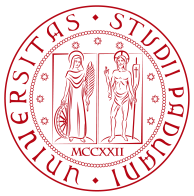


Università degli Studi di Padova

Department of Information Engineering

Master Thesis in ICT for internet and multimedia

A deep learning approach for object counting on embedded systems

Supervisor
Pietro Zanuttigh
Università di Padova

Co-supervisor
Andrea Cisco
E3 S.r.l

Master Candidate
Federico Chiarello

Academic year 2022-2023
Date 12/10/2023

Abstract

This thesis discusses the implementation of a piece counter, based on deep learning methods, to be used on industrial packaging lines. In particular, we aim at obtaining a system capable of detecting small objects (diameter < 1cm) and that can be executed on embedded devices. The object detection models SSD MOBILENET V2 and SSD RESNET 50 , after a brief analysis of their characteristics, result to be the most suitable for this application and their performance is measured by using bolts of different sizes as test object to be detected. In the experimental results, the pros and cons of the two models are analyzed in terms of accuracy, inference time and efficiency. All tests are performed on the development board NVIDIA JETSON NANO in order to optimize models using TensorRT and evaluate the results.

Sommario

In questa tesi viene discussa l'implementazione di un conta pezzi,basato su metodi di deep learning, finalizzato all'utilizzo su linee di confezionamento industriali. In particolare si vuole ottenere un sistema in grado di rilevare oggetti di piccole dimensioni (diametro < 1cm) e che possa essere eseguito su dispositivi embedded. I modelli di object detection SSD MOBILENET V2 e SSD RESNET 50, dopo una breve analisi delle loro caratteristiche, sono risultati i più adatti a questa applicazione e le loro prestazioni sono misurate usando bulloni di diverse dimensioni come oggetto da rilevare. In base ai risultati ottenuti vengono analizzati pro e contro dei due modelli in termini di precisione,tempo di inferenza e efficienza. Tutti i test sono eseguiti sulla scheda di sviluppo NVIDIA JETSON NANO in modo da poter ottimizzare i modelli tramite TensorRT e valutarne i risultati.

Contents

1 Introduction

Computer vision is a task that aims to retrieve meaningful information from digital images or videos. Thanks to the application of the artificial intelligence, and in particular the recent rise of deep learning and neural networks, in the last few years computer vision has improved to such an extent to be used in a lot of applications. In healthcare, inspecting magnetic resonance images, computer vision can guide doctors to a precocious diseases diagnosis[1], in agriculture animal tracking facilitates farmers work[2], and in transportation autonomous vehicle are able to identify pedestrians[3], to give some examples among all possible applications.

The possibility to automate operations like item counting or defect detections has attracted the interest of the industrial world towards computer vision too, in fact relying on machines to perform repetitive and tedious tasks speed up the production process and removes human errors.

In this project, carried out during the internship at the E3 S.R.L company, we focus on one of these industrial scenarios. E3 is a company, founded in 2007 and based in Altavilla Vicentina, that deals with the design and production of custom embedded systems for various applications, some examples are digital signal processing in tempering process, laser driver system for gas measurement in food products and power supplies for plating process. In particular one of the project developed during the years is a weighing system used to count the number of elements in automated packaging lines. Item counters based on weighing systems work well when dealing with few heavy objects, however their accuracy decreases if they have to count a lot of small parts. In recent years, owing to the advancements in image-processing techniques, counters that detect objects by extracting their edges have been developed, but the condition that the objects are clearly distinguished from the background is essential. In contrast, recent deep- learning-based counters have the

advantage of being able to detect objects robustly in any environment, for this reason the purpose of the company , in order to obtain a product suitable for all possible scenarios, is to develop an item counter exploiting machine learning.
This thesis, reporting studies and results obtained during the internship period, wants to be a proof of concept of a deep-learning-based object counter. The core aspect analyzed to determine the feasibility of the project is the relation between the computational capacity requested by deep learning frameworks and the necessary hardware resources to implement it. In fact, one of the main benefits of embedded systems is that they are designed to execute one specific task by optimizing the amount of processing power and memory, making them comparatively affordable. For this reason, to obtain a marketable product, it is crucial to find the best compromise between system performance, in terms of speed, accuracy and efficiency, and the required hardware capabilities.
In the first step of the thesis we study a set of object detection algorithms analyzing their strengths and drawbacks, to determine the most suitable for our application.Then we reduce the system inference time and memory footprint by using TensorRT optimizer, a key tool that optimizes the trained models for execution on NVIDIA GPUs. Subsequently, methodology and datasets used to train and test the selected object detection algorithms are described. Finally, the obtained results are analyzed and commented.

2

Deep Learning for Compture Visions Applications

In this section baselines are briefly introduced starting with a focus on deep learning model that significantly improves the performance of object detection algorithms. We next go into detail about the convolutional neural network architecture that is used to identify features and patterns in an image. The training procedure is finally explained with a particular emphasis on the transfer learning technique.

2.1 Deep Learning

The standard neural network structure is the following:

- **Input layer**: the first layer of the network, it is fed by the input data.
- **Hidden layers**: set of layers where applying a linear combination, and using an activation function, meaningful information is extracted from the input data.
- **Output layer**: it is the last set of neurons, it is used to show the outcome of the network.

A deep network maintains the same architecture but the number of hidden layers is greater, up to a hundred unit. For this reason, the number of variables is orders of magnitude higher than a standard neural network resulting in better performance. The higher degree of freedom exploitable is the strength of the deep learning networks and the reason of their wide use. This feature however has some drawbacks, in

particular it is necessary a large amount of training data in order to avoid overfitting issues and the computational power needed to train and run the model is not negligible. In computer vision are used convolutional neural networks, a type of deep learning architecture particularly suitable to operate with 2D vectors, like images. The major advantage of this network is that automates feature extraction process that is typically involved with machine learning.

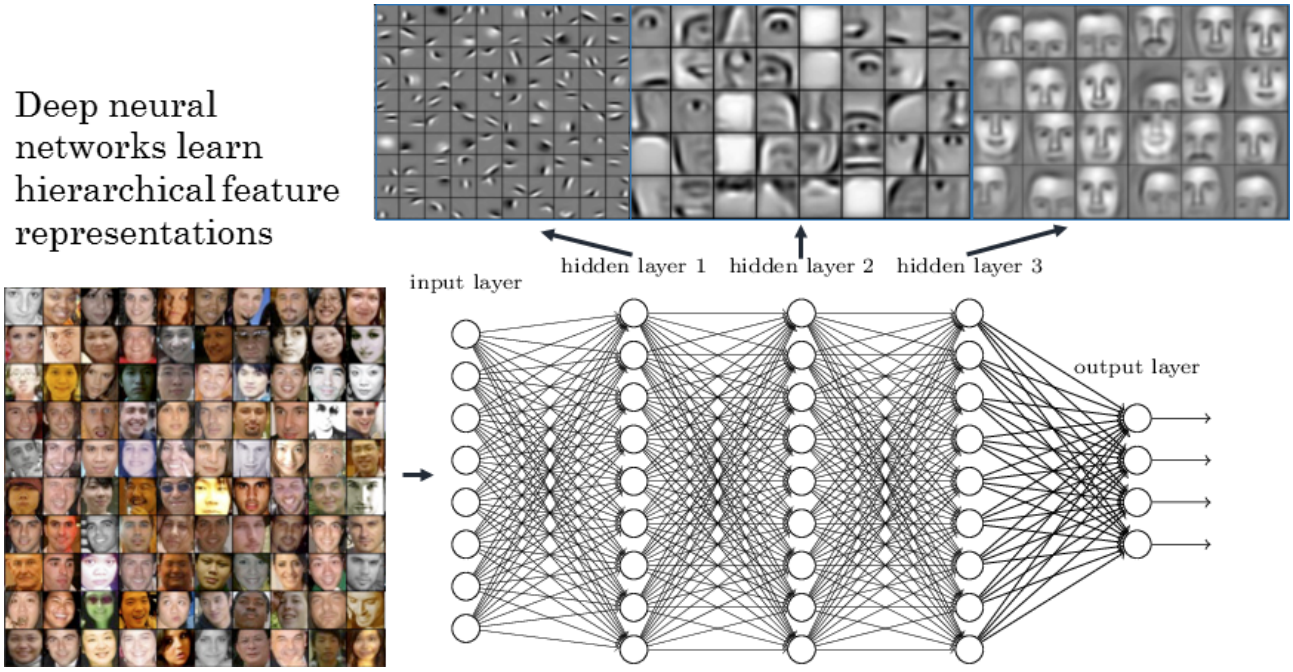


Figure 2.1: Convolutional Neural Network.

2.2 Convolutional Neural Network

Convolutional neural networks(CNNs) are specialized type of artificial neural networks that use convolution in place of general matrix multiplication.[4].
CNNs are particularly suitable for processing data that has grid-like topology, for this reason, since pictures are commonly represented as a 2D grid of pixels, they are widely used in image processing and recognition.
The CNNs architecture is composed by three main type of layers:convolutional layer, pooling layer and fully-connected layer, and it is designed to automatically learn spatial hierarchies of features through a back-propagation algorithm[5] .

Convolutional Layer

The convolutional layer is the core building block of a CNN, and its objective is to extract the high level features from the images. An element-wise multiplication between the input images(denoted as input tensors) and a 2D number array, called kernel, is calculated and the output summed, this process is repeated for each output position. In figure2.2 is shown a graphical presentation of the convolution operation.

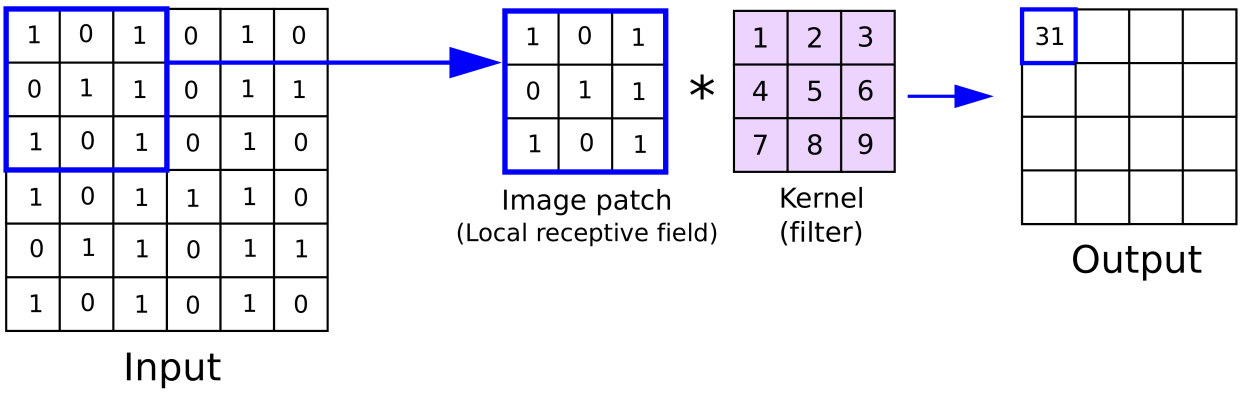


Figure 2.2: Example of convolution with a kernel 3x3.

The size of the convolutional operation output, for an input tensor of dimension $I_w \times I_h \times c$ and a $K_w \times K_h$ kernel, is:

$$O_w = \frac{I_w - K_w + 2P}{S_w} + 1 \quad O_h = \frac{I_h - K_h + 2P}{S_h} + 1 \tag{2.1}$$

Where S_w, S_h are the vertical and horizontal stride that indicate how far the filter moves in every step along one direction. P instead is the padding parameter. If no layer is added to the input image we talk about valid padding and the result of the convolution operation, named feature map, has a reduced size respect to the input tensor. On the other hand when the same padding is used, the feature map has the same size as the input data because the filters are applied to the input tensors with additional rows and columns of zero padding around the edges. This can help to preserve the information at the edges of the image and improve the performance of the CNN, but clearly due to the larger input size the overall computational cost increases. Finally, since the convolution is a linear operator, an activation function is applied to the output to ensure the non-linearity.

The most used activation functions are:

- Sigmoid function:

$$f(x) = \frac{1}{1 + e^{-x}} \tag{2.2}$$

- ReLU function:

$$f(x) = max(0, x) \tag{2.3}$$

(a) Valid padding

(b) Same padding

Figure 2.3: Illustration of the two types of padding

Pooling Layer

The pooling operation is often used after a convolutional layer to further reduce the feature map size. The dimension reduction is translated in several advantages, first the computational power necessary to train the network decreases, secondly the network is more robust to small-shifts and distortions of the input and finally the receptive field is increased. Usually in this layer are applied two types of function:max pooling and average pooling. Max pooling returns the maximum value from the portion of image covered by the kernel, while, for the same area, average pooling outputs the mean of the values. Max pooling also performs noise suppression and for this reason is most commonly used.

Fully Connected Layer

The final stage of the CNNs architecture is usually composed by one or more fully connected layer, the name is due to the fact that every node of the output layer is linked to a node in the previous one. The feature

map of the last convolutional or polling layer is transformed in a one-dimensional vector, in an operation known as flattening, and it passes through one or more fully connected layer to return the final outcome. In the standard approach used to solve the classification task every fully connected layer is linked to the successive one by a learnable weight and the last of these layers has a number of nodes equal to the number of considered classes. Based on the classification task to solve, there are different types of activation functions that are used. In case of binary classification the already cited sigmoid function (eq. 2.2) is used, while when a multi-classification task is treated it is often applied the softmax function which is defined as:

$$f(\boldsymbol{z})_i = \frac{e^{z_i}}{\sum_{i=1}^{K} e^{z_i}} \quad \text{for} \quad i = 1, \ldots, K \quad \boldsymbol{z} = (z_1, z_2, \ldots, z_k) \in \mathbb{R}^K \tag{2.4}$$

Where K is the number of classes and i is the i-th class.
The aim of these activation functions (eq.2.2, eq.2.4) is to normalize the output, in this way each node expresses the probability of the object to be identified to the correspondent class.

