POLITECNICO DI TORINO

Master's Degree in Computer Science Engineering

Master's Degree Thesis

DEEP LEARNING AT THE EDGE
Optimizations of object detection models for embedded devices

Supervisors

Prof. Andrea CALIMERA

Dott. Antonio DEFINA

Candidate

Ivan MURABITO

April 2022

Abstract

Deep learning algorithms, specifically those related to computer vision such as object detection, are increasingly used in various disciplines. However, these models have a very high computational cost and execute them (in inference phase) on devices with limited resources, in order to exploit them for real-time decision making, is a growing challenge. The simplest approach to bring intelligence to the edge is to use the device only for data acquisition and leave the computation to the cloud. However this approach is limited for near-real time applications because of the high latency of the network round-trip, and also can lead to problems of consumption and environmental impact of datacenters. The goal is to analyze, apply and evaluate the main optimization and compression techniques of object detection models (based on convolutional neural networks) to make them suitable for deployment on an embedded arm-based device. Specifically, two main techniques have been analyzed: Quantization, which purpose is to improve performance in terms of inference speed and reduce the model size, and Knowledge Distillation which aims to improve performance in terms of accuracy of a small network supervised by a larger one. Several experiments were carried out with different methods and combinations of networks, some evaluated directly on the edge device. The results suggest that with these techniques it is possible to optimize object detectors models, improving the inference time, accuracy and reduce the model size.

Sometimes it is the
people who no one
can imagine anything of,
who do the things that
no one can imagine.

Alan Turing

Ringraziamenti

Questo lavoro di tesi e' stato possibile grazie al supporto della LINKS Foundation e del mio relatore.

Arrivato alla conclusione di questo intenso percorso, desidero ringraziare chi lo ha reso possibile, supportandomi e sopportandomi durante questi anni.
Ai miei amici, quelli di giù e quelli di su, per aver reso le mie giornate più leggere, alla mia compagna, per la sua tenacia e per essere sempre stata il mio saldo riferimento, ma soprattutto alla mia famiglia, per avermi sempre e incodizionatamente supportato in tutte le mie scelte, permettendomi di coltivare le mie passioni.
Questo traguardo è per voi.

Table of Contents

List of Tables

List of Figures

Acronyms

AI
artificial intelligence

IOT
Internet of things

GPU
graphics processing unit

ANN
artificial neural network

DNN
deep neural network

CNN
convolutional neural network

SSD
single shot detector

FPS
frame per second

mAP
mean average precision

TFLITE
tensorflow lite

UINT8

unsigned int 8 bit

KD

knowledge distillation

PTQ

post training quantization

QAT

quantization aware training

OBKD

output based knowledge distillation

FIKD

feature imitation knowledge distillation

Chapter 1

Introduction

Machine learning is a branch of artificial intelligence that includes several algorithms and methodologies that aim to solve a problem through past experience (data) without being manually programmed to solve that particular task. We could say that a software can learn when its performance in solving a Task T improves with experience E while it is measured with metric P [1].
Compared to a traditional algorithm approach we delegate to a software the duty of find patterns and schemas from the data, this approach is convenient to solve problems that are too complex to be modelled manually, e.g: classification, regression, clustering on data but also object detection, autonomous driving, natural language processing, pathfinding.
Deep learning is that sub-set of machine learning techniques and algorithms inspired by the way the human brain works, so we can introduce the concept of artificial neural network as the basis of deep learning.
Unlike machine learning techniques where features of interest are explicitly defined in the data (problem specific features) the purpose of artificial neural networks is to extract these features of interest from data that have not been explicitly defined. Artificial neural networks are widely used in contexts such as computer vision, natural language processing, speech recognition, bioinformatics.

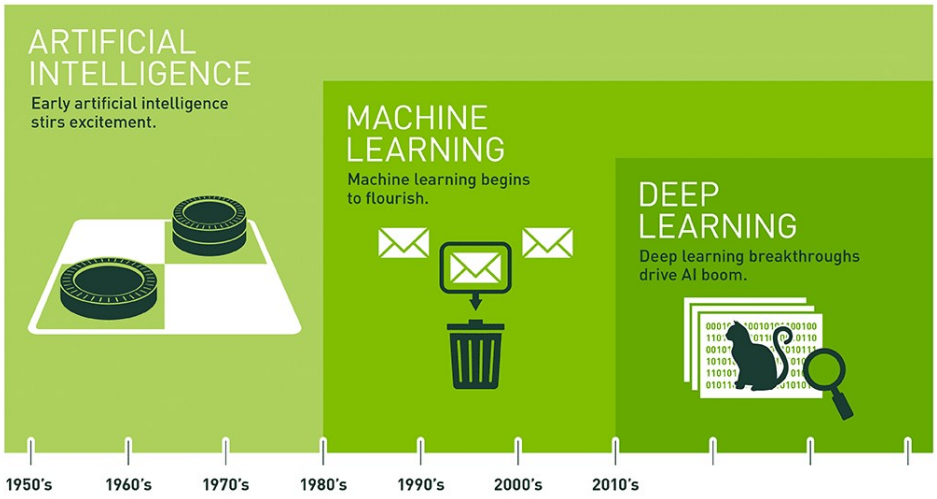


Figure 1.1: The Evolution of AI over time, from www.nvidia.com

Artificial Neural Networks are not a new concept, we can find the first neural network abstractions since the 60's [2] but in the last decade, since 2012 thanks to the disruptive advent of the use of GPUs for training Neural Networks and a better data availability [3] there 's been a significant increase in scientific research and its application fields, this has led to the development of models capable of performing perfectly tasks of computer vision as: image-classification, object-detection, segmentation and others.

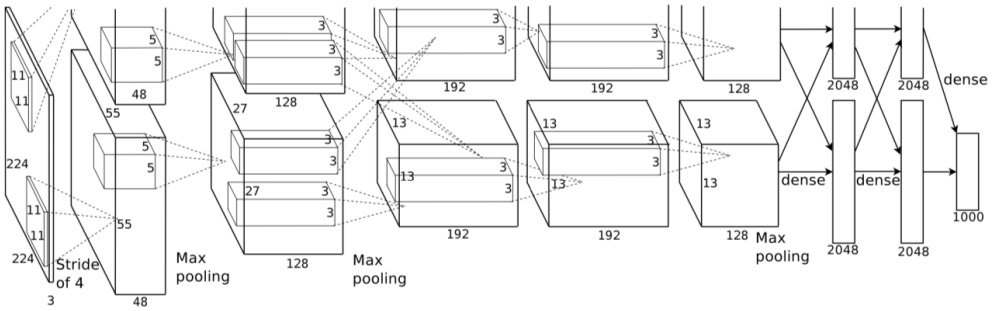


Figure 1.2: Alexnet architecture [3]

In parallel with the growth of computing power, these models have become increasingly accurate and resource intensive and at the same time there has been a strong increase of iot devices (internet of things), for iot device we can mean hardware such as sensors, actuators, and more, programmed for a specific purpose and able to transmit data over the network (not necessarily internet), this type of

device normally have low computational performance, in fact the first approach to bring artificial intelligence on these devices is to use them only as a final node and delegate to the cloud the processing of data.
Recently, the performance of these devices is growing, and devices capable of natively supporting AI applications have been introduced into the market, according to a research: *"Global shipments of edge AI devices to reach 2.5 billion by 2030"* [4].
However, running (in the inference phase) complex deep learning models, specifically those related to computer vision on this type of device is a growing challenge.
The purpose of this thesis in collaboration with Links Foundation is to explore and test methods and best practices of compression of deep learning models, more specifically of object detection models in order to be deployed efficiently on an arm device, optimizing performance and used resources. In the following chapters we will find a background on deep learning and artificial neural networks, object detection networks and the main compression methods, the tools I used, the framework I created, a series of tests and benchmarks of the tried methodologies and finally conclusions.

1.1 Edge AI Application Segments

Today, artificial intelligence combined with IOT and embedded devices is used in virtually every market sector, from mission critical to less important applications. the common goal is the use of artificial intelligence to make people's lives easier with automation of decision-making processes.
Bringing intelligence closer to the edge can benefit many areas of application, some examples:

- Self-Driving cars (*Detection, localization, control, ...*)
- Smart Buildings (*Non intrusive load monitor, smart BMS, ...*)
- Smart Industry (*Predictive maintenance, Monitoring , ...*)
- Smart Home (*Natural language processing, Domotics ...*)
- Smart Farming (*Decision support system, ...*)
- Drones (*Autonomous flight, Obstacle detection, ...*)
- Security (*Public surveillance, detection, recognition, ...*)
- Healtcare (*Disease diagnosis by data analysis, ...*)
- Wearable (*Body activity analysis, ...*)

1.2 Cloud AI Approach

To solve artificial intelligence tasks through devices with relatively few hardware resources (iot, mobile, ..) one of the approaches commonly used is to use the device only for data acquisition (sensors, photos, audio, ..) and delegate to the cloud the computation of data, and then return to the device the result of the task.
This solution has pros and cons, depending on the application context;
We have to consider the costs and efforts to manage an application that scales as the number of users grows, perhaps using SAAS (Software-As-A-Service), because managing the load of a few devices is computationally efficient, but as the number of users grows, the load can become an unsustainable engineering challenge for datacenters, not only in terms of performance and cost, but also in terms of environmental pollution.

1.2.1 Pros and Cons

- **Low power consumption**
  Embedded devices basically only need to send information through the network, this results in devices with low power consumption that are potentially cheaper and more efficient.

- **Easy to update and maintain**
  Developing a backend AI application is relatively easier for developers because they don't have to deal with hardware/software constraints. It is also very easy to upgrade and maintain, as there is no need to deploy the application to the embedded device. However, it does require some engineering effort to manage scalability.

- **Latency: no realtime application**
  Although network architectures have achieved great speed in terms of latency and bandwidth, the round trip time for a single device communication remains too slow and unstable to support real-time applications, there are contexts where the responsiveness requirement is fundamental, just think of autonomous driving, where it is unthinkable to have to wait for a response from a server to make a decision.

- **Privacy and safety**
  In the domain of deep learning, privacy is a hot topic. Taking an object detector algorithm as an example, a photo will have to travel over the network to data centers, which will probably save it to process a result and send it back to the sender. There is a need to assure the user that appropriate use will be made of those photos and data. Moreover, sending data over the network can

expose the application to various attacks such as man in the middle, reply attack and others.

- **Connection overhead**
  The use of a network connection brings with it some overhead, for example the amount of data to be exchanged can be too large and saturate the available bandwidth, in addition, comparable response times are not guaranteed. Some iot devices don't use the WiFi/Ethenet protocols to connect to the internet, but they can use special low power and long distance network protocols, e.g the LoRa protocol, this kind of protocols support very small payloads, in the order of tens of bytes.

- **Datacenter energy consumption** Since the workload of AI applications is proportionally greater than the normal workload of data centers, there is a need for a lot of computational power and this will lead to use more energy and pay more for cloud services, also in many contexts are required machines with GPUs that lead the expense to rise more and more, especially if you want to scale with the number of users. This surplus of energy used in the long run can turn into problems of environmental pollution.

1.3 From the cloud to the edge

Being able to process through AI the data directly on the edge can unlock the way to applications that can make a decision in real-time and solve some of the problems with the cloud approach. We will see that implementing this kind of approach from a developer's point of view is much more complex than implementing a cloud based solution, due to hardware constraints, and that's why optimizations are necessary. Moreover there is the need of more performing and specific hardware than off the shelf hardware used with cloud approach, since the workload moves from the cloud to the device itself and you need more powerful (and more expensive) devices.

1.3.1 Pros and cons

- **Low latency: Real Time Application**
  Executing the computation locally allows the device to make a choice or solve a type of problem without waiting for the response of the server, this unlocks the road to many applications in real-time as autonomous driving, flight control systems, error control in smart industries, cybersecurity etc.. It also guarantees a better user experience.

- **Privacy**
  Since the data is processed locally this drastically reduces the chances of a

data beach, and in some contexts it is important to ensure that the acquired data does not leave the device.

- **Expensive embedded devices and not so easy to deploy**
  Since in a cloud application theoretically there are few constraints regarding the development of the model, when you have to run in inference an AI model on an embedded device, especially if it is a deeplearning model, there are many constraints to respect for the deployment; - performance - compatibility - memory impact - operating system and programming languages - heterogeneous hardware
  Devices able to satisfy these characteristics are a small part of all iot devices, and therefore more expensive.

1.3.2 Mixed Approach

There is also a mixed approach that is increasingly used, i.e. delegating to the cloud the part of the computation that is heavier, while executing on the device the part of the computation that needs low latency.

An example can be the home voice assistants like *google home or amazon alexa*, where the logic to recognize the keyword of initialization is directly on the hardware of the device to allow a fast activation, while the subsequent analysis of the phrases is performed on the cloud of the producers. Another growing hybrid approach is to use intermediate nodes between the cloud and the edge for certain types of computation, we call them **edge (or fog) nodes**.

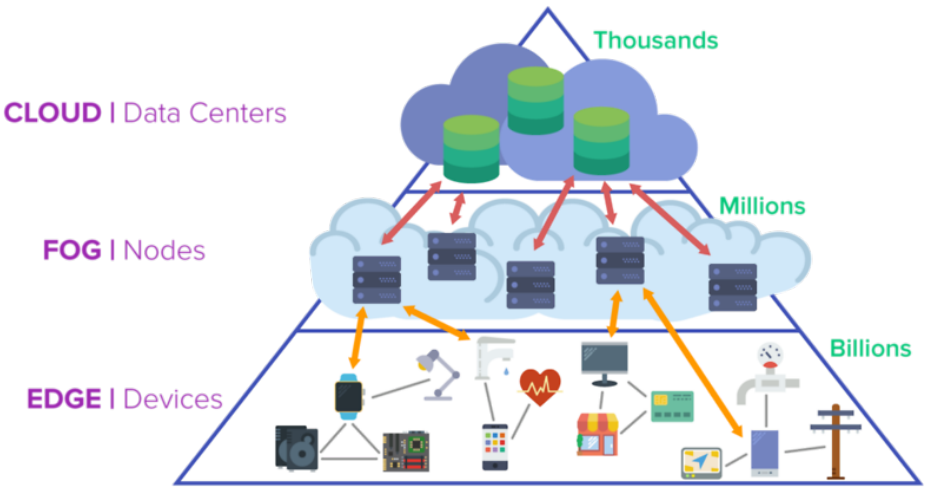


Figure 1.3: Edge and Fog computing, we can see the distribution of these devices.

Chapter 2

Background

In this chapter will be briefly explained the fundamentals of deep learning in order to understand this thesis work, will be initially made an introduction on neural networks and their training process. It will be analyzed some models of object detection important for this kind of study and finally there will be an analysis of the main methodologies and techniques of optimization and compression of neural networks.

2.1 Introduction to Deep Learning

2.1.1 Artificial Neuron and Activation Function

The artificial neuron is the smallest building block of a neural network, the first notion of an artificial neuron is from 1943 by McCulloch and Pitts taking inspiration from biology and how real neurons work. [5]

Abstractly we can define an artificial neuron as a function that takes multiple inputs and produces an output, each input Xi is associated with a weight Wi, as you can see in 2.1

Initially is performed a dot product between neuron's inputs and relative weights d, eventually summed by a bias factor b and then the result will go through an activation function. We can then represent the transfer function of an artificial neuron in this way:

$$f(x) = \phi((\sum_{i=0}^{n} w_i x_i) + b) \tag{2.1}$$

In equation 2.1 ϕ is the **Activation function** , which is used to bring an unbounded input into a bounded output,initially one of the most used activation functions was the **sigmoid** :

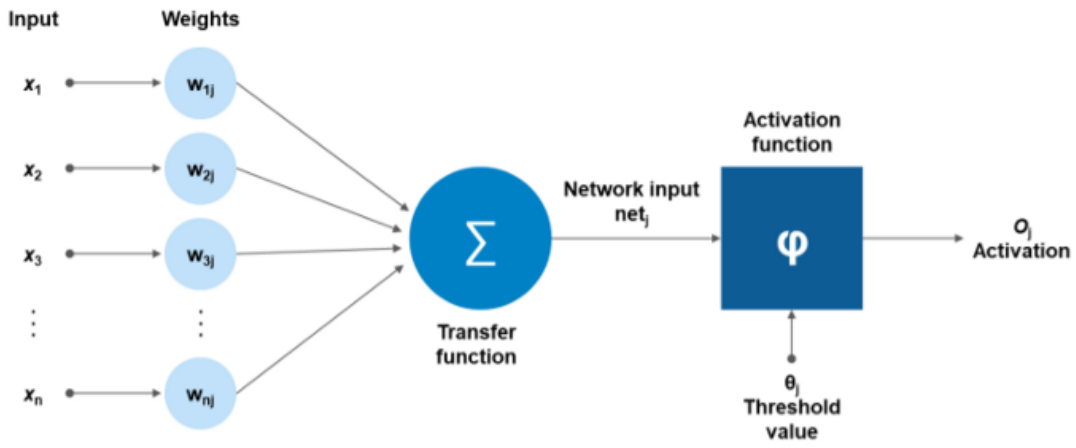


Figure 2.1: Representation of an artificial neuron (Perceptron Model), we can see the vector of inputs with related weights and the activation function, that can be a sigmoid.

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

in equation 2.2 $x = \sum_{i=0}^{n} w_i x_i$

Notice that the domain of sigmoid function is $[-\infty, +\infty]$, while the codomain is [0,1],this guarantees that whatever is the input the output is limited between 0 and 1.

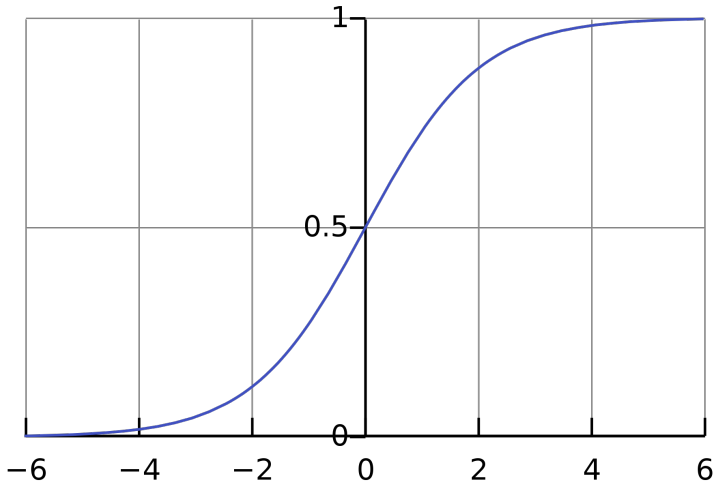


Figure 2.2: Sigmoid function

Sigmoid is also a nonlinear function, which is why it is often used to intruduce a non-linearity in Artificial Neural Networks.
However, its use is not recommended because it runs into the problems of vanishing or exploding gradient [6]. For this reason currently are more used activation functions like ReLU or leaky ReLU.

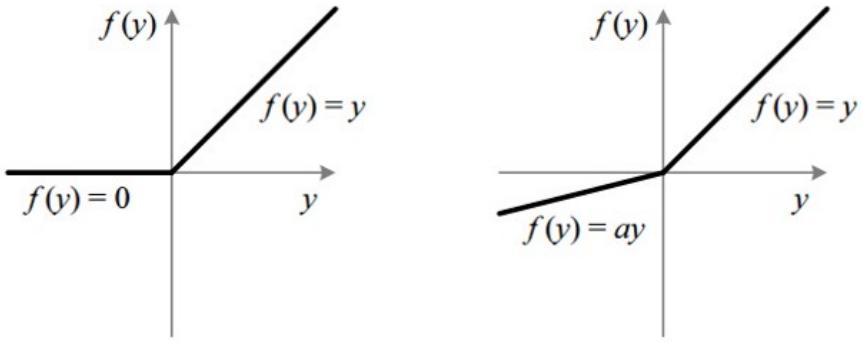


Figure 2.3: ReLU and Leaky ReLU activation functions

2.1.2 Artificial Neural Networks

Artificial Neural networks are widely used for their ability to autonomously learn from data patterns and decisions needed to solve a problem. We can consider an artificial neural network as a black box with inputs, a bit of maths and outputs, in the most basic way Neural networks are formed by more **layers** of interconnected neurons (2.1.1):*Input Layer,Hidden Layer,Output Layer*.
A Input (usually a multidimensional vector,**tensor**) starts from the input layers, passes through intermediate (hidden) layers and finally to the output layers, this kind of networks are called **Feed Forward Neural Networks**.
When there are many (dozens) hidden layers, we can call them **Deep Neural Network (DNN)**.

All the neural connections are associated with a **weight**, weights determines the strength of the signal on that connection,in each node will be processed the scalar product between the input signals and relative weights and next the information will be processed through an a Non-Linear function (**activation function**) 2.1.1 that will determine (also through a bias value) whether to "wake up" the neuron connection and forward the signal to the next layers. Finally the output layers will produce a prediction in terms of probability distribution.

In order to enable the learning phase of the network weights will be modified during the training phase through the **back propagation** in a **stochastic way**,i.e

the information about the error on the prediction will be back-propagated from output layers to input layers in order to optimize the weights for each network nodes. From here we understand that the main goal of an artificial neural network is to optimize the weights in order to minimizing the sum of the errors of output neurons .

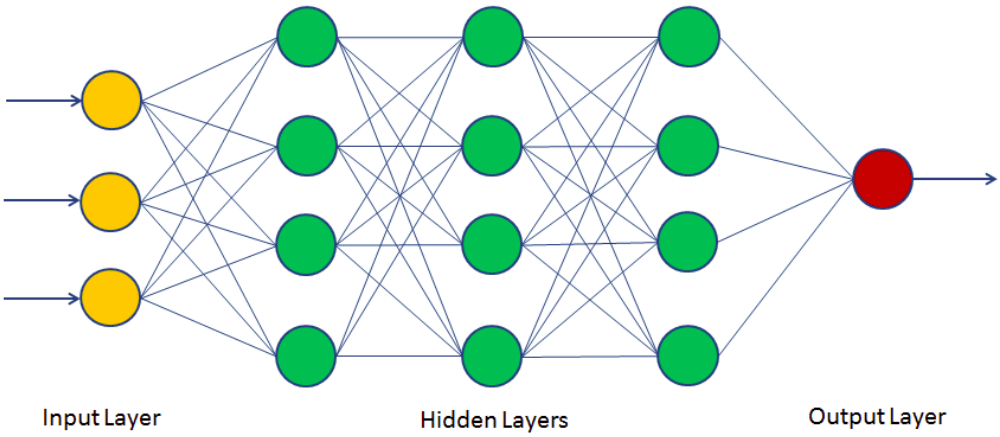


Figure 2.4: Example of Feed Forward and fully connected neural networks

According to the **universal approximation theorem** [7], an artificial neural network can be able to approximate any arbitrarily complex function, however, the theorem only guarantees the existence of a network that satisfies certain conditions, but does not provide a way to find the ideal structure or the weights of the model.

