Politecnico di Torino

Master of Science Degree in MECHATRONIC ENGINEERING

Master Thesis

Machine Learning Algorithms for Service Robotics Applications in Precision Agriculture

Supervisor:
prof. Marcello Chiaberge

Candidate:
Angelo Tartaglia
S242338

29 October 2018

Abstract

World population is growing faster than expected. Every year Earth resources are consumed faster, as proven by the early fall of the *Earth Overshoot Day* that claims the end of year resources. One of the simplest solutions to this problem would be investing in technologies and innovations towards a smart extraction of what population needs. In the last few years a lot of companies introduced automation and robots to improve production in terms of time and obviously cost. Agricultural world is not distant from this evolution; in fact many of the works attempted are now helped by a set of machines that simplifies human work. Hitherto the application in precision agriculture has been always under the human control; there are a lot of applications that see the participation of a worker and in parallel a machine used for a lot of tasks. In the future may an entire field will be managed by a group of automated robots that work together. Those machines will require a lot of specific features likes autonomous navigation, mapping, visual object recognition and many others. This thesis is part of that project and it regards the detection and classification of fruits for future application of auto-harvesting and health control. In a more specific way apples will be considered in this thesis work for fruit application and methods and algorithms as Y.O.L.O. and Mask R-CNN to do the processing of images will be described.These techniques permits the detection of apples in post processing and in real-time with accuracies that range over 32% to 78%. The final result can be used in future applications for spatial localization of fruits and for the detection of possible diseases. It should be emphasised that, even if the thesis shows the results of the object class apple, the algorithms can be applied in a wide range of objects with the only requirement of a different training images dataset.

Acknowledgements

Difficilmente sarà sufficiente ringraziare tutte le persone che mi sono state vicine con una sola citazione all'interno di un testo. Attimi di felicità e supporto che mi sono stati dati in questi due anni, che sono trascorsi più veloci del dovuto, necessiterebbero di un maggior premio. Spero di poter ripagare tutti quanti lautamente e far loro capire quanto per me siano importanti nei prossimi duri anni che mi si prospettano.
Come fortunatamente ovvio sia, i miei genitori meritano il primo pensiero: mamma Eleonora che si fa sempre in mille per me e papà Vito che è sempre pronto ad indicarmi la strada giusta in mezzo alla tempesta. Pluffa, la mia sorellina ormai grande, che mi riesce a proteggere anche quando io cerco di proteggere lei. Le mie nonne, Elena, la *Vecchia*, che se potesse mi donerebbe anche il suo cuore e Caterina che anche se da lontano mi pensa sempre. Non posso elencare tutto il resto della famiglia ma loro sanno quanto sono loro grato, o almeno spero di riuscire a dimostrarglielo.
Ringraziamento speciale poi per Nora, la piccina con le spalle larghe che mi fa respirare in un mondo senz'aria, e alla sua famiglia, Graziella, Melo e Gabri, sempre presenti per me ed amorevoli.
Al prof. Chiaberge ed a tutto il PIC4SeR, Vittorio, Luca, Gianluca, Jurgen, Lorenzo ed Aurora con i quali ho trascorso i mesi di tesi e probabilmente anche qualcos'altro in futuro: grazie per le chiacchiere e la caffeina.
Doveroso anche citare Daniele e Luca, compagni universitari, che hanno sofferto assieme a me e gioito nelle poche serate libere da pensieri ma piene di FIFA.
Impossibile non ringraziare anche i miei fratelli del collegio Einaudi: anche se ormai il mondo è cambiato noi ci saremo comunque l'uno per gli altri.
Sperando di non aver dimenticato nessuno, chiedo venia in tal caso, vorrei concludere ricordando a chi avrà l'onere della lettura di queste mie parole che:

"
La risposta alla domanda fondamentale...
sulla vita... l'universo e tutto quanto...
è...
42
"

Contents

List of Figures

Chapter 1

Introduction

World population is growing faster than expected and in parallel the request of food and beverage is increasing. During the last 15-20 years the major exponents of technology asked themselves on how to reduce the cost and to increase the production of all the raw materials using new advanced methods of analysis. One of the sectors that also nowadays is still in development is the agricultural field. Researchers devoted to the automation of the agricultural field try to use newest technology to make the work simpler and faster. During these years the methods have changed becoming more precise and making possible the usage of these techniques not only in theory but also in practice. It is interesting to see how in these years the methods have changed. In particular this elaborate is centred on the detection of fruits and specifically on apples.

1.1 State of Art

Precision agriculture is a branch of scientific research that has been continuously evolving from late '90s. The first real approach can be found in the following works listed in chronological order (only the important aspects are mentioned, for further information see [1, 2, 3, 4, 5]).

