POLITECNICO DI TORINO

Master degree course in Mechatronic Engineering

Master Degree Thesis

Applications of Artificial Intelligence and Neural Networks to automatic detection of defects on car bodies

Supervisor:
Prof. Massimo Poncino

Candidate:
Letizia Antonia Vaiana

Co-supervisor:
Prof. Daniele Jahier Pagliari

Internship tutors:
Ing. Alessandro Tessuti
Ing. Cristina Chesta

December 2020

Acknowledgements

Al termine di questo lavoro di tesi, vorrei ringraziare il mio relatore, il prof. Massimo Poncino, sempre pronto a darmi le giuste indicazioni in ogni fase della realizzazione dell'elaborato.

Ringrazio, inoltre, Gerry Nigro, prezioso trade-union tra il mondo universitario e quello aziendale, senza il quale questo lavoro di tesi non esisterebbe nemmeno. Ringrazio tutto lo staff di Reply, in cui ho svolto lo stage formativo, per l'ospitalità e per le skills acquisite sul campo.

Un ringraziamento speciale va a Cristina Chesta, importante punto di riferimento aziendale, per la sua disponibilità e grande professionalità. Mi ha fatto sentire totalmente integrata nel team di lavoro in cui sono stata inserita, creando un clima disteso e formando, così, un gruppo affiatato e produttivo, anche nei mesi in cui, a causa del lockdown, non c'è stata la possibilità di conoscerci e lavorare in presenza, ma solo da remoto.

Un ringraziamento particolare va a Gianluca Moret. Attraverso la sua guida costante e il suo prezioso supporto, ho imparato ad usare tutti gli strumenti di cui avevo bisogno per intraprendere la giusta strada e portare a compimento questo lavoro di tesi.

Infine, ringrazio tutte le persone che mi sono state vicine e mi hanno sostenuto durante il mio percorso universitario, la mia famiglia, i miei nonni e i miei amici che mi hanno sempre supportato, stando al mio fianco in qualunque momento.

Contents

1 Introduction 7
- 1.1 Context and project . . . 7
- 1.2 Challenges and goals . . . 7
- 1.3 Thesis outline . . . 9

2 Background 11
- 2.1 Overview . . . 11
- 2.2 Neural Networks . . . 12
  - 2.2.1 Neuron model and typical activation functions . . . 12
  - 2.2.2 Fully connected Neural Networks . . . 14
- 2.3 Convolutional Neural Network . . . 16
  - 2.3.1 General architecture . . . 16
  - 2.3.2 Convolutional layers . . . 17
  - 2.3.3 Pooling layers . . . 20
- 2.4 Defect detection algorithm . . . 22
  - 2.4.1 Deep learning-based anomaly detection . . . 22
  - 2.4.2 Object detection approach . . . 25
  - 2.4.3 Single Shot Detector . . . 27
- 2.5 Pre-trained models . . . 30
  - 2.5.1 Transfer learning and evaluation metrics . . . 30
  - 2.5.2 MobilNet . . . 31
  - 2.5.3 EfficientDet . . . 33

3 Damage detection 35
- 3.1 Case study system architecture . . . 35
  - 3.1.1 High-level system architecture . . . 35
  - 3.1.2 Acquisition system setup . . . 36
  - 3.1.3 Position of the components . . . 36

3.2 Damage recognition process 38
3.2.1 Functional requirements 38
3.2.2 Damage recognition steps 38
3.3 Defect detection software development 39
3.3.1 Dataset preparation 39
3.3.2 Development tools and frameworks 44
3.3.3 Model training and evaluation 46

4 Experiments and results 51
4.1 Experiments plan introduction 51
4.2 First test dataset: *"Dataset 200"* 51
4.3 Second test dataset: *"Dataset 500"* 52
4.4 Third test dataset: *"Dataset 750"* 53
4.5 Fourth test dataset: *"Dataset 1100"* 57
4.6 Fifth test dataset: *"Dataset 1100_new"* 57

5 Conclusion and future works 61

List of Figures 63

List of Tables 65

Bibliography 67

Chapter 1

Introduction

1.1 Context and project

The digital transformation that our society is undertaking nowadays is driving most companies to change their current business model by applying digital capabilities to products, processes and assets, improving efficiency, improving customer value, managing risk and discovering new income market opportunities.
One of the latest trends is car sharing services as a complementary alternative to the private vehicle. These companies are in continuous technological adaptation and looking for the optimization of resources and processes, to improve the service provided to its customers every day. From here, it is born the idea of an automated solution in new vehicle maintenance services to clean the vehicle and to automatically detect defects on the car body.
The use case analyzed in this thesis, in fact, is part of a wider project called SAROCA (SmArt RObot for CAr-sharing), whose aim is to reduce maintenance costs and optimize workforce for repetitive tasks by involving two independent components: a robot arm for cleaning the vehicle provided by Car Sharing Mobility Services and the object/defect recognition implementation developed by Santer Reply [1].

1.2 Challenges and goals

In details, the area of interest in this thesis is related to the second part of the above-mentioned macro-project (SAROCA), where the real contribution

in the improvement of the operational efficiency in the maintenance processes is given by Artificial Vision. The automatic detection of eventual damages and scratches on the car body is not a trivial task, especially in the case in which they are not so evident and recognizable also by humans, in specific disadvantageous light conditions. Depending on the product under analysis, different types of detection systems are applied, which are based on different kinds of inspection technologies and methods, like visual inspection through cameras and telecentric and pericentric optics or laser-based inspection.

In this use case, the high- level system architecture involves three main components: the acquisition system (cameras, motion detection sensors and video recorder), the processing element (hardware component that allows the execution of the software), the defect detection software (handles the image pre-processing and runs the neural network inference).
Focusing on the third component, through the study of the state of art of AI solutions applied to computer vision, we decided to adopt an object detection approach, by exploiting image recognition techniques. The latter allow us to identify people, places and objects in images or videos by combining the usage of a camera and the implementation of a versatile, self-standing neural network algorithm. Another additional task covered by object detection approach with respect to the other available deep learning algorithms, like semantic segmentation and classification, is the localization of the detected object, that, in this case, represents a specific damage on the car body.

Approaches based on machine learning algorithms and deep neural networks acquired more and more interest during last years, especially with the growing efficiency of the Convolutional Neural Network into extracting complex and abstract patterns from image data. Following this direction, it was conducted a survey in Deep Anomaly Detection (DAD) techniques, evaluating some necessary different aspects: the nature of input data, the availability of labels, the type of defect/anomaly, the output of DAD techniques and their corresponding model architectures.
Approximately, we can divide object detection algorithms into two categories: single-stage detector and two-stage detector. The main difference lies on whether to pool the feature maps for a second stage and, consequently, the different process speed. For this reason and after an accurate comparison between these different typologies of algorithms, also in terms of the most relevant metrics (like Average Precision, Average Recall), it was decided to lean toward the SSD (Single Shot Detector) architecture.

On the other side, one of the crucial parts of building deep learning systems based on neural network algorithm is gathering high quality dataset, since in general the model is as good as the data it learns from. Therefore, a relevant effort during the thesis was dedicated to the research of data: since there is no publicly available labelled dataset for car damage object detection containing suitable images for this goal, it was created a new personal appropriate dataset consisting of images belonging to different sources, that were manually annotated.

During the dataset collection, different pre-trained models were taken into account. The core idea was to perform transfer learning to train the chosen model. That is because typically training from scratch requires a huge amount of data (quite difficult to obtain to solve the project task) and time. Transfer learning, instead, allowed us to start from a pre-trained model: it is a model that has been already trained on a large amount of data (e.g. COCO dataset) and that represents a good and effective starting point to solve the project task. In order to deploy a production model, we decided to rely on the Tensorflow API, developed by Google, that is an open source library that offers the possibility to develop and train machine learning models.

Tests started using a small but fast model SSD MobileNet: this choice is associated with the fact that this model can be implemented in a relatively fast way, thanks to the powerful and efficient frameworks and APIs available in the context of machine learning systems development.
We also consider another typology of approach, EfficientNet model, that has gained the new state of the art accuracy with a relevant difference in terms of architecture size (9.6 times fewer parameters on average), as deepened in Chapter 2.

1.3 Thesis outline

This thesis work is structured as follows:

- **Chapter 2** contains a brief presentation of the main theoretical concepts in literature, which are at the basis of the implemented system;
- **Chapter 3** introduces the case study, providing a brief overview of the whole system architecture and describing the defect detection software through all the steps handled during its development phase;

- **Chapter 4** presents a description of the experiments performed during the model training and the evaluation results obtained;
- **Chapter 5** presents some final considerations and ideas for future works in order to further improve the system.

Chapter 2

Background

2.1 Overview

Artificial Intelligence (AI) is an important stakeholder in the rapid digital transformation that our society is undertaking nowadays. The organizations that want to prepare for an automated future should have a thorough understanding of AI. When we look at artificial intelligence, it can be divided into three different domains: Robotics, Cognitive systems and Machine learning [2]. The latter deals with the information world and, in particular, it is possible to say that machine learning algorithms use computational methods to derive meaning and to "learn" information directly from data without relying on a predetermined equation as a model [3]. A subset of machine learning is deep learning, that achieves good performance and flexibility to represent the data as a nested hierarchy of concepts within layers of the neural network [4]. The following sections contain a presentation of these main theoretical concepts more in details, since they are at the basis of the implemented system: after a brief introduction to the theory behind artificial neurons and neural networks, Convolutional Neural Networks (CNNs) are presented, because they have become one of the most used tool in computer vision field for solving problems of visual recognition, such as object detection.

2.2 Neural Networks

2.2.1 Neuron model and typical activation functions

The area of Neural Networks has originally been primarily inspired by the goal of modeling biological neural systems. The diagram in Figure 2.1 shows a symbolic drawing of a biological neuron (left) and a common mathematical model (right). Each neuron receives input signals from its dendrites and produces output signals along its (single) axon. The axon eventually branches out and connects via synapses to dendrites of other neurons.
In the computational model of a neuron, the signals that travel along the axons (e.g. x_0) interact multiplicatively (e.g. w_0x_0) with the dendrites of the other neuron based on the synaptic strength at that synapse (e.g. w_0). The idea is that the synaptic strengths (the weights w) are learnable and control the strength of influence (and its direction: excitory (positive weight) or inhibitory (negative weight) of one neuron on another.

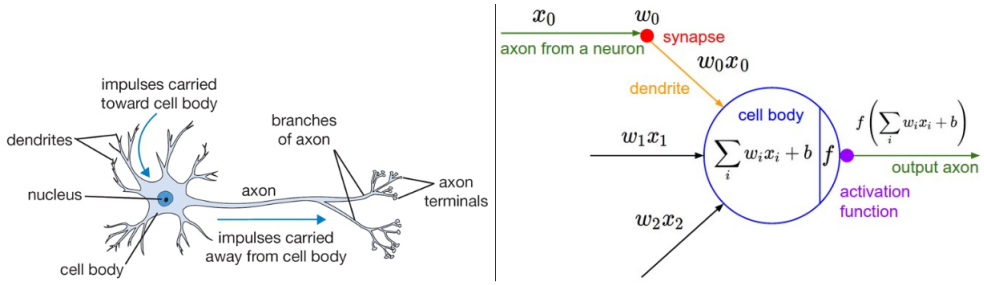


Figure 2.1: Symbolic representation of a biological neuron (left) and its mathematical model (right)

In the basic model, the dendrites carry the signal to the cell body where they all get summed. If the final sum is above a certain threshold, the neuron can fire, sending a spike along its axon. In the computational model, we model the firing rate of the neuron with an **activation function** f, which represents the frequency of the spikes along the axon.

Historically, a common choice of activation function is the **Sigmoid function** σ, since it takes a real-valued input (the signal strength after the sum) and squashes it to range between 0 and 1. Actually, the sigmoid non-linearity has recently fallen out of favor and it is rarely ever used, because it saturates and "kills" gradients: as it is possible to see in the first graph on the left side of Figure 2.2, very undesirable property of the sigmoid neuron is that when

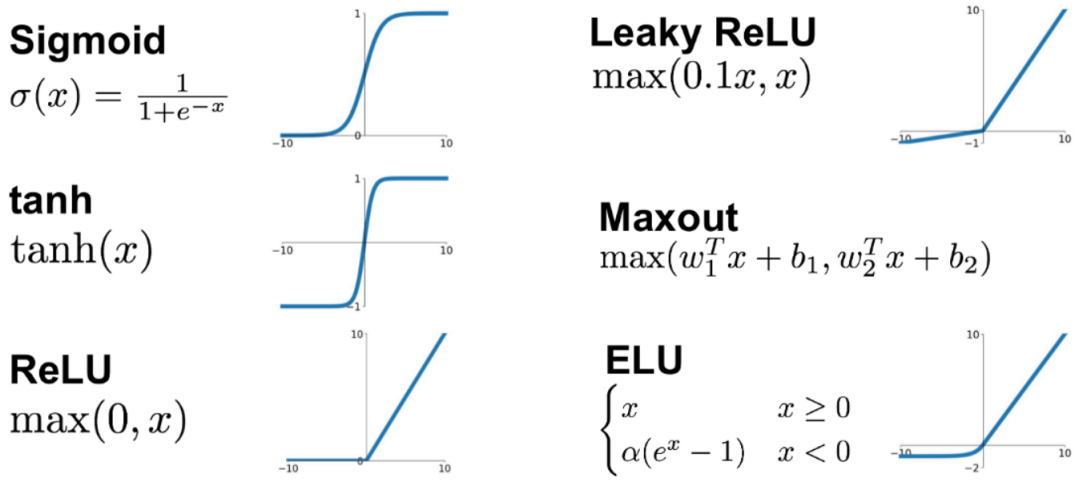


Figure 2.2: Commonly used activation functions

the neuron's activation saturates at either tail of 0 or 1, the gradient at these regions is almost zero [5].
Similarly, the **tanh** non-linearity (shown as second example on the left in Figure 2.2), which squashes a real-valued number to the range [-1, 1], saturates. Instead, a plus point is that, unlike the sigmoid neuron, its output is zero-centered.
Another typology of activation function, the Rectified Linear Unit (**ReLU**), differently from the previous ones, has become very popular in the last few years, because it has several pros with respect to the cons. From its formula and graph in Figure 2.2 (the last one on the left), it is immediate to say that the activation is simply thresholded at zero, it is not computational expensive and it greatly accelerates the convergence of stochastic gradient descent compared to the sigmoid/tanh functions, due to its linear, non-saturating shape. The only disadvantage consists in ReLU units, that could be fragile during training: a large gradient flowing through a ReLU neuron can cause the weights to update in such a way that the neuron will never activate on any datapoint. In this way, the unit will forever be zero from that point on and, consequently, it could irreversibly "dies" [5].
One attempt to fix the "dying ReLU" problem is done by the **Leaky ReLUs** activation functions, where, instead of the function being zero when x $<$ 0, a small negative slope (of 0.01, or so) is introduced, as visible in in the first graph on the right in Figure 2.2.
A generalization of the latter two reported typologies of activation functions

is contained in the **Maxout** neuron (the second one on the right in Figure 2.2), that represents one relatively popular choice nowadays. In it all the positive aspects of both ReLU and leaky ReLU are enclosed, but it doubles the number of parameters for every single neuron, leading to a high total number of parameters [6].
Also the last activation function reported on the right, **ELU** (Exponential Linear Unit - Figure 2.2) is characterized by the same problem: even though it reveals more robustness to the noise, it is computationally expensive.

