POLITECNICO
DI TORINO

POLITECNICO DI TORINO

Master degree course in Computer Engineering

Master Degree Thesis

Brain inspired Image Understanding

Smart Algorithm: semantic reasoning
to enhance deeplearning models

Supervisor
prof. Elena Baralis

Candidate
Andrea Pasini

October 2017

Abstract

In this thesis we propose an object recognition algorithm to improve the accuracy of convolutional neural network models. The goal is to classify multiple objects contained into the analyzed picture. Our method is inspired by the way we understand images, using both visual and semantic information. The key point of the reasoning is that objects of a particular class maintain specific spatial relationships with the others. For example ceiling lamps hang from the ceiling, floors lie to the bottom of the image. Computer screens are more likely found near keyboards, desks near to chairs, windows close to walls. Also the relative size between objects can be useful for scenery understanding, as many categories present particular relationships. For example floors are larger than many other elements, doors are taller, while lamps are smaller.

Considering a scene with multiple objects, each of them is classified firstly by the convolutional neural network, which is considered here the baseline model. Afterwards, our algorithm is applied to the whole scene with the aim of improving the baseline classifer results. The method is called Smart Algorithm because it uses the contextual information, as opposed to the neural network which only looks at the local characteristics of the image sub-regions. Our model can also provide interpretability of the results, since it is rule based; this cannot be easily done with the neural networks.

The dataset used for our experiments is Sun09, a collection of more than ten-thousands images annotated with XML. These pictures are described with the positions and shapes of the objects to be classified. The model is trained with indoor environments, containing elements like *ceiling*, *floor*, *lamp*, *oven*, *door*. After applying the Smart Algorithm we obtained an improvement of the 10 percent in accuracy with respect to the baseline classifier. The set of rules currently used involves the relative positions and sizes between objects, but many new features, such as textures and shapes, could be implemented in the future.

Contents

Part I

Background

Chapter 1

Introduction

1.1 Human Brain Skills

During million years of evolution nature has reached an increasingly level of complexity. Humans have always tried to mimic it with the purpose of developing technologies, realizing complex and autonomous systems which can simplify everyday life problems. Recently, computer science researchers together with neuroscientists have concentrated their efforts to study the working principles of our brain. The results of these studies can be exploited to obtain more performing machines which really emulate our behaviour and problem solving skills.

In particular the artificial vision and object recognition are among the most challenging fields in computer science. These very complex tasks cannot be achieved with common algorithms, but require complex mathematical models to analyze the pixels and understand the meaning of the images. Indeed, our brain owns an extraordinary ability in elaborating and interpreting the signals from our retinal photoreceptors. Computer vision techniques are performing increasingly better in the last ten years, but they produce results which are still less accurate than the ones humans can manually achieve.

In the next sections we try to focus some of the main principles at the base of our mind

processes. We will try to take these as a guideline and source of inspiration to develop our object recognition model. We point out three simple but fundamental concepts. These are the *hierarchy* of ideas and objects, the *contextual* information and the usage of past knowledge and memories to *reconstruct* missing information.

1.2 Hierarchy

Our thoughts are organized in hierarchies with different abstraction levels. [1, p. 239]

The image understanding process analyzes and decomposes images into sections organized with a hierarchical subdivision. Smaller image patches can be distinguished by color or texture. These are then grouped to construct bigger shapes: from a finer analysis we reach a coarser description of the main image components. At higher abstraction levels we can perform *semantic* reasoning to understand the relationships between the objects of the scene, while at lower levels we can only use *local* visual features.

All of these operations are performed without having to think about them. The conscious access occurs only at the last step. Indeed, in the first fractions of second after the image stimulus the only information we become aware are the main shapes and the coarser details. Afterwards, if our attention is captured by a smaller image region we can become conscious of its finer characteristics. In other words the unconscious process is bottom up, from finer to coarser image patches, while the conscious access occurs from the top level to the bottom one. Unconscious thoughts are much faster than active thinking and for this reason only the fundamental information of the image reaches our attention. Since the other details would slow down the process, they are taken in consideration only when required.

We previously talked about semantic reasoning over the components of the scene. To learn recognizing a new object we don't require to observe thousands of samples, but we only need to understand its general structure and the parts it is composed of. For example

vehicles can be described by elements like wheels, doors, handles and windows, located with specific spatial relationships from each other. These information help our brain to recognize objects not only from their visual characteristics, but taking in consideration the meaning of the whole image as well.

1.3 Context

Ideas and images are interpreted with the help of their context.

Both for speech and visual analysis we always consider the context surrounding the object of our interest. Our brain is provided with mechanisms which bring the conscious attention to an event every time we notice a mismatch with its context. Indeed, to solve the conflict it is necessary the active participation of our reasoning.

Some neuroscience experiments confirm this theory. For example while hearing a sentence like *"At breakfast, I like coffee with cream and socks"*, the areas of our brain used to recognize words in their context find the discrepancy between the last word and the others. A brain wave called *N400* originates at this point. Its amplitude is as big as the discrepancy found in the sentence. This wave takes the information to the conscious level with the purpose of generating more complex mechanisms to understand the reason of the conflict. [1, p. 107-108]

It is above all in the case of image understanding that the elements of the scene are interpreted using their context. To make an example, we don't have to clearly see the image of a smartphone to correctly recognize it, if it is held close to the ear of a man who is talking without being in front of an interlocutor. A combination of contextual information and hierarchical representation of a scene could be exploited to create more robust labeling systems, like shown in the paper *Exploiting Hierarchical Context on a Large Database of Object Categories* [2]. The usage of context to interpret the objects in a scene is a key point of the algorithm presented in this thesis.

1.4 Reconstruction

We reconstruct the missing information with the usage of our previous experiences.

When we cannot clearly hear some words in a sentence we introduce them according to a probabilistic guess based on the context and our memories. [1, p. 240]. The same process applies in the case of image understanding. For example our retina presents a blind spot located in the position of the optic disk, which is the optic nerve head. We don't notice a missing part or a black spot because our mind reconstructs this one and produces the completed image.

Another evidence emerges from the experiment which demonstrates the McGurk effect. This consists of showing the video of a person saying the *"da"* syllable, while the audio track speaks clearly the syllable *"ba"*. In this case the subject of the experiment seems to perceive the sound *"da"* because the cognitive processes of the visual cortex correct the information coming from the auditory nerve [1, p. 93-94]. The McGurk effect demonstrates how the reconstruction performed by our mind is not always correct and could mislead us. Nevertheless this process is essential to interpret the information which originates from our senses to quickly produce a complete representation of the external world.

1.5 Cognitive architectures

Trying to understand what is behind the cognitive processes of our brain is quite complex and neuroscientists are concentrating their efforts to achieve this task. These theories are commonly called *cognitive architectures*, since they describe the structure of the human mind. The most accepted descriptions compare the unconscious part of our brain to a set of statistical elaborators which work in parallel. They elaborate the local information such as image regions or the words in a sentence. Each of the elaborators produces

its probabilistic representation of the external world. This may be not correct, since it is based only on a portion of the input data. The produced information reaches the more abstract processes which choose the global interpretation and get a feedback to the local elaborators. These ones will correct their decisions and converge to a single interpretation [1, p. 133-138].

Our active perception is practically unaware of the complex mechanisms which hide in the unconscious part of the brain. The only information we can observe, according to the *Global Workspace Theory* (GWT), is the final result of this elaboration. In this way we are only aware of the essential information describing the external world, leaving out the statistical results of the intermediate areas which would otherwise be quite confusing [1, p. 258-260].

This description is confirmed by imaging techniques, such as the fMRI (Functional Magnetic Resonance Imaging). Thanks to these methods, neuroscientists have found long connections between remote areas in our brain. If the area A is connected to area B, we will likely observe neural signals in answer from B to A, generating a circuit of reverberating brain waves. The visual cortex, for example, strongly interacts with the higher brain regions such as the language area [1, p. 215, 233].

1.6 Objectives

The main objective of this thesis is to design an algorithm which emulates the brain mechanisms with the purpose of enhancing the accuracy of a baseline image recognition method. Specifically, given an input image representing multiple objects we would like to categorize them by assigning a class label. The baseline model is a local classifier because it produces the results by looking at the single objects in the image. The classification produced by the baseline model is taken as input from our algorithm which tries to improve the accuracy of the class labels using the information of the whole image.

This is called semantic reasoning as opposed to the local model which makes only use

of the visual patterns contained in the image pixels. The point of strength of our model is not only the improvement of accuracy, but also its characteristic of being a rule based algorithm. This feature can add more interpretability to the decisions because we can focus to the reason why the algorithm changed or kept some of labels conferred by the local model [3]. Instead, the baseline classifier, built with artificial neural networks, it is not interpretable and we cannot see the explanation of the results it produces.

The system we designed in our thesis work is called *Smart Algorithm*, as it reminds to the smart way of our brain to interpret the outside world by the usage of reasoning. In the first sections of this chapter we described three principles which are at the base of our brain mechanisms. We introduce now the way we were inspired by these concepts to design our model. The *hierarchy* is given by the way we approach the problem. Firstly, local models analyze the single pixels of a particular image region, then the Smart Algorithm computes the global interpretation at a higher abstraction level. The *context* is exploited to perform the correction, indeed the model tries to guess the new label of an object being aware of the other elements into the scene. This is achieved by comparing the current scene configuration to the ones analyzed during the learning process. The *reconstruction* feature refers to the ability of the Smart Algorithm to reconstruct the real labels of the objects even if the local model has given a wrong prediction for some of them.

1.7 Summary

The outline of this thesis goes from a presentation of the basic image understanding concepts to the design choices of our model. In chapter 2 we describe the concepts of classification and detection for image processing, followed by a brief summary of the metrics to evaluate the model results. In chapter 3 we present the working principles of the convolutional neural networks, which will be used as local models to classify the objects of the image. Since the choice of a good dataset is fundamental to test the algorithms, chapter 4 describes the main image collections that we analyzed before selecting the best one for

our experiments. The second part of this thesis presents in chapter 5 the preprocessing of our dataset and the local classifier architecture. Finally, in chapter 6 we will point out how we designed the Smart Algorithm and the way we use it to improve the accuracy of the baseline model.

Chapter 2

Automatic image recognition

During the last decade the rapid growth of computational power and the birth of big-data algorithms have allowed researchers to try new complex models in the field of computer vision [4], [5], [6], [7], [8], [9], [10], [11], [12], [13].
These algorithms require heavy computation, as they elaborate great amounts of data retrieved by increasingly bigger datasets [14], [15], [17], [18].

This chapter firstly aims to clarify which tasks are mainly focused by modern research about image recognition. Secondly, we will explain how evaluation metrics are defined to compare different models. In particular, a brief summary about the classic measures is given. Afterwards we will introduce the *mean average precision* metric, which became popular more recently. Finally, the last section shows how we can represent generic data or images with a mathematical concept called *tensor*. This is the basic object manipulated by the most recent models called *convolutional neural networks*.

2.1 Image recognition challenges

Automatic image recognition is a very complex task of computer vision. For years researchers have tried to find new models capable of achieving better performances and overcome their ancestors. The common goal between these kinds of algorithms is finding

an abstract description of the input image, for example a label or a list of the represented objects.

The first step of image processing is the *feature extraction*, a way of retrieving significant information to describe a picture. Once features have been extracted they can be provided to classifiers which can decide a label for the selected sample. Color histograms are an example of features that provide the distribution of each color in the image. They are thought as a way of summarizing all the pixels with a small list of values. Other characteristics that may be important to describe visual objects can be edges and corners because they define the key points of a shape. Many algorithms of this type can be found in literature [19], [20], [21].

There are two main drawbacks associated with this step. The first one is the need for algorithms that are specific to a particular goal and whose parameters must be tuned case by case. For example they could vary with image resolution, lightness and contrast. We must choose the best feature extractors keeping in mind the domain of our dataset. We could have to deal with landscape images, to automatically drive a car or a space exploration vehicle. A different approach would be necessary to analyze medical data, like *MRIs* (Magnetic Resonance Imaging), *CATs* (Computed Axial Tomography), or even images coming from soil drilling for civil engineering. The second one is the slowness of feature extraction, since it requires analyzing many times all the pixels of a picture.

The next chapter will explain how convolutional neural networks can solve the issues we pointed out here. This section will instead describe the difference between classification and localization tasks.

2.1.1 Classification

Classification for image recognition is defined as the task of assigning a label to describe a single image. It can be divided into binary and multi-class.

Binary classification aims to decide whether the sample belongs to a given class. For

example a system could distinguish images containing a particular object from the others. If the answer is affirmative the sample is considered *positive*, otherwise *negative*. These classifiers typically assign a score to support this decision. Positive images become those whose confidence value is greater than a particular threshold.

A *Multi-class* task consists of selecting one among a set of predefined classes. The complexity is greater than binary classification and grows with the number of classes. Figure 2.1 depicts an example of classification between the two classes *dog* and *cat*.

Figure 2.1: Image Classification Example, the number associated to the label represents the confidence of the result.

The result of a classified sample is a vector of scores between zero and one, each of them related to a particular class. The highest confidence value is selected to assign the label to the object being classified.

We illustrate this concept with an example. Supposing a multi-class task with three classes $[c1, c2, c3]$ and an output vector defined as: $[0.2, 0.6, 0.2]$
The class that will be assigned to this sample would be *c2* with a 0.6 confidence. Of course, results with higher confidence are preferred because they represent a strong decision with low uncertainty.

To address the problem of image classification all the models require a long training over big datasets. If the size of the training set is too small the system cannot generalize to the newer samples. This issue is commonly called *overfitting*: when the model conforms too much to the training data, the accuracy on the test set will be lower.

All modern deep learning models work with big datasets, composed of many thousands or even some millions images. The most popular image datasets are *MNIST* (a set of handwritten digits), Cifar-10, Cifar-100 with 10 and 100 classes, respectively. These present some thousands of images, but they are considered relatively small. Newer datasets, like *ImageNet* contain millions of samples. For this reason they are perfect to train deeplearning models.

In the nineties many models were proposed to accomplish these tasks. Artificial neural networks (*ANN*) were commonly used over the features extracted from an image, but they cannot be directly applied to the raw pixels. Recently, the invention of convolutional neural networks allowed to address the task of extracting features and proposing a label with a single algorithm [4], [5], [6].

2.1.2 Detection

Detection refers to the task of finding the position of the objects into an image, assigning them a class label. Images can contain one or multiple objects. We can distinguish two different objectives. The first one consists of finding a rectangular bounding box that fits around the object being classified. The second is called *semantic pixel labeling* (or *semantic segmentation*) and it is achieved by labeling each pixel of the image with a class. In this way the model outputs the real shape of each object, instead of showing only a bounding box. The result will be a three dimensional matrix with shape:

$$[nClasses \times imageWidth \times imageHeight]$$

that means one score for each pair $\{class, pixel\}$. The class with the highest score is selected for each pixel. Figures 2.3 and 2.4 show an example of detection with bounding boxes and semantic pixel labeling, respectively.

Figure 2.2: Original image

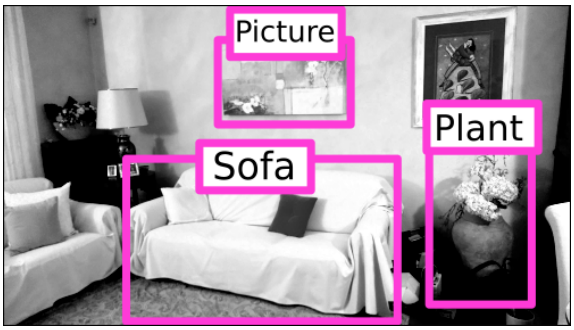

Figure 2.3: Detection with bounding boxes

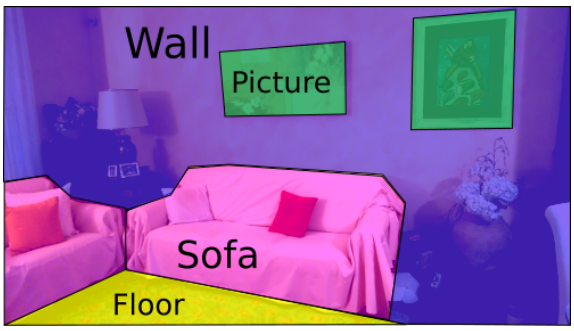

Figure 2.4: Semantic pixel labeling

Object detection has innumerable applications such as automatic driving, artificial vision for a grasping robot arm, search engines or even security systems. Detection can be

more challenging than simple classification and this is the reason why it is the main focus of modern research in the object recognition field.

This task is quite complex because models must consider all the possible subregions of the input image. These can vary between different sizes and shapes, and can assume many positions inside the matrix of pixels. For example, because of perspective, objects of the same class can be bigger or smaller and the model must recognize them in any case.

The first solutions we can find in literature used a technique called *sliding windows*. This consists of applying convolutional neural networks over rectangular subregions of the input picture. For each of them the algorithm reports whether an object is present and what class it belongs to. The approach we described here was very slow because the number of regions is in the order of thousands. For this reason many new attempts have been proposed in the last years [7], [8], [9], [10]. The common idea behind those new models is to apply the neural network only few times. This is achieved by analyzing only the regions that more likely contain an object. Chapter 3 illustrates some of these state-of-the-art models.

2.2 Model evaluation

Model evaluation is fundamental to compare different algorithms. There are many metrics that have been defined to measure the accuracy of a specific model. Some of them are specific for image recognition. From 2005 to 2012 the Pascal VOC (Visual Object Classes) project organized yearly challenges about image recognition [15] [16]. The evaluation metric they decided to use was the *mean average precision* (*mAP*). We will present how to compute this measure and will use it to evaluate the model proposed in this thesis. To better understand *mAP*, in the next paragraphs a brief summary of precision and recall is provided.

2.2.1 Precision and recall: binary classification

Binary classifiers perform the task of distinguishing between positive and negative samples. For example positive images can be divided by negative ones if they represent a particular object, such as a *car* or a *house*.

True-positives (*TP*) are the instances which are correctly recognized as positive by the classifier, while *false-positives* (*FP*) are the negative samples which are erroneously classified as positive. *False-negatives* are instead those positive instances that have been classified as negative. *True negatives* (TN) are the samples correctly assigned to the negative label. The ideal classifier will produce only true-positives and true-negatives, with no false-positives and false-negatives.

The *confusion matrix* is a table which contains these values. It is organized with the predicted class on the horizontal axis and the actual class on the vertical one. With this arrangement true-positives and true-negatives appear on the main diagonal. Table 2.1 shows the graphical representation of a confusion matrix.

	P'	N'
P	TP	FN
N	FP	TN

Table 2.1: Confusion matrix, binary classifier

Precision is the accuracy metric which specifies how many true-positives we can find among all the instances classified as positive. It is defined as:

$$precision = \frac{TP}{TP + FP} \tag{2.1}$$

The precision itself is not enough to evaluate the binary classifier. We illustrate this with an example. Suppose to have a dataset with 100 positive samples. The classifier labels 10 instances as positive: 9 true-positives and 1 false-positive. The confusion matrix is shown in table 2.2.

	P'	N'
P	9	91
N	1	9

Table 2.2: Confusion matrix, with example values

The computed precision is:

$$precision = 9/(9+1) = 0.9 = 90\%$$

We obtain a very high precision, but there are 91 false-negatives that have not been considered by the evaluation metric. This is the reason why recall was introduced.

Recall is the measure which specifies the percentage of true-positives among all the positive samples of the dataset. This also considers the false-negatives, which are the instances that should have been classified as positive by the model. The formal definition is:

$$recall = \frac{TP}{TP + FN} \tag{2.2}$$

In the example above recall is:

$$recall = 9/(9+91) = 0.09 = 9\%$$

which is very low if compared to the 0.9 precision. The global behaviour of the classifier is not good if we consider both precision and recall.

To summarize the values of the two measures described before we can use the F_1. It is a value between zero and one, defined as the harmonic mean of precision and recall:

$$F_1 = 2 \cdot \frac{p \cdot r}{p + r} \tag{2.3}$$

As expected, the value computed for the presented example is quite low, since the recall is only 0.09:

$$F_1 = 2 \cdot (0.9 \cdot 0.09)/(0.9 + 0.09) = 0.16$$

2.2.2 Multi-class evaluation

With multi-class tasks the confusion matrix becomes very useful to analyze the results. The x-axis contains the predicted values, while the y-axis contains the expected classes. It can be represented with two indexes as follows:

$$mat_{i,j} = mat_{actual,predicted}$$

where $i = actual\_class$ (y-axis) and $j = predicted\_class$ (x-axis); $i, j \in [0, N-1]$. Elements on the main diagonal ($mat_{i,i}$) are the instances correctly classified.

The overall accuracy of the model is computed as the sum of the elements in the diagonal divided by the sum of all the elements of the matrix.

$$accuracy = \frac{\sum_{i=0}^{N-1} mat_{i,i}}{\sum_{i=0}^{N-1}\sum_{j=0}^{N-1} mat_{i,j}} \tag{2.4}$$

We illustrate this with an example. Suppose a dataset with three classes, 40 samples for each of them. The confusion matrix is:

	c0'	c1'	c2'
c0	**40**	0	0
c1	0	**38**	2
c2	0	1	**39**

Table 2.3: Confusion matrix, multiple classes

c0', c1' and c2' are the predicted classes, while c0, c1, c2 are the actual ones of the samples. The overall accuracy is computed as follows:

$$accuracy = (40 + 38 + 39)/(120) = 0.975$$

Having established the overall accuracy, it would be interesting to compute precision and recall separately for each class. We define the precision for $class_c$ in the following way:

$$precision(c) = \frac{mat_{c,c}}{\sum_{i=0}^{N-1} mat_{i,c}} \tag{2.5}$$

The numerator refers to the number of correct instances for class c, while the denominator is the sum of the elements into the $column_c$ (in other words all the samples classified with class c). Example:

	c0'	c1'	c2'
c0	40	**0**	0
c1	0	**38**	2
c2	0	**1**	39

Table 2.4: Precision computation

$$precision(c_1) = 38/(0 + 38 + 1) = 0.97$$

The recall in multi-class classification is defined to be:

$$recall(c) = \frac{mat_{c,c}}{\sum_{i=0}^{N-1} mat_{c,i}} \tag{2.6}$$

In this case the denominator is the sum of all the elements into row_c, that is the set of instances with actual class c.

	c0'	c1'	c2'
c0	40	0	0
c1	**0**	**38**	**2**
c2	0	1	39

Table 2.5: Recall computation

$$recall(c_1) = 38/(0 + 38 + 2) = 0.95$$

2.2.3 Mean Average Precision (mAP)

We will now consider the definition of *mean average precision* (*mAP*) described in the Pascal VOC papers written by Mark Everingham and his team [15] [16]. The Pascal Visual Object Classes (VOC) challenges can be used as a benchmark in visual object recognition. The last challenge was organized in 2012.