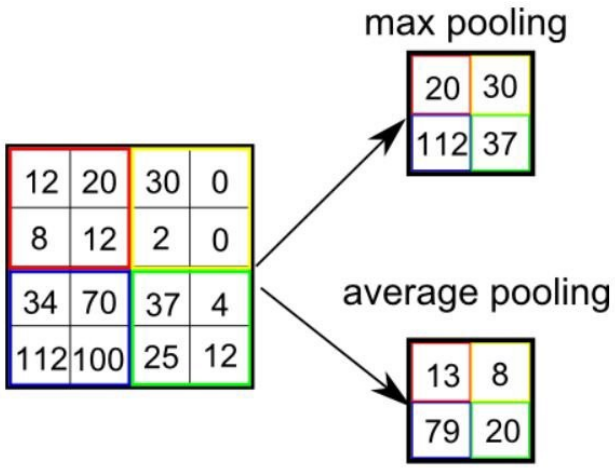


Figure 2.4: Pooling operation with a kernel 2x2

.

2.3 Network Training

The layers that make up the CNNs architecture have a lot of parameters to define, like the convolutional filter values or the weights of the fully connected layer. Training procedure aims to determine the parameter values in order to minimize the difference between the network output prediction and the ground-truth label given in the training dataset. This result is achieved thanks the application of backpropagation and gradient descent algorithms.

Backpropagation

Backpropagation repeatedly adjusts the weights of the connections in the network to minimize the difference between the actual output vector of the net and the desired output vector[6]. The "backward" part of the name stands for the fact that the method starts to calculate the gradient of the error from the final layer and, proceeding layer by layer, it reaches the first one. The algorithm could be divided in two parts:

- **Forward phase:** starting from the first layer to the output layer, for each input-output pair (x_n, y_n), calculates and stores the result $\hat{y}_n$, a_j^k and o_j^k for each node j in layer k.
- **Backward phase:** for each input-output pair (x_n, y_n) calculates and stores the result $\frac{\partial L_{CE}}{w_{ij}^k}$ for each weight w_{ij}^k connecting node i in layer $k-1$ to node j in layer k by proceeding from layer m, the output layer, to layer 1, the input layer.

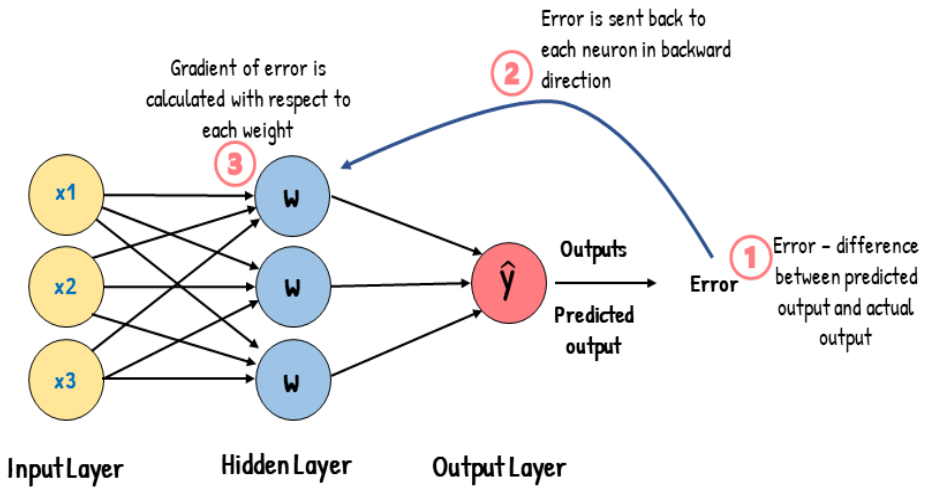


Figure 2.5: Illustration of backpropagation algorithm.

The forward phase is trivial, an input x_n is supplied to the first layer and the output of the network is computed performing the algebraic operations which are based on the network architecture, weights, bias and activation function. The backward phase instead is the core of the algorithm and it is also the reason of its efficiency. The aim of the backpropagation algorithm is to minimize the loss function, in order to do that is necessary to calculate $\frac{\partial L_{CE}}{\partial w_{ij}^k}$ for each w_{ij}^k, if we substitute the cross entropy loss function expression we obtain:

$$\frac{\partial L_{CE}(X,\theta)}{\partial w_{ij}^k} = \frac{\partial}{\partial w_{ij}^k}(-\sum_{n=1}^{N} y_n log(p_n)) \tag{2.5}$$

DEFINITION	EXPRESSION
Weight for node j in layer l_k for incoming node i	w_{ij}^k
Bias for node i in layer l_k	b_i^k
Number of nodes in layer k	r_k
Product sum plus bias for node i in layer l_k	a_i^k
Output of activation function for node i in layer l_k	o_i^k
Activation function	g
Network output for the $n\text{-}th$ input	$\hat{y}_n$
Cross entropy loss function respect to input-output pairs $X = (x, y)$ and set of parameter θ	$L_{CE}(X, \theta)$

Table 2.1: Terminology

where y_n is the truth label and p_n is the probability for the $n\text{-}th$ class. Since the derivative of the sum of function is the sum of the derivative of each function it possible to rewrite the equation (2.5) as:

$$\frac{\partial L_{CE}(X,\theta)}{\partial w_{ij}^k} = -\sum_{n=1}^{N} \frac{\partial}{\partial w_{ij}^k} y_n log(p_n) = \sum_{n=1}^{N} \frac{\partial L_{CE}^n}{\partial w_{ij}^k} \tag{2.6}$$

therefore the final result is obtained optimizing the error function for each input-output pairs separately and subsequently combining it. In the further derivation the subscript n, when not necessary, will be omitted for simplification.

Applying the chain rule, the error partial derivative can be expressed as:

$$\frac{\partial L_{CE}}{\partial w_{ij}^k} = \frac{\partial L_{CE}}{\partial a_j^k} \frac{\partial a_j^k}{\partial w_{ij}^k} \tag{2.7}$$

the first term, named error, is denoted $\delta_j^k \equiv \frac{\partial L_{CE}}{\partial a_j^k}$, while the second term can be computed from the a_j^k equation*

$$\frac{\partial a_j^k}{\partial w_{ij}^k} = \frac{\partial}{\partial w_{ij}^k} \sum_{l=0}^{r_{k-1}} w_{lj}^k o_l^{k-1} = o_i^{k-1} \tag{2.8}$$

*the bias term b_i^k is subsituted with a correspondent weight w_{0j}^k to obtain a more compact formulation

Using the chain rule, the error term is

$$\frac{\partial L_{CE}}{\partial a_j^k} = \sum_{l=1}^{r_{k+1}} \frac{\partial L_{CE}}{\partial a_l^{k+1}} \frac{\partial a_l^{k+1}}{\partial a_j^k} = \sum_{l=1}^{r_{k+1}} \delta_l^{k+1} \frac{\partial a_l^{k+1}}{\partial a_j^k} \tag{2.9}$$

and remembering that

$$a_l^{k+1} = \sum_{j=0}^{r_k} w_{jl}^{k+1} g(a_j^k) \tag{2.10}$$

the final equation is

$$\delta_j^k = \sum_{l=1}^{r_{k+1}} \delta_l^{k+1} \frac{\partial \sum_{j=0}^{r_k} w_{jl}^{k+1} g(a_j^k)}{\partial a_j^k} = \sum_{l=1}^{r_{k+1}} \delta_l^{k+1} w_{jl}^{k+1} g'(a_j^k) \tag{2.11}$$

Putting all together, for a single input-output pair, the partial error derivative can be expressed as:

$$\frac{\partial L_{CE}}{\partial w_{ij}^k} = \delta_j^k o_i^{k-1} = g'(a_j^k) o_i^{k-1} \sum_{l=1}^{r_{k+1}} \delta_l^{k+1} w_{jl}^{k+1} \tag{2.12}$$

From the equation (2.12) it is possible to observe the backward computation of the error, in fact the error δ_j^k is depended on δ_j^{k+1}. Moreover it is appreciable the efficiency of the algorithm, indeed all the value w_{jl}^{k+1}, δ_l^{k+1}, o_i^k and $g'(a_j^k)$ are already computed in the forward phase or in the previous layer partial derivative, and therefore, all the necessary parameters are always set up for the successive operation.

Gradient Descent Optimization

Training a network is a process of finding kernels in convolution layers and weights in fully connected layers which minimize differences between output predictions and given ground truth labels on a training dataset[5]. The difference to be minimized is measured by the loss function, also known as cost function, that expresses how much the output predictions of the network deviates from the given ground truth labels. The learnable parameters are updated through the gradient descent optimization algorithm. The gradient of a function, in fact, indicates the direction of the maximum increasing, therefore updating the parameters in the opposite direction the loss function will be minimized. A single update of a parameter is formulated as follows:

$$w := w - \alpha \frac{\partial L(X, \theta)_{CE}}{\partial w} \tag{2.13}$$

where α is the learning rate and it is one of the most important hyperparameters to be set in the training process. We can notice that to apply the gradient descent method is necessary to calculate the partial derivative of the loss function and it is here that the backpropagation algorithm previously described plays its crucial role. All the process is repeated for a set number of times, defined by a hyperparameter called step, and for each iteration the learnable parameters are updated in order to minimize the loss function. It is important to observe that in most of the cases it is used slight different version of gradient descent algorithm called mini-batch gradient descent. The difference between the two methods is that in mini-batch gradient descent only a subset of the training set is used to compute the gradient of the loss function, this is done for memory limitations or to accelerate the training process. The dimension of the subset, called mini-batch, is chosen to obtain a trade-off between optimization stability and computational cost.

Transfer Learning

A deep learning model requires huge dataset to be trained efficiently, and consequently a lot of computational power. These resources are usually own only by big companies and universities, therefore it's rare for a standard user to train a model from scratch. The common procedure indeed it is to use the transfer learning paradigm in order to reduce the size of the training dataset and at the same time speed up the training process. Transfer learning is a machine learning technique where the knowledge a model has acquired by training on a specific task is applied to solve a new problem. The base idea of this approach is that low-level features such as edges, shapes, corners and intensity can be similar along different tasks, hence a pre-trained neural network, that is obtained using a large amount of data, can be used as starting point to train a specific purpose model. The two most common transfer learning strategies are:

- **Feature Extraction**: all the layers of the pre-trained neural network are frozen(not updated during the training), but the last fully connected layers. In such a way the previous knowledge is used to extract features from the new dataset images and only the classifier part is trained accordingly to the specific task. In some cases instead of using only a fully connected layer, a simple linear classifier is added to the features extractor.
- **Fine Tuning**: in this strategy not only the pre-trained network output classifier is replaced, but also some of the previous layers are selectively retrained on the new dataset. In a convolutional neural network the first layers are generally used to capture generic features, that could be useful to many tasks, the later ones instead become progressively more specific. For this reason it is important to decide carefully, based on the size of the available dataset, if retraining the entire network or only the last portion, in order to avoid overfitting issues.

Reusing information from previously learned tasks to new tasks has the potential to significantly improve learning efficiency[7], the pretrained models released from big companies and major universities in fact are trained on thousands and thousands of images and therefore they have learned efficiently how to extract the generic features, the main goal of the first portion of a deep learning network. The second advantage of transfer learning strategies is that the training process involves only a subset of the total amount of the network weights therefore it is more rapid.

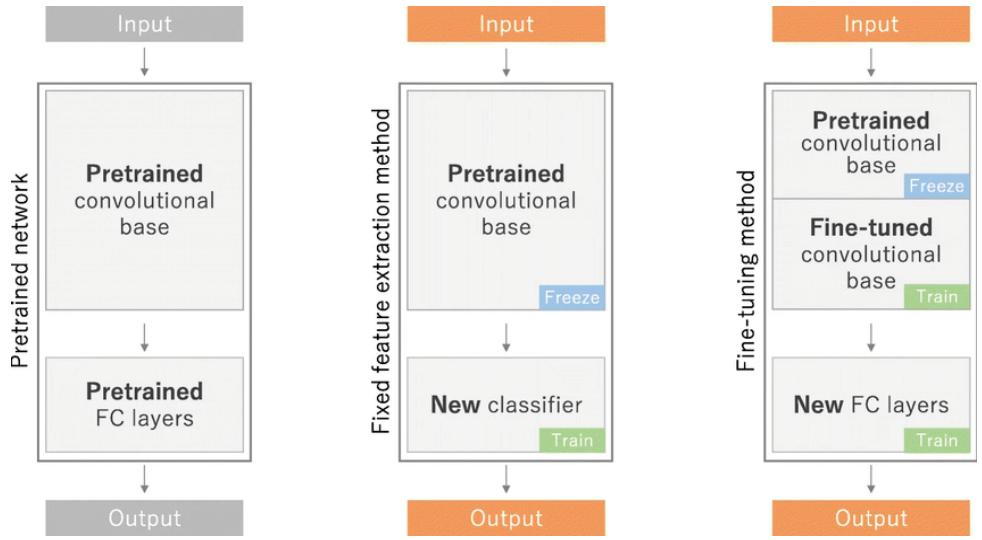


Figure 2.6: Feature Extraction vs Fine Tuning.