2.1.3 Training

The learning process of an artificial neural network can be achieved through the training process,there are two main macro-approaches: **Supervised** and **Unsupervised** learning. In supervised learning during the train phase, the network will be fed with a lot of examples in form of labeled data(i.e. input data and the desired output, *ground truth*). His goal is minimize the error (*calculated by a loss function*) of the predicted outputs respect the ground truth in order to find the optimal weights. After the training phase the network will be tested with a unlabeled data in order to understand if the network is able to generalize and perform even with never-before-seen data.
Otherwise in unsupervised learning, the network will be fed with unlabeled examples and his goal is to extract feature from the data in order to learn more about the data itself, there is no correct output to mimic, some type of applications of unsupervised learning: Clustering,Data Generation,Associations.

In this thesis work we will only deal with supervised learning, So the recipes for supervised training of an Artificial neural network are: labeled data , a loss (cost) function, a backpropagation algorithm and a optimal way to update network weights.

Loss function

Loss function in artificial neural networks quantifies the difference between the expected output (*ground truth*) and the network output, this difference is called **Loss**. Loss function determine the performance of a artificial neural network, there are no universal loss functions but there are different types, and there are different factors involved in choosing a function for specific problem, we can basically classify loss functions in two main categories: **Regression Loss** and **Classification Loss**. Regression problem deals with predicting a continuous value i.e given data with past market activity predict the next BTC conversion rate. Classification, on the other hand, deals with predict output into a set of finite categories i.e Given a dataset of properties of a plants,the network will predict the plant species.
As an example of regression loss function we can consider the **Mean Square Error** (identified also as Quadratic Loss or L2 Loss):

$$\textbf{MSE} \quad L(y, \hat{y}) = \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{n} \tag{2.3}$$

As the name suggest she measure the average of squared difference between predictions and ground truth,In the equation 2.3 y represent the ground truth,y with hat represent the network predictions,i is the index of the current example data,square instead is used for deal with negative value.

In a classification problem the network output is a probability distribution,the conversion between digit and probability is made in output layers by a *SoftMax function*. **Cross entropy loss** (or Negative Log Likelihood) measure the distance between these probabilities to the ground truth:

$$\textbf{Cross Entropy Loss} \quad L(y, \hat{y}) = -\sum_{i=1}^{n} y_i * log(\hat{y}_i) \tag{2.4}$$

Back Propagation

Backpropagation algorithm is a foundamental building block of artificial neural network,was introduced in the 1960s, but became popular much later, in the late 1980s . Backpropagation, according to [8]:
repeatedly adjusts the weights of the connections in the network so as to minimize a measure of the difference between the actual output vector of the net and the desired output vector.

In order to minimize the outputs of loss function (2.1.3) we need to tune the network's weights and biases,this tune operation is achieved by computing the negative **gradient** of the loss function respect to each weight. Remember that our goal is minimize loss. Mathematically,the gradient is the vector of partial derivatives of a function f in a point x,in other words it defines how the value of f will change with a modification (in positive or negative direction) of x, it is used to found a local minimum of a multi variable function.

In a neural network we need to compute gradient from last layers to initial layers,this is done by the **chain rule**, that allow us to find the derivative of a composite function,consider $h = g(f(x))$, its derivative are :

$$\frac{dh}{dx} = \frac{dh}{du} * \frac{du}{dx} \tag{2.5}$$

During training phase,initially weights and biases are randomly initialized for each network node,consider:

- C: Loss function e.g:CrossEntropy 2.4
- w, b weights and biases
- (x, y) inputs and target outputs
- η learning rate
- epoch: *an entire dataset samples feedforward process*

weights and biases will be modified in the following way:

$$w^1 = w - \eta\frac{dC}{dw} \tag{2.6}$$

$$b^1 = b - \eta\frac{dC}{db} \tag{2.7}$$

.

There may be different kind of timelines in which weights and biases are updated, the simplest consist of update weights for each sample at time,this mean that the forward propagation as well as the back propagation are run sequentially for each training sample at time, another kind is **batch** mode where the gradient is accumulated with respect of all training sample, forward propagate all samples in batch and backpropagate just once with accumulated results of the entire batch,this process is performed for multiple iterations, which we will call **epochs**.
The most used way is the **stochastic** mode,in which fixed size mini batches are defined randomly from the entire dataset for each training epoch.

The stochastic mode (**Stochastic gradient descend** - SGD) is preferable because if you choose randomly the mini batches you risk less to incur in a local optima, because at each iteration weights are updated considering an ever-changing mini batch of dataset,moreover it has better performance and converges quickly

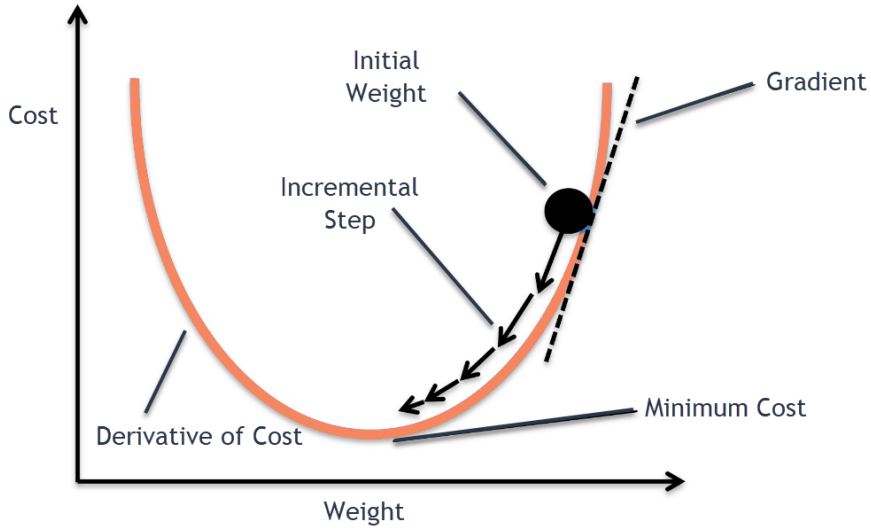


Figure 2.5: Example of Stochastic Gradient descent optimizer process.

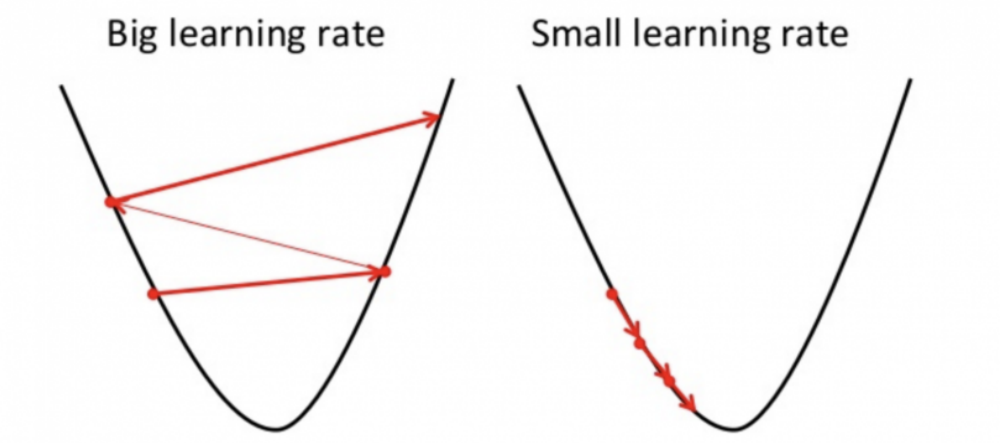


Figure 2.6: How Learning rate can affect the optimization

2.1.4 Dataset handling

Datasets play an important role in artificial neural networks, the way that the data are used during training can affect the behavior and the model performance. It is important to use the dataset to reduce as much as possible the phenomenon of overfitting and try to create a network that can generalize independently from the data on which it has been trained. Regardless of whether it is pure descriptive data or more complex data like images it is commonly used to **split the dataset** into multiple distinct parts:

- **Train split** labeled: it is the split used during train phase, the model will see only samples coming from here and the learning process refer to this data.
- **Validation split** is used for evaluate the model during train phase in order to tune model's hyperparameters.
- **Test split** unlabeled: Are data never seen by the network, are used to evaluate the performance of the final model after the train phase.

The proportions between the different splits depend a lot on the type of problem and the type of model, but also on the availability of data itself.

However, in order to obtain a robust train process there are extensions to the use of these splits, like **K-fold and cross validation**,which consists in further splitting the train dataset in k splits of equal size, for each split and for each iteration, k-1 splits will become the train dataset and the current split the validation dataset. However this technique can be very computationally expensive in case of large datasets.

It is also used apply **normalization** to data samples,e.g. $mean = 0 and std = 1$ in order to have comparable inputs. If we deal with an unbalanced dataset we need to balance it with over-sampling or under-sampling tecnique, to ensure that the network is not trained with to many, or to few, examples of a particular class. Instead, if the dataset has few samples, we can use **data augmentation** techniques.

2.1.5 Evaluation metrics

In order to estimate a model, is that it is a pure analytical model or an artificial neural network, different metrics are necessary,these metrics help us to understand how much is good and performant the model in solving a problem,like classification. for simplicity of exposure we will use a binary classification problem, whose the output of the model can be only: *Positive or Negative*, This prediction can be: *True Positive: TP,False Positive: FP, False Negative: FN* The True and False words refeer to the *ground truth*

- **Accuracy**: it is the simplest metric,is the proportion of the number of correct prediction (*True positive and True negative*) to the total. e.g in email spam detection *all emails correctly predicted as Spam or Non-spam compared to the total emails*

- **Precision**

$$\frac{TP}{TP + FP} \tag{2.8}$$

  Note that the denominator indicates the total of positive predicted,precision tell us how precise/accurate is your model , e.g in email spam detection *how many emails were correctly predicted as spam compared to all emails that are predicted as spam* this metric **doesn't keep in mind of false negative**, this mean that if an actual spam mail is predicted as normal mail the value of precision is not affected.

- **Recall (True Positive Rate)**

$$\frac{TP}{TP + FN} \tag{2.9}$$

  It is the ratio between true positive and the total of actual positive samples, often is called *sensitivity or True Positive rate* e.g *how many emails were correctly predicted as spam compared to all emails that are actually spam*

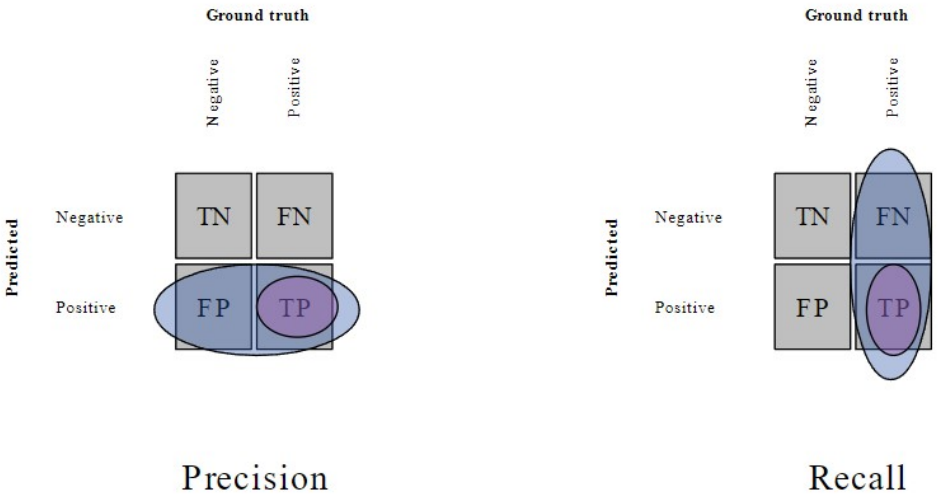


Figure 2.7: Representation of Precision and Recall through confusion matrix

- **F1 Score**

$$\frac{TP}{TP + \frac{FN+FP}{2}} \tag{2.10}$$

F1 score is a harmonic mean between precision and recall,refer to an arithmetical mean, harmonic mean gives more weight to small values,that mean an high F1 Score can be achieved only if both Precision and Recall are high. This metric is very useful when there is an a **imbalanced dataset** and it is more robust than previous ones.

- **ROC Curve and AUC** This is another aggregated metric which gives us a lot of information about the model performance, ROC is a probability curve generated by true positive rate and false positive rate. AUC represent the area under ROC curve and it tells how much the model is capable of distinguish between classes. An AUC near to 1 represent an ideal classifier that is able *To predict correct actual spam email as spam email and actual safe email as safe email*

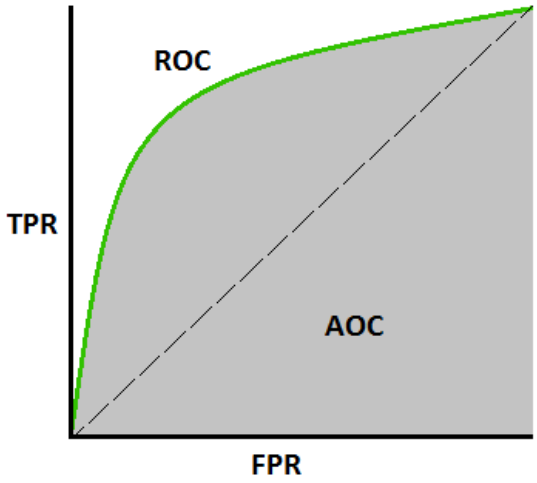


Figure 2.8: ROC is the green curve and AOC the area under it

All metrics have been exposed referring to a binary classification problem, but are easily extensible for a multi-class classification problem for example by calculating the same metric for each class.

2.1.6 Convolutional Neural Networks

Convolutional neural networks (**CNNs**) are a specialized type of neural networks, widely used in the field of computer vision, perfect for tasks related to **images** or videos such as: Image Classification, Object Detector, Image Segmentation, Pose Estimator and others. CNN's were inspired by the animal visual cortex, as neural networks are not a modern concept, but recently thanks to the disruptive advent of gpu for computing but also thanks to the availability of large datasets have become the de facto standard for computer vision applications. For simplicity of exposition from now on we will deal only with image classification, but CNNs are also applied in different areas such as natural language and audio processing.

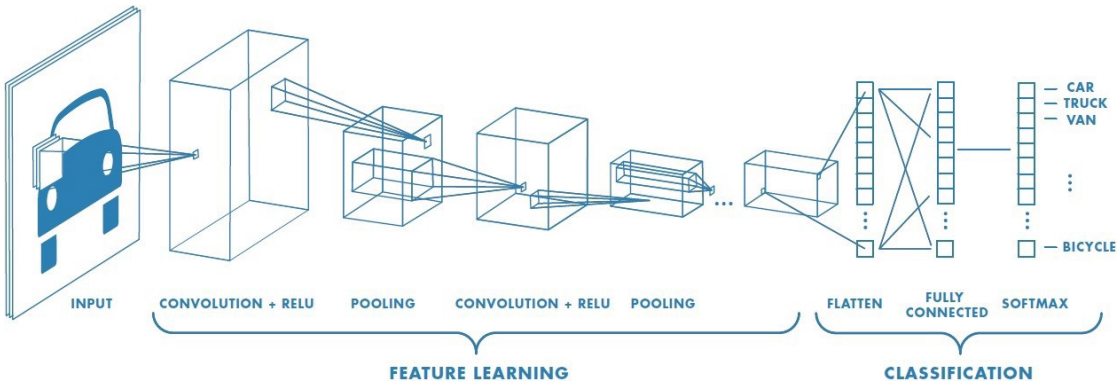


Figure 2.9: Example of a typical CNN architecture

Abstractly their purpose is to extract low and high level features from images, the main difference with *feedforward networks* is that CNN's doesn't have fully connected layer in input and inner layers,but just in the outputs layer, the input layers instead are replaced with **convolutional layers**.
Convolutional layers are used to extract information similar to that of fully connected layers, but using far fewer parameters. Images are just multi-dimensional arrays of pixels, we can consider a grayscale 128x128 image 2-dimensional array as input,the **convolution** process consists in multiplying a "sliding" filter (e.g 4x4) over the input image. The multiplication applied between filter and input is a dot product, that the result of a product is a single value. All the values you get by sliding the filter over the image is the extracted **feature map** which has reduced the input dimensionality. Different convolutions are performed on same input with different filters, the output of convolutional layer is formed by all the extracted feature maps,as we saw earlier with artificial neural network we will use a activation function like **ReLu** [2.3] to make non linear output of convolution. The amount that determines the sliding step of a filter over the input image is called **stride**.

In a CNN there can be more convolutional layers, usually the first ones have the task of extracting low-level features such as edges,cornes,curves, in different location of the image while the inner ones extract higher order features.

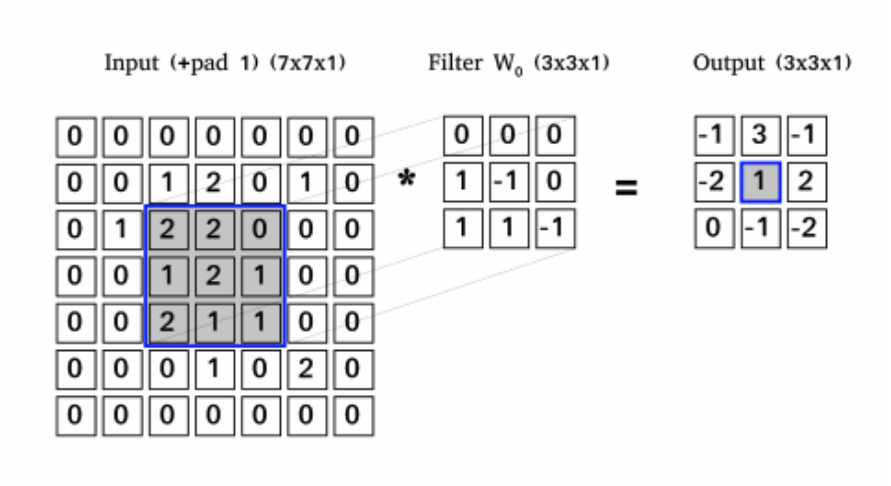


Figure 2.10: Example of the convolution process with a 3x3 filter on a 7x7 input, that output is obtained by sliding the filter along the input with 1x padding aka add 1 layer of zero to input

As we can see in 2.9 the feature extraction is performed by a sequence of same operations in block:

- **Conv -> ReLu -> Pool**

ReLu is used as Activation function in output of convolution block, already seen in 2.3,the reason why ReLu is so popular is that it is easy to calculate, it is simply the maximum value between the function's input and 0,and its derivative can be only 0 or 1, this mean that calculating the gradient during back propagation is computationally inexpensive, as a consequence the use of Relu as activation function prevent the exponential growth of computation required by the neural network.

Pooling layer instead is used to reduce the dimensionality of feature map in order to shrink the total number of weights and parameters needed by the network,consequently less computation effort, it is a sort of summary of previous layers information, also pooling help to keep overfitting under control. There are different type of pooling, the most frequent is **max pooling** which take the maximum value of each window of feature map,e.g: we can reduce with max pooling a 4x4 matrix into a 2x2 matrix where each element is the maximum value of each 2x2 quadrant of input matrix. At the end of the feature extract phase,

in a classification problem, there is always a fully connected layer and a softmax function that map the outputs of the network in a probability distribution of the N classes to classify.

The **training phase** works as already described for artificial neural networks [2.1.3], Filters are matrices of weights (and related biases), they too will be adjusted during train;

The optimizer used during train of a CNN often is **SGD** [2.5] or **Adam** (*adaptive moment estimation*, an SGD extension/alternative) and **Cross Entropy**[2.4] as loss function that can be expanded for binary and multi-class problem.

Dropout [9] is often used to reduce overfitting, during train,usually one time for epoch, some neuron outputs are randomly ignored (or *dropped out*), their contribution will be null in the network output of that epoch, at the next epoch dropped-out neurons are restored, and then other randomly neurons are dropped-out,and so on. We can see this operation as a sort of 'mask' applied to the network's layers that change in every epoch of the train that prevent the network to adapting to much to the data.

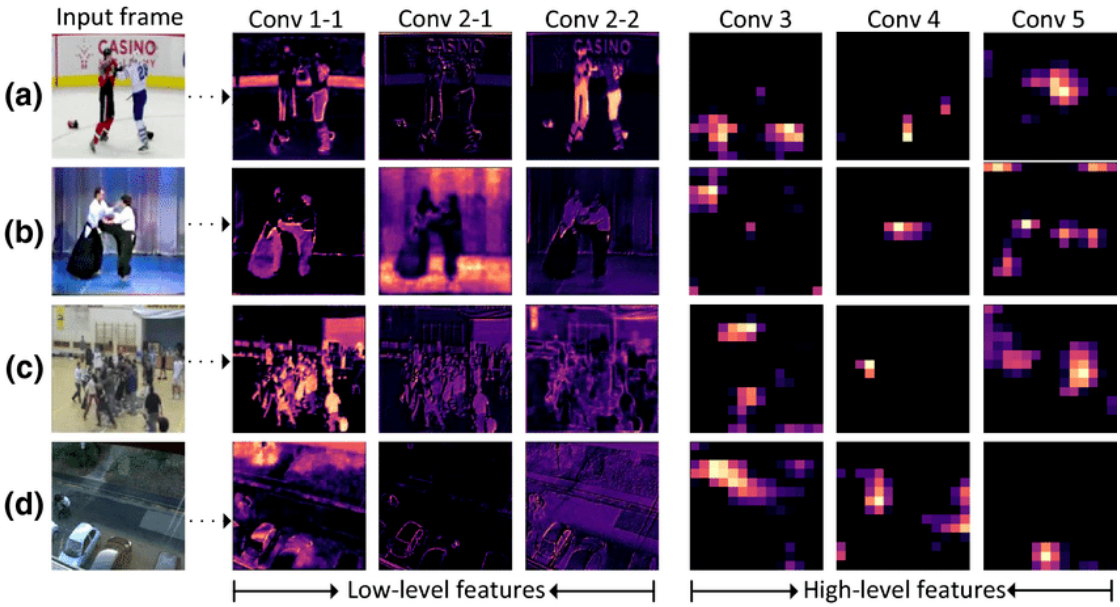


Figure 2.11: Visual example of low-level and high level feature maps from several conv layers

2.2 Object Detection

The term object detector refers to a process that involves two main tasks in computer vision:

- **Image Classification** it is the main and simple task of computer vision, predict the class label (from a fixed list) of an object in a image,usually fit well with images with one main object,the main purpose of convolutional networks.
- **Object Recognition** locate the presence of one or more relevant objects in a image and then indicate the position of these objects with **bounding boxes**, algorithm is not able to understand what kind of object it is, but is able only to identify the location of a possible objects in a image. Usually a Bounding Box is expressed in term of box center,width and height,expressed in pixel and normalized.

Object detection perform both recognition and classification; given an image the goal is to extract different object and then predict their class label, so the network output will be a set of bounding boxes with related label.

