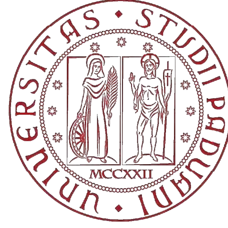



**Università degli Studi di Padova**

Department of Information Engineering


*Master Thesis in* ICT for Internet and Multimedia - Life & Health

# Automatic Product Recognition using Deep Learning: a Retail Industry Application


*Supervisor*
Prof. Stefano Ghidoni

*Co-supervisor*
Francesco Coccia
Reply Technology

*Master Candidate*
Gianluca Fighera



Academic Year 2018/2019

*"I've missed more than 9000 shots in my career.*
*I've lost almost 300 games.*
*26 times, I've been trusted to take the game winning shot and missed.*
*I've failed over and over and over again in my life.*
*And that is why I succeed."*

Michael J. Jordan

# Abstract

In this thesis, an automatic products recognition system has been developed using deep learning techniques. After a detailed study of the state-of-the-art object detection frameworks and the image classification networks, the dataset has been acquired and manually labelled. Several frameworks have been trained and tested to select the most suitable for this application. Results of each model are provided in this thesis, together with some considerations. At the end, the most performing object detection system has been encapsulated inside a web service environment that provides a friendly and easy-to-use interface to load an image, detect the products and return the results. The entire project has been developed using Python with TensorFlow as deep learning framework and Flask as web service environment.

# Acknowledgements

I would like to thank the professor Stefano Ghidoni for giving me the opportunity to carry on this project together with the company *Technology Reply S.R.L.* in Turin. I would also like to thank him for supervising my work and for giving me important advice on the completion of this thesis.

I would like to reserve a special thank to Francesco Coccia, the manager of *Technology Reply S.R.L.* in Turin, who presented me the project more than a year ago and kept in touch with me during my last semester in Padova before I went to Turin. I would also like to thank him for providing assistance during the course of this project.

I would like to thank Corrado De Bari, of *Oracle*, who participated in the project by assisting me and other colleagues in the developing steps. He also provided the hardware to train the models, one of the most important resource.

Finally, I would like to thank the company *Technology Reply S.R.L.* for giving me the opportunity to develop my master thesis in their offices gaining experience by working with more experienced colleagues. Moreover, I had the opportunity to observe closely the corporate issues and to learn how different is the job from the university.

# Contents


**1 Introduction** **1**

**2 State of the Art and General Concepts** **4**
- 2.1 General concepts . . . 4
  - 2.1.1 Deep Neural Networks . . . 8
  - 2.1.2 Convolutional Neural Networks . . . 8
- 2.2 History of Object Recognition . . . 11
  - 2.2.1 LeNet 1998 . . . 12
  - 2.2.2 AlexNet 2012 . . . 13
  - 2.2.3 OverFeat . . . 14
  - 2.2.4 ZFNet . . . 15
  - 2.2.5 GoogleNet and Inception v1 . . . 16
  - 2.2.6 VGGNet . . . 18
  - 2.2.7 Inception v2 and v3 . . . 19
  - 2.2.8 ResNet . . . 20
  - 2.2.9 Inception v4 and Inception ResNet v1 and v2 . . . 22
  - 2.2.10 ResNext . . . 23
  - 2.2.11 DenseNet . . . 24
  - 2.2.12 MobileNet v1 and v2 . . . 25
- 2.3 Object Detection Frameworks . . . 26
  - 2.3.1 R-CNN . . . 26
  - 2.3.2 Fast R-CNN . . . 28
  - 2.3.3 Faster R-CNN . . . 30
  - 2.3.4 YOLO . . . 31
  - 2.3.5 SSD . . . 33
  - 2.3.6 YOLOv2 and YOLO9000 . . . 34
  - 2.3.7 YOLOv3 . . . 35

**3 Description of the Project** **38**
- 3.1 Object Detection Module . . . 38
- 3.2 Web Service Module . . . 40

**4 Setup of Components** **41**
- 4.1 TensorFlow . . . 41
  - 4.1.1 TensorFlow Model Zoo . . . 42
- 4.2 Hardware . . . 43
- 4.3 LabelImg . . . 44

4.4 Flask . . . . . . . . 44

**5 Dataset and Models 46**

5.1 Dataset . . . . . . . . 46

5.1.1 Hierarchical representation of objects . . . . . . . . 47

5.1.2 Dataset *Oracle* . . . . . . . . 48

5.1.3 Dataset *Reply* . . . . . . . . 48

5.1.4 Dataset *Test* $\alpha$ . . . . . . . . 51

5.1.5 Dataset *Test* $\beta$ . . . . . . . . 51

5.2 V0: Models trained with Dataset *Oracle* . . . . . . . . 51

5.3 V1: SSDLite MobileNet v2 . . . . . . . . 54

5.4 V2: Faster R-CNN ResNet50 . . . . . . . . 55

5.5 V3: Faster R-CNN Inception v2 . . . . . . . . 56

**6 Results and Discussion 57**

6.1 V1: SSDLite Mobilenet v2 . . . . . . . . 59

6.2 V2: Faster R-CNN ResNet50 . . . . . . . . 64

6.3 V3: Faster R-CNN Inception v2 . . . . . . . . 68

**7 Applications 76**

7.1 Oracle - Proxima Smart City . . . . . . . . 76

7.2 Reply - Automatic Products Recognition . . . . . . . . 79

**8 Conclusions 83**

**Bibliography 86**

# List of Figures


2.1 Matrix representation of an image . . . . . 6
2.2 Classification and detection . . . . . 6
2.3 Machine learning approaches . . . . . 7
2.4 Artificial neuron . . . . . 8
2.5 Convolution operation . . . . . 10
2.6 Activation functions . . . . . 10
2.7 Pooling operation . . . . . 10
2.8 LeNet-1 . . . . . 12
2.9 LeNet-5 . . . . . 12
2.10 AlexNet . . . . . 14
2.11 OverFeat architecture . . . . . 14
2.12 OverFeat phases . . . . . 15
2.13 Deconvolutional layer of ZFNet . . . . . 16
2.14 Inception module of GoogleNet . . . . . 17
2.15 GoogleNet . . . . . 18
2.16 VGGNet . . . . . 19
2.17 Inception v3 . . . . . 20
2.18 Residual block of ResNet . . . . . 21
2.19 ResNet . . . . . 22
2.20 Inception v4 . . . . . 23
2.21 DenseNet . . . . . 25
2.22 Selective Search algorithm results . . . . . 27
2.23 Selective Search algorithm . . . . . 28
2.24 R-CNN . . . . . 28
2.25 Fast R-CNN . . . . . 29
2.26 Faster R-CNN anchors . . . . . 30
2.27 Faster R-CNN . . . . . 31
2.28 YOLO tensor . . . . . 32
2.29 YOLO algorithm . . . . . 33
2.30 SSD . . . . . 34
2.31 YOLO v2 boxes . . . . . 35
2.32 Word Tree . . . . . 36
2.33 YOLO v3 . . . . . 36

3.1 Project diagram . . . . . 38
3.2 Web app . . . . . 40

4.1 Example of TF code . . . . . 42

4.2 TF graph . . . 42
4.3 TensorFlow Model Zoo pipeline . . . 43
4.4 LabelImg tool . . . 44
4.5 LabelImg tool . . . 45
4.6 Flask Web Service example . . . 45

5.1 XML file with annotations . . . 47
5.2 BBox conventions . . . 48
5.3 Dataset *Oracle* classes distribution . . . 49
5.4 Dataset *Oracle* examples . . . 49
5.5 Dataset *Reply* classes distribution . . . 50
5.6 Dataset *Reply* objects per image distribution . . . 50
5.7 Dataset *Reply* examples . . . 51
5.8 Dataset *Test* $\alpha$ classes distribution . . . 52
5.9 Dataset *Test* $\alpha$ objects per image distribution . . . 52
5.10 Dataset *Test* $\alpha$ examples . . . 53
5.11 Dataset *Test* $\beta$ classes distribution . . . 53
5.12 Dataset *Test* $\beta$ objects per image distribution . . . 54
5.13 Dataset *Test* $\beta$ examples . . . 54
5.14 Model *v0* results . . . 55
5.15 Model *v0* results . . . 55
5.16 Parameters of model *v1* . . . 55
5.17 Parameters of model *v2* . . . 56
5.18 Parameters of model *v3* . . . 56

6.1 IoU definition . . . 58
6.2 Accuracy SSDLite MobileNet v2 ($2292 \times 1719$) . . . 60
6.3 Accuracy SSDLite MobileNet v2 ($1024 \times 768$) . . . 60
6.4 Accuracy SSDLite MobileNet v2 ($800 \times 600$) . . . 61
6.5 Accuracy SSDLite MobileNet v2 ($4032 \times 3024$) . . . 61
6.6 Accuracy SSDLite MobileNet v2 ($2297 \times 1292$) . . . 62
6.7 Accuracy SSDLite MobileNet v2 ($800 \times 450$) . . . 62
6.8 SSDLite MobileNet v2 metrics . . . 63
6.9 SSDLite output on dataset *Test* $\alpha$ . . . 64
6.10 SSDLite output on dataset *Test* $\beta$ . . . 64
6.11 Accuracy Faster R-CNN ResNet50 ($2292 \times 1719$) . . . 65
6.12 Accuracy Faster R-CNN ResNet50 ($1024 \times 768$) . . . 66
6.13 Accuracy Faster R-CNN ResNet50 ($800 \times 600$) . . . 66
6.14 Accuracy Faster R-CNN ResNet50 ($4032 \times 3024$) . . . 67
6.15 Accuracy Faster R-CNN ResNet50 ($2297 \times 1292$) . . . 67
6.16 Accuracy Faster R-CNN ResNet50 ($800 \times 450$) . . . 68
6.17 Faster R-CNN ResNet50 metrics . . . 69
6.18 Faster R-CNN ResNet output on dataset *Test* $\alpha$ . . . 69
6.19 Faster R-CNN ResNet output on dataset *Test* $\beta$ . . . 70
6.20 Accuracy Faster R-CNN Inception v2 ($2292 \times 1719$) . . . 70
6.21 Accuracy Faster R-CNN Inception v2 ($1024 \times 768$) . . . 71
6.22 Accuracy Faster R-CNN Inception v2 ($800 \times 600$) . . . 72
6.23 Accuracy Faster R-CNN Inception v2 ($4032 \times 3024$) . . . 72

6.24 Accuracy Faster R-CNN Inception v2 ($2297 \times 1292$) . . . . . . . . . . . 73
6.25 Accuracy Faster R-CNN Inception v2 ($800 \times 450$) . . . . . . . . . . . . 73
6.26 Faster R-CNN Inception v2 metrics . . . . . . . . . . . . . . . . . . . . . . 74
6.27 Faster R-CNN Inception output on dataset *Test* $\alpha$ . . . . . . . . . . . 74
6.28 Faster R-CNN Inception output on dataset *Test* $\beta$ . . . . . . . . . . . 75
6.29 Inference time analysis . . . . . . . . . . . . . . . . . . . . . . . . . . . 75

7.1 Oracle Proxima Smart City . . . . . . . . . . . . . . . . . . . . . . . . . 77
7.2 JSON file for the request . . . . . . . . . . . . . . . . . . . . . . . . . . 77
7.3 Detection information . . . . . . . . . . . . . . . . . . . . . . . . . . . . 78
7.4 JSON file for the response . . . . . . . . . . . . . . . . . . . . . . . . . 79
7.5 JSON file for the request . . . . . . . . . . . . . . . . . . . . . . . . . . 80
7.6 DM_PRODOTTI . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 80
7.7 DM_SCAFFALI . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 81
7.8 FT_PRODOTTI_SCAFFALI . . . . . . . . . . . . . . . . . . . . . . . . . 81
7.9 Power BI report . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 82
7.10 Power BI report . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 82

# Chapter 1

# Introduction

According to the Visual Networking Index (VNI) of Cisco, in 2022 the amount of data generated and exchanged over the Internet will be the same of the previous 32 years. The reasons behind this exponential increase is the ever growing number of digital devices that can generate data. Just to make some examples, nowadays' social networks like Facebook, Instagram and YouTube enable to load images and videos in a very simple way. Together with this evolution, there is the need for the telecommunication companies to keep abreast and to design efficient transmission protocols to manage the data traffic. On the other hand, the large quantity of data, in particular images and videos, available online has opened the doors to new frontiers in the statistical analysis. To be more specific, having the opportunity to analyze big quantities of data is of extreme interest for the companies in many fields. For example, having the transaction history is important for the banks in order to estimate their profits. Moreover, in the financial field, data from the past can be used to make predictions on the market stock.

This led to the development of a new field called machine learning. In short words, the goal of machine learning is to use the available data to build and train mathematical models with the objective to reproduce a real system and to handle it. It is clear that, without data, it would have been impossible to develop such models.

Machine learning has been a field of extreme interest for the last 40 years, and its applications cover many aspects. One of this is computer vision, where machine learning techniques have given the opportunity to develop better models (neural networks). In this way, the huge amount of images and videos available online can be used to train object detection models.

The goal of object detection is to determine whether there are instances of some pre-defined objects in a given image and, if present, to return the spatial location of each object instance. This raw information can be used to develop more complex systems such as pedestrian detectors in self-driving cars, activities recognizer, intelligent video surveillance and many others.

Nowadays, every daily-use electronic device equipped with a camera is able to "see" the world, hence, it is able to acquire a huge quantity of data and information. However, those are useless without a system that allows to understand the scene. If we were able to process that data in an intelligent way, we could extract information for higher level applications.

One of these applications of object detection is automated self-checkout. In the age of e-Commerce, where companies sell their products on the Internet allowing people to buy something with just a click, there is the need, for physical stores, to invest on technologies that enable them to keep abreast. One of the main downsides that emerges if you take a look at any store, is the huge checkout line, which is the bottleneck of the time spent doing shopping. In the recent years, some stores have placed some self-checkout where customers can get themselves the ticket by passing the goods on the screen like the attendant did in the old checkouts. This has speeded up the process, but what if we could make it completely automated?

What if we could watch over every customer in the store using some cameras (this is actually already done by the surveillance cameras, although the registrations are not processed for these purposes) and to detect when each of them takes a good from the shell and put it in the shopping cart? If we were able to do it, then we could build a sort of virtual ticket for each customer and, once he/she wants to exit, he/she just has to swipe the credit card. While the second part of the process only needs a database to be maintained and a web service running on it, the first part is a computer vision application, since every input video from the cameras needs to be processed by an object detection algorithm: some videos can be used to run people detection and tracking algorithms, while others can be used for detecting and classifying goods on the shelves. The final goal is to elaborate the inputs from all the cameras and to transform them to an information like "Customer *cust_15* has taken 2 instances of product *prod_34*". With this information, the virtual ticket of customer *cust_15* is updated with 2 instances of product *prod_34*. If the customer, even after a while, goes back to the shelf and leave one of the two products he has bought before, the system should update the virtual ticket accordingly. If such a system works properly, the process of automatic self-checkout becomes relatively simple, since the system only needs to associate each customer a credit card and, when the customer leaves the store, the bill is payed automatically.

The information produced by the system for each customer interaction is the result of a complex process that requires several parts: first of all, the surveillance system should be able to recognize and precisely track every people in the store since two different customers have to be associated with two different tickets; then, assuming that each camera is dedicated to a particular shell, it should be able to detect and identify each good in the particular shell; moreover, it should be able to perform activity recognition to detect the gesture of the customer that takes the product but also to discriminate whether the customer replaces it or puts it into the shopping cart.

This has been actually already done by *Amazon.com Inc*, the most famous american online retailer. In the recent years, they developed *Amazon Go*, which is a checkout-free store. The idea of Amazon Go is to install a set of sensors and cameras, as described above, to keep track of the customers' inside the store and their activity. When a customer enters the store and registers with his Amazon account, an empty virtual bill is created and associated to that account. While he moves through the store, the cameras track his activity real-time and, if he stops in front of a shelf, the cameras that check the products on that shelf are activated. When the customer takes a product and puts it into his shopping bag, his virtual bill is updated with the correspondent item. In the end, each customer exiting the store

with some products will have accumulated a virtual bill, which is automatically withdrew from the bank account associated to his Amazon account.

There are several aspects that emerge from such a system. The most important one is that the system should be reliable and the integrity of data has to be maintained. In fact, any error in the acquisition of information results in a incorrect payment, which is a damage both for the customer and for the store. For example, if a good is incorrectly classified by the object detection system, the ticket is wrongly updated. Moreover, if the customer replaces that good in the shelf and the system now correctly classifies it, it becomes aware that there was an error since the good doesn't appear in the virtual ticket. Also, mismatching of people should be handled properly, since tickets may be assigned to the incorrect person, who will pay 100$ for some apples. Finally, if the transaction fails and the customer exits the store, there should be a quick way to get aware of it and to handle the error.

However, this kind of technology is of interest not only for giant companies as Amazon or Google. In fact, having a system that keeps a database updated in real time with all the products in the shelves detected by cameras is helpful since it reduces staff effort of doing the inventory. Moreover, if the store runs out of a product, the cameras in the shelf detect it and may generate an order for the supplier. In general, once the results of object detection system are available, it is up to the company to decide how to use it.

# Chapter 2

# State of the Art and General Concepts

In this chapter, an overview of the general concepts of object detection is given, together with the analysis of the state-of-the-art in the field. Firstly, there is an introduction of machine learning and computer vision fields. Then, deep learning is introduced as a powerful tool to increase algorithm performances with its models like recurrent neural networks and convolutional neural networks, with a particular focus on the latter. Finally, the evolution of the object detection task is reviewed, starting from the pioneer models in the 80s until the modern architectures like Faster R-CNN, YOLO and SSD.

## 2.1 General concepts

Machine learning is the field of artificial intelligence which studies the algorithms that enable the computers to learn from experience, without being explicitly programmed. The process of learning is carried on by feeding a model with real data allowing them to find patterns and structures that may be useful in order to build a mapping function which is able to solve a particular problem. Once the learning process is over, the algorithm allows the system to interpret new data and to produce an output which is consistent with training data. Hence, using a trained model, the system is able to automatically interpret and process new incoming data.

Machine learning algorithms can be classified according to the philosophy of their training phase, in particular when can distinguish, among all, two types of algorithms:

- supervised learning: where the model is provided both with data and the expected output, and so it is explicitly guided to the solution;
- unsupervised learning: where the model is provided only with data, giving it the freedom to produce the output.

Supervised learning algorithms are used to solve tasks like classification or regression, where the aim is to build a mathematical function that maps the input data to an output, that is often a scalar value. In particular, in regression tasks the

output is a value sampled from real numbers, while in classification it is an integer chosen from a restricted set, which is to be meant as an identifier of the belonging class. These algorithms are useful when predicting some characteristic of data, basing on the other features of the sample: for example, a regression algorithm can be used to predict the price of an house, giving its characteristics (location, number of rooms, presence of domestic appliances); a classification algorithm can be used to predict the correct letter, given a graphical representation of it (handwritten text recognition). The major drawback of supervised learning algorithms is that they need a "supervisor", which is often a human, that has to manually label all the data in order to make them available for the system. This procedure is time consuming and, as it is done by humans, it is prone to errors, which are transferred permanently to the machine.

Unsupervised learning algorithms are used to extract meaningful structures and patterns in data. In particular, they are useful to group data into clusters, according to some similarity measure. For example, they can be used to split people into communities, according to some features like age, interactions, interests. Since they don't need labelled data, the preparation of the dataset is more comfortable. However, they cannot be used to solve specific mapping problems as supervised algorithms.

For this reason, the "brute-force" supervised learning algorithms are those used in industrial applications, while unsupervised learning is mostly used for research purposes.

Computer vision is an interdisciplinary field that deals with how computers can gain high-level understanding form digital images or videos. In particular, a computer is said to understand an image if it is able to extract meaningful information from it. In human visual system, the raw input from the retina is a set of electrical impulses that are transferred to the brain through the optical nerve; once there, they are processed from the visual cortex in order to create a representation of the scene that made us aware of what we are seeing. As it happens in our visual system, a digital camera is able to acquire an image by elaborating the light entering the lens aperture converting it to electrical impulses, which are further processed by some sensors (for example using a Bayer filter) in order to produce a virtual representation of the image. A digital image is a rectangular grid of pixels, where each of them is associated to a physical point in the real image. According to the type of image, each pixel can be represented with one (gray-scale image) or more values. For standard color images, three values are associated to each pixel, and they represent the RGB levels. Hence, the computer system "sees" an image as a matrix of numbers (figure [2.1]).

Computer vision techniques translate the matrix of numbers to meaningful information. For example, if we acquire an image of a dog, the aim is to translate the set of pixels into the label "dog". This task is often addresses as "image classification" in the literature, but there are several similar tasks (figure [2.2]):

- image classification: which type of a given object is in this image?
- object segmentation: what pixels belong to the object in the image?
- object detection: what objects are in this image and where are they?

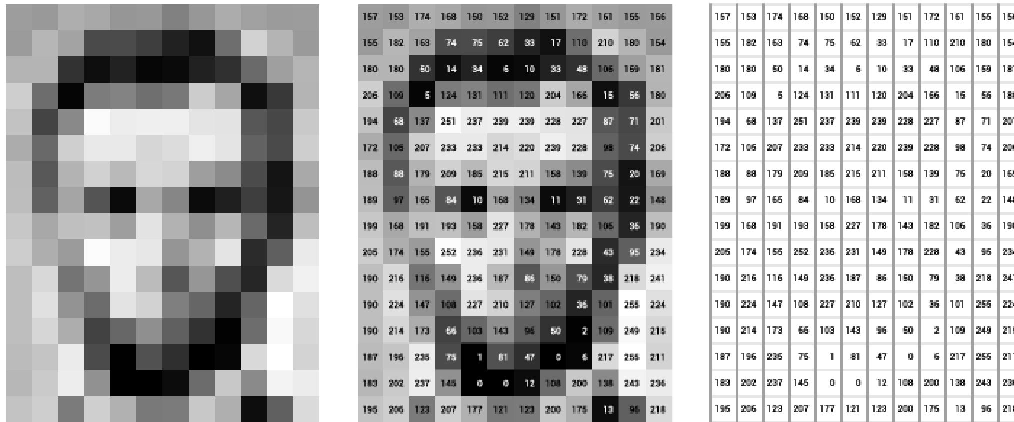
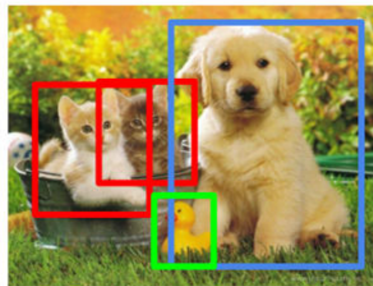


*Figure 2.1:* A computer "sees" an image as a matrix of values, each one indicates the color intensity.

**Classification**

CAT

**Object Detection**



CAT, DOG, DUCK

*Figure 2.2:* Left: image classification is assigning a label to the picture. Right: object detection is localizing and classifying objects in the image.

Before the introduction of deep learning, the computer vision algorithms were based on traditional machine learning approaches where there is a feature extraction phase followed by a classification phase. The idea was to collect as many features as possible for a class of objects and then to use this definition (bag-of-words) to look for that specific object in other images. Basiliar features can be edges, corners or gradient masks. For example, the Viola-Jones algorithm uses Haar features to detect faces in images; the SIFT and SURF algorithms extract features from an image in order to be able to identify the same object in different images. However, these algorithms need supervision, since we have to select the features to look for depending on the application, and then to use those features to train an object detector. For example, if we want to build a dog detector system, we have to predispose the algorithm to look for eyes, ears and paws in order to label the image

as containing a dog. As a result, this type of approach is not scalable since we have to manually adjust the algorithm for each desired application.

In order to be scalable, the entire system should be feature-independent, therefore being able to self adapt the features on the specific object to detect or classify. This is done with the introduction of the artificial neural networks in the machine learning field. Those models are able to extract features without being specifically trained, basing on the input image (figure [2.3]). A neural network is composed by three parts:

- the input layer: it is a set of neurons that have the role to accept the input data and forward it to the next layers;
- the hidden layers: typically 2 or 3, they are sets of neurons that transform the input data using a linear combination of them followed by non-linear activation functions that produce the input for the subsequent layer;
- the output layer: it is a set of neurons used by the network to present the output.

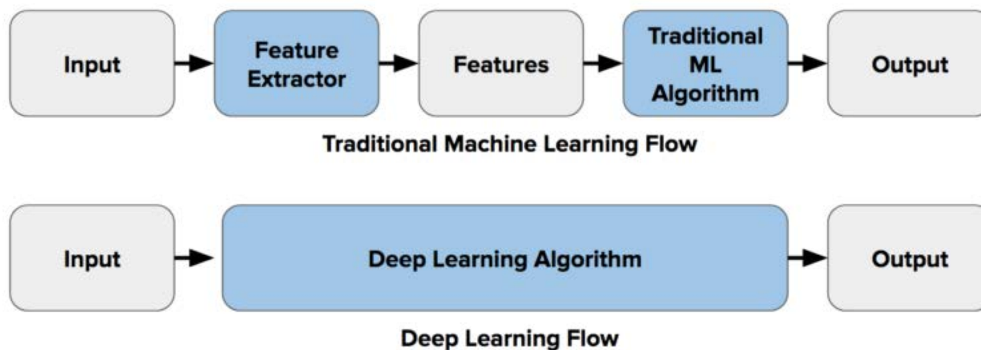



*Figure 2.3:* Above: in traditional machine learning approaches, the feature extractor has to be tuned by the user for each application. Below: deep learning techniques allow to hide this phase.

Each neuron (figure [2.4]) receives multiple inputs from the previous layer and produces a single output after a combination of functions; then, this output is forwarded to all the neurons of the subsequent layer. Such a structure is called fully-connected architecture, since all the neurons of $i^{th}$ layer are connected to all neurons of $(i-1)^{th}$ and $(i+1)^{th}$ layers. In particular, the neuron computes a weighted sum of its inputs, then it adds a bias term and finally computes a non-linear transformation to produce the output. So, during the training phase of the network, the weights and the bias values are initialized according to some random distribution, then the network is fed with each data sample from the input to the output layers (forward propagation phase). Then, the error is calculated by comparing the expected output to the actual one. Finally, the network computes the gradient of the error with the respect to each weight and bias and updates them accordingly (back-propagation

phase). In this way, the network is guided to a configuration where the error is minimized (local/global minimum): hence, when fed with new data (test phase) that belongs to the same distribution of training data, the network is able to produce the correct output.

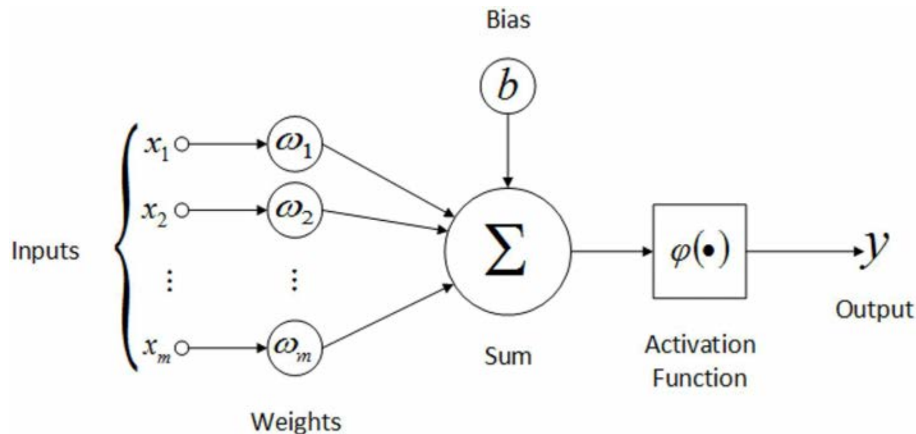



*Figure 2.4:* Representation of an artificial neuron model. The input values are weighted and summed; the results is given to an activation function to produce the output value.

### 2.1.1 Deep Neural Networks

A deep neural network is an artificial neural network with multiple hidden layers between the input and the output layers. As a result, the model has an higher number of variables, corresponding to an higher degree of freedom, that can be used to model the input data and to process them to produce the desired output. Differently from standard neural network, these networks have an arbitrary number of hidden layers, from 3 to even 150. The major drawbacks of this type of architectures are the computational power needed to train and run them, the high quantity of data needed to train them, the vanishing gradient problem and the overfitting. In general, increasing the number of hidden layers in a neural network leads to better performances, and this motivates their employment.

The two most used types of deep neural networks are:

- Recurrent Neural Networks, that are used in applications like speech recognition or time-series prediction, where there is a time relationship between the input data at different time steps;
- Convolutional Neural Networks, that are used in applications which operate with 2D data, as images.

### 2.1.2 Convolutional Neural Networks

A convolutional neural network is a deep neural network that is designed to handle 2D data like images. The neurons in the hidden layers of this type of networks are not fully connected as in standard artificial neural networks, but they are slightly

different, since they are grouped in filters. A filter is a set of neurons that are spatially arranged into rectangles (typically squares) that are shared across the full image. Indeed, if an image is reshaped into a one-dimensional vector to be fed into a standard neural network, we would lose the spatial correlation of pixels, which is extremely important in the extraction of visual features. In particular, this type of approach gives acceptable results for gray-scale images, but it fails with RGB images giving poor results. In convolutional neural networks, the input layer has the same dimensions of the input data: if the image to be processed is $300 \times 300 \times 3$, where 3 refers to the number of channels (RGB), the input layer will accept the structured 3D matrix as input. Then, each hidden layer is composed by a set of three operations:

- Convolution
- Pooling
- Activation function

In the convolution operation (figure [2.5]), the filter is moved over the image and a matrix multiplication is computed between the weights (filter) and the pixel values considered, producing one output value. All the parameters related to the filter dimensions and striding have to be set by the designer of the network. For example, if we use a $5 \times 5 \times 3$ filter over a $300 \times 300 \times 3$ image, we have to decide both the striding and padding values. The stride parameter refers to the amount of pixels to skip when moving the filter over the image; the padding parameters is used to decide the output dimension, since the convolutional operation reduces the original one: in particular, it is possible to pad the input image with zeros in order to increase its dimension in order to obtain an output that has the same input dimension. In general, the convolutional operations is designed to maintain the dimensions, since the pooling operations is meant to reduce them. The convolutional part is the only one where there are weights that learn to adapt to catch the features.

The activation function (figure [2.6]) is applied right after the convolution operation and it allows for the non linearity of the model, increasing its capabilities. There are several types of activation function and it is up to the designer to decide it: *Sigmoid*, *Softmax*, *Tanh*, *ReLU*, *Leaky ReLU*, *Maxout*, *ELU*. This function is applied to each neuron by transforming its output accordingly. In the selection of the most suitable activation function, both the range of the output and the differentiability of the function must be considered.

The pooling operation (figure [2.7]) is performed to the output of the activation function and it has the role to progressively reduce the data dimension. This operation starts from the output of the convolutional layer and selects one pixel among a predefined spatial mask. The dimensions of the mask are other parameters of the network. There are two types of pooling: average pooling, where the mean value among pixels is computed inside the mask and used as output; the max pooling, where the maximum value is selected as output.

