be as fast as possible with a small number of features while going further in the series they should be trained with an increasing number of features. In this way the number of false positives decreases with the number of stages that should be a compromise between the detection rate and the computational time.

The classifiers are trained with an increasing number of features until the target decision rate is reached.
This is done by re-weighting training examples after each stage giving an higher weight to samples wrongly classified in previous stages.

2.4 Hog Detector

The peculiarity of the Histogram Of Gradients (HOG) is that it is able to provide, from an input image, the number of gradient orientations in localized portions of such image.

In order to obtain such results the work by Dalal and Triggs goes trough a series of steps.

First of all it is noticed that, since the feature descriptor has an implicit normalization, image pre-processing can be omitted as it has little to no impact on the final result.

The gradient is then computed both horizontally and vertically using the 1D mask $[-1, 0, 1]$. Several other masks have been tested in [2] but the simplest one proved itself to be the best one for the people detection framework.

The next step is the orientation binning. The image is divided into cells of rectangular or circular shape, each cell produces an histogram of orientations evenly spaced between $0 - 180°$ or $0 - 360°$ (unsigned or signed orientation respectively). Each pixel within the cell produces a weighted vote based on the gradient magnitude and phase for one of the bins of orientations of the histogram.

Most images change a lot from cell to cell with respect to contrast and illumination, this results in very different gradient strengths. Normalization is hence essential for good performances.
The cells are grouped in bigger overlapping blocks, the redundancy caused by having the overlap and hence calculating the contribution of the same pixel more than once is essential in providing better performances.
The blocks can be either rectangular R-HOG or circular C-HOG.
The block normalization can then be computed using one of the four different methods tested by Dalal and Triggs. Considering as v the unnormalized descriptor vector containg all histograms in a given block, $||v||_k$ its $k = 1, 2$ norm and ϵ a small constant :

- L2 norm $v = \frac{v}{\sqrt{||v||_2^2+\epsilon}}$.
- L2 norm followed by clipping and normalizing
- L1 norm $v = \frac{v}{||v||_1+\epsilon}$
- L1 norm followed by its square root

All the normalization provide much better results than not using them and they are all similar in performances results with the exception of the simple L1 norm that performs slightly worse.

The HOG feature descriptor can be visualized in image 2.11 where each of the analyzed cells (in this specific case 8x8) reports the most significant gradient orientation. The Hog descriptors are then combined in a grid and used in a conventional SVM classifier giving the final human detection chain.

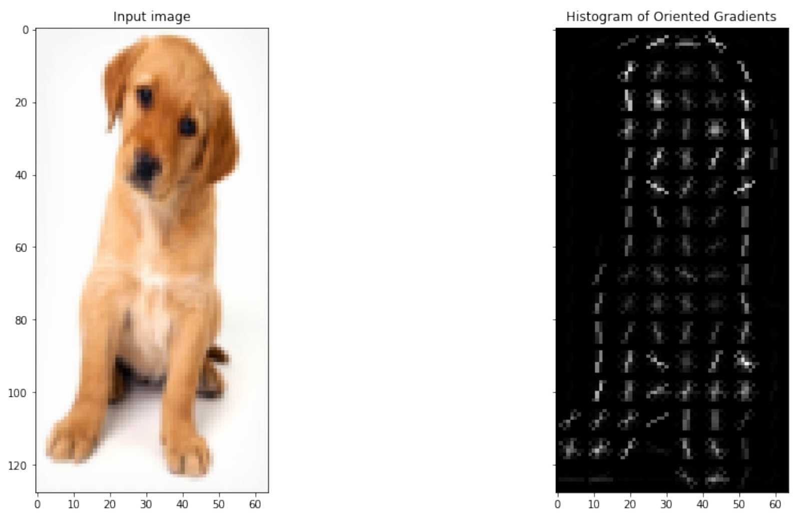


Figure 2.11: Visualization of the histogram of oriented gradients on a simple test image

2.5 R-CNN, Fast R-CNN and Faster R-CNN

In [5] Ross Girshick et al. introduced a new Deep Learning architecture for object detection to improve the performances achieved with detectors based on features like Hog and Sift.
Their paper has been the first to show that a CNN based approach leads to far better performances with respect to the previous feature extraction based ones.

Previous CNN object detection approaches were based on sliding windows or regression.
R-CNN solves the localization problem using the "recognition using regions" paradigm. Each image generates 2000 independent region proposal, fixed length features are then extracted from each region using a CNN and later classified with a linear SVM. The method is called R-CNN since it combines regions with a CNN. In figure 2.12 a brief overview of the method is given.

More in detail :

- Selective search; is the algorithm used to extract significant regions in the image. This method takes as input an image and returns all the patches that due to similarities such as pixel intensities are most likely to contain objects.

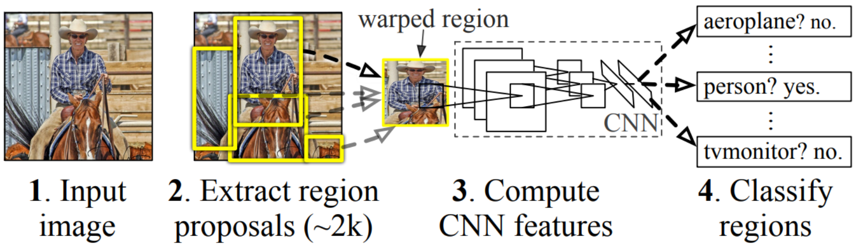


Figure 2.12: Overview of the Regions with CNN features approach

- Feature extraction; for each proposed region a 4096 feature vector is extracted. Each region is resized to a fixed $227x227$ shape and then the features are computed with a network composed of 5 convolutional layers and 2 fully connected one.
- Linear SVMs; there's one for each object category. The feature vector are run through each one of them and, as each of them gives a likelihood score, the one with the highest one will be responsible for classifying the region.

Furthermore the classified regions are refined using a using a class-specific bounding-box regressor that adjusts the coordinates of the bounding box.

This approach gave really meaningful results as it was presented but its main disadvantage was the Selective Search algorithm that bottlenecks the inference times of the approach. Furthermore, each part of the algorithm needs separate training making the method difficult to implement and modify.

R. Girshick in [4] introduced Fast R-CNN addressing the training processes speed as well as the inference time.

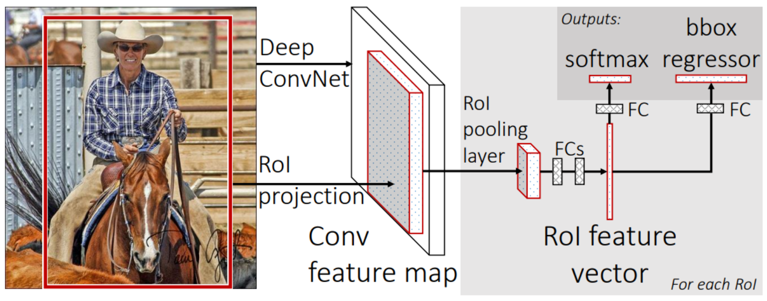


Figure 2.13: Overview of the Fast RCNN approach

The new network, as it can be noticed in figure 2.13, takes as input the entire image and a set of Regions of Interest (RoI) proposal. The image is then processed with several convolutional and max pooling layer to produce a feature map. Then for each RoI only the corresponding part of the feature map is pooled. In this way the convolutions are run only once for all the RoIs. Then a series of fully connected layers produce the softmax probabilities for the classes and separate fully connected layers redefine the position o the proposed bounding box.
The author then introduces a multi-task loss that is able to operate on the two sibling output layers and allows to train the network in a single process making it more elegant with respect to the previous one.

Fast R-CNN achieves near real-time performances when ignoring the time spent on the RoIs proposal. In [22] S. Ren et al. introduced Faster R-CNN proposing a novel Region Proposal Network (RPN). This method has the objective of sharing the computations with Fast R-CNN allowing the whole process to run in real-time. RPN takes as input an image and outputs a series of RoIs proposal using a set of convolutional layers that comprehends layers that are used in Fast R-CNN. Then a small network slides over the feature map of the last shared convolutional layer ultimately generating the proposal.
The training is then performed in 4 steps.

- First step; RPN is trained
- Second step; Fast R-CNN is trained using the RoIs proposed by RPN
- Third step; at this point the networks don't yet share the convolutional layers. Fast R-CNN is initialized to train RPN, the shared convolutional layers are fixed and only the ones unique to RPN are fine tuned.
- Fourth step; the unique layers of Fast R-CNN are fine tuned.

In figure 2.14 the proposed unique detector can be visualized.

Faster R-CNN achieves 5 fps on a GPU allowing to classify it as a real-time (not embedded) object detector.

2.6 SSD, Single Shot MultiBox Detector

The work by W. Liu et al. in [15] proposes a single network that does not re-sample feature maps of pixels based on bounding box hypothesis such as [5],

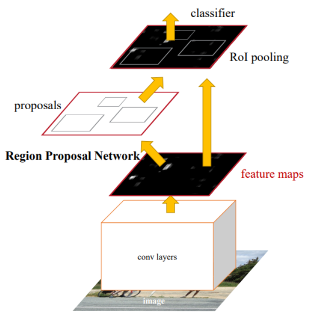


Figure 2.14: Overview of the Faster R-CNN unique detector approach

[4] and [22] making it extremely faster while being as efficient as the methods that do.

The network is a feed-forward and it produces a fixed size of bounding boxes and scores for the presence of a specific class on those object.
Multiple convolutional feature layers are added at the end of the base network in order to allow predictions at different sizes.
The output of each feature layer works like a grid each grid cell produces a prediction with a score and an offset relative to the default bounding box.
In figure 2.15 one of the output feature map can be analyzed. A set of default boxes is predicted for each location as well of the confidence scores for each class.

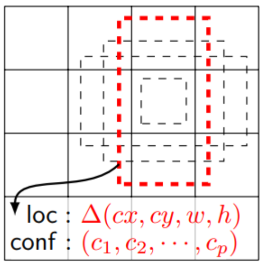

loc : $\Delta(cx, cy, w, h)$
conf : $(c_1, c_2, \cdots, c_p)$

Figure 2.15: Visualization of a 4x4 output feature map of the SSD network

It's important to note that the predictions are performed by different convolutional layers of the network at different sizes, allowing to achieve scale invariance. In figure 2.16 it can be noticed that multiple layers are used as output for the predictions allowing to produce a great number of predictions.

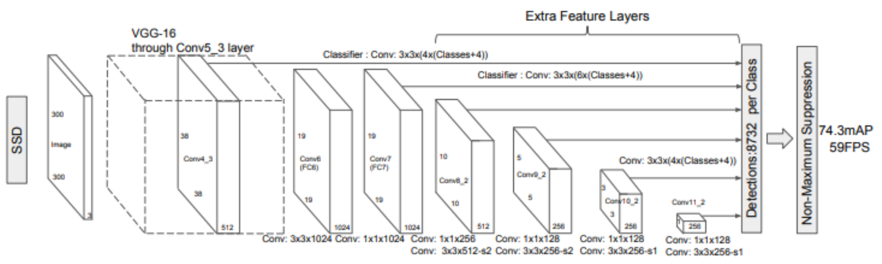


Figure 2.16: Architecture of the SSD network

At training time the ground truth bounding boxes are associated to the closest default one. Then all the default bounding boxes with Jaccard overlap with the ground truth bigger than 0.5 are matched.
This allows the network to predict multiple boxes with high scores rather than predicting only one.

The overall loss used for training is a weighted sum of the localization loss and the confidence loss.

The architecture of the network allows to achieve similar scores relative to the previous architectures used for object detection, this is mostly due to the choice of using feature maps at different layers to produce detections as well as associating the ground truth boxes with more than one of the default ones.
Furthermore using an end to end architecture and treating the detection problem in a regression fashion allows the network to achieve a really high frame rate, 59 FPS using Titan X and cuDNN v4 with Intel Xeon E5-2667v3@3.20GHz.

2.7 Lightweight CNN

The work by Nikouei et al. in [18] proposes a solution for real-time human detection in embedded devices.
The work is inspired by the SSD architecture in [15] and depthwise separable convolutions reported in section 2.2.

The proposed architecture has 26 layers considering depthwise and pointwise convolutions as separate ones. Only the first layer is a fully convolutional one,

downsizing takes places only using the strides of the filters.
Training has been performed by using the Mean Squared Error loss:

$$\frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2$$

This type of loss is one of the most popular losses used for regression problems, hence for problems where there's no need of a classification and only numbers have to be predicted.
This architecture allows to achieve an average of 1.76 FPS on a Raspberry Pi 3 with 1 GB of RAM memory with a rather low 6.6% False Positive Rate.

This results give some useful insights on the results that are achieved when squeezing a network and applying the depthwise separable convolutions.

2.8 Yolo Algorithm

Previous detection systems based on Convolutional Neural Networks re-purposed standard classifier by deploying them as the backbone of a sliding window approach at multiple scales or more recently with R-CNN on proposed regions or Single Shot Detector.
This processes, although they provide good results in terms of accuracy are not straightforward to implement as each element needs separate training and are typically not very efficient.

The You Only Look Once approach of Redmon, Divvala, Girshick and Farhadi offers an alternative solution, object detection is considered as a single regression problem having as input of the CNN the image matrix and obtaining as output a series of vectors containing the objects location and the correspondent class probablities. To do that it performs a series of operations.

The input image is resized to a specific size before passing it to the network, in the original paper by Redmon et al. it's $448x448$. The image is divided by the system into an SxS grid, the cell in which an object center falls is in charge of the detection of such object.

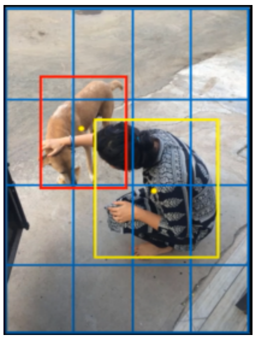


Figure 2.17: Division of the image into an SxS with $S = 4$ grid by the Yolo system

An example can be analyzed in figure 2.17. The image has been divided into a $4x4$ grid. In this image two objects are present, a person and a dog, to each of them a bounding is associated. Assigning to the top left cell the coordinates $(0, 0)$ then cell $(1, 1)$ is responsible for the detection of the dog and cell $2, 2$ for the person. This is because the center of the bounding boxes, represented in figure 2.17 as yellow dots, fall respectively in such cells.

Each cell is thus associated with a prediction and is represented as a vector, for example for the framework of detecting only persons and dogs:

$$\vec{v} = \begin{bmatrix} P_C & B_X & B_Y & B_W & B_H & C_1 & C_2 \end{bmatrix}$$

Where :

- P_C is the class probability that an object is present in such cell, if no object is present it should be equal to zero.
- B_X, B_Y, B_W and B_H are the coordinates of the bounding box associated with such cell.

  B_X and B_Y are the center of the bounding box within the cell in cell coordinates. The cell coordinates are expressed as a matrix having $(0, 0)$ as the top left corner and $(1, 1)$ as the bottom right one.
  So if, for example, the center of the bounding box is exactly as the center of the cell then B_X and B_Y will be both equal to 0.5.

  B_W and B_H are the bounding box width and height associated with the cell represented with respect to the cell dimensions.
  So if, for example a cell has dimension of $52x52$ pixels and the bounding box has dimensions $104x78$ then $B_W = 2$ and $B_H = 1.5$.
- C_1 and C_2 represent the classes. For the person $C_1 = 1$ and $C_2 = 0$ while for the dog $C_1 = 0$ and $C_2 = 1$.

However multiple object centers can fall within the same grid, to tackle this problem smaller cells can be considered and vectors of bigger dimension representing multiple detection can be associated with each cell.
So if $S = 7$, $B = 2$ representing the number of bounding boxes associated with each cell each containing the first 5 elements of $\vec{v}$ and the number of classes is $C = 20$ then the output of the network will be a $SxSx(Bx5 + C) = 7x7x30$ tensor.

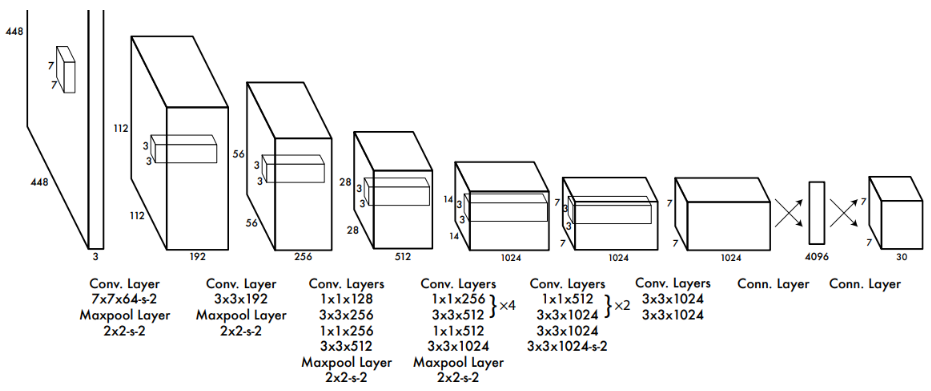

448
7
7
448
3
112
3
3
112
192
56
3
3
56
256
28
3
3
28
512
14
3
3
14
1024
7
3
3
7
1024
7
7
1024
4096
7
7
30
Conv. Layer 7x7x64-s-2 Maxpool Layer 2x2-s-2
Conv. Layer 3x3x192 Maxpool Layer 2x2-s-2
Conv. Layers 1x1x128 3x3x256 1x1x256 3x3x512 Maxpool Layer 2x2-s-2
Conv. Layers 1x1x256 3x3x512 ×4 1x1x512 3x3x1024 Maxpool Layer 2x2-s-2
Conv. Layers 1x1x512 3x3x1024 ×2 3x3x1024 3x3x1024-s-2
Conv. Layers 3x3x1024 3x3x1024
Conn. Layer
Conn. Layer

Figure 2.18: Network architecture of the original Yolo implementation by Redmon et al.

