POLITECNICO DI TORINO

Corso di Laurea Magistrale in Ingegneria Elettronica

Tesi di Laurea Magistrale

A Correction Method for Pedestrian Detection using a Convolutional Neural Network

Relatori:
prof. Guido Masera
prof. Maurizio Martina

Candidato:
Stefania Trapani

Aprile 2018

Acknowledgments

Table of contents

List of tables

List of figures

Chapter 1

Introduction

1.1 Pedestrian detection and Advanced Driver Assistance Systems

Advanced Driver Assistance Systems [1], referred to as ADASs, have been developed in order to help drivers avoiding traffic accidents. They can be also used for autonomous-driver cars, hence it's a very interesting and much studied field in research area. ADASs make use of image processing and many sensors, such as infrared, radars and lidars.
Some of the techniques included among ADASs are: Electronic Stability Control (ESC), Line Keeping Assistance (LKA), Adaptive Cruise Control (ACC), Lane Departure Warning System (LDWS), Anti-Lock Braking System (ABS), Cooperative Intersection Collision Avoidance System (CICAS) and Driver Drowsiness Detection.

Another task of the ADAS, maybe the most important due to its criticality, is the *Pedestrian Detection* [2]. Through the last decades, it has become a challenging topic, since the variety of peculiarities that can determine the features of a pedestrian. They can be different in size, clothes and in the way they appear (front, rear or side pose). The difficulties come not only from the pedestrians themselves, but also from the surrounding environment, such as light level or the presence of object between the camera and the person.
Since pedestrian detection is useful in other fields, like surveillance and robotics, this problem has been approached in different ways, changing both method and dataset.

Among the principal methods used for pedestrian detection, there are:

- Support Vector Machines (section 1.2.1)
- Histograms of Oriented Gradients (section 1.2.2)

- Neural Networks (sections 1.2.3 and 3)
- Deformable Part Detectors
- Mixed version of the previous ones

Datasets are a large set of photos, videos or frames taken from videos, and are very useful to train and evaluate the robustness of the algorithm. Depending on what is the goal pursued by the researchers, many datasets have been produced, such as:

- MNIST
- Caltech Dataset
- ETH Dataset
- INRIA Dataset
- PASCAL VOC
- DITS
- GTSRB and GTSDB
- KUL Belgium Traffic Sign dataset

A detailed description of the datasets can be found in sections 4 and 5.3.

The purpose of this thesis work is to design a module able to refine the probability to detect a pedestrian with the help of surrounding objects (road signs, cars, dogs, etc.). In order to accomplish this, a Convolutional Neural Network has been employed and some pre-existent datasets have been combined. The network is a Tiny-YOLO version [3], written in Python language, which will be presented in section 5.4.

1.2 Widely used methods

1.2.1 Support Vector Machines

One of the first and simpler algorithms used for pattern recognition is the *Support Vector* [4] one.
The Support Vector Machines (SVMs) have been created in order to solve two-group classification problems. The basic idea is to find the optimal hyperplane which

separates the input vectors into two classes and maximize the margin between them. The set of training vectors $\boldsymbol{x}_N$ are separated as follows:

$$y_i = \boldsymbol{w}^T \cdot \boldsymbol{x}_i + b \geq 1 \qquad \forall \boldsymbol{x}_i \in 1 \;\Rightarrow\; y_i \in Class1 \tag{1.1}$$

$$y_i = \boldsymbol{w}^T \cdot \boldsymbol{x}_i + b \leq -1 \qquad \forall \boldsymbol{x}_i \in -1 \;\Rightarrow\; y_i \in Class - 1 \tag{1.2}$$

where $\boldsymbol{w}$ is the weights vector and b is the bias.
Since the distance between $Class1$ and $Class - 1$ is

$$d = \frac{2}{\|w\|}, \tag{1.3}$$

in order to maximize it, the $\|w\|$ quantity has to be minimized.

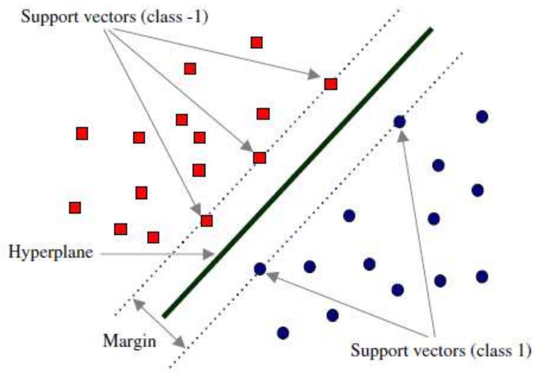


Figure 1.1: Support Vector Machine

The most successful fields of application for the Support Vector Machine are *handwritten character recognition*, *bioinformatics* and *machine vision*.

1.2.2 Histograms of Oriented Gradients

The *Histograms of Oriented Gradients* (HOG) method [5] is based on the idea that, in a restricted portion of the image, object shapes can be characterized by the distribution of intensity gradients or edge directions.
This implementation is composed of *five* phases, as pictured in 1.2:

1. The image window is subjected to a colour and gamma normalization, which helps to reduce illumination and shadowing effects on it. However, this preprocess stage is optional, since it does not affect performances.

2. The first order gradient of the image is computed. What comes out from this stage is a new image containing the object shapes present in the original one.

3. The image is divided into small *cells* and, in each of them, a local 1-D histogram of gradient or edge orientations is accumulated.

4. A further contrast normalization is applied to local groups of cells (called *blocks*), accumulating a measure of the local histogram *"energy"*. After that, the normalized block descriptors are named *Histogram of Oriented Gradient descriptors*.

5. The HOG descriptors from all blocks are collected in a feature vector.

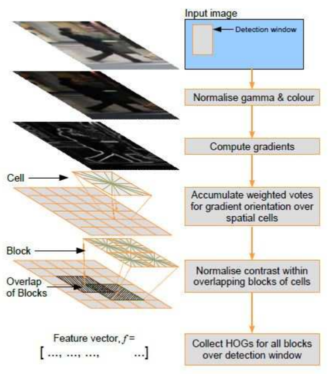

Input image
Detection window
Normalise gamma & colour
Compute gradients
Cell
Accumulate weighted votes for gradient orientation over spatial cells
Block
Overlap of Blocks
Normalise contrast within overlapping blocks of cells
Feature vector, f =
[..., ..., ..., ...]
Collect HOGs for all blocks over detection window

Figure 1.2: Histograms of Oriented Gradients

1.2.3 Neural Networks

In the last decade, *Neural Networks* became more popular, thanks to its usage in different fields. Depending on the way they process data, there exist many types of Neural Networks, but it can be stated that the data processing layers are similar (fig. 1.3). These layers can be grouped as three pipeline stages:

1. **region proposal** or **input layer**, where the entire frame is examined and image regions hopefully including, for example, a person are extracted. Therefore, its input is the complete image, while the output is a collection of regions which probably contain a pedestrian. One of the most used algorithms for region proposal is the *sliding window approach*, since it manages regions extraction at multiple scales and aspect ratios.

2. **feature extraction** or **hidden layer**, whose input is the set of image regions previously generated. The output is a set of real or binary values (depending on how many classes compose the network), which can be compacted in a *features vector*.

3. **region classification** or **output layer**, whose purpose is to identify which regions within the set of candidates correspond to a human shape. In order to accomplish to this task, the features vector has to be sent to this stage, which provides the probability that such region contains a person.

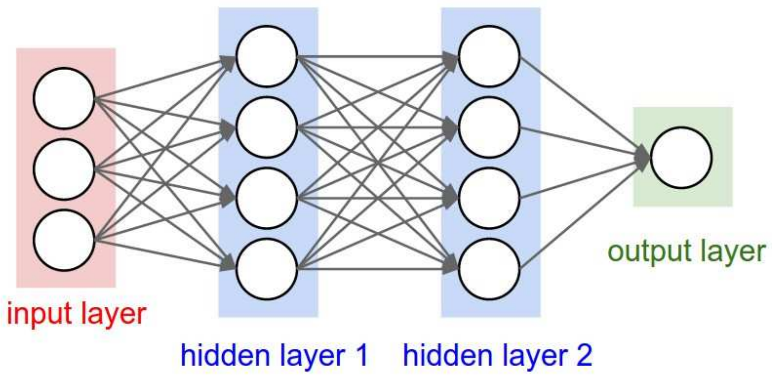


Figure 1.3: Layers composing a Neural Network

Further information about Neural Networks can be found in the next chapter.

Chapter 2

Convolutional Neural Network

2.1 Overview

A Neural Network, also called *Artificial Neural Network*, is a computing system inspired by natural neurons [6, 7]. It can be implemented as algorithms or actual hardware. Usually a brain has billions of neurons, while the Neural Network can have hundreds or thousands of them.

An Artificial Neural Network contains many highly-interconnected processing elements (fig. 2.1), which respond to external inputs, which play the role of the *synapses*. These inputs are then multiplied by weights ("activation functions"), computed by mathematical functions. In this way, the neuron has been activated and the output has been estimated. By adjusting the weights of an artificial neuron, the desired output for a given input can be stated.

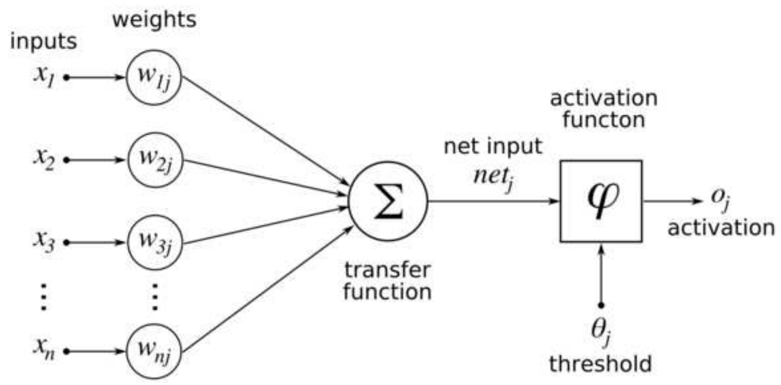


Figure 2.1: Artificial Neural Network

In order to revise the neurons weights, each neural network needs a learning rule, since Neural Networks learn by example. One of the most important and utilized learning rule is the *delta rule*: it is a supervised process, that occurs each time the network receives a new input. In other words, when a new input is applied to the network, it tries to guess the output. This kind of learning rule is the basis of the *Back-Propagation Neural Networks* (BPNNs), which allow the backward propagation of the error.

One crucial problem during the training of a network is the *overfitting* (fig. 2.2): the network only recognizes data from the training set, since it has been trained too extensively. In other words, overfitting occurs when a network memorizes training data rather than learning to generalize them. In order to avoid overfitting, some regularization techniques have to be employed, such as the *dropout* one.

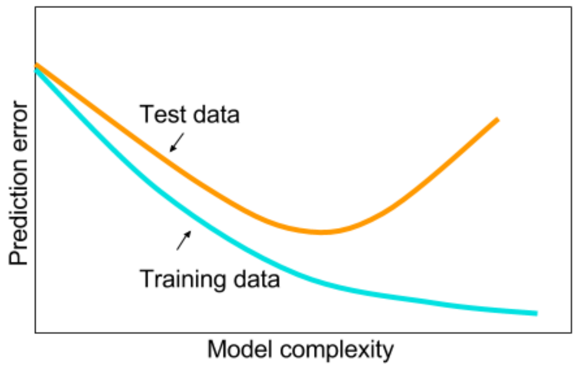


Figure 2.2: Overfitting Phenomenon

After training a Neural Network to an acceptable level, it can be used to analyse data, detect objects, etc. In order to do this, the network is used in forward propagation mode and each input is processed by the middle layers.

Depending on the connection pattern, in particular on the presence of loops, the network architecture can be divided in: *Feed-forward networks* and *Recurrent* or *Feedback networks.*

Convolutional Neural Networks (CNN) are a particular type of Neural Networks, composed by neurons with weights and biases. Their architectures are designed knowing that the inputs are images, and so the number of parameters in the network are reduced. The layers of a Convolutional Neural Network are composed of 3-dimension neurons: width, height, depth.

2.2 Typical structure of a Convolutional Neural Network

Depending on its application, a Convolutional Neural Network architecture [8] presents different sizes and shapes. However, some characteristics are common to all networks (fig. 2.3): they are typically organized in layers, to which some functions are applied. So, it can be stated that a CNN is composed of:

- Convolutional Layers
- Batch Normalization Functions
- Non-Linearity Functions
- Pooling Layers
- Fully-Connected Layers

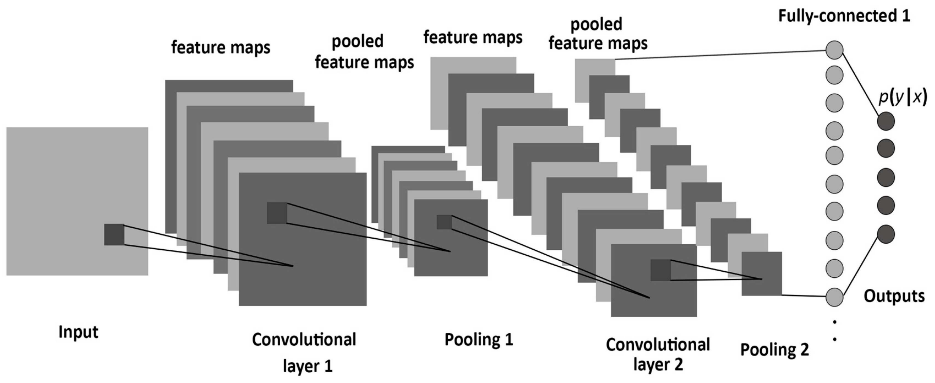


Figure 2.3: Typical structure of a Convolutional Neural Network

2.2.1 Convolutional Layers

The *Convolutional Layer* is a windowed and weight-shared layer much used during the design of a Neural Network.
This type of layer is composed of a set of filters, each of which is small in terms of width and height, but extends through the full depth of the input volume. In fact, during the convolutional stage, a small neighbourhood of the input is transformed in a weighted sum. To be more clear, the first layer of a Neural Network, for example, can be made of a filter with sizes $5x5x3$, which indicates *5 pixels* for width and height and *3 colour channels*.
Each filter is slid across the dimensions of the input and a dot product between

the filter elements and the input at any position is computed. The result is a 2-D activation map which is represented by the responses of that filter in every locations. Following these rules, the network will learn which filters are activated when recognizing some characteristics.
Recovering the example and supposing that the input has size $32x32x3$, it can be established that the number of weights is $5x5x3 = 75$ (+1 bias parameter) and each filter slides among the $32x32$ matrix.