These three operations form a hidden layer of a convolutional neural network. Stacking more hidden layers led to the progressive reduction of the input image to an activation map where the notable features are caught by the neurons in the previous

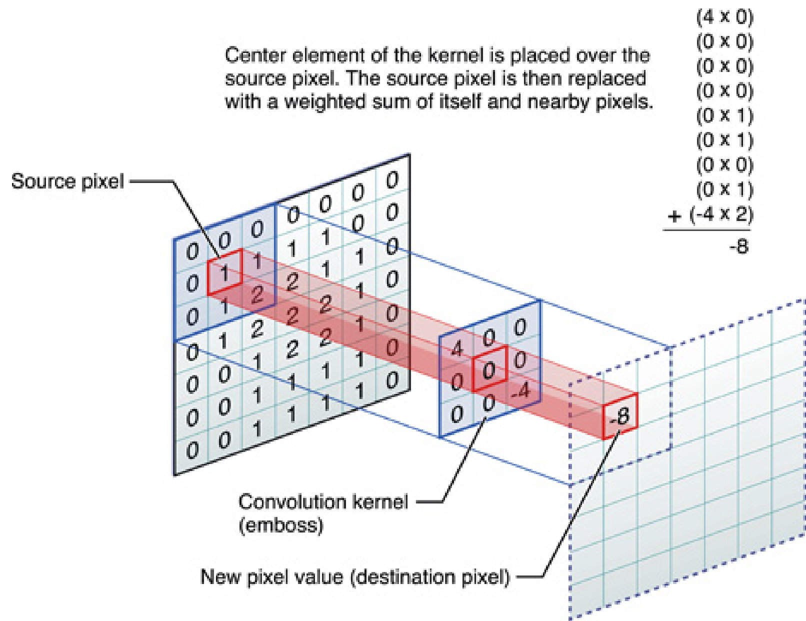



*Figure 2.5:* A representation of the convolution operation on a $7 \times 7$ image, with a $3 \times 3$ kernel, 0 padding and stride equal to 1.

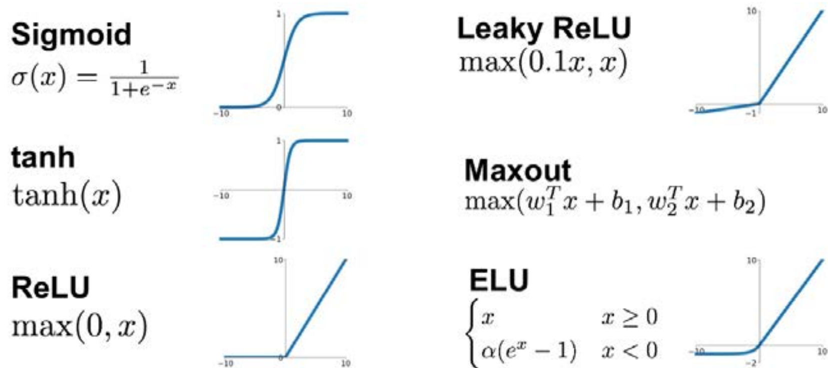



*Figure 2.6:* A list of most used activation functions.

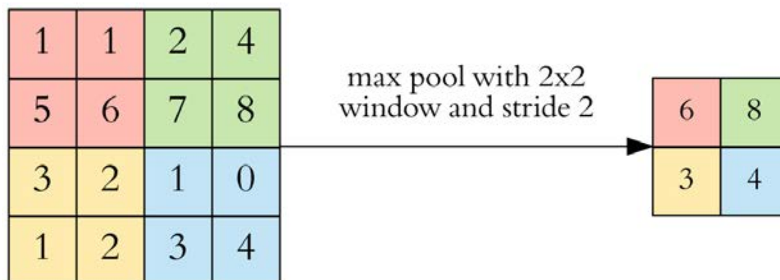



*Figure 2.7:* The pooling operation performs dimensionality reduction. In this case, the max pooling is performed.

layers. Moreover, batch normalization operation can be interleaved between layers: the output values of a layer are normalized in order to keep them bounded and to

rescale them in the same domain.

For the sake of simplicity, each hidden layer composed by these three operation is referred in the literature as “convolutional layer”. Convolutional layers are able to extract meaningful features from an image since the filter structure is designed to exploit the spatial correlation between pixels, which is a key-point in images comprehension. The features extracted from the image are then used to produce the output of the network: usually, these types of networks solve a classification problems by mapping an image into a category. The final part of the network is composed by a set of fully connected layers, with a decreasing number of neurons. Finally, the output layer has a neuron for each possible category.

Since the introduction of this type of architecture by Fukushima in 1980 with the “neocognitron” model, the general trend has been either to develop deeper models with an high number of parameters and to reuse information from previous layers in the computation of final features. However, the base structure of the convolutional layers is kept unchanged from the first architectures to nowadays complex models.

## 2.2 History of Object Recognition

Today’s object detection frameworks like YOLO, SSD and Faster R-CNN are the result of an intense research on the convolutional networks field. This research was guided by one worldwide challenge: ImageNet Large Scale Visual Recognition Competition (ILSVRC) that is an annual competition that evaluates object detection and image classification algorithms at large scale in order to allow researchers to compare their progresses. ImageNet group provides the dataset to be used both for training and testing, which consists in more than 14 million of images with annotations. Beyond object detection, there are also other challenges like object detection from video, scene classification and scene parsing. For these reasons, the most important achievements in this field can be found through the winners of each competition during years. ImageNet is not the only open source dataset that can be found online, but there are also PASCAL Visual Object Classes (PASCAL VOC) and Common Objects in Contest (COCO). Those dataset are often used to train the weights of object detection models, that will be fine-tuned on customized datasets by the users. This is possible since the convolutional layers are essentially feature extractor layers, while the fully connected layers at the end of the architecture use these features to perform the classification task. Indeed, the entire image processing can be divided into two phases:

- features extraction layers: convolutional layers that extract features, they are referred as the “backbone” of the model
- classification and regression layers: fully connected layers that are used to perform the classification and localization starting from the features, they are referred as “classification” layers

In the first years, the most challenging task was to design powerful feature extraction architecture and then to evaluate them by classifying entire images, where the detection/localization task was almost neglected. Then, the ability of these

models was also exploited to perform both classification and localization of objects in the image.

In this section, the most notable "backbone" architectures are analyzed in chronological order.

### 2.2.1 LeNet 1998

The first model designed to solve the image classification problem that used a set of convolutional layers concatenated was introduced by Yan LeCun in [Lec+95]. They introduced *LeNet-1*, *LeNet-4*, *Boosted LeNet-4* and *LeNet-5*, a family of architectures that used convolutional layers to extract features from the image and then were able to classify the correct digit with some fully-connected layers.

The first proposed architecture is showed in figure [2.8], *LeNet-1*. It takes in input a $28 \times 28$ gray-scale image, applies two blocks of convolution and average pooling layers ending up with 12 feature maps with $4 \times 4$ dimension. These are directly connected to the 10-dimensional output vector that represents the probability distribution of the input image over the 10 possible digits. This model was able to achieve 1.7% error rate on the test data.

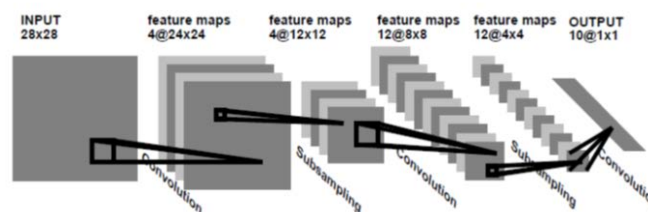



*Figure 2.8:* The architecture of the *LeNet-1* model.

The model known as *LeNet-4* is similar to the previous one, with differences in the size of the input image, which is increased to $32 \times 32$, and in the output layer, which is preceded by a fully connected layer with 120 neurons. This model was slightly more efficient, with an error rate of 1.1% on test data.

The most popular architecture when referring to LeNet family is *LeNet-5* (figure [2.9]), in which authors started from *LeNet-4* and, besides using an higher number of filters in each of the convolution layers, they added another fully connected layer before the output one. The two fully connected layers had, respectively, 120 and 84 neurons. The error rate reached with this architecture was 0.95%.

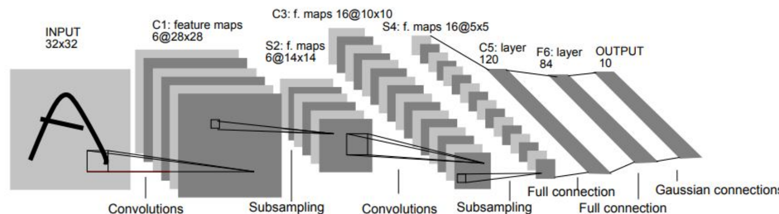



*Figure 2.9:* The architecture of the *LeNet-5* model.

The last architectures has nothing new with the respect to these previously discussed, but exploits the boosting technique to increase performances: in *Boosted*

*LeNet-4* model, three *LeNet-4* models are used in parallel to produce three results that are summed to end up with just one output. In this way, they increased the robustness of the overall model since, assuming that the probability that more than one single *LeNet-4* model makes an error on the same image is very low, the error of each model is corrected by the other two. Thank to this, the error rate of this architectures was 0.7%, even smaller than *LeNet-5* model.

The *LeNet* family was a breakthrough for the time, since all other models relied to features extraction techniques like SIFT, HOG and LPB, followed by classification models like the state-of-the-art of that time, which was SVM. Moreover, the trend suggested that adding more hidden layers helped reducing the error rate. However, there were two main problems in going deeper:

- these models used *Tanh* as activation function, which suffered the problem of vanishing gradient, hence going deeper would have resulted in no improvement
- the hardware of the time was not sufficient to train "deeper" models in a reasonable time

### 2.2.2 AlexNet 2012

After 14 years from *LeNet*, *AlexNet* [KSH12] was the winner of ILSVRC2012 contest for image classification. The authors of *AlexNet* could take advantage from a 1.2-million-images dataset provided by ImageNet divided upon 1000 classes and the GPUs to speed-up computations. Hence, they were able to reach a top-5 error rate of 15.3% on the test data, outperforming the first running-up model, which reached only 26.2%. The full 7-layers architecture is showed in figure [2.10]. In *AlexNet*, each image is re-scaled to a $256 \times 256 \times 3$ matrix and then, from each pixel, they subtract the average of intensity over the training set as a sort of normalization. Moreover, they used some techniques of data-augmentation to make the model more robust and stable. In particular, from each image they extract 10 $224 \times 224 \times 3$ patches (from each corner and the center, and then the horizontally-flipped versions). Besides this, they also altered the RGB intensities by adding to each pixel a vector that is a linear combination of eigenvalues and eigenvectors of the $3 \times 3$ covariance matrix of RGB pixel values, weighted by a Gaussian random variable. These techniques helped to reduce overfitting, as well as introducing a reduction of 1% in error rate. Inside the model, the authors used some modifications with the respect to *LeNet* models. First of all, they used *ReLU* as activation function instead of *Tanh*: this helped solving the problem of vanishing gradient when the model is deep, as in this case. Although *ReLU* activation function doesn't require normalized input, authors applied a sort of response normalization after the $1^{st}$ and $2^{nd}$ convolutional layers that implements a kind of lateral inhibition between neurons that are forced to learn more robust features. Based on the same principle, they used the recently-proposed dropout technique with rate 0.5 during training, halving the weights of neurons at test-time to cope with this. Finally, instead of using average pooling layers, they used overlapping max pooling layers. All these tricks applied together allowed to reduce the error rate of 2%, aside from reducing overfitting.

The model was trained for 90 cycles on 2 GPUs as to maximize the average across training images of the log-probability of the correct label under the predicted

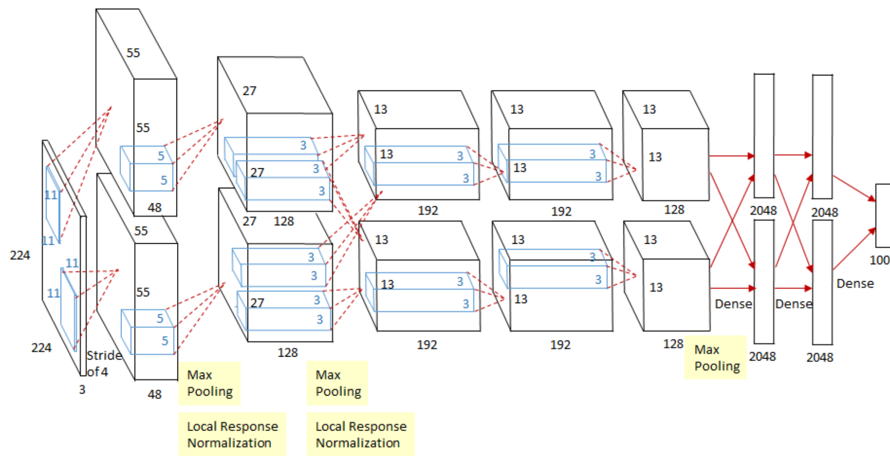



*Figure 2.10:* The architecture of the *AlexNet* model.

distribution. The training technique used is SGD with *batch_size* = 128, *momentum* = 0.9 and *weight_decay* = 0.0005. The learning rate was initialized equal to 0.01 for all layers and then reduced by a factor of 10 when error rate has stopped improving. The entire training took 6 days to be completed.

Evidences provided by the authors are:

- the depth of the network is a crucial aspect for the accuracy
- removing any of the hidden layers results in a less accurate model

### 2.2.3 OverFeat

The main point of Sermanet *et al.* [Ser+13] was to show how to train end-to-end a model to classify, locate and detect objects in the image. *OverFeat* was the winner of the ILSVRC2013 localization task, that required a model able to classify the objects in the images, besides localizing them with a rectangular bounding box. The authors solved the problem by designing a network as shown in figure [2.11].

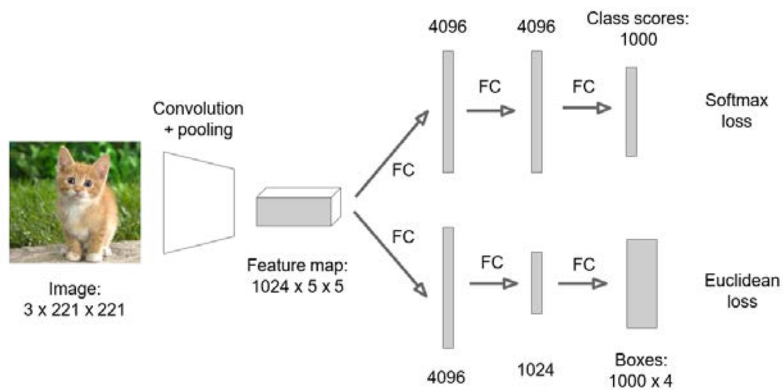



*Figure 2.11:* The architecture of the *OverFeat* model.

The features extraction layers are taken from *AlexNet*, with some modifications. First of all, here authors didn't use any contrast normalization and pooling regions

are non overlapping. Moreover, they used larger kernels in $1^{st}$ and $2^{nd}$ layers but with smaller stride to increase the accuracy. The feature layers are trained according to the classification task, as already done by *AlexNet*. In order to increase the robustness, they trained the model over 6 different scaled versions of the original image. Then, they removed classifier layers and substitute them with regression layers (2 fully connected layers with 4096 and 1024 neurons) to perform the localization task. Their weights are trained to minimize the $\ell 2$ norm between the ground-truth bounding box coordinates and the predicted.

At the detection time (as shown in figure [2.12]), the model firstly performs classification at each location. Then, it predicts object bounding boxes on all classified regions generated by the classifier. Finally, it merges bounding boxes with sufficient overlap from localization and sufficient confidence of being the same object from the classifier.

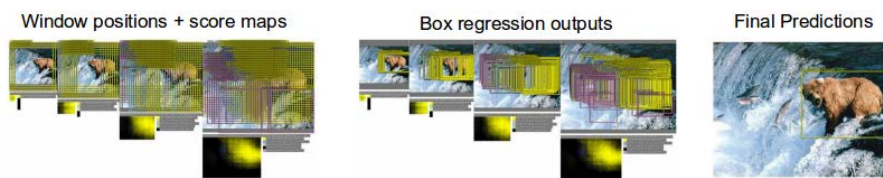



*Figure 2.12:* The three phases of the *OverFeat* detection model.

This pioneer model was able to integrate three different tasks related to computer vision (classification, localization and detection) in a single model. The model was able to outperform the other approaches in the localization task, reaching the top positions also in the other two tasks.

### 2.2.4 ZFNet

The winner of ILSVRC2013 classification task was the model proposed by Zeiler and Fergus in [ZF13]. They thought that without a clear understanding of how CNNs encoded the images and how features look like in the hidden layers, it would be impossible to design meaningful models. They started from the architecture of *AlexNet*, but they inserted some deconvolutional networks in the features extraction pipeline in order to reproject them back to the pixel input space. As show in figure [2.13], the three steps of a convolutional network are performed in the inverse way: max-pooling is a non-invertible operation, and so it is replaced by keeping track of the max locations in the previous layer and placing there the maximum value, setting all the others to zero; rectification is no longer needed, since only positive values are returned from the previous operation; the inverse of convolution is performed using transposed versions of those filters applied to rectified maps. The data-augmentation strategy, together with the training pipeline, is the same used for *AlexNet*, except for the fact that here only one GPU is used. During training, they observed the learned features, making changes to cope with issues.

Authors found that layer 2 is responsible for learning corners, edges and color conjunctions; layer 3 captured similarities in the texture, while layer 4 and layer 5 are more class-specific. During training, the first hidden layers converged after a few

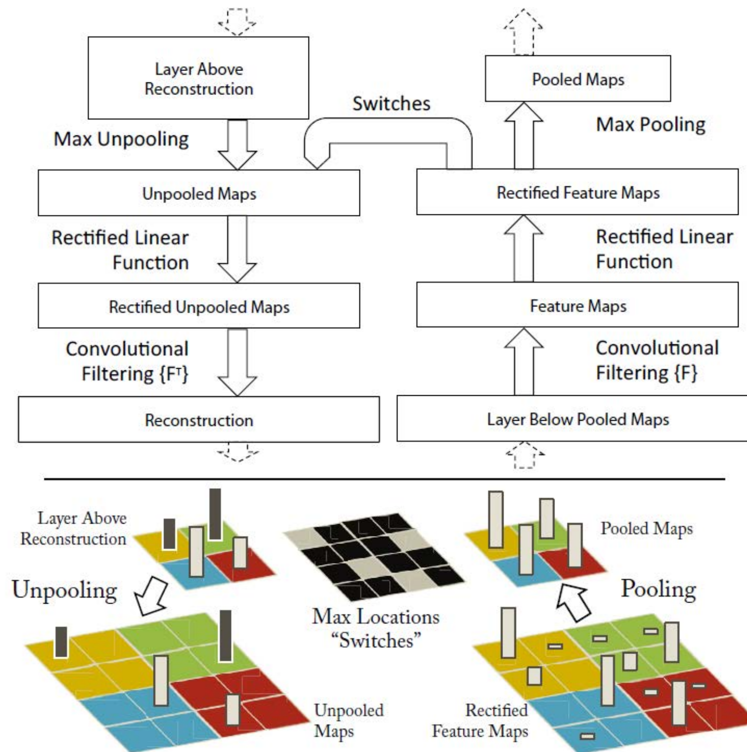



*Figure 2.13:* The deconvolutional layer of the *ZFNet* architecture.

epochs, whereas it took longer to train more deep layers. Moreover, they showed that although the first layers are very sensitive to small intra-class variations, these differences are absorbed in successive layers. The aliasing effects caused by the large strides of kernels in the first and second layers are solved reducing both kernel size and kernel stride. The authors also provided some experiments to understand whether the model is localizing the object basing on the background boundaries rather than on its internal structure. They occluded different portions of the input images, showing that the model failed to correctly classify the objects when occluded.

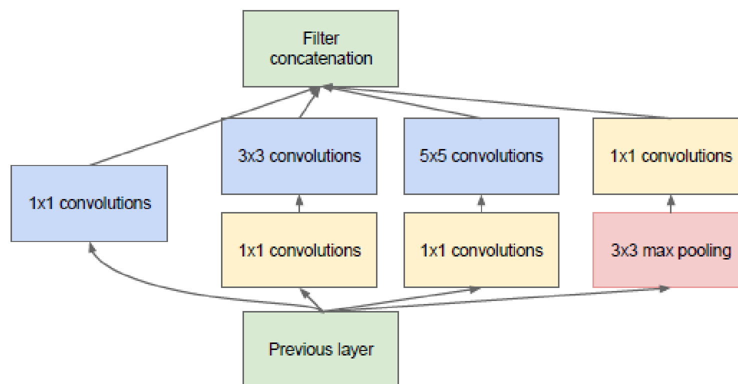
With these modifications to *AlexNet* and combining different architectures varying the number of feature maps, authors were able to reach 14.8% of test error. Authors also provided some results when using their model (trained on ImageNet 2012 dataset) with different datasets, like Caltech-101, Caltech-256 and PASCAL VOC 2012. They both tried to re-train the model from scratch or to use the pre-trained weights of the feature extraction layers just modifying the classification neurons according to the dataset specifics. They discovered that while training from scratch ends in very poor results, using the feature extraction layers of their model they were able to beat the state-of-the-art object detection algorithms on those datasets. This confirmed the feature extraction power of their architecture.

### 2.2.5 GoogleNet and Inception v1

The previous work done in *AlexNet* and *ZFNet* papers has shown that going deeper with hidden layers is the right direction to have more powerful object detection

models. On the other hand, increasing the number of layers also means exponentially increasing the number of parameters to train. Hence, if we don't have a system capable to manage a wide quantity of variables in reasonable time, this results in models impossible to train and, most importantly, to use. Moreover, since many real-world applications are designed to be run over simple architectures like smartphones with a reduce computing capability, those models have to be relatively thin.

In their paper [Sze+15], authors addressed this problem, designing an architecture that, more than going deeper with layers in the conventional way, introduces a new module named "inception" that goes wider. The "inception" module takes in input the feature maps of the previous layer and then applies convolution using different kernel sizes ($1 \times 1$, $3 \times 3$ and $5 \times 5$). In addition, as introduced by Lin *et al.* in [LCY13], "inception" layer uses $1 \times 1$ convolution as a dimensionality reduction technique. Before being passed to the next layer, the features extracted with different kernel sizes are concatenated. The architecture can be seen in figure [2.14].




(b) Inception module with dimensionality reduction

*Figure 2.14:* The Inception module of the *GoogleNet* architecture.

Together with a very deep network comes the problem of vanishing gradient. Hence, authors suggested a technique to propagate back the errors among all layers in an effective manner, stacking some auxiliary classification layers in parallel in the middle of the network. In this way, during training, not just the loss of the output is computed, but also the loss at intermediate levels. The total loss is then computed averaging these quantities.

At the end of the model, instead of using fully-connected layers that requires a lot of parameters, authors exploited another technique proposed in [LCY13] which consists of using global averaging pooling. In this way, only one value (the maximum) is extracted from each feature map and weights are no longer needed.

The full model (figure [2.15]) consists of 9 "inception" modules stacked together, with 2 auxiliary classifiers in the middle. All in all, the network has 27 layers, which is a great step forward from 7-layers *AlexNet* of 2012. The network is trained using asynchronous SGD with 0.9 momentum and fixed learning rate that decreases by 4% every 8 epochs. From each image, a total of 144 crops are extracted for data-augmentation. Finally, boosting technique is used averaging the results of 7 independently trained *GoogLeNet* models. The authors' final submission to the

challenge obtains a top-5 error rate of 6.67%, outperforming every other competitor and winning ILSVRC2014 classification task.

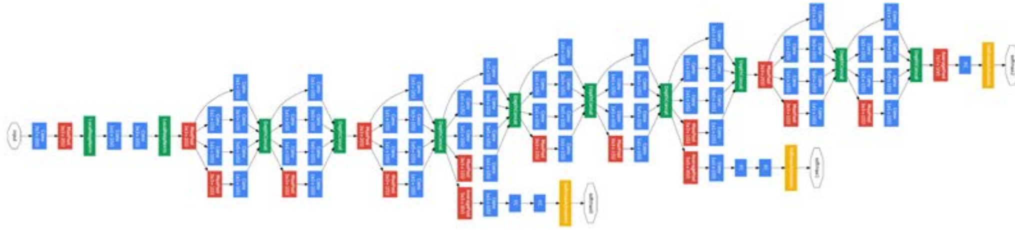


*Figure 2.15:* The full architecture of the *GoogleNet* model.

Authors also competed in the ILSVRC2014 detection task, using the "inception" architecture to gain precision. They do not use bounding box regression. However, they were able to reach 38.02% as *mAP* ranking second, just 2 points less than *Deep Insight* model (40.2%) that used bounding box regression.

### 2.2.6 VGGNet

The winner of ILSVRC2014 localization task, that was able to beat *GoogLeNet*, was *VGGNet* by Simonyan and Zisserman [SZ14]. The key characteristic of their model is the small size of the convolutional filters. They showed that the feature maps obtained using a cascade of two $3 \times 3$ filters is able to capture the same spatial resolution of a unique $5 \times 5$ filter, but significantly reducing the number of parameters. They also showed that the local response normalization used by Krizhevskty *et al.* in [KSH12] after the $1^{st}$ and $2^{nd}$ convolutional layers can be omitted without any loss of precision.

The authors compared different architectures (figure [2.16]) varying the number of hidden layers from 11 up to 19, discovering that the model with 19 layers is slightly worse than the one with 16 layers. This fixed the maximum depth in their experiments. They also confirmed that a deep network with small filters outperforms a smaller network with larger filters.

In addition, authors exploited multi-scale training, where every image is scaled with smaller size in a pre-defined range and then multiple crops with fixed size of $224 \times 224$ are taken to feed the models. This helped reducing the error rate of about 1%. Moreover, also performing multi-scale testing led to a slightly improvement. Finally, during testing, all the classification fully-connected layers are replaced by convolutional layers, where $1 \times 1$ convolutions are used to reduce dimensionality.

When compared to *GoogLeNet*, *VGGNet* achieved comparable results (around 6.8%) using boosting with just 2 models instead of 7. Without exploiting the boosting technique, *VGGNet* achieved 7% top-5 error, which is less than 7.9% achieved by *GoogLeNet*.

Nevertheless, *VGGNet* was the winner of the ILSVRC2014 localization task with 25.3% top-5 test error. The classification layers are replaced by regression layers to predict the coordinates of the bounding boxes, in particular their center and the dimensions. Instead of using cross-entropy, the Euclidean distance between predicted and ground-truth boxes is used as loss metric. The authors exploited both single-class regression (SCR) where only one 4-D vector is predicted among

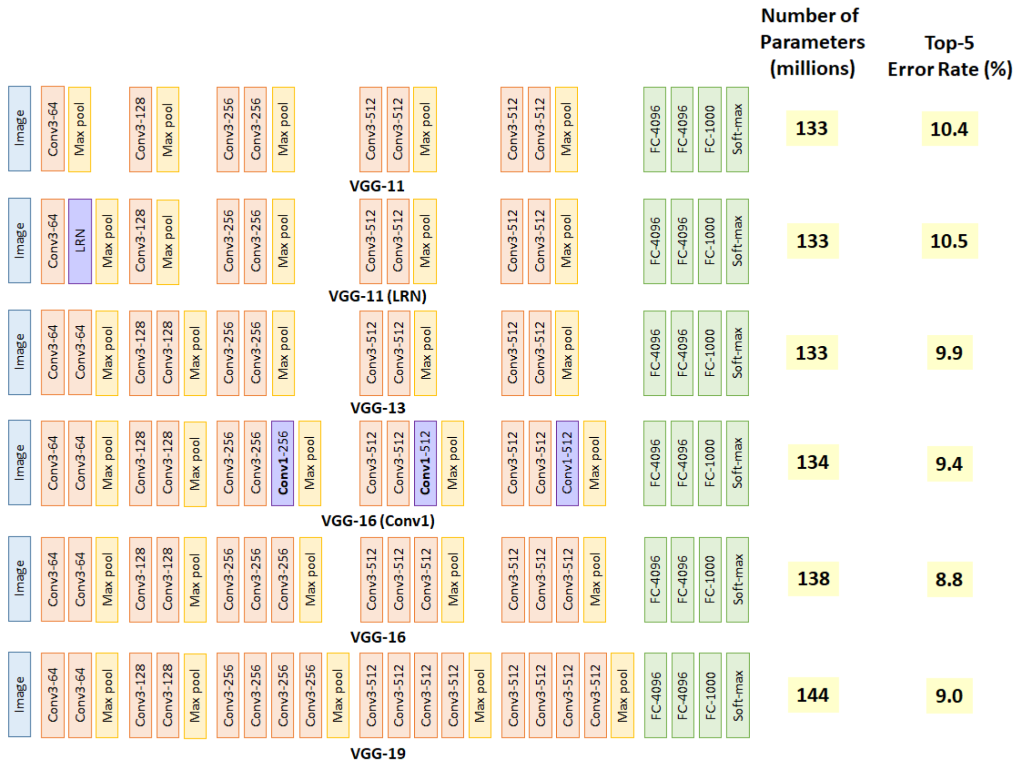



*Figure 2.16:* The different architectures tested by authors of *VGGNet*.

all classes, or per-class regression (PCR) where 1000 4-D vectors are produced, one for each class. They also tried both fine-tuning all layers and fine-tuning only the fully-connected layers. They found that PCR outperformed SCR and fine-tuning all layers led to better results instead of just tuning the fully-connected ones.

### 2.2.7 Inception v2 and v3

In 2015, the same authors of *GoogLeNet* proposed some design modifications specifically to reduce the computation complexity, more than increasing the performances. This was because although *VGGNet* was a very simple architecture, it had an extremely high number of parameters making infeasible its use in real-time applications, for example with mobile devices. Moreover, the "inception" module introduced in *GoogLeNet* is not easily up-scalable without introducing delays. These modifications introduced a new generation of "inception" architectures that are addressed as *Inception v2* and *Inception v3*.

The first modification, introduced in [IS15], is to use Batch Normalization to normalize value distributions before going on to the next layer of the model. When the distributions are almost fixed to the same range, the gradients are no longer dependent on the precise value, but better on its variations. Hence, an higher learning rate can be used and dropout can be reduced, with a significant reduction in training time. Applying Batch Normalization to *GoogLeNet* led to a new model called *Inception v2* with a top-5 error rate of 4.82%.

Another issues was that using filters with a large spatial resolution like $5 \times 5$ and

$7 \times 7$ led to bottlenecks in the pipeline. Hence, in [Sze+16], authors introduced some techniques to tackle this issues, that results in 3 different new "inception" modules, namely A, B and C. In module A, they used a cascade of multiple $3 \times 3$ filters to produce the same result of higher dimensional filters but saving some parameters: for example, a $5 \times 5$ filter has 25 parameters, while 2 $3 \times 3$ filters have 18 parameters. In module B, they divided the spatial factorization into asymmetric convolutions (vertical and horizontal) using filters with dimensions $1 \times n$ and $n \times 1$ and then merging the results: this way, a $3 \times 3$ filter with 9 parameters can be replaced by a $1 \times 3$ and a $3 \times 1$ filters with a total of 6 parameters. Finally, in module C, authors proposed an architecture to promote high dimensional representation where convolutional operations are executed in parallel and then concatenated at the end.