Mean Average Precision is an evaluation metric which summarizes the behaviour of the model over all the recognized classes. It is simply computed with the mean of another measure called *Average Precision* (*AP*), which is defined for each class. More formally:

$$mAP = \frac{1}{N} \cdot \sum_{c=0}^{N-1} AP(c) \tag{2.7}$$

To illustrate how the *average precision* is computed, we have to consider the whole model as a set of independent binary classifiers, one for each class. Considering class *c*, the binary task consists of deciding whether a sub-image contains an object of that class. For each subregion being classified we take the output score associated to class *c* and, if it is greater than a particular threshold, the assigned label will be *positive*, otherwise *negative*. For example we could select all positives for class *horse* with confidence greater than $thr = 0.7$

Having selected the class and a particular threshold, we count all the true-positives, false-positives and false-negatives in the test set. Precision and recall can be then computed. We repeat this procedure for many values of threshold in the range 0-1. For each value, a pair precision-recall can be drawn as a point of a curve. The x-axis represents the recall, while the y-axis is related to the precision.

Precision and recall are related by a trade-off in this curve: when recall grows, precision becomes lower. With threshold values closer to one only few samples will be labeled as positive and the recall will be lower, with high precision. On the other hand, if the threshold is closer to zero, there will be many positives and the recall will become closer to one. For this reason the shape of the curve goes the from top-left to the bottom-right corners in the

graph. Figure 2.5 depicts an example of precision-recall curves.

According to Pascal VOC paper, the precision-recall curve must be corrected to avoid wiggles as follows:

$$p_{interp}(r) = \max_{\bar{r} \geq r} p(\bar{r}) \tag{2.8}$$

where $\bar{r}$ are the recall values greater than the recall for which we want to compute the precision. This interpolated curve is monotonically decreasing. The *interpolated average precision* is defined as the area under the curve, more formally:

$$AP = \sum_{i=0}^{N-1} p_i^{interp} \cdot (r_i - r_{i-1}) \tag{2.9}$$

where r_i is the recall value for the point i of the curve, $r_{-1} = 0$, p_i^{interp} is the precision of the interpolated curve at point i.

Each class has its own precision-recall curve and the AP value. When the average precision is closer to one the classifier works correctly and the curve will be like the blue one shown in figure 2.5.

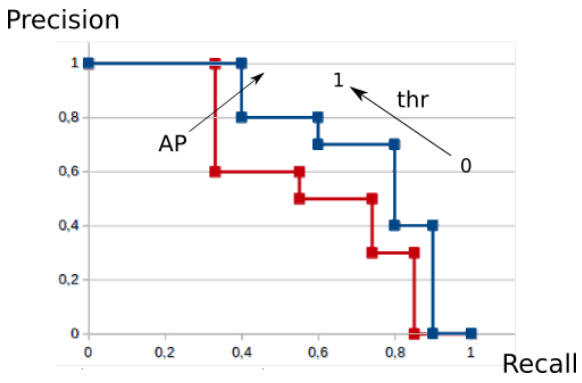

Figure 2.5: Precision-recall curve. The blue curve (the one with higher precision values) has a greater AP score.

In relation to the complexity of the task, significant values of AP can also be in the order of $0.3 - 0.5$ (or $30\% - 50\%$ if written in percents). Some examples for Pascal VOC challenge (2010) are:

mAP	Paper
24.98%	*Learning Collections of Part Models for Object Recognition*, University of Illinois at Urbana-Champaign, CVPR 2013
35.1%	*Search for Object Recognition*, J.R.R. Uijlings, K.E.A. van de Sande, T. Gevers, and A.W.M. Smeulders, IJCV 2013
40.4%	*Fisher and VLAD with FLAIR*, Koen E.A. van de Sande, Cees G.M. Snoek, Arnold W.M. Smeulders, CVPR 2014

Table 2.6: mAP examples

2.3 Image representation

Before analyzing how convolutional neural network are defined, we have to specify a formal way to represent images with any color depth. Commonly pictures are represented with a set of colored points, called pixels. They are disposed one after the other to form the image. Grayscale images can be represented with one value per pixel, associated to the intensity of the light. Color images need three channels to represent blue, green and red intensities. We use the shorthand *BGR image* to name this encoding. Hence, to represent a color image we need three bidimensional matrices, one for each channel. We will now see a more general and formal representation for multidimensional data.

2.3.1 Tensors

A tensor is a mathematical entity representing a multidimensional vector that can store any kind of numerical data. This can be seen as a general way to represent data organized in matrices with an arbitrary number of dimensions. Like vectors, tensors are composed

of numbers characterized by the same type (single or double precision floating point numbers, integers, or even complex numbers). We adopt here the notation introduced with the machine learning library *Tensorflow* [22].

Each tensor is characterized by a rank and a shape. The *rank* specifies the number of dimensions. A scalar number has rank zero, common vectors (such as one-dimensional arrays in programming languages) have rank one, while matrices are rank-two vectors. The *shape* defines, for each dimension, the number of elements it contains. The shape of a vector is equal to its length, for example $[a, b, c, d]$ is a rank-one tensor with shape $[4]$ because it contains four elements. Matrices are tensors with rank two and shape equal to their width and height. Example:

$$\begin{bmatrix} a & b & c \\ d & e & f \end{bmatrix}$$

This matrix can be represented with a rank-2 tensor of shape $[2, 3]$, since its width is 3 and its height is 2. The general shape for a matrix is $[height, width]$. Continuing with this reasoning, we can generalize for rank-three tensors which have shape $[depth, height, width]$. The depth value can be used to represent the color channels for images. For example a bitmap which is 512 pixels in width and 128 pixels in height, with three BGR color channels, is a rank-three tensor with shape $[3, 128, 512]$. Table 2.7 shows a summary of the concepts presented in these paragraphs.

Example	Rank	Shape
Scalar	0	$[\ ]$
Vector	1	$[length]$
Matrix	2	$[height, width]$
BGR image	3	$[channels, height, width]$
Generic Tensor	n	$[s_{n-1}, ...s_1, s_0]$

Table 2.7: Tensor examples

2.3.2 Tensor internal representation

We have analyzed the mathematical representation for tensors. To complete the overview about these entities we have to understand how to represent them into the main memory. It is convenient to find the most efficient solution to facilitate the application of the classification algorithms. The majority of deeplearning libraries, like *TensorFlow* and *Deeplearning4j* serialize tensors into a single vector of floating point numbers. This is done by making use of the *stride* concept.

A stride is a vector that specifies, for each dimension, the separation of contiguous elements. To better understand this definition, we provide some examples. A vector has stride $s = [1]$ because it has only one dimension and elements are separated by one place. Instead, matrices have stride equal to $s = [width, 1]$. When moving along a row the step is equal to one element. Again when moving from one row to another, keeping the same column, we have to skip a number of elements equal to the matrix width. When dealing with a BGR image, in order to move from one channel to the other, keeping the same row and column, we have to skip $width \cdot height$ elements. Table 2.8 shows a recap of the stride definition.

Example	Rank	Stride
Scalar	0	$[\ ]$
Vector	1	$[1]$
Matrix	2	$[width, 1]$
BGR image	3	$[width \cdot height, width, 1]$

Table 2.8: Strides examples

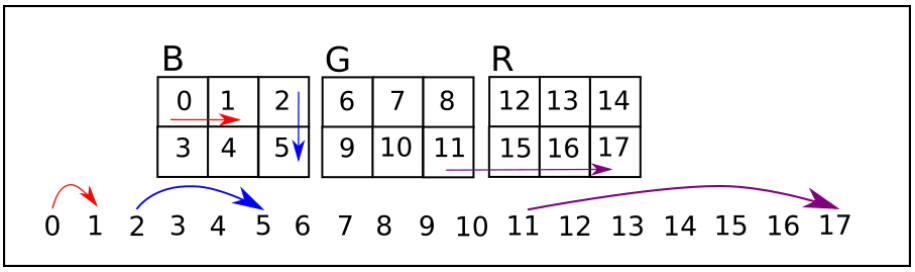

Figure 2.6: Strides Example

The matrices shown in figure 2.6 compose a BGR tensor with rank three. Pixel values are numbered from 0 to 17 and can be serialized as shown in the lower part of the picture. Each rectangular matrix represents the disposition of the pixels for a specific color channel. The shape of the represented tensor is defined to be:

$$shape = [nChan, height, width] = [3,2,3]$$

While moving from pixel 0 to pixel 1 we just skip one element. The transition from pixel 2 to 5 requires skipping a row of three elements. Between value 11 and 17 the number of places is equal to six ($width \times height$). In this way the strides assume the following form:

$$strides = [6,3,1]$$

Chapter 3

Convolutional Neural Networks

Convolutional neural networks provide a solution to the feature extraction problem since they are capable of analyzing and classifying directly the raw pixels. Their point of strength is the structure composed of multiple layers. While data flows through these ones it is transformed to abstract features. When the information arrives to the last layer it is ready to produce the classification result.

In the next sections the structure of convolutional neural networks will be analyzed in detail. To better illustrate their working principles, some examples are provided. We will describe the way this model can learn from data and how it can be used to classify images characterized by one or more color channels. Finally, some practical information about training and overfitting are given. The last section of this chapter presents some of the most popular state-of-the-art models.

3.1 Deeplearning origins

Instead of creating algorithms for each specific task, artificial neural networks can learn our goal from a set of examples called *training set*. The first model of neural network was presented by Frank Rosenblatt with his paper "*The Perceptron - a perceiving and recognizing automaton.*", in 1957. Inspired by biological neurons, he created a linear model

which computes a weighted sum of its input values to produce the result:

$$y = \sum_{i=0}^{N-1} w_i \cdot x_i \tag{3.1}$$

where x_i represents the element i of the input vector and w_i are the weights.

The output value y is subjected to an *activation function* which mimics the ability of neurons to produce an output signal only if sufficiently excited. Mathematically, the purpose of the activation function is to introduce non-linearity into the network. In particular this is necessary for multi-layer models that will be described later. Figure 3.1 depicts how a perceptron is composed.

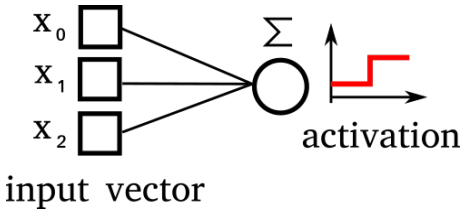

Figure 3.1: Perceptron

Perceptrons work with the binary step activation function. It is characterized by a threshold value, which is typically zero. If the activation y of the neuron is greater than this value then the selected class is positive, otherwise negative. This acts as a binary classifier.

$$result = \begin{cases} 1 & \text{if } y \geq 0 \\ 0 & \text{otherwise} \end{cases} \tag{3.2}$$

Modern neural networks can choose between many different kinds of activation functions. We will see some of them in the next sections.

Later models, called *feed-forward neural networks*, work with multiple perceptrons. The first implementations can be found in the 70s and 80s. A multi-layer neural network is composed of two or more neuron arrays. Each neuron is connected to all the others of the previous layer. For this reason the topology is also called *fully connected*. The feed-forward name is justified by the fact that data flows layer by layer as far as the output array.

Figure 3.2 shows a graphical representation of a multi-layer neural network. In the case of a classifier, the output layer can contain one neuron for each class to be recognized. The neuron with the higher activation will represent the classification result.

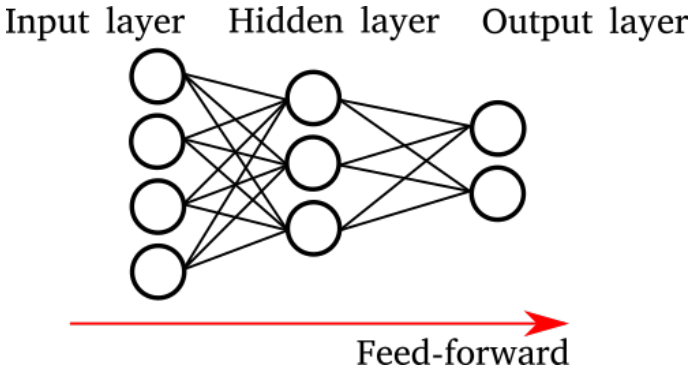

Figure 3.2: A feed-forward neural network

Finally, the learning process is called *back-propagation* because, when a sample is presented to the network, we compute the error of the predicted result and propagate it back from the output layer to the input one. With a *gradient-descent* process, weights are adjusted to minimize the error function of the model over the training data. Equation 3.3 shows the main principles of back-propagation:

$$\Delta W_{i,j} = -k \cdot \frac{\partial E_p}{\partial W_{i,j}} \tag{3.3}$$

where $W_{i,j}$ is the weight j of layer i that is being updated, $\Delta W_{i,j}$ is the adjustment of the weight and it is proportional to the error gradient along the variation of $W_{i,j}$, E_p is the error of the prediction for the presented sample p (input pattern). The error of the last layer, given an input patter p, is computed with:

$$E_p = \frac{1}{2} \cdot \sum_j (d_{p,j} - o_{p,j})^2 \tag{3.4}$$

where $d_{p,j}$ is the desired output for neuron j and $o_{p,j}$ is the actual prediction made from the model.

The idea behind *deeplearning* is to use artificial neural networks (*ANN*) with many hidden layers. To achieve better results, instead of making bigger layers, we increment their number. This operation allows the neural network to ”understand” more abstract concepts since each layer can be seen as a new abstraction level of the input data. The main issues with these models are the long training time, because with deep topologies the gradient computation becomes very complex, and the need for bigger datasets to provide a wider range of examples.

3.2 Building blocks

Having presented a quick recap about older models, we will now describe the main structure of convolutional neural networks (*CNN*). Some standard topologies have been proposed in literature, but researchers can customize them to fit best their goals. These models are structured by a concatenation of building blocks.

The first ones are a succession of *convolutional* and *pooling* layers which extract the features from the input image. Instead, the last module is composed by a fully connected feed-forward neural network. This has the task of performing the actual classification by producing one score for each output class. In the next paragraphs we will go deeper with some examples and illustrations.

3.2.1 Convolution

Convolutional layers are the main building blocks of CNNs. They take an input tensor and produce a new one with a different shape, typically smaller to reduce data dimensionality. The result represents the extracted features. While going deeper into the network, these features assume a more ”abstract” meaning until they are ready to produce the classification result.

A convolutional layer is composed of many squared filters which slide along the pixels

of the image (or the input tensor for hidden layers). In this way they select a small squared patch of data for each position. For each selected patch the sliding window produces a new value that will be inserted into the output tensor. This is achieved by multiplying element by element the filter with the subregion of the input data. All the multiplied values must be summed up to a single result. Mathematically, this operation is defined as the *dot product* between two tensors.

To represent more formally this concept, we specify a notation for selecting a subregion from a tensor:

$$X[\quad][y:y\prime][x:x\prime] \tag{3.5}$$

This describes a rectangular patch of pixels which extends from x to $x\prime$ in width and from y to $y\prime$ in height. All the three color channels are selected and for this reason the first squared parenthesis are void. The pixel indexes of the image X are numbered from the *top-left corner* ($x = 0, y = 0$) to the *bottom-right* ($x = width - 1, y = height - 1$). The set of filters that will be applied to the input image is:

$$W = \{W_0 ... W_n ... W_{N-1}\} \tag{3.6}$$

where N is the number of filters. All of them must have the same shape. Working with tree color channels, the generic filter W_n has shape $[3, w, w]$ and its top-left corner is located at position x, y of the input image. The selected subregion has the same shape of the filter and can be written as:

$$X[\quad][y:y+w-1][x:x+w-1] \tag{3.7}$$

The dot product between W_n and the patch of input image is defined to be:

$$\begin{aligned} Y[n][y][x] &= W_n \cdot X[\quad][y:y+w-1][x:x+w-1] \\ &= \sum_{i=0}^{2} \sum_{j=0}^{w-1} \sum_{k=0}^{w-1} W_n[i][j][k] \cdot X[i][y+j][x+k] \end{aligned} \tag{3.8}$$

As we can see from equation 3.8 the number of channels of the output tensor is equal to the number of filters W_n that we have applied.

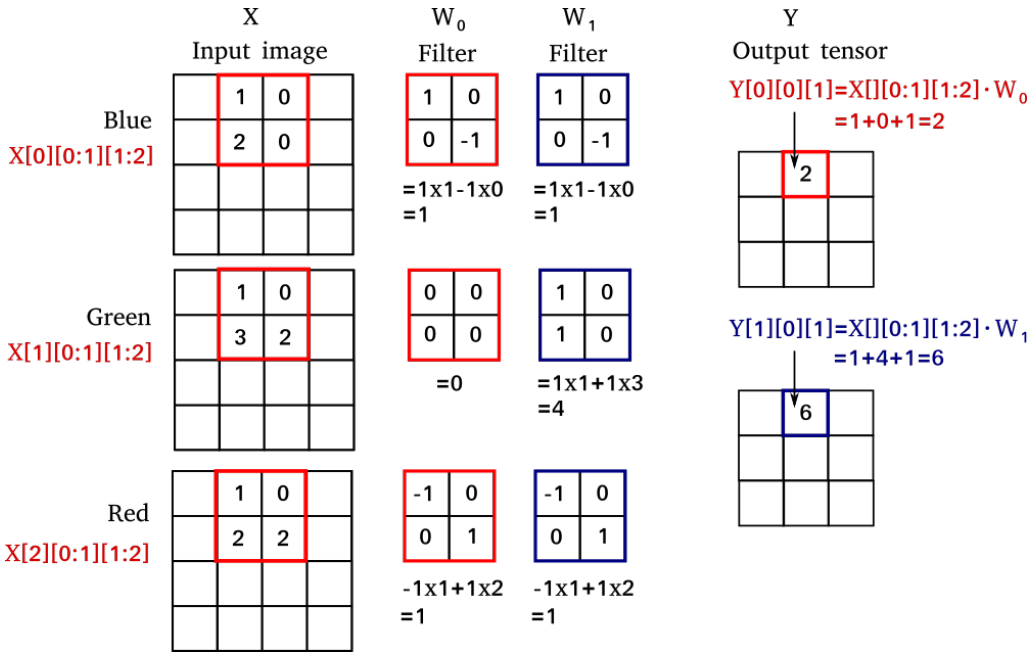

Figure 3.3: Dot product and convolution: the two filters W_0 and W_1 are applied to a patch of image at position $x = 1, y = 0$. Each filter is 2x2 with three depth channels. The operation produces an output tensor with shape 3x3 and two channels (equal to the number of filters).

While performing the convolution, filters can move along the input tensor with a step greater than one pixel to reduce the size of the result. Suppose an input grayscale image with shape $[4,4]$. We compute the convolution with a 2×2 filter moving with step $s_x = 1$ along the x-axis and step $s_y = 2$ along the y-axis. With this configuration the filter can assume three horizontal positions and two vertical ones, for a total of six. The result will be a matrix with shape $[2, 3]$ instead of $[3,3]$ if we had used a constant step along both axes.

Tensor	Shape
Input image	$[4, 4]$
Filter	$[2, 2]$
Output image	$[2, 3]$

Table 3.1: Convolution examples, non unitary step: $s_x = 1$, $s_y = 2$

We recap the concepts presented earlier with an illustrated example. A grayscale input image has size 4×4 and it is convoluted with a 2×2 filter moving with non unitary step: $s_x = 1$, $s_y = 2$. Picture 3.5 shows, for each position of the filter, the result of the dot product with the corresponding patch of input image. Finally, figure 3.6 depicts the result of the whole operation. The output tensor is only one channel in depth because we applied a single filter. This is a toy example, typically the number of filters used for a convolutional layer is in the range from 64 to 512.

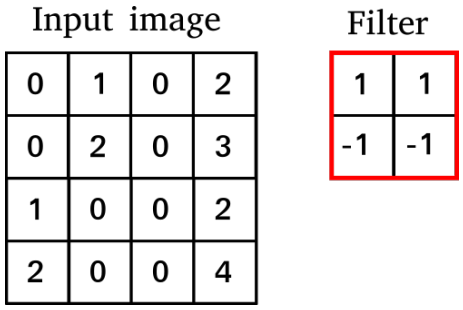

Figure 3.4: Input image and filter

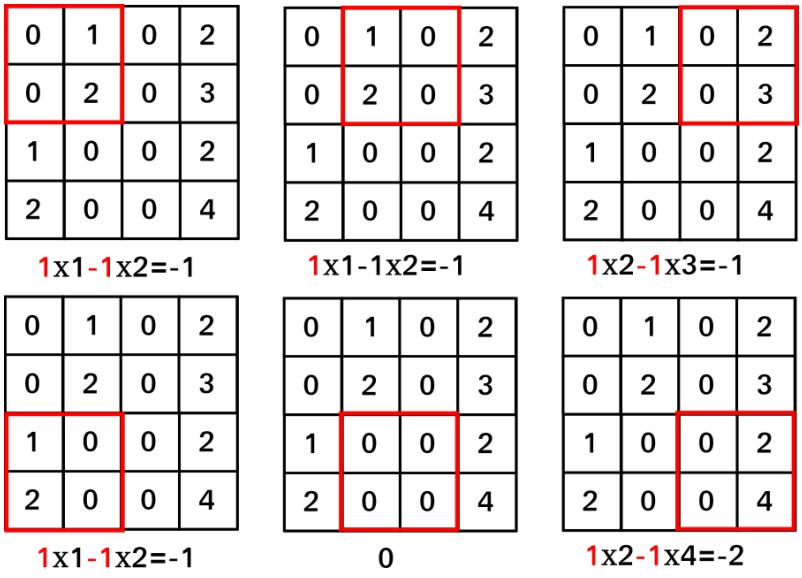

Figure 3.5: Convolution: the red square represents the filter moving along the image

Output tensor

-1	-1	-1
-1	0	-2

Figure 3.6: The output tensor

The values contained into convolutional filters are equivalent to the weights of a common feed-forward neural network and they are computed during the training process. Each of these tensors can be thought as a pixel pattern to be recognized into the input image. When the dot product assumes a high value, the model recognizes a pattern in a particular position and will propagate this information to the next layers. The combination of all these patterns will allow the last layer to perform the actual classification.