2.2.2 Fully connected Neural Networks

A neural network is composed of a collection of neurons that are connected in an acyclic graph and organized in layers. According to this structure, the outputs of some neurons can become inputs to other neurons, as shown in Figure 2.3, where the scheme of two example Neural Network topologies are reported. In details, the displayed neurons organization in Figure 2.3 represents the most common layer type, fully-connected layer, in which neurons between two adjacent layers are fully pairwise connected, but neurons within a single layer share no connections.

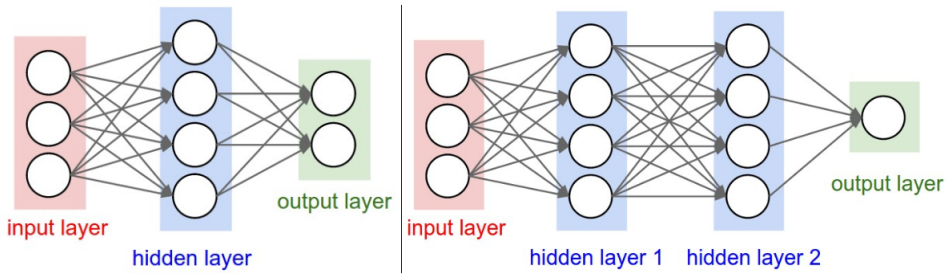


Figure 2.3: Fully-connected layers in two examples of Neural Networks

Fully Connected Neural Networks, also known as Multilayer Perceptrons (MLP), are a category of Artificial Neural Network (ANN) which aim to approximate a function f able to map an input feature x to an output class (or label) y. The architecture of a generic FCNN is shown in Figure 2.3 and it is composed by:

- an input layer;

- one or more hidden layers;
- an output layer.

Focusing on the latter, unlike all layers in a Neural Network, the output layer neurons most commonly do not have an activation function, because they are usually taken to represent the class scores, given by a **score function** that maps the raw data [5]. Hence, with an appropriate **loss function** that quantifies the agreement between the predicted scores and the ground truth labels on the neuron's output, we can turn a single neuron into a linear classifier. In other words, we cast this as an optimization problem in which we will minimize the loss function with respect to the parameters of the score function [6]. The advantage of this parametric approach is that once we learn the parameters we can discard the training data (unlike other kinds of classifiers, as for example kNN classifier).

Some possible choices could be: **Softmax** layer, which converts the previous layer output to a probability distribution, given a probabilistic interpretation through the Softmax function, as formula shown in Figure 2.4; alternatively, another solution could be represented by a max-margin hinge loss to the output of the neuron in order to train it to become a **Multiclass Support Vector Machine** (whose loss for the i-th example is formalized in Figure 2.5).

$$P(y_i \mid x_i; W) = \frac{e^{f_{y_i}}}{\sum_j e^{f_j}}$$

Figure 2.4: Softmax function with a probability distribution

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + \Delta)$$

Figure 2.5: Loss function in Support Vector Machine

More specifically, SVMs are based on a representation of the examples as points in space, so those examples of different categories are divided by a gap as wide as possible and the goal is to find a decision boundary between two or more classes that are maximally far from any point in the training data [7]. In fact, referring to Figure 2.6, the Multiclass Support Vector Machine aimed

Figure 2.6: Class scores scheme in SVM

to achieve scores of the correct classes to be higher than all other scores by at least a margin of delta, so that if any class has a score inside the red region (or higher), then there will be accumulated loss, otherwise the loss will be zero.
The main objective will be to find the weights that will simultaneously satisfy this constraint for all examples in the training data and give a total loss that is as low as possible [6].

2.3 Convolutional Neural Network

2.3.1 General architecture

During last years, in the world of machine vision and the image recognition, approaches that recur to machine learning algorithms and deep neural networks acquired more and more interest, especially with the growing efficiency of the **Convolutional Neural Network (CNN)** into extracting complex and abstract patterns from image data relatively quickly respect to other conventional methods. Convolutional Neural Networks, in fact, have become one of the most used tool in computer vision field for solving problems of visual recognition, such as image classification, object detection and recognition. CNN is a kind of Artificial Neural Networks which belongs to the category of the Feed Forward Neural Networks, i.e. that networks for which the information flows in only one direction, from the input nodes forward to the output nodes, passing through some hidden layers as the structure that was previously shown in Figure 2.3 [8].

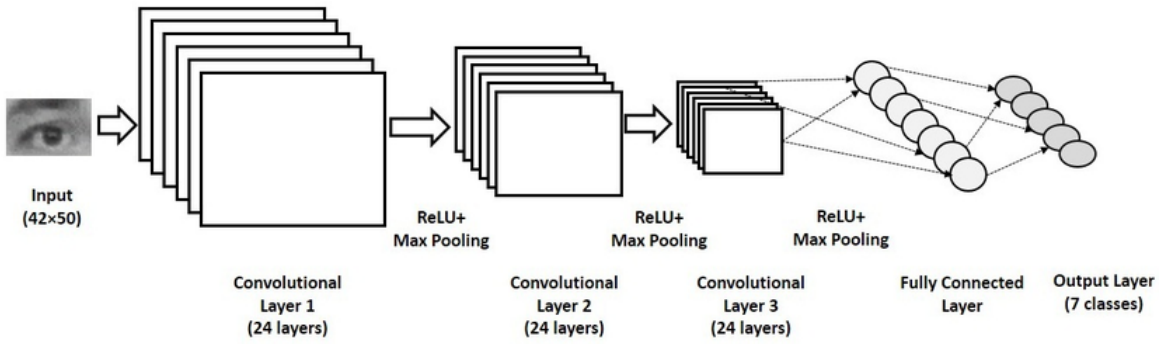


Figure 2.7: Architecture of the Convolutional Neural Network

The difference between convolutional neural networks and feed-forward neural networks lies in the initial and central part of the architecture and consists in the lack of fully connected layers; while, CNNs exploit the convolutional operator instead of the general matrix multiplication in at least one of the layers. Their structure (Figure 2.7) can be divided into two main blocks:

- the first block, made up of a sequence of **convolutional layers** interleaved with non-linear functions and some sub-sampling layers (respectively ReLU and Max Pooling in Figure 2.7), is aimed at learning patterns and local and spatial features directly from the annotated training images that are input to the network;
- the second block, used to actually implement the visual recognition task, is a series of **fully connected layers** which makes the aggregation of the previous convolutional map output and get definitive scores output for a class, through the classification probability distribution.

2.3.2 Convolutional layers

In the first section of a CNN, the **convolutional layers** are organized in a sort of hierarchical structure, meaning that the first layers are dedicated to learning low-level features, such as edges, color, gradient orientation, curves, while, going up with the layers, the learnt features become more and more complex, like corners (mid-level), blobs (high-level). The role of the CNN is to reduce the images into a form which is easier to process, without losing features which are critical for getting a good prediction [9]. In fact, thanks

to the introduction of the convolutional part, CNNs can learn even complex features of a set of images, that are seen as compositions of simpler features, directly from the data, without requiring the effort of extracting them manually.
The element implicated in performing the convolution operation in the first part of a Convolutional Layer is called the **Kernel/Filter**, as displayed in Figure 2.8.

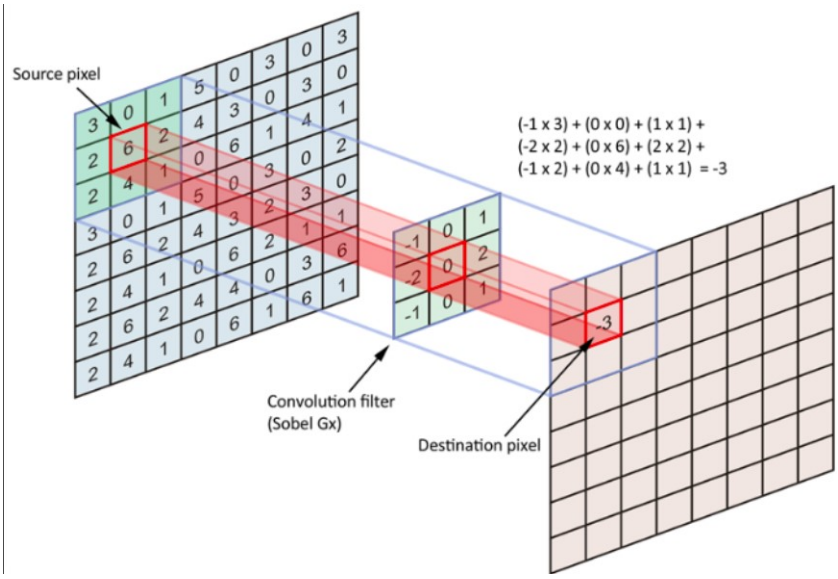


Figure 2.8: Filtering operation in Convolutional layer

Every filter is small spatially (along width and height), it is applied and slided on the entire extension of the input image in order to learn a specific feature. Therefore, the convolutional layer's parameters consist of a set of learnable filters.
The filtering operation is performed through a series of multiplications and sums of the input values by some parameters, called weights, whose values identify the features to be learnt. For example, a typical filter on a first layer of a CNN might have size 5x5x3 (i.e. 5 pixels width and height, and 3 because RGB images have depth 3, the color channels).
During the forward pass, we convolve each filter across the width and height of the input volume and compute dot products between the entries of the filter and the input at any position: the convolution operation is applied [5]. Then, each intermediate result, given by all the multiplications, is summed in

order to provide the final result. After its storage, it is located into an output matrix: **feature map** or activation map.
During the forward pass, we convolve each filter across the width and height of the input volume and compute dot products between the entries of the filter and the input at any position. As we slide the filter over the width and height of the input volume we will produce a 2-dimensional activation map that gives the responses of that filter at every spatial position.
Actually, **three hyperparameters** control the size of the output volume: the depth, the stride and zero-padding.

1. The **depth** of the output volume corresponds to the number of filters we would like to use, each learning to look for something different in the input. For example, if the first Convolutional Layer takes as input the raw image, then different neurons along the depth dimension may activate in presence of various oriented edges, or blobs of color. A set of neurons that are all looking at the same region of the input is called depth column.

2. The **stride** is referred to how the filter is going through the input layer. When the stride is 1 then we move the filters one pixel at a time, while if the stride is 2 then the filters jump 2 pixels at a time, so that they produce smaller output volumes spatially.

3. The feature of **zero-padding** is that it will allow us to control the spatial size of the output volumes, by padding the input volume with zeros around the border; the main reason of this possible choise is that it is useful to preserve the spatial size of the input volume, that so the input and output width and height are the same [10].

$$\left\lfloor \frac{N + 2P - F}{S} \right\rfloor + 1$$

Figure 2.9: Computation of the output feature map size

The spatial size of the output volume can be calculated through the formula reported in Figure 2.9, as a function of the input volume size, in terms of width/height (N), the receptive field size of the Convolutional Layer neurons (F), the stride with which they are applied (S) and the amount of zero padding

used (P) on the border [6].
There are two types of results to the operation: one in which the convolved feature is reduced in dimensionality as compared to the input, and the other in which the dimensionality is either increased or remains the same. This is done by applying Valid Padding in case of the former, or Same Padding in the case of the latter. When the width/height dimension is augmented in each layer (for example, from a 5x5x1 image into a 6x6x1 image) and then the 3x3x1 kernel is applied over it, it is evident from Figure 2.10 that the convolved matrix turns out to be of dimensions 5x5x1 [11]. For this reason, this kind of computation is called **Same Padding**.

