POLITECNICO DI TORINO

Corso di Laurea in Ingegneria Matematica

Tesi di Laurea Magistrale

People counting using detection networks and self calibrating cameras on edge computing

Relatore
prof. Francesco Vaccarino

Candidato
Simone Luetto

Corelatore
dott. Stiven Kulla

Dicembre 2019

Contents

1 Introduction

This thesis project consists in constructing a model able to detect and count people from digital images in an indoor context, specifically in a classroom, using a properly trained neural network and different computer vision techniques. In particular the scope is to develop a software such that it is utilized on an embedded device and such that it is as autonomous as possible. In this way the digital images can be processed by the embedded device without being sent to external computers and without the human assistance.

In the first chapter are presented the main theoretical arguments useful to understand the model, Computer vision and deep learning are introduced, and the convolutional neural network are explained in detail.

In the second chapter is faced the choice of the hardware, two development board build specifically for neural network inference are compared through a benchmark, the main aspects that are considered are also the dimension and price of the boards but mostly their inference speed, in fact that the software should be able to give almost real time statistics.

Subsequently is presented the object detection task in order to define the model used for people counting, are described the neural networks employed and is addressed the problem of counting from the detections.

After the model definition is described the training procedure, this includes description of the datasets used and the presentation of the training results.

When the counting model is developed with a satisfying accuracy and inference speed, in order to improve the model different computer vision techniques are explored aiming at improving the efficiency and the self-reliance of the network. This process is defined as auto-calibration of the camera and includes three different techniques:

- The vanishing point detection, is useful to have a better prediction on the expected dimension of the bounding boxes in different position of the room.

- The computation of the optical flow, it is the pattern of apparent motion, this information is used to exclude from the detection the zones of the camera field of view that cannot contain people.
- Matching of images from different cameras, this is needed when more than a camera must be used to point at an entire classroom, it is used to avoid the risk of double counting the same person in two images.

These computer vision techniques combined with the network trained and optimized ensure a resulting software that is accurate but also efficient on embedded devices and versatile for different contexts. The reason of this development is to avoid any privacy problem and to create an autonomous device that can be utilized without any software or data analysis knowledge, resulting in an accessible and realistic application.

2 Deep learning for computer vision

In this chapter will be presented a wide overview on computer vision and on deep learning applied to computer vision.

Most of the technical argument explained in this chapter are the fundamental building blocks to understand how a neural network application works and to understand more specific argument discussed in following chapters.

Definition 2.1. *Computer Vision* is an interdisciplinary science, which goal is to construct algorithms and techniques in order to obtain an high level knowledge about real physical objects and scenes based on sensed image [1].

This task concerns all the stages of the learning from images, not only the image analysis, for example are also part of computer vision:

- Acquisition of data and data encoding.
- Processing and representation of sensed images.
- Images analysis.
- Interpretation of the results to make decisions about real physical world.

Talking about images to refer to computer vision input data is not referred only to standard images but to a variety of different data.

For examples typical inputs are: single 2D images, video sequences, images from different cameras of the same scenes, multidimensional data of different sensors like medical scanners or thermal cameras.

It is a wide scientific discipline and can be divided in many different sub-domains, for example typical tasks addressed by computer vision are:

- Recognition, is the principal sub-domain and contains a lot of different problems like: classification, object detection, identification. But also, more specific tasks as facial recognition or pose estimation.
- Motion Analysis, applied on sequence of images to solve tasks of tracking (of vehicles or pedestrian usually) or through optical flow to find the zones characterised by movement between sequentially images.
- Scene reconstruction, which aims at computing a 3D model of the scene based on sensed images.
- Image restoration, a variety of techniques to remove the noise or to reconstruct deteriorated part of the images.
- Moreover a lot of other specific application are solved using computer vision algorithms, both as separated tasks and as preprocessing or post-processing in different models. For example, the edge detection or the matching of images.

The majority of the problems has been tackled constructing specific mathematical models for that task, this has given good results on a lot of problems, but for each new application a specialized model with hand-designed features should have been constructed.

In a lot of specific problems this approach is currently the best one, for example to detect the movement or the edges in an image a lot of efficient algorithm based on gradient analysis are state-of-the-art models.

However, some different problems with less specific instances like the classification or the object detection this approach is too expensive.

Let imagine facing image classification, algorithms based on the retrieval of hand-designed features were popular, but they have an important defect, the features have to be hand designed, when the problem change considering a different set of images and classes most of the work done is completely useless and not reusable.

In these cases the rise of neural network, but even before the evolution of machine learning, was a revolution for the computer vision.

A lot of specific problems like the matching, the edge detection or movement analysis are still faced without the use of neural networks but using refined and specifically designed algorithms. However more difficult and generic problems, like most of the ones concerning the recognition sub-domain, are nowadays solved exclusively using neural networks.

The neural networks in fact are now the state-of-the-art solution for: classification, object detection, facial recognition, segmentation, pose estimation and even more tasks.

Remark 2.1. The deep learning, or more generally the machine learning, are not a sub-domain of computer vision, neither the computer vision can be considered part of the machine learning.

Indeed, the machine learning has a lot of application fields that are not part of computer vision, like the speech recognition and the natural language processing; the same is valid also for computer vision that uses a lot of algorithms that are not based on any type of automated learning.

However, for some of the most fundamental problems in computer vision the best available solutions are based on the use of neural networks, for this reason these disciplines are indissolubly linked.

2.1 Generalities on Machine Learning

From the beginning of the computer science one of the dream has always been the possibility to create an artificial intelligence, with the rise of programmable computer this has given rise to an official scientific field called *artificial intelligence (AI)* in 1956.

At first the focus was on difficult problems for human mind but relatively easy to encode for a computer, for example the creation of programs able to play at strategic games like chess.

For example, develop a program that does optimal moves in tic-tac-toe is really easy:

1. If someone has a *threat* (that is, two in a row), take the remaining square. Otherwise,
2. if a move *forks* to create two *threats* at once, play that move. Otherwise,
3. take the centre square if it is free. Otherwise,
4. if your opponent has played in a corner, take the opposite corner. Otherwise,
5. take an empty corner if one exists. Otherwise,
6. take any empty square.

The real revolution of artificial intelligence has been when the focus moved to problems that could seem intuitive for humans, but are really difficult to describe analytically, like the speech recognition or the understanding of images. Any human can solve these problems without even think about them but is really hard to understand how we do that, we have simply *learned* to do that through our whole lifespan.

In fact facing these problems has led to the development of a completely new approach, based no more on analytic programming, and more on the attempt to exploit information from the experience, following this change of perspective was born the field defined *machine learning* by Arthur Samuel in 1959.

Definition 2.2. The *machine learning* is a subset of artificial intelligence that study algorithms and models that enable the computers to perform different tasks without specific instruction, relying on patterns and data instead to learn a solution.

Despite an early birth of neural networks, between the 60's and the end of the 80's the artificial intelligence had periods with difficulty in obtaining founding, this was due to the failure in solving real problems and in creating productive projects, this period has been called *AI winter*.

The neural networks have been left, because at that time they were to heavy computationally and there wasn't an effective way to train them, the backpropagation algorithm gained recognition only in 1986. In the 90's the target has moved from the theoretical idea of constructing an artificial intelligence to the effort in finding concrete solution to real problems, from that time until now programs that can learn from experience have proven to be a real effective research field, machine learning and deep learning definitely explodes and now they are some of the most active fields in computer science.

Machine learning can be classified in three categories according to the learning process utilized:

- *Supervised Learning*: The model in the learning phase receive both the input values and their corresponding true output values, the target of learning is to create a model that maps the input in the outputs, such that after training it can effectively compute a good estimate of the output for new input data.

  Most of the recognition problems are faced using supervised learning, like classification, object detection and face recognition, or the problems about speech recognition.

- *Unsupervised Learning*: The model receive non-labeled input data, there is not any target output, the aim of the learning process is to find a structure in the data or to extract useful information that can be used also on new data.

  The principal tasks tackled in unsupervised learning are the clustering and the association rules.

- *Reinforcement Learning*: In this paradigm the model interact with a dynamic environment, the software tries to reach a target and the learning takes place using a system of reward or penalties based on the quality of the partial result of the software that have to improve maximizing the reward.

  It is applied to a very wide variety of problems because it is a general approach, some classical examples are tasks like learning to drive autonomously or to play against an opponent in strategic games.

The main components of each machine learning model are:

- The task T: it represent the problem that has to be solved.
- The performance measure P: it measures the quality of the solution of our algorithm, usually is defined by a *loss function* that measures the error made on a set of data.
- The experience E: in supervised learning is represented by each couple of data with the corresponding label (x, y).

Definition 2.3 (citation by Tom M. Mitchell)**.** A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E.

In our case the focus will be on supervised learning, that is based on learn to solve a problem from a set of n training examples $\{(x_1, y_1), ..., (x_n, y_n)\}$ where x_i is the feature vector and y_i is the corresponding label. The learning process is focused on seeking a function $f : X \to Y$, where X is the feature space and Y is the label space, such that it assign the right label to the data. The error of the predictive function on a labeled datum (x, y) is given by a loss function $\mathcal{L}(x, y, f(x))$.

The aim of the training process is to minimize the loss on some test data $\{(x_{n+1}, y_{n+1}), ..., (x_m, y_m)\}$ whose label is unknown and for this reason the loss cannot directly be computed. For this reason is defined the *risk* of the function $f(\cdot)$ as the expected loss on the feature and label spaces:

$$\begin{aligned} Risk: \quad R_{\mathcal{L}}(f) &= \int_{X,Y} \mathcal{L}(x, y, f(x)) dP(x, y) = \\ &= \mathbb{E}[\mathcal{L}(X, Y, f(x))]. \end{aligned} \tag{2.1}$$

The objective becomes minimizing the risk, because the empirical loss on the test data will converge to the risk. For this reason, is defined the best achievable risk:

Definition 2.4. The *Bayes Risk* is defined as the best possible Risk considering any measurable predictive function, for a specific loss function.

$$Bayes\,risk: \quad R^*_{\mathcal{L}} = \inf_{f:X\to Y} R_{\mathcal{L}}(f) \tag{2.2}$$

The goal of learning is to build a function g whose risk is as close as possible to Bayes risk. However also the probability distribution of feature space and label space are usually unknown, in order to estimate the risk is used its empirical counterpart computed using the known training labeled data:

$$Empirical\,risk: \quad \hat{R}_n(f) = \frac{1}{n}\sum_{i=1}^{n} \mathcal{L}(x_i, y_i, f(x_i)) \tag{2.3}$$

Minimizing the empirical on any possible function is a terrible idea and cause extreme overfitting, for this reason the model consists in choosing a small subset of functions $\mathcal{F}$ and minimize only over that functions, this doesn't guarantee the possibility to reach the Bayes risk and for this reason the model error is splitted into two components:

1. The *structural risk* or model risk: is the difference between the best risk reachable using a function belonging to $\mathcal{F}$ and the Bayes risk:

$$Structural\,risk: \quad \inf_{f\in\mathcal{F}} R_{\mathcal{L}}(f) - R^*_{\mathcal{L}} \geq 0. \tag{2.4}$$

2. The error due to the optimization of the function f in $\mathcal{F}$, this is the error that the training aims to reduce.

$$R_{\mathcal{L}}(\hat{f}) - \inf_{f\in\mathcal{F}} R_{\mathcal{L}}(f) \geq 0. \tag{2.5}$$

When building a machine learning model these considerations on learning theory should always be taken into account, the choose of the model aims at reducing the structural risk while the training of the model aims at optimizing the predictive function in the given subset.

A lot of different models have been developed in machine learning, a machine learning system for example can be developed using: decision trees, support vector machines, bayesian network, genetic algorithms and *artificial neural networks*. The machine learning algorithms that use deep artificial neural networks are considered as a separated sub-field called *deep learning*, it will be presented in details in the next sections because it is the cornerstone of the software developed.

2.2 Basic Deep Learning

Artificial neural networks (ANN or NN) are computing models composed by a sequence of layers, each consisting in some computational units called neurons, the structure is vaguely inspired by the idea of replicating in a simpler way a biological neural network.

The rise of models based on artificial neural networks more and more complex have given birth to the *deep learning*, that nowadays is the subfield of machine learning most active both in industrial application and in research.

Definition 2.5. The *deep learning* is a class of machine learning algorithms, based on artificial neural networks, that using multiple layers aim at extracting features of progressively higher level from input data.

As we have already said the neural networks are a relatively *old* idea, the recent rise of deep learning is due principally to the overcoming of two important limitations:

- To have a proper training the neural networks need a huge availability of input data, only in the last 15 years this has become possible with the creation of big datasets of labeled data.
- The neural networks are also computationally heavy, only when the GPUs became available for computing was made possible the extensive use of deep neural networks. Modern neural networks can have easily more than 5 million of parameters, they are very effective but only in the last years the necessary computational power has been available to use such deep networks.

The fundamental idea of deep learning is to use sequence of layers in order to extract features from raw data, then using these features to solve the assigned task, that could be almost any type of recognition problem. Recalling the learning theory presented in the previous chapter the utility of having deep networks that depends on millions of parameter is that the set of predictive function that this model can build is huge, in this way the *structural error* is way smaller with respect to simple approach that use linear functions and even with respect to methods that use kernels. As already mentioned, one of the main problems of considering big function spaces is that is easy to find overfitting problems, for this reason the neural networks implement a lot of regularization techniques that will be discussed in detail later.

To understand the idea that gives life to deep learning is better to start describing the most simple neural network, that is the *multilayer perceptron (MLP)*, or equivalently *deep feedforward networks*.

The multilayer perceptron consists in three or more layers, an input and an output layer plus at least one hidden layer, each layer is composed by multiple neurons, a simple example of minimum feedforward network is shown in image 2.1.

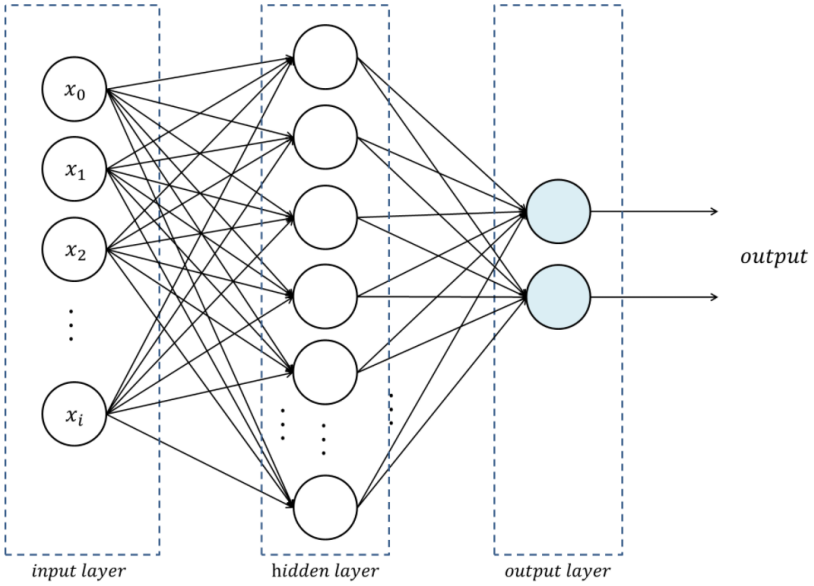


Figure 2.1: Example of a multilayer perceptron with only one hidden layer (from [2])

Definition 2.6. An *artificial neuron* is the elementary unit of an artificial neural network, each layer is composed by several neurons, it receives one or more input and combines them to produce an output.

As suggested from the name the elementary unit of an MLP is the perceptron, it is a simple algorithm that receive multiple inputs and combines them to produce a single output. The algorithm does a weighted sum of the inputs adding also a bias term, the bias and the weights represent the parameters of the algorithm, and then it applies a non-linear function to the result to produce the output. In image 2.2 can be seen an example of the working of each neuron.

The nonlinear function can be of different types but usually in the perceptron is used the sigmoid function described by the equation:

$$g(x) = \sigma(x) = \frac{1}{1 + e^{-x}}. \tag{2.6}$$

To better understand the computation performed by each perceptron let denote for each neuron:

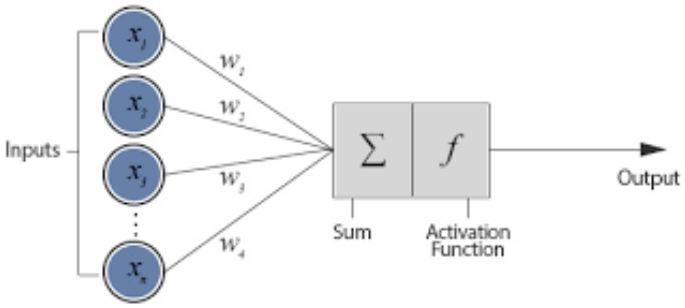


Figure 2.2: Example of a single perceptron, used as neuron in MLP (from [2])

- x_i the inputs.
- w_i the weights
- b the bias.
- y the output.
- $g(\cdot)$ the sigmoid function.

The functioning of a single neuron can be resumed by:

$$y = g\left(\sum_i w_i * x_i + b\right) \tag{2.7}$$

These computations are performed in each neuron of the deep feedforward network and each one is linked to the neurons of the previous and following layer; the output of a node works as input for the nodes in the next layer. In the MLP all the layers are *fully connected layers*, this means that each input of the layer is connected to each neuron of the previous layer, while the output of each neuron is used as input in all the neurons of the following layer.

The resulting structure is a composition of successive simple computations; however, the result depends on a lot of parameters, in fact for each neuron there are $n + 1$ parameters where n is the number of inputs, in particular n weights and one bias. In a very small network with only one hidden layer composed by 10 nodes that receive 5 inputs and gives a single value as output the number of parameter is 6 for each neuron in the hidden layer and 11 for the output node, so the network depends on 71 parameters that is already a quite high number. Considering that modern networks have easily more than 30 layers composed by hundreds of neurons is easy to understand why the modern neural networks are architectures with millions of parameters.

The utility of using multiple layers is that the model during the learning creates a hierarchical structure of features, the first layers are utilized to detect low level features, usually these are really general and less dependent on the single problem, going deeper the layers start to learn to detect higher level features, more specific for the type of input. Finally, the last layers use the last features, usually quite specific, to extract the output of the neural network

The easiest way to visualize this procedure is considering a convolutional neural network, that is the standard network class used in image related tasks. In the following section they will be described in detail because is the class of networks used in object and people detection, but for now we will only see a typical result to understand the functioning of the hierarchical layer structure.

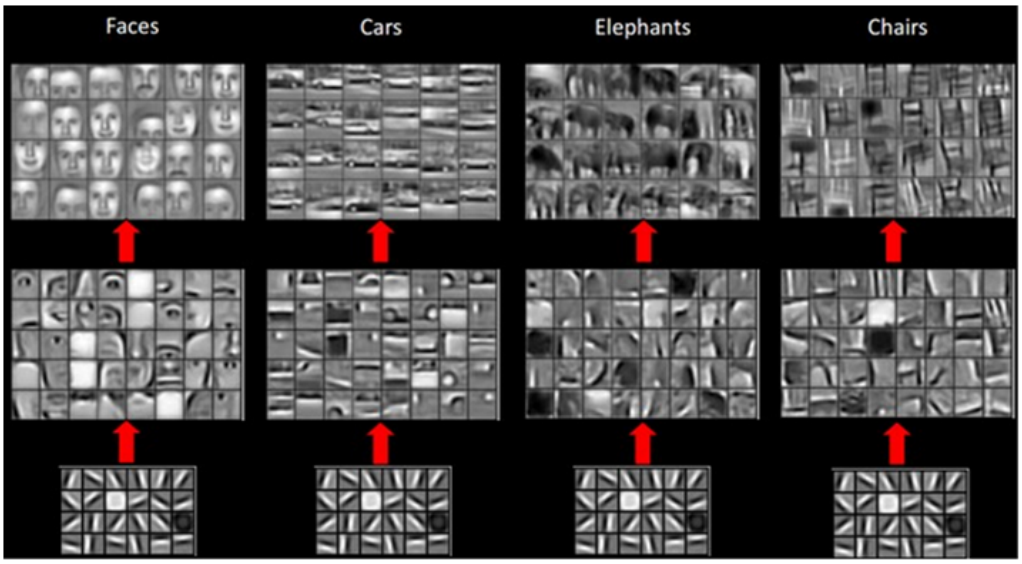


Figure 2.3: Example of hierarchical feature learning in a classification network (from [3])

As can be seen in image 2.3, the features recognised in the first layer are really generic, as a matter of fact they are almost identical between different type of input images, because the recognition of any image should start by detecting edges or other simple shapes.

In the following layer the learned features start to depend on the input images used, they are constructed on the base of the previous features and are clearly of a higher level. For example, in case of faces as inputs the visible features are mostly about section of the face like nose, eyes or mouth, or in the car case between the features are clearly visible some wheels. In the end in the terminals layers the features became specific is in fact possible

to see complete faces or cars.

This hierarchical structure is really effective in solving recognition tasks, but it also has some other advantages. In fact, having a structure that starts with very generic features made possible to reuse some of the trained layers to solve similar problems.

One of the peculiarity of the neural networks is exactly the possibility to perform the so called *transfer learning*, namely starting with a neural network trained to solve a problem similar or more generic than the target task, it can be utilized retraining only a determined fraction of the layer, maintaining the preceding layers that are trained to find low level features useful in both the problems. This practice is really useful because it widely reduces both the number of data needed for learning and the necessary computational power. The fraction of layers that needs to be retrained depends on a lot of factor, principally it depends on the similarity between the two tasks and on the dimension of the available dataset.

Transfer learning is not a prerogative of neural networks, but is probably it's best application field, a more general definition of transfer learning is the following:

Definition 2.7. Let $\mathcal{D}$ be the domain of the problem, it consists of: a feature space $\mathcal{X}$ and a probability distribution on it $P(\mathcal{X})$, and let $\mathcal{T}$ be the task, composed of: a label space $\mathcal{Y}$ and an objective prediction function $f(\cdot)$.

Given a source domain $\mathcal{X}_s$, a source task $\mathcal{T}_s$, a target domain $\mathcal{X}_t$ and a target task $\mathcal{T}_t$; the *transfer learning* aims at improving the learning of the target predictive function $f_t(\cdot)$ in $\mathcal{D}_t$ using the knowledge contained in $\mathcal{X}_s$ e $\mathcal{T}_s$.

These qualities of neural networks shall ensure their versatility, for this reason nowadays they are the most used technique in a lot of practical problem in the recognition field. However, they are definitely not free from defects, one of the principal critics about neural networks is their nature of *black-box* models.

Definition 2.8. A model is called *black-box* if it can be observed only in terms of its input and outputs, without any additional knowledge of its internal working.

Obviously, a neural network is not a completely black-box model, the internal structure is known and some information about the features extracted in the middle layer can be obtained. However even if we can see what the internal layers are computing there are too much link and parameters in the layers to really understand why they are working in that way, as

opposed to other machine learning algorithms like support vector machines that gives a separator and a margin in the input space.

The problem with neural networks in fact is to give an answer to the question *why is the neural network giving this result?*. In problems that are easy for humans like the image classification this aspect can be neglected, in fact the fairness of the output can be checked easily. But there are a lot of application fields that need an answer, let imagine a neural network that automatically detect tumours from medical scanner images, or a financial application with a neural network that predict market course, in order to manage savers' money. This is still one of the main open problems in deep learning, however it is a case-specific issue and in the majority of the problems the neural networks can be used without any concern.

Even if nowadays networks with millions of parameters can be efficiently trained and deployed, the number of parameters for a multilayer perceptron explodes too fast augmenting the network depth (number of layers), or the network width (number of neurons in each layer). This create easily overfitting problems and for this reason in state-of-the-art networks the fully connected layers are used only as last layers of the networks, to collect together all the information gathered by the previous layers and determine the result of the network. The main structure used in computer vision is represented by the convolutional neural networks.

2.3 Convolutional Neural Networks

The training process and the specifics layers and techniques utilized will be explained in the corresponding chapter. Although it is useful to present the *Convolutional neural networks*, they represent the main class of networks for computer vision tasks, almost all the neural network that receive images as input are implemented with the utilization of convolutional layers.

Definition 2.9. The name *convolutional neural network* indicates that the network employs a mathematical operation called convolution. Convolution is a specialized kind of linear operation.

Convolutional networks are simply neural networks that use convolution in place of general matrix multiplication in at least one of their layers [4]

In its most general form, the convolution is an operation on two function of real-valued argument, it is denoted with an asterisk and is defined as follows:

$$s(t) = (x * w)(t) = \int x(a)w(t-a)da \tag{2.8}$$

or

$$s(t) = (x * w)(t) = \sum_{a=-\inf}^{\inf} x(a)w(t-a) \tag{2.9}$$

In machine learning application the sum is restricted to non-zero element and usually the input is a two-dimensional image.

The convolution is done between the image (or a part) as input and a two-dimensional kernel, resulting in:

$$s(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n) \tag{2.10}$$

Remark 2.2. The convolution is a commutative operation, so is usually rewritten as:

$$s(i,j) = (I * K)(i,j) = \sum_m \sum_n I(i-m, j-n)K(m,n) \tag{2.11}$$

The two-dimensional convolution can be seen as a multiplication between the input and a doubly block circulating matrix constructed from the kernel, however this representation doesn't give an intuitive idea of the functioning.

To have a better understanding of what really the convolution does, it can be seen simply as a sequence of point-wise multiplication between a block of the input of the layer I and the kernel K followed by a sum on the resulting matrix, due to the fact that kernel K is usually significantly smaller than the input, consecutive blocks of input are each multiplied by the kernel giving the value for one entry of the output matrix.

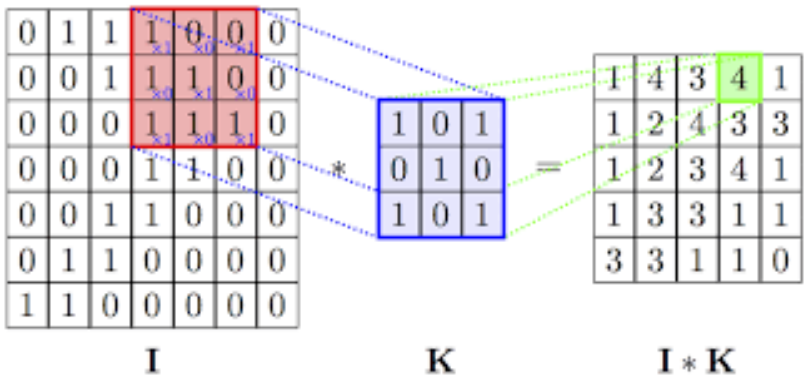


Figure 2.4: Functioning of a two dimensional convolution in a neural network (from [5])

As can be seen from the figure each 3x3 block of the input is element-wise multiplied by the 3x3 kernel, then the sum of the result is stored as the corresponding entry in the output matrix.

Remark 2.3. The output matrix of a two-dimensional convolution in the neural networks is called *activation map*.

The use of convolution leverages three useful ideas:

Fully connected layers have one parameter for each input pixel, while convolutional layer exploit the idea of *sparse interaction*, the kernel is a lot smaller than input, it saves a lot of memory and computational time, and is efficient in detecting small features, like edges.

Another idea is the *parameter sharing*, in convolution each parameter of the kernel is used a lot of times on different blocks of the input, theoretically on almost any position of the input, this is useful because in this way the weights are constrained to learn to detect meaningful features, not only learn what usually is in a particular location of the input.

