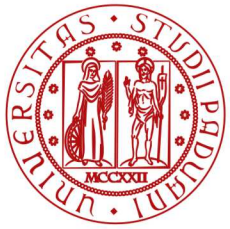


Università degli Studi di Padova

Dipartimento di Ingegneria dell'Informazione

Master Thesis in Control Systems Engineering

Comparative analysis of object density estimation algorithms for anomaly detection and object counting tasks.

Master Candidate

Christian Cusinato

Student ID 2045252

Supervisor

Prof. Cenedese Angelo

University of Padova

Academic Year
2023/2024

To the people
I care the most

Abstract

Camera usage in industrial applications is becoming more common in recent times due to its wide-ranging versatility in solving various emerging problems in a production environment. Instances of camera use can be found in diverse tasks, namely quality control, production monitoring, workplace safety, security surveillance, automation and robotics. In addition to this, nowadays, several industrial applications require the use of smart cameras capable of performing built-in computer vision and machine learning operations, thereby expanding their applicability in vision-based solutions.

This work proposes a comparison between Machine Learning and Deep Learning models, addressing the problem of manufactured object distribution density for the purpose of adversity detection and object counting. These algorithms can be useful for supervising the correctness of the production procedure and also for obtaining an estimate of the number of objects produced. The solutions explored would range from more traditional computer vision and machine learning-based approaches to custom deep learning models suited for object counting purposes. Furthermore, the proposed study addresses the applicability of such algorithms in a real-time scenario to gain insights into a control problem.

Sommario

L'uso di telecamere sta diventando sempre più frequente in tempi recenti a seguito della loro versatilità ad ampio raggio nel risolvere diversi problemi emergenti in ambiente produttivo. La presenza di telecamere si può trovare in diverse applicazioni, come il controllo della qualità della produzione e monitoraggio della stessa, sicurezza nel posto di lavoro, nei sistemi di automazione industriale e robotici. Inoltre molteplici applicazioni industriali richiedono l'uso di telecamere smart, capaci di eseguire algoritmi di Computer Vision e Machine Learning, espandendone dunque la loro applicabilità in soluzioni basate sulla visione.

Questo studio propone un confronto tra modelli Machine Learning e Deep Learning che affrontano il problema della densità di distribuzione di componenti prodotti con lo scopo di conta e di supervisione della correttezza della procedura di produzione. Le soluzioni esplorate variano da un più tradizionale approccio basato su algoritmi di Computer vision associati a soluzioni Machine Learning, al più moderno approccio basato su modelli Deep Learning allenati per ottenere una stima della conta degli oggetti presenti. Inoltre questo studio affronterà il tema dell'applicabilità di queste soluzioni ad applicazioni real-time, dimodochè si possa determinarne la possibile applicazione al controllo.

Contents

List of Figures

List of Tables

List of Algorithms

List of Acronyms

CNN Convolutional Neural Network

ML Machine Learning

DL Deep Learning

PPM Pyramid Pooling Module

MSS Multi-Sigma Stages

RVC2 Robotic Vision Core 2

SoC System on Chip

TOPS Tera Operations per second

FoV Field of view

SIFT Scale Invariant Feature Transform

FFNN Feed Forward Neural Networks

ReLU Rectified Linear Units

SGD Stochastic Gradient Descent

1 Introduction

1.1 Literature and Methodologies

Modern production environments require a substantial monitoring of the production process to ensure that the manufactured goods are up to given standards. Due to production requirements in both speed of manufacturing and increasing quality standards, the task of quality control and production monitoring has become more demanding in recent times, thus resulting in the shift from human supervision to automatic remedies capable of solving complex tasks in less time and with comparable or higher precision.
Many problems require the usage of cameras to implement vision-based solutions since vision sensors provide great flexibility in application. In fact camera sensors are widely used in pose estimation tasks in robotics approaches involving robotics manipulators, ground vehicles and aerial drones. In addition to this, many applications implement Computer Vision and Learning algorithms that exploit visual information to acquire insight in the state of the production process in both supervision and quality control tasks. Furthermore the advance in deep learning architectures has expanded the applicability of autonomous solutions and the introduction of smart cameras capable of performing stand alone Computer Vision, Machine Learning and Deep learning algorithms with small inference times.

The focus of this thesis is to approach the task of object density distribution using Machine Vision solutions both related to Machine Learning applications and Deep learning ones.

The literature in object counting algorithms is vast and proposes several solutions approaching the problem from diverse points of view. The main methodologies implemented in object counting tasks are the following:

1. Regression-based Methods
2. Density map estimation
3. Detection based methods
4. Transfer Learning

Regression based methods Initial research focus on visual counting tasks was dedicated to solve crowd counting problems for monitoring and surveillance.
One of the first approaches introduced to solve the problem was based on obtaining an estimate of the count by regression of feature vectors describing the image globally.

In the work of Chan et al. [6] the feature vector is extracted from the segmentation ground truth. This approach holds since the extension of the segmented area belonging to the crowd, is shown to be closely related to the global count by an almost linear relation. Such relation, although almost linear, brings some local non-linearities which are addressed by augmenting the feature vector with an additional 28 features related to each crowd segment, internal edge features and textures.To obtain the global count on the image, a suitable Gaussian Process performs regression on such feature vectors.
Subsequent studies like [7] and [8] by Chen et al. address the feature mining process from different points of view, gaining some advantages in count accuracy against previously introduced regression models.

Density Map estimation methods Another method that can be applied in solving object counting tasks, can be based on retrieving an estimate of the distribution density maps of the objects in the image.
Density maps can be interpreted also as probability distribution of objects since, as it will be shown later, it is convenient to construct a density maps by summing position-shifted probability distribution. Furthermore the proprieties following from the adoption of probability density distributions allow to obtain the total count of objets in the image by integration.

One of the first solution addressing the density map estimation task, was presented in the work of Lempitsky et al. [13], which formulated the problem as a minimization of a regularized risk quadratic loss function.
The introduction of Deep Learning models, in the field of visual based tasks, via the development of Convolutional Neural Network (CNN), provided considerable improvements in building suitable feature vector descriptors. Such improvements can be attributed to the capability of CNNs in performing feature extraction and subsequent feature descriptors generation automatically, since they don't rely on user defined feature extraction algorithm, thus generalizing the feature extraction phase of the algorithm.
In the work of Zhang et al. [22] a custom CNN is proposed to solve the crowd counting problem, trained exploiting a switching training approach to obtain a better optimum on the Network cost. Contemporary works like [19] proposed a multi stage structure, implemented with a suitable Feature Map extraction CNN as the first stage. Such structure is subsequently followed by a Pyramid Pooling Module (PPM), responsible of constructing a feature vector comprehensive of global and local information. Finally a Multi-Sigma Stages (MSS) refinement module capable of improving the density map for object detection purposes, receives the output of the PPM and enhances the peaks of the distribution by reducing the variance, thus highlighting the positions of the objects in the image.

Detection Based Methods Traditional Object detection algorithms, based on a Machine Learning approach, highlight certain limitations once applied to modern applications. In fact such algorithms present to be hard to optimize and with slow inference, once optimization is obtained. Such limitations are due to the way such algorithms operate, which is based on classification of sub windows and their subsequent optimization.
Recent developments in Deep Learning models shifted the process of object detection based on a classification of sub windows to more efficient regression problems. The most important one is introduced by [17], which describes the YOLO (You Only Look Once) network, which represents the state of the art approach in deep detection models. This change of approach provided the possibility of processing images in real time up to 155fps in the case of Fast YOLO, outperforming other object detection algorithms in real time applications. In the case of YOLO the number of objects instances present in an image can be obtained by summing the number of bounding boxes detected by the algorithm,

but can present some issues in counting partial or severely occluded objects.

Transfer Learning methods The concept of transfer learning is based on the capability of a Machine Learning or Deep Learning model to be fine tuned starting from a previously trained version on similar tasks. This approach can be useful when the training data available is limited and there is the necessity of building a synthetic dataset in order to train an object counting model. This approach provides a faster convergence to a local minimum solution, requires less data in order to successfully complete the training phase and improves generalisation.

The motivations for having a vision based solution capable of performing an object counting tasks, as stated earlier, is the need to supervise the correctness of the production process. For instance there could be some issues with some occlusion happening upstream of the camera, resulting from objects accumulation. This provides the information to the system that some issues occurred along the production pipeline, thus allowing an instructed operator to remove the obstruction and continuing with the production.
On the other hand there could be the case where, in a certain part of the production line, a considerable amount of unprocessed components could build up creating a bottleneck effect. With a density measurement, it is possible to control the feeding speed of the components thus allowing the occlusion to be correctly handled in order to obtain optimal feeding of components.

In this work the problem would be addressed exploiting density distribution map estimation procedure, following the process introduced in [13] and [22], since the use of density maps will provide more spatial information on the objects distributions and can provide a more stable handling of partially occluded objects with respect to the detection based methods.

1.2 Experimental setup

The experimental setup is composed by a three stage conveyor belt with the possibility of controlling each step velocity via speed knobs. The third stage of such conveyor belt structure presents a light that allows back lighting of the belt for easier segmentation of components on the belt. The feeding of the components is done by a vibrating feeder that can be filled with objects. Once operating this device would provide a uniform stream of pieces depending on the vibration intensity, which can be modified by the corresponding knob present in the control panel. At the end of the third stage a vertical structure is mounted, where the chosen camera will be allocated thus allowing a perpendicular top view of the back-lit section of the conveyor belt (see Fig.1.1).

The components of choice are washers with inner circumference diameter of 13mm, outer circumference diameter of 24mm and a thickness of 2.5mm. It is important to note that such choice was arbitrary and it will not affect the generality of the solution adopted since the goal of this work is to find a suitable process that can be applied to diverse object counting setups and possibly provide transfer learning capabilities for other counting tasks.

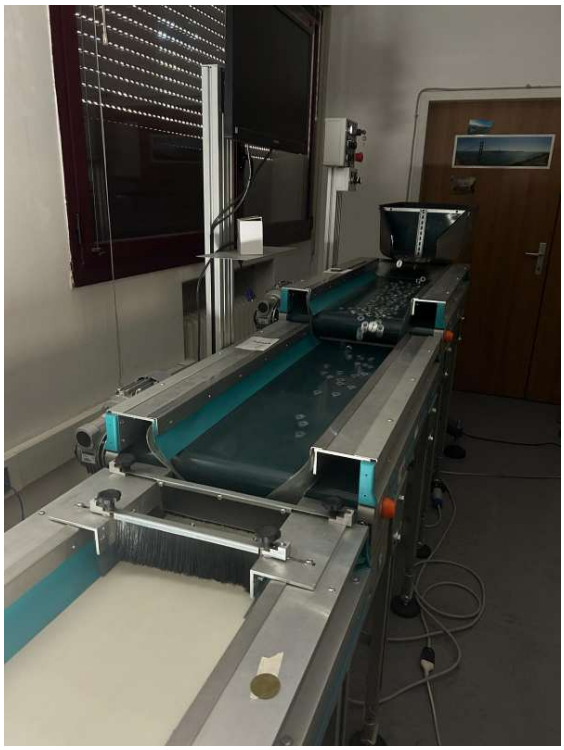


Figure 1.1: Experimental Setup

1.2.1 Camera choice

Smart cameras adopted in industrial setups are capable of performing numerous built in tasks related to computer vision, such as reshaping of the image (i.e. resizing and warping), thresholding, feature detecting and many more. Furthermore some camera models admit the possibility of running custom built computer vision functions, thus expanding the applicability on various tasks. In addition most smart cameras are capable of running Machine Learning (ML) and Deep Learning (DL) models, both already built in and custom ones. On the other hand many cameras adopted in industrial setups are inherently costly, thus the extensive use of cameras in factories further compounds to the overall expense.

(a) OAK-D Lite (b) OAK-D Pro

Figure 1.2: Images of OAK-D Lite and Pro versions

The cameras of choice for this work are belonging to the OAK-D family of the brand Luxonis, given that these cameras present a lesser cost when compared to other more common industrial cameras, providing a convenient quality over cost performances. The models in exam are the OAK-D lite and OAK-D pro versions (see Fig.1.2), which present a considerable amount of built in features. The main features are Artificial Intelligence inference, namely the possibility of running ML and DL models, Computer Vision routines, video encoding and Python Script execution among the many others. Furthermore the family of OAK-D camera present built-in stereo camera pair, enabling stereo depth perception integrated with filtering. Here lies the main difference between the two versions of the model, in fact the pro version of the camera presents an Infra-red laser dot projector that allows an active stereo perception, a feature that the lite model is missing. Both models are capable of generating 3D coordinates of detected objects, with the pro model having better measurement precision in low light

conditions. The specifics of the hardware in these cameras can be consulted in Tab. 1.1

	Color camera		**Stereo pair**	
Camera Specs	**OAK-D Lite**	**OAK-D Pro**	**OAK-D Lite**	**OAK-D Pro**
Sensor	IMX214 (PY014 AF, PY114 FF)	IMX378 (PY004 AF, PY052 FF)	OV7251 (PY013)	OV9282 (PY091 BP @ 940nm)
DFOV/H-FOV/VFOV	81°/69°/54°	81°/69°/55°	86°/73°/58°	89°/80°/55°
Resolution	13MP (4208x3120)	12MP (4056x3040)	480P (640x480)	1MP (1280x800)
Focus	AF: 8cm - ∞ FF: 50cm - ∞	AF: 8cm - ∞ FF: 50cm - ∞	FF: 6.5cm - ∞	FF: 19.6cm - ∞
Max framerate	35 FPS	60 FPS	120 FPS	120 FPS
F-number	2.2 ± 5%	1.8 ± 5%	2.0 ± 5%	2.0 ± 5%
Lens size	1/3.1 inch	1/2.3 inch	1/7 inch	1/4 inch
Effective Focal Length	3.37mm	4.81mm	1.3mm	2.35mm
Pixel size	$1.12\mu m$x$1.12\mu m$	$1.55\mu m$x$1.55\mu m$	$3\mu m$x$3\mu m$	$3\mu m$x$3\mu m$

Table 1.1: Combined Camera Specifications for OAK-D Lite and OAK-D Pro.

The OAK-D family of cameras presents three camera sensors, one color (RGB) placed in the center of the structure and two monochromatic camera sensors placed at the sides allowing precise stereo vision depth measurements in optimal light conditions. The OAK-D family presents various models suited for different ranges and applications (for instance either long range and short range), with Infra Red dot projector and flood illuminator for more precise depth estimate. The Computer Vision tasks are handled by the Robotic Vision Core 2 (RVC2), which is present throughout the whole OAK-D family. Furthermore most Luxonis cameras come with a Power over Ethernet (PoE) version for communication and power, which are better suited for industrial applications since they are provided with a built-in flash memory that allows the camera to operate

in a setup without a host device. It is possible to consult what the OAK family of cameras offers in the Tab. 1.2.

OAK Model	AI + CV	Spatial data	PoE, IP67	IR	Stan-dalone mode	Copro-cessor	Camera
1 **1 lite**	✓						
D **D lite**	✓	✓					
D S2	✓	✓					Wide FOV option
D pro	✓	✓		✓			Wide FOV option
1-PoE	✓		✓		✓		
D-PoE	✓	✓	✓		✓		
D S2 PoE	✓	✓	✓		✓		Wide FOV option
D pro PoE	✓	✓	✓	✓	✓		Wide FOV option
D-CM3, D-CM4	✓	✓			✓	RPi CM	
D-CM4 PoE	✓	✓	✓		✓	RPi CM	
FFC-3P, FFC-3P-OG	✓	✓					Custom

Table 1.2: Comparative main features of OAK models

AI inference is handled by the RCV2 module, as stated earlier, which is the second iteration of the RCV, being at the basis of most of OAK devices. It consists of two primary components namely:

- **Depth AI features :** Specifically tuned for the respective System on Chip (SoC) present on the camera.
- **High performance SoC**: Including supporting circuitry with efficient heat dissipation.

On the software side the RCV2 provides a vast selection of features:

- 4 Tera Operations per second (TOPS) with 1.4 TOPS dedicated to AI inference
- Compatibility with any AI models, including custom ones (conversion is needed)
- Video Encoding Options in various formats, resolution and framerates.
- Computer Vision capabilities with the ImageManip node (responsible for resizing, warping and cropping), and Edge Detection, feature tracking and the possibility of running custom Computer Vision functions.
- Stereo Deptth perception implemented with filtering, post-processing, RGB-depth alignment ad high configurability.
- Object tracking node offering both 2D and 3D tracking

The performance of the cameras presenting the RVC2, in running the available DL models, is summarize in Tab. 1.3

Model name	**Image size**	**FPS**	**Latency [ms]**
MobileOne S0	224x224	165.5	11.1
Resnet18	224x225	94.8	19.7
DeepLab V3	256x256	36.5	48.1
DeepLab V3	513x513	6.3	253.1
YoloV6n R2	416x416	65.5	29.3
YoloV6n R2	640x640	29.3	66.4
YoloV6t R2	416x416	35.8	54.1
YoloV6t R2	640x640	14.2	133.6
YoloV6m R2	416x416	8.6	190.2
YoloV7t	416x416	46.7	37.6
YoloV7t	640x640	17.8	97.0
YoloV8n	416x416	31.3	56.9
YoloV8n	640x640	14.3	123.6
YoloV8s	416x416	15.2	111.9
YoloV8m	416x416	6.0	273.8

Table 1.3: Performance Metrics of the camera with available DL models

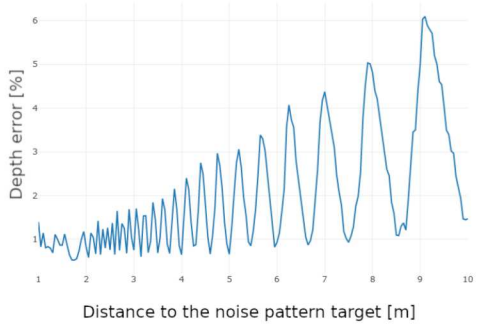
For what regards the depth accuracy of measurements with both cameras depends on several factors, like:

- **Camera Calibration:** Wrong calibrations of the cameras involved can affect the depth measurements by introducing considerable errors
- **Camera Field of view (FoV):** The FoV greatly affects depth measuramnets performed by cameras, in fact the wider the FoV the less accurate are the measurements.

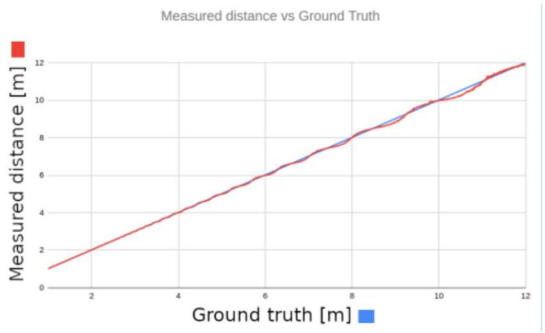
- **Resolution of stereo camera pair:** Higher resolution in stereo pair allows for better measurement accuracy
- **Baseline Distance:** The wider the baseline the better accuracy of measurements the stereo camera has, on the other hand by making the baseline wider the minimum measurable distance increases.

Most OAK-D models with a resolution of 800P (for OV9282 or OV9782 stereo pair) and a Baseline Distance of 75mm have depth accuracy and error following the graphs reported in Fig. 1.3a and Fig. 1.3b which show that:

- **Below 4m:** The absolute depth error is below 2% of the ground truth
- **Between 4m-7m:** The absolute depth error is below 4% of the ground truth
- **Between 7m-10m:** The absolute depth error is below 6% of the ground truth



(a) Depth accuracy of a 800p, 75mm baseline OAK camera



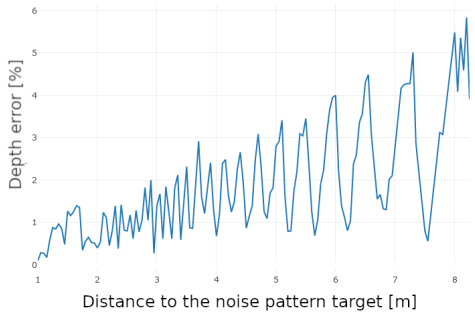
(b) Measured distance compared with ground truth 800p, 75mm

Figure 1.3: Depth accuracy and ground truth comparison with 800p, 75mm Baseline distance

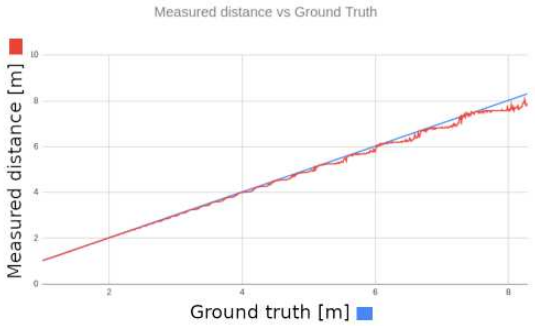
On the orther hand, for the OAK cameras with 480p resolution (OV7251), namely the OAK-D-Lite camera in our case, the depth accuracy would be:

- **Below 3m:** Below 2% absolute depth error
- **Between 3m-6m:** Below 4% absolute depth error
- **Between 6m-8m:** Below 6% absolute depth error

The error behaviour can be seen in the graphs shown in Fig. 1.4a and Fig. 1.4b



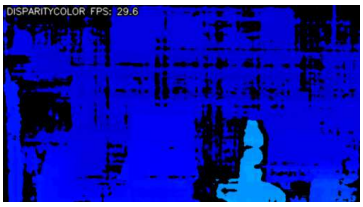
(a) Depth accuracy of a 480p, 75mm baseline OAK camera



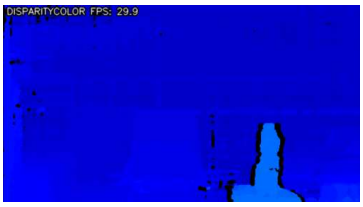
(b) Measured distance compared with ground truth 480p, 75mm

Figure 1.4: Depth accuracy and ground truth comparison with 480p, 75mm Baseline distance

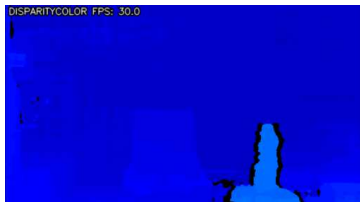
Exploiting the demo available on the website it is possible to highlight the improvements in depth perception introduced by the IR flood illuminator and dot projector. In fact by showing the raw measurements mapped into a RGB image it is seen that the OAK-D-Lite version retrieves noisier measurements since not only because it presents a monochromatic sensor of lower resolution in the stereo pair but also due to the absence of the active Stereo Depth perception. The difference can be seen clearly in Fig. 1.5: in fact the disparity depth measurements of the OAK-D Lite model presents to be the noisiest of the three, meanwhile for the Pro version of the camera the noise is reduced for both the passive and active setup.