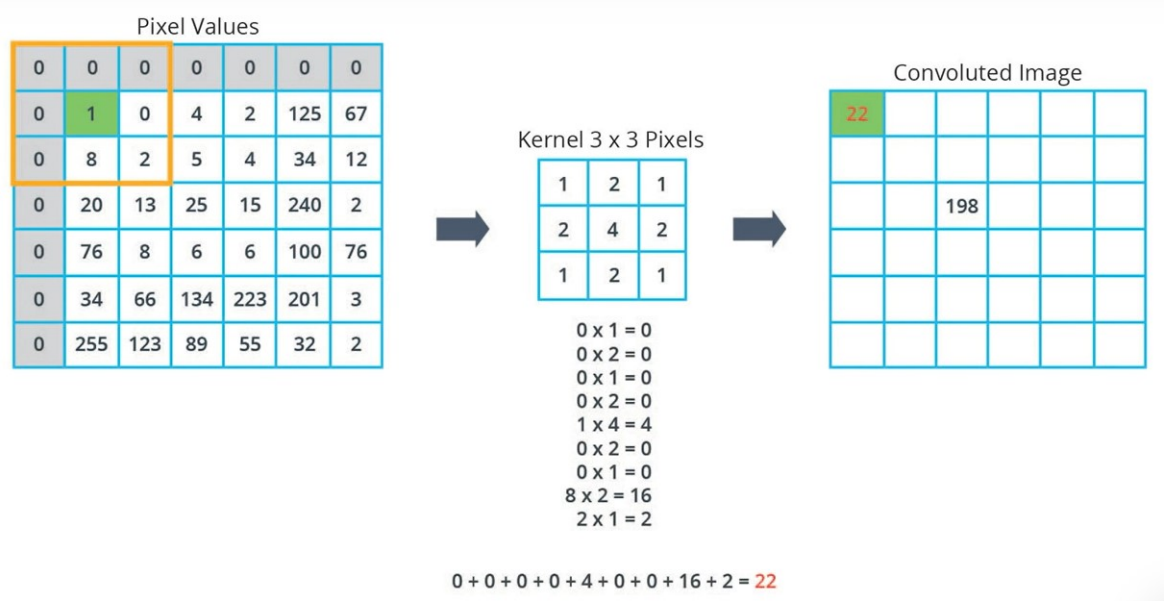


Figure 2.10: Filtering operation with "Same zero-padding" mode

On the other hand, if we perform the same operation without padding, specifying "Valid Padding", it means our convolutional layer is not going to pad at all and, in this way, the input size will not be maintained.

2.3.3 Pooling layers

Referring to the structure explained in general in Section 2.3.1 and in Figure 2.7, after a nonlinearity (e.g. ReLU) the **pooling layer** is a new layer added after the convolutional one. The addition of a pooling layer after the convolutional layer is a common pattern used for ordering layers within a

convolutional neural network that may be repeated one or more times in a given model.
The pooling layer operates upon each feature map produced by a certain convolutional layer in a separate way in order to create a new set of the same number of pooled feature maps. The result of using a pooling layer and creating down sampled or pooled feature maps is a summarized version of the features detected in the input in order to reduce the amount of parameters and computation in the network, and hence to also control overfitting. Moreover, they are useful as small changes in the location of the feature in the input detected by the convolutional layer will result in a pooled feature map with the feature in the same location. This capability added by pooling is called the model's invariance to local translation and it helps to make the representation become approximately invariant to small translations of the input. Invariance to translation means that if we translate the input by a small amount, the values of most of the pooled outputs do not change.
Pooling involves selecting a pooling operation, much like a filter to be applied to feature maps. The size of the pooling operation or filter is smaller than the size of the feature map; specifically, it is almost always 2×2 pixels applied with a stride of 2 pixels. This means that the pooling layer will always reduce the size of each feature map by a factor of 2. For example, a pooling layer applied to a feature map of 6×6 (36 pixels) will result in an output pooled feature map of 3×3 (9 pixels).

Two common functions used in the pooling operation are reported in Figure2.11 :

- **Average Pooling**: involves calculating the average for each patch of the feature map, so this means that each 2×2 square of the feature map is down sampled to the average value in the square;
- **Maximum Pooling**(or Max Pooling): implicates the maximum value for each patch of the feature map, where the results are down sampled or pooled feature maps that highlight the most present feature in the patch [12].

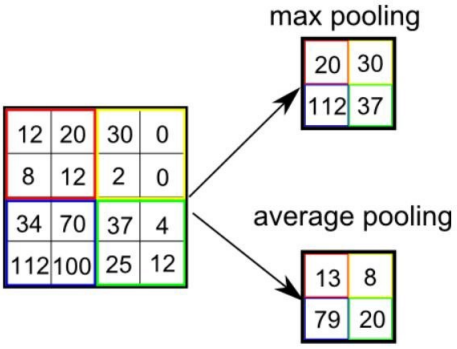


Figure 2.11: Average and max pooling oparation schemes

2.4 Defect detection algorithm

2.4.1 Deep learning-based anomaly detection

Deep learning is a subset of machine learning that achieves good performance and flexibility to represent the data as a nested hierarchy of concepts within layers of the neural network. In particular, deep learning-based anomaly detection algorithms have become increasingly popular: by analyzing real-world datasets, their aim is to determine which instances stand out as being dissimilar to all others and, for this reason, it could represent the key- point for defect recognition algorithms.
First of all, it is crucial to understand what an anomaly is: it also referred to as abnormality, deviant or outlier in the data extraction and statistics. The anomalies appear as few data points which are located further away from the bulk of the standard data points (inliers), as to arouse the suspicions they are generated from a different mechanism, and hence are considered anomalies (outliers). *"An outlier is an observation in a dataset which appears to be inconsistent with the remainder of that set of data"* (Johnson, 1992). It is important do not confuse anomalies with novelties, that are not considered as anomalous data points; instead, they are been applied to the regular data model. In order to distinguish them it is fixed a certain decision threshold, so that the points which significantly deviate from this may be considered essentially as anomalies or outliers.
The choice of a deep neural network architecture in **Deep Anomaly Detection(DAD)** methods primarily depends on the nature of input data. Input data can be broadly classified into sequential (e.g. voice, text, music, time

series recognition) or non-sequential data (e.g. images, video detection).
In order to have a general overview, it is shown a scheme in Figure 2.12, where state-of-the-art DAD research techniques were grouped into different categories by underlying the features and the various aspects that determine the formulation of the problem [4].
Looking at Figure 2.12, in the first column, several interesting applications are reported (e.g. damage detection) and a first classification is made on the base of the types of the existing anomalies (that could be point anomalies, contextual or behavioral anomalies, collective or group anomalies).

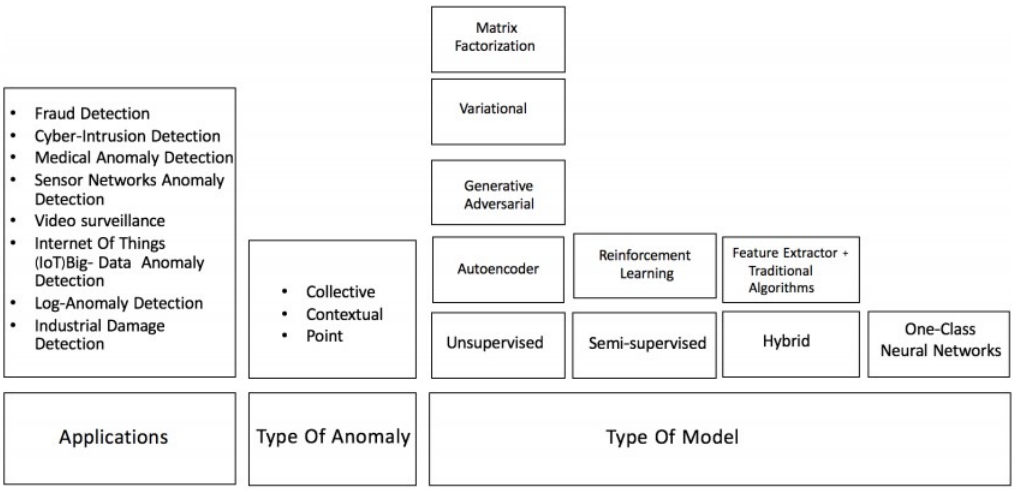


Figure 2.12: Key components associated with deep learning-based anomaly detection technique

Focusing on the various types of DAD models, we can distinguish different **categories of techniques**: (1) Unsupervised deep anomaly detection (2) Supervised deep anomaly detection (3) Semi-supervised deep anomaly detection – based on availability of labels – (4) deep hybrid models (DHM) or (5) one-class neural network (OC-NN) – based on training objectives.

1. **Unsupervised anomaly detection** algorithm is identified as a cost effective technique since it does not require annotated data for training the algorithms able to distinguish the "normal" regions from the "anomalous" ones in the original or in the latent feature space producing an outlier score of the data instances based on intrinsic properties of the dataset

such as distances or densities. The computational complexity of this technique depends on the number of the operation, networks parameters and hidden layers of the neural networks that characterized the autoencoders, which represents the most common unsupervised deep architecture exploited in anomaly detection. Basically, the procedure consists in reconstructing the input data. When the autoencoders are trained solely on normal data instances (which are the majority in anomaly detection tasks), they fail to reconstruct the anomalous data samples; therefore, they output a large reconstruction error and the data samples which produce high residual errors are considered outliers [4].

2. **Supervised deep anomaly detection** technique is superior in performance compared to unsupervised anomaly detection one, since it uses labeled samples. Deep supervised learning methods depend on separating data classes whereas unsupervised techniques focus on explaining and understanding the characteristics of data. Multi-class classification based anomaly detection techniques assume that the training data contains labeled instances of multiple normal classes such that a classifier is able to distinguish between anomalous class from the rest of the classes. In fact, the architecture of this model is composed by two sub-networks: a feature extraction network followed by a classifier network. Due to lack of availability of clean data labels and the computational complexity that could be very high depending on the input data dimension, supervised deep anomaly detection techniques are less cost effective with respect to the above-mentioned techniques.

3. **Semi-Supervised deep anomaly detection** techniques learn a discriminative boundary around the normal instances by assuming that all training instances have only one class label and the single test instance that does not belong to the majority class is labelled as anomalous. The computational complexity of semi-supervised DAD methods based techniques is similar to supervised DAD techniques, which primarily depends on the dimensionality of the input data and the number of hidden layers used for representative feature learning. Use of labeled data (usually of one class), can produce considerable performance improvement over unsupervised techniques, but the hierarchical features extracted within hidden layers may not be representative of fewer anomalous instances, hence are predisposed to the over-fitting problem.

4. For what concerns the last two techniques classified training objectives employed, we have the **Deep Hybrid Models (DHM)**, where the representative features learned within deep models are input to traditional algorithms as common classifiers and the features are extracted within hidden layers of the deep neural network, in order to discriminate the irrelevant features which can conceal the presence of anomalies.

5. Moreover, the last method presented in the above-mentioned list is **One-class Neural Network (OC-NN)**, which combines the ability of deep networks to extract a progressively rich representation of data with the one-class objective in order to separate all the normal data points from the outliers. In fact, OC-NN models extract the common factors of variation within the data distribution, working in the hidden layers of the deep neural network, and then, the algorithm produces an outlier score for a test data instance [4].

2.4.2 Object detection approach

In order to carry on the algorithm evaluation suitable for this use case, it was executed a deep investigation, since today machine learning algorithms provide a wide set of powerful possible approaches that cover many different vision tasks.
There are four main different widespread tasks (Figure 2.13) which can be handled by deep learning algorithms:

- **semantic segmentation**;
- **classification**;
- **instance segmentation**;
- **object detection**.

The first one, the semantic segmentation, in which the output is a decision category for each pixel, was not considered. It does not differentiate instances, only cares about pixels and usually it is not used in machine vision tasks.
Instead, the classification approach could have been proper in order to distinguish if the image given as input represents a damaged car or not. In this case, the problem was represented by the difficulties in the dataset preparation: it is impossible to collect all the necessary images that cover the huge amount of

potential damages, in different light conditions. There are two further possible methods that can be chosen using this kind of approach: binary classification and one-class classification. The first one uses a neural network with just two classes, damaged and non-damaged. The problem is that, in reality, in this case there are basically a huge amount of possible damages and different light conditions, therefore, it results impossible to produce a balanced dataset that provides all the necessary pitures as representative examples for the training of the neural network. Instead the one-class approach works in a different way because usually the neural network is trained using a dataset with just the images that do not present any damage. Also in this case, it is common to have again a problem in the dataset preparation, because if it does not include every possible conditions of lights in which the system is working, the detection could fail. The algorithm knows only what is "correct" from the provided dataset, it does not have any information of what is a damage. It is a crucial point, especially in the one-class approach, because, for example, if an input image is characterized by a specific different kind of illumination, outside the training dataset, it might be classified as damaged, incorrectly.

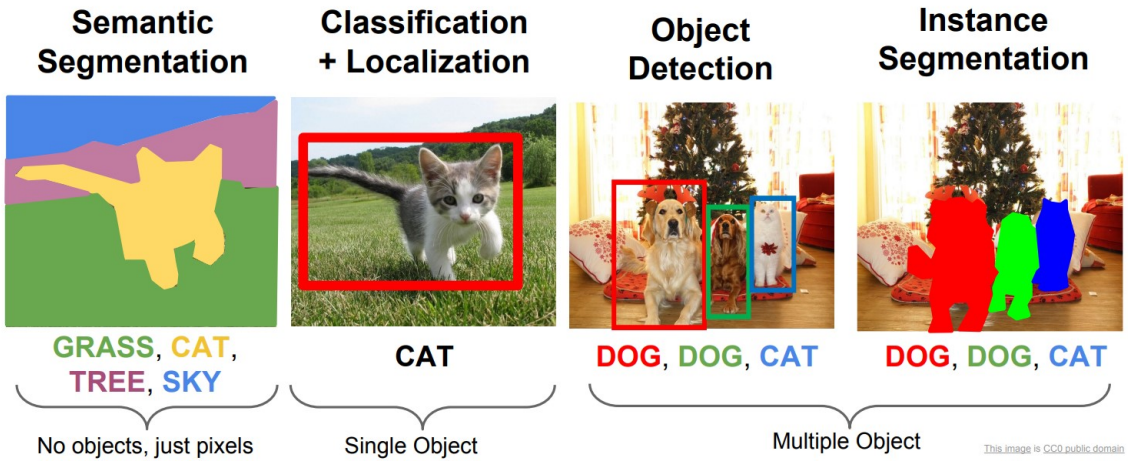


Figure 2.13: Main machine vision approaches (from left): Semantic segmentation, Classification + Localization, Object detection, Instance segmentation

Also in instance segmentation method there are some problems related to the dataset preparation: here the labelling is very time expensive, since it is necessary to provide the algorithm with the precise coordinates of the object shape, in order to obtain the correct segmentation mask. For this reason,

even if the algorithm is able to achieve a better precision in the identification and in the localization of the desired object, it was discarded for our use case.