3
Object Detection

Object detection is the task of identifying objects of interest within a picture. This task involves image classification to predict the class of the objects and object localization to determine their position in the image. Object detection is the base to build more complex tasks, such as segmentation or object tracking, and therefore it is in continuous evolution. Over the years several object detection algorithms have been developed and each of these has brought progresses in terms of efficiency, speed or accuracy.
In this section we will give a generic overview of the most important object detection frameworks in order to determine, based on their qualities and weaknesses, their most suitable application. Firstly the two-staged based models will be briefly described, understanding why they are inadequate for the purpose of this thesis. Then, the chapter continues with an in-depth analysis of the one-stage based models, in particular of the SSD (Single Shot Detector) algorithm.

3.1 Object Detction Problem Definition

Before starting the description of the object detection frameworks it is useful to formally define the task we have to solve. A set of N images $\{x_1, x_2, ..., x_N\}$ and the corresponding annotations are given as input. The annotation of the $i-th$ image containing M_i objects belonging to C classes has the following format:

$$y_i = \{(c_1^i, b_1^i), (c_2^i, b_2^i), ..., (c_{M_i}^i, b_{M_i}^i)\} \tag{3.1}$$

where $c_j^i \in C$ is the class of the object and b_j^i its bounding box. The aim of the detector, parametrized by

θ, is to generate predictions with the same format:

$$y^i_{pred} = \{(c^i_{pred_1}, b^i_{pred_1}), (c^i_{pred_2}, b^i_{pred_2}), ...,)\} \tag{3.2}$$

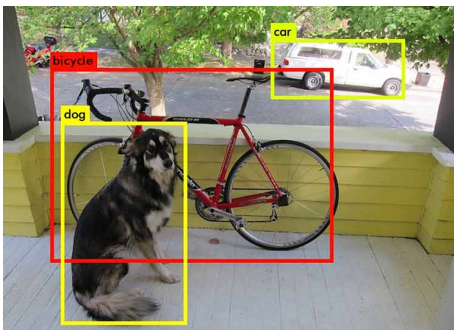
that minimize a loss function l defined as:

$$l(x, \theta) = \frac{1}{N} \sum_{i=1}^{N} l(x_i, \theta, y_i, y^i_{pred}) + \lambda \sum_{j=1}^{R} \theta_j^2 \tag{3.3}$$

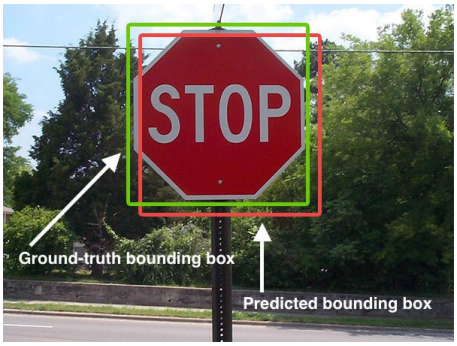
Finally, during the evaluation of the model, an object is considered correctly detected if it is assigned the proper label and the intersection-over-union (IoU) index is greater than a threshold T. The IoU index

$$IoU(b^i_{pred}, b^i_{gt}) = \frac{Area(b^i_{pred} \cap b^i_{gt})}{Area(b^i_{pred} \cup b^i_{gt})} \tag{3.4}$$

measures how well the predicted bounding box (b^i_{pred})covers the ground truth one (b^i_{gt}). Others very common metrics used in the evaluation of the object detection framework are mean average precision (mAP) for the accuracy and frame per second (FPS) for inference time.



(a) Object Detection example



(b) Illustration of the IoU index

Figure 3.1

3.2 Two-Stage Based Model

In 2014 Ross Girshick proposed the *R-CNN* object detection algorithm[8] that significantly improves the detection performance compare to the frameworks used at the time. The idea at the base of this approach

is to divide the detection process in two steps. In the first stage, through the selective search algorithm[9], the model identifies portion of image where is likely to find a particular object, in the following step a CNN takes as input each of these regions, known as "region proposals", and generates the feature vectors that feed an SVM classifier. Finally the extracted features are used by a bounding box regressor to refine the original proposal coordinates.

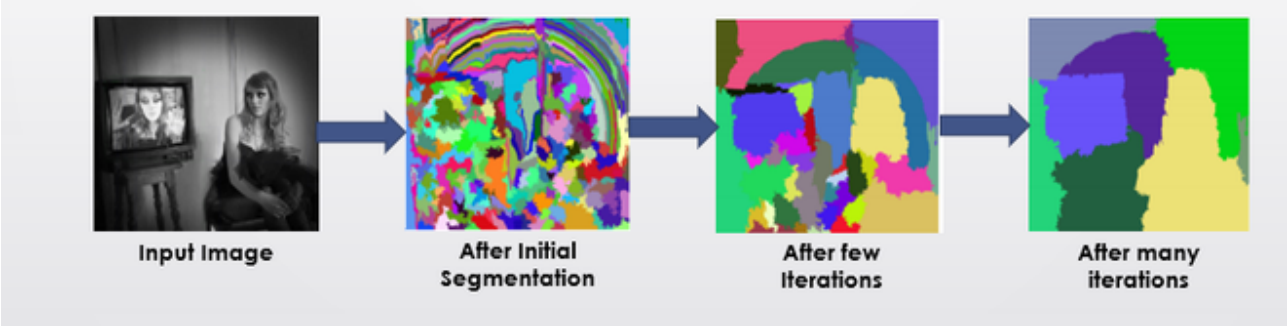


Figure 3.2: Example of selective search algorithm.

The main drawback of this approach is that the CNN has to process every region proposal, and since the selective search algorithm generates 2000 of them computation and memory usage become really large. For this reason in 2015 the same Girshick proposed an *R-CNN* improved implementation to tackle the problem, the *Fast R-CNN*[10]. In this second version feature map is computed for the whole input image and fixed size region features are extracted from it. The region dimension is reshaped by the Region of Interest (RoI) polling layer. The RoI pooling layer is a special case of the spatial pyramid pooling (SPP) layer with just one pyramid level, its function is to partition the region proposal retrieved from the feature map into a fixed number of divisions and for each of them performing a max pooling operation. The extracted features vectors then feed a set of fully connected layers before two siblings output layers: a softmax classifier and a bounding box regressor. Fast R-CNN achieves significantly faster inference time since it performs the convolutional operation only once instead for each region proposal avoiding duplicate computations. In addition, unlike R-CNN, in Fast R-CNN feature extraction, region classification and bounding box regression steps are optimized all together increasing accuracy and without using extra cache space to store the features. Despite of the considerable improvements in terms of speed and accuracy Fast R-CNN architecture still uses the selective search algorithm to generate the region proposals, and that introduces two remarkable limitations:first it is a time-consuming process based on CPU architecture therefore it can't exploit GPUs parallel computation, and then the selective search algorithm is not optimized in data-driven-manner like the rest Fast R-CNN pipeline thus it could produce misleading proposals. To face this issues, Shaoqing Ren has developed the *Faster R-CNN* [11], in which a neural network is used as region proposals generator in

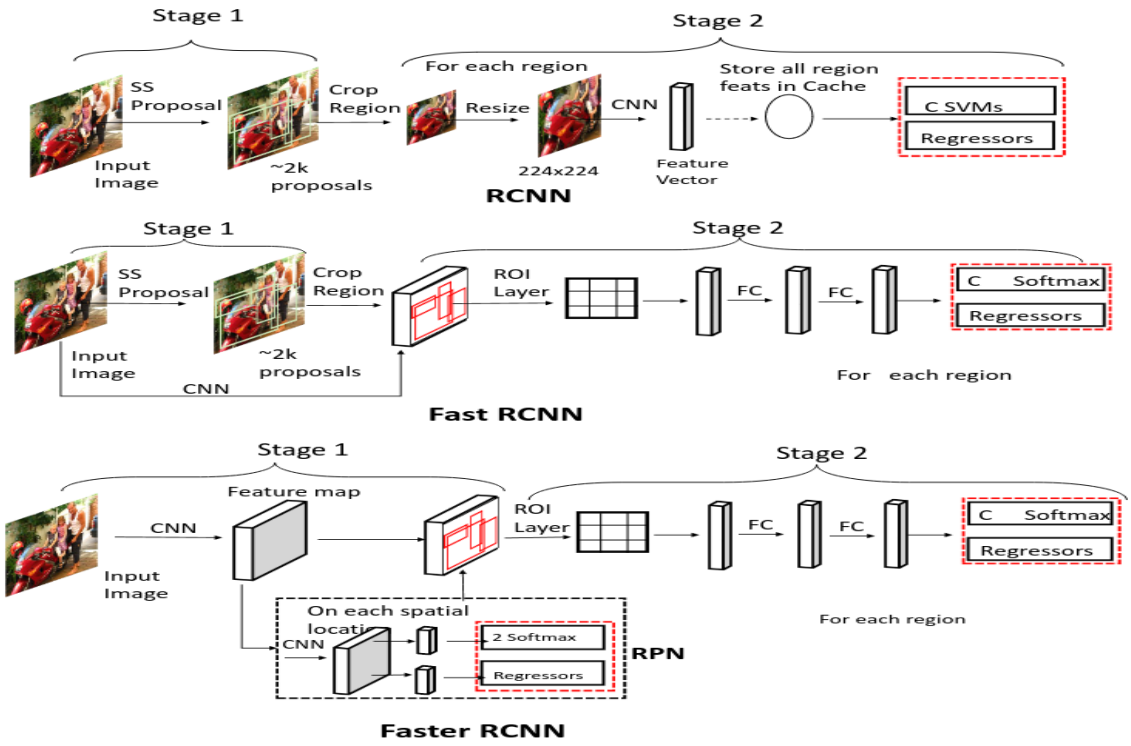


Figure 3.3: R-CNN vs Fast R-CNN vs Faster R-CNN.

place of the selective search algorithm. This new generator, named regional proposal network (RPN), is a fully convolutional network which takes an image of arbitrary size and generates a set of object proposals on each position of the feature map. The RPN mitigates the Fast R-CNN problems since it can be learned via supervised learning permitting an end-to-end optimization and the computing time is sharply reduced. The current version of Faster R-CNN is capable to reach several FPS on GPUs and it achieves 70% mean average precision (mAP) on the public benchmark dataset PASCAL VOC 2007.

System	Time	07 data	07 + 12 data
R-CNN	~ 50s	66.0	-
Fast R-CNN	~ 2s	66.9	70.0
Faster R-CNN	**~ 198ms**	**69.9**	**73.2**

Figure 3.4: Inference time and accuracy comparison between R-CNN architecture version on PASCAL VOC 2007 and 2012 dataset.

3.3 Single Shot Multibox Detector

The two-stage based models have definitely permitted to achieve remarkable results in terms of accuracy. However, the complex pipeline of these methods is too computationally intensive for embedded systems and only with modern, and expensive, high-end hardware suitable for real time applications. These limitations have been faced using a different approach, based on a single-stage architecture, whose most famous implementations are *SSD*(Single Shot Multibox Detector)[12] and *YOLO*(You Only Look Once)[13]. SSD and YOLO methods share the same core idea, instead of generating proposal regions and then applying a classifier on them, like R-CNN models , object detection is treated as a regression problem in which spatially separated bounding boxes and associated class probabilities are learned. The advantage of this approach is that all the computation is encapsulated in one single neural network resulting in a faster detection and an easier training process.

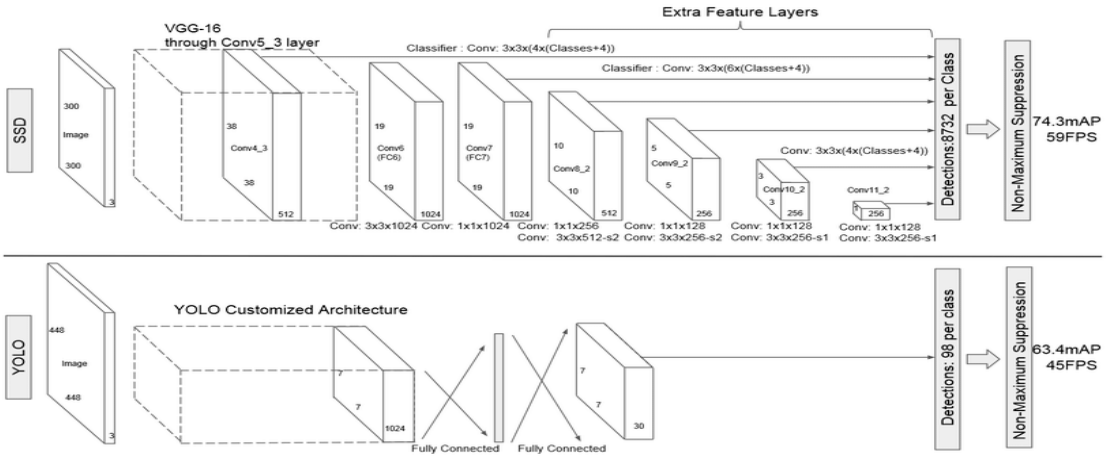


Figure 3.5: Comparison between YOLOv1 and SSD architectures.