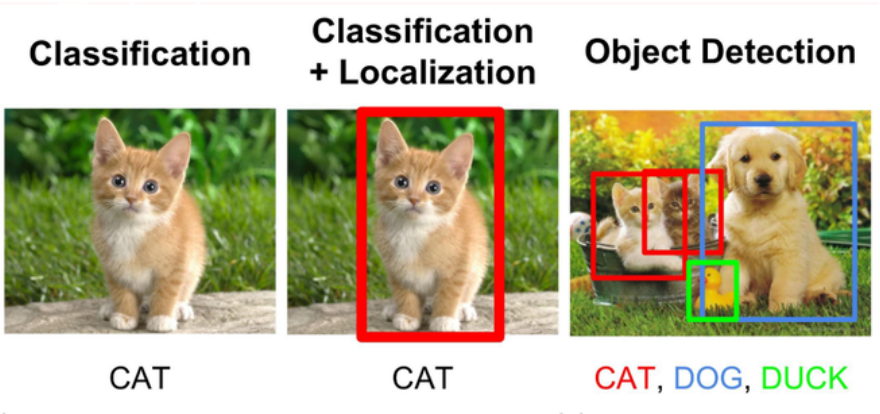


Figure 2.12: Difference between classification,recognition and detection.

Object detection process can be achieved in two main ways:

- **Two Stage detectors**: Detection process is executed in two different stages, in first stage there is a network *(often called region proposal network)* that generate region of interest inside the image, in the second stage will be another distinct network that will classify these proposed regions. This kind of detector usually are slower in inference phase but more accurate than single stage,this architecture is used in **RCNN-family** *"Region-Based Convolutional Neural Network"* (2014-2015). [10]

- **One Stage**: Whole detection process is executed by just a single network that take the full resolution image at input and will predict bounding boxes without the region proposal phase. The process it is generally possible thanks to the use of fixed boxes of different aspect ratios in different part of the image, networks with this architecture are usually faster in inference phase than two stage detector,however are less accurate. These networks can fit well in a real-time applications, some examples of this architecture:
  - **SSD** *Single Shot MultiBox Detecor* (2015) [11].
  - **YOLO** *You look Only Once* (2016-2021) [12].

2.2.1 Object detection metrics

Since as we seen, object detection concerns two distinct problem: *Classification and Regression*, order to evaluate object detector performance, we need metrics that in some ways combines these two problems.Some of these metrics are similar to those described here 2.1.5 .

Intersection over the union: IoU

IoU describe the amount of overlap of two boxes,is used to calculate the distance between bounding boxes generated by the model and ground truth ones. With two distinct box define:

$$IoU = \frac{Area\ of\ intersection}{Area\ of\ union} \tag{2.11}$$

IoU can have values between 0 and 1,higher IoU mean that two boxes are almost overlapping. We can use a **IoU Threshold value** to display only the bounding boxes that have an IoU value above that threshold.

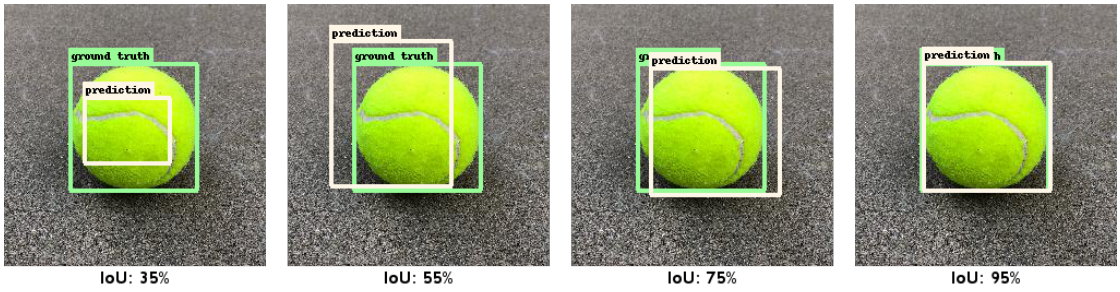


Figure 2.13: Example of different values of IoU

mAP, mAP@.5, mAP@.5:.95

The **precision-recall curve**, similarly to already seen in 2.1.5 is the curve generated by the plot of precision 2.1.5 on y-axis and recall 2.1.5 on x-axis across many threshold values. Threshold is that minimum probability value for which a sample can be classified as positive (or belonging to a certain class) like a *confidence*, the PR curve show us the tradeoff between precision and recall for different confidence values,We can see it as a tradeoff between false positives and false negatives. Object detection often deal with multi-class problem,this mean that the PR curve must be calculated for each class.

AP (average precision) is the area under the PR-curve, An high value of AP means high precision and high recall for a certain class.

mAP instead is the average of AP calculated for each class:

$$\textbf{mAP} = \frac{1}{n}\sum_{i}^{m} AP_i \quad \textit{for } n \textit{ classes} \tag{2.12}$$

Usually mAP is evaluated with different values of **IoU threshold**,e.g mAP@.5 mean the mean of AP calculated for each class with a IoU Threshold > 0.5, if not specified mAp is calculated across IoU values. Generally we can consider a positive result an IoU >0.5, the metric that gives us the most information about the goodness of an object detection and the most used in competitions [13] is the **mAP@.5:.95** that mean the mAp calculated over IoU threshold between 0.5 and 0.95. Having an high score in this metric implies that the model is effective in recognizing the objects in an image and is also able to correct classify them.

2.2.2 R-CNN,Fast R-CNN and Faster R-CNN

R-CNN

Region-based Convolutional Neural Networks are one of the first approach that that applied deep learning models to object detection,RCNN mean *Regions with CNN features* and it is a two stage detector. The main idea behind R-CNN [10] is that the first stage propose some regions of interest from an image,indipendent from the category and then send those regions at second stage that is a fully convolutional neural network, in order to compute features from each proposed region,then the final stage is a fully connected layer (SVM in original paper).

Region proposals can be generated in several ways, in the original paper was used **selective search** that is a segmentation algorithm,which generates a **segmentation map** of the image based on pixel intensity using a graph based segmentation method [14]. This segmentation map gives us an idea of potential objects in the image, the bounding boxes are generated by the iterative merge of

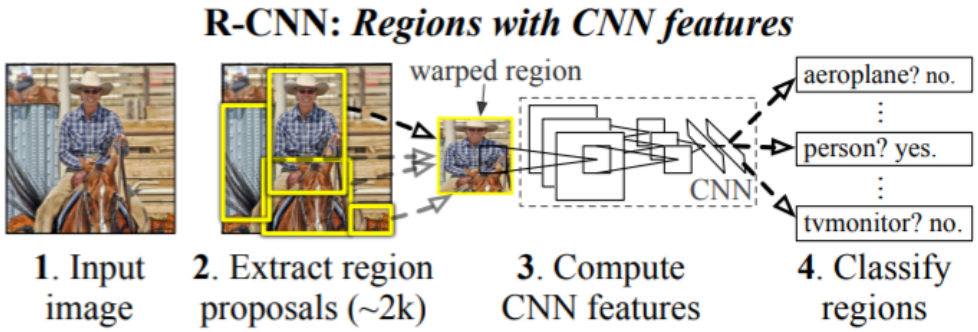


Figure 2.14: Example of Region proposal network pipeline

the segmentation map,and will be drawn from the generated bigger areas, this process can be seen in fig 2.15.

Figure 2.15: Example on how bounding box are generated by a segmentation map

Bounding boxes generated by the segmentation map may not be centered to respect of where the subject is,for this reason the author also has included at the end of first stage a *bounding box classifier* in order to refine the bounding box coordinates. Next,each proposed region is cropped in order to be compatible with CNN input layer; In the second stage there is an CNN that will extract features from each proposed regions. In the paper the author use the convolutional part of AlexNet and then an SVM that classify the CNN output.

To deal overlapping of two bounding boxes that have same class,a bounding box will be reject if has an high IoU with another box that has an higher confidence score.

Although R-CNN achieves very good results in terms of accuracy, the limitation of the architecture is easy to see: If there is a single CNN processing a single region, and in an image there can be hundreds, it requires a lot of computation and consequently a lot of time, this does not make it suitable for real time applications.

The train of a R-CNN has as distinct train phases:

- CNN network with softmax classidier
- SVM classifier
- Bounding-box regressions

The training phase is very slow (about 84h) and inference phase it also slow, about 50s at image.

Fast R-CNN

In order to solve the problems and drawbacks of R-CNN the same author proposed **Fast R-CNN** [15], the basic idea is very similar, but instead of use a CNN to extract feature map from each region proposal, which is a very expensive process, a sort of context switching is performed: The CNN is used to extract feature map from the entire image and then regions will be proposed from this feature map, this mean that in fact the CNN phase is executed only once for image. A new layer has been inserted between the convolutional and the fully connected layers **ROI Pooling layer**, that map regions of interest in a fixed size input for the output fully connected layers,for the classification task instead,SVM is replaced with a softmax function. The bounding box regressor also is used.

During train from a single image,Fast-R-CNN process mini-batches of ROI

The result is that Fast R-CNN is much more efficient than a regular R-CNN, reporting a boost of performance in terms of inference speed of 10x than old one, has also increased performance in terms of mAP.

The train of a FAST R-CNN unlike R-CNN,is single step with multi-loss problem:

- CNN network (classification loss)
- Bounding-box regressions (localization loss)

Training phase of Fast R-CNN is about 10x factor less than original R-CNN, the test time,instead has been reduced from more than 50s to about 2 seconds, on gpu. We are talking huge improvements compared to R-CNN,like a 10x improvement.

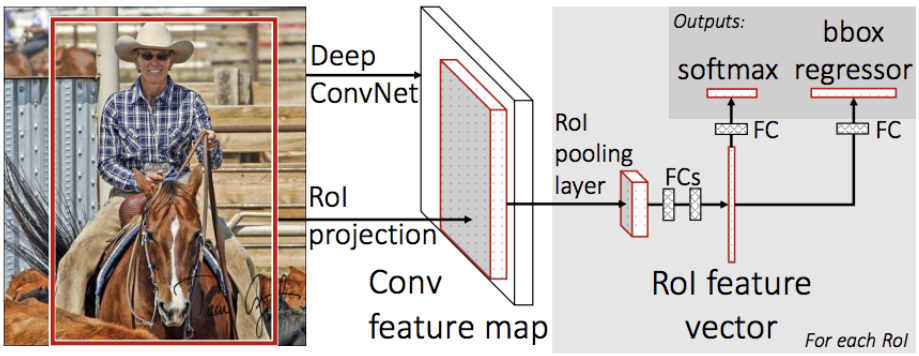


Figure 2.16: Fast R-CNN architecture from original paper [15]: An input image and multiple regions of interest (RoIs) are input into a fully convolutional network. Each RoI is pooled into a fixed-size feature map and then mapped to a feature vector by fully connected layers (FCs). The network has two output vectors per RoI: softmax probabilities and per-class bounding-box regression offsets. The architecture is trained end-to-end with a multi-task loss.

Faster R-CNN

Faster R-CNN is the latest model belonging to the R-CNN family, developed in 2016 and as the name suggests is also the fastest and performing of the family. Although Fast-RCNN has optimized execution times by reduce convolutional layers for feature map extraction, the region proposal part remains computationally intensive, and is considered the bottleneck while CNN phase can be achieved easy with GPU but the region poroposal method was implemented on CPU. The author of **Faster R-CNN** [16] proposed as a solution a **Region Proposal Network (RPN)** to predict proposals from feature map, RPN share convolution layers with the convolutional network.RPN receive at input the feature map generated from convolutional layers and generate a list of possible proposal by sliding a small CNN over the feature map. For each sliding window, RPN predict if the portion of the image over window is an object or not, the prediction on each window is made on K fixed size boxes called **Anchors**.

RPN model is trained separately from the classification part, but the difference,thanks to the use of anchors is that the ROI pooling has a fixed size determined by the K number of anchors. Faster-R-CNN achieves even better results than the previous ones, with an inference time of about 300ms.

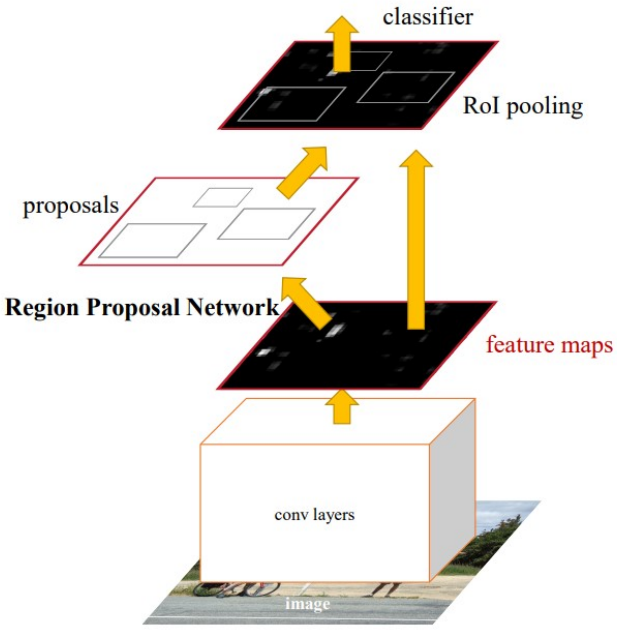


Figure 2.17: Faster-RCNN architecture

2.2.3 SSD: Single Shot Multibox Detector

Like the name suggest,and unlike the previous models, SSD is a single shot detector,this mean that the tasks of object recognition and classification are achieved by a single forward pass of the image in a single network. Proposed in late 2016 [11], The author of SSD speeded up the object detection process by eliminating the need of a region proposal network by a simple idea: compute both boxes and classification by the use of small (3x3) convolution filters applied to feature map for a fixed set of default bounding boxes for each image cell,similar concept to anchor boxes in Faster-R-CNN however in SSD are applied to several feature maps of different resolutions. Since the number of generated bounding boxes and relative classes can be really a lot and they can overlap,there is also a **non-maximum suppression** step that produce final detections aggregating these boxes. The base network of the original SSD is **VGG-16** Training is executed end to end, ans also here the training objective is a multi loss problem; localization loss and confidence loss.

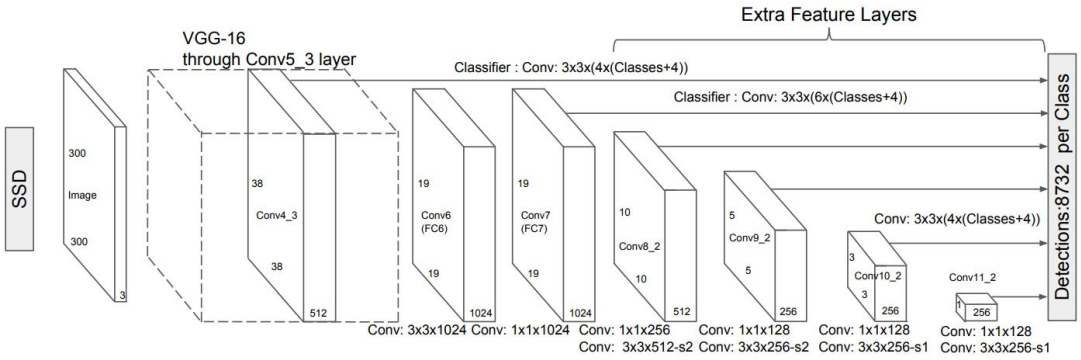


Figure 2.18: SSD architecture from the original paper

2.2.4 YOLO: You Only Look Once

YOLO (You Only Look Once) is a very fast object detection architecture born in 2016 by Joseph Redmon [12], Like the SSD network,yolo perform both classification and localization with a single convolutional neural network. The operation is similar to that seen for the SSD network: The input image is divided into a grid of $S * S$ cells,and for each cell the network predict B bounding boxes with relative confidence scores and C class probabilities. These confidence scores reflect how confident the model is that the box contains an object,if no object present in that cell the value should be 0,otherwise the objective will be the IoU 2.13 between above box and ground truth box.

The C class probabilities are related to grid cell,regardless of the number of bounding boxes that it contains,represent the probability that the object belongs to a certain class.In order to generate the final network output will be a **Non-Maximum suppression** layer, as already seen for ssd, which will aggregate the bounding boxes that seem overlapping in terms of coordinates and class probabilities based by an IoU 2.2.1 threshold. over time this architecture has evolved and new models have been introduced:

- **YOLO v2**: It is not a radical change of architecture, but several improvements have been made to the original model in order to improve speed and accuracy,like biggest input size (416*416),use of Anchor boxes,and the C class probabilities are now calculated at box level instead of the cell level.
- **YOLO v3**: Others small improvements, like the change of loss function,more bounding box predicted by the network and **multiscale box prediction**

These three versions of yolo are the latest from the original author,are based on a framework that he created: **Darknet**. The author, for ethical reasons abandoned the development of the yolo family and darknet framework, but subsequently the

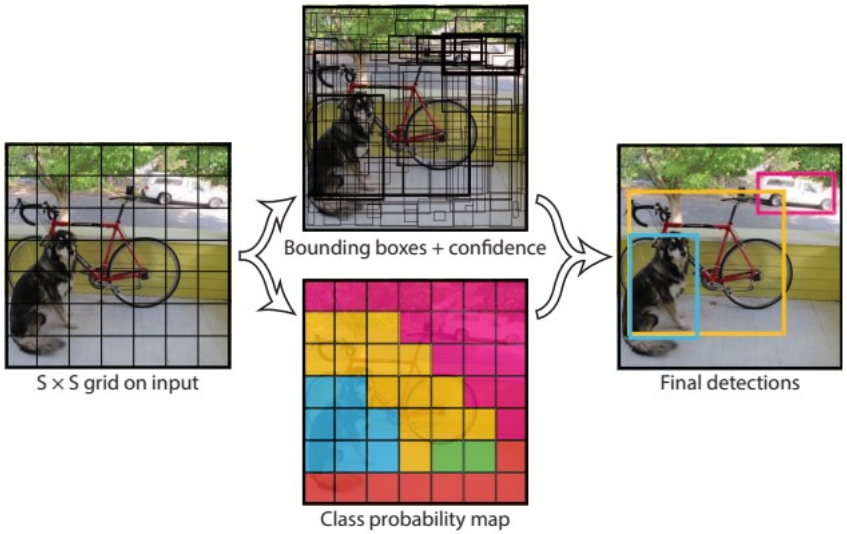


Figure 2.19: Example of YOLO process, we can see how the high number of bounding boxes predicted and the class probability map, are associated in order to generate final result

work with this architecture continued with several forks no longer based on darknet original framework e.g: yolov4, yolov5, yoloX.

2.3 Edge/Embedded Devices

As embedded device we identify all that series of special purpose systems, that are programmed to perform a specific task in a larger system. The Embedded device integrates all the hardware and software necessary for its operation and are often used to control the system that hosts it, simple examples could be: the control unit of a car, traffic light controller, digital medical system, avionics system etc.
We need to make a first major separation in:

- Embedded device based on **microcontrollers (MCU)** , a microcontroller self-contain cpu, memory and peripherals on same chip,typically MCUs are to be programmed at low level as they are designed to be special purpose. Device based on MCU are extremely energy efficient but typically have low computational power, they are not exactly suitable for artificial intelligence applications, however there is growing interest in this area, an interesting project is tinyML (`https://www.tinyml.org/`) An organization that grows ultra-low power machine learning technologies.

  Some examples of MCUs can be **Arduino, STM32, ESP8266, ...**

- Embedded device based on **microprocessors** This type of devices are more complex, based on microprocessors, which use memory and external peripherals. In this case **ARM** architecture is the main market player for low power microprocessors. It is also possible to integrate several microprocessors with memory, peripherals and possibly a gpu into a single chip called system-on-chip. Typically devices based on microprocessors or SOCs allow the creation of real "mini-computers" (Single Board Computer **SBC**) with a real operating system in execution and several input/outputs interfaces, this means that they can be programmed in a general purpose way, they are more powerful devices but less efficient than the previous ones and they are the subject of this thesis. Some examples of this kind of devices: Raspberry pi, Nvidia jetson nano, Coral Dev Board

ARM

ARM are a family of processors designed to have a low energy impact, dominating the mobile and embedded sector where energy saving is fundamental. Arm does not sell the hardware directly but licenses the production of ARM architecture cores.

It's a RISC (reduced Instruction Set) architecture as opposed to CISC (x86) architectures commonly used in desktops, the main difference is that RISC use only simple and fixed length instructions that can be executed in one clock cycle, this makes predictable the behavior of the system from the energy point of view.

2.3.1 NXP iMX8M-Plus

The board that I had available to carry out the experiments is based on the NXP: **imx8m plus** processor. It is a quad core arm-based device,at cpu level comparable with a raspberry pi4. According to nxp: *The i.MX 8M Plus family focuses on machine learning and vision, advanced multimedia, and industrial automation with high reliability. It is built to meet the needs of Smart Home, Building, City and Industry 4.0 applications.* features:

- **Quad-Core 1.8 Ghz** arm based Cortex-A53 CPU
- **NPU** (Neural processing unit) operating at up to **2.3 TOPS**
- Dual image signal processors (**ISP**) and two camera inputs
- **H.265 Encode/Decode**

Specifically I used the **nxp evaluation kit** for the i.MX 8M that exploit processor performance,is a SOM (system on module) which houses the processor,and it includes:

- RAM **4GB LPDDR4**
- eMMC 16 GB and support for microsd
- HDMI, Bluetooth 4.1, Ethernet, **Wi-Fi AC**

A first class equipment for this type of devices, on the board is possible to run an **embedded linux distribution** with the yocto project.

Yocto

OpenEmbedded is a software framework used for creating Linux distributions aimed for, but not restricted to, embedded devices. The build system is based on BitBake recipes which behave like Gentoo Linux ebuilds.
The **Yocto** Project is a set of templates, tools and methods that allow to build custom embedded Linux-based systems. Yocto is not to be seen as a ready-made embedded linux distribution, but we can see it as a base that allows you to create a custom one according to your needs. Core components:

- **BitBake**: It is the build engine, it interprets metadata in order to to configure and build packages and images.
- **OpenEmbedded-Core** is a set of base layers shared beween all openEmbedded based systems.

Figure 2.20: i.MX8M plus evaluation kit

- **Poky Distribution** it is the reference embedded OS.

I've build the official nxp recipes related to iMX8 family in order to update the evaluation kit packages.

NXP eIQ®

NXP offers custom versions of machine learning libraries optimized for their processors, and it is contained in a custom yocto layer, this layer integrates execution providers that allow machine learning libraries to take advantage of the integrated NPU, for example a custom version of tensorflow lite or onnxruntime, unfortunately free access to NPU computation is still immature,and today we can access NPU only through these custom libraries and not directly from a library that you can get in official python repositories.

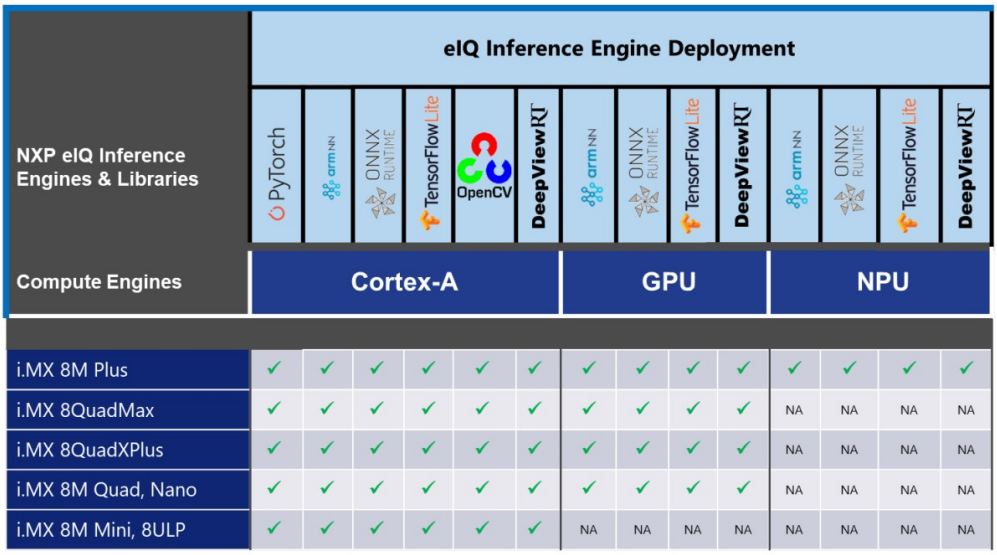


Figure 2.21: eiQ Supported inference engines for i.MX 8M Family

2.4 Network Optimizations

Some of the information in this section was obtained from [17], [18] ,[19] *and* [20].