The last possible application in the list above-mentioned, **object detection**, was revealed the most suitable approach for the task under analysis. In most of the cases the object detection models are chosen instead of image classification ones because their first significant advantage consists in recognizing multiple and different relevant objects in a single image, by providing also their own localization. The localization task is attained drawing boxes around objects, as it is possible to see in the third image in Figure 2.13 and it is defined in terms of width/height and box coordinates. On the other side, dataset creation and labelling are time expensive, but less demanding with respect to instance segmentation. Although it is still necessary to have a high quality dataset, this algorithm should be more robust even if it may not contain information about all the possible damages. From the literature, in fact, it is come to light that object detection models learn to identify defects and should be independent from the background on which the defects are found.This peculiarity could represent a very big advantage in this case study, since the car body is not uniform and this skill could well separate the situation in which there is the presence of a real damage from an abrupt inhomogeneity on the external surface of the car.
Nowadays, with the rapid development of convolutional neural network, most object detection algorithms started to use deep learning techniques. Approximately, we can divide them into two categories: **single-stage detector** and **two-stage detector**. The main difference lies on whether to pool the feature maps for a second stage [13].
SSD (Single Shot Detection), DSSD (Deconvolutional Single Shot Detector), YOLO (You Only Look Once) and RetinaNet are some widely used single-stage object detector with efficient speed. As for two-stage object detector, the most representative models are Fast R-CNN, Faster R-CNN (Region-based Convolutional Neural Network) and R-FCN (Region-based Fully Convolutional Networks).

2.4.3 Single Shot Detector

The **Single Shot Detection** method is much faster compared with two-shot RPN-based approaches.SSD achieves 74.3%- 76.9% mAP. This value outperforms Faster R-CNN (73.2% mAP) and YOLOv1 (63.4% mAP). Thus, SSD

is one of the object detection approaches to be analyzed. Focusing on its internal architecture, after going through a certain number of convolutions for feature extraction, we obtain: a feature layer of size m×n (number of locations); for each location, we got k bounding boxes, which have different sizes and aspect ratios; for each of the bounding box, we will compute c class scores and 4 offsets relative to the original default bounding box shape. Thus, we got (c+4)kmn outputs. Through this formula, SSD reaches an amount of bounding boxes which is much more than that of YOLO (Figure 2.14).

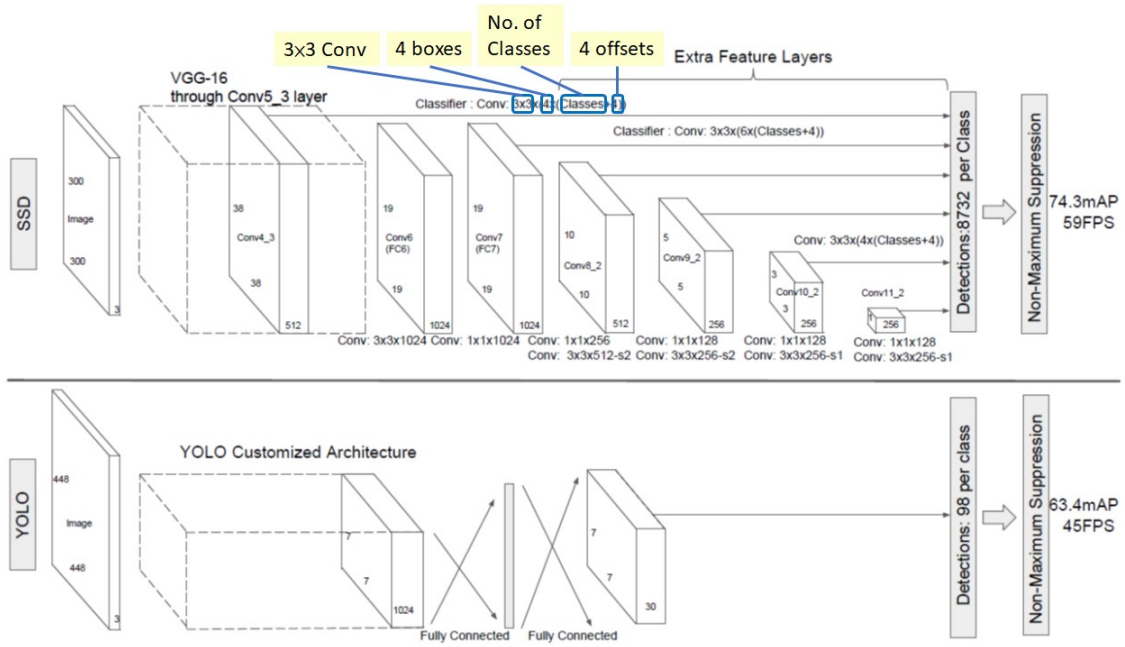


Figure 2.14: SSD and YOLO architectures

Other interesting characteristics of the Single Shot Detector are related to its structure. It has mainly two components (Figure 2.15): a backbone model (layers with white boxes) and SSD head (layers with blue boxes). Backbone model usually is a pre-trained image classification network as a feature extractor. The SSD head is just one or more convolutional layers added to the backbone, and the outputs are interpreted as the bounding boxes and the classes of objects in the spatial location of the final layers activations.
Instead of using sliding window, SSD divides the image using a grid and have each grid cell be responsible for detecting objects in that region of the image. Detection objects simply means predicting the class and location of an object

within that region. If no object is present, we consider it as the background class and the location is ignored.

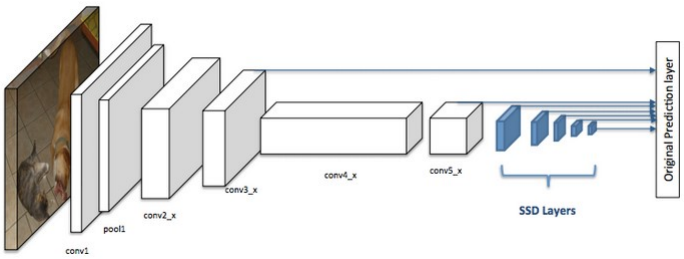


Figure 2.15: Architecture of a convolutional neural network with a SSD detector

Each grid cell in SSD can be assigned with multiple anchor/prior boxes. These anchor boxes are pre-defined and each one is responsible for a size and shape within a grid cell. SSD uses a matching phase while training, to match the appropriate anchor box with the bounding boxes of each ground truth object within an image. Essentially, the anchor box with the highest degree of overlap with an object is responsible for predicting that object's class and its location. This property is used for training the network and for predicting the detected objects and their locations once the network has been trained. The central premise of the SSD architecture is the receptive field that enables us to detect objects at different scales and gives in output a tighter bounding box. Receptive field is defined as the region in the input space that a particular CNN's feature is looking at.
The extra step taken by SSD is that it applies more convolutional layers to the backbone feature map and has at each of these convolution layers output an object detection result. As earlier layers bearing smaller receptive field can represent smaller sized objects, predictions from earlier layers help in dealing with smaller sized objects, like for example very little damages. Because of this, SSD allows us to define a hierarchy of grid cells at different layers. For example, we could use a 4x4 grid to find smaller objects, a 2x2 grid to find mid-sized objects and a 1x1 grid to find objects that cover the entire image [14].

2.5 Pre-trained models

2.5.1 Transfer learning and evaluation metrics

After a first algorithm analysis, the main idea was to apply **transfer learning** by adapting the neural network of pre-trained models to this specific application (see Section 3.3.3). For this purpose, we conducted a survey of the pre-trained neural networks already available on the literature, in order to exploit one of them as a starting point.
The existing models are previously trained on huge datasets, containing millions of images, so that they are able to have good performance in recognizing hundreds of different objects. The large-scale dataset, in fact, is an important motivation for the continuous improvement of the object detection algorithms, especially for deep learning based techniques. Some examples of the earliest datasets are ImageNet and VOC, but the most recent high-quality dataset that has an important role in the image classification and object detection community is COCO (Common Object in COntex). It contains more than 300,000 images (>200K labeled) and 80 object categories [15] and features object detection tasks using either bounding box output or instance segmentation output [16]. Therefore, all the pre-trained models chosen in our training phase (for further details, see Section) had made use of COCO dataset.
In order to make a comparison between these different typologies of models, firstly, it is conducted an analysis in terms of mean Average Precision, mean Average Recall, where the reported scores are based on small, medium and large objects in the data (general formulation in Figure 2.16) and they are really taken into account in object recognition:

$$Precision = \frac{TP}{TP + FP}$$

$$Recall = \frac{TP}{TP + FN}$$

TP = True positive
TN = True negative
FP = False positive
FN = False negative

Figure 2.16: Evaluation metrics formulas

- **Average Precision (AP)**: it indicates the amount of your correct predictions;

- **Average Recall (AR)**: it measures how good are all the positives you found.

Another important parameter, that could be useful to analyze among the outputs during the training phase is the **Intersection of Union (IoU)** (look at Figure 2.17).

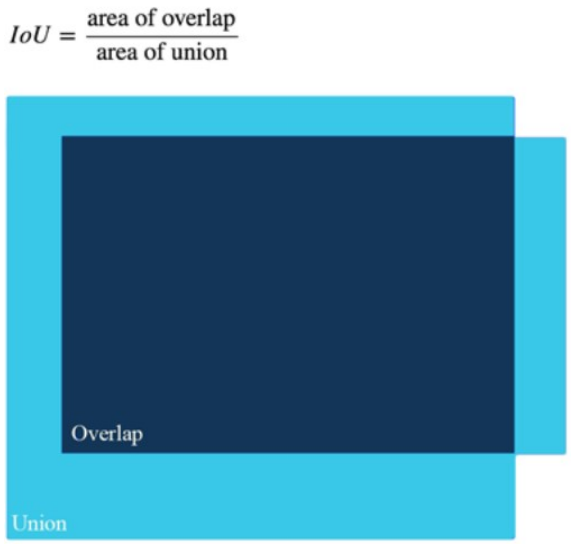


Figure 2.17: Intersection of Unit scheme definition

The Average Precision, defined previously, is measured at different values of IoU, which indicates how much our predicted boundary overlaps with the ground truth (the real object boundary). As standard value, its threshold should be fixed to 0.5, but during the evaluation phase, often it is present also the value "0.75" (usually it stands in the range 0.5- 0.95). This factor is crucial in order to classify if the produced prediction is a true positive or a false positive, so that it can be inserted in the AP formula.

2.5.2 MobilNet

If we merge both SSD, analyzed previously in Section 2.4.3 and the **MobileNet** architecture, that is an example of pre-trained model included in the lists of models belonging to the Model Garden and Model Zoo of Tensorflow, we could arrive at a fast, efficient deep learning-based method to object detection. From the moment that, if we combine the MobileNet and SSD, it

may get better accuracy with a higher mAP, they represent one pre-trained model and one of the object detection approaches that might also be worth investigating. Other architectures, like Resnet or VGG or Alexnet, have a large network size and the number of computations increases whereas in Mobilenet there is a simple architecture, as reported in Figure 2.18, consisting of a 3×3 depthwise convolution followed by a 1×1 pointwise convolution [17].

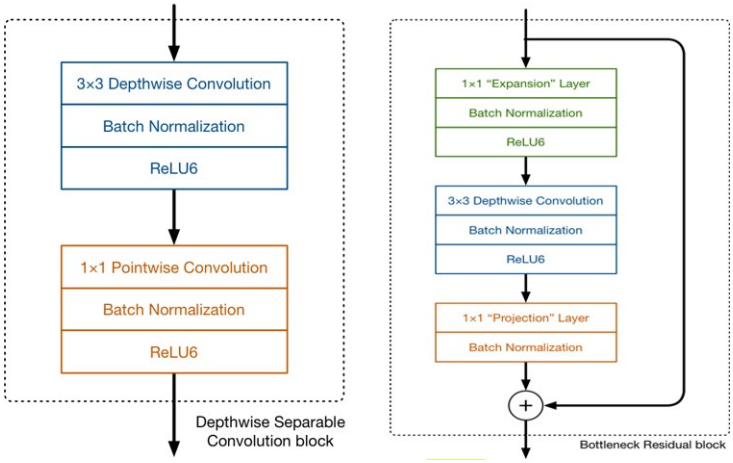


Figure 2.18: MobilNet v1 and v2 architectures

MobileNet V2 still uses depthwise separable convolutions. In V1 the pointwise convolution either kept the number of channels the same or doubled them. In V2 it does the opposite: it makes the number of channels smaller. This is why this layer is now known as the projection layer — it projects data with a high number of dimensions (channels) into a tensor with a much lower number of dimensions. Using low-dimension tensors is the key to reducing the number of computations. Let's compare **MobileNet V1** to **MobilNet V2** (from Figure 2.19), starting with the sizes of the models in terms of learned parameters and required amount of computation (MACs) [18].

Version	MACs (millions)	Parameters (millions)
MobileNet V1	569	4.24
MobileNet V2	300	3.47

Figure 2.19: MobilNet v1 and v2 parameters

From the number of MACs alone, V2 should be almost twice as fast as V1. Also in terms of parameters, V2 has the advantage too: it only has 80% of the parameter count that V1 has.

2.5.3 EfficientDet

We also consider another typology of approach, **EfficientNet** model, that has gained the new state of the art accuracy for 5 out of the 8 datasets, with 9.6 times fewer parameters on average. From Figure 2.20, we can observe an analysis in which the authors firstly find out the relationship between the accuracy and the scaling (size) of a model: initially the accuracy increases radically (of course along with the computational cost), but after sometime the curves almost flatten. The red curve displayed EfficientNet models performance on the ImageNet dataset compared with other models of similar Top-1 accuracy and it is possible to see how EfficientNet models show the best values.