SSD architecture is composed by a feature extractor in the first portion, called base network, and then by a set of convolutional layers that generates output grids of different size. A high quality image classifier, deprived of any classification layers, performs the feature extraction process, in the original *SSD* implementation is used *VGG-16*, but it can be replaced by other classification models making possible architecture customization. At the end of the base network the feature map passes through the extra convolutional layers that are designed to obtain:

- **Multiscale feature maps**: because of convolutional operation, feature maps at different level of the network have different receptive field. Receptive field is defined as the region in the input image that a particular feature map is looking at. Feature map dimension decreases after each convolutional layer,

therefore one single cell represents a wider area of the starting image. SSD algorithm uses this property to execute multiscale predictions. The first feature maps, which a higher resolution are used to detect small objects, while large ones are identified in the last level of the network.

- **Detection predictions**: in order to determine position and class of the object a set of small convolutional filters is applied on each entry of the extracted feature maps, in this way every cell is used for the detection in the nearby region of the image. Assuming a feature map of size $mxnxp$ for each output value to predict, therefore $N + 1$(one more for background case) class probabilities and 4 bounding box coordinates, it is applied a $3 \times 3 \times p$ filter.

Feature maps grid indicates to the detector where to search, but it doesn't give any information about the shape of the objects to find. If we use the standard deep learning approach, starting with a random bounding box prediction, model training will struggle to determine which of several object shapes it has to optimize. *SSD* solves the problem associating to the feature map cells a set of default bounding boxes, each of them is centered within the cell and has different aspect ratio and sizes that are modified by the 3×3 filters described before in order to optimally fit the object to detect. Default boundary boxes scale is different for each level of the network, since, as we have seen, *SSD* model is designed to perform multiscale predictions. The scale of the default boxes is linearly spaced along the m levels of the added convolutional layers:

$$s_k = s_{min} + \frac{s_{max} - s_{min}}{m - 1}(k - 1), k \in [1, m] \tag{3.5}$$

where $s_{min} = 0.2$ is the scale of the first layer and $s_{max} = 0.9$ is the ones of the last layer. The aspect ratios considered instead are $a \in (1, 2, 3, \frac{1}{2}, \frac{1}{3})$ from which is possible to obtain bounding box width and height as:

$$w_k^a = s_k \times \sqrt{a} \quad h_k^a = \frac{s_k}{\sqrt{a}} \tag{3.6}$$

aspect ratio 1 is computed also in a different scale

$$s'_k = \sqrt{s_k \times s_{k+1}} \tag{3.7}$$

resulting in a total of 6 default bounding boxes per cell. The overall amount of output value computed for each feature map therefore are $(((N + 1) + 4) \times 6) \times m \times n$ (except the first and last layer that use only 4 default bounding boxes).

SSD training process is divided in two phases, during the first step is necessary to pair a ground truth bounding box j with a default bounding box i, a matching is considered as positive if the IoU index between the two boxes is higher than a threshold, more formally is defined as:

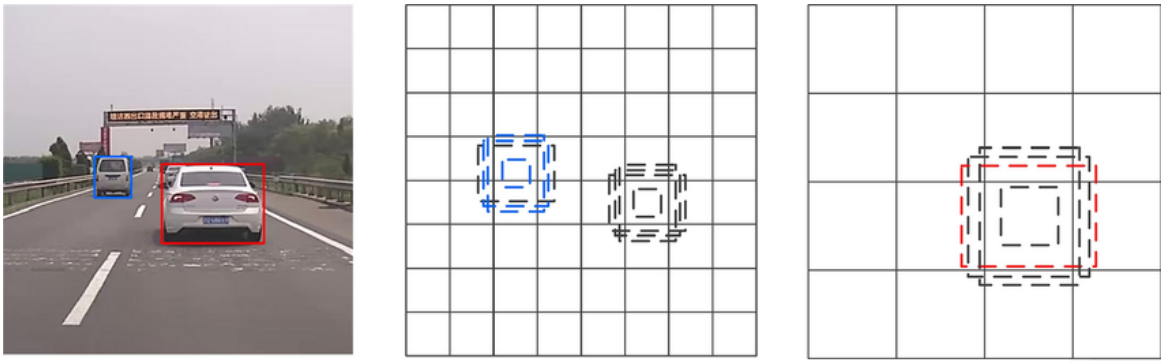


Figure 3.6: Prespective changes the scale of the objects that are hence detected using different default boxes and feature maps.

$$x_{ij} = \begin{cases} 1 & \text{if IoU} \geq T \\ 0 & \text{otherwise} \end{cases} \tag{3.8}$$

in case of positive match SSD model penalizes the difference between predicted offset and ground truth ones according the localization function

$$\begin{aligned} L_{loc}(x, l, g) &= \sum_{i \in Pos}^{N} \sum_{m \in (cx,cy,w,h)} x_{ij}^{k} smooth_{L1}(l_i^m - g_j^m) \\ \hat{g}_j^{cx} &= \frac{(g_j^{cx} - d_i^{cx})}{d_i^w} \quad \hat{g}_j^{cy} = \frac{(g_j^{cy} - d_i^{cy})}{d_i^h} \\ \hat{g}_j^{w} &= log(\frac{g_j^w}{d_i^w}) \quad \hat{g}_j^{y} = log(\frac{g_j^h}{d_i^h}) \end{aligned} \tag{3.9}$$

where $\hat{g}_{cx}, \hat{g}_{cy}, \hat{g}_w, \hat{g}_h$ are the offsets for the center (cx, cy), width w and height h of the default bounding box d respect to the ground truth box g. The predicted offsets instead are indicated as l_{cx}, l_{cy}, l_w, l_h.
In terms of classification error positive matches are penalized by the probability assigned to the corresponding ground truth class $\hat{c}_i^p$, while for negative matches it is used the background class $\hat{c}_i^0$. The resulting confidence loss is:

$$L_{conf}(x,c) = -\sum_{i \in Pos}^{N} x_{ij}^{p} log(\hat{c}_i^p) - \sum_{i \in Neg} log(\hat{c}_i^0)$$
$$\text{where} \quad \hat{c}_i^p = \frac{exp(c_i^p)}{\sum_p c_i^p} \tag{3.10}$$

The overall loss function minimized in the training process is the weighted sum of confidence and localization loss:

$$L(x,c,l,g) = \frac{1}{N}(L_{conf}(x,c) + \alpha L_{loc}(x,l,g)) \tag{3.11}$$

4

TensorRT Optimizer

The request for more and more accurate models has led to a significant rise of neural network complexity and dimension. Working with embedded systems available hardware resources are limited, therefore become essential to exploit hardware acceleration in order to obtain low response time and contained memory footprint. To answer this needed NVIDIA develops TensorRT[14], an SDK specifically designed for its GPUs that optimizes deep learning models trained in any framework. TensorRT structure and principal optimization operations will be described in the following of this chapter.

4.1 TensorRT Structure

TensorRT includes both deep learning inference optimizer and runtime, the two separated working mode are defined respectively as build and deployment phases. The build phase imports a trained model and based on several parameters, such as target deployment GPU, batch size, working memory and quantization, performs the model optimization. The output of this phase is an optimized inference engine that is saved in a serialized file called plan. TensorRT has been developed to working with any deep learning framework, for this reason the most common platforms like TensorFlow and PyTorch are integrated with TensorRT by high-level APIs that facilitate import operation. However, given the deep learning frameworks variety and to favor interoperability is a good procedure to convert trained model in the *ONNX* format before starting the build phase. Open neural network exchange(*ONNX*)[15] is an open format that provides a common language any machine learning framework can use to describe its models.

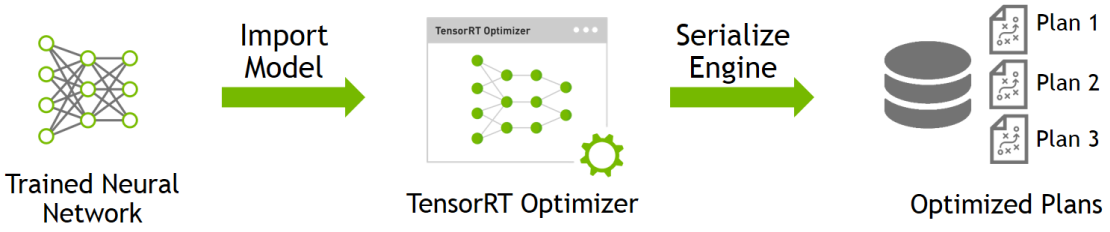


Figure 4.1: TensorRT build phase.

In the deployment phase the plan file generated in the previous step is deserialized to create a TensorRT engine object that is used to run inference on the target platform. The advantage of working with plan files is that inference devices don't have to install any deep learning frameworks.

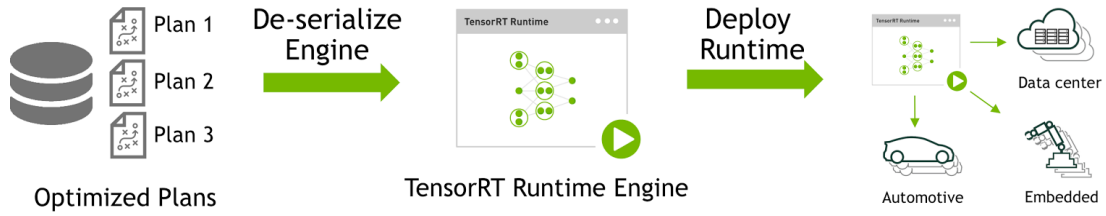


Figure 4.2: TensorRT deployment phase.

4.2 TensorRT Optimizations

TensorRT applies transformation to neural network structure and uses memory management techniques in order to optimize the imported model. Since the optimization operations are designed for the specific hardware architecture it is important that the build phase is run in the target platform. The main optimizations performed are:

- Precision reduction
- Layer and tensor fusion
- Target specific auto-tuning
- Dynamic tensor memory
- Multi-stream execution

Precision Reduction

During training deep learning frameworks use 32-bit single-precision to achieve the highest possible accuracy in gradient descent and backpropagation algorithms. However, it is experimentally proven[16] that inference phase requires less numeric precision, with some attentions it can work in FP16 half-precision, or even INT8 precision, without affect heavily accuracy. In case of reduction to INT8 precision, since the range is very tight respect FP32, TensorRT requires a representative input dataset to determine the best weights representation. Precision reduction optimization allows to noticeably decrease the model dimension and to obtain lower inference time.

Layer and Tensor Fusion

TensorRT optimizer analyzes neural network structure searching for pattern of layers that can be optimized. This operation modifies the network architecture obtaining a lighter and more efficient model, but the underlying computation remains the same. Firstly TensorRT performs vertical layer fusion of very common set of operations that are applied one after the other, such as convolution, bias and ReLU. This fusion into one single layer allows to reduce kernel launches and avoids writing into and reading from memory between layers, operations that in several cases are even slower than kernel computation, becoming the pipeline bottleneck. Another type of fusion is layer aggregation, or horizontal fusion, this operation merges layers that apply the same operation to the same source tensor but with different weights. The output of the combined layer is disaggregated to feed different portion of the network. Horizontal fusion decreases memory overhead since the input tensor are copied from host to GPU only once resulting in further improving model efficiency.

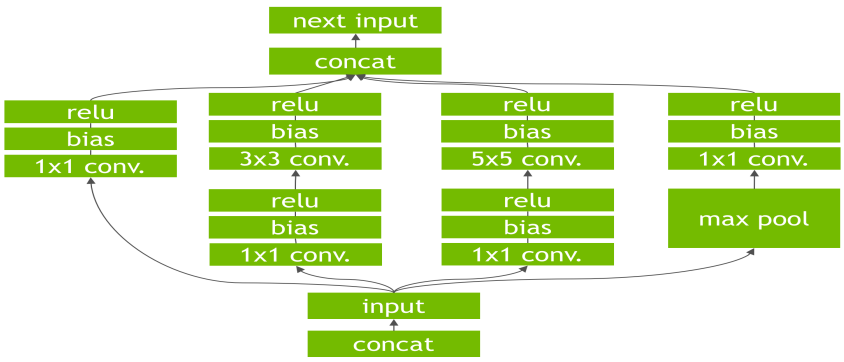
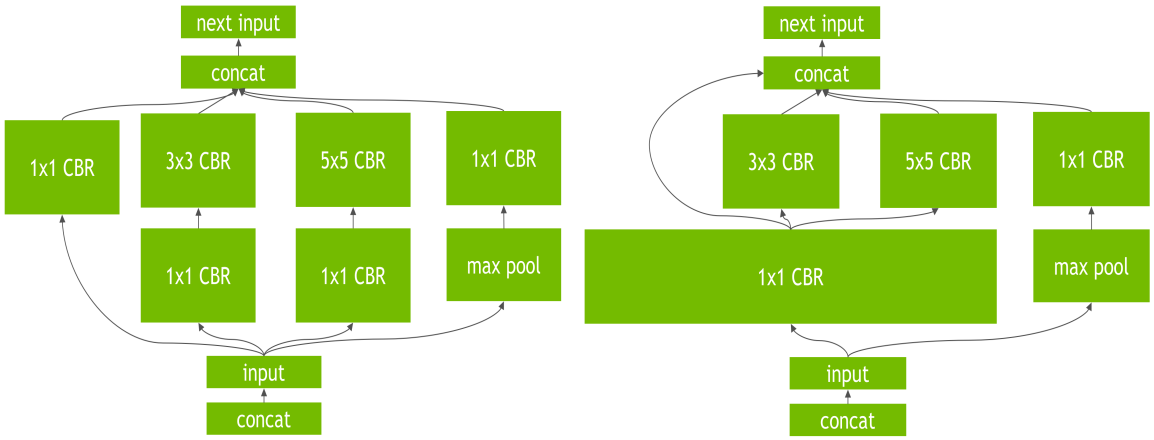


Figure 4.3: Original network before fusion optimization.