Three hyper-parameters control the input dimensions:

- the **depth**, which corresponds to the number of filters used in the stage;
- the **stride**, which specifies the slide of filter, i.e. how many pixels are moved each time;
- the **zero-padding**, which defines how many zeros have to be added around the border. It is useful to have the same dimensions of the input.

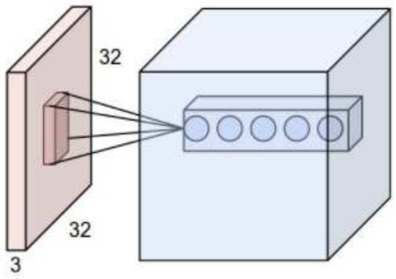


Figure 2.4: Convolutional Layer example

2.2.2 Batch Normalization Functions

In order to increase the stability and performances of a Neural Network, a *Batch Normalization function* needs to be added. This stage leads to improve accuracy in the training phase without loosing speed.
The input of each layer is normalized such that the mean output activation (μ) is *zero* and the standard deviation (σ) is *one*. Then, the normalized value is scaled and shifted:

$$y = \frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}}\gamma + \beta \tag{2.1}$$

where ϵ is a small constant to avoid numerical problems, while γ and β parameters are learned from training.
The Batch Normalization is usually performed between a convolutional or a fully-connected layer and the non-linearity function.

2.2.3 Non-Linearity Functions

A *Non-Linearity function* is an activation function, which is usually adopted after a Convolutional or a Fully-Connected layer, in order to introduce a non-linearity into the network. They are very important, since they are in charge of deciding if a neuron should be triggered or not. This stage simply works applying one activation function to the inputs and sending the transformed outputs to the next layer.
Various types of non-linear activation functions can be chosen, as depicted in figure 2.5:

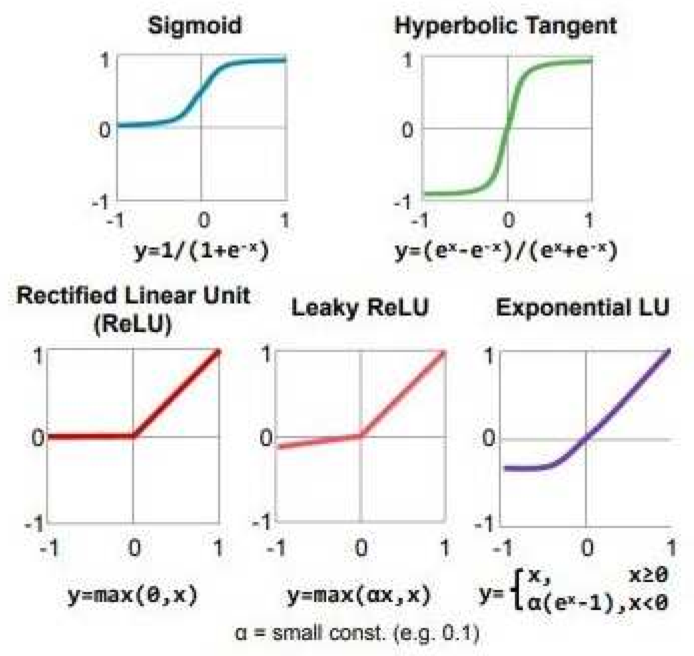


Figure 2.5: Non-Linearity Functions

2.2.4 Pooling Layers

A *Pooling Layer* is employed each time is necessary to progressively reduce the dimensions of a feature map, in order to limit over-fitting problems and to decrease network parameters and computation.
This stage is applied to each channel of the input separately, which is spatially resized using a *maximum* or *average* operation. A further feature, the *stride*, has to be established, in order to know which degree of reduction has to be used.

Both maximum and average operations on the input matrix can be observed in figure 2.6, where the input is represented by a 4x4 matrix and it has to be transformed in a 2x2 one (so the stride number is 2).

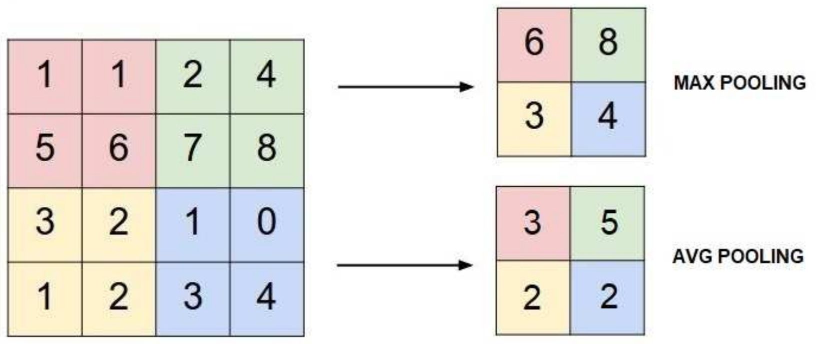


Figure 2.6: Pooling Layer example

2.2.5 Fully-Connected Layers

In a *Fully-Connected layer*, all neurons involved in the previous layer (convolutional, fully-connected or pooling) are connected to each of its neurons. In other words, the output neurons are composed of the weighted sum of each input neuron:

$$output = (weights \times input) + bias \tag{2.2}$$

After using this type of layer, the output dimensions are *1x1xN*, as can be seen in figure 2.7. When there are more than one fully-connected layers, the last one (called *output layer*) will compute the ratings for each category. This happens when the network is used for classification purposes.

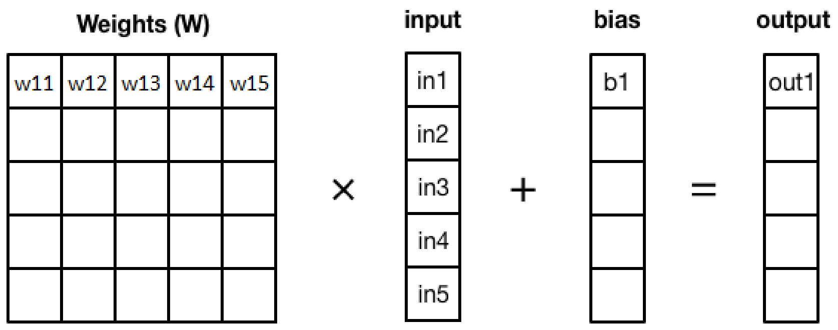


Figure 2.7: Fully-Connected Layer example

2.3 Fields of use

Nowadays, Neural Networks are very popular in many applications and every day new challenges are introduced to improve people's life. The well-known fields of use for Neural Networks are:

- ***Image and video***: they are useful, among the other tasks, for objects classification or security surveillance. Their application is widespread and computer vision helps to extract meaningful information from them. Videos can be also employed for real-time systems, for which fast responses are needed.
- ***Speech and language recognition***: Neural Networks are employed for machine translation, speech recognition and language processing.
- ***Medical***: Neural Networks have been very important to understand the genetics of some diseases and to detect them. Thanks to this type of Neural Network, it has been possible to detect various types of cancers, such as brain, breast and skin ones.
- ***Robotics***: the success of these type of Neural Networks is stated by their numerous applications, such as the self-driving cars and the motion of a robotic arm.

Chapter 3

Model examples for CNNs

As already mentioned in the previous chapters, depending on the pursued purpose, Convolutional Neural Networks [8] presents different architectures. The main features of an architecture are:

- number of filters
- filter sizes
- number of layers
- types of layers

In the next sections, the most popular Convolutional Neural Networks will be introduced.

3.1 LeNet

The first application of Convolutional Neural Network, named *LeNet* [9], was developed by Yann LeCun in the late 1980s.
This model was designed primarily for the handwritten digit recognition in the MNIST dataset, which is composed of 70.000 grayscale images of dimensions 28x28.
In order to accomplish its task, the network (fig. 3.1) was composed of:

- **2 convolutional layers**, with 6 filters in the first layer and 16 in the second one (each layer uses 5x5 filters);
- **Sigmoid function** for the non-linearity;
- **Average pooling function** of 2x2 after each convolutional layer;

- **2 fully-connected layers**.

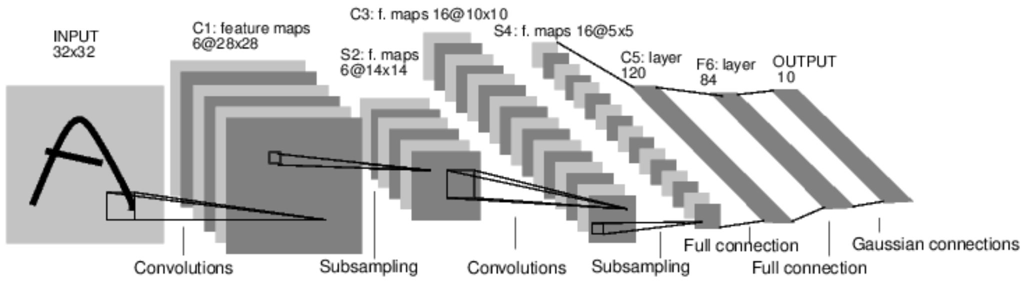


Figure 3.1: LeNet architecture

3.2 AlexNet

The first Convolutional Network conceived for Computer Vision purposes was the *AlexNet* architecture [10]. It was proposed at the ImageNet Competition in 2012, introducing the concept of *Local Response Normalization*.
This model (fig. 3.2) consists of:

- **5 convolutional layers**, with different number and sizes;
- **ReLU function** for the non-linearity;
- **Max pooling function** of 3x3 after convolutional layers no. 1, 2 and 5;
- **3 fully-connected layers**.

The first convolutional layer contains 3 channels, corresponding to the RGB (Red, Green and Blue) components of the input image.
The main difference between AlexNet and LeNet is that the number of weights is much larger and the shapes vary according to the layers.

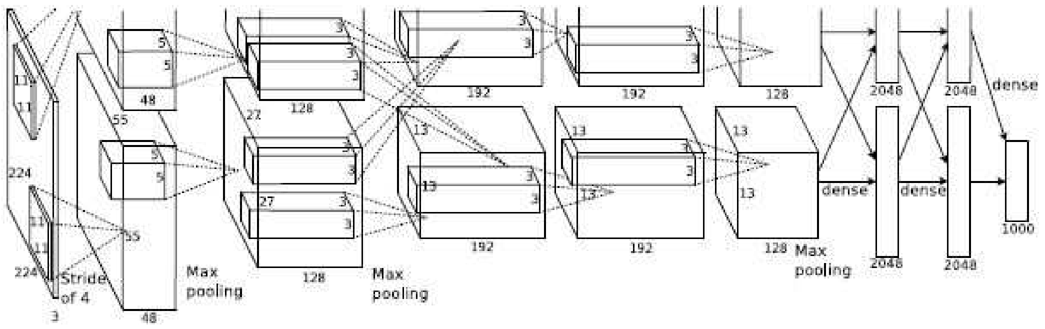


Figure 3.2: AlexNet architecture

3.3 VGG Networks

In 2014 Karen Simonyan and Andrew Zisserman discovered that the depth of a network is critic for good performance. This led to the realization of the *VGGNet* [11].
The final version of this network, called *VGG-16*, as its name suggests, is 16 layers deep. In particular, it consists of:

- **13 convolutional layers**, with the same filter size (3x3);
- **ReLU function** for the non-linearity;
- **Max pooling function** of size 2x2;
- **3 fully-connected layers**.

In the convolutional layers, large filters are decomposed in smaller ones, which have fewer weights. This explains why, in that stages, filters have the same size. The principal disadvantage of the VGG-16 network is that it is more expensive to evaluate and uses a lot of memory and parameters.

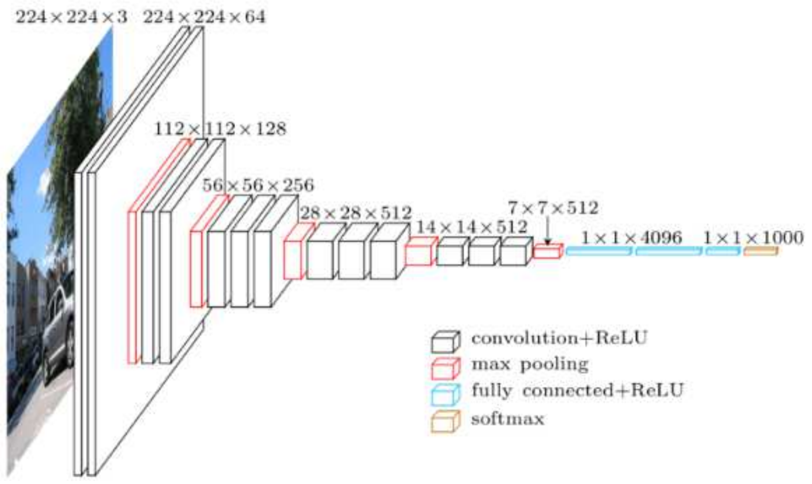


Figure 3.3: VGG-16 architecture

3.4 ResNet

The *ResNet* [12], also known as *Residual Net*, was proposed at the ImageNet Challenge in 2015, being the first Neural Network that exceeded human-level accuracy. Its idea is based on residual connections to build a deeper network; indeed, it is

composed of 34 or more layers. This model is motivated by the fact that the ability to update the weights in the earlier layers is degraded, since, as the error back-propagates through the network, the gradient decreases.
The Residual Network introduces a module which contains an identity connection in order to skip the weight layers (convolutional layers). Rather than learning the function for the weight layers *F(x)*, this module learns the residual mapping:

$$F(x) = H(x) - x \tag{3.1}$$

In addiction to this, in order to reduce the number of the weights, the two 3x3 layers involved in the module are replaced by three layers of sizes 1x1, 3x3, 1x1. This transformation is observable in fig. 3.4:

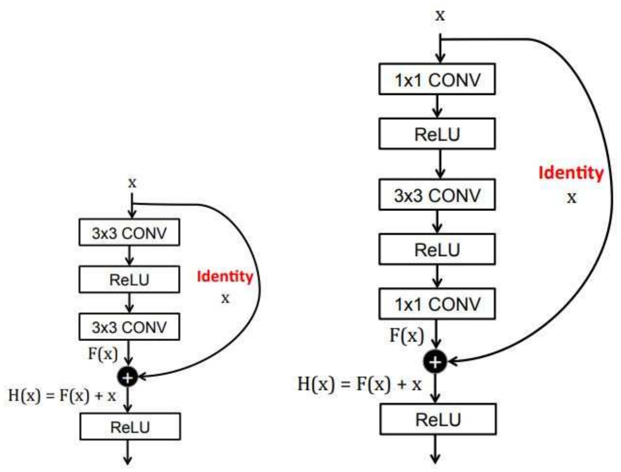


Figure 3.4: ResNet module

3.5 GoogLeNet

The *GoogLeNet* [13] is a Convolutional Network proposed at the ImageNet Competition in 2014. Its key point is the introduction of an *Inception Module*, which, realizing parallel connections, reduces the number of the parameters in the network. In this architecture, as can be seen in fig. 3.5, parallel connections are designed with the help of a 3x3 max-pooling layer and of filters with sizes 1x1, 3x3 and 5x5. The choice of using different filters sizes helps to process the input at multiple scales.