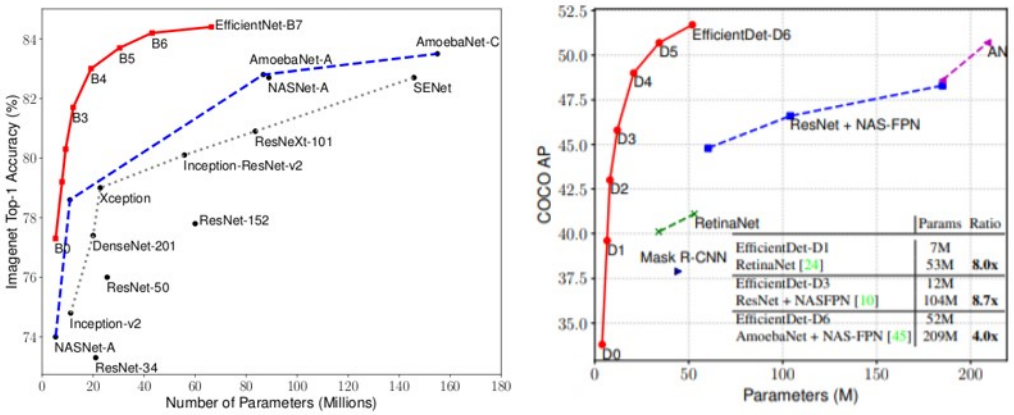


Figure 2.20: EfficientNet accuracy on Imagenet (on the left) EfficientDet AP on COCO (on the right)

Also on the right (Figure 2.20), where EfficientDet models values are reported, AP shows a visible rise with respect to the others.
Recent works show remarkable performance on image classification by jointly scaling up all dimensions of network width, depth, and input resolution. Inspired by these works the keypoint for the construction of **EfficientDet** models was to reuse the same width/depth scaling coefficients of EfficientNet-B0

to B6. In fact, the considerable difference with respect to the previous works in object detection field, which mostly scale up a baseline detector by employing bigger backbone networks (e.g. ResNeXt or AmoebaNet), is the choice of exploiting the ImageNet-pre-trained checkpoints of EfficientNet-B to B6 models.
Moreover, in Figure 2.21, if we analyze in particular the rightmost column, we can notice that in FLOPS (the measure of computational power needed) there is a remarkable difference between EfficientDet models and their comparable models. Also in this case, we can observe a very low number of architecture parameters and interesting results relative to the AP, measured at different values of IoU (Figure 2.17). The latter indicates how much our predicted boundary overlaps with the ground truth (the real object boundary). Usually, as standard value, in order to classify if the produced prediction is a true positive or a false positive, its threshold should be fixed (for example 0.5) and it is inserted in the AP formula.

EfficientDet models consistently achieve better accuracy and efficiency than the prior art across a wide spectrum of resource constraints. In particular, it achieves state-of-the-art accuracy with much fewer parameters than previous object detection models [19].

Model	test-dev			val					Latency (ms)	
	AP	AP_{50}	AP_{75}	*AP*	Params	Ratio	FLOPs	Ratio	TitianV	V100
EfficientDet-D0 (512)	**34.6**	**53.0**	**37.1**	**34.3**	**3.9M**	**1x**	**2.5B**	**1x**	**12**	**10.2**
YOLOv3 [34]	33.0	57.9	34.4	-	-	-	71B	28x	-	-
EfficientDet-D1 (640)	**40.5**	**59.1**	**43.7**	**40.2**	**6.6M**	**1x**	**6.1B**	**1x**	**16**	**13.5**
RetinaNet-R50 (640) [24]	39.2	58.0	42.3	39.2	34M	6.7x	97B	16x	25	-
RetinaNet-R101 (640)[24]	39.9	58.5	43.0	39.8	53M	8.0x	127B	21x	32	-
EfficientDet-D2 (768)	**43.9**	**62.7**	**47.6**	**43.5**	**8.1M**	**1x**	**11B**	**1x**	**23**	**17.7**
Detectron2 Mask R-CNN R101-FPN [1]	-	-	-	42.9	63M	7.7x	164B	15x	-	56‡
Detectron2 Mask R-CNN X101-FPN [1]	-	-	-	44.3	107M	13x	277B	25x	-	103‡
EfficientDet-D3 (896)	**47.2**	**65.9**	**51.2**	**46.8**	**12M**	**1x**	**25B**	**1x**	**37**	**29.0**
ResNet-50 + NAS-FPN (1024) [10]	44.2	-	-	-	60M	5.1x	360B	15x	64	-
ResNet-50 + NAS-FPN (1280) [10]	44.8	-	-	-	60M	5.1x	563B	23x	99	-
ResNet-50 + NAS-FPN (1280@384)[10]	45.4	-	-	-	104M	8.7x	1043B	42x	150	-
EfficientDet-D4 (1024)	**49.7**	**68.4**	**53.9**	**49.3**	**21M**	**1x**	**55B**	**1x**	**65**	**42.8**
AmoebaNet+ NAS-FPN +AA(1280)[45]	-	-	-	48.6	185M	8.8x	1317B	24x	246	-
EfficientDet-D5 (1280)	**51.5**	**70.5**	**56.1**	**51.3**	**34M**	**1x**	**135B**	**1x**	**128**	**72.5**
Detectron2 Mask R-CNN X152 [1]	-	-	-	50.2	-	-	-	-	-	234‡
EfficientDet-D6 (1280)	**52.6**	**71.5**	**57.2**	**52.2**	**52M**	**1x**	**226B**	**1x**	**169**	**92.8**
AmoebaNet+ NAS-FPN +AA(1536)[45]	-	-	-	50.7	209M	4.0x	3045B	13x	489	-
EfficientDet-D7 (1536)	**53.7**	**72.4**	**58.4**	**53.4**	**52M**		**325B**		**232**	**122**
EfficientDet-D7x (1536)	**55.1**	**74.3**	**59.9**	**54.4**	**77M**		**410B**		**285**	**153**

We omit ensemble and test-time multi-scale results [30, 12]. RetinaNet APs are reproduced with our trainer and others are from papers.
‡Latency numbers with ‡ are from detectron2, and others are measured on the same machine (TensorFlow2.1 + CUDA10.1, no TensorRT).

Figure 2.21: EfficientDet performance on COCO

Chapter 3

Damage detection

3.1 Case study system architecture

3.1.1 High-level system architecture

The scenario where the system will be working is characterized by a designated garage placed in Madrid, Spain, that is the city where Zity headquarters is located and operational and maintenance activities take place [20].

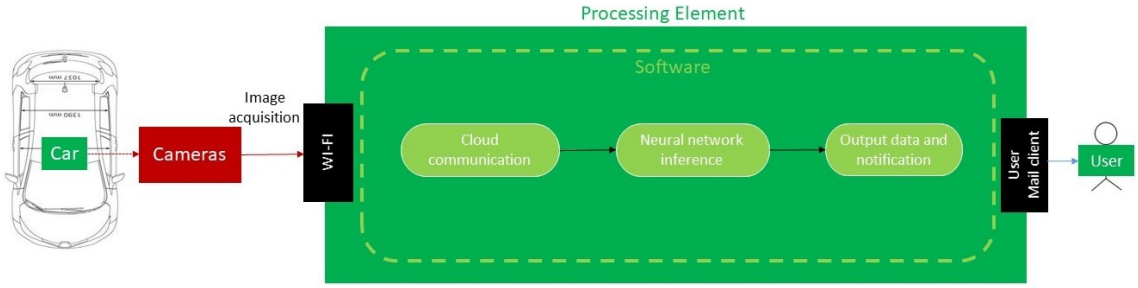


Figure 3.1: High-Level System Architecture

The high level system architecture is described in Figure 3.1 and includes:

- **Acquisition system**: including cameras, motion detection sensors and video recorder.
- **Processing element**: hardware component that allows the execution of the software.
- **Defect detection software**: handles the image pre-processing and runs the neural network inference.

In the next sections (Section 3.1.2, Section 3.1.3 and Chapter 3) each component and its own position are further detailed.

3.1.2 Acquisition system setup

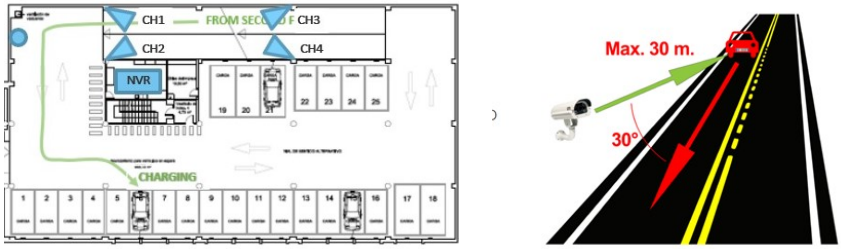
The setup of the acquisition system required to make decisions regarding:

- The selection of the cameras taking into account the technical characteristics (resolution, frame rate, connectivity, etc.) as well as considerations related to cost and robustness.
- The identification of the optimal position considering the field of view, the distance from the car, the light conditions.

After analysing different possibilities, the best choice regarding hardware components to run the defect detection software seemed to be a video surveillance kit WiFi from DSE, including an NVR and four cameras.
The IP cameras have a resolution of 8MP (4K) and 15 fps that provides a good level of detail. They support H265 video compression formats to record high quality video stream with the minimum bandwidth requirement. The cameras come with 3.6mm. wide-angle lens and a built-in infrared illuminator, which makes it possible to monitor in the dark up to 30m. The optical sensor is 1/2.5" CMOS 2704x1950 pixel The video signal between the cameras and the NVR, the control unit of the system, is WiFi, without wires. The latest generation WiFi modules allow an excellent range: up to 300 m. without obstacles and 4/5 cameras indoor, but it is possible to extend the range through a WiFi repeater.
The motion detection sensor is directly integrated in the cameras and allows to activate the video recording when motion is activated.

3.1.3 Position of the components

Considering several aspects such as the light conditions, the position of available Ethernet connection and electric cabinet, we decided to position the NVR in the available office, and the four cameras (*CH1*, *CH2*, *CH3*, *CH4*) along the ramp in the four angles of the walls respectively, as shown in Figure 3.2a. In this way, every car will inevitably go through the AOV (angle of view) of all the four cameras and the WiFi connection between cameras and NVR is good.



(a) Position of NVR and cameras (b) Optimal angle of the cameras

Figure 3.2: Position of the components

The optimal angle relative to the direction of travel of the vehicles is about 30 degrees (Figure 3.2b). In this way, you limit the night glare of the headlights. The maximum distance from where a human face can be detected is 4.9m, while from an optimal distance (calculated using the link *https://www.dseitalia.it/calcolatore_obiettivi.htm*) of 1.5m an area of 2.4m x 1.8m can be framed. From a practical point of view, it has been seen that the best position of cameras was about 1.2 m (Figure 3.3), since the damages are usually localized in the central-lower part of the car body and empirically, from the dataset analysis, the best results have been obtained in these conditions (for more details, see Chapter 4).

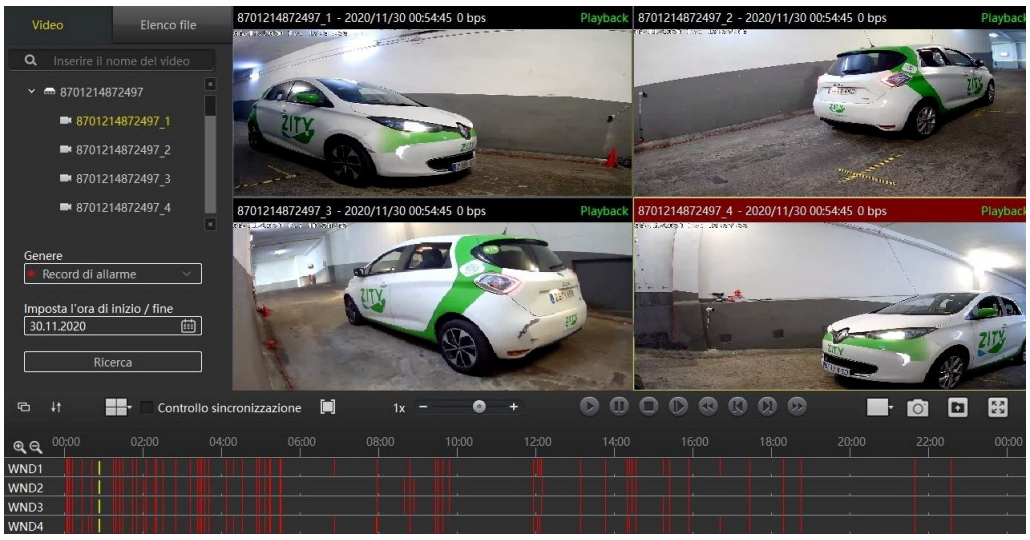


Figure 3.3: Real view from the cameras

3.2 Damage recognition process

3.2.1 Functional requirements

Focusing on the part of the whole structure dedicated to the maintenance activities, the main general functional requirements of the specific case study are the following.

- The solution should be able to **identify** the presence of any new defects that the body car may have.
- The solution should to be able to **localize** the recognized defects, in order to give also an idea about their own approximate extension.
- The solution should be able to **send** an alert to the operator in case a new damage is identified.

3.2.2 Damage recognition steps

The damage recognition process is illustrated in Figure 3.4 and summarized below.

1. When the car passes in the range of a camera, a sensor detects the movement and activates the camera recordings.
2. The camera takes a picture or video of the car.
3. The defect detection algorithm processes the picture/video and identifies if a damage is present.
4. Optionally a license plate identification algorithm identifies the plate number.
5. Optionally the information collected from multiple cameras nearly at the same time and then associated to different views of the same car is aggregated.
6. If a damage is detected on at least one of the sides, a notification including the information is sent to the operator.
7. The information is optionally also saved in a DB for further analysis [20].