(a) Vertical fusion

(b) Horiziontal fusion

Figure 4.4: An example of vertical and horziontal layer fusion

Target Specific Auto-tuning

The same operation can be executed in different manner, TensorRT exploits this property in the optimization process. As an example, convolution operation can be performed by several algorithms[17], but some implementations work better than other based on hardware architecture and working parameters. For this reason TensorRT selects hand-tuned and optimized implementations for each layer based on the target GPU, input data, batch size and filter dimension in order to ensure the highest possible performance.

Dynamic Tensor Memory

Model memory footprint is reduced allocating memory for each tensor only for its usage time and identifying chances of memory re-use. It helps to optimally exploit the available memory, a crucial operation in embedded platform.

Multi-stream Execution

TensorRT exploits parallelism computations processing multiple CUDA streams simultaneously. CUDA (or Compute Unified Device Architecture) is a parallel computing platform and application programming interface (API) that allows software to use certain types of graphics processing units (GPUs) for general purpose processing [18]. In the standard serial CUDA approach the computation is divided in three steps:first input memory is copied from host to the device then it executes the kernel and finally output memory is

transferred back from device to host. This process is defined as CUDA stream. TensorRT enables to execute multiple streams at the same time dividing the memory in chunks, in this way while a piece of memory is on transfer from host to the device a kernel execution is running and another portion of memory is copied from device to host. With this approach three steps of different streams are processed concurrently.

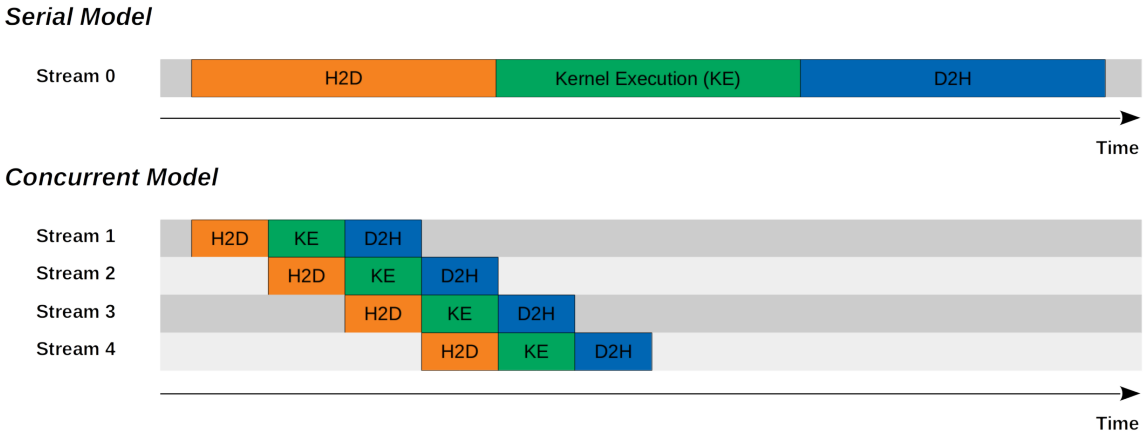


Figure 4.5: Cuda stream,serial and concurrent mode[19].

5 Experimental Setup

In this section are discussed the project objective and the working environment, it starts with a brief description of the hardware and software tools used to obtain the results then it continues introducing models and datasets employed during the test sessions.

5.1 Project objective

The objective of the internship period, and therefore the one of this thesis project, is to study the feasibility of a deep learning-based object counter on an embedded system. More in the details the final system has to be suitable to counting object of small-size in automated packaging lines. E3 company has several years of experience in the design, develop and implementation of embedded systems, however it is still taking its first steps on the fields of machine learning and artificial intelligence, that is why it wants to analyze the potentialities and limitations of this new project before starting to invest in technological resources and workforce to develop it. In particular the purpose is to measure the performance in terms of:

- **Frame rate** : we want to obtain a system that can detect in real-time the objects of interest.
- **Accuracy detection** : the usage of the deep learning approach has to ensure a low error-rate ($5\% <$) and a good localization of the object(IoU > 0.8).

The results are collected using the bolts as example of object to detect, but the consequent considerations want to be extendable, as much as possible, to all small-size objects in order to achieve a complete overview of the problem.

5.2 Setup

TensorFlow

TensorFlow[20] is an open source framework for machine learning with a particular focus on training and inference of deep neural networks. It has been released in 2015 by Google as successor to DistBelief[21], a closed-source library used internally at the company, and currently it is one of the most popular machine learning frameworks. The main reasons that bring to the decision to use TensorFlow in this thesis project are:

- **Scalability**: TensorFlow optimally performs training procedure on GPUs cluster, but it is also very flexible in inference mode, in fact it can run trained models on various platforms, ranging from large distributed clusters in a datacenter, down to running locally on mobile devices[22].
- **Open source**: TensorFlow is an open-source library therefore it has a good documentation and an active community working on it. Moreover, it offers a collection of pre-trained detection models that are useful for initializing new ones when training on novel datasets.
- **Object Detection API**: TensorFlow provides specific object detection API, that makes it easy to construct, train and deploy object detection models.

Training Hardware

Deep neural networks have permitted remarkable improvements in many AI applications, including computer vision, but the performance growth has come at a cost:a significant computational demand. For example, ImageNet 2012 challenge winner AlexNet[23] (84.7% accuracy) takes 1.4GOPS to process a single 224×224 image, while ResNet-152[24] that won the 2015 edition (96.5% accuracy), takes more than an order of magnitude more computations, 22.6GOPS.In particular, for any deep learning model the training phase is the most resource-intensive task and almost in every case requires GPUs high-parallel computations capabilities. The traditional Central Processing Unit (CPU) handles all the main functions in computer hardware architectures and has few powerful cores (usually from 2 to 64) that allows to execute sequential tasks efficiently. Instead, Graphic Processing Units (GPUs), originally designed for computer graphic purpose, are specialized components with thousands of cores that excel in repetitive and highly-parallel computing tasks that makes this hardware particularly suited to run deep learning algorithm based on matrix multiplications and convolutions. Unfortunately, E3 S.r.l company doesn't have such hardware, therefore we make use of

a cloud GPUs platform to work around the problem. Cloud GPUs are platforms that provide hardware acceleration for an application using GPUs on a cloud server, several big companies offer this service such as Microsoft with Azure, Amazon with AWS and Google with Colab, the latter is the one used in this project as it provides the most beginner-friendly environment. A brief benchmark test is conducted to estimate GPUs computation advantages in our specific case, and in this way evaluate possible company investments in this technology in the future. The model trained in the test is InceptionV3[25] at the end of which a small neural network classifier is added, this is done to retrieve results with different amount of trainable parameters using InceptionV3 as feature extractor or fine-tuning the entire model.

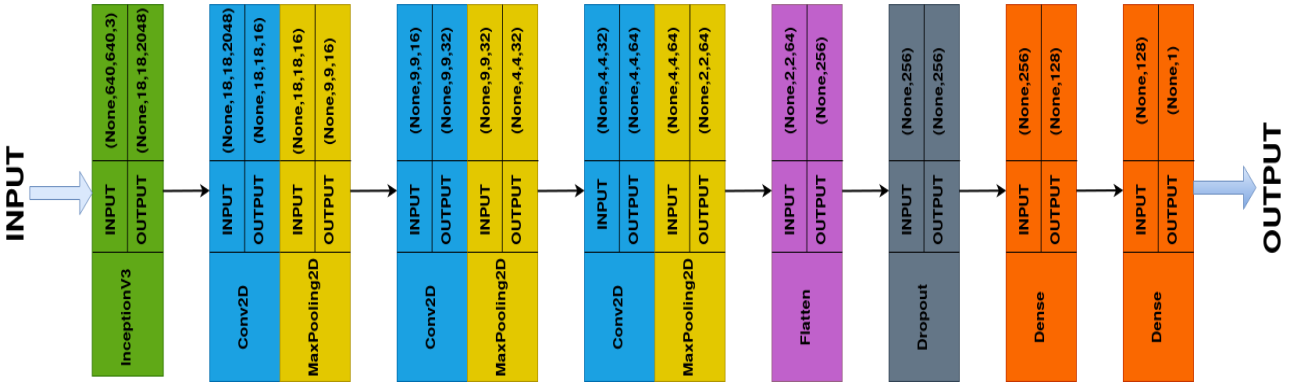


Figure 5.1: Model tested.InceptionV3 architecture has 25M trainable parameters while the added classifier 350K.

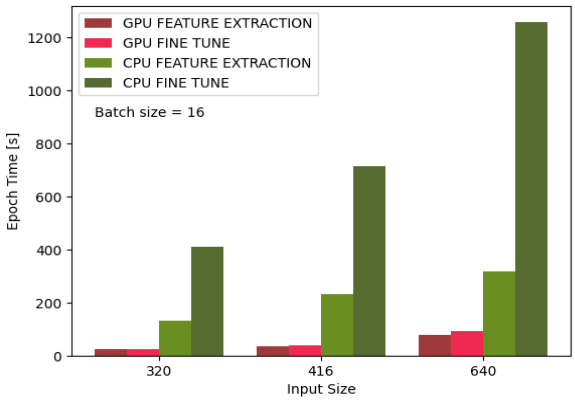


Figure 5.2: Training times.

The results in figure 5.2, obtained for different image input size, depict the training time using AMD Ryzen 5500 CPU and NVIDIA Tesla T4 GPU, the standard one available in Colab, and they confirm as CPUs training is feasible only for very small models and using low image resolution.

Component	Cores	Clock Speed	Memory	Price	Speed
AMD RYZEN 5500	6	4.2 GHz	System Memory	100$	434 GFLOP in FP32
NVIDIA Tesla T4	2560	1.6 GHz	16GB GDDR6	2000$	8.1 TFLOP in FP32

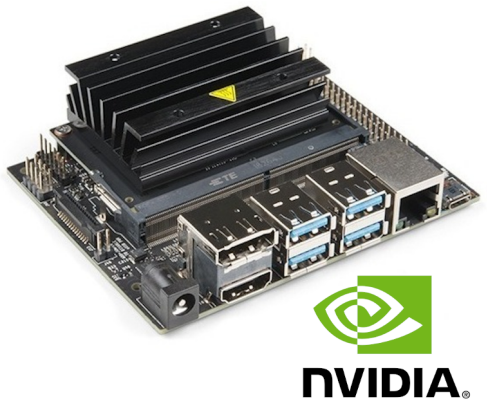
Table 5.1: Hardware used for training time comparison

Inference Hardware

There are many single-board computer to develop AI on embedded systems[26], but NVIDIA Jetson is without a doubt the world's leading computing platform, it offers a set of edge devices that for their high-performance, form factor, low power consumption, and advanced thermal management system are ideal for various industrial applications. These reasons, joined to the possibility to exploit TensorRT optimization on NVIDIA GPUs, lead to adopt a Jetson platform also in our work case. In this project we use the Jetson Nano Development board, however Jetson family comprehend more powerful solutions like Jetson Orin and Jetson Xavier that can be considered for future improvements.

AI performance	472 GFLOPs
GPU	128-core NVIDIA Maxwell™ GPU
GPU Max Frequency	921 MHz
CPU	Quad-core ARM® Cortex® A57 MPCore processor
CPU Max Frequency	1.43 GHz
Memory	4 GB 64-bit LPDDR4 25.6 GB/s
Storage	16 GB eMMC 5.1
Power	5W-10W

(a) Jetson Nano specification



(b) Jetson Nano board

Figure 5.3: Jetson Nano board and specification

5.3 Dataset

The training dataset is one of the main factors that determines deep neural network performances [27]. A high-quality, large-size dataset is crucial to obtain an accurate model and complete the final task successfully. To be as close as possible to these features the collected samples must be uncorrelated to each other and in a plentiful number, however the creation of such a dataset is not trivial since it requires a lot of time and effort in the process of retrieving and annotating the necessary images. For this reason, techniques as data augmentation and transfer learning are widely used to reduce the cardinality of the training dataset and therefore lighten the workload. Following these considerations we built a training dataset of 500 images, using the bolt as element to detect, although there is no theoretical results that say which is the optimal dataset dimension we have empirically verified that, for our specific case, a drop in performance occurs below this value. We decide to use the bolt as case of study, among all the possible alternatives, principally for two reasons, first it is a classical example of element that is difficult to count precisely with the standard weighing approach which determines the number of element dividing the total measured weight by the weight of a single item, and then because bolts are produced in various forms factor therefore we can analyze the system performance with the same object taken in different size. In order to do that two test dataset are collected:

- **DATASET A**: Contains 100 images of bolts M3 , 5.5mm in diameter
- **DATASET B**: Contains 100 images of bolts M6 , 10mm in diameter

To fairly compare the results the sample of both dataset are acquired with the same device (Samsung S10e camera) and at the same distance (30cm).