3.2.2 Activation Function

Once the output tensor Y has been computed, the convolutional layer performs another important operation: the application of the activation function. Each element of tensor Y is transformed:

$$Y\prime[c][y][x] = f(Y[c][y][x]) \tag{3.9}$$

This operation introduces the non linearity factor into the model. Some examples of activation functions can be found in figure 3.7.

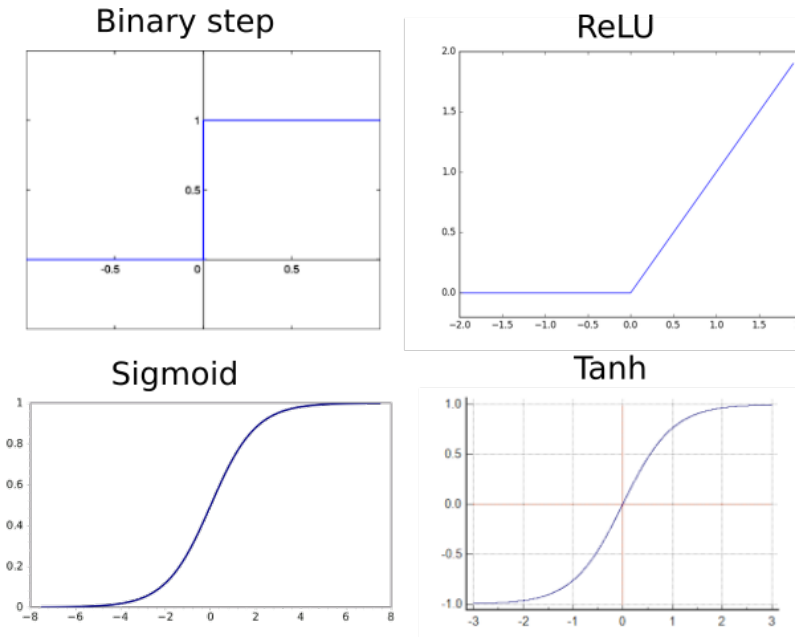

Figure 3.7: Activation functions

3.2.3 Pooling operation

The output produced by the convolution increases data dimensionality because of the big number of filters. To avoid this drawback and perform a better data summarization, *pooling layers* are inserted.

A pooling layer divides the input tensor into squared sections and for each of them it extracts a single value. The result will be a tensor whose size is given by the number of sections. The most commonly used pooling operation is *maxpool*. This process simply takes the maximum value as the representative one of each region. For example, if we have an input tensor of shape $[4, 4]$, we could divide it into four sections and take the maximum value for each of them.

Input tensor:

$$input = \begin{bmatrix} 0 & 1 & 3 & 0 \\ 2 & 0 & 2 & 1 \\ 0 & 1 & 3 & 0 \\ 4 & 0 & 4 & 1 \end{bmatrix}$$

Divide into sections:

$$sections = \left[\begin{array}{cc|cc} 0 & 1 & \mathbf{3} & 0 \\ \mathbf{2} & 0 & 2 & 1 \\ \hline 0 & 1 & 3 & 0 \\ \mathbf{4} & 0 & \mathbf{4} & 1 \end{array}\right]$$

Result of max-pooling:

$$output = \begin{bmatrix} 2 & 3 \\ 4 & 4 \end{bmatrix}$$

3.2.4 Fully connected layer

The last layers of a convolutional neural network are designed to produce a single vector with the classification scores. The sum of these values must be equal to one in order to assume the meaning of a categorical distribution. Each score represents the probability of belonging to a particular class. To perform this task common *fully-connected* feed-forward networks are used. They can be single or multi-layer.

In order to ensure that the output values will add up to one, we use the *softmax* activation. Unlike the other functions, softmax analyzes all the neurons of a layer, instead of one at a time. The activation of a neuron z_j is defined as follows:

$$softmax(z_j) = \frac{e^{z_j}}{\sum_{i=0}^{N-1} e^{z_i}} \tag{3.10}$$

where z_i are the output values of all the neurons of the last layer.

3.3 The complete architecture

In the previous sections we have described the main blocks which can be used to compose a convolutional neural network. The typical architecture can be structured as follows.

First the input image is serialized to an input tensor. Then data flows through a composition of convolutional and pooling layers. Smaller networks can have only one or two layers of this type. Bigger ones can have even ten or more. Of course, the higher is the number of layer, the longer is the training time. Finally, one or two fully connected layers are added to perform the actual classification. Figure 3.8 shows an example of complete convolutional neural network.

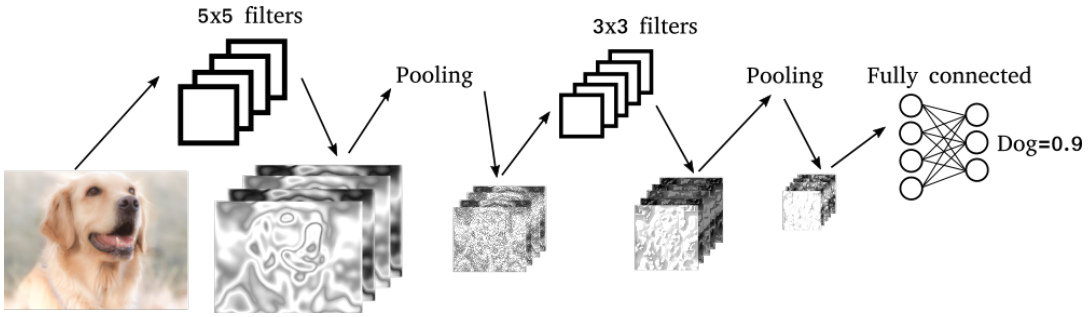

Figure 3.8: An example of CNN with two convolutional layers.

3.4 Training algorithms and regularization

The training phase of convolutional neural networks is performed with a generalized version of backpropagation. Samples can be presented one at a time or in batches. For each of them the error gradient is computed from the fully connected network back to the convolutional layers. The weights are updated to minimize the error. When all the samples of the training set have been presented to the network an epoch is completed. After each epoch the system can evaluate the model on the test set. This validation error will slowly decrease until the process reaches its maximum accuracy.

There are many variants of this gradient descent algorithm. Some of them are the *Stochastic Gradient descent*, *Adagrad* [23], *AdaDelta* [24] and *ADAM* [25]. The main differences can be found in the usage of *momentum*. Momentum avoids rapid changes of direction in the update of the weights. This expedient can also prevent situations where the algorithms gets stuck into a *local minimum* of the error function and guarantees better results. Figure 3.9 shows the gradient descent for two algorithms with and without momentum.

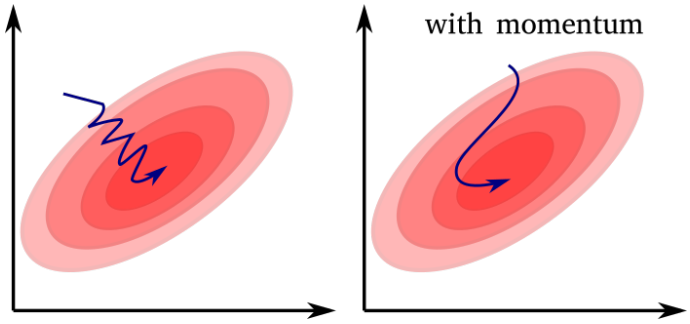

Figure 3.9: The algorithm on the right makes use of momentum which allows a smoother descent, without zig-zagging.

3.4.1 Early Stopping

After some epochs the neural network will start overfitting the training set: the accuracy over the test data will begin to decrease. *Regularization techniques* are designed to solve this issue. Among these we can find a procedure called *early stopping*. It simply terminates the training process when the accuracy on the test set starts to get worse. The complete algorithm is described in the next lines.

Algorithm 1 Early Stopping

```
procedure TRAINING(trainingSet, testSet)
    bestErr ← ∞
    for epoch = 0 to max-1 do
        for all sample ∈ trainingSet.samples do
            cnn.fit(sample)
        error ← cnn.evaluateTestSet(testSet)
        if error < bestErr then
            bestErr ← error
            bestModel ← cnn
        else
            break;
```

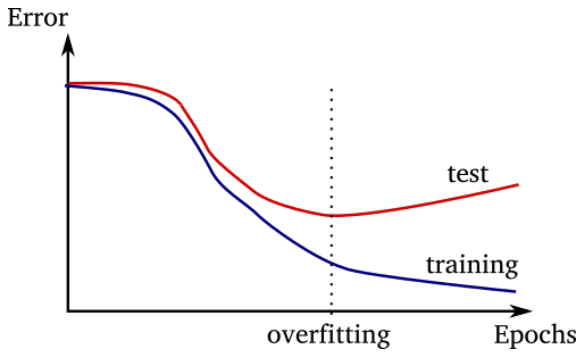

Figure 3.10: Overfitting example. The curves represent the error loss evaluated on the test set (the red one) and on the training data (the blue one). When the red curve starts growing, the model is overfitting the training set and test accuracy starts getting worse.

3.4.2 Dropout

Dropout is another important regularization technique thought to avoid the overfitting issue [26]. The idea behind this methodology originates from *bagging*. Bagging is a method which is realized by training m models on m different datasets, called *bootstrap samples*. Bootstrap samples have the same size of the original training set and their observations are extracted from this one by sampling uniformly and with replacement. Some observations can be duplicated, but the m datasets will be statistically different. Each model is trained on a single bootstrap sample and will learn different parameters. When testing is performed, the result of the m models over a single sample is combined by averaging or voting. This avoids overfitting because the models are trained only on a subset of the original data.

Dropout is inspired by this technique, but works with a single neural network. Instead of partitioning the dataset, at each training iteration we select a random subset of the neural network connections. The new observation will adjust only those weights, while the others will stay unchanged. The proportion of connections that is randomly selected every time is specified by a parameter of the algorithm, which is typically set between 0.2 and 0.5 (20% and 50% of the connections). In this way the neural network won't overfit because each partition is trained on different data. Dropout can be applied to CNNs to both the fully connected and the convolutional layers.

3.4.3 Dataset augmentation

To improve the learning process, samples of each class should be presented to the network with the same frequency. If this rule were not satisfied, the classes with a smaller cardinality would be hardly recognized by the model. When the dataset contains classes with many more elements than others, we say that it is *unbalanced*.

Datasets can be re-balanced with a generative process called augmentation. Images are randomly transformed with horizontal flips, zoom in and out, changes of lightness and

contrast to generate new samples. This can fill the gap between the smaller classes and the bigger ones. Augmentation can also be used with balanced datasets to increment their size. It can be necessary in some cases where the examples are not sufficient and the model overfits too easily the training data. The model trained for this thesis project works with a small and unbalanced dataset. For this reason augmentation is performed to achieve better results.

3.5 State of the art models

3.5.1 Classification

Chapter 2 presented how the classification task is defined for image recognition. We briefly review now some of the standard convolutional neural network architectures which were proposed in the last years for this purpose. One of the most popular model, called *Alexnet*, was presented by Alex Krizhevsky in 2012 [4]. This neural network contains five convolutional and three fully-connected layers. The filter shapes vary from 11×11 to the smaller 3×3. The last fully connected layer performs the classification by using the softmax activation function. Alexnet was tested with the *ImageNet LSVRC-2010* contest for 1000 different classes.

With the developing of big data techniques and more powerful GPUs the size of the most recent models increased considerably. The *VGG* architecture is characterized by 16 layers for the smaller version and 19 for the deeper one [5] . This CNN is only built with small 3×3 filters and 2×2 pooling layers.

Google researchers proposed *GoogleNet*, a convolutional neural network built with a new kind of internal module, called *inception layer* [6]. Instead of using a single filter size, the inception layer applies to the input tensor either 1×1, 3×3 and 5×5 filters. The results produced by the different filters are then put together with the *concatenation layer*. In this way the architecture tries to achieve better results by combining more accurate features

with coarser ones. Figure 3.11, extracted from the original GoogleNet paper, depicts the structure of this neural network.

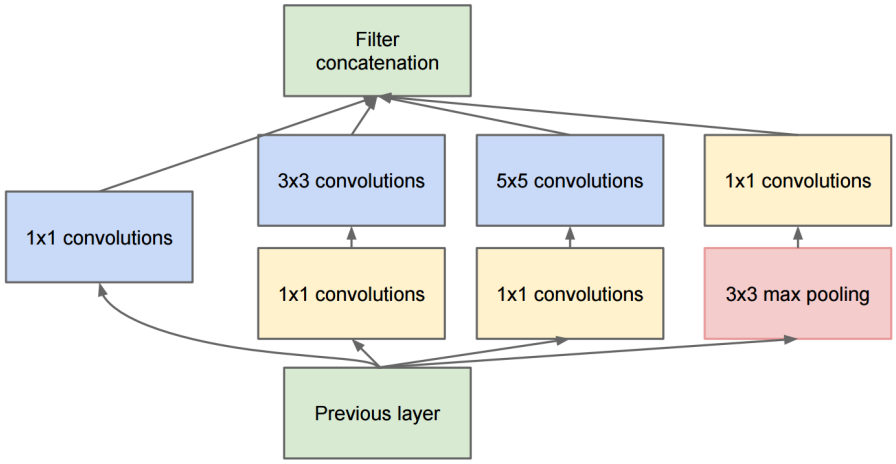

Figure 3.11: GoogleNet architecture [6].

3.5.2 Detection

Detection refers to the task of finding and classifying rectangular bounding boxes, each containing one of the objects in the image. Selecting for every position and size all the possible sub-regions of the input image and then applying a CNN to classify them is too expensive in terms of computational time. This method was called *sliding windows* and was soon overcome by the newer systems.

The first efficient solution was given by the model proposed in the paper *Rich feature hierarchies for accurate object detection and semantic segmentation* [7]. The architecture, called R-CNN (*Regions with CNN features*), is composed by three modules: region proposal, feature extraction and SVM classifiers. The region proposal computes the positions of the candidate bounding boxes which will be analyzed by the convolutional neural network. Instead of trying every possible region, this module outputs only the ones which more likely contain an object. This can be done with different methods, for example with

selective search they perform a hierarchical color-based segmentation of the image to extract the regions and the candidate bounding boxes. The feature extraction module is composed by a convolutional neural network without the fully connected layer. The actual classification is performed by the last SVMs module.

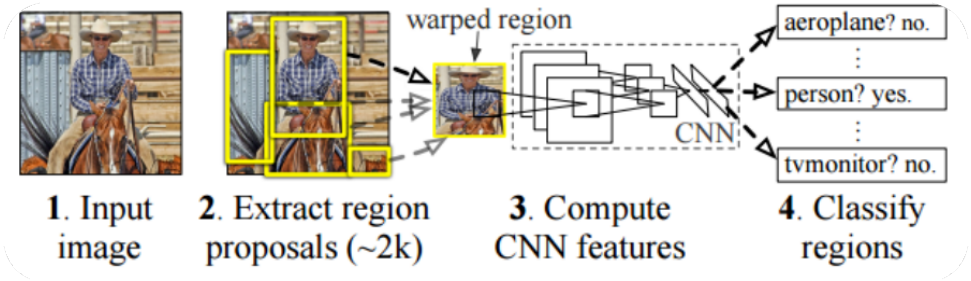

Figure 3.12: R-CNN, Regions with CNN features [7].

One of the main efficiency issues of the R-CNN architecture is the need for applying the convolutional neural network to every sub-region being classified. A remarkable improvement was achieved with the *Fast R-CNN* model [8]. The new approach executes the CNN over the whole image, extracting a feature map which describes the entire scenery. This map has the same shape of the input image, but each pixel presents the feature values instead of containing the color information in the BGR encoding. As the previous method, the region proposal module computes the candidate bounding boxes. For Faster R-CNN, instead of extracting the proposed regions from the original image, this operation is performed directly to the feature map. With the operation called *ROI (region of interest) pooling*, the feature map regions are subsampled and resized to squared tensors. The last step applies a fully connected neural network to classify each of the produced tensors. The presented method is faster than the simple R-CNN model, since the convolutional neural network producing the features is applied only once.

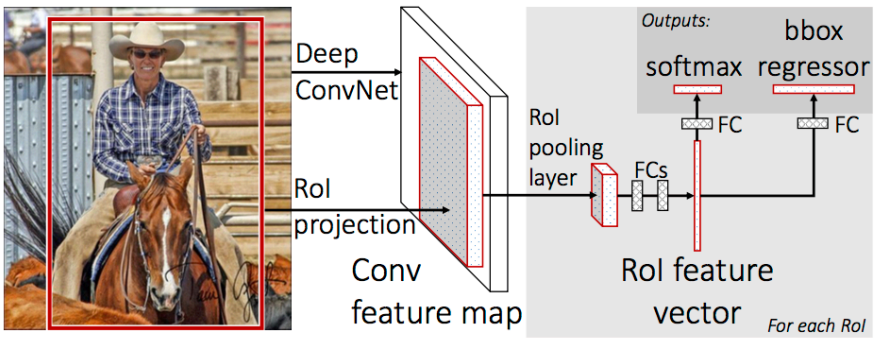

Figure 3.13: Fast R-CNN [8].

The last bottleneck with the detection task was the slowness of the region proposal methods. The solution was proposed in the paper *"Faster R-CNN: Towards Real- Time Object Detection with Region Proposal Networks"* [9], in 2015. Instead of computing the candidate bounding boxes from the original image, the Faster R-CNN uses directly the feature map extracted by the CNN. The region proposal works with a neural network (*RPN*, region proposal network) which takes as input the feature map and whose output represents the position and sizes of the proposed bounding boxes. Afterwards, as shown for Fast R-CNN, the fully connected layer classifies the squared tensors produced with the ROI pooling method.

Another state-of-the-art model which makes use of these concepts is called YOLO [10]. It is built with a single convolutional neural network. The fully connected module, instead of producing a score vector with the softmax function, outputs directly a matrix with the classified objects. Each cell of the matrix contains a set of rectangular bounding boxes, with their size, position and class scores.

3.5.3 Semantic Segmentation

The semantic segmentation task consists of assigning the class scores to each pixel of the image. Three of the most popular models proposed in the last years are DeepLab [11], SegNet [12] and PSPNet [13]. They are fully built with convolutional neural networks. These models are characterized by being divided in two modules, one for computing the features and the other for generating the output segmentation.

The first module, called *encoder network*, is a CNN which is responsible for computing the feature map. The encoder network contains both convolutional and pooling layers to generate a tensor which is smaller in width and height with respect to the input image. The depth of the feature tensor is established by the number of applied convolutional filters. The second half of the architecture, the *decoder network*, is designed to upsample the feature map and obtain the segmented output image. Each pixel is associated to the class scores generated by the softmax activation function. The decoder network is composed of trainable convolutional filters and upsampling modules.

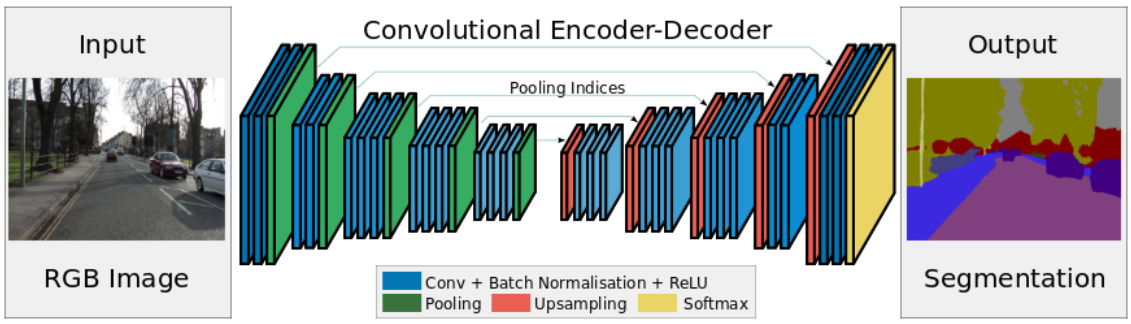

Figure 3.14: Segnet Architecture [12].

Part II

Model implementation

Chapter 4

Image Datasets

4.1 Choosing the dataset

In chapter 2 and 3 we pointed out how image segmentation and classification models require big datasets for training and testing. Many resources of this kind can be found on the web. These datasets contain images that can be annotated with single or multiple labels. The case with single label has been studied for many years with the purpose of creating models capable of predicting the image class given its pixels. Multi-label datasets propose more interesting challenges because models must learn to find and classify objects into a scene. These kinds of image collections can provide the object positions with rectangular regions, usually called *bounding boxes*. They can also specify polygons around the object boundings or even provide pixelwise segmentation. Currently, a lot of competitions are organized to evaluate models capable of locating objects and segmenting images. As we said in the previous chapters, the operation of assigning a label to each pixel is called *semantic image segmentation*. This is one of the most complex tasks with image recognition and understanding.