In figure 2.18 the original Yolo architecture can be analyzed. The network deploys 24 convolutional layers followed by 2 fully connected layers and outputs the $7x7x30$ tensor previously mentioned.

Furthermore, during training the following loss function is optimized:

$$\begin{aligned}
&\lambda_{coord} \sum_{i=1}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{i,j}^{obj} [(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2] \\
&+ \lambda_{coord} \sum_{i=1}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{i,j}^{obj} [(\sqrt{w_i} - \sqrt{\hat{w}_i})^2 + (\sqrt{h_i} - \sqrt{\hat{h}_i})^2] + \sum_{i=1}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{i,j}^{obj} [(C_i - \hat{C}_i)^2] \\
&+ \lambda_{noobj} \sum_{i=1}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{i,j}^{noobj} [(C_i - \hat{C}_i)^2] + \sum_{i=1}^{S^2} \mathbb{1}_{i,j}^{obj} [(p_i(C) - \hat{p}_i(c))^2]
\end{aligned} \tag{2.3}$$

Where $\lambda_{coord} = 5$ increases the loss for bounding boxes that contain prediction and $\lambda_{noobj} = 0.5$ decreases the loss for bounding boxes that don't contain any

prediction. This is to avoid an overpower of the cells containing predictions over the ones that don't that leads to instability and model divergence.
Furthermore $\mathbb{1}_{i,j}$ denotes if the object appears on cell i and j represent the the number of the bounding box predictor responsible for such detection.

A series of incremental improvements has been made to the Yolo network by the original authors J. Redmond et al.
In particular Yolo v2 uses higher resolution for training, exploited batch normalization to improve the performances and uses anchor boxes. Anchor boxes are given in a list of predefined boxes that best match the desired objects, the predicted box is then scaled with respect to the anchors.
Yolo v3 instead adds an objectiveness score to bounding boxes predictions and make predictions at three separate levels of granularity to improve performance on smaller objects.
More recent versions of Yolo have then been pursued by different authors making use of new techniques such as losses based on the IoU and adopting state of the art classification CNNs as a backbone.

2.9 Embedded Person Detection

The advancements in embedded hardware and deep learning architectures allowed the research to explore the particular task of detecting persons in resource constrained devices.
The work by Kim et al. in [10] gives a good overview of person detection on embedded devices. Indeed it explores some of the most popular CNNs such as YOLO, SSD and R-CNN. Then it analyzes how well they perform over an in house proprietary dataset and finally it reports how fast they are when deployed on the Nvidia Jetson tx2.
All the models where adapted for the task of person detection, trained and tested over an in house proprietary dataset that considers costumers in stores.

The results reported in table 2.19 show that R-FCN, Faster RCNN and some of the version of YOLO are the most indicated for their particular application. However it has to be noted that their proprietary dataset has been annotated fine tuning the annotations proposed by a version of Faster RCNN and for this reason the results of this particular network probably gain some advantage over

the other ones.
Finding a good CNN for an application that deploys an embedded device is not only a matter of how well they perform in terms of precision, it must also consider a compromise with the inference speed.

#	Model	Framework	AP [IoU=0.95]	AP [IoU=0.50]
1	Faster RCNN (ResNet-101)	Tensorflow	**0.245**	**0.476**
2	YOLOv3-416	Darknet	0.143	0.367
3	Faster RCNN (Inception ResNet-v2)	Tensorflow	**0.317**	**0.557**
4	YOLOv2-608	Darknet	0.198	**0.463**
5	Tiny YOLO-416	Darknet	0.035	0.116
6	SSD (Mobilenet v1)	Tensorflow	0.094	0.233
7	SSD (VGG-300)	Tensorflow	0.148	0.307
8	SSD (VGG-500)	Tensorflow	0.183	0.403
9	R-FCN (ResNet-101)	Tensorflow	**0.246**	**0.486**
10	Tiny YOLO-608	Darknet	0.06	0.185
11	SSD (Inception ResNet-v2)	Tensorflow	0.116	0.267
12	SqueezeDet	Tensorflow	0.003	0.012
13	R-FCN	Tensorflow	0.124	0.319

Figure 2.19: Comparison of the Average Precision for person detection of popular CNNs in the work by Kim et al. in [10]

As it can be noticed in figure 2.20 more efficient versions of the CNNs such as Tiny Yolo are the faster ones. However YOLO v3-416 and SSD (VGG-500) are the best trade-off between average precision and throughput.

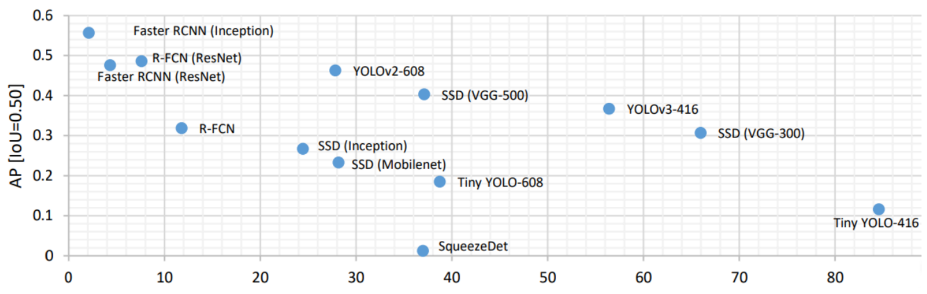


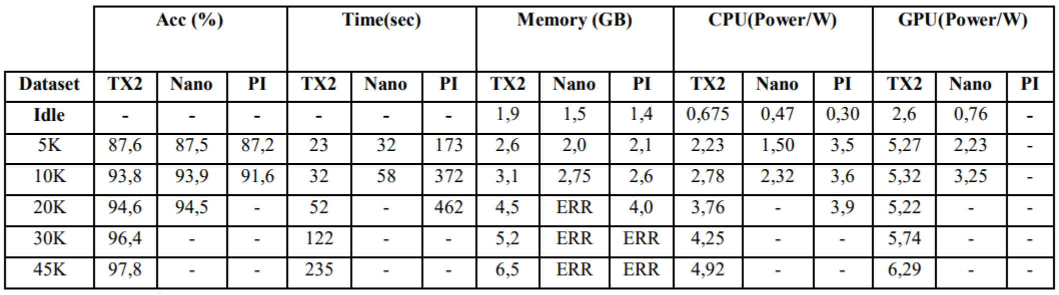
Figure 2.20: Comparison of the Average Precision versus the throughput FPSs performed by popular CNNs on the Jetson tx2 in the work by Kim et al. in [10]

Although the work by kim et al. is similar to the comparison that will be carried out in chapter 3 of this thesis some key differences must be noted.
Kim et al. adapted all the CNNs by training and testing them over an in house proprietary dataset. Indeed their work is particularly interested in a in-store application that is rather different from a work environment. Persons will be less likely to be occluded and the surroundings will have less variability (work environments can also be outdoors and are subject to weather conditions while stores are most likely to be indoor).
Furthermore the device that is used for their tests is a Nvidia Jetsion tx2 that is more indicated for running the parralel computations required by CNNs. The devices that are tested in this thesis instead, are the Raspberry Pi 3 and 4 that without a powerful GPU perform much worse. This is because they are the closest to the radio-commands developed by the company in which this thesis has been developed.

2.10 Convolutional Neural Networks on Embedded Devices

As Convolutional Neural Networks are becoming more efficient and embedded devices more powerful applications that combine both are becoming an interesting area to resarch.
High level GPUs can deploy with small inference times, high performance deep learning CNNs. However, this comes with high expenses and power consumption.

The work by A. Suezen et al. in [26] discuss the implementation of a 13 classes classification CNN on three different single-board computers: NVIDIA Jetson Nano, NVIDIA Jetson TX2 and Raspberry Pi 4.
Although all the devices can be considered embedded as they have limited resources a key difference must be noted.
The two boards developed by NVIDIA have a CUDA capable GPU, this means that such GPU is optimized for performing the big parallel computations required by CNNs.



	Acc (%)			Time(sec)			Memory (GB)			CPU(Power/W)			GPU(Power/W)		
Dataset	TX2	Nano	PI	TX2	Nano	PI	TX2	Nano	PI	TX2	Nano	PI	TX2	Nano	PI
Idle	-	-	-	-	-	-	1,9	1,5	1,4	0,675	0,47	0,30	2,6	0,76	-
5K	87,6	87,5	87,2	23	32	173	2,6	2,0	2,1	2,23	1,50	3,5	5,27	2,23	-
10K	93,8	93,9	91,6	32	58	372	3,1	2,75	2,6	2,78	2,32	3,6	5,32	3,25	-
20K	94,6	94,5	-	52	-	462	4,5	ERR	4,0	3,76	-	3,9	5,22	-	-
30K	96,4	-	-	122	-	-	5,2	ERR	ERR	4,25	-	-	5,74	-	-
45K	97,8	-	-	235	-	-	6,5	ERR	ERR	4,92	-	-	6,29	-	-

Figure 2.21: Benchmarking of the NVIDIA Jetson Nano ,TX2 and the Raspberry Pi 4

Table 2.21 reports the results based on the same classification network trained and tested based on 5 different dataset sizes.
Each dataset has been split in 70% training images and 30% testing ones.
The time refers to the time that is needed to classify all the testing images of the corresponding dataset.

It's clear that having a device equipped with a GPU as the two NVIDIA gives a huge advantage in terms of speed. In each dataset indeed, the Pi 4 is outperformed by one order of magnitude. The results however show that this comes with a cost, although faster performances are obtained the two Jetson require require more power to run, that in a resource constrained environment with limited energy may not be desirable.
Moreover it must be considered that the Pi 4 is typically 10 times less expensive than the TX2 and 3 times less than the Nano.

It's also interesting to notice that the accuracy increases when the training dataset is bigger.

3 Comparison of Different Off the Shelf Person Detectors

In this chapter three approaches to the task of person detection are going to be analyzed and implemented.
These are Viola and Jones, which makes use of a cascade of classifiers trained over Haar based features, a linear SVM detector based on the Histogram of Oriented Gradients features and finally Yolo version 3, a Convolutional Neural Network for object detection.
The aim is to perform a comparison between some popular off-the shelf person detectors, without adapting the code too much and implementing the algorithms as the authors originally intended. Major performance metrics such as precision and recall are going to be compared with a particular interest over the inference times on the Raspberry Pi 3 and 4, two embedded platforms.
The two devices are going to be used as test platforms through the thesis. They have been chosen as they are the ones that mimic the best the performances of the remote control device produced by the company in which this thesis has been developed.

The Pi 4 is equipped with 8 GB of LPDDR4 RAM and Quad core Cortex-A72 CPU, while the Raspberry Pi 3 that has a quad-core ARM Cortex A53 CPU and 1 GB of LPDDR2 RAM. Hence, while both devices are powerful enough to run a light OS, the Pi 4 outperforms the Pi 3.
In figure 3.1, the Pi 4 can be visualized. To its right the Pi Camera v2.1 can also

be observed. The camera will serve as the webcam for all the run-time tests of the algorithms.

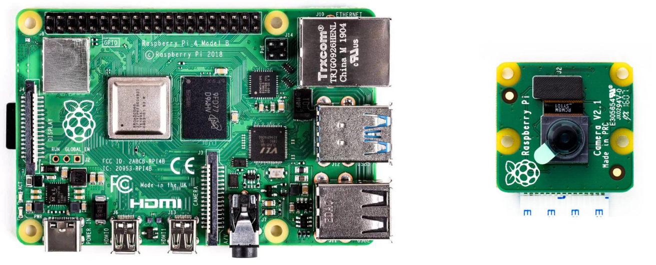


Figure 3.1: The Raspberry Pi 4 (on the right) and the Pi Camera v2.1 (on the left)

The two devices, without a powerful GPU are not indicated for running the big parallel computations required by Convolutional Neural Networks.
This motivates the choice of the three algorithms to test. Viola and Hog were indeed developed in a time where powerful GPUs were not yet available. Their algorithms don't require as much parallelism as the recent CNNs and are more indicated for running on the CPUs of the Raspberry Pi 3 and 4.
Yolo instead, has been chosen to represent the state of the art of CNNs for object detection and to give a meaningful comparison with the selected machine learning based approaches.

3.1 Viola and Jones

A first approach to person detection has been developed through the P. Viola and M. Jones method for rapid detection using a boosted cascade of simple filters introduced in [27].
The main ideas introduced by their paper were :

- Integral Image; a way of representing images that allows fast feature evaluation.

- AdaBoost; a method that selects a small number of features to build a classifier.
- Cascade classifier; combines simple classifiers in a cascade structure to considerably enhance the speed of the detector.

In this section, these methods will be exploited and implemented on OpenCV running on the Raspberry Pi 3 and 4, the results will be later compared with other off-the shelf methods described in this chapter.

3.1.1 Viola and Jones Implementation

In this subsection a general explanation of how the code exploits some of the OpenCV classes in order to detect persons, will be given.
The Viola and Jones approach has been implemented for body detection on the Raspberry Pi 3 and 4. The devices have been programmed using C++ and some libraries from OpenCV, the most interesting one for the scope of the application is the "objdetect.hpp" class.
Viola and Jones has also been implemented in Python on the Google Colab platform in order to perform more testing.

The "CascadeClassifier()" class allows to perform the detection on a gray-scale image. The classifier, trained on a huge number of data by the OpenCV organization has been provided to users in the form of an xml file that can be loaded with a dedicated function.
Two different implementations have been considered, one with just one classifier "haarcascade_fullbody.xml" and one with two additional classifiers "haarcascade_upperbody.xml" and "haarcascade_lowerbody.xml". Indeed the detector should detect bodies at different sizes and location in the image and part of them could also be occluded. For this reason using multiple classifier trained to detect multiple part of the bodies should lead to better results.

A series of modification have then been applied to the analyzed images. Each image has been transformed to the gray-scale single channel domain, their histogram has been equalized in order to enhance the image and produce more significant features.

The function "detectMultiscale()" has subsequently been exploited. This function takes as input the previously elaborated image and produces a template class for 2D rectangles containing the top left corner (x and y), width and height of a rectangle that should contain the detected body.

The code that has been written then proceeds to draw a rectangle in the location of the object if something has been detected.

```
    String body_cascade_name = ".../haarcascade_fullbody.xml";
    body_cascade.load(body_cascade_name);


    Mat frame_gray;
    cvtColor(frame, frame_gray, COLOR_BGR2GRAY);
    equalizeHist(frame_gray,frame_gray);

    std::vector<Rect> bodies;

    body_cascade.detectMultiScale(frame_gray, bodies);

    for(vector<Rect>::iterator i = bodies.begin(); i != bodies.end();
    ++i)
    {
        Rect &r = *i;
        rectangle(frame, r.tl(), r.br(), Scalar(0,255,0), 2);
    }
```

Code 3.1: Viola and Jones single image and single classifier Person detection

A snippet of the code implementation can be examined in code listing 3.1.

Furthermore a Real Time implementation for the target embedded device has been developed. The implementation makes use of the GStreamer class in order to communicate with the Pi Camera v2.1 that has been mounted on the embedded devices. Then each frame produced by the camera is analyzed with the same classes described before and displayed to the user with its results.
The code for this type of implementation can be found on the Github repository mentioned in the introduction.

3.2 Hog Based Person Detector

A second approach using the Histogram of Oriented gradients introduced by Dalal and Triggs in [2] has been considered and implemented. It is based on Hog features combined with a linear Support Vector Machine (SVM) and it presents a valid solution to the problem of person detection. An overview of the chain to detect objects is given in figure 3.2.

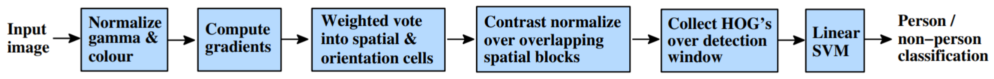

Input image
Normalize gamma & colour
Compute gradients
Weighted vote into spatial & orientation cells
Contrast normalize over overlapping spatial blocks
Collect HOG's over detection window
Linear SVM
Person / non−person classification

Figure 3.2: Overview of the HOG feature extraction and object detection chain introduced by Dalal and Triggs

In this section, this method will be exploited and implemented using the OpenCV library installed on the Raspberry Pi 3 and 4, the results will then be compared with other off-the shelf methods later in this chapter

3.2.1 Hog detector Implementation

The detection of persons using Hog features has been implemented on the Raspberry Pi 3 and 4.
A set of libraries from OpenCV has been exploited, in particular "objdetect.hpp".

The Hog descriptor is initialized with default values, the cell size is set to 8x8, the block size to 16x16 (hence a R-HOG) and the block stride to 8x8.
A linear SVM detector, trained for detecting peoples in images is then set by passing its coefficients to the "HOGDescriptor" class.
Two Support Vector Machines have been tested, both provided by OpenCV. One is the default People Detector that has been trained on windows of dimension 64x128, the other one is the Daimler People Detector, trained over 48x96 windows.

The "detectMultiScale" function is then able to perform the detection based on the HOG features and the linear SVM that has been previously set. In particular it returns a vector of Rect as in subchapter 3.1.1, containing all the locations of the identified objects accompanied by their confidence level. The hit threshold defines the distance between features and the SVM classifying plane, in this framework it has been set to 0. The window stride that must be a multiple of the block stride has been set to 8x8. The scale defines the coefficient of the detection window increase, its setting is a compromise between performances in terms of detection rate and number of computations. Finally the group threshold avoids overlapping rectangles by grouping similar ones.

```
    HOGDescriptor hog;
    hog.setSVMDetector(HOGDescriptor::getDefaultPeopleDetector());
    vector<Rect> bodies;

    hog.detectMultiScale(img, bodies, 0, Size(8,8), Size(), 1.05, 2,
   false);

    for(vector<Rect>::iterator i = bodies.begin(); i != bodies.end();
    ++i)
    {   Rect &r = *i;
        rectangle(image, r.tl(), r.br(), Scalar(0,255,0), 2);
    }
```

Code 3.2: Hog based person detector implementation

Some of the code used for the tests can be seen in code snippet 3.2, as it can be noticed it is really similar to the Viola and Jones implementation in code snippet 3.1, this is because both methods are derived from the OpenCV "objdetect.hpp" class.