Due to the property of convolution it has the property of being *equivariant to translation*, in fact if we apply a translation function g to the input I, than the output is equal to the output of original I translated by g.

This means that theoretically if we move an object in different position of the input images, the convolution will extract the same information and the same features in the object's location.

The standard layer's structure of a convolutional network is composed by three consecutive stages:

- Convolutional Layer: the layer performs several convolutions in parallel to produce a set of linear activation maps.
- Nonlinearity: each activation map is run through a nonlinear activation function, that is applied element-wise to the activation map.
- Pooling Layer: a Pooling function is applied on the activations to modify the output

The nonlinear activation function is used also in classical neural network, it is fundamental because the matrix multiplication operations are all linear, and is known that a composition of linear functions is again a linear function.

For this reason, if the fully connected or convolutional layers are used sequentially without a nonlinearity the result would be the same of a single linear function applied to the input, given the fact that is required a non-linear activation function, the problem now is to decide with should be the best one.

As we have seen at first most neural networks used logistic sigmoid activation function:

$$g(x) = \sigma(x) = \frac{1}{1+e^{-x}}. \tag{2.12}$$

or the hyperbolic tangent activation function:

$$g(x) = tanh(x) = \frac{e^{2x}-1}{e^{2x}+1}. \tag{2.13}$$

They are similar in fact $tanh(x) = 2\sigma(2x) - 1$.

However, both these two activation functions have two shortcomings:

- They are quite expensive to compute, in fact the activation function is computed for each entry of each activation map in the network, this means that is computed millions of times in a single forward pass, even a slight improvement can reduce a lot the inference time.
- It could sound counter intuitive but they are too much nonlinear, the nonlinearity is needed to avoid that the use of a lot of layers becomes useless. But the optimization of the neural networks is fundamentally gradient based, and gradient optimization works far better with function that are close to be linear.

For this reason these are not the standard activation function nowadays, in 2011 it has been demonstrated that an elementary function, the *rectified linear units (ReLU)*, enable a better training of deep neural networks.

$$ReLU \ : \ g(z) = x^{+} = max(0, x). \tag{2.14}$$

Relu is not only computationally convenient, but due to the similarity with a linear unit is also easier to optimize that the previous activation function, the plot of these activation functions can be seen in image 2.5.

Even if the ReLU is really effective a lot of generalization has been developed, the principal criticism against ReLU is that all the information about negative values is lost, because negative values are flattened to zero.

For this reason has been tried a variant called *Leaky ReLU*, the idea is to maintain the information on the negative values but also to maintain the easiest possible nonlinearity.

$$Leaky\, ReLU \ : \ g(z) = \begin{cases} x & if \quad x \geq 0 \\ \alpha\, x & if \quad x < 0 \end{cases} \quad with\ \alpha << 1 \tag{2.15}$$

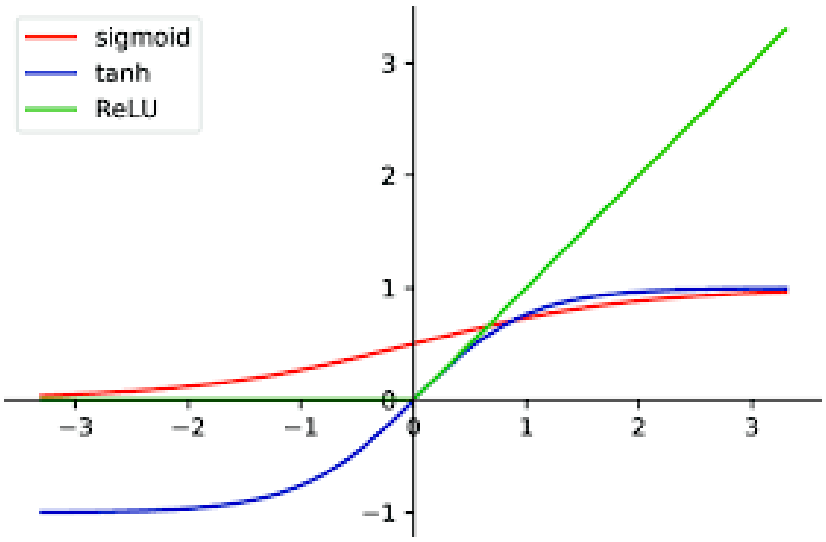


Figure 2.5: Plot of the three principal activation functions (from [6])

The idea is simple instead of flattening the negative values to zero they are multiplied by a fixed parameter α, that is usually in the order of $\frac{1}{100}$.

Another possible generalization is the *parametric ReLU*, the formula is identical to the Leaky ReLU, but the parameter α is learned and not fixed.

Between ReLU and leaky ReLU there is not a clear advantage, the ReLU is faster and the fact that some neurons can die (if they have always negative values) could help to improve training, however in some rare cases, like with bad initialization, all the neurons could die causing the early finish of the training and leaky ReLU solves this problem.

There are also other generalization like *Noisy ReLU* or *Exponential linear unit (ELU)*, but are less used practically in standard neural networks.

To properly describe a standard convolutional block only the *pooling layer* is missing, it simply apply a pooling function to each activation map modified by the nonlinearity.

Definition 2.10. A *Pooling function* is a function that replaces the output of the net at a certain location with a summary statistics of the nearby values.

There are a lot of different pooling function, but the most popular (visible in image 2.6) are:

- Max pooling: this operation transforms a rectangular neighbourhood in a single value given by the maximum output in that rectangle.
- Average pooling: this operation transforms a rectangular neighbourhood in a single value given by the average output in that rectangle.

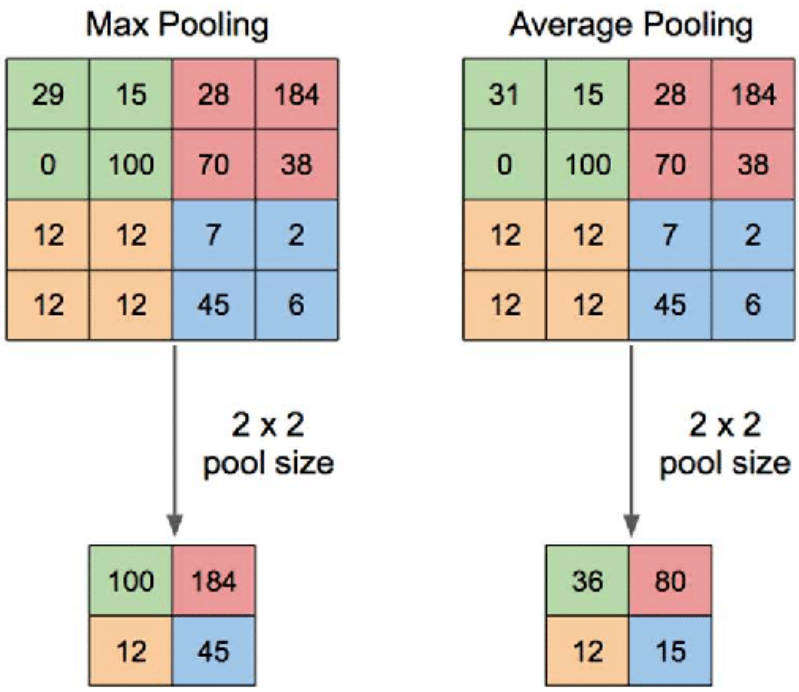


Figure 2.6: Example of pooling operations with a 2x2 pool size (from [7])

Obviously, the pooling operation reduce the dimension of the output maps, usually the pool size is 2x2, so the resulting matrix is a quarter than the output of convolutional layer.

This reduction in dimensions is a controversial aspect of the pooling operation, some famous researchers like Geoffrey Hinton argue that is a major loss of input information and is the real problem of the convolutional networks.

On the other hand, the pooling operation has proven its effectiveness, and the dimensional reduction has the advantages that the following convolutional layer will consider a rectangular patch that corresponds to bigger and bigger sections of the input images.

This is one of the reasons of the hierarchical structure of learned features, in fact the features of the first layers correspond to small portions of input, while the features of the last layers corresponds to way bigger sections of the original image.

Remark 2.4. Despite the critics coming from a major researcher, the pooling operation is fundamental in convolutional neural networks, the only way to get rid of this problem is to create a completely new network structure, Hinton did it two years ago with capsule networks, but in terms of efficiency they are not yet comparable with CNN.

The pooling operation has also other advantages, it obviously reduces the computational cost, in fact it reduce the number of weights by at least three quarter at each occurrence, for this reason is way better to use pooling to augment the convolution's *field of view* rather than use bigger convolutional kernels.

Another important characteristic of pooling is to make the representation approximately invariant to small translations of the input; this is a very useful property if the aim is towards the presence of a feature and not towards the exact location at a pixel precision. And obviously in almost all the computer vision tasks information of location with a pixel-perfect accuracy is useless or at least insignificant compared to the accuracy on features presence.

Understood the basic layers and operations in the convolutional neural networks we can see how a standard classification CNN architecture looks like in figure 2.7

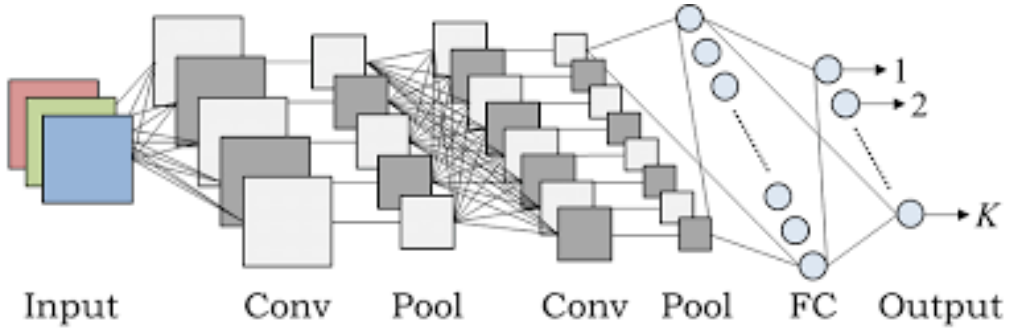


Figure 2.7: Example of a standard classification CNN architecture (from [8])

As can be seen there are a series blocks composed by a convolutional layer and a pooling layer, in between of them is always applied also the activation function, in the end usually are present two fully connected layers, with the last one being the real classifier.

Another classical characteristic of the CNN is that while the dimension of each activation map decreases, the number of activation map usually increases continuing through the network.

The structure of state-of-the-art convolutional neural network is way more complex that these basic architectures, usually it implements different techniques to do regularization and avoid overfitting. In the following sections both the theory behind the training procedure and all the useful regularization techniques are described.

2.3.1 Backpropagation

The training of neural network is performed using an iterative method based on the fundamental algorithm of *backpropagation*, in fact the necessary non-linearity between neural networks' layers causes most interesting loss functions to become non-convex. This means that no closed form exists and the majority of available methods to optimize the networks are iterative gradient-based methods.

A lot of different optimizer exists for neural networks and is an active research field, however all of them are gradient-based so before describing the different optimization methods is important to describe how a simple gradient descent can be performed on a neural network, whose gradient is not intuitive to compute.

When an input x is given to the network it will propagate through the hidden layers and will produce an output y, this process is called *forward propagation*, during the training the same thing happens and a scalar corresponding the value of the loss function is computed. At this point the question is on how to compute in an efficient way the gradient of the loss function, this is done through *backpropagation*.

Definition 2.11. The *backpropagation* algorithm allows the information from the cost to flow backward through the network in order to compute the gradient of the loss function, necessary to update the weights in order to train the network.

Remark 2.5. Often as backpropagation is considered the whole training algorithm, this is not exactly true in fact backpropagation refers only to the method to compute the gradient, while other algorithms such as gradient descent are used to perform learning and optimize the training.

All the backpropagation algorithm is based on the *chain rule*, that states:

Theorem 2.1. *Let x be a real number, let f and g both be functions mapping from a real number to a real number. Suppose that $y = g(x)$ and $z = f(g(x)) = f(y)$.*

Then the chain rule states that:

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}. \tag{2.16}$$

These can be generalized beyond the scalar case. Suppose that $x \in \mathbb{R}^m$, $y \in \mathbb{R}^n$, $z \in \mathbb{R}$ with g maps from $\mathbb{R}^m$ to $\mathbb{R}^n$ and f maps from $\mathbb{R}^n$ to $\mathbb{R}$. If $y = g(x)$ and $z = f(y)$, then:

$$\frac{\partial z}{\partial x_i} = \sum_{j=1}^{n} \frac{\partial z}{\partial y_j} \frac{\partial y_j}{\partial x_i}, \qquad \forall i = 1 : m. \tag{2.17}$$

The idea of the backpropagation is to start computing the gradient of the loss $L(y, t)$ with respect to the network's output y, this is easy because these are all known quantities, including the true label t because we are in the training phase. After that using the chain rule it is possible to flow backward through the network computing the gradient of the loss with respect to the weights of the previous layer and from its inputs, the inputs coincide with outputs of the layer preceding it and again is possible to compute the gradient with respect to weights and inputs. This process continue until the first layer is reached, in this way the gradient of the loss with respect to every weight in the network is computed and the gradient-based optimizer can be applied.

Let's give some notation and let's consider as example a multilayer perceptron with one hidden layer, the same visible in figure 2.1, so a network with a single hidden layer and a single output:

- $\vec{x}$ is the input vector with i entries for $i = 1...n$.
- y is the output of the network, t is the true label associated to $\vec{x}$, $L(y, t)$ is the loss function that have to be optimized.
- $\vec{w_f}$ is the weight vector associated to the final layer and b_f the corresponding bias (it has only one neuron so only one weights vector).
- $\vec{w_j}$ is the weight vector for the j-th neuron in the hidden layer and b_j is the corresponding scalar bias, with $j = 1...m$.
- we define with h_j the value of the output of neuron j before passing through the activation function f, and with o_j the definitive output of the neuron.

We are considering a multilayer perceptron, so the hidden layer is a fully connected layer and it is modelled by the equations:

$$h_j = \vec{w_j} \cdot \vec{x} + b_j \qquad \forall j \in 1 : m \tag{2.18}$$

$$o_j = f(h_j) \qquad \forall j \in 1 : m \tag{2.19}$$

Where $f(\cdot)$ can be any of the classical activation functions (ReLU, sigmoid or tanh).

For the final layer the equation is given by:

$$y = \vec{w_f} \cdot \vec{o} + b_f. \tag{2.20}$$

Where with $\vec{o}$ we are considering the vector composed by the hidden layer's outputs o_j.

As we have said backpropagation runs backward so it starts computing $L(y,t)$ that is a known quantity, after that the gradient with respect to the parameters of the output layer are computed by chain rule:

$$\frac{\partial L(y,t)}{\partial \vec{w_f}} = \frac{\partial L(y,t)}{\partial y}\frac{\partial y}{\partial \vec{w_f}} \tag{2.21}$$

$$\frac{\partial L(y,t)}{\partial y} \quad \textit{is a quantity that depends on the loss used and is known.} \tag{2.22}$$

$$\frac{\partial y}{\partial \vec{w_f}} = \frac{\partial \vec{w_f} \cdot \vec{o} + b_f}{\partial \vec{w_f}} = \vec{o} \tag{2.23}$$

$$\Rightarrow \frac{\partial L(y,t)}{\partial \vec{w_f}} = \frac{\partial L(y,t)}{\partial y} \cdot \vec{o} \tag{2.24}$$

Using the same formulas on the bias b_f and on each of the hidden layer inputs o_j leads to:

$$\frac{\partial L(y,t)}{\partial b_f} = \frac{\partial L(y,t)}{\partial y} \cdot 1 \tag{2.25}$$

$$\frac{\partial L(y,t)}{\partial o_j} = \frac{\partial L(y,t)}{\partial y} \cdot w_{f_j} \qquad \forall j \in 1:m \tag{2.26}$$

Where w_{f_j} represent the j-th entry of vector w_f.

In this way the gradient relative to the parameters of the output layer are computed, the same idea is used backward to compute the gradients for hidden layer. In this case also the activation function has to be considered and the chain rule has to be applied twice.

$$\frac{\partial L(y,t)}{\partial \vec{w_j}} = \frac{\partial L(y,t)}{\partial o_j}\frac{\partial o_j}{\partial \vec{w_j}} = \frac{\partial L(y,t)}{\partial o_j}\frac{\partial o_j}{\partial h_j}\frac{\partial h_j}{\partial \vec{w_j}} \tag{2.27}$$

$$\frac{\partial o_j}{\partial h_j} = \frac{\partial f(h_j)}{\partial h_j} = f'(h_j) \tag{2.28}$$

$$\frac{\partial h_j}{\partial \vec{w_j}} = \frac{\partial \vec{w_j} \cdot \vec{x} + b_j}{\partial \vec{w_j}} = \vec{x} \tag{2.29}$$

$$\Rightarrow \frac{\partial L(y,t)}{\partial w_j} = \frac{\partial L(y,t)}{\partial o_j} \cdot \vec{x} f'(h_j) \qquad \forall j \in 1:m \tag{2.30}$$

In the same way the chain rule leads to:

$$\frac{\partial L(y,t)}{\partial b_j} = \frac{\partial L(y,t)}{\partial o_j} f'(h_j) \qquad \forall j \in 1:m \tag{2.31}$$

$$\frac{\partial L(y,t)}{\partial \vec{x}} = \frac{\partial L(y,t)}{\partial o_j} \cdot w_j f'(h_j) \qquad \forall j \in 1:m \tag{2.32}$$

Having the gradient of the loss function with respect to every parameter is possible to apply the preferred gradient-based optimizer.

Remark 2.6. In deeper networks the algorithm continues in the same way, in fact we have computed also the gradient with respect to input, but it can be thought as the outputs of a previous layer. Using the chain rule the same computations could be performed. In this way the gradient is computed on networks of arbitrary depth and width.

2.3.2 Optimization methods

The backpropagation is an efficient method to compute the gradients in neural networks, however we have to see how the gradient should be used to optimize the network parameter.

Definition 2.12. An *optimization algorithm* is a procedure executed iteratively that aims at reaching the optimum or a satisfactory solution in a set of values. Usually the problem consists in maximizing or minimizing a function.

In deep learning the optimization algorithm is the procedure that repeatedly update the learnable parameters (weights and biases) in order to minimize the value of the loss function.

To define and describe an optimizer we need to use:

- θ that represent in general the learnable parameter considered.
- μ is the learning rate, is an hyperparameter defined at the beginning of the training, usually it is not fixed for all the training, but it slightly decreases during the process.
- $\nabla L(\theta)$ is the gradient of the loss function with respect to the parameter considered.

There are a lot of different optimizer that can be used to train a neural network, however almost all of them are based on the *gradient descent*, that is the simplest optimizer available. Gradient descent is a first order iterative algorithm based on the updating formula:

$$\theta \leftarrow \theta - \mu \cdot \nabla L(\theta). \tag{2.33}$$

It relies on the fact that the negative of the gradient corresponds to the direction of the fastest decrease of a function, so it updates the parameter in the direction of the fastest decrease of the loss function.

This method is guaranteed to find a local minimum; however, the loss function is usually non-convex, so a local minimum is not also a global minimum and for this reason is not guaranteed to be a good solution. This method is also affected by a lot of oscillations and for this reason the training usually is not as fast as it could be with better optimizer.

The gradient descent method can be used some in different ways:

- Using all the training data at once, in this way the real gradient is computed, is the simplest method but the less effective.
- On single training data, this method is called *stochastic gradient descent (SGD)* because in this way the parameter updates have high variance, but it is not a defect because in this way new and better local minimum could be discovered.
- On mini-batch (*mini-batch gradient descent*), so the gradient is computed every batch of training data and the parameters are updated. Is the most used of the three approaches because is a good compromise between having variance and a good approximation of the real gradient.

To avoid getting trapped in one of the non-optimal local minima and to solve the problem of the slow convergence with bad values of the learning rate parameter some more refined algorithm have been proposed to improve the gradient descent efficiency.

To solve the excessive oscillations in SGD a technique called *momentum* has been added, the idea is to maintain a fraction of the previous update's direction also in the successive updates. This has the effect of a sort of time average to determine the update direction. It is described by the formulas:

$$V(t) = \gamma V(t-1) + \mu \nabla J(\theta), \tag{2.34}$$

$$\theta \leftarrow \theta - V(t). \tag{2.35}$$

Where $V(t)$ represent the update direction at the t-th iteration of the algorithm. The parameter γ is the momentum term and is usually set to a value slightly smaller than one, like 0.9 or similar. This method reduces

the oscillation and for this reason has usually a more stable and faster convergence than simple SGD.

A researcher named Nesterov noticed that a momentum has a big overcome, when a local minimum is found the momentum term is still big and this could cause the miss of the minimum. To solve this problem the idea is to compute the gradient not in the point where the function is, but in the point where we know that the momentum will lead to. The resulting formulas are:

$$V(t) = \gamma V(t-1) + \mu \nabla J(\theta - \gamma V(t-1)), \tag{2.36}$$

$$\theta \leftarrow \theta - V(t). \tag{2.37}$$

In this way the gradient will try to compensate the momentum effect in case that the momentum lead to a point beyond the minimum.

At this point the main problem of optimizer is that they update all the parameters using the same learning rate, moreover the learning rate is an hyperparameter that is hard to set in an optimal way for all the network, for this reason some methods with adaptive learning rates are described.

Adagrad is an algorithm that allows the learning rate μ to adapt for each parameter based on their previous gradients. In particular the learning rate is divided by the square root of the sum of all the previous gradients squared. In formulas:

$$S(\theta) \leftarrow S(\theta) + (\nabla J(\theta))^2, \tag{2.38}$$

$$\theta \leftarrow \theta - \frac{\mu}{\sqrt{S(\theta) + \epsilon}} \cdot \nabla J(\theta). \tag{2.39}$$

In this way the learning rates is automatically smaller for parameters that have already changed a lot and higher with parameters that have still to be optimized. However, the main disadvantage is that the learning rate is constantly decreasing, and the risk is that become too small before the convergence or that it leads to very slow convergence.

For this reason, in the extension called *RMSProp* the same computations are performed but the sum of the squared gradients is computed as a decaying mean so it is not always increasing.

$$S(\theta) \leftarrow \gamma S(\theta) + (1-\gamma)(\nabla J(\theta))^2, \tag{2.40}$$

$$\theta \leftarrow \theta - \frac{\mu}{\sqrt{S(\theta) + \epsilon}} \cdot \nabla J(\theta). \tag{2.41}$$

The last algorithm that presented is currently the most used optimizer for neural network training, *Adam* is a further improvement of adaptive

methods that takes into account both mean and variance of the previous gradients to adapt the learning rate value. It can be seen as a variant of the combination of RMSProp and momentum in fact it uses both the linear (like momentum) and the squared (like RMSProp) values of the previous gradients.t also add a bias correction to the estimates of both first and second order momentum of the gradient. The algorithm is given by this iterative method:

- Update first and second moment estimates:

$$V(\theta) \leftarrow \gamma_1 V(\theta) + (1-\gamma_1)\nabla J(\theta), \tag{2.42}$$

$$S(\theta) \leftarrow \gamma_2 S(\theta) + (1-\gamma_2)(\nabla J(\theta))^2, \tag{2.43}$$

- Correct biases:

$$\hat{V}(\theta) = \frac{V(\theta)}{1-\gamma_1}, \tag{2.44}$$

$$\hat{S}(\theta) = \frac{S(\theta)}{1-\gamma_2}, \tag{2.45}$$

- Update the parameters:

$$\theta \leftarrow \theta - \mu \cdot \frac{\hat{V}(\theta}{\sqrt{\hat{S}(\theta)} + \epsilon}. \tag{2.46}$$

The choice of the optimization algorithm depends strongly on the learning task, there is no definitive consensus on this choice, however with sparse data and difficult learning tasks the adaptive algorithms usually perform better and between the adaptive methods usually Adam is the one that performs better.

2.3.3 Regularization techniques

The training data contains information about the regularities in the mapping between input and output, but they also contain a lot of sampling errors or outliers. When a neural network is trained it could fit well all the real regularities, but it could also fit a lot of sampling errors because the neural networks have millions of parameters, for this reason one the main problems that have to be tackled is the *overfitting*.

Definition 2.13. *Overfitting* is the production of an analysis that corresponds too closely or exactly to a particular set of data and may therefore fail to fit additional data or predict future observations reliably.

Figure 2.8: Typical plot of training and test error depending on model complexity (from [9])

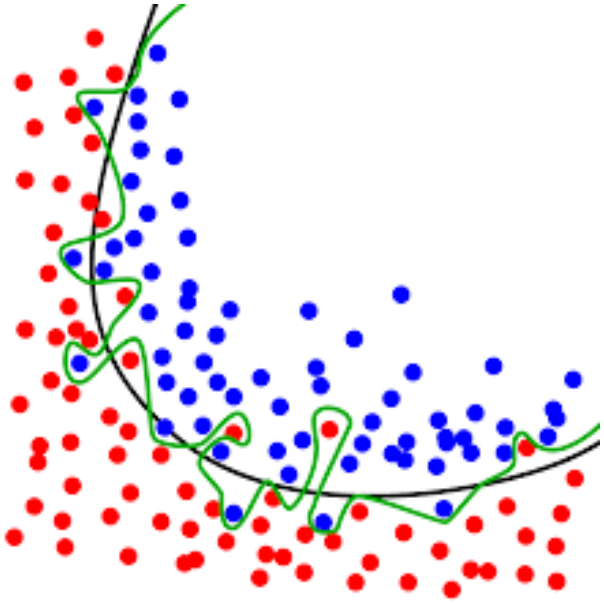


Figure 2.9: Example of overfitting in a standard 2D classification problem (from [10])

As can be seen in figure 2.8 by increasing the model complexity is not always an improvement in the performance, in fact when the model becomes too precise in fitting the training data usually it lost the ability to generalize and find good prediction also on future test data.

In the image 2.9 is showed the typical how it works overfitting on a simple separator that could be computed for example with SVM. The green separator performs better on training data, but it will probably find problems in fitting also test data, because trying to fit all the training data the separator found doesn't represent the real division between the classes' distributions,

while the black separator has an higher training error but it represent the real regularities of the classes and not also the sampling errors.

The very first method to avoid overfitting is to pay attention and understand when the training has to be stopped. Usually when training a network, the trained weights are saved as checkpoints many times, in this way the checkpoints are used to compute the performance on the validation set and the best checkpoint can be chosen. This way of working is called *early stopping* in fact usually with long training the best checkpoint are not the last ones, and when the performance on validation begin to get worse the training is stopped.

The neural networks are very complex models, for this reason a variety of regularization techniques have been developed to avoid overfitting.

Data augmentation

The overfitting often is simply caused by a wide difference between the number of parameters and the number of training data. For this reason, the simplest way to reduce it is to increase the size of the training set. Working on images there are some techniques known as *data augmentation* that aims at increasing the size of the training set without the need to find completely new images.

The principal data augmentation algorithms for computer vision are:

- Horizontal flipping
- Cropping
- Rotation or translation
- Gaussian noise
- Colour jitter

The horizontal flipping is probably the most used and safe method, it consists in simply flipping the image with respect to the horizontal direction. It is a safe bet because only in very particular task a flipped image can represent something different, but in any classification or object detection task it maintains the label and double the number of training data with almost zero effort. The vertical flipping instead is not used often because it usually creates images that doesn't represent nothing.