(a) OAK-D Lite disparity depth perception



(b) OAK-D Pro passive disparity depth perception



(c) OAK-D Pro active depth perception

Figure 1.5: Mapped raw measurements highlighting the depth perception differences between the two cameras

Furthermore the pro model is capable of performing depth measurements in low light conditions due to the presence of IR Active Stereo which significantly

enhances environmental conditions. The contribution of the IR flood illuminator and dot projector can be seen in Fig. 1.6

(a) Environment with low light conditions (b) Flood illumination active (c) IR dot grid active

Figure 1.6: Low light condition scene without lighting, with flood IR lighting and no dot projection, with just dot projection

These devices can be used to setup a safety zone that detects if a foreign object (or a worker) is in the FOV of the camera without training a specific DL model but just by setting a distance threshold for which the safety system stops the production line. For the purpose of washer counting the depth measurement doesn't provide a sufficient resolution to distinguish between the object and the background due to the washer limited thickness. In order to apply disparity to our experimental setup the OAK-D SR (Short Range) model would suit better the application, due to the better accuracy subsequent to the narrower base line distance.

2

Machine Learning Solution

2.1 "Learning to count Objects" by Lempitsky et. al.

This paper, introduced in 2010, addresses the problem of object counting via a supervised learning approach. The solution proposed by this paper was applied in diverse instance counting tasks, more specifically on cell counting and crowd counting in images. One of the main advantages of such an approach is avoiding the detection task, which, before the introduction of suitable DL models, was computationally demanding. Furthermore, the paper addresses the dataset annotation part by adopting an annotation method which is both relatable to the way humans count and also reduced when compared to other annotation methods.

Annotations and Dataset generation Dataset dedicated to object counting tasks require some level of annotation that, associated with the images belonging to the dataset, highlights the position or the number of the objects in the images. The bare minimum of annotation is of course the ground truth count of object in each image. On the other hand such annotation doesn't provide any additional information on the locality of the distribution of objects in the image.

The subsequent higher level of annotations, which would be suitable for the problem in question, is point annotations on the center of the objects. Such annotations are implemented by associating the source images belonging to the dataset with a file where the point annotations are contained. Some examples of point annotations provided in the original paper are shown in Fig. 2.1, where

on the left there is an instance of such annotations applied to the cell counting task, meanwhile the right example is related to the crowd counting tasks. For both instances, the annotation in question are highlighted by colored crosses.

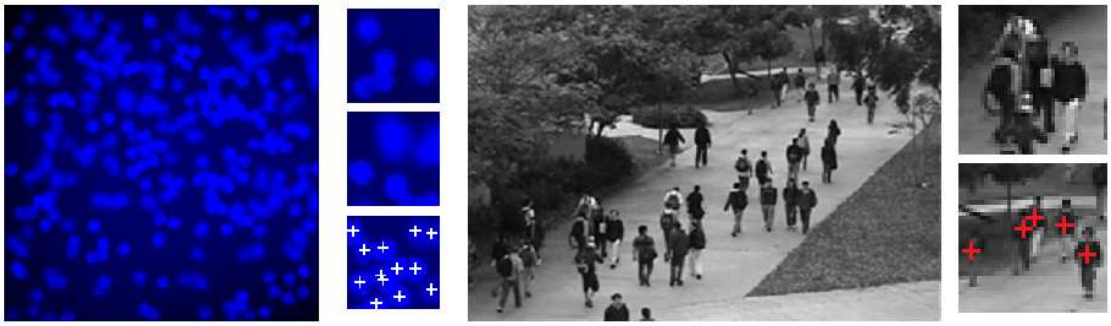


Figure 2.1: Examples of image annotations provided in [13].

The associated density maps for each example in the dataset can be obtained by superimposing a suitable density distribution centered on each annotated point. In the paper the distribution of choice is a normal, bi-dimensional distribution.

Consider the set of N training images $\mathbf{I} = \{I_1, I_2, ..., I_N\}$ and p the generic pixel of a given image $I_i \in \mathbf{I}$. With this notation the ground truth density map associated to the image I_i can be written as in expression (2.1).

$$\forall p \in I_i,\ F_i^0(p) = \sum_{P \in \mathbf{P}_i} \mathcal{N}(p; P, \sigma^2 \mathbf{1}_{2\times 2}) \tag{2.1}$$

with $\mathbf{P}_i = \{P_1, P_2, ..., P_{C(i)}\}$ being the set of points annotated by the user for each image and $\mathcal{N}$ being representative of a normal bi-dimensional distribution evaluated in p. The density map is a real valued function over the pixel grid and the ground truth count of the objects in frame can be obtained through integration of such map, namely by denoting as y_i the ground truth count it holds that:

$$y_i = \sum_{v \in F_i^0} F_i^0(v) \tag{2.2}$$

Note that the count would not be a perfect match on the number of objects present in the image due to the fact that, by integrating the contribution of the density map of objects that are not completely in frame, only a part of the object is considered in the count thus providing a "decimal" count. Such property is although desirable since objects that are not fully included in the picture natu-

rally are considered as part of an object.

Linear transformation Once a suitable training set is built, the goal shifts in learning a the linear transformation relating the feature vector associated to the image with the respective density map distribution, namely learning ω^T such that:

$$\forall p \in I_i,\ F_i(p|\omega) = \omega^T x_p^i \tag{2.3}$$

$\omega \in \mathbb{R}^K$ is the parameter vector of the linear transformation, which is the goal of the learning procedure, $F_i(\cdot, \omega)$ is the estimate of the density map for a certain ω. The parameter x_p^i is the feature vector associated to the image I_i, although its nature would be addressed later in detail.

A considerable amount of Machine Learning problems can be transposed into regularized risk frameworks, which has the goal of minimizing a given loss function with an additional term addressing the problem of over fitting. Given a set of N images and the definitions introduced above the problem introduced in the paper can be written as follows:

$$\omega^* = \arg\min_{\omega} \left(\omega^T \omega + \lambda \sum_{i=1}^{N} \mathcal{D}(F_i^0(\cdot), F_i(\cdot|\omega)) \right) \tag{2.4}$$

where $\mathcal{D}$ is a suitable loss function for the problem in question, which measures the differences in the estimated density $F_i(\cdot|\omega)$ and the ground truth density F_i^0. A further analysis is provided in the following paragraphs.

Choice of loss function The choice of a suitable loss function for the density estimation problem in question is not necessarily trivial. A natural choice would be to use either the L_1 metric (i.e. the sum of absolute differences pixel-wise) or the L_2 metric (i.e. the sum of squared differences) thus turning (2.4) into a Ridge Regression problem. On the other hand this choice would focus the problems on metrics not closely related to the task in question, which is the overall object count. The resulting map would present to be greatly affected by noise and would not correlate with the total count of the image.

Another natural choice for a counting problem would be to adopt the overall absolute count difference between two density maps F_1 and F_2 referring to image

I, namely:

$$\mathcal{D}_c(F_1(\cdot), F_2(\cdot)) = \left| \sum_{p \in I} F_1(p) - \sum_{p \in I} F_2(p) \right| \tag{2.5}$$

The two summatory symbols return the total object count of each of the two density maps for which the comparison is performed. On the other hand this choice of loss function would transform the problem into a Regression problem discussed in Section 1.1, which is not the goal of this work. Furthermore such measure requires the training set to be composed of more images when compared with other methods due to the fact that, by considering only the global count absolute difference as loss function, local information on the image are lost during the computation of each global count.

In order to both consider the global count absolute difference and keeping additional information about the local distribution of objects in the image, the paper introduces a custom loss function with the aim of implementing a substantial contribution in the training from both aspects. The function in question would be the MESA distance, described in the following paragraph.

Considering an image I and two functions $F_1(p)$, $F_2(p)$ defined over the pixel grid of I. Consider also $\mathbf{B}$ as the set of all box sub-arrays of I, the MESA distance between the two is the is the largest absolute difference between the two functions in $\mathbf{B}$, which can be written as a mathematical expression:

$$\mathcal{D}_{MESA}(F_1, F_2) = \max_{B \in \mathbf{B}} \left| \sum_{p \in B} F_1(p) - \sum_{p \in B} F_2(p) \right| \tag{2.6}$$

The advantages of adopting such metric are several, namely:

1. Given the fact that the MESA distance is defined over the set of all box sub-arrays in the image, the whole image is included in the set as well. Thus meaning that the MESA distance offers an upper bound on the absolute objects count difference.

2. The effect of zero mean independent noise between two functions would not affect the MESA distance much, since over large regions positive and negative contributions of such noise would compensate each other.

3. MESA distance is greatly affected by the overall shape of the two functions, more precisely on their spatial layout. In fact for two sub-arrays presenting

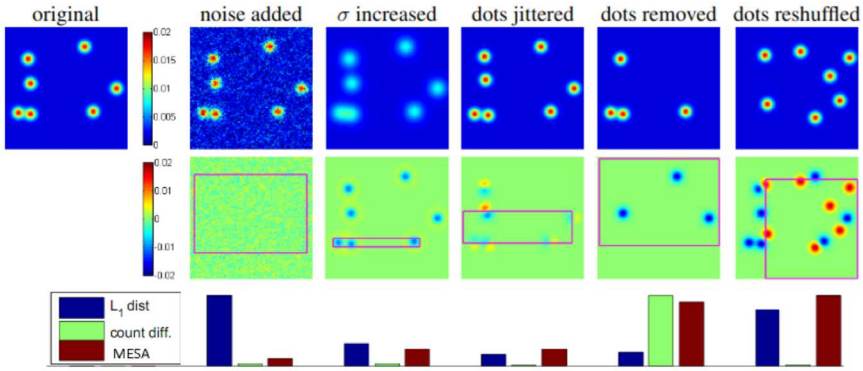


Figure 2.2: Comparison between MESA distance and other loss functions provided in [13].

the same object count but with different layouts would provide a large MESA loss.

4. MESA distance can be computed efficiently after "rephrasing the problem".

In Fig. 2.2 it is shown a comparison between MESA distance and other loss functions that were addressed previously. The "noise added" example highlights how the L_1 metric is greatly affected by noise when compared to the count difference and the MESA distance, the "σ increased" example shows how both the L_1 distance and the MESA distance return a considerable error when the object count error is close to zero, a similar outcome is highlighted in the "dots jittered" example where the two maps present similarities but with small displacements between the distributions. MESA distance provides also considerable values for the "dots removed" example and "dots reshuffled" example showing the versatility of such measure.

To explain what point 4 in the previous list means a further elaboration is needed. It is possible to rewrite the MESA distance, by transforming the absolute value into a maximum between the two differences:

$$\mathcal{D}_{MESA}(F_1, F_2) = \max\left(\max_{B\in\mathbf{B}} \sum_{p\in B} \left(F_1(p) - F_2(p)\right), \max_{B\in\mathbf{B}} \sum_{p\in B} \left(F_2(p) - F_1(p)\right)\right) \quad (2.7)$$

thus turning the problem into a maximum sub-array problem search, namely finding the box sub-array that encloses the maximum between $\max_{B\in\mathbf{B}} \left(F_1(p) - F_2(p)\right)$ and $\max_{B\in\mathbf{B}} \left(F_2(p) - F_1(p)\right)$. The solution of the maximum sub-array problem

can be found efficiently by a bi-dimensional version of the *Kadane's Algorithm*, which is addressed in [3].

Kadane's Algorithm 2D Consider a 1-dimensional array composed by N numbers, the goal is to find the contiguous sub-array containing the maximum sum of it's values.

It is possible to provide an example to better visualize the goal of the algorithm: Consider the one dimensional row array $X = [31, -41, 59, 23, -53, 58, 97, -93, -23, 84]$ represented in Fig. 2.3, for such instance the maximum sub-array is $X[2...6]$ with a sum of $\sum X[2...6] = 187$, with indexes in the image highlighted by the arrows.

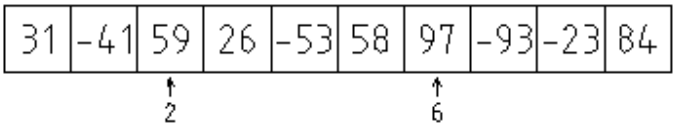


Figure 2.3: Example of one dimensional maximum sub-array

The algorithm scans the vector from left to right keeping track of the maximum sub array seen so far. Supposing that the problem is solved for some index i, namely we found the maximum sub-array in $X[0...i-1]$, in order to extend the search for subsequent elements a dynamic programming like approach is adopted. In fact the maximum at index i is either the maximum sub-array found so far (*MaxSoFar*) or it is a sub-array with ending index i, which for convenience it's called *MaxEndingHere*. In order to compute the maximum sub-array at index i the maximum sub-array at index $i-1$ is needed, indeed the maximum ending at index i falls into two possible cases:

- **CASE 1:** Adding $X[i]$ to *MaxEndingHere* yields a non negative number, thus *MaxEndingHere* is updated by adding $X[i]$.
- **CASE 2:** Adding $X[i]$ to *MaxEndingHere* yields a negative number, thus *MaxEndingHere* is updated to be 0.

Once the *MaxEndingHere* is updated it is possible to update *MaxSoFar* by comparison between the two, in fact *MaxSoFar* either remains the same or it's updated to *MaxEndingHere* if it's value is greater than *MaxSoFar* of the previous iteration. A pseudocode description of the one dimensional Kadane's algorithm

Algorithm 1 Kadane's Algorithm for one dimensional arrays

Require: $Row\ array\ X[0, .., n] \in \mathbb{R}^n$
Ensure: $MaxSoFar\ =\ Maximum\ sub-array\ sum$
  $MaxSoFar \leftarrow 0.0$
  $MaxEndingHere \leftarrow 0.0$
  for $i \leftarrow 0\ to\ n-1$ **do**
    $MaxEndingHere \leftarrow \max(0.0, MaxEndingHere\ +\ X[i])$
    $MaxSoFar \leftarrow\ \max(MaxSoFar,\ MaxEndingHere)$
  end for

is provided in Algorithm 1. This algorithm, although simple, ensures optimality since it requires a single scan of the array, thus ensuring linear complexity, namely $T(n) = O(n)$, outperforming other algorithms designed to solve this task.

This algorithm is at the base of its two dimensional version, which is the one used in this solution. Furthermore it could be useful to keep track of the indexes defining the upper-left and bottom-right corner of the two dimensional sub-array that has the maximum value of it's sum among all other sub-arrays. The pseudocode for the algorithm can be seen in Algorithm 2.

The indexes *left* and *right* are auxiliary indexes keeping track of the column index, *left* iterates over every column meanwhile *right* starts from the value of *left* and iterates over the columns on the right of the *left* index.
Once set the *left* and *right* the auxiliary vector containing the current cumulative row-wise sum *current_sum* is updated by adding the column of index *right*.
Once *row_sum* is updated then the one dimensional Kadane's algorithm is applied updating the scalar variable *max_sum*. During the iterations the indexes indicating the upper-left and lower-right corner are kept track of for the purpose of obtaining additional information about the sub-array position if needed.

This algorithm is more efficient than the more natural brute force approach which has complexity of $T(n) = O(N^3\, M^3)$, in fact Algorithm 2 has a complexity of $T(n) = O(N\, M^2)$ yielding a great improvement in performance.

Optimization phase With the use of the MESA distance as the loss function, the optimization needed to solve (2.4) is not trivial since, as it would be addressed later, the MESA distance theoretically will introduce a combinatorial number of constraints due to it being defined over the set of all sub-arrays of a given image. The first step is to write the problem as a convex quadratic program with (2.8)

Algorithm 2 Kadane's Algorithm for 2D arrays

Require: $N \times M$ matrix s.t. $X \in \mathbb{R}^{N \times M}$
Ensure: max_sum
indexes a, b, c, d s.t $\sum X[a:b, c:d] = \max_{B \in \mathbf{B}} \sum_{p \in B} X(p)$

```
max_sum ← −∞
upper, left, lower, right ← 0, 0, 0, 0
for left ← 0 to M − 1 do
    row_sums ← [0]_{N×1}
    for right ← left to M − 1 do
        row_sums ← row_sums + X[:, right]
        current_sum ← 0
        for lower ← 0 to N − 1 do
            current_sum ← current_sum + row_sums[lower]
            if current_sum < 0 then
                current_sum ← 0
                upper ← lower + 1
            else if current_sum > max_sum then
                max_sum ← current_sum
            end if
        end for
    end for
end for
return max_sum, (upper, left), (lower, right)
```

being the objective function and (2.9) being the set of linear constraints:

$$\min_{\omega,\xi_1,\ldots,\xi_N} \omega^T\omega + \lambda \sum_{i=1}^{N} \xi_i \text{ Subject to:} \tag{2.8}$$

$$\forall i, \forall B \in \mathbf{B}_i : \ \xi_i \geq \sum_{p\in B} \left(F_i^0(p) - \omega^T x_p^i\right), \ \xi_i \geq \sum_{p\in B} \left(\omega^T x_p^i - F_i^0(p)\right) \tag{2.9}$$

where $\mathbf{B}_i$ is the set of all box sub-arrays in the image Ii and ξ_i is the slack variable associated with the image i. When in the optimization procedure the optimum $\hat{\omega}$ is reached the value of the slack variable would coincide with the MESA distance.

As already stated, the combinatorial nature of the MESA distance brings a combinatorial number of constraints, highlighted by the fact that we need constraints for every box sub array of every image belonging to the training set as written in (2.9). A traditional Quadratic Programming solver would not be suitable to apply, so to overcome this problem a standard cutting-plane procedure is adopted that defines the constraint to update before each iteration of the optimization process.

The procedure is as follows:

1. A small subset of the training set is chosen to initialize the problem with a limited amount of constraints, in the paper 20 boxes with rndom dimensions and positions are selected.

2. At each iteration the quadratic programming problem (2.8)-(2.9) is solved with a subset of constraints. At the iteration j the solution would be ${}^i\omega, {}^i\xi_1, {}^i\xi_2, \ldots, {}^i\xi_N$ and one can find the box which the constraints in 2.9 are violated the most, namely for each image in the training set the goal is to find the box sub-array that maximise the right hand side of the constraints or in other words the box sub-arrays maximising $F_i^0(\cdot) - F_i(\cdot|{}^j\omega)$ and $F_i(\cdot|{}^j\omega) - F_i^0(\cdot)$. The boxes sub-arrays in question are annotated as ${}^jB_i^1$ and ${}^jB_i^2$

3. Once the boxes sub-arrays are found it is necessary to check is the constraint related would be active for the next iteration of the solving procedure. For this purpose the respective sums are computed, namely $\sum_{p\in {}^jB_i^1} \left(F_i^0(p) - {}^j\omega^T x_p^i\right)$ and $\sum_{p\in {}^jB_i^2} \left({}^j\omega^T x_p^i - F_i^0(p)\right)$, the constraint is then active for the next iteration if these quantities exceed ${}^j\xi_i \cdot (1-\varepsilon)$.
   The constant $\varepsilon << 1$ is there to guarantee the convergence in reduced

number of steps at the cost of a sub optimal solution that doesn't affect generalisation.

4. The iterations terminate once for every images the sums corresponding to the maximum sub-arrays are within $(1 + \varepsilon)$ factor from ${}^{j}\xi_i$, thus no constraints are activated for the next iteration.

2.2 Dataset Creation

2.2.1 Density maps generation

For the purpose of our object counting task a suitable dataset needs to be created. The camera is placed above and centered at the back lit part of the conveyor belt at a height of $h = 434mm$. The speed of the last step of the conveyor belt was set to be around 20% of the speed range. Batches of 200 washers are dropped inside the vibrating feeder with the vibration intensity tweaked to obtain different distribution densities. The capturing of the image is done only when washers are present to avoid empty images at a varying sampling time. A total amount of 2566 images were taken, with some examples are highlighted in Fig. 2.4

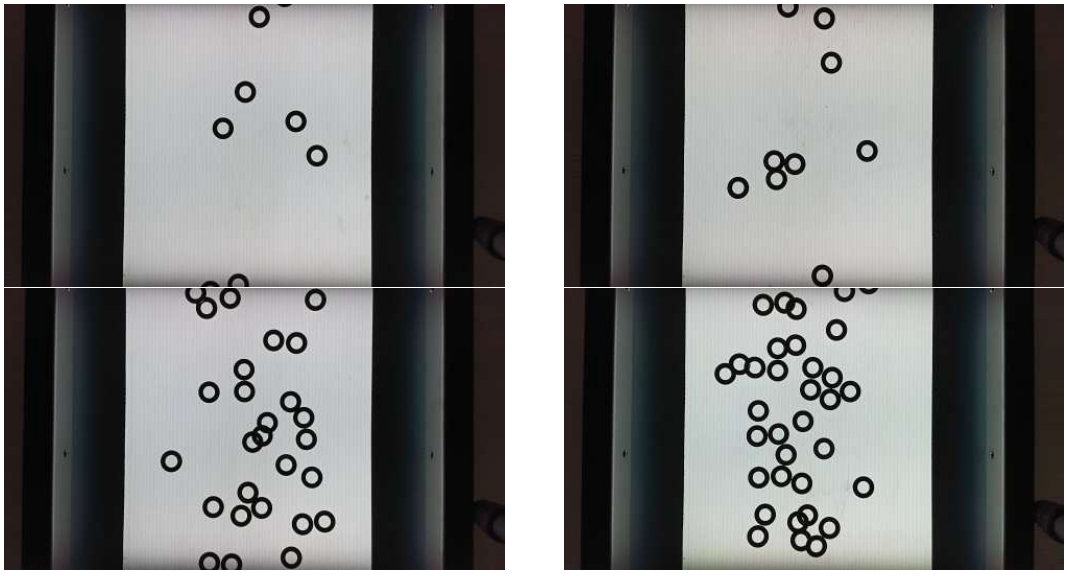


Figure 2.4: Some examples of images taken to construct the training set

The images are taken with a resolution of 1920×1080 pixels which is too high of a resolution to be handled in a manageable time, in fact in subsequent steps these images are going to be cropped and resized accordingly.