Moreover, while in *GoogLeNet* three auxiliary classifiers are used to prevent gradient vanishing, in *Inception v3* only one classifiers is used for regularization. Another regularization technique is to use label smoothing, keeping probability values well distributed among all classes, hence reducing the cross-entropy. Finally, an efficient grid size reduction substitutes simple max pooling between different "inception" modules. The overall architecture of *Inception v3* can be seen in figure [2.17].

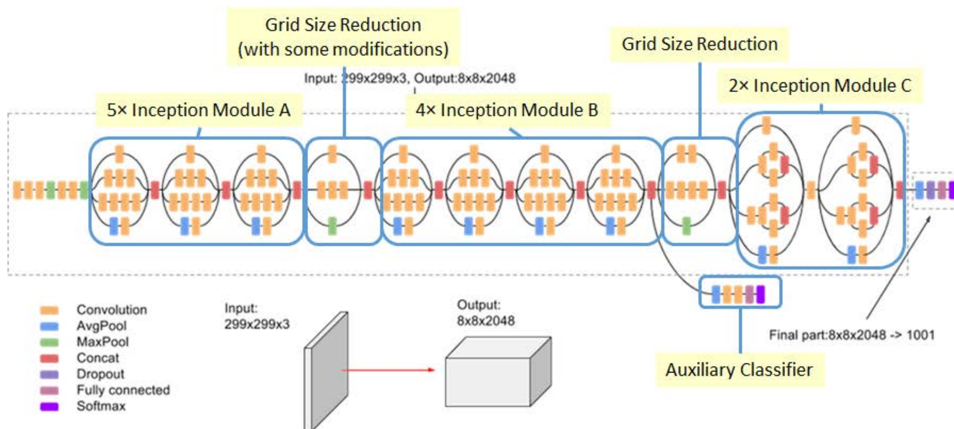



*Figure 2.17:* The full architecture of the *Inception v3* model.

The top-5 error rate achieved by *Inception v3* is 4.2%, which outperformed both *Inception v2* and *PReLU-Net*. Using boosting, *Inception v3* established a top-1 error rate of 17.2% and a top-5 error rate of 3.58%, reaching $2^{nd}$ position in ILSVRC2015 challenge.

### 2.2.8 ResNet

Although one of the most trivial technique for improving the learning accuracy of the model is to stack layers to create very deep networks, this increases the number of weights to be trained, resulting in a very computationally demanding model. Authors of *Inception* networks have exploited the technique to go wider instead of deeper with layers reaching quite good results. Their choice was motivated by the vanishing gradient problem that occurs when the model is very deep. Batch-Normalization techniques have been used to limit the range of weights values between

different layers in order to avoid saturation. However, experiments showed that the train and test errors of deeper models are unexpectedly higher. This is mainly caused by the degradation problem, a phenomena in which a cascade of multiple layers is not able to approximate an identity mapping. To solve this, He *et al.* introduced *ResNet* in [He+16a], an architecture in which shortcut connections with no weights between layers are added to simplify the representation of features. Suppose that x is the input of the first of a series of layers and $H(x)$ is the underlying mapping to be learned. Rather than train the weights to approximate $H(x)$, they are trained to learn the residual function $F(x)$, defined as $H(x) - x$. In this way, the input x can be added at the end of the layers, after some operations to match the dimensions, to obtain the desired feature map. In the case an identity mapping has to be learned, during training the weights are driven towards zero, and the input just skip the layers and replicates at the end. This new module is illustrated in figure [2.18].

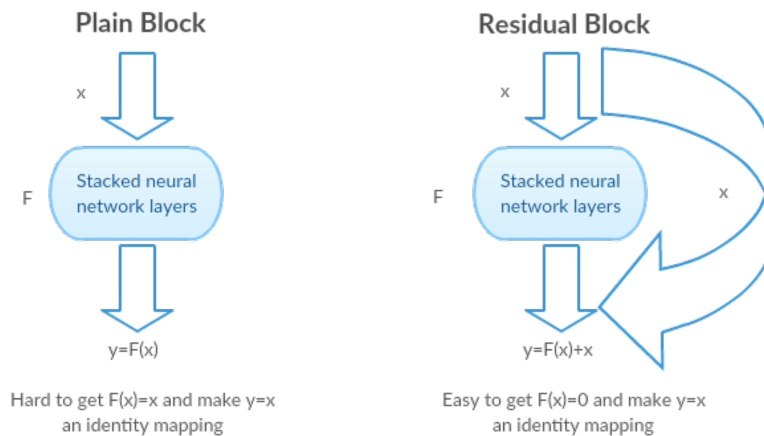



*Figure 2.18:* The residual block introduced in the *ResNet* architecture.

This technique allows for deeper architectures to be trained without suffering the vanishing gradient problem. Authors started from the *VGGNet* network with 19 layers and built a model with 34 hidden layers. They trained two models: one without shortcut connections (plain) and one with residual modules as described above. Those models are shown in figure [2.19]. They found that using residual networks they are able to achieve a lower training error with a deeper model, overtaking the vanishing gradient issue. With an architecture that could exploit the advantages of going deeper without losing the gradients, the authors tested several models adding more and more layers, using $1 \times 1$ convolutions, as introduced in [Sze+15], to reduce the number of parameters. They were able to reach 19.38% top-1 and 4.49% top-5 error rate using a single model approach with a 152-layer network called *ResNet-152*. An ensemble of 6 *ResNet* models achieved 3.57% top-5 error rate, winning ILSVRC2015 contest. They also built a 1202-layer network and they trained it with no optimization difficulty. However, the test error was 7.93%, higher than 6.43% achieved with a 100-layer network. This is probably due to overfitting, since authors didn't register vanishing gradient problems.

*ResNet* was also the winner of both ILSVRC2015 detection and localization tasks. The authors used their *ResNet-50* and *ResNet-101* models as backbone for

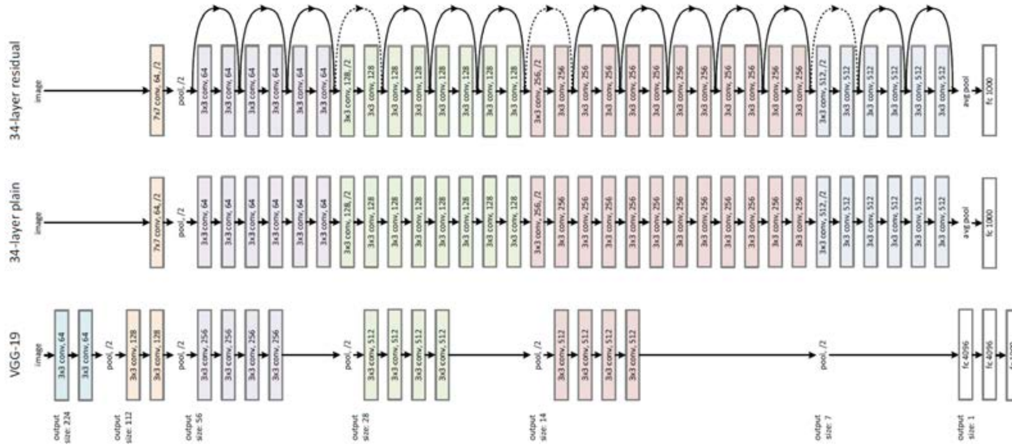



*Figure 2.19:* The two architectures (one with residual connections) trained by authors, compared to *VGGNet-19*.

the *Fast R-CNN* framework. They found that the 101-layer model increased the mAP by over 3% on PASCAL VOC and over 6% on MS COCO with the respect to *VGGNet-16* due to a better features learning strategy. For the localization task, they used RPN framework designed in a per-class form with two modules: one for classification and one for regression. An ensemble of networks they achieved 9.0% top-5 error on the test set, which significantly improved the *GoogLeNet* results of the previous year (26.7%).

In [He+16b] authors exploited the advantages of modifying the position of the modules in the network pipeline. They found that moving the ReLU and BN layers before activation can led to better accuracy. In particular, they were able to take advantages of the depth of the network by training a 1001-layers model and making it outperform the previous *ResNet* architectures and reaching 4.62% error rate on CIFAR-10 and 22.71% on CIFAR-100.

### 2.2.9 Inception v4 and Inception ResNet v1 and v2

In [SIV16], Szegedy *et al.* applied the residual module introduced in [He+16a] to the *Inception* architectures developed so far, building *Inception ResNet-v1* and *Inception ResNet-v2*. They also tried to make the *Inception v3* model deeper and wider, introducing *Inception v4* network, which structure can be seen in figure [2.20]. In this model, all techniques used in the previous *Inception* architectures are used, but the number of layers is increased to about 100.

In the *Inception ResNet* versions, a shortcut connection is added to by-pass each inception layer. If compared to *Inception v3*, *Inception ResNet-v1* has the same computational cost but converged earlier. The same happens to *Inception ResNet-v2* if compared to *Inception v4*. These results both proved the benefit to have a deeper model and also showed that the residual module technique can generalize to more complex network structures improving their performances.

Wrapping up, the three new introduced architectures were compared to *Inception v3*, showing that they all provide better results both with single and multiple crop strategies. In particular, *Inception ResNet-v1* reached 18.8% top-1 error rate and

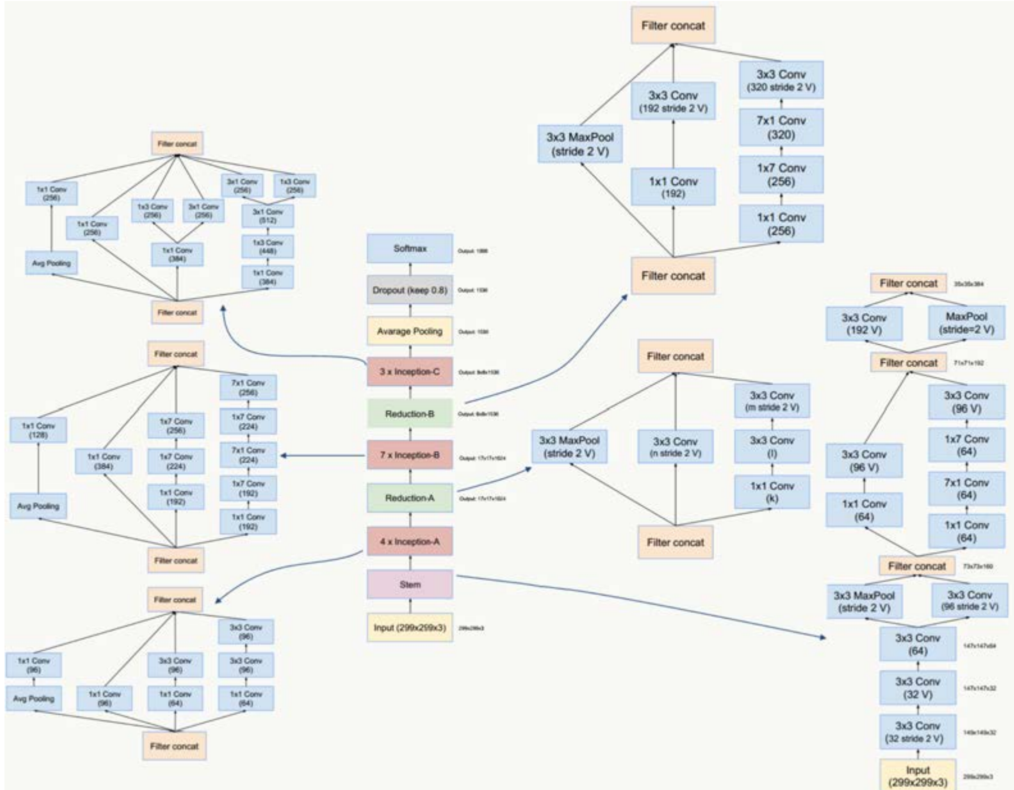



*Figure 2.20:* The architecture of the *Inception v4* model with all its modules.

4.3% top-5 error rate; *Inception v4* and *Inception ResNet-v2* are slightly better, with 17.8% top-1 and 3.8% top-5 error rate. Ensembling 4 Resnet-type models allowed to reach 16.5% top-1 and 3.1% top-1 error rate.

There is also another version of inception architectures that was introduced by Chollet in [Cho17] and it is called *Xception*. In the previous inception architectures, the authors assumed that the cross-channel and spatial correlations are sufficiently decoupled and so they can be calculated separately by performing depth-wise first and then point-wise convolution. Here, they assumed that those correlations are perfectly separable, hence the *Xception* model has 36 convolutional layers structured into 14 modules with linear residual connections among them. Inside each module, $1 \times 1$ convolution is performed one time and then shared by all the filters at each kernel size. This model reached the state-of-the-art performances of *Inception v3*.

## 2.2.10 ResNext

The *ResNeXt* architecture by Xie *et al.* [Xie+17] aggregates the concepts of *Inception* modules and *ResNet* to build a new features extractor network that was able to reach the second position in the ILSVRC2016 classification challenge with 17.7% and 3.7% top-1 and top-5 errors respectively with a single model structure, outperforming its predecessors. The key idea of *ResNeXt* was to aggregate a set of transformations with the same topology into a module and then to stack multiple modules. They took the same split-transform-merge strategy of *Inception* networks,

where the input is splitted into lower dimension with $1 \times 1$ convolution, then the filters with different sizes computes the features maps in parallel and finally those are aggregated before passing to the next module. Inside each module, the cardinality refers to the number of paths and each of them has the same topology. So, at the end, the *ResNeXt* architecture has several residual blocks where, inside each block, a set of transformations is computed in parallel and aggregated at the end before adding the input.

In their experiments on ImageNet, they replaced every residual module in *ResNet-50* and *ResNet-101* with *ResNeXt* modules. They found that the new module increased the performances of *ResNet* architecture both for 50 and 101 layers models. Moreover, they discovered that increasing the module's cardinality instead of the depth led to a more accurate network.

### 2.2.11 DenseNet

In [Hua+17], Huang *et al.* published a new backbone architecture based on the concept of residual connections introduced in [He+16a] where short paths are created between layers. However, they tried to take advantage from the connections by allowing every layer to be directly connected to every succeeding layer. In this way, each layer is discovering some knowledge about the features and, when passed to the next layer, this knowledge is not fully transformed as in previous architectures but it is kept unchanged by concatenating the feature maps of all preceding layers. This compact structure was addressed as *DenseNet*. Although this concatenation of feature maps is intuitively increasing exponentially the network complexity, authors discovered that, since previous layers feature maps are available at each layer, the number of filters could be significantly decreased, resulting in fewer parameters. Moreover, since previous works that randomly dropped some layers in very deep *ResNet* architectures showed that the features learned are somehow redundant, using fewer feature maps combined with their availability at all levels resulted in significant improvements in accuracy.

In order to keep the number of feature maps low, the full network is divided into "dense" blocks. Inside each block, batch normalization, $ReLU$ and $3 \times 3$ convolution are used to produce new feature maps with the same size, so they can be easily concatenated. At the end of each block, a $1 \times 1$ convolution is performed to reduce the number of feature maps to a fixed value, $k$, that is an hyper-parameter of the model and then $2 \times 2$ average pooling with stride 2 is performed to decrease the input size. These two operations are addressed in the paper as bottleneck layers. At the end of the last dense block, global average pooling is performed and then a *Softmax* layer computes the output vector. Authors also investigated the possibility to further decrease the number of feature maps between dense blocks by rescaling them by a factor $\theta$. The full architecture can be seen in figure [2.21].

The architecture with $\theta$ compression and bottleneck layers is the most efficient reaching the same accuracy of *ResNet* using 1/3 of parameters. The best result of the *DenseNet* architecture is on ImageNet dataset where a 264-layer model with $k = 48$ reached 20.27% and 5.17% top-1 and top-5 errors.

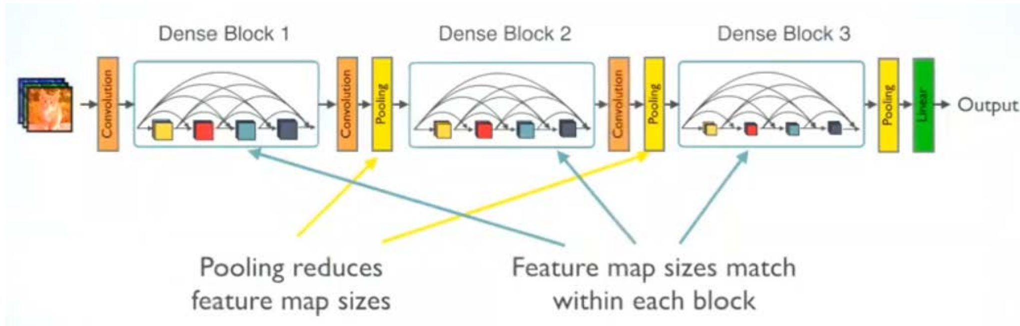



*Figure 2.21:* The architecture of the *DenseNet* model.

### 2.2.12 MobileNet v1 and v2

*MobileNet* architecture was introduced by Google researchers Howard *et al.* in [How+17] as a model that exploits the depth-wise separable convolution to build light weight deep neural networks with around 30 layers. In this way, the model can be run by simple devices with a limited capacity and computational power like smartphones and, in general, portable devices. As in models like *Inception v4* [SIV16], the convolutional operation is splitted into a depth-wise stage and a point-wise stage. This allows to reduce the number of parameters (and multiplications) by a factor of 9 when using, for example, $3 \times 3$ kernels.

In addition, authors proposed some modifications to the original *MobileNet* architecture to further reduce the number of parameters and multiplications. They introduced two hyper-parameters: $\alpha$ that is the width multiplier and $\rho$ that is the resolution multiplier. The $\alpha$ is used to shrink the feature maps uniformly at each layer and helps reducing the complexity by a factor of $\alpha^2$. For example, using $\alpha = 0.25$ allows to reduce the number of parameters of the network by 8 times but sacrificing 20% of accuracy. The $\rho$ is used to reduce the input dimension and has a milder effect if compared to $\alpha$, allowing for a reduction of factor $\rho^2$ in the number of multiplications but keeping the number of parameters fixed.

Using it as backbone for object detection frameworks led to good results when some points in accuracy are sacrificed but the number of parameters is highly reduced by a factor of 5 in *SSD* to a factor of 23 in *Faster R-CNN*, much better than *Inception v2* and *VGG-16*.

In [San+18], the same group proposed some modifications to the *MobileNet* architecture discussed above and released the model under the name *MobileNet v2*, with two main innovations. The first is the addition of the $1 \times 1$ convolution called expansion layers before the depth-wise convolution in each block. In this way, the feature maps are projected into a tensor with an higher number of channels before applying separated convolution as in *MobileNet v1*. The second modification is the addition of residual connections as in *ResNet* that allows for better back propagation of the gradient during training.

The main motivation of these choices is that keeping small tensors between layers allows for lighter networks. On the other hand, if the feature maps is too small, it is not capable of learning correlations in the image to classify the objects. The compromise introduced in *MobileNet v2* is to keep small tensors and then to expand them temporarily inside the blocks using $1 \times 1$ expansion layers, reducing them

after the depth-wise convolution again using $1 \times 1$ piece-wise convolution. It is like unzipping the tensor for the computations and then zipping it again before propagating its values through the network. This allows to build architectures even more efficient than *MobileNet v1* both in terms of accuracy and model complexity.

## 2.3 Object Detection Frameworks

In the previous section, the most notable backbone architectures in the history of object detection have been analysed. These models can be used as powerful feature extractor networks to be used for detection and classification tasks. In the literature, there are three main object detection frameworks that provide a complete end-to-end processing pipeline to perform classification and localization of objects in an image, and they are: Region based Convolutional Neural Networks (R-CNN), You Only Look Once (YOLO) and Single-Shot multibox Detector (SSD). In this section, each of these frameworks is analysed and strong and weak aspects are listed for each.

### 2.3.1 R-CNN

In [Gir+13], Girshick *et al.* combined *AlexNet* and the *Selective Search* algorithm [Uij+13] to build the first powerful object detection framework that outperformed any other approach. The algorithm was based on the observation that, even thought *AlexNet* was an accurate classification model, real world images are generally not focused on a single object, but they are better composed of many objects at different locations and scales. An object detection algorithm should be able to output the class of any object together with the bounding box coordinates of the patch containing it. Hence, applying *AlexNet* to an image containing, for example, a cat in the bottom-left corner, will probably be the same as applying it to an image where the cat is at the center, occupying the entire image. In the first case, we would like the model to draw a bounding box around the cat in the bottom-left corner, while in the second case it doesn't really matter.

A brute-force approach used at the time was to extract patches at different locations and scales from the input image using a sliding window that passes the whole image, and then to feed every patch to a CNN. In this way, there is an high probability that there exists at least one patch that perfectly contains the object to detect, and that patch will be classified by the CNN. It is clear that this approach had some drawbacks:

- the number of patches exponentially grows with number of scales and image size, making the approach computationally expensive
- even with an accurate model, the higher the number of patches, the higher the number of false positives

In *R-CNN* framework, the authors used an algorithm to extract meaningful patches from the input image in order to avoid the computationally expensive sliding window approach. In this way, instead of feeding the CNN with hundreds of thousands patches where the most of them are not significant, they only pass a

small fraction, addressed as candidates to contain objects. This algorithm is called *Selective Search*. In figure [2.22] the result of the algorithm is shown.

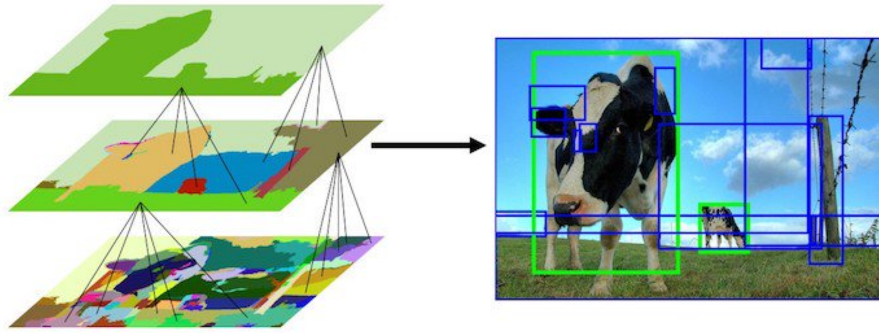


*Figure 2.22:* The *Selective Search* algorithm progressively aggregates pixels in the image that could represent objects.

The *Selective Search* [Uij+13] algorithm is a regional proposal algorithm that takes in input an image and extracts a set of patches that are likely to contain objects. It starts from making an over-segmentation of the image at very low level using the Felzenszwalb and Huttenlocher segmentation method [FH04], based on colors. Then, the algorithm extracts bounding box containing homogeneous patches and adds them to the proposal regions list; finally, it hierarchically aggregates regions. These two steps are repeated iteratively until the entire image falls on the same area. The similarity measure used to aggregate regions is a linear combination of four characteristics as color, texture, size and shape: for colors, a 25-bins histogram is calculated for each of RGB channel and the similarity is calculated using histograms intersection; for texture, a 240-dimensional feature vector is computed by extracting Gaussian derivative at 8 orientations and using 3 10-bins histograms; for size, smaller regions are encouraged to merge together; for shape, a bounding box is considered around the two regions and they are merged if one contains the other. The pseudo-code of the algorithm is showed in figure [2.23].

With the *Selective Search* algorithm, the number of proposed regions is significantly decreased, allowing to speed-up the overall process of detection and classification. The *R-CNN* pipeline is shown in figure [2.24]. Proposed regions are extracted, resized to $227 \times 227$ and fed into the CNN model to extract a 4096-dimensional feature vector. Finally, class-specific linear SVMs are used to classify the patch. This approach obtained above 50 mAP on the PASCAL VOC 2010 dataset. Moreover, it outperformed the winner of ILSVRC2013 *OverFeat* (24.3%) by achieving 31.4% in mAP. On PASCAL VOC 2007, authors used *VGGNet-16* as feature extractor, achieving 66% in mAP.

To improve localization performances, authors used a bounding box regression stage in which a new set of coordinates are predicted for each detected object by adjusting the RoIs towards objects. During the training, each ground-truth bounding box is identified by a set of 4 values $[G_x, G_y, G_w, G_h]$ where the first two are the center, while the others are the dimensions. The procedure consists in mapping the predicted set P towards the ground-truth G, learning the linear transformation functions by optimizing the regularized least squares objective. Predicted boxes that are too far from any ground-truth box are not considered, while those with

**Algorithm 1**: Hierarchical Grouping Algorithm

DontPrintSemicolon **Input**: (colour) image
**Output**: Set of object location hypotheses $L$

Obtain initial regions $R = \{r_1, \cdots, r_n\}$ using Felzenszwalb and Huttenlocher (2004) Initialise similarity set $S = \emptyset$;
**foreach** *Neighbouring region pair* $(r_i, r_j)$ **do**
  Calculate similarity $s(r_i, r_j)$;
  $S = S \cup s(r_i, r_j)$;

**while** $S \neq \emptyset$ **do**
  Get highest similarity $s(r_i, r_j) = \max(S)$;
  Merge corresponding regions $r_t = r_i \cup r_j$;
  Remove similarities regarding $r_i : S = S \setminus s(r_i, r_*)$;
  Remove similarities regarding $r_j : S = S \setminus s(r_*, r_j)$;
  Calculate similarity set $S_t$ between $r_t$ and its neighbours;
  $S = S \cup S_t$;
  $R = R \cup r_t$;
Extract object location boxes $L$ from all regions in $R$;

*Figure 2.23:* The pseudo-code of the *Selective Search* algorithm.

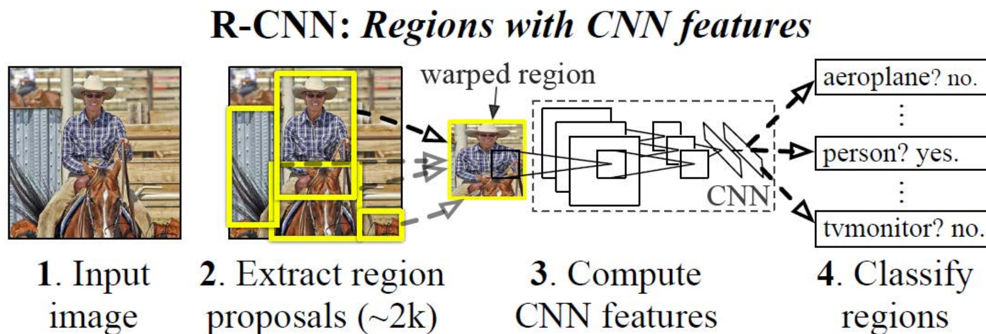



*Figure 2.24:* The pipeline of the R-CNN object detection framework.

an IoU above a certain threshold (fixed to 0.6 by authors in this framework) are adjusted accordingly. Moreover, non-maximum suppression technique is used to discard multiple detections for the same object.

### 2.3.2 Fast R-CNN

*Fast R-CNN* was proposed in [Gir15] as an improvement of the previous versions *R-CNN* and *SPP-Net*. In this new framework, authors tackled the drawbacks of their previous one *R-CNN*: the multi-stage training pipeline, the space and time requirements during training and the inference time during testing. Together with *SPP-Net*, *R-CNN* was very slow both during training and testing because a convolutional neural network is run for each proposed region (that are around 2000). Hence, given a test image, the framework required 47 s to detect and classify objects. With a fps lower than 0.03 it is almost impossible to build a real-time object detec-

tion system. Moreover, *R-CNN* has to be trained in a multi-stage pipeline, where SVM weights are trained after the convolutional layers. This process is inelegant and doesn't allow for the convolutional weights to adapt to the SVM ones, prohibiting hidden relationships. Despite accelerating both training and testing processes by using a single convolutional network over the entire image, *SPP-Net* cannot be completely fine-tuned since weights before the spatial pyramid pooling layer are fixed.

The architecture of *Fast R-CNN* is shown in figure [2.25]. The backbones used to extract features are the same of *AlexNet* and *VGG-16*. The model takes in input an image and a set of proposed region generated by an algorithm like *Selective Search*. The image is passed through convolutional layers to extract the feature maps and then, for each proposed region, a RoI pooling layer extracts a fixed-length vector. This vector is then passed through fully connected layers to extract features. Finally, two separate modules are stacked above the features vector: one is responsible for computing the probability distribution among classes; the other is a bounding box regression layer that computes the box center (x, y) and dimensions (h, w). Differently from *R-CNN* and *SPP-Net* that used SVM, this framework uses *softmax* layer.

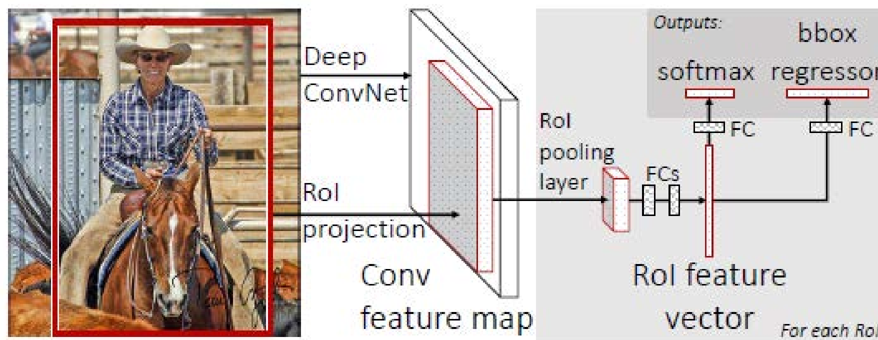



*Figure 2.25:* The pipeline of the *Fast R-CNN* object detection framework.

One of the main advantages of *Fast R-CNN* is the possibility to jointly train the classification, regression and convolutional layers end-to-end. This is done by labelling each RoI with a ground-truth class and a ground-truth set of coordinates. The loss function is a combination of the classification loss and the localization loss: the classification loss is computed as the log loss with the respect to the ground-truth class; the localization loss is a smoothed $\ell 1$ loss that is considered by the authors more stable than the $\ell 2$ loss used in *R-CNN* and *SPP-Net*. The localization loss is added only when a class different from "0" (which is the *background* class) is predicted. A detection confidence is assigned to each bounding box and then non-maximum suppression is performed for each class.

Finally, authors discovered that the 38.7% of time during testing is due to fully connected layers, that are simple matrix multiplications. They proposed to use SVD algorithm to extract singular values and reduce computational effort during testing. With this technique, they were able to reduce the fraction of time for fully connected layers to 17.5%.

The main drawback that authors leaved for future work is the need for an external algorithm to extract the proposal regions. However, they were able to achieve good results both in term of speed and accuracy. They reduced the training time by 8.8× with the respect to *R-CNN* and they also reduced the testing time per single image from 47 s to 0.32 s. The proposed framework outperformed every approach on all PASCAL VOC datasets (2007, 2010 and 2012) reaching, respectively, 70.0, 68.8 and 68.4 as mAP. Finally, they tried to use SVM instead of *softmax* layer for the outputs and they discovered that SVM performed worse by 1 point in mAP.