Another vastly used technique is the image cropping, it consists in cutting the image taking only a part of the original image. It is the standard choice if the input images are bigger than the network's input, in this way the image are also resized. Usually the crops are taken at the four corners and

at the centre of the image, so from one input image 5 resized crops are given as input.

Another simple technique is the rotation or translation of the image, this are less used because they can modify dimension or content of the image, so they should be used only if they well adapt on the task considered. In principle any method that creates image with different pixel values without altering the content can be used as data augmentation. Some other methods consist in directly modify the pixels, for example adding some Gaussian noise on random pixels, or modifying the contrast of the image changing the pixel values.

However, these techniques are less used because flipping and cropping works very well in keeping the content while augmenting the dataset dimension, in fact using the standard cropping to resize the image and the horizontal flipping on the resulting crops the result is to have a dataset that is 10 times bigger than the original one.

Parameter norm penalties

Usually the principal methods for regularization are based on adding to the loss function a parameter norm penalty. This kind of methods are common in regression models and corresponds to Lasso and Ridge regression models.

The idea is the same also for neural networks, the loss function is modified in order to account also for a penalty on the parameter values, the effect is to avoid to have too big parameters and leads to models that in case of weights connected to the same neuron will tend to give similar values to the weights and not to strongly rely on only a subset of weights with high values to produce the desired output.

Thinking about convolutional neural networks this means that, on average the network will try to found features relying on all the pixels, so even if the result on training could be effective the network will avoid to have weights of specific pixels with too high values, instead it will rely on all the neighbour pixels to have the same output value.

The norm penalty can be linear and is called L^1 *regularization*, or quadratic that gives L^2 *regularization*. The L^1 *regularization* use the absolute value as norm penalty and gives an objective (loss) function for each parameter that is given by:

$$\hat{L}(\theta) \;=\; L(\theta) + \lambda \mid \theta \mid . \tag{2.47}$$

This is not often used in neural networks, because it doesn't give difference for example in two scenarios like $\theta_1 = 0.9\,,\ \theta_2 = 0.1$ and $\theta_1 = 0.5\,,\ \theta_2 =$

0.5, but we have seen that to avoid overfitting is better to tend to similar weights in order to depend less on single pixels.

For this reason, the most used regularization is the L^2 *regularization*, in this case the loss is modified using a quadratic factor.

$$\hat{L}(\theta) = L(\theta) + \frac{\lambda}{2}(\theta)^2. \tag{2.48}$$

This gives a result known as *weight decay*, that means that the network learning try both to optimize the loss and to solve the problem using weights that are as small as possible, furthermore this time the network will tend to average the weights that are linked to the same neurons and is a useful result because avoiding single weights with high values is avoided also the strong dependence of the network from single pixels. This result can be seen computing the resulting gradient and the update for example using a gradient descent optimizer.

$$\nabla_\theta \hat{L}(\theta, x, y) = \nabla_\theta L(\theta, x, y) + \lambda\theta. \tag{2.49}$$

$$\theta \leftarrow \theta - \mu\lambda\theta - \mu \cdot \nabla_\theta L(\theta, x, y) = (1 - \mu\lambda) \cdot \theta - \mu \cdot \nabla_\theta L(\theta, x, y). \tag{2.50}$$

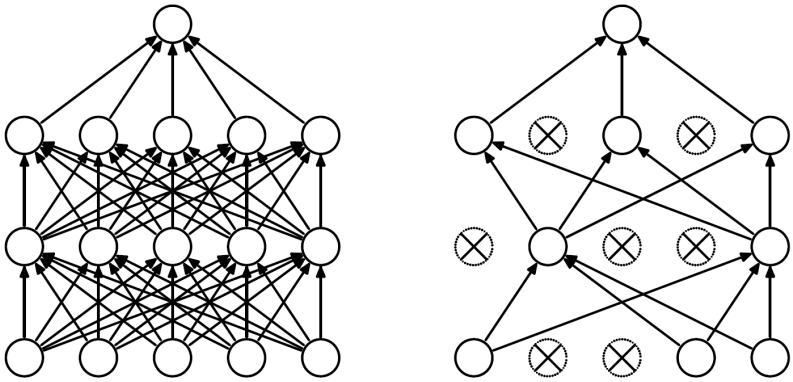
Is visible from the equation that regardless the value and the direction of the gradient there is a term that tend to decrease the weights' values, furthermore this term is proportional to the weight value for this reason the bigger weights decrease faster giving also an effect of averaging in the parameters' values.

Dropout

In big layers the overfitting can happen under the form of co-adaptation, this means that the neurons don't work as independent feature extractor, but they adapt to depend one on the other in order to gives output that fits the training data. This is an overcoming for training because usually in this way the neurons stop detecting real feature and start to overfit adapting one to the others.

In order to solve the co-adaptation problem, the best algorithm available is *dropout*. *Dropout* is a powerful method of regularization that is computationally inexpensive. The method consists in removing, or better deactivating, random neurons in each training iteration.

As shown in figure 2.10, eliminating some neurons the number of weights involved drop significantly, this operation is done each training batch on random neurons, the fraction of neurons removed is an hyperparameter but is usually set equal to 0.5.



(a) Standard Neural Net (b) After applying dropout.

Figure 2.10: Effect of dropout on fully connected layers (from [11])

This obviously prevent the co-adaptation phenomenon because the neurons are forced to find useful features even when their neighbour neurons are deactivated. This is also useful as general prevention of overfitting because the network cannot rely on single neurons but have to be able to find features with almost any subset of neurons.

Practically dropout is performed adding this operation on each fully connected hidden layer, in convolutional layers is less used, it could help against overfitting but is also risky because it has fewer clear effects. In dropout training also the loss is computed only on active neurons, so it optimizes only the weights that the network is practically using to generate the output. This operation is performed only at training time, while during the test inferences the dropout is eliminated, and all the neurons are used at every iteration.

2.3.4 Batch normalization

Another general problem in any training algorithm is the normalization problem. Almost any training algorithm works better on normalized data because in this way the weights will not depend on the intensity of the input data but only on its content that is given by the relative differences between pixels.

Another important consequence of normalization is that reduce the problem of coordinating updates on all the layers, in fact using gradient-based optimization we update each weight thinking about the other weights as fixed, instead they are all updated simultaneously and having many different values can create an effect of updates that propagating through the network change the network at the point that the last updates have no more the effect to improve performance. Having the inputs normalized reduce the

effect because it reduces the differences on the possible inputs of each layer and for this reason also the influence of the updates of the previous layers.

Batch normalization is an approach to reparametrize any deep network in order to have data and activations that are almost always standardized. Also, batch normalization allows each layer of a network to learn by itself a little bit more independently of other layers. The problem is that during the training we don't know the true mean and variance of the data that flows between all the layer, but we have information only on the values computed on the current batch.

Consider with X the minibatch of activations of the layer that we want to normalize and suppose that each row represents the activation of one unit (neuron) for the different example in the batch. So X has one row for each neuron in the layer, and one column for each example in the batch. In order to normalize we would compute:

$$\hat{X} = \frac{X - \vec{\mu}}{\vec{\sigma}} \tag{2.51}$$

Where $\vec{\mu}$ should be the vector of the means of the activations for each neuron, and $\vec{\sigma}$ should be the vector of standard deviations for each neuron. However, as we have said these vectors are unknown because we could compute them only at the end of the training when we have used all the data but at that time obviously the network would be already trained without normalization.

Remark 2.7. The arithmetic in the equations about batch normalization should always be considered as element-wise computations, in fact each activation is normalized subtracting the corresponding mean and dividing by the corresponding standard deviation.

Remark 2.8. The activations considered for each neuron are the result of the layer and the application of the nonlinearity, in fact applying a normalization before the nonlinearity could be quite useless.

To solve this problem the mean and the variance are computed on every batch during the training, let $\vec{x}_i$ with $i = 1 : m$ be the activations for a batch of m data, so $\vec{x}_i$ is a column of the previous X matrix. The mean and variance for the batch are computed for each layer independently and given by:

$$\vec{mu}_B = \sum_{i=1}^{m} \vec{x}_i, \tag{2.52}$$

$$\vec{\sigma}_B^2 = \sum_{i=1}^{m} (\vec{x}_i - \mu_b)^2, \tag{2.53}$$

Each dimension k of the d-dimensional vectors $\vec{x}_i$ is normalized separately:

$$\hat{x}_i^{(k)} = \frac{x_i^{(k)} - \mu_B^{(k)}}{\sqrt{(\sigma_B^{(k)})^2 + \epsilon}}, \quad where\, k \in [1, d]\, and\, i \in [1, m]. \tag{2.54}$$

The resulting activation $\hat{X}$ has rows with zero mean and unit variance. To restore the representation power of the network then a transformation is applied:

$$y_i^{(k)} = \gamma^{(k)} \hat{x}_i^{(k)} + \beta^{(k)}. \tag{2.55}$$

The resulting matrix Y is the real output of that activation and will be the input of the next layer in the next iteration. The parameters $\vec{\gamma}$ and $\vec{\beta}$ are learned parameters that allow the network to maintain the possibility to have input of any mean and standard deviation, however now the mean is determined only by the value of the parameter $\vec{\beta}$ and doesn't depend on previous layers in this way the layers are more independent during the training phase.

During the test phase the mean and variances are no more compute as estimates on a single batch, but they can be computed on the entire training population, in this way the operation performed are always the same on any input image and basically consists in some linear transformations between the layers. In this way the output is a deterministic function of the input data as it should be during inference.

Another important consequence of the batch normalization is that having a more reliable learning is possible to use bigger values for the learning rate, improving the training speed. Moreover, the network is less dependent to different initialization schemes.

3 Hardware vs CNN

This chapter gives an overview on edge computing, motivation, optimization procedures and hardware available will be presented.

Definition 3.1. The *edge computing* is a distributed computing paradigm which brings computation closer to the location where it is needed to improve response times and save bandwidth.

This is a general definition of edge computing, in a machine learning environment it consists in performing some or all the computations directly in the location where the data are collected in order to don't have the need to move the data from their original source.

Obviously in order to apply machine learning algorithm to the data a powerful processor of small dimensions is needed, in fact is inconceivable that for each application an entire computer with proper computational power is installed at the data source. In particular applied on computer vision is interesting to see what can be implemented using processors that can be installed inside or together with a video camera.

The edge computing is useful for a lot of different reasons:

- Reliability, having the computation distributed on the single application nodes gives a better management of failures. In this way every failure compromise only the single nodes in which it happens, while having a centralized computation has the risk of a systemic failure that compromise all the application that rely on it.
- Scalability, it doesn't rely on connection and to increase the number of nodes is only needed to deploy a new edge processor that will take care of all the computation, the number of devices is not constrained by the bandwidth or the centralized computational power.
- Privacy and security, this is probably the most important feature of edge computing, the data collected are no longer sent to be analysed, performing the computation on edge the data can be analysed and cancelled if needed without never leaving their source. This is an extremely

useful property in fact often the data are protected by privacy and having the possibility of analysing the data directly on source solves most of the privacy and security issues.

Usually the edge devices share two other qualities, the, usually, low price compared to bigger and more powerful processors, and the low power consumption, that is extremely useful in order to deploy big workload on these small devices.

However, the edge computing has also some issues and complications. The primary issue in performing the edge computing is represented by the computational power. Obviously using small devices installed on edge the processors involved are far less powerful than usual computers, however the technology is rapidly improving and in the last times a lot of interesting edge processors have been developed.

We will consider two different hardware, *Coral Dev Board* and *NVIDIA Jetson Nano*, they represent the latest devices available and they share some common useful properties. Both the devices are standalone development board, these means that they are almost ready to use and any program or algorithm can be deployed on them, most of the concurrent edge devices are simple processors that should be implemented in some hardware or accelerating devices that should be used together with a board.

Furthermore, both these devices are built specifically for computing deep learning inferences, that is exactly what was needed, the both contains a CPU a GPU and the special tensor cores that are built for matrix multiplication, the most recurrent operation in convolutional neural networks.

Even if they share a lot of features these two devices works in very different ways, this is due to the tensor units. In Coral Dev Board is composed by a Google's own processing unit called tensor processing unit (TPU). The Jetson Nano is provided with an NVIDIA GPU with cuda cores, it doesn't feature the tensor cores developed by NVIDIA but the GPU it has is still enough powerful to perform neural network inference. These differences have an impact on the workflows using these board, the two particular workflow are described in the following sections.

3.1 Coral Dev Board

The first device considered is the Coral Dev Board developed by Google, it is a single-board computer that is built specifically to perform machine learning inference with high efficiency on a small edge device. Specifically, the board is designed to perform neural network inference.

The hardware contains all the usual features, a low power CPU, a small GPU, all the needed connectivity and memory units. What really makes this board more than a usual raspberry is the *tensor processing unit (TPU)*.

A *tensor processing unit* is an application specific integrated circuit built to be an artificial intelligence accelerator, it is developed by Google specifically for neural network machine learning. For some years it was used only internally by google, this board, available from January 2019, is the first commercial device that use the TPU.

The standard computations in each neuron are:

- Multiply the input data with the weights.
- Aggregate the results adding them together.
- Apply a nonlinear activation function to the resulting value.

The sequence of multiplication and addition can be written as a matrix multiplication and is the most computationally expensive part of running a trained neural network. TPU is designed specifically to address these computations faster, in fact the most important computational resource of TPU is an 8-bit matrix multiplication unit.

The efficiency of TPU is given mostly by this unit, however it works only with 8-bit precision and that is the principal weakness of the Coral Dev Board. All the standard neural networks work on standard float precision (32-bit), for this reason strong optimization procedure have to be performed in order to perform inference. Another important drawback of this board is that it is able to perform only the inference, the training phase have to be implemented on another device and if optimized correctly the neural network should be deployed on the board.

3.1.1 Quantization aware training

The requirement of 8-bit precision is a fairly strong requirement, however it is not even the only requirement for the Coral.

The act of transforming a neural network from 32-bit precision to 8-bit precision is called *quantization*, this is due to the fact that the 8-bit precision corresponds to the use of integer number and no more floats, for this reason the range of values for the parameter is "quantized" and cannot take any real value as before. The neural networks can be quantized with two different approaches:

- Taking quantization into account already during the training, this is called *quantization aware training*, it has an higher accuracy but is

also more complex and less flexible, in fact it has to be implemented before than starting the training and all the network already available on the various model zoo can't be used but have to be retrained from the start.

- After the training, is the *post-training quantization*, it is a bit less accurate than quantization aware training, but it is way more flexible. In this approach the model is optimized after the training, for this reason any model that is already trained on 32-bit can be optimized and run on 8-bit precision.

Unfortunately, the Coral Dev Board is compatible only with the quantization aware training, for this reason not all the models are available on this board and only few models can be found already trained with 8-bit quantization.

In order to perform an efficient and accurate inference the quantization aware training approach consider principally two aspects:

- The operator fusions performed at inference time are modelled already at training.
- The effects of quantization on inference are considered at training time.

For what concerns the first point, the principal modification is that the batch normalization layers will be folded into the previous convolutional or fully connected layer.

To take into account the future quantization during the training the idea, developed in [12], is to add *fake quantization* nodes during the forward passes to simulate the effect of quantization during inference. The backpropagation happens as usual and the weights are still stored in floating point, but this approach aims at having better performances when the quantization will be used during the inference.

In order to maintain the precision of 32-bit precision the best way should be finding an affine mapping between the real valued weights and the new quantized weights, in this way no information is lost, so the objective is to find:

$$r = S(q - Z) \tag{3.1}$$

Where: r is the real value number (32-bit), q is the quantized respective (8-bit), S is called scale parameter and Z is called zero-point because it represents the 8-bit value that corresponds to the real value 0.0. Obviously, the real values are too much to find a proper mapping, so the idea is to

restrict the number of possible real value of the weights already during the training, this is the operation performed by the fake quantization nodes.

In order to restrict the range of possible values the weights are mapped on a subset of still real values using the following equations:

$$clamp(r; a, b) := min(max(r, a), b), \tag{3.2}$$

$$s(a, b, n) := \frac{b - a}{n - 1} \tag{3.3}$$

$$q(r; a, b, n) := \left\lfloor \frac{clamp(r; a, b) - a}{s(a, b, n)} \right\rceil s(a, b, n) + a. \tag{3.4}$$

Where q is the quantized value of the real number r, $[a, b]$ is the quantization range, and n is the number of values of the quantized space that is fixed and equal to $2^8 = 256$ for 8-bit precision. The quantization range is computed simply putting $a := min(w)$ and $b := max(w)$. As we have already said the mapping is done such that 0.0 corresponds to $z(a, b, n)$.

3.1.2 Edge TPU optimization procedure

The general procedure to do quantization aware training and inference is:

- Create the training graph for a normal floating-point model.
- Insert the fake quantization nodes.
- Train in simulated quantized mode until convergence.
- Create and optimize the trained graph for 8-bit inference.
- Run inference using the quantized inference graph on specialized device.

Using the Coral Dev board, the same procedure is performed but some specific instruments have to be used. The training graph must be implemented using TensorFlow and must be trained using quantization aware training. The trained graph is then converted to TensorFlow lite, that is the lighter version of standard TensorFlow library, it is specialized in building and optimizing neural network for lower precisions and is used mostly with light networks suited for mobile or edge applications. Finally, the lite model has to be ran with the online compiler of Google Coral, specific for edge TPU, and is finally ready to be deployed on the device.

This procedure seems to be quite straightforward and easy to use, however the real problem is that these steps introduce a lot of constraints to the

available models and for this reason the Coral Dev Board is not a flexible device.

The bigger constraint is given by the quantization aware training, all the state-of-art networks are available already trained usually on general purpose datasets, unfortunately only a real small subset of the networks is present also with quantization aware training. In theory to solve this problem is only needed to repeat the training from scratch adding fake quantization, however it is a really long process that need a lot of computational power and rarely gives results better than the ones already available.

Some other constraints are given by the conversion to TensorFlow lite and the use of edge TPU compiler. These framework in fact doesn't support all the layer used in complex networks for this reason the network that are not already available are practically useless on this board.

However, is worth to say that this process is really effective, the speedup is huge, and the resulting performances are really promising as it is shown in the last section of this chapter.

3.2 Jetson Nano

The *Jetson Nano* is another single-board computer, developed by NVIDIA, it is the smaller product of the Jetson family that feature two other devices of bigger dimensions and computational power, like the Coral it is specifically built to run neural network applications on edge with high performance and low power consumption.

The main feature of the hardware is given by the 128-core integrated NVIDIA GPU provided of cuda cores to perform inferences with high computing performances.

This board is designed to work on networks optimized with TensorRT platform and gives its best performance when used to perform inference on network accelerated on 16-bit precision (half precision). The nano doesn't support the 8-bit precision that could give a further speedup but has the advantage of working with any network available with any of the principal frameworks.

deploying neural network application on the jetson nano is quite the same as doing it on a standard computer with an NVIDIA GPU, this versatility is useful not only in the choice of the neural network but also in every other software portion that is not related to deep learning, for example any computer vision algorithm used to preprocess or postprocess the image or to extract information useful also for the neural network.

On the other hand, this comes with a less specific hardware that features less speed up compared to other devices.

Another important difference is that the Jetson Nano is able to perform also the neural network training, however this is not recommendable because it would result in a very long process, this feature is indeed useful in fact some specific finetuning could be performed on the device without spending too much time.

3.2.1 Post-training quantization

Even if the Jetson nano can run standard flost-32 models, it is highly recommended to use it with networks quantized on half precision. In fact, this is the way the jetson nano can have competitive performances against any edge device.

In order to perform 16-bit quantization is not needed the quantization aware training, but is enough using a *post-training quantization* approach, furthermore is not even needed a calibration step that is instead needed to quantize post training in int8 precision.

The first advantage of using half precision is reducing the storage space, this improvement is underestimated using computers that contains big memories, but edge devices like the Jetson Nano usually features only small fast memories. In fact, some of the bigger network could even create failures if used with single precision on the Nano.

In order to perform the quantization a specific instrument is used, *tensorRT* is a platform developed by NVIDIA that optimize neural network model and speed up for inference across GPU-accelerated platforms. Obviously being developed by the same company the Jetson Nano is built to deploy network optimized with TensorRT, the official benchmarks are computed precisely on network optimized and quantized with this platform.

3.2.2 TensorRT optimization procedure

TensorRT performs a lot of optimization procedure in addition to the quantization procedure. The first operation done is the elimination of layers with unused outputs to avoid useless computations. Next each block of convolutional, ReLU and batch normalization layer is fused in a single layer that performs all these operations, moreover it performs also horizontal layer fusion.

Remark 3.1. The fusion operations don't change the underlying computation in the graph, it is only done to restructure the graph to perform the operations much faster and more efficiently.

To better understand what layer are fused together using TensorRT, in figure 3.2 is shown the vertical and horizontal layer fusion to the GoogLeNet Inception module graph.

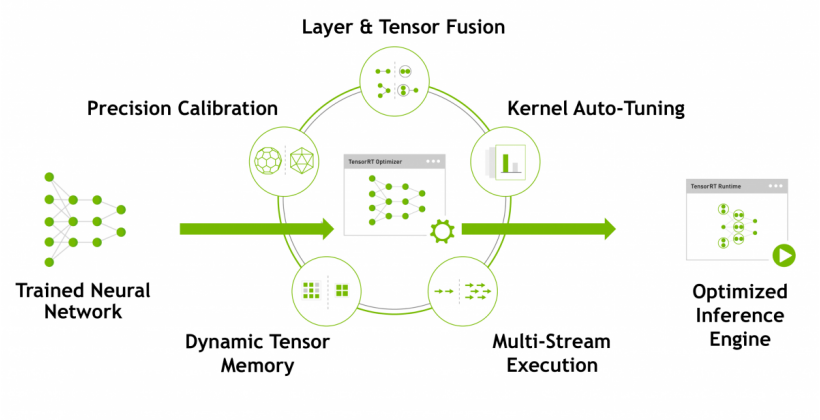


Figure 3.1: Representation of the principal optimization techniques performed by TensorRT (from [13])

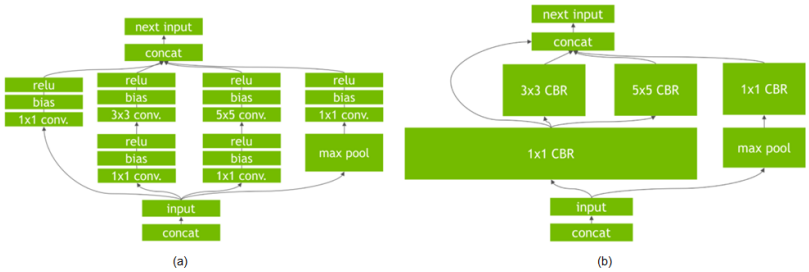


Figure 3.2: Example of TensorRT vertical and horizontal layer fusion (from [13])

The quantization procedure in this case is way easier than with int8, in fact the range of values that can be described using half precision is huge compared to the 8-bit precision, for this reason there no need of any calibration or fine tuning operation, the only operation done is mapping the values of all the weights from 32 bit to 16 bit maintaining all the other operations. The loss of accuracy with this quantization is almost negligible.

The most important thing using TensorRT optimization tool is to make sure that the optimization procedure is performed on the same NVIDIA GPU that will be used for inference, in fact part of the procedure optimize the graph structure relying on the specific hardware structure and using a different GPU in inference usually cause failures.

3.3 Benchmarks

The comparison between the two boards will be focused mostly on the software flexibility and on the inference speed, however a lot of other factors could be considered like the price and dimension of the boards, or the power

consumption and the module temperature during high workloads. Jetson Nano and Coral Dev Board are an interesting comparison because they share similar characteristic in terms of price, dimension and power consumption, for this reason they could be considered as direct competitor and the performance comparison is meaningful also because as we have seen they work with completely different techniques.

Both Nvidia and Google have released official benchmarks, however these are usually inflated by over specific optimization, moreover the official benchmarks shared few common networks to compare the devices and more data could be useful to understand the differences of the deployment of networks on these boards.

I tried to optimize and deploy the all the possible neural networks that can be ran on both the devices, the optimization workflow used is the one described in the previous section and quantized graph are used. Unfortunately, the number of networks considered is limited to eight common networks, these limitations as we have seen are due all to the optimization procedure required by the Coral Dev Board, in fact all the networks available with quantization aware training have been considered. The only network that is added to the analysis without being available on both the devices is the SSD Inception V2, that is one of the principal candidate networks to be used in people counting, for this reason it has been added anyway to understand the possible performance of deploying it on Jetson Nano.

Neural network	Inference time		Inference per second		Standard deviation	
	Coral	Jetson	Coral	Jetson	Coral	Jetson
MobileNet V1	4.50 ms	25.56 ms	222.22	39.12	0.09 ms	0,52 ms
MobileNet V2	4.75 ms	29.91 ms	210.52	33.43	0.10 ms	0.42 ms
Inception V1	6.33 ms	15.73 ms	157,98	63.59	0.23 ms	0.53 ms
Inception V2	20.75 ms	20.33 ms	48.19	49.19	0.11 ms	0.54 ms
Inception V3	57.44 ms	50.08 ms	17.41	19.97	0.25 ms	0.69 ms
Inception V4	107.81 ms	98.30 ms	9.28	10.17	0.22 ms	0.77 ms
SSD MobileNet V1	15.32 ms	58.37 ms	65.27	17.13	1.55 ms	2.33 ms
SSD MobileNet V2	20.97 ms	70.45 ms	47.69	14.19	2.43 ms	2.53 ms
SSD Inception V2	ND	65.28 ms	ND	15.32	ND	2.59 ms

Figure 3.3: Inference time benchmark for Jetson Nano and Coral board

Before analysing the results is important to give an overview on the neural networks considered in the benchmarks. First of all, it has to be noted that the first 6 networks (MobileNets and Inception) are image classification networks of increasing complexity and predictive power in the various version indicated. The last three network considered are network composed by an image classification network as feature extractor and the Single Shot multibox Detector (SSD) that is an algorithm to perform object

detection, to understand the Benchmark is not fundamental the knowledge of its structure, for this reason it is presented in the section dedicated to the object detection in the following chapter about the model construction. The most important feature that has to be known is that Mobilenets are network specifically design to be fast and light because are developed to be implemented on mobile or edge devices. While the Inceptions are quite famous networks implemented by Google researchers and each version is an improvement of the previous ones but also a bigger version of it, Inception V4 is by far the bigger network considered in the analysis and it can no longer be considered as a light network for edge application.

Analysing the results is clear that the Coral Dev Board has a clear advantage using MobileNets, both for image classification than as feature extractor. This result is probably due to the stronger optimization performed by the int8 quantization, but the results are really remarkable and the speed of Coral on this network is well over the real-time needed speed. However, the performance of the Jetson Nano, even if way worse than the Coral, is still a good result because it is still a result that allow any real-time application.

The surprising result of the benchmark is that this great difference is valid only for what concerns the smaller networks considered (MobileNets and Inception V1), as soon as the network dimension increase the gap between the two boards is completely erased and indeed the Jetson Nano features higher performance on the last three version of Inception architecture. To understand this results a note on the difference of the boards has to be done, we have seen that the Coral performs a stronger optimization using int8 precision, it also features a tensor processing unit, that is a specialized processor for deep learning absent in Jetson Nano architecture, however fundamentally the Jetson Nano is a more powerful processor that has a better CPU and RAM but above all that feature a far bigger GPU. This is the reason of this difference of results based on the dimension of the network considered, with small and optimized networks the Coral has a great advantage, but when the depth of the network increases the difference is mitigated and then erased by the more powerful architecture of the Jetson Nano that in fact performs better on the bigger networks.