In order to obtain the annotations for the images a custom built tool is built using the OpenCV library for Python, since there are none available for the point annotations needed. Due to the circular shape of the washer the Hough transform can be applied to detect some of the centers of the inner circle of each washer.
The choice of exploiting the Hough transform to detect the centers of the washer is done to simplify the annotation speed, on the other hand such solution is not perfect since for superimposed washers the Hough transform can't always detect the center of the washer and thus the annotation should be completed by hand.
One could argue that the Hough transform could be a substitute of the entire process, since it provides the exact center positions of the washers; for this purpose the process explained in this work is to discuss a generalized approach on solving the object counting problem without relying on the particular shape of the object in case. Furthermore, as stated earlier, the Hough transform is not a perfect way to detect the center of superimposed washers since some centers would remain undetected.
A text file associated to each image belonging to the training set, containing the position of the centers of the washers in each image, is created accordingly. At this point the annotated training set is obtained and the subsequent step is to create the density maps.

One of the main advantages of adopting a density estimation approach is creating a custom density function that resembles the shape of the object in question, thus the distribution used doesn't necessarily have to be a bi-dimensional Normal distribution. In fact for the task in exam it could be useful to define a custom density function that is shaped like the washers. The distribution in question is obtained by computing the average value of the radius of the inner and outer circumference resulting in $\rho = \frac{1}{2}(\rho_1, \rho_2) = 32\, pixels$ and constructing a normal distribution developing radially. Consider a window W over which the distribution in question is defined as a 150×150 pixel grid. Assume the center of this window to be denoted as $\left[c_x,\, c_y\right]$. The value for a generic pixel

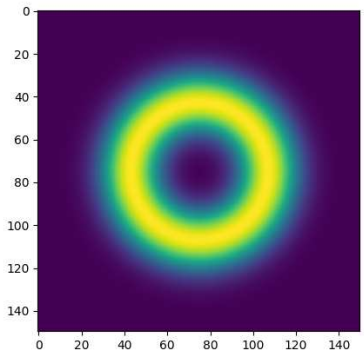


Figure 2.5: Resulting distribution shaped accordingly to the washers in exam

$p = [x, y]^T \in W$ can be computed as:

$$D_p = \frac{1}{\eta_{norm}} e^{-\frac{\left(\sqrt{(x-c_x)^2+(y-c_y)^2}-\rho\right)^2}{2\sigma^2}} \tag{2.10}$$

The quantity under the square root is the distance of the point p from the center of the distribution, σ is the radial variance that, for our purposes assumes the value of $\sigma^2 = 100.5$. The result is a washer shaped distribution that it is shown in Fig. 2.5.

Combining this distribution with the annotations provided generates the density map of every image in the training set. Some examples can be seen in Fig 2.6.

2.2.2 Feature vector

In the previous paragraphs it is discussed how the learning procedure for the Machine Learning solution has the goal of estimating the linear transformation parameters ω^T such that the expression (2.3) yields an estimate density map close to the ground truth map. The expression clearly shows the need of obtaining a suitable feature vector associated to each training sample, furthermore such feature vector should describe globally the image in exam.

A feature extraction algorithm highlights keypoints in an image, or in other words, the pixels in the image that can provide some information based on their surroundings. The information are associated to a keypoint descriptor which contains attributes of the detected location. There are several feature

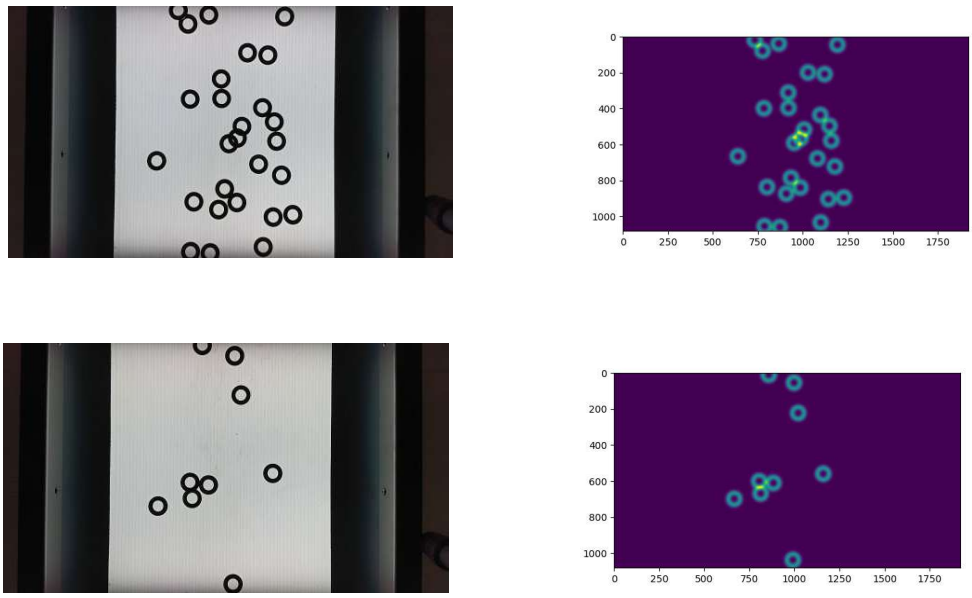


Figure 2.6: An example of acquired Image and the respective density map.

extraction algorithms capable of describing characteristics in key-points in the image, which descriptors should have the core characteristic of being invariant with respect to a family of transformations, namely:

- Variation in scale, rotation and translation of a keypoint shouldn't affect the keypoint descriptors thus meaning that the keypoints should be scale, rotation and position invariant.
- Key Point descriptors should be independent on illumination changes to guarantee a invariance with respect to light changes in changing environments.

In the setup under exam the changes in light conditions can be ignored since the environment is consistent between each image in the training set; the images are in fact taken in a dark environment with the only illumination source being the back-lit part of the conveyor belt.

Scale Space and keypoint candidates Regarding the invariance with respect to scale, rotation and position, several algorithms have been developed during the years.
One of the most important ones is the Scale Invariant Feature Transform (SIFT)

feature extraction algorithm, introduced in [14] and [15] in the works of Lowe, which is location, rotation and scale invariant.
The goal of the SIFT feature extractor is to find features that are stable across multiple scales to ensure scale invariance, for this purpose a function of scale named *Scale Space* is exploited. The Scale Space represents an image among difference scales by simulating the loss of information of lower scales images by convolving the source image with Gaussian kernels of various variance as highlighted in (2.12), where $G(x, y, \sigma)$ represents the family of smoothing kernels of variance σ which is called *Scale Parameter*. This family of smoothed images is the Scale Space representation of the source image and it is at the base of the SIFT feature extraction.

$$L(x, y, \sigma) = G(x, y, \sigma) * f(x, y) \tag{2.11}$$

$$G(x, y, \sigma) = \frac{1}{2\pi\sigma^2} e^{-(x^2+y^2)/2\sigma^2} \tag{2.12}$$

The kernels convolving with the source image $f(x, y)$ are of standard deviations that are multiplied by subsequent powers of a factor k, namely $\sigma, k\sigma, k^2\sigma, k^3\sigma...$, generating a stack of smoothed images separated by a constant factor k (a visual representation of the scale space is shown in Fig. 2.7). Once the standard deviation is doubled the stack of smoothed images forms an octave. The number of images in an octave is determined by a integer numbers representing the intervals s, as a consequence the factor k is computed as follows:

$$k^s\sigma = 2\sigma \rightarrow k = 2^{1/s} \tag{2.13}$$

In order to detect the keypoints that are stable across different scales it is exploited the difference between adjacent images in the same octave, convolved with the input images of the correspondent octave, namely:

$$D(x, y, \sigma) = \big[G(x, y, k\sigma) - G(x, y, \sigma)\big] * f(x, y) \tag{2.14}$$

Local extrema are found by exploiting the Taylor expansion of $D(x, y, \sigma)$, namely:

$$D(\mathbf{x}) = D + \left(\frac{\partial D}{\partial \mathbf{x}}\right)^T \mathbf{x} + \frac{1}{2}\mathbf{x}^T \frac{\partial}{\partial \mathbf{x}} \left(\frac{\partial D}{\partial \mathbf{x}}\right) = D + (\nabla D)^T + \frac{1}{2}\mathbf{x}^T \mathbf{H}\mathbf{x} \tag{2.15}$$

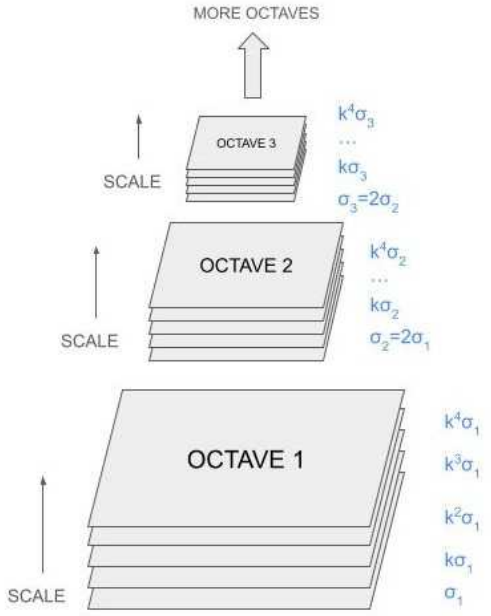


Figure 2.7: Scale Space representation

with ∇ being the gradient operator and $\mathbf{H}$ being the *Hessian Matrix*.

$$\nabla D = \frac{\partial D}{\partial \mathbf{x}} = \begin{bmatrix} \partial D/\partial x \\ \partial D/\partial y \\ \partial D/\partial \sigma \end{bmatrix} \tag{2.16}$$

$$\mathbf{H} = \begin{bmatrix} \partial D/\partial x^2 & \partial D/\partial x \partial y & \partial D/\partial x \partial \sigma \\ \partial D/\partial y \partial x & \partial D/\partial y^2 & \partial D/\partial y \partial \sigma \\ \partial D/\partial \sigma \partial x & \partial D/\partial \sigma \partial y & \partial D/\partial \sigma^2 \end{bmatrix} \tag{2.17}$$

Assuming the Hessian matrix to be invertible and imposing the derivative of $D(\mathbf{x})$ to be zero it hold that:

$$\hat{\mathbf{x}} = -\mathbf{H}^{-1}(\nabla D) \tag{2.18}$$

Thus $\hat{\mathbf{x}}$ yields local extrema in the corresponding scale.
Key points found at this stage of the feature extraction algorithm consist in points that can be unstable, thus the value of the function $D(\hat{\mathbf{x}})$ is used to identify which

points should be discarded. Experimental results introduce by Lowe [15] show that a suitable threshold could be set for $D(\hat{\mathbf{x}}) = 0.03$ based on normalized images in the interval $[0, 1]$. In addition to discarded weak key points, there is the need to address the emerging edge like key points from the difference of Gaussians.

Eliminating edge-like keypoints SIFT prioritizes corner-like key points that are significantly more localized. To eliminate edge like key points in the DoG it is necessary to compute the Hessian matrix at that given level:

$$\mathbf{H} = \begin{bmatrix} \partial^2 D/\partial x^2 & \partial^2 D/\partial x \partial y \\ \partial^2 D/\partial y \partial x & \partial^2 D/\partial y^2 \end{bmatrix} = \begin{bmatrix} D_{xx} & D_{xy} \\ D_{yx} & D_{yy} \end{bmatrix} \tag{2.19}$$

The eigenvalues of this matrix are related to the curvature of D which is used to highlight edge like keypoints. Eigenvalues are not computed explicitly but, given the fact that the trace and determinant of H are related to the sum and the product of it's eigenvalues. Assuming that α, β are the eigenvalues sorted by magnitude it holds that:

$$Tr(\mathbf{H}) = D_{xx} + D_{yy} = \alpha + \beta \tag{2.20a}$$

$$Det(\mathbf{H}) = D_{xx}D_{yy} - (D_{xy})^2 = \alpha\,\beta \tag{2.20b}$$

If the determinant is negative then the curvatures have different signs and the key point in question can't be an extremum, thus it is discarded. Now define as r the ratio between the largest and smallest eigenvalue, thus $\alpha = r\beta$ and:

$$\frac{[Tr(\mathbf{H})]^2}{Det(\mathbf{H})} = \frac{(\alpha+\beta)^2}{\alpha\beta} = \frac{(r\beta+\beta)^2}{r\beta^2} = \frac{(r+1)^2}{r} \tag{2.21}$$

The quantity in (2.21) is related to the ratio of the eigenvalues and has it's minimum if the eigenvalues are equal. To discard edge like key points the ratio of principal curvatures must be below a certain threshold, namely:

$$\frac{[Tr(\mathbf{H})]^2}{Det(\mathbf{H})} < \frac{(r+1)^2}{r} \tag{2.22}$$

The experimental results provided by Lowe [15] suggest a value of $r = 10$, thus key points with curvature higher than $r = 10$ are discarded.

Keypoint Orientation The subsequent step is to assign orientation information to the Key-point descriptor based on local image properties, in order to achieve invariance to image rotation. The smoothed image L closest to the keypoint scale is then selected and a neighborhood of the keypoint at that scale is analyzed. On this region it is calculated the magnitude and orientation of the gradient exploiting pixel differences, exploiting the equations in (2.23a) and (2.23b).

$$M(x,y) = \left[(L(x+1,y) - L(x-1,y))^2 + (L(x,y+1) - L(x,y-1))^2\right]^{1/2} \quad (2.23a)$$

$$\theta(x,y) = \tan^{-1}\left[\left(L(x,y+1) - L(x,y-1)\right) / \left(L(x+1,y) - L(x-1,y)\right)\right] \quad (2.23b)$$

An histogram of orientations weighted by the magnitude of the gradient is then created, with 36 bins spanning over 360°. The dominant peak of the histogram is then considered as the orientation of the keypoint, if another peak is present within 80% of the dominant peak, another keypoint with that orientation is then created. Thus meaning that in such a case the keypoints would have same location and scale but different orientations. This instance occurs only 15% of the times but it contributes considerably in image matching tasks.

At this point the keypoints found have scale and orientation information associated, thus the next step is to compute the keypoint descriptor which, in fact describes the region surrounding the keypont. Considering the keypont to be at the center of a 16×16 window divided by 4×4 blocks like in Fig. 2.8

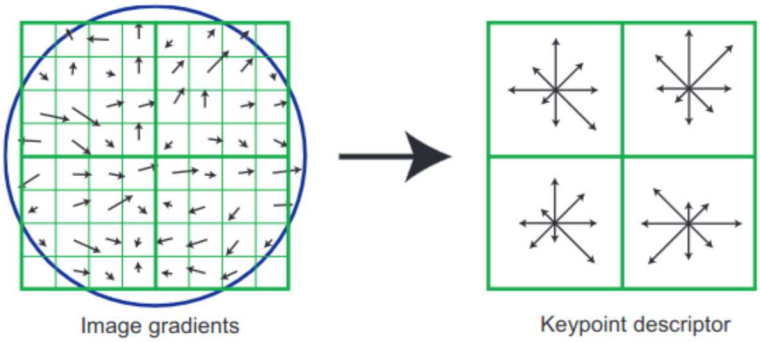


Figure 2.8: Sift 16×16 window selected to compute the feature vector associated to the keypoint.

Exploiting pixel gradient magnitude and direction computed at each pixel for each 4×4 window an eight-directional histogram is computed (shown as randomly oriented arrows in the left of Fig. 2.8). Thus meaning that every 4×4 window introduces 8 dimensions to the feature vector, implying that the feature

vector associated to the keypoint considered is of dimension 128.

The feature vector x_p^i introduced in (2.3) is a global feature vector describing the image. For this reason a feature extraction algorithm like SIFT has to be tweaked since, in the described procedure, only few points are selected as keypoints. To overcome the issue arising, the SIFT descriptors are calculated over every pixel in the image without performing the keypoint detection phase, this process creates a dense feature vector of the image and it's called dense SIFT. With the resolution of the images taken during the collection of the dataset, the process of extracting a dense feature vector associated to the image is computationally demanding in terms of time.
For this reason the instances should be reduced in dimension in order for the training sample to be processed in a suitable time, in the following ways:

- **Cropping:** In the images collected, the region of interest is defined only in the back-lit section of the conveyor belt, thus meaning that the computations performed outside of this region are not useful and therefore can be left out.

- **Down sampling:** It is possible to reduce the dimension of the instances by down sampling the image into one of lower resolution. The advantage of having images at reduced size comes at the cost of a loss of detail, thus meaning that the resizing factor should be chosen wisely in order to not compromise the learning procedure.

The images are then cropped and down sampled into frames of dimensions 216×188 and some examples are shown in Fig. 2.9.

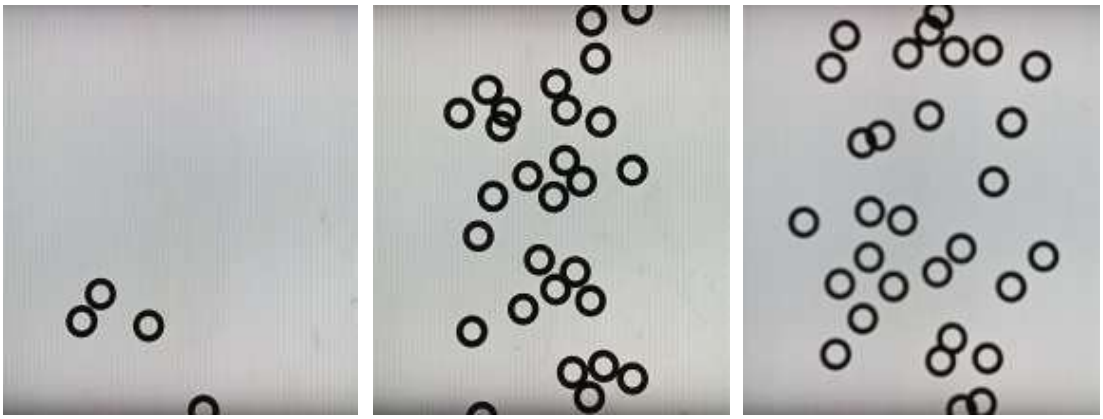


Figure 2.9: Cropped and down sampled images

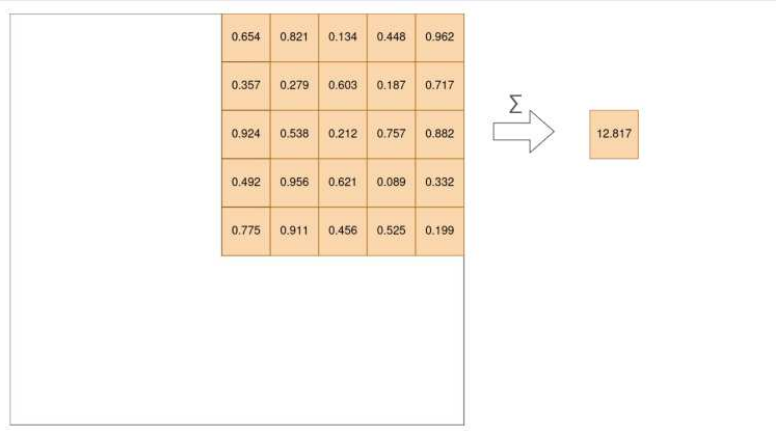


Figure 2.10: Visualization of downsampling for the density maps.

Once the images are down sampled to the size written above, the subsequent step is to reduce the densities to have the same dimensions of the image. After the cropping of the unnecessary parts, the image has dimension 1080×940 and the subsequent down sampling reduces the images to 20% of the cropped image size to obtain the dimension of 216×188 as written above.

The first step in resizing the ground truth densities accordingly is to crop the 2D array defining such densities at the same indexes for which the corresponding image was cropped. Subsequently, in order to down sample the density map accordingly to the image, it is possible to reduce every 5×5 window into a single value which is the sum of every value of the window. This results in the density map being the same size of the images in the training set, without changing the total count of the objects in the image.

An efficient implementation of the dense SIFT algorithm can be found in the VLFeat library with Matlab and C implementation. This library offers the Matlab routine *vl_dsift* which outputs the key point positions and descriptors.

It is to denote that the computation of the dense descriptors is performed without implementing the scale space, in fact the dense descriptors are computed on the input image considered to be already at a certain scale. This means that at each feasible pixel just the SIFT descriptor is computed, with the same magnitude/orientation method.

2.2.3 Bag Of Words for feature vector quantization

Applying the SIFT feature descriptor computation directly to each pixel in the image would create a feature vector of dimension $[h \times w \times 128]$ where 128 is

the length of a single SIFT descriptor and $w \times h$ is the dimension of the image in exam. It is useful to reduce the size of the feature vector in order to simplify the optimization process for estimating the density feature parameter ω. To reduce the dimensionality of the feature vector obtained by dense SIFT, the descriptor associated to each point of the image is quantized. In order to perform the quantization procedure it is necessary to create a codebook of visual word, i.e. a dictionary that associates a SIFT descriptor to a codeword.
This encoding process is at the base of a bag of visual words approach, which is one of the simplest algorithms for classification tasks, but it's principles can be extended in this case.

To construct a dictionary a selection of images $\mathbf{I} = \{I_1, I_2, ..., I_n\}$ is considered, for each of them the dense SIFT descriptors are computed and collected in a set of individual descriptors $\mathbf{x} = \{x_{1,1}, x_{1,2}, ..., x_{1,m}, ..., x_{n,1}, ..., x_{n,m}\}$. These descriptors are considered as training set for a clustering algorithm with the centers μ_k of each cluster being the visual word.
The simplest clustering algorithm to apply is the K-Means algorithm, which works ad follows: The goal is to partition the data points into K groups named clusters, the choice of K is not trivial and a tentative based approach can be used. For our case it is chosen $K = 256$. The goal is to assign each data point to a cluster center μ_k such that the sum of the squared distances between data points and their relative centers is at the minimum. To describe the action of assigning a data point to a cluster it could be useful to define a set of indicator binary variables $r^k_{i,j} \in \{0, 1\}$ with $k = 1, .., K$ describing which cluster the point is assigned to, $i = 1, ..., n$ the image on which the descriptor is extracted and $j = 1, ..., m$ being the specific index of the descriptor in the image. The objective function that can be defined is the distortion measure given by:

$$J = \sum_{i=1}^{n} \sum_{j=1}^{m} \sum_{k=1}^{K} r^k_{i,j} ||x_{i,j} - \mu_k||^2 \tag{2.24}$$

The indicator variable would be either $r^k_{i,j} = 1$ if k is the assigned cluster or $r^c_{i,j} = 0, \ \forall c \neq k$.
In order to find the $r^k_{i,j}$ and μ_k that minimize the distortion measure J an iterative approach can be exploited, which for every iteration two successive optimization steps with respect to $r^k_{i,j}$ and μ_k are performed.
The initialization of the algorithm if performed by fixing the initial values of

μ_k and minimize J with respect to $r^k_{i,j}$ while keeping μ_k fixed. Now, while keeping $r^k_{i,j}$ fixed, the distortion measure is optimized with respect to μ_k thus updating the cluster centers. This two step optimization procedure is repeated until convergence. The algorithm is named EM algorithm where E stands for Expectation, which is the first optimization process with respect to $r^k_{i,j}$, and M which stands for Maximization and it's related to the second optimization step. This two steps algorithm is at the base of the K-means algorithm.