### 2.3.3 Faster R-CNN

*Faster R-CNN* was proposed in [Ren+15] to improve performances of *Fast R-CNN*, where the speed bottleneck was due to the *Selective Search* algorithm that extracts proposal regions. In *Faster R-CNN*, the model learns itself the interesting regions where to look for objects by using a RPN (Region Proposal Network) that works on top of the convolutional layers and uses the features map extracted to rank the regions proposing the ones that are more likely to contain objects. This is conceptually different from the *Selective Search* approach that is an external method that doesn't share computations with the convolutional layers. The feature maps, alongside with the proposed regions, are then passed through the classification and regression modules that are the same of *Fast R-CNN* framework. This allows for very fast processing of a single image, hence approaching real-time object detection.

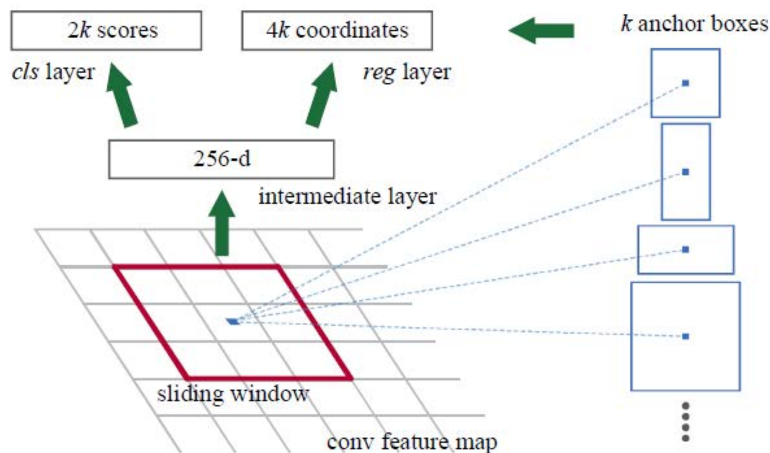



*Figure 2.26:* The anchors are pre-defined bounding boxes.

The input image is processed using a bunch of convolutional layers like *ZFNet* or *VGG-16* backbones and the last layer outputs the feature map. This is fed to the RPN that extracts region proposals using anchors. Anchors (figure [2.26]) are bounding boxes at different locations and with different sizes and aspect ratios that are placed in the image and are fine-tuned during training phase towards ground-truth object annotations. In *Faster R-CNN*, authors used 3 different sizes (128, 256 and 512) and 3 different aspect ratios (1 : 1, 1 : 2 and 2 : 1), leading to 9 anchors for each location. For each anchor, the RPN outputs a classification loss for classes "object" and "not-object" and a bounding box proposal regression (4 coordinates). The anchors boxes are compared to the ground-truth boxes and only those with an

high IoU are kept. Non-maximum suppression is used to remove redundancy. The regions with the higher scores are passed to the RoI pooling layer that computes feature maps with fixed size and finally to the detector network on top of the model that classifies the object in each bounding box and refines them to better enclose the object. The detector network is similar to the one of *Fast R-CNN*. The full architecture is shown in figure [2.27].

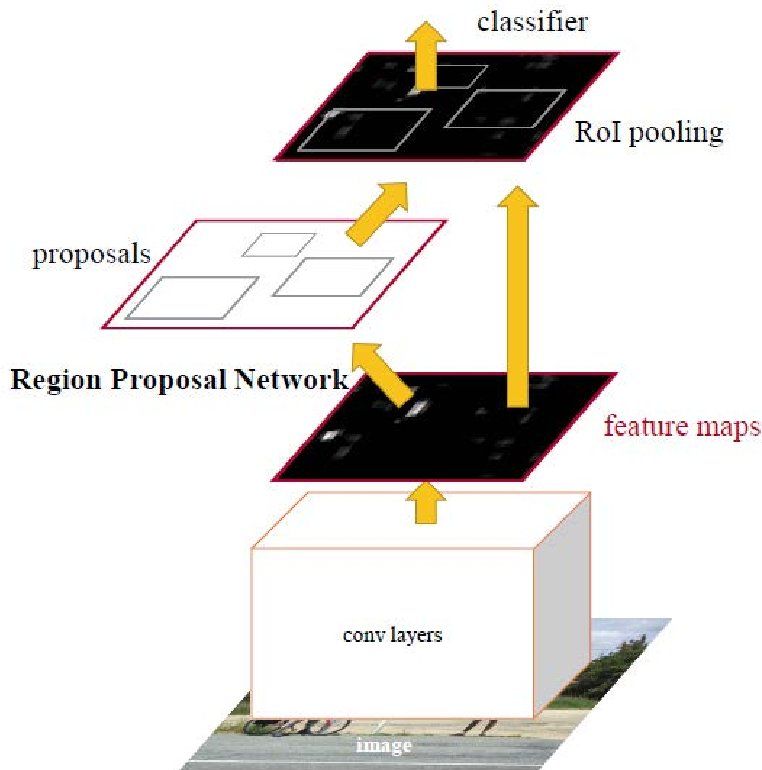



*Figure 2.27:* The pipeline of the *Faster R-CNN* object detection framework.

The training pipeline of *Faster R-CNN* framework is composed by four steps. In the first step, the RPN is trained minimizing a combination of localization and classification loss. In the second step, a *Fast R-CNN* framework is trained using the proposals generated by the first step. In the third step, the detector network is used to fine-tune RPN layers. In the last step, the detection network is fine-tuned.

The *Faster R-CNN* framework outperformed the other methods by reaching 75.9 mAP in the PASCAL VOC 2012 with 300 proposals, less than the 2000 of *Fast R-CNN*. Moreover, with *VGG-16* backbone, the framework was able to elaborate images at 5 fps and with *ZFNet* it was even faster reaching 17 fps, which is a 250× speed-up with the respect to *R-CNN*.

### 2.3.4 YOLO

Introduced in [Red+16] by Redmon *et al.*, *You Only Look Once* is the first object detection algorithm that, as the name suggests, performs object localization and classification by analyzing the full image once, without decoupling the region proposal and classification stages as *R-CNN* models. In practice, the *YOLO* algorithm

starts by dividing the image into a grid of $S \times S$ cells (in the original paper $S = 7$). Then, each cell predicts $B = 2$ bounding boxes, called anchor boxes, with different shape. Each bounding box is identified by its center normalized coordinates and dimensions (relative to the cell), together with the confidence, *i.e.* the probability that it contains an object (without specifying the class), calculated as the IoU between the predicted box and any ground-truth box in the image. At the end, for each cell we have a vector where the first ten numbers are coordinates, dimension and confidence for the first and the second bounding box, followed by the conditional probabilities for each class that we want to detect (20 in the original paper). The predicted tensor can be seen in figure [2.28]. The confidence of the boxes are combined with the class probabilities to compute the final object scores that are used to draw the bounding boxes in the image, as seen in figure [2.29].

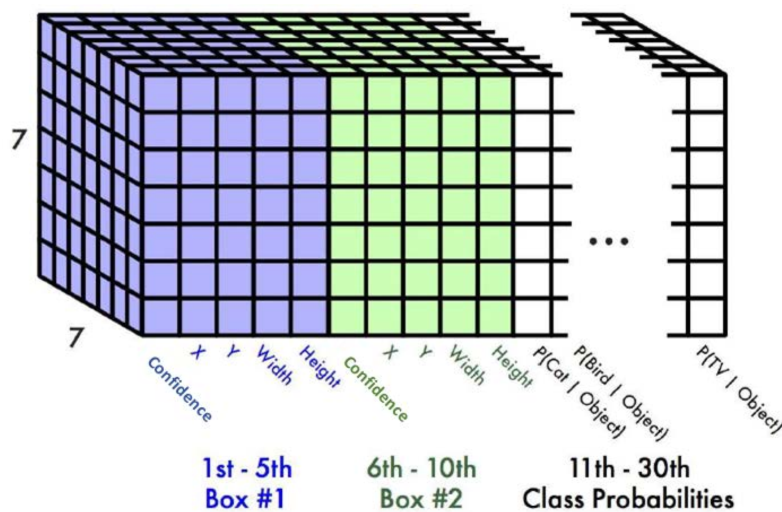



*Figure 2.28:* The tensors produced by the *YOLO* algorithm.

The *YOLO* network has 24 layers in which convolution and max pooling are alternated, followed by two fully connected layers at the end. During training, the IoU is calculated for each bounding box of each cell and the one with the highest value is kept to predict the localization; hence, each cell can predict at most one object. The loss function minimized by the algorithm is the combination of three factors: the first is the classification loss, calculated as the square error of the class conditional probability for each class; the second is the localization loss, calculated as the mean squared error between the predicted box and the ground-truth box, weighted using the hyperparameter $\lambda_{coord}$; the third is the confidence loss, weighted by hyperparameter $\lambda_{noobj}$ if the cell doesn't contain any object. Non-maximum suppression is performed by looking at the IoU between predicted boxes and keeping only the one with the higher score.

The advantage of the *YOLO* architecture is that it can process an image much faster compared to other object detection systems, making it a suitable framework for real-time applications. In fact, *YOLO* can process 45 fps and authors also provided a lighter version with just 9 convolutional layers that can reach 155 fps. This is a significant improvement with the respect to *Faster R-CNN* that has an elaboration speed of 17 fps, without sacrificing to much in accuracy, with results

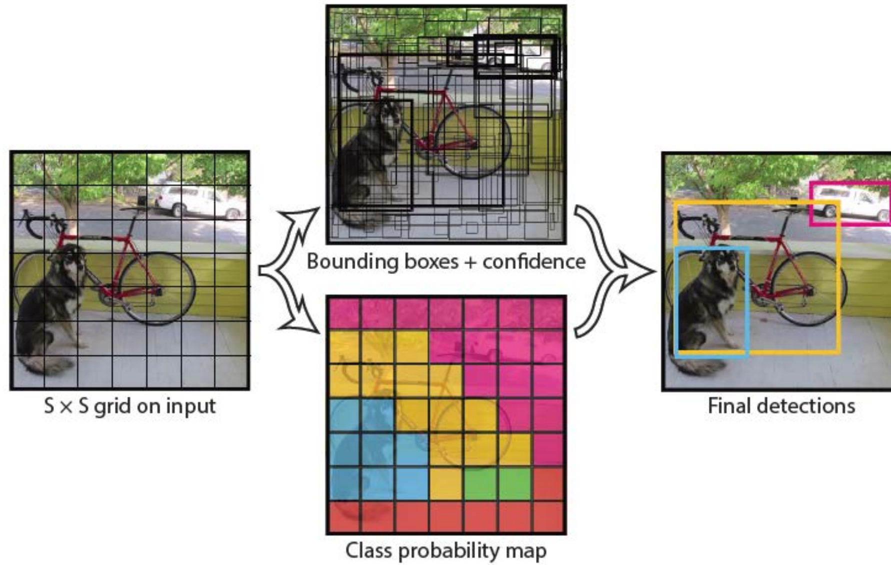



*Figure 2.29:* Bounding boxes and class probability map are used to produce the final detections.

near 63.4% as mAP. Moreover, *YOLO* is much better in background classification if compared to *R-CNN* models, hence reducing false positive errors. On the other hand, *YOLO* algorithm pays the speed in term of localization performances, with more localization errors than *Fast R-CNN*. Moreover, the grid cell imposes spatial constraints on detecting grouped small objects. However, being able to elaborate the images at real-time can reduce the impact of these errors since they are related to each frame and the flow of images somehow mitigates the effect.

### 2.3.5 SSD

The Single Shot Multibox Detector (SSD) was introduced by Liu *et al.* in [Liu+15]. It is an object detection framework that uses an approach similar to the one used by *YOLO*, hence it is able to work with a single pass over the input image, proving to be a valid approach to real-time object detection applications. It was introduced soon after *YOLO*, outperforming it by running at 59 fps on the VOC2007 dataset with a mAP of 74.3%.

The SSD framework uses *VGG-16* as feature extractor network, but modifies the layers on top of it for the localization and classification purposes. In particular, from the $4^{th}$ convolutional layer, the feature map is convolved with a $3 \times 3$ kernel that produces an output with dimension $m \times n \times k \times (num_class + 4)$ where $m \times n$ is the feature map dimension, $k$ is the number of anchors to use at that level, $num_class$ is the number of classes to detect (plus background class) and 4 is related to the offset of the bounding boxes. In parallel, the feature map is again convolved and max-pooled to further reduce the dimension. In this way, the subsequent classification layers that operates over lower dimensional feature maps are suitable for detecting larger objects. Finally, for the last two convolutional layers, the dilated convolution ... is used to increase the receptive field without having too much parameters. In total, 6 additional convolutional layers are added upon *VGG-16* network, some with

6 anchors and some with 4 anchors. The full architecture is shown in figure [2.30].

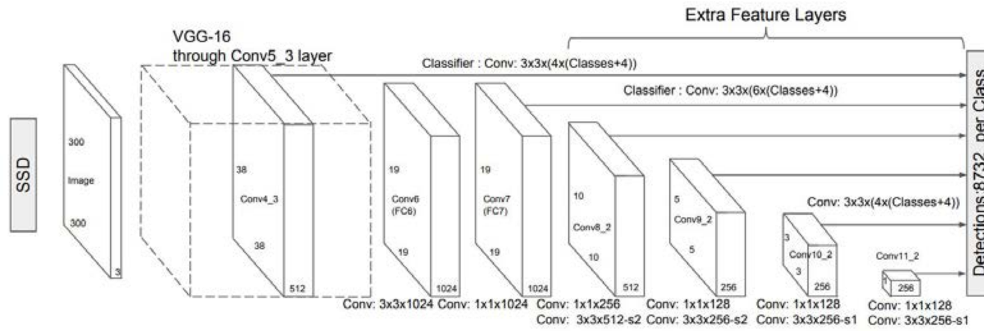



*Figure 2.30:* The pipeline of the *SSD* object detection framework.

Compared to *Faster R-CNN*, the *SSD* framework performs badly with small objects because the first feature map layer on which anchors are used to detect objects has a dimension of $38 \times 38$, much smaller than the input resolution (300 or 500). Moreover, the higher the input resolution, the higher the classification and localization accuracies, but the lower the frame rate (22 fps with *SSD-512* and 59 fps with *SSD-300*).

### 2.3.6 YOLOv2 and YOLO9000

In [RF16], the authors of *YOLO* provided some modifications to their object detection algorithm to make it namely "better, faster and stronger". Moreover, they trained the new *YOLO* architecture called *YOLOv2* on 9000-class dataset organized with a tree structure in order to refine the classification task and make the algorithm more specific when predicting the class.

In the *YOLO v2* algorithm, batch normalization is added for regularization purposes and helps improving the accuracy. Moreover, higher resolution images ($448 \times 448$ instead of $224 \times 224$) are used to fine-tune the network in order to increase the classification accuracy. Instead of hand-picking the bounding boxes during training as in *YOLO v1*, here authors proposed to use *k-means* algorithm on ground-truth boxes to extract the more frequent dimensions and aspect ratios. According to the paper, the distance metric used is $1 - IoU(box, centroid)$. In this way, the network is more likely to predict the correct bounding boxes because it starts from similar ones. In the paper the number of clusters is 5, hence 5 anchors are predicted for each grid cell. In addition, authors used logistic activation to constrain the offset predictions of the bounding boxes and relating its dimensions to the grid cell, making the network more stable (figure [2.31]). To fine-tune the model on higher resolution features and improve performances on small objects, a passthrough layer concatenates the $26 \times 26$ features with the $13 \times 13$ features of the subsequent layer (in a similar way of the residual connections in ...) to provide richer features. Finally, multi-scale training is used randomly scaling the images.

In *YOLOv2* a new backbone architecture called *Darknet-19* is introduced with 19 convolutional layers and 5 max-pooling layers. This feature extraction network is very powerful with 72.9% top-1 and 91.2% top-5 accuracy on ImageNet and a

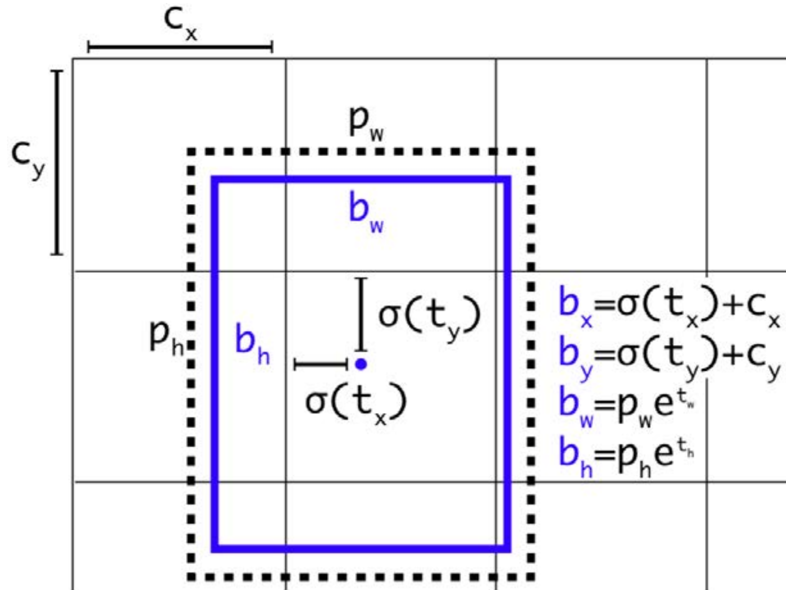



*Figure 2.31:* The coordinates of the new bbox (blue) are predicted with the respect to the grid cell coordinates and the prior anchor (dashed black), in order to make the network more stable.

reduction of a factor 6 in number of operations for a single image. On top of it, other 11 layers are added for object detection.

In order to make the model stronger and more powerful in the classification task, authors proposed to jointly train it on object detection and image classification, using simply labelled images to expand the number of detectable classes. In this way, it is possible to use multiple datasets to train the same network. In particular, authors built a WordNet-inspired graph (figure [2.32]) fusing COCO and the 9000 top classes of ImageNet datasets. This dataset is used to train a model called *YOLO9000* where the predictions are class specific, for example predicting not only "dog" but also the dog breed. In this case, only 3 anchors are computed for each cell grid instead of 5. To cope with different tasks (detection and classification), the loss for images where only classification is required is back-propagated only to classification layers without refining the bounding box predictions.

For what concern performances, *YOLO v2* has been tested with multiple size images showing that it is more accurate than *YOLO v1* with 78.6% mAP in PASCALVOC2007 dataset running at 40 fps with $544 \times 544$ input images. Compared to *SSD300* and *SSD500*, this algorithm has similar accuracy but can process images at doubled speed.

### 2.3.7 YOLOv3

The *YOLO v3* version was published in [RF18] as a technical report explaining the improvements made over *YOLO v2* model. Authors decided to increase the dimension of the features extractor network in order to cope with the main issue of the previous model: the detection of small objects. Besides being a bit slower, running at 30 fps, this architecture improves the overall mAP and is able to detect

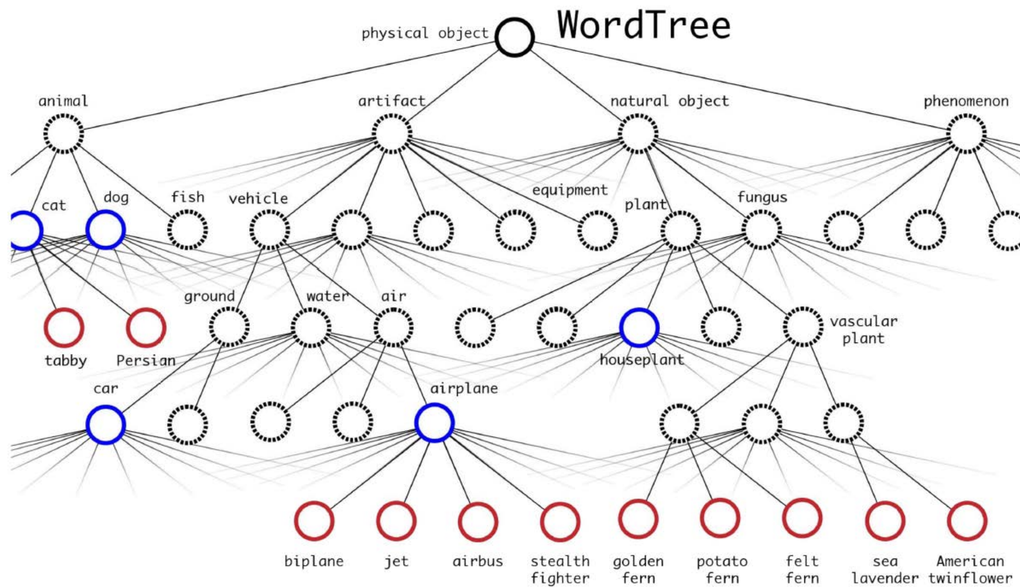



*Figure 2.32:* The hierarchical model for classification obtained by fusing COCO dataset (blue) and ImageNet dataset (red).

also small objects that are missed by *YOLO v2*.

The backbone architecture of *YOLO v3* is a *Darknet* network with 53 layers for features extraction pre-trained on ImageNet. Then, other 53 layers are stacked for detection purposes, producing a 106-layers model. In figure [2.33] the full architecture is shown.

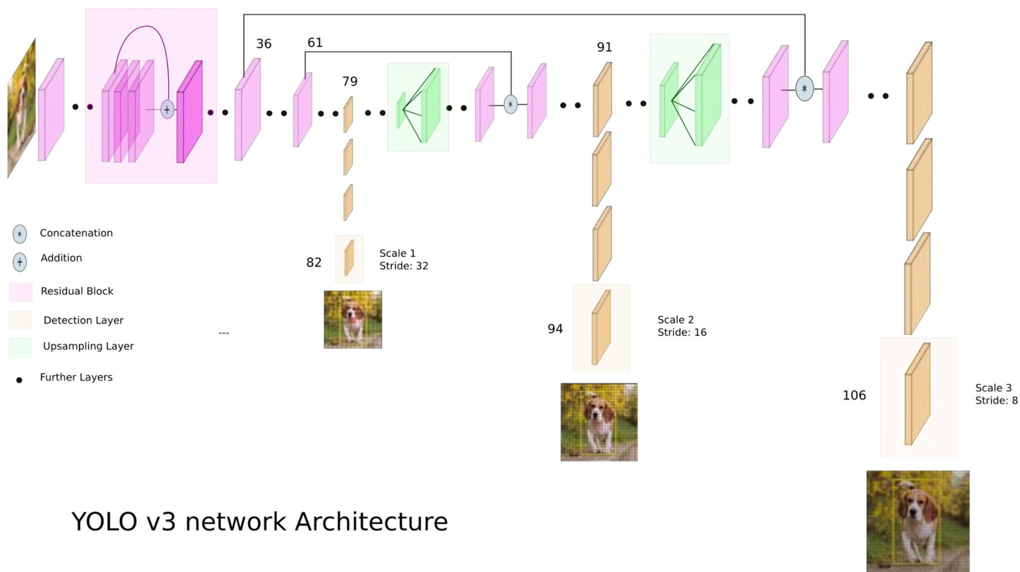



*Figure 2.33:* The full architecture of the *YOLO v3* model.

The most powerful feature introduced with *YOLOv3* is the detection at multiple scales. In fact, the previous models failed to detect small objects because, after many max-pooling layers, the resolution is very low and details are lost. For multiscale

detection, a $1 \times 1$ convolution kernel is used on feature maps three times in the network. In this way, at the first scale (stride of 32) small objects are detected since the resolution is still high. Then, the feature map is upsampled and concatenated with the previous one before going deep with convolutions. At the second scale (stride of 16), medium size objects are detected, while the last scale (stride of 8) is responsible for detecting big objects occupying the most part of the image. The connections between layers inspired by *ResNet* architecture are useful to keep track of the features at different resolutions and this improves the accuracy. *YOLO v3* uses 3 anchors for each scale selected using *k-means* clustering, for a total of 9 anchors. This led to a number of proposed boxes that is $10\times$ higher than *YOLO v2*. Finally, authors replaced *softmax* layer for classification with *logistic regression* layer followed by a thresholding function because the classes, in general, are not mutual exclusive (think, for example, at "person" and "woman").

# Chapter 3

# Description of the Project

In this work, I exploited the state-of-the-art object detection algorithms and frameworks developing a complete application for the fashion retail industry. The final goal of the project is to build a system that can be used in the industry as a powerful tool for several applications both for the customers and for the staff. Examples of applications are self-checkout, automatic inventory, products recommendation or commercial contracts supervision. In order to build such applications, the fundamental requirement is a virtual mapping of the store and all the merchandise in the shelves. The acquisition of the virtual mapping of the items on the shelves is the topic of this thesis. The system can be divided in two modules: the Object Detection and the Web Service (figure [3.1]).

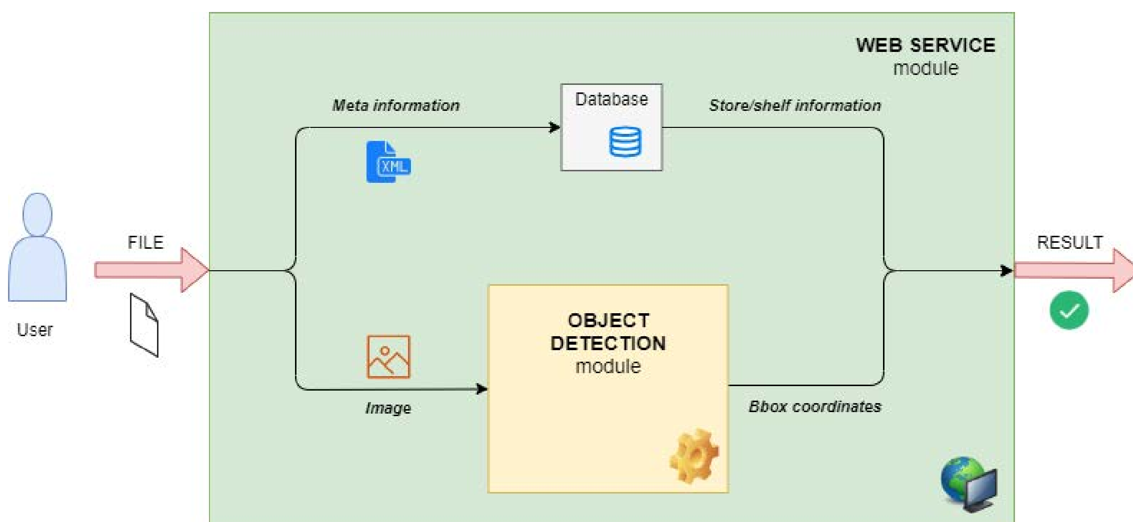



*Figure 3.1:* The overall diagram of the project.

## 3.1 Object Detection Module

The object detection module is the core of the application. This module is basically an artificial deep neural network trained on a customized dataset, which takes in input an image and returns a series of annotations on the absolute position of some predefined items, if they are present in the image. The common way to show the

results of the detection is to draw the bounding boxes on the original image and to indicate the class of each object, together with the accuracy of the detection. For this module, the main tasks are:

- the acquisition of the dataset
- the annotation of the images
- the training of the models
- the selection of the most suitable model
- the post-detection processing

The acquisition of the dataset is the starting point of the processing pipeline. Since all deep-learning based frameworks for object detection downsize the input images, high quality images are not needed. As a result, professional cameras are not needed for this kind of applications, and images are acquired using smartphones. Modern smartphones can acquire images with up to 4K resolution, hence images have to be rescaled after acquisition. In acquiring the images, artifacts like illumination, contrast and camera position have to be considered: on one hand, having images with some variance helps to have a more robust model; on the other hand, if the artifacts are very strong, changes are that the model encodes the "errors" as discriminating features. After the images acquisition, it is possible to rescale them in order to be able to test the object detection models with different resolutions and provide the one that performs better.

In the annotation phase, bounding boxes coordinates and product classes have to be assigned to each image. To do this, there exist some open source software with a graphical interface that gives helpful tools. This process is done just for one resolution, since the bounding box coordinates can be rescaled for images with a different resolution using some code.

In order to be trained, a model needs the images and the bounding box coordinates. The annotation files created with the labelling tool are converted in the correct format using some scripts. Then the model is fine-tuned to detect and classify only the desired items.

The model selection is dependent of the application, since each model has some pros and cons: as a general rule, the more accurate is a model, the slower it is. As a result, the correct model has to be chosen as a compromise between speed and accuracy. In this project, since the application is not meant to work in real-time, an higher accuracy is preferred over a fast model.

Since the object detection models don't have always 100% accuracy, it is necessary to further process their output in order to make the system more robust. Dealing with object detection, the more frequent artifacts are false positives and false negatives, together with localization and classification errors. In this project, techniques are used to reduce those errors before providing results to users.

## 3.2 Web Service Module

The Web Service module encapsulates the object detection module in order to use it in an higher level application, in this case a client/server service. In particular, this module takes care of receiving the client request with the image to elaborate and to pass it to the detection module, together with elaborating the results before replying to the client request. In this module, the main tasks are:

- the design of the architecture
- the front-end/back-end interaction
- the resource management

The object detection module is a routine that takes an image from a directory and, after the elaborations, displays it with the bounding boxes. This routine is suitable to rapidly test the model on a given image (or set of images), but not for a client/server application. The architecture is designed in order to hide the object detection module from the client perspective and to provide only the readable information.

In this project, the front-end is a web page (figure [3.2]) that provides a form to be filled with the image and information about it. The information flow between the web page and the object detection routine and the other way around is handled using classes.

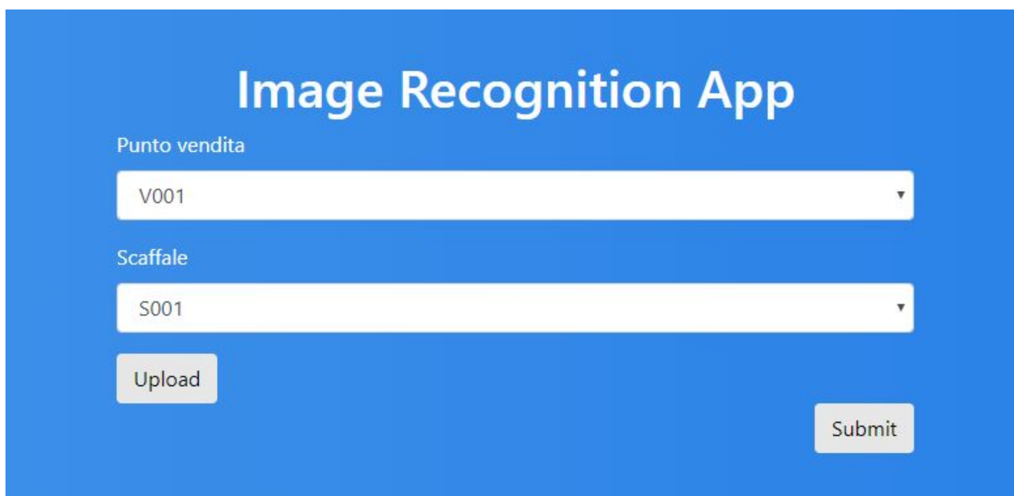



*Figure 3.2:* The initial page of the web application provided to the users.