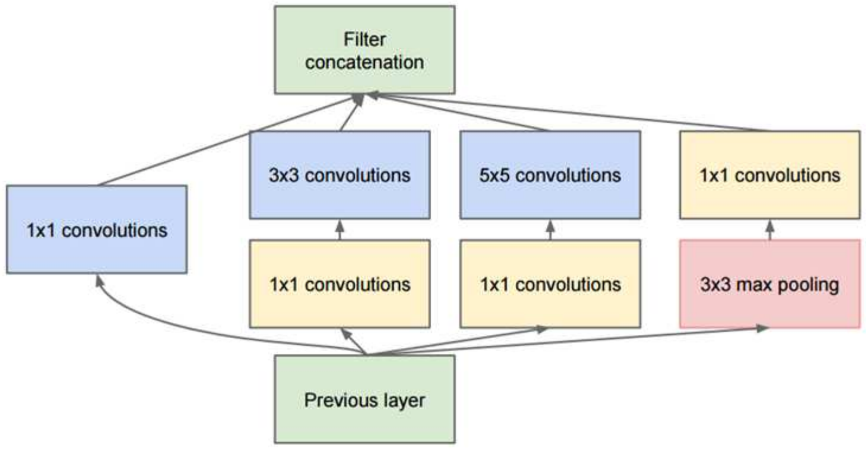


Figure 3.5: GoogLeNet Inception Module

The whole GoogLeNet architecture (fig. 3.6) is composed of 22 layers:

- **3 convolutional layers**

- **9 inception layers**

- **1 fully-connected layer**

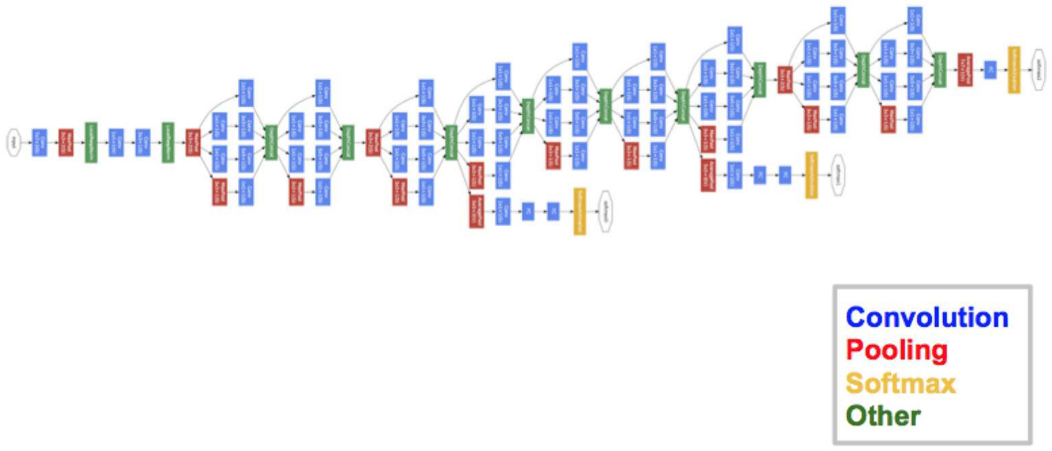


Figure 3.6: GoogLeNet architecture

3.6 Darknet and YOLO

Darknet is an open source Neural Network framework, which supports CPU and GPU computation. It relies on the fact that the Neural Network receives the whole image, which is then divided into regions. At this point, the network predicts multiple bounding boxes and the related probabilities for each class. This constitutes the main difference from other networks, where the model is applied to an image at multiple locations and scales and the regions with higher probabilities are considered *positives*.

YOLO [3], which stands for *"You Only Look Once"*, belongs to the Darknet types of networks. Its basic idea is to design the network merging the components of object detection into a single Neural Network.
YOLO uses features from the full image to predict simultaneously the bounding boxes and the right class among the trained ones. Its design allows end-to-end training and real-time speeds while maintaining high average precision.
The input image is divided into an SxS grid. The grid cell is in charge of detecting an object, if the center of it falls into a grid cell. Each grid cell predicts B bounding boxes and the confidence results for those boxes. Confidence is defined as:

$$confidence = Pr(obj) \cdot IOU_{truth}^{pred} \tag{3.2}$$

where IOU represents the *Intersection Over Union* between the predicted box and the ground truth, which is *zero* when there is no object in the cell.
Each bounding box consists of *5 predictions*:

- **x** and **y**, which are the coordinates of the center of the box relative to the bounds of the grid cell
- **w** and **h**, which represent width and height relative to the whole image
- **confidence**, defined in equation 3.2

Each grid cell also predicts C conditional class probabilities, $Pr(Class_i|Obj)$. YOLO only predicts one set of class probabilities per grid cell, regardless the number of bounding boxes B. The predictions are encoded as a tensor with shape:

$$S \times S \times (B \cdot 5 + C) \tag{3.3}$$

Making an example for evaluating YOLO network on the PASCAL VOC dataset, where the parameters are $S = 7$, $B = 2$ and $C = 20$, the output is a 7 x 7 x 30 tensor.

YOLO architecture (fig. 3.7) is similar to the GoogLeNet (section 3.5) model for image classification. The network consists of:

- **24 convolutional layers**;
- **leaky-ReLU function** for the non-linearity;
- **Max pooling function**;
- **2 fully-connected layers**.

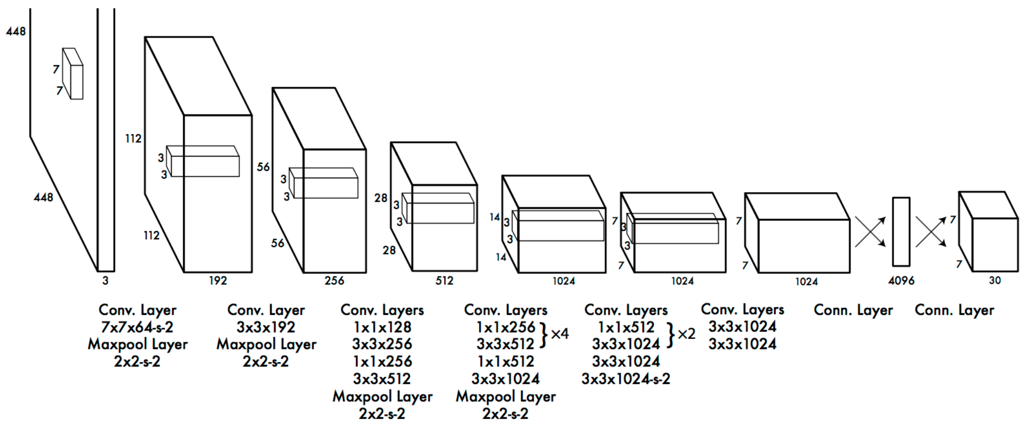


Figure 3.7: YOLO architecture

The convolutional layers at the beginning of the network extract features from the image, while the fully-connected layers predict the output, providing probabilities and coordinates. While GoogLeNet makes use of the inception modules, in YOLO 1x1 reduction layers are used followed by 3x3 layers.

There also exists a fast version of YOLO, which uses 9 convolutional layers and fewer filters in those layers.

Chapter 4

Datasets for Classification Purposes

As mentioned in section 1.1, dataset can be a large amount of photos, videos or frames from videos. The choice of a dataset is based on the classes to be detected and so on its task. In general, among the others, the most important classes are:

- Digits
- Persons
- Animals
- Vehicles for Transportation
- Road Signs

4.1 Caltech Dataset

The *Caltech Pedestrian Dataset* [14] is made of frames extracted from a 10 hours video taken from a vehicle driving through city traffic. In order to label each frame, researchers implemented an interactive annotating tool, which operates depending on the pedestrian visibility. For every frame containing a completely visible pedestrian, the annotator drew a Bounding Box (BB) indicating the region where to find the entire pedestrian. Instead, for pedestrians which are covered by something, the visible region is bounded as usual and an estimation of the hidden parts location is performed. Then, each bounding box is marked according to one of the following labels:

1. Individual pedestrians as *'Person'*

2. Large groups of pedestrians as *'People'*
3. Ambiguous or easily mistaken pedestrian as *'Person?'*

Pedestrians are categorized according their image height in pixels into three scales: near (above 80 pixels), medium (between 30-80 pixels) and far (below 30 pixels). For automotive applications, a medium scale detection is fundamental, since allows to have sufficient time to alert the driver about the incoming danger.

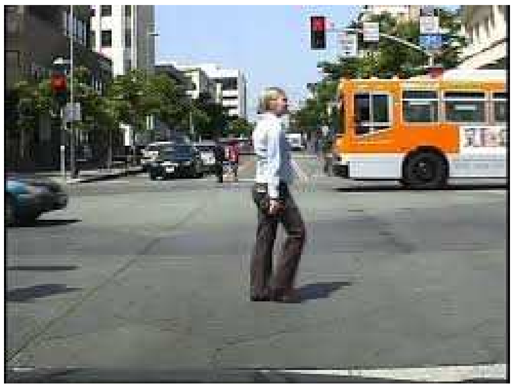


Figure 4.1: Caltech Pedestrian dataset example

Another type of Caltech dataset is the *Caltech-256 Object Category Dataset* [15], which involves classes different from the pedestrian one. It is an evolution of the Caltech-101 dataset and contains more than 30.000 images. These images don't need pre-processing steps, since, after collecting them from Google and Picsearch, they were labelled following some criteria:

- **Good**, if the image represents a clear example for the involved category;
- **Bad**, if the image is a drawing or leads to confusion;
- **Not applicable**, if the image does not contain the involved category.

and only the *good* ones constitute the dataset.
Among the others, the most important categories for urban recognition are:

- People
- Pram
- Dog
- Traffic Light

- Bike (both touring and mountain)
- Bulldozer
- Fire-truck

Figure 4.2: Caltech-256 dataset example

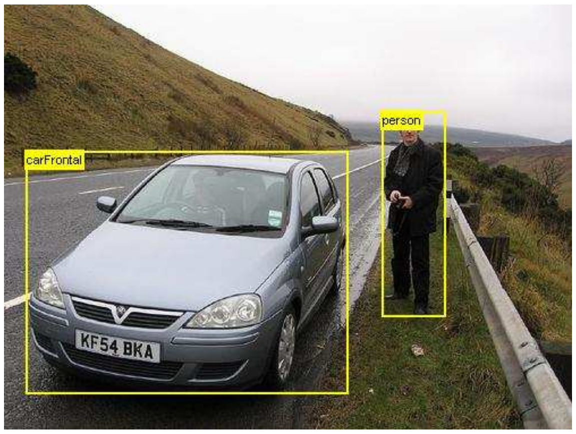
4.2 PASCAL VOC

The *PASCAL VOC dataset* [16] provides a large image set for object class recognition, which is created concurrently to an annual challenge.
The biggest dataset (*VOC2012*), which contains images from 2005[1] to 2012, is composed of more than 11.000 images representing 20 classes (*person* and varieties of *animals*, *vehicles* and *indoor objects*). Images are provided along with the associated annotation files (in *.xml* format), which contain the coordinates of the regions of interest (ROI) and the object class and pose (frontal, rear or lateral).
Depending on the pursued task, the challenges are different:

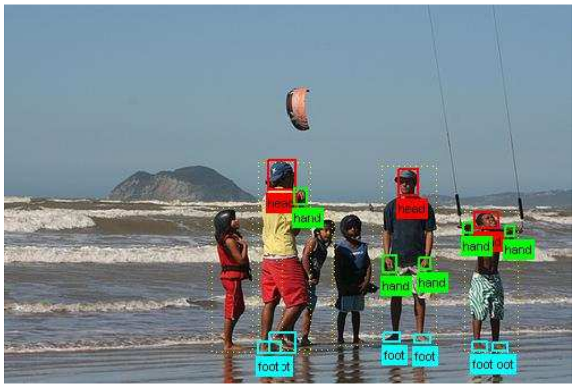
- **Classification**: for each of the 20 classes, it has to predict the presence or the absence of at least one object of that class in a test image;
- **Detection**: for each of the 20 classes, it has to predict the bounding boxes of each object of that class in a test image (fig. 4.3a);
- **Person Layout Evaluator**: for each person in a test image, it has to predict the bounding box of the person, the presence or the absence of human parts, such as *head/hands/feet*, and the bounding boxes of those parts (fig. 4.3b).

[1]This dataset contains images from Caltech, ETH, INRIA, TUGraz and UIUC

All of them have to produce also the real-valued confidence of the related bounding box.