To explain in detail how the first step of the optimization procedure the linearity of J with respect to $r^k_{i,j}$ is exploited to obtain a closed form solution. The terms in i and j are independent from each other, thus it is possible to optimize for every $x_{i,j}$ by fixing $r^k_{i,j} = 1$ for whichever value of k returns the smallest value of $||x_{i,j} - \mu_k||$, or in other words this assigns the i,j-th data point to the closest cluster center.

$$r^k_{i,j} = \begin{cases} 1 & \text{if } k = \arg\min_c ||x_{i,j} - \mu_c||^2 \\ 0 & \text{otherwise} \end{cases} \tag{2.25}$$

Once the $r^k_{i,j}$ are fixed then the optimization is then performed over μ_k. The function J is a quadratic function of μ_k, thus in order to minimize the derivative with respect to μ_k is taken and set to zero, resulting in:

$$2 \sum_{i=1}^{n} \sum_{j=1}^{m} r^k_{i,j}(x_{i,j} - \mu_k) = 0 \tag{2.26}$$

which solved for μ_k yields:

$$\mu_k = \frac{\sum_{i=1}^{n} \sum_{j=1}^{m} r^k_{i,j} x_{i,j}}{\sum_{i=1}^{n} \sum_{j=1}^{m} r^k_{i,j}} \tag{2.27}$$

By inspection of the terms in (2.27) it is possible to denote that the denominator is the total count of points associated to the cluster k, thus meaning that the center of each cluster is obtained by taking the average point between every entry of the respective cluster. Once obtained the cluster centers $\boldsymbol{\mu} = \{\mu_1, \mu_2, ..., \mu_K\}$, which in our case are the visual words of the training set, it is possible to proceed with the quantization process.

To obtain the feature vector that would be used for the learning procedure, the dense SIFT features extractor is applied to every pixel in a given image. The

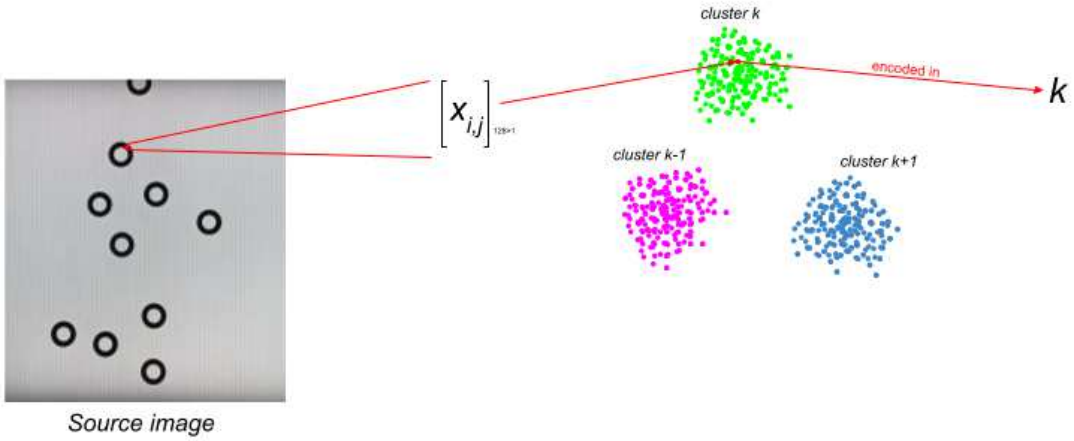


Figure 2.11: Visualization of encoding process

individual pixel descriptors would then be assigned to the closest cluster center μ_k and thus encoded into an index $k \in [0, 255]$ indicative of the closest visual word associated to that given pixel descriptor. This means that the resulting feature vector x_p^i is a matrix with the same dimensions of the training image sample and, in correspondence of every pixel position, the respective encoded visual word as it's value (a visual representation of the encoding process is provided in Fig. 2.11).

2.3 Algorithmic implementation

Considering the fact that, in order to build the dense feature vector associated with the single training instance in an efficient manner, the vlFeat library for Matlab was adopted, the whole optimization algorithm associated with the learning procedure was implemented in the same environment. Such choice is also advantageous due to the fact that the Optimization Toolbox provided in Matlab presents to be very efficient in solving linear and quadratic programming problems.

In this section only the pseudo code is presented in order to provide some generality of the solution, it is to denote that the indexes for the cell-arrays and matrices follow Matlab notation, namely the first element of both structures is at index 1.

The first algorithm that would be addressed (Algorithm 3) is the section of the code that created the dictionary of visual words. Assuming that $I_s \subset \mathbf{I}$ is a subset of the images in the training set, $k = 256$ the number of visual words adopted, the algorithm should output the dictionary containing the coordinates of the cluster centers. In the first row of the pseudo-code an empty Matlab vector

Algorithm 3 Dictionary creation

Require: $I_s = \{I_1, ..., I_s\} \subset \mathbf{I}$; k
Ensure: $\mu = \{\mu_1, \mu_2, ..., \mu_k\}$
1: **keypoints** is initialized as empty vector
2: **for** $i \leftarrow 1$ **to** s **do**
3: [points{i}, values{i}] ← vl_dsift(image{i})
4: keypoints ← cat(2, keypoints, values{i})
5: **end for**
6: [idx, μ] ← kmeans (keypoints,k)

is created, for each image of this subset the dense descriptors found are then concatenated to such key points variable exploiting the Matlab routine cat(). In the following step, the K-Means algorithm from the Statistics and Machine Learning Toolbox is applied, which results in the cluster centers contained in μ. These are the saved in a *.mat* file which would be the dictionary of visual words.

The algorithm implementing the optimization process has it's pseudo-code written in Algorithm 4. It is important to address some lines of Algorithm 4 which are important to highlight the learning procedures and understand the implementation.

Algorithm 4 Learn to count Objects

Require: $k, features, \lambda, densities, maxIter$
Ensure: ω

$size \leftarrow [features\{1\}.shape(1), features\{1\}.shape(2)]$
n $\leftarrow len(features)$
$preAlloc \leftarrow 1 + initial + maxIter \cdot numImages \cdot 2$
$Rects \leftarrow [0]_{preAlloc \times 5}$
$nRects \leftarrow 0$
A $\leftarrow sparse(2 \cdot preAlloc, k + n)$
b $\leftarrow [0]_{2 \cdot preAlloc \times 1}$
for i $\leftarrow$0 **to** initial **do**
  while $|x(2) - x(1)| < 1$ **or** $|y(2) - y(1)| < 1$ **do**
    $x \leftarrow \lceil rnd_{2\times1} * size(1) \rceil$
    $y \leftarrow \lceil rnd_{2\times1} * size(2) \rceil$
    $i \leftarrow \lceil rnd_{1\times1} * n \rceil$
  end while
  $AddRect(x, y, i)$
end for
$H \leftarrow \begin{bmatrix} \mathbb{I}_{[k\times k]} & \mathbf{0}_{[k\times n]} \\ \mathbf{0}_{[n\times k]} & \mathbf{0}_{[n\times n]} \end{bmatrix}$; $f \leftarrow \begin{bmatrix} \mathbf{0}_{[k\times 1]} \\ \lambda \cdot \mathbf{1}_{[n\times 1]} \end{bmatrix}$; $lb \leftarrow \mathbf{0}_{[(k+n)\times 1]}$
for $iter \leftarrow 0$ **to** $maxIter$ **do**
  $sol \leftarrow$ Solution of $min_x \frac{1}{2}\mathbf{x}^T H\mathbf{x} + f^T\mathbf{x}$subject to: $A\mathbf{x} < b$ and $lb < \mathbf{x}$
  $\omega \leftarrow$ first n elements of sol
  $\xi \leftarrow$ remaining elements of sol
  for $im \leftarrow 1$ **to** n **do**
    $density = \omega^T x_p^i$
    $diff =$ Ground truth density$\{im\}$ − Estimated density$\{im\}$
    $[v_1, y_{m1}, y_{M1}, x_{m1}, x_{M1}] \leftarrow$ Maximum sub-array of *diff*
    $[v_2, y_{m2}, y_{M2}, x_{m2}, x_{M2}] \leftarrow$ Maximum sub-array of - *diff*
    $slack \leftarrow \xi\{im\}$
    if $max(v_1, v_2) < slack \cdot 1.01$ **then**
      continue
    end if
    if $v_1 > v_2$ **then**
      $AddRect([x_{m1}, x_{M1}], [y_{m1}, y_{M1}], im)$
    else
      $AddRect([x_{m2}, x_{M2}], [y_{m2}, y_{M2}], im)$
    end if
  end for
end for

The rows annotated from 1 to 16 belong to the initialization phase. It starts by storing the dimension of the image in the variable *size* and the number of images in the raining set in the variable n.
After that a variable named *preAlloc* is initialized, which is related to the maximum number of iterations that can be imposed in the learning process. In fact the pre-allocated dimension for the matrices that store the information about the constraints emerging during the iterations of the optimization procedure are related to twice the number of images in the training set, since for each image two constraint are generated, as expressed in (2.9).

The first $while$ loop initializes the constraints at the beginning of the optimization procedure. The algorithms draws two random points (x, y) which define the box sub-arrays top-left and bottom-right corners, and a random index i for the associated image for which the box sub-array is drawn. Once a suitable box-array is drawn (namely $[x, y]$ define a rectangle of non negligible dimensions) a custom built function named $AddRect(x, y, i)$ adds the coordinates defining the box sub-array as well as updating the matrices A and b introducing the new constraint to the optimization problem.

In line 16 of the pseudo-code are initialized the matrices related to the objective function. In fact one can write, assuming the notation $\mathbf{x} = [\omega^T, \xi^T]^T$:

$$\min_{\omega,\xi_1,\xi_2,\ldots,\xi_N} \omega^T\omega + \lambda \sum_{i=1}^{N} \xi_1 = \min_{\mathbf{x}} \mathbf{x}^T H\mathbf{x} + f^T\mathbf{x} \tag{2.28}$$

with:

$$H = \begin{bmatrix} \mathbb{I}_{[k\times k]} & \mathbf{0}_{[k\times N]} \\ \mathbf{0}_{[N\times k]} & \mathbf{0}_{[N\times N]} \end{bmatrix}; f = \begin{bmatrix} \mathbf{0}_{[k\times 1]} \\ \lambda \cdot \mathbf{1}_{N\times 1} \end{bmatrix} \tag{2.29}$$

The right hand side of (2.28) is the notation that the Matlab routine $quadprog()$ expects, except for the fact that the documentation states that the objective function is of the type $\min_{\mathbf{x}} \frac{1}{2}\mathbf{x}^T H\mathbf{x} + f^T\mathbf{x}$.
The presence of the coefficient $1/2$ is not mandatory and doesn't affect the solution value at the optimum.

An analogous manipulation of the terms defining the constraints results in a deeper understanding on how the matrices $\mathbf{A}$ and $\mathbf{b}$ are constructed in the algorithm. Consider the *AddRect()* function is called at a certain iteration of the optimization algorithm, thus there is the need to add the resulting constraint in

the optimization process, namely considering just the first constraint:

$$\xi_i \geq \sum_{p \in B} \left(F_i^0(p) - \omega^T x_p^i \right) \tag{2.30}$$

$$-\sum_{p \in B} F_i^0(p) \geq \left[-\sum_{p \in B} x_p^i \right]^T \omega - \xi_i \tag{2.31}$$

This results in the matrix **A** and the column array **b** having two rows dedicated to each box sub array introduced, or as a mathematical expression:

$$\begin{array}{c} \\ i \\ \downarrow \\ \end{array}$$

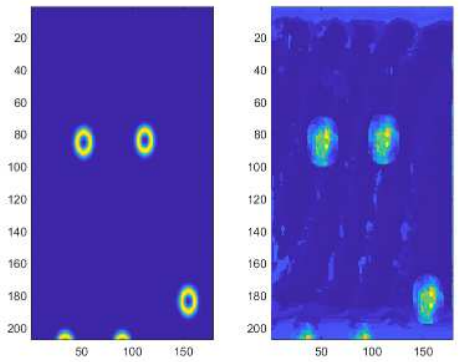
$$\underbrace{\begin{bmatrix} & \vdots & & & \\ -\sum_{p \in B_i} x_p^i & 0 & \dots & -1 & \dots & 0 \\ \sum_{p \in B_i} x_p^i & 0 & \dots & -1 & \dots & 0 \\ & \vdots & & & \end{bmatrix}}_{\mathbf{A}} \underbrace{\begin{bmatrix} \omega \\ \xi \end{bmatrix}}_{\mathbf{x}} \leq \underbrace{\begin{bmatrix} \vdots \\ -\sum_{p \in B_i} F_i^0(p) \\ \sum_{p \in B_i} F_i^0(p) \\ \vdots \end{bmatrix}}_{\mathbf{b}} \tag{2.32}$$

The **for** loop highlighted between the lines 17 and 21, shows the part of the code where the quadratic program is solved and the values for ω and ξ at the optimum are then retrieved.
The subsequent for loop iterates over the images in the training set computing the density estimates (line 23), computing the difference with the ground truth and obtaining the maximum sub-arrays for the computation of the MESA distance. Now considering the ξ at the optimum found at that point, the code checks which constraint is violated and adds it to the formulation of the quadratic problem for the next call of the solver. Once an optimum solution is obtained, namely no more constraints are violated within the threshold $\epsilon = 0.01$.

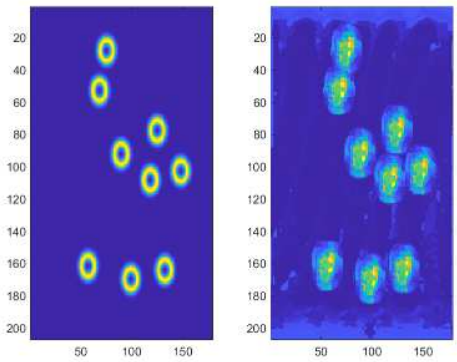
2.4 Training process and validation results

The training procedure was performed with a reduced data set composed of 256 images where a total of 224 are used in the optimization procedure an the other 16 are kept out of the learning process and used for testing the results of the optimization. A maximum number of 10 iteration can be performed, such number can be thought as small but it will be shown that the optimum solution is found in less iterations of the optimization algorithm.

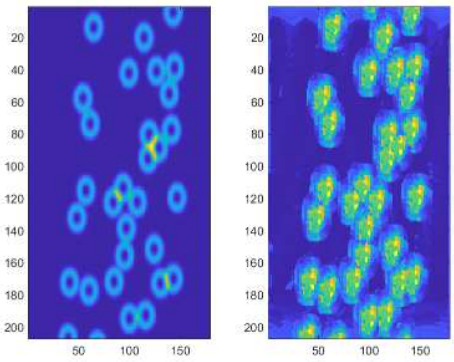
After the first iteration of the optimization algorithm, the estimated guesses are far from the ground truth as expected. The performance after the first iteration of the optimization procedure can be seen in Fig. 2.12, where three different instances with low, medium and high number of washers are compared.



(a) Training instance with low number of washers



(b) Training instance with medium number of washers

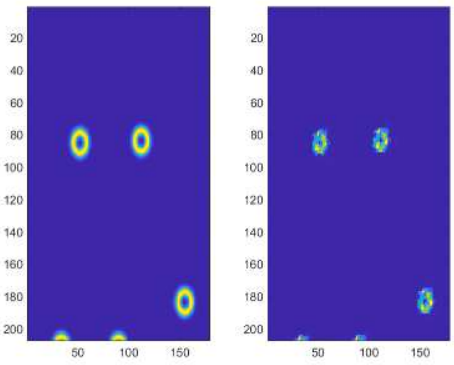


(c) Training instance with high number of washers

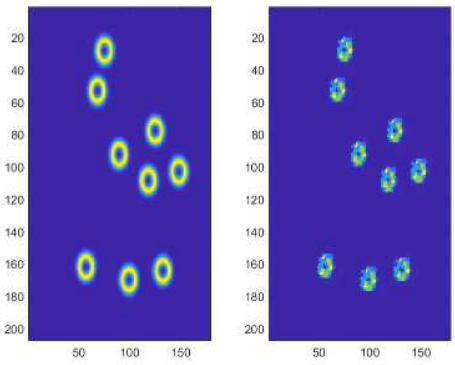
Figure 2.12: Training instances and density estimates after 1 iteration

After a single iteration of the optimization process the average error on the object count in the whole training set is $count\_err = 3.0$.

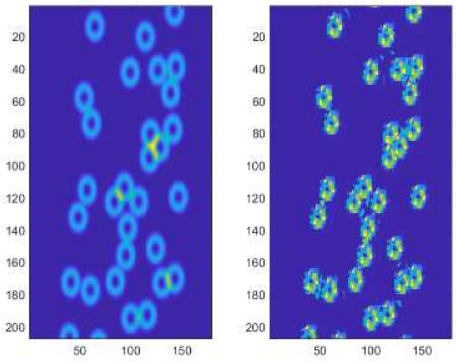
Major results can be seen after the second iteration, where the estimated densities start to resemble, in shape and location of the washers, the ground truth densities. Furthermore it is shown, even at this point of the optimization procedure, how objects that are not fully in frame provide a contribution to the estimated density map.



(a) Training instance with low number of washers



(b) Training instance with medium number of washers

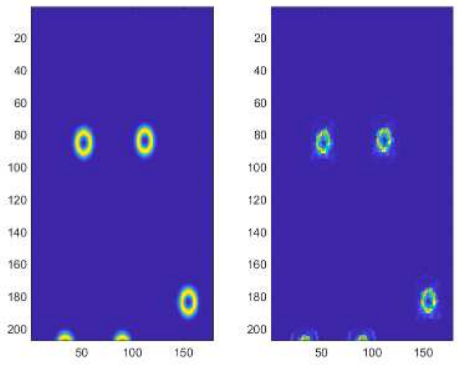


(c) Training instance with high number of washers

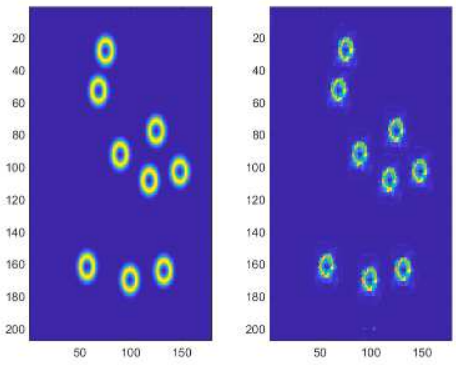
Figure 2.13: Training instances and density estimates after the second iteration

The algorithm converges in 6 iterations where the constraint generation procedure can't find more violated constraints. The results at the conclusion of the optimization process is not much different from the ones seen after the second iteration in both overall map shape and count difference (Fig. 2.13)

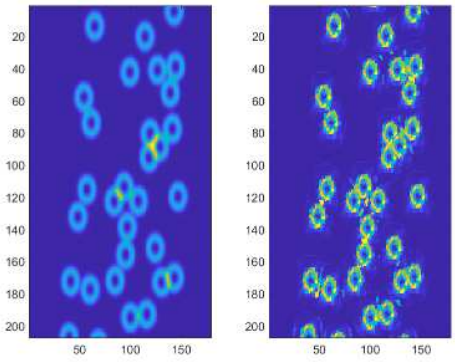
In fact, by focusing on the object count difference from the ground truth



(a) Training instance with low number of washers



(b) Training instance with medium number of washers



(c) Training instance with high number of washers

Figure 2.14: Training instances (left) and density estimates (right) after convergence

(Fig. 2.15), computed for every instance of the training set, it is highlighted that such measure converges rather quickly to a minimum and becomes almost stationary for the majority of the training iterations after the second one. By focusing on the estimated maps obtained after convergence is reached, it is shown that both the positions of the objects and their shape are correctly estimated by the algorithm as it is shown in Fig. 2.14 suggesting that the resulting linear mapping ω^T is estimated correctly.