Taking only these results in consideration the clear winner still seems to be the Coral Dev Board, however has to be taken into account also the difference in network availability on the two boards. The Jetson could have been used on multiple other different networks without problems, while the Coral Dev Board has a lot more constraints. Obviously, this board are built to work with network suitable to edge computing and the possibility of using all the bigger networks on Jetson Nano is in practice useless, however the concrete possibility of using residual networks, tiny YOLO or networks for segmentation or pose estimation is an important feature. The only

reasonable conclusion of this benchmark is that fundamentally these boards are very different and the best one can be determined only based on the specific application that must be implemented.

For what concerns our problem, as will be seen in the next chapter, the speed necessary was to have an inference in the order of 10 images per second, for this reason the constraint of having only object detection network based on MobileNets of the Coral Dev Board are way too restrictive and the speed given by the Jetson Nano is in the right order of magnitude. For this reason, from now on the only device used is the Jetson Nano.

4 Models choosing for people counting

After the presentation of deep learning theory and the analysis of the hardware used, is the time to define what will be the model used to solve the problem.

As already mentioned in the introduction the main task that must be tackled is the people counting specifically in crowded classrooms. There are a lot of possible solution to people counting, these technologies are commonly known as people counters, however most of them are based on sensors on the entrances of a room or a building, like infrared beams, thermal sensors or computer vision systems. All these methods are based on counting the people that enter or exit from a specific door, they have the shortcoming that a sensor for each door is needed and that they can't count instantly the people in the room without being active from the first to the last person that enters.

The target task that is faced in this thesis is the development of a software that count the people from a single digital camera that points to the entire room, or at the most on two cameras that points to the room in case of too wide classrooms. To count the people from a digital image the only way is to use an algorithm of object detection, that aims at finding all the people, and then an algorithm to count the distinct detections found.

In the next section will be discussed the problem of object detection in general and in particular will be presented the deep learning methods to solve it, however that doesn't solve completely the problem. One of the main issues in counting people in crowded rooms is that the number of detections that should be performed on the single image is way higher than the usual number of detections that a neural network is trained to find. This problem is discussed in the last section of this chapter.

In order to improve the performances of the model some computer vision techniques are implemented; their main purpose is to account for the possibility of bad or different installation of the cameras with respect to the expected. These techniques face the problem of camera's autocalibration,

however they are not part of the main detection model that solve the people counting and will be discussed in the last chapter after the full definition and construction of the detection model.

4.1 Object Detection

Definition 4.1. The *Object Detection* is one of the main subfields of recognition in computer vision, this computer technology aims at detecting multiple instances of semantic objects belonging to a predefined list of classes in digital images or videos.

Multiple typical problems in computer vision are related or part of object detection:

- People detection.
- Vehicles and pedestrian tracking
- Segmentation
- Face detection

As explained in the introduction the target of this project is not a standard problem of object detection, however people counting from camera images falls into the problems related to it. In fact, the software is based on an object detection technique, specifically a neural network properly trained to detect people.

The object detection algorithms could be divided into two main groups, depending on the techniques they rely on:

- Machine learning-based algorithms: these techniques rely on hand-designed features, usually simple shapes like rectangle, edges or lines. First in each image the features are found. Then the objects are found using classical machine learning algorithms like SVM trained on a proper dataset to detect the objects using the hand designed features found as data.
- Deep learning-based algorithms: are more recent algorithms, they don't need hand designed features, the features are autonomously constructed by neural network during the training and on that basis the different objects are then detected during inference.

In 2012 for the first time a neural network won the largest object recognition competition (ImageNet Large-Scale Visual Recognition Challenge or ILSVRC), from that time until now all the best classification or object detection algorithms are deep learning-based.

As we have seen in the previous sections, convolutional neural networks are really effective in performing classification. The object detection is quite a different task with respect to a simple classification and more refined neural networks are needed.

In the classification the only need is to give the right class to the whole image. In the object detection a lot more information is needed, usually there are multiple objects that should be spotted in a single image, for this reason the output should contain for each detection both the class and the location, expressed in the form of a bounding box that contains the object.

Even if object detection is a more complicated task the ability of classifying an image is needed, for this reason the neural networks constructed for object detection have similar structure to the standard convolutional networks, actually in some algorithms an entire classification network is applied to predefined regions in the image or is only modified in the last layers.

There are principally three models to perform object detection using deep neural networks, they also correspond to three different paradigms to exploit the potential of convolutional neural network:

- Region Proposal (R-CNN, fast R-CNN, faster R-CNN) [14]
- You Only Look Once (YOLO) [15]
- Single-Shot multibox Detector (SSD) [16]

These methods represent three different approaches and specifically: YOLO rely on a brand new network structure that directly perform object detection having the multiple detections and bounding boxes as output, Region Proposal is based on creating a set of regions on which a standard CNN is applied, SSD uses a CNN as feature extractor and has a new structure substituted to the last layers to find multiple detections. To have an overview of these methods they are presented in more details.

4.1.1 R-CNN

R-CNN stands for Region proposal Convolutional Neural Network, as a matter of fact under this name three different methods have been developed and they utilize a classification CNN together with an algorithm that computes

a proposal for interesting regions to search for the different objects in the image.

the first method created is R-CNN, it simply utilizes a selective search algorithm to define roughly 2000 regions and on each of them is applied the convolutional neural network. The selective search algorithm is based on hierarchical segmentation constructing the region proposal with a bottom-up approach, it can be described as follows:

1. Do an oversegmentation of the image based on pixel intensity (divide the image in zones of same colour).
2. add all the bounding boxes corresponding to segmented regions to the list of regional proposal
3. using a greedy algorithm combine adjacent regions according to their similarity.
4. repeat from point 2 until the regions reach a stopping criterion (on dimension or number)

This method is quite effective, the real concern with R-CNN is that it creates too much regions, and for this reason almost 2000 inferences of the CNN are needed and both training and testing are too slow.

The same author solved some of the drawbacks of R-CNN, the new algorithm has been called Fast R-CNN[17], the basic idea is the same as R-CNN, but this time not every proposed region is given to the CNN as input. Instead the whole image is given to the CNN (not using it until the end) generating a convolutional feature map, then from the feature map the proposed regions are recognised, they are reshaped and given to the fully connected layers such that they are classified.

This algorithm is clearly faster because most of the convolutional network is applied only one time on the whole image, however with this approach the bottleneck becomes the selective search algorithm. It is both the slowest part of Fast R-CNN and the one that can cause errors because it is a static algorithm, that means that it doesn't learn from data and in some cases, it could give bad region proposals.

Subsequently a third method has been proposed, this time in order to get rid of the selective search algorithm. This third method is called Faster R-CNN[18], also in this case the whole image is given to the convolutional neural network in order to compute the convolutional feature map, but selective search is not applied, this time a separate network is used to identify the region proposals, then the regions of the feature map are reshaped and given to the fully connected layers to compute the outputs.

This is way faster than selective search, especially on test-time when the network that does the region proposal is already trained, furthermore the possibility to learn the regions gives more flexibility and the possibility to safely apply the method to any kind of dataset.

4.1.2 YOLO

You Only Look Once (YOLO) is an object detection algorithm based on a single convolutional neural network, it doesn't use external algorithms or second network to find the regions, but it is specifically constructed from the start to the end to perform multiple object detections on a single image. The output of YOLO in fact is composed by a list of detections and for each of them the class probabilities and the bounding box coordinates.

The input image is divided into a grid of $S \times S$ cells, for each object in the image the cell that contains its centre is the cell that will be responsible for detecting it.

In each cell the network is constructed to predict B bounding boxes and the C class probabilities (not one for each bounding box only for the entire cell).

For each of the bounding box the prediction contains 5 values:

- the four values that define the bounding box position, they are the coordinates of the centre and the relative value of height and width with respect to the cell dimension.
- A numerical value called confidence, it is defined as: $Pr(\exists\ Object) * IoU(pred, truth)$

Definition 4.2. The Intersection Over Union (IoU) is a statistic used to measure the similarity or diversity of sample sets. It is fundamental for object detection, because it is used as a metric for distance between estimated bounding box and real ones. It is computed as:

$$IoU(pred, truth) = \frac{Area\,of\,intersection\,between\,predicted\,and\,true\,boxes}{Area\,of\,union\,between\,predicted\,and\,true\,boxes} \tag{4.1}$$

For each of the C classes is also computed the conditional class probability:

$$Pr(Class(i)\,|\,Object) = Probability\,of\,having\,object\,of\,class\,i,\ given\,the\,grid\,containing\,one\,object. \tag{4.2}$$

The YOLO network is similar to a convolutional neural network, but the output of the last fully connected layer is not only the list of the class probabilities as usual, instead it is composed by the $S * S * (B * 5 + C)$ values described. S is the number of cell grid's length and height, B is the number of boxes detected in each cell, these are hyperparameters of YOLO structure, while C is given by the specific problem because it is the number of different class that must be detected.

The most interesting part of YOLO network, to understand its functioning, is given by the *loss function*, it is the most complex part because has to account for a lot of different factors. The loss function is composed by four different parts that are briefly described separately in order to understand how the network evaluate the output and what are the errors that the network try to optimize and reduce.

1. The first component of the loss function considers for each predicted bounding box the distance from the real bounding box:

$$\lambda_{coord} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{obj} (x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2 \tag{4.3}$$

2. The second part takes into account the difference in terms of dimension between the predicted and the corresponding real bounding box:

$$\lambda_{coord} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{obj} (\sqrt{w_i} - \sqrt{\hat{w}_i})^2 + (\sqrt{h_i} - \sqrt{\hat{h}_i})^2 \tag{4.4}$$

3. The third factor considers for each predicted bounding box the difference between the estimated value of the confidence and the real one:

$$\sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{obj} (C_i - \hat{C}_i)^2 + \lambda_{noobj} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{noobj} (C_i - \hat{C}_i)^2 \tag{4.5}$$

4. The last component is the standard classification loss but computed conditionally on the presence of an object in each cell:

$$\sum_{i=0}^{S^2} \mathbb{1}_{i}^{obj} \sum_{c \in classes} (p_i(c) - \hat{p}_i(c))^2 \tag{4.6}$$

The summation of these four different elements results in the fact that the network tries both to find the right objects, to avoid false positives and

also to build the best bounding boxes in terms of dimensions and position. The relative weight of these four different components can be modified using the two constant parameters λ_{coord} e λ_{noobj}.

Different version of YOLO exists and even some smaller and faster version denoted as Tiny YOLO, however these will not be described because YOLO will not be used in the solution found, in fact the network used is based on SSD that is described in detail in the next section.

4.1.3 SSD

The paper about *SSD: Single Shot Multibox Detector* was released at the end of 2016, it reaches new records in terms of performance and precision on classic object detection dataset like *PascalVOC* and *COCO* scoring over 74% mAP (mean average precision). To understand the principal features of this detector let's see the meaning of its name:

- *Single Shot*: it means that the object localization and classification are done in a single forward pass of the network, this is similar to the approach of YOLO, while the region proposal methods need two shots, one to generate region proposal and one to detect the objects.
- *Multibox*: this is the name of a previous algorithm to find the bounding box proposals developed by *Szegedy*[19].
- *Detector*: is simply referred to the fact that the network is an object detector, that localize and classify objects.

SSD consists in a feed-forward convolutional neural network that produces a fixed-size collection of bounding boxes and scores, this is followed by a non-maximum suppression step in order to reduce the number of detection and produce the final detections. The difference with respect to previous single shot methods like YOLO is that they operated on single scale feature map, while SSD tries to find detection from feature maps at different scales as can be seen in image4.1.

The starting feature extractor of SSD architecture could in principle be any CNN, in the original paper the base network is VGG-16, however a lot of implementation of SSD with other networks exist nowadays. The base network is obviously truncated before the classification layers (fully connected), some feature maps of the base network can already be used to perform detection and that is the case of the original paper that uses also an intermediate layer of base network. After the VGG architecture other convolutional layer are added to progressively reduce the dimension of the feature map and after each reduction the feature map is also used

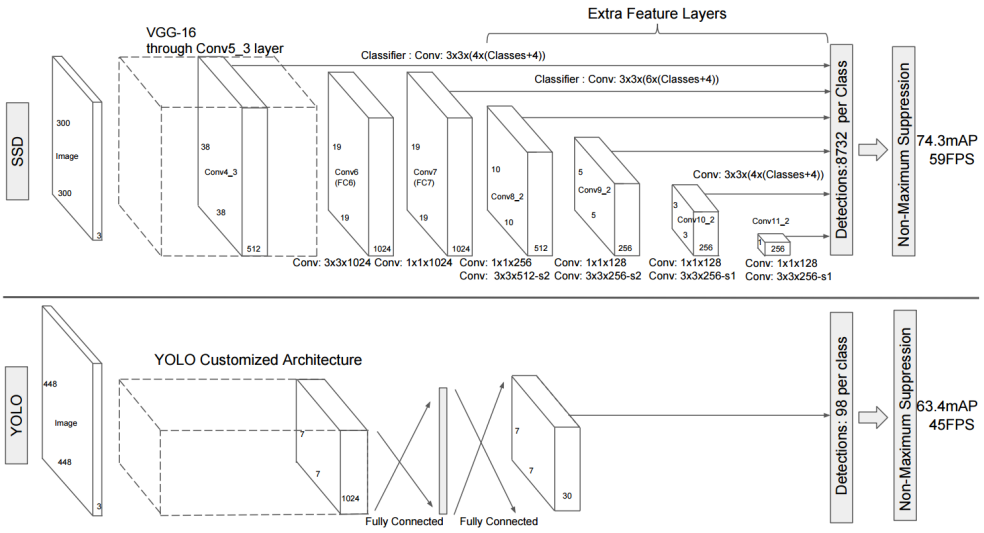


Figure 4.1: Comparison of SSD and YOLO architecture (from [16])

for detection, this process is repeated until the last feature map has the dimension of a single point.

This process of feature extraction could seem both overcomplicated and computationally heavy, instead it gives great performance because it consists only in adding some convolutional layers to already deep networks. This process has also a precise motivation, as has already been explained proceeding through the layers of a CNN the feature extracted are bigger, in fact each single value of the feature map is computed on the base of a region of pixels of the original image, in particular smaller is the dimension of a feature map bigger is the section of image represented by each value in the map. The effect of this method is that the first layers used for detection are great in finding small objects and moving on the feature map are ideal to detect bigger and bigger object until the last single value map that is suited to detect object that occupy the majority of the image.

Now let explain how each single feature map is used to extract the detections, this is where the multibox algorithm is used. The algorithm consists in three main steps:

- The starting point is a feature map of dimension $m \times n \times p$ that can come from different point of the feature extraction part, on this feature layer a convolution with $3 \times 3 \times p$ kernel is applied.
- For each of the $m * n$ values we got k fixed bounding boxes that have different sizes and aspect ratios to fit different type of objects.
- For each of the bounding boxes are computed the c class scores, one for

each class, and are also returned the 4 values to identify the bounding box location and shape. This gives an output for each feature map of $m * n * k * (c + 4)$ values.

For example in the original paper the first feature map has dimension of $38 \times 38 \times 512$, a 3×3 kernel is applied and 4 bounding box are used at each location, considering Pascal VOC dataset that has 21 classes means that the output of that feature map will be composed by $38*38*4*(21+4) = 144400$ values that corresponds to $38 * 38 * 4 = 5776$ different bounding boxes. Similarly, for the other layers this gives a total of 8732 bounding boxes in total compared to the 98 boxes computed by YOLO architecture.

In order to perform the training with this huge number of bounding boxes an important operation is the assignation of each ground truth box to a specific default box. The first idea could be of simply assign each true box to the fixed output box that maximize their intersection over union, that is the idea used in Multibox algorithm. However, in this way the training is difficult because there are a lot of false negatives because a lot of different boxes can represent well the true one, for this reason the true box is assigned to all the ones with an intersection over union grater that 0.5.

The SSD training objective is composed by two main factors, the *localization loss* L_{loc} and the *confidence loss* L_{conf}. The overall loss is a weighted sum of localization and confidence losses:

$$L(x, c, l, g) = \frac{1}{N}(L_{conf}(x, c) + \alpha L_{loc}(x, l, g)) \tag{4.7}$$

Where N is the number of default boxes, l represent the predicted box, g represent the ground truth box. Let $x_{ij}^{p} = \{1,0\}$ be an indicator function for the matching, such that it is equal to 1 if the i-th default box has the j-th ground true box assigned of category p and equal to 0 otherwise.

The localization loss is computed as a Smooth L1 loss between the predicted box and the ground truth box, also in this case the bounding box are represented by four parameters, the centre coordinates (cx, cy), the box width (w and the box height (h). The values of the ground truth box are scaled using the default box dimensions (D), in order to give the same importance to boxes of different dimensions and shapes. The localization loss is computed only on default boxes that has an assigned detection ($i \in Pos$).

$$
\begin{aligned}
L_{loc}(x,l,g) &= \sum_{i\in Pos}^{N} \sum_{m\in\{cx,cy,w,h\}} x_{ij}^{p} smooth_{L1}(l_i^m - \hat{g}_j^m). \\
\hat{g}_j^{cx} &= (g_j^{cx} - d_i^{cx})/d_i^{cx}) \quad , \quad \hat{g}_j^{cy} = (g_j^{cy} - d_i^{cy})/d_i^{cy}). \\
\hat{g}_j^{w} &= \log\left(\frac{g_j^w}{d_i^w}\right) \quad , \quad \hat{g}_j^{h} = \log\left(\frac{g_j^h}{d_i^h}\right)
\end{aligned}
\tag{4.8}
$$

Where the smooth L1 loss is given by the formula:

$$
smooth_{L1}(x) = \begin{cases} 0.5x^2 & if \quad |x| < 1 \\ |x| - 0.5 & if \quad |x| \geq 0 \end{cases}
\tag{4.9}
$$

The confidence loss is computed as the softmax loss over each of the c different classes:

$$
\begin{aligned}
L_{conf}(x,c) &= -\sum_{i\in Pos}^{N} x_{i,j}^{p} \log(\hat{c}_i^p) - \sum_{i\in Neg}^{N} x_{i,j}^{p} \log(\hat{c}_i^0) \\
\hat{c}_i^p &= \frac{exp(c_i^p)}{\sum_p exp(c_i^p)}
\end{aligned}
\tag{4.10}
$$

Another important choice that has to be made is the dimension and shape of the boxes in each feature map, for what concern the dimension is represented by the scale s_k of the boxes of k-th feature layer, the values for each scale are assigned as regularly spaced values from 0.2 to 0.9. To decide the shape of the boxes is chosen the ratio between height and width, in fact given the ratios $a_r \in \{1, 2, 3, \frac{1}{2}, \frac{1}{3}\}$ the height and width are computed as:

$$
w_k^a = s_k\sqrt{a_r} \quad , \quad w_k^a = \frac{s_k}{\sqrt{a_r}}
\tag{4.11}
$$

The only exception is given by the ratio equal to 1 that gives a second smaller box computed with $s'_k = \sqrt{s_k s_{k+1}}$, resulting in the six default boxes used in most of the layers.

One of the drawback of having this much default boxes is that usually the number of negative boxes is by far higher than the positive ones, this can create problem of unbalancing in the loss computation, for this reason the negatives are sorted by highest confidence loss and only the first are maintained in order to have no more than 3 negatives for each positive box.

Nowadays have been found object detection network with accuracies better than SSD, however if the objective is a compromise between the accuracy and the efficiency of the network SSD or some newest version of YOLO are

the best choice available, one of the advantages of SSD compared to YOLO is that can be used with different base networks including network suitable for edge computing like mobilenets or inception V2. For this reason, the best choice according to both accuracy and efficiency in our problem has been using the SSD method with Inception V2 as feature extractor that will be described better in the next section.

4.2 Feature extractor used: Inception V2

We have seen that using the Single Shot Multibox Detector is needed a convolutional neural network as base network. As always, the choice is done considering both accuracy and inference speed. Is visible from the benchmarks that the fastest base networks are the mobilenets, however the difference with respect to inception V2 are not enough to represent a better choice, in fact mobilenets are less accurate and less powerful, while network bigger and deeper than inception V2 on the Jetson Nano are almost impossible to be deployed, not only because of the inference speed but principally because they can easily create failures because you ran out of memory.

For this reason, having the confirm given by a good benchmark result, the chosen network for training is *Inception V2*, it is part of the inception networks family, also named GoogleNets, that is composed by principally five popular version developed in three different times:

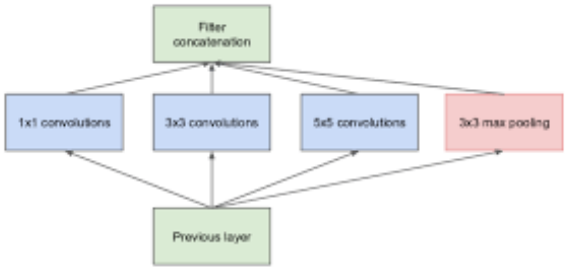
- *Inception V1* or *GoogLeNet*: it has been presented in 2014, in a paper by Google team [20]. This network was one of the first to consider the efficiency of the network, in fact it won the 2014 ILSVRC competition using 12x fewer parameter than previous winner.

- *Inception V2* and *Inception V3*: the improvements of these network are presented in two different papers, the first[21] that presented the batch normalization method and the second one[22] that presented a series of techniques to improves efficiency of convolutional layer in order to increase the networks' depth. In the paper a lot of different techniques are presented and the differences between these two architectures usually depends on the single implementation, however they rely on the same ideas, usually inception v3 has a bigger input size and some more layers.

- *Inceptionv4* and *Inception ResNet*: they are introduced in the same paper[23], inception v4 is a bigger and refined version of the previous network, while inception resnet is the result of using both the inception modules and the residual connection[24].

In order to understand the structure of inception v2 is important to understand the idea at the root of these networks, the *inception module*. Until 2014 the main method to increase accuracy was building deeper networks stacking convolutional and pooling layers on top of each other, obviously this method is constrained both from overfitting and from computational cost and couldn't be the solution.

The premise is that the part of the image that have to be recognised can have large variation in size and finding a kernel size suited for different situation is difficult, for this reason the main idea behind inception module is to find a good local block that is computationally efficient and address the problems for different sizes. The result is given by substituting a single big convolutional layer with multiple convolution of different kernel sizes that operates at the same level, their outputs are then concatenated to be sent to next inception module.



(a) Inception module, naïve version

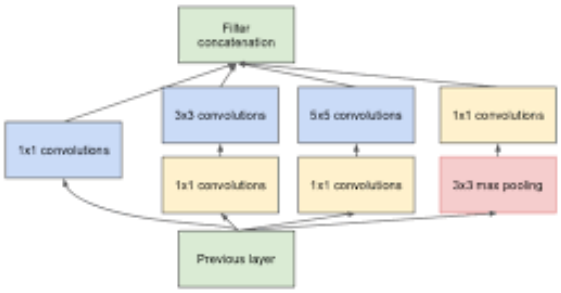
Figure 4.2: First version of inception module developed in GoogLeNet (from [20])

As can be seen in figure 4.2, the solution has been to simply take away layer with big kernels and using instead a module that is composed by three different convolutional layers respectively with kernel of 1×1, 3×3 and 5×5 and by a simple pooling layer.

This naive version of the inception module has a big drawback, in fact the pooling layer preserves the features depth and adding the result of the three convolutions result in a major increase in feature depth at each inception module, this also means a huge number of operations.

The solution has been to add layers of 1×1 convolutions that preserves the height and width of the features but can reduce the depth, these layers has been added to all the layers in the inception module resulting in the module visible in the figure 4.3 that is the one concretely used in GoogLeNet.

The resulting network has been a network deep 22 layers (27 counting the pooling layers) composed by 9 inception modules stacked linearly in addition to the initial layers (called stem) and the last classification layers. The result



(b) Inception module with dimension reductions

Figure 4.3: Inception module developed in GoogLeNet with dimensionality reduction (from [20])

is a pretty deep network and for this reason it suffers of vanishing gradient problem, for this reason another innovation introduced is the addition of two auxiliary classifier in the middle part of the network. These are structured exactly like the last classifier and are used only for training, their loss is weighted in the total loss computation and has a weight smaller the last real loss.

The later version *Inception V2* is the result of two main improvements. The first improvement is the addition of *batch normalization* layers, these techniques has already been explained in section 2.3.4, its addition improves the training. The second main improvement is the *factorization of the bigger convolutions*, the idea is that the bigger kernels can be substituted with more smaller kernels without changing the result but reducing the operations needed.

The first idea is to factorize into smaller convolutions, as can be seen in figure 4.4 one $5x5$ kernel is exactly equivalent to two consecutives $3x3$ kernels, but if it is equivalent why perform this substitution? The answer is trivial, two 3×3 filters have $(9 + 9) * d = 18d$ weights with d representing the depth, one 5×5 kernel has $25d$ weights, that result in a relative gain of 28%.

The second idea is factorize $n \times n$ filters in two consecutive asymmetric filters respectively $n \times 1$ and $1 \times n$ as in figure 4.5, also in this case the result is the same but the number of weights is reduce from n^2 to $2n$, that result in a relative gain that goes from 33% considering 3×3 filters up to $\sim 71\%$ for 7×7 filters (usually no bigger filters are used and factorized).

In figure 4.6 is visible how these ideas change the architecture of the inception module, these great gain in terms of computational cost gives the possibility to increase the network depth, for this reason inception v2 and inception v3 are respectively 42 and 48 layers deep.

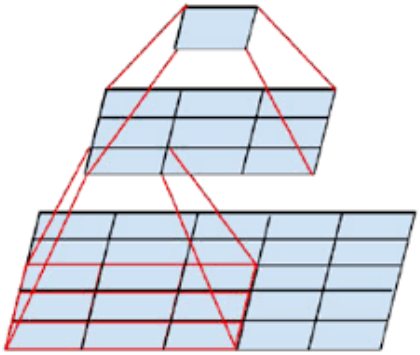


Figure 4.4: Factorization of bigger filters with $3x3$ filters (from [22])

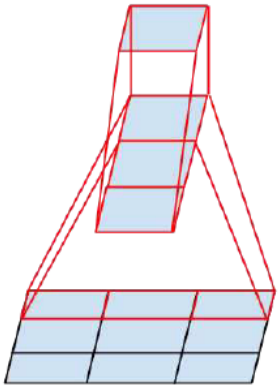


Figure 4.5: Factorization of nxn filters with $nx1$ and $1xn$ filters (from [22])

These are the main techniques introduced in the development of inception v2, the result is a network that is deep but maintains a reasonable number of parameters and can be applied without huge computational power. For this reason, it is a good choice for the base network to use SSD object detection method.

4.3 Manage box quantity

After the choice of the neural network the model definition is not done, we will see in the next chapter that is fundamental the network training procedure, in fact none of the available networks is already trained to detect only people in the context of a room.