(a) Detection



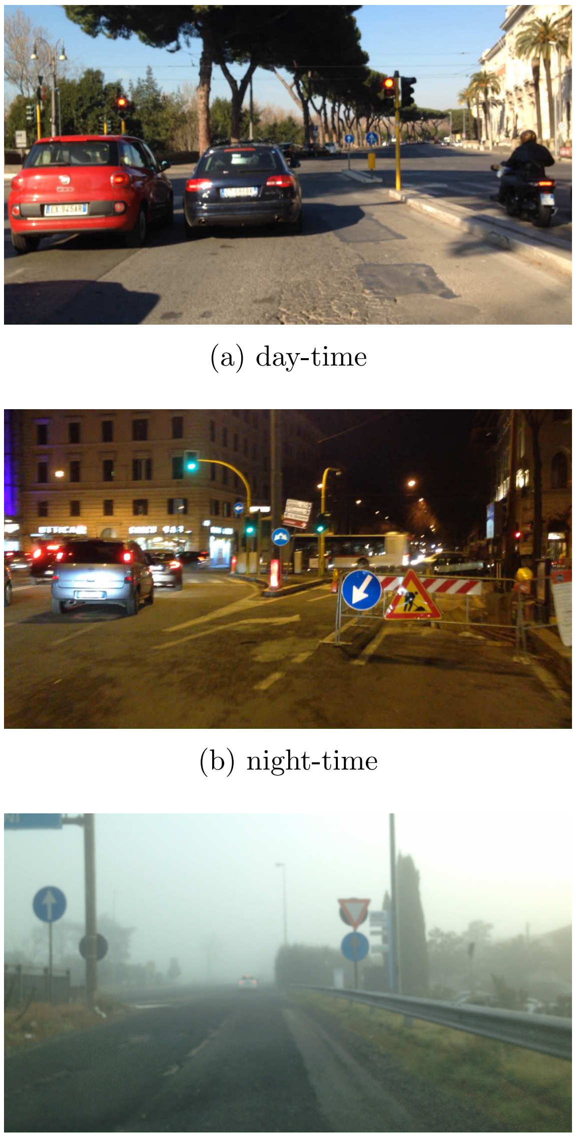
(b) Person Layout Evaluator

Figure 4.3: PASCAL VOC dataset examples

4.3 DITS

The *Data set of Italian Traffic Signs* (DITS[2]) is composed of images extracted by several hours of videos recorded in Italy and divided into 58 classes folders. It's a project managed by some researchers in the Department of Computer, Control, and Management Engineering (Sapienza University of Rome).
One of the main advantages of using this dataset is that some images are taken in different weather and illumination conditions, such as day-time (fig. 4.4a) or night-time (fig. 4.4b) and fog ones (fig. 4.4c).

[2]`http://www.dis.uniroma1.it/~bloisi/ds/dits.html`



(a) day-time

(b) night-time

(c) fog conditions

Figure 4.4: DITS dataset examples

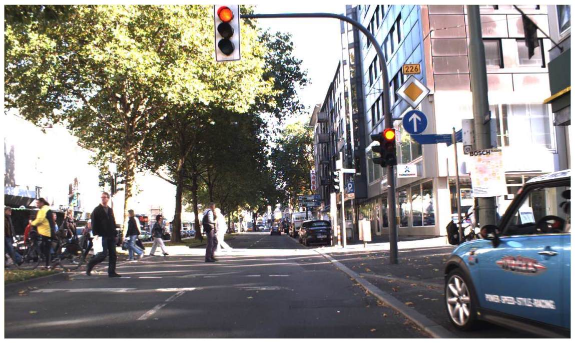
4.4 GTSRB and GTSDB

The *German Traffic Sign* datasets have been created by a research team of the Ruhr University of Bochum[3].

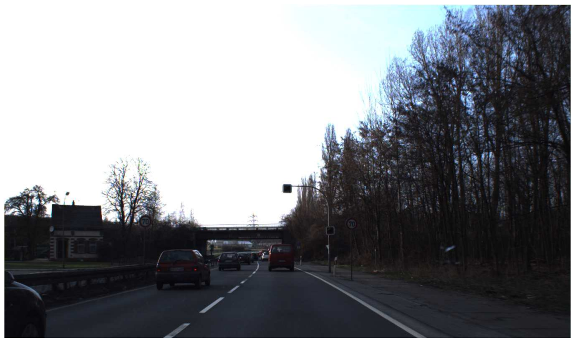
[3]`http://benchmark.ini.rub.de`

The *German Traffic Sign Recognition Benchmark* (*GTSRB*) is composed of more than 50.000 images depicting only road signs. They have different sizes (from 15x15 to 250x250 pixels) and are divided in 43 classes folders, each of which contains a *.csv* file, were the coordinates of the Regions of Interest are annotated.

The *German Traffic Sign Detection Benchmark* (*GTSDB*) is composed of 900 images, whose dimensions are 1360 x 800 pixels. They are captured by a vehicle not only in a urban scenario (fig. 4.5a), but also in a highway one (fig. 4.5b).



(a) Urban scenario



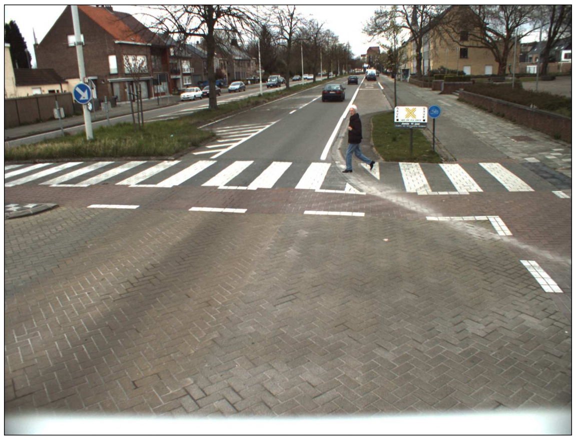
(b) Highway scenario

Figure 4.5: GTSDB dataset examples

4.5 KUL Belgium Traffic Sign dataset

The *Belgium Traffic Sign dataset* [17] consists of more than 9000 images, where traffic signs are visible at less than 50 meters from the camera. It has been created by a research team at Catholic University of Leuven.
The number of included classes, so of the traffic signs, is 65. Data can be divided in subsets, for both training and validation purposes. In order to include also negative

samples for the training phase, researchers use urban scenario images without traffic signs.


(a) image with 2 traffic signs

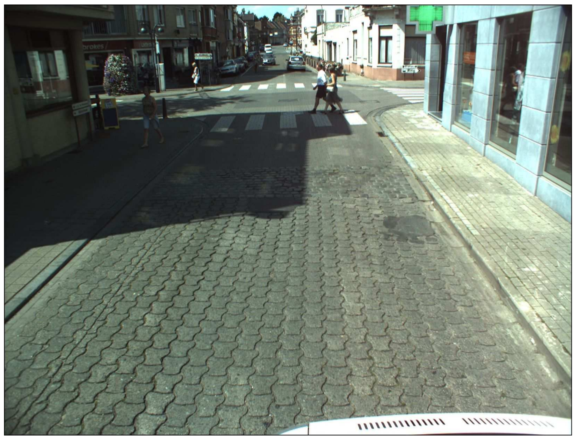
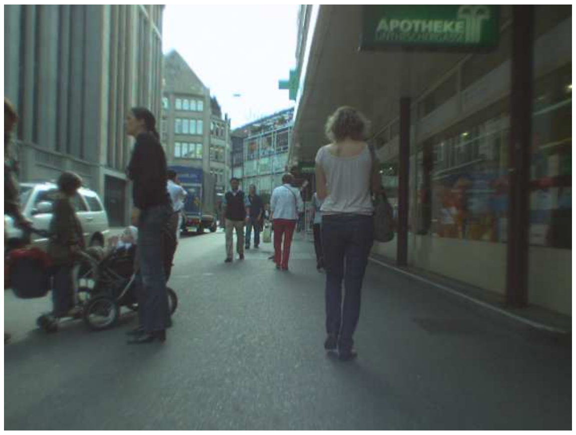

(b) image with no traffic signs

Figure 4.6: KUL Belgium Traffic Sign dataset examples

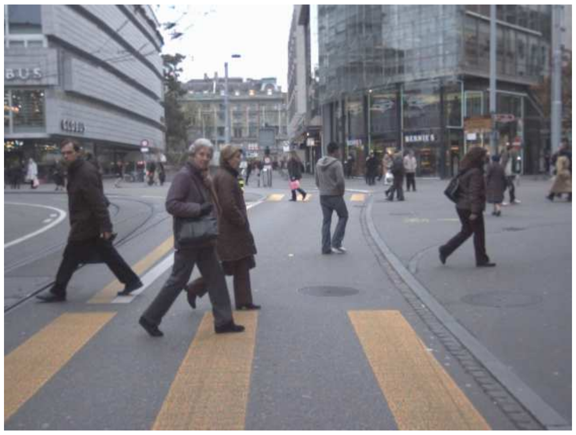
4.6 ETH dataset

The *ETH dataset* [18, 19] has been created by a research team of the Swiss Federal Institute of Technology in Zurich.
Each image of the dataset contains at least one pedestrian, which can be occluded or not. This constitutes a good detection dataset, since, in this way, it's possible to determine how robust is a network.

If this dataset is used for training, in addition to this, a set of bounding box annotations is provided.
Examples of the ETH dataset are presented in the figure below.



(a)



(b)

Figure 4.7: ETH dataset examples

Chapter 5

Implementation and Results

5.1 Overview

The basic idea behind this thesis work is to refine the Pedestrian Detection results produced by the Convolutional Neural Network. This is done using the probabilities of other detected objects in the same image.
In other words, speaking about probability theory, this situation can be seen as the measure of the probability of an event given that another event has occurred. This is the formal definition of *conditional probability*:

$$P(A|B) = \frac{P(A \cap B)}{P(B)} = \frac{P(A) + P(B) - P(A \cup B)}{P(B)} \tag{5.1}$$

Obviously, in this particular case, the intersection or the union of two events is not feasible, since formally there is no correlation between two different bounding boxes. One idea could be to annotate how many times, during the training phase, the two classes coexist in an image. However, this could lead to influence the samples, since desired images could be added in order to obtain a certain correlation between the classes.

Alternatively, classes can be correlated through a set of pseudo-random constants, based on common experiences. This approach is the one used for this thesis activity. Accordingly to this concept, the formulas to be applied are:

$$P(A \cup B) = \alpha \cdot (P(A) + P(B)) \tag{5.2}$$

$$P(A|B) = \frac{(1-\alpha) \cdot (P(A) + P(B))}{P(B)} \tag{5.3}$$

where α corresponds to the pseudo-random quantities, which are real values between 0 and 1.

Further information about α and the correlation of the employed classes can be found in section 5.2.

5.1.1 Flowchart

The implementation of this thesis (fig. 5.1) is composed of 4 stages:

1. **Region Of Interest (ROI) Annotation**

   - *input*: the set of training images;
   - *output*: annotated files in PASCAL VOC format (*.xml* files);

2. **Network Training**

   - *input*: the set of training images and the related annotation files from the previous stage;
   - *output*: the network weights;

3. **Objects Detection**

   - *inputs*: the weights from the previous stage and the set of images to be detected;
   - *outputs*: a modified copy of the original images, containing the bounding boxes, and the related annotation files with the coordinates and the confidence of each identified class;

4. **Probability Correction**, required only in case of wrong detection

   - *inputs*: the coordinates and confidence of the wrong bounding box, the label and probability of the helping class, and the correction file (*alphas.txt*) where pseudo-random quantities are saved in order to exploit the conditional probability formula;
   - *outputs*: a modified copy of the original images, containing the revised bounding boxes, and the related annotation files with the coordinates and the confidence of each class.

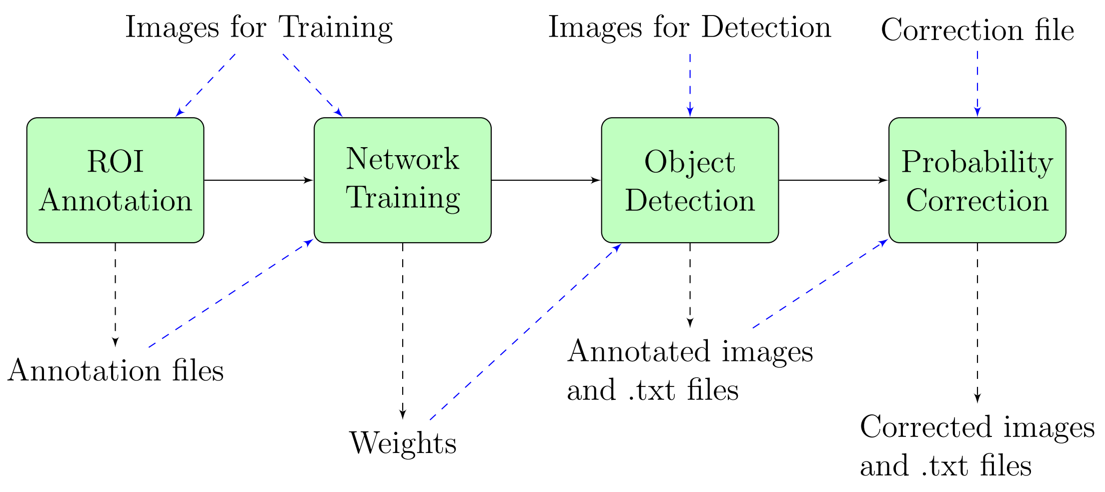


Figure 5.1: Flowchart of the implementation

5.2 Classes

Since persons have to be observed in urban scenarios, for the purpose of this thesis, some new classes have been created. The whole list of classes is the following:

1. **dog**
2. **person**
3. **car**
4. **bus**
5. **PBM**, i.e. Person on Bicycle or Motorcycle. This class has been created since the shapes they assume together are, in this work, more meaningful than those of a bicycle or of a motorcycle alone.

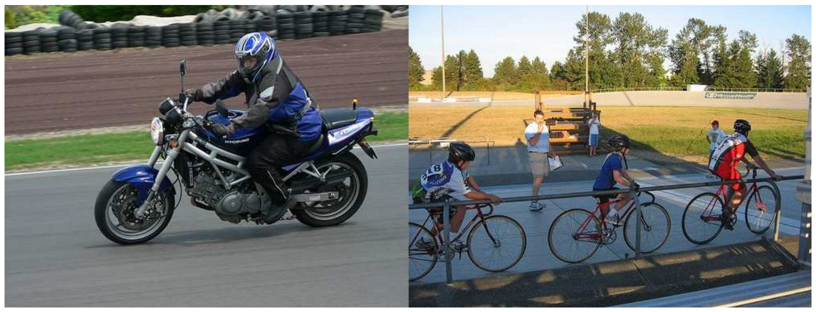


Figure 5.2: PBM class examples

6. **warningPos**, i.e. warning signals which increase pedestrian probability. Even if the pedestrian crossing signal with the blue background is an indication signal, it can be included in this class because of the features inside it.