Considering the results obtained on the guesses performed over the validation set, it holds that the model commits an average error on the ground truth count of $count\_diff = 0.47$. Such error is comparable with the average error on the training set, thus suggesting that the model is likely not over-fitted on the training data. Some results on the validation images can be seen in Fig. 2.16

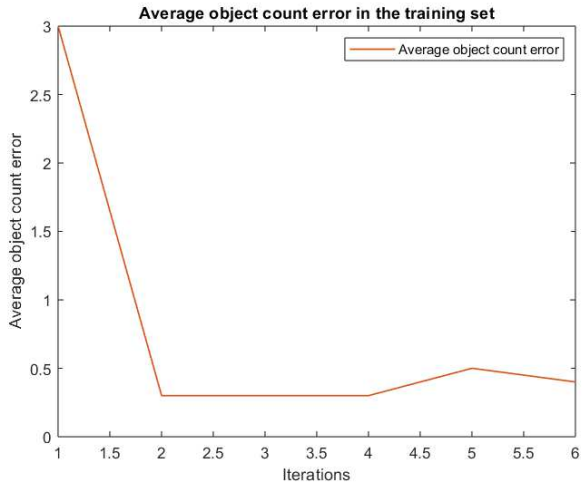
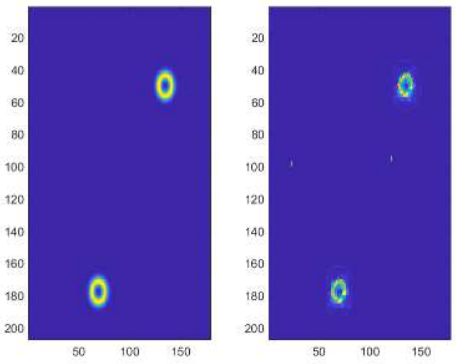
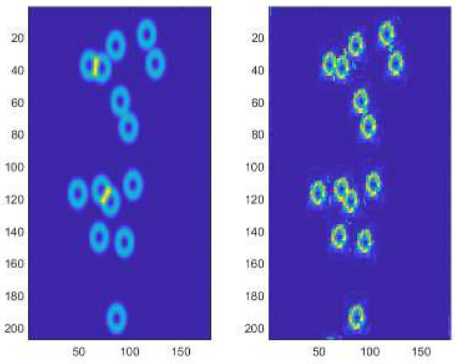


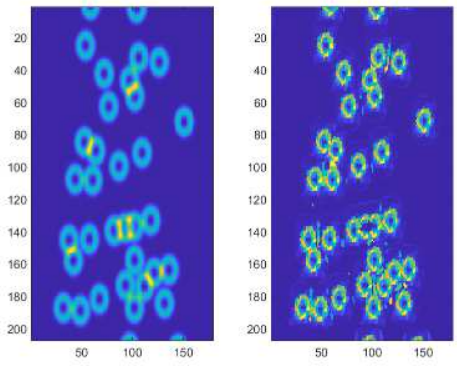
Figure 2.15: Average object count error at each iteration of the training algorithm.



(a) Test Image 1. True Count = 2.00 Predicted Count = 2.26



(b) Test Image 11. True Count = 14.00 Predicted Count = 13.6



(c) Test Image 11. True Count = 32.47 Predicted Count = 31.66

Figure 2.16: Some examples of low, medium and high density of objects in the test set and their respective estimates.

3

Convolutional Neural Network solution

3.1 Overview of Convolutional Neural Networks

In order to address the concept of CNN it is necessary to overview Feed Forward Neural Networks (FFNN) from a general point of view, since CNNs are a specific application of FFNN.

The goal of a FFNN is not different from every Machine Learning model, namely to approximate a function f as a function $\hat{f}$ such that, for a given input x, it holds that $f(x) \sim \hat{f}(x)$. Thus meaning that there is the need to define a mapping $y = f(x, \theta)$ and, via a suitable learning procedure, retrieve the parameters θ that best estimate the function f.
The term *Feed Forward* is related to the path on which information travels, namely the structure of the model doesn't include feedback connections.

FFNN can be represented as a direct graphs composed by subsequent layers of nodes, an example can be seen in Fig. 3.1. Considering f as a function that can be written as a composition of other functions, namely:

$$f = f^l \circ f^{l-1} \circ \cdots \circ f^1(x) \tag{3.1}$$

each function f^i is a layer of the neural network, and the overall number of functions l is the **depth** of the network, highlighting the motivations behind the term "deep" in the nomenclature.

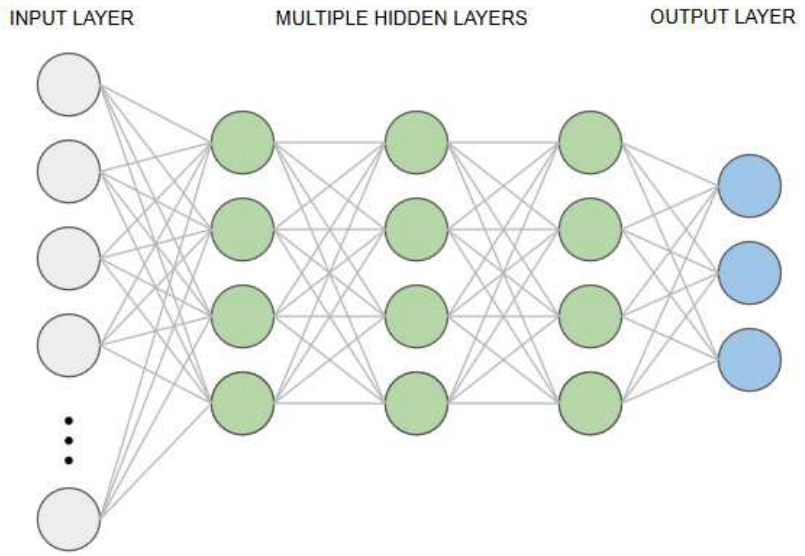


Figure 3.1: Example of a Neural Network representation

The final layer of the Network is called **output layer** .

Consider the training set being built by instances $\{\mathbf{x}, y\}$ where $\mathbf{x}$ are the training inputs and y are the corresponding outputs such that $y \approx f(\mathbf{x})$. The goal of a FFNN is to produce, at the output layer, a result close to the given label y associated to the training instance.
The behaviour of the layers in between the input layer and the output layer is non explicitly specified and is a result of the training process. This means that the learning algorithm must evaluate how each layer in between contributes in building the desired output, thus implementing the best approximation of f. The layers between the first layer, called **input layer**, and the **output layer** are called **hidden layers**.

The term *neural* refers to the fact that these networks are loosely inspired by an interpretation of the inner workings of a brain, where each node is compared to a single neuron which is connected to other nodes acting as a network of neurons.

Considering traditional Machine Learning models, it is fair to state that they can be used to approximate linear functions fairly well. Linear Machine Learning models can be also extended to represent non linear functions by addressing some preliminary remarks. In fact almost every non-linear function can be approximated by a linear function applied to a transformed version of

the input by a non linear transformation, chosen accordingly.

$$y = f(\mathbf{x}, \theta, \omega) = \Phi(x, \theta)^T \omega \tag{3.2}$$

In the context of FFNN modelled to approximate non linear functions, it holds that the goal of such structures is to obtain an approximation of the non-linear mapping Φ. In other words the parameters of the Network are tweaked so that the inner layers behaviour best approximates Φ. The parameters ω define the linear map that relates $\Phi(\mathbf{x}, \theta)$ to the desired output.

The introduction of hidden layers require the choice of activation functions used to compute hidden layers outputs. Furthermore network designers should choose the structure of the network, namely the number of layers (depth of the network), how they are connected between each other and the number of units composing each layer.

In order to perform optimization on the Network parameters, it is necessary to compute the gradient of the "cost" of the Network (which is a measure defined similarly to a loss function in the traditional Machine Learning sense). The task of computing the gradient, with respect to the Network parameters, it is usually not trivial and, for more complex structures, can be unfeasible to compute analytically. The solution of this issue is provided by the back-propagation algorithm, which doesn't compute the gradient exactly but a close estimate.

These concepts of activation functions and back-propagation algorithm will be addressed in the following sections.

3.1.1 Hidden Units

Hidden units (or nodes) are the basic blocks building the inner layers of a FFNN and can be defined by elements that receive as input a vector $\mathbf{x}$ and computing an affine transformation $z = W^T x + b$. W is the matrix of **weights** and b represents the **bias** of each hidden unit, such variables are referred as Network parameters. After the affine transformation is applied, the result is provided as the input of an activation function, resulting in the output of the hidden unit. This function is an element-wise non linear relation $g(z)$ and it's characteristic of each hidden unit. Some examples of activation functions are provided in the following paragraphs.

Rectified linear units Rectified Linear Units (ReLU) are characterized by the activation function $g(z) = \max\{0, z\}$. In the context of gradient based optimization, the ReLU activation function provides the derivatives to be non negligible for whenever the unit is active. Furthermore the use of this function helps address some arising problems in training deep learning models, such as vanishing gradients, due to the consistency of the derivatives. In addition the ReLU activation function avoids introducing second order effects in the optimization process, due to the seconds order derivative being null almost everywhere. ReLU are usually applied on top of an affine transformation as following:

$$h = g(W^T x + b) = \max(0, W^T x + b) \tag{3.3}$$

A representation of the ReLU function is given in Fig. 3.2

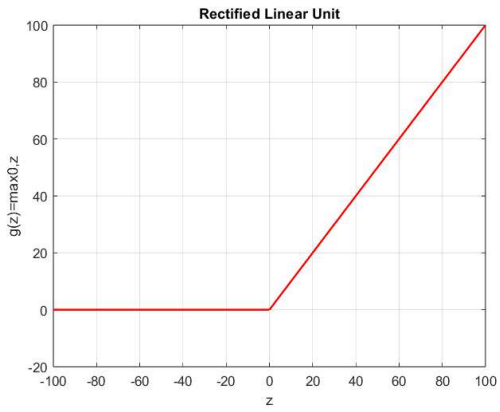


Figure 3.2: Representation of ReLU function

Logistic Sigmoid and Hyperbolic Tangent These two activation functions were more commonly used before the introduction of the ReLU activation function. The sigmoid function is defined as follows:

$$g(z) = \sigma(z) \tag{3.4}$$

and the hyperbolic tangent function:

$$g(z) = tanh(z) \tag{3.5}$$

Such functions are closely related to each other since it holds the relation $tan(h) = 2\sigma(2z) - 1$ between the two. Furthermore they share some simi-

larities in the behaviour and, unlike ReLU, they saturate across most of their domains due to their particular shape. The function is strongly sensitive to the input only for values of z close to zero. This particular behaviour makes gradient-based learning difficult to perform with these functions and thus their use is discouraged in most applications. A graphical representation of these two particular activation functions is provided in Fig. 3.3.

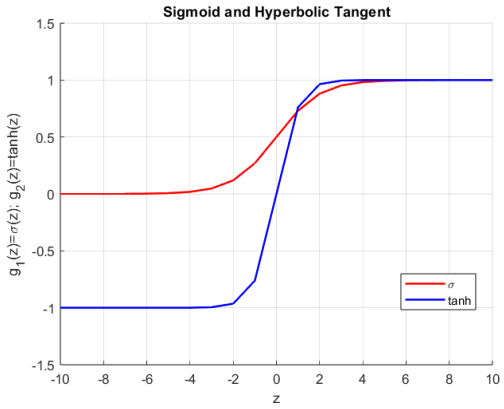


Figure 3.3: Sigmoid and Hyperbolic Tangent

3.1.2 Gradient Based Learning

Gradient based learning is at the core of training deep learning models and can be applied after an estimate of the gradient is used to shift the parameters towards a global minimum of the cost of the Network. As already stated for FFNNs the computation of the gradient is performed by the back propagation algorithm, explained in the following paragraphs.

Back Propagation: When a FFNN accepts an input x and produces an output $\hat{y}$, information propagates through the network, performing **forward propagation**. Alongside the forward propagation pass it is associated the cost of the Network as function of the Network parameters $J(\Theta)$.

This measure is used to pass information from the last layer to the first layer through the process of **back-propagation** (*Rumelhart et al. [18]*), allowing to compute the gradient in a simple and inexpensive procedure when compared to evaluating it numerically.

It is important not to mix the back-propagation algorithm and the optimization of the Network. In fact the process of back-propagation is only a part of

the learning procedure, being responsible of solving only the task of estimating the gradient at that specific iteration of the optimization procedure. Other algorithms such as **gradient descent** and **stochastic gradient descent** are responsible of updating the **weights** and **biases** defining the FFNN, exploiting the gradient and thus performing the optimization step.

Recalling now the *Chain Rule of Calculus* in vectorial form, assuming $\mathbf{x} \in \mathbb{R}^m$, $\mathbf{y} \in \mathbb{R}^n$ and the relations $g : \mathbb{R}^m \to \mathbb{R}^n$, $f : \mathbb{R}^n \to \mathbb{R}$ it holds that:

$$\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j} \frac{\partial y_j}{\partial x_i} \tag{3.6a}$$

or in generalized vector notation:

$$\nabla_{\mathbf{x}} z = \left(\frac{\partial \mathbf{y}}{\partial \mathbf{x}} \right)^T \nabla_y z \tag{3.7}$$

The concept can be extended to tensors, providing some further insights in understanding the same concept applied to CNN. To denote the chain rule with respect to operations on tensors, assuming $\mathbf{X}$ being the notation for which we refer to a tensor and $\mathbf{Y} = g(\mathbf{X})$, $z = f(\mathbf{Y})$ being the relations in question; it holds that:

$$\nabla_x z = \sum_j \left(\nabla_{\mathbf{X}} Y_j \right) \frac{\partial z}{\partial Y_j} \tag{3.8}$$

In order to proceed in the explanation of the back-propagation algorithm it is necessary to address some notation for clarity purposes.

Consider a layer of index l. The weight associated to the linear transformation that is applied to the output of the node of index k belonging to the $l-th$ layer, producing the input of the node j of the layer of index $l+1$, is denoted as w^l_{jk}. Regarding the bias contribution before applying the activation function, the notation is b^l_j.

Assume now to have a cost function J of the network, the derivative of such function with respect to the weights and the biases, following the chain rule, can be written as:

$$\frac{\partial J}{\partial w^l_{jk}} = \frac{\partial J}{\partial z^l_j} \frac{\partial z^l_j}{\partial w^l_{jk}} \tag{3.9}$$

But since z^l_j is the input of the activation function of node j of the layer $l+1$,

it is possible to write it as:

$$z_j^l = W_j^l a^{l-1} + b_j^l = \sum_{k=1}^{m} w_{jk}^l a_k^{l-1} + b_j^l \tag{3.10}$$

with a^{l-1} being the output of the layer of index $l-1$, decomposed in the individual contributions of the m nodes. By differentiating it holds that:

$$\frac{\partial z_j^l}{\partial w_{jk}^l} = a_k^{l-1} \tag{3.11}$$

obtaining the final value:

$$\frac{\partial J}{\partial w_{jk}^l} = \frac{\partial J}{\partial z_j^l} a_k^{l-1} \tag{3.12}$$

The same reasoning can be applied to the biases obtaining:

$$\frac{\partial J}{\partial b_j^l} = \frac{\partial J}{\partial z_j^l} \tag{3.13}$$

The term common for both expression (3.12) and (3.13) is often called **local gradient** and it's expression is the following:

$$\delta_j^l = \frac{\partial J}{\partial z_j^l} \tag{3.14}$$

which can be obtained with the chain rule recalled earlier. By computing the gradient from the output layer to the first hidden layer an indication on how much the weights and biases should change in order to reduce the cost, is obtained. These adjustments, as stated earlier, are performed by an optimization algorithm such as gradient descent or stochastic gradient descent.

Gradient Descent and Stochastic Gradient Descent The gradient of the cost of the Network highlights the direction of steepest ascend of the measure. Therefore the optimization is performed towards the opposite direction thus moving to a global minimum in an iterative way. The magnitude of the step is defined by the learning rate α, an hyper-parameter defining how much the parameters are adjusted towards the steepest descent direction. This process can be executed for the whole training set for each iteration and the algorithms responsible of

performing such task are called **batch** or **deterministic** gradient methods. Optimization applied to the whole training set, although providing a more precise estimate of the gradient, can be computationally costly for large datasets. The exact opposite approach is to compute the gradient only for a singular instance of the training set at the time, resulting in faster computations but with worse precision in the estimate of the gradient. Such methods are called **stochastic**. Most DL models are trained with a procedure that is an in-between approach, namely the training set is divided in groups of more than one instance and the gradient is computed for every batch providing a sweet spot in the tradeoff of precision and computational speed. Such methods are called **minibatch stochastic** methods. There are some consideration to address for such optimization method.

- The batch size affects the estimate of the gradient, in fact the larger the batch the more precise the gradient estimate is. On the other hand the improvement is not linear with respect to the increment in batch size thus the batch size should be chosen wisely.
- If the examples in the batch are processes in parallel, the amount of memory required scales with the size of the batch. This could be a limiting factor for many hardware setups.
- For some hardware it is convenient to choose the batch size to be a power of 2. Typical powers of 2 chosen, range between 32 to 256 with some larger models benefiting of a batch size of 16.
- The advantage of smaller batches is the introduction of a regularization-like effect. To maintain stability smaller batches are associated with a smaller learning rate.

The sampling of the mini-batches is required to be randomly performed since the computation of the gradient should be based on independent training samples. Furthermore two subsequent minibatches should also be independent from each other since two subsequent gradient estimates should be also independent from each other. The gradient descent performed in such way is called **minibatch stochastic gradient** descent or just **Stochastic Gradient Descent (SGD)**.

A crucial parameter for the SGD algorithm is the learning rate α, since for such algorithm the learning rate should be reduced as the Network cost converges to a minimum.

Another important parameter, that reduces the learning time, is the **momentum** μ. Momentum is responsible of solving two problems, namely the ill-conditioning of the Hessian Matrix and variance of stochastic gradient. The momentum is a measure that can be interpreted as a directional speed in the parameters space, thus a larger value is helpful in avoiding local minimum that would cause the optimization algorithm to get stuck at a sub-optimal solution.

3.1.3 Convolutional Neural Networks

CNNs are a particular application of FFNN, where the input data are images. This approach has become widely used in image pattern recognition tasks given that it introduces some considerable advantages when compared to more traditional visual pattern recognition algorithms.

One of the main advantage introduced by these networks, is the overcoming of the issue of feature points descriptors defined by a human operator. In fact one of the strength of CNN is the capability of learning pattern features directly from training data on it's own. Furthermore the intuitive approach of transforming image information in linear vectors and considering such vectors as input data of a traditional FFNN, would cause the loss of spatial information that are essential in images.

In CNN the input of a layer is a single value obtained through spatial convolution over a neighborhood in the output of the previous layer, therefore the term "convolutional" in the denomination. The convolutional operator applied to images performs a sum of products between pixel values and a set of kernel weights, over every spatial location in the image. To conceptualize this in the sense of FFNN think of such a value being the output of a single hidden unit in the network. Now by adding a bias and passing the result through an activation function a complete analogy between CNN and traditional FFNN is obtained.

Given an input image, a *receptive field* of the image is selected, we assume it to be a square $k \times k$ window of pixels with k being an odd number. Starting from the top-left corner $(x, y) = (0, 0)$, the window can be centered in the pixel $(\lfloor k/2 \rfloor, \lfloor k/2 \rfloor)$. The pixels values in that window are then multiplied by a *kernel*, which is a set of weights shaped like the receptive field. The resulting sum of the previous multiplications is then added to a bias and the passed as input of an activation function to generate a single value. Such value is then considered as the value of the position $(0, 0)$ of the resulting 2D array called *feature map*. This

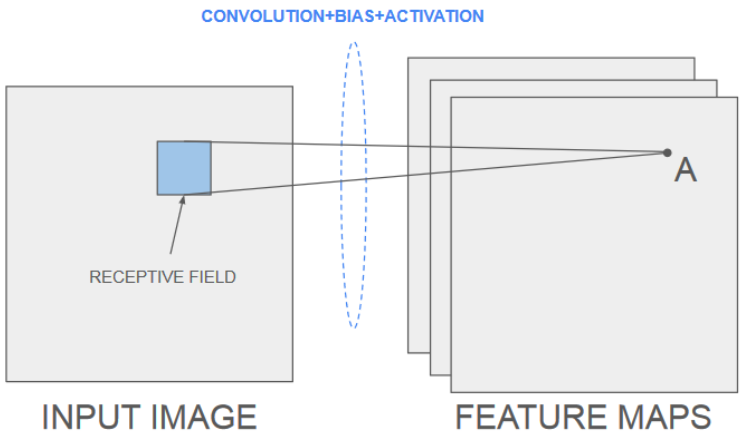


Figure 3.4: Convolutional layer visualization

operation causes a reduction in the shape of the *feature maps* when compared to the dimension of the input image; such secondary result could be either advantageous or undesired depending on the application.
The same weights and bias are shared through all convolutions performed on each input image location, since every kernel can be associated with a particular feature extracted (such as edges, corners or blobs) and thus having the same "feature extractor" throughout the full image is a desirable trait; such concept is called *weight sharing*. Furthermore the convolution with a kernel is not necessarily performed for each adjacent pixel, instead the window slides with a *stride*, skipping some pixels and thus obtaining a reduction in size. For every layer of the network a different numbers of different kernels can be used. At each stage there is a feature map associated with each kernel. The set of maps belonging to the same layer are collectively referred as *convolutional layer*. A visual representation of the convolutional layer is shown in Fig. 3.4, where the square in blue highlights the receptive field and the point **A** represents the correspondent point in the feature map, resulting from the convolutional operation. The convolutional operator, combined with the addition of the bias and the application of the activation function, is drawn as a dotted ellipse for simplification purposes. After convolution and activation most CNN require a subsampling or pooling layer. This layer works analogously with respect to a convolutional layer with the difference that the resulting value is selected with a criterion and the pooling is performed on adjacent pixels.

The pooling methods are various, some examples are:

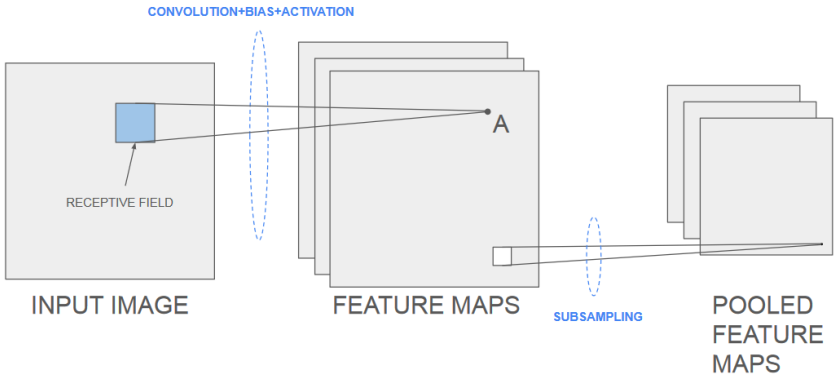


Figure 3.5: Visual representation of a Pooling Layer

- **average pooling:** The values in neighborhood are replaced with the average value of the neighborhood
- **max-pooling:** In max-pooling the resulting value is obtained by taking the maximum value of the neighborhood.

Pooling associates a pooled feature map to every feature map resulting from a convolutional layer thus building a *pooling layer*. A visual representation of the pooling process is shown in Fig. 3.5, where the pooling operation maps the values from the white window into a single value resulting from the pooling criterion chosen.