"A New Method for Fruit Recognition System"

This was a work dated back to 2009. It is based on the *k-Nearest Neighbours* (KNN) algorithm. The system uses a colour-based, shape-based and size-based feature analysis to classify the fruit in the images. The authors assure a value for the accuracy up to 90%. This particular system works as follows:

1. Ask to the user the generation of a shape that encloses the fruit to recognise;
2. Calculate the average RGB of the fruit;
3. Calculate the fruit shape roundness: $shape = 4\pi \left(\frac{Area}{Perimeter^2}\right)$;
4. Compute the KNN algorithm and generate the result.

This model is based on a particular database that includes details of colour (minimum and maximum RGB values) and shape roundness for each type of fruit in the trained system. In fact 50 images were processed and catalogued according to these features. Then, when an image is given to this model, the KNN algorithm measures the distance between feature values detected and the stored ones choosing the smallest distance that gives the highest probability class of belonging. This was an old model that requires also a user intervention for the classification but needs few data for training and very low hardware performance.

"Thai Fruit Recognition System (TFRS)"

This was a work dated back to 2009. It is similar to the previous explained one: it uses edge shape, size and colour features to determine which kind of thai fruit is detected among the 39 classes in the training dataset. The only differentiation from the previous model is the computation with the *chain code method* of the shape of the object: it is not asked to the user the generation of the shape of fruit, but the image has to be very clear. The dataset used is only formed by 128 images subdivided into 39 classes (an average of 3.28 images per class) but the authors assure an accuracy about 87% and an average processing time per image 55sec.

"Fruit Recognition using Color and Texture Feature"

This was a work dated back to 2010. The authors' aim was the development of a fruit recognition system based on intensity, colour, shape and texture feature of the image. The idea at the base is always the same: using a dataset for the training of the model and then compare it to the processed picture. The dataset used is composed by 15 classes of fruits and 2633 images (175.53 images per class). The different approach used is about the usage of *HSV* format for the processed images: a module transforms the picture from RGB into HSV and then the chrominance channels 'H' and 'S' give colour features while luminance channel 'V' gives texture features. The test image is then compared with the stored values and a classification is done using the *Minimum Distance Criterion*. This is one of the first algorithms that during the classification makes use of the RoIs (§4).

"Automatic Fruit Recognition and Counting from Multiple Images"

This was a work dated back to 2014. It represents one of the first similar applications of what has been done on this thesis work. The researchers' aim was the development of a system able to count the pepper on a plant starting from multiple images. The overall system is formed by an initial colour transformation module (from RGB to a custom value useful for peppers) followed by a *Naïve Bayes* classifier; then a *Bag-of-Words* model with a SVM classifier processes the test image and gives the result. It should be underlined that it was used a *Support Vector Machine* classifier, one of the latest; also it was created an algorithm able to identify the same pepper on different images taken in succession. This allows a count of peppers with an accuracy of 74.2%. Therefore in this application it is possible to see a dataset composed by 28,000 images and the *sliding window* technique to detect a pepper.

"Automatic Fruit Recognition from Natural Images using Color and Texture Features"

This was a work dated back to 2017. The authors' aim was the classification of 8 different fruits using a training database of 240 images. The methodology introduces a new *GrabCut* algorithm to split the fruits from the background. This procedure makes easier the creation of the RoIs that are processed by the feature extractor (colours and texture) and finally given to the SVM classifier. The results have an accuracy about 83%, making this method a good classifier for fruits.

1.2 Motivations

As it can be clearly seen in previous works described in SoA section, all the solutions found until this thesis are referred to a classification of the fruit from images. The aim of my work is instead the *detection* of fruit that implies not only the comprehension of the presence of an

apple in pictures but also the identification of its location. This is a big deal considering the power needed by the hardware and the huge dataset needed for the training of a network. The final platform can be applied in several manners, from auto-harvesting to the monitoring of health conditions of the plant and fruits. The purpose of this thesis is to set the base for a general methodology for future applications and to make clear how a neural network works.

1.3 Objectives

This thesis can be subdivided into different objectives:

- Making a general overview of the machine learning world;
- Explaining what are the neural networks for object detection;
- Instructing neural networks for the detection of apples with high confidence.

Chapter 2

The History of Machine Learning

This chapter discusses several general concepts from machine learning science used in the thesis to devise the classification algorithm. Machine learning is becoming over the time, a wide field of research with almost eighty years of history. For this reason only neural networks, a sub-field of machine learning, evolved in the past few years in a branch of study dubbed *deep learning*, are covered in this part of the work.

2.1 Historical Overview of Deep Learning Techniques

"Machine learning is the science of getting computers to act without being explicitly programmed". It is the sub-field of computer science that, as explained by Arthur Samuel (1959) gives *computers the ability to learn without being explicitly programmed.* Teaching instead of programming. The tendency of building an aware machine can be traced back to the origin of modern computing, if not even before. The time-line 2.1 represents key points of machine learning history that will be treated in the following.