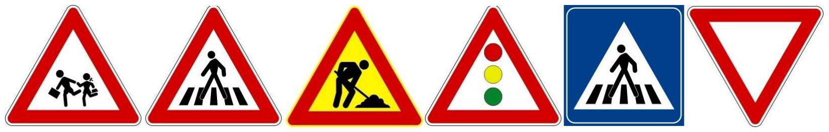


Figure 5.3: warningPos class examples

7. **trafficLight**
8. **stop**
9. **zebraCrossing**
10. **pram**
11. **prohibitoryPos**, i.e. prohibitory signals which increase pedestrian probability

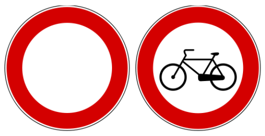


Figure 5.4: prohibitoryPos class examples

12. **obligationPos**, i.e. obligation signals which increase pedestrian probability

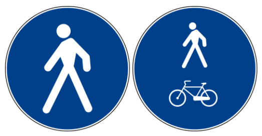


Figure 5.5: obligationPos class examples

13. **warningNeg**, i.e. warning signals which decrease pedestrian probability

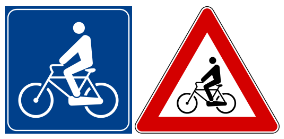


Figure 5.6: warningNeg class example

14. **prohibitoryNeg**, i.e. prohibitory signals which decrease pedestrian probability

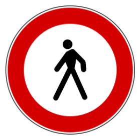


Figure 5.7: prohibitoryNeg class example

Road signs are collected into few classes, in order to avoid too many filters at the output layer of the Network. Recalling that the output of a YOLO-type Network is a predictions tensor with shape $S \times S \times [B \cdot (5 + C)]$ (equation 3.3), the number output filters are:

$$N_{filters} = B \cdot (5 + C) = 5 \cdot (5 + 14) = 95 \tag{5.4}$$

As mentioned in section 5.1, the pseudo-random quantities α are generated taking into account the correlation between classes. For example, the give-way road sign (*warningPos* class) helps the pedestrian probability to grow, while the detection of a no-pedestrian road sign (*prohibitoryNeg* class) is not.
In table 5.1, the objectives of each class are shown; in particular:

- "+" indicates that the class helps increasing the pedestrian probability;
- "-" indicates that the class helps decreasing the pedestrian probability;

class	**P**	**class**	**P**	**class**	**P**
dog	+	person	+	car	-
bus	+	PBM	-	warningPos	+
trafficLight	+	stop	+	zebraCrossing	+
prohibitionPos	+	obligationPos	+	pram	+
warningNeg	-	prohibitoryNeg	-		

Table 5.1: Pedestrian Helpers

In order to accomplish this task, taking into account the information given above, a Python file for the generation of the "alphas" is created (*alphaGen.py*):

```
import sys
import random
```

```
def probFunc(lab, start, stop):
   probs = lab + ":\n"

   for i in range(len(labels)):
      if (labels[i] == "person"):
         rdm = random.uniform(start, stop)
      else:
         rdm = random.uniform(0.84, 0.91)

      probs = probs + "\t%s %f\n" % (labels[i], rdm)

   return probs

labFile = open("labels.txt", "r")
labels = []

for line in labFile:
   labels.append(line[:len(line)-1])

file = open("alphas.txt","w")

file.write(probFunc("dog", 0.7, 0.8) + "\n")
file.write(probFunc("person", 0.6, 0.7) + "\n")
file.write(probFunc("car", 0.85, 1.0) + "\n")
file.write(probFunc("bus", 0.7, 0.8) + "\n")
file.write(probFunc("PBM", 0.85, 1.0) + "\n")
file.write(probFunc("warningPos", 0.65, 0.8) + "\n")
file.write(probFunc("trafficLight", 0.65, 0.8) + "\n")
file.write(probFunc("stop", 0.65, 0.8) + "\n")
file.write(probFunc("zebraCrossing", 0.6, 0.7) + "\n")
file.write(probFunc("prohibitoryPos", 0.6, 0.75) + "\n")
file.write(probFunc("obligationPos", 0.62, 0.77) + "\n")
file.write(probFunc("pram", 0.6, 0.65) + "\n")
file.write(probFunc("warningNeg", 0.9, 1.0) + "\n")
file.write(probFunc("prohibitoryNeg", 0.95, 1.0) + "\n")

file.close()
```

The produced output is in the form:

```
dog:
   dog 0.840572
   person 0.776773
   car 0.878812
   bus 0.892800
   PBM 0.877044
   warningPos 0.877657
   trafficLight 0.843473
   stop 0.854105
   zebraCrossing 0.844662
   prohibitoryPos 0.848044
   obligationPos 0.883571
   pram 0.885746
   warningNeg 0.886785
   prohibitoryNeg 0.905894

person:
   dog 0.857985
   person 0.647749
   car 0.882685
   bus 0.854004
   PBM 0.883794
   warningPos 0.880917
   trafficLight 0.865681
   stop 0.905100
   zebraCrossing 0.856251
   prohibitoryPos 0.876428
   obligationPos 0.901422
   pram 0.896362
   warningNeg 0.844165
   prohibitoryNeg 0.894666

PBM:
   dog 0.885336
   person 0.956782
   car 0.901154
   bus 0.895608
   PBM 0.897947
   warningPos 0.897834
   trafficLight 0.889612
   stop 0.886631
   zebraCrossing 0.889097
   prohibitoryPos 0.886647
   obligationPos 0.897668
   pram 0.844802
   warningNeg 0.849538
   prohibitoryNeg 0.885323
```

In this way, each class has different constants, which depend on the helping class. Obviously, if the results are not satisfying, it's possible to re-run the script to change the constants.

5.3 Dataset

Because of the introduction of new classes, for this thesis, two custom datasets have been created, one for the training phase and another for the detection phase.

The set of training images, which is composed of about 7.700 images, is a combination of some datasets discussed in section 4:

- *Caltech-256 Object Category*;
- *PASCAL VOC*;
- *Dataset of Italian Traffic Signs* (DITS);
- *German Traffic Sign - Recognition and Detection - Benchmarks* (GTSRB and GTSDB).

Even if some images are already annotated, some files need to be edited and others need to be created from scratch. In order to do this, a tool named LabelImg (section 5.5.2) has been used. In table 5.2, the statistics of the training dataset are shown.

class	**samples**	**class**	**samples**
person	9.430	dog	1.318
car	2.225	bus	565
pram	30	PBM	1.255
stop	197	warningPos	699
warningNeg	54	zebraCrossing	100
trafficLight	323	obligationPos	75
prohibitoryPos	85	prohibitoryNeg	3

Table 5.2: Training Dataset Statistics

The set of images for the detection phase is composed of 379 images taken from *KUL* and *ETH* datasets (sections 4.5 and 4.6). The results of the detection phase are presented in section 5.7.

5.4 Network

The network implemented in this thesis work is a modified version of the YOLO network, called *Tiny-YOLO*, which is smaller and faster, but less accurate. It, as shown in figure 5.8 and in table 5.3, is composed of *9 Convolutional layers* with 3x3 filters, followed by *6 Max pooling layers*. Each Convolution layer, except the last one, is followed by the Batch Normalization function and a Leaky ReLU as Non-Linearity function.

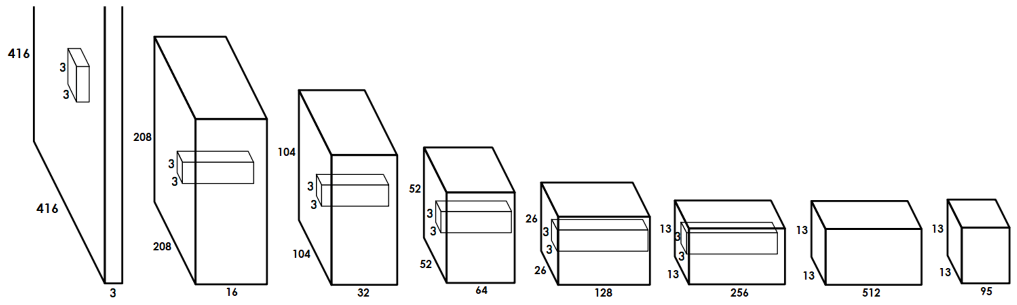


Figure 5.8: Tiny-YOLO Network architecture

Layer type	Filters	Output size
Input		$416 \times 416 \times 3$
Convolutional	16	$416 \times 416 \times 16$
Max Pooling		$208 \times 208 \times 16$
Convolutional	32	$208 \times 208 \times 32$
Max Pooling		$104 \times 104 \times 32$
Convolutional	64	$104 \times 104 \times 64$
Max Pooling		$52 \times 52 \times 64$
Convolutional	128	$52 \times 52 \times 128$
Max Pooling		$26 \times 26 \times 128$
Convolutional	256	$26 \times 26 \times 256$
Max Pooling		$13 \times 13 \times 256$
Convolutional	512	$13 \times 13 \times 512$
Max Pooling		$13 \times 13 \times 512$
Convolutional	1024	$13 \times 13 \times 1024$
Convolutional	1024	$13 \times 13 \times 1024$
Convolutional	95	$13 \times 13 \times 95$

Table 5.3: Tiny-YOLO Network

The code for the network configuration is presented below, where the *outputFilters* variable is determined by the equation 5.4:

```
def build_networks(self):
      self.x = tf.placeholder('float32',[None,416,416,3])
      self.conv_1 = self.conv_layer(1,self.x,16,3,1)
      self.pool_2 = self.pooling_layer(2,self.conv_1,2,2)
      self.conv_3 = self.conv_layer(3,self.pool_2,32,3,1)
      self.pool_4 = self.pooling_layer(4,self.conv_3,2,2)
      self.conv_5 = self.conv_layer(5,self.pool_4,64,3,1)
      self.pool_6 = self.pooling_layer(6,self.conv_5,2,2)
      self.conv_7 = self.conv_layer(7,self.pool_6,128,3,1)
      self.pool_8 = self.pooling_layer(8,self.conv_7,2,2)
      self.conv_9 = self.conv_layer(9,self.pool_8,256,3,1)
      self.pool_10 = self.pooling_layer(10,self.conv_9,2,2)
      self.conv_11 = self.conv_layer(11,self.pool_10,512,3,1)
      self.pool_12 = self.pooling_layer(12,self.conv_11,2,2)
      self.conv_13 = self.conv_layer(13,self.pool_12,1024,3,1)
      self.conv_14 = self.conv_layer(14,self.conv_13,1024,3,1)
      self.conv_15 = self.conv_layer(15,self.conv_14,outputFilters,1,1)

      self.sess = tf.Session()
      self.sess.run(tf.initialize_all_variables())
      self.saver = tf.train.Saver()
      self.saver.restore(self.sess,self.weights_file)


def conv_layer(self,idx,inputs,filters,size,stride):
      channels = inputs.get_shape()[3]
      weight =
          tf.Variable(tf.truncated_normal([size,size,int(channels),filters],
          stddev=0.1))
      biases = tf.Variable(tf.constant(0.1, shape=[filters]))

      pad_size = size//2
      pad_mat =
          np.array([[0,0],[pad_size,pad_size],[pad_size,pad_size],[0,0]])
      inputs_pad = tf.pad(inputs,pad_mat)

      conv = tf.nn.conv2d(inputs_pad, weight, strides=[1, stride, stride,
          1], padding='VALID',name=str(idx)+'_conv')
      conv_biased = tf.add(conv,biases,name=str(idx)+'_conv_biased')
      return
          tf.maximum(self.alpha*conv_biased,conv_biased,name=str(idx)+'_leaky_relu')
```

```
def pooling_layer(self,idx,inputs,size,stride):
     return tf.nn.max_pool(inputs, ksize=[1, size, size, 1],strides=[1,
         stride, stride, 1], padding='SAME',name=str(idx)+'_pool')
```

5.5 Tools

5.5.1 TensorFlow

$TensorFlow^{TM}$ [20] is an open source software library for data-flow programming. It was originally developed by researchers and engineers working on the *Google Brain Team*, a deep learning artificial intelligence group.
It is used for a wide range of activities, such as speech recognition, image classification and object detection.
The name TensorFlow derives from the usage of multidimensional data arrays, named *tensors*, which are easily to handle in complex systems like Neural Networks. Its flexible architecture allows to distribute computations to one or more CPUs or GPUs in a desktop, server, or mobile device with a single API.

5.5.2 LabelImg

LabelImg [21] is a graphical image annotation tool written in Python language. It helps labelling images in a fast and precise way, translating the regions of interest (ROI) in *.xml* files.

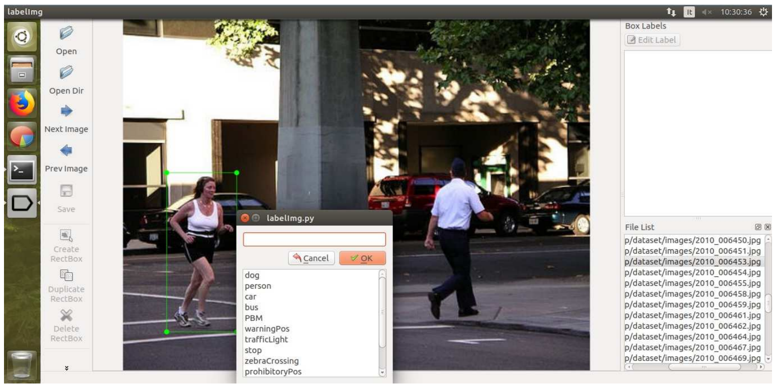

labelImg
10:30:36
Open
Open Dir
Next Image
Prev Image
Save
Create RectBox
Duplicate RectBox
Delete RectBox
labelImg.py
Cancel
OK
dog
person
car
bus
PBM
warningPos
trafficLight
stop
zebraCrossing
prohibitoryPos
Box Labels
Edit Label
File List
p/dataset/images/2010_006450.jpg
p/dataset/images/2010_006451.jpg
p/dataset/images/2010_006453.jpg
p/dataset/images/2010_006454.jpg
p/dataset/images/2010_006455.jpg
p/dataset/images/2010_006458.jpg
p/dataset/images/2010_006459.jpg
p/dataset/images/2010_006461.jpg
p/dataset/images/2010_006462.jpg
p/dataset/images/2010_006464.jpg
p/dataset/images/2010_006467.jpg
p/dataset/images/2010_006469.jpg

Figure 5.9: labelImg usage example

The formatting style of the output files is the same as PASCAL VOC annotation ones, which will be presented below.

```
<annotation>
     <folder>images</folder>
     <filename>2010_006453.jpg</filename>
     <path>/home/sti/Desktop/dataset/images/2010_006453.jpg</path>
     <source>
          <database>Unknown</database>
     </source>
     <size>
          <width>500</width>
          <height>375</height>
          <depth>3</depth>
     </size>
     <segmented>0</segmented>
     <object>
          <name>person</name>
          <pose>Unspecified</pose>
          <truncated>0</truncated>
          <occluded>0</occluded>
          <difficult>0</difficult>
          <bndbox>
               <xmin>46</xmin>
               <ymin>163</ymin>
               <xmax>121</xmax>
               <ymax>333</ymax>
          </bndbox>
     </object>
</annotation>
```

5.6 Training

The computational resources for the training phase are provided by HPC@POLITO[1], which is a project of Academic Computing within the Department of Control and Computer Engineering at the Politecnico di Torino.