Choosing the right dataset is a crucial point that must be faced out before developing such algorithms. For this reason a set of the most commonly used image collections are

described in the following paragraphs.

4.2 CIFAR-100

CIFAR-100 is a relatively rich dataset composed of 60,000 images. The classification problem defines 20 superclasses divided into 100 subgroups. Each class contains 500 images for training and 100 for testing. All of them have the same size of 32×32 pixels.

The collection of samples is easy-to-use, since all images have the same size and they are stored as simple binary files containing the label and a row vector for the pixels. The class hierarchy with two levels is an interesting feature that could be exploited by a labeling model working with multiple levels of abstraction. The main limitation of this dataset is that each image contains only one object, not allowing reasoning with multi-label contextual information. Table 4.1 shows all the 100 classes and their hierarchy.

Table 4.1: CIFAR-100 classes

Class	*Sub-classes*
aquatic mammals:	beaver, dolphin, otter, seal, whale
fish:	aquarium fish, flatfish, ray, shark, trout
flowers:	orchids, poppies, roses, sunflowers, tulips
food containers:	bottles, bowls, cans, cups, plates
fruit and vegetables:	apples, mushrooms, oranges, pears, sweet peppers
household electrical devices:	clock, computer keyboard, lamp, telephone, television
household furniture:	bed, chair, couch, table, wardrobe
insects:	bee, beetle, butterfly, caterpillar, cockroach
large carnivores:	bear, leopard, lion, tiger, wolf
large man-made outdoor things:	bridge, castle, house, road, skyscraper
large natural outdoor scenes:	cloud, forest, mountain, plain, sea
large omnivores and herbivores:	camel, cattle, chimpanzee, elephant, kangaroo
medium-sized mammals:	fox, porcupine, possum, raccoon, skunk
non-insect invertebrates:	crab, lobster, snail, spider, worm
people:	baby, boy, girl, man, woman
reptiles:	crocodile, dinosaur, lizard, snake, turtle
small mammals:	hamster, mouse, rabbit, shrew, squirrel
trees:	maple, oak, palm, pine, willow
vehicles 1:	bicycle, bus, motorcycle, pickup truck, train
vehicles 2:	lawn-mower, rocket, streetcar, tank, tractor

4.3 ImageNet

The *ImageNet* dataset [14] organizes images according to the WordNet hierarchy. WordNet is a big lexical database defining all the English nouns, verbs and adjectives, connected with semantical relations. The samples of this dataset vary from animals to landscapes and indoor objects. Differently from CIFAR-100 the number of classes is quite big, with thousands of samples each. All of them are quality-controlled and human-annotated. The database only provides thumbnails and URLs of the images because they are protected by copyright, since they have been extracted from the web. For this reason the collection cannot be used for commercial purpose. Another issue is the absence of a single file download: data must be indeed downloaded in separate blocks. Even if this dataset is very big and contains a lot of classes, it doesn't meet the requirement of having multiple objects for a single image.

The importance of ImageNet is reinforced by the annual *ImageNet Large Scale Visual Recognition Challenge* (*ILSVRC*), active since 2010, where research teams can submit algorithms for image classification, trying to achieve the best trade-off between efficiency and accuracy.

4.4 Pascal VOC (visual object classes)

Pascal VOC (Visual Object Classes) [15] [16] presents a set of 14,743 images, annotated with bounding boxes. Each image can contain multiple objects. A subset of the samples is also annotated with pixelwise segmentation of the regions being classified.

From 2005 to 2012 the Pascal VOC project organized yearly challenges. There were three main types of competition: classification, detection and segmentation. Classification was concerned about predicting the presence/absence of an object of a particular class in the test image. Detection was about predicting the bounding box and the label of each object in the test image. The version proposed in 2012 is characterized by 20 target classes,

divided into 4 groups. Table 4.2 shows the taxonomy in question. Figure 4.1 depicts in the upper part an example of pixelwise segmentation then in the lower side two samples with bounding boxes around objects.

Table 4.2: Pascal VOC classes

Class	*Sub-classes*
Person	person
Animal	bird, cat, cow, dog, horse, sheep
Vehicle	airplane, bicycle, boat, bus, car, motorbike, train
Indoor	bottle, chair, dining table, potted plant, sofa, tv/monitor

Figure 4.1: Pascal VOC dataset

The interesting feature of Pascal-VOC is the presence of bounding boxes, which could be used to train models like *Fast R-CNN* to find some objects into an image. Since the

number of classes is not very high and each sample contains only few labeled regions, this dataset doesn't allow semantic reasoning over the elements of a scene. For this reason it doesn't meet the requirements for models like the one proposed in this thesis.

4.5 NYU Depth Dataset V2

The *NYU-Depth V2* dataset [17] contains a big video collection of indoor scenes, recorded with RGB-depth Microsoft Kinect cameras. Depth can be used by classification models to better understand the image they are requested to analyze. The most interesting subset of NYU-Depth is composed of 1,449 densely labeled samples, for about 2.8 gigabytes of data. The remaining images are still unlabeled for now.

Each single scene is characterized by multiple objects, that are pixelwise annotated. The annotation related to a sample is a *HxWxN* matrix, where *N* is the number of classes and *H*, *W* the size of the image. The dataset is divided into folders, where each one corresponds to a different scene being filmed. Each image is associated with the timestamp in the video sequence. Annotations are represented in Matlab format, which is not very portable and requires a bit of preprocessing before being used in custom applications.

Concluding, NYU-Depth is a useful resource for training semantic image-segmentation models because it is multi-object and regions are pixelwise annotated. Figure 4.2 depicts some examples of annotated images. For each sample the first column is the original BGR image, the second one is the color representation of the depth channel, the third one depicts the pixelwise segmentation.

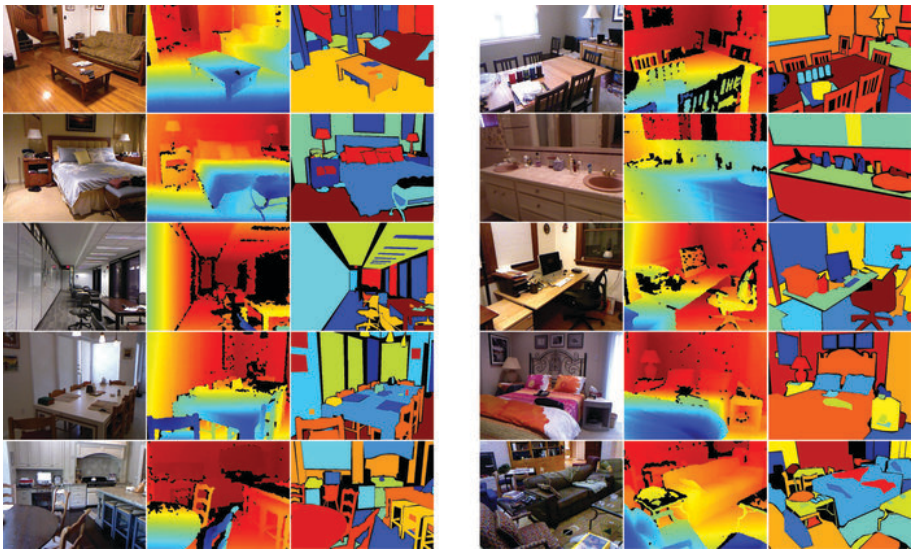

Figure 4.2: NYU Depth Dataset

4.6 Our choice: Sun09

Presented by researchers of Massachusetts Institute of Technology at IEEE CVPR 2010 (Computer Vision and Pattern Recognition conference) [2], *Sun09* provides augmented annotations for Pascal-VOC dataset. It contains 12,000 images, divided into different scenery types, for example kitchen, office, street or bathroom.

Annotations are represented by polygons around the boundings of each objects being recognized. They are handmade and very accurate, even smallest objects like handles, faucets and glasses are labeled. This dataset is perfect to realize hierarchical models, grouping elements into a complex taxonomy and allows studying the contextual information given by the presence of multiple objects into an image. These reasons make it perfect for our experiments. The Sun09 paper also presents a model which exploits this contextual information to improve the results of the baseline object recognition methods [2]. The system described in the paper is a *probabilistic model* and differs from our proposal, which is a *rule based* algorithm.

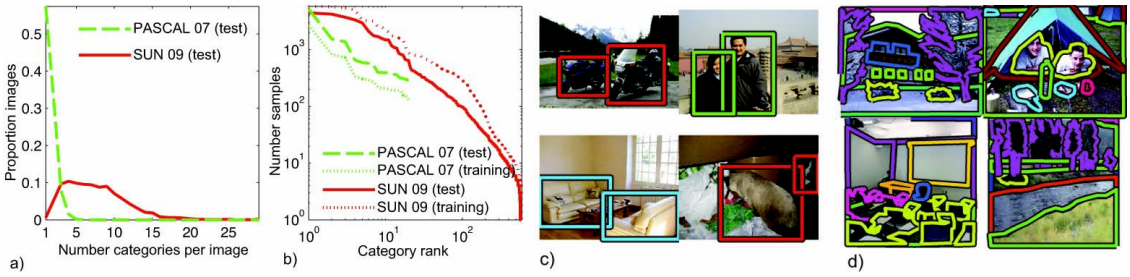

Figure 4.3: Pascal VOC compared to Sun09

After download, Sun09 presents two main folders: Images and Annotations. The former contains the images in jpeg format, which can have different sizes and whose name describes the environment being represented. This could be useful to filter samples by type. The latter contains XML annotations, where each object is described by a label and a list of points, defining a polygon around its boundings.

Picture 4.3 shows the differences between annotations for Pascal-VOC (c) and Sun09 (d). The latter presents a more accurate segmentation if compared with Pascal-VOC. In listing 4.1 there is an example of XML description representing an object labeled as *armchair*, followed by its boundings coordinates.

Listing 4.1: Sun09 annotations

```
<object>
        <name>armchair</name>
        <date>05-Oct-2009 21:55:32</date>
        <id>7</id>
        <polygon>
                <pt><x>12</x><y>156</y></pt>
                <pt><x>32</x><y>210</y></pt>
                ...
        </polygon>
</object>
```

Chapter 5

Base Model

5.1 Dataset Preparation

In chapter 4 we have described the reasons why *Sun09* is the correct choice for our experiments. We will exploit its characteristic of presenting multiple objects for each scene to work with contextual information and correct convolutional neural network results. The model proposed in this thesis will not make use of the entire data, but only of a subset. We will present here the main phases of data preparation before its usage in our experiments.

The main steps that will be analyzed are: scenery selection, class names cleaning, creation of *k-fold* partitions, object extraction by reading XML annotations, samples augmentation and batch balancing.

5.1.1 Sceneries and classes

Sun09 is stored into the file system with two main folders. Their names are `Images` and `Annotations`. The first one contains the set of all the image samples, each of them representing a scenery which can be indoor or outdoor. These images are in *jpeg* format and their names are related to the type of scenery. We can find for example "*kitchen_i.jpg*", "*road_i.jpg*" or "*office_i.jpg*". Each image contains the objects that will be recognized by

our model. For example inside a "*road*" scenery we could find objects like "*street*", "*sky*", "*road sign*" or "*car*".

For each scenery file the `Annotations` folder contains an XML document with the same name. This stores the image segmentation by means of a set of polygons which specify the object boundings. Each object is characterized by a class label that will define the goal of our algorithm. Since they are manually annotated, object of the same type can have slightly different labels. For this reason the first step of data preparation is the label cleaning.

We created 18 classes from two indoor scenery types: "*kitchen*" and "*office*". Each class is represented by set of labels which refers to objects of the same type. An XML document, called `classGroups.xml`, defines these name collections for all the 18 classes. We report here a partial view of this file.

Listing 5.1: classGroups.xml

```
1 ...
2 <group name="door">
3         <el>door</el>
4         <el>door occluded</el>
5         <el>door crop</el>
6         <el>open door</el>
7         <el>doors crop</el>
8 </group>
9 ...
```

Listing 5.1 shows how classes are defined. In this example, all the objects with a label which is contained into the `<group>` element will be assigned to the class "*door*".

5.1.2 k-fold partitions

The Java class responsible for extracting the objects from the scenery files is called `SunDBcreationPipeline`. The algorithm selects all the images whose name contains the words "*kitchen*" or "*office*" and prepares a list to index these files. We decided to use a k-fold cross-validation with $k = 10$ partitions in order to evaluate our model. For this

purpose the list of the selected scenery files is randomly splitted into 10 parts, numbered from 0 to 9. Each model will be trained on 9 of them and evaluated on the remaining one. In particular we call $model_i$ the model that will be evaluated on $partition_i$. The subdivision of the file names is stored into the `datasetConfig.xml` file. Figure 5.1 depicts this dataset organization.

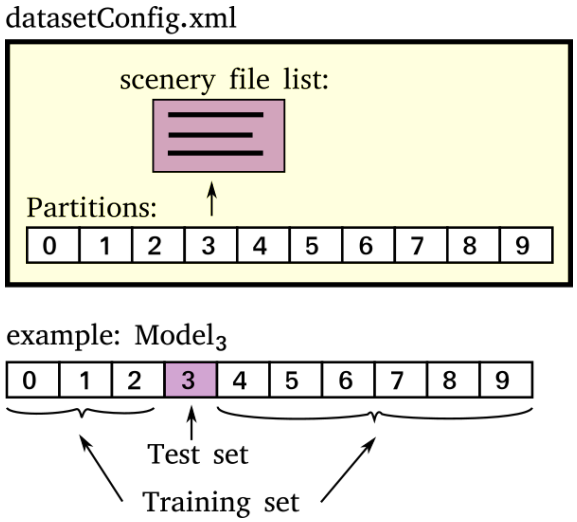

Figure 5.1: K-fold subdivision of the generated dataset. The example shows how $model_3$ will be evaluated on $partition_3$

The selected sceneries are 510. Each partition contains about 50 of them. Indeed, the subdivision with $k = 10$ allows us to use the 90% of data as training set. The number of objects for each class is slightly different among each of the partitions. The explanation is that the sceneries are randomly splitted into 10 parts without caring about the samples distribution inside them. The objective is to divide training and test set considering the scenery as unit, instead of a single object. This is more correct to evaluate the final model, which works with an entire scenery to analyze the contextual information. Table 5.1 shows, for each class, the total number of objects that can be found into the generated partitions. As we can see the dataset is highly unbalanced and provides a relatively small number of samples, which makes our task quite complex.

Class	Training	Test	Total
wall	879	98	977
cupboard	611	68	679
chair	384	43	427
window	376	42	418
floor	359	40	399
surface	298	33	331
plant	291	32	323
ceiling lamp	276	31	307
ceiling	215	24	239
sink	212	23	235
bottle	179	20	199
desk	137	15	152
screen	128	14	142
handle	123	14	137
door	111	12	123
oven	111	12	123
outlet	98	11	109
keyboard	88	10	98

Table 5.1: Class cardinalities relative to the 510 sceneries. Training and test values are computed with the 90% and 10% of the total: these only represent the average number of objects that we could find in one of the 10 k-fold subdivisions.

5.1.3 Sample extraction

The sample extraction phase consists of producing an image for each of the object samples contained into a scenery file. Each scenery presents typically more than one object. This operation is achieved by reading the XML annotations and extracting the sub-images defined by the object boundings. Since convolutional neural networks work with squared images of a predefined size, we also need to resize the rectangular sub-images to squares.

The XML structure of the scenery annotations is built with a list of `<object>` elements, each of them containing the object label and the coordinates of the points which define a polygon around its boundings. The algorithm computes the smallest rectangle which fits around this polygonal region. Listing 5.2 shows a view taken from an example of annotation file.

Listing 5.2: Example of scenery annotations

```
...
<object>
        <name>open door</name>
        <date>05-Oct-2009 21:47:59</date>
        <id>1</id>
        <polygon>
                <pt><x>19</x><y>209</y></pt>
                <pt><x>24</x><y>190</y></pt>
                ...
        </polygon>
</object>
```

Once all the polygons have been transformed to rectangular bounding boxes, the system loads the *jpeg* file of the given scenery and extracts the pixel subregions. Each object sample will be represented in memory by a label, a bounding box and the raw pixels.

We created a Java class, called `Scenery`, which contains the raw pixels of the scenery image and the list of the bounding boxes computed from the polygonal regions. Each bounding box is stored as `RectangularROI` (Region Of interest) class, which specifies its size and position in percentage with respect to the complete picture.

After this point every `RectangularROI` is extracted from the whole image and stored as a `Sample` instance, with a label and the pixels of the subregion. In this way, from a `Scenery`, we obtain a `Sample` instance for each object. The `FlatImage` class, contained into `Scenery` and `Sample`, provides the method to extract rectangular subregions from a picture and stores the pixels into the form of a serialized tensor, as we described in section 2.3.2.

The last transformations required to prepare tensors before providing them to a convolutional neural network are the resizing to squared images and data *normalization*. The second operation is required since each of these tensors has been extracted from a BGR image and for this reason the pixel values vary from 0 to 255. These numbers are normalized between 0 and 1 with a simple division by 255.

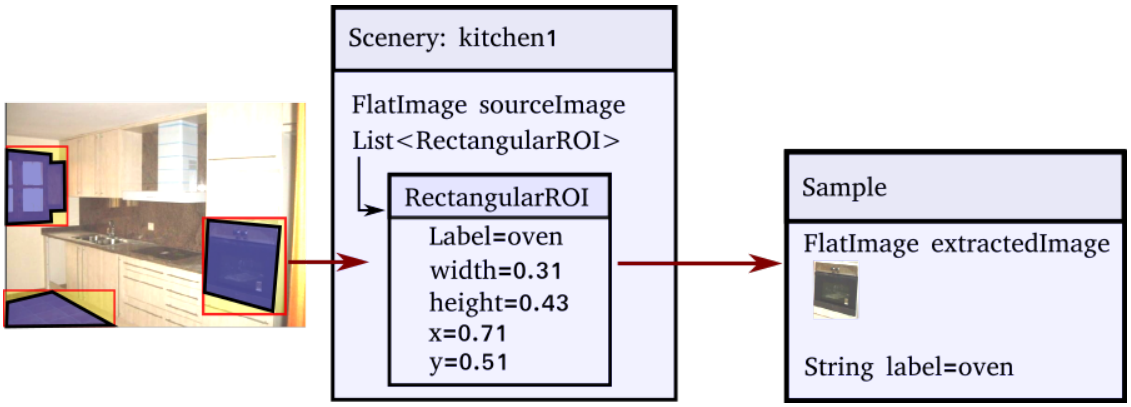

Figure 5.2: Sample extraction. The list of bounding boxes for a segmented image is firstly stored in a `Scenery`, then each subregion is extracted and transformed to a `Sample`. The `Sample` objects are ready to be provided to the neural network.

5.1.4 Batch creation and balancing

Before training the neural network each k-fold partition has to be divided into batches. A *batch* is a group of data samples that will be fitted with a single training iteration. When using *gradient descent* algorithms, providing more than one sample at once is a way of describing better how data is distributed. Larger batches will give better results because the gradient is computed on many samples from all the classes. Using only one example would create a gradient which takes care only of that particular instance, making the training process slower and less accurate.

Each batch must be balanced: all the 18 classes must provide the same number of examples. For our experiments the batch size was set to 486, giving 27 samples for each class. Every k-fold partition is composed of 4 batches. The technique we adopted to balance the class samples is called *data augmentation*. When more examples are needed for smaller classes, these are automatically generated from the existing ones with a set of transformations. We first applied random horizontal flips, then some changes of lightness and contrast, performed by choosing two random values *gain* and *bias*. Pixels are updated

with the following rule:

$$newValue = bias + gain \cdot oldValue \tag{5.1}$$

We analyze the algorithm for batch balancing by focusing on how it works with a single k-fold partition. The procedure is the same for all the 10 partitions. For each scenery, all the object samples which belong to one of the 18 classes are extracted. Every sample is normalized and converted to tensor, then stored into a bucket with the corresponding class label. Once the process is finished we obtain 18 buckets, each serialized to a binary file. The bucket with the greatest number of elements is selected as *reference*. All the other buckets will be augmented with new generated samples until they reach the size of the reference: *ref.numElements*. We obtain one bucket for each class, all with the same size. The total number of instances, including the generated images, will be equal to the number of elements of the reference bucket multiplied by the number of classes:

$$numObjects(partition_i) = ref.numElements \cdot numClasses \tag{5.2}$$

With this procedure all the classes will be represented into a batch by the same number of elements. The number of batches is computed as follows:

$$numBatches(partition_i) = ceil\left(\frac{numObjects(partition_i)}{batchSize}\right) \tag{5.3}$$

where $batchSize$ is the number of elements contained into a single batch. In our experiments the class with the highest cardinality is *wall*, with its average of 98 samples for each k-fold partition. The number of batches will be:

$$numBatches(partition_i) = ceil(98 \cdot 18\ /486) = ceil(3.6) = 4$$

with $numClasses = 18, batchSize = 486, numSamplesPerclass = 486/18 = 27$.

For each k-fold partition we obtained 4 batches, each of them containing 486 images, 27 for each class. The reference bucket *wall* will only contain original elements, while the other ones, among the 27 elements, will also include generated images. The created dataset is *600 MB* in size and can be stored into the main memory for a faster training process.

This is how the whole algorithm works:

Algorithm 2 Batch creation and balancing

```
procedure CREATEBATCHES
    for b = 0 to numBatches-1 do
        for i = 0 to numSamplesPerClass-1 do
            for c = 0 to numClasses-1 do
                batch[b].add(bucket[c].getSample())
```

For every batch the algorithm extracts 27 elements from each bucket. The method `bucket[k].getSample()` takes a sample from the bucket of class k or generates it if no more images are available. Figure 5.3 shows an example of k-fold partition with the described organization.