Edge Devices, such as the one the subject of this thesis introduced in 2.3.1, have limited performance and memory compared to desktop computers, to deploy deep object detection models such as those described in 2.2 on these devices in real-time application is a great challenge, compression and acceleration techniques are therefore necessary so that such models can be run within device memory and performance constraints.
The purpose of these methods is to perform model compression/optimization without significantly decrease the model accuracy, we can summarize recent work in the area of efficient deep neural network optimizations [21] into:

- **Quantization**: Reducing the number of bits required to represent each model's weights. e.g from float32 to uint8. [22]
- **Knowledge distillation**: Training a compact neural network with distilled knowledge of a large model, Teacher and Student. [23]
- Parameter Pruning *untreated* Reducing redundant parameter that are not sensitive to the performance,e.g cut from the network weights close to 0 value.
- Low-Rank factorizaztion *untreated* : Using matrix/tensor decomposition to estimate the informative parameters.
- Transferred/compact convolutional filters *untreated* : Design special structural convolutional filters to save parameters.

In this thesis work, we have primarily focused on quantization and knowledge distillation, The pruning method has untreated, because if it is theoretically useful, pruning is done by zeroing model's weights close to zero, so the tensor become sparse, and' relatively easy therefore to compress the model in terms of dimension (since there are so many zeros), however in the common frameworks the operations on the tensors by default assume dense tensor representation, unfortunately there's not a lot of hardware and software that can benefit directly from sparse Tensors.

2.4.1 Quantization

Quantization is an approach that has shown great and consistent success in both training and inference of DNN models. While the problems of numerical representation and quantization are as old as digital computing, deep learning networks offer unique opportunities for improvement [19].

After the training phase of an artificial neural network, its weights (parameters) are stored as 32 bit floating-point values, reducing the number of bits used to represent network's weights and activation, can lead to a significant reduction in MAC (Multiply–accumulate) operations required for an inference and reduce the size of the network. Quantization, then, consist into mapping float32 (weights or/and activations) into a lower precision ones (e.g. uint8,int8).

Uniform Quantization

First we need a function that takes the floating point values of the models and maps them to a lower precision range, a good choice would be the following function:

$$Q(r) = int(\frac{r}{S}) - Z \tag{2.13}$$

where:

- Q is the quantization operator
- r is the real input value (of activation or weights)
- S is the scale factor
- Z is the zero point

Int operator a real value to a integer value by the rounding operation (e.g round to nearest integer and truncate), this type of quantization is called **Uniform Quantization**, since the resulting quantized values (quantization levels) are uniformly spaced, as we can see in figur 2.23 on the left side. It is possible to do the reverse operation, that is from a quantized value back to the real value, with an operation called de-quantization (nota that the real values will not exactly match r, due to tounding operation):

$$r = S(Q(r) + Z) \tag{2.14}$$

Symmetric and Asymmetric quantization

It is important the choice of the scaling factor, since it essentially divides a range of real values r into a number of partitions ([24])

$$S = \frac{\beta - \alpha}{2^b - 1} \tag{2.15}$$

[α,β] represent the bounded **clipping range**, b value represent instead the **quantization bit width**. So to identify the scaling factor, we need to determine the

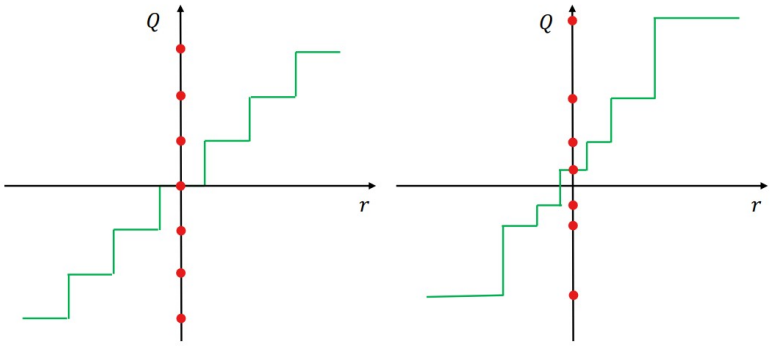


Figure 2.22: *Comparison between uniform quantization (left) and non-uniform quantization (right). Real values in the continuous domain r are mapped into discrete, lower precision values in the quantized domain Q, which are marked with the red bullets. Note that the distances between the quantized values (quantization levels) are the same in uniform quantization, whereas they can vary in non-uniform quantization.* [19]

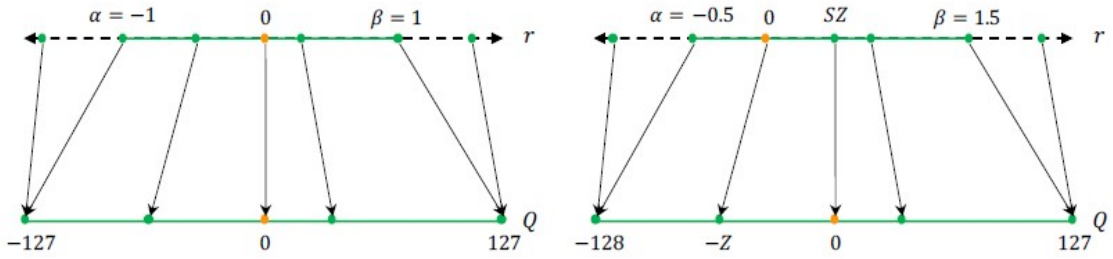


Figure 2.23: *Comparison of symmetric and asymmetric quantization, Symmetric quantization with restricted range maps real values to [-127, 127], and full range maps to [-128, 127] for 8-bit quantization.*

clipping range, this process is often called **calibration**. The simplest choice is to use the **min/max** of the signal to determine the clipping ranges, this approach is called Asymmetric, since the range is not necessarily centered on the origin (when $-\alpha \neq \beta$). If $-\alpha = \beta$ it is considered symmetric quantization, we can see a comparison in figure 2.23 If we use symmetric quantization we can set Z=0 in the equation2.13, which become:

$$Q(r) = int(\frac{r}{S}) \tag{2.16}$$

Basically, applying symmetric quantization method is easier because of the zero point Z=0, however it is not the optimal approach when the ranges are skewed and asymmetric, in this case asymmetric quantization is preferred.

Static and Dynamic quantization

So far we have discussed how to determine the clipping range [α,β] to perform the quantization, this process can be done in different times and modes, there are two main classes of range calibration algorithms: Static and Dynamic.
After the process of training a model, its weights have fixed values during inference, which makes it relatively easy to perform offline clipping range calculations, however the activation maps are different for each input,hence the need for two different approaches to calculate the clipping range. In dynamic quantization the range is calculated dynamically at runtime inference, for each activation map, which can result in a performance overhead, given by the extra computation, however this method is the most accurate, since the range is calculated for each activation. In static quantization the clipping range is pre-calculated offline and is static during inference. To pre-calculate the range, one method used is to use "mini-datasets" of calibration, to be given as input to the network in order to calculate the fixed values of clipping range. Static quantization has less computational overhead than dynamic quantization, but is less accurate because the clipping ranges are fixed, and may be different for each activation.

Quantization aware training

So far we have discussed the quantization of a pre-trained model, this process is defined as **Post Training Quantization (PTQ)**, however it is possible to perform the quantization process during the model's training phase , this approach is called **Quantization Aware Training (QAT)**.
In QAT forward and backwarard phases are performed on the floating point model, but the model parameters are quantized after each gradient update. It is important that the quantization is executed after the weights update (performed in floating point precision) as the accumulated gradient can result in zero or high error gradients [25][19].
The int (rounding) operation in equation 2.13, is not-differentiable and without any approximation the gradient will be zero almost everywhere. To approximate the gradient of this operator, a Straight Through Estimator (STE) was introduced, which essentially ignores the rounding operation and approximates it with an identity function, the process is described in the figure 2.25. Despite the coarse approximation introduced by the STE operator, QAT it often works well in practice, except for ultra low-precision quantization such as binary quantization, however, the main disadvantage of QAT is the computational cost of retraining the model, and it may take several epochs to recover the original accuracy.

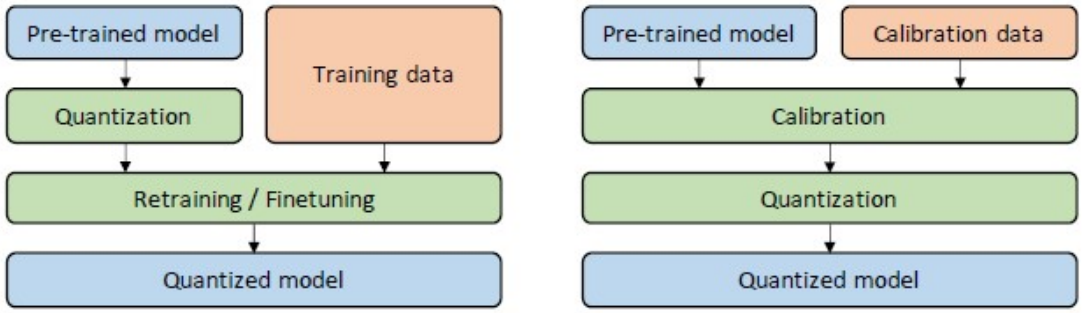


Figure 2.24: *Comparison between Quantization-Aware Training (QAT, Left) and Post-Training Quantization (PTQ, Right). In QAT, a pre-trained model is quantized and then finetuned using training data to adjust parameters and recover accuracy degradation. In PTQ, a pre-trained model is calibrated using calibration data (e.g., a small subset of training data) to compute the clipping ranges and the scaling factors. Then, the model is quantized based on the calibration result. Note that the calibration process is often conducted in parallel with the finetuning process for QAT.* [19]

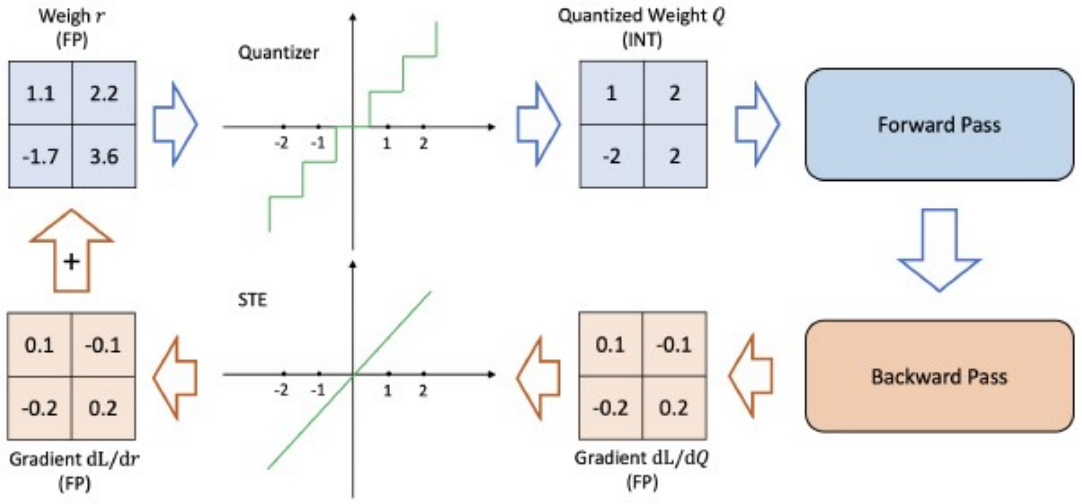


Figure 2.25: *Quantization-Aware Training with Straight Through Estimator*

2.4.2 Knowledge Distillation

Knowledge distillation is a set of methods whose purpose is to distill knowledge from a large deep neural network into a smaller network [23].
In order to address the issue of deploying deep models on low power devices, respect to quantization, knowledge distillation methods uses a different point of view/approach : instead of optimizing in terms of perfomance speed an existing model, we intervene during the training of a lighter model to improve its performance in terms of accuracy by introducing the "suggestions" of a more accurate model.
In Knowledge distillation Teacher and Student networks are formally introduced, the basic idea is that the student model must mimic the teacher model, in order to achieve competitive performance, compared to stock-trained student model, However, the main problem is how and in what way, to transfer the knowledge. Basically, there are three core of Knowledge Distillation:

- Knowledge
- Distillation Algorithms
- Teacher-Student Architecture

In this thesis work we will focus on two main types of knowledge distillation: output based and feature based.

Response (Output) based Knowledge Distillation

In Response based knowledge distillation, the student is only affected by the response of the last output layer of the teacher model. It is the simplest method of knowledge distillation, the core concept is to mimic the final prediction of the teacher [20].
Given a vector of logits (predictions) z as the output of final network layer, we can define a **distillation loss** for response-based kd in this way:

$$L_{RBD}(z_t, z_s) = \mathcal{L}(z_t, z_s) \tag{2.17}$$

Where $\mathcal{L}$ is the divergence loss of predictions, and z_t and z_s are the logits of teacher and student. Response-based knowledge distillation can be applied in different contexts, for example in object detection task, the response may contain the predictions in addiction with the bounding box coordinates [26]. We can formally introduce soft targets in the context of image classification, as the probability that the input belong to a certain class, and can be expressed with a **softmax function** (*as already described in previous chapters*). Essentially, the total loss of the kd process is the join of the distillation loss and the student loss, we can see the process in figure 2.26, please note that the student loss is always defined as

cross entropy loss (2.4) between the ground truth label and the soft logits of the student model.

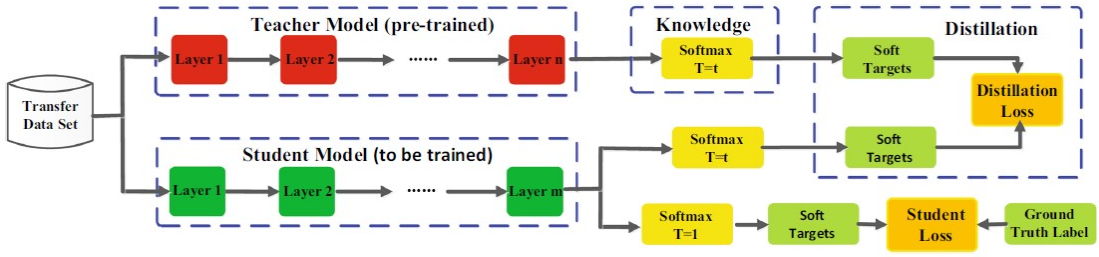


Figure 2.26: *Detail of Output Based Knowledge Distillation architecture, we can see the total loss is the combination of distillation loss (calculated from soft targets of both teacher and student) and the student loss calculated from student soft target and ground truth. Please Note that it is possible to intervene on the influence of the teacher, using a multiplication factor on the distillation loss.* [23]

Feature Based Knowledge Distillation

As we have seen in the previous chapters, deep neural networks are able to learn different levels of feature representation, called feature maps, which the representation is more and more abstract going deep along the network layers. Therefore, as well as the output of last layer of the teacher model, we can use also the output of intermediate layers as knowledge to supervise the training of the student model. This knowledge representation is often called **Feature Based** or **Feature Imitation** and we can see in figure 2.27 the generic architecture of feature-based knowledge distillation. We can define the **distillation loss** for feature based knowledge distillation that use feature maps as knowledge as follows:

$$L_{FBD}(f_t(x), f_s(x)) = \mathcal{L}(\Phi_t(f_t(x)), \Phi_s(f_s(x))) \tag{2.18}$$

Where the $f_t(x)$ and $f_s(x)$ are the **feature maps** of intermediate layers between teacher and student models. When the intermediate layers doesn't have the same shape among the two networks, we need a **transformation function**, shown in the equation as Φ.
$\mathcal{L}$ instead is the similarity function that match feature maps from teacher and student models. Over time, different combinations of loss functions *(l2-norm, l1-norm, cross-entropy and maximum mean discrepancy)*, knowledge types *(feature maps, attention maps, feature representation, ..)* have been explored, however, the biggest focus, is in how to effectively choose the hint layers from teacher model and the guided layers from the student model. [27].

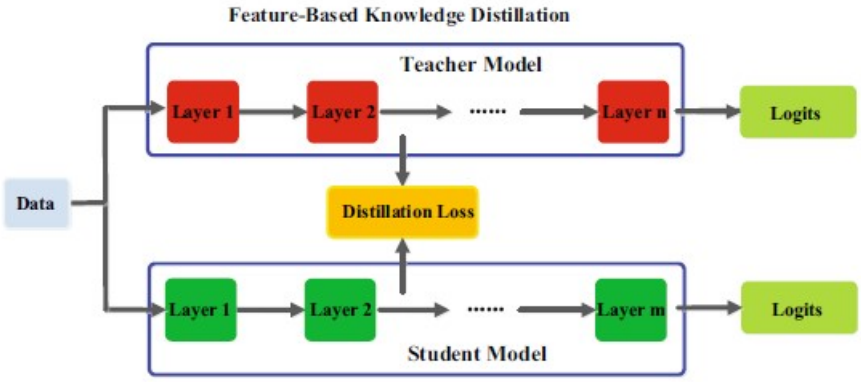


Figure 2.27: *Generic Feature-based knowledge distillation* [23]

Chapter 3

Model Compression Box

Model compression box is a custom python framework that I developed in order to provide useful tools for compression, export, test and validate object detection models with different engine/format.
It has been designed in a completely modular and configurable way in order to easily and quickly expand it with new features like supported models, inference engines, input sources, and more. At the architecture level of the software to make it modular I followed the
Strategy Pattern: *"Strategy is a behavioral design pattern that turns a set of behaviors into objects and makes them interchangeable within original context object."* [28].

For the development of this framework have been used some of the most famous libraries related to machine learning and deep learning such as: *numpy, opencv, sklearn, pytorch, onnxruntime, tensorflow and others.* It is possible to use the framework both in desktop and embedded environment, obviously the embedded device must be able to run python code and resolve dependencies.
Model compression box is formed by two main distinct module: Detector and Compressor that perform very different but complementary tasks.

3.1 Detector

The main goal of the detector module is to provide robust tools to evaluate both qualitatively and quantitatively object detector models in inference. Natively it supports several object detection models and inference engines such as onnxruntime, tensorflow lite and pytorch. It is easy to update with new models, you only need to define the pre and post process phases.

Where supported, the inference phase can be performed on both cpu and gpu, several metrics will be tracked such as: inference latency, pre/post process latency, accuracy, mAp (on coco dataset), model size and others device hardware informations.
It is possible to archive the inference results, both in terms of performance reports and images/live video with bounding boxes. All detector features are configurable by startup arguments or by configuration file.

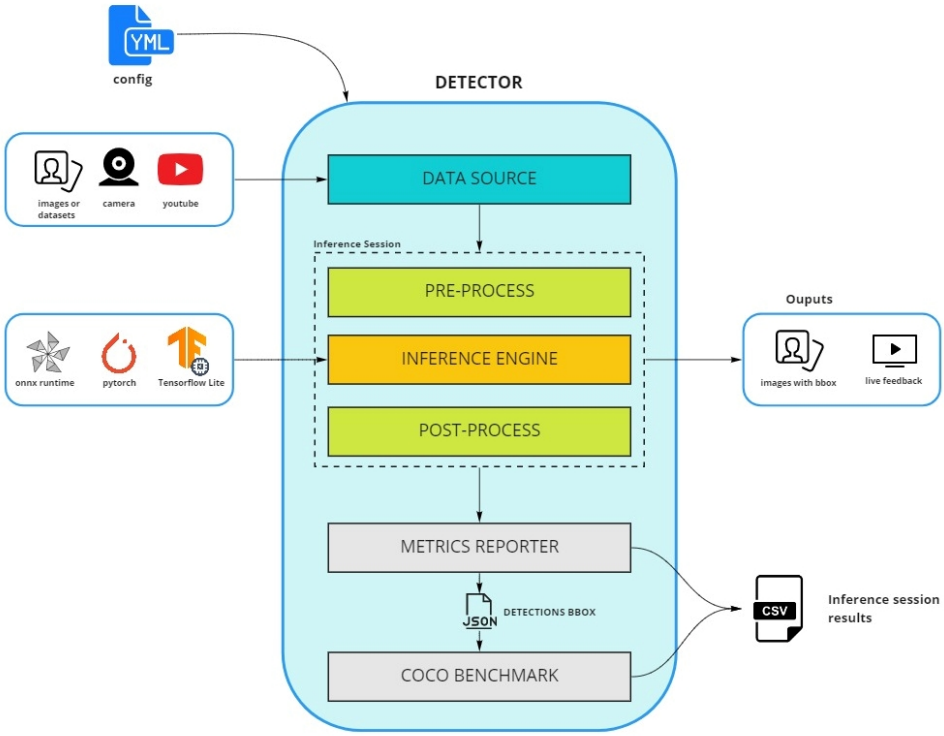


Figure 3.1: High-level architecture of the detection module

3.1.1 Configurations

In order to get orderly and schematic results I decided to make the whole framework parametric, the detector can be configured through command line arguments or from a configuration file in the project root (I chose yaml format for compactness and readability). Supported configurations are:

Model

This field defines which model to use for inference, the file extension determines which engine to use for inference,available engines:

- **.onnx - Onnxruntime**
- **.tflite - TensorFlow Lite**
- **.pt .pth (WIP) - Pytorch or OpenCV-dnn**

All object detection models with weights in this format are compatible, however must be specified the phases of pre and post process that can vary according to the architecture of the model,hose currently available are:

- **Yolo v5**
- **Yolo v2**
- **SSD Mobilenet v1**
- **Faster-RCNN**

Images folder

Indicates the path to the image directory to be used as a dataloader for inference phase (detections). all major image extensions and any resolution are supported, all images in the folder will be processed. this option is mutually exclusive with video and webcam.

Video

Indicates the path of a video on which perform inference, as video formats are supported: *.mov, .avi, .mp4, .mpg, .mpeg, .m4v, .wmv, .mkv* .
Alternatively you can specify a url of a youtube or RTMP video to do streaming video detection. this option is mutually exclusive with image folder and webcam.

Webcam (Boolean)

Flagging this value the detections will be made directly live from the webcam, a window will be automatically opened with the webcam live output with the detected bounding boxes.

Labels

Indicates which labels to associate to the detected classes, currently the labels of the **VOC** and **COCO** datasets are supported.