5.7 Results

As already said in the section 5.3, the detection stage is composed of 379 images. Each image contains at least one person, which could be a pedestrian or not: for

[1]HPC@POLITO Website: http://hpc.polito.it

example images with persons in cars are also included in order to check if the network gives the right output.

Depending on the results of the network in this stage, the detected images can be divided into several groups:

1. *right response*: 109 images

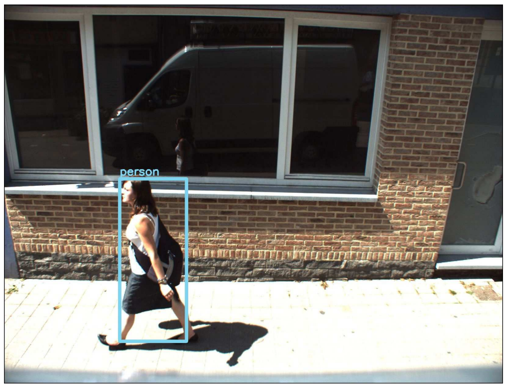


Figure 5.10: Example of image with right response

2. *partially right response*: 125 images;

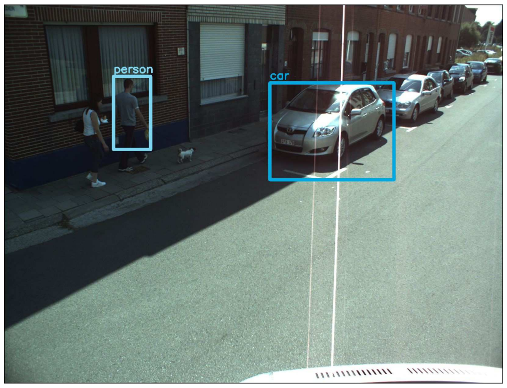


Figure 5.11: Example of image with partially right response

3. *wrong response* ***with*** *the possibility of improvement*: 16 images

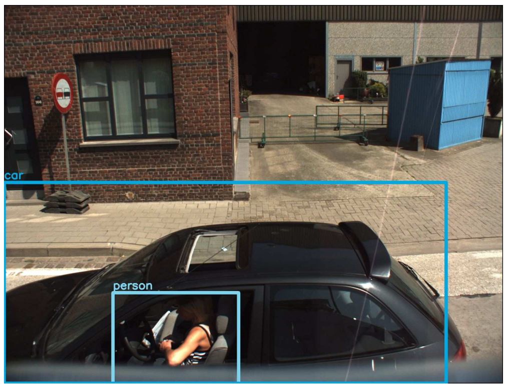


Figure 5.12: Example of image with possible improvements

4. *wrong response* ***without*** *the possibility of improvement*: 78 images

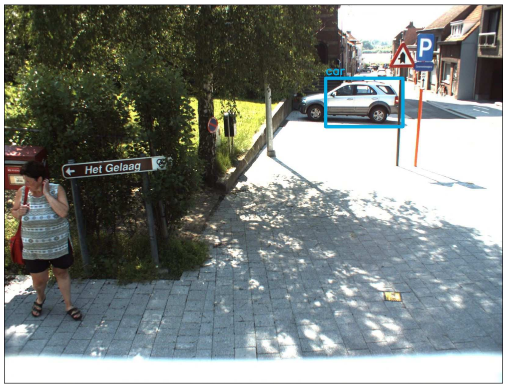


Figure 5.13: Example of image without possible improvements

5. *without bounding boxes*: 51 images

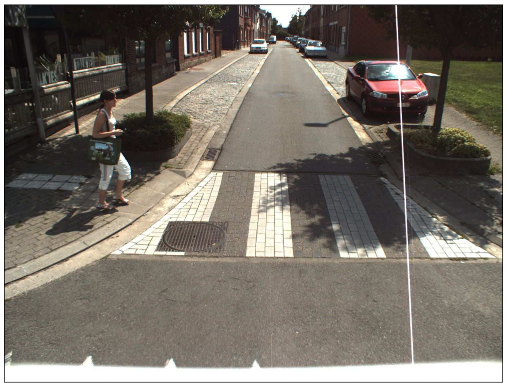


Figure 5.14: Example of image without bounding boxes

The partially right response category is different from the wrong ones, because it points out that some pedestrians have been recognized but, unfortunately, nothing can be done for the others in the same image. This is due to the absence of bounding boxes around them.

After finding the possibly correctable images, a script file is created in order to launch a simulation. It is the form:

```
python3 ./correctionModule.py --imgName arg1 --wrongBox
   tl_x,tl_y,br_x,br_y --helperLabel arg3 --helperProb arg4
```

where:

- **arg1** is the name of the image to be modified;
- tl_x and tl_y are the coordinates at the top-left of the bounding box to be changed;
- br_x and br_y are the coordinates at the bottom-right of the bounding box to be changed;
- **arg3** is the label of the bounding box in the same image, which indicates the helper;
- **arg4** is the confidence of the helper.

There are, obviously, two possibilities:

1. the image contains erroneously a pedestrian and the label needs to be changed into another one;
2. the network detects something which is not a pedestrian, but the label need to be changed into it.

Since the network has been modified in order to generate all the guesses of each bounding box, the conditional probability formula (equation 5.3) is applied to all of them. Then, the maximum value among these new probabilities is obtained, and, checking if it is above the threshold chosen in the detection stage, it is used to create a new image with the right labels and its *.txt* file with the annotations. This could lead to reject a good label with a low confidence.

Using this detection dataset, the correctable images are 16, which contain one of more label to be changed (the total number of wrong labels is 22). The correction script file is the following:

```
python3 ../correctionModule.py --imgName image.001756.jpg --wrongBox
    355,939,765,1235 --helperLabel car --helperProb 0.2
python3 ../correctionModule.py --imgName image.001964.c00.jpg
    --wrongBox 1159,311,1290,416 --helperLabel car --helperProb 0.54
python3 ../correctionModule.py --imgName image.002376.c00.jpg
    --wrongBox 899,193,955,288 --helperLabel PBM --helperProb 0.36
python3 ../correctionModule.py --imgName image.004877.c01.jpg
    --wrongBox 615,292,662,363 --helperLabel car --helperProb 0.53
python3 ../correctionModule.py --imgName image.008750.c02.jpg
    --wrongBox 992,264,1154,520 --helperLabel person --helperProb 0.47
python3 ../correctionModule.py --imgName image.010539.c02.jpg
    --wrongBox 1136,250,1231,421 --helperLabel car --helperProb 0.78
python3 ../correctionModule.py --imgName image.010539.c02.jpg
    --wrongBox 1109,241,1187,430 --helperLabel car --helperProb 0.78
python3 ../correctionModule.py --imgName image.011705.c02.jpg
    --wrongBox 883,103,946,201 --helperLabel car --helperProb 0.64
python3 ../correctionModule.py --imgName image.011706.c02.jpg
    --wrongBox 1246,108,1336,197 --helperLabel car --helperProb 0.65
python3 ../correctionModule.py --imgName image.012831.c05.jpg
    --wrongBox 209,451,333,553 --helperLabel car --helperProb 0.81
python3 ../correctionModule.py --imgName image.016032.c05.jpg
    --wrongBox 108,397,303,664 --helperLabel car --helperProb 0.74
python3 ../correctionModule.py --imgName image.016032.c05.jpg
    --wrongBox 208,475,258,595 --helperLabel car --helperProb 0.74
python3 ../correctionModule.py --imgName image.016032.c05.jpg
    --wrongBox 176,465,259,599 --helperLabel car --helperProb 0.74
python3 ../correctionModule.py --imgName image_00000265_0.jpg
    --wrongBox 380,236,390,261 --helperLabel person --helperProb 0.79
python3 ../correctionModule.py --imgName image_00000265_0.jpg
    --wrongBox 365,232,386,265 --helperLabel person --helperProb 0.79
python3 ../correctionModule.py --imgName image_00000265_0.jpg
    --wrongBox 420,236,445,269 --helperLabel person --helperProb 0.79
python3 ../correctionModule.py --imgName image_00001077_0.jpg
    --wrongBox 308,115,425,378 --helperLabel person --helperProb 0.63
python3 ../correctionModule.py --imgName image_00001077_1.jpg
    --wrongBox 305,70,444,476 --helperLabel person --helperProb 0.43
python3 ../correctionModule.py --imgName image_00001450_1.jpg
    --wrongBox 12,161,144,382 --helperLabel person --helperProb 0.70
python3 ../correctionModule.py --imgName image_00001527_0.jpg
    --wrongBox 198,196,254,274 --helperLabel person --helperProb 0.60
python3 ../correctionModule.py --imgName image_00001527_0.jpg
    --wrongBox 175,178,278,300 --helperLabel person --helperProb 0.60
```

```
python3 ../correctionModule.py --imgName image_00005693_0.jpg
    --wrongBox 489,182,506,221 --helperLabel car --helperProb 0.21
```

The results of this stage are:

- **9 right** labels;

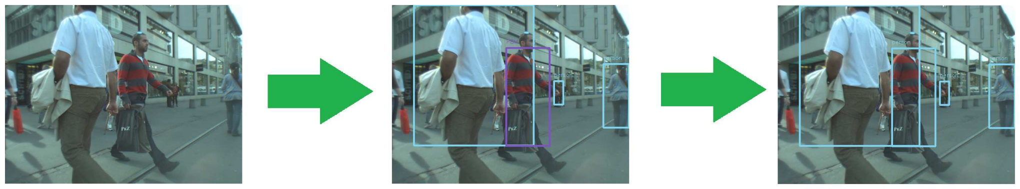


Figure 5.15: Example of image through each stage of the flowchart: the result is a right label

- **3 positively disappeared** labels (a person inside a car is not considered as pedestrian, so he/she is deleted);

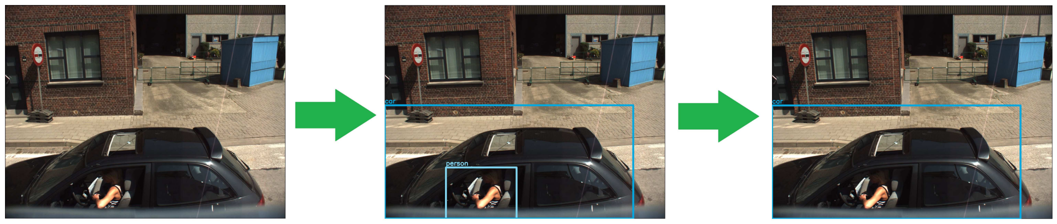


Figure 5.16: Example of image through each stage of the flowchart: the result is a positively disappeared label

- **5 negatively disappeared** labels (the confidence of a pedestrian is below the threshold, so he/she is deleted);

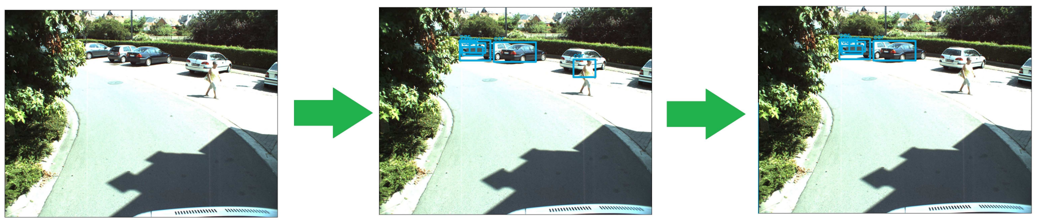


Figure 5.17: Example of image through each stage of the flowchart: the result is a negatively disappeared label

- **5 still-wrong** labels (the applied formula does not helped to reach the desired result, but the same label with a lower confidence).