However, this is not the only problem in our environment, one of the main problems that have to be tackled is that the number of people that should be detected effectively is far higher than the number of detections

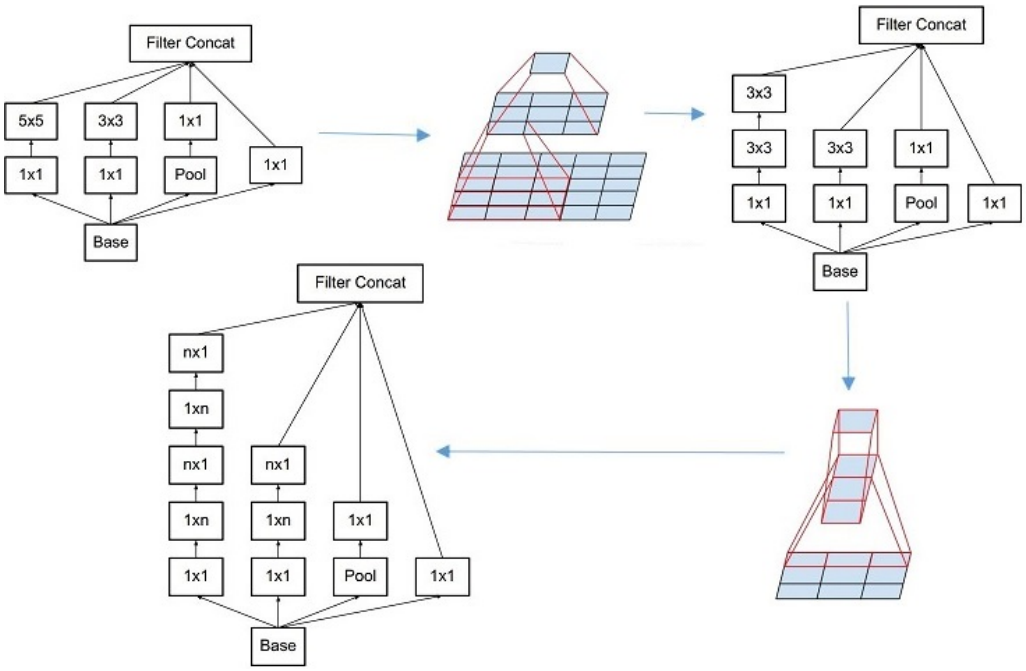


Figure 4.6: Modification of the inception module using factorization techniques on convolutions (from [22])

that any network is capable to find. In fact a network trained to find dogs on a dataset that has 5 dogs per image on average won't be able to detect all the dogs on a test set that has more than twenty dogs per image and this applies on any network for object detection.

One idea to solve this problem could be training from scratch the network on a suited dataset similar to the test images, however completely train from scratch a deep neural network is a long and expensive task and need to much images to be performed. Obviously a finetuning on a good dataset could reduce the problem but should be considered that having an image with ~ 50 or even ~ 100 bounding box they will result in very small portion of original image and is unlikely that the network will be able to detect all of them.

In our particular case the chosen network SSD inception v2 receive in input images of 300x300 pixels, a good camera (not excessively) takes frames with 1080p (usually $1920x1080$) or QHD (usually $2560x1440$), obviously the hypothesis of resizing the image have to be discarded without even thinking about it, most of the information would be lost during the resizing. A standard cropping is also not suitable is important to count all the people present in the image and the risk of cropping away a person near a border is high.

The only solution to maintain both the quality and the entire image is to

divide it in a lot of different sections (crops) and count the people in each section separately summing the results afterwards. The idea is to maintain the whole resolution of the camera image, and depending on its resolution will change only the number of different crops that have to be processed by the neural network, in this way we have two advantages:

- The network receive as input images that has the exact dimension of the network input, in this way no resizing is performed and all the information contained in the input are maintained.
- The number of bounding boxes in each input crop is not excessive and with crowded rooms should be at most around 15 or 20 boxes but usually it should stay around 5 to 10 bounding box that is similar to a good dataset used to pretrain an object detection network.

This approach creates also some concerns that has to be faced. The first one regards the efficiency of the algorithm, with this approach in fact to have a single count of the people in the original image have to be performed multiple inferences of the neural network. This is one of the reasons of the attention towards the efficiency in the network's choice. In particular considering crops of $300x300$ depending on the input resolution and some other parameters the number of inferences needed can go from around 50 up to even 200, not even considering the use with 4K cameras.

Another issue that has to be faced is to decide how the input image should be divided in sections. The cropping of the input image should be done without any knowledge on the bounding box position, for this reason taking completely distinct crop have the result to cut a lot of people between two crops, obviously the network has higher probability to fail the count having to count some people that are not completely visible.

To solve this problem, I decided to consider a margin of overlapping between the different crops, in this way (with a good margin value) all the people should be completely visible in at least one crop. The shortcoming of this approach is that increase the number of crops and above all that creates a lot of duplicates in the detections.

4.3.1 Duplicates elimination

The problem of the elimination of the duplicates is not trivial as it can seems, the main reason is that in crowded rooms is likely to have a lot of detections that are slightly overlapped. At the same time, we have no guarantee that a person that is partially or completely inside the margin will be detected in both the crops, often even if it is detected in both the crops the bounding box found could have a completely different dimension

and shape. Moreover, a person could be detected in up to four different crops if it is both in a lateral margin than in the above or below margin, substantially if it belongs to the corner.

These variety of different cases makes the duplicates elimination a problem that should not be underestimated, particularly since the final result will depend strongly on the network accuracy but even more on the correct counting of the network's detections.

The first processing of the network result is a simple filter to eliminate bad detection and boxes with strange dimensions. All the detections with a score below a threshold are eliminated, moreover the area of each bounding box is computed and all the bounding boxes with areas excessively low or high are eliminated, this approach is improved by the computation of the vanishing point in fact is considered also the perspective to obtain stronger constraints on the boxes area.

The first simple approach to eliminate the duplicates consists in checking each couple of boxes and computing if they are distinct, overlapped, or one is contained in the other one. the distinct boxes are obviously maintained, while for the contained boxes they are discarded. For the overlapped boxes is computed a metric to determine the entity of the overlap.

The standard metric used also to determine the quality of the predicted boxes is the IoU (intersection-over-union), however during the analysis of the results I found out that is not a good metric for many usual cases. In fact many times a person could be detected even if a small portion of its figure is present in the crop, comparing this detection with the detection in the crop that contains the entire person the intersection over union is typically small, while with two people sited sideways the intersection over union could be higher, for this reason it is impossible to find a threshold that satisfy both the cases. The solution is to consider a different metric suitable for this case, the idea is to consider the *intersection-over-minimum*, defined as the area of the intersection fraction the area of the smaller between the two boxes. This solution works better because in the case of two detection with different sizes of the same person its value is typical high and near 1, while in the case of two close people the value is obviously higher than IoU but typically remains under a reasonable threshold of 0.7.

This approach gives positive results but it completely ignores the information about the location of the bounding boxes in the correspondent crop. In order to exploit this information a more sophisticated approach has been developed. The first thing that should be noted is that with good values of overlapping between the crops any person that is detected at the border of a crop should be completely contained in the crop besides, and the second detection should be more accurate because it contains the entire person. For this reason, in the first filter also the box location is considered, all

the boxes that are against the border or distant a maximum of 5 pixels are checked, if their dimension is smaller than the overlap between crops, so they should be completely contained in the next crop, they are eliminated without further checks. In this way a lot of detection of a small portion of a person are already eliminated. After this first check the normal algorithm to eliminate duplicates is applied.

The result of this process can be seen in figures 5.11 and 5.12, shown in the next chapter to analyse the result of the trained neural network, these images display the bounding boxes detected by the trained model before and after the elimination procedure.

The managing of the bounding boxes is further improved using some computer vision techniques, that compose the process of camera autocalibration, in order to generalize the method and to speed up the inference. The idea of this improvement can be summarized explaining the three main objectives of the autocalibration:

- Improve the dimensional filter: one of the main defects of filtering the bounding box on the base of their dimension is that the bounding box location is not considered, and obviously the perspective can highly affect this value. The improvement consists in computing the vanishing point of the image and normalize the dimensions of the bounding box on that base to make them more comparable.
- Avoid the inference in useless section of the image: the camera could easily see part of the rooms like walls or the roof that will never contain a person to detect, for this reason is computed the optical flow of the room while the people are moving inside it, in this way the sections that don't contain any movement can be eliminated improving the inference speed.
- Make possible the use of multiple cameras: in big classroom a single camera could not be enough to frame the entire room, for this reason a technique to perform automatic matching between frames from different cameras is attempted.

These techniques are described in detail in the chapter 6 where their results are shown.

5 Chosen model training

After the choice of the neural network used in the model the next step is the training of the network. The training is the most important procedure in order to obtain an effective people counter. We have already explained the theory behind the training procedure, in particular the backpropagation algorithm and the optimization methods, however to implement it some good practices have to be taken into account.

The neural network used is taken from the open source library TensorFlow, as usual with object detection networks it is pretrained on the COCO dataset, that is a dataset containing instance of 80 classes of general type of objects. One of the classes is the *person* class, so in theory it is already able to detect people, however it gives really poor results and it is essential training it properly.

In all the different training performed the dataset has been divided between a training set and a validation set. The validation is performed simultaneously with the training procedure in order to understand how the procedure is progressing and to understand if the network start overfitting and is better to take an intermediate result in place of the last one.

Obviously to have a good training is important also the dataset preprocessing, the choice of the optimizer and the learning rate.

5.1 Performance measures for training

The effectiveness of the training is addressed through three main statistics, the classification loss, the localization loss and the mean average precision.

The first two loss are the ones already described and are computed simultaneously on training and validation sets, they are useful to understand if the training is improving network predictive power and to understand if problems of overfitting are occurring. In fact, the best way to find overfitting issues is to compare the losses on the training and the validation sets, when they start to diverge is usually better to stop the training or to avoid using the weights found in the following of the training.

The mean average precision is the most popular metric in measuring the accuracy of object detectors. It is the mean value of the average precisions on all the classes considered, in our case is the average precision on the *person* class. To explain how the average precision is computed we have to do a recap on the classical concepts of true and false positives and negatives in the object detection context and the formulas for *precision* and *recall*.

The true positives (TP) are the predicted bounding box that have an intersection over union higher than a threshold (usually 0.5 or 0.75) with a true bounding box of the same class. The false positives (FP) are the cases in which the predicted bounding doesn't correspond to any ground truth box, while the false negatives (FN) are the bounding box that correspond to a ground truth box but are predicted with the wrong class, the true positives are not computed in object detection because usually all the images in the datasets contains at least one object. With these values is possible to compute precision and recall.

- Precision: is the percentage of the prediction found that are correct.

$$Precision = \frac{TP}{TP + FP} = \frac{TP}{total\,predictions} \tag{5.1}$$

- Recall: is the percentage of object of the considered class in the image that are found.

$$Recall = \frac{TP}{TP + FN} = \frac{TP}{true\,objects\,in\,the\,image} \tag{5.2}$$

To compute average precision, we rank all the detection in decreasing order of confidence, and we compute the precision and recall at each step. The recall will obviously result in an increasing value, in fact it remains the same at each negative prediction and it increases at each positive prediction. The precision instead will have a zigzag pattern, it increase at each true prediction and decrease at each false prediction, usually the trend will however be decreasing because on average the true prediction will have higher values of confidence.

The idea behind average precision is to plot from these values the precision against recall value and to compute the area behind that curve, however usually the precision is smoothed to obtain a decreasing value considering at each step not the precision of that step but the maximum precision of the following steps. The average precision is then computed as the area under the resulting smoothed curve. The idea of the average precision is almost the same of the classical *area under the curve (AUC)* used in statistic models but its computation slightly changes also between a dataset to another due to the process of smoothing of the curve.

5.2 Efforts of training on public datasets

To improve the network predictive power the first attempts has been done using big public dataset available to retrain the network. The principal task faced including the people detection are usually:

- Face or people recognition: the dataset used in these cases are completely useless, the reason is that in this problem the people should be recognised one from another and the datasets usually contains few people that have to be linked to the correct identity of the single individual. We need a dataset that contains more detections and we are only interested in locating the people in an anonymous way.
- Pedestrian detection or tracking: These datasets usually contain a lot of instances more in each image and are usually aimed at detecting only the people without recognising them. For this reason, a dataset of this type is worth an attempt to understand if it can improve the efficiency of our pretrained network.
- General purpose datasets: most of the principal object detection dataset contains the class *person*, obviously they contain also other classes, but they can be eliminated in order to consider only people. The main problem here is that usually they contain between 5 and 10 instances per image considering all the classes, so this number is far lower considering only people.

I decided to do an attempt at using two public datasets to retrain the network. The first dataset used is *Microsoft COCO*, it is the most famous general purpose dataset for object detection and is the dataset on which the network is already trained, however the idea is that eliminating all the annotations relative to the classes that are not *person* the accuracy of the network could improve avoiding the risk of detecting a person as an object of some other class.

The second dataset utilized is the *EuroCity Persons (ECP)*, it is one of the biggest publicly available datasets regarding the pedestrian detection, it contains also some annotations for bicycles or motorcycles, but they represent a minor part of the total bounding box and can be easily eliminated in order to maintain only the pedestrian annotations.

5.2.1 Microsoft COCO

COCO (Common Objects in Common Context)[25] is a large-scale object detection, segmentation, and captioning dataset. We are interested only in

object detection and in particular only in the first class *person*. It contains more than 200.000 images and 80 object categories divided in the usual three sections:

- Training set: composed by 64115 images with 262465 total bounding boxes annotated, so an average of ~ 4.1 annotations per image.
- Validation set: composed by 2693 images with 11004 total bounding boxes annotated, so an average of ~ 4.1 annotations per image.
- Test set: n images for which obviously are not available the annotations, it is in fact used for an object detection competition, however we are not interested in this set because it is used as a dataset to pretrain our network.

Obviously, the dataset as it is useless because SSD inception v2 is already pretrained on it, for this reason all the instances of the other classes are eliminated and also the images that doesn't contains people are not considered. The network has been retrained on the resulting dataset that is slightly smaller than the original one.

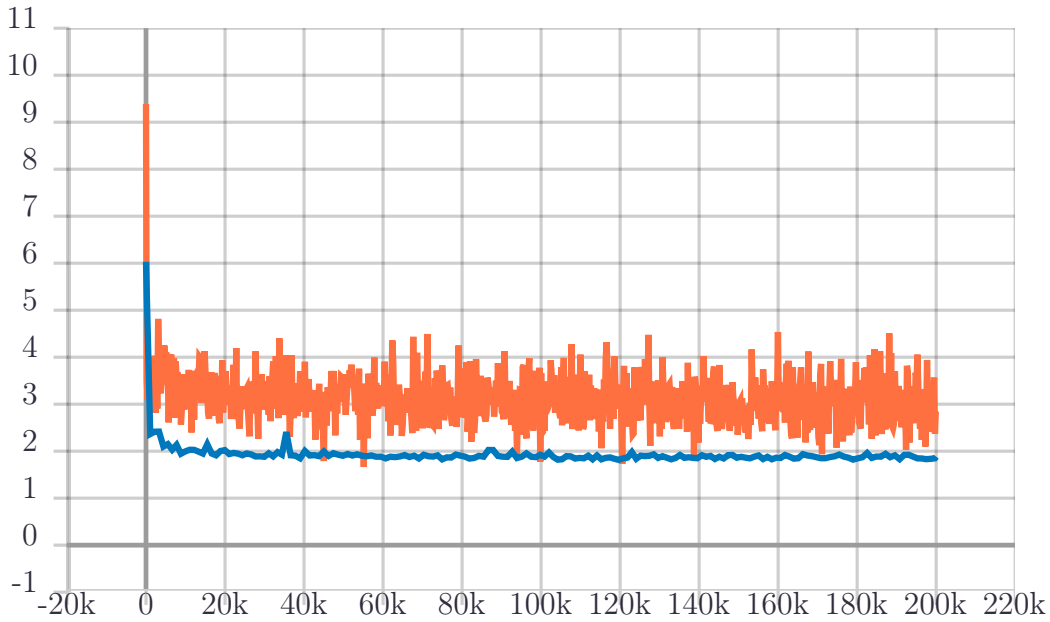


Figure 5.1: Classification loss on training and validation COCO dataset

From the images 5.1 and 5.2 can be observed that the network needs very few iterations to reach good values of the loss, this is expected because it is retrained on almost the same dataset, however the epochs' number is not

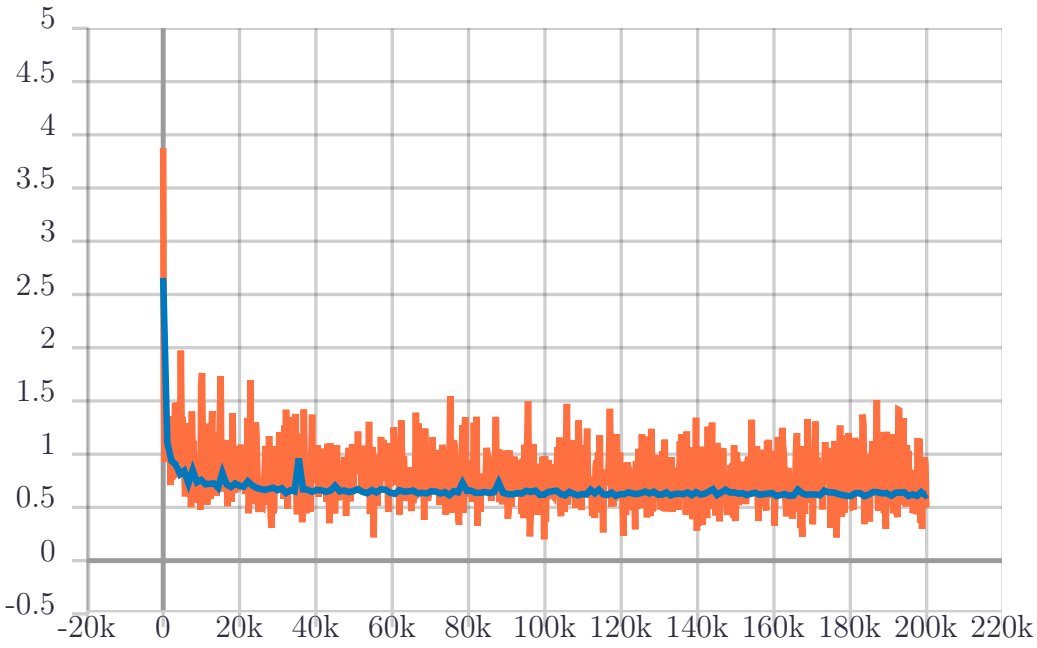


Figure 5.2: Localization loss on training and validation COCO dataset

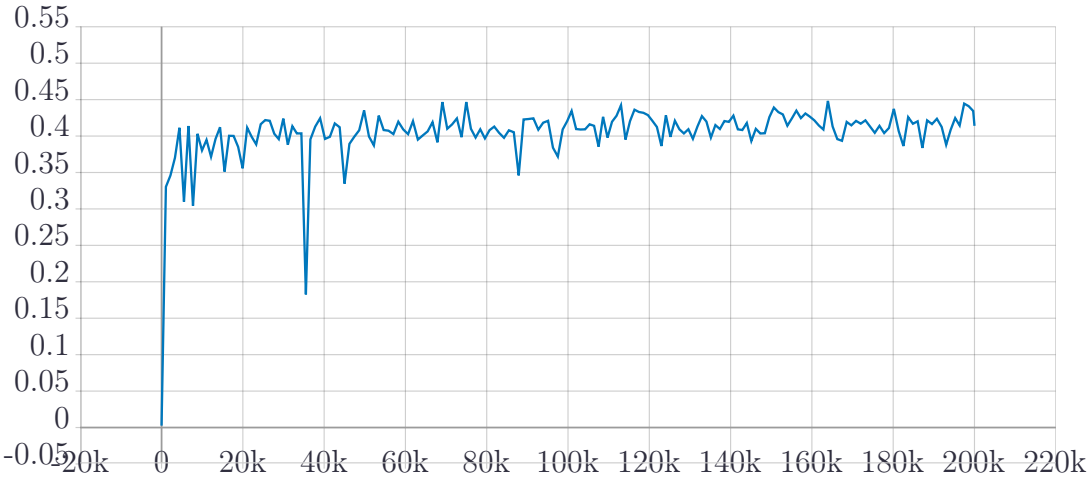


Figure 5.3: Mean average precision with $IoU > 0.5$ on validation COCO dataset

excessive in fact no overfitting is visible from these results. The accuracy on the validation set in image 5.3 is measured considering the mean average precision of bounding box that are correctly detected, and a bounding box is considered to be correctly identified if the intersection over union between the predicted and the ground truth box is more than 0.5.

The training can be considered as ended well and the values of accuracy found are the ones expected for this network on COCO (an idea on these

values can be found on Tensorflow detection model zoo), however the results are not better than the ones of the original pretrained network.

This suggest that probably the original training on the whole COCO dataset is effective and the network had few problems of false negatives with respect to the *person* class. This first attempt of improving the network effectiveness has not gone wrong but is neither particularly useful to improve the people detecting.

5.2.2 EuroCity Persons

ECP (EuroCity Persons) is a huge dataset of person instances in urban environment, it contains annotations both for persons and the vehicle in case of riders. The dataset further contains images both from day and night situations. As before it has been preprocessed in order to maintain only the useful information, for this reason all the annotations regarding the vehicles has been removed, furthermore the night dataset has not been used, because we are interested in detecting people in indoor situations and the illumination can be considered guaranteed.

The dataset contains $\sim$ 40000 images from 31 different cities all over Europe, it also the only dataset in this context that contains images taken during all the 4 seasons and with different weather condition. It contains $\sim$ 183000 annotations of pedestrian so almost 5 detections per image, that is a number far higher than COCO considering that we had removed all the classes except the people. In this case only the riders are eliminated but they are not counted in the number of annotations reported.

Also, in this case we tried to retrain the SSD inception v2 pretrained on COCO. Is important to note that the images in ECP are bigger than the ones in COCO and also with respect to the expected input of our network that is 300x300. For this reason, an approach similar to the one explained in section 4.3 has been used, the images are cropped in smaller images but this time we considered the position of the bounding box. Considering a training dataset we know the location of each bounding box, for this reason instead of considering an overlapping margin the idea is to cut the crops as soon as no bounding box is not cut.

As can be seen from the plots the training is effective also in this case, the process is a bit slower than the previous training on COCO but it is a very different dataset, so it is expected. In this case is visible from the classification loss than after 40000 iteration the network slowly starts to suffer from overfitting, in fact the loss on the validation set start to increase whereas the training loss continue to decrease.

In order to understand what could be a good stopping time for this training is useful to pay attention to the plot of the mean average precision

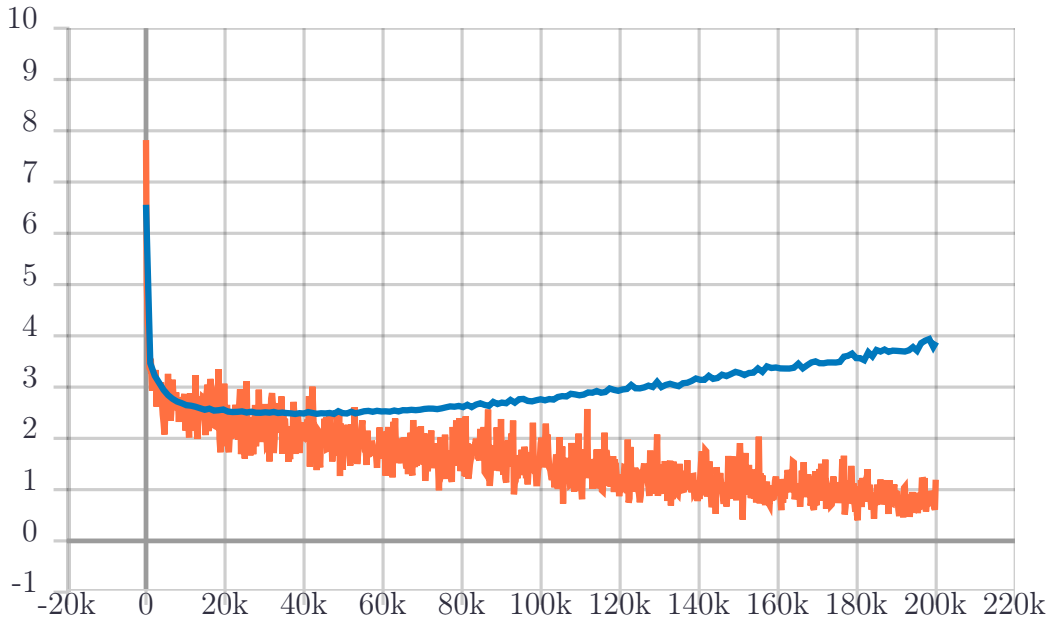


Figure 5.4: Classification loss on training and validation ECP dataset

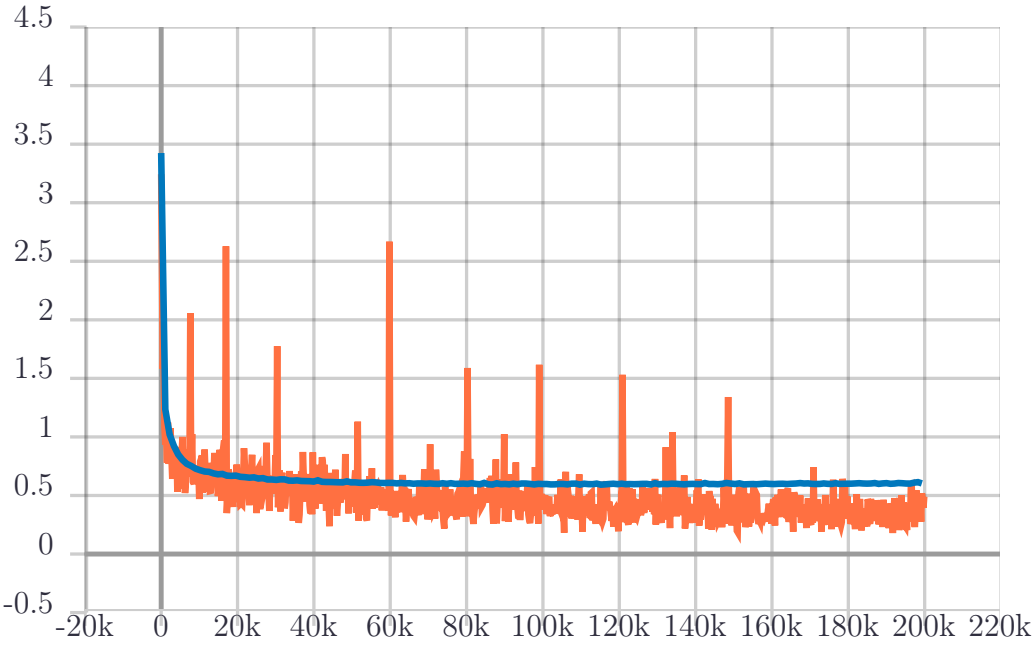


Figure 5.5: Localization loss on training and validation ECP dataset

on the validation set. The values of MaP reached are really good results and far higher than the ones on COCO, with the exceeding of 0.7 of average precision, furthermore can be seen that the overfitting didn't cause problems

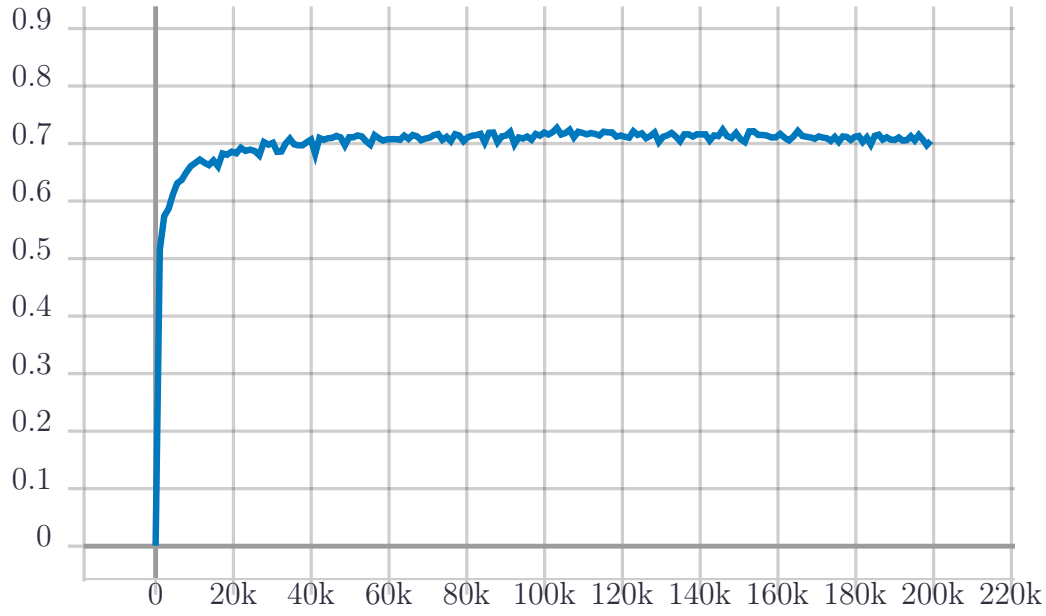


Figure 5.6: Mean average precision with $IoU > 0.5$ on validation ECP dataset

to the network's precision, however the result is almost unaffected from the last $100000 - 150000$ iterations of the training. A good range to search for the best trained network could be between the 40000 and the 60000 iteration.