Resolution

Images during pre-processing will be resized with this value. this value can depend on the model used, some models accept any type of resolution as input, while in most models the resolution is fixed.

Threshold

Indicates the minimum detected class accuracy allowed, detected classes with an accuracy below the threshold value will be discarded.

IoU threshold

Indicates the value of Intersection over union to be used, a value close to 1 indicates an almost total overlapped area of the predicted bounding box compared to the real one, so very close. Bounding boxes with IoU below the set threshold value will be discarded. for more information refer to [2.2.1]

Test (Boolean)

Enabling this flag allows to benchmark the performance of a model in terms of accuracy and mAp [2.2.1], this option is valid only on models trained on the COCO dataset and on any subset of COCO-2017 images and labels. However all other performance metrics will always be recorded regardless of this flag.

Show (Boolean)

Displays an on-screen output of the generated images with their detected bounding boxes and classes.

Save (Boolean)

Save the generated images with their detected bounding boxes and classes. The function is available only if the data sources are images. The feature of saving videos with added bounding boxes is under development.

3.1.2 Session Result: Report and Benchmarking

Tracking all the performances of the tested models is essential for the analysis that I will do in the next chapters, it is a fundamental aspect of the Model Compression Box because the detector module allows us to analyze the models in different contexts but representative of reality.
Regarding the inference speed metrics we will average the results on all the images of the inference session (or frames if we are using a video) both for the inference time and the pre and post process time and then we will calculate the total FPS.
Regarding accuracy metrics like accuracy, mAp 0.5 and mAp 0.5-0.9, available only if you use a subset of the images of the COCO dataset and its annotations file, first will be saved a .json file with all the detected detections and then will be processed with the official tools of the COCO dataset (pycocotools) in order to have results comparable with the community.
Then a summary of the information of the inference session in tabular format will be append in a single .CSV file containing all the information of the past experiments, in order to create a single dataset containing all the information that I will need for analysis.

```
{
    "model_name": "yolov5l",
    "node_name": "DESKTOP-U10***",
    "device": "GPU",
    "resolution": 640,
    "runtime": "onnx",
    "fps": 17.54,
    "average_inference_time": 0.057,
    "average_preprocess_time": 0.003,
    "average_postprocess_time": 0.006,
    "mAP@0.5:0.95": 0.526,
    "mAP@0.5": 0.706,
    "data": "./data/nanoval2017/images",
    "model_size": 177.973,
    "System Info": {
        "System Info": { ... },
        "CPU Info":  { ... },
        "RAM Info":  { ... },
    }
}
```

Listing 3.1: Detail of information tracked for each inference session.

3.2 Compressor/Exporter

In the Exporter/compressor module i developed tools to compress, optimize and finally export object detection models. I choose **.onnx** as output format for the models, because in my tests it was found to be the most compatible and versatile format. More info about onnx and onnxruntime in **??**.

This module will mainly deal with **quantization** 2.4.1 and **pruning**

UINT8 Dynamic Quantization is available for any input .onnx model, through onnxruntime there are three ways of quantizing a model:

- **Dynamic quantization**
  Quantization parameters (scale and zero point) are calculated dynamically on-flight for each activation.
- Static quantization *untreated*
  Quantization parameters are calculated offline with an calibration dataset,all the activations have same parameters.
- Quantize-Aware training quantization *untreated*
  Quantization parameters are calculated in the training phase.

The dynamic quantization obtains better results even if it is computationally more expensive. Since the main way, and also the way used in this thesis, to represent a quantized model in onnx is to convert the operators into a quantized counterpart defined by onnx standards, it is very important to pay attention to the **op-set** used for export model from another framework to .onnx format,since many 8-bit operators are unavailable for older onnx op-set. It is not possible to update the op-set of an onnx model but you need a re-train the and then an export to onnx with updated op-sets. Another way to represent quantized models is through the quantize and de-quantize paradigm, however it is not treated in this work.

The pruning phase is carried out by the pytorch framework

Whether you want to apply pruning, quantization or simply export a model, the output of this module will always be an .onnx model.

3.2.1 Configurations

As with the detector module, all behaviors of the exporter module are configurable from startup parameters or by a configuration file.

Model

The input model of exporter, can be **any model in .onnx** format or one of these models on torch-hub:

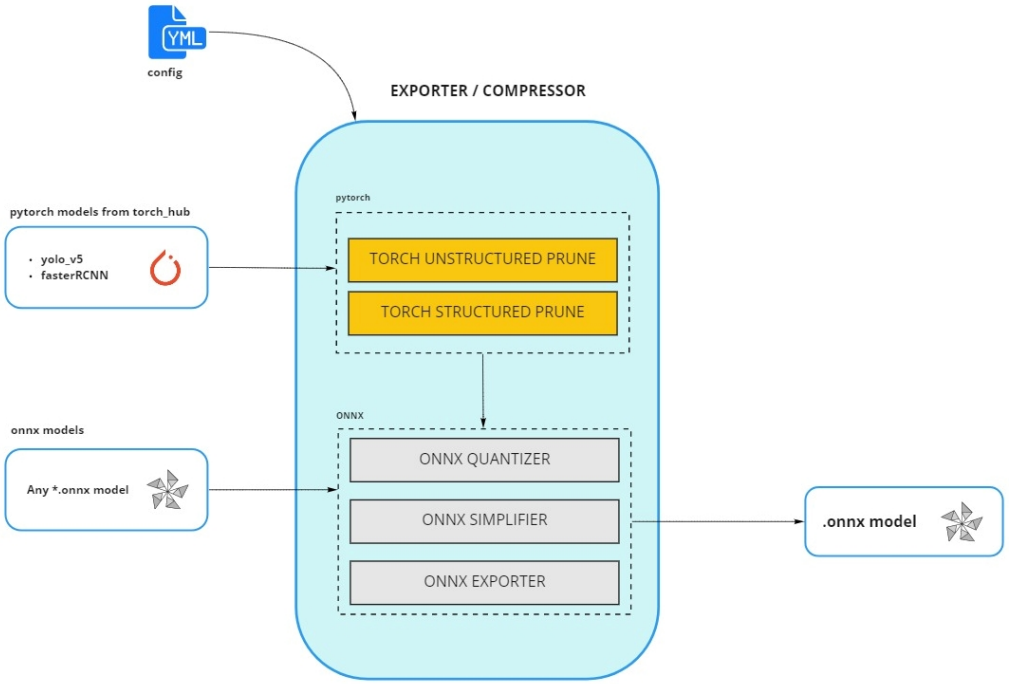


Figure 3.2: High-level architecture of the export module

- Yolov5 family (n-s-m-l)
- FasterRCNN

Res

The resolution (in terms of pixels) of the input images to the model, this option will change the input layers of the model fixing the input tensors to the given resolution.

nb: on some models through onnxruntime is possible to define dynamic axes input range, this allow to use images of any resolution as input of the model.

Quant (Boolean)

This flag indicates whether to quantize the model, it is available for all supported models (both torch and onnx). Quantization will be applied to all operators of the model with these properties:

- Uint8 Weights and Uint8 activation

- Dynamic Quantization
- Per channel Quantization

Simplify (Boolean)

This flag allow us to simplify an onnx model; Analyzes the entire computational graph created and replaces redundant operations with their constant outputs. Powered by onnx-simplifier python library .

Gpu (Boolean)

Use gpu in compression/export computation phase,pay attention that some quantized operators cannot be compatible with GPU computation.

Prune-u (Number)

Applies l1 unstructured per-layer pruning on a pytorch model, with the specified sparsity value. It is the easiest way to do pruning,as a unstructured will be pruned weights per-layer. L1 means that is used L1-distance (Manhattan) to measure the the contributions of the weights to be pruned with a given sparsity.

Prune-ug (Number)

Like the previous command, it applies unstructured pruning to the model, But now the pruning parameters are not calculated for each layer, but for the whole model.

Prune-s (Number)

Applies l1 structured prune on a pytorch model, with the specified sparsity value,with structured prune approach weights will be removed in group for channel.

Chapter 4

Processes, Methodologies and Tools

The main purpose of this thesis work is to analyze the behavior and possible optimizations of object detection models in order to perform the inference operation (i.e predict bounding boxes and relative classes) on an image or a video stream as quickly as possible in an embedded device context, in our case the device is **i.Mx 8M plus** (more details in the appropriate chapter). The device is able to run python code and consequently different deep learning frameworks,however as we will see not all frameworks and the respective representation of the model's weights are suitable for this context. The analysis was performed by dividing the problem into two macro categories.

- **Optimizations that are unaware of the training process**: **t**esting of the behavior of mainly **quantized pre-trained** models on different engines.
- **Optimizations that require re-train the model**: Train from scratch models to apply different types of knowledge distillation.

For both phases different metrics will be used to evaluate performances of the models, both in terms of purely speed and mAp,almost all operations will be performed on a split of the COCO dataset in order to have comparable and replicable results. I developed the **Model compression Box** (3) to help me in both phases in order to easily evaluate on the board different models and optimizations, the evaluation phase will be carried out directly on the board by the detector module (3.1), the unaware training optimization instead by the Exporter module (3.2). As for the optimizations that need to retrain the model, I've been fork project from the original model's repository and modified the original training operations,all the code written in this thesis is purely Python.

4.1 Post-Training Optimizations: Quantization

In this phase I was mainly involved in applying quantization and other training unaware optimizations to different pre-trained object detection models with the purpose of evaluating the differences between quantized and not quantized models in the **inference phase** on the embedded board. It will be also evaluated the behavior of the same models to vary of the used engine, e.g. TFlite or Onnx Runtime.

Although in theory the i.Mx 8M plus board has a **NPU** (neural process unit) that would greatly speed up inference times, **all tests on the device were performed on CPU**.This is because to access the NPU computationally you need different execution providers (like VSI NPU/NNAPI/VX), that will be the engine of the different deeplearning frameworks, however the support for this type of NPU and providers is limited for both onnxruntime and tensorflow lite, in summary the NPU computation is possible through the use of operators supported by the execution provider. for some operators there is a 1 to 1 conversion with pytorch models, while other operators are still not supported, this implies that if we want to run a model on the NPU of the board we must act and modify the network topology possibly modifying or removing unsupported operators.

Object detection models used in this comparision:

- YOLO v5 (s, n)
- SSD Mobilenet v1.10
- Faster-R-CNN

In these models a quantization will be applied, typically a **per-channel dynamic quantization**,after the quantization additional optimizations will be applied if necessary (such as node-graph simplify 3.2.1) it will be applied a fixed input to the model, that is it will be fixed the resolution in terms of pixels of the images that the model accept,then model will be exported in .onnx or .tflite format. for each model (both quantized and non quantized) will be evaluated on-device these metrics:

- FPS *(frame per second)*
- Average Inference time *ms*
- Average pre process time *ms*
- Average post process time *ms*

- mAP@0.5 *percentage*
- mAP@0.5:0.95 *percentage*

Time-related metrics can be collected for all types of images, including for example from the camera. However, metrics related to model accuracy can only be analyzed on supervised dataset,of course. Subsets of the coco dataset were used for both types in the tests, but the tests regarding the mAP to be at least comparable with the literature were done on the **official coco validation split**, which is too heavy to be loaded on the board, for this reason the mAP calculation was done on a desktop machine.

In order to understand which is the more suitable engine for the inference on device, the same quantized model will be evaluated on:

- OnnxRuntime
- TensorflowLite

Performance evaluations in this phase are performed by the model compression box reports, as already described here 3

4.2 Training aware optimizations: Knowledge Distillation

In this type of optimization, unlike the previous ones, we don't try to optimize an already (and possibly deep and accurate) existing network in terms of execution speed, but the focus is different;
We start from a very small network, already fast but not so much accurate who will be the **student** and we train it with the support of a bigger and accurate network that will be the **teacher**. The goal of the training process is to make the accuracy of the student network tend to that of the teacher.

In this second phase I analyzed different forms of knowledge distillation during training of different combinations of teacher-student networks. Different trainings were performed to analyze the behavior as the hyperparameters related to knowledge distillation varied.

Specifically I focused on the YOLO v5 family, since they are already very fast and well documented object detection networks, the basic idea is to use a deep version of YOLO v5 as teacher network and a smaller one as student network, used networks are:

- YOLO v5 L **(46.5 M params)** as teacher network
- YOLO v5 M **(21.2 M params)** as teacher network

- YOLO v5 N **(1.9 M params)** as student network

Using the same basic network architecture for both student and teacher has several advantages, the networks (barring the use of different resolutions) share the same pre- and post-processing steps for images, this implies that the manipulations required to make the two networks "talk" is minimal.

Instead of rewriting the training process from scratch, I created a **fork** from the official repository:`https://github.com/ultralytics/yolov5`
that you can find in my public repository: `https://github.com/CuriousDolphin/yolov5-knowledge-distillation` I have modified the train script by including the loading of a pre-trained teacher network from the yolo v5 family, which will perform the prediction that would affect the student training, specifically we will be working with two different kind of knowledge distillation:

- **Output-Based**:the student will only be influenced by the final outputs of the teacher model, in order To do this I have implemented in training process the loss functions described in this paper:[29], more info in relative chapter.

- **Feature-Imitation**: The student will also be affected by the behavior of the intermediate layers of the teacher network, needs "anchor points" between the intermediate layers of the teacher and student,Is an implementation of this paper: [30] , more info in relative chapter.

Pre-trained yolo family networks are trained with 300 epochs,since the train phase is very slow (about 10 days with the hardware that I have available) and since the training process is not a precise science but we needs to find the right hyperparameters with different iterations, I have done some testing on shorter trainings for example 80 epochs.

Wandb: Weights and Biases

In order to evaluate and track all training experiments I've used a tool called **Wandb: Weights and Biases** (`https://wandb.ai/site`), wandb integrates perfectly with pytorch, and through the web interface allows you to monitor and **track experiments**, **compare** different experiments (with a lot of metrics defined in train process) and **versioning** models and hyperparameters. Creates visual plots that allow us, for example, to compare the performance of the loss of two distinct training sessions, also stores in the cloud all the information that it tracks, allowing us to analyze it at any time.

4.3 Dataset: Microsoft COCO

Most of the tests in this thesis use images coming from the COCO dataset, COCO is the acronym for *Common Objects in Context* and it is a large-scale object detection and segmentation dataset widely used in computer vision challenges,it consists of a collection of images that include **80 different classes** of common objects.

It contains more than 200k images annotated for different tasks, **not only object detection**,including:

- Object Detection: bounding boxes and instance segmentation with **80 classes**
- Captioning: caption descriptions for each image
- Keypoints detections: about 200k images with **person istances labeled with keypoints**.
- DensePose: annotation, like points on a person instance, that map person with a 3d model.

The first version was launched in 2015 [31], then updated in 2017 based on community feedback. In the tests present in this thesis was used the 2017 release that contains the following splits:

- **Train**: 118k samples (18GB) this split has been used to train networks with knowledge distillation .
- **Val** : 5k samples (1GB) this split has been used to evaluate the different networks performance in term of accuracy and **mAP**, most data in the literature refer to this split.
- **Test**: 41k samples (6GB) The annotations of this split are not public, it's possible to upload the network results and run a benchmark in the coco servers,used for competitions, but not used in this work.

I also created a **mini-val split** (1000 samples) and **nano-val split** (500 samples), both are random subset of stock val split, in order to evaluate directly on the board the performance in terms of **speed and latency**. This because even just evaluating very heavy models on a hardware with reduced performance could take several hours, and as far as performance in terms of speed is concerned an average on a few samples is sufficient.

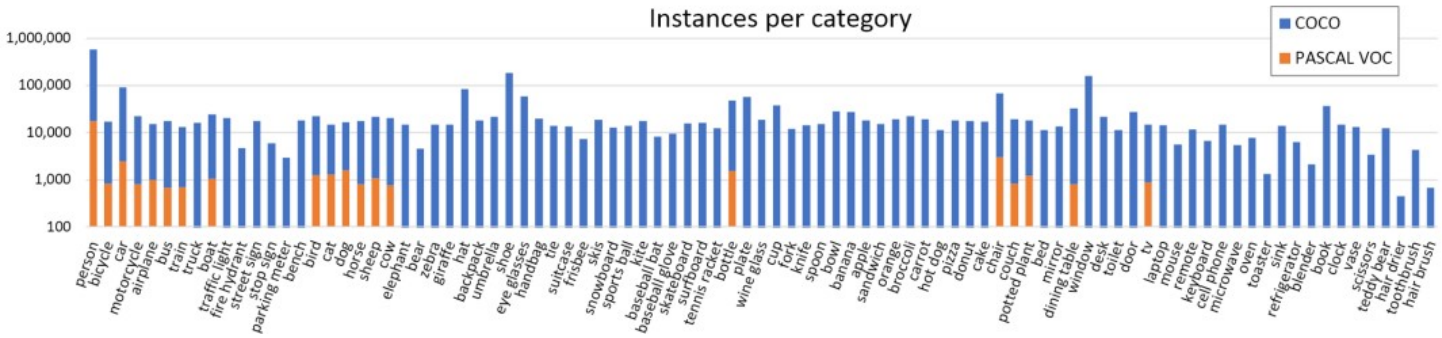


Figure 4.1: This plot shows the number of instances for each of the 80 classes, compared to the dataset VOC (Visual Object Classes) which is another important dataset in computer vision, we can see 80 classes for COCO and only 20 for VOC. The number of instances for the same class is on average 10 times higher for coco, this makes us understand that it is a relatively large dataset.

Chapter 5

Experiments and Results

5.1 Post Training Dynamic Quantization

In this section will be exposed the experiments concerning quantization, specifically **post-training dynamic quantization** on different types of object detection models, introduced in 2.2.
Latency, **model size** and **mAP** performance will be analyzed on the nxp imx8mp evaluation board 2.3.1, it will also analyze the behavior of the main engines optimized for Edge AI: OnnxRuntime and TensorflowLite, all metrics collected and the quantization process is done by the framework introduced in 3. For each quantization model, the baseline for comparison is the same model but not quantized, and both refer to the same engine.

Experiment Config

- **Quantization type**: post-training dynamic uint8 (both on activation and weights)
- **Device**: imx8mp (cpu) 2.3.1
- **Models**: [Yolov5n, Yolov5s, SSD-mobilenet, faster-RCNN]
- **Input Resolution**: [640px, 320px]
- **Runtime**: [OnnxRuntime, TensorFlow Lite]
- **Metrics**: average latency (ms), mAP .5:.95, model size (MB)
- **Dataset (mAP measures):** coco-val2017 (5000 samples)
- **Dataset (Latency measures)** : nano-val (500 samples subset from coco-val)

5.1.1 Yolo v5n

The nano version is the smallest model of the yolo v5 family, it has only 1.9 million parameters. It's a model already highly optimized on performance, and we expect a slight increase with quantization. This model was quantized with the two different engines, tensorflow lite and onnx, including a quantized onnx version with the input image size fixed to 320px (compared to 640px for the other models).
In figure 5.1 we see the average results obtained from running the models on the board, grouped by inference time, mAP and model size, next, the results are represented in terms of percent improvement from baseline.
We can see that in all experiments the weight of the model has decreased dramatically for both onnx and tflite models. The mAP metrics , as we expected, decreased in all experiments, with a negative peak in the network with 320px input. Also the inference speed increased in all experiments, in this field (with input at 640px) the quantized tensorflow lite model performed better, with a 30% decrease of inference time. The onnx model with input at 320px instead improved in speed by 4x, however with a notable loss in accuracy.

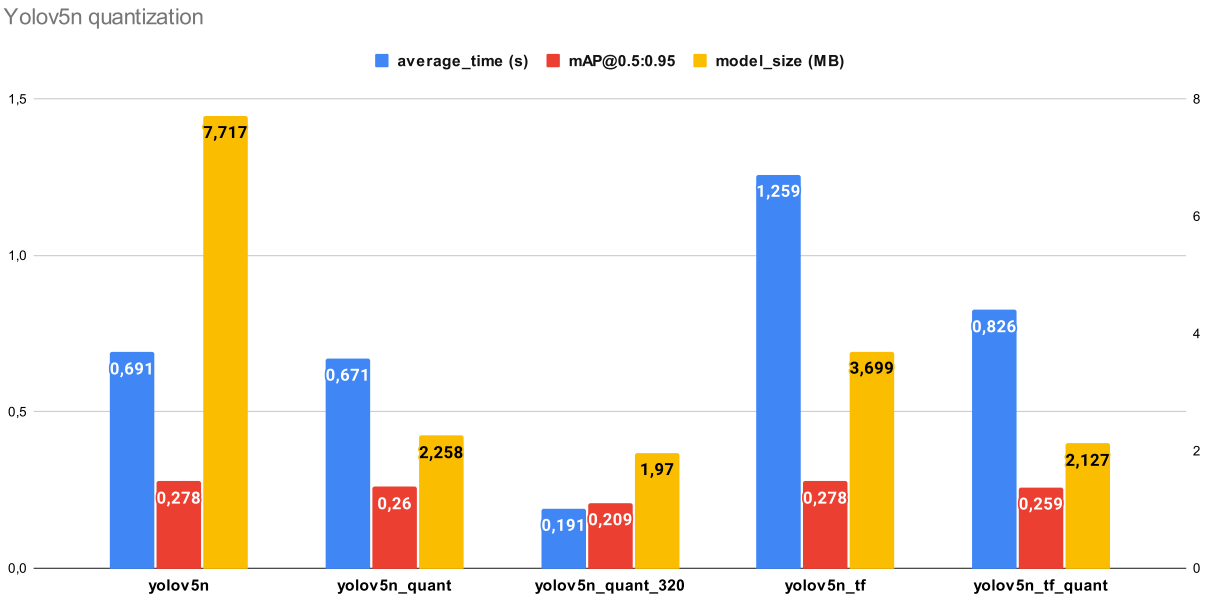


Figure 5.1: Yolo v5n dynamic metrics quantization comparison on imx8mp

Onnx Quantization

- **Latency**: -3% (20 ms faster)
- **mAP.5:95**: **-6%**
- **Size**: **-71%** (From 7.8MB to 2.26MB)

Onnx Quantization with 320px

- **latency**: **-72% (480ms faster)**
- **mAP.5:95**: **-25%**
- **Size**: **-74%** (From 7.8MB to **1.97MB**)

TensorFlow Lite Quantization:

- **latency**: **-34% (435ms faster)**
- **mAP.5:95**: -7%
- **Size**: **-42%** (From 3.7MB to **2.13MB**)

5.1.2 Yolo v5s

In this experiment we are going to quantize the small version of the family yolov5, it is larger, accurate and slower than the nano version and contains 8 million parameters, however this model is already highly optimized and we expect results in line with previous ones. As in the previous experiment, the model was quantized with the two different engines, tensorflow lite and onnx, including a quantized onnx version with the input image size fixed to 320px (compared to 640px for the other models).
In figure 5.2 we see the average results obtained from running the models on the board, grouped by inference time, mAP and model size, next, the results are represented in terms of percent improvement from the baseline.
We note, as in the previous experiment a strong reduction in weight for all models tested. Compared to the Nano version, the s-version quantized with onnx has achieved much better performance, since although the latency is reduced by almost 20%, the loss in accuracy is only 2%, all with a model whose size is less than half of the original. The quantized model with tensorflow lite has similar inference times as onnx, but significantly lower accuracy.