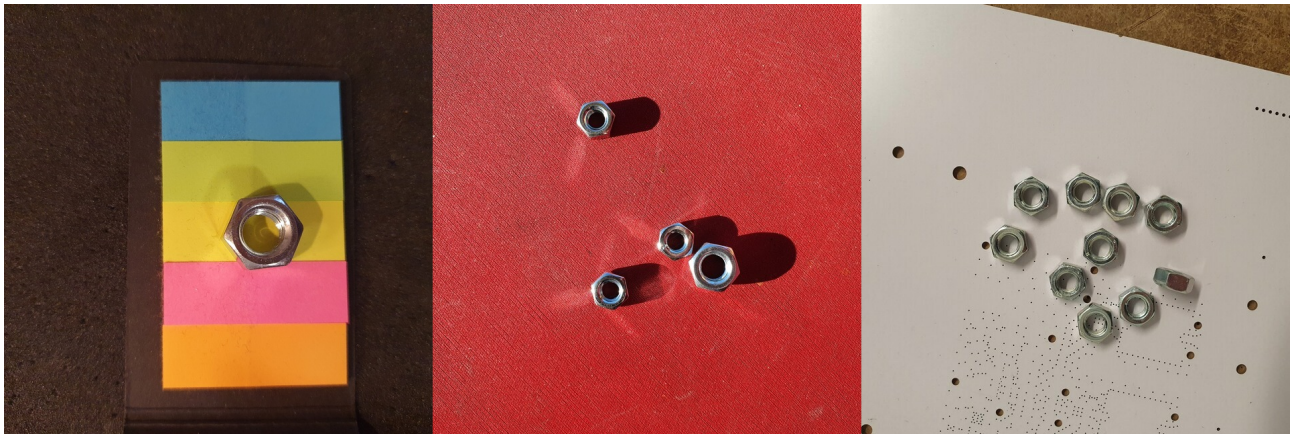


Figure 5.4: Training dataset samples. The samples are the most possible uncorrelated to achieve a good generalization of a real world scenarios.

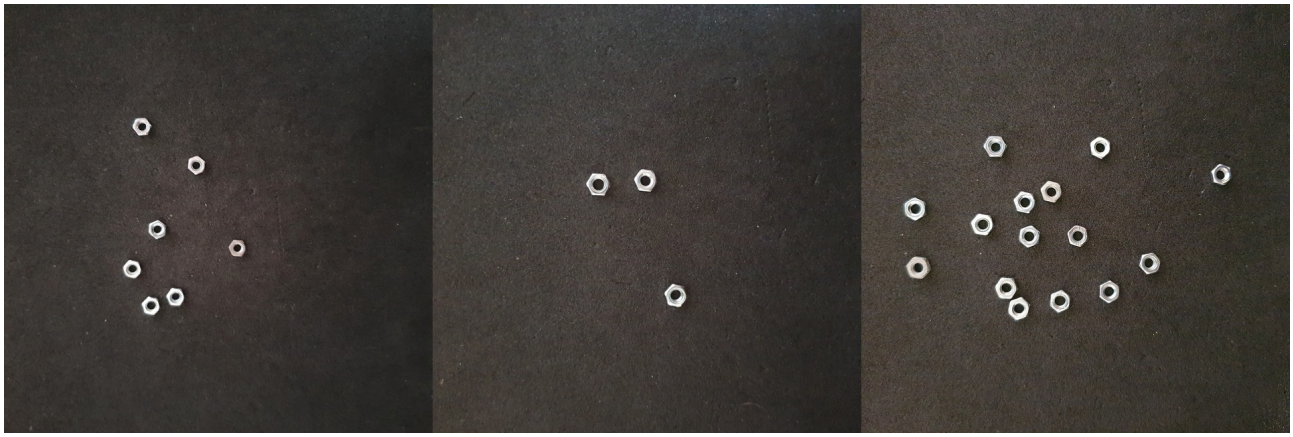


Figure 5.5: Dataset A samples examples.

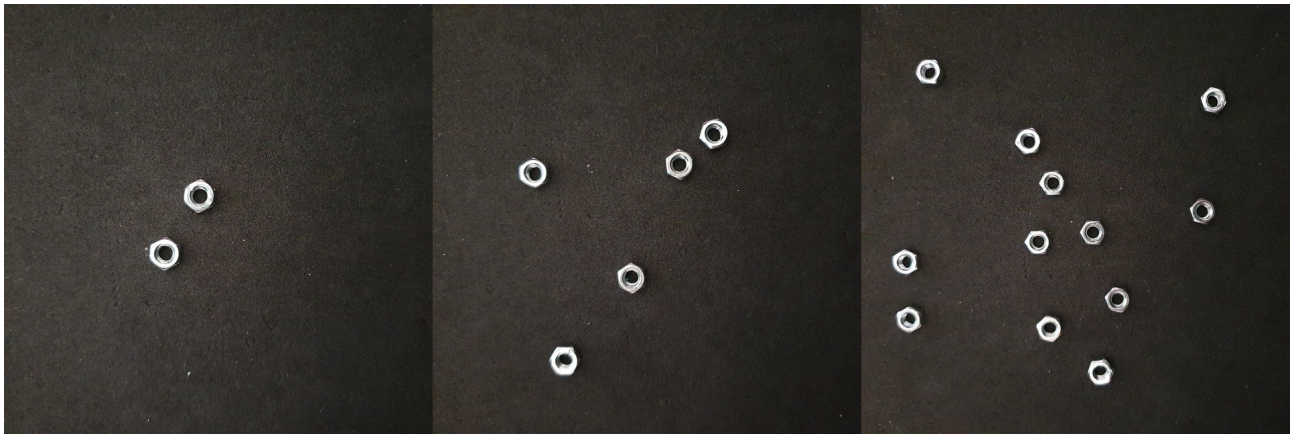


Figure 5.6: Dataset B samples examples.

5.4 Selected Models

As we have seen in chapter 2 SSD is the most suitable method when working with embedded systems. During the years several SSD versions have been released, each of these replaces the original backbone layers with other architectures.Among all the possible solutions we decide to compare and analyze these two versions:

- **SSD MOBILENET V**2: is a lightweight implementation that is designed for efficient computation on mobile devices and embedded systems.
- **SSD RESNET**50: is a more accurate version that improves feature extraction ability by exploiting residual module in the backbone layers.

The choice falls in these two networks to study the results of models that prioritize different metrics, as reported in Tab5.2 ResNet50 favors accuracy at the expense of computation complexity, while MobileNetV2[28] ensures a low inference time, but less robust detection.

Model	Parameters	FLOPs	Top 1-err
MobileNetV2	3.4M	300M	28
ResNet50	25M	3.8B	20

Table 5.2: Comparison between MobileNetV2 and ResNet50 backbones

ResNet50

Intuitively, by increasing the number of layers the resulting neural networks should be more and more accurate, however this assumption is true up to a certain limit. It is well-known in fact that deeper neural networks are affected by the vanishing gradient problem which causes accuracy saturation and degradation of the performance. To mitigate the problem ResNet architecture proposes a new building block structure that makes use of skip connections. Skip, or shortcut, connections are paths that directly link the input of a layer to the output of one 2 or 3 hops away, the stack of layers connected by a skip connection is called residual block. The advantage of residual blocks is that, driven to zero weights and bias, they can easily learn the identity function, in this way whenever a block are not retrieve any meaningful information is skipped preserving the overall performance of the network. Skip connections therefore facilitate the task of each layer making it incremental respect to the previous ones, the result is a more stable and efficient training procedure.

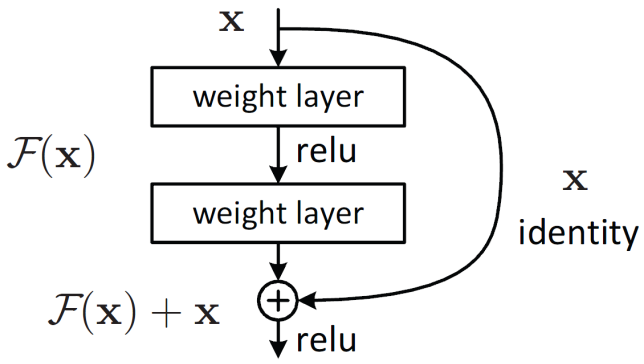


Figure 5.7: Residual Block.

Another way to interpret residual block is that because of skip connections backpropagation algorithm is capable to propagate larger gradients to initial layers given the possibility to train deeper networks. As results ResNet architecture can stack multiple residual blocks without running into the vanishing gradient problem ensuring robust detection on a wide range of tasks.

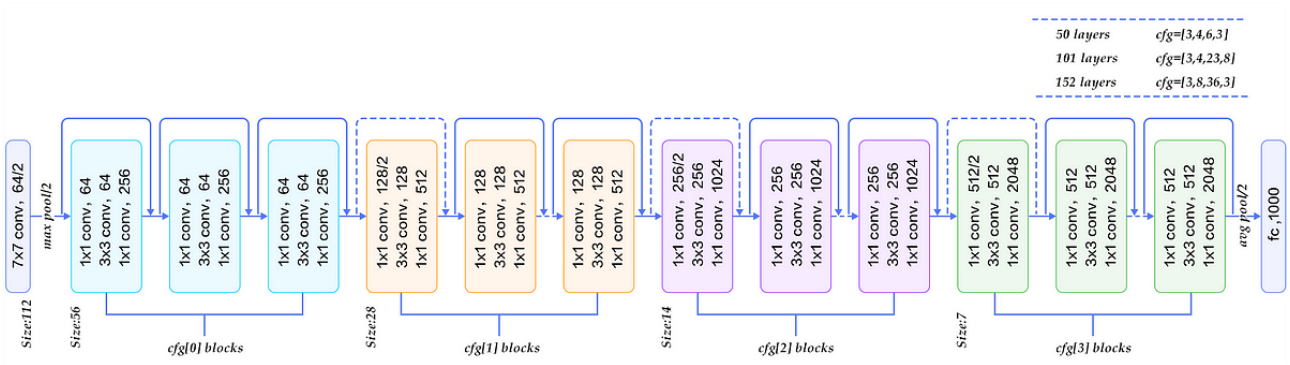


Figure 5.8: ResNet architecture.

MobileNet V2

MobileNetV2 is an architecture specifically designed for mobile and resource constrained environments. The novelty respect to its predecessor, MobileNetV1[29], is the introduction of the bottleneck residual block (Fig.5.9). The block is made up by two 1×1 pointwise convolution and one 3×3 depthwise filter. The first pointwise convolution has the purpose to expand the number of the channel of the input tensor, and for this reason it is also known as expansion layer. While, the second 1×1 convolution, the so-called projection

layer, performs the opposite operation bringing back the filtered data to a low-dimension.

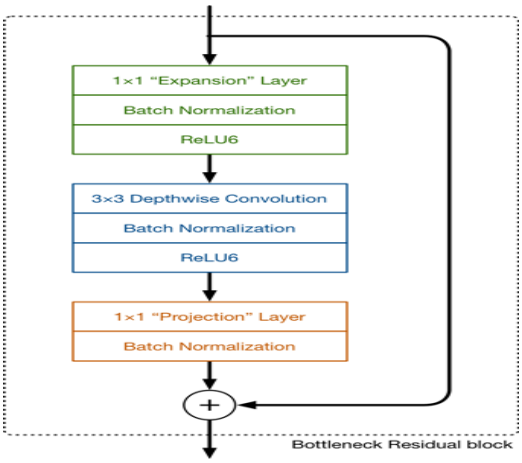


Figure 5.9: Bottleneck residual block.

The expansion factor t is the hyperparameter that defines how much to expand the data processed into the block (the standard value is 6). For example,suppose an input tensor with 24 channels, the expansion layer provides to the depthwise filter a $24 \times 6 = 144$ channels tensor, subsequently the projection layer reduces the filtered data channels, let's say to 32. Following this procedure the tensors that flow between the blocks have low-dimension reducing the overall number of operations, however thanks to the compression/decompression approach the filtering operations is performed on high-dimensional tensors avoiding to loss too much information. The trick that makes this all work, of course, is that the expansions and projections are done using convolutional layers with learnable parameters, and so the model is able to learn how to best (de)compress the data at each stage in the network.

Input size: $224 \times 224 \times 3$				
Layer	Expansion Factor	Stride	Repetition	Output size
conv	-	s2	1	$112 \times 112 \times 32$
bottleneck	1	s1	1	$112 \times 112 \times 16$
bottleneck	6	s2	2	$56 \times 56 \times 24$
bottleneck	6	s2	3	$28 \times 28 \times 32$
bottleneck	6	s2	4	$14 \times 14 \times 64$
bottleneck	6	s1	3	$14 \times 14 \times 96$
bottleneck	6	s2	3	$7 \times 7 \times 160$
bottleneck	6	s1	1	$7 \times 7 \times 320$
conv pw	-	s1	1	$7 \times 7 \times 1280$
average pool 7×7	-	s1	1	$1 \times 1 \times 1280$
conv pw	-	s1	1	$1 \times 1 \times k$

Table 5.3: MobileNetV2 architecture

5.5 Work pipeline

The procedure followed to train, optimize and test the object detection models can be sum up in these principal stages:

1. In the first step the collected training samples are annotated using LabelImg, a lighten software implemented in python that generates annotation files in the .xml extension.
2. The annotation files, one for each image, are converted in a unique TFRecord file. TFRecord is the Tensorflow's own binary storage format that makes easy to import and preprocess data with the functionality provided by the TensorFlow Object Detection (TFOD) API[30].
3. Starting from the models provided in the tensorflow modelzoo, pre-trained on the COCO 2017 dataset, we use TFOD API to perform the training process on our dataset.
4. The trained models are converted in the ONNX format to facilitate interoperability with the Jetson Nano board and the optimization procedure.
5. TensorRT is used to generate serialized engine file optimizing the previously trained ONNX model. We perform the optimization in both single and half precision to analyze possible different behaviors.
6. The optimized models, in single and half precision, and the original one are evaluated on the two test datasets.