With some initial testing it has been noticed that the Daimler people detector performs much worse than the Default one. This is because it uses smaller windows that lead to more computations and more false positives.
For this reasons in the results analysis only the Default People Detector has been considered.

Furthermore, as for the Viola and Jones algorithm, a Real Time implementation for the target embedded device has been developed. The implementation makes use of the GStreamer class in order to communicate with the Pi Camera v2.1 that has been mounted on the Raspberry Pi 3 and 4. Then each frame produced by the camera is analyzed with the same classes described before and displayed to the user with its results.
The code for this type of implementation can be found again on the Github repository mentioned in the introduction.

3.3 Yolo Object Detection

More recently the research for object detection and image classification has been developed over Convolutional Neural Networks.
The main difference with a Machine Learning based approach is that CNN don't use hand crafted features such as Haar for Viola and Jones or Hog, instead they perform an end to end learning that means that they learn the best feature representation that fits the problem.

Convolutional Neural Networks differ from classical Deep Neural Networks because they consider the input image as a two dimensional matrix where spatial neighborhood is important and interesting features to be learned are local. For this reason they deploy the concept of local connectivity meaning that for each pixel there's a receptive field, only neighboring nodes are connected and the convolutional filters share the same weights. Using this concept CNN achieve a spatially invariant response. Furthermore some polling layers are used throught the network to subsample the image.

In this chapter You Only Look Once, a popular state of the art CNN, first introduced by J. Redmond in [21] is going to be analyzed. It addresses the task of object detection by using a regression approach.
Its methods will be exploited and implemented on OpenCV running on the Raspberry Pi 3 and 4, the results will be then compared with other off-the shelf methods described in this chapter.

3.3.1 Yolo Implementation

The detection of people using Yolo has been implemented using OpenCV on the Raspberry Pi 3 and 4.
The scope of this subsection has been to perform a straightforward, off the shelf implementation meaning that the network has not been re-trained and default config files and weights have been downloaded from the author website [1]. No adaptation concerned only with the task of person detection has been found. The original implementation is indeed capable of detecting 80 different classes,

[1]https://pjreddie.com/darknet/yolo

one of which is person.
Yolo v3 has been chosen as a step by step guide based on the Keras framework has been found in a github repository [2]. This is because Keras is the chosen framework for developing a custom solution to the problem of this thesis.

The OpenCV "dnn.hpp" library has been exploited, with its methods it allows to load the configuration files and trained weights into a "Net" class.
The input image then needs to be adapted for the network, for this reason the "blobFromImage" function converts the image values in format $[0, 1]$ from $[0, 255]$ and scales them down to a $416x416$ tensor.

The image is then passed to the network and the output has to be post processed. Only bounding boxes associated with a probability higher than a certain threshold will be maintained.
Furthermore, non maxima suppression is performed in order to remove overlapping boxes.

```
Net net = readNetFromDarknet(modelConfiguration, modelWeights);
net.setPreferableBackend(DNN_BACKEND_OPENCV);
net.setPreferableTarget(DNN_TARGET_CPU);

blobFromImage(frame, blob, 1 / 255.0, cv::Size(416, 416), Scalar(0,
    0, 0), true, false);

net.setInput(blob);

vector<Mat> outs;
net.forward(outs, getOutputsNames(net));

postprocess(frame, outs);
```

Code 3.3: Yolo object detector implementation

As it can be noticed in code snippet 3.3 the classes of OpenCV easily allow to load the weights of the network, however the output tensor needs to be post-processed in order to retrieve the required data.

[2]https://github.com/qqwweee/keras-yolo3

In appendix code snippet A.1 the post-processing can be analyzed.
For each cell the bounding boxes are retrieved and only the ones with high confidence score are maintained. The box class label is assigned from the class with the highest score.
Redundant bounding boxes with lower confidence are hence removed with the OpenCV "NMSBoxes" function.

3.4 Results

In this section the results of the implementation of the Viola and Jones, Hog and Yolo person detectors will be analyzed and compared.

Popular metrics in evaluating object detectors such as Precision and Recall are going to be exploited over the Coco 2017 dataset.
The metrics are going to be discussed in details with a particular interest over the time performances on the test embedded platforms, the Raspberry Pi 3 and 4.

All the code written to perform this evaluations can be found on the Jupiter Notebooks "Stat_evaluation.ipynb" and "Scale_Stat_Evaluation.ipynb" on the Github repository mentioned in the introduction.

3.4.1 The Coco 2017 Dataset

The evaluation has been performed over images and ground truth bounding boxes from the Microsoft COCO 2017 dataset.

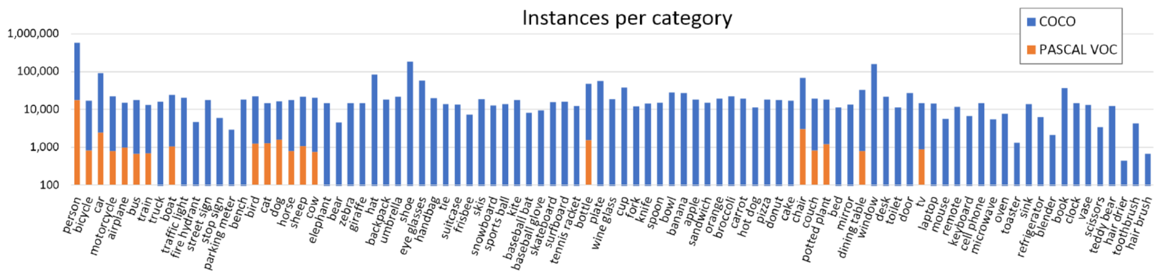
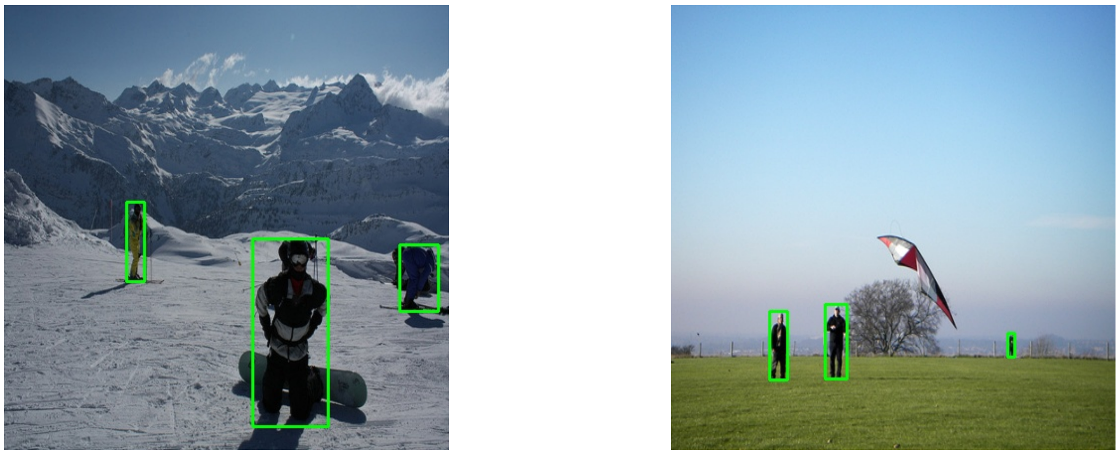


Figure 3.3: Number of annotated instances per category for Coco and Pascal datasets

COCO is a large-scale dataset, first introduced in [14], addressing three core research problems in scene understanding : detecting non iconic views of objects, contextual reasoning and precise 2D localization.
In particular, for the scope of this thesis, the non iconic view has been considered as the main advantage of using COCO over different datasets. Indeed the analyzed methods are considered over a real world industry setting where persons could be occluded by large objects and may have different poses. Detecting those persons, in a safety application, is as crucial as detecting the ones non occluded and in canonical poses.

In figure 3.3 an overview of the number of annotated instances per category of Coco is given and compared to the popular Pascal VOC dataset, it can be noticed how the number of instances for every category and in particular for the "Person" category is far superior.
Coco has several features such as object segmentation and recognition in context, it has more than 330K images (of which more than 200K labeled), 80 object categories and 1.5 million object instances. In this discussion the object category "Person" and the Coco feature concerning its bounding boxes are going to be exploited.
A test dataset with only images, of different size, containing "Person" has been



(a) (b)

Figure 3.4: Some example images and their ground truths of the Coco dataset regarding the "Person" class, the true bounding boxes are drawn over the images as green rectangles.

downloaded from the open source tool Fiftyone. In figure 3.4 some examples can be visualized.
The download genereates a json file containing all the bounding boxes.
In particular the file contains not only the "Person" bounding boxes but also all the others that are present in the downloaded image, so some filtering had to be performed in order to retrieve only the needed data.

Furthermore, images not containing persons have been downloaded with Fiftyone from Coco and added to the test dataset. This has been done in order to get a more accurate evaluation.

3.4.2 Performance Metrics

In order to compare the results, it is useful to report the metrics used in this thesis.
In an object detection task, the object has to be identified and the coordinates of the bounding box around it have to be compared with the ground truth.

In order to perform this comparison the concept of Intersection over Union (IoU) is often applied.

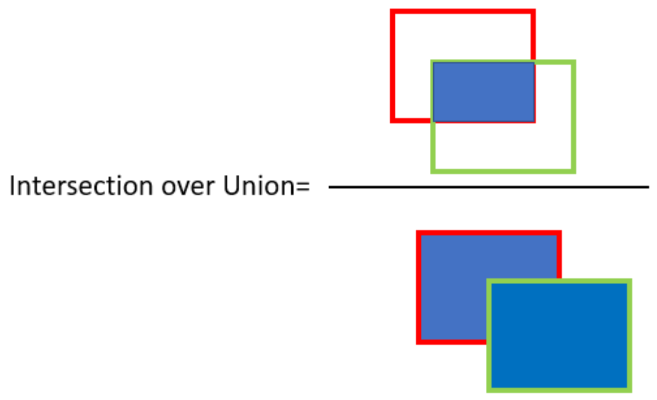


Figure 3.5: Graphic visualization of the Intersection over Union

Consider the predicted bounding box and the one corresponding to the ground truth. Then, as it can be visualized in figure 3.5 the IoU is considered as the area

in which the bounding boxes overlap and the total area of the union.
Using the IoU as a threshold value, for example $IoU = 0.5$:

- $IoU \geq 0.5$, then the predicted bounding box is classified as a True Positive (TP).
- $IoU < 0.5$, then the predicted bounding box is classified as a False Positive (FP) hence the prediction is considered wrong.
- If a ground truth box is present in the image and the model failed to detect it then it is classified as a False Negative (FN).

With this three classes (TP, FP and FN), Precision and Recall can be defined to measure the performances of a model :

$$Precision = \frac{TP}{TP + FP} \qquad Recall = \frac{TP}{TP + FN} \tag{3.1}$$

Precision gives an indication of how good the model is at predicting the positive class when the actual outcome is positive, indeed the number of missed predictions (False Negatives) is not considered in the equation.
Recall instead considers how good the model is at predicting the positive class in general considering also the missed predictions.

Furthermore the F1 score can be considered, giving an armonic mean between Precision and Recall it's an indication of how the two values vary in relation with the other:

$$F1\,score = 2 * \frac{Precision * Recall}{Precision + Recall} \tag{3.2}$$

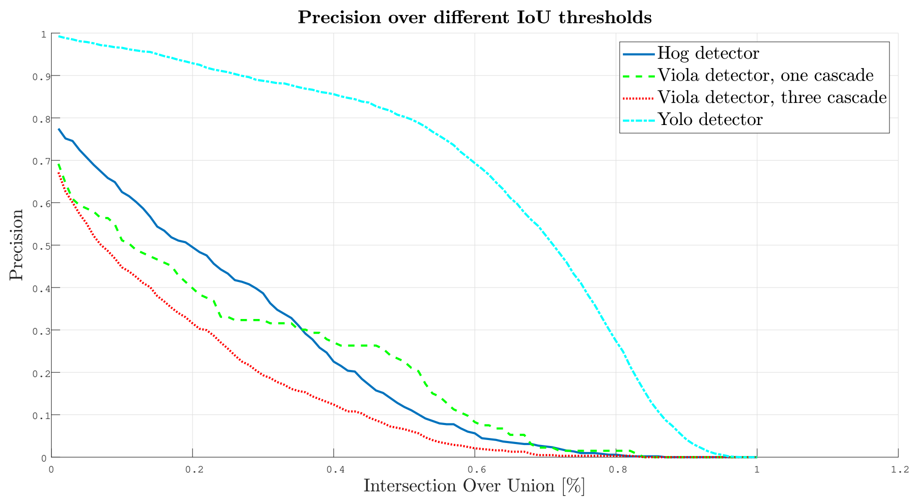
The total number of the three classes over the test dataset will be evaluated considering an increasing Intersection over Union threshold, then Precision, Recall and F1 score are going to be computed considering those numbers.

Furthermore, as the scope of this thesis is considering an embedded application inference times on the Raspberry Pi 3 and 4 are going to be considered observing how they evolve over decreasing input scales. Precision, Recall and F1 score are also going to be computed considering a decreasing input image scale.

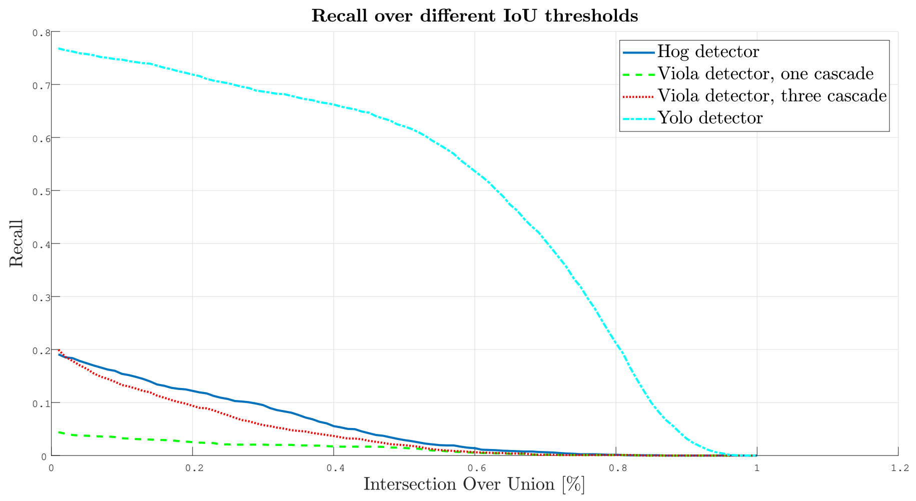
3.4.3 Results Based On Different IoU Thresholds

A first evaluation can be performed over the Precision, Recall and F1 score over different IoU thresholds considering the three models described and im-

plemented in this chapter.
The Viola and Jones method will consider two implementations, one with only the full body cascade classifier and one with three classifiers comprehending also the upper and lower body ones in order to obtain better results.



(a)



(b)

Figure 3.6: Comparison of the different Person detector methods described in this chapter. In (a) the Precision over different IoU is given. In (b) the Recall over IoU.

As it can be noticed in figure 3.6 (a) the precision of the method based on Yolo performs better especially when increasing the IoU threshold, while the others have similar performances. Thus Yolo gives more accurate predictions and is generally better at producing the correct bounding box.

In figure 3.6 (b) instead the superior performances of Yolo can be visualized with the Recall metric. Yolo, when the IoU threshold is not too high misses always less than 50% of the persons while the other methods have at least 50% worse performances.

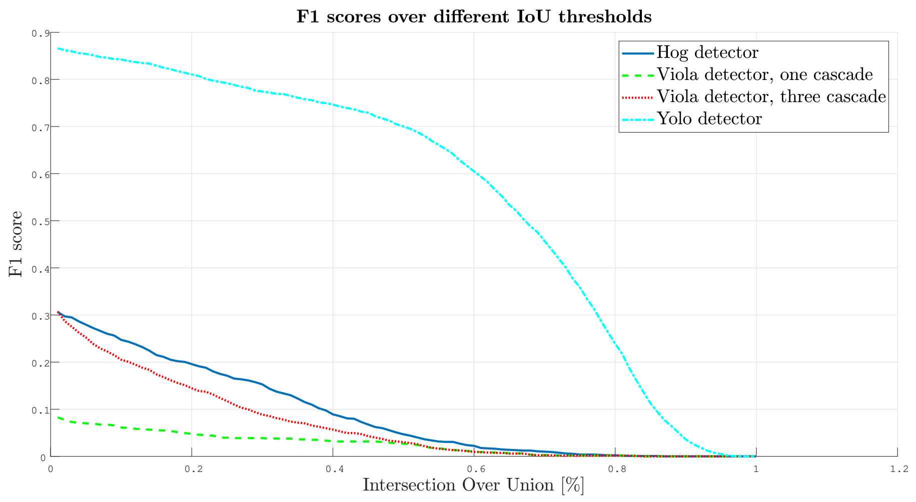


Figure 3.7: Comparison of the F1 score for the different person detectors analyzed in this chapter over different IoU thresholds

The F1 scores, as shown in figure 3.7, giving an armonic average of the two metrics in figure 3.6, confirms that the Yolo detector is the one that performs better.