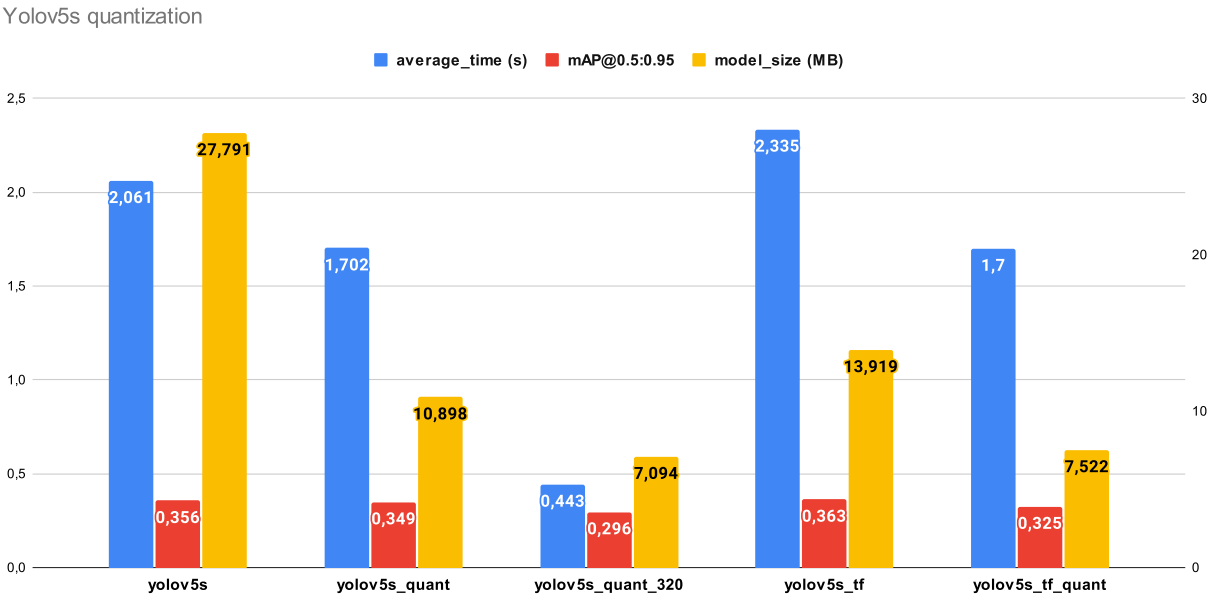


Figure 5.2: Yolo v5s dynamic quantization comparison

Onnx Quantization

- **Latency**: **-17% (300 ms faster)**
- **mAP.5:95**: **-2%**
- **Size**: **-61%** (From 27.8MB to 10.9MB)

Onnx Quantization with 320px input

- **Latency**: **-79% (1.6s faster! from 2s to 0.4s)**
- **mAP.5:95**: **-17%**
- **Size**: **-74%** (From 27.8MB to **7.1MB**)

TensorFlow Lite Quantization:

- **Latency**: **-34% (665ms faster)**
- **mAP.5:95**: -7%
- **Size**: **-42%** (From 14MB to 7.5MB)

5.1.3 SSD Mobilenet

The network ssd-mobilenet, as the name suggests is designed for use on low performance devices, is a single shot network like the previous ones and it has 4.2 million parameters. The model in this experiment has been quantized only with the onnx runtime, and has been tested with input at 640px and 340px.
In figure 5.3 we see the average results obtained from running the models on the board, grouped by inference time, mAP and model size, next, the results are represented in terms of percent improvement from the baseline.
We note that the model weight reduction is in line with past experiments, however compared to the yolo family the quantization with onnx has deteriorated a lot the performance in accuracy, almost by 1/3. The model was on average faster than 35ms per inference, the curious fact to note is that the model with input at 320px was on average slower than the model with input at 640px.

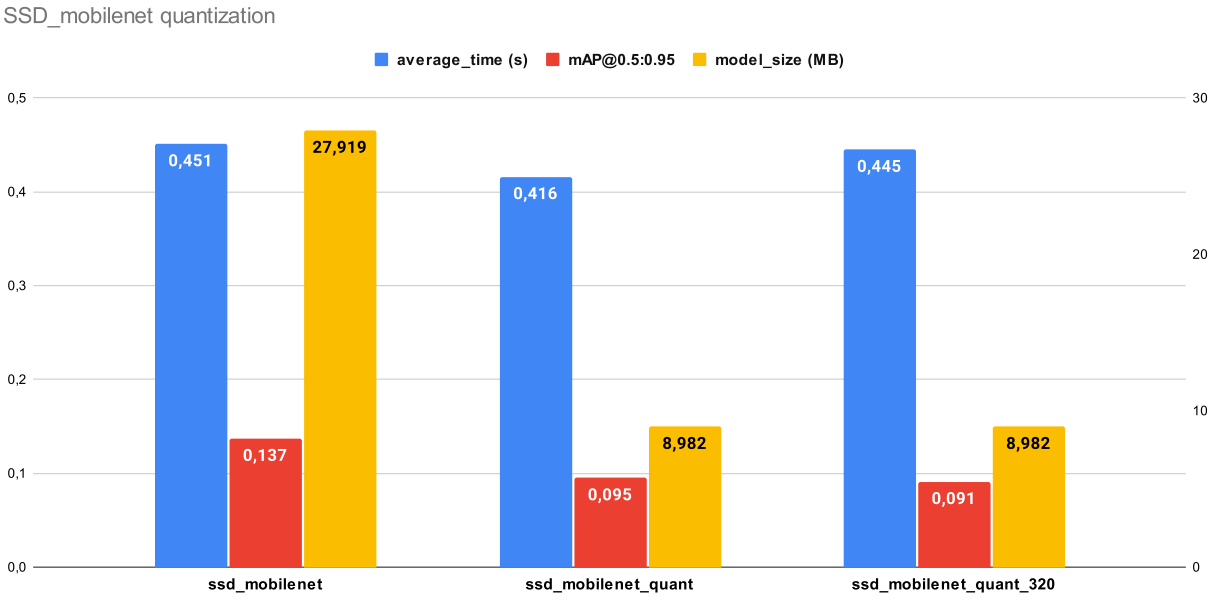


Figure 5.3: SSD Mobilenet onnx dynamic quantization comparison

Onnx Quantization

- **Latency**: -8% (35 ms faster)
- **mAP.5:95**: **-32% (very bad)**
- **Size**: **-68%** (From 27.9MB to 9MB)

Onnx Quantization with 320px input

- **Latency**: -1% (5ms faster)
- **mAP.5:95**: **-34% (very bad)**
- **Size**: **-68%** (From 27.8MB to **9MB**)

5.1.4 Faster-RCNN

FasterRCNN is the largest model analyzed, its size is about 160 MB (the size of previous models was less than 30MB). It is also the slowest, taking 20 seconds for a single inference on the test board, it is a multiphase model, less optimized than the previous ones, in fact the accuracy is not the best.
Although it is not a model designed for use on low performance devices, FasterRCNN can be a good benchmark to test how quantization affects such heavy and slow models. The model in this experiment has been quantized only with the onnx runtime, and has been tested with input at 640px and 340px.
In figure 5.4 we see the average results obtained from running the models on the test board, grouped by inference time, mAP and model size, next, the results are represented in terms of percent improvement from the baseline.
We immediately notice the incredible weight reduction, in fact it behaved better than all other experiments reaching, a weight reduction of 75%, reducing the weight of the model of 120MB, the inference time was reduced by 5 seconds with a loss in mAP .5:95 really negligible of 1%.
Input at 320 px further reduced the inference time by as much as 13 seconds, however, accuracy dropped dramatically by 42%.

Onnx Quantization

- **Latency**: **-25% (4.7s faster, from 19s to 14s)**
- **mAP.5:95**: **-1% (very very good)**
- **Size**: **-75% (From 160MB to 40MB)**

Onnx Quantization with 320px input

- **Latency**: **-69% (13s faster, from 19s to 6s)**
- **mAP.5:95**: **-42% (very bad)**
- **Size**: **-75% (From 160MB to 40MB)**

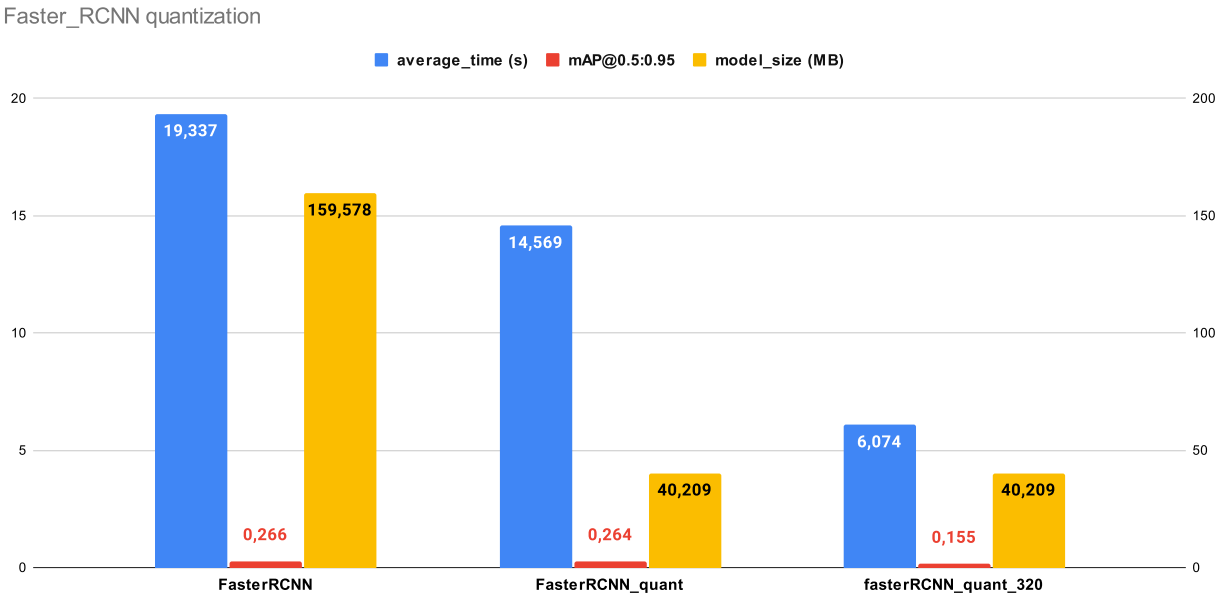


Figure 5.4: Faster-RCNN onnx dynamic quantization comparison

5.1.5 General results

The table 5.1 shows in detail the results of all the experiments performed, remember that the accuracy tests refer to coco-val 2017 and the latency test refer to 500 sub-samples. In the next page we find the bubble chart, which gives us a visual information of all the models analyzed in the metrics of interest: latency, accuracy, size. Inference times decreased in all tests, and mAP metrics decreased too (as expected), It is necessary to evaluate the cost/benefit ratio, which in some cases is really convenient as in the case of FasterRCNN and Yolo v5s where the benefits in terms of performance are much greater than what is lost in accuracy. This suggests that post-training quantization can give excellent results on both medium and large models. In all the experiments performed, the weight reduction of the models is significant, both with onnxruntime and tensorflow lite. Quantized Tensorflow Lite models has slightly lighter than the quantized onnx models, however yolo v5 n in tensorflow is both slower and less accurate than the onnx version. In addition, onnx proved to be more portable and practical than tensorflow lite, also in terms of hardware compatibility. If you need more performance you can consider half the input to 320px, however the decrease in accuracy can be significant, the best drop recorded is 17% in the case of yolo v5s, in all other cases it is worse.

Onnx quantization average results 640px and 320px

Average results over all tested models, mAP is heavy affected by the negative result of ssd-mobilenet (other models are under 6%.)

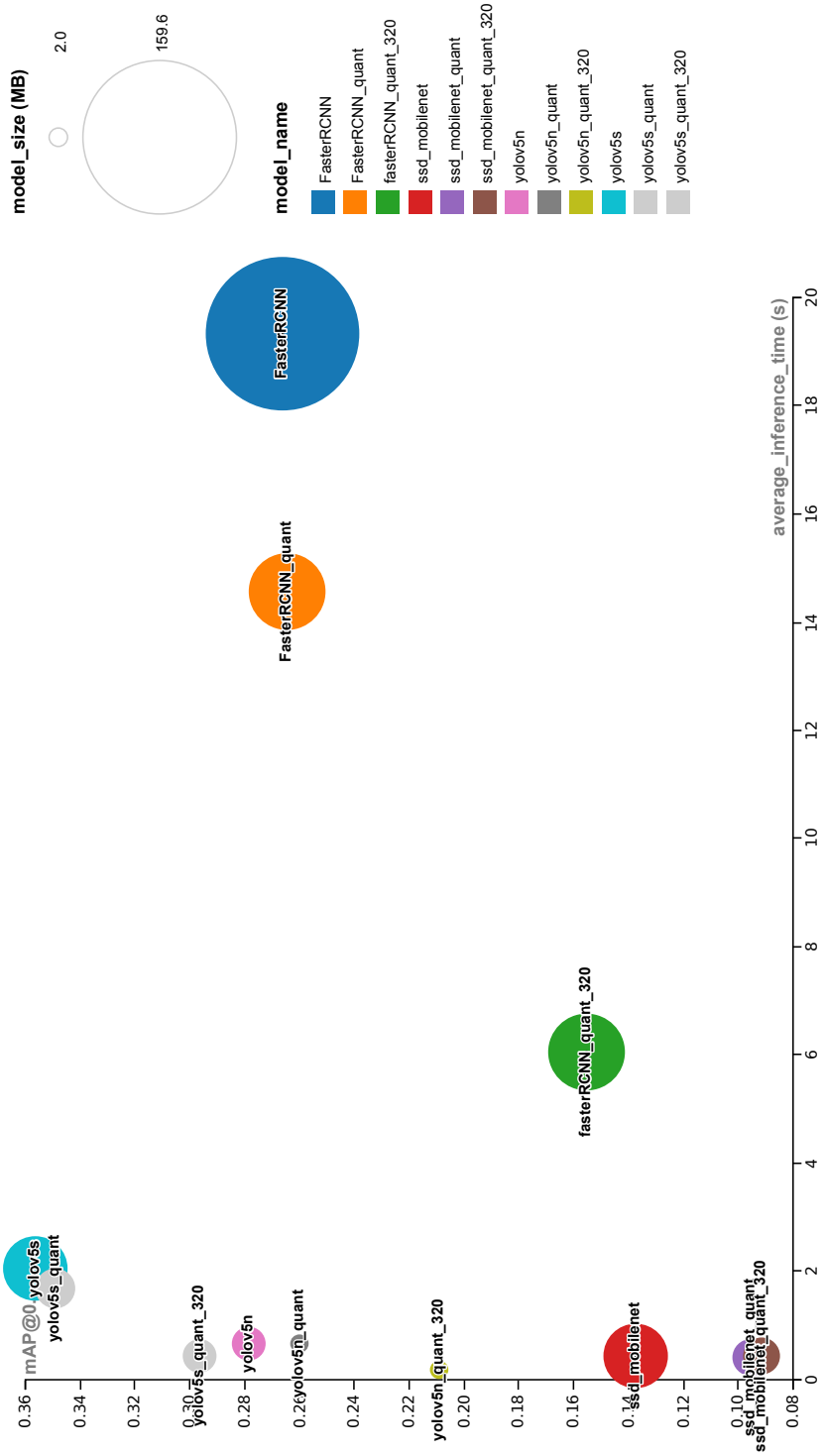
- **Latency:** -13% (-55% at 320px)
- **mAp:** -10% (-30% at 320px)
- **Size:** -69% (-73% at 320px)

TensorFlowLite Quantization average results

- **Latency:** -31%
- **mAp:** -9%
- **Size:** -44%

name	res	engine	fps	time	mAP.5:.95	mAP.5	size
yolov5n	640	onnx	1.45	0.691	0.278	0.454	7.72
yolov5n_quant	640	onnx	1.49	0.671	0.26	0.438	2.26
yolov5n_quant_320	320	onnx	5.24	0.191	0.209	0.35	1.97
yolov5s	640	onnx	0.48	2.061	0.356	0.539	27.79
yolov5s_quant	640	onnx	0.59	1.702	0.349	0.533	10.898
yolov5s_quant_320	320	onnx	2.26	0.443	0.296	0.462	7.094
ssd	640	onnx	2.22	0.451	0.137	0.216	27.919
ssd_quant	640	onnx	2.4	0.416	0.095	0.155	8.982
ssd_quant_320	320	onnx	2.25	0.445	0.091	0.149	8.982
faster	640	onnx	0.05	19.337	0.266	0.443	159.58
faster_quant	640	onnx	0.07	14.569	0.264	0.439	40.20
faster_quant_320	320	onnx	0.16	6.074	0.155	0.268	40.20
yolov5n	640	tflite	0.79	1.259	0.278	0.454	3.699
yolov5n_quant	640	tflite	1.21	0.826	0.259	0.431	2.127
yolov5s	640	tflite	0.43	2.335	0.363	0.547	13.92
yolov5s_quant	640	tflite	0.59	1.7	0.325	0.529	7.522

Table 5.1: General comparison on models with post training dynamic quantization executed on imx8mp, mAP refeer to coco-val2017, size is expressed in MB and latency in milliseconds


model_size (MB)
2.0
159.6
model_name
FasterRCNN
FasterRCNN_quant
fasterRCNN_quant_320
ssd_mobilenet
ssd_mobilenet_quant
ssd_mobilenet_quant_320
yolov5n
yolov5n_quant
yolov5n_quant_320
yolov5s
yolov5s_quant
yolov5s_quant_320
mAP@0.5
average_inference_time (s)
FasterRCNN
FasterRCNN_quant
fasterRCNN_quant_320
ssd_mobilenet
ssd_mobilenet_quant
ssd_mobilenet_quant_320
yolov5n
yolov5n_quant
yolov5n_quant_320
yolov5s
yolov5s_quant
yolov5s_quant_320

5.2 Knowledge Distillation

In this section we will expose all experiments regarding knowledge distillation, we will start by defining a student network as a baseline, and experiments with different combinations of knowlege distillation methodologies and teacher networks will be performed.

5.2.1 Student Baseline: YOLO v5n train 80 epochs

I have chosen Yolo v5n as the baseline student network, it is the fastest model of the YOLOv5 family and being able to improve the performance in terms of mAP of this model is the goal of the experiments in this section. The authors recommend a train of 300 epochs, however with the resources available it would have been unthinkable to run all the experiments on 300 epochs due to time constraints, so it was decided to use as a reference a training reduced to **80 epochs**.

Hyperparameters

The hyperparameters used to train the baseline yolov5 are the official ones released by the yolov5 authors: `https://github.com/ultralytics/yolov5/blob/master/data/hyps/hyp.scratch.yaml`

```
batch_size: 64
lr0: 0.01  # initial learning rate (SGD=1E-2, Adam=1E-3)
lrf: 0.1  # final OneCycleLR learning rate (lr0 * lrf)
momentum: 0.937  # SGD momentum/Adam beta1
weight_decay: 0.0005  # optimizer weight decay 5e-4
warmup_epochs: 3.0  # warmup epochs (fractions ok)
warmup_momentum: 0.8  # warmup initial momentum
warmup_bias_lr: 0.1  # warmup initial bias lr
box: 0.05  # box loss gain
cls: 0.5  # cls loss gain
cls_pw: 1.0  # cls BCELoss positive_weight
obj: 1.0  # obj loss gain (scale with pixels)
obj_pw: 1.0  # obj BCELoss positive_weight
iou_t: 0.20  # IoU training threshold
anchor_t: 4.0  # anchor-multiple threshold
fl_gamma: 0.0  # focal loss gamma (efficientDet default gamma=1.5)
# hyperparameters related to data augmentation are not displayed
# ...
```

Listing 5.1: Detail of Hyperparameters used to train baseline model, pretty stock from yolo v5n authors.

Baseline Results after 80 epochs:

mAP .5:.95	0.2459
mAP .5	0.414
Precision	0.573
Recall	0.375
train box loss	0.0528
train cls loss	0.0286
train obj loss	0.0724

Table 5.2: Yolo v5n baseline metrics after 80 training epochs

Authors' yolo v5n declared these results **after 300 epochs**: map.5:95: 0.284 map.5: 0.460 However, the hyperparameters used are not known.