6 Results

In this section we comment the performance of the models, optimized and not, previously described. The training procedure is conducted setting a cosine-decay learning rate with an initial value of 2×10^{-3} and 15000 steps, while the batch size, because memory constraint, is of 16 samples. After several attempts we find that this combination of learning rate and iterations offers the best solution ensuring low fluctuation of the loss metrics and at the same time the convergence of the network parameters to the optimal value.

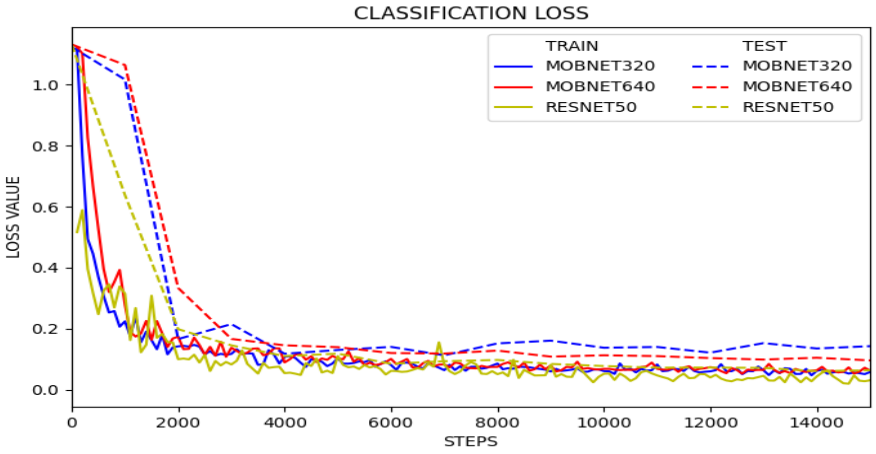


Figure 6.1: Classification loss.

We train the SSD-MOBILENETV2 using two different input image size, (320×320 and 640×640) in

order to measure the impact of the image resolution on the overall performance of the model. However, in the training step, as shown in the figures Fig 6.1 and Fig 6.2 there aren't noticeable differences between the two versions in terms of classification and localization errors. Instead, as expected, SSD RESNET50(640×640 input image) achieves slightly better results in both metrics.

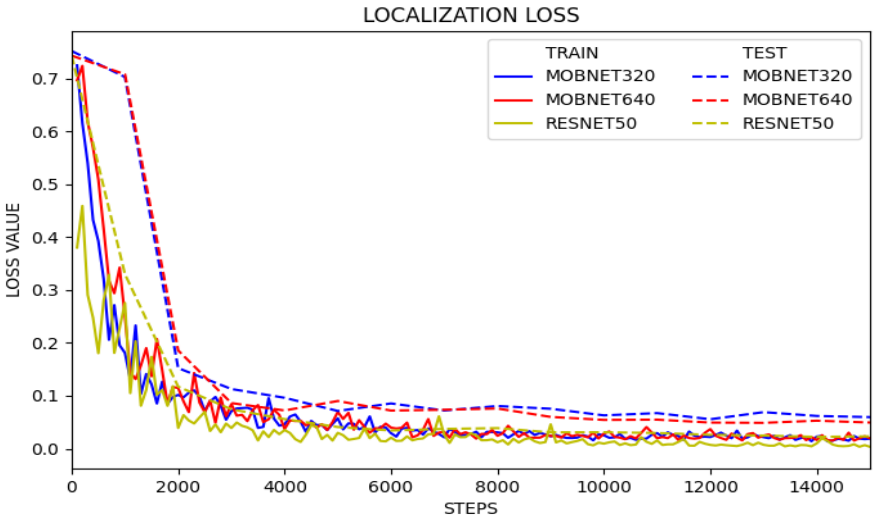


Figure 6.2: Localization loss.

Regarding the detection accuracy, the performance of the models are evaluated with recall and precision metrics:

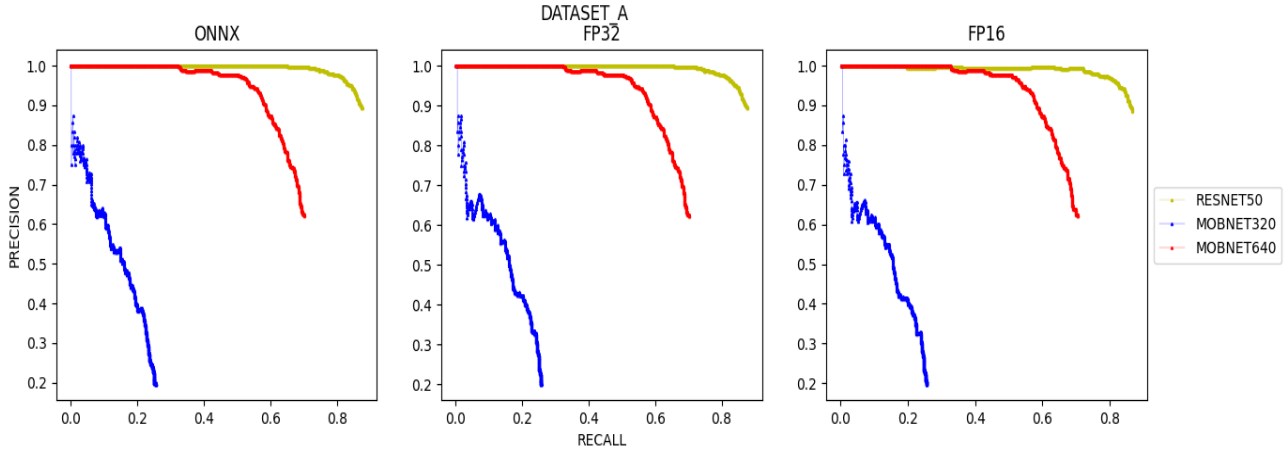
		PREDICTED LABEL	
		POS	NEG
ACTUAL LABEL	POS	**TP**	**FN**
	NEG	**FP**	**TN**

Figure 6.3: Confusion matrix.

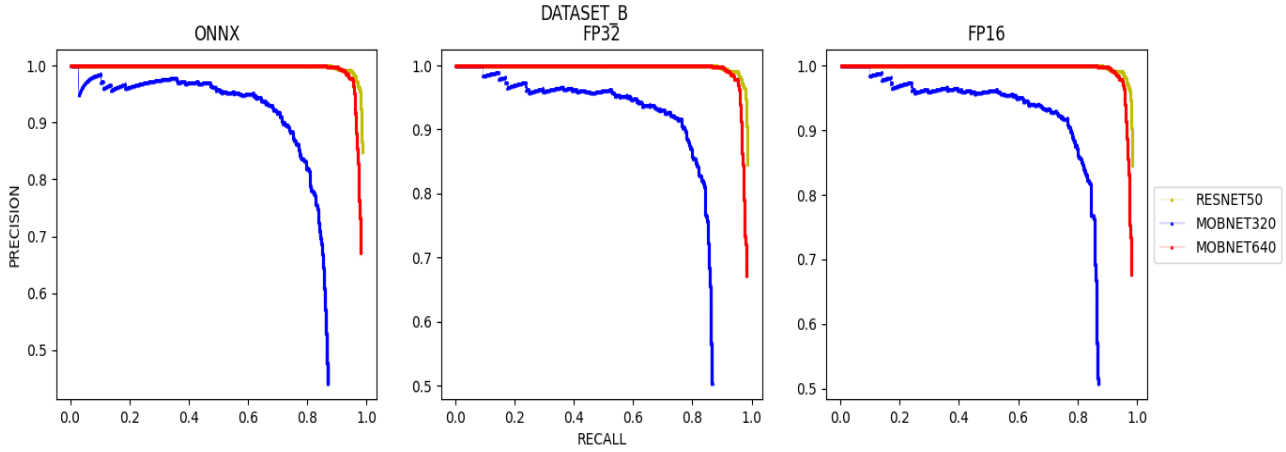
$$Precision = \frac{TP}{TP + FP} \quad Recall = \frac{TP}{TP + FN} \tag{6.1}$$

Recall is the fraction of relevant elements detected, while *Precision* measures the rate of correct detections among all the retrieved ones. The two metrics are not particularly useful when used in isolation, in fact a system with high recall but low precision returns many results, but most of its predicted labels are incorrect,

instead a system with high precision but low recall returns very few predictions, but correctly identifies most of them. Therefore, to properly estimate the better trade-off, we plot precision against recall curve as a function of the model's confidence score threshold. More specifically recall and precision are computed sorting the detections in descending order of confidence score. The resulting curves are shown in Fig.6.4.



(a) Dataset A.



(b) Dataset B.

Figure 6.4: Precision-Recall curve for the different optimizations

For each model we obtain the best balance between recall and precision maximizing the F1 score function, defined as:

$$F1 = 2 \times \frac{Precision \times Recall}{Precision + Recall} \tag{6.2}$$

We can notice that the lower resolution version of the SSD-MOBILENETV2* performs much worse re-

*in the following the two versions are abbreviated as SSD-MOBNET320 and SSD-MOBNET640

DATASET A			
Model	Precision	Recall	Conf. Threshold
SSD-MOBNET320	0.43	0.20	0.28
SSD-MOBNET640	0.93	0.57	0.25
SSD-RESNET50	0.95	0.85	0.14

Table 6.1: Dataset A, FP32 optimized models precision and recall at the correspondent confidence threshold

DATASET B			
Model	Precision	Recall	Conf. Threshold
SSD-MOBNET320	0.92	0.76	0.31
SSD-MOBNET640	0.98	0.95	0.30
SSD-RESNET50	0.98	0.96	0.32

Table 6.2: Dataset B, FP32 optimized models precision and recall at the correspondent confidence threshold

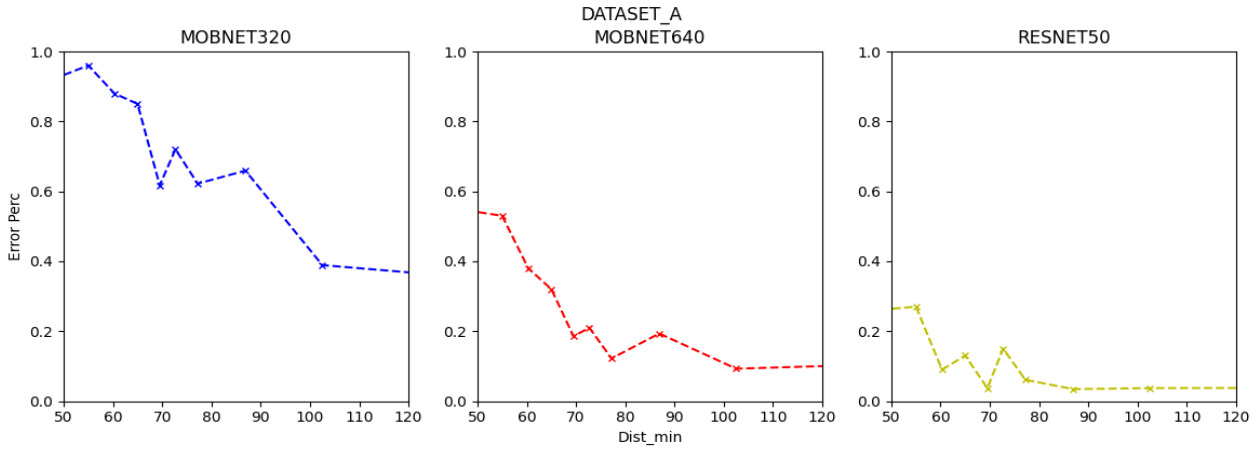
spect to the other two models, especially in the case of the smaller bolts(DATASET A). On the DATASET B SSD-MOBNET640 and SSD-RESNET50 are able to detect over the 95% of the test samples with a remarkable precision, however the performance, as for SSD-MOBNET320, drops when the models are tested on the DATASET A. This problem is not unexpected, in fact it is well known that SSD algorithm decreases its accuracy when has to detect objects that are too small respect the resolution of the image[31]. Finally, the precision-recall curves in Fig. 6.4 confirm that the optimization procedure, in single and half precision, doesn't damage the overall detection performance of the models.

Object-detection methods encounter difficulties in detecting heavily compacted scenes, an issue that must be considered in an object-counting system where in some frames it is possible to have a lot of items to detect in a limited portion of the image. To study the behavior of the selected models the fraction of missed detections is computed as the density of objects in an image varies. We express the density of items in each test image as the average of the minimum distances, measured in number of pixels, between the centers of the bounding boxes that belong to the annotation set B.