Consider the feature maps after the first convolutional layer composed of N two dimensional feature vector. Consider also that the second convolutional layer generates k feature maps. The question that arises is determining how many sets of kernels and biases are needed to construct the feature maps at the second layer. In fact if we want to pass through another convolutional layer after the first one and a subsequent pooling layer, it holds that every 2D array in the feature map is convolved with a set of k filters generating k responses. The convolution is performed with different sets of k kernels characteristic of the individual feature map resulting from the first convolutional layer. This means that a total of $N \times k$ different sets of kernels and biases are needed, thus generating $N \times k$ responses. In order to obtain a feature maps composed of k 2D array at the second convolutional layer the responses of the same kernel index in the set are added together. This process is resulting from the linearity of the convolution operation, thus the superposition effect holds (Fig. 3.6).

After that the same pooling procedure is applied thus completing the second

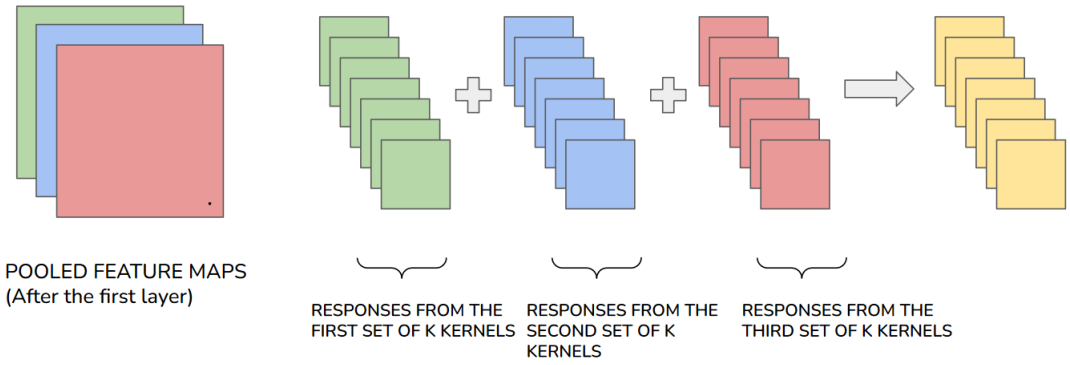


Figure 3.6: An example of Convolutional layer after the first one

layer of the network.

So far the convolutional layer discussed addresses the feature extraction phase of the Network. These features maps are then used to produce some kind of response useful to perform a given task. One of the first tasks for which CNNs were implemented is the image classification task, thus a classifier needs to be placed in the network. This is done by feeding the resulting pooled maps of the last layer of the CNN into a fully connected FFNN. In order to have suitable input for a FFNN that performs classification, there is the need to vectorize the 2-D pooled feature maps by concatenating every pooled feature map to form a column vector. Such input then propagates through the FFNN, with the last layer having a node for each class to classify. The node that outputs the highest value indicates which class the input image belongs to. Such implementation is not only characteristic of classification tasks as it will be shown in the following paragraphs, as the number of nodes in the output layer can be chosen to solve different tasks. The complete representation of a CNN can be seen in Fig. 3.7

3.1.4 Backpropagation for CNNs

The equation related to the forward pass through a CNN are similar to the one introduced in the FFNN section, with the only difference being the product operator is replaced by the convolutional operator. Consider a kernel w and an input array $a^{\ell-1}_{x,y}$, the convolution results in $z_{x,y}$ being:

$$z^{\ell}_{x,y} = \sum_{l}\sum_{k} w^{\ell}_{l,k} a^{\ell-1}_{x-l,y-k} + b^{\ell} \tag{3.15}$$

$$= w^{\ell} * a^{\ell-1}_{x,y} + b^{\ell} \tag{3.16}$$

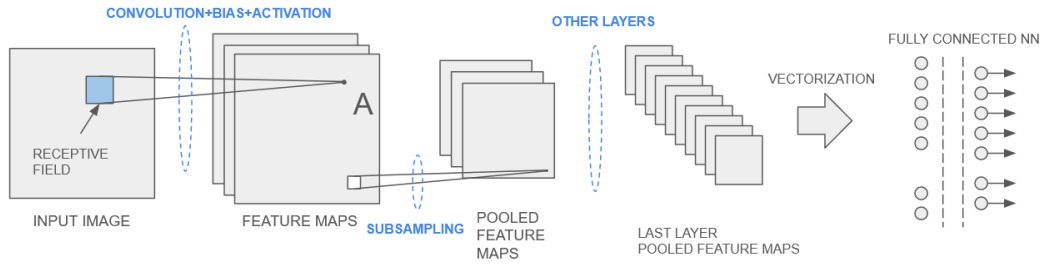


Figure 3.7: Structure of a complete CNN with convolutional layers, pooling layers and FFNN

where l and k are the indexes spanning the kernel dimensions, x and y are the indexes spanning the dimensions of the input vector a and b is the bias. The result of the convolution after the activation function is:

$$a^{\ell}_{x,y} = h(z^{\ell}_{x,y}) \tag{3.17}$$

Now it is possible to explicitly write the equations for back-propagation in CNNs. Consider the output error on a batch of p instances, produced by a CNN to be denoted as E (this could be for example the Mean Squared Error), the goal is to estimate how this error changes with respect to the shifting parameters in the Network, thus obtaining an estimate of the gradient useful to perform an optimization step with a gradient descent approach. For notational purposes we denote as H and W the dimensions of the input map and assume the dimension of the kernel as $k_1 \times k_2$.

$$\frac{\partial E}{\partial w^{\ell}_{x,y}} = \sum_{i=o}^{H-k_1} \sum_{j=0}^{W-k2} \frac{\partial E}{\partial z^{\ell}_{i,j}} \frac{\partial z^{\ell}_{i,j}}{\partial w^{\ell}_{x,y}} \tag{3.18}$$

$$= \sum_{i=o}^{H-k_1} \sum_{j=0}^{W-k2} \delta^{\ell}_{i,j} \frac{z^{\ell}_{i,j}}{\partial w^{\ell}_{x,y}} \tag{3.19}$$

where $\delta^{\ell}_{i,j}$ is a therm analogous to the **local gradient** introduced in the section regarding the back propagation algorithm of FFNN. Recalling now that $z^{\ell}_{i,j}$ is the result of the previous convolution, as shown in (3.15), it is possible to

differentiate (3.19) with respect to $w_{x,y}$ resulting in (3.19) becoming:

$$\frac{\partial E}{\partial w^{\ell}_{x,y}} = \sum_i \sum_j \delta^{\ell}_{i,j} a^{\ell-1}_{i-x,j-y} \tag{3.20}$$

When compared with the equation from the forward propagation in holds that the indexes are changed in sign. Thus meaning that if we want to express (3.20) as a convolution between two terms it results in:

$$\frac{\partial E}{\partial w^{\ell}_{x,y}} = \sum_i \sum_j \delta^{\ell}_{i,j} a^{\ell-1}_{-(x-i),-(y-j)} \tag{3.21}$$

$$= \delta^{\ell}_{x,y} * a^{\ell-1}_{-x,-y} \tag{3.22}$$

$$= \delta_{x,y} * rot_{180}(a^{\ell-1}) \tag{3.23}$$

where rot_{180} is an operator that rotates by 180° the output of the previous layer $a^{\ell-1}$. Similarly the effects of the biases on the error results in:

$$\frac{\partial E}{\partial b^{\ell}} = \sum_i \sum_j \delta^{\ell}_{i,j} \tag{3.24}$$

Exploiting the equations found a step performed by a gradient descent algorithm on the CNN parameters is applied:

$$w^{\ell}_{x,y} = w^{\ell}_{x,y} - \alpha \frac{\partial E}{\partial w_{x,y}} \tag{3.25}$$

$$= w^{\ell}_{x,y} - \alpha \delta^{\ell}_{x,y} * rot_{180}(a^{\ell-1}) \tag{3.26}$$

$$b^{\ell} = b^{\ell} - \alpha \frac{\partial E}{\partial b^{\ell}} \tag{3.27}$$

$$= b^{\ell} - \alpha \sum_x \sum_y \delta^{\ell}_{x,y} \tag{3.28}$$

3.1.5 Additional remarks on designing CNN

One of the main parameters in designing the structure of a CNN is to decide how many hidden layers should a Network have. Theoretically networks with a simple structure are quite powerful, following from the *Universality ap-*

proximation theorem, that states that under mild conditions a simple network with a single feedforward network with a single hidden layer can approximate arbitrarily complex decision functions.

From experimental results it was shown that deep neural networks (namely with two or more hidden layers) perform better than single layer neural networks in learning abstract representations. There is no systematic approach in deciding how many layers should compose a network. Therefore, specifying the number of layers is generally performed exploiting experience and experimentation.

Adding layers that are not strictly necessary to the operations of the neural network can lead to problems in training, in fact with higher numbers of layers the back propagation algorithm could run into some issues like *vanishing gradients*, namely gradients become so small that the gradient descent step ceases to be effective.

Another important issue, which was already introduced in the previous sections, is the decrease in size of the feature maps as the information passes through the network. If such secondary effect is undesirable the input vectors can be either *padded* in order to reach the desired size.

In addition to the number of layers there is the need to decide the number of *neurons* (FFNN) or *kernels* (CNN). Such number has no theoretical optimum value either but, following the objective of keeping the number of layers the lowest possible, the number of *kernels* should increase to increase the power of the Network.

Another aspect to decide in the structure of the neural network is the activation function characteristic of each layer. Most networks rely on a ReLU activation function but some applications show good performances using either the sigmoid or the hyperbolic tangent function. Once the network architecture is specified the fundamental aspect of making an architecture useful is the training of such network. Such structures can be constructed by millions of parameters and require a considerable amount of time to train. To accelerate matrix operations most networks are trained by exploiting GPUs. Training a CNN could arise some issues. One of the most common ones is *over-fitting*, which results in acceptable performance in the training set but poor performances for instances not yet encountered by the Network. This means that the network is not capable to *generalize* what has learned and apply it to never seen before input instances. Therefore generalization can be achieved by having the largest training set possible, which sometimes is not feasible to either construct

or obtain. A solution could be to artificially enlarge the training set instances by applying some transformations, such as geometric distortions and intensity variations. Another approach applied often to avoid over fitting is to use the *dropout* technique, which is performed by randomly omitting some connections between nodes during training. Furthermore, to decrease the time needed for training the network, there are some additional procedures that can be implemented, such as shuffling the training set at each epoch to avoid the **cycling** effect arising when features are repeated at regular intervals.

To conclude this practical focus on the designing and training of CNNs it is necessary to state that, in order to make the training procedure faster, mini-batch stochastic gradient descent can be implemented. This results in faster convergence when compared to optimizing among the whole training set a the same time, for each epoch.

3.2 Cross Scene Crowd Counting Via Deep CNNs - Zhang et al.

3.2.1 INTRODUCTION

Following the work of Lempitsky the work presented by Zhang et al. [22] introduced a Deep CNN model capable of recreating density maps useful to estimate the number of person in a crowded images. The basic goal of this approach is the same as the one presented in [13], namely to learn a suitable mapping $\mathcal{F} : \mathcal{X} \rightarrow \mathcal{D}$, where $\mathcal{X}$ is a set of low-level features extracted from the training images, $\mathcal{D}$ is the crowd density map of the image.

Consider now that the training set is labeled analogously to the annotations introduced in Ch.(2), namely point annotations are associated to each position of an individual in the image. The training samples are built by randomly extracting patches from the training images and retrieving the corresponding density map as ground truth. The same applies to the density maps ground truth, thus for every point in the annotation file a normalized density distribution is superimposed, as it is shown in Fig. 3.8. To estimate the number of people in the image (as a decimal number) the same property of density maps applies, namely such value can be obtained by integration.

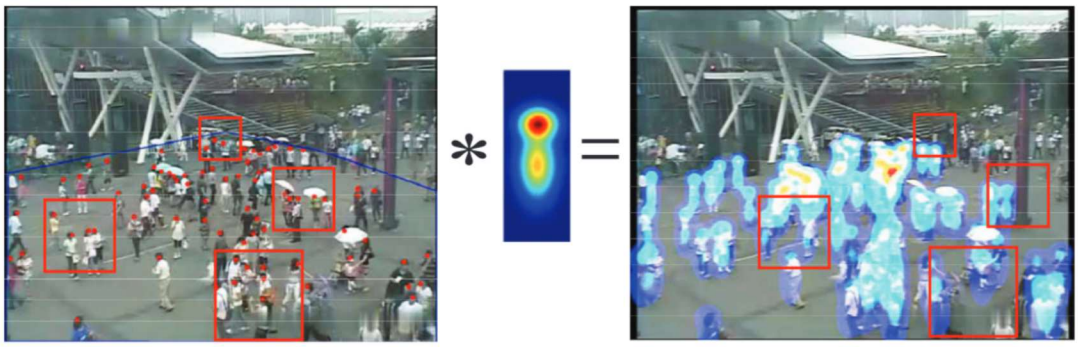


Figure 3.8: Density map generation shown in the paper [22]

It is to denote that the shape of the distribution used in this work is related to the shape of the human body and, due to the choice of placing the point annotation on on the head of the subject, centered on the head of each individual in the image. In addition to this, such distribution is scaled following the perspective warping of the image. This means that the mathematical expression

of the density maps ground truth is the following:

$$\mathcal{D}_i(p) = \sum_{P \in \mathbf{P}_i} \frac{1}{||Z||} (\mathcal{N}_h(p; P_h, \sigma_h) + \mathcal{N}_b(p; P_b, \Sigma)) \tag{3.29}$$

where $\mathcal{N}_h$ is the normal distribution representing the head and $\mathcal{N}_b$ is the one representing the body. The parameters P_h and P_b are the position of the head and the body of the person. The variances of these two normal functions are dependent on the perspective value, namely $\sigma_h = 0.2\,M(p)$, $\sigma_{bx} = 0.2\,M(p)$, $\sigma_{by} = 0.5\,M(p)$ where $M(p)$ is the value of the perspective map at pixel p.

3.2.2 Network structure

In this section it is discussed the structure of the network introduced by the paper. Given an source image a region of dimensions $3m \times 3m$ is selected considering the perspective maps measurements. This section is warped into a $72\,pixels \times 72\,pixels$ which is the input of the custom CNN.

The structure of the network is composed by three convolutional layers and a fully connected FFNN composed of other three layers. The first convolutional layer manages to produce a feature map composed by 32 2D arrays of dimensions 72×72, thus meaning that such layer is composed of 32 filters of dimensions $7 \times 7 \times 32$, where a padding of size 3 is applied to the input image in order to obtain such a size of convolved feature maps. After the first convolutional operator a max pooling layer is places, with kernel size 2×2, which reduces the convolved maps into 32×36×36 pooled feature maps. Subsequently the network adopts an activation layer defined by a ReLU activation function. The placement of the activation layer after the pooling layer instead of before is a design choice that doesn't affect the correctness of the Network, in fact this choice can also bring some advantages like reducing the instance size before computing the activation layer result, thus reducing computational time.

The second convolutional layer receives the output of the first layer and produces feature maps at the second stage by convolving the 32×36×36 pooled feature maps from the first layer with 32 sets of 7×7×32 kernels. Analogously to what operations were performed before performing convolutions with these kernels, the feature maps were padded with a padding size of 3. This results in the convolved feature maps resulting from the second convolutional layer being of size 32×36×36. The same max pooling layer and ReLU activation layer are

also present after the second convolutional layer. This block of layer produces a 32×18×18 pooled feature vector at the second stage.

The third convolutional layer is composed of 64 sets of 5×5×32 kernels resulting, with a padding of 2 applied before hand, in a convolved feature vector at the third stage of dimensions 64×18×18.

At this point the convolved feature maps at the third stage comprehend the features extracted from the convolutional part of the network. Following the reasoning applied in Section 3.1.3 such convolved maps should be adapted to be suitable for the FFNN part of the Network. This means that the feature maps are re-shaped to a column vector of dimensions 20736×1. The first layer of the fully connected part of the Network reduces the size of the feature vector to 1000×1, the second layer transforms the output of the first fully connected layer to a column vector of size 400×1.

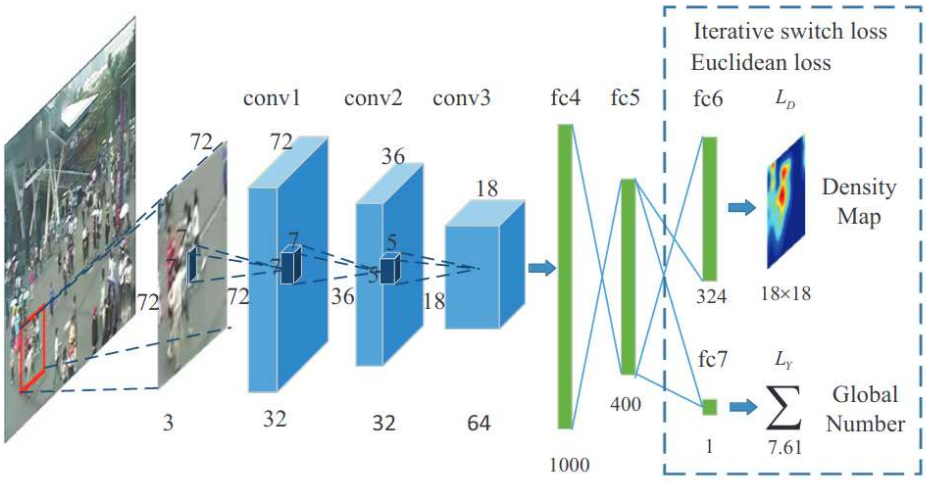


Figure 3.9: Representation of the Crowd Counting Network Structure found in [22].

Regarding the last layer there is more to address since it depends on the training procedure. Since the primary goal of the Network is to estimate the total number of people in the image the last layer could be implemented as a single node which output would be the total count. On the other hand if the goal is to obtain the density map estimate the last layer is built considering an input of size 400×1 and outputs a column vector of size 324×1. Such column vector represents the density map reshaped as a column vector, this means that the Network outputs a map 18×18 from a window of size $72pixels \times 72pixels$

once the column output is reshaped accordingly. The full representation of the model adopted in [22] is shown in Fig. 3.9.

Such particular structure follows from the training procedure implemented, in fact the paper suggest an **iterative switching learning process** that alternatively optimizes two different objective, namely the density map generation and the total count of individuals in a patch. The two optimization task alternatively assist each other, thus obtaining a better solution. The two loss functions adopted are represented in (3.30) and (3.31).

$$L_D = \frac{1}{N}\sum_{i}^{N} ||F_d(X_i;\Theta) - D_i||^2 \tag{3.30}$$

$$L_Y = \frac{1}{N}\sum_{i}^{N} ||F_y(X_i;\Theta) - Y_i)||^2 \tag{3.31}$$

with Θ referring to the parameters of the CNN model, D_i the ground truth density map of the training instance i, Y_i the ground truth count, $F_d(\cdot,\cdot)$ being the output of the density network and $F_y(\cdot,\cdot)$ the estimated count. Both measures are based on the Euclidean distance and the loss is minimized using stochastic gradient descent and back propagation.

The switching learning procedure starts by optimizing L_D as the first objective loss to minimize, due to density maps bringing more spatial information to the CNN model. Once the network cost converges, the training procedures changes function to optimize the global count regression.

The minimization of the count difference L_Y function converges faster with respect to the L_D loss function and after convergence is obtained the learning algorithm switches the objective in minimizing L_D again.

The pseudocode associated to the learning procedure is shown in Algorithm 5.

Algorithm 5 Training with iterative switching losses

Require: Training Set consisting of patches and density maps
Ensure: Θ set of parameters of the CNN
1: L_D set as the objective function
2: **for** $t = 1$ **to** T **do**
3: **while** loss drop rate $\Delta L > \epsilon$ **do**
4: Backprop. to learn Θ
5: **end while**
6: Switch objective loss function
7: **end for**

3.3 Application to the washer counting task

In this section we discuss the implementation of the CNN introduced in the previous sections to the washer counting task in exam. The code defining the Network and the iterative switching learning procedure was implemented by exploiting the PyTorch library. The execution of such code was performed in Google Colaboratory environment (also known as Colab). The reasons for such choices are many, such as:

- **Ease of use:** The PyTorch library offers high-level interface that highly simplify the implementaton of FFNN and CNN. This results in the code to be easier to define, modify and debug with respect to static graphs framework
- **Better Numpy integration:** The two Python libraries are seamlessly integrated with each other, allowing to convert easily from PyTorch tensors and Numpy array
- **Community and Documentation:** The large community of users, and the large amount of tutorials combined with a rich documentation offer a vast support to implement and train complex Networks
- **Research-Oriented**: The flexibility combined with the ease of use made PyTorch popular among researchers.

On the other hand there are some disadvantages that PyTorch can introduce due to it's simplicity of use when compared to other environments like TensorFlow, which is one of the most used library for Deep Learning tasks. Such disadvantages are:

- **Performance Overhead:** PyTorch's dynamic computation graph leads to slightly slower execution when compared to static graphs representation provided by TensorFlow
- **Not Ideal for production development:** Deploying PyTorch models in industrial setups require more effort when compared to other frameworks like TensorFLow.

Regarding the choice of writing the code in Google Colab to train the CNN in exam, it was performed based on the following advantages that the platform provides:

- **Free GPU:** Google Colab offers free access to GPU resources which provide an accelerated training of CNNs when compared to using CPUs. Furthermore the use of GPUs admits the possibility of training with larger Datasets without the need of powerful local hardware.
- **Pre-Installed libraries:** Popular Deep Learning libraries are pre-installed in Colab like TensorFlow, Keras and PyThorch. Furthermore other commonly used libraries are also installed like NumPy and matplotlib.
- **Cloud Storage and integration:** Due to the seamlessly integration with Google Drive it is easy to access and save files directly from the cloud. Such features allows to store Datasets, model checkpoints and other project files.

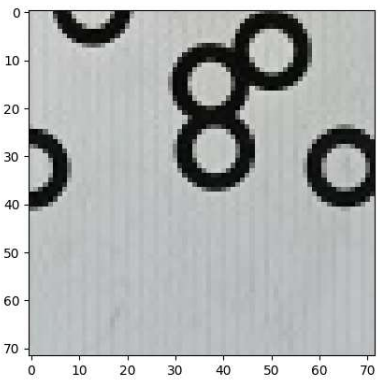
3.3.1 Dataset and Training Instances

The training set built in Section 2.2 can be easily transposed to be suitable to the DL solution presented. Consider the already **cropped** and **down-sampled** source images and ground truth densities. Starting from the original dataset composed of 2543 image-density pairs, the dataset is artificially augmented to enlarge the number of training instances. The only transformations applied to the training instances pairs is a rotation of 180°, thus doubling the size of the dataset. Such choice is arbitrary and can be replaced by other transformations like flipping the image in both axis. On the other hand such transformations would not be applied on this solutions, since the rotation would be sufficient for the convergence of the model.