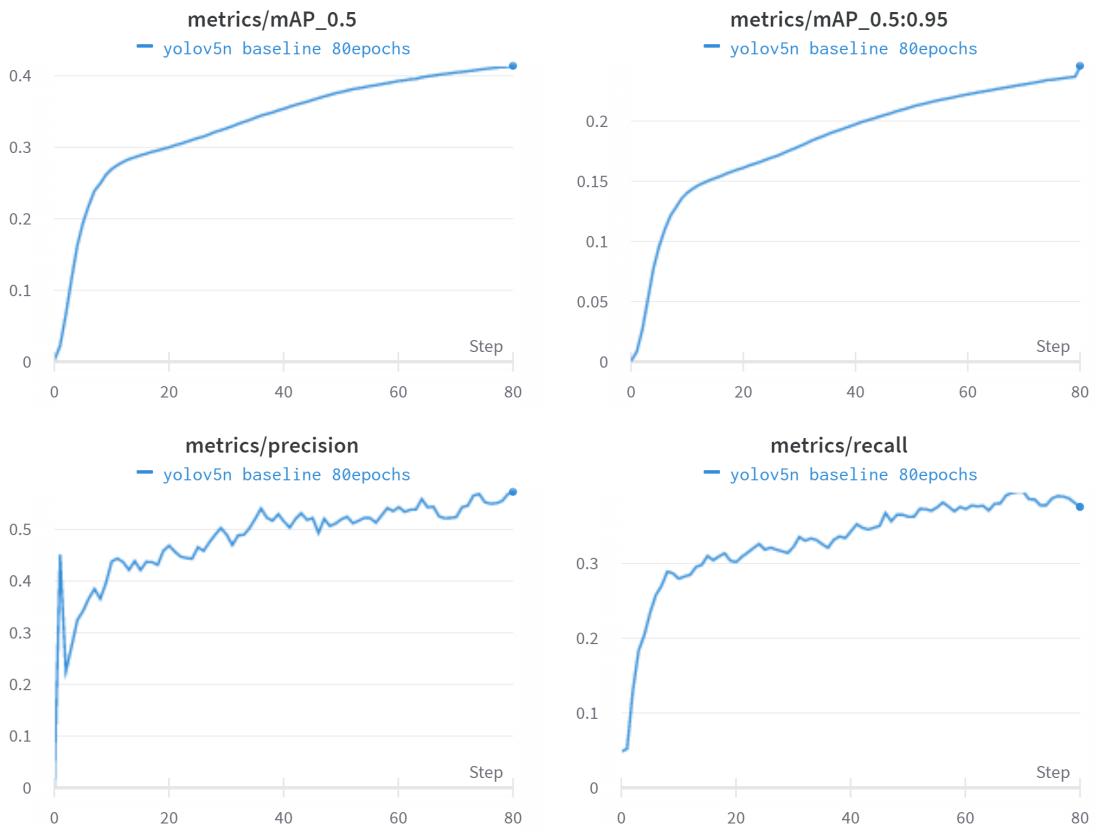


Figure 5.5: Yolo v5n baseline training analysis

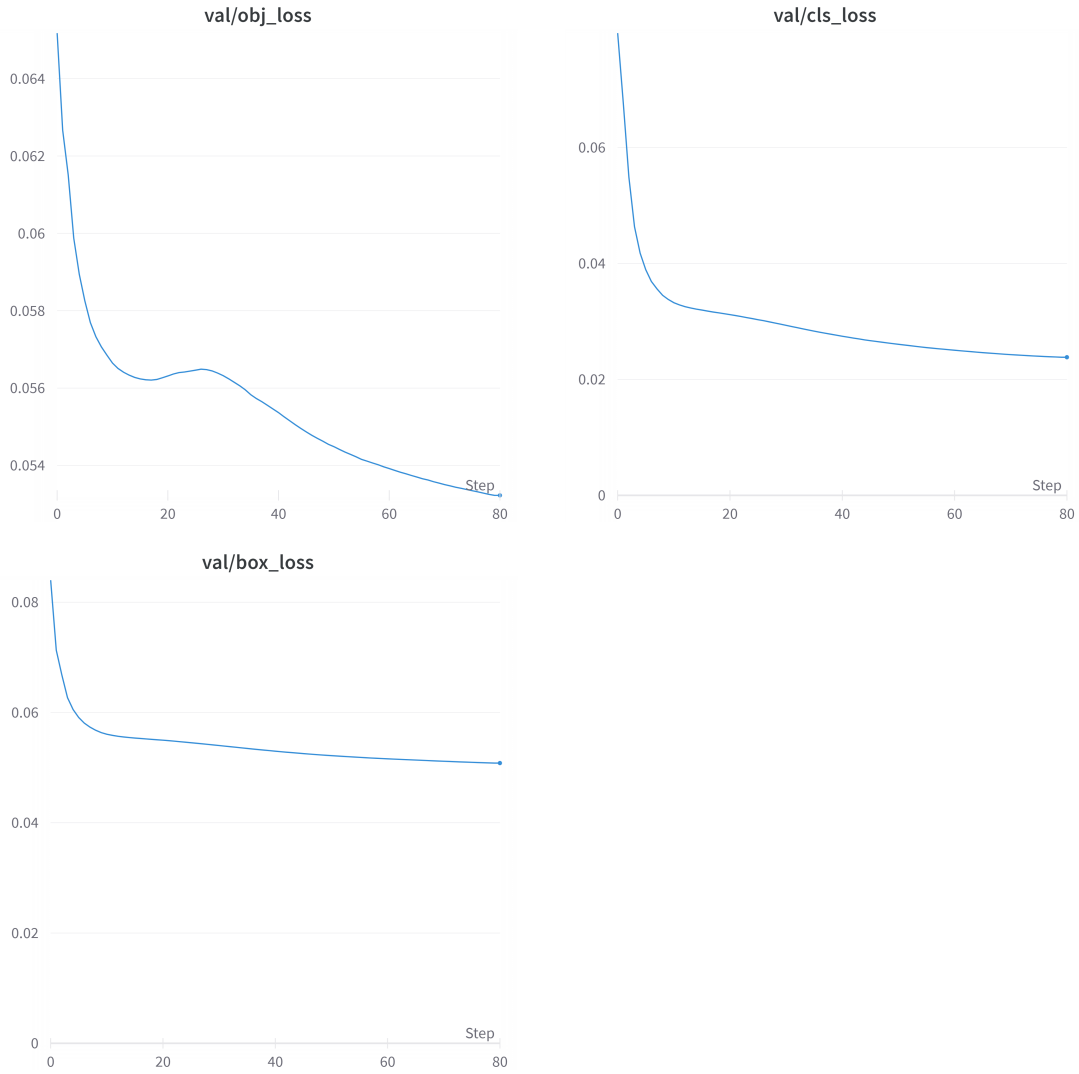


Figure 5.6: Yolo v5n baseline loss analysis

5.3 Output Based Knowledge Distillation (OBKD)

In this section will be exposed the experiments concerning output based knowledge distillation, they will be divided into two approaches, train the student network from scratch or starting from one already trained,also exploring transfer learning .

5.3.1 OBKD with Yolov5 M as Teacher

In this first experiment with output-based knowledge distillation I choose yolo v5m as a teacher network, two different trainings were performed to evaluate the impact of the distillation factor (the multiplier of the teacher's influence for the student's predictions),both trainings are performed on 80 epochs.

- **Student**: Yolo v5n
- **Teacher**: Yolo v5m (mAP.5:.95 = 0.452 mAP.5 = 0.639)
- **Distillation Factor**: [1,2]
- **Other Hyperparamters**: same as 5.2.1

Training Results:

Between the two distillation factors , the network that has demonstrated more performance in terms of mAP .5:.95, but also all the other metrics, has been the one trained with a **distillation factor of 1**, this means that 2 is a parameter too high and can be invasive for the training process of the student. However, the baseline results are slightly better, except for **recall**, where the **model with distillation factor 1 is almost 1% better than the baseline** ,that's an encouraging result.

Metric	OBKD dist2	OBKD dist1	Baseline
mAP 0.5:0.95	0.2396	**0.2447**	**0.2459**
mAP 0.5	0.3954	**0.4034**	**0.4141**
recall	0.3679	**0.3833**	0.3756
precision	0.5566	**0.5476**	**0.5729**
box loss	0.0549	**0.0541**	/
obj loss	0.0893	**0.0834**	/
cls loss	0.0339	**0.0320**	/

Table 5.3: OBKD comparative with different distillation factor after 80 epochs,we can see that the results are very similar with the baseline.

Training Analysis

In figure 5.7 we can see the mAP .5:.95 (evaluated on coco-val) trend of the different models in the 80 train epochs, In the first 20 epochs the baseline has a steeper course in comparison to the OBKD models, subsequently instead the course of the model with dist 1 seems even to overtake the baseline for the remaining epochs, even if of little, **this indicates that effectively the OBKD method can improve the performances of the model**. Instead the model with factor 2 seems to go on average worse than the others, this indicates that **2 is a too aggressive parameter** and therefore the teacher influences negatively on the training of the student.
In figure 5.8 in addition to the mAP.5, we can see the **loss analysis**, it is normal that the 3 different losses are always lower for the baseline, this is because in the models with OBKD the total loss is given by the loss calculated between the output of the student and the ground truth, to which is added the loss related to the influence of the teacher. As before, the network that seems to learn best is the one with distillation factor 1, the largest spread is found in the object loss, i.e the confidence related to the presence of objects, in this case the baseline model reaches very early the horizontal asymptote, which indicates that OBKD models learn more slowly to predict the presence of an object.

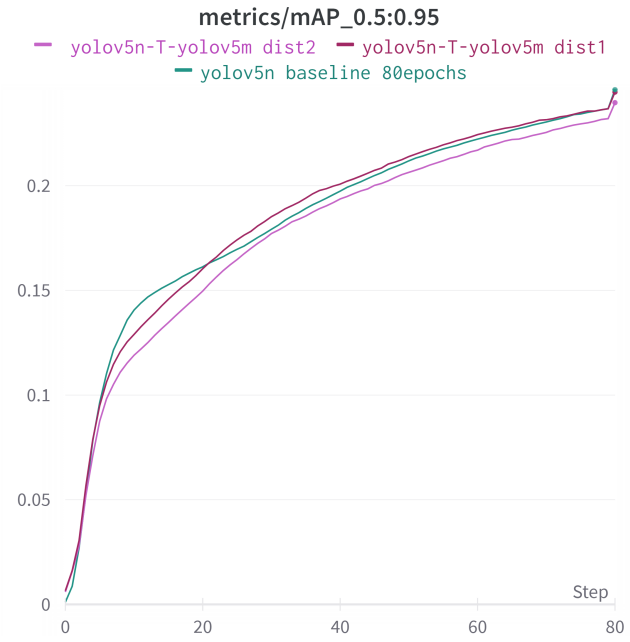


Figure 5.7: OBKD mAP .5:95 analysis with different distillation factors.

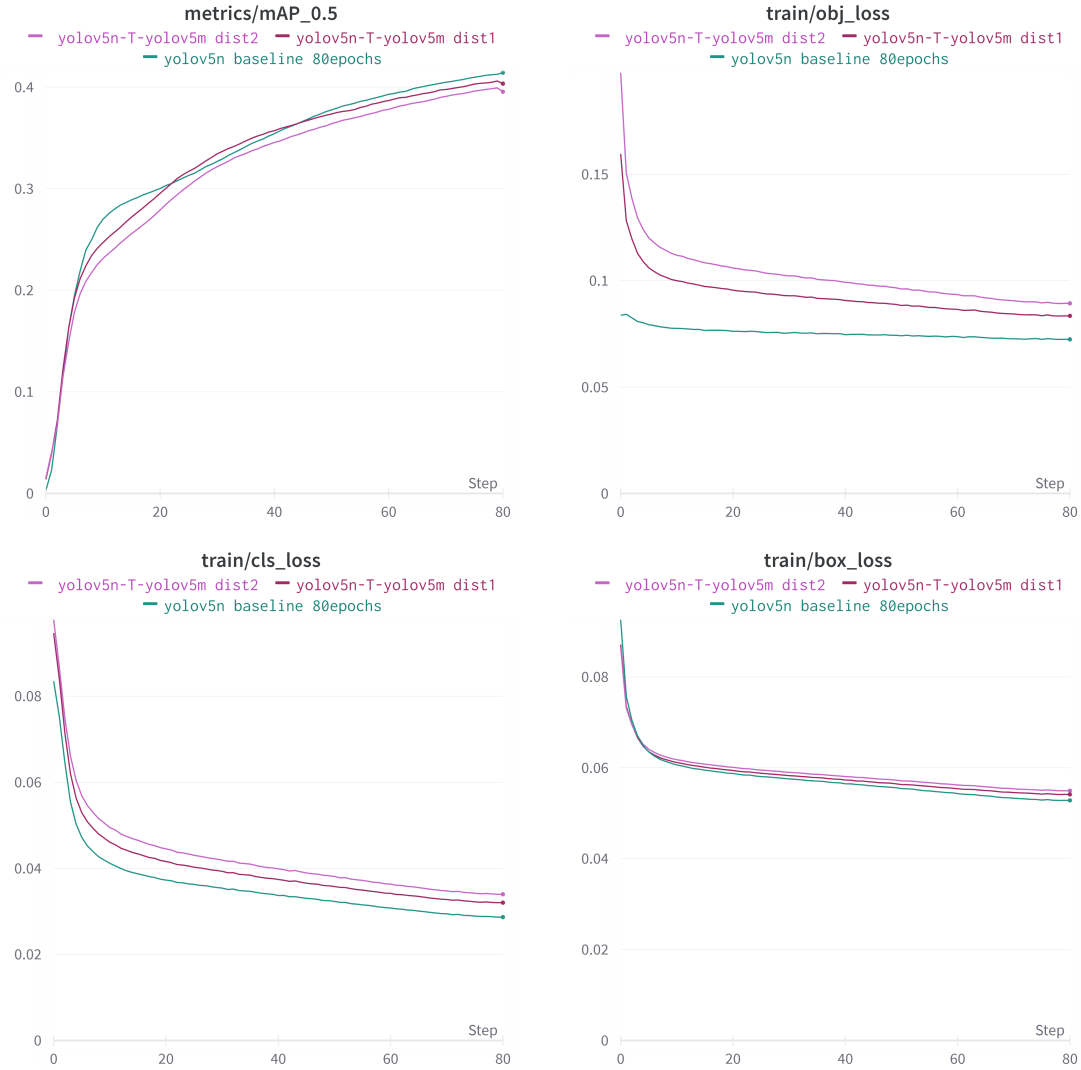


Figure 5.8: OBKD mAP.5 and loss analysis with different distillation factors, We can see that roughly all the losses have the same trend, except for the one concerning the confidence on object recognition, where the baseline network converges to the horizontal asymptote much faster than the others.

5.3.2 OBKD and Transfer Learning: Pre-trained Student with frozen backbone

In this experiment with output-based knowledge distillation I choose yolo v5l as a teacher network, but this time with a different approach. The idea is to apply **transfer learning to knowledge distillation**, i.e. starting from a pre-trained student, freezing the backbone and training only the last layers of the network with the help of the teacher's predictions,This process is normally used to fine-tune a network trained on a different dataset.
Since we start from a pre-trained student we need less epochs for the training, for this reason I decided to make different training with different **epochs** [**30-50-80**] and different distillation factors.

- **Student**: Yolo v5n pretrained with frozen backbone
- **Teacher**: Yolo v5L (mAP.5:.95 = 0.488 mAP.5 = 0.672)
- **Distillation factor**: [0.5 (80 epochs), 0.8 (50 epochs),1 (30 epochs)]
- **Other Hyperparamters**: same as 5.2.1

Training Results:

The results as we seen in 5.4 are worse than the previous experiment, even all the networks trained in this way are worse than the starting network, this means that transfer learning does not give us any improvement, **in fact the influence of the teacher deteriorate the performance of the base network**. We must also consider that the teacher is different from the previous experiment, and the greater separation between the student network and the teacher network can influence learning.

Metric	80e dist0.5	50e dist0.8	30e tl dist1	300e pre-trained*
mAP 0.5:0.95	**0.2723**	0.2674	0.2518	**0.284**
mAP 0.5	**0.4447**	0.4369	0.4332	**0.460**
recall	**0.4115**	0.4063	0.4014	/
precision	0.5791	0.5770	**0.5960**	/
box loss	**0.0513**	0.0516	0.0521	/
obj loss	**0.0802**	0.0843	0.0869	/
cls loss	**0.0263**	0.0269	0.0274	/

Table 5.4: OBKD and transfer learning comparative with different distillation factor and different epochs,*unknown hyperparameters .

Training Analysis

There is no baseline plot in the graph because the student's model is pre-trained and a comparison would not make sense with a train-from-scratch model. it is also difficult to make an analysis on models with different distillation factors because they are trained on different epochs and consequently different adaptations of learning rate but all networks have in common that they have **performed worse** than the starting network, however network that has performed better has been the one with distillation factor 1 and 80 epochs.
With the transfer learning should be enough much less epochs to converge to the result, this confirms that the contribution of the teacher in the learning of the student with the freeze backbone **affects negatively**.

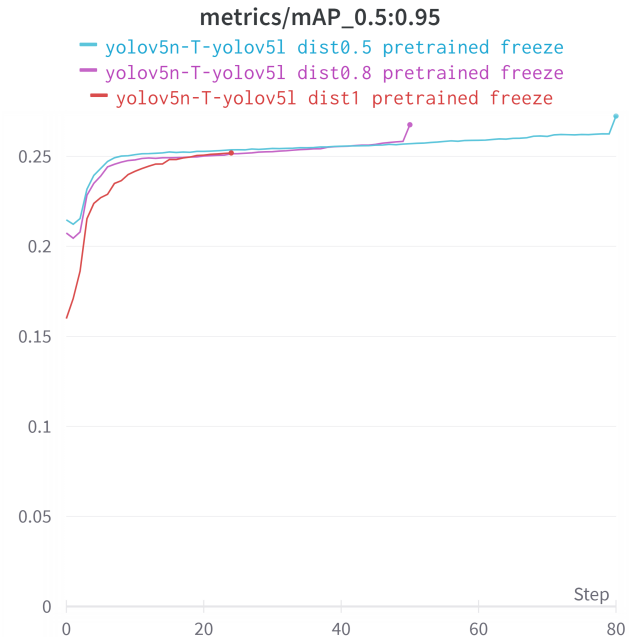


Figure 5.9: OBKD with transfer learning mAP .5:95 analysis with different distillation factors and different epochs.

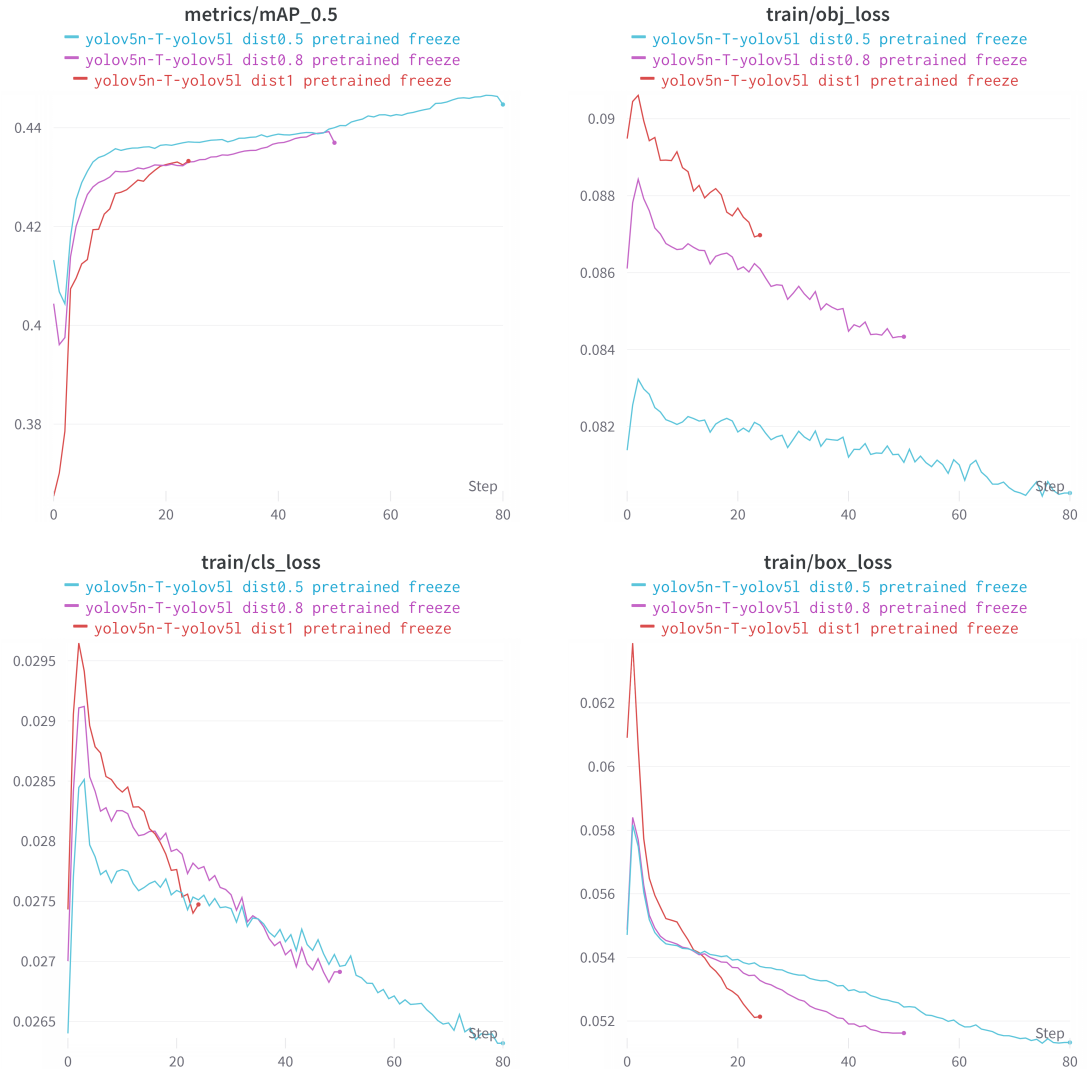


Figure 5.10: OBKD with transfer learning loss analysis, like as previous experiments we can see the major spread in obj loss,can be one of the most impactful parameters from OBKD.

5.3.3 OBKD and pre-trained student

In this experiment I want to compare one of the previous models (OBKD with transfer learning) with another OBKD training with a always pre-trained student, but without freeing his backbone, this should allow the student network to adapt better to the presence of the teacher and should lead to results in less time than the train from scratch.
The teacher is always Yolov5 L and both tests are evaluated over 80 epochs.

- **Student**: Yolo v5n pretrained
- **Teacher**: Yolo v5L (mAP.5:.95 = 0.488 mAP.5 = 0.672)
- **Distillation factor**: 0.5
- **Other Hyperparamters**: same as 5.2.1

Training Results:

They are not very good results if we compare them with the same model pre-trained but without OBKD, **both models have worse performance relative to mAP**. we can notice that after 80 epochs the model with transfer learning obtains better results than the only pre-trained network, this can indicate that probably starting from a pre-trained student is not the right approach for OBKD, evidently **the network needs to learn through the influence of the teacher directly from scratch and not to be "confused" by the pre-trained weights.**

Metric	OBKD-p	OBKD-tl	300e pre-trained*
mAP 0.5:0.95	0.2707	**0.27237**	**0.284**
mAP 0.5	0.4395	**0.4447**	**0.460**
recall	0.4101	**0.4115**	/
precision	0.5782	**0.5791**	/
box loss	0.0522	**0.0513**	/
obj loss	0.0814	**0.0802**	/
cls loss	0.0287	**0.0263**	/

Table 5.5: OBKD comparative with pretrained student and transfer learning, both with dist 0.5 after 80 epochs,we can see that the results are very away from the authors baseline.

Training Comparative: pre-trained student vs transfer learning

In the figures 5.11 and 5.12 we can see that both the trend of the mAP and the Loss of the OBKD network with only pre-trained student (the orange ones) is much steeper than the network with transfer learning, this actually indicates that **using OBKD the network learns better when also the weights inside the backbone are updated**. Probably with a few more epochs the orange network would have overtaken the blue, **however even if it could pass the baseline it is not a noteworthy result if a lot of epochs are needed to reach it**.