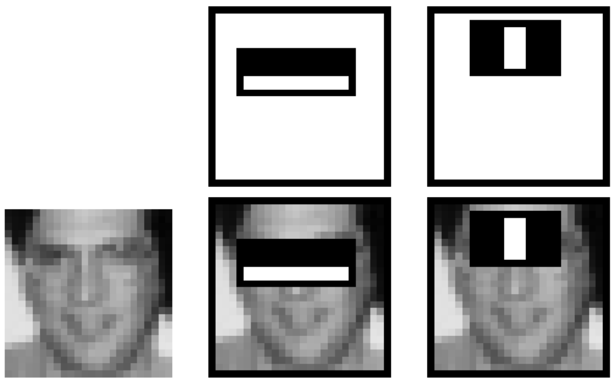


Figure 3.8: Visualization of the first two selected feature by AdaBoost in the Viola and Jones algorithm

It can be considered that the Yolo off-the shelf implementation used in this evaluation has been trained with the scope of detecting 80 classes and not only persons. This means that the network knows how to detect other objects resulting in less false positives.
It can also be noticed that the Hog features are more suitable for the detection of persons with respect to the Haar features used by Viola and Jones even when three classifiers trained for different body parts are used. This is because the Haar features are historically employed for the task of face detection and are not that powerful when they are applied over bodies. Indeed, as it can be noticed in image 3.8 particular Haar features return interesting results when applied over parts like eye-nose-eye or eyes-cheeks while for the whole body this type of results are, in general, not obtained.
The performances of Hog and Viola are arguably not suitable for an aid safety

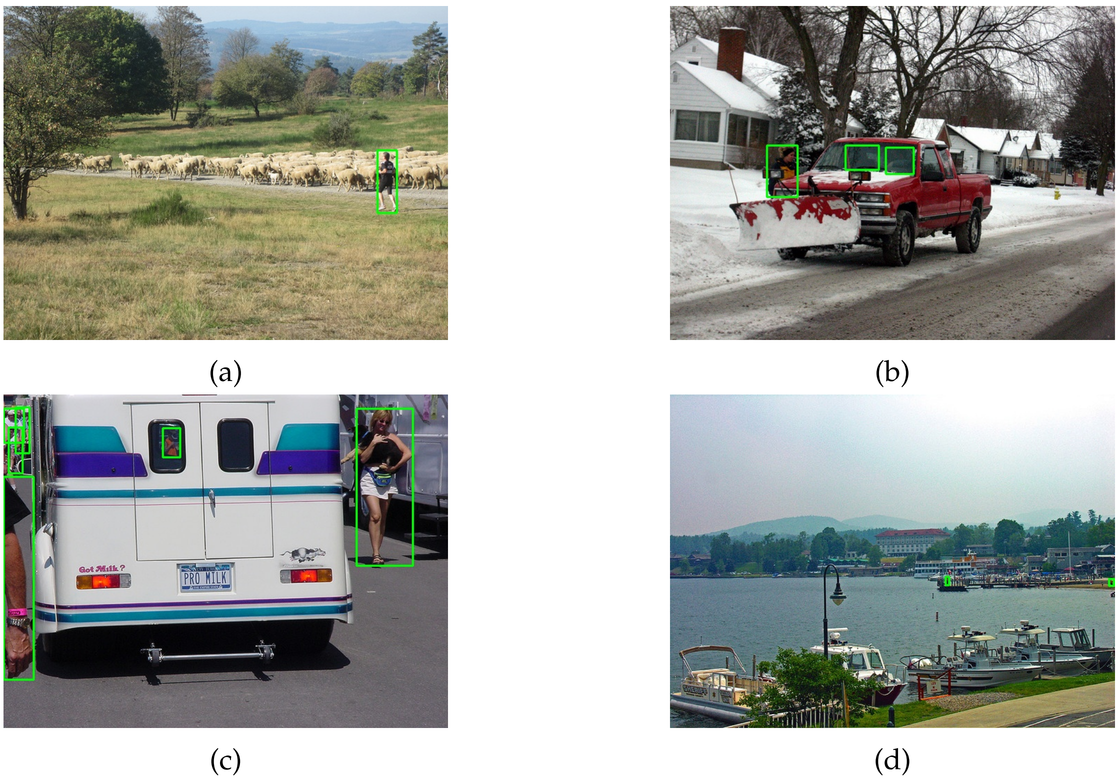


Figure 3.9: Some challenging ground truths of the Coco dataset regarding the "Person" class, the true bounding boxes are drawn over the images as green rectangles.

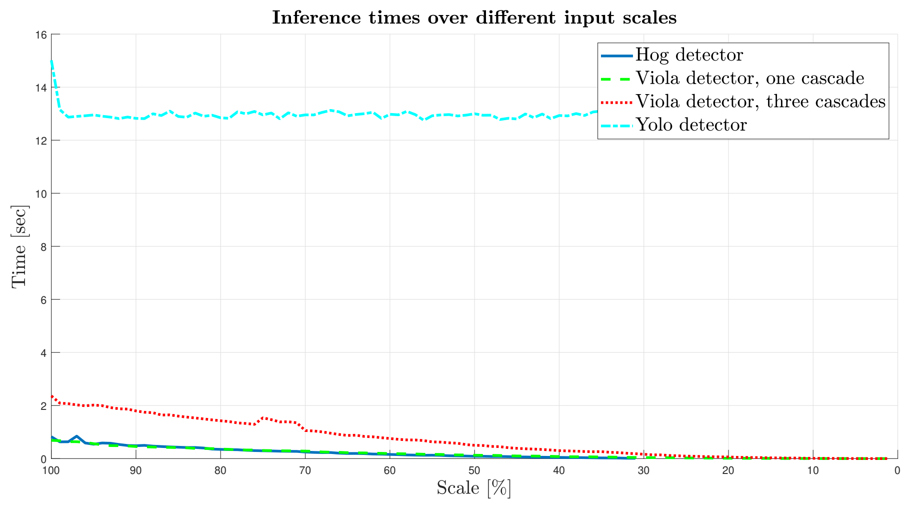
application where it's important to miss little to no persons, however it needs to be considered that the Coco dataset used to produce this results classifies per-

sons in non canonical poses, at multiple scales, in the background of the images and occluded by one or more objects. For this reason it is very challenging to achieve an high score over the test dataset.
Some example of uneasy test images can be visualized in figure 3.9. Here the ground truths obtained by Coco are displayed as green bounding boxes over the persons present in the images.
It can be noticed that significantly occluded persons are considered like the ones behind the front glass of a car in figure 3.9 (b). Body parts are also considered as truths like the arm in the left of figure 3.9 (c). Moreover persons in the far background are also examined like in figure 3.9 (d).
Figure 3.9 (a) is reported to display a full body picture that is not considered challenging as the others.

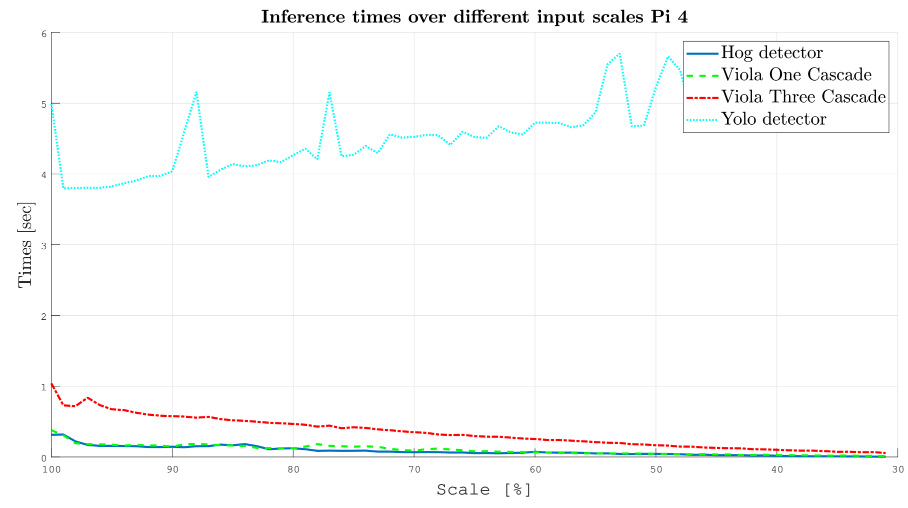
3.4.4 Results Based on Different Input Scales

An additional test has been performed on the target embedded devices.
The inference times were computed for all the models considering an iterative decreasing size of the input image. The test has been performed with C++ code that feeds to the classifiers or the Neural Network the same input image over and over after applying to it a scale factor that decreases its size every time by 0.1 ($new\ scale = old\ scale - 0.1$) with a linear interpolation as the default OpenCV method.
The input image considered has an original size of $640x480$ as for the VGA standard often used in cameras applied over a work environment. As for this test only the inference times are meaningful a single random image has been chosen.

Analyzing the results in figure 3.10 important consideration can be made.
The inference time of Yolo is considerably bigger than the others and does not decrease with different scales. This is because the input size of the network is set in Yolo v3 to $416x416$, so each image needs to be scaled to that dimension.
Furthermore the inference time of Yolo is really big, indeed the network with its 52 convolutional layers and 40549216 parameters requires a huge number of operations to be performed by the CPU.
This type of network indeed provides a guarantee for the detection accuracy



(a)



(b)

Figure 3.10: Comparison of inference times of the different models based on the input size of the image. In (a) the times on the Raspberry Pi 3. In (b) the times on the Pi 4.

in multiple categories applications. The considered network is indeed able to detect more than 80 categories while in this application only the "Person" detections are considered.

The other detectors instead perform better and the inference time linearly decreases with a decreasing scale. In particular it can be noticed that the Viola detector that uses three classifiers and hence infers the same image three times at every scales increases its inference time more or less by 3 times over the standard

Viola detector.

Furthermore a significant improvement in performances can be noticed in figure 3.10 (b). This indicates how the better hardware of the Pi 4, even if it's still an embedded device, can provide faster computations.

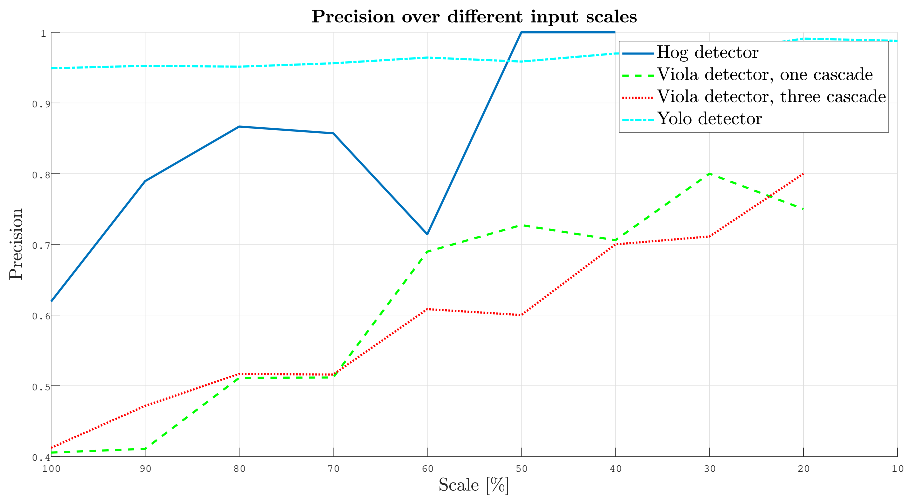
It has been analyzed how decreasing the input size of the image decreases the inference time in all the detectors but Yolo.
It's then interesting to evaluate the performance metrics analyzed in sub-chapter 3.4.3 for input images of decreasing sizes.

In order to perform this comparison 300 images of size $640x480$ as for the VGA standard have been downloaded from COCO and Precision, Recall and F1 score have been evaluated with a decreasing scale with a fixed IoU threshold.

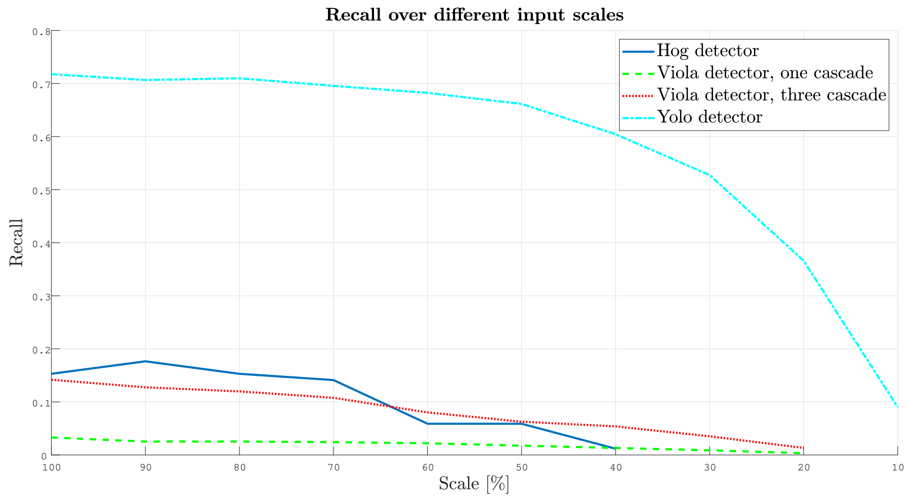
The IoU threshold in the differentiation of TP, FP and FN classifies the correct and incorrect predictions, the bigger it is the more accurate to the ground truth the predictions are, but less number of objects are going to be considered as true positives.
The literature usually sets $IoU = 0.5$, however by looking at figure 3.6 it can be noticed that too many persons will be missed. In a safety application it's more important that the system tells that there's a person instead of missing it, for this reason the IoU threshold has been set to an arguably low $IoU = 0.1$. To perform this evaluation the ground truth bounding boxes had to be scaled down in order to be compared with the predicted ones.

As it can be noticed in figure 3.11 (b) and further considered in figure 3.12 the performances for F1 score and Recall decrease for all the methods as expected. Giving as input images with lower resolution means that more and more features are lost and the predictions resent it producing a low number of more inaccurate predictions.

However the precision increases for all the methods when decreasing the scale. Precision is indeed a measurement of how good the model is at predicting the



(a)



(b)

Figure 3.11: Comparison of the different Person detector methods described in this chapter. In (a) the Precision over different scales is given. In (b) the Recall over scales.

positive class without taking into account the number of misses (FN), when the resolution of the input image is decreased less pixels are taken into account but they represent more relevant aspects of the image. Structures visible at coarse spaces are simplification of the strongest elements in the image.
For this reason the models are less likely to produce false positive samples that are often triggered by small local maxima or minima present in the image at bigger scales.
Additionally with less pixels in the image the predicted and true bounding boxes

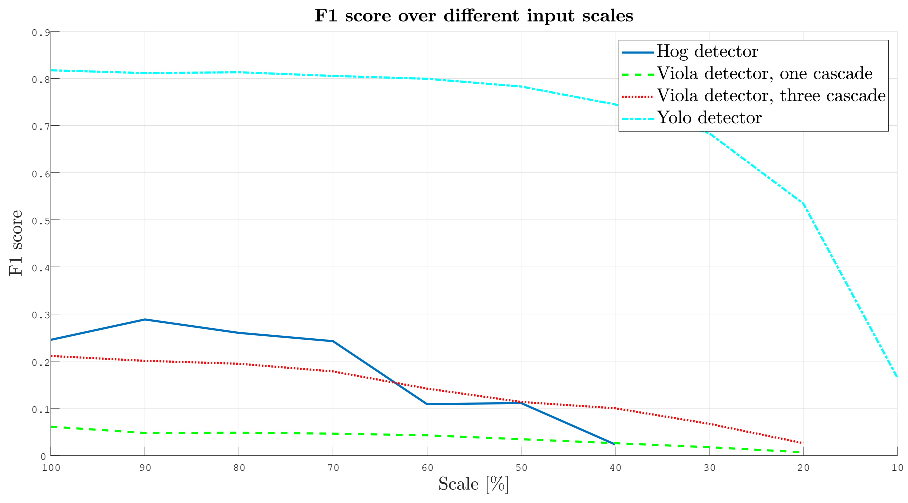


Figure 3.12: Comparison of F1 scores of the different models based on the input size of the image

have less variability in terms of their positioning leading again to less overall false positives.

Furthermore it can be noticed that the Yolo detector maintains its supremacy over the other feature-based methods as it was previously noticed in figures 3.6 and 3.7. Hence, clearly the performances of Yolo remain the best ones, however decreasing the input size, as previously stated, does not lead to better performances in terms of inference times because every image needs to be resized to a size of $416x416$ before being passed to the CNN.

3.4.5 Results for a Live Application

With the performances analyzed in sub-chapter 3.4.4 in mind a live application can be considered for the feature based detectors on the Raspberry Pi 3 and 4. As previously explained Yolo inference times are too high for the target embedded devices and is thus not considered.
The IoU threshold has been set equal to $IoU = 0.1$ to classify correct and incorrect predictions in order to miss as little persons as possible with the same reasoning of sub-chapter 3.4.4. A target performance on a video stream input of 10 fps has been considered.

In order to consider the performances an inference time of $t = \frac{1}{10} \approx 0.1sec$

has been set on figure 3.10 and for each detector the corresponding input image scale has been obtained.
Then, with the scales needed to obtain the desired performances in terms of fps the corresponding Precision, Recall and F1 score have been obtained from figures 3.11 and 3.12.

	FPS	Scale	Precision	Recall	F1 score
Hog detector	10 fps	**55%**	**95%**	**7%**	**11%**
Viola one cascade	10 fps	50%	72%	2%	3%
Viola three cascades	10 fps	30%	71%	4%	7%

Table 3.1: Comparison of different feature based detectors performances considering a target performance of 10 Fps and data reported in figures 3.10 (a), 3.11, 3.12 for the Raspberry Pi 3

	FPS	Scale	Precision	Recall	F1 score
Hog detector	10 fps	**75%**	**87%**	**16%**	**26%**
Viola one cascade	10 fps	70%	52%	2%	5%
Viola three cascades	10 fps	40%	78%	6%	10%

Table 3.2: Comparison of different feature based detectors performances considering a target performance of 10 Fps and data reported in figures 3.10 (b), 3.11, 3.12 for the Raspberry Pi 4

As it can be noticed by looking at table 3.1 and 3.2 the more viable option for a Real-time application is the Hog detector although it does not provide really good results.
It can indeed be noticed that the performances in term of Recall and F1 score are very poor, however, as previously stated, the Coco dataset provides a very challenging test benchmark and results in a real world application would certainly be more satisfying.