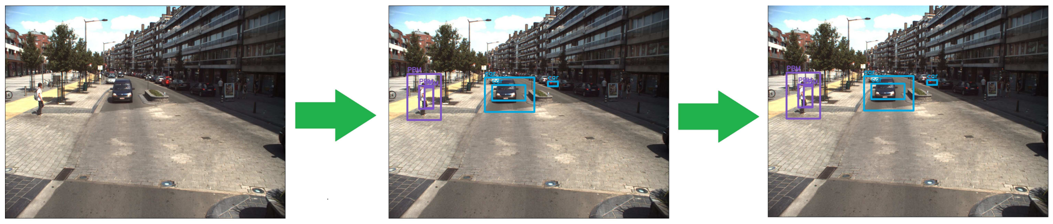
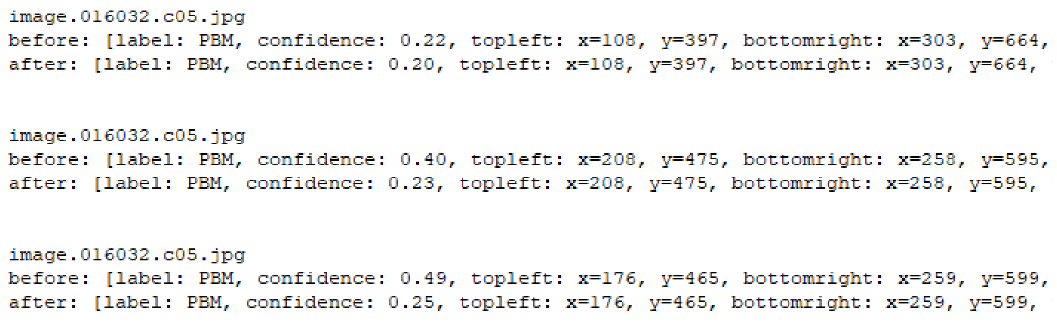


Figure 5.18: Example of image through each stage of the flowchart: the result is a still-wrong label



```
image.016032.c05.jpg
before: [label: PBM, confidence: 0.22, topleft: x=108, y=397, bottomright: x=303, y=664,
after: [label: PBM, confidence: 0.20, topleft: x=108, y=397, bottomright: x=303, y=664,


image.016032.c05.jpg
before: [label: PBM, confidence: 0.40, topleft: x=208, y=475, bottomright: x=258, y=595,
after: [label: PBM, confidence: 0.23, topleft: x=208, y=475, bottomright: x=258, y=595,


image.016032.c05.jpg
before: [label: PBM, confidence: 0.49, topleft: x=176, y=465, bottomright: x=259, y=599,
after: [label: PBM, confidence: 0.25, topleft: x=176, y=465, bottomright: x=259, y=599,
```

Figure 5.19: Still-wrong labels example

Results in terms of error rate are summarized in tables 5.4 and 5.5.

Before/after correction	**# wrong images**	**# total images**	**Error rate**
before	145	379	38.26%
after	137	379	36.15%

Table 5.4: Results per image

Before/after correction	**# wrong labels**	**# total labels**	**Error rate**
before	480	1421	33.78%
after	468	1421	32.93%

Table 5.5: Results per label

Chapter 6

Conclusion and future works

Clearly, the set of images for the detection part is too small with respect to the popular ones to constitute solid and reliable results. For this reason, it can be said that the results produced in this thesis work are not comparable with the ones from the well-known architectures. Anyway, they can be considered a good start point.

In the following, the *top-5 error rates* of the most popular Convolutional Neural Networks (mentioned in chapter 3) are presented in table 6.1.

AlexNet	**VGG-16**	**ResNet**	**GoogLeNet**	**YOLO**
16.4%	7.4%	5.3%	6.7%	12%

Table 6.1: Top-5 error rates of the most popular CNNs

The goals of this thesis were to implement a correction module for a Convolutional Neural Network designed for the pedestrian detection and to synthesize the hardware.
Even if the first goal was achieved, obviously, further steps can be considered to improve the robustness of the Neural Network, such as:

- more training epochs;
- more and differentiated training images;
- different thresholds;
- different learning rate;
- different helping classes.

Regarding the hardware goal, it was not possible, due to lack of time, to achieve it. However, scientific articles on this subject have been produced, so this can be a valid starting point for a possible task for the future. For example, in the article named *Hardware Implementation and Optimization of Tiny-YOLO Network* [22], the authors explain that, since a Convolutional Neural Network needs intensive computations and lots of memory, designing its hardware is very complex. Both of these problems can be overcome as follow:

1. in order to reduce the memory, they employ data reusing and data sharing techniques;

2. in order to reduce computing time, some processing elements work simultaneously.

In this way, parallel processing elements share the same input data.

Appendix A

Code

This appendix includes the command lines and some of the principal code files involved for the purpose of this thesis. Some of these files are located at
https://github.com/thtrieu/darkflow

CommandLines

```
python3 ./flow.py --model ./cfg/tiny-yolo-14c.cfg --load
    ./bin/tiny-yolo-voc.weights --train --annotation
    ./dataset/annotations/ --dataset ./dataset/images/ --batch 5 --epoch
    30 --save 10000 --savepb

python3 ./flow.py --pbLoad built_graph/tiny-yolo-14c.pb --metaLoad
    built_graph/tiny-yolo-14c.meta --imgdir ./images/detImg/ --threshold
    0.2

python3 ./alphaGen.py

python3 ../correctionModule.py --imgName arg1 --wrongBox
    tl_x,tl_y,br_x,br_y --helperLabel arg3 --helperProb arg4
```

tiny-yolo-14c.cfg

```
[net]
batch=64
subdivisions=8
width=416
height=416
channels=3
```

```
momentum=0.9
decay=0.0005
angle=0
saturation = 1.5
exposure = 1.5
hue=.1

learning_rate=0.001
max_batches = 40100
policy=steps
steps=-1,100,20000,30000
scales=.1,10,.1,.1

[convolutional]
batch_normalize=1
filters=16
size=3
stride=1
pad=1
activation=leaky

[maxpool]
size=2
stride=2

[convolutional]
batch_normalize=1
filters=32
size=3
stride=1
pad=1
activation=leaky

[maxpool]
size=2
stride=2

[convolutional]
batch_normalize=1
filters=64
size=3
stride=1
pad=1
```

```
activation=leaky

[maxpool]
size=2
stride=2

[convolutional]
batch_normalize=1
filters=128
size=3
stride=1
pad=1
activation=leaky

[maxpool]
size=2
stride=2

[convolutional]
batch_normalize=1
filters=256
size=3
stride=1
pad=1
activation=leaky

[maxpool]
size=2
stride=2

[convolutional]
batch_normalize=1
filters=512
size=3
stride=1
pad=1
activation=leaky

[maxpool]
size=2
stride=1

[convolutional]
```

```
batch_normalize=1
filters=1024
size=3
stride=1
pad=1
activation=leaky

[convolutional]
batch_normalize=1
size=3
stride=1
pad=1
filters=1024
activation=leaky

[convolutional]
size=1
stride=1
pad=1
filters=95
activation=linear

[region]
anchors = 1.08,1.19,3.42,4.41,6.63,11.38,9.42,5.11,16.62,10.52
bias_match=1
classes=14
coords=4
num=5
softmax=1
jitter=.2
rescore=1

object_scale=5
noobject_scale=1
class_scale=1
coord_scale=1

absolute=1
thresh = .5
random=1
```

darkflow/net/prova.py

```
import numpy as np
from numpy import exp
from ..utils.box import BoundBox
from .provaNMS import NMS

#findMatch
def findMatch(nms_prob, cl_prob):
    for el in nms_prob:
        #print("el: {}".format(el))
        for elRow in range(cl_prob.shape[0]):
            #print("elRow: {}".format(elRow))
            for elCol in range(cl_prob.shape[1]):
                #print("elCol: {}".format(elCol))
                if(el == cl_prob[elRow, elCol]):
                    return cl_prob[elRow, :]
    return nms_prob


#expit
def expit_c(x):
    y= 1/(1+exp(-x))
    return y

#MAX
def max_c(a, b):
    if(a>b):
        return a
    return b

#BOX CONSTRUCTOR
def box_constructor(meta, net_out_in):

    threshold = meta['thresh']
    arr_max=0
    summation=0

    anchors = np.asarray(meta['anchors'])
    #print("anchors {}".format(anchors))

    H, W, _ = meta['out_size']
    C = meta['classes']
    B = meta['num']

    #print(meta)

    net_out = net_out_in.reshape([H, W, B, net_out_in.shape[2]//B])
    Classes = net_out[:, :, :, 5:]
    Bbox_pred = net_out[:, :, :, :5]
    probs = np.zeros((H, W, B, C), dtype=np.float32)

    clProb = np.zeros(C, dtype=np.float32)
    firstClProb = True

    for row in range(H):
        for col in range(W):
            for box_loop in range(B):
                arr_max=0
                summation=0;

                Bbox_pred[row, col, box_loop, 4] = expit_c(Bbox_pred[row,
                    col, box_loop, 4])
                Bbox_pred[row, col, box_loop, 0] = (col +
                    expit_c(Bbox_pred[row, col, box_loop, 0])) / W
                Bbox_pred[row, col, box_loop, 1] = (row +
                    expit_c(Bbox_pred[row, col, box_loop, 1])) / H
                Bbox_pred[row, col, box_loop, 2] = exp(Bbox_pred[row, col,
                    box_loop, 2]) * anchors[2 * box_loop + 0] / W
                Bbox_pred[row, col, box_loop, 3] = exp(Bbox_pred[row, col,
                    box_loop, 3]) * anchors[2 * box_loop + 1] / H
                #SOFTMAX BLOCK, no more pointer juggling

                for class_loop in range(C):
                    arr_max=max_c(arr_max,Classes[row,col,box_loop,class_loop])

                for class_loop in range(C):
                    Classes[row,col,box_loop,class_loop] =
                        exp(Classes[row,col,box_loop,class_loop]-arr_max)
                    summation+=Classes[row,col,box_loop,class_loop]

###############
                prova = np.zeros(C, dtype=np.float32)
                toProb = False
                for class_loop in range(C):
                    tempc = Classes[row, col, box_loop, class_loop] *
                        Bbox_pred[row, col, box_loop, 4]/summation

                    prova[class_loop] = tempc

                    if(tempc > threshold):
                        toProb = True
                        probs[row, col, box_loop, class_loop] = tempc

                if(toProb):
                    if not (firstClProb):
                        clProb = np.vstack([clProb, prova])
                    else:
                        clProb = prova
                        firstClProb = False

    #print("clProb probs: {}\n".format(clProb))

###############



    #NMS
    probsContArray = np.ascontiguousarray(probs).reshape(H*W*B,C)
    boxPredContArray = np.ascontiguousarray(Bbox_pred).reshape(H*B*W,5)

    NMSboxes = NMS(probsContArray, boxPredContArray)



########
    #print("nmsSHAPE before {}".format(np.shape(NMSboxes)))
    toDel = -1
    for b in NMSboxes:
        toDel += 1
        #print("nms PROBS before: {}".format(b.probs))

        if (np.array_equal(b.probs, np.zeros(C, dtype=np.float32))):
            NMSboxes = np.delete(NMSboxes, toDel, axis=0)
            toDel -= 1
            #print("deleted row {}\n".format(toDel))
        else:
            b.probs = findMatch(b.probs, clProb)
            #print("nms PROBS after: {}\n".format(b.probs))
    #print("nmsSHAPE after {}\n".format(np.shape(NMSboxes)))
########

    return NMSboxes
```

darkflow/net/provaNMS.py

```
import numpy as np
from numpy import exp
from ..utils.box import BoundBox




#OVERLAP
def overlap_c(x1, w1 , x2 , w2):

    l1 = x1 - w1 /2.
    l2 = x2 - w2 /2.
    left = max(l1,l2)
    r1 = x1 + w1 /2.
    r2 = x2 + w2 /2.
    right = min(r1, r2)
    return right - left;


#BOX INTERSECTION
def box_intersection_c(ax, ay, aw, ah, bx, by, bw, bh):

    w = overlap_c(ax, aw, bx, bw)
    h = overlap_c(ay, ah, by, bh)
    if w < 0 or h < 0: return 0
    area = w * h
    return area


#BOX UNION
def box_union_c(ax, ay, aw, ah, bx, by, bw, bh):

    i = box_intersection_c(ax, ay, aw, ah, bx, by, bw, bh)
    u = aw * ah + bw * bh -i
    return u
```

```
#BOX IOU
def box_iou_c(ax, ay, aw, ah, bx, by, bw, bh):
    return box_intersection_c(ax, ay, aw, ah, bx, by, bw, bh) /
        box_union_c(ax, ay, aw, ah, bx, by, bw, bh);




#NMS
def NMS(final_probs, final_bbox):
    boxes = []
    indices = set()

    pred_length = final_bbox.shape[0]
    class_length = final_probs.shape[1]
    for class_loop in range(class_length):
        for index in range(pred_length):
            if final_probs[index,class_loop] == 0: continue
            for index2 in range(index+1,pred_length):
                if final_probs[index2,class_loop] == 0: continue
                if index==index2 : continue
                if box_iou_c(final_bbox[index,0], final_bbox[index,1],
                    final_bbox[index,2], final_bbox[index,3],
                    final_bbox[index2,0], final_bbox[index2,1],
                    final_bbox[index2,2], final_bbox[index2,3]) >= 0.4:
                    if final_probs[index2,class_loop] > final_probs[index,
                        class_loop] :
                        final_probs[index, class_loop] =0
                        break
                    final_probs[index2,class_loop]=0

            if index not in indices:
                bb=BoundBox(class_length)
                bb.x = final_bbox[index, 0]
                bb.y = final_bbox[index, 1]
                bb.w = final_bbox[index, 2]
                bb.h = final_bbox[index, 3]
                bb.c = final_bbox[index, 4]
                bb.probs = np.asarray(final_probs[index,:])
                boxes.append(bb)
                indices.add(index)
```

```
   return boxes
```

darkflow/net/yolov2/predict.py

```
import numpy as np
import math
import cv2
import os
#import json
#from scipy.special import expit
#from utils.box import BoundBox, box_iou, prob_compare
#from utils.box import prob_compare2, box_intersection
from ...utils.box import BoundBox
#from ...cython_utils.cy_yolo2_findboxes import box_constructor
from ..prova import box_constructor

def expit(x):
   return 1. / (1. + np.exp(-x))

def _softmax(x):
    e_x = np.exp(x - np.max(x))
    out = e_x / e_x.sum()
    return out

def findboxes(self, net_out):

   # meta
   meta = self.meta
   boxes = list()
   boxes=box_constructor(meta,net_out)
   return boxes

def postprocess(self, net_out, im, save = True):
   """
   Takes net output, draw net_out, save to disk
   """
   boxes = self.findboxes(net_out)

   #print("POSTPROCESS YOLO2")

   # meta
   meta = self.meta
   threshold = meta['thresh']
```

```
colors = meta['colors']
labels = meta['labels']
if type(im) is not np.ndarray:
   imgcv = cv2.imread(im)
else: imgcv = im
h, w, _ = imgcv.shape


outfolder = os.path.join(self.FLAGS.imgdir, 'out')
img_name = os.path.join(outfolder, os.path.basename(im))
textFile = os.path.splitext(img_name)[0] + ".txt"
with open(textFile, 'w') as f:
   for b in boxes:
      print("im {} PROBS: {}".format(im, b.probs))
      boxResults = self.process_box(b, h, w, threshold)
      if boxResults is None:
         continue
      left, right, top, bot, mess, max_indx, confidence = boxResults
      thick = int((h + w) // 300)

      totProbs = ''
      for p in range(len(b.probs)):
         totProbs = totProbs + ' ' + str(b.probs[p])
      print('totProbs {}'.format(totProbs))

      resultsForTXT = '[label: %s, confidence: %.2f, topleft: x=%s,
          y=%s, bottomright: x=%s, y=%s, totProbs: %s]' % (mess,
          confidence, left, top, right, bot, totProbs)
      f.write(resultsForTXT + "\n")

      cv2.rectangle(imgcv, (left, top), (right, bot), colors[max_indx],
          thick)
      cv2.putText(imgcv, mess, (left, top - 12), 0, 1e-3 * h,
          colors[max_indx],thick//3)


#if not save: return imgcv
cv2.imwrite(img_name, imgcv)
```