When the number of requests is high, the information of each should be handled correctly by classifying them and storing the essential data, together with logging the actions performed by the system.

# Chapter 4

# Setup of Components

In this chapter, the framework and the software used for the project are illustrated. Firstly, an overview of *TensorFlow* framework is given, by explaining why and how it is used in this work. Then, other tools that are used are briefly presented.

## 4.1 TensorFlow

*TensorFlow* is python-friendly end-to-end open source framework for machine learning. It was developed by *Google Brain* team and provides a wide range of functions to build, train and test mathematical models. Besides offering a set of tools to develop any learning model, from SVM to deep neural networks, *TensorFlow* has some built-in functions to manipulate and process data before feeding the models. *Tensorflow* provides also an easy-to-use API written in Python to assist the model development, while running it with the highly-performing C++. This frameworks enables users to create *dataflow* graphs, that are structures that describe how the data moves inside the model. Each node in the graph is a specific operation, while edges are multidimensional arrays, that are called *tensors*. Python language provides high level of abstraction giving the user the possibility to write relatively few lines of code, that are translated into C++ instructions by the compiler. The major benefit of *TensorFlow* is the possibility to focus on the overall logic of the applications instead of dealing with the precise implementation of the algorithms. Moreover, it provides a web-based dashboard to monitor the training phase of the models, which is *TensorBoard*.

Every graph in *TensorFlow* is composed by a set of elementary objects. In particular, there are objects that store data (Tensors) and objects that make transformations (Operational node). Any algorithm can be written in this form.

By concatenating these objects, it is possible to create a computational graph, which is a set of nodes, each of them running a single operation. Then, a *Tensorflow* Session is initialized and input data are fed to the graph. Once the computation is over, output data are returned. A Session encapsulates the environment where all the operations are computed.

An example of *TensorFlow* code is given in figure [4.1], where some elementary operations are performed. In particular, after creating two vectors with dimension $1 \times 3$ ($x$ and $y$), they are either added, multiplied and subtracted in 3 different operational nodes (*z1*, *z2* and *z3*). When the snippet of code is executed, the com-

putational graph showed in figure [4.2] is generated, and output data are produced only when the operational node has been given to the *run()* function.

```
import tensorflow as tf

session = tf.Session()

x = tf.constant([1, 2, 3], name="x")
y = tf.constant([4, 5, 6], name="y")

z1 = tf.add(x, y, name="z1")
z2 = x*y
z3 = z2-z1

session.run(z3)
```

*Figure 4.1:* This snippet of code performs some elementary operations on two tensors.

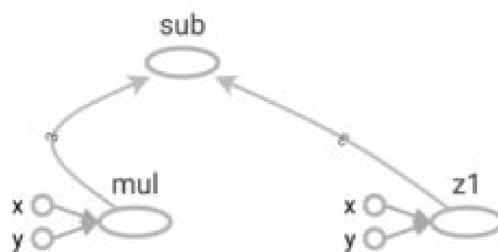



*Figure 4.2:* This is the computational graph generated when executing the snippet in figure [4.1].

### 4.1.1 TensorFlow Model Zoo

*TensorFlow Model Zoo* (figure [4.3]) is a collection of models that are pre-trained on some datasets and are made available. The features extraction layers of each model are adjusted during training and then fixed, while the classification layers can be fine-tuned by users to adapt the model to the specific application. For each model, authors provide:

- a *graph proto*, which is a dictionary id/class of the detectable items
- a *checkpoint*, which is the model state after the last training iteration
- a *frozen graph proto*, which is the model available for detection
- a *configuration file*, which contains training information

The frozen graph, which is actually the object detection model, can be directly used to detect the objects listed in the *graph proto* in any image. However, it is also

possible to fine tune the model on a customized dataset with specific classes. In order to do this, the configuration file should be changed by indicating the location of the training and test images and bounding bounding box files, together with the training parameters (batch size, learning rate, images size, data augmentation functions, ...). The *graph proto* should be changed in order to list the specific classes of the customized dataset. After these settings, the checkpoint model is further trained with the customized dataset and, at the end, the frozen graph is exported.

The available models are divided according to the dataset on which they are pre-trained. The majority of models are pre-trained on the COCO dataset, but there are some of them pre-trained on Kitti, Open Images, iNaturalist Species, AVA datasets. The two most used object detection frameworks are SSD and Faster R-CNN, that are available with different backbone architectures for feature extraction. Each model is provided with speed and mean average precision. The speed is the time needed by the model to produce the output on a $600 \times 600$ image using a NVIDIA GeForce GTX Titan X card, and it is expressed in milliseconds. The mean average precision is specific for each dataset, and it is a measure of the accuracy of the model. As suggested by the authors, these measures are not to be considered in an absolute way, but better for comparing the models before selecting the most suitable for the customized application. The general rule is that the faster is the model, the lower is its accuracy.

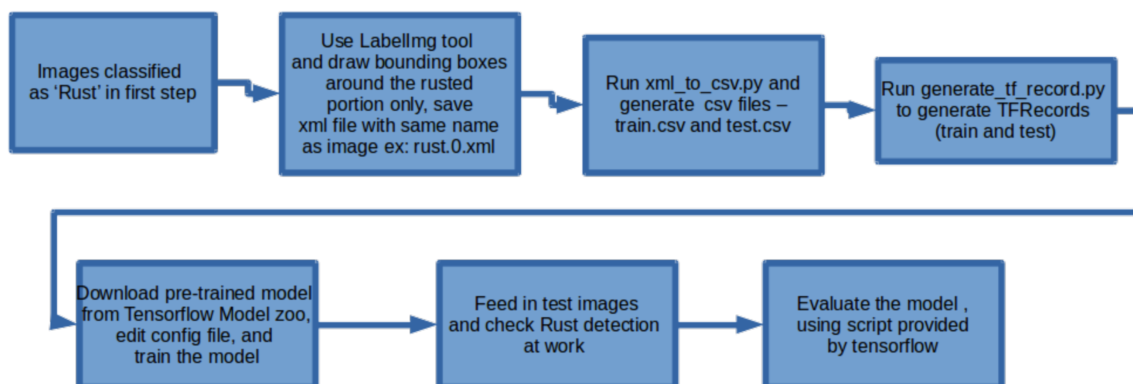



*Figure 4.3:* TensorFlow Model Zoo provides an end-to-end way to fine-tune object detection models on customized dataset.

## 4.2 Hardware

The training of the object detection models is the most computationally demanding part of the project. In order to be able to train such powerful models with an high quantity of images (about 800), a Virtual Machine and Bare Metal GPU provided by Oracle Cloud has been used. The machine was powered by a NVIDIA Tesla P100 single GPU with 2.0 GHz Intel Xeon Platinum 8167M CPU. The object detection models, together with the training images and the training pipeline configuration files, were uploaded on a NVIDIA Docker instance running on the Oracle Cloud and connected to the GPU. With this configuration, the time required to train a single model on the entire dataset was from 16 to 24 hours.

The laptop used to test the models and to develop the web service was a Dell Inspiron 15 with a Quad Core Hyper Thread Intel Core i7-8565U@4 60Ghz (Turbo) CPU and Ram 8GB DDR4@2.666Mhz.

## 4.3 LabelImg

*LabelImg* is a tool written in Python that can be used to label the images in order to prepare the dataset to train an object detection model. This tool provides a user-friendly way to store the bounding boxes coordinates of objects in the image by supplying the user with a graphical interface. After uploading the image (figure [4.4]), it is possible to draw a box with the mouse and then to select the correct class from a drop-down menu (figure [4.5]). The bounding box coordinates can be saved in the PASCALVOC format (an *xml* file for each image) or in the YOLO format (a *txt* file for each image).

*Figure 4.4:* The *LabelImg* GUI after loading the image.

## 4.4 Flask

Flask is a lightweight WSGI (Web Service Gateway Interface) web application framework, written on the top of Werkzeug, which is a Python library that allows to build objects like HTML requests and responses, and Jinja, which is a template engine for Python programming language.

With Flask, it is possible to create a web service and to handle some web pages. Flask provides the structure of the service, while the content of the web page and the functions to call are handled separately (figure [4.6]).

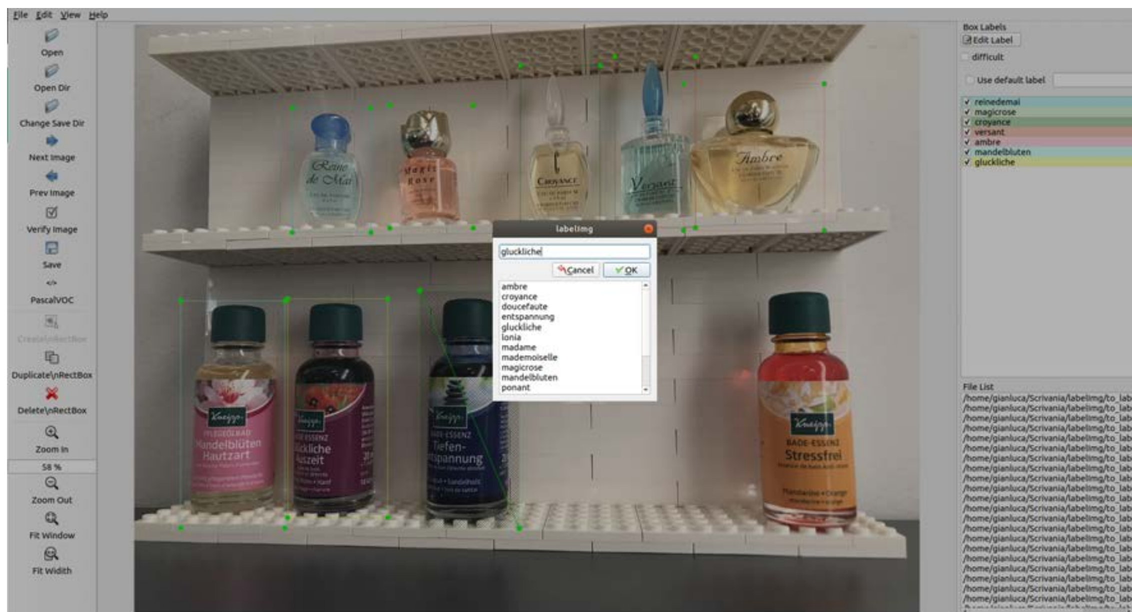



*Figure 4.5:* After having drawn the bounding box, a drop-down menu lets you choose the correct class.

```
from flask import Flask

home = Flask(__name__)

@home.route("/route/to/destination", methods=["POST","GET"])
def name_of_a_function():
    ...
    ...
    ...
    ...
    return ...

if __name__ == "__main__":
    home.run(debug=True)
```

*Figure 4.6:* The *Flask* environment allows to create a Web Service where the function to run on the web page are those called by the function "*name_of_a_function()*"

# Chapter 5

# Dataset and Models

In this chapter, the datasets used for the application are presented and explained after an introduction on the hierarchy of the classification task. Finally, the architectures chosen for the training are presented.

## 5.1 Dataset

Every application that uses a deep learning model needs an high number of images to be successfully trained and so the dataset is one of the most important part of the object detection module. While for image classification we just need a set of images with one attribute (the class), for object detection we need a more structured annotation file for each image.

The acquisition of a dataset is composed of three phases: in the first part, the acquisition part, the images are physically acquired using a device; in the second part, the processing part (which is not mandatory), it is possible to process the images by resizing or cropping them, changing the illumination or the contrast and so on; in the third part, the labelling part, every object in each image has to be manually annotated.

In the first part, we have to consider the properties of the device as important parameters for the system. In particular, the resolution of the image is directly proportional to its dimension in the disk: hence, if we acquire images with an extremely high resolution, the computational time required to process them increases. On the other hand, choosing a very low resolution reduces the dimension of the image, but details that are crucial for the classification may be lost. Moreover, we have to consider also the illumination conditions and the background of the images. The general rule is to have a dataset of images that is as much similar as possible to the scenario where the application will work. In fact, if the images used for training are too different from the images that the system will work with, changes are that the model is not able to generalize and details that are typical of the training set are assumed to be crucial for the specific object. For example, if we acquire all the images using a black background, the contrast between the object and the background will become a feature of the object that can affect the accuracy of the system in case that images for real application are acquired with, for example. white background.

In the second part, we can digitally adjust the image properties using a framework like *OpenCV*. However, since the aim is to have high variability in the dataset

in order to train the model to "see" all possible types of images, this part is often used for a process called data augmentation. In data augmentation, the dataset is enriched by providing copies of the real images with some changes in the scale, illumination, contrast, noise and so on. There are two types of transformations: the label-preserving and the label-changing. The former transformations don't alter the absolute coordinates of the bounding box in the image, so they can be performed after labelling phase, since we just need to assign to the new image the same annotation file of the real image. The latter transformations generate an image in which the position of the object has changed, and so they have to be performed before labelling.

In the third part, we use a labelling tool to annotate the bounding boxes class and coordinates for each image. Since this procedure cannot be automatic, it is the most onerous and we need to pass all the images one by one. For this reason, the label-preserving operations are preferred over the others. The labelling tool used in this project gives the possibility to produce two types of annotation file: the default used by *YOLO* framework, where bounding boxes are saved in a *txt* file, one line for each image; the one defined by *PASCAL VOC 2007*, where an *xml* file is produced for each image (figure [5.1]). Each bounding box is defined by a set of 4 parameters, and there are two ways to define them (figure [5.2]): [x_center, y_center, width, height] or [x_min, x_max, y_min, y_max]. In this project, the convention is to use the second way.

<table>
<tr><td>folder</td><td colspan="6">dataset_gian_2592x1944</td></tr>
<tr><td>filename</td><td colspan="6">train1.jpg</td></tr>
<tr><td>path</td><td colspan="6">/home/Scrivania/dataset_gian_2592x1944/train1.jpg</td></tr>
<tr><td>size</td><td>depth</td><td>3</td><td>width</td><td>800</td><td>height</td><td>600</td></tr>
<tr><td colspan="7">objects</td></tr>
<tr><td></td><td colspan="2">name</td><td>xmin</td><td>ymin</td><td>xmax</td><td>ymax</td></tr>
<tr><td></td><td colspan="2">tiefen</td><td>225</td><td>252</td><td>323</td><td>469</td></tr>
<tr><td></td><td colspan="2">mandelbluten</td><td>340</td><td>253</td><td>430</td><td>454</td></tr>
<tr><td></td><td colspan="2">gluckliche</td><td>468</td><td>254</td><td>565</td><td>452</td></tr>
<tr><td></td><td colspan="2">stressfrei</td><td>608</td><td>250</td><td>711</td><td>450</td></tr>
</table>

*Figure 5.1:* For each image, the *xml* file contains information about the coordinates and classes of the bounding boxes. In this table, those information are resumed.

### 5.1.1 Hierarchical representation of objects

In object detection applications, an important aspect to consider is the hierarchy of objects representation. Indeed, each object is not uniquely defined by a tag, but can be indicated at different hierarchy levels. For example, if our aim is to detect pedestrians in the street for self-driving car systems, we don't need a fine representation of each person in the city, because we are interested in detecting the class "person" better than "John" or "Paul". However, "John" and "Paul" are instances of the class "person", so these tags are not an error. On the other hand, if our aim is to detect people in an office using surveillance cameras, the set

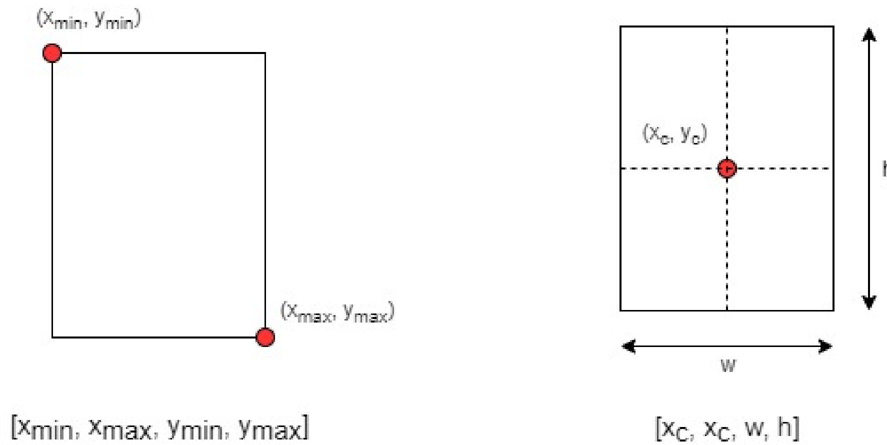



*Figure 5.2:* Two possible ways to represent the coordinates of a bounding box. For this application, the left one is used.

of people allowed to enter the office is perhaps predefined. Here, a system that labels each person with the tag “person” is too trivial, and we may need something more sophisticated with a lower level of representation where the classes are “John”, “Paul” and all the other employees. Again, while in the self-driving system we just need to label cars as “car”, a dealership may be not interested in such a system, since all objects that they deal with are cars. Hence, they need a finer representation level, where classes are the specific car models.

In this project, where objects are specific perfume and shampoo phials, we are not interested in marking them as “phials” or “bottles”, but we need a finer representation, where we can assign the class names. For a specific application, the datasets available online are not useful since they are made for an high level of hierarchy, where classes are “car”, “ball”, “person”, “bottle” and so on. Hence, we built a customized dataset to train the models.

### 5.1.2 Dataset *Oracle*

The first dataset of images is composed by a restricted set of 10 objects among the 16 available. For each object, almost 100 *jpg* images have been acquired using a fixed camera and a mobile platform that rotates: in this way, each item is photographed from 360° angles. The resolution of each RGB image is $640 \times 360$ and it contains just one object in a neutral white background. In this dataset, objects are acquired at only one scale since the distance between the camera and the platform is fixed. The number of images per item is showed in figure [5.3].

### 5.1.3 Dataset *Reply*

The second dataset of images is composed by all the 16 class of objects. In total, 1000 *jpg* images have been acquired with a resolution of $4032 \times 3024$. Each image contains from 2 to 7 objects arranged as in a shelf, with some of them in the foreground and others in the background. Moreover, there are some objects that are partially occluded by others. In order to have flat basement and background, white papers

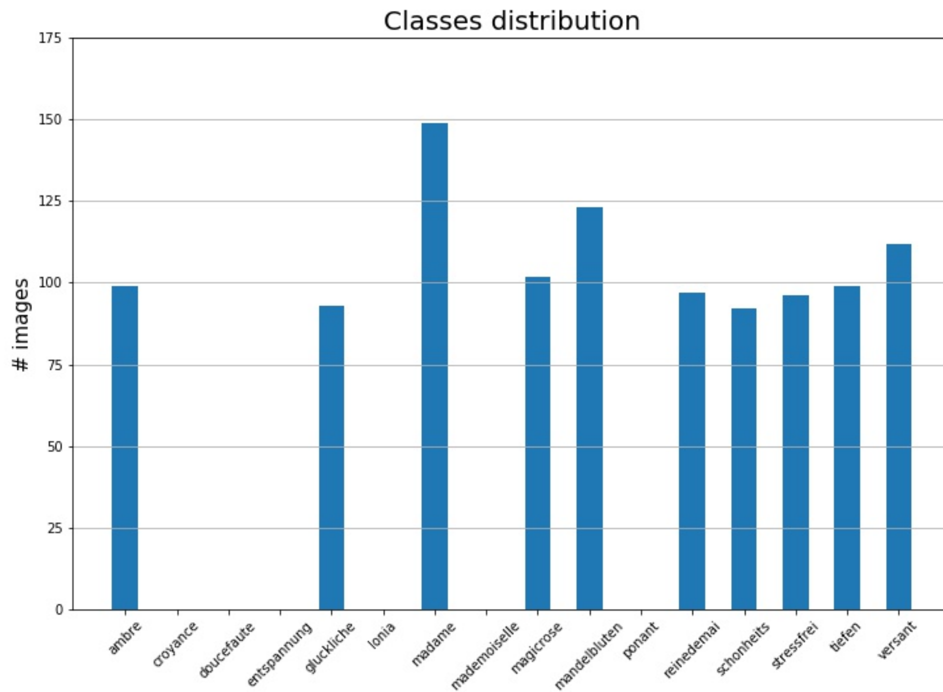



*Figure 5.3:* Distribution of the classes in the dataset *Oracle*. The empty bars are for items that were not used for this dataset.

*(a)* "ambre"

*(b)* "stressfrei"

*Figure 5.4:* Two images of dataset *Oracle*.

have been placed under and behind the objects. Differently from the other dataset, this time the basement is fixed and the camera is moved to change the angles of acquisition. In addition, each object has been slightly rotated or moved between two acquisitions. The classes distribution is showed in figure [5.5] and the distribution of the number of items per image in figure [5.6]. Since the resolution is extremely high, in the processing phase those images has been scaled to $2592 \times 1944$ and $800 \times 600$ resolutions, that are more suitable to be fed to the deep neural network models.

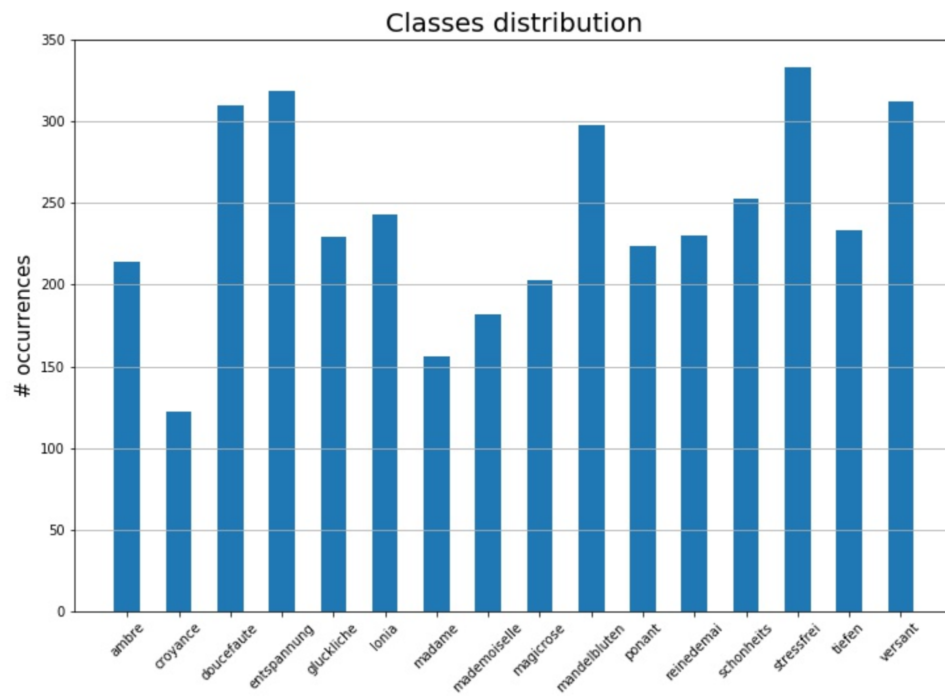



*Figure 5.5:* Distribution of the classes in the dataset *Reply*.

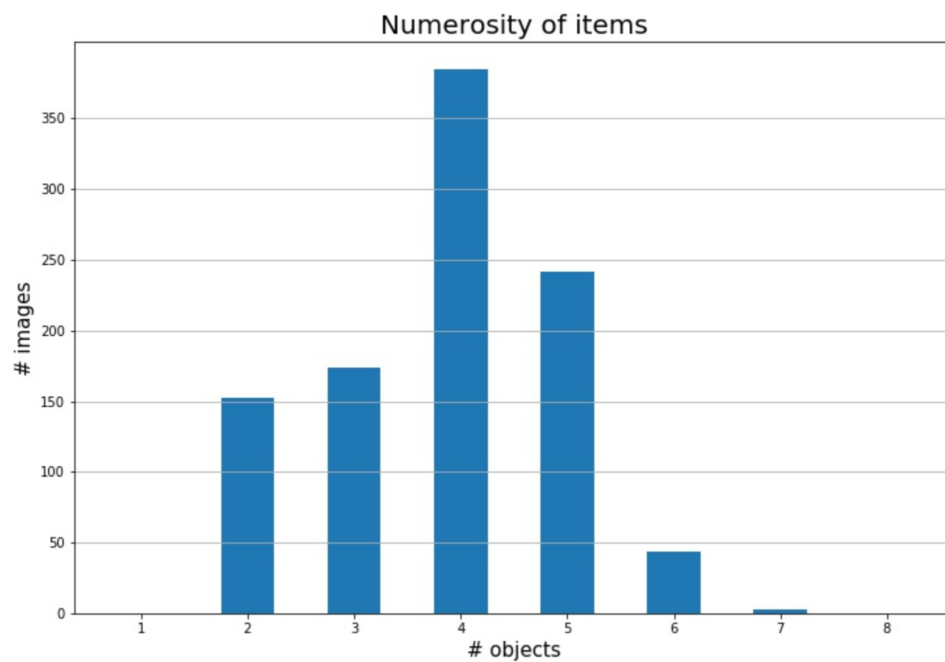



*Figure 5.6:* Distribution of the items per image in the dataset *Reply*.

(a) (b)

*Figure 5.7:* Two images of dataset *Reply*.

### 5.1.4 Dataset *Test* $\alpha$

The third dataset of images is similar to version *Reply*. In fact, this dataset was not intended to train a model, but it is used to test the already trained networks comparing them according to the most used metrics in the field. The images of this dataset are acquired placing the items in a shelf that is designed according to one possible scenario of application of the system. In particular, the shelf is built using white *LEGO* bricks and is composed by two corbels, each of which can contain up to 7 items. In total, this dataset is composed by 100 *jpg* images. The classes distribution of this dataset is showed in figure [5.8] and the number of objects per image in figure [5.9].

### 5.1.5 Dataset *Test* $\beta$

This dataset was also intended to test the models and it is similar to dataset *Test* $\alpha$. While dataset *Test* $\alpha$ contains approximately 7/8 products in each image, this dataset is mostly composed by images with only one item which is moved in different places in the shelf, in order to study the performances of the model among all possible positions in the shelf. The dataset is composed by 303 *jpg* images. In figure [5.11] it is shown the classes distribution of the dataset, while in figure [5.12] it is shown the distribution of the number of items for each image.

## 5.2 V0: Models trained with Dataset *Oracle*

The first attempt was to train some models using the dataset *Oracle*, with images of just one item from different points of view. We trained on both SSD and Faster R-CNN framework. These training stages were thought to infer the ability of the models to learn to detect items and to analyze the strength of the model against resizing and illumination changes in the scene. However, the results were very poor, since each model was not able to correctly classify and locate any of the object. For this reason, the results of these models are briefly presented in this chapter. The model used to produce the following 2 figures has been trained only on the item

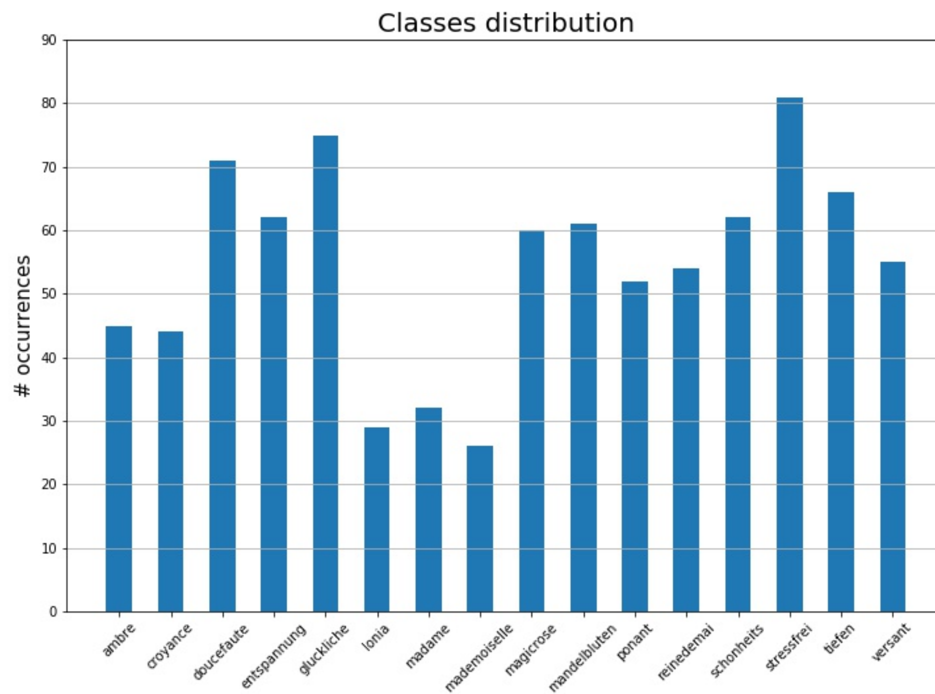



*Figure 5.8:* Distribution of the classes in the dataset *Test* $\alpha$.

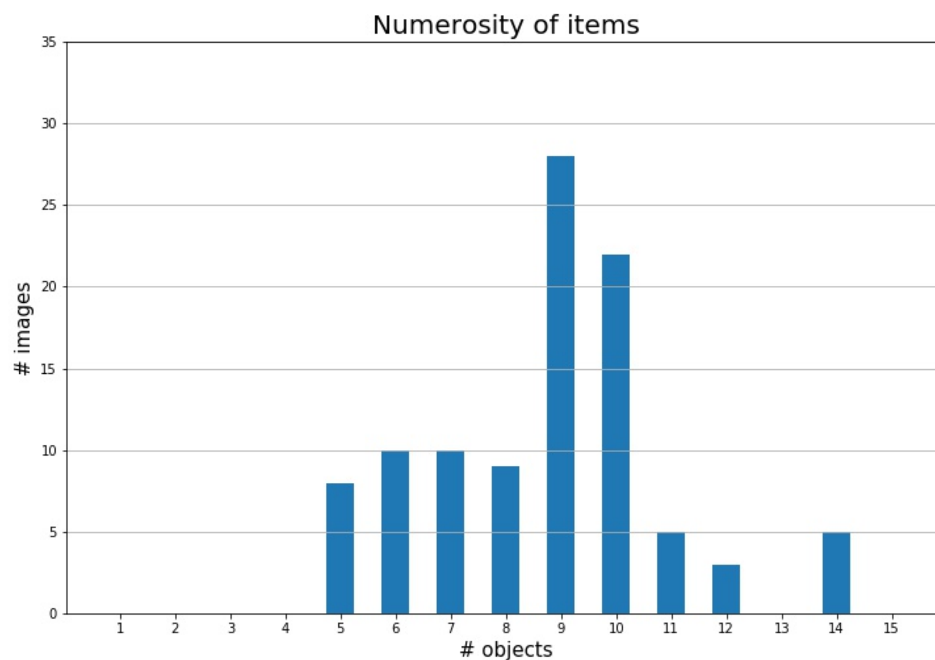



*Figure 5.9:* Distribution of the items per image in the dataset *Test* $\alpha$.

(a) (b)

*Figure 5.10:* Two images of dataset *Test* $\alpha$.