Furthermore, it can be remarked how a better hardware, such as the Raspberry Pi 4, leads to far better results. Requiring a device with better specification leads to better results and a more deployable solution. However this comes with more expensive devices that may not be considered in a large scale application.

To conclude, the results are not satisfying in most cases for a live Person detection application.

The Yolo detector has been proven to lead to far better results in terms of detection however the Raspberry devices without a GPU are not capable of performing efficiently the computations required to infer a stream of images through the network.
For this reason the focus of the thesis will now shift in the custom development of a light weight version of Yolo for detecting the single class "Person" on an Embedded Device.

4 Custom Architectures for Person Detection

In this chapter some custom architectures based on Convolutional Neural Networks and Yolo are going to be explored and implemented for the task of person detection.
The developments will concentrate on finding a light-weight architecture suitable for a live application on embedded devices without a GPU.

The performance metrics discussed in the previous chapter are going to be considered and results of the evaluation are going to be compared with the off-the shelf methods analyzed in chapter 3.

4.1 Architecture Redesign

The original Yolo architecture was designed to detect with a good accuracy objects belonging to 20 different categories.
Considering that the application of this thesis requires to detect only a single class (i.e. "Person") some modifications can be made to the original network to reduce the number of parameter and computations.

The main things that have been considered are reducing the output size of the network and removing some of its layers. This reduction is motivated by the fact that the network does not need to detect 20 different classes hence it should not need to perform as many computations.

The original paper considers an input rgb image of $448x448$. The output is encoded as $7x7x30$ tensor.
This means that the image is divided into a $7x7$ grid and each cell of such grid outputs a vector of length 30 composed by :

- 2 bounding boxes having the Yolo format as discussed in the previous chapter
$$\vec{B} = [P_C \quad B_X \quad B_Y \quad B_W \quad B_H]$$
- 20 entries, one for each class.

The scope of this thesis is predicting a single class hence the output has been encoded as a $7x7x5$ tensor where each cell has associated a vector of length 5 that is equal to a single $\vec{B}$.
This means that the architecture will predict a single bounding box per cell and there's no need of adding to the vector the classes since there's only one and each prediction will belong to the "Person" class.

With this considerations in mind many new architectures were designed and tested following a trial and error approach changing the number of filters, the number of convolutional layers, the type of pooling and many other parameters. The three more relevant ones are reported, each one accepts as input an image of size $448x448x3$ where 3 are the channels of the image. These architectures will allow to make some considerations that will later serve as a basis to make some improvements to the network ideas.

The first reported architecture is "Yolo Small", it has been designed by shrinking as much as possible the original Yolo architecture, indeed it has significantly less layers and a small overall number of filters. This new architecture only has 6 convolutional layers followed by a fully connected one versus the original architecture that has 24 convolutional layers followed by 2 fully connected one.

Yolo small reduces the input resolution of the original image via max-pooling layers and some convolutional ones (with stride $s = 2$) and finally produces the prediction via a fully connected layer.

Yolo Small		
Layer type	Filters/Filter size	Output shape
Conv2D	8/(3x3)	448x448x8
MaxPooling2D	(2x2)s2	224x224x8
Conv2D	16/(3x3)	224x224x16
Conv2D	8/(1x1)	224x224x8
Conv2D	16/(3x3)s2	112x112x16
MaxPooling2D	(2x2)s2	56x56x16
Conv2D	32/(3x3)	56x56x32
MaxPooling2D	(2x2)s2	28x28x32
Conv2D	64/(3x3)s2	14x14x64
MaxPooling2D	(2x2)s2	7x7x64
Reshape	-	3136
Dense	-	245
Reshape	-	7x7x5

Table 4.1: Overview of the Sequential layers adopted by the Yolo Small architecture

Each convolutional layer is followed by batch-normalization that makes training faster and more stable by re-centering and re-scaling the outputs of the layers. Furthermore every convolutional layer is followed by the leaky relu activation function defined as :

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ 0.1 & \text{otherwise} \end{cases} \tag{4.1}$$

This activation function is the same adopted by the original Yolo algorithm and it was introduced in order to solve the dying neuron problem where neurons

that produced large negative values remained stuck at zero. Indeed relu sets all the negative values to zero while leaky relu partially mantains them with a small slope.
The last fully connected layer uses a sigmoid activation function, this is because all of the output values of the $7x7x5$ tensor are expected to have values in range $[0, 1]$ as it will be better explained in the next chapters.

Yolo Big		
Layer type	Filters/Filter size	Output shape
Conv2D	32/(3x3)s2	224x224x32
MaxPooling2D	(2x2)s2	112x112x32
Conv2D	/64/(3x3)	112x112x64
MaxPooling2D	(2x2)s2	56x56x64
Conv2D	128/(3x3)	56x56x128
Conv2D	64/(1x1)	56x56x64
Conv2D	128/(3x3)	56x56x128
MaxPooling2D	(2x2)s2	28x28x128
Conv2D	128/(3x3)s2	14x14x128
MaxPooling2D	(2x2)s2	7x7x128
Reshape	-	6272
Dense	-	245
Reshape	-	7x7x5

Table 4.2: Overview of the Sequential layers adopted by the Yolo Big architecture

The second reported architecture is "Yolo Big".
The aim of this architecture is still to produce a light-weight version of Yolo but

without a dramatical decrease of some of the parameters such as the number of filters as in Yolo Small.
However the network remains similar as there are still 6 convolutional layers followed by a fully connected and the same activation functions are used as well as batch-normalization.

Yolo Big Depthwise		
Layer type	Filters/Filter size	Output shape
DW-Conv2D	32/(3x3)s2	224x224x32
MaxPooling2D	(2x2)s2	112x112x32
DW-Conv2D	/64/(3x3)	112x112x64
MaxPooling2D	(2x2)s2	56x56x64
DW-Conv2D	128/(3x3)	56x56x128
DW-Conv2D	64/(1x1)	56x56x64
DW-Conv2D	128/(3x3)	56x56x128
MaxPooling2D	(2x2)s2	28x28x128
DW-Conv2D	128/(3x3)s2	14x14x128
MaxPooling2D	(2x2)s2	7x7x128
Reshape	-	6272
Dense	-	245
Reshape	-	7x7x5

Table 4.3: Overview of the Sequential layers adopted by the Yolo Big Depthwise architecture

The last reported architecture is "Yolo Big Depthwise".
It is identical to Yolo Big so with a higher number of filters w.r.t Yolo Small but

it uses depthwise separable convolutions instead of standard ones. This type of convolution has been explained in detail in section 2.2 and it aims to reduce the inference time of the network. The three new architecture will be trained later in this chapter and their results will be analyzed.

4.2 Loss Function

In order to train the network it's necessary to perform some modifications to the loss function. Indeed the original loss adopted by Yolo is no longer applicable as the output has a different shape and it's no longer necessary to consider the loss regarding the classes.
Considering the output encoded as explained in subsection 4.1 the loss function will change considerably and will assume this form :

$$\lambda_{coord} \sum_{i=1}^{S^2} \mathbb{1}_i^{obj} [(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2] + \lambda_{coord} \sum_{i=1}^{S^2} \mathbb{1}_i^{obj} [(\sqrt{w_i} - \sqrt{\hat{w}_i})^2 + (\sqrt{h_i} - \sqrt{\hat{h}_i})^2]$$
$$+ \lambda_{noobj} \sum_{i=1}^{S^2} \mathbb{1}_i^{noobj} [(p_i(C) - \hat{p}_i(c))^2] + \sum_{i=1}^{S^2} \mathbb{1}_{i,j}^{obj} [(p_i(C) - \hat{p}_i(c))^2] \tag{4.2}$$

Where $\lambda_{coord} = 5$ increases the loss for cells containing objects and $\lambda_{noobj} = 0.5$ decreases the loss for the cells not containing objects.

As it can be noticed in equation 4.2 with respect to the original 2.8 there are quite a few changes.
Indeed there's only one box for each cell so there's no need to consider the inner sum for each $\vec{B}$. Furthermore $\mathbb{1}_i^{obj}$ was previously computed for all the cells, containing a ground truth, considering only the bounding box prediction with the biggest IoU against the ground truth. In 4.2 instead $\mathbb{1}_i^{obj}$ considers only the cells with a ground truth different from zero and since there's only one prediction per cell it directly uses its bounding box to compute the loss.
Furthermore there's no need to compute the loss of the classes since there's only one and the probability of such class will be equal to the probability of the bounding box.

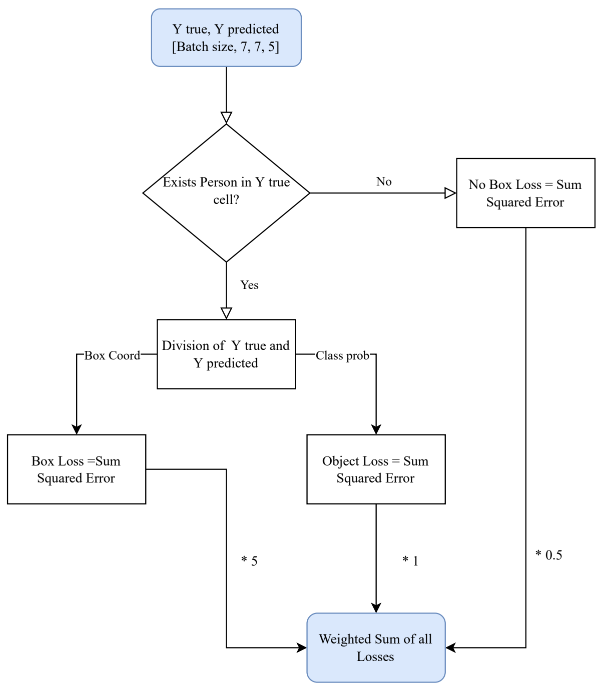

Y true, Y predicted
[Batch size, 7, 7, 5]
Exists Person in Y true cell?
No
No Box Loss = Sum Squared Error
Yes
Division of Y true and Y predicted
Box Coord
Class prob
Box Loss =Sum Squared Error
Object Loss = Sum Squared Error
* 5
* 1
* 0.5
Weighted Sum of all Losses

Figure 4.1: Flowchart diagram describing the computation of the redesigned custom loss

In figure 4.1 a flowchart diagram describing how the custom loss is computed can be visualized.
The code to compute the bounding box has been developed using the Tensorflow Keras framework in Python and can be analyzed in appendix code snippet A.2.

4.3 Batch Generator

A challenge in developing and training the neural network has been the preparation of the dataset.
In order to get a good result the neural network needs to train on a huge number of data that, for this thesis, are rgb images and their associated bounding boxes. Each image however requires a considerable amount of memory, the creation of a single vector of images containing all the training data is unfeasible since it will require more than all the available RAM memory to store it.

The training of the neural network works with batches, a collection of sam-

ples that the network works through before computing the loss and updating the weights with back propagation.
For this reason it is convenient to develop a batch generator that creates a single collection of data after each batch iteration during training.

The training is performed for a number of epochs meaning a complete pass of the training dataset through the algorithm. The batch generator at the beginning and at the end of every epoch generates an array of indices that are shuffled so that the batches between epochs do not look alike leading to a more robust network.

After each batch pass a subset of those indices are taken and the data is generated.
The input and the ground truth however have to be transformed in Yolo format. Each image has to be scaled down to the required input size of the network (for all the tested architectures it's $448x448$) and the bounding boxes need to be transformed to the grid format proposed by Yolo.

The bounding boxes associated with each image are given with respect to the original image size and in format $\vec{B} = \begin{bmatrix} B_X & B_Y & B_W & B_H \end{bmatrix}$ where B_X and B_Y refer to the top left corner.
Each bounding box is then scaled down by the factors $scale_w$ and $scale_h$ (if for example the original width is 1792 and the target width is 448 then B_X will be multiplied by $scale_w = 448/1792 = 0.25$).

The center of each bounding box is then computed and with it the coordinates of the grid cell in which it falls.
Finally the bounding box are transformed into cell notation, meaning that the center will assume values between $(0, 0)$ and $1, 1$ being the top left and bottom right corner of the cell, while width and height will be considered w.r.t the whole image $new\ width = width/image\ width$.

The newly converted bounding box will then be associated to the corresponding cell in the output matrix and its probability will be added, since it's a ground truth such probability will be equal to 1.

So the ground truth will contain vectors of type $\vec{y} = \begin{bmatrix} P & B_X & B_Y & B_W & B_H \end{bmatrix}$ for cells where there's a box and empty vectors of $size = 5$ for all the others.

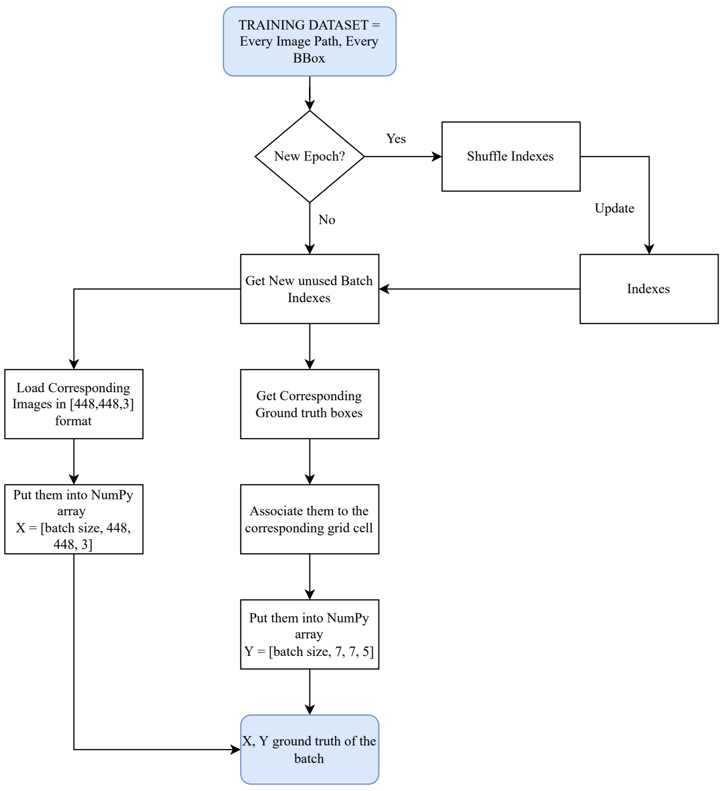

TRAINING DATASET = Every Image Path, Every BBox
New Epoch?
Yes
Shuffle Indexes
Update
No
Indexes
Get New unused Batch Indexes
Load Corresponding Images in [448,448,3] format
Get Corresponding Ground truth boxes
Put them into NumPy array X = [batch size, 448, 448, 3]
Associate them to the corresponding grid cell
Put them into NumPy array Y = [batch size, 7, 7, 5]
X, Y ground truth of the batch

Figure 4.2: Flowchart diagram describing the generation of the training batches for the custom network

In figure 4.2 a flowchart description of the general working principles of the batch generator can be visualized.
The batch generator has been developed using the TensorFlow keras framework, the data generation discussed in this subchapter can be analyzed in appendix code snippet A.3.

4.4 Live Implementation

An implementation of the network on the Raspberry Pi with the OpenCV set of libraries has been developed. The code has been written in C++ since it is efficient in terms of inference times.

First of all the network and its weights have to be converted into a format that can be read by the OpenCV library.
Code snippet 4.1 reports the conversion of the custom trained network called "yolo_cus" into a protobuf file. This type of file contains the graph definition as well as the weights of the model.

```python
from tensorflow.python.framework.convert_to_constants import convert_variables_to_constants_v2

f = tf.function(yolo_cus).get_concrete_function(tf.TensorSpec(yolo_cus.inputs[0].shape, yolo_cus.inputs[0].dtype))
f2 = convert_variables_to_constants_v2(f)
graph_def = f2.graph.as_graph_def()

with tf.io.gfile.GFile('yolo_cus.pb', 'wb') as f:
    f.write(graph_def.SerializeToString())
```

Code 4.1: Conversion of a Tensorflow network into a protobuf file for OpenCV

Having obtained the protobuf file, the network weights and the architecture can be read with the OpenCV "dnn::readNetFromTensorflow" class. The default container "dnn::Net" will then represent the network and all the other operations can be performed on it.