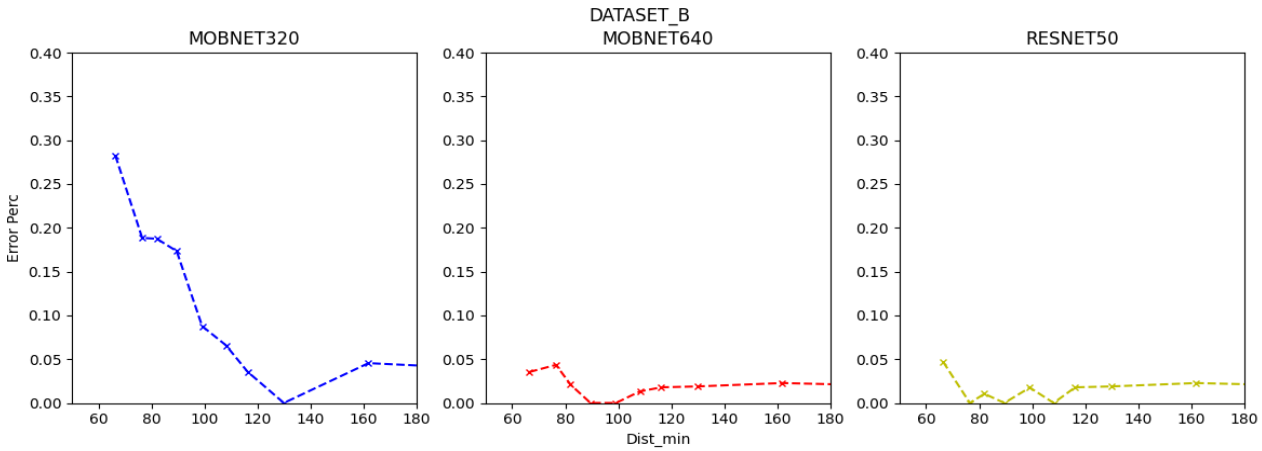
$$D = \frac{\sum_{x \in B} \min_{y \in B} d(x_c, y_c)}{|B|} \tag{6.3}$$

Clearly SSD MOBNET640 and SSD RESNET50 have a low error rate at any distance on the DATASET B, as depicted in Fig.6.5b, since as we have seen previously they detect the bigger bolts efficiently, while we

can appreciate how the worse performance of SSD MOBNET320 is principally due to the poor results in detecting very close objects. On the DATASET A, dealing with the smaller bolts, only SSD RESNET50 ensures reliable detections (error rate $< 5\%$) when the bolts are sufficiently spaced, $D_{min} > 80$, in all other cases the number of errors committed is too high to be accepted.

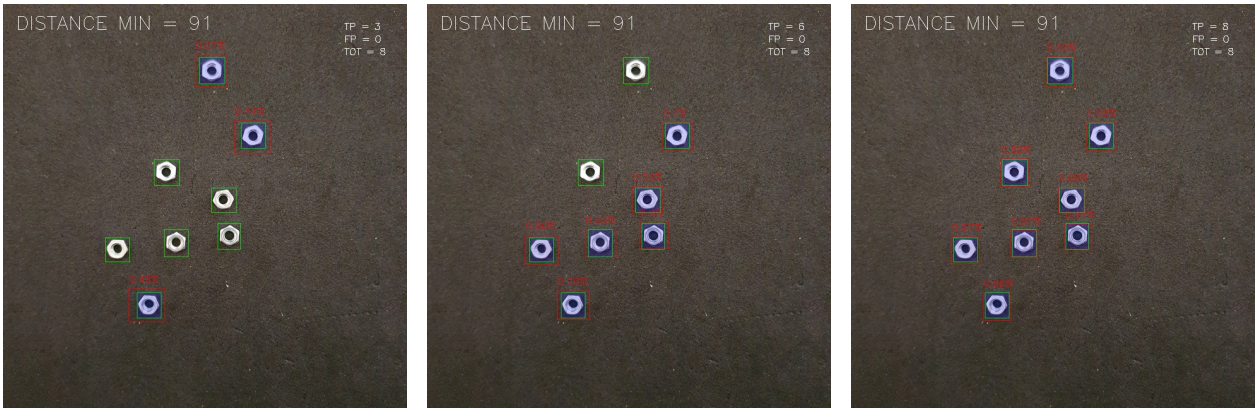


(a) Dataset A.



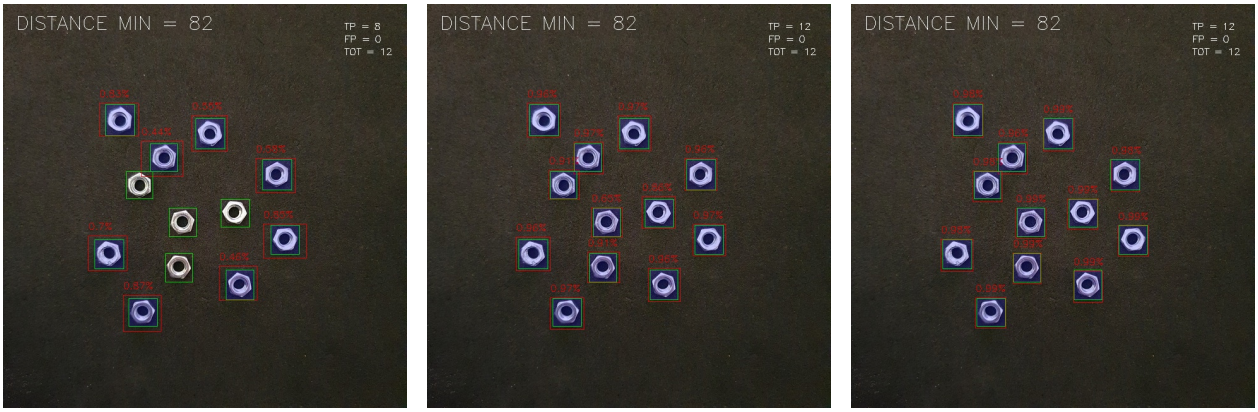
(b) Dataset B.

Figure 6.5: Error-Distance curves for the different models



(a) MOBNET320 **(b)** MOBNET640 **(c)** RESNET50

Figure 6.6: Detection example on Dataset A



(a) MOBNET320 **(b)** MOBNET640 **(c)** RESNET50

Figure 6.7: Detection example on Dataset B

The next step is the evaluation of the models inference time. On the Jetson Nano board, SSD MOBNET320, SSD MOBNET640 and SSD RESNET50 achieve 30FPS, 10FPS and 2.5FPS respectively. As shown in Tab6.3, these results are obtained by exploiting the TensorRT optimizer which allows to reduce the inference time from 2 to 3 times. In addition, we also observe that half-precision significantly improves the performance compare to single-precision only in the case of SSD RESNET50, on the other lighter models the detection time obtained by the two versions is basically the same.

INFERENCE TIME [ms]			
Model	ONNX	FP32	FP16
SSD-MOBNET320	100	36(−64%)	33(−67%)
SSD-MOBNET640	236	110(−53%)	107(−55%)
SSD-RESNET50	890	648(−27%)	385(−56%)

Table 6.3: Inference time measuraments for the different models and optimizations

The usage of TensorRT also reduces the amount of necessary GPU memory to run the different models, like depicted in Fig.6.8. The memory reduction due to half and single precision optimization on the three considered models follows the same trend described for the inference time.

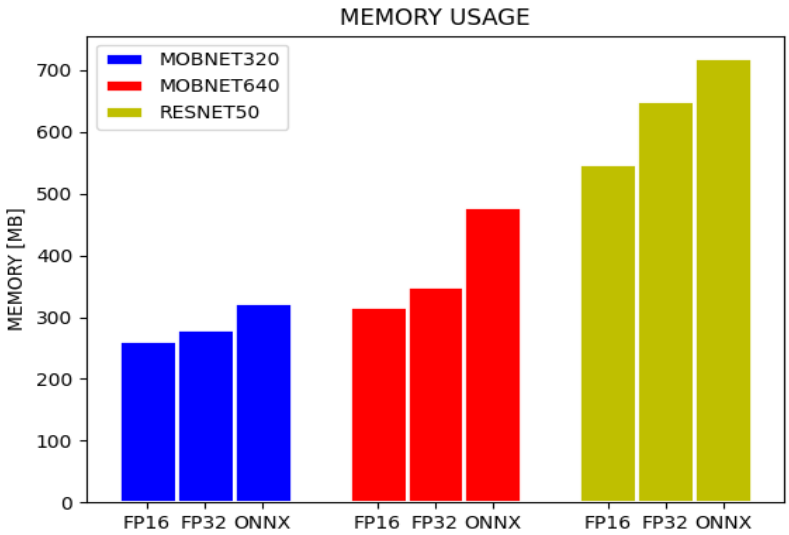


Figure 6.8: GPU memory usage.

7 Conclusion

This thesis work analyzes a deep learning based object counter implemented on an embedded system. Although object counting is not a new application of deep learning and there are many examples of this task in the literature, it is still a difficult problem to solve on embedded devices because of the specific hardware resource constraints. Nowadays effective application of deep learning techniques within embedded system is generating considerable interest, in fact moving the computation from cloud servers to edge devices reduces power consumption and network bandwidth, ensures low-latency and preserves data privacy. The work focuses on the implementation of a deep learning system running on the Jetson Nano board that is able to count bolts in automated packaging lines. The performance of the counter is far from being perfect, especially in the case of the smaller bolts, but it is satisfactory considering the relatively small dataset used for training and the light models used for the detection. In particular, among the different models tested in this work, we observe that SSD MOBNET640 ensures the best trade-off between accuracy(98%) and inference time(100ms) in the case of bolts in diameter greater than 1cm. When the dimension of the object to detect decrease it is necessary to use more complex models to maintain an acceptable accuracy, like SSD RESNET50, and in this case TensorRT optimizer provides remarkable results on the reduction of the inference time and memory usage. However, on Jetson Nano board the frame rate obtained, about 2.5FPS, is still too small to work on real-time condition. A future direction is to exploit more powerful Jetson modules in order to obtain low inference time also with complex model architecture. Another possible improvement is the implementation of a defect detection system along with the object counter, in this way during the counting phase it is also performed the quality check of the pieces. Despite the specific application on bolts, it is important to highlight the generality of the implemented pipeline, that can be adapted to different contexts.

Reference

1. J. Bernal, K. Kushibar, D. S. Asfaw, S. Valverde, A. Oliver, R. Martí, and X. Lladó, "Deep convolutional neural networks for brain image analysis on magnetic resonance imaging: a review," *Artificial Intelligence in Medicine*, vol. 95, pp. 64–81, 2019.

2. M. Kashiha, C. Bahr, S. Ott, C. P. Moons, T. A. Niewold, F. Ödberg, and D. Berckmans, "Automatic identification of marked pigs in a pen using image pattern recognition," *Computers and Electronics in Agriculture*, vol. 93, pp. 111–120, 2013.

3. M. Heimberger, J. Horgan, C. Hughes, J. McDonald, and S. Yogamani, "Computer vision in automated parking systems: Design, implementation and challenges," *Image and Vision Computing*, vol. 68, pp. 88–101, 2017. Automotive Vision: Challenges, Trends, Technologies and Systems for Vision-Based Intelligent Vehicles.

4. I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

5. R. Yamashita, M. Nishio, and R. K. G. Do, "Convolutional neural networks: an overview and application in radiology," *Insights into Imaging*, vol. 9, no. 10, pp. 611–629, 2018.

6. D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, pp. 533–536, 1986.

7. T. G. Karimpanal and R. Bouffanais, "Self-organizing maps for storage and transfer of knowledge in reinforcement learning," *Adaptive Behavior*, pp. 111–126, 2019.

8. R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," 2014.

9. J. R. R. Uijlings and K. E. A. van de Sande, "Selective search for object recognition," *International Journal of Computer Vision*, 2013.

10. R. Girshick, "Fast r-cnn," *2015 IEEE International Conference on Computer Vision (ICCV)*, 2015.

11. S. Ren, K. He, R. Girshick, and J. Sun, "Faster r-cnn: Towards real-time object detection with region proposal networks," 2016.

12. W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg, "SSD: Single shot MultiBox detector," in *Computer Vision – ECCV 2016*, pp. 21–37, Springer International Publishing, 2016.

13. J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," 2016.

14. TensorRT NVIDIA, 2023. https://developer.nvidia.com/tensorrt.

15. ONNX Open Neural Network Exchange. https://onnx.ai/, 2023.

16. Z. Song and K. Shui, "Research on the acceleration effect of tensorrt in deep learning," *Scientific Journal of Intelligent Systems Research Volume*, vol. 1, no. 01, 2019.

17. K. Pavel and S. David, "Algorithms for efficient computation of convolution," in *Design and Architectures for Digital Signal Processing* (G. Ruiz and J. A. Michell, eds.), ch. 8, Rijeka: IntechOpen, 2013.

18. Wikipedia contributors, "Cuda — Wikipedia, the free encyclopedia," 2023.

19. CUDA Stream: Serial Model vs Concurrent Model. https://leimao.github.io/blog/CUDA-Stream/, 2022.

20. M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015. Software available from tensorflow.org.

21. J. Dean, G. S. Corrado, R. Monga, K. Chen, M. Devin, Q. V. Le, M. Z. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, and A. Y. Ng, "Large scale distributed deep networks," in *NIPS*, 2012.

22. M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: A system for large-scale machine learning," 2016.

23. A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, p. 1097–1105, Curran Associates Inc., 2012.

24. K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

25. C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," 2015.

26. H. A. Imran, U. Mujahid, S. Wazir, U. Latif, and K. Mehmood, "Embedded development boards for edge-ai: A comprehensive report," 2020.

27. J. G. A. Barbedo, "Impact of dataset size and variety on the effectiveness of deep learning and transfer learning for plant disease classification," *Computers and Electronics in Agriculture*, vol. 153, pp. 46–53, 2018.

28. M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "Mobilenetv2: Inverted residuals and linear bottlenecks," 2019.

29. A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, "Mobilenets: Efficient convolutional neural networks for mobile vision applications," 2017.

30. TensorFlow Object Detection 2 API. https://github.com/tensorflow/models/tree/master/research/object_detection, 2023.

31. J.-S. Lim, M. Astrid, H.-J. Yoon, and S.-I. Lee, "Small object detection using context and attention," 2019.