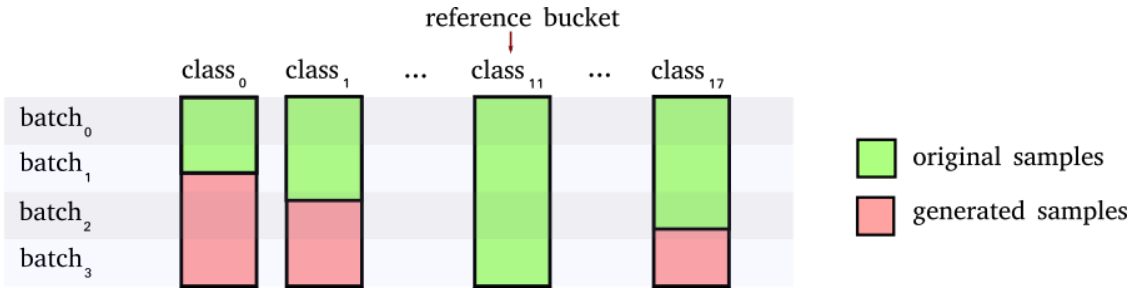

Figure 5.3: Batch balancing: this example shows, for a given k-fold partition, the subdivision of the 18 bucket into 4 batches. The reference bucket contains only original samples, while the others include generated images (highlighted in red).

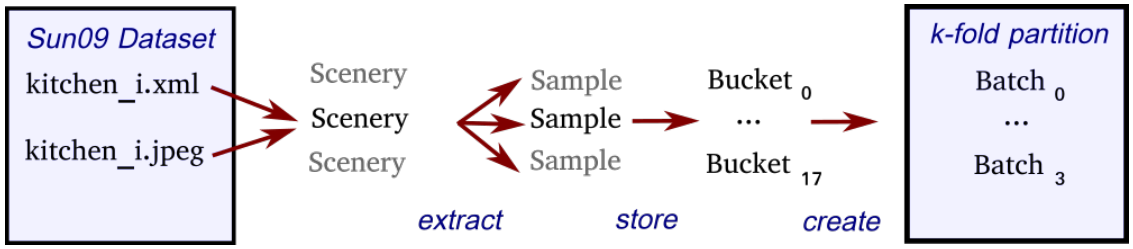

Figure 5.4: Batch creation pipeline. From sceneries to batches.

5.1.5 File system organization

Before continuing with the description of our experiments, we give an overview of how the training and testing platform is organized in the file system. The environment is divided into two folders. The first one, called `cnnSamples`, contains two XML files describing the class labels subdivision and how the scenery files are splitted into the k-fold partitions. This folder also includes, in one directory for each partition, the generated batches that will be used to train the neural networks. The second folder is named `cnnModel` and contains the 10 trained convolutional neural networks. For each model the evaluation results over the training and test sets are stored into the files `trainingSet.cnnr` and `testSet.cnnr`, respectively. These are required to save time, instead of executing the neural networks every time we need the classification outputs for each scenery, for example when varying the threshold to compute the mAP score or when running the *Smart Model*. The `cnnModel` folder also contains an XML file which stores the configuration and the learning curves for each of the trained convolutional neural networks. Figure 5.5 shows the described file system organization.

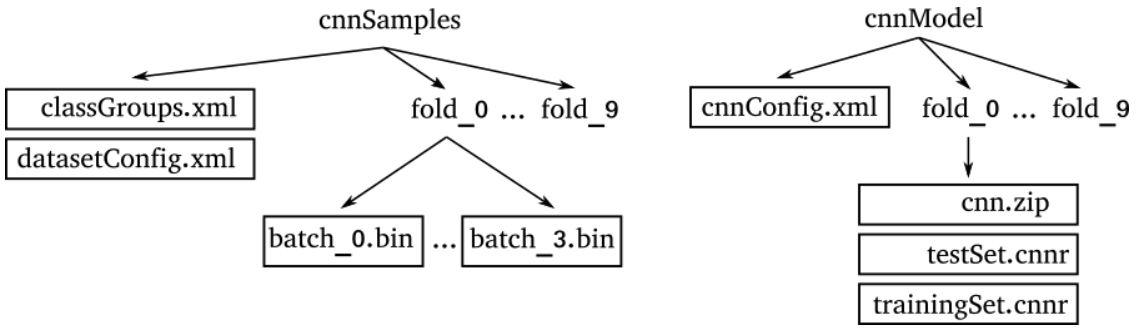

Figure 5.5: Directories for the k-fold evaluation platform.

5.1.6 BGRC: context layer

The main problem with convolutional neural networks is their usage as local classifiers on a subregion of the input image, the scenery in our case. This region is resized

to a squared matrix, loosing the information about its original shape. We tried to add a fourth layer in addition to the BGR channels, obtaining a tensor with rank 3 and shape $[4, height, width]$. We call this format *BGRC* image, where *C* stands for *context layer*.

This layer is a matrix which represents the information about the size of the selected subregion and its position inside the original scenery. Size and positions are computed in percents with respect to the input scenery. For example a bounding box position with $x = 0.5$ and $y = 0.5$ will be located in the middle of the picture. The four values $x, y, width, height$ which describe the selected bounding box are represented with grayscale squares inside the context layer. Lighter pixels will depict higher percent values: the neural network can use the contextual information as if it were just another color layer. Training some CNN architectures with this approach gave better results than simply using BGR images. For this reason all the final model presented in the next sections will use the BGRC encoding. Figure 5.6 shows how the context layer would look if represented with a grayscale image.

To understand how this context layer can help neural networks in the recognition phase we provide some examples. Considering the class *ceiling*, objects of this type can be found in the scenes at higher vertical positions and their shape is typically larger than taller. Floors have the same shape of ceilings, but can be found more likely in lower vertical positions. Objects of class *door* will be taller than larger and *ceiling lamps* will be smaller and in higher vertical positions. For these classes, figure 5.7 shows some graphical examples of context layer. Without knowing the real meaning of the squares drawn into the context layer, one can easily recognize the separation between these classes, simply because the grayscale images look different.

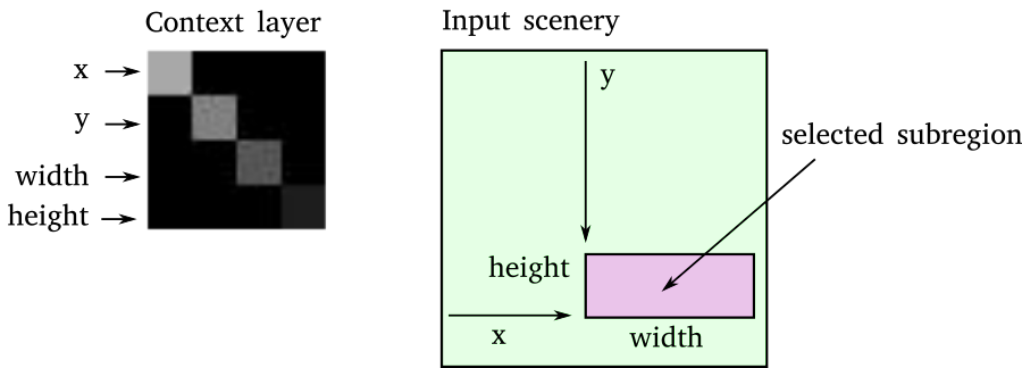

Figure 5.6: BGRC format: each sample contains a context layer describing its size and position with respect to the input scenery

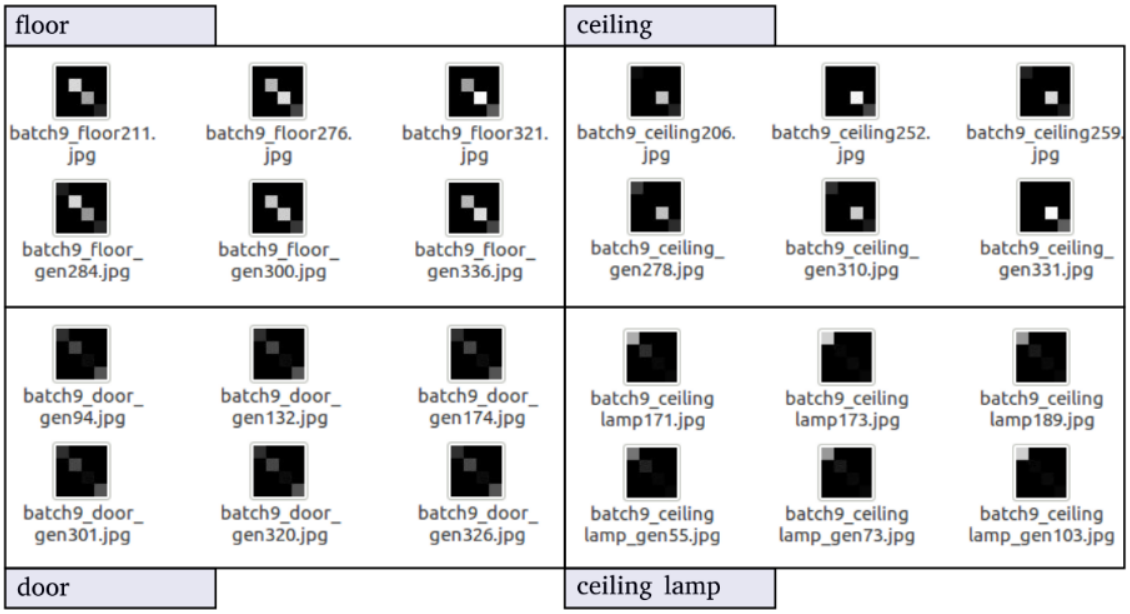

Figure 5.7: Context layer for different classes. Note how images look different without thinking at the real meaning of the grayscale map.

5.2 Training pipeline

We describe now how the neural networks are trained, how they are evaluated and which are the architectures we tried for our experiments. The Java class responsible for training the CNNs is called `NetTrainerPipeline`. We built it as an automatic system able to train all the 10 models and test their performances. It uses the regularization technique called early stopping: at the end of every training epoch the algorithm computes the *mAP* score related to the evaluation on the test set. If the value is better than the previous one, the best CNN model is replaced, otherwise the old one will be kept. Since the *mAP* score can have some wiggles between the training epochs, the process doesn't stop at the first sign of overfitting. It will instead continue until it reaches a predefined maximum number of epochs.

`NetTrainerPipeline` trains one model at once, providing to the network all the batches of the training set. The procedure works as described in algorithm 3.

Algorithm 3 Training pipeline

```
procedure TRAINMODEL(m, maxEpochs)
    bestMAP ← 0
    for e = 0 to maxEpochs-1 do
        for b = 0 to numBatches-1 do
            for p = 0 to numKfoldPartitions-1 do
                if p != m then
                    model[m].fit(partition[p].batch[b])
        mAP ← model[m].evaluate(partition[m])
        if mAP > bestMAP then
            bestMAP ← mAP
            bestModel ← cnn
        else
            break;
```

The pseudocode shown above has the task of training a single model *m*. The neural network fits all the partitions except for the one with index *m* because it represents the test set associated to the selected model. The XML file `cnnConfig.xml` stores the current batch,

partition, epoch and the mAP value for the test set. Each training epoch is associated to a timestamp, making possible to reconstruct the learning curve after the process is finished. In this way if something goes wrong, a backup of the temporary data is ready to restart the process. Terminated the training for all the 10 models, the system automatically takes the best epoch for each of them and generates the classification results for every scenery in the test set.

The classified images are stored into the `testSet.cnnr` files, one for each model. The classification results for the training set sceneries are computed as well and saved to the `trainingSet.cnnr` files. These results will be used as the training data for our *Smart Model* as we will show in the following chapter.

5.3 Evaluation pipeline

We describe in this paragraph how the classification results are represented with Java objects. Once a scenery has been classified, it is represented by a `ClassifiedScenery` object. This contains a list of `ClassifiedROIs` which are the bounding boxes associated to the class scores predicted by the neural network. Each of them represents an object in the scenery image. Every `ClassifiedScenery` also stores the associated file name to retrieve the original annotation and jpeg files from the Sun09 folder. The `testSet.cnnr` file contains the list of `ClassifiedScenery` objects generated from the test set. Figure 5.8 depicts how these Java objects are organized. Since the CNN classification results are stored in the `testSet.cnnr` file, we save some time every time we need them for our experiments. Indeed, performing the classification for all the objects takes about *15 minutes*.

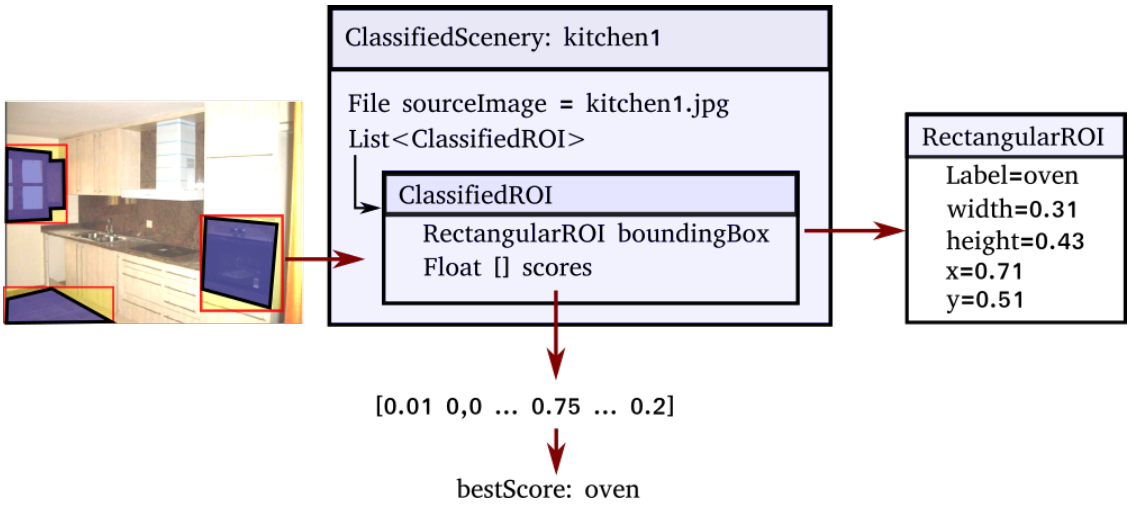

Figure 5.8: `ClassifiedScenery`: contains the list of the classified bounding boxes. Each `ClassifiedROI` stores the bounding box position and size into the `RectangularROI` object. The maximum value of the `scores` vector is corresponding to the selected class. If this is equal to the label contained into the `RectangularROI`, then we consider the classified object as a True-Positive.

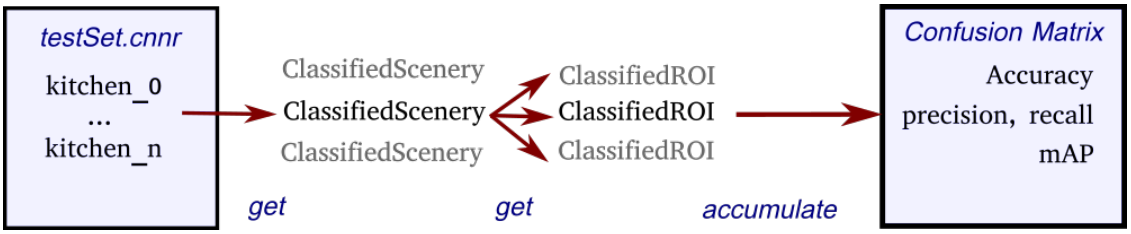

Figure 5.9: Evaluation pipeline. The classified sceneries stored into the `testSet.cnnr` file are collected to build the confusion matrix and compute the mAP score.

5.4 Architecture Experiments

Having described how the neural network models are trained, we provide now a summary with the architectures evaluated in our experiments. This first step of evaluation was performed without cross-validation. The proportion of dataset for training and test set were respectively 80% and 20%.

We started with a small custom CNN, with two convolutional layers and one fully

connected layer. We chose first smaller models because they are easier and faster to train. The result with the first network was 36.7 accuracy and 31.5 mAP.

The second model, *CNN2*, has the same structure of the first one, but the number of filters for the first convolutional layer is duplicated. Also the neurons in the last layer have been increased twice. We obtained a slight improvement, with 38.2 accuracy and 31.7 mAP. Both these two models work with an image size of 40×40 pixels.

The neural network *CNN3* was instead built with three convolutional layers and works with 43×43 images. This gave the best result we achieved with our attempts: 46.2 accuracy and 42.1 mAP. Having added one more convolutional layer improved a lot the performances.

After that point we tried to use a network inspired by the famous *AlexNet* architecture. *CNN4* has five convolutional layers and two fully connected ones. Applying this model didn't give the expected improvement: we could get only 26.4 accuracy. The explanation we give, according to what we learned about CNNs, is that our dataset is too small and bigger networks, like AlexNet, overfit soon the training set.

We tried with two more architectures that gave us some noteworthy results. *CNN10*, with the same structure of *CNN3*, but a higher number of filters and neurons, obtained the 46.4 of accuracy. The last network, *CNN11*, has five convolutional layers, but few filters to avoid overfitting. It gave the 47.1 accuracy only after 3 epochs, while the *CNN3* took 10 epochs to achieve its maximum accuracy. However the training time for *CNN11* is longer because with deeper network the gradient takes more time to be computed .

After the attempts presented in the previous lines, we decided to adopt the *CNN3* structure as the classifier for our experiments with the context-aware *Smart Model*. The following pages show the architectures we described here. In the next tables, convolutional layers kernel size refers to the shape of the filters. For *maxpool* layers the kernel size represents the shape of each patch of the input tensor that will be squashed to a single value. The fully connected layers are described with the number of neurons.

Table 5.2: CNN1

Layer	Kernel size	Number
image size	40x40	
conv	5x5	16
maxpool	2x2	
conv	3x3	64
maxpool	3x3	
fully c.		256

Table 5.3: CNN2

Layer	Kernel size	Number
image size	40x40	
conv	5x5	32
maxpool	2x2	
conv	3x3	64
maxpool	3x3	
fully c.		512

Table 5.4: CNN3

Layer	Kernel size	Number
image size	43x43	
conv	5x5	32
maxpool	3x3	
conv	3x3	64
conv	3x3	64
maxpool	3x3	
fully c.		512

Table 5.5: CNN10

Layer	Kernel size	Number
image size	43x43	
conv	5x5	64
maxpool	3x3	
conv	3x3	64
conv	3x3	128
maxpool	3x3	
fully c.		1024

Table 5.6: CNN4

Layer	Kernel size	Number
image size	100x100	
conv	11x11	64
maxpool	3x3	
conv	5x5	128
maxpool	3x3	
conv	3x3	128
conv	3x3	128
conv	3x3	128
maxpool	3x3	
fully c.		512
fully c.		512

Table 5.7: CNN11

Layer	Kernel size	Number
image size	48x48	
conv	5x5	32
conv	3x3	32
maxpool	2x2	
conv	3x3	64
conv	3x3	64
maxpool	2x2	
conv	3x3	64
fully c.		512

5.5 The final architecture: results

We have presented in the previous section how the *CNN3* architecture was the best attempt for classifying our data. This model presents three convolutional layers which work

Architecture	Accuracy	Epochs
CNN1	36.7	16
CNN2	38.2	18
CNN3	**46.2**	13
CNN10	46.4	15
CNN4	26.2	3
CNN11	47.1	3

Table 5.8: CNN classification results.

with the *RELU* (Rectified Linear Unit) activation function. Pooling layers use the *max-pooling* method, as they take the maximum value to represent a single tensor partition. The fully connected layer works with a *Softmax* activation function, which ensures that the output scores will add up to one. This layer is subjected to the *dropout* regularization technique, with probability 0.5, which means that the 50% of the connections will be randomly turned off at each training iteration. Finally, the optimization algorithm we used is the *Stochastic Gradient Descent*, with momentum 0.9 and learning rate 1×10^{-2}.

Architecture *CNN3* was then evaluated with a k-fold cross-validation, with 10 partitions. Since this allows using the 90% of the dataset as training, we obtained a slightly better result than the one presented in the previous experiments. The training time for all the partitions took *16 hours and 20 minutes*, with 12 epochs per model.

Measure	Value
mAP	0.448
accuracy	0.475
avg Precision	0.446
avg Recall	0.484
avg F1	0.431

Table 5.9: CNN3 classification results, k-fold with $k = 10$

	floor	ceil.	wall	door	oven	han.	win.	surf.	cupb.	chair	desk	sink	plant	bottle	outlet	screen	keyb.	c. lamp	tot.
floor	303	0	0	0	6	1	1	18	3	12	43	10	0	0	0	1	1	0	399
ceiling	0	205	8	0	0	0	6	2	4	0	0	1	0	0	0	0	0	13	239
wall	12	25	530	161	13	3	113	21	19	16	29	6	4	4	0	9	7	5	977
door	0	1	15	58	10	0	25	0	4	5	2	0	0	0	0	3	0	0	123
oven	8	0	0	7	21	0	7	0	9	40	9	4	2	0	0	14	2	0	123
handle	1	0	0	1	1	40	6	1	4	3	0	8	2	5	49	4	7	5	137
window	1	8	23	27	5	2	274	5	10	2	6	1	5	3	3	25	9	9	418
surface	58	0	0	0	4	4	0	124	2	11	48	53	0	0	3	5	19	0	331
cupb.	26	13	52	19	40	2	179	10	89	116	72	13	6	4	3	20	3	12	679
chair	35	0	1	1	57	4	6	1	15	155	43	35	6	8	13	30	17	0	427
desk	17	0	2	0	0	0	1	12	4	15	80	8	2	1	0	8	2	0	152
sink	12	0	0	0	2	3	1	23	1	2	8	132	0	0	3	6	42	0	235
plant	1	3	4	2	14	12	28	0	7	19	4	18	60	25	44	66	11	5	323
bottle	0	0	0	0	5	21	3	0	4	2	0	2	18	65	59	4	9	7	199
outlet	0	0	0	0	0	5	0	0	0	0	0	1	0	1	89	6	6	1	109
screen	1	0	0	1	8	0	8	1	1	5	6	8	6	1	19	68	9	0	142
keyb.	1	0	0	0	0	1	0	3	0	0	2	43	0	1	5	0	42	0	98
c. lamp	0	8	1	0	0	3	36	1	6	0	0	0	6	5	1	1	1	238	307

Figure 5.10: Confusion Matrix for all the 18 classes being classified. The main diagonal corresponds to the items correctly recognized.