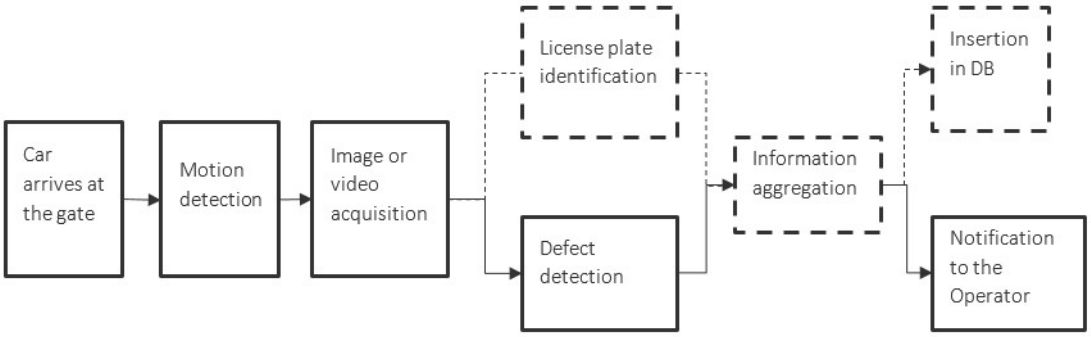


Figure 3.4: Damage detection flow chart

3.3 Defect detection software development

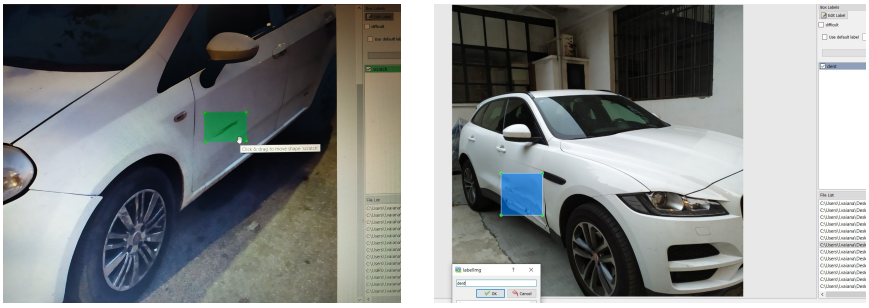
3.3.1 Dataset preparation

One of the crucial parts of building deep learning systems based on neural network algorithm is gathering high quality dataset, since in general the model is as good as the data it learns from. For this reason, a significant effort was dedicated to the research of data.
Since there is no publicly available dataset for car damage object detection containing suitable images for our task, we created our own dataset. This included images belonging to different sources, both because at the beginning of the project we didn't have the camera system installed in the final location and because the objective was to obtain a general model.
At first, some damaged car photos were collected from web (www.kaggle.com, google images). We selected in detail images with proper characteristics for the training dataset (look at Figure 3.5): their resolution (Figure 3.5d) and their sharpness have to be sufficient in order to well distinguish also small damages, as requested in the project; they must not contain labels (Figure 3.5a), or people, or other kind of elements that overlap the damages of the shot car, so that they can be clearly visible; the zoom level in the framing of the subject (Figure 3.5b) has to be approximately constant and comparable with the data acquired by the cameras placed in the ramp of the garage, depending on their own position (Figure 3.5c).

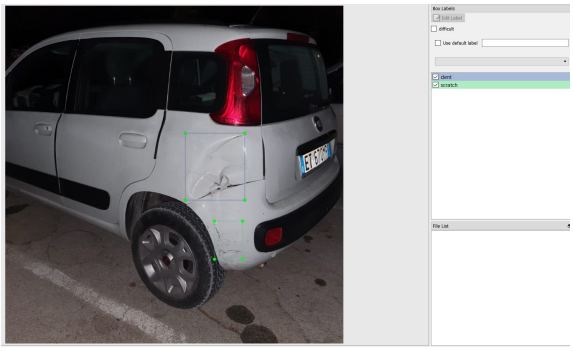
(a) label overlapped (b) too zoomed in picture (c) right point of view (d) right resolution

Figure 3.5: Examples of: discarded images (a)-(b) and included images (c)-(d)

Moreover, most of the images are taken empirically by hand, trying to follow the above-mentioned features, including also similar light conditions with respect to the internal space of the garage and the color of the car body of Zity cars.
All the images were collected, resized through a square shape and a resolution of 1080x1080 pixels.
Afterwards, they were manually annotated by the annotation tool LabelImg (shown in Figure 3.6, because we have not available labelled images for our use case. For object detection data, we need to draw the bounding box on the object (Figure 3.6a) and we need to assign the textual information to the object (Figure 3.6b), with the possibility to include multiple relevant objects in a single image. Even though it was not requested in the project, we generated two different classes of labels (*"scratch"* and *"dent"*, see Figure 3.6c) to better identify the several characteristics of each kind of bounded damage, with the aim to obtain good results during the following phases. For simplicity, initially we have chosen only two classes, because they are the two most common cases of damages in our dataset.



(a) bounding box draw (b) label assignment



(c) list of the classes of label: *"scratch"* and *"dent"*

Figure 3.6: Labelling steps representation

Starting from this initial set of images, we have carried out subsequent cycles of experiments, analysis of the results (as reported in Chapter 4) and collection of new sets of images with the objective to solve some of the identified issues and then improve the performances of the system.
Firstly, we noticed that the results corresponding to the dent recognition are less precise with respect to the scratch detection. For this reason, during the preparation of a second round of photos (about 500), we tried to increase the number of pictures relative to damaged cars containing dents on the car body, by providing more labelled images as samples corresponding to the second class of the model (*"scratch"*), in order to obtain a more balanced effect during the training.
In addition, in a **third dataset** collection (about 750 images), we also included car images in light conditions that seemed to be closer to the real situation in the garage: most of the photos were shot in the evening, with street light switched on or inside underground car parking.
In this way, we enlarged our dataset integrating new photos with similar features until about 1000 images (**fourth dataset**), by following a linear increase

in the amount of the added ones.
In the initial plan, we have scheduled a further extension of the dataset, also including about 100-200 pictures showing the Zity cars in final pilot setting, so that the final complete dataset (**fifth dataset**) contains about 1200 images.
Over the several experiments carried on during the training, validation and test phases featured more in details in the final part of Chapter 4, we observed that the introduction of the Zity cars may make it necessary to reconsider the number and the typology of classes set in the earlier labelling operation. This reassessment is due to the substantial difference in specific features which characterized Zity cars from the other samples included in the previous versions of dataset. In particular, the two additional elements we have focused on during the analysis of the whole dataset are: the front fog lamp characterized by an uncommon shape with respect to the other inserted cars and the presence of any trace of text, specifically the text *Zity* on the frontal, rear and lateral side of the car body. Therefore, a new labelling action was performed on the last added photos and two further classes of label (*"Zity"* and *"front fog light"*) were added into the list previously mentioned in Figure 3.6c.
LabelImg tool produces an output file (it is possible to choose between *PascalVoc XML* or *YOLO* format) for each labelled image that contains information about the class of each bounded object, the relative coordinates of the bounding boxes and also the width/length depending on the format selected for the annotation files. An example of a typical output file (*PascalVoc XML* format), including more than one object simultaneously assigned to different labels of the final class list, is reported below (Figure 3.7): the annotation file inform us that in the image of the car named *Car1170* are present three objects belonging to *"front fog light"*, *"Zity"* and *"scratch"* classes with their own coordinates corresponding to the relative bounding boxes manually drawn.

```
- <annotation>
    <folder>Images</folder>
    <filename>Car1170.jpg</filename>
    <path>E:\Datasets\Fifth test Dataset\Images\Car1170.jpg</path>
  - <source>
      <database>Unknown</database>
    </source>
  - <size>
      <width>497</width>
      <height>398</height>
      <depth>3</depth>
    </size>
    <segmented>0</segmented>
  - <object>
      <name>front fog light</name>
      <pose>Unspecified</pose>
      <truncated>0</truncated>
      <difficult>0</difficult>
    - <bndbox>
        <xmin>74</xmin>
        <ymin>243</ymin>
        <xmax>116</xmax>
        <ymax>293</ymax>
      </bndbox>
    </object>
  - <object>
      <name>Zity</name>
      <pose>Unspecified</pose>
      <truncated>0</truncated>
      <difficult>0</difficult>
    - <bndbox>
        <xmin>303</xmin>
        <ymin>241</ymin>
        <xmax>408</xmax>
        <ymax>291</ymax>
      </bndbox>
    </object>
  - <object>
      <name>scratch</name>
      <pose>Unspecified</pose>
      <truncated>0</truncated>
      <difficult>0</difficult>
    - <bndbox>
        <xmin>190</xmin>
        <ymin>272</ymin>
        <xmax>285</xmax>
        <ymax>295</ymax>
      </bndbox>
    </object>
</annotation>
```

Figure 3.7: Example of LabelImg output file in *PascalVoc XML* format

3.3.2 Development tools and frameworks

During the model training and evaluation phase described in Section 3.3.3, several tools and frameworks were exploited. For this reason, we have provided a brief introduction of all of them in the following subsections, by describing their relative main features.

Google Colaboratory

Colaboratory from Google is a platform that provides a free Jupyter notebook environment to run code entirely in the cloud. In its environment some of the most common libraries for deep learning applications development are available, such as Keras, Tensorflow, PyTorch and others. Also, it makes available GPU, which accelerates workloads, where significantly higher amounts of data have to be analyzed faster than traditional CPUs, in applications from energy exploration to deep learning, that is the case study of this thesis. In addition, NVIDIA accelerators provide the power needed to run larger simulations with speeds never achieved before. NVIDIA GPUs provide the high performance required for desktops, applications, and virtual workstations.

Keras

Keras is a high-level and open-source API (Application Programming Interface) for neural network development. It supports both CNN and RNN (Recurrent Neural Network) and also other arbitrary network architectures (multi-input or multi-output models, layer sharing, model sharing). This means that Keras is suitable for developing any deep learning model. It is written in Python and can run on top of Tensorflow. It also allows to easily import and integrate well known and pre-trained Neural Networks in a project; in fact, Keras Applications include efficient deep learning models (like MobileNet, RetinaNet, EfficientNet and others) available with pre-trained weights that can be employed for prediction, features extraction and fine-tuning purposes.

Tensorflow and Pytorch

TensorFlow is an end-to-end open source deep learning framework. It has a comprehensive, flexible ecosystem of tools, libraries and community resources that lets researchers push the state-of-the-art in ML and developers easily

build and deploy ML powered applications. There is also another popular framework for deep learning that is diffused more and more, Pytorch, therefore it is done a brief comparison, by analyzing which one to choose out of the two.
The two frameworks had a lot of major differences in terms of design, paradigm, syntax. Both work on fundamental data type called Tensors which are nothing but multi-dimensional arrays, amenable to high performance computation.
Both represent computation as a directed acyclic graph often called Computation Graph [21]. For this reason, the Tensorflow API was very difficult to start with; while Pytorch API felt like writing native Python code and immediate to debug.
Pytorch is faster than Tensorflow, but it presents less features, lower level function, smaller scale applications, it needs to Python support or specific interface and requires more documentation with respect Tensorflow, which collects several previous use cases, examples and tutorials.

Tensorflow 2.0

Recently it is introduced another major milestone, TensorFlow 2.0, that focuses on simplicity and ease of use, featuring updates like :

- Easy model building with Keras and eager execution;
- Robust model deployment in production on any platform;
- Powerful experimentation for research;
- Simplifying the API by cleaning up deprecated APIs and reducing duplication[22].

The new architecture, describing TensorFlow 2.0, is shown in the conceptual diagram in Figure 3.8

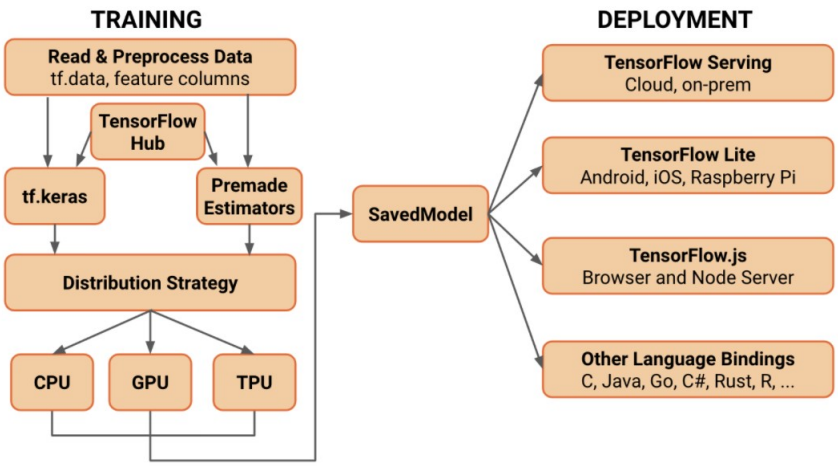


Figure 3.8: Tensorflow 2.0 architecture

Over the last few years, there have been a number of versions and API iterations, while with TensorFlow 2.0, these will be packaged together into a comprehensive platform that supports machine learning workflows from training through deployment. With the rapid evolution of ML, the platform has grown enormously and now supports a diverse mix of users with a diverse mix of needs; TensorFlow 2.0, offers the opportunity to clean up and modularize the platform.