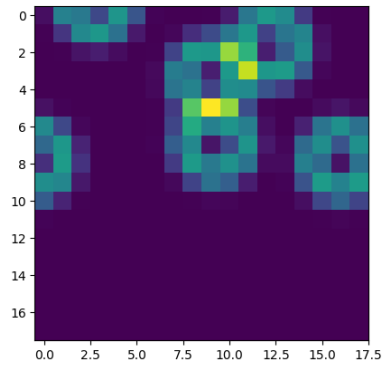
Due to the objects being approximately at the same distance from the camera it holds that there is no need to perform some perspective normalization like it was done in the paper. From this reason it follows that the training snippets of images can be obtained by simply selecting a $72pixels \times 72pixels$ regions in the image, subsequently such regions are transformed into Tensors.

Regarding the target densities some additional transformation are performed to be consistent with the Network output. After transforming the Numpy array defining the ground truth density map into a PyTorch tensor, cropping in the same 72×72 window, a down sampling is required. The down-sampling procedure applied is the same that was addressed previously and it can be visualized in Fig. 2.10.

Although the down sampling follows the same procedure, the scaling factor is different. In fact since the goal of this transformation is to obtain ground truth maps that are sized accordingly to the output of the CNN, a 4×4 window was used instead of the 5×5 used for down sampling the density maps in the Machine Learning solution. This means that the ground truths density are down sampled into 1×18×18 tensor. An example of a input/target pair can be seen in Fig. 3.10.



(a) Example of input window



(b) Down-sampled target density

Figure 3.10: Washer counting Network training instance

3.3.2 Implementation of the CNN and training procedure

The implementation of the Network also required some modification with respect to the one presented in the paper. In fact a direct implementation of the network would lead to *vanishing gradients* during training with the washer

dataset. For the same reason the densities were scaled by multiplying the density maps by a constant scalar that would increase the response of the Network during training and thus lead to gradients with substantial magnitude to perform gradient optimization.

The other main change introduced was the implementation of a layer normalization [2] step after every convolution step. Normalization layers in general have shown to greatly affect performances on training time. The objective of layer normalization is normalizing the activation of the layer on every single training instance in the mini batch. The resulting CNN is structurally very similar with respect to the model suggested in [22] with the added normalization layers represented in Fig. 3.11 in red, placed between the convolutional layer and its respective pooling layer.

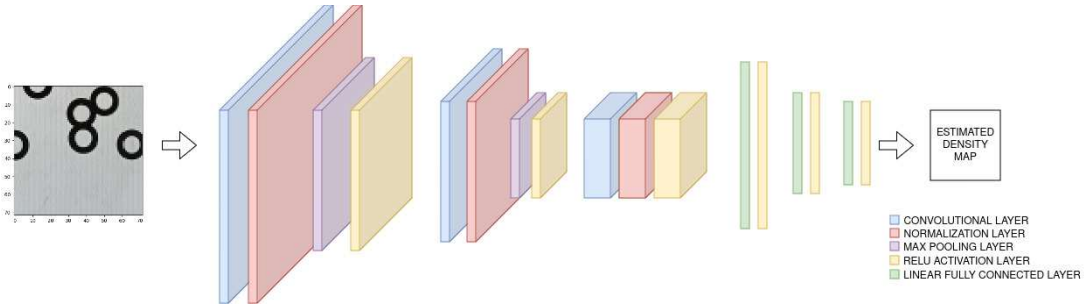


Figure 3.11: Structure of the actual CNN trained for the washer counting task

The full dataset is thus split into a training set and a small validation set. The training set is composed by 5056 images and densities pairs.
The small validation set is composed of 30 pairs, and it is useful just to understand how the model would perform on never seen before data.
Another difference from the paper is the choice of loss function. In fact the procedure of switching loss functions was kept but instead of using the Euclidean distance from the ground truth count the choice for the second loss function fell on the L1 distance. The choice of such loss function was performed on the basis that the main objective of the solution implemented is not related directly in obtaining the ground truth count, instead retrieving a as refined as possible density map. Thus meaning that the two loss function adopted are the

following:

$$L_D^2 = \frac{1}{N}\sum_{i=1}^{N} ||F_d(X_i;\Theta) - D_i||^2 \tag{3.32}$$

$$L_D^1 = \frac{1}{N}\sum_{i=1}^{N} |F_d(X_i;\Theta) - D_i| \tag{3.33}$$

Where for both loss measures $F_d(X_i;\Theta)$ is the estimated density maps that the model produces for each batch and Θ is the Network parameters at the current iteration. The other therms are D_i which is the corresponding ground truth density for each window and N the number of samples composing each batch.

The training was performed with stochastic gradient descent, meaning that the training set was sub-divided in mini-batches of 32 training pairs. Since the batch size is quite small then a smaller learning rate is preferable in order to facilitate convergence to the optimum, thus a value of $\alpha = 0.0005$ was chosen. such choice was empirical since for such value of convergence rate the Network would show some improvements in learning fundamental features for the task, while for larger values of the learning rate the training measures would not improve even after several epochs.

Regarding the number of epochs for the first training loss function a maximum number of $epochs = 11$ was chosen, such choice was also empirical and was made after a first iteration of the algorithm showed that for fewer epochs the model showed evident improvements even at the last epoch. It is to denote that such choice would not result in the over-fitting of the Network since, before the switching is performed, the model that shows the best validation result is saved as a starting point for the next optimization. The last parameter for training the Network is the momentum of the gradient descent algorithm which is large to avoid local minimum where the algorithm would stop to a sub-optimal solution, namely $\mu = 0.9$. The training hyper-parameters are summarized in Tab. 3.1
The parameters of the Network can be initialized by following the Kaiming Normal initialization which is an initialization method that works best for ReLU activation layers, with the goal of keeping gradients at a reasonable range.

As already stated the first part of the training process is performed by using the L2 loss. For the first 200 iterations of the learning algorithm it was observed that there wasn't any noticeable improvement in reducing the loss function. In fact despite variations in the input data the network's predictions showed erratic

$n°$ epochs	11
α	0.0005
μ	0.9

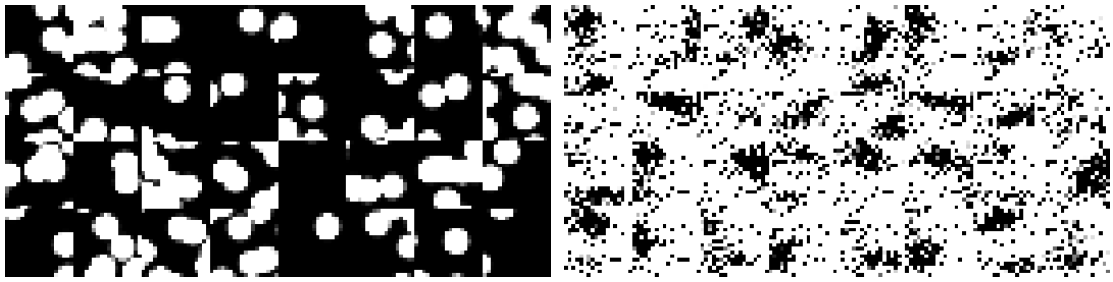
Table 3.1: Training hyper-parameters

fluctuations and lacked meaningful convergence towards a desired output. This behaviour is shown in Fig. 3.12

(a) Ground Truth Densities (b) Network Output during Training

Figure 3.12: Comparison between ground truth and output at itaration 155

After the first 200 iterations, a significant improvement in the training output was observed. Such behaviour shows that the model around the 200th iteration started learning meaningful features and patterns from the input data and adapt the Network parameters accordingly. This improvement can be seen in the training output as shown in Fig. 3.13, where empty regions started showing in the training output in correspondence of empty regions in the ground truth densities.



(a) Ground Truth Densities (b) Network Output during Training

Figure 3.13: Comparison between ground truth and output at iteration 315

As the training process progresses it can be seen that the training error has a converging behaviour as expected. The fluctuations in the training loss

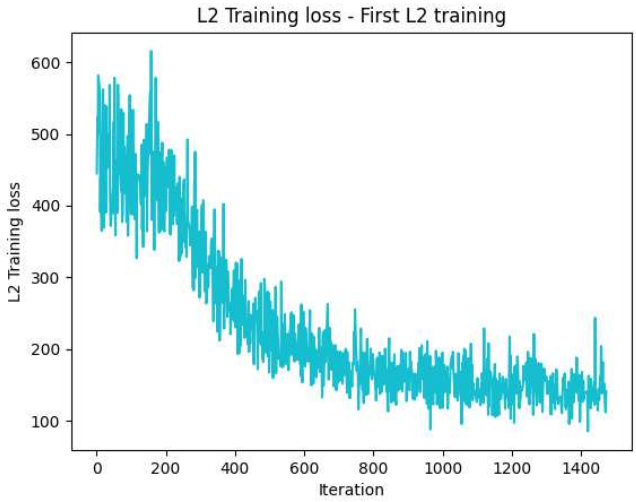


Figure 3.14: Training Error - L2 loss

are expected and follow the behaviour that the Stochastic Gradient Descent optimization method produces. The evolution of the training guesses can be seen in Fig. 3.16, where it is highlighted that the guesses become more consistent with the desired output. The first training process was stopped at the ninth epoch due to the slowing improvements in training error and the model chosen for the subsequent training with L1 loss was the one with the best validation error, namely the one resulting from epoch 7. The validation error evolution can be seen in Fig. 3.15 and shows how for each epoch before the 7th such measure decreases as expected.

The evolution of the L1 error (Fig. 3.19) after the switch of loss function shows that, by maintaining the same training hyper-parameters, a decreasing behaviour exists only on the initial phase. After a slight decrease in training error the convergence rate slowed significantly, becoming almost stationary.

By focusing on the Validation Error (Fig. 3.18), it shows that the model obtained after the second epoch displays an inconsistent decrease in the magnitude of such error and after the fifth epoch the validation error has an increasing behaviour. On the other hand by letting the training process complete the 11 epochs it holds that the minimum value of such error was reached at the very last epoch. This yields that the starting model for the next switching of loss function to be the one resulting from the 11th epoch. The Training performed with the L1 loss, although it didn't affect the shape of the maps much, it helped in reducing the random "noise like" peaks in the estimated density maps, as it can

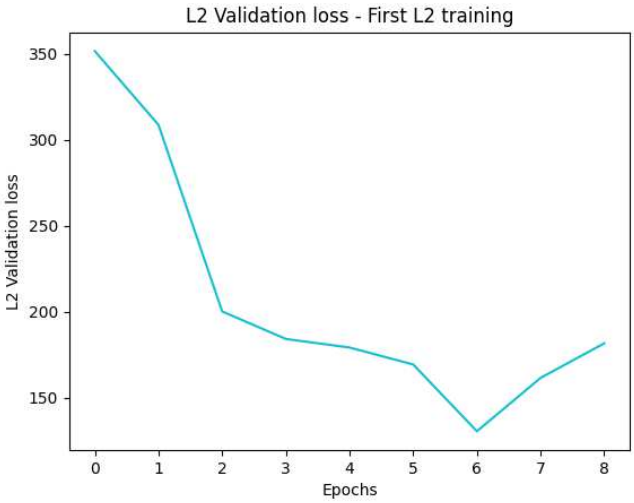


Figure 3.15: Validation Error - L2 loss

be seen in Fig. 3.19. As a result of this second training it holds that true empty regions in the density maps ground truth are represented as empty regions in the Network output, behaviour that can also be highlighted in Fig. 3.19.

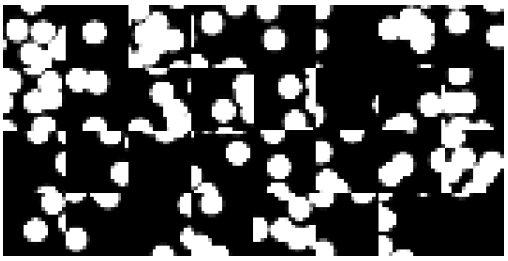
The second phase of the training procedure, namely the second "switching" of loss function from L1 to L2 measure, was performed with some additional remarks. In fact in order to try and improve the training performance the training hyper-parameters were slightly tweaked. To improve the training error the learning rate α was increased to be $\alpha = 0.001$. In addition the momentum parameter was slightly lowered to be $\mu = 0.6$ to avoid bringing too much inertia in the learning process (Tab. 3.2). Such choice of parameters greatly improves

n° epochs	11
α	0.001
μ	0.6

Table 3.2: Training hyper-parameters for second L2 and subsequent training

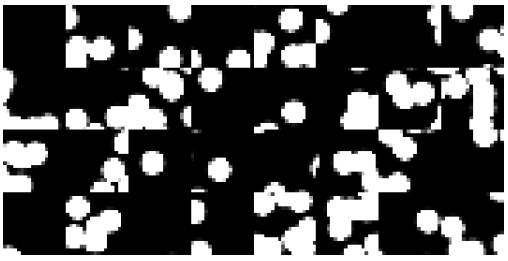
the convergence of the model towards a minimum of the L2 training error whose behaviour can be highlighted in Fig. 3.20. Furthermore this part of the training improves the performance on the validation set as it is highlighted in Fig. 3.21 On the other hand such training introduces again "noise like" artifacts as can be highlighted in Fig. 3.22.

For this reason the subsequent (and last) training phase with L1 loss was im-



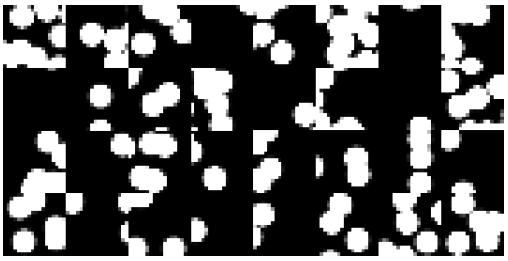
(a) Ground Truth Densities - iteration 610

(b) Network Output during Training - iteration 610

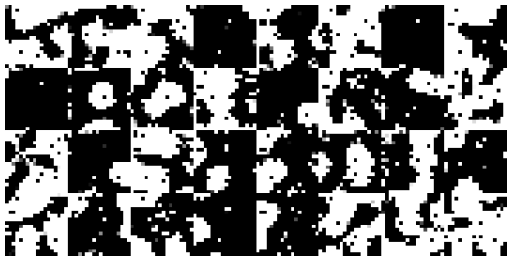


(c) Ground Truth Densities - iteration 705

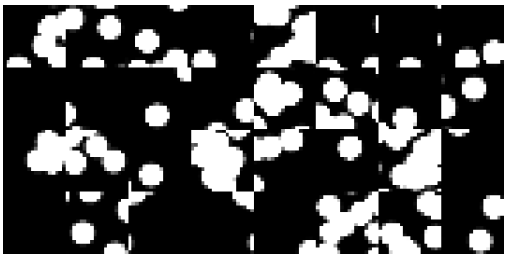
(d) Network Output during Training - iteration 705



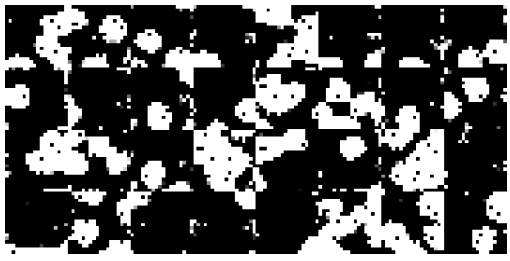
(e) Ground Truth Densities - iteration 1075



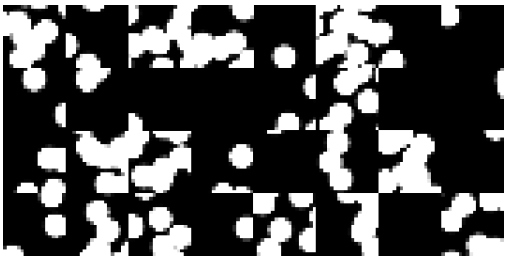
(f) Network Output during Training - iteration 1075



(g) Ground Truth Densities - iteration 1155



(h) Network Output during Training - iteration 1155



(i) Ground Truth Densities - iteration 1305

(j) Network Output during Training - iteration 1305

Figure 3.16: Evolution of the guesses with respect to he Ground Truth during training

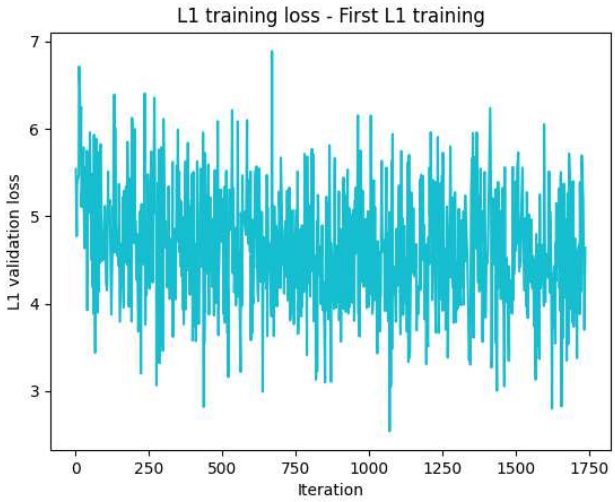


Figure 3.17: L1 Training Error - First L1 training

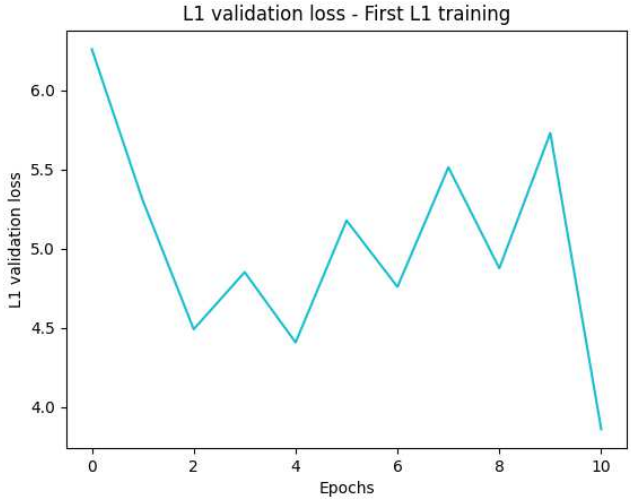
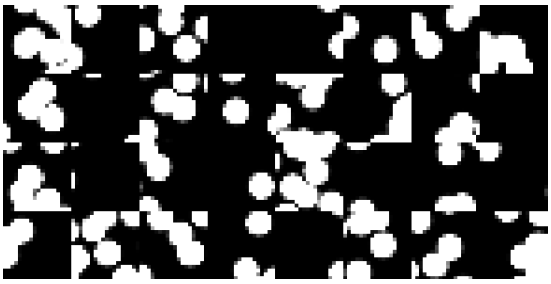
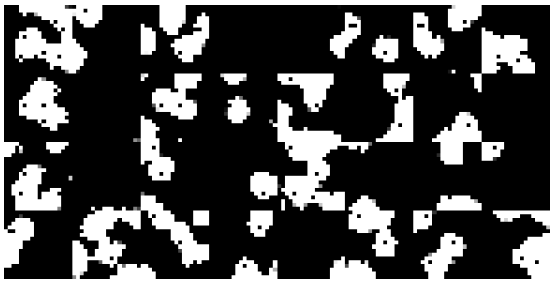


Figure 3.18: L1 Validation Error - First L1 training

plemented, in order to obtain a cleaner result in the representation of the output density maps. The last training phase showed a fast convergence regarding the training loss, as expected and analogously to the first training procedure performed with the L1 measure. In fact the Training L1 loss reaches it's minimum regarding the second training with such loss at around 100 iterations (or in less than one epoch). On the other hand the training was performed fully and by checking the validation error it was proved that the best validation result was provided by the model derived in the last training epoch of the algorithm as highlighted by the evolution of the validation error in Fig. 3.24.

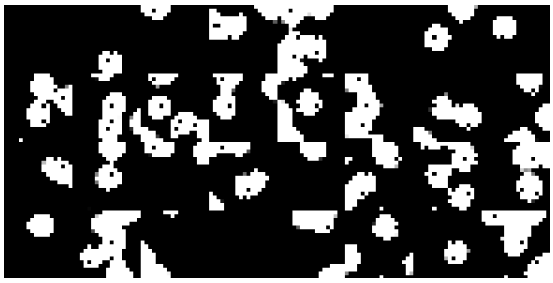


(a) Ground Truth Densities - iteration 170

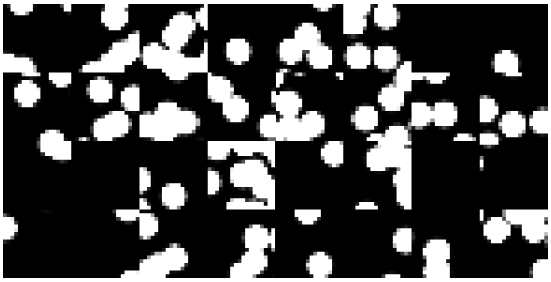


(b) Network Output during Training - iteration 170

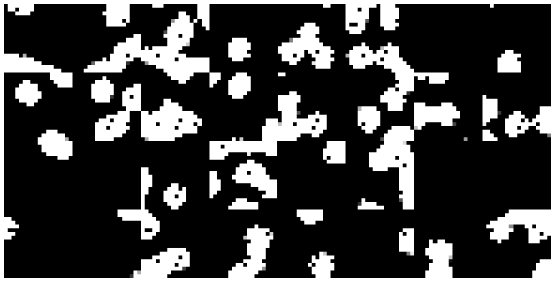
(c) Ground Truth Densities - iteration 340



(d) Network Output during Training - iteration 340



(e) Ground Truth Densities - iteration 650



(f) Network Output during Training - iteration 650

Figure 3.19: Evolution of the guesses with respect to he Ground Truth during training with L1 loss

The results in training instances of the second training performed with the L1 loss show how the Network is capable of reducing the noise like peaks if such loss is adopted. Results can be seen in Fig. 3.25.