Precision:

wall	ceiling lamp	ceiling	floor	surface	bottle	plant	cupboard	handle	window
0.83	0.81	0.78	0.64	0.56	0.53	0.51	0.49	0.40	0.39
sink	chair	outlet	screen	desk	keyboard	door	oven		
0.38	0.38	0.31	0.25	0.23	0.22	0.21	0.11		

Recall:

ceiling	outlet	ceiling lamp	floor	window	sink	wall	desk	screen	door
0.86	0.82	0.78	0.76	0.66	0.56	0.54	0.53	0.48	0.47
keyboard	surface	chair	bottle	handle	plant	oven	cupboard		
0.43	0.37	0.36	0.33	0.29	0.19	0.17	0.13		

F1 score:

ceiling	ceiling lamp	floor	wall	window	sink	surface	outlet	bottle	chair
0.82	0.79	0.69	0.66	0.49	0.46	0.45	0.45	0.40	0.37
handle	screen	desk	keyboard	door	plant	cupboard	oven		
0.34	0.33	0.32	0.29	0.29	0.27	0.21	0.14		

AP score:

ceiling lamp	ceiling	wall	floor	surface	bottle	sink	window	outlet	cupboard
0.84	0.82	0.78	0.74	0.48	0.47	0.45	0.44	0.39	0.36
chair	handle	plant	screen	desk	keyboard	door	oven		
0.36	0.34	0.34	0.31	0.30	0.29	0.22	0.11		

Figure 5.11: Evaluation metrics separated by class. The values of the *AP* score are averaged to compute the mean average precision (*mAP*). The classes with the lower scores are typically those which presents less samples into the dataset.

Chapter 6

The Smart Algorithm

6.1 Smart Model, overview

In this chapter we will present the main contribution of our thesis work. The novel approach is called *Smart Model*. This name refers to the ability of our brain to *smartly* collect and elaborate the rich, but noised information coming from the five senses. It is able to merge all these data with its memories and reconstruct the external world like a complex puzzle with many missing parts.

In particular we use contextual information to provide a better classification than the one obtained directly from the convolutional neural networks, which are only local models. If the image of a particular object is not clear enough and the system is not able to recognize it, the context can be used to infer additional information. Indeed, in our life we are used to see objects in their context and we hardly recognize them when they appear in unusual places. Important information can be retrieved from relative comparisons between the elements into the scene we are looking at. In particular we can compare their positions and sizes or colors and shapes. For example, if the convolutional neural network confuses the class *floor* with *ceiling*, we can correct it because the *ceiling* is more likely above every other object into the scene.

Even the semantical information of the class labels can be very helpful, indeed objects with related functions will appear more likely together. To make an example, a *bath soap* can be easily found in the scenery bathroom, near the *sink* and will be rarely seen into another room like an *office*.

The contextual information which will be used for our experiments refers to a concept that we call *relative features*. They are defined to describe pairs of objects which belong to the scene. These features are compared to the statistical information learned from the training set and allow to *correct* the local predictions of the CNN, obtaining better results. They are also a way of providing more *interpretability* to the model. Neural networks are not interpretable because of their internal structure. Indeed, after training we are not able to easily understand the meaning of the learned weights. Enhancing the model with a rule based algorithm would provide interpretability since these can explain the reason of the correction applied to the neural network scores.

To summarize, our model works with relative features related to object pairs and compares them to the *Smart Rules* which are the entity to represent the contextual information learned from the training set. The result of this operation is used to update the CNN scores and obtain the new classified images. In the next sections we will explain deeper the concepts we described here.

6.1.1 A working example

As we pointed out in the introductory section the Smart Model is designed to update the output scores of neural networks. We can provide the main idea of the algorithm with an example. The following lines represents parts of the scores vector predicted by the convolutional neural network for two object samples.

CNN scores for $object_1 = [$**floor = 0.6**$, ceiling = 0.3, ...]$

CNN scores for $object_2 = [$**ceilingLamp = 0.9**$, floor = 0.00001, ...]$

The class assigned to $object_1$ is *floor* and the one for $object_2$ is *ceiling lamp*. From the dataset we know that the prediction for $object_1$ is wrong, as the actual class is *ceiling*. The scenery image presents the bounding box for $object_1$ at the same vertical position of the $object_2$, like shown in figure 6.1.

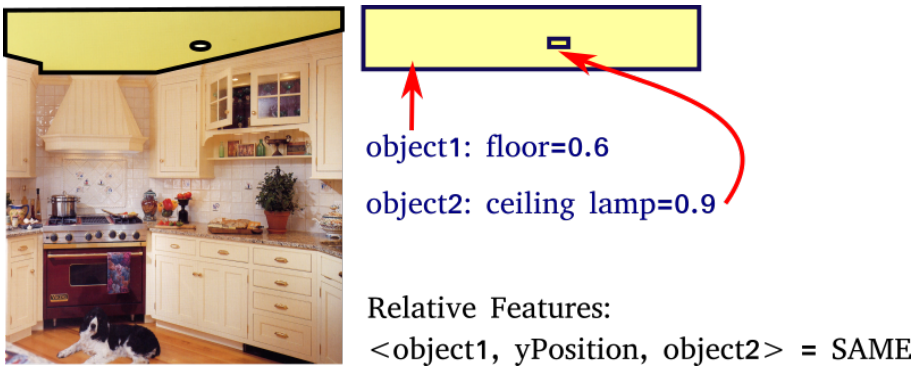

Figure 6.1: Smart Rules. $Object_1$ and $object_2$ are at the same vertical position.

By analyzing the training set we obtained that the 0% of *floors* are at the same vertical position of a *ceiling lamp*. This can be considered as a general rule useful to correct the prediction. Moreover, from the dataset we know that the 90% of the *ceiling* objects are at the same height of *ceiling lamps* and the remaining 10% are above them. Indeed, lamps always hang from a ceiling in our sample sceneries.

Since the assigned label for $object_1$ is wrong, we try to apply a correction to its CNN scores using the two rules presented in the previous lines. Using the contextual information we can easily understand that the model is confusing the class *ceiling* with *floor*. $Object_1$ is a ceiling because it is at the same height of a *ceiling lamp*, assuming that the class for $object_2$ is correct.

The algorithm should vary the output scores for $object_1$ by increasing the value associated to *ceiling* and decreasing the one related to *floor*. The correction should be strong enough to make the score of the first class overcome the second one. A variation of 0.2 would be sufficient for this example.

$$\text{Correction for } object_1 = [floor = 0.6 \textbf{ - 0.2}, ceiling = 0.3 \textbf{ + 0.2}, ...]$$
$$= [floor = 0.4, \textbf{ceiling = 0.5}, ...]$$

After the update of the scores we obtain the correct class for $object_1$. The algorithm that we will present in the following sections use this kind of reasoning as a guideline.

6.2 Smart-Rules: formalization

As previously anticipated, our model works with the concept called *relative features*. The main entity used to represent these relationships between objects is the *triplet*:

$$< subject, featureType, reference >$$

The *subject* is the object that will be compared with the *reference*, according to a particular *feature type*. We designed some different feature types, according to the ways we can compare visual objects jus by looking at their position, shape and color. Each feature type is associated to a set of discrete values. For example the feature *xDimension*, which refers to the bounding box width, can assume the values *smaller*, *same* or *bigger*. Table 6.1 presents for each feature type its possible values.

Triplets can be divided into *instances* and *rules*. A *triplet instance* refers to a pair of objects in the input scenery which are compared with respect to a specific feature type. By looking at the positions and sizes of the two objects, the system chooses one of the possible discrete values associated to the feature type. We adopt the following notation to represent a *triplet instance* and its assigned value:

$$< subject, featureType, reference > = value$$

For example, to express "$object_1$ larger than $object_2$" we can write:

$$< object_1, xDimension, object_2 > = bigger$$

A *rule* doesn't refer to a particular scene instead, but it is thought to summarize the information learned from the training set. It is composed by a *triplet* and a *histogram*. The triplet involves the *class labels*, instead of the *objects* of a particular scenery. For example:

$$< floor, yPosition, ceiling >= [histogram]$$

which refers to the relative position between *floors* and *ceilings*. We will see in the next sections how the *histogram* can describe the statistical distribution of the samples in the dataset.

Feature type	Values
yPosition:	{above, same, below}
xPosition:	{left, same, right}
xDimension:	{smaller, same, bigger}
yDimension:	{smaller, same, bigger}
distance:	{near, far}
luminosity:	{higher, same, lower}

Table 6.1: Feature Types and their possible values.

6.3 Triplets value computation

In this section we present how the system computes the values relative to each feature type. All of these involve a pair of objects and make use of the bounding box coordinates obtained from the input scenery.

Each object is associated to a position which refers to the coordinates of the top-left corner of its bounding box. These coordinates are relative to the top-left corner of the scene frame and they are defined in the range between zero and one. For example with $x = 0.5$ and $y = 0.5$ we obtain a rectangle whose top-left corner is in the middle of the scenery image. The size of a bounding box is in percents, as well. A shape of $width = 0.5$ and

$height = 0.5$ refers to a rectangle which is exactly half in size with respect to the input scenery.

6.3.1 Relative position

The relative position between a pair of objects is described by two feature types: *xPosition*, referring to the horizontal alignment, and *yPosition* which refers to the vertical alignment. While computing these values, both take in consideration the size of the bounding boxes. Considering the *yPosition*, the subject is above the reference if its bottom margin is over the top of the reference. Practically, to make this definition less rigid we introduced the concept of soft margin. Instead of considering strictly the bottom of the bounding box, we say that the feature value is *above* if at least the α percent of the subject height is above the top of the reference. A reasonable value for alpha can be in the range from 0.8 to 0.9. More formally:

$$\textit{above: } subj.y + \alpha \cdot subj.height < ref.y$$

Instead, subject is *below* the reference if at least the α percent of its height is under the subject.

$$\textit{below: } subj.y + (1 - \alpha) \cdot subj.height > ref.y + ref.height$$

All the remaining cases are assigned to the value *same* position. This reasoning can also be applied for the horizontal position. The definitions of *left* and *right* are the following:

$$\textit{left: } subj.x + \alpha \cdot subj.width < ref.x$$
$$\textit{right: } subj.x + (1 - \alpha) \cdot subj.width > ref.x + ref.width$$

The vertical position is very useful for our model because the scenes are not symmetric along this direction. Think to ceilings and floors that cannot be exchanged along the vertical axis. The horizontal position is less significant because the images can often be flipped horizontally without producing any change of meaning. Actually, there is one case when this feature becomes important: if the *xPosition* value is *same* and the *yPosition* is *above*

or *below*, we can identify objects that are directly one on the top of the other. For example *surface* and *bottle* would often be found at the same horizontal position because the latter leans on the former.

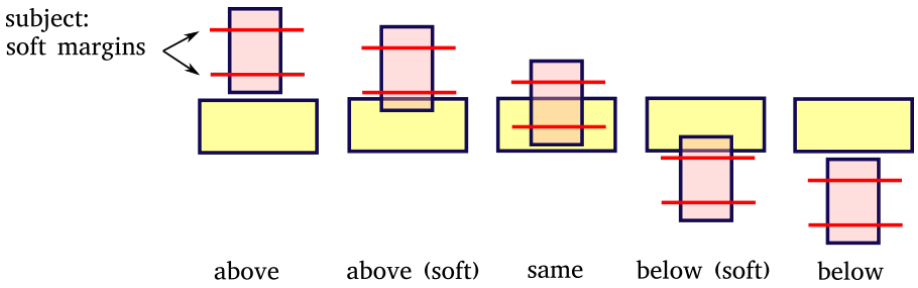

Figure 6.2: Examples of yPosition computation

6.3.2 Size

The relative size is computed separately for width and height. The reason is that we want to distinguish the concept of taller from larger. Considering the horizontal dimension, we say that the subject is bigger than the reference if the ratio of their width is greater than a threshold α. We set the value of α to 1.5, which means that the subject must be the 50% larger than the reference to be considered bigger in width.

$$xDimension = \begin{cases} bigger & \text{if } subj.width/ref.width > \alpha \\ same & \text{if } subj.width/ref.width < \alpha \\ smaller & \text{otherwise} \end{cases}.$$

The same reasoning can be applied to the *yDimension*, by computing the ratio between the bounding boxes height.

This kind of information can be used for example to distinguish smaller objects, such as bottles or lamps, from the bigger elements of the scene. For example a door will be always taller than floors, but not larger. This is the reason why we separated the measure of width and height in two different features. Even if the relative size of the objects is quite

important for our correction system, perspective may be an issue that could confuse the algorithm. Indeed, some sceneries could present objects whose size is difficult to interpret because of perspective reasons.

6.3.3 Distance

The concept of distance is fundamental to identify objects that more likely appear together in the sceneries. For example we can find that *bottles* are always near *surfaces* or *shelves* because they need a support to lay on. To compute this measure we consider the distance between the middle of the bounding boxes, which is defined as follows:

$$x1 = subj.x + 0.5 \cdot subj.width$$
$$y1 = subj.y + 0.5 \cdot subj.height$$
$$x2 = ref.x + 0.5 \cdot ref.width$$
$$y2 = ref.y + 0.5 \cdot ref.height$$

Then the euclidean distance is calculated:

$$dist^2 = (x1 - x2)^2 + (y1 - y2)^2$$

The squared distance is compared to a threshold to decide if the subject is *near* or *far* from the reference.

6.3.4 Luminosity

The relative luminosity between objects could help in identifying light sources like windows, lamps or screens. Another interesting feature could be considering separately the three BGR colors to compare subjects and references. For each object we compute the

average color, defined as the mean of each color channel like shown in equation 6.1.

$$\begin{aligned} avgBlue &= \frac{1}{N}\sum_{i=0}^{N-1} pixel_i.B \\ avgGreen &= \frac{1}{N}\sum_{i=0}^{N-1} pixel_i.G \\ avgRed &= \frac{1}{N}\sum_{i=0}^{N-1} pixel_i.R \end{aligned} \tag{6.1}$$

The luminosity is a value in the range $0{,}1$ and can be computed as follows:

$$luminosity = 0.21 \cdot avgRed + 0.72 \cdot avgGreen + 0.07 \cdot avgBlue \tag{6.2}$$

The system computes the difference of luminosity between subject and reference, then it compares this value to a threshold, deciding which is the lighter object of the pair.

6.4 Histograms

The last concept we have to introduce before explaining the main algorithm of our *Smart Model* is called *histogram*. This entity is designed to represent the information that the model learns from the training set sceneries. Indeed, histograms can collect statistics about the relative features computed on the training samples. As we anticipated, we define *rule* as the association of a histogram with a triplet. For each of the possible values related to the feature type of the triplet, the histogram provides the percentage of the training samples that satisfy the relationship.

We provide an example to better understand this concept. Consider the triplet describing the relative vertical position between *cabinet* and *floor*. For each value of the *yPosition* feature the histogram specifies the number of examples that satisfy this relationship in the

training set:

$$< cabinet\ \textit{yPosition:above}\ floor > = 78$$

$$< cabinet\ \textit{yPosition:same}\ floor > = 7$$

$$< cabinet\ \textit{yPosition:below}\ floor > = 0$$

$$< cabinet, yPosition, floor >= [above, same, below][78, 7, 0]$$

The obtained histogram $[78, 7, 0]$ means that in 78 object pairs cabinets are above floor, while in the remaining 7 they are at the same height. In total we have $78 + 7 = 85$ pairs cabinet-floor into our training set. Note that a scenery can contain zero, one or more pairs for the rule in analysis. As shown in the example, we can find that a cabinet is at the same height of the floor because of perspective reasons, like depicted in figure 6.3.

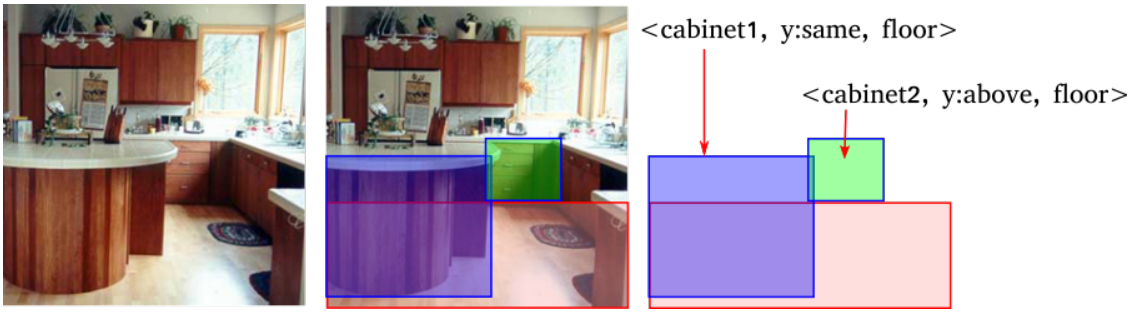

Figure 6.3: Example of perspective issues.

Histograms can also be normalized to obtain a discrete distribution of probabilities. In this way all the numbers will add up to one. For the example in analysis, by dividing all the histogram elements by 85, we obtain:

$$< cabinet, yPosition, floor >= [above, same, below][0.918, 0.082, 0.0]$$

We must point out that, when interpreting these values as percents, the 92% or the 8% are referred to the number of pairs *cabinet-floor* and not to the number of *cabinets* (which we would have obtained dividing the histogram elements by 78).

6.4.1 Impurity measure associated to histograms

The number of different histograms that can be computed on the training set is given by the combination of all the class pairs with the feature types:

$$nFeatureTypes \times nClasses^2$$

With 6 features and 18 classes we obtain $1{,}944$ histograms. Actually, this is not a huge number for modern computational resources, but trying to reduce it by filtering only the most significant statistics would improve the performances.

To evaluate the quality of a histogram we need to define an impurity measure, index of how its discrete probabilities are unbalanced. A lower impurity means a significant rule, for example *"all the lamps are on the ceiling"* or *"many cabinets are above the floor"*. Instead, when the impurity is higher the histogram is useless because all the probabilities are equally distributed. Inspired by the Gini-index measure, the impurity of a histogram is defined to be:

$$impurity = 1 - \sum_{i=0}^{\|h\|-1} \left(\frac{h[i]}{sum(h)} \right)^2 \tag{6.3}$$

where:

$$\|h\| = \text{n. elements of the histogram}$$

$$h[i] = \text{element } i \text{ of the histogram}$$

$$sum(h) = \sum_{i=0}^{\|h\|-1} h[i]$$

We provide here a practical example with two histograms.

$$impurity([0, 2, 235]) = (1 - (2/237)^2 + (235/237)^2) = 0.017$$

$$impurity([0, 61, 235]) = (1 - ((61/296)^2 + (235/296)^2) = 0.33$$

The first rule has lower impurity, for this reason is more relevant than the second one.

The $impurity$ vary in the range between 0 and $normFactor$, whose value is:

$$normFactor = 1 - \frac{1}{\|h\|} \tag{6.4}$$

Sometimes it could be useful to normalize the impurity, obtaining a value which does not depend to size of the histogram:

$$impurity_{norm} = \frac{impurity}{normFactor} \tag{6.5}$$

6.4.2 Histogram importance and Jaccard index

We could think about using the sum of the elements of a histogram as another importance index. Indeed, this value represents the number of sample pairs which support the associated rule. Actually this method generates problems because it penalizes the rules which are related to classes with lower cardinality. For example, since we have few examples of class *keyboard*, all the histograms which are related to this class would have low importance and would be filtered out.

We should define another measure which takes in consideration both the number of samples that support the histogram and the cardinality of the two involved classes. Let's call A and B the classes of the *subject* and the *reference* associated to a given rule. The sum of the elements of the histogram is index of how many times we find sceneries containing both objects of class A and B. We want to compare this value with the number of times that we find either objects of class A or class B in the training set.

The definition of the *Jaccard index* comes in our help. In statistics, it is used to compare two sample sets by computing the ratio between the size of their intersection and the size of their union. In our case the index takes in consideration how many times the two classes appear together in a scene with respect to the number of times they appear in the dataset. The higher is the coefficient, the higher is the number of pairs A-B with respect to the occurrences of two classes considered separately. In other words it is an index of how much the two classes are related. If A and B always appear together in the scenes, the

value will be maximum. More formally, it can be written as:

$$jaccard(A, B) = \frac{\|A \cap B\|}{\|A \cup B\|} = \frac{\|A \cap B\|}{\|A\| + \|B\| - \|A \cup B\|} \tag{6.6}$$

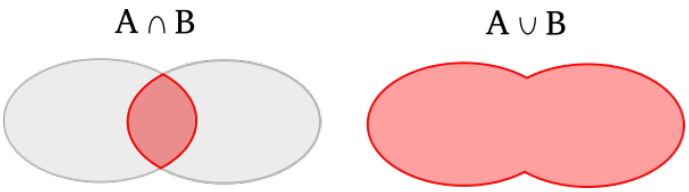

Figure 6.4: Jaccard index computation: intersection and union

The procedure to compute the Jaccard coefficient works by analyzing each image in the training set. For each scenery we get all the possible object pairs, calling A and B the classes for the subject and the reference respectively. The loop counts the number of couples containing both A and B, the ones containing A and the ones containing B. When computing the index for two classes A' and B', the intersection is equal to the number of pairs containing both of them. Instead, the union is equal to the number of pairs containing at least an object of class A' or B'. The pseudocode is described in algorithm 4.