3.3.3 Model training and evaluation

In order to deploy a production model, it was decided to rely on the Tensorflow API (look at Section 3.3.2), developed by Google . The core idea was to perform transfer learning to train the chosen model. That is because typically training from scratch requires a huge amount of data (quite difficult to obtain to solve the project task) and time. Transfer learning, instead, allowed us to start from a pre-trained model, that is a model that has been already trained on a large amount of data.
Therefore, the idea was to exploit a model that has already "learned" how to extract main features from an image. Even if the model has been trained for a different task and have different classes, it constitutes a good and effective starting point to solve the project task. For example, in the one trained on COCO dataset (described in Section 2.5.1), we have changed only the last layers to have just two classes we need to solve the problem, that were not

present in the pre-trained model. Tensorflow provides what they call the "Tensorflow Model Garden" (for Tensorflow 2 described in Section 3.8). This last is a collection with the state-of-the-art models, pre-trained on COCO dataset, that can be used for computer vision tasks like image classification, object detection, and segmentation. At the beginning of the project, the choice was forced to use the old Tensorflow 1 release, even though Tensorflow 2.x was already available, because there was no support for the object detection API. As previously described, after several considerations, it was decided to build a four-class model (look at the label map in Figure 3.9), with the following classes (look at Section 3.3.1 for more details):

- *dent*;
- *scratch*;
- *Zity*;
- *front fog light*;

```
item {
    id: 1
    name: 'dent'
}

item {
    id: 2
    name: 'scratch'
}

item {
    id: 3
    name: 'Zity'
}

item {
    id: 4
    name: 'front fog light'
}
```

Figure 3.9: Label map containing all the classes

The pipeline to perform the training operation can be divided in steps as follows:

1. **collect data** (images in our case) to build the dataset;
2. **label** such **data** with bounding boxes;

3. **split the dataset** in three sections: training, validation and test;
4. **create** the **record file** for each partition of the dataset;
5. **set** correctly the **configuration** and the **hyperparameters** both of the model and for the training phase

Concerning the first two points, they are deeply analyzed in Section 3.3.1.
Regarding the subdivision of the dataset, we have three groups of data that are exploited for different aims in different phases in building a model:

- **training dataset** consists in all the samples used to fit the model during the learning process of the model;
- **validation dataset** is made of all the samples used to provide an unbiased evaluation of a model fit on the training dataset and to fine-tune and to update the model hyperparameters during the "development" stage of the model;
- **test dataset** provides the gold standard used to evaluate the model and it contains all the samples used to provide an unbiased evaluation of a final model already completely trained.

Due to the small size of the entire dataset and the models with a not huge number of hyperparameters (see Chapter 2), we decided to reduce the size of the validation set in this case study, so that it was a partition complying with the following dimensions: the training dataset and the validation dataset contains respectively about 90% and 10% of the whole number of the image data.
Moreover, as regards the test dataset preparation, we referred to a new collection of images out of the previous ones, so that we test properly the models on an amount of pictures, comparable with respect to the validation dataset.
During the test phase, firstly, we also exploited some videos containing cars in similar conditions relative to the real set in Zity garage, and then, we had the possibility to test the models on the recording Zity cars in the definitive cameras position. In this way, we have been able to get an idea of the processing time corresponding to the different models and their own behavior also in the object detection on videos.
In order to start the training of the model, it was necessary to produce the record file for each partition of the dataset, because it is a particular file format required by Tensorflow serializing both data images and annotations.

The starting point was represented by the annotation files in PascalVoc XML format (see Section 3.3.1 and Figure 3.7) which are first converted in CSV (Comma Separated Values) in order to have a complete overview of all the objects labelled in the images by means of a table. In the structure of the produced file, in each row the following information are reported: the filename corresponding to the specific image, the width and the height of the bounding box drawn, its relative class label and its coordinates (*xmin, ymin, xmax, ymax*).
From this conversion we implemented a script (reported in Figure 3.10), by following the Tensorflow hints, aimed to provide the final proper record files.
The final step to start the training and evaluation phase was to set correctly the configuration file relative to the model. First of all, it was necessary to modify the number of classes with respect to the pre-trained model, as abovementioned. Focusing on the training and the evaluation configuration, it was important to indicate the right paths corresponding to the record files of the different partitions, to the label map (previously shown in Figure 3.9) and to the fine tune checkpoint, by specifying also its typology, that in this case is *"detection"*.
During the training phase specified in the configuration file, there are other fundamental hyperparameters of gradient descent, which consists in an iterative optimization process; in particular, we have the **batch size** and the **number of steps**. The former controls the number of training samples to work through before the model's internal parameters are updated and both are strictly linked to the measure of the **number of epochs**, which represents the number of passes of the entire training dataset that the machine learning algorithm has completed. Just to give an example of the relationship between all these terms, if you have 1,000 images and use a batch size of 4, an epoch consists of: 1,000 images / (4 images / step) = 250 steps.
Depending on the model, these hyperparameters are set properly and we can see these choices more in detail in the next Chapter 4, where all the experiments are reported.

```
#Generating TF records
def class_text_to_int(row_label):
  if row_label == 'dent':
    return 1
  if row_label == 'scratch':
    return 2
  if row_label == 'Zity':
    return 3
  if row_label == 'front fog light':
    return 4
  else:
    None

def split(df, group):
  data = namedtuple('data', ['filename', 'object'])
  gb = df.groupby(group)
  return [data(filename, gb.get_group(x)) for filename, x in zip(gb.groups.keys(), gb.groups)]
```

(a) Tf record generation: cell 1

```
def create_tf_example(group, path):
  with tf.io.gfile.GFile(os.path.join(path, '{}'.format(group.filename)), 'rb') as fid:
    encoded_jpg = fid.read()
  encoded_jpg_io = io.BytesIO(encoded_jpg)
  image = Image.open(encoded_jpg_io)
  width, height = image.size
  filename = group.filename.encode('utf8')
  image_format = b'jpg'
  xmins = []
  xmaxs = []
  ymins = []
  ymaxs = []
  classes_text = []
  classes = []
  for index, row in group.object.iterrows():
    xmins.append(row['xmin'] / width)
    xmaxs.append(row['xmax'] / width)
    ymins.append(row['ymin'] / height)
    ymaxs.append(row['ymax'] / height)
    classes_text.append(row['class'].encode('utf8'))
    classes.append(class_text_to_int(row['class']))
  tf_example = tf.train.Example(features=tf.train.Features(feature={
    'image/height': dataset_util.int64_feature(height),
    'image/width': dataset_util.int64_feature(width),
    'image/filename': dataset_util.bytes_feature(filename),
    'image/source_id': dataset_util.bytes_feature(filename),
    'image/encoded': dataset_util.bytes_feature(encoded_jpg),
    'image/format': dataset_util.bytes_feature(image_format),
    'image/object/bbox/xmin': dataset_util.float_list_feature(xmins),
    'image/object/bbox/xmax': dataset_util.float_list_feature(xmaxs),
    'image/object/bbox/ymin': dataset_util.float_list_feature(ymins),
    'image/object/bbox/ymax': dataset_util.float_list_feature(ymaxs),
    'image/object/class/text': dataset_util.bytes_list_feature(classes_text),
    'image/object/class/label': dataset_util.int64_list_feature(classes),
    }))
  return tf_example
```

(b) Tf record generation: cell 2

```
#creates tfrecord for both csv's
for csv in ['train_labels', 'test_labels']:
  writer = tf.io.TFRecordWriter(data_base_url + csv + '.record')
  path = os.path.join(image_dir)
  examples = pd.read_csv(data_base_url + csv + '.csv')
  grouped = split(examples, 'filename')
  for group in grouped:
    tf_example = create_tf_example(group, path)
    writer.write(tf_example.SerializeToString())
  writer.close()
  output_path = os.path.join(os.getcwd(), data_base_url + csv + '.record')
  print('Successfully created the TFRecords: {}'.format(data_base_url +csv + '.record'))
```

(c) Tf record generation: cell 3

Figure 3.10: Script sections for the record file generation

Chapter 4

Experiments and results

4.1 Experiments plan introduction

As explained in the previous section 3.3.1, the dataset was in continuous expansion. In general, it was decided to follow a linear increase in the amount of the added pictures, by fixing some partitions to be tested on the path (except for the last one, where we made some different modifications regarding the labelling operation, fully detailed in section 4.6):

- *Dataset200*;
- *Dataset500*;
- *Dataset750*;
- *Dataset1100*;
- *Dataset1100_new*.

4.2 First test dataset: *"Dataset 200"*

Tests started on a first version of dataset of about 200 images, using a small but fast model, already analyzed in Section 2.4.3 (SSD MobileNet).
The result were satisfactory compared to the expectations, but following the literature on the state-of-the-art models for machine learning (as explained deeply in Section 2.5), we decided to take into account the object detection API support for Tensorflow 2 and to concentrate our efforts on pre-trained

models on COCO dataset, as EfficientDet models.

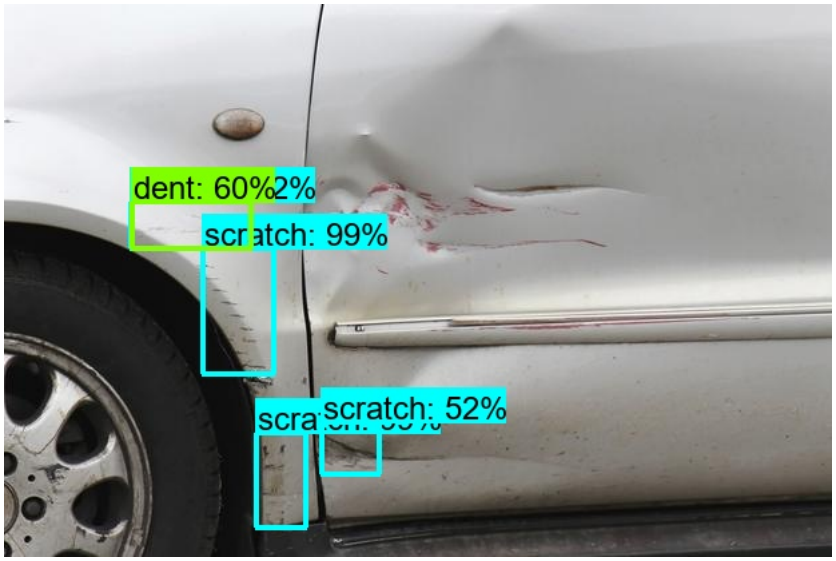


Figure 4.1: Model based on EfficientDetD2 (*Dataset 200*)

This was a crucial step: the initial dataset, we built, was producing the first results (look at Figure 4.1 and Table 4.1) and we could focus building a larger one in order to improve them significantly.

Pre-trained model	**mAP(%)**	**mAR(%)**
EfficientDetD0	7.7	11.5
EfficientDetD1	8.6	11.9
EfficientDetD2	9.1	14.0

Table 4.1: Mean values of the evaluation metrics - Average Precision (AP) and Average Recall (AR) - relative to models trained on *Dataset 200*

4.3 Second test dataset: *"Dataset 500"*

Carrying on with a larger dataset, of about 500 images, prevents us to train more accurate models (e.g. EfficientDetD3).
Within this second image dataset, in general, more pictures relative to damaged cars containing dents on the car body were included, in order to obtain a more balanced effect during the training between the two classes (*"scratch"*

and (*"dent"*). Most of them were collected from web sources, with a different resolution with respect to the rest of the other photos taken manually. It could be this, the reason why the improvement visible in the obtained values reported in Table 4.2 did not result as meaningful as expected.

Pre-trained model	**mAP(%))**	**mAR(%)**
EfficientDetD0	10.9	20.1
EfficientDetD1	12.3	18.2
EfficientDetD2	15.0	28.4
EfficientDetD3	14.9	23.2

Table 4.2: Mean values of the evaluation metrics - Average Precision (AP) and Average Recall (AR) - relative to models trained on *Dataset 500*

4.4 Third test dataset: *"Dataset 750"*

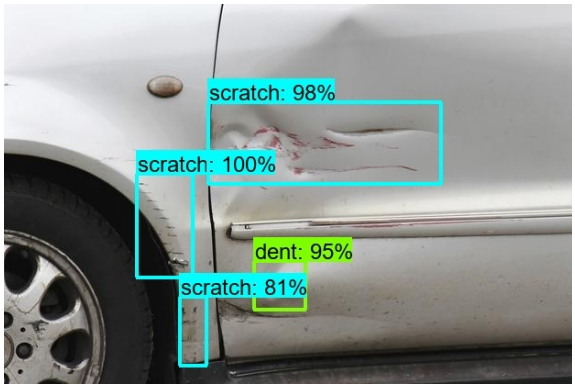
During the tests made by exploiting a third version of dataset (with an amount of 750 pictures), we obtained results which point out the improved quality (and quantity, of course) of the dataset.

Figure 4.2: Object detection on a sample using SSD EfficientDetD2 - on the left - the ground truth bounding boxes made during labelling, used for training - on the right

In Figure 4.2, it is possible to observe a comparison between the detections performed by the neural network and the ground truth, that is a real sample, labelled by hand for *Dataset 750*. This kind of comparison is done on the validation set, to evaluate the performance of the network and obtain the evaluation metrics reported on Figure 4.3a.
In Figure 4.3b is reported the same image of Figure 4.1, processed with EfficientDetD2 network after being trained on *Dataset 750*. To be noted the improvement in the detection and in the superposition of the bounding boxes (considering that often the damages are a combination of dents and scratches).

```
Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.149
Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.391
Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.086
Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.411
Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.139
Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.164
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.135
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.253
Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.299
Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.421
Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.286
Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.308
```

(a) Model evaluation metrics



(b) Model behaviour on a test sample

Figure 4.3: Model based on EfficientDetD2 (*Dataset 750*)

We managed to build a dataset for any generic kind of car, in generic light conditions, both inside (e.g. in a garage) or outside. In the former case, during the test on the *Dataset 750*, we started working to improve the accuracy, even though the results show still a low performance. This could be due to lack of samples in those specific conditions within the dataset used during the training phase and, furthermore, in a closed place with artificial

lights, reflexes created on some car surfaces can be easily mistaken and be interpreted as scratches (keeping in mind that also for a human this task is not so easy, as in Figure 4.4).