To understand if the training on this dataset has been useful the network has to be tried on some test images from our target dataset, so it has to be tried on some image of people inside a classroom. Unfortunately, the result is worse than the starting network trained on COCO, but it is interesting to analyse the motivation of this result in order to understand better what could be the difficulties in the target task.

From the results is visible than one of the biggest problems is given by the shape of the bounding box, is clear in fact that most of the bounding box have a wrong vertical shape that create problems like taking two persons in the same box if they are one in front of the other. Thinking about the context of the ECP dataset the reason is straightforward, in the urban environment taking only pedestrian they are all in standing position, for this reason the network is trained to detect people from the head to the feet and is trained to detect shape that are always far more high than wide.

For the same reason the network has also problems in detecting people that are covered by the desks and for which almost only the head is visible. These problems make almost impossible to separate the different boxes and to remove the duplicates because a lot of bounding box contains more than

one person, the techniques used to eliminate duplicated detections from different crops will tend to eliminate this duplicates underestimating the number of people in the room, this effect create even worse result than the supposed one.

For this reason, can be excluded the idea to use any pedestrian dataset publicly available, however all the main public datasets regarding people are composed by people in urban context and hence in standing position. This means that the efforts in training the network on a public dataset before the last training on the target dataset has been in vain.

However these attempts are never useless and this training has been useful to understand some peculiarities in the problem of counting people in a classroom, furthermore it has given some insights on how the network train from the experience, even if the network was successfully trained to detect people it couldn't be able to generalize enough to detect and locate precisely people in new context, for this reason is important to have training dataset that precisely reflects the target task, and that possibly covers all the different possibilities in order to have a resulting network able to generalize.

5.3 Dataset creation and processing

After the attempt on improving the network through training on public datasets is time to perform the last and definitive training of the network on the target dataset. The dataset is composed by more than 100000 frames of high-resolution images taken from cameras in crowded classroom, privately obtained by Addfor. The images contain a number of people that ranges from empty rooms from really crowded rooms with almost 100 people to be detected.

Before starting the training, the first problem that should be addressed is that the dataset is not manually labeled and doesn't contains annotations contrary to the public datasets. The manual annotations of this much images unfortunately is a work that could take some months to be performed correctly and that is clearly a workload not suitable for this thesis work.

The best solution to get around this problem is to do the labeling work using the best possible neural network available. Obviously, this won't give a perfect result and could add some false negatives already in the dataset but should be a minor problem considering the dimension of the dataset and the high number of detections in each image.

The choice of the network has been straightforward, without giving importance to the inference speed I took the best available network pretrained on COCO dataset in order to obtain the best dataset possible. From Tensorflow detection model zoo can be seen that the best network available is

the *faster RCNN Nas network* that is a region-proposal based network of last generation. The faster rcnn method has already been explained, the base network in this case is NASNet[26], it a recent state-of-the-art network based on a particular idea coming from reinforcement learning.

The idea of NASNet is to have a basic structure of repeated convolutional and reduction layers, however the internal structure of this layer is not fixed at training time, in fact the network uses a recurrent neural network (RNN) that combines a defined set of operations to create the internal structure of a module, the possible operations principally are:

- Convolutions of different sizes: $7x1$ then $1x7$, $3x1$ then $1x3$, $1x1$ or $3x3$ standard convolutions, some variations of the previous like dilated and depthwise convolutions.
- Different types of pooling layers: $3x3$, $5x5$ or $7x7$ max pooling, $3x3$ average pooling.
- Identity layer.

the idea to create new blocks is to start from two simple hidden layers compose them together with these operations and create in this way a new hidden layer to be added to the hidden state space. These hidden blocks are used in the child network that computes the standard losses during training and gives the results also to the RNN that tries not to optimize the weights but the blocks' structure. This process is really complicated and need an enormous amount of computational power, it has been performed on 500 GPUs for more than four days to provide the first resulting network.

The resulting network is pretty big network and is by far the slowest network in TensorFlow detection model zoo, however to create the dataset all the effort is in having the best detections possible and the time needed to create it is secondary in this context because it has to be performed only one time.

After the choice of the network the usual problem of the dimension of the images should be tackled. This network can receive way bigger images as input, however trying different configurations I have seen that dividing each image in smaller crops result in a major improvement to the accuracy, this is due to the problem already mentioned regarding the number of detections. Using a network trained on COCO even if a such powerful network it won't be able to generalize at the point to detect all the people in image with more than 50 instances. For this reason, also in this case the images have been divided in crops to improve the network accuracy.

The dataset that has to be constructed should be divided in a training, a validation and a test set, for what concerns the training set and the validation set it is enough to divide the starting image in crops of the right

dimension, apply the faster-RCNN-Nas network to compute the bounding box and saving the crop as one of the training or validation images, in fact during the training is useless to reconstruct the images and crop them again, the SSD inception v2 network has to be trained on the single crops the cropping and reconstruction operations are preprocessing and postprocessing of the data, it is also needed the presence of a simultaneous validation during the training to control the risk of overfitting and to understand how to improve parameters like the learning rate.

The counting will be performed in the real application on camera, for this reason I decided to reconstruct the image for the test set, in this way the entire software can be fairly tested and all the operations can be improved using the test set.

The last operation performed to create the definitive dataset has been the elimination of most of the images without any person, only a really small number of them has been included in order to train the network to *not detect* some features of objects near the people, like chairs, as features belonging to people.

5.4 Training on private dataset

After the creation of the dataset the next step is the final training of the neural network. We have seen that the attempt of training on the ECP dataset has worsened the network's performances, for this reason the starting network is the SSD inception v2 trained on the modified version of COCO dataset containing only people. Probably starting from the original network pretrained on the whole COCO would have given the same results, in fact there is no noticeable difference between these two pretrained networks.

One of the most important choice for the training is the optimizer. The most famous gradient-based optimizers are described in section 2.3.2, we have seen that optimizers that implement a sort of *inertia* or *adaptive learning rate* are the best ones to avoid the stuck in local minima and trying to reach the global optimum. For this reason, the best choice for the optimizer is the *Adam optimizer* that combines both these approaches.

Even using an adaptive optimization method, the learning rate is still an important hyperparameter. At the beginning of the training the network needs a high learning rate for two reasons:

- To move faster towards the target and cause a shorter training.
- To avoid getting stuck in local minima

Proceeding in the training is better to decrease the learning rate value, in this way the trained network becomes more stable and the results are less variable from a step to another. Furthermore, towards the end of the training the network should stop to avoid the minima and start approaching a minimum that should be as close as possible to the global minimum, but a small learning rate is needed for that. The learning rate used, visible in figure 5.7, will be a stepwise descending function that will in total decrease by a factor of 10 assuming three different values.

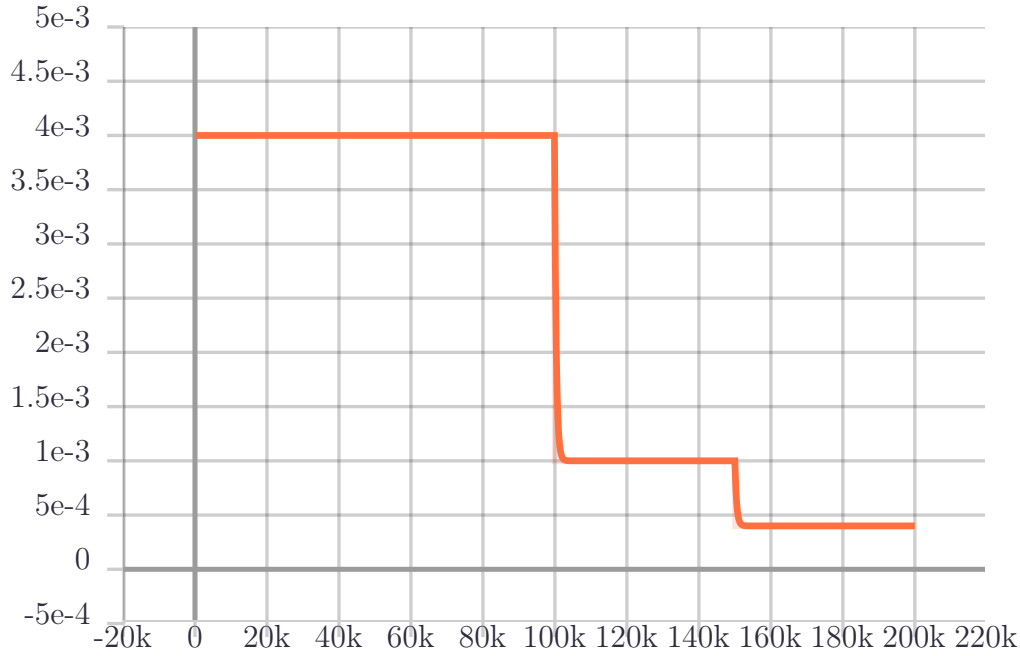


Figure 5.7: Learning rate for training on classroom dataset

The training is performed in 200000 steps and the validation is done simultaneously every 200 steps. From the resulting plot is clear that the training is incredibly effective, and the choices gave good results.

Analysing the plot of the two losses, in images 5.8 and 5.9 is visible that there is no sign of overfitting, the validation losses never diverge from the training losses and both have a huge and fast decrease in the first 10 thousands steps as always, however the training seems to be effective during all its duration. In fact, the losses have a slow but constant lowering during the entire training, the difference between training and validation loss is almost constant and small, this means that the training is effective not only for training data, instead the network is able to generalize and give good results also for the validation set.

An even greater result is given from the plot of the mean average precision

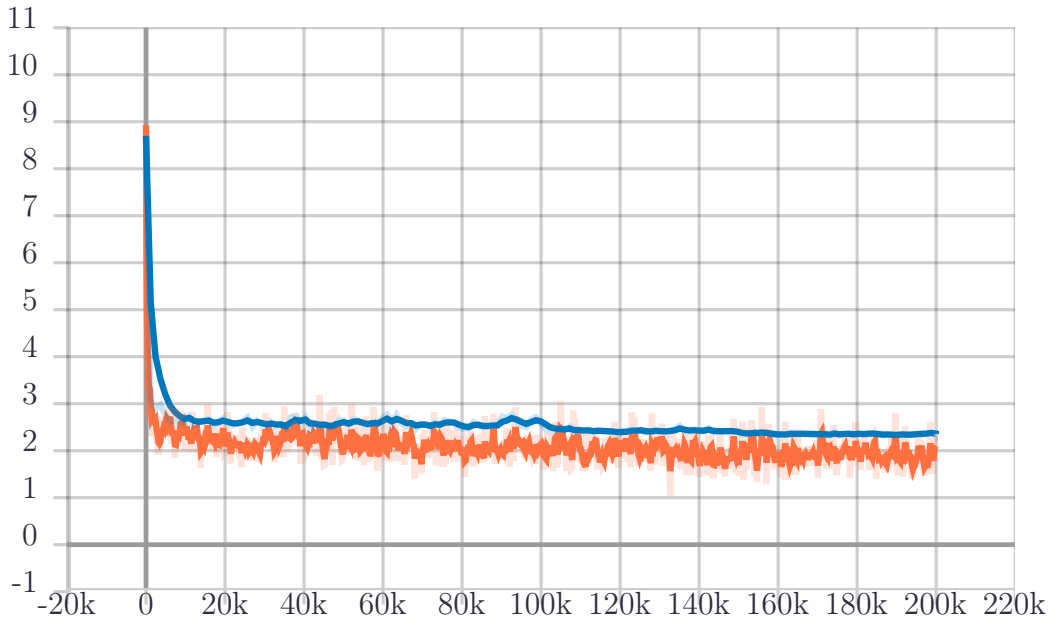


Figure 5.8: Classification loss on training and validation classroom dataset

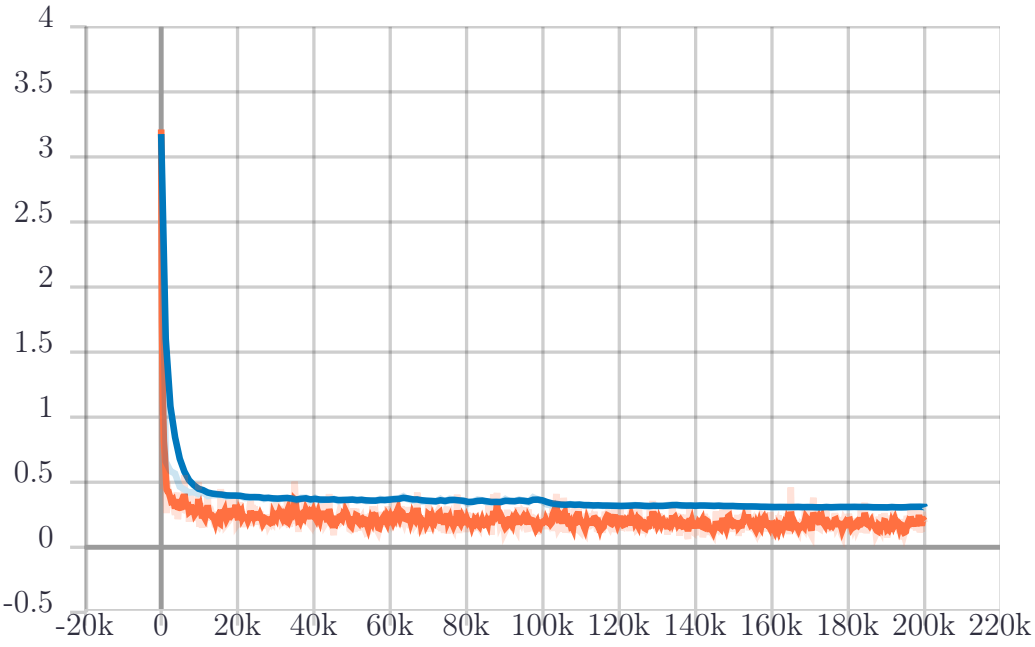


Figure 5.9: Localization loss on training and validation classroom dataset

(in image 5.10) on the validation set. After a fast starting increase the precision oscillates between 75% and 80% until the half of the training, After the half of the process the precision exceed the 80% and from that

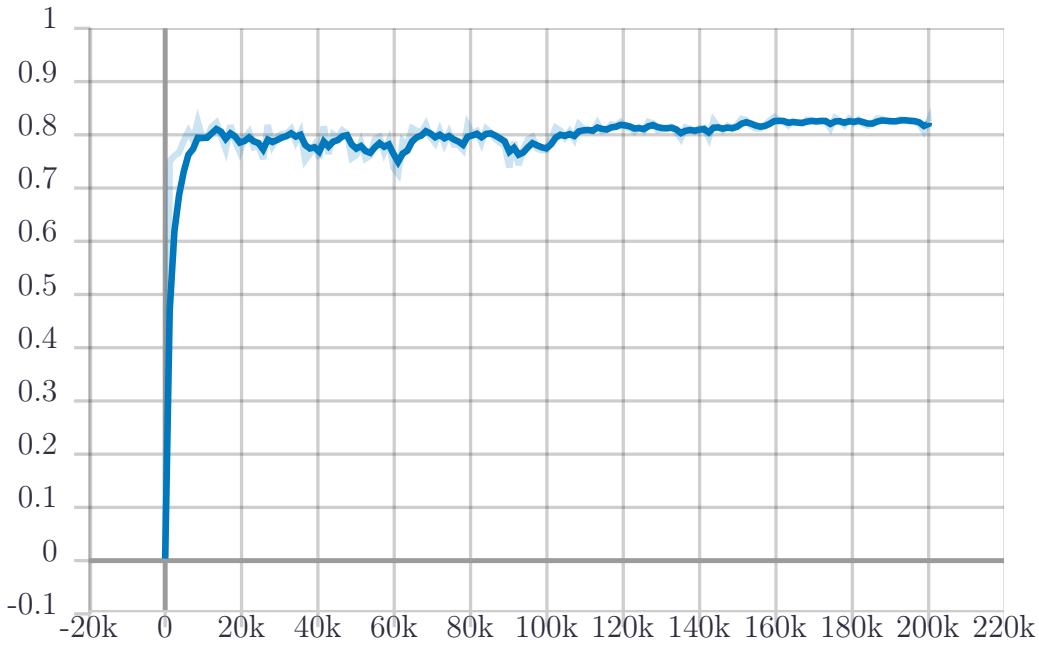


Figure 5.10: Mean average precision with $IoU > 0.5$ on validation classroom dataset

time on it stays constantly over that mark and becomes more stable. The reason is simple, the $100000 - th$ step is when the first decrease in the learning rate is performed, the consequences are clear, the network starts to give stable results and increase by some percentages point the accuracy. After the three quarter of the training (step 150000) the learning rate is lowered again, and the accuracy of the network became definitely stable around 82%. A mean average precision of this kind is a great result, on the modified COCO dataset the peak was at 45% and the result is almost doubled.

These results confirm the choices of the learning rate and the optimizer, the training can be considered as successful and the resulting network can be further analysed.

5.4.1 Training results

Even if training and validation results are encouraging, the only way to verify the effectiveness of the network is using it in the real-application environment, for this reason to analyse the results of the training the network is tried out on the test set.

The principal difference between validation and test process is that the test images have been reconstructed and the whole process of cropping and

duplicates elimination described in section 4.3 must be carried out. This is also the only way to validate the algorithm of duplicates elimination and to decide the best approach that have to be used. In fact, the result depends on the shape of boxes founded by the network and should be aimed at fixing all the problems that can arise but principally the most common one and it should be done only on the definitive network. The algorithm that manages the duplicates counting has already been described and now the focus is on the analysis of the results.

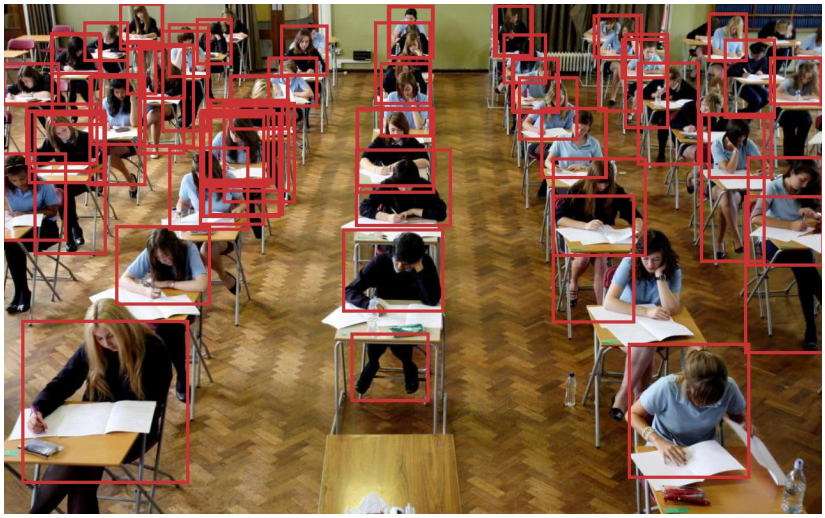


Figure 5.11: Result of neural network inference, without duplicates elimination (from [27])

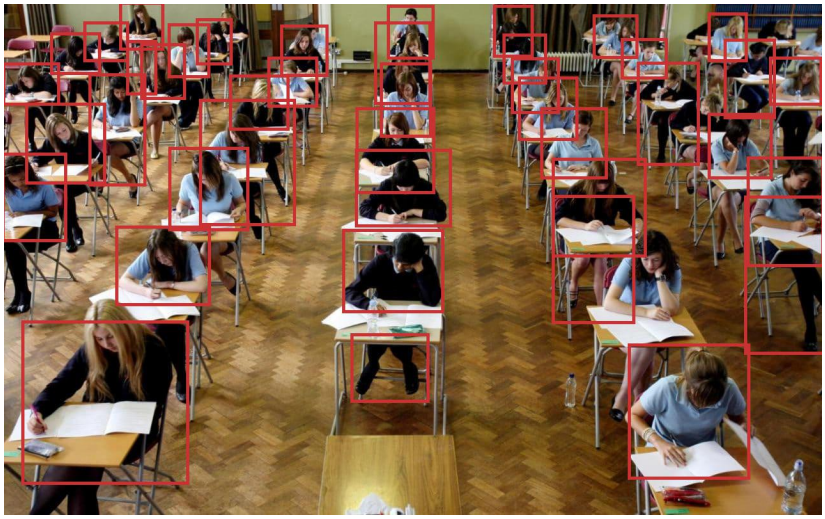


Figure 5.12: Result of duplicates elimination on previous result (from [27])

An example of the inference results can be seen in images 5.11 and 5.12. The first image represents the total detections performed by the neural network except for the ones filtered based on the score or the dimension. The second image represent the result of the duplicates elimination algorithm applied to that results.

The first thing that has to be noticed is that the shape and position of the bounding box is not always right, in some cases the box that is maintained in the duplicate's elimination is a box that contains only a part of the person's body. This is due to the fact that the effort is in maintaining the most right as possible only the number of boxes and the elimination algorithm prioritize this with respect to the quality of the box location or shape.

As can be seen in the images the algorithm is effective and also the neural network gives good results in terms of accuracy. The only students that are not detected at all are the ones in the last rows covered by the previous mates. Another criticality of the software is visible in this example, one of the guy in the first row is detected both from its upper body than from its legs, unfortunately it is not detected entirely, for this reason is impossible for the algorithm to understand that the two bounding box belong to the same person because they are not overlapped. This problem could also due to the fact the almost all the university class, so also the images used for training, are composed of desks that cover the legs. For this reason, this problem is quite rare and not severe.

Given the errors already mentioned the network is able to count 41 persons in this image and it contains 47 people, however some of them are almost completely covered and could be difficult to detect even for humans, in fact this image has been chosen in order to describe both qualities and shortcomings of the detection software.

The network inference on test set had given far better results compared with the previous pretrained networks, in order to compute a reliable value for the accuracy a test set of 120 images has been used, none of these images was also present in the training dataset. The images are from 6 different classrooms and the true number of people in each image is manually annotated. In this way the results are reliable, to have a better understanding of the accuracy the results are computed on the entire set and on a fraction of it containing only the crowded images, the threshold to include an image in the second set is 40 people. The results are also computed considering three different thresholds for the minimum score to consider each of the network's detections. In particular as threshold the value considered are 0.3, 0.4 and 0.5, higher values give poor results while lower values are too susceptible to bad detections of jackets and other objects.

The results are visible in table 5.1, as can be seen the trained model is able to detect and count the people with an accuracy around 95% in crowded

Table 5.1: Accuracy on test set using different score thresholds

Dataset	Score > 0.3	Score > 0.4	Score > 0.5
Entire test set	98.87%	98.20% s	97.76%
Crowded test set	97.13%	96.19% s	95.48%

rooms and reaches even 98% considering also easier contexts. Considering the threshold value, it can be seen that the difference is minimal and for this reason it could be safer to avoid the choice of the lowest ones 0.3, preferring a value between 0.4 and 0.5 in order to decrease the possibility of false positives in new contexts.

These results ensure the success of the training process; however the model is further improved, in particular in terms of performance using some computer vision techniques already mentioned, they are analysed in detail in the next chapter.

6 Self-calibration

The scope of this project can be summarized into two main tasks:

- Create a software that can detect and count people from digital images with the highest possible accuracy.
- Make sure that the software is able to work with the least possible monitoring.

The first task has been addressed in the chapters 4 and 5, with the creation of a counting model using a network properly trained to detect people. The second one is partially addressed with the use of an edge device to perform the computations, in fact this make sure that the camera images should not be viewed by any person and the device is able to locally count the people returning only information on the number of people detected.

However, in this way we can be sure that the software works well with a single camera in a classroom installed properly to point only at the right part of the classroom. In order to improve the versatility and independence of the software is important to have a software that works without having prior knowledge on its position and orientation.

This problem has been named *camera self-calibration*, in order to give to the software, the ability to adapt to different contexts three tasks have been considered:

- Computation of vanishing point: a useful information to avoid network errors like detecting in a single box a group of people or detect some particularly smaller objects as people is the expected bounding box dimension. We can't give a global constraint on boxes' dimension because due to the perspective we could have people with great differences in terms of dimension. For this reason the first improvement is the computation of the vanishing point, in this way is possible to project all the boxes to the same perspective point and give more strict constraints on the boxes dimension.

- Optical flow: in simple terms the optical flow is the computation of the motion of objects. This information is useful because in this way is possible to eliminate from the camera view all the parts that are motionless while the people are in the room, for example the camera could point most of the walls or some of the ceiling. These sections can be eliminated for the detection part and this could improve the efficiency of the network that doesn't have to continuously search for people where the people cannot be.
- Multiple cameras matching: in a lot of big classroom a single camera could not be enough to point at the entire room and multiple camera should be needed. The problem could be easily solved with a particular attention during camera installation to avoid overlap between the views, however we want to maximize the versatility and for this reason is implemented also a software to match the images from different cameras in order to have a unique count for all the people in the room including the ones that are in the visual of both the cameras.

These different problems are all belonging to the computer vision discipline and will be faced using different algorithms explained in detail in next sections.

6.1 Computation of vanishing point

the first task considered is the computation of the vanishing point.

Definition 6.1. A vanishing point is a point on the image plane of a perspective drawing where the two-dimensional perspective projections of mutually parallel lines in three-dimensional space appear to converge.

The vanishing point is not unique in an image and in general an image can contains more vanishing point, however we are interested in finding the one corresponding to the principal direction of the scene, that we can assume to be above the image or in its upper part, in fact objects at the end of the room will be closer to the vanishing point and will result smaller in the image.