Algorithm 4 Jaccard computation

```
procedure computeJaccard
    for all scenery ∈ trainingSet do
        for all a ∈ scenery.objects do
            for all b ∈ scenery.objects, a ≠ b do
                A ← a.label
                B ← b.label
                countPairs[A, B]++
                count[A]++
                count[B]++

procedure getJaccard(A', B')
    intersection ← countPairs[A', B']
    union ← count[A'] + count[B'] - intersection
    return intersection / union
```

The condition $a \neq b$ avoids counting each pair two times. Note that since we don't care about the couple order, `countPairs[A, B]` is equivalent to `countPairs[B, A]`.
We present here a practical example of computation.

$$A \cap B = 20, A = 100, B = 50, \qquad Jaccard = \frac{20}{100 + 50 - 20} = 0.15$$
$$A \cap B = 20, A = 30, B = 20, \qquad Jaccard = \frac{20}{30 + 20 - 20} = 0.67$$
$$A \cap B = 20, A = 100, B = 100, \qquad Jaccard = \frac{20}{100 + 100 - 20} = 0.11$$

All the three cases have the same value for the intersection, but different cardinalities for the individual classes. For this reason the histograms are evaluated differently, giving more importance to the case where A and B have less elements.

6.4.3 Transition and label probabilities

The algorithm responsible for updating the class scores should also take in consideration the number of times that the neural network gives the wrong prediction for a given class. In particular the *label probability* is defined as the probability of finding an object belonging to class *A* labeled with the class *B*. More formally:

$$P(actual = A \quad | \quad predicted = B) = \frac{count(actual = A \cap predicted = B)}{count(predicted = B)} \tag{6.7}$$

During our experiments we used this measure to apply the Smart-Rules associated to a particular reference object. Since we don't know the real class of the reference, the algorithm weights the importance of the Smart-Rules over all the possible classes. The weight is computed by taking in consideration both the CNN class scores and the *label probability*, as we will see later. By looking at the confusion matrix relative to the CNN results extracted from the training set, we observe that this probability is simply computed as the ratio between the element *mat[actual=A][predicted=B]* and the sum of the values in the column related to the objects with predicted class *B*.

When the correction of the Smart-Algorithm tries to move an object from class B to class A, we should take in consideration another measure that we call *transition probability*. It specifies how many objects of class A have been classified by the neural network with class B. The formal definition is:

$$\begin{aligned} P(B \rightarrow A) &= P(predicted = B \quad | \quad actual = A) \\ &= \frac{count(actual = A \cap predicted = B)}{count(actual = A)} \end{aligned} \tag{6.8}$$

The transition probability is computed from the confusion matrix with the ratio between *mat[actual=A][predicted=B]* and the sum of the elements in the row related to the objects with actual class *A*, as shown in the right half of figure 6.5.

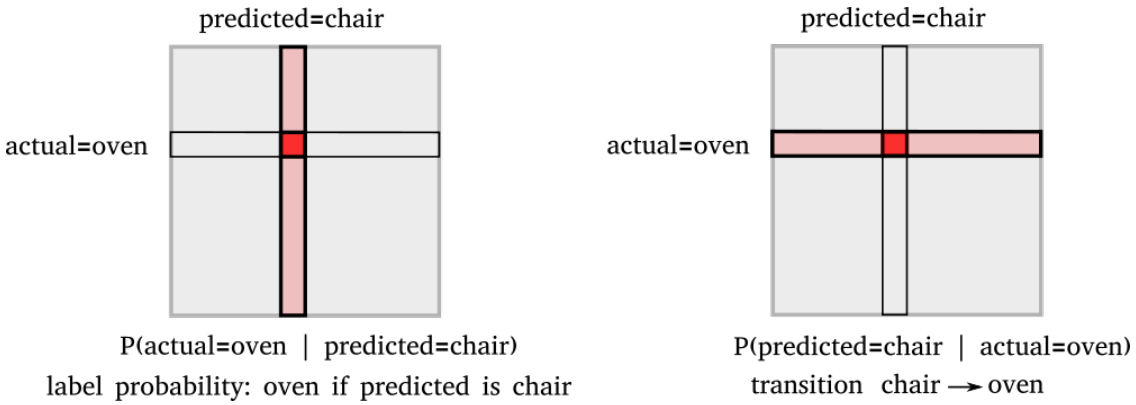

Figure 6.5: Transition and label probabilities computed on the training set confusion matrix.

6.5 Building the histograms

We have seen in chapter 5 how the the neural network results are stored into the `trainingSet.cnnr` and `testSet.cnnr` files in the form of `ClassifiedScenery`. Each classified scenery contains for every object the bounding boxes coordinates and the classification scores. Starting from this data and using the procedures defined in section 6.3, the histogram computation considers all the possible couples of objects belonging to a scenery and computes a value for each feature type. This procedure generates many triplet instances of the type:

$$< subject, featureType, reference > \quad = value$$

For every triplet instance the system retrieves the rule with its histogram:

$$< subject.label, featureType, reference.label > \quad = hist$$

Afterwards it adds an occurrence to the histogram element corresponding to the computed feature value. At the end of the process we obtain the statistics about the object pairs into the training set. The algorithm works as follows.

Algorithm 5 Histogram computation

```
procedure computeHistograms
    for all scenery ∈ trainingSet do
        for all subj ∈ scenery.objects do
            for all ref ∈ scenery.objects, ref ≠ subj do
                for all featureType ∈ featureTypes do
                    subjL ← subj.label
                    refL ← ref.label
                    triplet ← <subjL, featureType, refL>
                    histogram ← histograms[triplet]
                    value ← computeValue(<subj, featureType, ref>)
                    histogram[value]++
```

Suppose an example with two objects *lamp1* and *ceiling1*. The `computeValue()` method produces the feature values. Let's focus for example to *"yPosition=below"*. The object

`histograms` is encoded with a Java `Map` which can retrieve the histogram given a triplet. In our case:

```
histograms[<lamp, yPosition, ceiling>]
```

could return something like `histogram = [below, same, above][55, 12, 0]`. The line `histogram[value]` selects the element corresponding to the computed feature value; in our example `histogram["below"]` returns 55. The selected element is then incremented with a new occurrence:

```
[below, same, above][56, 12, 0]
```

6.6 Smart Rules application

In this section we present how the system will apply the learned rules and histograms to improve the convolutional neural network results. We will explain the algorithm in steps, with two different approximation levels. The process works, one scenery at a time, with the objects classified by the convolutional neural network. These are encoded with the Java object `ClassifiedROI`. We introduce some notations that will be used in the pseudocode shown later.

Global lists:

`labels` = list of all the possible class labels (18 in our experiments)

Sceneries and objects:

`scenery.objects` = list of the objects (`classifiedROI`) into the scenery

`obj.predicted` = label predicted by the CNN for the object

`obj.score[label]` = predicted CNN score for the specified label

Statistics:

`jaccard[A,B]` = Jaccard coefficient of the two classes

`pLab[A, B]` = label probability $P(actual = A \quad | \quad predicted = B)$

`pTrans[A, B]` = transition probability $P(predicted = B \quad | \quad actual = A)$

Histograms:

`histograms[triplet]` = histogram associated to the triplet.

`h[featureValue]` = histogram element related to the specified feature value

`h.length` = histogram length, equal to the number of feature values

6.6.1 The main loop

The main idea behind the algorithm is to apply a correction considering as *subject* one object at a time and comparing it with the other elements in the scenery. At each iteration, the class scores of the selected subject are updated by exploiting the information of the Smart-Rules.

Algorithm 6 Smart-Rules Application

```
procedure APPLYSMART(scenery)
    for all subject ∈ scenery.objects do
        updatedSubject ← applyToSubject(subject)
        outputScenery.put(updatedSubject)
    return outputScenery
```

The procedure `applyToSubject(subject)` generates the new *subject* with the updated class scores. This method internally selects as reference the other objects in the scene, one at a time, generating all the possible pairs <*subject, reference*>. Afterwards, it performs the correction using the generated couples and the statistics collected into the histograms. Once the subject has been updated, its newer version is inserted into the `outputScenery`, which is the result produced by the algorithm.

6.6.2 Smart Algorithm: first approximation

Now we can analyze how the procedure `applyToSubject()` is defined. This method works always with the same subject, which is going to be updated. The references will remain untouched. The algorithm picks one class at a time and pretends the *subject* belongs to this one. With this idea in mind we extract the relative features between the *subject* and the *reference* objects. If these are consistent to the statistical information learned by the histograms, then the score relative to the pretended class of the *subject* is increased, otherwise it is penalized. In this way the adjustments will support the most probable classes and oppose to the others. If the re-balancing of the scores is sufficient, the result will be a change of the predicted class for the *subject*.

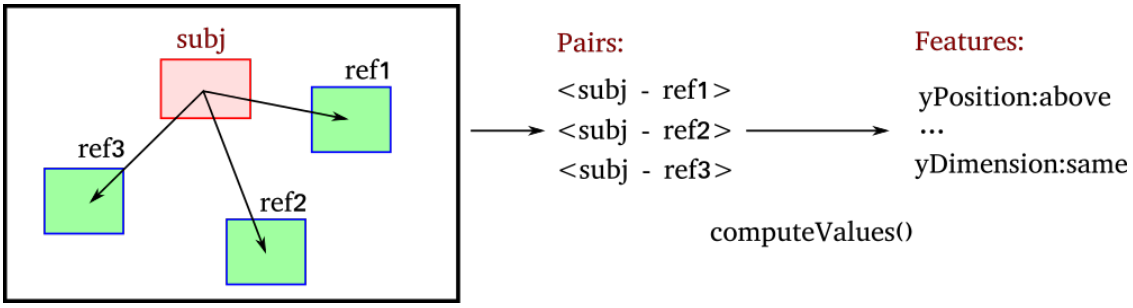

Figure 6.6: Feature extraction for all the pairs composed by the *subject* and the other objects considered as *reference*.

The algorithm described informally above is composed by two nested loops. The outer one selects the class pretended by the *subject*. Instead, the inner cycle computes the corrections for all the possible references. The relative features between the *subject* and the *reference* are extracted with the `computeValues()` method. This produces a Java `Map` containing for each feature type the corresponding value. The `accumulate()` procedure takes as input the pretended label of the subject, the class predicted by the neural network and the feature values. By comparing the features with the information stored into the histograms, it produces a value that we call decision, represented in figure 6.7 with the bold arrows. The arrows supporting the pretended class point up, while the others point

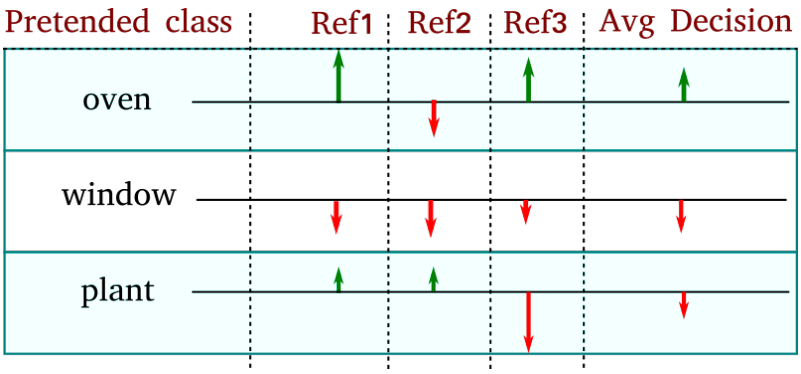

Figure 6.7: The rows in the table represent the pretended classes of the *subject* (outer loop), while the columns specify the *reference* used to compute the decisions (inner loop). The arrows pointing up are supporting the pretended class, while the ones pointing down are penalizing it. Finally, the last column represents the average of the previous decision values.

down. Once all the references have been presented to the `accumulate()` method, we call `apply()` to obtain the *average decision* and update the *subject* score. If the most part of the decisions was in favour of the pretended label then the score is increased, otherwise decreased. Before returning the result, the *subject* scores are processed with the *softmax* function to obtain a vector whose elements add up to one.

Algorithm 7 Smart Algorithm, first approximation

```
procedure APPLYTOSUBJECT(subj)
    var updatedSubj ← subj.clone()
    for all subL ∈ labels do
        var corr ← emptyCorrection
        for all ref ∈ scenery.objects, ref ≠ subj do
            var values ← computeValues(subj, ref)
            var refL ← ref.predicted
            corr.accumulate(subL, refL, values)
        updatedSubj.score[subL] ← corr.apply(subj, subL)

    updatedSubj.scores ← softmax(updatedSubj.scores)
    return updatedSubj
```

6.6.3 Smart Algorithm: complete version

The algorithm presented in the previous section makes an assumption: the label `refL` conferred by the CNN to the *reference* is correct. This might be not true and could influence negatively the functioning of the procedure, as it could bring the correction in a wrong direction. To overcome this issue we applied the rules considering for the *reference* not only the class predicted by the neural networks, but all the possible labels with their associated score.

Given a pair *subject-reference* the algorithm selects one class at a time for the reference. Afterwards it computes a decision value using the histograms. The decisions are weighted with the probability that the class assigned to the *reference* is correct, then averaged. We call this as the *inner average* because it belongs to the inner loop. The *outer average* is instead computed with the mean of the inner averages, after all the references have been analyzed. Figure 6.8 depicts the difference between inner and outer average.

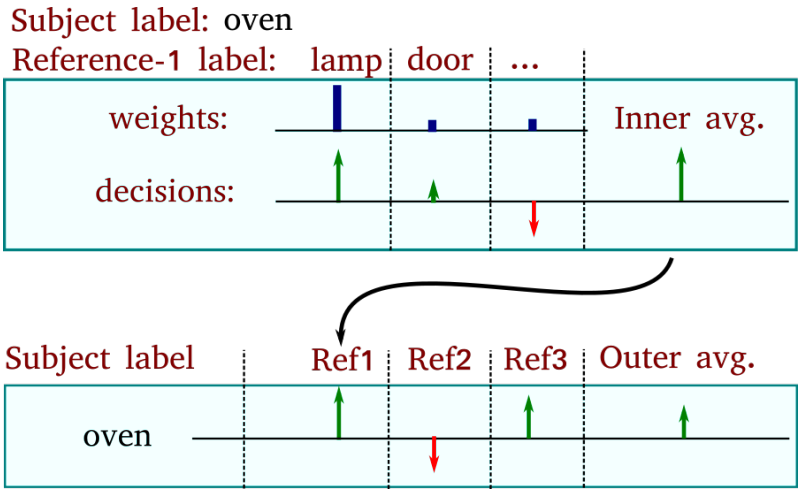

Figure 6.8: Inner and Outer average. The upper table shows how the inner weighted average is computed among the possible reference labels. In particular the weight for *lamp* is very high and the result takes in consideration almost only this class. The inner average for reference-1 is then inserted in the second table, where the outer average is computed.

The weights of the *inner average* are computed with the class scores assigned by the neural network and the *label probabilities* defined in section 6.4.3. The value `weights[i]` is index of how likely the reference belongs to class *i*. It is computed with the product between the class score and the probability *P(actual = i | predicted = ref.predicted)*, which is learned from the training set. The vector containing the weights is then scaled until its values add up to one. The pseudocode to compute the weights is shown in algorithm 8.

Algorithm 8 Reference Weights Computation

```
procedure computeWeights(ref)
    for all i ∈ labels do
        weights[i] ← ref.score[i] · pLab(i, ref.predicted)
    sum ← sum(weights)
    for all i ∈ labels do
        weights[i] /= sum
    return weights
```

Resuming the working principles of the algorithm, one *subject* at a time is selected to apply the correction. The subject pretends to belong to a particular class and it is compared with the other objects. For each of these references the system computes the decisions related to every possible class label and obtains the inner average. The calculated values are then collected to retrieve the outer average. This last number is used to actually correct the subject score relative to its pretended class.

We describe now the procedure more in detail. The pseudocode is composed by three loops. The the most inner one selects the possible classes for the reference. For each of them a decision value is computed by calling the `accumulate()` method. When the loop is terminated the method `nextReference()` is invoked and the `Correction` object performs the *inner average* of the accumulated decisions. The middle loop runs through the references found into the scenery and when it is completed the *outer average* is performed calling the `apply()` method. Finally, the outer cycle is the one responsible for selecting the subject labels. The pseudocode is shown in algorithm 9.

Algorithm 9 Smart Algorithm, complete version

```
procedure APPLYTOSUBJECT(subj)
    var updatedSubj ← subj.clone()
    for all subL ∈ labels do
        var corr ← emptyCorrection
        for all ref ∈ scenery.objects, ref ≠ subj do
            var values ← computeValues(subj, ref)
            var weights ← computeWeights(ref)
            for all refL ∈ labels do
                corr.accumulate(subL, refL, values, weights[refL])
            // inner-average
            corr.nextReference()
        // outer-average
        updatedSubj.score[subL] ← corr.apply(subj, subL)

    updatedSubj.scores ← softmax(updatedSubj.scores)
    return updatedSubj
```

6.7 Score computation

We have described in detail the working principles of the *Smart Algorithm*. The last important module to be explained is the `Correction` object, which actually updates the subject scores. Its life cycle starts when the pretended label `subL` is decided and terminates as the update is applied to the subject. The local variables of the `Correction` object store the values necessary to build the *inner* and the *outer* averages. The former is computed using the variables whose name starts with the prefix `"inner"`, while the latter uses variables with the prefix `"outer"`.

We distinguish between positive and negative decisions, which were depicted by arrows pointing up or down in the graphical representation of figure 6.7 and 6.8. The variables associated to positive decisions have the suffix `"pos"`, while the negative ones have the suffix `"neg"`. The `innerSum` and `outerSum` variables are the denominators of the two averages.

Algorithm 10 The *Correction* object

```
// Initialization: empty Correction
var innerPos ← 0
var innerNeg ← 0
var innerSum ← 0

var outerPos ← 0
var outerNeg ← 0
var outerSum ← 0

procedure CORRECTION.ACCUMULATE(subL, refL, values, weight)
    var positiveRules ← 0
    var negativeRules ← 0
    var countRules ← 0

    for all feat ∈ values do
        hist ← histograms[<subL, feat.type, refL>]
        jacc ← jaccard(subL, refL)
        score1 ← (hist[feat.value] - 1/hist.length) · jacc
        if score1 > 0 then
            positiveRules += score1
        else
            negativeRules += score1
        countRules++

    innerPos += (positiveRules / countRules) · weight
    innerNeg += (negativeRules / countRules) · weight
    innerSum += weight

procedure CORRECTION.NEXTREFERENCE
    // Inner-average
    innerAvgPos = innerPos / innerSum
    innerAvgNeg = innerNeg / innerSum

    outerPos += innerAvgPos
    outerNeg += innerAvgNeg
    outerSum ++

    innerPos ← 0
    innerNeg ← 0
    innerSum ← 0
```

As shown in algorithm 10 the `accumulate()` method takes as parameters the pretended label of the subject (`subL`), the label selected for the reference, its associated weight and the `Map<>` of computed feature values. For each entry *<FeatureType, FeatureValue>* in the Java `Map`, the algorithm creates the triplet instance:

$$<subL, feat.type, refL> = feat.value$$

Afterwards, the histogram related to the triplet is extracted into the variable `hist`. With the instruction `hist[feat.value]` the algorithm selects the histogram element related to the feature value. This is normalized in the range 0-1 and refers to percentage of object pairs in the training set which satisfy the relationship described by the triplet instance.

To discriminate between positive and negative decisions we have to define a threshold and compare it with `hist[feat.value]`. A histogram with maximum impurity presents all the elements with the same value. Since they represent percentages and they add up to one, these values are equal to `1/hist.length`. If the selected histogram element is greater than this threshold, then it is sufficiently high to be considered as positive decision. According to this reasoning we compute the difference:

```
hist[feat.value] - 1/hist.length
```

If it is greater than zero, the value is added to the positive decisions, otherwise to the negative ones. To weight the importance of the rules, each decision value is multiplied by the Jaccard coefficient of the two classes involved in the relationship. Once all the features have been analyzed, the positive and the negative rules are averaged, multiplied by the weight associated to the reference label and added to the variables `innerPos` and `innerNeg` respectively. When the `nextReference()` method is called, the inner average is computed dividing `innerPos` and `innerNeg` by the sum of the weights stored into `innerSum`. These inner averages are accumulated to the `outerPos` and `outerNeg` variables.

Algorithm 11 Rule Application

procedure CORRECTION.APPLY(subj, subL)
// Outer-average
`outerAvgPos = outerPos / outerSum`
`outerAvgNeg = outerNeg / outerSum`

if `Method == 1` **then**
$p = pLab(subL, subject.predicted)$
else if `Method == 2` **then**
$p = pTrans(subL, subject.predicted)$

$finalCorrection = (outerAvgPos + outerAvgNeg) \cdot gain_1 - bias$

if `finalScore > 0` **then**
$retval = subj.score[subL] + (p) \cdot finalCorrection$
else
$retval = subj.score[subL] + (1 - p) \cdot finalCorrection$
return $gain2 \cdot retval$

Finally, the `apply()` method actually elaborates the new score for the subject. To achieve this task the outer average is computed separately for the positive and the negative decisions into `outerAvgPos` and `outerAvgNeg`. At this point, the algorithm considers two methods which differ from the choice between using the *label probability* or the *transition probability*. They both are capable of estimating how likely we could have a change from the predicted label to `subL`. The variable `p` contain either of these two values.

The final correction is computed by adding together the positive and the negative decisions, since they are opposite in sign. To make this value effective for the score update, a gain and a bias are added. These are the parameters of the algorithm and must be manually tuned. A positive final correction means that the majority of the decisions support the subject label. In this case, the subject score is updated with the correction multiplied by the probability `p`. Instead, when the final correction is negative, it is multiplied by `1-p` which is the probability for the subject of maintaining its original label. The result is amplified with `gain2`, necessary to have bigger numbers before applying the softmax function.