alphaGen.py

```
import sys
import random
```

```
def probFunc(lab, start, stop):
   probs = lab + ":\n"

   for i in range(len(labels)):
      if (labels[i] == "person"):
         rdm = random.uniform(start, stop)
      else:
         rdm = random.uniform(0.84, 0.91)

      probs = probs + "\t%s %f\n" % (labels[i], rdm)

   return probs

labFile = open("labels.txt", "r")
labels = []

for line in labFile:
   labels.append(line[:len(line)-1])

file = open("alphas.txt","w")

file.write(probFunc("dog", 0.7, 0.8) + "\n")
file.write(probFunc("person", 0.6, 0.7) + "\n")
file.write(probFunc("car", 0.85, 1.0) + "\n")
file.write(probFunc("bus", 0.7, 0.8) + "\n")
file.write(probFunc("PBM", 0.85, 1.0) + "\n")
file.write(probFunc("warningPos", 0.65, 0.8) + "\n")
file.write(probFunc("trafficLight", 0.65, 0.8) + "\n")
file.write(probFunc("stop", 0.65, 0.8) + "\n")
file.write(probFunc("zebraCrossing", 0.6, 0.7) + "\n")
file.write(probFunc("prohibitoryPos", 0.6, 0.75) + "\n")
file.write(probFunc("obligationPos", 0.62, 0.77) + "\n")
file.write(probFunc("pram", 0.6, 0.65) + "\n")
file.write(probFunc("warningNeg", 0.9, 1.0) + "\n")
file.write(probFunc("prohibitoryNeg", 0.95, 1.0) + "\n")

file.close()
```

correctionModule.py

```
import sys
import numpy as np
import cv2
from helper import correctionHelper


def extractAlpha(lab):
   tmp = []

   fl = open("alphas.txt","r")
   fileL = list(fl)

   for i in range(len(fileL)):
      if (lab+":") in fileL[i]:
         for k in range(len(labels)):
            sl = fileL[i+k+1].split(" ")
            tmp.append(float(sl[1]))

   #print('tmp = {}'.format(tmp))
   fl.close()

   return tmp


def finder(ms):    #find nClasses and colors in .meta file
   iAbsolute = ms.find('"absolute"')
   iAnchors = ms.find('"anchors"')
   iBiasMatch = ms.find('"bias_match"')
   iClasses = ms.find('"classes"')
   iClassScales = ms.find('"class_scale"')
   iColors = ms.find('"colors"')
   iCoords = ms.find('"coords"')
   iCoordScale = ms.find('"coord_scale"')
   iInpSize = ms.find('"inp_size"')
   iJitter = ms.find('"jitter"')
   iLabels = ms.find('"labels"')
   iModel = ms.find('"model"')
   iName = ms.find('"name"')
   iNet = ms.find('"net"')
   iNoObjectScale = ms.find('"noobject_scale"')
   iNum = ms.find('"num"')
   iObjectScale = ms.find('"object_scale"')
   iOutSize = ms.find('"out_size"')
   iRandom = ms.find('"random"')
   iRescore = ms.find('"rescore"')
```

```
    iSoftmax = ms.find('"softmax"')
    iThresh = ms.find('"thresh"')
    iType = ms.find('"type"')

    mList = [iAbsolute, iAnchors, iBiasMatch, iClasses, iClassScales,
        iColors, iCoords, iCoordScale, iInpSize, iJitter, iLabels, iModel,
        iName, iNet, iNoObjectScale, iNum, iObjectScale, iOutSize, iRandom,
        iRescore, iSoftmax, iThresh, iType]
    #print("mList {} \nmList size {}\n".format(mList, len(mList)))
    mList = sorted(mList)
    #print("sorted mList {}\n".format(mList))

    clStart = mList.index(iClasses)
    nClasses = int(ms[mList[clStart]+11:mList[clStart+1]-2])
    # print("nClasses {}".format(nClasses))

    coStart = mList.index(iColors)
    colors = ms[mList[coStart]+12:mList[coStart+1]-4]
    # print("colors {}".format(colors))

    return nClasses, colors


def changeProbs(probs, alpha, help):
    newProbs = []

    for j in range(len(labels)):
        new = ((1 - float(alpha[j])) * (float(probs[j]) + help))/help
        newProbs.append(float(format(new, '.7f')))

    #print("newProbs {}".format(newProbs))
    return newProbs


def extractLabels():
    labFile = open("labels.txt", "r")
    labels = []

    for line in labFile:
        labels.append(line[:len(line)-1])

    #print("LABELS {}".format(labels))
    return labels
```

```
def extractMeta(metaName):
   metaFile = open(metaName, "r")
   metaStr = metaFile.read()
   #print("metaStr {}".format(metaStr))

   nClasses, colors = finder(metaStr)
   print('classes {}'.format(nClasses))
   print('colors {}'.format(colors))

   for i in range(nClasses-1):
      c = colors.find('], [')
      colors = colors[:c] + ', ' + colors[c+4:]

   print('colors before {}\n'.format(colors))

   colors = colors.split(", ")

   temp = []
   for j in range(nClasses):
      temp2 = []
      temp2.append(str(int(float(colors[3*j]))))
      temp2.append(str(int(float(colors[3*j+1]))))
      temp2.append(str(int(float(colors[3*j+2]))))
      temp.append(temp2)

   colors = temp
   print('colors after {}\n'.format(colors))

#  threshold = meta['thresh']

   return colors



def extractObjs(line, tp):
   lo1 = line.find('label')
   lo2 = line.find('confidence')
   lo3 = line.find('topleft')
   lo4 = line.find('bottomright')
   lo5 = line.find('totProbs')
   tl = line[lo3+11:lo4-2]
   c_idx = tl.find(',')
   left = int(tl[:c_idx])
   top = int(tl[c_idx+4:len(tl)])

   br = line[lo4+15:lo5-2]
   c_idx = br.find(',')
   right = int(br[:c_idx])
   bot = int(br[c_idx+4:])

   if not tp:
      lab = line[lo1+7:lo2-2]
      conf = line[lo2+12:lo3-2]

      return (lab, conf, left, top, right, bot)

   else:
      tmp = line[lo5+11:len(line)-2]
      totProbs = [float(s) for s in tmp.split()]
      #print("TOT PROBS {}".format(totProbs))

      return (left, top, right, bot, totProbs)



# print ('arguments lenght: {}'.format(len(sys.argv)))
# print ('arguments: {}'.format(sys.argv))
ch = correctionHelper()
ch.parseArgs(sys.argv)

#print(ch.imgName)
#print(ch.coord)
#print(ch.helpLab)
#print(ch.helpProb)
#print(ch.probOld)
#print('')

new_imgName = 'new_' + ch.imgName
#print('new image name: {}'.format(new_imgName))

coord_split = ch.coord.split(",")
brIN = 'bottomright: x=%s, y=%s' % (coord_split[2], coord_split[3])
tlIN = 'topleft: x=%s, y=%s' % (coord_split[0], coord_split[1])
```

```
#print('')
#print("coord {}".format(coord_split))
#print(brIN)
#print(tlIN)

#colors =
extractMeta("./built_graph/tiny-yolo-14c.meta") # tiny-yolo-14c.meta
labels = extractLabels()


txt1 = ch.imgName[:len(ch.imgName)-4] + '.txt'  # .jpg extension is
    changed in .txt
txt2 = 'new_' + ch.imgName[:len(ch.imgName)-4] + '.txt'
#print('txt1 file: {}'.format(txt1))
#print('txt2 file: {}'.format(txt2))

file1 = open(txt1,"r")
file2 = open(txt2, "w")

if type(ch.imgName) is not np.ndarray:
   imgcv = cv2.imread(ch.imgName)
else:
   imgcv = ch.imgName
h, w, _ = imgcv.shape
thick = int((h+w)//300)

for line in file1:
   if tlIN in line:
      if brIN in line:
         left, top, right, bot, totProbs = extractObjs(line, True)
         alpha = extractAlpha(ch.helpLab)
         newTotProbs = changeProbs(totProbs, alpha, ch.helpProb)
         probNew = max(newTotProbs)
         lab = labels[newTotProbs.index(probNew)]

         newline = '[label: %s, confidence: %.2f, %s, %s, totProbs: %s]\n'
            % (lab, probNew, tlIN, brIN, newTotProbs)
         print(newline)

   else:
      newline = line
      lab, conf, left, top, right, bot = extractObjs(line, False)
```

```
  file2.write(newline)

  detIdx = labels.index(lab)
  #print("detIdx {}".format(detIdx))

  cv2.rectangle(imgcv, (left, top), (right, bot),
      (int(colors[detIdx][0]), int(colors[detIdx][1]),
      int(colors[detIdx][2])), thick)
  cv2.putText(imgcv, lab, (left, top - 12), 0, 1e-3 * h,
      (int(colors[detIdx][0]), int(colors[detIdx][1]),
      int(colors[detIdx][2])),thick//3)


cv2.imwrite(new_imgName, imgcv)


file1.close()
file2.close()
```

helper.py

```
class correctionHelper(dict):
  def __init__(self):
    __getattr__ = dict.get
    __setattr__ = dict.__setitem__
    __delattr__ = dict.__delitem__
    self._descriptions = {'help, --h': 'help message'}
    self._descriptions['imgName'] = 'path to the image to be corrected'
    self._descriptions['wrongBox'] = 'coordinates of the wrong box
       tl_x,tl_y,br_x,br_y BEWARE: no spaces between numbers!'
    self._descriptions['helperLabel'] = 'label of the helper'
    self._descriptions['helperProb'] = 'probability of the helper'


  def parseArgs(self, args):
    print('')
    i = 1

    if (len(args) != (2*len(self._descriptions)-1)):
      if args[i] == '--h' or args[i] == '--help':
            print('Example usage: python3 correctionModule.py --imgName
                arg1 --wrongBox tl_x,tl_y,br_x,br_y --helperLabel arg3
                --helperProb arg4')
            print('')
            print('Arguments:')
            spacing = max([len(i) for i in self._descriptions.keys()]) + 2
            for item in self._descriptions:
                currentSpacing = spacing - len(item)
                print(' --' + item + (' ' * currentSpacing) +
                    self._descriptions[item])
            print('')
            exit()
        else:
            print('Wrong input!! Missing argument/s, run python3
                correctionModule --help')
            print('')
            exit()

    while i < len(args):

        if args[i] == '--imgName':
            self.imgName = args[i+1]

        if args[i] == '--wrongBox':
            self.coord = args[i+1]

        if args[i] == '--helperLabel':
            self.helpLab = args[i+1]

        if args[i] == '--helperProb':
            self.helpProb = float(args[i+1])

        i += 1
```

Bibliography

[1] A. Ziebinski, R. Cupek, D. Grzechca, and L. Chruszczyk. Review of advanced driver assistance systems (ADAS). *AIP Conference Proceedings*, 1906(1), 2017.

[2] D. Gerónimo, A.M. López, A.D. Sappa, and T. Graf. Survey of pedestrian detection for advanced driver assistance systems. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 32(7), 2010.

[3] J. Redmon, S. Divvala, R. Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. *arXiv:1506.02640*, 2015.

[4] C. Cortes and V. Vapnik. Support-Vector Networks. *Machine Learning*, 20(3), 1995.

[5] N. Dalal. *Finding People in Images and Videos.* PhD dissertation, Institut National Polytechnique de Grenoble, 2006.

[6] C. Maureen. Neural networks primer, part I. *AI Expert*, 2(12), 1987.

[7] N. Gupta. Artificial neural network. *Network and Complex Systems*, 3(1), 2013.

[8] V. Sze, Y. Chen, T. Yang, and J. Emer. Efficient processing of deep neural networks: A tutorial and survey. *Proceedings of the IEEE*, 105(12), 2017.

[9] Y. Le Cun, L.D. Jackel, B. Boser, J.S. Denker, H.P. Graf, I. Guyon, D. Henderson, R.E. Howard, and W. Hubbard. Handwritten digit recognition: Applications of neural network chips and automatic learning. *IEEE Communications Magazine*, 27(11), 1989.

[10] A. Krizhevsky, I. Sutskever, and G.E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, 2012.

[11] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *ICLR*, 2015.

[12] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016.

[13] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. CVPR, 2015.

[14] P. Dollár, C. Wojek, B. Schiele, and P. Perona. Pedestrian detection: An

evaluation of the state of art. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 34(4), 2012.

[15] G. Griffin, A. Holub, and P. Perona. Caltech-256 object category dataset. Tech. rep., California Inst. of Technology, 2007.

[16] M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman. The pascal visual object classes (VOC) challenge. *International Journal of Computer Vision*, 88(2), 2010.

[17] R. Timofte, K. Zimmermann, and L. Van Gool. Multi-view traffic sign detection, recognition, and 3D localisation. *Machine Vision and Applications*, 25(3), 2014.

[18] A. Ess, B. Leibe, and L. Van Gool. Depth and appearance for mobile scene analysis. In *IEEE 11th International Conference on Computer Vision*, 2007.

[19] A. Ess, B. Leibe, K. Schindler, and L. Van Gool. A mobile vision system for robust multi-person tracking. In *IEEE Conference on Computer Vision and Pattern Recognition*, 2008.

[20] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015.

[21] Label Annotation VOC Pascal website. *https://github.com/manhcuogntin4/Label-Annotation-VOC-Pascal*, 2015.

[22] J. Ma, L. Chen, and Z. Gao. Hardware implementation and optimization of tiny-yolo network. In *Digital TV and Wireless Multimedia Communication*. Springer Singapore, 2018.