Figure 4.4: Object detection on a car surface with the presence of reflexes (model based on EfficientDetD2)

In the following table (Table 4.3) an overview of the best average results obtained on *Datast 750* is presented, in terms of evaluation metrics, corresponding to all the implemented models.

Pre-trained model	**mAP(%))**	**mAR(%)**	**Data augmentation notes**
EfficientDetD1	18.6	29.1	*Random horizontal flips*
EfficientDetD1	17.0	30.1	*Random horizontal flips* *Random scale, crop , pad to square* *Random rgb to gray*
EfficientDetD2	23.5	28.1	No data augmentation
EfficientDetD2	22.7	29.0	*Random horizontal flips*
EfficientDetD2	17.4	30.1	*Random horizontal flips* *Random scale, crop, pad to square* *Random rgb to gray*
EfficientDetD3	25.1	28.7	*Random horizontal flips*
EfficientDetD4	11.6	26.7	*Random horizontal flips*

Table 4.3: Mean values of the evaluation metrics - Average Precision (AP) and Average Recall (AR) - relative to models trained on *Dataset 750*

As it is possible to notice in Table 4.3 some tests were made by introducing in the configuration file also an important regularization technique, that is **data augmentation**. It is a widely adopted procedure used to increase the diversity of the dataset by applying some pre-processing methods. Commonly they are applied randomly, but they have to be meaningful and realistic for the current use case. As example, firstly we tried to apply only the *random horizontal flips*, because it makes sense for a car, while vertical flip is quite unrealistic. In a second moment, we have taken into account also the *random scale, crop and pad to square* and the *random rgb to gray*, but, from the results, it is evident that they are not proper for this use case.
Moreover, even if there are no strict timing constraints during execution, we performed a lot of tests and decided to evaluate a comparison between the processing time distinguishing each model. In the table below (Table 4.4), some results are reported regarding the time to process a test video of 38 seconds (resolution 1280x720, 30 fps) on an I7-8550-U Intel processor. Considering 30 frames per seconds, in 38 seconds we have 30x38=1140 images, thus, just to give an example, for the case of EfficienDetD2, the inference time for a single image is (60x21)/1140, that is about 1.1 sec.

Model	**Processing time [minutes]**
EfficientDetD0	~ 7
EfficientDetD1	~ 12
EfficientDetD2	~ 21
EfficientDetD3	~ 39
EfficientDetD4	~ 68

Table 4.4: Processing time relative to all the implemented models – the video resolution is 1280x720, 30 fps and it lasts 38 seconds

From Table 4.4, it is possible to observe the huge amount of processing time corresponding to the model EfficientDetD4; for this reason, after some considerations, we have analyzed a trade-off between speed and accuracy and we have discarded this model in the following test described in the next sections.

4.5 Fourth test dataset: *"Dataset 1100"*

In a fourth dataset collection of about 1100 images, we tried to enlarge our dataset integrating new photos with similar features to the previous one. Following a linear increase in the amount of the added pictures, we included car images in light conditions that seemed to be closer to the real situation in the garage. Most of the photos were shot in the evening, with streetlights switched on or inside underground car parking in order to improve this feature in the learning process of the neural network.
The gained results are reported in the following table (Table 4.5).

Pre-trained model	**mAP(%))**	**mAR(%)**	**Data augmentation notes**
EfficientDetD1	11.9	24.6	*Random horizontal flips*
EfficientDetD2	15.3	29.2	No data augmentation
EfficientDetD2	13.3	26.6	*Random horizontal flips*
EfficientDetD3	10.5	22.1	*Random horizontal flips*

Table 4.5: Mean values of the evaluation metrics - Average Precision (AP) and Average Recall (AR) - relative to models trained on *Dataset 1100*

As the low values in almost all the parameters, in spite of the improvement both in quality and in quantity of the dataset, we decided to make some modifications in the preparation of the next dataset version (*"Dataset 1100_new"*, section 4.6).

4.6 Fifth test dataset: *"Dataset 1100_new"*

In the last collection of images, we observed that the introduction of the Zity cars pictures led to reconsider the number and the typology of classes set in the earlier labelling operation, as previously specified in Section 3.3.1. As visible in Figure 4.5, the two additional elements that caused problems were: the front fog lamp, characterized by an uncommon shape, and the presence of the text "Zity" on the car body. Therefore, during the preparation of (*Dataset 1100_new*), the labelling action, performed on the new added photos, included two further classes of label ("Zity" and "front fog light"), as previously shown in the label map in Figure 3.9.

Figure 4.5: Object detection on a Zity car sample where the front fog light and the text are recognized as scratches (model based on EfficientDetD2)

After these modifications and a further cleaning of the previous versions of dataset (we just deleted most of the images with too low resolution downloaded from the web), we obtained satisfactory results, as reported in Table 4.6 and displayed in Figure 4.6.

Pre-trained model	**mAP(%))**	**mAR(%)**	**Data augmentation notes**
EfficientDetD1	34.8	38.7	No data augmentation
EfficientDetD1	38.2	42.6	*Random horizontal flips*
EfficientDetD2	35.4	38.9	No data augmentation
EfficientDetD2	46.4	49.9	*Random horizontal flips*
EfficientDetD3	44.8	51.5	*Random horizontal flips*

Table 4.6: Mean values of the evaluation metrics - Average Precision (AP) and Average Recall (AR) - relative to models trained on *Dataset 1100_new*

Figure 4.6: Object detection on a Zity car sample where the object detection is performed correctly (model based on EfficientDetD2)

Chapter 5

Conclusion and future works

The study presented in this thesis showed that the image recognition and detection approach selected is pretty functional from a point of view of the accuracy performance, even if the task was very challenging. The choice of transfer learning as starting point, made after an accurate comparison between the different typologies of object detection algorithms (single-stage detector/two-stage detector) and pre-trained models on huge datasets (COCO dataset), as deepened in Section 2.5, proved to be suitable for this use case. In particular, we observed that EfficientDet pre-trained models(D0, D1, D2, D3, D4) obtained good performances, in terms of the most relevant evaluation metrics (like Average Precision, Average Recall).
The proposed method was carried out by growing up the dataset by successive subsets, collected, on the basis of the results (reported in Chapter 5) achieved along the path: it was possible to notice that the optimal preparation of the whole dataset have represented the key of the improvement for the final results (highlighted in Table 4.6) . The turning-point it has been seen especially in the introduction of realistic images (*"Dataset 1100_new"*, section 4.6) in the initial version of datasets, where the neural network had the possibility to train the specific features belonging to the real cases.
The final results show that, thanks to the quality of gathered dataset and the refinement technique, the built models are able to improve by about 25% the mean Average Precision (mAP) and the mean Average Recall (mAR) score with respect to the starting values. This is a relevant goal, because they are the standard accuracy scores that provide an idea of the amount of possible

false positive or false negative cases during the object detection implementation.
In fact, during the experiments we have seen that also the bounding boxes drawn on images samples have identified the objects more precisely, avoiding other elements representing false positive (e.g. reflexes, front fog lights, labels/text on the car body surface). In particular, meaningful results have been given by the models based on the pre-trained models EfficientDet D2 and EfficientDetD3, where the confidence score reached very high value as previously displayed in Figure 4.6 (Section 4.6).

In spite of the main objective of the project has been successfully achieved, some improvements should be investigated to enhance the system for a refined application.
Working toward that, from a dataset point of view, a further research of images could be achieved in order to act on its quantity and its quality: the total number of the Zity car pictures, specific for this use case, should be increased with respect to the rest showing the generic cars typologies and also their resolution should be improved.
In addition to an high-quality dataset, it could be useful investigate some other common neural network "tricks", including both further regularization techniques (e.g. dropout) or the implementation of a variety of optimization algorithms, such as mini-batch gradient descent, RMSprop, Adam, so that the models already gained could be further customized in terms of efficiency and best performance.

List of Figures

2.1 Symbolic representation of a biological neuron (left) and its mathematical model (right) 12
2.2 Commonly used activation functions 13
2.3 Fully-connected layers in two examples of Neural Networks . . 14
2.4 Softmax function with a probability distribution 15
2.5 Loss function in Support Vector Machine 15
2.6 Class scores scheme in SVM 16
2.7 Architecture of the Convolutional Neural Network 17
2.8 Filtering operation in Convolutional layer 18
2.9 Computation of the output feature map size 19
2.10 Filtering operation with "Same zero-padding" mode 20
2.11 Average and max pooling oparation schemes 22
2.12 Key components associated with deep learning-based anomaly detection technique 23
2.13 Main machine vision approaches (from left): Semantic segmentation, Classification + Localization, Object detection, Instance segmentation 26
2.14 SSD and YOLO architectures 28
2.15 Architecture of a convolutional neural network with a SSD detector 29
2.16 Evaluation metrics formulas 30
2.17 Intersection of Unit scheme definition 31
2.18 MobilNet v1 and v2 architectures 32
2.19 MobilNet v1 and v2 parameters 32
2.20 EfficientNet accuracy on Imagenet (on the left) EfficientDet AP on COCO (on the right) 33
2.21 EfficientDet performance on COCO 34

3.1 High-Level System Architecture 35
3.2 Position of the components 37
3.3 Real view from the cameras 37
3.4 Damage detection flow chart 39
3.5 Examples of: discarded images (a)-(b) and included images (c)-(d) 40
3.6 Labelling steps representation 41
3.7 Example of LabelImg output file in *PascalVoc XML* format . . 43
3.8 Tensorflow 2.0 architecture 46
3.9 Label map containing all the classes 47
3.10 Script sections for the record file generation 50

4.1 Model based on EfficientDetD2 (*Dataset 200*) 52
4.2 Object detection on a sample using SSD EfficientDetD2 - on the left - the ground truth bounding boxes made during labelling, used for training - on the right 53
4.3 Model based on EfficientDetD2 (*Dataset 750*) 54
4.4 Object detection on a car surface with the presence of reflexes (model based on EfficientDetD2) 55
4.5 Object detection on a Zity car sample where the front fog light and the text are recognized as scratches (model based on EfficientDetD2) 58
4.6 Object detection on a Zity car sample where the object detection is performed correctly (model based on EfficientDetD2) 59

List of Tables

4.1 Mean values of the evaluation metrics - Average Precision (AP) and Average Recall (AR) - relative to models trained on *Dataset 200* 52
4.2 Mean values of the evaluation metrics - Average Precision (AP) and Average Recall (AR) - relative to models trained on *Dataset 500* 53
4.3 Mean values of the evaluation metrics - Average Precision (AP) and Average Recall (AR) - relative to models trained on *Dataset 750* 55
4.4 Processing time relative to all the implemented models – the video resolution is 1280x720, 30 fps and it lasts 38 seconds . . 56
4.5 Mean values of the evaluation metrics - Average Precision (AP) and Average Recall (AR) - relative to models trained on *Dataset 1100* 57
4.6 Mean values of the evaluation metrics - Average Precision (AP) and Average Recall (AR) - relative to models trained on *Dataset 1100_new* 58

Bibliography

[1] P. SAROCA, "D01 - early bird activity report," 2019.

[2] "How to prepare for an automated future: 7 steps to machine learning." https://vanrijmenam.nl/prepare-for-automated-future-7-steps-machine-learning/, 2019.

[3] "What is machine learning?." https://www.mathworks.com/discovery/machine-learning.html.

[4] R. Chalapathy and S. Chawla, "Deep learning for anomaly detection: a survey." https://arxiv.org/pdf/1901.03407.pdf, Jan 2019.

[5] "Convolutional neural networks for visual recognition." https://cs231n.github.io/neural-networks-1/, 2020.

[6] "Neural networks: a modern introduction." https://compsci682.github.io/notes/neural-networks-1/, 2020.

[7] "Support vector machines (svms)." https://medium.com/@aaaanchakure/support-vector-machines-svms-4bcccbd78369, 2019.

[8] "Explained: Deep learning in tensorflow — chapter 0." https://towardsdatascience.com/explained-deep-learning-in-tensorflow-chapter-0-acae8112a98, 2019.

[9] "You only look once (yolo) real-time object detection." https://www.cpp.edu/~honorscollege/documents/convocation/EGR/ECE_Keil.pdf.

[10] "Cnn convolutional layer in depth." https://sameerbairwa07.medium.com/convolutional-layer-ab60395b8b27.

[11] "A comprehensive guide to convolutional neural networks - the eli5 way." https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53, 2018.

[12] "A gentle introduction to pooling layers for convolutional neural netwoks." https://machinelearningmastery.com/pooling-layers-for-convolutional-neural-networks/, 2019.

[13] Z. L. Shuai Shao, "Objects365: A large-scale, high-quality dataset for object detection." https://openaccess.thecvf.com/content_ICCV_2019/papers/Shao_Objects365_A_Large-Scale_High-Quality_Dataset_for_Object_Detection_ICCV_2019_paper.pdf, 2019.

[14] "How single-shot detector (ssd) works?." https://developers.arcgis.com/python/guide/how-ssd-works/.

[15] https://cocodataset.org/.
[16] O. Liu, L., "Deep learning for generic object detection: A survey." https://link.springer.com/ONLINE/10.1007/s11263-019-01247-4, 2020.
[17] "Single shot object detection ssd using mobilenet and opencv." https://honingds.com/blog/ssd-single-shot-object-detection-mobilenet-opencv/.
[18] "Mobilnet version 2." https://machinethink.net/blog/mobilenet-v2/.
[19] R. P. Mingxing Tan, "Efficientdet: Scalable and efficient object detection." https://arxiv.org/abs/1911.09070, Jul 2020.
[20] P. SAROCA, "D03 - report on technical specification for the real-time object/defect recognition solution," Sep 2020.
[21] "Pythorch vs tensorflow in 2020." https://towardsdatascience.com/pytorch-vs-tensorflow-in-2020-fe237862fae1, 2020.
[22] "Tensoflow 2.0.0." https://github.com/tensorflow/tensorflow/releases/tag/v2.0.0, 2019.