The vanishing point of the room is computed with a single image of the empty room, in fact it is not needed to compute it multiple times, moreover the presence of people in the room is useless and could only make it harder to compute the straight lines in the image to find the vanishing point.

The method to compute the vanishing point is composed by different computer vision techniques and is inspired by the work in [29], but with some modification in the computation of edges and without the need to

compute more the second vanishing point to rectify the image. The method can be divided in two main processes:

- Hough line transform[28]: is a method to compute straight lines inside an image.
- Ransac algorithm[29]: it is an iterative method, robust to outliers, to estimate parameters, in our case used to find the estimation of the vanishing point.

6.1.1 Hough lines transform

Definition 6.2. The *Hough transform* is a technique used to isolate features of a particular shape within an image, originally invented to detect lines6.3 and then generalized for simple and complex curves or shapes. It is particularly useful to compute a global description of a feature given multiple and possibly noisy measurements.

In order to apply the hough transform to the image however we have to find the local measurements for the lines that we want to detect. This process is done performing some preprocessing to the image. The objective is finding small edges in the image and reduce all the noise possible.

In order to do that first of all the image is converted to grayscale in fact we only need the edges we are not interested in colours. After the conversion a technique to remove the noise, called *opening* is performed. Opening is an operator from mathematical morphology applied to remove noise, it is composed of two operations named *erosion* and *dilation*.

Erosion and dilation are both operations applied to binary or grayscale images: Erosion receive as input the image and a kernel and simply every time it found a set of ones in the image equal to the kernel it reduces it to a single one. The dilation does exactly the opposite, each one in the input image is expanded to a set of ones equal to the kernel. The opening is given by an erosion followed by a dilation, the effect is that all the objects in the image are shrank down and the noise point are eliminated, then the objects are expanded to the original dimension, but the noise point does not return, the effect can be seen in figure 6.1.

After the noise removal the edge are found using the *Canny edge detector*.

Definition 6.3. The *Canny edge detector* is an edge detection operator that uses an algorithm with multiple stages to detect a wide range of edges in an image. It has been developed by John F. Canny in 1986.

It is a standard and famous algorithm of edge detection, the idea is to compute for each pixel the edge gradient (sum of squared horizontal and

Figure 6.1: Effect of an erosion followed by a dilation, process named opening (from [30])

vertical gradient), and the edge angle (from the ratio between vertical and horizontal gradient). After this stage using a non-maximum suppression all the pixels that are not a local maximum for the direction of the gradient, the direction of the gradient is perpendicular to the edge direction in this way only thin edges are maintained, and all the adjacent pixels are removed. The last stage is a threshold process, all the pixels with intensity gradient above a threshold $maxVal$ are maintained, all the ones below a second threshold $minVal$ are eliminated. For the ones in between they are maintained only if they are connected to pixel above the threshold. An example of the result of Canny edge detector on an empty classroom is visible in image 6.2.

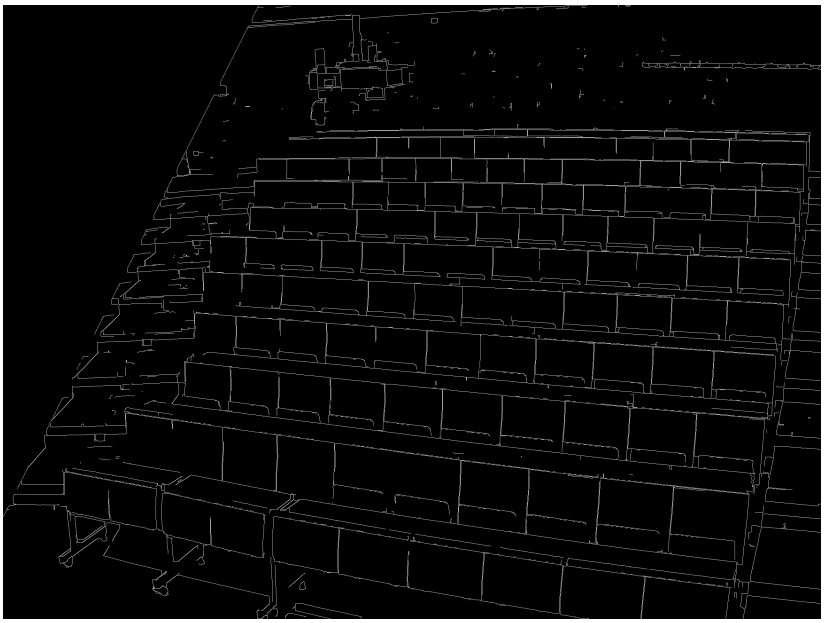


Figure 6.2: Result of Canny edge detector on an empty room

As can be seen in the image the edges are detected effectively but from this data is difficult to understand how to find a vanishing point, what is needed is to find from the edges the principal lines in the image.

For the Hough transform all the lines are expressed in polar coordinate system, with parameters (r, θ). Then for each point that belongs to an edge is defined the family of lines that goes through that point, so for point (x_0, y_0) we will have the line written as:

$$r_\theta = x_0 \cdot \cos\theta + y_0 \cdot \sin\theta. \tag{6.1}$$

Each couple (r_θ, θ) represent one of the lines and the family can be plotted in the polar Hough parameter space $r - \theta$, the resulting plot is a sinusoid in that space. This process is done on each point of the edges found, in this way a lot of different curves will be plotted in the $r - \theta$ space. The reason of this transformation is that it makes easy to search for lines, in fact every time that two curves intersect in a specific point $(\hat{r}, \hat{\theta})$ this means that the line described by that parameters goes through both the points. In other words, collinear points in the cartesian space yield to curves that intersect in a common point in Hough parameter space and that point describes the line to which both the points belong.

To find strong evidence that a line exists in the image at this point is only needed to find the point in the $r - \theta$ plane that are the intersection of the highest number of curves. The simplest idea at this point is to define a threshold that represent the minimum number of points needed to consider as existing a line.

To find the vanishing point however we are not interested in any possible lines in the image but only in image that are parallel in the real world but that point to the vanishing point in the $2 - D$ image, in particular we need the lines pointing towards the end of the room. For this reason, all the lines that have an angular coefficient below a threshold are discarded.

Furthermore, the algorithm is strongly dependent to the threshold, and in different images can have completely different results in terms of quantity and quality of lines detected. The solution proposed is to start finding only very strong lines and saving them, then trying to lower the threshold parameter until a minimum number of lines is detected. In this way the algorithm is finding always the lines with the strongest constraints possible without finding an amount of lines too small to detect a vanishing point. The result on the same empty room showed before is visible in image 6.3.

As can be seen in the figure some strong and right lines are detected however it doesn't give perfect results and some outliers seems to appear anyway, for this reason is important that the algorithm to find the vanishing point is strong to outliers. For this reason, the resulting set of lines is

Figure 6.3: Result of Hough line transform algorithm on the same empty room

then used to find the vanishing point using a method based on RANSAC algorithm.

6.1.2 Ransac algorithm

The terms RANSAC derives from *RANdom SAmple Consensus* it is an iterative method to estimate parameters from data that contains outliers. It is a non-deterministic algorithm that gives a good result with increasing probability depending on the number of iterations performed. The assumption of this method is that the data can be divided into inliers and outliers, for this reason the model is never fitted using all the data but only the one considered inliers.

The general idea of this method is to select a random set of data, from them a model is computed, then for all the other data is defined if they are inliers (fit the model within a threshold error) or outliers (do not fit the model). After that a new model is computed using all the data considered inliers, then it is compared with the best model found until that point and if it is better it become the new best model. This process is repeat for a prefixed number of times and the best model is returned at the end.

To perform this algorithm the lines detected with Hough transform are encoded as edgelets, each edgelet is composed by three components:

- Location: is a point representing the location of the line, I have considered the midpoint.

- Direction: unit vector representing the direction of the line.
- Strength: length of the line segment detected.

The algorithm randomly selects two lines, the corresponding model or vanishing point is the intersection of these two lines.

For each edgelet is then computed a vote, first for each edgelet is computed the angle between its direction and the line that connects its location with the candidate vanishing point, if the angle is below a predefined threshold that the edgelet is an inlier otherwise it is an outlier. The vote for each edgelet correspond to its strength if it is considered an inlier and it is zero otherwise.

Then the votes are summed, and the resulting value is compared with the best ones found until that iteration, if it is higher this candidate becomes the temporary best model, then the process is reiterated selecting two new random lines.

The algorithm is a really simple heuristic however it is effective with outliers by construction and with a good number of iterations it always gives a good estimate of the vanishing point. Furthermore, the precision of the vanishing point location is not fundamental in our application, it is more important to efficiently find a good estimate of its position because is used only to have an idea of the relative dimensions of bounding box in different part of the image. The result of RANSAC algorithm on the usual empty room is visible in image 6.4.

6.1.3 Utilization of vanishing point

The information of the vanishing point location is useful in order to understand the relative dimension of bounding boxes in different position of the image, in order to define constraints on the dimension of the bounding boxes the best way is to normalize the dimension of the bounding box to a fixed zone of the image, in this way is easier to compare them.

The choice has been to project all the boxes towards the vanishing point until they are in correspondence of the upper border of the image, in this way the dimension of each bounding box should depend almost only on the object they contain and most of the perspective effects should be eliminated. An example of this projection is visible in figure 6.5

On this resulting bounding box, a dimension constraint is applied, all the boxes smaller than 400 pixels (a square of $20x20$) and all the bounding box bigger than 10000 pixels ($100x100$) are eliminated.

In this way all the smaller objects like hats or pencil cases in particular positions should be eliminated if the network wrongly detects them as a

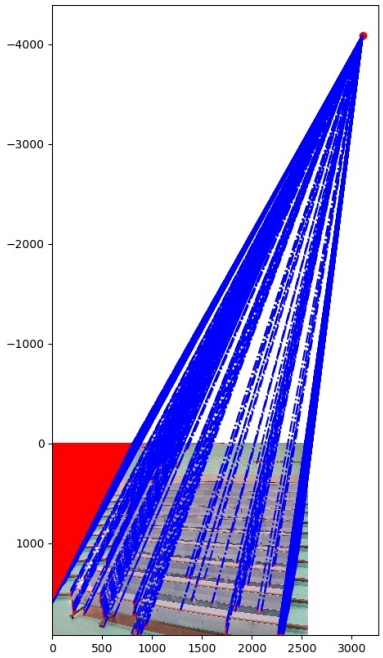


Figure 6.4: Result of RANSAC algorithm to compute the vanishing point of an empty room

person. Another problem that should be avoided is the error of the network that could detect as a single person a group of people close one to the others in a particularly crowded part of the room.

Table 6.1: Accuracy on test set using the vanishing point

Dataset	Score > 0.3	Score > 0.4	Score > 0.5
Entire test set	98.77%	97.81% s	97.64%
Crowded test set	96.30%	95.20% s	94.49%

The test performed in section 5.4.1 are repeated adding also the dimensional constraints given by the vanishing point computation. The upper threshold used for the projected box area dimension is 30000 while the lower threshold is 1500. The results are shown in table 6.1, as can be seen the use of vanishing point does not improve the accuracy, this is due to

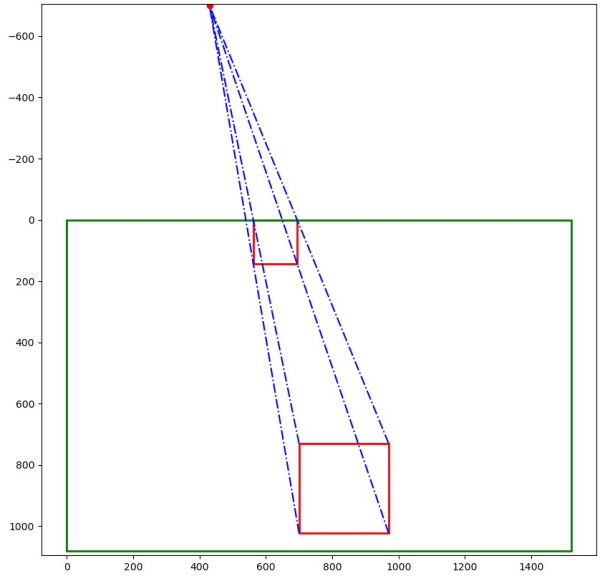


Figure 6.5: Projection of a hypothetic bounding box on the upper border using vanishing point

the fact that the network is really accurate on the test set and there are almost no macroscopical errors that can be avoided considering the bounding box dimension. However, is also important to notice that the accuracy is stable this means that computing the vanishing point is possible to add constraints, that are aimed at solving potential criticality, without lowering the software accuracy and for this reason this could anyway be useful in new contexts.

Finally, the computational time needed to compute the vanishing point could be considered. It is worth noting that the vanishing point must be computed only one time for each camera just after the installation and for this reason is not important the efficiency. The Jetson nano needs only $\sim$ 18 seconds to perform the hough line transform of the image and $\sim$ 1.5 seconds to estimate the vanishing point using the ransac algorithm.

6.2 Optical flow

The second autocalibration techniques is based on the computation of the *optical flow* of the image and it aims at finding the zones of the frame that point at zones of the room in which no people can be detected, like walls or

the ceiling.

Definition 6.4. The *Optical flow* is the pattern of apparent motion of objects, surfaces and edges in visual scenes. It can also be defined as the distribution of apparent velocities of movement of brightness pattern in an image.

It has been introduced in 1940 in study related to optics. However, in computer vision is used to define the apparent motion of image objects between consecutive frames from a camera.

The methods to determine the optical flow of an image can be divided in three main categories:

- Sparse optical flow: is a family of methods that computes the optical flow only for a set of features that should be decided with a proper feature detection algorithm.
- Dense optical flow: is a family of methods that computes the optical flow for each pixel of the frames, it is computationally more expensive but also more accurate.
- FlowNet: the last method to compute the optical flow is using a neural network that has been constructed and trained to compute the dense optical flow of a set of frames, it is usually faster than dense optical flow techniques, but the result is only an estimation of the dense optical flow.

All the optical flow methods start with the assumption that consecutive frames are divided by a small amount of time such that the objects have only a small displacement. This is important because using frames with too much difference in time the objects will have bigger movements and all the optical flow methods will not be able to recognise the movement.

From this assumption is derived an equation known as optical flow equation. Assuming a small time increment the intensity ($I(x, y, t)$) of a pixel that is moving can be considered constant between frames, this means that we can generically write:

$$I(x, y, t) = I(x + dx, y + dy, t + dt). \tag{6.2}$$

That means that the pixel is moving but is not changing value. Taking the Taylor series at the first order for the right-hand side it becomes:

$$I(x + dx, y + dy, t + dt) = I(x, y, t) + \frac{\partial I}{\partial x}dx + \frac{\partial I}{\partial y}dy + \frac{\partial I}{\partial t}dt \tag{6.3}$$

Eliminating the common factor $I(x, y, t)$ and dividing by dt a simple equation can be found.

$$\begin{aligned} \frac{\partial I}{\partial x}\frac{dx}{dt} + \frac{\partial I}{\partial y}\frac{dy}{dt} + \frac{\partial I}{\partial t} &= 0 \\ \Rightarrow \frac{\partial I}{\partial x}V_x + \frac{\partial I}{\partial y}V_y + \frac{\partial I}{\partial t} &= 0 \\ \Rightarrow I_x V_x + I_y V_y + I_t &= 0 \end{aligned} \tag{6.4}$$

Where V_x and V_y are the components of the velocity of $I(x, y, t)$ so they are the optical flow for that pixel. This equation has two unknowns and cannot be solved as it is, for this reason each method is based on adding an assumption and define an efficient method to solve this equation given that assumptions.

In order to understand the best method in terms of both accuracy and efficiency all the three different methods have been tried, the methods used are briefly described in the next sections.

6.2.1 Lucas-Kanade method for sparse optical flow

This method has been introduced in the paper [31] in 1981, it is a method to estimate the movement of some interesting features in successive frames of a scene. The computations are performed only for the pixels that are detected as interesting by a previous feature detector.

This algorithm starts assuming that the displacement of the image is not only small but also approximately constant in the neighbourhood of the pixel considered. So, it can be assumed that the optical flow equation holds not only for the considered pixel but for a neighbourhood of 3x3 of pixels.

$$\begin{gathered} I_x(x+\Delta x, y+\Delta y)V_x + I_y(x+\Delta x, y+\Delta y)V_y + I_t(x+\Delta x, y+\Delta y) = 0 \\ with \quad \Delta x \in \{-1,0,1\} \quad and \quad \Delta y \in \{-1,0,1\}. \end{gathered} \tag{6.5}$$

This gives for each of the selected pixels a system of nine equations with two unknowns, so now the system is over-determined, and the solution is found by least squares.

The Lucas-Kanade is a simple and efficient method to compute the optical flow, however it is suitable only for computing it for a small subset of pixels and for this reason its performance is strongly related to the feature detector. The most used feature detector is the Shi-Tomasi algorithm that detects corners, the reason is that the condition on conditioning useful for

Lucas-Kanade are the same considered by this corner detector and for this reason they work at the best together.

The principal problem of using a sparse optical flow is that in a crowded context is easy that some of the moving people doesn't have any pixel tracked by the algorithm and that movement is not detected. Trying it in our scenario is visible that the sparse optical flow is not the best option for two reasons. The first is that a lot of simultaneous movement can happen and not all of them will be detected, the second one is that the feature detector will tend to identify as interesting features corners or other geometric elements that are mostly part of the room equipment and will rarely take as feature pixel belonging to a person.

6.2.2 Farneback method for dense optical flow

This method has been developed in paper [32] in 2003, it consists in an algorithm to estimate the optical flow between two consecutive frames in each pixel of the image.

The first operation performed is approximating the neighbourhood of each pixel using a quadratic polynomial expansion.

$$I(x) \sim x^T A x + b^T x + c \tag{6.6}$$

The coefficient of the symmetric matrix A, the vector b and the scalar c are computed from least squares fitting the intensities in the neighbourhood. Now suppose that the neighbours are all moved by the same displacement d

$$\begin{aligned} I_2(x) &= I_1(x-d) = (x-d)^T A_1 (x-d) + b_1^T (x-d) + c_1 \\ &= x^T A_1 x + (b_1 - 2A_1 d)^T x + d^T A_1 d - b_1^T d + c_1 \\ &= x^T A_2 x + b_2^T x + c_2. \end{aligned} \tag{6.7}$$

So, the equations that link the coefficient in the first frame with the coefficient in the second frame can be derived.

$$A_2 = A_1 \tag{6.8}$$

$$b_2 = b_1 - 2A_1 d \tag{6.9}$$

$$c_2 = d^T A_1 d - b_1^T d + c_1. \tag{6.10}$$

From equation 6.9 can be easily derived the value of the displacement that we are interested in.

$$d = -\frac{1}{2}A_1^{-1}(b_2 - b_1). \tag{6.11}$$

Considering a more realistic local displacement $d(x)$ and defining $\Delta b(x) = -\frac{1}{2}(b_2(x) - b_1(x))$, the main equation becomes:

$$A(x)d(x) = 1\,\Delta b(x). \tag{6.12}$$

In principle this computation can be performed pointwise however the result turns out to be noisy. The problem is solved in a neighbour of each pixel, so the algorithm tries to find $d(x)$ that satisfy eq. 6.12 in x but also over its neighbours. The algorithm contains some refinement to improve efficiency of the computations, but this is the idea behind the computation.

The result is completely different from the one given by sparse optical flow and the movement is usually visualized as coloured patches on a black background as in figure 6.6.

Figure 6.6: Example of the result of the optical flow between two consecutive frames of a room with multiple people

In the images can be easily spotted some figures that recalls the movement of people, the result is exactly the one needed in our application and is also easier to account for all the movements during a lot of frames summing the coloured patches. However, the algorithm is quite slow and is worth to search for a solution that gives good results with a higher efficiency.

6.2.3 FlowNet

A much more modern approach to compute optical flow is given by *FlowNet*, developed in 2015 in the paper [33], it is a convolutional neural network trained to solve the problem of optical flow estimation as a supervised learning task.

The original paper proposed and compared two different architectures, *FlowNetSimple* and *FlowNetCorr*, the first architecture simply stacked the two frames together and uses as the union as input, while the second architecture first produce a representation of the two frames separately and then they are combined in the *correlation layer* before learning higher representation of their union. The general structure of the two networks is visible in figure 6.7

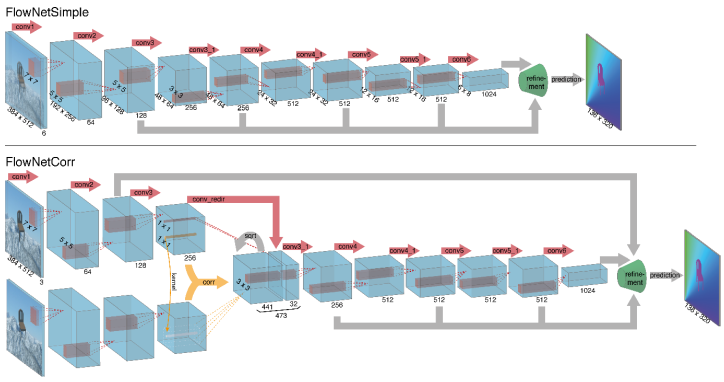


Figure 6.7: Structure of the two architectures proposed for Flownet1 (from [33])

The most interesting part of the architecture are the Correlation layer and the last refinement that generates the flow estimate. The rest of the structure is composed by usual convolutional layers that extract features.

The correlation layer performs multiplicative patch comparison between the two feature maps derived from the two frames. Given two feature maps f_1 and f_2, both with width w, height h and number of channels c, the correlation layer performs a comparison between two patches. Considering a patch of f_1 centered at x_1 and a patch of f_2 centered at x_2, both the patches squared of dimension $2k + 1 \times 2k + 1$ the correlation between $x1$ and x_2 is computed as:

$$c(x_1, x_2) = \sum_{o \in [-k,k] \times [-k,k]} \langle f_1(x_1 + o), f_2(x_2 + o) \rangle. \tag{6.13}$$

The equation is the same of a convolutional layer, but instead of convolving the feature map with a filter two feature maps are convolved together.

The CNN are good at extracting high-level features and the feature maps are shrank through the network, however the objective is to compute a dense optical flow, so we need a result in the same resolution of the input image or at least in a resolution that is a reasonable fraction of the original one. The idea is to add in the end of the network a refinement part, that is constituted by so called upconvolutional layers. They consist in an unpooling and a convolution, the unpooling extends the feature maps at the contrary of pooling. The result of this process is concatenated with the feature map of the same dimension of the first part of the network, in this way are maintained both high and low level features. The structure of this refinement is visible in figure 6.8

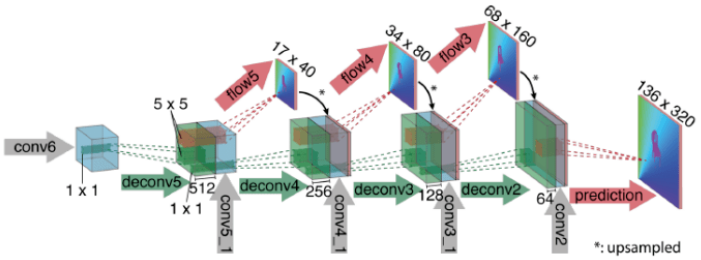


Figure 6.8: Structure of the refinement section of FlowNet (from [33])

After each upconvolution is also concatenated an up sampled coarse flow prediction. This process is repeated until the last feature map is 4 times smaller than the input image, this is because the authors have found that further refinement does not improve the performances but increase the computational complexity.

The FlowNet structure has been later improved in the paper [34], the main idea is to stack multiple time FlowNetS and FlowNetC as is shown in figure 6.9. The main structures in the network remains the same already described.

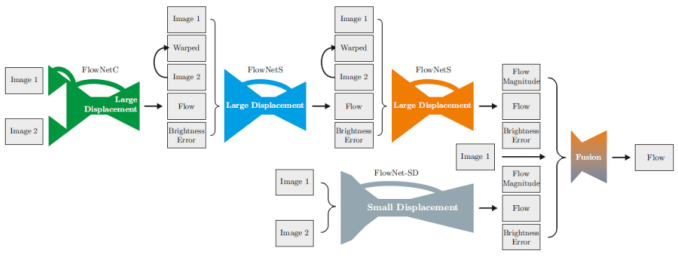


Figure 6.9: Structure of FlowNet2 (from [34])

As can be seen the networks are not only stacked one after another, but

the network is divided in two main parts, this is due to the fact that the two principal branches are trained to detect different displacements, one is specialized in small displacements and the other one in large displacement.

The result of using FlowNet is visible in figure 6.10, the image is encoded in the exact opposite with respect to the previous dense optical flow algorithm, however is visible that the kind of result is quite similar, at the same time the efficiency is improved, and the computation is way faster using FlowNet.

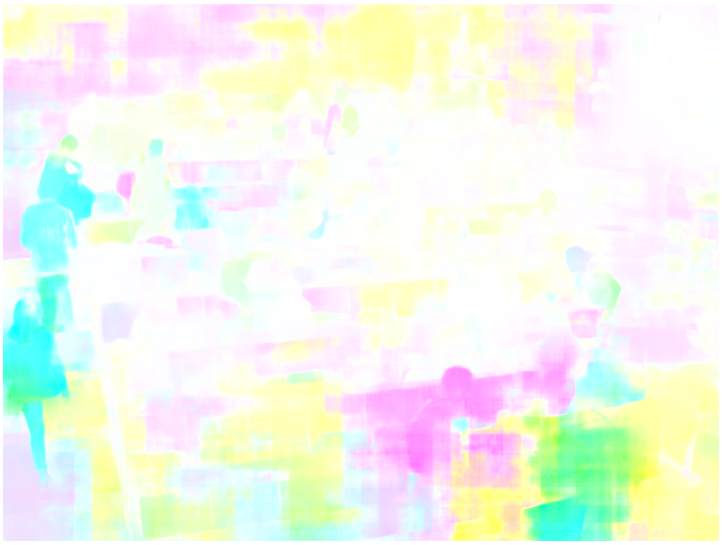


Figure 6.10: Example of the result of the FlowNet between two consecutive frames of a room with multiple people

6.2.4 Utilization of optical flow to reduce crops considered

The objective of optical flow computation is to avoid the application of the neural network on the entire field of view of the camera. I principle we cannot know the percentage of the images given by the camera that are useful to detect the people, however the optical flow can be computing even a single time to eliminate the useless sections of the frames.

To have the highest possible reliability is better to introduce some way to control this eliminated areas, a simple idea could be to perform the detection on the entire image one time when the room is particularly crowded, in this way if the optical flow have eliminated a useful zone that will be recovered if any people is inside it.

The total results of the dense optical flow are a lot noisy 6.11 and some small movement can be accidentally detected also for motionless objects.