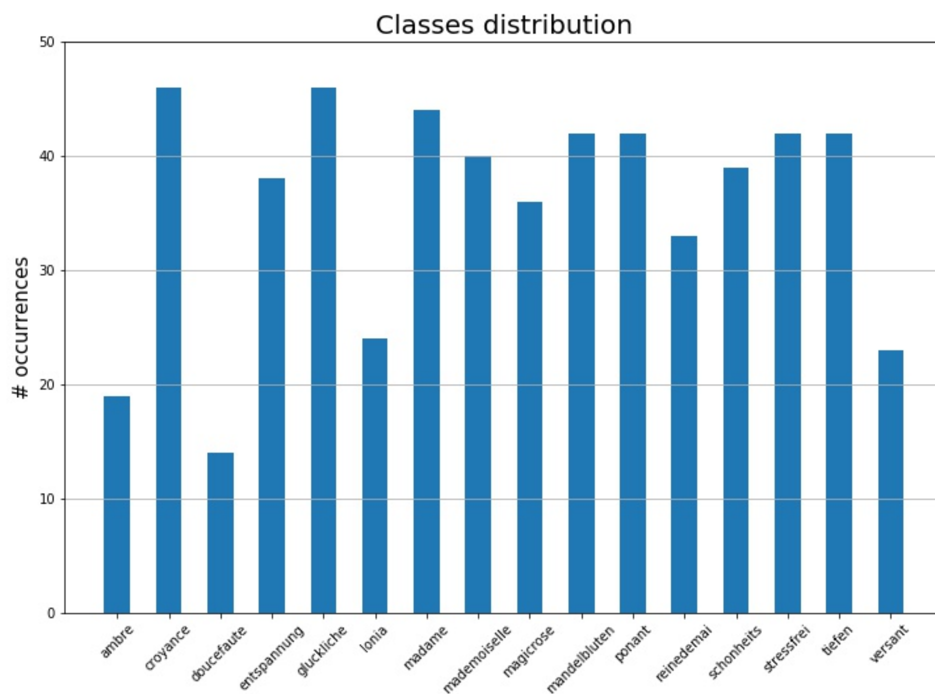



*Figure 5.11:* Distribution of the classes in the dataset *Test* $\beta$.

*"ambre"* for simplicity. In figure [5.14], the image on the left is the original one acquired with the camera, with very low contrast. The object detection model was not able to detect the phial *"ambre"*. If the contrast is increased, as in the right image, the model correctly detects the product at the right position.

In figure [5.15], the left image is the original one, with 9 items plus the *"ambre"* on the bottom right corner. When the object detection algorithm is run over this image, the class *"ambre"* is wrongly assigned to a bounding box that fills the majority of the image. If the image is cropped, as in the right one, the algorithm correctly identifies the item *"ambre"* at the correct position.

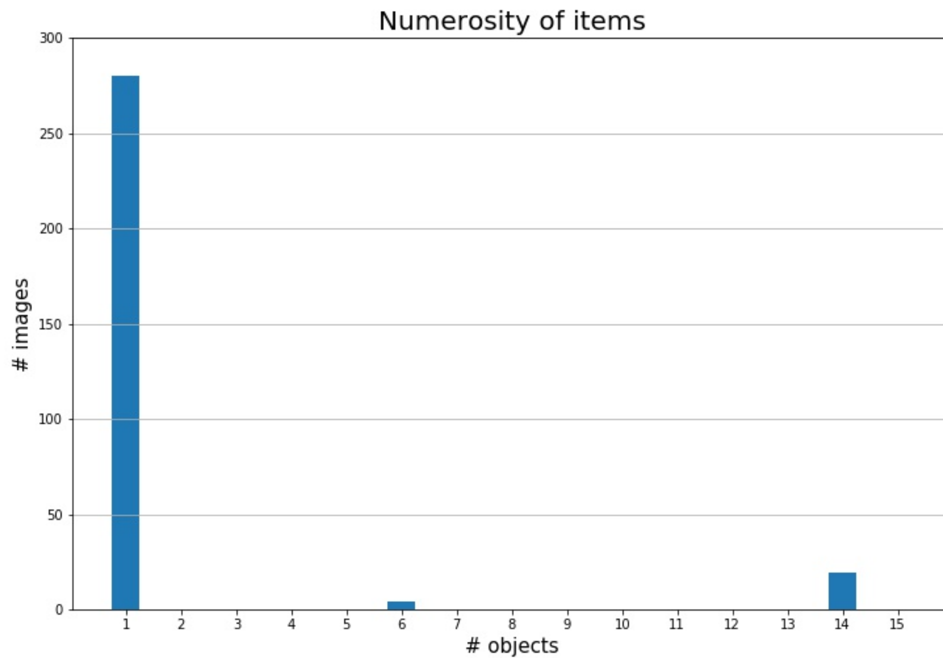



*Figure 5.12:* Distribution of the items per image in the dataset *Test β*.

*(a)*

*(b)*

*Figure 5.13:* Two images of dataset *Test β*.

## 5.3 V1: SSDLite MobileNet v2

The SSDLite model was selected instead of SSD because it is a lighter version, faster than it and with a lower number of parameters. The goal was to have a model which is suitable to be run on devices with low computation capabilities, such as smartphones and laptops without the GPU. Moreover, it was meant to be run on video streaming from a webcam, in order to build a real-time object detection system. According to the *GitHub* repository, this model has a processing speed of 27 ms per image and an accuracy of 22 COCO mAP, which place it at the top positions for what concern speed and at the end of the accuracy ranking. The model is pre-trained on the COCO 2014 training set and evaluated on the COCO2014 validation set. For this application, this model was fine-tuned on dataset *Reply* $2592 \times 1944$.

*(a)* Low contrast *(b)* High contrast

*Figure 5.14:* Results of models trained with dataset *Oracle*.

*(a)* Full image *(b)* Crop image

*Figure 5.15:* Results of models trained with dataset *Oracle*.

| **model** | SSDLite MobileNet v2 |
|---:|:---:|
| **# classes** | 16 |
| **dataset** | Reply |
| **train data** | 850 x2 images |
| **test data** | 150 images |
| **batch size** | 24 |
| **learning rate** | 0.004 |
| **training steps** | 75'000 |
| **image dimension** | 800 x 600 |
| **data augmentation** | Gaussian blurring |

*Figure 5.16:* List of parameters used to train the SSDLite MobileNet v2 model.

## 5.4 V2: Faster R-CNN ResNet50

As discussed in Chapter 2, the Faster R-CNN processing steps are different from those of SSD framework, and the result is a model with different evaluation metrics. A Faster R-CNN framework is always slower than an SSD, but we gain in accuracy and precision. This model is pre-trained and evaluated on the COCO2014 dataset, the same of SSDLite MobileNet *v2*, and its results reflect the general trend. This model has a processing speed of 89 ms per image, 3 times slower than SSDLite, but it has an mAP of 30. The dataset used to fine-tune the model is dataset *Reply* after

resizing all the images to $800 \times 600$.

| model | Faster R-CNN ResNet 50 |
|---|---|
| # classes | 16 |
| dataset | Reply |
| train data | 850 x2 images |
| test data | 150 images |
| batch size | 12 |
| learning rate | 0.0003 |
| training steps | 200'000 |
| image dimension | 800 x 600 |
| data augmentation | Gaussian blurring |

*Figure 5.17:* List of parameters used to train the Faster R-CNN ResNet 50 model.

## 5.5 V3: Faster R-CNN Inception v2

The last model used is similar to the previous, since it is again a Faster R-CNN framework, but this one uses another feature extraction network, the Inception. Again, also this model is pre-trained on the COCO2014 dataset and it is placed between the other two in terms of metrics. The processing speed is 58 ms per image, while the COCO mAP is 28. The dataset used for the fine-tuning phase is dataset *Reply* with images of $800 \times 600$.

| model | Faster R-CNN Inception v2 |
|---|---|
| # classes | 16 |
| dataset | Reply |
| train data | 850 x2 images |
| test data | 150 images |
| batch size | 12 |
| learning rate | 0.0002 |
| training steps | 75'000 |
| image dimension | 800 x 600 |
| data augmentation | Gaussian blurring |

*Figure 5.18:* List of parameters used to train the Faster R-CNN Inception v2 model.

# Chapter 6

# Results and Discussion

In this section, the results of each model will be illustrated. In order to compare the three models used, they are run on a dataset of new images (dataset *Test* $\alpha$ and dataset *Test* $\beta$) and the main important metrics are calculated. For each image in each dataset, the location and class id of each item have been manually annotated. Then, after running the object detection model on the image, the result is compared with the ground-truth in order to calculate those metrics. In particular, the metrics used are: FP (false positives), FN (false negatives), accuracy, IoU (intersection over union) and inference time per image.

The metrics are defined as follows:

- *FP*: it is the total number of false detection in the image
- *FN*: it is the total number of items that are not detected by the model. For this value, also the normalized version is calculated.
- *IoU*: it is the ratio between the total area of intersection of the ground-truth box and the box produced by the algorithm and the total area of union (figure [6.1])
- *Accuracy*: it is the ratio between the detected items over the total number of items present for that class
- *Inference time*: it is the time required by the model to produce the results

Each bounding box produced by the object detection algorithm comes together with a probability value, which indicates the confidence of the model about that detection. The higher the confidence, the higher is the probability that the detection is correct. However, the algorithm produces a lot of bounding box proposals and it is up to the user to discard those with a low confidence value in order to keep only the results that are expected to be correct. This selection is done using a threshold parameter: only the boxes with a confidence value higher than the threshold are drawn in the image.

The number of *FP* reflects the ability of the model to highlight only the true items in the image, and it is highly correlated with the threshold on the confidence level of the detection. Indeed, if this threshold is low (for example 15%), even a bounding box with a confidence of 20% is drawn. However, since every model

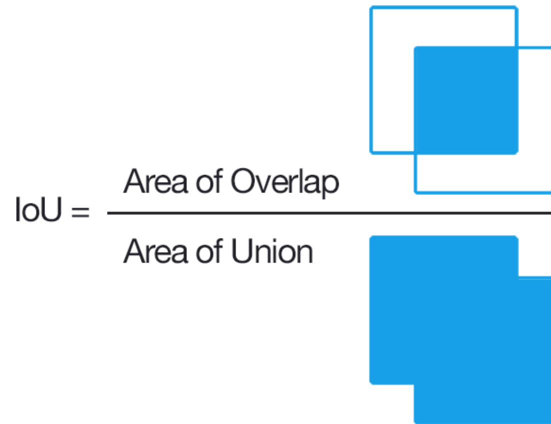



*Figure 6.1:* The Intersection over Union score is the ratio between the area of intersection and the total area of union of the two bounding boxes.

used is able to label the correct bounding boxes with a very high confidence (over 85%), all those detection between 15% and 85% are likely to be wrong. Hence, the threshold should be selected *a posteriori*, after having analyzed the confidence level of the correct classifications. If the confidence threshold is low, there will be an high number of *FP*, while with an higher threshold the number of *FP* is reduced. In this project, the detection threshold has been fixed to 50%.

The *FN* (and the normalized *FN*) value reflects the ability of the model to detect all the items in the image. A missing detection can be originated by two causes: the bounding box around that item has a confidence level under the threshold or the algorithm has not produced any bounding box around it. In the first case, it is sufficient to tune the confidence threshold in order to minimize the *FN*, while in the second case the error cannot be restored and it is due to the performances of the model.

Since in this algorithm the image is given to the object detection module and then all the other elaborations are made on top of its output, it is important to ensure that all items are detected: indeed, if an item is missed, there is no way to detect it anymore and it will be labelled as a *FN*. On the other hand, if all items are detected and even some *FP* are picked out, it is possible to discard them in subsequent steps by, for example, analyzing their position or by using another classifier on top of the object detection model. Hence, the goal of the application is to minimize the number of *FN* even without caring about the number of *FP*, since the latter can be adjusted at a later time.

The $IoU$ value indicates the ability of the model to draw the bounding boxes at the correct location and with the correct dimension. In order to be compared, the two bounding boxes need to have the same class id. If the $IoU$ is high (over another threshold) then the localization is correct, while if it is low there is a localization error. An important aspect to point out is that if there are two items of the same class that are adjacent in the shell, changes are that the two identified bounding boxes are slightly overlapping and so the $IoU$ is different from 0 and very low. In this case, there would be 2 localization errors (one for each couple item/bbox of the

other item) even if it is all correct. In this application, since there is only one item for each class, this problem does not subsist, but it is important to consider it for future development. For this project, the $IoU$ threshold has been fixed to 90%.

The accuracy is calculated over the entire dataset, since it is not possible to calculate it for every image. Indeed, since there is just one item for each class, the accuracy per image can only be 0 in case of missing detection or 1 in case of correct detection. Hence, the accuracy is defined as the total number of detected items of each class over the total number of items actually present.

The inference time is the time required to the model to produce the output results on the image. Of course, this time is to be considered only for comparing the different frameworks and the different image sizes. The results that are illustrated below are calculated using a laptop computer with no GPU installed.

## 6.1 V1: SSDLite Mobilenet v2

The SSDLite model is the one with the lower inference time per image. Its configuration allows to use large images for training (up to $2592 \times 1944$) and so the model can work also with high resolution inputs. However, this has the counter effect that the model is not very precise in detecting the items. The low number of $FP$ is balanced by the relatively high number of $FN$: the model is not able to graphically catch the items, so it "forgets" some of them; however, this fact comes with the advantage of having also low $FP$. Thanks to the low number of parameters, this model is suitable to be run on small portable devices such as smartphones and surveillance cameras, at the price of sacrifice some accuracy.

The most important analysis that can be done is on the accuracy of the model for each class id. In the following graphics it is shown the fraction of items localized over the total number of items actually present in the dataset. The figure [6.2] shows the accuracy results with images with a resolution similar to those used for the training. In figure [6.3] and [6.4] there are the results with the same images but resized. In general, those results are good but far from optimum, since almost 2/3 of the items are not detected by the model. The class with the lower accuracy is *"mademoiselle"* with a score of just 23%. It can be noticed that the performances slightly increases with low quality images, but these improvements are not remarkable: with this considerations, it is clear that the best practice will be to use low quality image to reduce the inference time, since the results will have almost the same accuracy. On the dataset *v3*, this model is less accurate, and there are classes like *"croyance"*, *"lonia"* and *"mademoiselle"* with a poor accuracy score (under 5%) ad so they are almost never detected by the model (figures [6.5], [6.6] and [6.7]). Since the classes *"croyance"* and *"mademoiselle"* are two of the less represented in the dataset *v1* used for the training, these results can be explained by saying that the model didn't have enough images to memorize the graphical structure of them. In particular, there are 120 images of *"croyance"* and 175 images of *"mademoiselle"*, while all the other items are represented by over 200 images. Except for the class *"ambre"* which is always detected, the scores for the other classes are around 50%.

It is also possible to exploit the ability of the model when the input image has different resolutions. In particular, the following graphics represent the number of

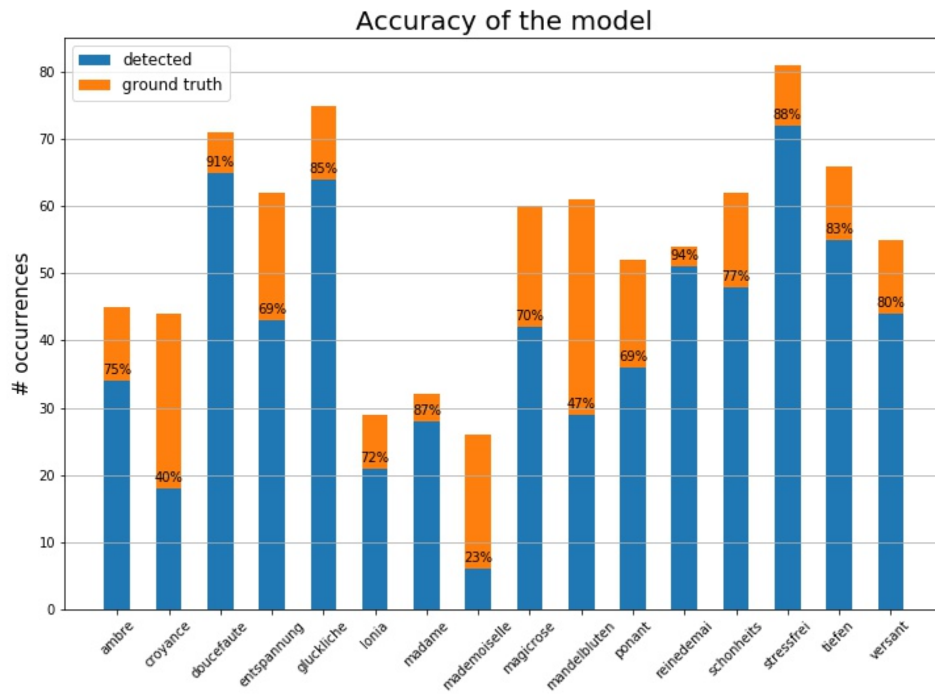



*Figure 6.2:* Accuracy of the SSDLite MobileNet v2 model on dataset *Test* $\alpha$ with images of $2292 \times 1719$ resolution.

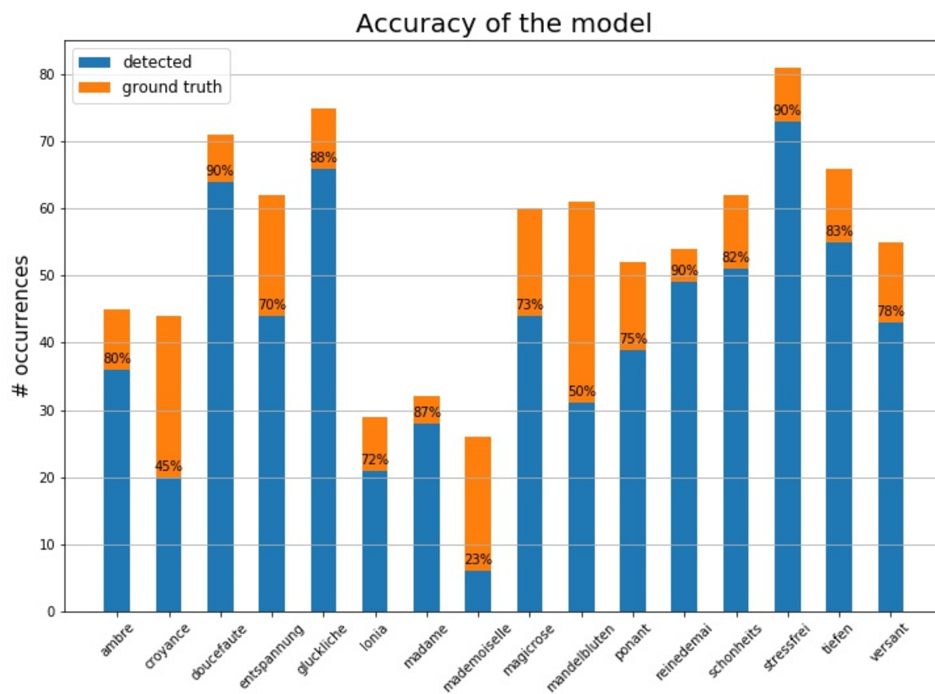



*Figure 6.3:* Accuracy of the SSDLite MobileNet v2 model on dataset *Test* $\alpha$ with images of $1024 \times 768$ resolution.

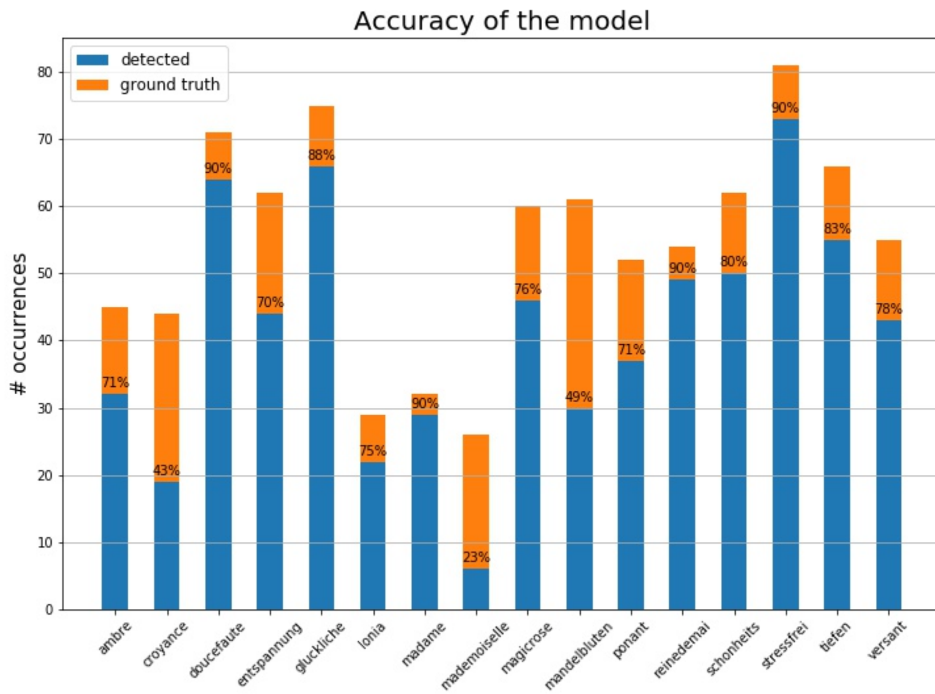



*Figure 6.4:* Accuracy of the SSDLite MobileNet v2 model on dataset *Test* $\alpha$ with images of $800 \times 600$ resolution.

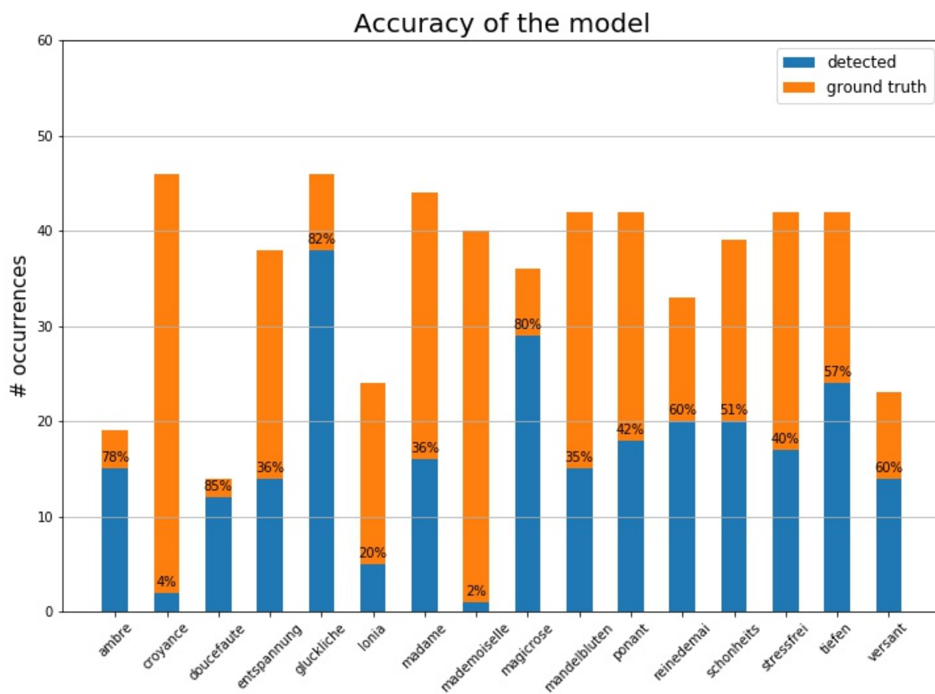



*Figure 6.5:* Accuracy of the SSDLite MobileNet v2 model on dataset *Test* $\beta$ with images of $4032 \times 3024$ resolution.

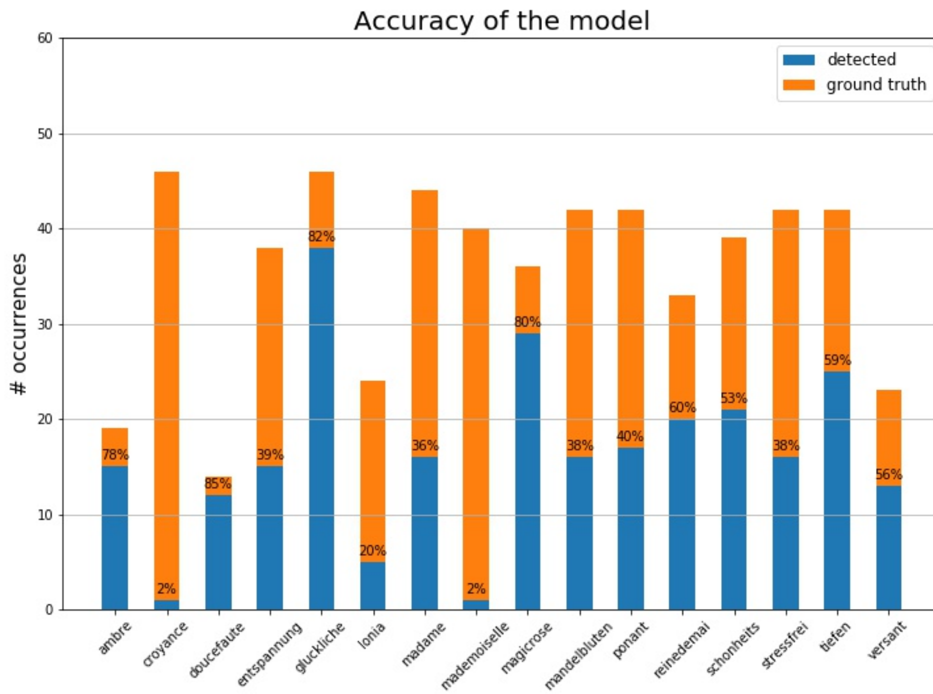



*Figure 6.6:* Accuracy of the SSDLite MobileNet v2 model on dataset *Test* $\beta$ with images of 2297 $\times$ 1292 resolution.

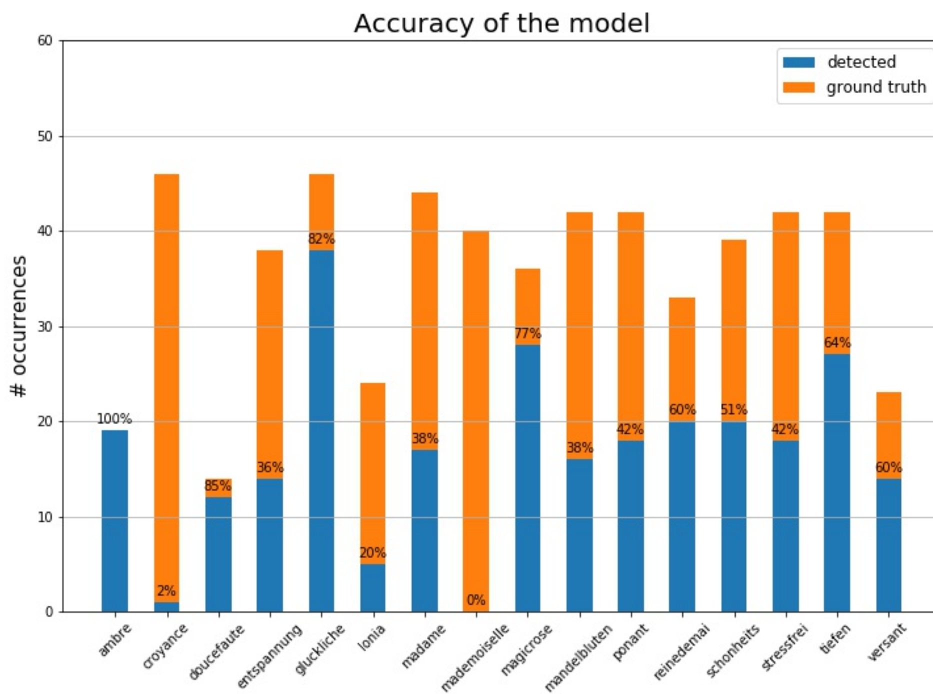



*Figure 6.7:* Accuracy of the SSDLite MobileNet v2 model on dataset *Test* $\beta$ with images of 800 $\times$ 450 resolution.

*FP*, normalized *FN* and localization errors when compared to the image resolution. The *FP* value is obtained by averaging the number of *FP* for each image, while the normalized *FN* is obtained in the same way, after having normalized the *FN* for each image. Also the localization errors are normalized by the number of items in the image and then averaged over the all dataset. In the figure [6.8] these results are shown.

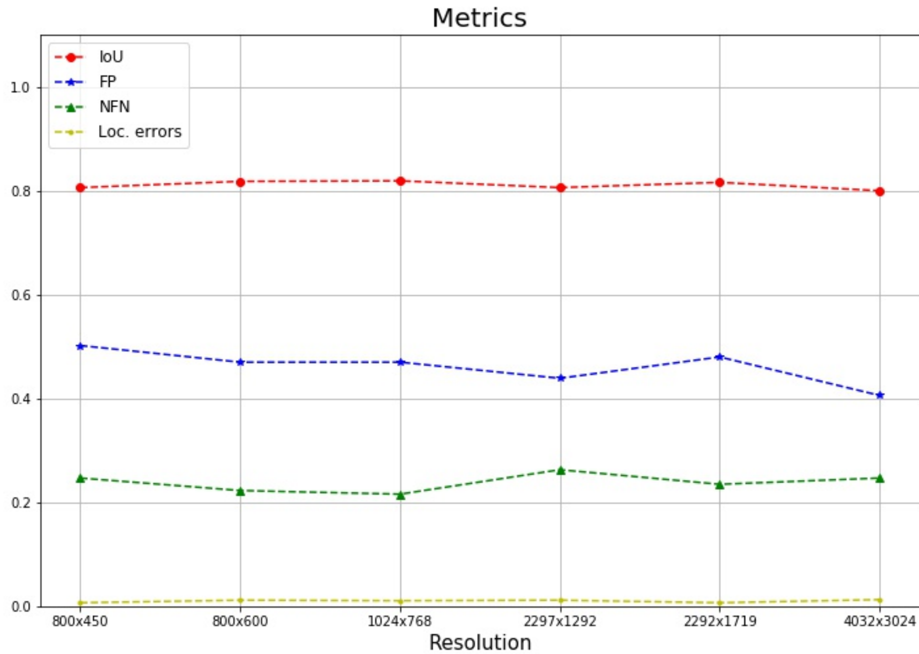



*Figure 6.8:* FP, NFN, IoU and localization errors for different image resolutions for the SSDLite MobileNet v2 model.