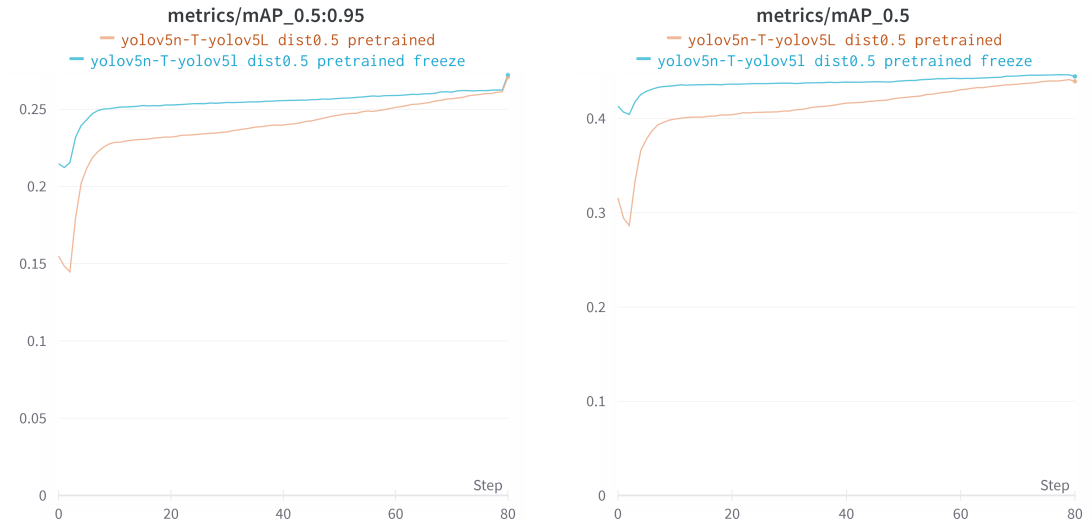


Figure 5.11: OBKD map comparison between pre-trained student and pre-trained student with frozen backbone

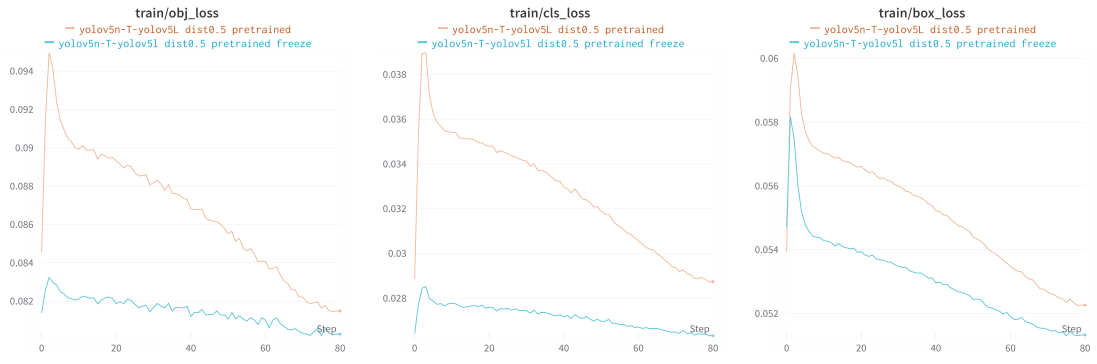


Figure 5.12: OBKD loss comparison between pre-trained student and pre-trained student with frozen backbone

5.4 Feature Imitation Knowledge Distillation (FIKD)

5.4.1 FIKD with Yolov5L and Yolov5L as Teachers

In this experiment we are going to analyze another method of knowledge distillation: **Feature Imitation**, it is **more invasive** than output based, the student is not influenced only by the output of the teacher but also by the behavior of intermediate layers, more information about FIKD in the appropriate chapter.
In theory with this method the student **should learn better** to "imitate" the teacher, specifically we will analyze the behavior of training from scratch with FIKD models with **different teachers** (Yolov5M and Yolov5L), we will also analyze the behavior with a pre-trained student network (no frozen backbone), but in this case, it would not make much sense to apply transfer learning and freeze backbone, because this method also exploits the intermediate layers of the student network.
All experiments are based on 80 training epochs and Yolov5N as a student.

- **Student**: Yolo v5n
- **Teacher**: [Yolo v5M,Yolo v5L]
- **Other Hyperparamters**: same as 5.2.1

Training Results:

From the results we can immediately see that the network that has performed better is the one with **yolov5L as teacher**, this indicates that actually the knowledge distillation methods perform better with a more robust and performing teacher. Even if just a very little, the network **FIKD-v5n-T-v5l exceeds the baseline** in the **mAP.5:95** metric that we consider most important to evaluate the performance of the model. **This result shows us that the FIKD method can improve the performance of a student network during his training.**

Metric	FIKD-v5m	FIKD-v5l	base 80e	FIKD-v5l pre	300e pre
mAP .5:.95	0.2422	**0.2465**	0.2459	0.2748	**0.284**
mAP 0.5	0.4021	0.4079	**0.4141**	0.4451	**0.460**
recall	0.3763	**0.3806**	0.3756	0.4076	/
precision	**0.5391**	0.5347	**0.5729**	0.5713	/
box loss	0.0530	**0.0529**	/	0.0514	/
obj loss	0.0726	**0.0725**	/	0.07116	/
cls loss	0.0289	**0.0287**	/	0.0265	/

Table 5.6: FIKD metrics and methods comparison after 80 training epochs

However exactly as in the previous experiments from 5.6 we notice that **the pre-trained approach does not work** and even worsens the performance of the basic network, this leads us to think that the presence of the teacher tends to "forget" what the student has already learned.

Training Comparative

The dotted line in the figures 5.13 and 5.14 represents FIKD with a pre-trained student, and it is interesting to note that although there is a lot of spread between the "virgin" networks and the pre-trained ones, the slope of the latter is steeper. This indicates that with knowledge distillation the models must start from the scratch in order to learn efficiently from the teacher, otherwise there is a risk of worsening the starting position.
Regarding instead the comparison with the baseline, practically in all the plots the three networks have a similar trend, between the two networks with FIKD the one that however perform a little bit better is the one with the greatest teacher: **yolov5L**,this is not an obvious behavior, as a teacher network who is too distant in term of performance from the student network may negatively affect them.

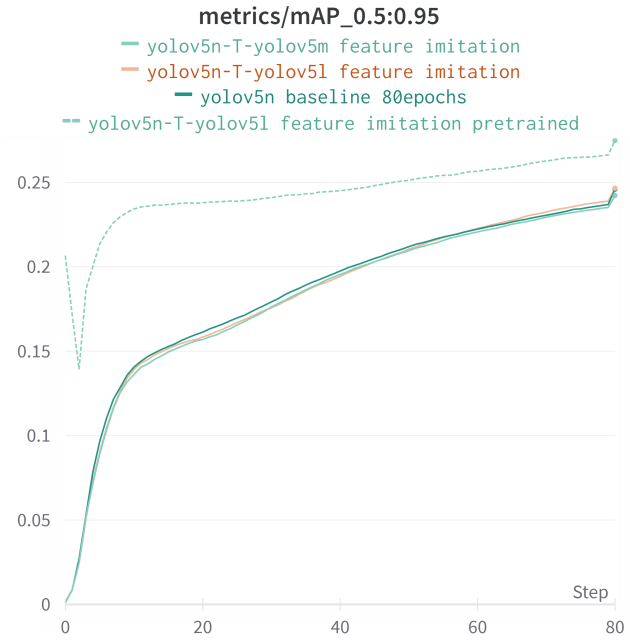


Figure 5.13: FIKD mAP .5:95 analysis with different teachers, we can see that the trend of the trained-from-scratch fikd networks is steeper than the pre-trailed one.

Similar to the OBKD experiment we can see in the figure 5.13 that the baseline trend is slightly above FIKD-v5l up to about 50-60 epochs, then **FIKD behaves slightly better**. This transition in the experiment on **obkd happens earlier**, close to 20 epochs as we can see in 5.7, this thing indicates that probably more time is needed for feature imitation than output based in order to influence student performance.

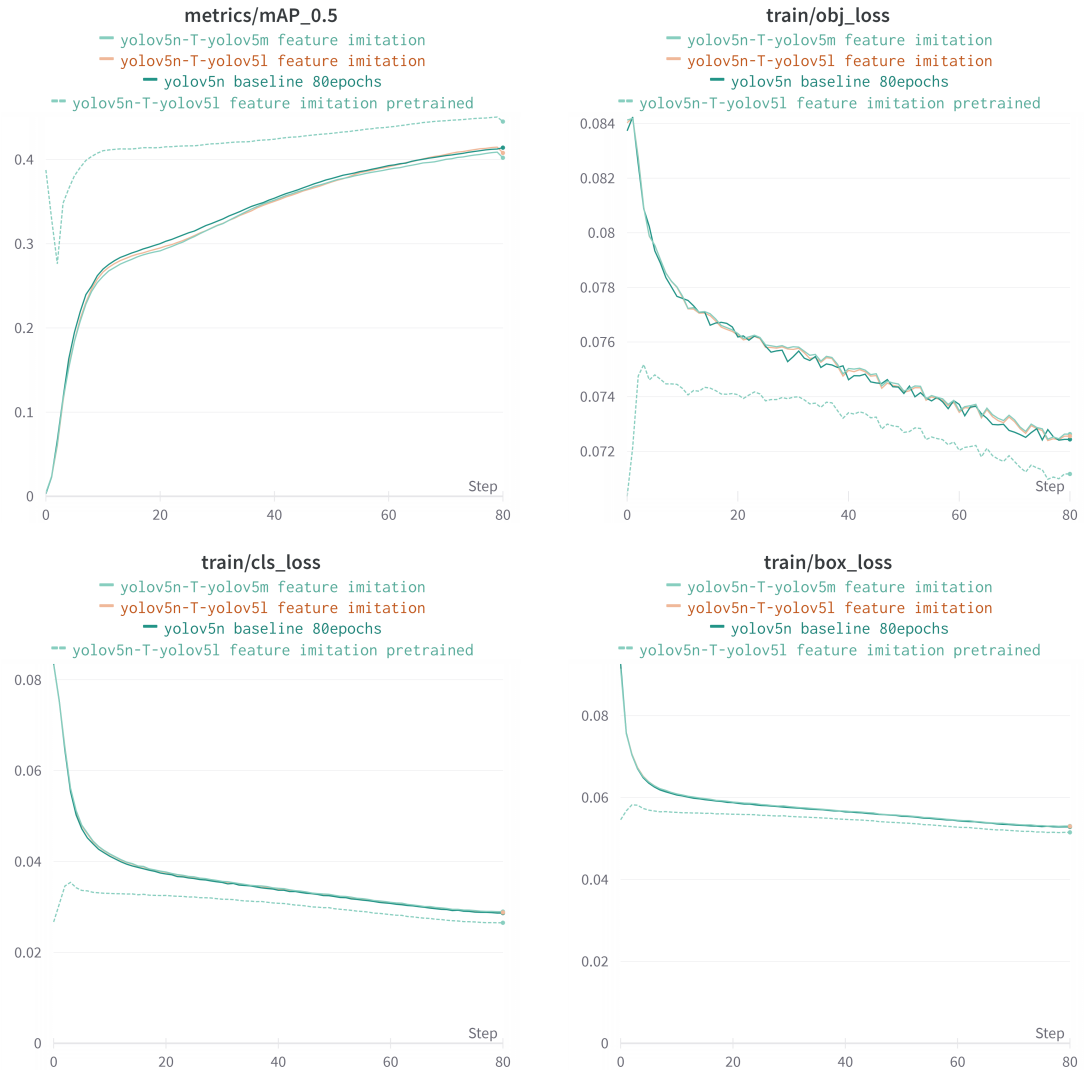


Figure 5.14: FIKD loss and map.5 analysis of diferent teacher networks.

5.5 Knowledge Distillation Methods Comparison

In this chapter we are going to make a final comparison with all the knowledge distillation methods previously analyzed, dividing it into two parts: train from scratch student and pre-trained approach

Train Student From Scratch

Between the two approaches, OBKD and the more advanced FIKD, there is not so much distance on the accuracy metrics, although the influence of the teacher should be more invasive in the case of the feature imitation, which anyway seems to be the method that has performed better, managing to **increase (very slightly)** the score compared to the baseline network of mAP .5:95 that I consider the best metric to evaluate an object detection algorithm in terms of accuracy.
It is interesting to note that the performance of the two different knowledge distillation techniques is very similar, but the performance of the same method (OBKD) with different distillation factors is very different, indicating that this is an important parameter to consider.
While both knowledge distillation approaches, have shown that it is possible to improve the training performance of a small network by the hint of a larger one, but it may take a lot of effort and time in order to find the right configuration of parameters and teacher networks to achieve tangible results with knowledge distillation.

Model	**mAP.5:.95**	**mAP.5**	**recall**	**precision**
yolov5n-T-yolov5m dist2 OBKD	0.2396	0.3954	0.3679	0.5566
yolov5n-T-yolov5m dist1 OBKD	0.2447	0.4034	**0.3833**	0.5476
yolov5n-T-yolov5m FIKD	0.2422	0.4021	0.3763	0.5391
yolov5n-T-yolov5l FIKD	**0.2465**	0.4079	0.3806	0.5347
yolov5n baseline 80epochs	0.2459	**0.4141**	0.3756	**0.5729**

Table 5.7: Comparison between different Knowledge Distillation methods, all networks are trained from scratch for 80 epochs

Pre-trained student

Both knowledge distillation methods, starting from a pre-trained student (even with the backbone frozen in the case of OBKD) perform worse than the same network without the re-train with knowledge distillation. Both knowledge distillation methods, starting from a pre-trained student (even with the backbone frozen in the case of OBKD) **perform worse than the same network** without the re-train

with knowledge distillation. The idea of starting from a pre-trained student derives from the fact that it is very likely to save time refining an already trailed model rather than re-training the model from scratch for all the epochs needed, so even if in twice the number of epochs used (80) the knowledge distillation approach would perform better it would not be a great result compared to the total time spent on training the network itself.We can therefore affirm that the techniques of knowledge distillation have little sense if the student is already a trailed net, therefore with a "knowledge" already defined, it could be a similarity with the fact that a man learns in a better way in the first phases of his life, where he has not yet learned to much.

Model	mAP.5:.95	mAP.5	recall	precision
yolov5n-T-yolov5l p FIKD	0.2748	0.4451	0.4076	0.5713
yolov5n-T-yolov5L dist0.5 p OBKD	0.2707	0.4395	0.4100	0.5781
yolov5n-T-yolov5l dist0.5 tl OBKD	0.2723	0.4447	0.4115	0.5791
yolov5n-T-yolov5l dist0.8 tl OBKD	0.2674	0.4369	0.4063	0.5770
yolov5n 300 epochs	**0.284**	**0.460**	/	/

Table 5.8: Pre-trained Knowledge Distillation comparison, "p" stands for pre-trained,"TL" stands from Transfer Learning.

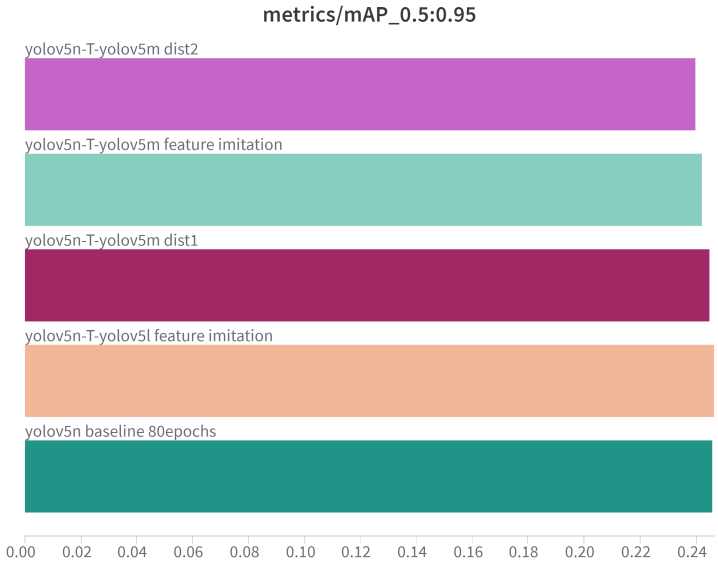


Figure 5.15: Comparison between different KD networks trained from scratch for 80 epochs, we can see that yolov5L work better as a Teacher respect to Yolov5M

Chapter 6

Conclusion and Future Work

In this thesis we analyzed the challenge of optimizing deep learning models for inference on edge devices, we took two different approaches: one is to optimize an existing trained model in terms of execution speed and also compress its weight with post-training quantization.
The second approach, instead, concerns to intervening during the training of a lightweight and fast model to improve its performance in terms of accuracy with knowledge distillation.
The target device of this thesis is a board based on multi-core arm cpu, with an embedded linux os and all tests were performed on its cpu.
The experiments with post training quantization, have given positive results, the performances have averagely improved in terms of execution speed on the board, the weight of all tested models has been reduced significantly, usually with an acceptable trade-off in terms of accuracy drop. In some cases it was possible to achieve an increase in fps of more than 20%, a reduction in model weight up to 75% and an accuracy trade-off (mAP.5:95) of 1-3%. To get more performance from the quantization, we can use lower resolution inputs, but in some cases the trade-off with accuracy has been high. We notice that higher is the model complexity, more efficient is the quantizazion on that model, resulting in better improvement rates than smaller models.
Results achieved, makes post-training quantization recommended for deployment of object detection models on embedded devices, **future tests could be a comparison with quantization-aware-training**.
We consider onnx and onnxruntime a good almost universal support for exporting and optimizing deep learning models, however, compatibility of quantized operators with the original ones is not always available. Regarding knowledge distillation, experiments were less successful than the quantization ones, however if only slightly, in some cases the performance in terms of mAP of the student network was improved compared to the baseline. Two different knowledge distillation methods have

been tested, output based (the simplest) and feature imitation, their performances alternate, however feature imitation seems more promising, training-from-scratch approach proved to be significantly better than the pre-trained student ones.
Knowledge distillation is a method for those seeking extreme optimization of the model training process, and with in-depth study and research of hyperparameters, it is possible for this technique to improve the student model performance in a more robust way. However, find the optimal configuration can take a lot of effort and time. Future Knowledge distillation tests can be:

- **Comparison of knowledge distillation with different datasets**: COCO (the dataset used in this thesis) is a very-large dataset, smallest or custom dataset can give different results.
- **Test different types of student-teacher combinations**: The experiments on kd were done with teacher and student of the same model family, it would be interesting to see the differences with different teacher and student architectures.
- **Combine Knowledge Distillation and Quantization**: The two optimizations are completely independent, combining them can bring to achieve model optimization at 360°.
- **Epochs Adjustment**: To be able to test different configurations, trainings have been performed on less epochs than those necessary to complete the process, however the ideal would be to test these methods on all the necessary epochs, but there is also to consider that the presence of the teacher in the training process makes it slower than the vanilla training.
- **Different Devices**: Our experiments were tested on a single device, it would be interesting to compare these methods on different types of same category devices (and with and without tensor accellerator), even smartphones, where the use of artificial intelligence is increasingly growing and close to the end user.

Bibliography

[1] Tom Mitchell. *Machine learning*. McGraw hill New York, 1997 (cit. on p. 1).

[2] Frank Rosenblatt. «The perceptron: a probabilistic model for information storage and organization in the brain.» In: *Psychological review* 65.6 (1958), p. 386 (cit. on p. 2).

[3] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. «Imagenet classification with deep convolutional neural networks». In: *Advances in neural information processing systems* 25 (2012), pp. 1097–1105 (cit. on p. 2).

[4] Abiresearch. «Global Shipments of TinyML Devices to Reach 2.5 Billion by 2030». In: (2020) (cit. on p. 3).

[5] Warren S McCulloch and Walter Pitts. «A logical calculus of the ideas immanent in nervous activity». In: *The bulletin of mathematical biophysics* 5.4 (1943), pp. 115–133 (cit. on p. 7).

[6] Bing Xu, Ruitong Huang, and Mu Li. «Revise saturated activation functions». In: *arXiv preprint arXiv:1602.05980* (2016) (cit. on p. 9).

[7] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. *Multilayer feedforward networks are universal approximators*. 1989 (cit. on p. 10).

[8] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. «Learning representations by back-propagating errors». In: *nature* 323.6088 (1986), pp. 533–536 (cit. on p. 11).

[9] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. «Dropout: a simple way to prevent neural networks from overfitting». In: *The journal of machine learning research* 15.1 (2014), pp. 1929–1958 (cit. on p. 19).

[10] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. «Rich feature hierarchies for accurate object detection and semantic segmentation». In: (2014), pp. 580–587 (cit. on pp. 20, 22).

[11] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C Berg. «Ssd: Single shot multibox detector». In: (2016), pp. 21–37 (cit. on pp. 21, 26).

[12] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. «You only look once: Unified, real-time object detection». In: (2016), pp. 779–788 (cit. on pp. 21, 27).

[13] Rafael Padilla, Sergio L Netto, and Eduardo AB da Silva. «A survey on performance metrics for object-detection algorithms». In: (2020), pp. 237–242 (cit. on p. 22).

[14] Pedro F Felzenszwalb and Daniel P Huttenlocher. «Efficient graph-based image segmentation». In: *International journal of computer vision* 59.2 (2004), pp. 167–181 (cit. on p. 22).

[15] Ross Girshick. *Fast R-CNN*. 2015. arXiv: `1504.08083 [cs.CV]` (cit. on pp. 24, 25).

[16] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. *Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks*. 2016. arXiv: `1506.01497 [cs.CV]` (cit. on p. 25).

[17] Yu Cheng, Duo Wang, Pan Zhou, and Tao Zhang. «A survey of model compression and acceleration for deep neural networks». In: *arXiv preprint arXiv:1710.09282* (2017) (cit. on p. 33).

[18] Tejalal Choudhary, Vipul Mishra, Anurag Goswami, and Jagannathan Sarangapani. «A comprehensive survey on model compression and acceleration». In: *Artificial Intelligence Review* 53.7 (2020), pp. 5113–5155 (cit. on p. 33).

[19] Amir Gholami, Sehoon Kim, Zhen Dong, Zhewei Yao, Michael W Mahoney, and Kurt Keutzer. «A survey of quantization methods for efficient neural network inference». In: *arXiv preprint arXiv:2103.13630* (2021) (cit. on pp. 33, 35–37).

[20] Jianping Gou, Baosheng Yu, Stephen J Maybank, and Dacheng Tao. «Knowledge distillation: A survey». In: *International Journal of Computer Vision* 129.6 (2021), pp. 1789–1819 (cit. on pp. 33, 38).

[21] Yu Cheng, Duo Wang, Pan Zhou, and Tao Zhang. «Model compression and acceleration for deep neural networks: The principles, progress, and challenges». In: *IEEE Signal Processing Magazine* 35.1 (2018), pp. 126–136 (cit. on p. 33).

[22] Jiaxiang Wu, Cong Leng, Yuhang Wang, Qinghao Hu, and Jian Cheng. «Quantized convolutional neural networks for mobile devices». In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 4820–4828 (cit. on p. 33).

[23] Geoffrey Hinton, Oriol Vinyals, Jeff Dean, et al. «Distilling the knowledge in a neural network». In: *arXiv preprint arXiv:1503.02531* 2.7 (2015) (cit. on pp. 33, 38–40).

[24] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. «Quantization and training of neural networks for efficient integer-arithmetic-only inference». In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2018, pp. 2704–2713 (cit. on p. 34).

[25] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. «Training deep neural networks with low precision multiplications». In: *arXiv preprint arXiv:1412.7024* (2014) (cit. on p. 36).

[26] Guobin Chen, Wongun Choi, Xiang Yu, Tony Han, and Manmohan Chandraker. «Learning efficient object detection models with knowledge distillation». In: *Advances in neural information processing systems* 30 (2017) (cit. on p. 38).

[27] Adriana Romero, Nicolas Ballas, Samira Ebrahimi Kahou, Antoine Chassang, Carlo Gatta, and Yoshua Bengio. «Fitnets: Hints for thin deep nets». In: *arXiv preprint arXiv:1412.6550* (2014) (cit. on p. 39).

[28] Refactoring Guru. «Strategy in Python». In: (2016) (cit. on p. 41).

[29] Rakesh Mehta and Cemalettin Ozturk. «Object detection at 200 frames per second». In: (2018), pp. 0–0 (cit. on p. 52).

[30] Tao Wang, Li Yuan, Xiaopeng Zhang, and Jiashi Feng. «Distilling object detectors with fine-grained feature imitation». In: (2019), pp. 4933–4942 (cit. on p. 52).

[31] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. «Microsoft coco: Common objects in context». In: (2014), pp. 740–755 (cit. on p. 53).