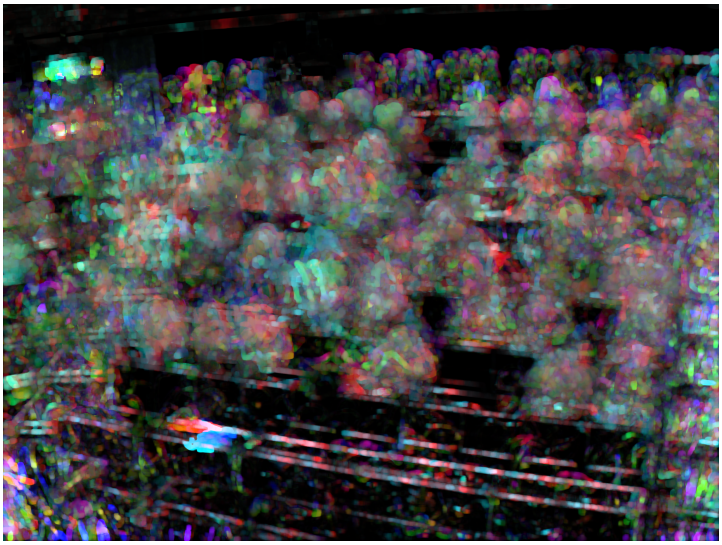


Figure 6.11: Example of the typical result of the total optical flow of a big set of frames

For this reason, the result is postprocessed, all the pixels that have an intensity of colour (movement), under a defined threshold are transformed in black pixel, so in pixels that does not indicate movement. Furthermore, for each pixel is also checked the neighbourhood, the assumption is that any movement due to a person should regard a lot of close pixels, for this reason all the pixels that doesn't contains enough movement are transformed into black.

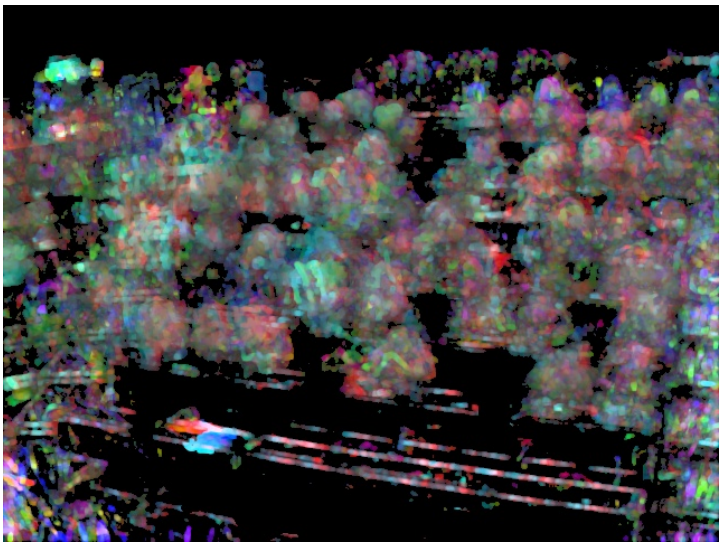


Figure 6.12: Resulting optical flow after the postprocessing

In this way the colour map is less noisy, as can be seen in the resulting figure 6.12 and it is easier to detect zones that are completely black and useless to detect people. Then the zones that doesn't contains any movement are computed and a mask for that class is added, the relative crops will not be given to the CNN in order to save some computational time. The result of this operation is visible in image 6.13.

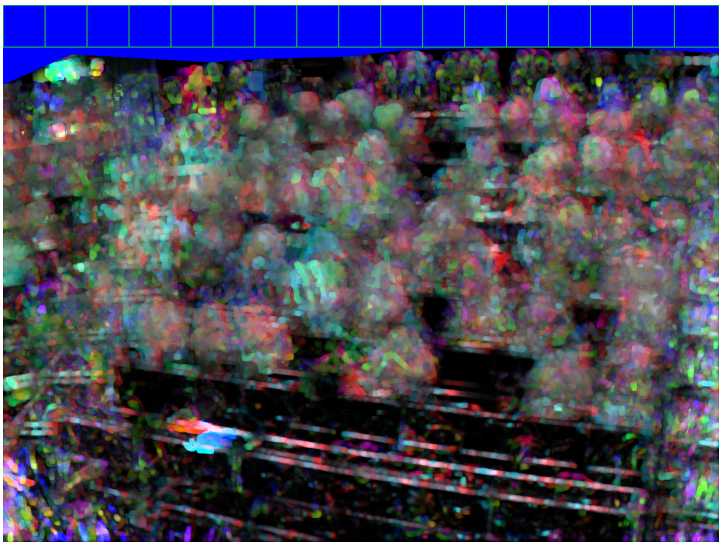


Figure 6.13: Resulting mask computed from optical flow

In this image are also showed the boxes that are eliminated with this optical flow. This image has a resolution of 2560×1920 that gives a partition of 204 crops, however using optical flow the upper row can be neglected and this eliminates 17 crops, lowering the number of inferences needed to 187, this is a reduction of $\sim 8.3\%$ of the computational cost. It is a good result for two reasons:

- Even if the optical flow is slow and computationally heavy, it must be computed only one time to obtain the crops that could be eliminated, while the reduction of computational cost is applied on each iteration of the counting process.
- It also have to be noticed that this was a case with a camera installed with a good field of view, so theoretically the speedup could be higher in other contexts.

The optical flow computation is quite slow both with standard techniques like Farneback method and using flownet. In fact, the Jetson needs $\sim 6s$ to perform the computation on a single pair of frames, while the flownet

needs a bit less than one second per image and could be used in real time. The computational time are shown in detail in the next chapter resuming all the results found.

The optical flow result strongly depends on the time interval considered because it should contain people moving all around the room, in this case the computation has been performed on a time interval of 20 minutes with students moving and considering one frame per second, so the dataset is composed of 1200 consecutive frames. Theoretically to have a better result the optical flow could be used on an entire school day however not enough data where available.

6.3 Multiple cameras matching

The last autocalibration technique implemented is the multiple cameras matching. A lot of rooms could be too wide to be checked using a single camera, however installing multiple cameras in a classroom is probable that some parts of the fields of view will be in common.

A possible solution could be to simply hand-check this issue eliminating the common part from one of the two camera's frames. However, looking forward to an independent software is better to try to implement an automatic detection for the common part, in order to merge the result from multiple cameras avoiding counting multiple times the same person.

The matching task in computer vision includes a lot of different problems, to images that have to be matched can correspond to the same images modified through rotations or other transformation, one of the two images can represent an object contained in the other image, or the two images can contain only a common part as it is in our case. In order to solve the matching, the best way is trying to find a mapping of 2D coordinates that transform the points of the first image in the corresponding point of the second one.

The most important algorithms to match different images are composed of two different algorithms:

- A feature detection algorithm: is a standard algorithm to detect useful information to perform the matching between the two images. We have already seen a feature detection algorithm in the section regarding the vanishing point. However, for each application a proper algorithm should be used, in fact in this case a feature algorithm specifically designed to perform matching will be used.

- A feature descriptor: in order to match different images is not sufficient

to detect some interest point because they usually represent small geometric objects like corners, for this reason the idea is to find a local or global description of the features that is ideally invariant under image transformation.

- A matcher: is an algorithm that aims at establishing correspondence between the feature detected in the two images, in order to define a map that links the same objects in the two images.

In our case is not necessary the mapping between the frames, in fact count the people in both the images than perform a mapping is an elegant solution but a waste of computational resources. The best use of the mapping is to mask in each of the two frames the sections of image that are already detected from the other camera, obviously the overlapped parts should be maintained in the camera that is more frontal to them in order to visualize the people in a better perspective avoiding too small bounding boxes.

6.3.1 Matching with Orb

The principal modern feature detection algorithm for matching are:

- SIFT[35] Scale-invariant feature transform
- SURF[36] Speeded up robust features, it is an improved algorithm inspired by SIFT.
- ORB[37] Oriented FAST and rotated BRIEF, it is based on the keypoint detector FAST and the descriptor BRIEF, it has been developed as an alternative to SIFT and SURF.

The main problem of using SIFT and SURF is that they are patented algorithm and for this reason they are not available on standard free libraries, ORB has been developed exactly for this reason to provide a robust and efficient alternative that is free to use. In order to describe the working mechanism of ORB is important to understand the two algorithms on which it is constructed.

FAST[38] is a corner detection method famous for its efficiency, suitable for real-time video processing. In order to define if a pixel p is an interesting point the algorithm consider a circle of 16 pixels and radius 3 around the one under test. The central pixel intensity is Ip and a threshold t is defined, the pixel p is considered to be a corner if at least n of the pixels in the radius are brighter than $Ip+t$ or darker than $Ip-t$. In the original algorithm the value for n was 12 and in order to make the algorithm faster the surrounding pixels are checked starting from four opposite pixels, if at least three of them

are not brighter or darker the other ones are not checked and the pixel p is not a corner.

Finally, in order to avoid the detection of adjacent corners in this case for the corners is computed the absolute value of the difference between p and the 16 surrounding pixels value, then only the pixel with the higher value is maintained, so it is used a Non-maximum Suppression.

One of the main disadvantages of FAST is that the detected features (corners) do not have an orientation, in order to add an orientation in ORB is computed the centroid of the circle's intensities C using the formulas:

$$\begin{aligned} m_{pq} &= \sum_{x,y} x^p y^q I(x,y) \\ C &= \left(\frac{m_{10}}{m_{00}}, \frac{m_{01}}{m_{00}} \right) \end{aligned} \tag{6.14}$$

Then the angle of the vector between the central pixel p and the centroid C is computed and it is used as the orientation of the feature.

After the detection of the feature points is applied a feature descriptor called BRIEF, it is based on describing the features using binary strings, it is fast both to compute and to match. The aim is to differentiate each feature from the other ones in order to match them between different images.

The image is smoothed to be less noise-sensitive then for each keypoint is considered a rectangular patch around it, to define a binary feature vector of length n it run n tests for each keypoints. A test τ consists in selecting randomly two pixels in the patch using a gaussian distribution, the first x is drawn from a gaussian of standard deviation σ and the second y from a gaussian of standard deviation $\sigma/2$, then the result of the test is:

$$\tau(p;x,y) = \begin{cases} 1 & if \quad p(x) < p(y) \\ 0 & if \quad p(x) \geq p(y) \end{cases} \tag{6.15}$$

Where $p(\cdot)$ is the intensity of a pixel. The results of each test are one entry of the binary feature vector, so the BRIEF descriptor is:

$$f_n(p) := \sum_{1 \leq i \leq n} 2^{i-1} \tau(p; x_i, y_i) \tag{6.16}$$

In order to take into account, the orientation computed previously in ORB is modified transforming the random points matrix S using the orientation matrix R_θ, this defines a steered version of the points matrix $S_\theta = R_\theta S$ and the resulting operator becomes:

$$g_n(p,\theta) = f_n(p) | (x_i, y_i) \in S_\theta \tag{6.17}$$

This is the functioning of ORB algorithm that detects and describes feature to perform the matching but in order to effectively perform the match between the images a matcher have to be used. The most used matcher is also the simplest that can be thought about, a *Brute-Force Matcher*, it takes the descriptor of one feature of the first image and try to match it with all the features of the second image, the closest feature based on a distance calculation is returned.

6.3.2 Matching results

Unfortunately the matching encounter some problems, the principal problem of matching two images of the same classroom from different perspective is that most of the objects are present multiple times like chairs and desks, when the matcher tries to associate the descriptor of an image with the other one it cannot know how to match a chair with the right chair of the second frame. For this reason, most of the matches are indeed between same objects in the two frames but not exactly the same, each descriptor of a chair will be matched with the most similar chair descriptor in the other image that can belong to virtually any chair in the room.

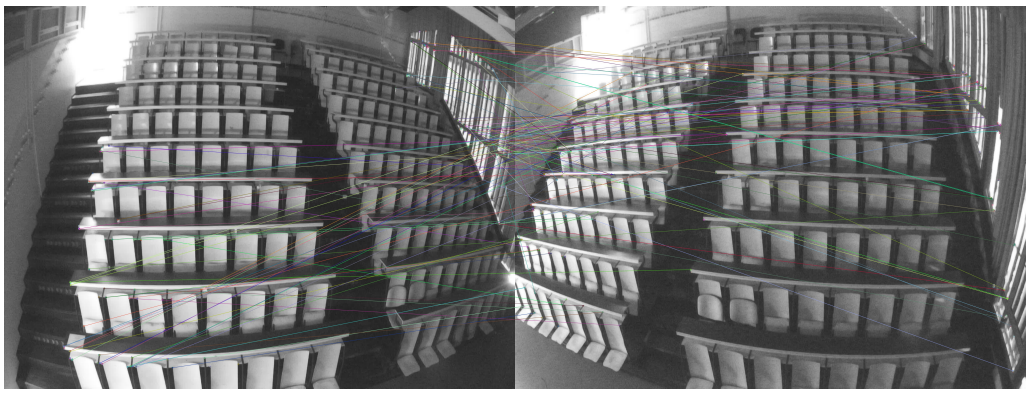


Figure 6.14: Result of the matching between frames of the same room from different perspective

As can be seen in figure 6.14, the result is a lot of matching that goes in different directions, from this matches is impossible to create a map to translate the boxes from a frame to the other, neither can be putted a mask on the sections that are seen better from the other camera.

The best solution in this case is to perform a manual elimination of the overlapped part of the two frames, obviously for each camera is maintained the frontal part of the room that is seen with a better perspective and will guarantee a better detection.

In order to understand how the frames are masked the result on the previous camera is shown in image 6.15.

Figure 6.15: Manual masking of frames of the same room from different perspective

This approach guarantees a correct counting of the people in the room and improves also the inference speed in fact in this case where almost the entire room was seen by both the cameras the frames are reduced of almost the 30% of the image.

7 Final Results analysis

In order to analyse the final result of the project two main aspects have should considered:

- The software efficiency, it is given by the processing time of the different part of the software and should allow its utilization in almost real time.
- The software effectiveness, in particular the most important result is the accuracy of the network and the postprocessing algorithms to count the number of people.

Considering the efficiency is important to consider the running times for each functionality on the Jetson Nano, the autocalibration techniques are not performed at each iteration of the counting process, they are computed at the beginning and the information are stored to improve the inference speed or accuracy, the computational times relative to these algorithms can be seen in the table 7.1.

Table 7.1: Computational times for autocalibration techniques

Technique	Subprocess	Computational time per image	Total time
Vanishing point	Hough lines Ransac	17.9259s 1.3960s	19.3219s
Optical flow	dense FlowNet	6.1467s 0.4453s	2h 3m 1.06s 9m 1.82s
Matching	Descriptor Matcher	1.3391s 0.4972s	1.8363s

The slowest technique is by far the optical flow computation however this does not affect the possibility of counting the people in real-time, in fact the optical flow computation is performed only at the beginning of the installation along with the other autocalibration techniques. However, is

important to notice the great speedup using flownet instead of a classical dense optical flow algorithm. This is due in part by the efficiency of flownet that has been developed with this objective, but also by the nature of the hardware utilized. The jetson nano is specialized in neural network inference in fact is slower than a normal laptop running classical algorithms, but way faster running neural networks. This major speedup gives the possibility to perform in real time the optical flow computation during an initial autocalibration period.

For what concern the inference time it is composed of multiple parts, the preprocessing time that includes the crops computation, the inference time on each crop and the postprocessing time that is given mostly by the algorithm of duplicates elimination. The processing times for these three factors are visible in table 7.2.

Table 7.2: Processing times on a single neural network inference

Process	Processing time	Runs per second
Preprocessing	0.6628s	1.5087
Crop inference	0.0571 ms	17.5131
Postprocessing	0.0199 ms	50.2513

ù

The main component to compute the total time needed to compute the number of people in an image is given by the time needed for each inference, in fact the preprocessing and the postprocessing are computed once for each image while the neural network has to perform between $150-200$ inferences depending on the image size and on the techniques utilized. For this reason, the higher value of preprocessing time is almost negligible because it will affect the total time by only $\sim 6\%$, while the postprocessing time is completely negligible, the remaining $\sim 94\%$ of the computational time is used to perform inference on each crop.

The total inference time depends on these three factors and on the number of crops that are considered, as we have seen the use of a mask based on optical flow and one based on multiple cameras overlapping reduce the number of crops needed. The results of the total time needed to process an image using different techniques to reduce the number of crops is shown in table 7.3.

The images used in the test set have a resolution of 2560×1920, that is divided into $204 totalcrops$. The optical flow has been computed on a single room and as we have already seen it has made possible the elimination of

Table 7.3: Total computational times with different techniques

Model	Avg crops	Computational time	Runs per second
No autocalibration	204	12.3331 s	0.081
OptFlow	187	11.3622 s	0.088
Match mask	173.5	10.5912 s	0.094
OptFlow+Match mask	161	9.8773 s	0.101

17 crops in that case. The manual mask used in place of the automatic matching has reduced the number of crops from 204 to an average of 173.5. Finally to compute the combined effect of these two techniques have been computed the crops eliminated using both, that are not exactly the sum of the crops eliminated by the two technique because some crops are eliminated in both cases, the result is an average on the different classes of 43 inferences avoided resulting in 161 crops on average that have to be considered.

Obviously, the crops elimination has a great impact on the total computational time because it is almost completely given by the single inferences on each crop.

Finally, in table 7.4 are shown the accuracy results both with and without the use of the vanishing point to try to improve the bounding box filtering.

Table 7.4: Final accuracy results with different parameters and techniques

Dataset	Score > 0.3	Score > 0.4	Score > 0.5
Entire test set	98.87%	98.20% s	97.76%
Crowded test set	97.13%	96.19% s	95.48%
Entire test set + VP	98.77%	97.81% s	97.64%
Crowded test set + VP	96.30%	95.20% s	94.49%

Taking the raw accuracy without understanding the underlying mechanism seems that avoiding the use of vanishing point and using the lower threshold value are the best choices available. However, it must be considered that we are evaluating only the ability of counting in the test phase, however is not guaranteed that all the bounding box counted really represent a person. Usually the neural network would tend to find a lower number of people than the reality because some could be covered or difficult to find for a lot of reason, this means that having false positive would tend to increase

the counting accuracy. It is noticeable in fact that sometimes with the score threshold of 0.3 the estimated count is higher than the real one.

For this reason, considering the minimal difference in accuracy between this result should probably be a safer choice the use of vanishing point together with a score threshold between 0.4 and 0.5, that maintains the accuracy above the remarkable value of 95%

7.1 Conclusions

In conclusion the project is composed by a software able to count effectively the people inside a room from digital images, together with some computer vision techniques useful to gain information about the context in order to improve the autonomy and versatility of the model.

From the results is clear that performing such task on an embedded device is possible with clearly good results. For what concern the accuracy the result is way better than expected in fact the starting idea was to achieve a 90% accuracy, however it has to be considered that asking only for the number of people two opposite errors eliminate themselves, in fact having a false positive and a person not detected gives the same result as a perfect detection. A possible future improvement could be the creation of a test dataset that contains all the ground-truth bounding boxes in order to evaluate the effective neural network accuracy in terms both of classification and localization. Another thing that should be noticed is that the images from test set where different from the training ones but taken from the same context and for this reason the accuracy is certainly overestimated with respect to the expected one on completely new contexts.

Considering the efficiency of the software the objective was to develop a software able to provide information about every minute and the total computational time needed for an image has been lowered to 10 seconds. This result leaves room for improvements like the use of a deeper and more powerful network or the developing of a system of moving average between consecutive frames to improve the accuracy. It also has to be noticed that the hardware used is quite new and during the time span of the thesis a lot of improvements has been carried out, for this reason it should be possible to improve the results using new network and refined optimization techniques.

In conclusion the main objectives of the project can be considered as achieved and the development has allowed the possibility to delve into the mathematical details of deep learning and computer vision applied in a specific context.

Bibliography

[1] Shapiro Linda, Stockman George, *Computer Vision*, Prentice Hall PTR, (2001).

[2] Feedforward Neural Networks - Bigdata Bootcamp, "http://www.sunlab.org/teaching/cse6250/fall2222/lab/dl-fnn/"

[3] Gowtham, Deep Learning - our shot towards real AI, Medium, "https://medium.com/gowtham_palani/deep-learning-our-shot-towards-real-ai-c27a5766081f"

[4] Goodfellow Ian, Bengio Yoshua and Courville Aaron, *Deep Learning*, The MIT Press, (2016)

[5] Anh Vo, Deep Learning - Computer Vision and Convolutional Neural Networks, "https://anhvnn.wordpress.com/2018/02/01/deep-learning-computer-vision-and-convolutional-neural-networks/"

[6] Maolin Shi, *Recurrent neural networks for real-time prediction of TBM operating parameters*, Research gate

[7] Yani Muhamad, Irawan (2019). *Application of Transfer Learning Using Convolutional Neural Network Method for Early Detection of Terry's Nail*, Research gate

[8] Takio Kurita, *Consecutive Dimensionality Reduction by Canonical Correlation Analysis for Visualization of Convolutional Neural Networks*, Research gate.

[9] Julien Despois, Memorizing is not learning - 6 tricks to prevent overfitting in machine learning, Hackernoon, "https://hackernoon.com/memorizing-is-not-learning-6-tricks-to-prevent-overfitting-in-machine-learning-820b091dc42"

[10] Overfitting, Wikipedia, "https://en.wikipedia.org/wiki/Overfitting"

[11] Rinat Maksutov, Deep study of a not very deep neural network, Medium, "https://medium.com/@maksutov.rn/deep-study-of-a-not-very-deep-neural-network-part-5-dropout-and-noise-29d980ece933"

[12] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, Dmitry Kalenichenko, *Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference*, 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2704-2713.

[13] NVIDIA TensorRT - NVIDIA Developer, "https://developer.nvidia.com/tensorrt"

[14] Girshick, Ross, Donahue, Jeff, Darrell, Trevor, Malik, Jitendra. (2013). *Rich Feature Hierarchies for Accurate Object Detection and Semantic Segmentation.* Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition. 10.1109/CVPR.2014.81.

[15] Redmon, Joseph, Santosh Kumar Divvala, Ross B. Girshick and Ali Farhadi. *You Only Look Once: Unified, Real-Time Object Detection.* 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR) (2015): 779-788.

[16] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, Alexander C. Berg, *SSD: Single Shot MultiBox Detector*, ECCV (2016).

[17] R. Girshick, *Fast R-CNN*, 2015 IEEE International Conference on Computer Vision (ICCV), Santiago, 2015, pp. 1440-1448. doi: 10.1109/ICCV.2015.169

[18] Ren, Shaoqing, Kaiming He, Ross B. Girshick and Jian Sun. *Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks.* IEEE Transactions on Pattern Analysis and Machine Intelligence 39 (2015): 1137-1149.

[19] Szegedy, Christian, Scott E. Reed, Dumitru Erhan and Dragomir Anguelov. *Scalable, High-Quality Object Detection.* ArXiv abs/1412.1441 (2015)

[20] Szegedy, Christian, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott E. Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke and Andrew Rabinovich. *Going deeper with convolutions.* 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR) (2014): 1-9.

[21] Ioffe, Sergey, and Christian Szegedy. *Batch normalization: Accelerating deep network training by reducing internal covariate shift.* arXiv preprint arXiv:1502.03167 (2015)

[22] Szegedy, Christian, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens and Zbigniew Wojna. *Rethinking the Inception Architecture for Computer Vision.* 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR) (2015): 2818-2826.

[23] Szegedy, Christian, Sergey Ioffe, Vincent Vanhoucke, and Alexander A. Alemi. *Inception-v4, inception-resnet and the impact of residual connections on learning.* In Thirty-First AAAI Conference on Artificial Intelligence. 2017.

[24] He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. *Deep residual learning for image recognition.* In Proceedings of the IEEE conference on computer vision and pattern recognition, pp. 770-778. 2016.

[25] Lin, Tsung-Yi, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C. Lawrence Zitnick. *Microsoft*

coco: Common objects in context. In European conference on computer vision, pp. 740-755. Springer, Cham, 2014

[26] Zoph, Barret, Vijay Vasudevan, Jonathon Shlens, and Quoc V. Le. *Learning transferable architectures for scalable image recognition.* In Proceedings of the IEEE conference on computer vision and pattern recognition, pp. 8697-8710. 2018.

[27] StudyBlue.com, *How StudyBlue can help you tackle your SATs*, Image credit: tutornerds.com

[28] Hough, P.V.C. *Method and means for recognizing complex patterns.* U.S. Patent 3,069,654, Dec. 18, 1962.

[29] Chaudhury, Krishnendu, Stephen DiVerdi and Sergey Ioffe. *Auto-rectification of user photos.* 2014 IEEE International Conference on Image Processing (ICIP) (2014): 3479-3483.

[30] OpenCV-Python tutorials, Morphological transformations.

[31] Lucas, Bruce D., and Takeo Kanade. *An iterative image registration technique with an application to stereo vision.* (1981).

[32] Farnebäck, Gunnar. *Two-frame motion estimation based on polynomial expansion.* In Scandinavian conference on Image analysis, pp. 363-370. Springer, Berlin, Heidelberg, 2003.

[33] Dosovitskiy, Alexey, Philipp Fischer, Eddy Ilg, Philip Hausser, Caner Hazirbas, Vladimir Golkov, Patrick Van Der Smagt, Daniel Cremers, and Thomas Brox. *Flownet: Learning optical flow with convolutional networks.* In Proceedings of the IEEE international conference on computer vision, pp. 2758-2766. 2015.

[34] Ilg, Eddy, Nikolaus Mayer, Tonmoy Saikia, Margret Keuper, Alexey Dosovitskiy, and Thomas Brox. *Flownet 2.0: Evolution of optical flow estimation with deep networks.* In Proceedings of the IEEE conference on computer vision and pattern recognition, pp. 2462-2470. 2017.

[35] Lowe, David G. *Distinctive image features from scale-invariant keypoints.* International journal of computer vision 60, no. 2 (2004): 91-110.

[36] Bay, Herbert, Tinne Tuytelaars, and Luc Van Gool. *Surf: Speeded up robust features.* In European conference on computer vision, pp. 404-417. Springer, Berlin, Heidelberg, 2006.

[37] Rublee, Ethan, Vincent Rabaud, Kurt Konolige, and Gary R. Bradski. *ORB: An efficient alternative to SIFT or SURF.* In ICCV, vol. 11, no. 1, p. 2. 2011.

[38] Viswanathan, Deepak Geetha. *Features from accelerated segment test (fast).* (2009): 1-5.

[39] Calonder, Michael, Vincent Lepetit, Christoph Strecha, and Pascal Fua. *Brief: Binary robust independent elementary features.* In European conference on computer vision, pp. 778-792. Springer, Berlin, Heidelberg, 2010.

Acknowledgements

Foremost, I would like to thank my advisor Prof. Francesco Vaccarino, for his availability and for the possibility to work on this thesis.

My sincere thanks also to all the Addfor group, the internship and thesis experience has been great primarily for their sympathy, cleverness and disposition; in particular thanks to Steven that has been a great tutor during this experience.

I would sincerely thank all the friends that have been close to me in this journey.

First, all my classmates and in particular Luca Paolo and Alessandro that made the courses way more pleasant.

Then I have to thank the friends that I have made in the first years, for all the nights and weekend together that was the best pauses ever from the study.

I am also grateful to my basketball teammates, they are great friends and the practices and more together helped me to forget about the university when needed.

But most of all I really thank my older friends; Prone, Giaco, Paolo, Tempo, Claudio, Coria and all the others, you really are my second family and I am sure that I can always count on you.

Above all i have to express gratitude to my family, to my parents Graziella and Dario for all the thing that you do for me, to my brother Matteo and to Beatrice for being friends even before than relatives, and to my grandmothers Clara and Mariuccia that are a constant positive presence in our lives.

Finally I really have to thank Giorgia's family to have always welcomed me at their best; and most of all I have to thank Giorgia, is difficult to find a single reason, because you have been the most important thing in my life for all these years, and probably I wouldn't even be the same person without you.