The input images have to be converted into a suitable format in order to be inferred by the network. As reported in code snippet 4.2 they have to be converted into RGB format, then resized to $448x448$ and the values of the pixels scaled by 255 in order to be in range $[0, 1]$.
Finally the transformed input frame can be converted into a blob and inferred, this is because the network works with batches and accepts inputs with dimensions $batchsizex448x448x3$. The "blobFromImage" class indeed, converts the input frame having dimensions $448x448x3$ into a blob of dimension $1x448x448x3$.

```cpp
std::vector<cv::Mat> preProcessAndInfer(cv::dnn::Net yolo, cv::Mat frame)
{
    Mat inImg;
    cvtColor(frame, inImg, COLOR_BGR2RGB); //convert into rgb format as used by the network
    resize(inImg,inImg,Size(448,448)); //resize image to 448x448
   format
    inImg.convertTo(inImg, CV_32F); //convert into float with values
   in range [0,1]
    inImg = inImg/255.0;
    Mat inNet = blobFromImage(inImg); //Blob to (1x448x448x3)
    yolo.setInput(inNet);
    Mat out = yolo.forward(); // out = 1x5x7x7
    vector<Mat> outs; // outs = 1*(7x7x5)
    imagesFromBlob(out,outs);

    return outs;
}
```

Code 4.2: Preprocessing and Infer of the input frames for the custom yolo networks in C++

The obtained output has to be decoded in order to get the bounding boxes predictions.
As it can be analyzed in code snippet 4.3 two for loops are able to go through all the output matrix and by analyzing the first value of each cell only bounding boxes with a probability higher than 80% are mantained.
The proposed boxes are then scaled back to the original frame size and converted into the rectangle format.

```
std::vector<cv::Rect> decodeNetOut(cv::Mat output)
{
    typedef Vec<float, 5> Vec5f; //Definition of a vector of 5 floats
    int cell_size = 64; //Cell size (448/7)

    vector<Rect> propBoxes;

  //Go through all the output matrix and mantain only bboxes with
   associated
  //probability higher than 0.8
    for(int i = 0; i < 7; i++)
    {
        for(int j = 0; j < 7; j++)
        {
            if(output.at<Vec5f>(i,j)[0] > 0.8)
            {
                float x = output.at<Vec5f>(i,j)[1];
                float y = output.at<Vec5f>(i,j)[2];
                float w_cell = output.at<Vec5f>(i,j)[3];
                float h_cell = output.at<Vec5f>(i,j)[4];
                double ox = x*cell_size + cell_size*j;
                double oy = y*cell_size + cell_size*i;
                double w = w_cell*1280; //Scale back to original image size
                double h = h_cell*720;
                double lx = (ox/scale_w) - w/2;
                double ly = (oy/scale_h) - h/2;
                propBoxes.push_back(Rect(static_cast<int>(lx), static_cast<int>(ly), static_cast<int>(w), static_cast<int>(h)));
            }
        }
    }
    return propBoxes;
}
```

Code 4.3: Decode the output of the custom yolo networks in C++

All the reported networks (i.e. "Yolo Small", "Yolo Big", "Yolo Big Depthwise") share the same input and output format. For this reason all the functions defined to pre-process, infer and decode the output of the networks can be used for each of them without applying any change.

4.5 First Results

The three proposed architectures (i.e. "Yolo Small", "Yolo Big" and "Yolo Big Depthwise") have been trained and their performances have been tested in terms of precision, recall and F1 score.
Furthermore a live application has been developed.
In this section the results concerning the performances of the networks on the test dataset and on the target embedded devices, will be reported.

4.5.1 Training

Training is a long and computationally expensive process.
Luckily the company in which this project has been developed provided access to a workstation equipped with a Nvidia Quadro P4000 GPU. This considerably decreased the amount of time required for training and allowed the trial of more architectures and datasets. This is because GPUs are much better for running

neural networks as they are generally better at working with big matrices than CPUs.

However the training process still requires considerable time depending on the CNN and the training dataset, this is why the first tests have been performed on a arguably small size dataset.

Indeed the three redesigned networks have been initially trained on a subset Coco 2017 dataset.
10000 images containing instances of persons have been downloaded as well as 500 images without any for a total of 10500 images of which 1000 have been used for validation.
The size of this initial dataset allowed to validate the model and to make some small changes to the architectures without waiting too much to validate them.

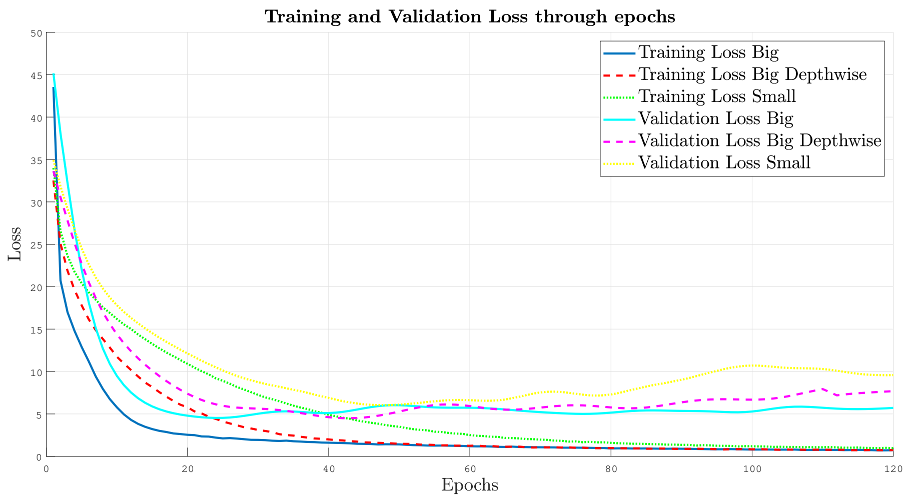


Figure 4.3: Comparison of the custom loss behaviour through training epochs for the three reported re-designed architectures

During all the trainings the optimizer that has been used is the Adam optimizer, an optimization algorithm that can be used instead of the traditional stochastic gradient descent. The main advantage is that it allows to use different learning rates (the speed at which the neural network trains) during the training. So, Adam allows to use a bigger learning rate for the first epochs of training and

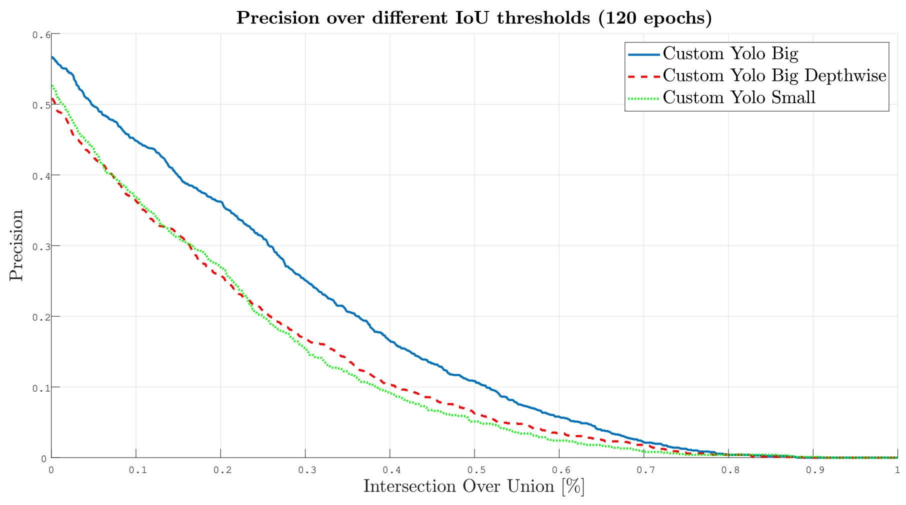
gradually reduces it towards the end of training.
The loss has been computed using the custom loss defined in section 4.2.
In figure 4.3 the training losses for the three reported models through the epochs of training are reported. It can be noticed that all the three losses decrease and thus the optimizer is working well.
Furthermore it's interesting to notice that the more parameters a model has (such as Yolo Big) the faster the loss is initially decreasing. This is because when more parameters are available the network can better approximate the seeked out function.
In figure 4.3 the trend of the validation losses can also be observed. Although all the curves are not diverging and thus the models are not over-fitting, they seem to have reached a plateau as none of them is decreasing anymore.
This could be because the initial training dataset is not that big and thus is not more "learnable". Another reason can be imputed to the limited size of the networks, since they're really small they aren't capable of learning the desired function.

The number of epochs has been initially set to 120 in order to make small modifications on the networks without having to wait too much for training.
Furthermore, this faster comparison followed by the live application and the precision, recall and f1 score metrics evaluated in the next sub-sections gave interesting insights on where to concentrate the research for developing the network.

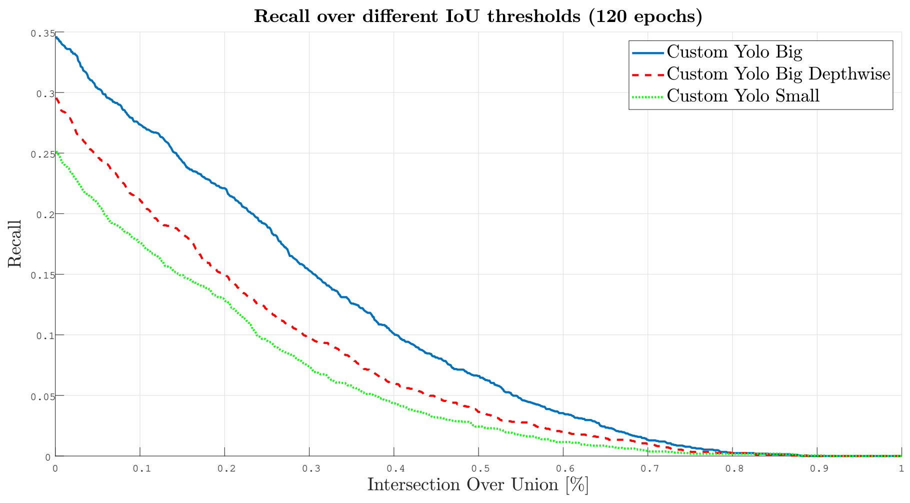
4.5.2 Precision, Recall and F1 Score Results

In order to validate the models the performance metrics discussed in section 3.4 have been evaluated for increasing IoU thresholds over the same test dataset. The results reported in figures 4.5 and 4.4 give interesting insights on the models and where to concentrate the direction of the research. It can be indeed noticed that with more parameters the network generally performs better obtaining higher precisions and more importantly recall and F1 score.

Furthermore it can be noticed that adopting depthwise convolutional layers leads to circa 5% worse performances w.r.t traditional ones but the performances remain better than decreasing the number of filters like in Yolo Small.



(a)



(b)

Figure 4.4: Comparison of the different redesigned yolo architectures reported in this chapter with a training of 120 epochs. In (a) the Precision over different IoU thresholds is given. In (b) the Recall over IoU.

This results give a clear indication that, as expected, the more parameter the network has, the better it will perform. It's indeed straightforward to reason that with more parameters and convolutional layers the network will approximate better the seeked function. This however increases the inference times that can become not suitable for an embedded application.

For this reason a compromise between performances in terms of precision, recall, f1 score and inference time has to be found.

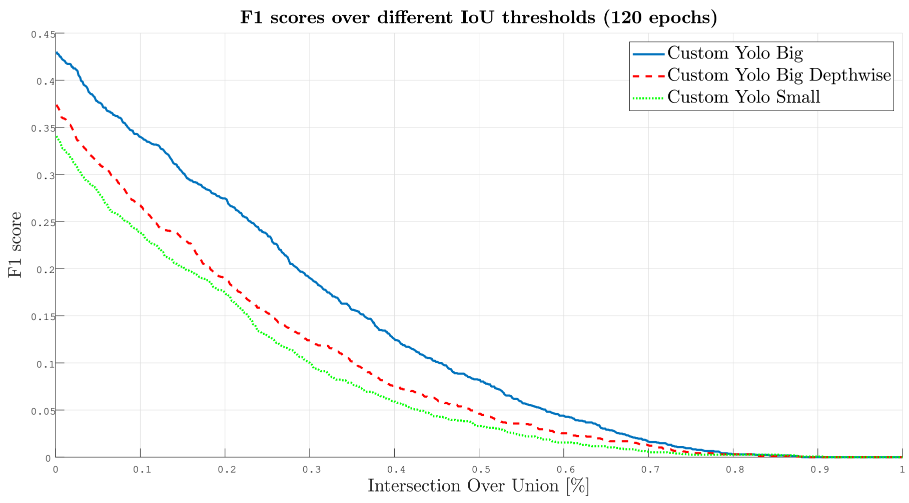


Figure 4.5: Comparison of the F1 scores of the different redesigned yolo architectures reported in this chapter with a training of 120 epochs

4.5.3 Live Application Results

The live application described in section 4.4 has been tested out with the three architectures reported in the previous sections on the Raspberry Pi 3 and 4.
The scope of this test is to comprehend if the proposed architectures are capable of running live on the target embedded device as well as providing more insights on how to improve the architecture design.

As it can be noticed in figure 4.6 the inference time of all three network, in the less powerful Pi 3, is less than half a second.
Furthermore, as expected, a bigger number of parameters like in Yolo Big considerably increases the inference time. It can also be noticed the significant advantage of using depthwise separable convolution; indeed Yolo Big Depthwise has a bigger precision, recall and f1 score w.r.t Yolo Small while mantaining a similar or even lower inference time.

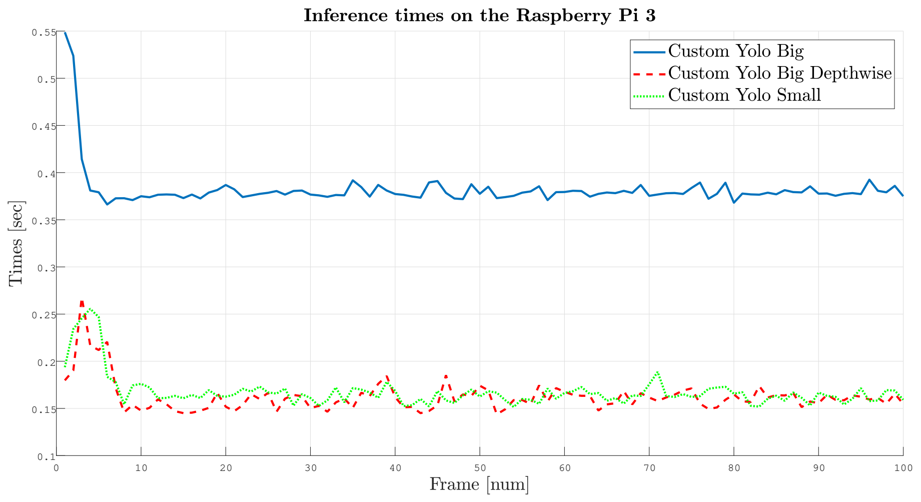


Figure 4.6: Comparison of the inference times of the different redesigned yolo architectures reported in this chapter on the Raspberry Pi 3

Even if decreasing the number of parameters and convolutions will certainly decrease the inference time, it will badly effect the performances in terms of precision, recall and f1 score making the application worse than the off-the shelf methods Hog and Viola.

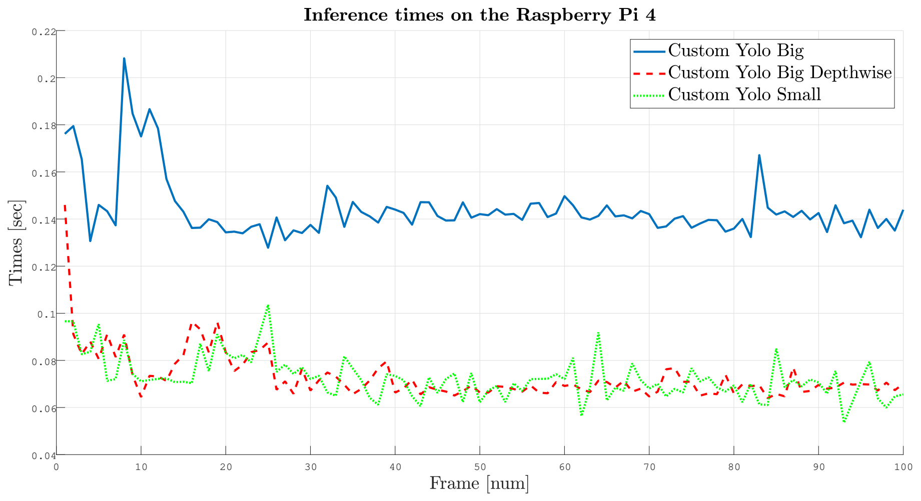


Figure 4.7: Comparison of the inference times of the different redesigned yolo architectures reported in this chapter on the Raspberry Pi 4

As shown in figure 4.7, the Raspberry Pi 4 is able to considerably reduce the inference times with respect to the Pi 3.

5 New Solutions for Person Detection

Having studied the limitations of the off-the shelf algorithms and of the custom network proposed in chapter 3 and 4, the work proposed in this thesis will now propose two final solutions for the task of person detection.
The first one tries to improve the custom network by adopting some strategies like adopting a bigger training dataset over a bigger network.
The second one takes advantage of a nano version of Yolo v6 by training it over the same dataset.

5.1 Improving the Custom Network

By analyzing the results of the previous chapter in sub-sections 4.5.2 and 4.5.3, it has been proven that the redesigned Yolo network is capable of running live on the Raspberry Pi 3 and can reach at least 15 fps on the Pi 4. The results in terms of precision, recall and f1 score are however far from the original Yolo results reported in sub-section 3.4.3 while they remain similar to the Viola and Hog machine learning approaches.

In figure 5.1 the Custom Yolo Big architecture has been compared to the original Yolo and Hog detector; the f1 score has been chosen as the more meaningful parameter to compare as it gives a weighted average of precision and recall.
The two reported off the shelf methods have been chosen for this comparison as they give an upper and lower limit of where the desired performances of the

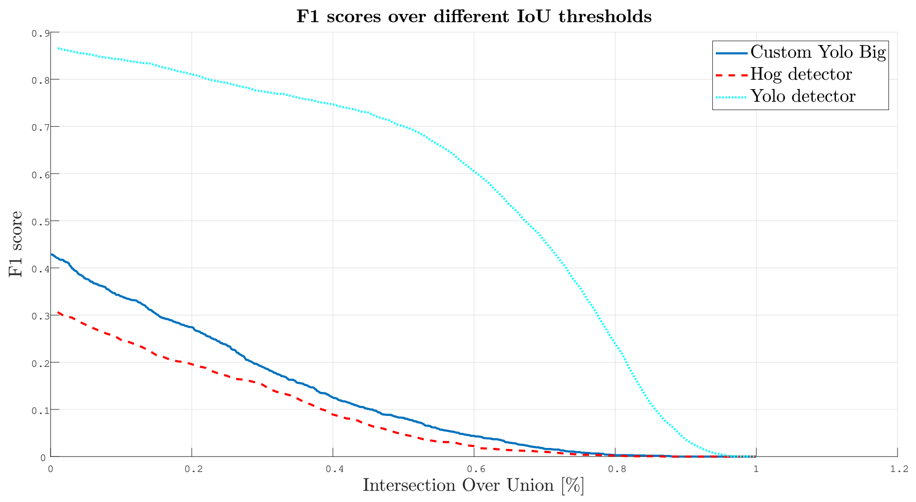


Figure 5.1: Comparison of the Custom Yolo F1 score with respect to meaningful off the shelf algorithms analyzed in 3.4.3

custom network should be.
It's indeed unrealistic to outperform the original Yolo with its huge number of convolutional layers and parameters, while it's desirable to top the Hog based detector.
It can be noticed in figure 5.1 that this result has been already achieved as Custom Yolo Big positions itself between the two off the shelf methods, it's however interesting to apply some considerations in order to improve this performance.