The main consideration that emerges from this analysis is that the SSDLite model is quite unpredictable to correctly identify and locate items in the image. Indeed, there are classes for which the totality of occurrences are detected, while there are also classes with a very low accuracy. For example, the *"doucefaute"* class always obtains an accuracy value near 0.8, while the class *"mademoiselle"* has very poor results. Also, the *FP* is always near 0.5, so there are not many cases where the pruning algorithm has to remove false detection. The *NFN* is even lower, around 0.2, but here it is useful to analyze the *FN* values without normalizing: in the dataset *Test* $\beta$, where there the majority of images have just one item, the *FN* is around 0.8, so there is 80% of probability that the algorithm will not be able to detect the item in the image. This is reflected by very low accuracy results in the graphics above. In a real scenario, if the image contains an high number of items and someone is not detected by the algorithm, then it is not a big problem but, if in the image there is just one item and that one is not detected, the application is far from optimum. This is what happens with dataset *Test* $\alpha$, where images have more items in the shelf: the *FN* is around 2, so we have 2 missed detection in the average

case, and this is a more acceptable result. The normalized localization error value is very low, around 0.01 so, when there is a detection, this is at the right position. This fact is reflected also by the $IoU$ indicator, which is always around 0.8.

For what concerns the inference time, the analysis is made at the end with the data from the other two models. An anticipation is that this model is thought to sacrifice the accuracy in order to be very fast in processing an image, so it is the model with the lowest inference time.

In figures [6.9] and [6.10] there are some examples of the output of this model.

*(a)* *(b)*

*Figure 6.9:* SSDLite output on dataset *Test* $\alpha$.

*(a)* *(b)*

*Figure 6.10:* SSDLite output on dataset *Test* $\beta$.

## 6.2 V2: Faster R-CNN ResNet50

The Faster R-CNN ResNet model is the second model considered for this application. It uses an higher number of weights, compared to the SSDLite model, and this has two effects: on one hand the model is more precise and the accuracy scores increases; on the other hand, the inference time for processing one image is higher and also the model requires more space to be run on the device. The accuracy scores are shown in the following graphics. In figures [6.11], [6.12] and [6.13] there are the results for the dataset *v2*, where the number of items per image is around 6/7. It is clear that the accuracy scores increases with the respect of the previous model: here, the

majority of classes are always detected and the few mistakes are due to illumination artifacts and partial occlusions. However, for all classes the accuracy score is above 85%, which is a great result. Moreover, when reducing the image resolution, these accuracy scores increase up to 90% even for the class *"mandelbluten"*, which is the Achille's heel of this model. These results are quite the same for dataset *v3* (figures [6.14], [6.15] and [6.16]), with scores that are above 90% for all the classes except for the class *"mademoiselle"* for which the model is not able to go beyond 52% of accuracy even with the images with lowest quality.

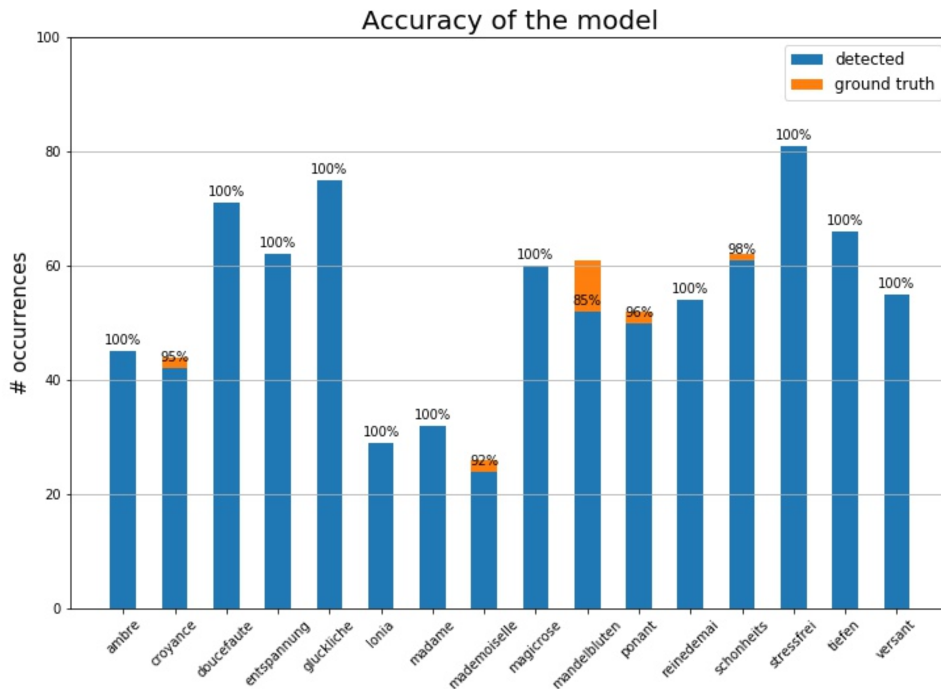



*Figure 6.11:* Accuracy of the Faster R-CNN ResNet50 model on dataset *Test* $\alpha$ with images of $2292 \times 1719$ resolution.

The results in figure [6.17] show that the *NFN* value is considerably reduced to 0.01 for dataset *Test* $\alpha$ and 0.04 for dataset *Test* $\beta$. Moreover, the *FN* value without normalization is under 1 for both the datasets, so the probability to have a false negative is very low. On the other hand, as a counter effect, the *FP* value increases. This is because the model's weights have learn the graphical features of the items in such a precise way that it "sees" items also in other positions in the image, obviously with a lower confidence score. The reason for which the number of *FP* in higher for dataset *Test* $\beta$ is that those images were acquired with some coloured objects in the background that are interpreted by the model as items to detect, while dataset *Test* $\alpha$ has always a flat background. These performances are accompanied with an higher localization precision, so there are 0 localization errors and the *IoU* is a bit higher than those of SSDLite.

The inference time per image, for this model, is higher than SSDLite for two reasons: the first is intrinsic in the framework, since the Faster R-CNN module needs more computational effort than SSD; the second is that the ResNet50 feature

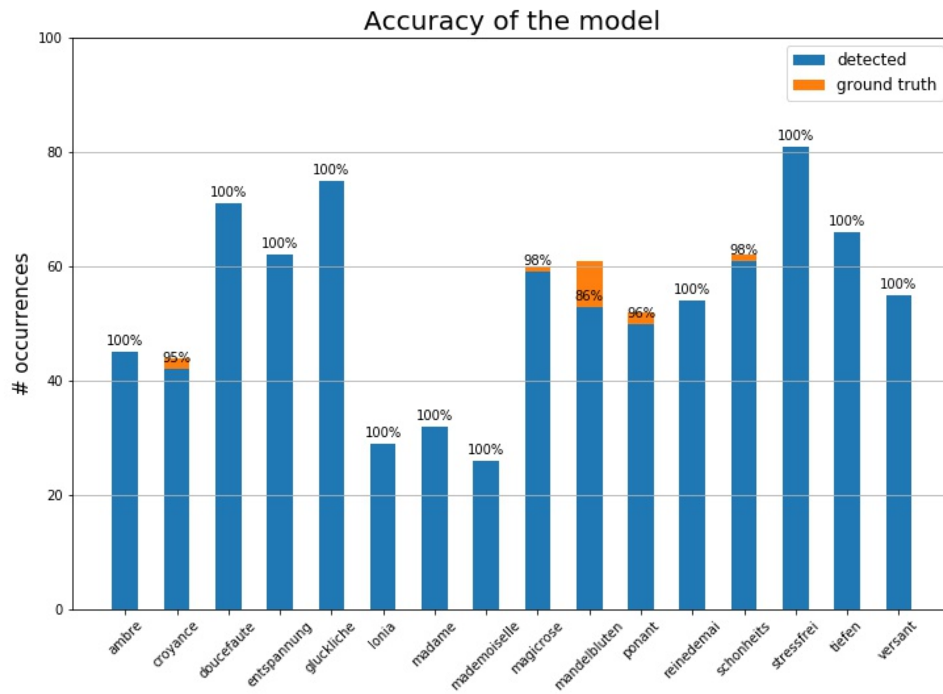



*Figure 6.12:* Accuracy of the Faster R-CNN ResNet50 model on dataset *Test* $\alpha$ with images of $1024 \times 768$ resolution.

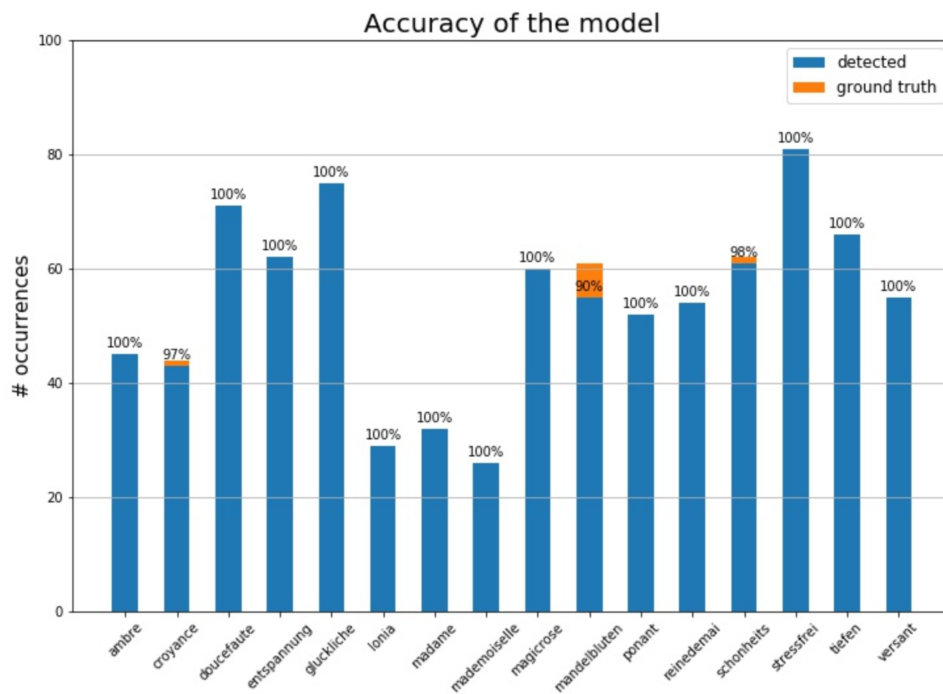



*Figure 6.13:* Accuracy of the Faster R-CNN ResNet50 model on dataset *Test* $\alpha$ with images of $800 \times 600$ resolution.

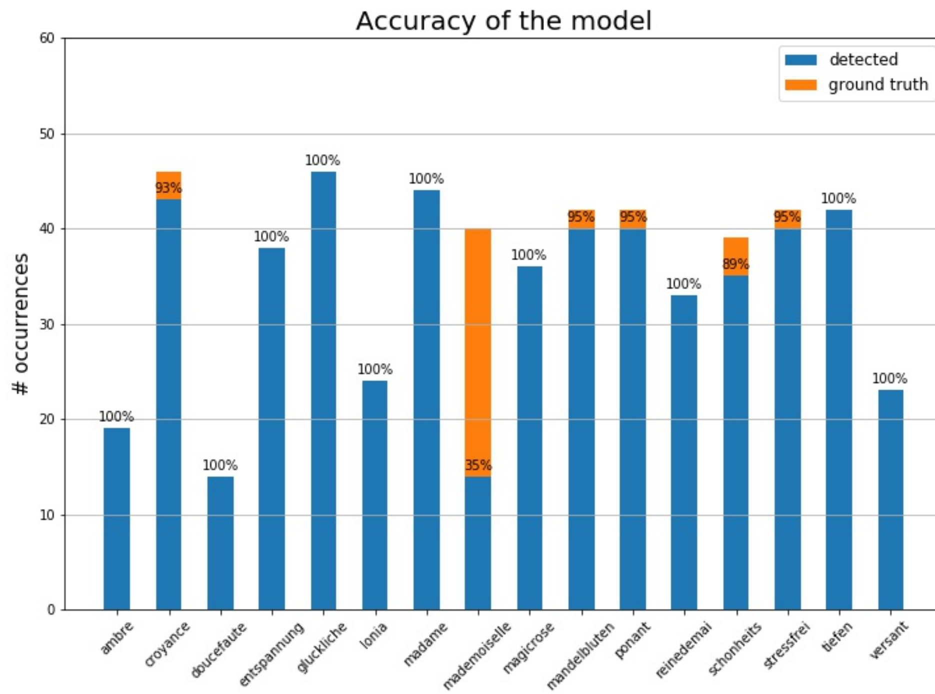



*Figure 6.14:* Accuracy of the Faster R-CNN ResNet50 model on dataset *Test β* with images of 4032 × 3024 resolution.

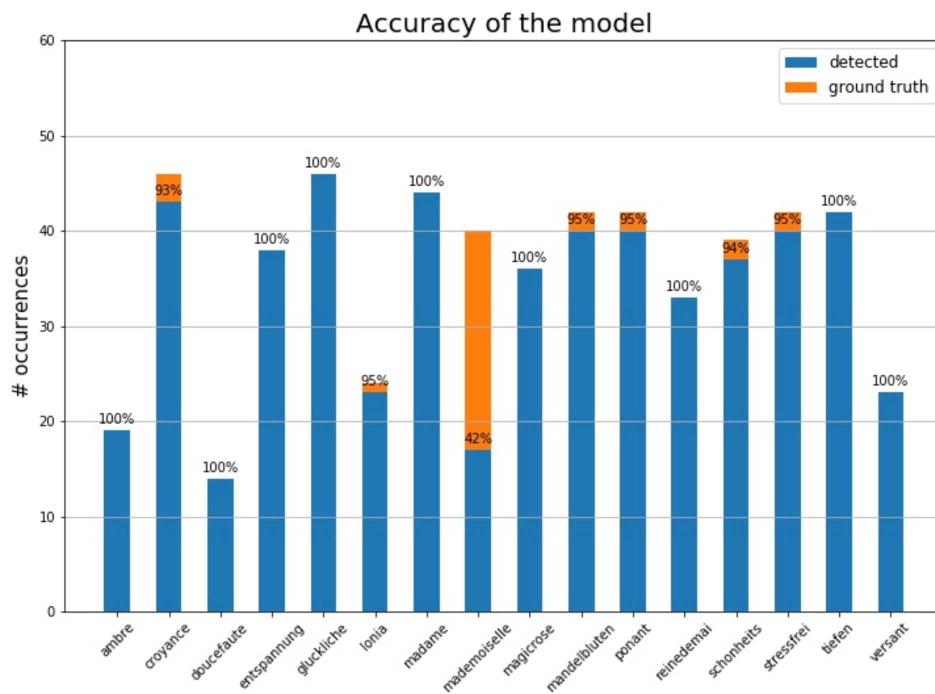



*Figure 6.15:* Accuracy of the Faster R-CNN ResNet50 model on dataset *Test β* with images of 2297 × 1292 resolution.

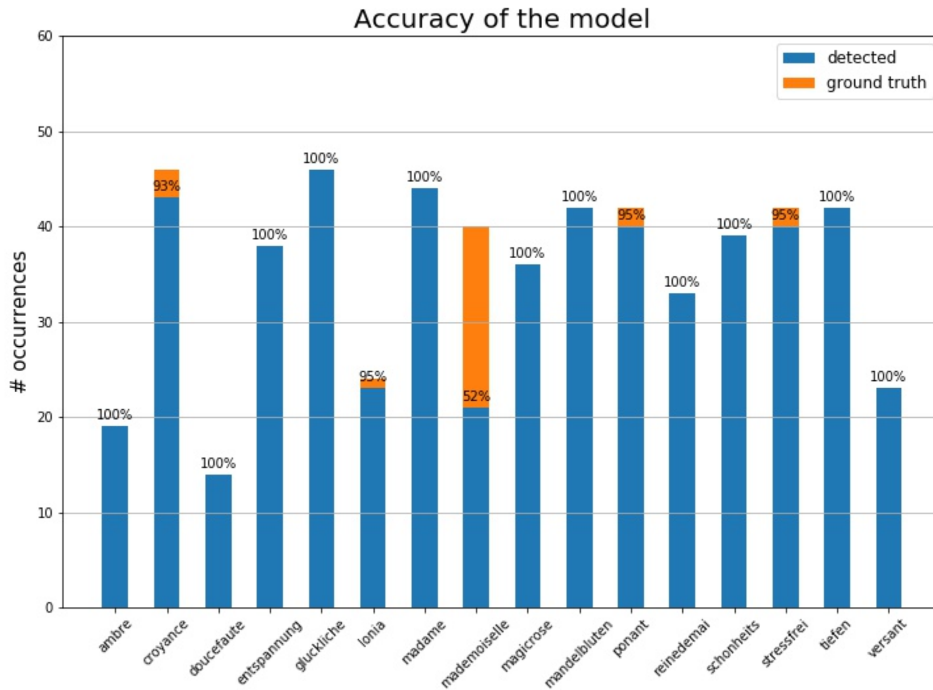



*Figure 6.16:* Accuracy of the Faster R-CNN ResNet50 model on dataset *Test* $\beta$ with images of $800 \times 450$ resolution.

extraction network is a very deep architecture (50 layers).

In figures [6.18] and [6.19] there are some examples of the output of this model.

## 6.3 V3: Faster R-CNN Inception v2

The third model used in this application is again a Faster R-CNN, but with a different feature extraction architecture, the InceptionNet. This model is used to compare the same framework with two different backbone networks. The accuracy scores are comparable to the one of the previous model and are illustrated in the graphics below. For dataset *Test* $\alpha$ (figures [6.20], [6.21] and [6.22]), the accuracy is above 92% for all classes, with many of them with a score of 100%. In particular, this model with $1024 \times 768$ images gave the best results of the application, with just 3 classes that are not perfectly recognized, but with significantly high scores: 93%, 96% and 97%. Also for dataset *Test* $\beta$ the accuracy scores are very high, with almost all classes at 100%; however, the model failed to recognize the class *"entspannung"* with all the resolutions, with results of 55%, 47% and 73%.

One of the main drawbacks that emerges from the analysis of the metrics (figure [6.26]) is that the number of *FP* for dataset *Test* $\beta$ is around 3 for every image. This indicates that this model is prone to errors when detecting graphical features. For the other dataset, the results are comparable with the other model. However, the *FN* score is close to 0, as for the previous model, so the majority of items are always detected by the model. In the analysis, no localization errors are registered on both the datasets, and the $IoU$ score is always above 80%, as for the other models.

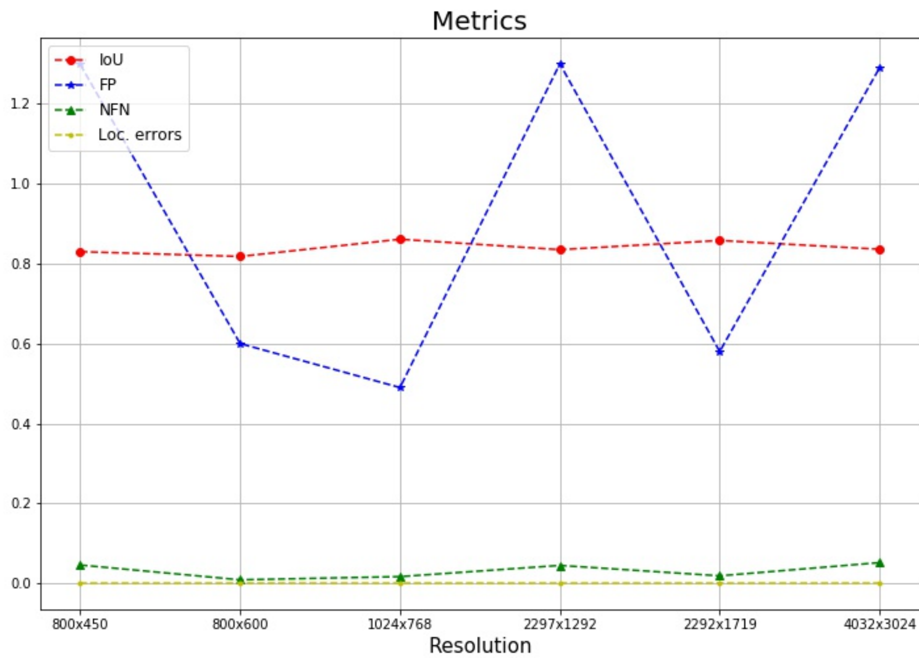



*Figure 6.17:* FP, NFN, IoU and localization errors with different image resolutions for the Faster R-CNN ResNet50 model. The peaks in the FP are due to the variety of the two datasets.

*(a)* *(b)*

*Figure 6.18:* Faster R-CNN ResNet output on dataset *Test* $\alpha$.

In figures [6.27] and [6.28] there are some examples of the output of this model.

The inference times of the three models are compared in figure [6.29]. As expected, the SSDLite model has always the lower value, while the Faster R-CNN model requires more time to compute the results. Furthermore, the inference time increases with the image resolution, since there are more pixel to analyze. In the best case, the time required to process one $800 \times 450$ image is above 1 second,

(a) (b)

*Figure 6.19:* Faster R-CNN ResNet output on dataset *Test* $\beta$.

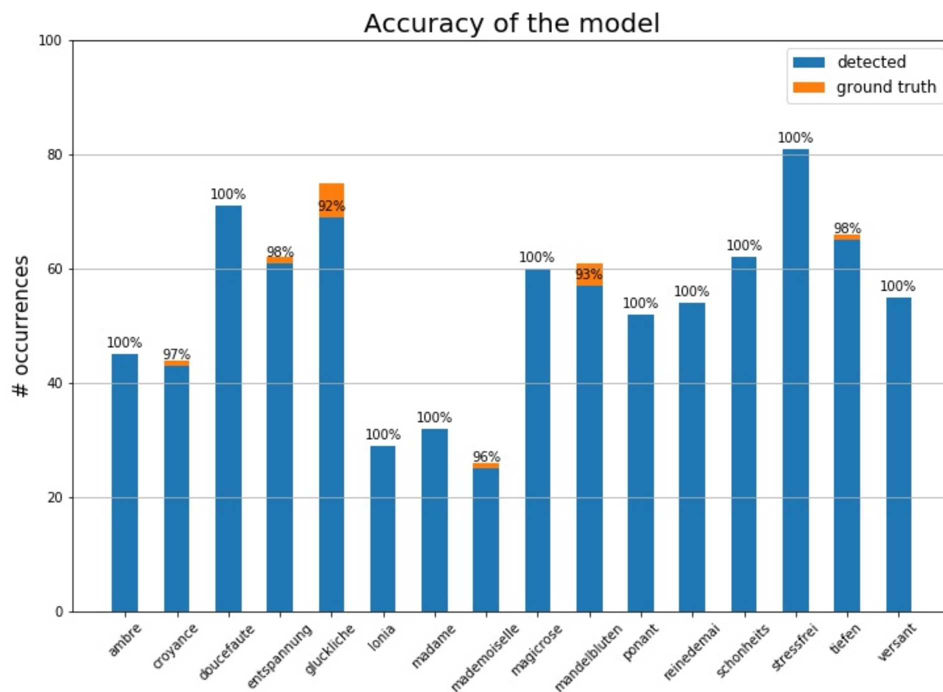



*Figure 6.20:* Accuracy of the Faster R-CNN Inception v2 model on dataset *Test* $\alpha$ with images of $2292 \times 1719$ resolution.

so none of this models is suitable for building a real-time object detection system. However, it is important to remember that these data are calculated on a laptop without any GPU, so they can be substantially reduced by using a different hardware architecture. The time required to the system to process a 4K image goes from the 10 seconds for the SSDLite model to 15.6 seconds for the Faster R-CNN with ResNet50. This interval, besides being prohibitive for a real-time system, is also restrictive for any other application which uses this image. However, as said before, changes are that using a highly performing GPU will reduce this delay even under 1 second. On the other hand, the real-time systems usually work with low quality images (for example surveillance cameras), so there's no need to process 4K images.

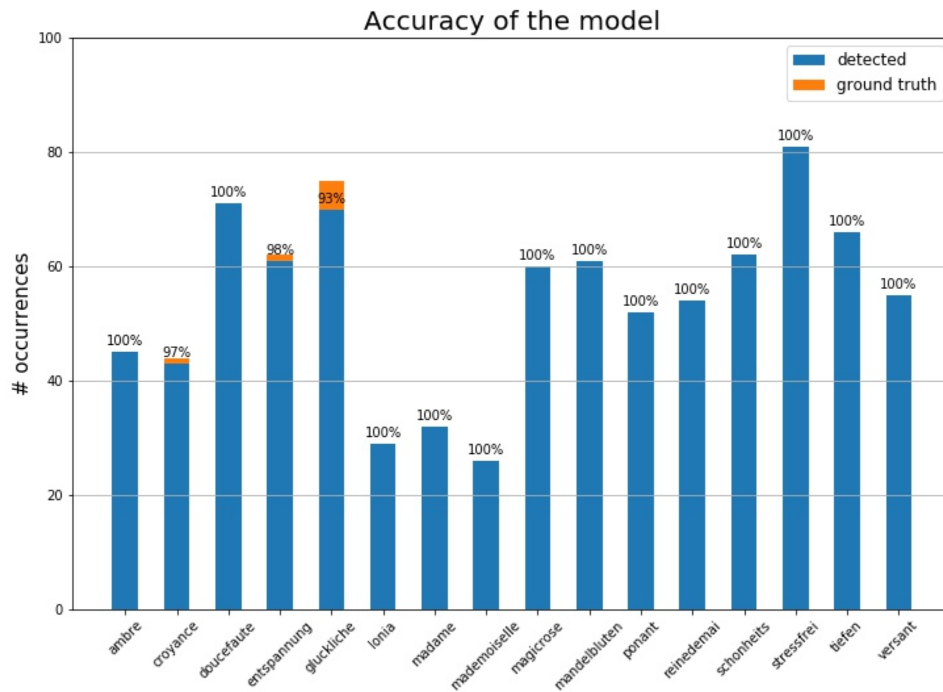



*Figure 6.21:* Accuracy of the Faster R-CNN Inception v2 model on dataset *Test* $\alpha$ with images of $1024 \times 768$ resolution.

Finally, it is important to point out that the resizing of an image is an instantaneous operation and, hence, it is possible to create a system that receives in input images of any resolution and then resizes them to the same (low) quality before feeding the object detection model.

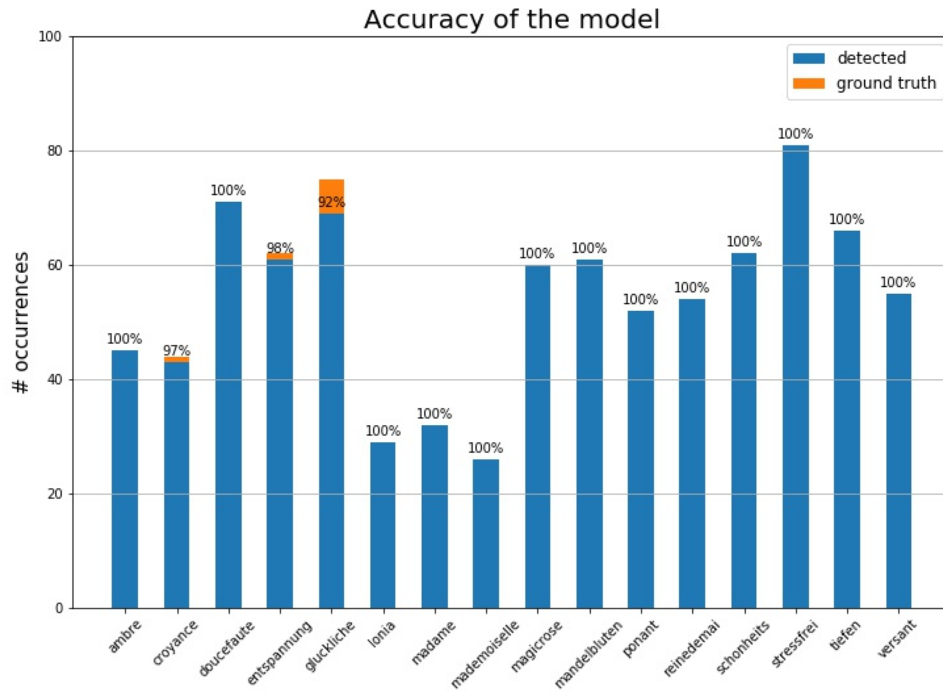



*Figure 6.22:* Accuracy of the Faster R-CNN Inception v2 model on dataset *Test* $\alpha$ with images of $800 \times 600$ resolution.

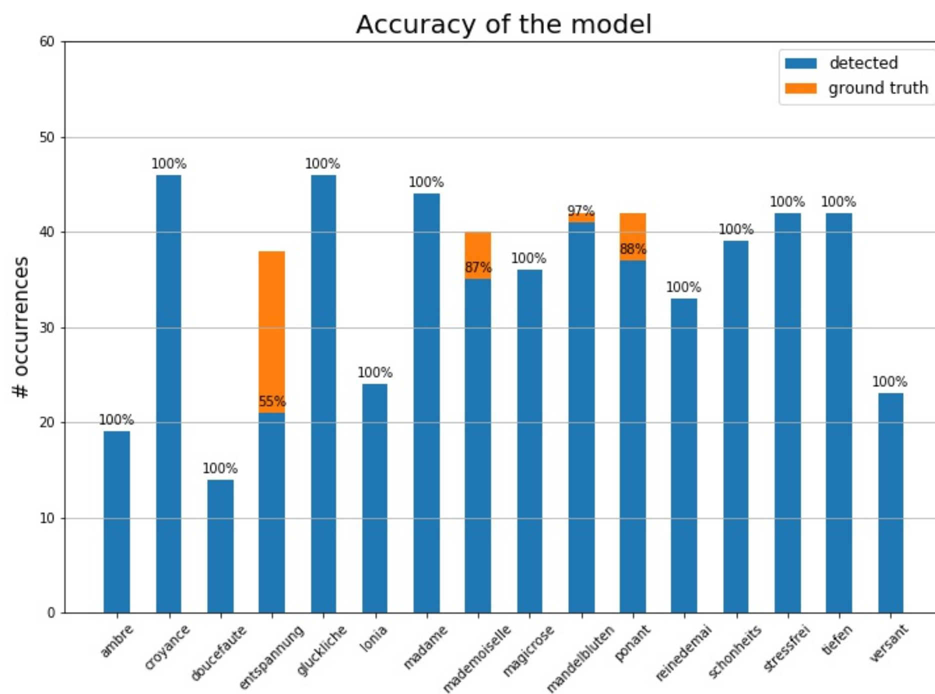



*Figure 6.23:* Accuracy of the Faster R-CNN Inception v2 model on dataset *Test* $\beta$ with images of $4032 \times 3024$ resolution.