Chapter 7

Results

7.1 Evaluation of the Smart Algorithm

In the previous chapter we presented how our model learns the contextual information from the training set and the way it uses the Smart-Rules to correct the neural network predictions. We applied the algorithm to the 510 sceneries of the dataset, already labeled by the convolutional neural network that we presented in chapter 5. Maintaining the same 10 partitions for the cross-validation technique, we trained 10 Smart Models. Each was evaluated on the related test set and the results were collected into a single confusion matrix. In this way we can easily compare the improvement brought from the Smart Algorithm with respect to the baseline model composed only by the neural network.

Our model is quite **fast**, indeed to apply the CNN to all the sceneries it took about ten minutes, while **few seconds** to perform the Smart correction. After tuning properly the parameters of the `apply()` method in the `Correction` object we obtained some remarkable results. We succeeded in improving the **accuracy of the 10%** with respect to the base CNN-only model, with an increasing of the **7% for the mAP** score.

As anticipated in section 6.7 we designed two methods to compute the correction into the `apply` method. The first one defines the value p, which specifies how likely the subject

will change its predicted class, with the label probability:

$$P(actual{=}A \mid predicted{=}B)$$

The second one defines p with the transition probability:

$$P(predicted{=}B \mid actual{=}A)$$

After trying the method-1 we obtained a very low F1 for the class *oven*. The neural network is very imprecise with this class and the Smart Model didn't manage to improve the results. We noticed from the confusion matrix that the most of the *ovens* were confused with *chairs* by the CNN. When the Smart Model selects *oven* as the pretended class of the subject, if the label predicted by the neural network is *chair* we would like to have a high value of p in order to encourage the transition from *chair* to *oven*. This requirement is not satisfied with the label probability $P(actual{=}oven \mid pred{=}chair)$. Indeed, the class *chair* has many more samples than *oven* and for this reason the ratio:

$$\frac{count(actual = oven \cap pred = chair)}{count(pred = chair)}$$

has always a low value. In other words the label probability counts the percentage of objects which actually belong to the class *oven* among those with predicted class *chair*. Since we have many chairs and few ovens, this value is always low.

Instead, when using the transition probability the denominator is defined with the number of objects whose actual class is *oven*:

$$\frac{count(actual = oven \cap pred = chair)}{count(actual = oven)}$$

In other words, this ratio refers to the percentage of *ovens* which have been classified as *chairs*. This new value is independent from the number of *chairs* and gives better results.

Another attempt we tried to improve the F1 of the classes with smaller cardinality was to avoid correcting objects whose predicted label has a low recall in the confusion matrix. In the *apply()* method we just added a condition which leaves untouched the score if the

recall of the predicted class is lower than a threshold. In this way we avoid the Smart-Algorithm to reduce the recall for classes like *oven*, which are hardly recognized by the neural network. We obtained a slight improvement of the average F1 and a loss of accuracy. Choosing between the two method depends to a trade-off between accuracy and F1, but the differences are very small. The table below shows the scores for the neural-network-only method, the improvement with the application of method-1 and and the results with method-2.

Measure	Before (%)	After 1 (%)	Delta 1 (%)	After 2 (%)	Delta 2(%)
mAP	44.77	52.43	+ 7.66	51.50	+ 6.72
accuracy	47.49	57.77	+ 10.28	57.51	+ 10.02
avg Precision	44.66	54.13	+ 9.47	51.45	+ 6.80
avg Recall	48.44	50.93	+ 2.49	51.65	+ 3.22
avg F1	43.11	50.18	+ 7.07	50.62	+ 7.50

Table 7.1: Smart Model - final results.

In the next two section we will present a set of tables with the evaluation for the individual classes. Firstly we show the difference cell-by-cell between the confusion matrix before and after the application of the Smart Model. A cell with a positive number means that new samples were added by the algorithm. A negative number specifies that some objects were removed and positioned to another cell. The model works properly when we obtain positive numbers only on the main diagonal, where the actual class is equal to the predicted one. The other tables represent the variation of the precision, recall and F1 scores for the single classes. Positive values represent an improvement with respect to the base model.

7.1.1 Method-1, results

	floor	ceil.	wall	door	oven	han.	win.	surf.	cupb.	chair	desk	sink	plant	bottle	outlet	screen	keyb.	c. lamp
floor	25	0	0	0	-6	0	-1	-3	5	6	-20	-4	0	0	0	-1	-1	0
ceiling	0	-1	2	0	0	0	-4	0	3	0	0	0	0	0	0	0	0	0
wall	-1	-4	222	-138	-13	-1	-64	3	25	-5	-23	1	11	-2	0	-8	-5	2
door	0	0	45	-41	-10	0	-4	0	14	-1	-2	0	1	0	0	-2	0	0
oven	2	0	10	-7	-20	0	0	2	18	15	-8	-3	5	0	0	-12	-2	0
handle	-1	0	0	-1	-1	2	-4	0	10	0	0	-2	7	13	-20	-4	-2	3
window	0	-1	41	-25	-5	-2	-16	-2	30	2	-5	3	8	-3	-1	-23	-7	6
surface	5	0	4	0	-4	-1	1	37	26	2	-35	-11	1	0	-2	-4	-19	0
cupb.	15	-4	19	-19	-39	0	-80	-2	231	-39	-45	-7	5	-4	-3	-19	-3	-6
chair	0	0	3	-1	-57	-3	-4	7	66	40	-23	-5	11	-2	-11	-9	-12	0
desk	9	0	4	0	0	0	-1	5	4	4	-17	-5	0	0	0	-4	1	0
sink	-4	0	2	0	-2	-2	0	8	14	5	-5	19	3	1	-2	-4	-33	0
plant	0	-2	6	-1	-14	-6	-11	1	38	2	-3	-2	72	-7	-17	-51	-9	4
bottle	0	0	0	0	-5	-6	0	0	8	5	0	0	12	22	-25	-4	-8	1
outlet	0	0	0	0	0	4	0	0	3	0	0	0	5	2	-16	-4	-1	7
screen	-1	0	1	-1	-8	0	0	1	36	4	-4	-3	12	6	-11	-27	-5	0
keyb.	2	0	0	0	0	0	0	2	1	4	-2	-20	1	-1	-1	0	13	1
c. lamp	0	-2	-1	0	0	-2	-7	0	7	0	0	0	-2	-3	-1	-1	0	12

Figure 7.1: Method-1, Confusion Matrix. Actual class on the vertical axis, predicted on the horizontal one.

Delta Precision:

oven	keyboard	screen	door	desk	window	sink	handle	outlet	bottle
0.39	0.36	0.19	0.19	0.17	0.12	0.11	0.10	0.10	0.07
chair	ceiling	surface	floor	plant	c. lamp	cupboard	wall		
0.05	0.04	0.01	-0.01	-0.02	-0.04	-0.05	-0.08		

Delta Recall:

cupboard	wall	plant	keyboard	surface	bottle	chair	sink	floor	c. lamp
0.34	0.23	0.22	0.13	0.11	0.11	0.09	0.08	0.06	0.04
handle	ceiling	window	desk	outlet	oven	screen	door		
0.01	0.00	-0.04	-0.11	-0.15	-0.16	-0.19	-0.33		

Delta F1 score:

keyboard	cupboard	plant	wall	sink	bottle	desk	surface	chair	window
0.28	0.25	0.17	0.11	0.10	0.10	0.09	0.08	0.07	0.07
outlet	handle	ceiling	screen	floor	c. lamp	door	oven		
0.06	0.04	0.02	0.02	0.02	0.00	-0.09	-0.12		

Figure 7.2: Method-1. Evaluation metrics separated by class.

7.1.2 Method-2, results

	floor	ceil.	wall	door	oven	han.	win.	surf.	cupb.	chair	desk	sink	plant	bottle	outlet	screen	keyb.	c. lamp
floor	24	0	0	0	-4	0	-1	-3	4	5	-20	-5	0	0	0	-1	1	0
ceiling	0	-1	1	0	0	0	-3	0	3	0	0	-1	0	0	0	0	1	0
wall	-1	-4	207	-133	-7	-1	-59	4	26	-7	-21	-1	8	-1	0	-8	-4	2
door	0	0	34	-35	-7	0	-2	0	14	-1	-2	0	1	0	0	-2	0	0
oven	2	0	7	-5	-10	0	-1	1	17	7	-7	-3	4	0	0	-12	0	0
handle	-1	0	0	0	-1	10	-5	0	8	-1	0	-4	8	8	-25	-4	3	4
window	0	-1	33	-23	-5	-2	-18	-2	37	2	-5	2	8	-1	-3	-22	-7	7
surface	6	0	3	0	-3	-2	1	40	23	0	-35	-17	1	0	-2	-4	-11	0
cupb.	16	-4	18	-18	-32	1	-79	-2	224	-41	-45	-8	2	-3	-3	-19	-1	-6
chair	-3	0	3	-1	-42	-3	-4	8	50	34	-21	-14	10	2	-11	-9	1	0
desk	10	0	4	0	0	0	-1	4	4	1	-17	-8	2	0	0	-4	5	0
sink	-4	1	1	0	-2	-1	0	6	12	6	-5	3	2	0	-2	-4	-13	0
plant	0	-2	3	-1	-5	0	-11	1	36	-5	-3	-8	71	-3	-23	-51	-1	2
bottle	0	0	0	0	-4	-3	0	0	8	3	0	1	11	24	-29	-4	-6	-1
outlet	0	0	0	0	0	10	0	0	3	0	0	0	4	5	-23	-4	-1	6
screen	-1	0	2	-1	-8	0	-3	0	33	4	-5	-4	12	8	-12	-26	1	0
keyb.	2	0	0	0	0	0	0	2	1	3	-2	-28	1	-1	-1	0	22	1
c. lamp	0	-2	-1	0	0	-2	-8	0	6	0	0	0	-2	-3	-1	-1	0	14

Figure 7.3: Method-2, Confusion Matrix.

Delta Precision:

screen	door	sink	desk	keyboard	outlet	window	oven	chair	handle
0.19	0.17	0.16	0.16	0.14	0.12	0.12	0.08	0.07	0.07
ceiling	bottle	surface	plant	floor	c. lamp	cupboard	wall		
0.04	0.03	0.03	-0.01	-0.01	-0.03	-0.04	-0.06		

Delta Recall:

cupboard	keyboard	plant	wall	surface	bottle	chair	handle	floor	c. lamp
0.33	0.22	0.22	0.21	0.12	0.12	0.08	0.07	0.06	0.05
sink	ceiling	window	oven	desk	screen	outlet	door		
0.01	0.00	-0.04	-0.08	-0.11	-0.18	-0.21	-0.28		

Delta F1 score:

cupboard	plant	keyboard	wall	sink	bottle	surface	desk	chair	handle
0.25	0.18	0.17	0.11	0.10	0.09	0.09	0.08	0.08	0.07
window	outlet	screen	ceiling	floor	c. lamp	oven	door		
0.06	0.05	0.02	0.02	0.01	0.01	-0.01	-0.04		

Figure 7.4: Method-2. Evaluation metrics separated by class. Note the improvement of the F1 with respect to method-1, in particular for the classes *door* and *oven*.

7.2 Smart Algorithm - Illustrated Examples

In the following illustrations we show some examples of classified sceneries. The images to the left represent the results with the baseline model, while the ones to the right depict the same scenery after the application of the Smart Algorithm. For each bounding box we show the corresponding class label and confidence value. Blue regions represent the objects correctly classified by the baseline model. Instead, the red regions have been assigned to a wrong class label. The green rectangles into the right images represent the objects whose label was incorrect with the baseline model and which were correctly updated by the Smart Algorithm.

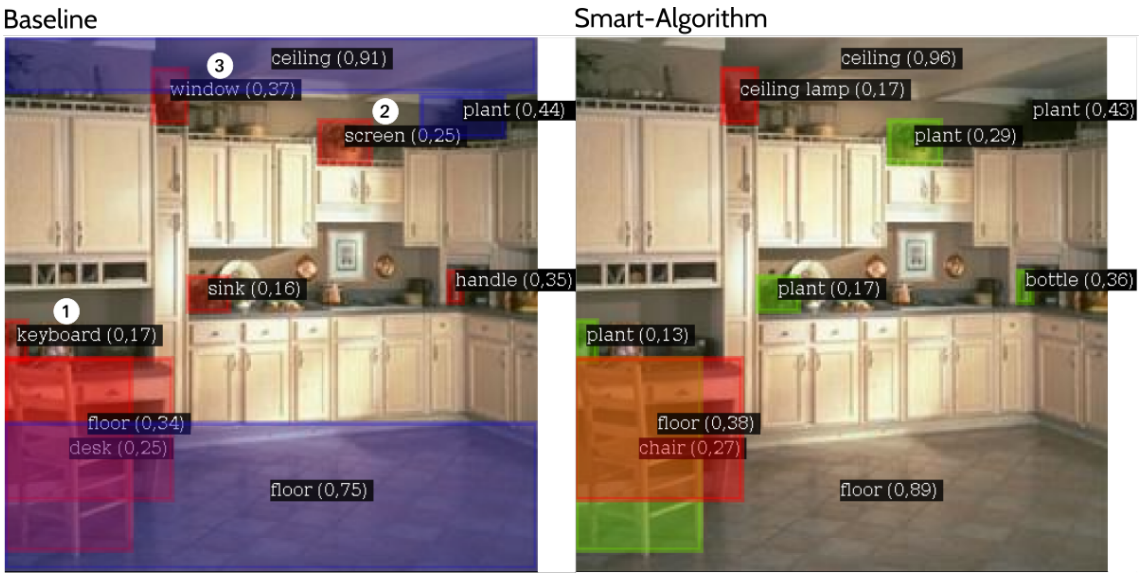

Figure 7.5: Example 1. We can notice in the left image that the *keyboard* (1) and *screen* (2) objects are clearly out of context, since they more likely appear in the office sceneries and not into kitchens. Also object (3) is incorrect because too small for being a *window*. Object (1) and (2) were correctly converted to the *plant* class. Instead, object (3) was updated to the *ceiling lamp* label because it is near to the ceiling: the reasoning is correct, but in this case brings to an error as the actual label would have been *plant*.

Figure 7.6: Example 2. In this picture we can notice the transition between *surface* and *floor* for object (1), motivated by its position to the bottom of the image. The window object (2) was already correct with the baseline model, but after the Smart Algorithm application has increased its confidence.

Figure 7.7: Example 3. The example shows an office scenery where the algorithm succeeded in updating object (1) from *oven* to *screen*.

Figure 7.8: Example 4. Kitchen scenery: *chair*, *plant* and *ceiling lamp* were correctly recognized by the Smart Algorithm.

Figure 7.9: Example 5. Object (1) is near to a *screen* and its label is more likely a *keyboard* than a *sink*.

Chapter 8

Conclusions

8.1 Summary

The aim of this thesis was to develop a new algorithm which could exploit semantics to enhance deeplearning models in the field of computer vision. We have presented in the first chapters the common challenges for image understanding and object recognition, pointing out the difference between classification and detection. These methods require standard measures to be evaluated and compared, in particular we described the mAP score which was used in the Pascal VOC contest for object detection. Modern models try to achieve these tasks by using deeplearning techniques, such as the convolutional neural networks. These can solve the issue with feature extraction and can work directly on the raw pixel of the image. To understand how convolutional neural networks can perform the image processing we explained the concept of tensors, which can represent multidimensional data or multi-channel pixel arrays. By analyzing the input tensors, the neural networks let data flow through their layers and transform the information into more abstract features. The last fully connected layer actually performs the classification.

We trained some CNN architectures on the Sun09 dataset, over 510 indoor sceneries with 18 class labels. To improve the performances, the context layer was added. The BGRC

format adds the information about the real shape and position of the object into the scenery. This helps the convolutional neural network which is a local model and it is not able to consider the whole scene. We learned that since our dataset is small the results cannot be very accurate, but they could be enhanced with the usage of semantics.

The novel method proposed in our thesis is called Smart Model. This uses the contextual information related to the objects inside a single scenery and corrects the CNN predictions. To achieve this task we defined some relative features between object pairs. They involve characteristics like size, position and luminosity. For each pair of objects the model tries to guess a correction of the predicted class label. The class scores update is based on the statistical information stored in the histograms, which are built by analyzing the training set sceneries. We obtained a method which improves the accuracy of about the 10% with respect to the baseline model. The algorithm is also quite fast, since its application on the 510 sceneries takes only few seconds on a common laptop.

8.2 Future Works

The Smart Algorithm presented in this thesis can really increase the accuracy of a pure CNN based model. There are many possibilities to improve our system and extend its usage to other state-of-the-art deeplearning model, instead of applying it to a small custom network. The main limit was the Sun09 dataset, which seemed great at the beginning of our work, but revealed quite small during the experiments. A better idea would be to use for example the state-of-the-art SegNet as the baseline model. This is a deeplearning network for pixelwise labeling of the input image, presented in CVPR 2015 [12]. There is an available instance pre-trained on the Sun RGB-D dataset, with labeled indoor sceneries [18]. The images in this dataset are captured with 3D cameras and contain a depth map which could be exploited to enhance the Smart Rules and create new types of relative features.

The pixelwise segmentation obtained as output from SegNet presents some noise, like small isolated regions labeled with an incorrect class. The semantic model should remove

this noise and use the contextual information to correct the label predictions. Instead of looking at the rectangular bounding boxes, we could also try to retrieve information from the actual shape of the classified regions. It could be interesting trying to generate a 3D reconstruction of the scene starting from the segmented image and merging it with the depth map. The reconstruction could be enhanced with the semantical data computed with our Smart Model.

Bibliography

[1] Stanislas Dehaene, *Coscienza e cervello. Come i neuroni codificano il pensiero*, Scienza e Idee, Raffaello Cortina Editore, 2014.

[2] Myung Jin Choi, Joseph J. Lim, Antonio Torralba, Alan S. Willsky, *Exploiting Hierarchical Context on a Large Database of Object Categories*, Massachusetts Institute of Technology, CVPR 2010.

[3] Marco Tulio Ribeiro, Sameer Singh, Carlos Guestrin, *"Why Should I Trust You?" Explaining the Predictions of Any Classifier*, KDD 2016.

[4] Alex Krizhevsky, Ilya Sutskever, Geoffrey E. Hinton, *ImageNet Classification with Deep Convolutional Neural Networks*, NIPS, 2012.

[5] Karen Simonyan, Andrew Zisserman, *Very Deep Convolutional Networks for Large-Scale Visual Recognition*, ICLR, 2015.

[6] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, Andrew Rabinovich, *Going deeper with convolutions*, CVPR, 2015.

[7] Ross Girshick, Jeff Donahue, Trevor Darrell, Jitendra Malik, *Rich feature hierarchies for accurate object detection and semantic segmentation*, CVPR, 2014.

[8] Ross Girshick, *Fast R-CNN*, ICCV, 2015.

[9] Shaoqing Ren, Kaiming He, Ross Girshick, Jian Sun, *Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks*, NIPS, 2015.

[10] Joseph Redmon, Santosh Divvala, Ross Girshick, Ali Farhadi, *You Only Look Once:*

Unified, Real-Time Object Detection, CVPR, 2016.

[11] George Papandreou, Liang-Chieh Chen, Kevin P. Murphy, Alan L. Yuille, *Weakly- and Semi-Supervised Learning of a Deep Convolutional Network for Semantic Image Segmentation*, ICCV, 2015.

[12] Vijay Badrinarayanan, Alex Kendall, Roberto Cipolla, *SegNet: A Deep Convolutional Encoder-Decoder Architecture for Image Segmentation*, IEEE Transactions on Pattern Analysis and Machine Intelligence, 2017.

[13] Hengshuang Zhao, Jianping Shi, Xiaojuan Qi, Xiaogang Wang, Jiaya Jia, *Pyramid Scene Parsing Network*, CVPR, 2017.

[14] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li and Li Fei-Fei, *ImageNet: A Large-Scale Hierarchical Image Database*, Dept. of Computer Science, Princeton University, USA, CVPR, 2009.

[15] Mark Everingham, Luc Van Gool, Christopher K. I. Williams, John Winn, Andrew Zisserman, *The PASCAL Visual Object Classes (VOC) Challenge*, International Journal of Computer Vision, 2010.

[16] Mark Everingham, John Winn, *The PASCAL Visual Object Classes Challenge 2012 (VOC2012) Development Kit*, May 18, 2012.

[17] Nathan Silberman, Rob Fergus *Indoor Scene Segmentation using a Structured Light Sensor*, ICCV, 2011.

[18] Shuran Song, Samuel, P. Lichtenberg, Jianxiong Xiao, *SUN RGB-D: A RGB-D Scene Understanding Benchmark Suite*, Princeton University, CVPR, 2015.

[19] Gaurav Kumar, Pradeep Kumar Bhatia, *A Detailed Review of Feature Extraction in Image Processing Systems*, Advanced Computing & Communication Technologies (ACCT), 2014.

[20] Dong ping Tian, Baoji, Shaanxi, *A Review on Image Feature Extraction and Representation Techniques*, International Journal of Multimedia and Ubiquitous Engineering, 2013.

[21] Rajkumar Goel, Vineet Kumar, Saurabh Srivastava, A. K. Sinha, *A Review of Feature*

Extraction Techniques for Image Analysis, ICACTRP, 2017.

[22] Martìn Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viègas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng, *TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems*, 2015.

[23] John Duchi, Elad Hazan, Yoram Singer, *Adaptive Subgradient Methods for Online Learning and Stochastic Optimization*, Journal of Machine Learning Research 12, 2011.

[24] Zeiler, Matthew D, *ADADELTA: An adaptive learning rate method*, arXiv, 2012.

[25] Diederik P. Kingma, Jimmy Lei Ba, *Adam: A Method for Stochastic Optimization*, arXiv, 2014.

[26] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, Ruslan Salakhutdinov, *Dropout: A Simple Way to Prevent Neural Networks from Overfitting*, Journal of Machine Learning Research 15, 2014.

[27] Chenxia Wu, Ian Lenz and Ashutosh Saxena, *Hierarchical Semantic Labeling for Task-Relevant RGB-D Perception*. Department of Computer Science, Cornell University, USA. Robotics: Science and Systems (RSS) 2014.