The previous section highlighted that increasing the number of filters and convolutional layers should lead to better precision, recall and f1 score performances, as with more parameters the network should be more capable of approximating the desired function. This will however lead to larger inference times so it's important to find a compromise.
With this considerations in mind, some strategies can be employed in order to improve the network.

5.1.1 Bigger Dataset and more Epochs

In order to improve the results, two main strategies have been first considered.

First of all the size of the training dataset has been considerably extended adding

up to more than 60 thousand images. This will allow the network to analyze and train over more cases and environments containing persons.
The main disadvantage of this strategy is the considerably longer training time with respect to the previous one. Indeed with the first training dataset each epoch took more or less 2:00 minutes, on Yolo Big, while with the increased dataset each epoch now takes more than 13:00 minutes. This means that a training of 120 epochs will take approximately 26 hours, considerably decreasing the possibility of changing some of the hyper-parameters in the network and observing the results after short amounts of times.

The second strategy consists in increasing the number of training epochs, in this way the network can learn from the dataset for a longer period of time and approximate better the seeked behaviour.

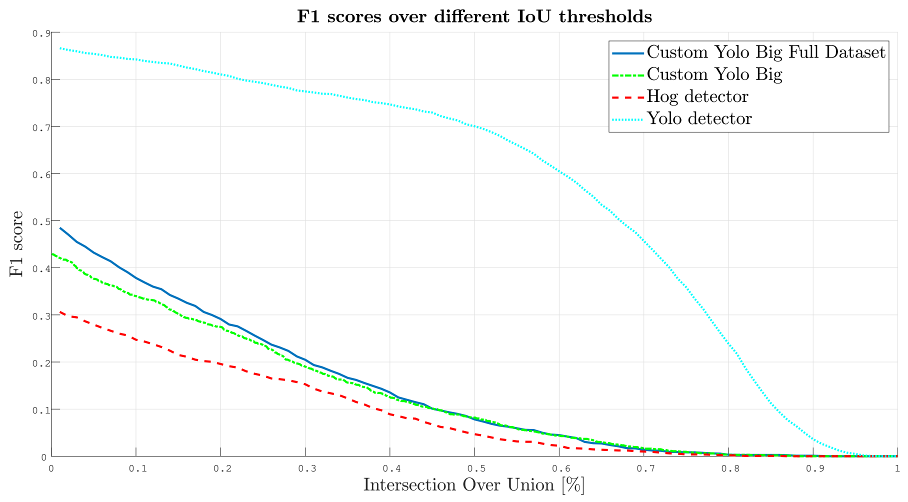


Figure 5.2: Comparison of the Custom Yolo F1 score trained for 1000 epochs on the full dataset and the same network trained over the smaller dataset with respect to meaningful off the shelf algorithms analyzed in 3.4.3

However, as it can be noticed in figure 5.2 training on a bigger dataset for more epochs have only increased slightly the performances.
This is because the number of convolutional layers and filters used by the network, limited in order to achieve high fps on a live application for an embedded device, have evidently met their limit in approximating the desired function.

5.1.2 Bigger Network

As highlighted in subchapter 5.1.1 the network has met its limit in terms of improvements.
The performances in terms of precision, recall and f1 score can thus be improved by increasing the number of filters and adding convolutional layers to the custom network. This change however will certainly decrease the fps performances of the live embedded application. It's thus important to find a balance between fps and f1 score without adding too many layers to the network.

After some design iteration a final custom network solution has been developed and can be examined in detail in table 5.1.

Custom Yolo v2		
Layer type	Filters/Filter size	Output shape
Conv2D	16/(7x7)	448x448x16
MaxPooling2D	(2x2)s2	224x224x16
Conv2D	48/(3x3)	224x224x48
MaxPooling2D	(2x2)s2	112x112x48
Conv2D	32/(1x1)	112x112x32
Conv2D	64/(3x3)	112x112x64
Conv2D	32/(1x1)	112x112x32
Conv2D	64/(3x3)	112x112x64
MaxPooling2D	(2x2)s2	56x56x64
Conv2D	64/(1x1)	56x56x64
Conv2D	128/(3x3)	56x56x128
Conv2D	64/(1x1)	56x56x64

Continued on next page

Table 5.1 – continued from previous page

Custom Yolo v2		
Layer type	Filters/Filter size	Output shape
Conv2D	128/(3x3)	56x56x128
MaxPooling2D	(2x2)s2	28x28x128
Conv2D	128/(1x1)	28x28x128
Conv2D	256/(3x3)	28x28x256
Conv2D	128/(1x1)	28x28x128
Conv2D	256/(3x3)	28x28x256
Conv2D	256/(3x3)	28x28x256
Conv2D	256/(3x3)s2	14x14x256
Conv2D	256/(3x3)	12x12x256
Conv2D	256/(3x3)	10x10x256
Flatten	-	25600
Dense	-	128
Dense	-	256
Dense	-	245
Reshape	-	7x7x5

Table 5.1: Overview of the Sequential layers adopted by the Custom Yolo v2 network

With respect to previous designs that had only 6 convolutional layers, the network now has 18 of them, additionally two fully connected layers have been added.
It can be noticed that the layers at higher resolutions (i.e. the first ones) have a

smaller number of filters and a bigger kernel. This leads to overall less computations in the first few layers and allows to increase them at lower resolutions feature maps.
The last two convolutional layers don't apply any type of padding and thus decrease the resolution.

The same design has also been developed by adopting depthwise separable convolutions instead of the regular ones and will be later referred as "Custom Yolo v2 DW". The results of this two final designs will be analyzed in the next sub-chapter.

5.1.3 Improved Results

The two improved networks, Custom Yolo v2 and Custom Yolo v2 DW, have been trained for 500 epochs on the full dataset containing more than 60000 images, taking more than 4 days each to complete the training.

As it can be noticed in figure 5.3 the two custom networks have improved with respect to previous iterations such as figure 4.5. This improvement however comes with the cost of losing speed in terms of inference times.
Indeed as it can be noticed in figure 5.4 the Yolo Custom v2 is able to provide only a speed of 2.7 fps while Yolo Custom v2 DW can run at 4.7 fps.

As expected the solution that adopts depthwise separable convolutions performs faster but with worse F1 score.

This result and the whole analysis given through the chapter show that building a custom CNN is not a straightforward operation.
It's indeed really complicate to find a good combination of hyperparameters and convolutional layers that allow to construct a good architecture. Although inspiration can be taken from state of the art networks, it's necessary to squeeze them in order to run them live on an embedded device and it's time consuming to find a good balance between speed and detection accuracy. This is mainly

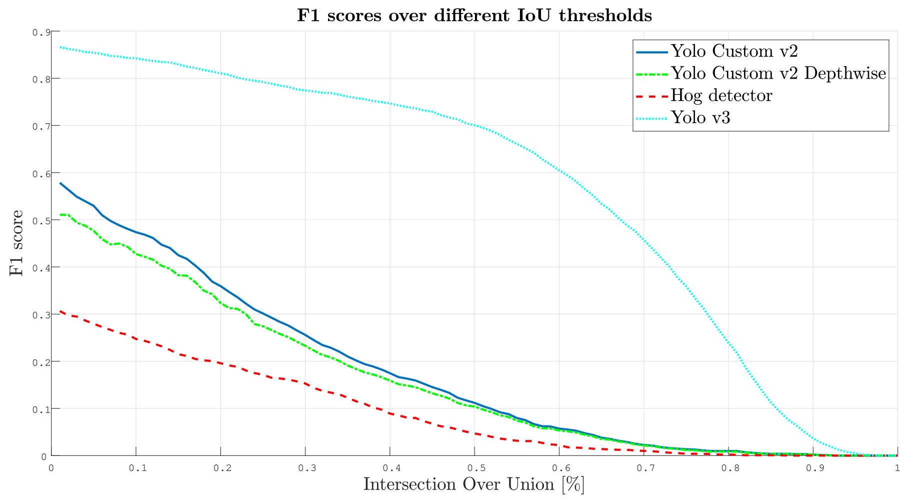


Figure 5.3: Comparison of the Custom Yolo v2 and Custom Yolo v2 DW F1 score trained for 500 epochs on the full dataset with respect to meaningful off the shelf algorithms analyzed in 3.4.3

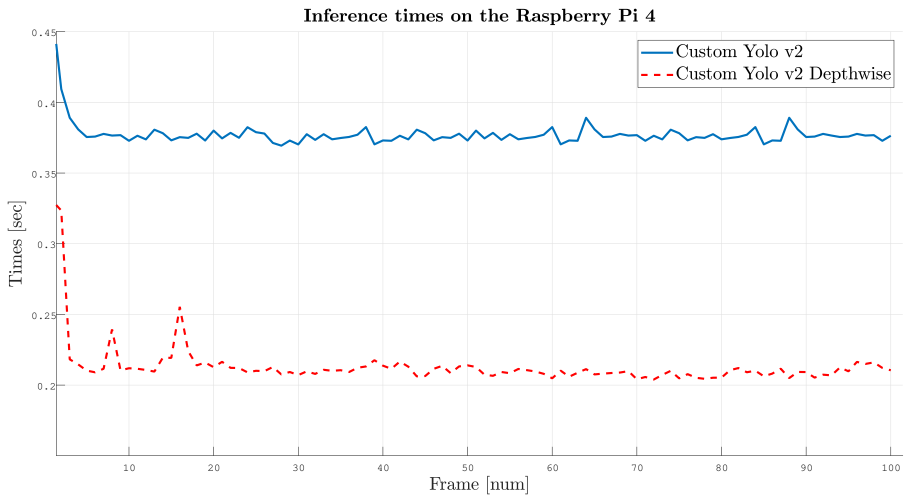
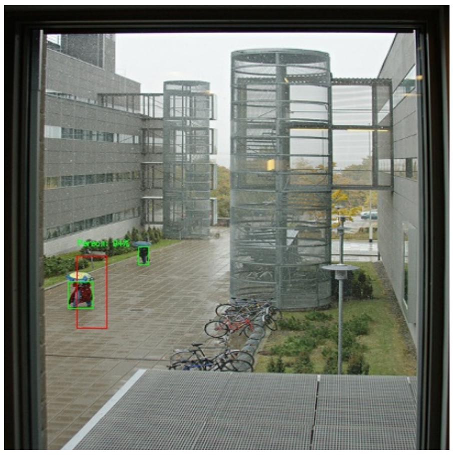


Figure 5.4: Comparison of the Custom Yolo v2 and Custom Yolo v2 DW inference time on the Raspberry Pi 4

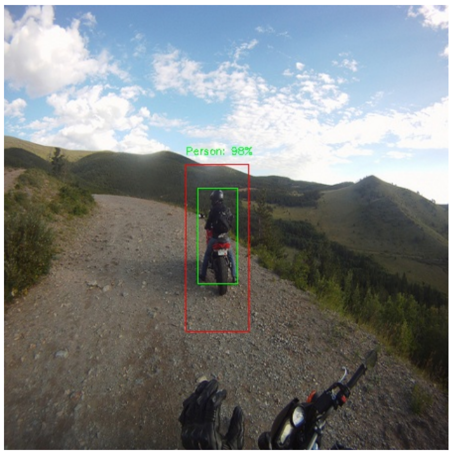
because each solution needs to be trained separately taking hours or days before being able to analyze the results.

This thesis tried to propose a simple and straightforward DNN without too many complex layers as the more recent state of the art architectures. This however comes with a decrease in performances that is mainly noticeable in the loss of accuracy when the IoU threshold increase.

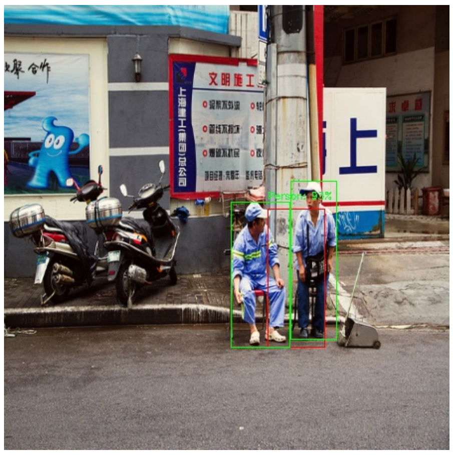
It has been indeed noticed that the custom solution often proposes bounding



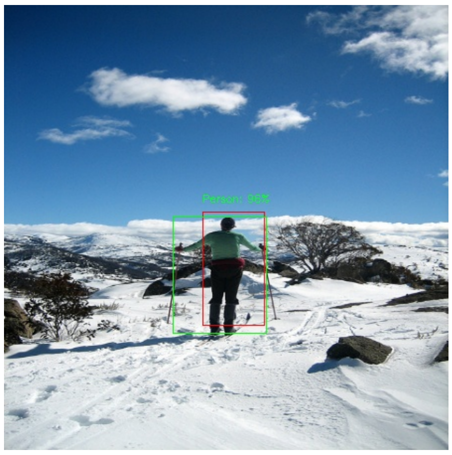
(a) IoU = 30%



(b) IoU = 37%



(c) IoU = 41%



(d) IoU = 60%

Figure 5.5: Some detections that have a low IoU w.r.t the ground truth, in the caption of each image the IoU percentage is reported

boxes that are too big, too small or de-centered with respect to the ground truth while they generally identify the person. This means that when increasing the IoU threshold most of this quasi-correct bounding boxes will be treated as false positives considerably decreasing the general performances.
When the IoU threshold is low instead, the performances are much better.
In figure 5.5 some examples of proposed bounding boxes with low IoU thresold with respect to the ground truth are reported. In the caption of each image it can be noticed how the proposed bounding boxes that are not tight with respect to the ground truth produce a low IoU score.
This problem is mainly imputable to the network and its combination of hyper-parameters as previously explained. However, it can be also considered that the training dataset, containing positive sample for body parts and group of

persons, probably confuses the network while it's learning.

5.2 Yolo v6 Nano, an Alternative Approach

The results of the custom CNN architectures for person detection highlighted that building a network from scratch is not a straightforward operation.
It's indeed difficult to find the right combination of convolutional layers and hyperparameter that can achieve the desired behaviour. The solution that has been delivered can be considered satisfying but there are other approaches that can deliver better solutions.

The Yolo network during the years reached its 6th iteration and its authors created a set of functions that allow to train a customized solution based on a dataset chosen by the user.
Furthermore Yolo v6 comes in different versions, the most interesting one for the scope of this thesis is a ligthweight alternative called nano.

5.2.1 Network Design

C. Li et al. in [12] introduced a new version of Yolo, whom it has been given the name of Yolo v6, framing it as on of the most recent design of the Yolo family. The object detector, as it can be visualized in figure 5.6, is a single stage detector and is composed of different parts.

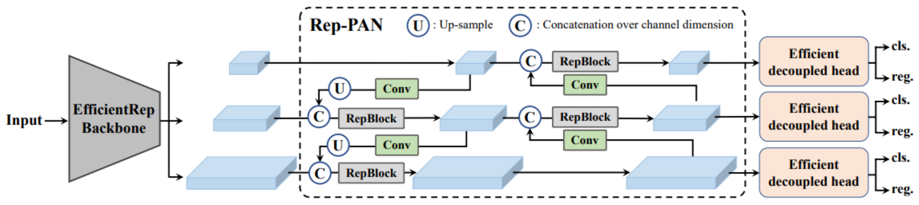


Figure 5.6: Visualization of the Yolo v6 framework

The backbone is responsible of representing features; the design proposed is re-parametrizable and scalable allowing to propose multiple networks of differ-

ent size. It produces feature representation at three different scales, in this way small objects can be detected at bigger scales and big objects at smaller scales.
The neck of the network performs feature integration of the three different scales.
The head is decoupled, at each detection scale two branches are produced, one regression branch for the bounding boxes and one classification branch for the classes.
The design of the network is anchor-free allowing to reduce computations in the post-processing part.

The loss is computed as a multi part loss. The classification loss and the box regression loss. In particular the box regression loss is computed as an IoU loss that regresses the four bounds of the box as a whole unit.

Furthermore some other improvements are introduced when training the network. More training epochs are used to obtain better performances and RepOptimizer is used to obtain quantization friendly weights. With this strategy the run time is further optimized without losing much performances.

5.2.2 Training

The training of the network has been performed thanks to the Github repository[1] by the user meituan.
The dataset has been prepared using Roboflow, an online tool that allows to convert annotations from the COCO json format to the one required by Yolo v6 (txt files, a custom YAML file and organized directories). The dataset used has 10454 images and is the smallest one of the two used through this thesis project.
This is because with the Roboflow free account it's not been possible to upload the whole full dataset.

The training has been performed on the Yolo v6 nano architecture, the smallest one of the v6 family, allowing to obtain faster inference times. Furthermore the input dimension has been selected as a 416x416 image smaller than the original 640x640 allowing to achieve even faster performances as the resulting network will be smaller in size.

[1]https://github.com/meituan/YOLOv6

The network has been trained for 200 epochs. This is because it had to be trained on the Google Colab platform that doesn't allow long sessions.

5.2.3 Final Results

After training, the Yolo v6 nano architecture has been tested over the same test datest used through the thesis.