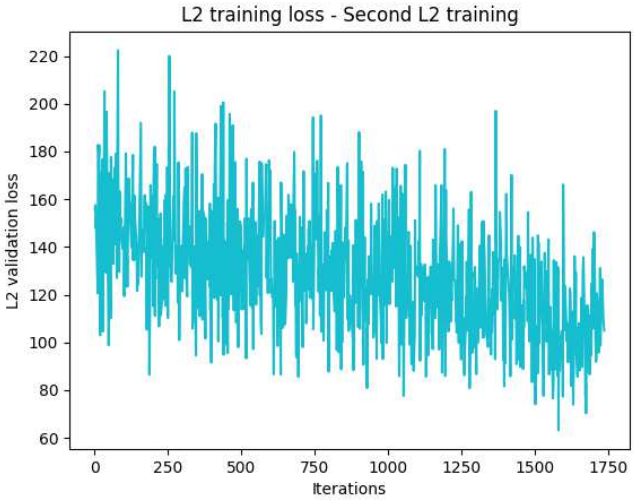


Figure 3.20: L2 Training error - Second L2 training

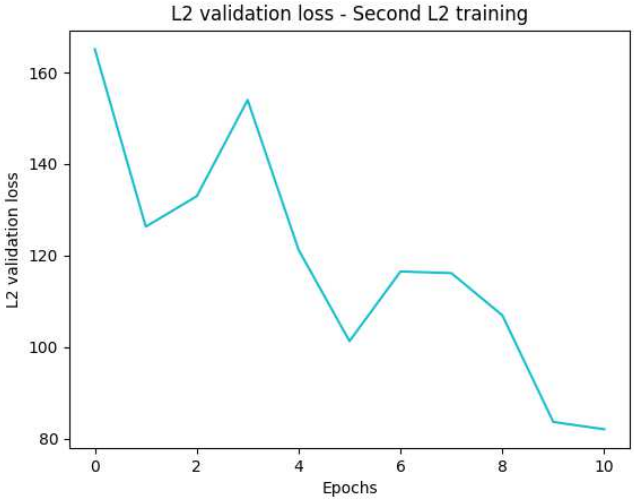
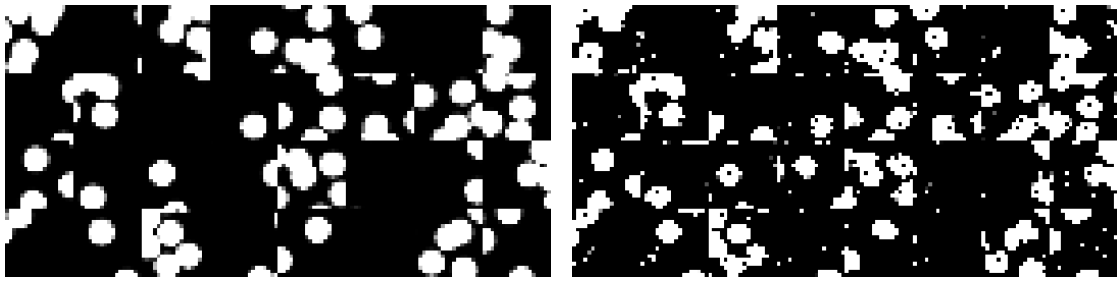


Figure 3.21: L2 Validation Error - Second L2 training



(a) Ground truth (b) Estimated density maps

Figure 3.22: Output density maps with the best model after second L2 training

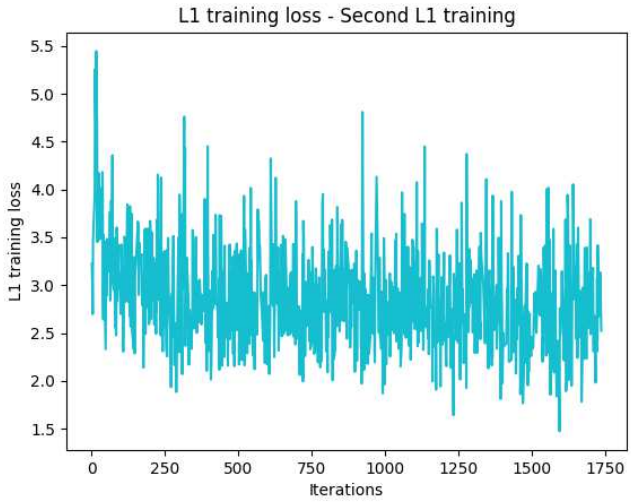
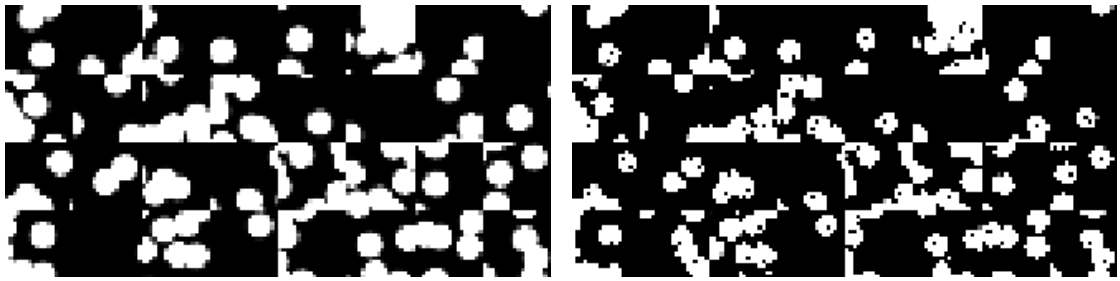


Figure 3.23: L1 Training Error - Second L1 training

Figure 3.24: L1 Validation Error - Second L1 training

Regarding the results of the final Trained Network with respect to the instances belonging to the validation set it holds that the ground truth count and the estimated count in every window is comparable. The best results are obtained with isolated objects where the estimated count is almost exact, however for the instances where objects are overlapped there is a noticeable difference in the estimated count with respect to the ground truth count, although not large.

Furthermore one could state that by comparing such results with the detection based methods this algorithm performs better since even if severely occluded object are present in the frame they will still partially add to the over-



(a) Ground Truth density batch (b) Estimated density batch

Figure 3.25: Density map comparison at the end of the training procedure

all count. Meanwhile for the detection based methods if an object is not detected due to being too hidden it will not be considered in the overall count. Results on some validation examples are shown in Fig. 3.26.

It is important to denote that after each switch of the loss function adopted the training set is re-shuffled and sampled again on a random window thus providing new data for the Network to learn, effectively enlarging the dataset further.

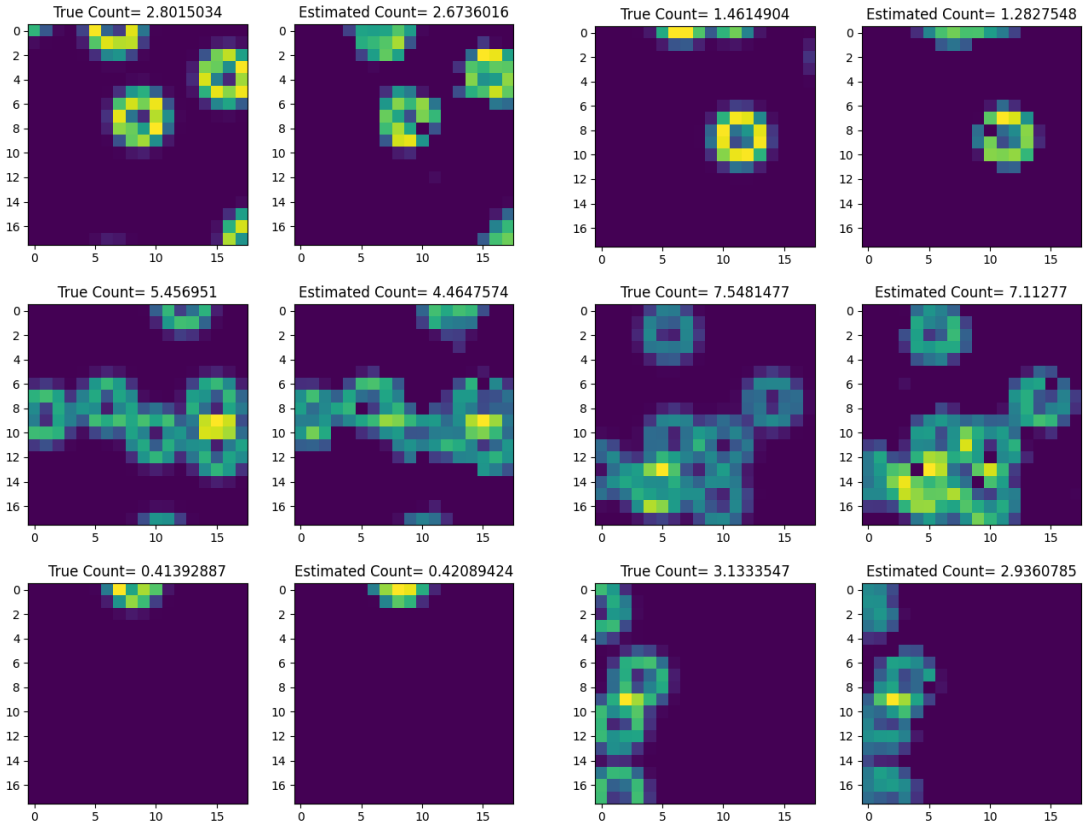


Figure 3.26: Validation results with count comparison between ground truths (left) and estimates (right)

4

Implementation on a Real Setup

Both solutions showed to be accurate in both estimating the ground truth density map and the total count of objects in frame. The structure of the density estimation system would have the same three steps for both solutions, namely:

- **Step 1:** Washers are dropped on the first stage of the conveyor belt from the vibrating sift. The washers travel along the whole conveyor belt eventually reaching the back-lit part.
- **Step 2:** The camera, mounted on top of the back-lit part of the conveyor belt, samples the image with a sampling time depending on the computational capability of either the smart camera or the external computational device
- **Step 3:** The image is cropped and resized in order to obtain input samples having the same characteristics of the Washer Dataset.

Once these initial steps are performed the following processes are dependent on the solution adopted.
Regarding the ML solution the whole image can be considered as input instance and computations can be performed on the whole frame at one time. The image is then converted into grey-scale due to the Dense SIFT algorithm requiring a single channel image.

Dense SIFT descriptors having the same dimensions of the input frame are computed, subsequently the encoding process based on the k-means dictionary quantizes the feature vector to reduce its dimensions. Finally the feature map transforms the feature vector into the estimated density map.

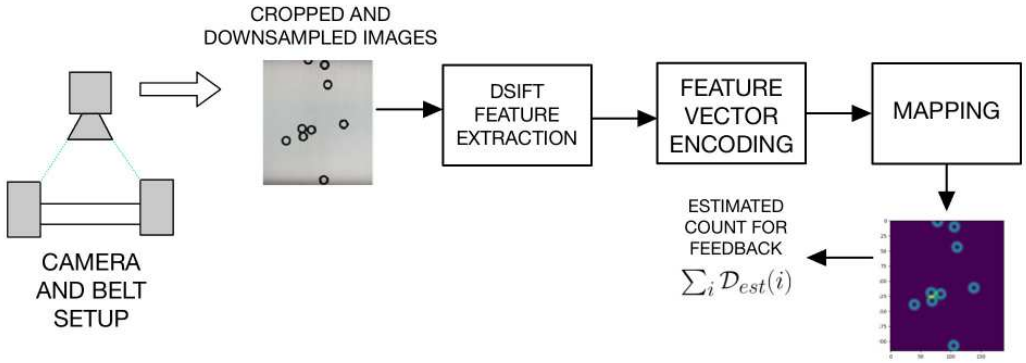


Figure 4.1: Structure of the Machine Learning solution

It is to denote that such reasoning is based on the MATLAB implementation of such solution. On the other hand a comparable, if not faster, solution written in C++ can be obtained. However this implementation demands a custom implementation of the Dense SIFT feature extraction and the subsequent encoding if needed. A graphical interpretation of the Structure of the Machine Learning solution implemented is shown in Fig. 4.1.

Regarding the CNN solution further steps are applied on the sampled image in order to generate feasible inputs for the Input layer of the washer counting CNN. In fact such solution requires the input window of being of dimensions $72\,pixels \times 72\,pixels$ thus the input image needs to be pre-processed further.

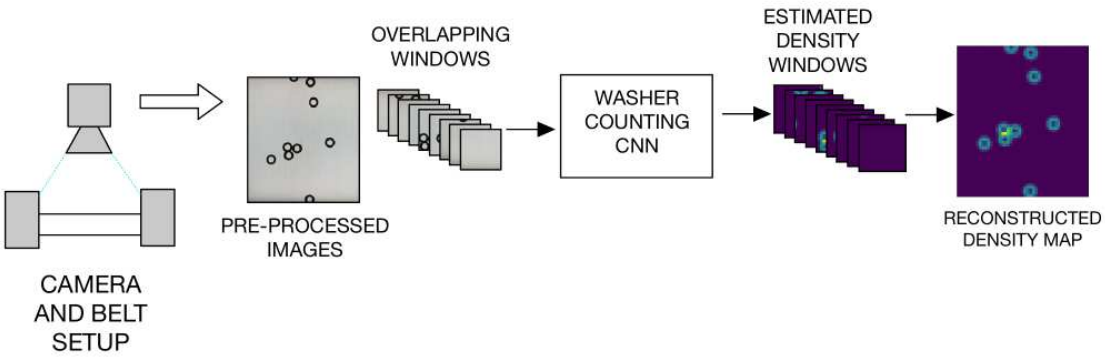


Figure 4.2: Structure of the CNN solution

The pre-processed image of dimensions $216\,pixels \times 188\,pixels$ is divided into 9 overlapping windows of dimensions $72\,pixels \times 72\,pixels$ accordingly with the input instances that the Network allows. Overlapping of the windows is not inherently an issue as long as, once the corresponding estimated density windows are obtained, they are treated accordingly by considering the over-

lapping regions of the estimated density maps only once. The structure of the whole density map estimation is illustrated in Fig. 4.2.

4.1 Performance comparison and applicability

Regarding the precision of both solutions on the estimated count, the best between the two has proved to be the Machine Learning one. In fact on average it provided a count difference in never seen before input instances of 0.47 washers per image, which compared with the 1.74 of the Convolutional Neural Networks solution shows the better performance of the first. On the other hand the ML solution is relatively slow when compared to the other solution. The feature vector generation process is mostly responsible of a "bottle neck" effect that greatly slows down the computation. In fact even by exploiting the optimal algorithm provided in the VLFeat library the computation of the feature vector took on average 0.8s, accounting also for the encoding process. For such reasons such application can introduce some limitations in a industrial setup and can be better suited is some tasks that don't require real time counting but need to be precise.

Regarding the computational time required by the CNN solution, the Network took 0.95s to compute the respective density of 120 windows. This amounts to a total time of 0.071s per full image without considering the generation of the overlapping windows and the reconstruction of the density maps. Keep in mind that these results are obtained in the same environments on which the respective solutions are trained and thus there would be an hardware specific contribution in the results. The results are summarized in Tab. 4.1

	CNN Solution	**ML Solution**
Average count error	1.74 washers	0.47 washers
Time performance	~0.07s	>0.8s

Table 4.1: Comparison between the two solutions in therms of computational time and count error.

Furthermore the CNN solution has also another advantage when compared with the ML solution. In fact the first is of easier implementation in a real setup due to the camera of choice being better suited to run Deep Learning models after a suitable conversion, thus being an almost a ready to use solution.

Regarding the ease of training of the respective implementations, it can be highlighted that the Machine Learning model has an advantage when compared to the CNN implementation. Indeed the convergence on the optimal solution of the regularized quadratic programming problem was obtained with a reduced Dataset and in very few iterations of the solver, thus proving to be faster in the training process.

Another important aspect that needs to be highlighted is the feature engineering aspects of the whole process. In fact for the Machine Learning solution a certain level of domain knowledge is needed to choose wisely the engineering of the feature descriptor that would be suitable for generating the density maps. On the contrary the Deep Learning solution doesn't require any engineering on the feature descriptors due to the model being capable of learning salient feature on it's own. Furthermore the model complexity introduced by the Deep Learning application, is better suited in learning intricate patterns and relationship between the data.

Accounting for everything stated so far, it is fair to state that the better suited model in the washer counting task would be the Convolutional Neural Network solution due to it's versatility and ease of implementation, in addition to the real-time potential application. Furthermore the promising transfer learning properties of such implementation can be advantageous in application of other setups.

4.2 Future works

The work presented in this thesis has provided valuable insights in the strength and weaknesses of the more traditional Machine Learning approach and the more contemporary Convolutional Neural Network solution.

Several developments following this study can be implemented in order to further address the implementations of such solutions in diverse scenarios and tasks. In fact there is the possibility of exploring the performance metrics (in both inference time and precision) of these two algorithms with other state of the art approaches like the ones addressed in the introduction, being several and based on different approaches.

Further investigation could arise in the structural implementation of the two: For instance one could focus on the fastest and suitable dense feature extraction algorithm that could provide increase performances of the Machine

Learning approach. In the case of the Convolutional Neural Network approach one could analyze and modify the structure of the Network in order to improve performances and reduce the size of the overall structure removing superfluous layers depending on the task. Furthermore one could test the improvements in performance that eventually other cost function would bring in the training of the CNN solution. Moreover the transfer learning capabilities of the Convolutional Neural Network model need to be tested on various scenarios, effectively proving such property.

In addition one could test whether or not the precision of the Machine Learninìg model can be useful in an hybrid solution, paired with a state of the art detection algorithm. For instance consider a YOLO Network that can detect reliably most of the instances of objects in the image. The objects instances that are not detected can become the input of the Machine Learning procedure that can fine tune the guess of the detection based algorithm.

Finally it could be wise to test the performances of such algorithms in a real time industrial setup, providing insights on the actual applicability in a production scenarios.

References

[1] Mohamed Abdou and Abdelkarim Erradi. "Crowd Counting: A Survey of Machine Learning Approaches". In: *2020 IEEE International Conference on Informatics, IoT, and Enabling Technologies (ICIoT)*. 2020, pp. 48–54. DOI: 10.1109/ICIoT48696.2020.9089594.

[2] Jimmy Ba, Jamie Kiros, and Geoffrey Hinton. "Layer Normalization". In: (July 2016).

[3] Jon Bentley. "Programming pearls: algorithm design techniques". In: *Commun. ACM* 27.9 (Sept. 1984), pp. 865–873. ISSN: 0001-0782. DOI: 10.1145/358234.381162. URL: https://doi.org/10.1145/358234.381162.

[4] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Berlin, Heidelberg: Springer-Verlag, 2006. ISBN: 0387310738.

[5] Stephen Boyd and Lieven Vandenberghe. *Convex optimization*. Cambridge university press, 2004.

[6] Antoni B. Chan, Zhang-Sheng John Liang, and Nuno Vasconcelos. "Privacy preserving crowd monitoring: Counting people without people models or tracking". In: *2008 IEEE Conference on Computer Vision and Pattern Recognition*. 2008, pp. 1–7. DOI: 10.1109/CVPR.2008.4587569.

[7] Ke Chen et al. "Cumulative Attribute Space for Age and Crowd Density Estimation". In: *2013 IEEE Conference on Computer Vision and Pattern Recognition*. 2013, pp. 2467–2474. DOI: 10.1109/CVPR.2013.319.

[8] Ke Chen et al. "Feature Mining for Localised Crowd Counting". In: Jan. 2012. DOI: 10.5244/C.26.21.

[9] Rafael C. Gonzalez and Richard E. Woods. *Digital image processing*. Upper Saddle River, N.J.: Prentice Hall, 2008. ISBN: 9780131687288 013168728X 9780135052679 013505267X. URL: http://www.amazon.com/Digital-Image-Processing-3rd-Edition/dp/013168728X.

[10] Ian J. Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. Cambridge, MA, USA: MIT Press, 2016.

[11] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning*. Springer Series in Statistics. New York, NY, USA: Springer New York Inc., 2001.

[12] Reinhard Klette. *Concise Computer Vision - An Introduction into Theory and Algorithms*. Undergraduate Topics in Computer Science. Springer, 2014, pp. 1–413. ISBN: 978-1-4471-6319-0. URL: http://dx.doi.org/10.1007/978-1-4471-6320-6.

[13] Victor Lempitsky and Andrew Zisserman. "Learning To Count Objects in Images". In: *Advances in Neural Information Processing Systems*. Ed. by J. Lafferty et al. Vol. 23. Curran Associates, Inc., 2010.

[14] D.G. Lowe. "Object recognition from local scale-invariant features". In: *Proceedings of the Seventh IEEE International Conference on Computer Vision*. Vol. 2. 1999, 1150–1157 vol.2. DOI: 10.1109/ICCV.1999.790410.

[15] David G. Lowe. "Distinctive Image Features from Scale-Invariant Keypoints". In: *International Journal of Computer Vision* 60 (2004), pp. 91–110. URL: https://api.semanticscholar.org/CorpusID:174065.

[16] Luxonis. "Hardware Documentation". In: n.d. URL: https://docs.luxonis.com/projects/hardware/en/latest/.

[17] Joseph Redmon et al. "You Only Look Once: Unified, Real-Time Object Detection". In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016, pp. 779–788. DOI: 10.1109/CVPR.2016.91.

[18] David E Rumelhart et al. "Backpropagation: The basic theory". In: *Backpropagation*. Psychology Press, 2013, pp. 1–34.

[19] Mauro dos Santos de Arruda et al. "Counting and locating high-density objects using convolutional neural network". In: *Expert Systems with Applications* 195 (2022), p. 116555. ISSN: 0957-4174. DOI: https://doi.org/10.1016/j.eswa.2022.116555. URL: https://www.sciencedirect.com/science/article/pii/S0957417422000549.

[20] Shai Shalev-Shwartz and Shai Ben-David. *Understanding Machine Learning - From Theory to Algorithms.* Cambridge University Press, 2014, pp. I–XVI, 1–397. ISBN: 978-1-10-705713-5.

[21] Richard Szeliski. *Computer vision algorithms and applications*. London; New York: Springer, 2011. ISBN: 9781848829343 1848829345 9781848829350 1848829353. URL: http://dx.doi.org/10.1007/978-1-84882-935-0.

[22] Cong Zhang et al. "Cross-scene crowd counting via deep convolutional neural networks". In: *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2015, pp. 833–841. DOI: 10.1109/CVPR.2015.7298684.