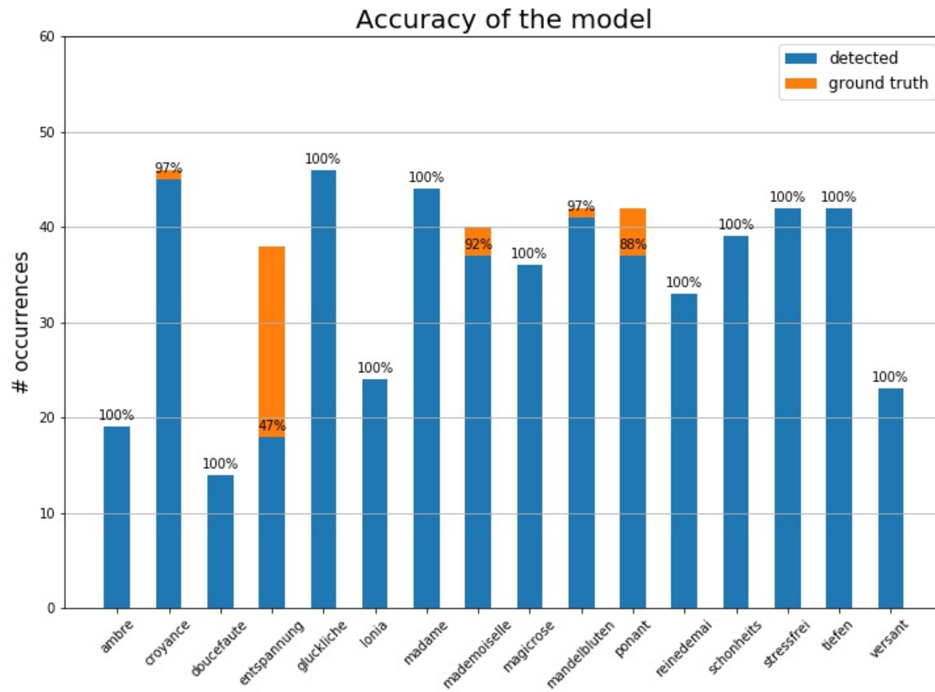



*Figure 6.24:* Accuracy of the Faster R-CNN Inception v2 model on dataset *Test* $\beta$ with images of $2297 \times 1292$ resolution.

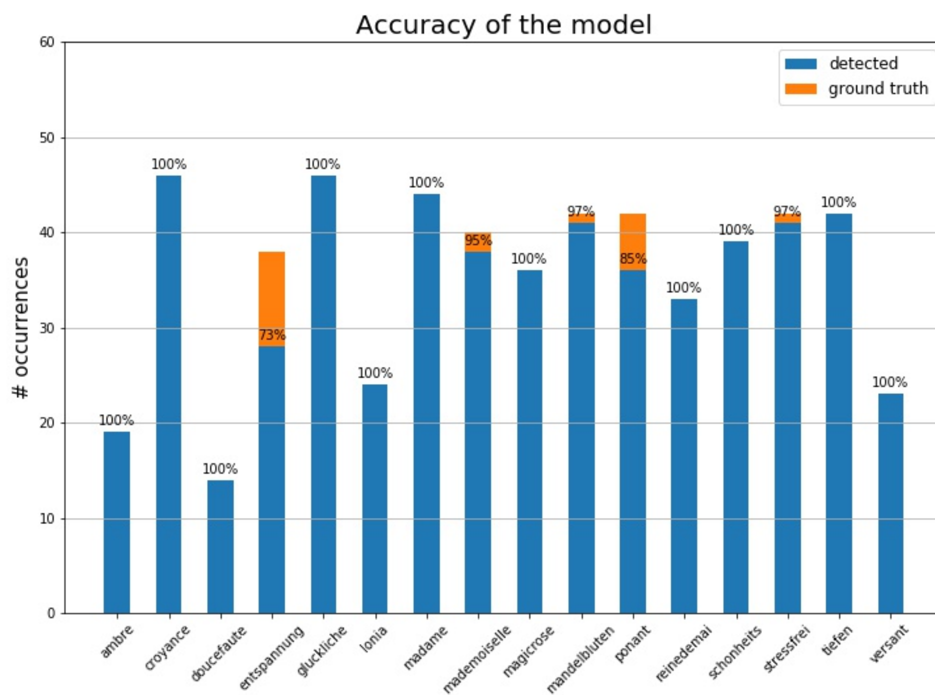



*Figure 6.25:* Accuracy of the Faster R-CNN Inception v2 model on dataset *Test* $\beta$ with images of $800 \times 450$ resolution.

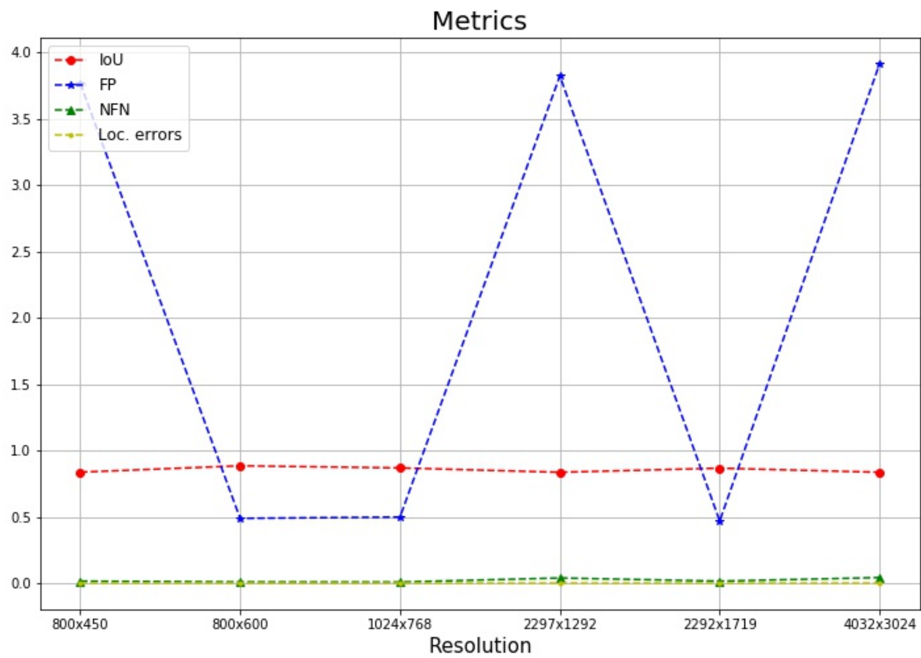



*Figure 6.26:* FP, NFN, IoU and localization errors with different image resolutions for the Faster R-CNN Inception v2 model. The peaks in the FP are due to the variety of the two datasets.

*(a)* *(b)*

*Figure 6.27:* Faster R-CNN Inception output on dataset *Test $\alpha$.*

(a) (b)

*Figure 6.28:* Faster R-CNN Inception output on dataset *Test β*.

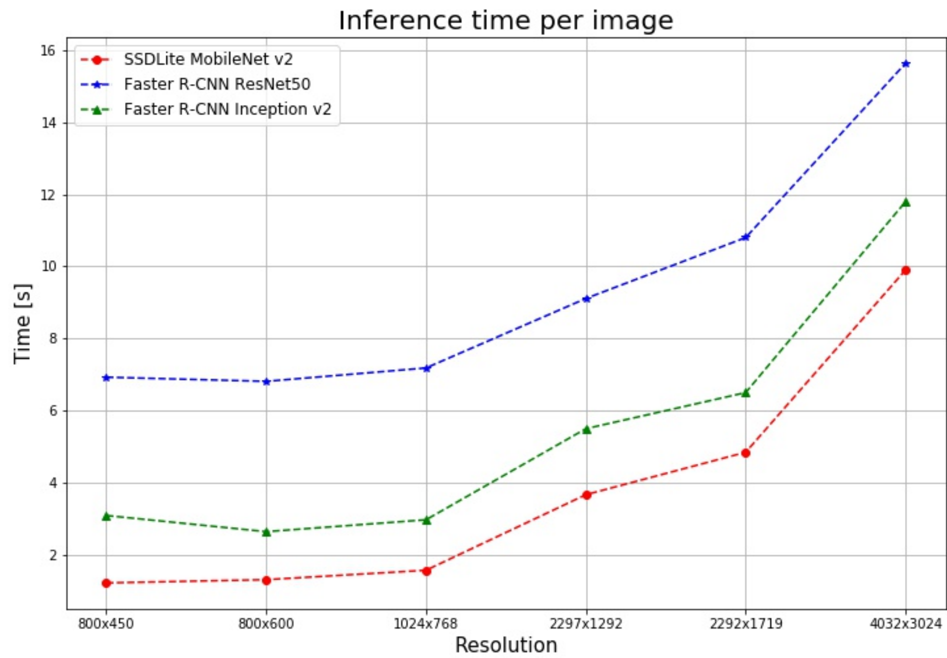



*Figure 6.29:* Time required by the model to produce the output given the image resolution.

# Chapter 7

# Applications

In this chapter, two applications of the object detection system are presented. Indeed, as discussed in chapter 3, the object detection module is encapsulated inside a more complete system which is thought to be used in real industrial applications. These are self-checkout, marketing analysis on the stores or customers' advisor on products. Both the application has been fully developed using the Flask library in Python.

## 7.1 Oracle - Proxima Smart City

The Oracle Proxima Smart City (figure [7.1]) is a project made my Oracle whose aim is to show the possibilities of creating a city where the main services are automated. The Proxima is built with LEGO and it is powered by a set of sensors and open source gateways (Raspberry PI, Arduino) that are connected to the Oracle Cloud. In the demo, it is possible to observe the typical day of a citizen in the smart city and how the automated services can help him. For example, it is possible to adjust the light of the street lamps according to the presence of vehicles in order to save energy; moreover, is it possible to ask for park availability to a Digital Assistant which search for free parks using machine learning techniques. These services are thought in two ways: the first is to help the citizens' life by automating some processes; the second is to make an intelligence surveillance over the environment in order to save energy and to increase the standard of living.

An idea for the Oracle Proxima Smart City is an automated self-checkout system which will help customers inside a store to reduce the time spent shopping. Ideally, the system should be able to track the activity of any customer in real-time and to produce a virtual ticket of his/her purchases. On the other hand, if the customer acquires a photo of the goods in the shelf, the system could enrich that image with information on the product and so on. All these services can be run only if the system is firstly able to detect and classify the items on the shelf and return their position. On top of that, any kind of elaborations can be made.

In this thesis, the object detection module described in the previous chapters is used to detect and classify the items on the shelf starting from an image of it. Since the acquisition of the photo is up to the customer, the system needs a front-end part (an App) that can be used by people to send images to the detection module, which is running on a remote server. In order to perform this, a Web Service has

*Figure 7.1:* Representation of the Oracle Proxima Smart City. Source of the image: *https://www.oracle.com/it/corporate/pressrelease/italia-startup-milan-digital-week-2019-03-07.html*

been developed.

From the front-end perspective, the user is asked to acquire a photo and to select the store ID and the shelf ID from a list in order to identify the specific shelf. Indeed, the shelves could have different sizes and structures. Then, after acquiring the picture, this information is sent to the server which feeds the object detection model with the image and uses the shelf identifier to retrieve the structure and to place the items at the correct position. For example, if the shelf is identified by the code *S001* and is composed by two trays (upper and lower), then the system will place the items either on one of these trays depending on the position calculated by the object detection module.

The image is sent to the server in the payload of an HTTP request, while the other information are encapsulated inside a JSON file which is automatically produced by the system (figure [7.2]). The JSON file also contains a field with the image name. The JSON file request is serialized into a Python class named *ProximaRequest* whose attributes are the JSON fields:

```
{
        "meta": "aaa",
        "storeID": "001",
        "shelfID": "S002",
        "imageName": "test013.jpg"
}
```

*Figure 7.2:* The JSON file produced by as an HTTP request and sent to the server, together with the image to analyze.

- *meta* is an unassigned field
- *storeID* is the identifier of the store
- *shelfID* is the identifier of the shelf inside the store
- *imageName* is the name of the loaded image

When the server receives that request, it checks both the *imageName*, that has to be not empty and the image extension, that should be in a list of allowed (*jpg*, *jpeg*). If these tests don't succeed, an *ErrorResponse* class creates an error message and sends it to the user deserializing it into a JSON file in order to be readable. Otherwise, the server saves the image inside a specific directory and creates another object of class *ProximaResponse* that somehow extends the *ProximaRequest* class and that will be populated with the information from the object detection module.
At this point, an object of class *Detection* is created by passing the absolute path of the image saved in the server. This method has a private attribute which is called *detect_info* that is an array where the results of the detection are saved (figure [7.3]). The coordinates of the center of the bounding box are calculated so that the items can be ordered from left to right in the image.

| class_id | x_min | y_min | x_max | y_max | accuracy | x_c | y_c |
|---|---|---|---|---|---|---|---|
| 2 | 100 | 80 | 200 | 120 | 0.96 | 150 | 100 |
| 3 | 120 | 130 | 200 | 250 | 0.86 | 160 | 190 |
| 9 | 200 | 130 | 300 | 250 | 0.98 | 250 | 190 |
| 15 | 200 | 60 | 350 | 125 | 0.91 | 275 | 93 |
| 11 | 360 | 75 | 500 | 150 | 0.89 | 430 | 113 |
| 7 | 355 | 160 | 510 | 250 | 0.90 | 432 | 205 |

*Figure 7.3:* A representation of the array that contains the information extracted by the object detection module. The last two columns (*x_c* and *y_c*) are calculated from the other coordinates and they are the center of the bounding box.

The last step is to populate the attributes of the class *ProximaResponse* with the detection information. In particular, the response contains the same attributes of the request (*meta*, *storeID*, *selfID* and *imageName*) but is enriched with two others:

- *numProd* is the number of items detected by the algorithm
- *products* is a list where each element is composed by:
  - *name* of the detected item
  - *bbox* as a list of the 4 rectangle coordinates: [*xmin*, *xmax*, *ymin*, *ymax*]

This response is deserialized to a JSON file and returned to the user (figure [7.4]). There are several ways to return the information to the user: for example, it is possible to return the original image with the bounding box and the class names. However, this format has been discarded to reduce the response time of the system. Indeed, if the object detection module runs on a server with high computational

capabilities, the bottleneck of the entire system is the transmission of the image from the client to the server. If the server has to process the image and then return it to the client, the time required for the response would have been doubled. However, since drawing the bounding box on an image could be done locally, the server just sends to the client the coordinates in a JSON file, which is softer.


```
{
        "meta": "aaa",
        "storeID": "001",
        "shelfID": "S002",
        "imageName": "test013.jpg",
        "numProd": 2,
        "products": [
                {
                        "name": "ambre",
                        "bbox": {
                                "xmin": 20,
                                "xmax": 202,
                                "ymin": 56,
                                "ymax": 103
                        }
                },
                {
                        "name": "croyance",
                        "bbox": {
                                "xmin": 252,
                                "xmax": 343,
                                "ymin": 45,
                                "ymax": 98
                        }
                }
        ]
}
```


*Figure 7.4:* The JSON file produced by the server and sent back the client.

When the file is correctly received by the client, then it is up to the front-end to present it to the user, depending on the application. This last part is out of the scope of this thesis, but all the possible applications just need the information on the JSON file.

## 7.2 Reply - Automatic Products Recognition

The second application of the object detection module, in this thesis, is an automatic product recognition for marketing analysis. Indeed, when the suppliers sell their goods to the stores, they virtually "buy" the shelves to place their products. Once the trade agreements are concluded, the suppliers need to check whether their goods are placed in the correct way. On the other hand, the store manager can carry out some marketing analysis on the number of sales as a function of the product position. For example, people that buy chips are likely to buy also ketchup if it is placed close to them as against it is placed on another shelf. Another example, two companies that produce the same good are more interested in placing them as far as possible. On the other side, the store manager has the interest in placing similar goods in

the same sector to facilitate the customers. Hence, for both the parts, a system that automatically detects the goods in the shelves and highlights also the empty positions would be of great interest.

In this case, the real-time system is not needed, since the analysis is made on images acquired by the analysts. The system uses a Web Service where the object detection module is at the server side, while at the client side there is an App that allows for loading an image with some information. In the real scenario, since the post processing operations are done by the server, the response given to the client is less detailed. Once the client has sent the image to the server, the server just returns an *ACK* for the correct reception and then further process the image locally. The detection info that are produced by the object detection module are the same for the previous application (figure [7.3]), but are elaborated in a different way. The JSON file attached to the image sent by the client contains the shelf identifier and the image name/path (figure [7.5]).

```
{
        "imagePath": "test001.jpg",
        "idScaffale": "S001"
}
```

*Figure 7.5:* The JSON file produced by the client and sent to the server, together with the image file.

In the server, there are two tables *"DM_SCAFFALI"* and *"DM_PRODOTTI"* that contain the information about the shelves and the products, with a key value to identify them. These tables are shown in figures [7.6] and [7.7].

| id_prodotto | nome_prodotto | class_label | desc_prodotto | foto_prodotto |
|---|---|---|---|---|
| P001 | ambre | 1 | ... | ambre.jpg |
| P002 | croyance | 2 | ... | croyance.jpg |
| P003 | doucefaute | 3 | ... | doucefaute.jpg |
| P004 | entspannung | 4 | ... | entspannung.jpg |
| P005 | gluckliche | 5 | ... | gluckliche.jpg |
| P006 | lonia | 6 | ... | lonia.jpg |
| P007 | madame | 7 | ... | madame.jpg |
| P008 | mademoiselle | 8 | ... | mademoiselle.jpg |
| P009 | magicrose | 9 | ... | magicrose.jpg |
| P010 | mandelbluten | 10 | ... | mandelbluten.jpg |
| P011 | ponant | 11 | ... | ponant.jpg |
| P012 | reinedemai | 12 | ... | reinedemai.jpg |
| P013 | schonheits | 13 | ... | schonheits.jpg |
| P014 | stressfrei | 14 | ... | stressfrei.jpg |
| P015 | tiefen | 15 | ... | tiefen.jpg |
| P016 | versant | 16 | ... | versant.jpg |

*Figure 7.6:* The table *DM_PRODOTTI* that contains the information about the products to detect in the shelves.

| id_scaffale | num_rows | desc_scaffale |
|---|---|---|
| S001 | 2 | ... |
| S002 | 3 | ... |
| S003 | 1 | ... |
| S004 | 1 | ... |
| S005 | 2 | ... |
| S006 | 3 | ... |
| S007 | 1 | ... |
| S008 | 2 | ... |
| S009 | 1 | ... |
| S010 | 3 | ... |

*Figure 7.7:* The table *DM_SCAFFALI* that contains the information about the shelves. The number of rows is the number of trays of the shelf.

With the information extracted by the object detection module and post processed, the server fills out another table *"FT_SCAFFALI_PRODOTTI"* (figure [7.8]) where each product is assigned to a position in the shelf, according to its bounding box coordinates.

| id_scaffale | row | col | id_prodotto |
|---|---|---|---|
| S001 | 1 | 0 | P002 |
| S001 | 1 | 1 | P003 |
| S001 | 0 | 0 | P004 |
| S001 | 1 | 2 | P009 |
| S001 | 0 | 1 | P014 |
| S001 | 1 | 3 | P011 |
| S001 | 1 | 4 | P016 |
| S001 | 0 | 2 | P013 |

*Figure 7.8:* The table *FT_PRODOTTI_SCAFFALI* that contains the information about the products placed in the shelf. The *row* column indicates the tray in the shelf, from down on). The *col* column is referred to the relative position of the items in the same row, from left to right.

Once this table is filled with the products, then it is possible to use another software for the analysis of the merchandising, for example Power BI. Microsoft Power BI is a software for the visualization of data, which retrieves the information from a database and allows to create an interactive report with tables and graphics. In the following pictures (figure [7.9] and [7.10]), there are some screenshots from the Power BI products analysis.

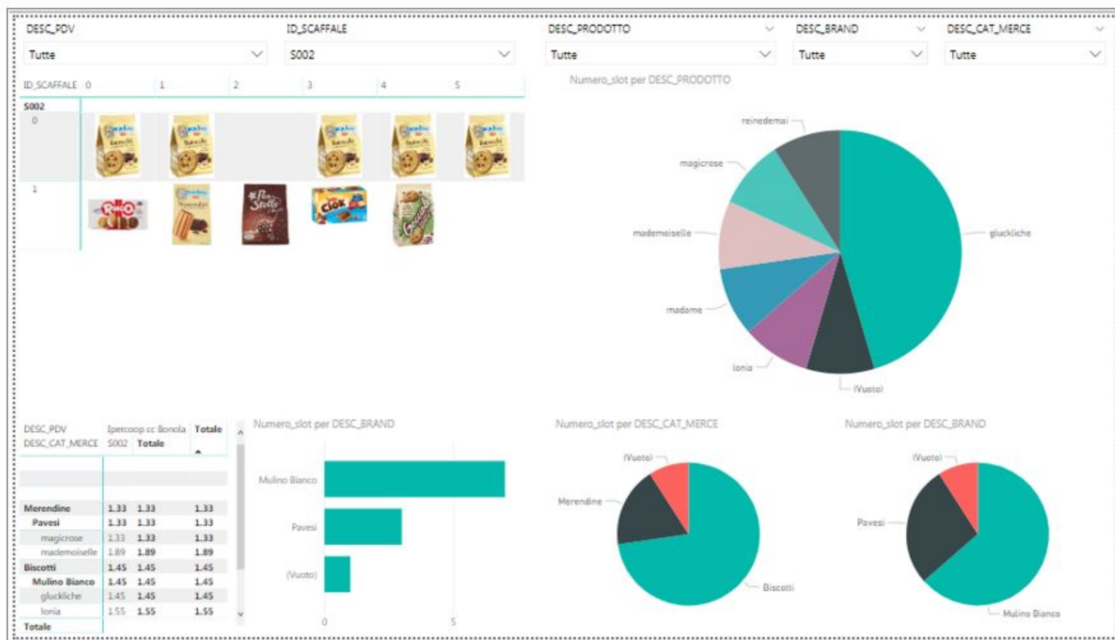



*Figure 7.9:* Top left: disposition of the items on the shelf. Top right: pie chart of the occupation of the products. Bottom left: price analysis. Bottom center: histogram of the product brands. Bottom right: pie charts of the category and brand occupation.

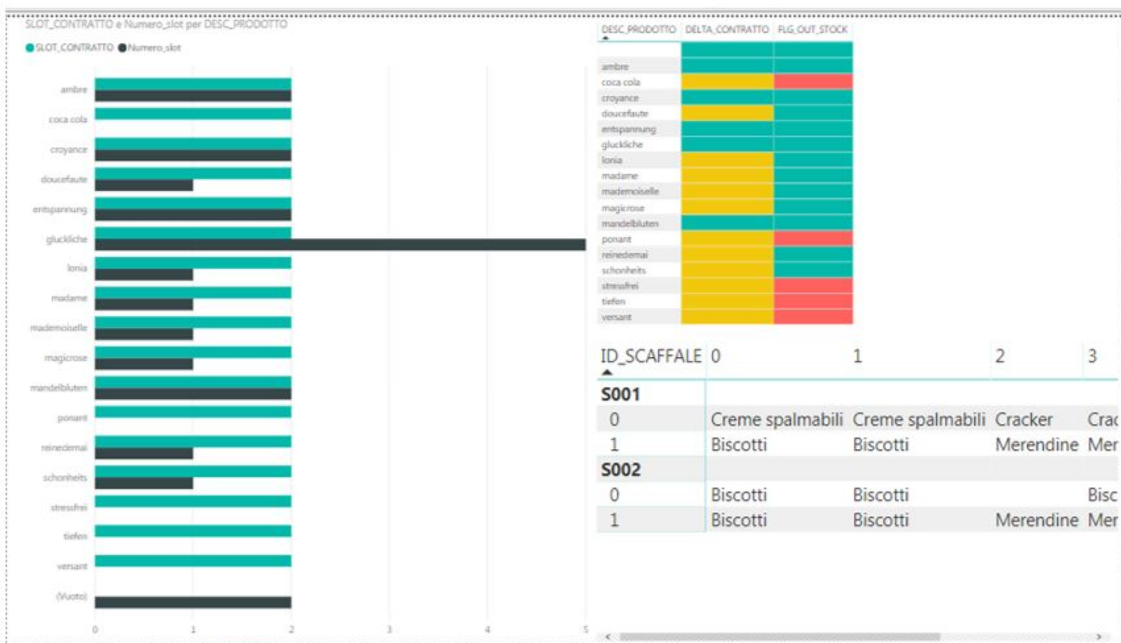



*Figure 7.10:* Left: comparison between occupied slot and agreed slot. Top right: analysis of agreed correspondence. analysis of the adjacent categories of the products.

# Chapter 8

# Conclusions

In this thesis, an object detection framework has been used to build an automatic products recognition system that is able to detect a catalogue of 16 types of items placed on a shelf. As discussed in chapter 1, such a system is of extreme importance in the mapping of the store and the information extracted with it can be used in several applications. Two of these applications have been analysed and discussed in chapter 7: the Oracle Proxima Smart City and a commercial brand analyzer. While the first aims at introducing a new kind of shopping where cameras and sensors are able to detect and take care of the customers' bills without creating a bottleneck at the check-out, the second application is more related to the monitoring of the commercial agreements. These are two examples of the various applications achievable with an object detection system. Indeed, the main important thing, as discussed in chapter 1, is to maintain an updated database of the items in the shelves, and then this information can be used as wished. This is what is actually done by the Object Detection module of this project, which uses a deep neural network to identify and locate the items in the image.

The choice of TensorFlow as deep learning environment is justified by the fact that it is the most widespread and documented tool for this kind of applications. Moreover, it provides some easy-to-use classes and Python scripts to train and test the neural networks. Finally, it comes with some pre-trained models (TensorFlow Model Zoo) that can be fine-tuned by users on some specific datasets, as it was our case. The only drawback was the preparation of the dataset, since every image had to be manually labelled. The dataset preparation phase is perhaps the bottleneck of these kind of applications, since the network has to be retrained with new images every time that an item is added to the catalogue. Indeed, the models trained in this thesis are able to detect with high accuracy all of the 16 items of the list, but adding a new item would require a completely new dataset and a new training phase. This is the main open issue of this project, since any amendment in the catalogue of items requires a lot of work. On the other hand, we can be satisfied with the results on the restricted dataset used. Moreover, as suggested by the authors of the pre-trained models, the maximum number of item classes that can be detected and classified can reach also 90, which is enough to cover the catalogues of many specific stores.

The model selection is guided by the QoS required by the application. The general trend of the object detection models is that the accuracy is inversely pro-

portional to the inference time required to produce the result. As discussed in chapter 6, among the three models, the SSDLite is the fastest and the less precise at the same time. If the model was installed on a powerful server with high computational capabilities, the SSDLite would probably be the most suitable choice for building a system that works in real-time. The low accuracy would be compensated with an high number of possibility to identify the items. Things are different when considering applications that works offline, since the object detection module is not required to give results instantly. In this case, the inference time is sacrificed in favour of a more accurate and precise network, as Faster R-CNN.

The analysis of the different networks, together with their capabilities and limits is the part where I focused the most during the development of the project. Indeed, the Web Service module has nothing to do with the deep learning system, but it is thought to give users a readable and easy-to-use interface to send the image to the server and receive the results.

# Bibliography


[Lec+95] Yann Lecun et al. “Comparison of learning algorithms for handwritten digit recognition”. In: Jan. 1995.

[FH04] Pedro Felzenszwalb and Daniel Huttenlocher. “Efficient Graph-Based Image Segmentation”. In: *International Journal of Computer Vision* 59 (Sept. 2004), pp. 167–181. DOI: 10.1023/B:VISI.0000022288.19776.77.

[KSH12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton. “ImageNet Classification with Deep Convolutional Neural Networks”. In: *Neural Information Processing Systems* 25 (Jan. 2012). DOI: 10.1145/3065386.

[Gir+13] Ross Girshick et al. “Rich Feature Hierarchies for Accurate Object Detection and Semantic Segmentation”. In: *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition* (Nov. 2013). DOI: 10.1109/CVPR.2014.81.

[LCY13] Min Lin, Qiang Chen, and Shuicheng Yan. “Network In Network”. In: *CoRR* abs/1312.4400 (2013).

[Ser+13] Pierre Sermanet et al. “OverFeat: Integrated Recognition, Localization and Detection using Convolutional Networks”. In: *International Conference on Learning Representations (ICLR) (Banff)* (Dec. 2013).

[Uij+13] J. R. Uijlings et al. “Selective Search for Object Recognition”. In: *Int. J. Comput. Vision* 104.2 (Sept. 2013), pp. 154–171. ISSN: 0920-5691. DOI: 10.1007/s11263-013-0620-5. URL: http://dx.doi.org/10.1007/s11263-013-0620-5.

[ZF13] Matthew Zeiler and Rob Fergus. “Visualizing and Understanding Convolutional Neural Networks”. In: vol. 8689. Nov. 2013. DOI: 10.1007/978-3-319-10590-1_53.

[SZ14] Karen Simonyan and Andrew Zisserman. “Very Deep Convolutional Networks for Large-Scale Image Recognition”. In: *arXiv 1409.1556* (Sept. 2014).

[Gir15] Ross Girshick. “Fast r-cnn”. In: (Apr. 2015). DOI: 10.1109/ICCV.2015.169.

[IS15] Sergey Ioffe and Christian Szegedy. “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift”. In: (Feb. 2015).

[Liu+15] Wei Liu et al. “SSD: Single Shot MultiBox Detector”. In: *CoRR* abs/1512.02325 (2015). arXiv: 1512.02325. URL: http://arxiv.org/abs/1512.02325.

[Ren+15] Shaoqing Ren et al. “Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 39 (June 2015). DOI: 10.1109/TPAMI.2016.2577031.

[Sze+15] Christian Szegedy et al. “Going deeper with convolutions”. In: June 2015, pp. 1–9. DOI: 10.1109/CVPR.2015.7298594.

[He+16a] Kaiming He et al. “Deep Residual Learning for Image Recognition”. In: June 2016, pp. 770–778. DOI: 10.1109/CVPR.2016.90.

[He+16b] Kaiming He et al. “Identity Mappings in Deep Residual Networks”. In: vol. 9908. Oct. 2016, pp. 630–645. ISBN: 978-3-319-46492-3. DOI: 10.1007/978-3-319-46493-0_38.

[RF16] Joseph Redmon and Ali Farhadi. “YOLO9000: Better, Faster, Stronger”. In: (Dec. 2016).

[Red+16] Joseph Redmon et al. “You Only Look Once: Unified, Real-Time Object Detection”. In: June 2016, pp. 779–788. DOI: 10.1109/CVPR.2016.91.

[SIV16] Christian Szegedy, Sergey Ioffe, and Vincent Vanhoucke. “Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning”. In: *AAAI Conference on Artificial Intelligence* (Feb. 2016).

[Sze+16] Christian Szegedy et al. “Rethinking the Inception Architecture for Computer Vision”. In: June 2016. DOI: 10.1109/CVPR.2016.308.

[Cho17] Francois Chollet. “Xception: Deep Learning with Depthwise Separable Convolutions”. In: July 2017, pp. 1800–1807. DOI: 10.1109/CVPR.2017.195.

[How+17] Andrew Howard et al. “MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications”. In: (Apr. 2017).

[Hua+17] Gao Huang et al. “Densely Connected Convolutional Networks”. In: July 2017. DOI: 10.1109/CVPR.2017.243.

[Xie+17] Saining Xie et al. “Aggregated Residual Transformations for Deep Neural Networks”. In: July 2017, pp. 5987–5995. DOI: 10.1109/CVPR.2017.634.

[RF18] Joseph Redmon and Ali Farhadi. “YOLOv3: An Incremental Improvement”. In: (Apr. 2018).

[San+18] Mark Sandler et al. “MobileNetV2: Inverted Residuals and Linear Bottlenecks”. In: June 2018, pp. 4510–4520. DOI: 10.1109/CVPR.2018.00474.