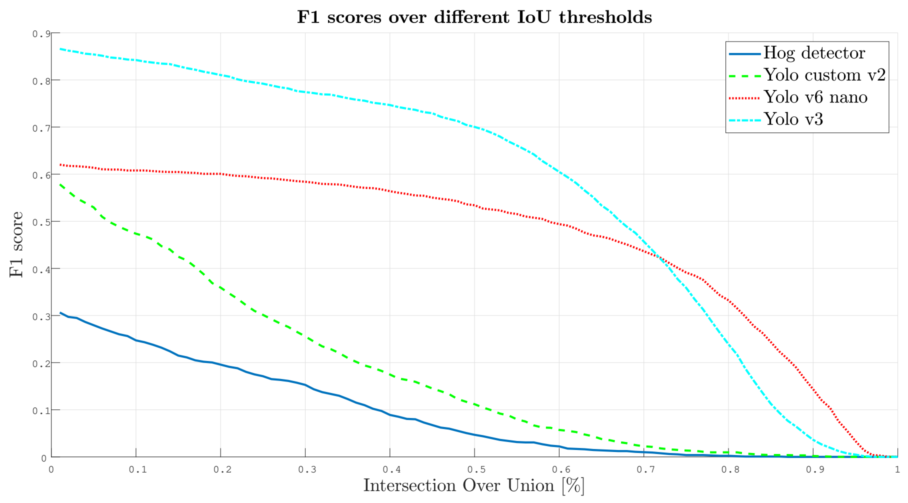


Figure 5.7: F1 score over different IoU thresholds, comparison between Yolo v6 nano, Hog, Yolo v2 custom and Yolo v3

As it can be noticed in figure 5.7 the proposed architecture gives really interesting results. The curve indeed is much similar to the one obtained with Yolo v3 while being approximately 20% worse in performances. Indeed the new architecture is much smaller and optimized for embedded devices. This however, as pointed in table 5.2, allows to obtain faster inference times with respect to Yolo v3 that allow to run the network fast enough on an embedded device such as the Raspberry Pi 4.
The table however shows that the results are not suitable for an application on the Raspberry Pi 3, that with its more limited hardware is only capable of analyzing less than 1 fps.

The result of the Yolo v6 nano show that the state of the art architectures are

	Yolo v3	Yolo v6 nano	Yolo Custom v2	Yolo Custom v2 DW
FPS Pi 3	0.07 fps	0.92 fps	1.18 fps	1.8 fps
FPS Pi 4	0.2 fps	**2.5 fps**	**2.7 fps**	**4.7 fps**

Table 5.2: Comparison of the Real Time capability of the different proposed architectures for the task of Person detection on the Raspberry Pi 3 and the Raspberry Pi 4

pushing towards being faster. However this solutions can be used in some application that require to analyze only some frame of a real time feed and cannot be yet defined suitable for a real time implementation on an embedded device.

6 Conclusions

In this chapter the conclusions over the work reported in this thesis are going to be given.
First a summary and a discussion over the result obtained is going to be given.
Then, a quick overview of the future works that rose as a result of the studies of this thesis is going to be reported.

6.1 Summary and Discussion

The work of this thesis argued on the possibility of implementing a computer vision based algorithm able to detect persons in work areas on an embedded system as an aid to other security systems.
The work has been developed as a Research and Development project of a company based in Vicenza.
The seeked solution, had to be as good as possible at detecting persons while being fast enough to run live on devices such as the Raspberry Pi 3 and 4 that, it's important to remark, don't have a GPU that would allow to run modern CNNs efficiently.

First of all some off-the shelf solutions have been analyzed and implemented.
The traditional machine learning approaches such as Hog and Viola proved themself to be satisfying in terms of inference times. These two historical solutions were indeed designed for machines that had resources comparable to

a modern day Raspberry. They showed however to be not suitable in terms of accuracy as it has found that it's very low.
Deep learning architectures, such as Yolo, instead, proved to be really accurate when detecting persons but required too much time to infer a single frame, making them not suitable for a GPU-less embedded device application.

Having studied the state of the art and understood the limitations of the off-the shelf architectures, the work of the thesis shifted on implementing a customized solution.
This solution was strongly inspired by the original Yolo architecture, with many strategies applied to squeeze the original network as much as possible in order to achieve a satisfying inference speed. All the strategies employed to design the network from the bottom up, required to develop functions such as the custom loss used for training and classes and the batch generator for preparing the training data.
The development of a custom solution from scratch however, is not straightforward. Finding the right combination of hyperparameters and convolutional layers without increasing them too much in order to not lose too much inference speed is a long process. This is mainly because each solution needs to be trained from scratch and this process takes hours or even days depending on the size of the training dataset and the size of the network.
After some iteration of the architecture two solutions were proposed, one faster but less accurate and the other one slower but more accurate. The results of this implementations are suitable as an aid to other security system but cannot be the only employed system. Indeed the results are not very accurate when the metric employed is the IoU threshold as the proposed bounding boxes are often too big or too small with respect to the ground truth.

Having seen the limitations of developing a new architecture from scratch a new solution has been proposed.
This consisted in using an already developed network, Yolo v6 in its nano declination.
The Yolo v6 nano architecture has been trained using the same dataset as the custom architecture and, although it is slightly slower and not suitable for really limited embedded devices such as the Raspberry Pi 3, it is much better in accuracy. At its sixth iteration indeed, Yolo v6 with years of development is

capable of delivering more than satisfying performances for the purpose of this thesis. Furthermore it showed that the deep neural networks are getting better and more efficient. This, together with the development of better embedded devices, will probably allow in the future to have impressive CNN detection architectures run in real time on edge technologies.

6.2 Future Works

The work reported in this thesis can be further developed in order to achieve better performances both in detection and inference times in embedded devices. In particular some possible future studies are here reported :

- Safety with a classification system. Convolutional Neural Networks that are concerned with classifying images are notably smaller and faster than the object detection ones. For this reason it is interesting the implementation of a simple classifier (Person - No Person) on an embedded device. This system could be an useful aid to safety as it should correctly identify the presence of a human in the area covered by the camera. However it won't give the useful information relative to the location of such person.
- Detection based on centroid. As it has been reported in this thesis, decreasing the number of parameters is a useful technique to increase the inference speed of the networks. It is then interesting to study the implementation of a CNN for object detection based only on the center of an object and not giving the additional informations relative to the bounding boxes.
- Newer algorithms. Research in computer vision for more efficient Convolutional Neural Networks is advancing fast. During the work of this thesis newer algorithms, such as Yolo v7, came to life. Implementing these state of the art algorithms and observing their behaviour relative to the problem of this thesis can be an useful study.

References

[1] Antonio Brunetti et al. "Computer vision and deep learning techniques for pedestrian detection and tracking: A survey". In: Elsevier, 2018.

[2] N. Dalal and B. Triggs. "Histograms of oriented gradients for human detection". In: *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*. Vol. 1. 2005, pp. 886–893.

[3] Y. Freund and E. Schapire. "A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting". In: *Journal of Computer and System Sciences*. Vol. 55. 1. 1997, pp. 119–139.

[4] Ross Girshick. "Fast R-CNN". In: arXiv, 2015. DOI: `10.48550/ARXIV.1504.08083`.

[5] Ross Girshick et al. "Rich feature hierarchies for accurate object detection and semantic segmentation". In: arXiv, 2013. DOI: `10.48550/ARXIV.1311.2524`.

[6] Andrew Howard et al. "Searching for MobileNetV3". In: vol. abs/1905.02244. 2019.

[7] Andrew G. Howard et al. "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications". In: arXiv, 2017. DOI: `10.48550/ARXIV.1704.04861`.

[8] Peiyuan Jiang et al. "A Review of Yolo Algorithm Developments". In: *Procedia Computer Science*. Vol. 199. 2022, pp. 1066–1073.

[9] Jordan Johnston, Kaiman Zeng, and Nansong Wu. "An Evaluation and Embedded Hardware Implementation of YOLO for Real-Time Wildfire Detection". In: *2022 IEEE World AI IoT Congress (AIIoT)*. IEEE. 2022.

[10] C. E. Kim et al. "A Comparison of Embedded Deep Learning Methods for Person Detection". In: *Proc. of the 14th Int. Conf. on Computer Vision Theory and Applications (VISAPP)*. 2019.

[11] Darren Anando Leone et al. "A Survey: Crowds Detection Method on Public Transportation". In: *2021 1st International Conference on Computer Science and Artificial Intelligence (ICCSAI)*. IEEE. 2021.

[12] Chuyi Li et al. "YOLOv6: A Single-Stage Object Detection Framework for Industrial Applications". In: arXiv, 2022. DOI: `10.48550/ARXIV.2209.02976`. URL: `https://arxiv.org/abs/2209.02976`.

[13] R. Lienhart and J. Maydt. "An extended set of Haar-like features for rapid object detection". In: *Proceedings. International Conference on Image Processing*. 2002.

[14] Tsung-Yi Lin et al. "Microsoft COCO: Common Objects in Context". In: *Computer Vision – ECCV 2014*. Ed. by David Fleet et al. Springer International Publishing, 2014, pp. 740–755.

[15] Wei Liu et al. "SSD: Single Shot MultiBox Detector". In: *Computer Vision ECCV 2016*. Springer International Publishing, 2016, pp. 21–37. DOI: `10.1007/978-3-319-46448-0_2`.

[16] Y. Ma, X. Chen, and G. Chen. "Pedestrian Detection and Tracking Using HOG and Oriented-LBP Features". In: *Network and Parallel Computing*. Springer Berlin Heidelberg, 2011, pp. 176–184.

[17] Seyed Yahya Nikouei et al. "Real-Time Human Detection as an Edge Service Enabled by a Lightweight CNN". In: *2018 IEEE International Conference on Edge Computing (EDGE)*. 2018, pp. 125–129. DOI: `10.1109/EDGE.2018.00025`.

[18] Seyed Yahya Nikouei et al. "Real-Time Human Detection as an Edge Service Enabled by a Lightweight CNN". In: *2018 IEEE International Conference on Edge Computing (EDGE)*. 2018, pp. 125–129. DOI: `10.1109/EDGE.2018.00025`.

[19] Joseph Redmon and Ali Farhadi. "YOLO9000: better, faster, stronger". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017, pp. 7263–7271.

[20] Joseph Redmon and Ali Farhadi. "YOLOv3: An Incremental Improvement". In: *CoRR*. Vol. abs/1804.02767. 2018.

[21] Joseph Redmon et al. “You Only Look Once: Unified, Real-Time Object Detection”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016.

[22] Shaoqing Ren et al. “Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks”. In: arXiv, 2015. DOI: 10.48550/ARXIV.1506.01497.

[23] Mark Sandler et al. “MobileNetV2: Inverted Residuals and Linear Bottlenecks”. In: arXiv, 2018. DOI: 10.48550/ARXIV.1801.04381.

[24] MS Sruthi et al. “YOLOv5 based Open-Source UAV for Human Detection during Search And Rescue (SAR)”. In: *2021 International Conference on Advances in Computing and Communications (ICACC)*. IEEE. 2021.

[25] Zhihong Sun et al. “A survey of multiple pedestrian tracking based on tracking-by-detection framework”. In: IEEE, 2020.

[26] A. A. Süzen, B. Duman, and B. Sen. “Benchmark Analysis of Jetson TX2, Jetson Nano and Raspberry PI using Deep-CNN”. In: *Int. Congress on Human-Computer Interaction, Optimization and Robotic Applications (HORA)*. 2020.

[27] P. Viola and M. Jones. “Rapid object detection using a boosted cascade of simple features”. In: *Proceedings of the 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition. CVPR 2001*. 2001.

[28] P. Warden and D. Situnayake. “TinyML: Machine Learning with TensorFlow Lite on Arduino and Ultra-low-power Microcontrollers”. In: O’Reilly, 2020.

[29] Zhewei Xu et al. “Human-Cascaded network for Robust Detection of Occluded Pedestrian”. In: *2022 3rd International Conference on Computing, Networks and Internet of Things (CNIOT)*. IEEE. 2022.

A Appendix

A.1 Code Snippets

```cpp
void postprocess(Mat& frame, const vector<Mat>& outs)
{
    vector<int> classIds;
    vector<float> confidences;
    vector<Rect> boxes;

    for (size_t i = 0; i < outs.size(); ++i)
    {

        float* data = (float*)outs[i].data;
        for (int j = 0; j < outs[i].rows; ++j, data += outs[i].cols)
        {
            Mat scores = outs[i].row(j).colRange(5, outs[i].cols);
            Point classIdPoint;
            double confidence;

            minMaxLoc(scores, 0, &confidence, 0, &classIdPoint);
            if (confidence > confThreshold)
            {
                int centerX = (int)(data[0] * frame.cols);
                int centerY = (int)(data[1] * frame.rows);
                int width = (int)(data[2] * frame.cols);
                int height = (int)(data[3] * frame.rows);
                int left = centerX - width / 2;
                int top = centerY - height / 2;

                classIds.push_back(classIdPoint.x);
                confidences.push_back((float)confidence);
                boxes.push_back(Rect(left, top, width, height));
            }
        }
    }

    vector<int> indices;
    NMSBoxes(boxes, confidences, confThreshold, nmsThreshold, indices);

}
```

Code A.1: Yolo detector post processing applied to the output of the network

```python
def custom_loss(y_true, y_pred):
    # y_true (Batch size, 7, 7, 5)
    # y_pred (Batch size, 7, 7, 5)

    mse = tf.keras.losses.MeanSquaredError(reduction = "sum") # Define the SUM squared error loss
    predictions = tf.reshape(y_pred,(-1,7,7,5)) # The predictions are a tensor, need some reshaping to manipulate it

    exists_box = tf.expand_dims(y_true[...,0], 3) # A box exists if the first entry of the cell is equal to 1

    #------------#
    #| BOX LOSS |#
    #------------#

    pred_box = exists_box*predictions[...,1:5] #Calculate only loss for the cells that contain a box
    target_box = exists_box*y_true[...,1:5] #Target boxes

    epsilon = tf.fill(tf.shape(pred_box[..., 2:4]), 1e-6) #Needed to avoid divergence of square root derivatives in back propagation

    # width and height are penalyzed using the square root, however predictions can be negative so multiply by sign in order to obtain positive
    # and take absoulte value in the square root
    wh_pred = tf.math.sign(pred_box[...,3:5]) * tf.math.sqrt(tf.math.abs(pred_box[...,3:5] + epsilon))
    wh_targ = tf.math.sqrt(target_box[...,3:5] + epsilon)

    # Get also centers
    xy_pred = pred_box[...,1:3]
    xy_true = target_box[...,1:3]

    # Concatenate the new xy and wh in order to calculate sum squared root
    final_pred_box = tf.concat([xy_pred,wh_pred], axis = 3)
    final_true_box = tf.concat([xy_true,wh_targ], axis = 3)
    box_loss = mse(tf.reshape(final_pred_box, (-1, tf.shape(final_pred_box)[-1])),tf.reshape(final_true_box, (-1, tf.shape(final_true_box)[-1])))

    #---------------#
    #| OBJECT LOSS |#
    #---------------#

    # Take only the first entry of each box corresponding to the probability that there's an object
    pred_obj = predictions[...,0:1]
    true_obj = y_true[...,0:1]

    #Calculate object loss as in the paper
    object_loss = mse(tf.reshape(exists_box*pred_obj, (-1, )), tf.reshape(exists_box*true_obj, (-1, )) )

    #------------------#
    #| NO OBJECT LOSS |#
    #------------------#

    # Calculate the loss for cells that don't have objects
    non_exists_box = 1 - exists_box
    no_object_loss = mse(tf.reshape(non_exists_box*pred_obj, (-1, )), tf.reshape(non_exists_box*true_obj, (-1, )))

    #--------------#
    #| FINAL LOSS |#
    #--------------#

    # Penalize more the box loss and less the no object loss
    total_loss = 5*box_loss + object_loss + 0.5*no_object_loss
    return total_loss
```

Code A.2: Yolo custom loss computation for a single class predictor

```
class DataGenerator(tf.keras.utils.Sequence):
    'Generates data for Keras'

    def __data_generation(self, indexes):
            'Generates data containing batch_size samples'
            # X : (n_samples, *dim, n_channels)
            # Initialization
            X = np.empty((self.batch_size, *self.dim))
            Y = np.empty((self.batch_size,self.S,self.S,5))

            batch_num = 0
            # Generate data
            for i in indexes:
                original_img = load_img(self.path_list[i])
                width, height = original_img.size
                # load the image with the required size
                # and calculate scale factors
                image = load_img(self.path_list[i], target_size=(448,
    448))
                scale_w = 448 / width
                scale_h = 448 / height
                image = img_to_array(image)
                # scale pixel values to [0, 1]
                image = image.astype('float32')
                image /= 255.0
                y_img = np.zeros((self.S,self.S,5))
                for box in self.bboxes_list[i]:
                    xleft = int(box[0] * scale_w)
                    yleft = int(box[1] * scale_h)
                    b_width = int(box[2] * scale_w)
                    b_height = int(box[3] * scale_h)
                    ox = xleft + b_width/2
                    oy = yleft + b_height/2
                    # Calculate the coordinates of the cell in
                    # the grid that contains the center
                    grid_col = trunc(ox/self.cell_size)
                    grid_row = trunc(oy/self.cell_size)
                    # Calculate the coordinates of the center of
                    # the bbox w.r.t the associated cell; (0,0)
                    # top left and (1,1) bottom right corners of
                    # the cell
                    ox_cell = (ox - (grid_col)*self.cell_size)/self.cell_size
                    oy_cell = (oy - (grid_row)*self.cell_size)/self.cell_size
                    # Calculate the width and height of the bbox
                    # in terms of cell size, a bbox of width
                    # 448/S(cell size) will have grid_width = 1
                    grid_width = b_width/self.cell_size
                    grid_heigth = b_height/self.cell_size
                    # Put the results into y; 1 represent the
                    # probability of the class
                    y = [1,ox_cell,oy_cell,grid_width,grid_heigth]
                    y_img[grid_row][grid_col] = y

                # Store sample
                X[batch_num,] = image

                # Store grid
                Y[batch_num,] = y_img

                batch_num += 1

            return X, Y
```

Code A.3: Yolo batch generator for a single class detector