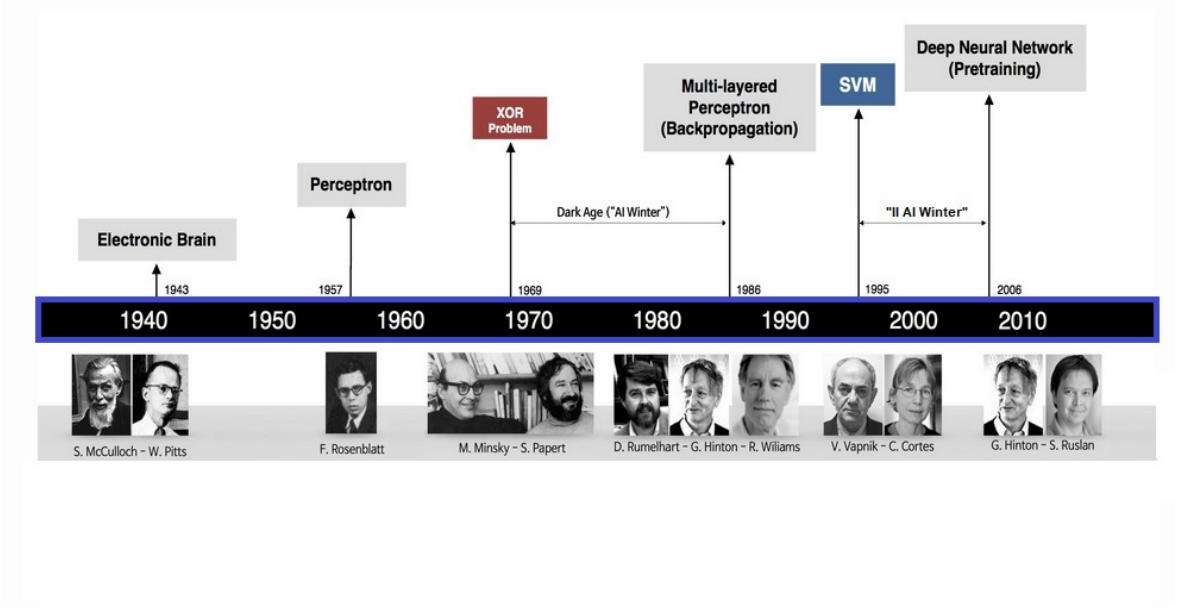
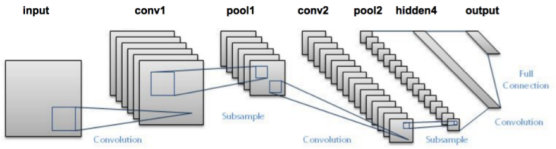


FIGURE 2.1: Visual representation of historical key events.

Early works in machine learning field were largely inspired by the first studies on human brain. The first researchers to make a contribute were Warren McCulloch and Walter Pitts. Together they invented an approach called *thresholded logic unit* (TLU) basing the main

structure on biological neuron: this was an explicit intent to emulate the internal behaviour of our brains.
The year of that first step was 1943. However, it isn't until Frank Rosenblatt's *perceptron* that it is possible to observe the first real ancestor of an actual neural network. For that year, 1957, it was a huge innovation. Through a learning procedure, it has the capability to improve its performances and with that it could detect correctly numbers and letters from an image. Rosenblatt was so convinced that his invention would drive to a mature and actual AI, that in 1959 he remarked: *"(the perceptron is) the embryo of an electronic computer that (the Navy) expects will be able to walk, talk, see, write, reproduce itself and be conscious of its existence"*. Perceptron is considered an important step forward, mainly for its capability to learn from its mistakes, and it was the first real example of *supervised learning*. Rosenblatt's perceptron began to earn popularity between researchers; the 60s in fact saw a huge amount of *intelligent* programs in very different fields of application: programs able to play table games, plan sequences of actions and able to elaborate opinions in precise fields.
But in particular, Marvin Minsky started to gain interest in the perceptron idea. Largely considered as one of the father's of artificial intelligence, perceived the hidden limitations of Rosenblatt's perceptron. In 1969 he published, along with Seymor Papert, a textbook called *Perceptrons* that stifled the rising Ronsenblatt's idea, ending the embryonic idea of neural networks. They gave a demonstration of inability to learn the trivial exclusive-or function (XOR problem), also giving it an unlimited training time. Indeed, the exclusive-or is a non-linear function and the perceptron, being a linear model, is incapable to learn it. This unlucky discover turned out, in a matter of weeks, all researches on neural networks and push the field of machine learning in what has later been called *dark age* or *AI winter*.
Lately, a well known researcher, Geoffrey Hinton, in 1986, turned the doom of machine learning. With a famous publication, written in cooperation with Ronald Williams and David Rumelhart, entitled *"Learning representations by back-propagation errors"* he gave a proof that neural networks with multiple hidden layers could be quickly trained. This was done by means of a new algorithm called *backpropagation* that still today represents a pillar of machine learning. Non-linear functions were not anymore a limitation because the additional layers provided networks with the capability to learn every type of function. Indeed, after this publication, in the same year, it was mathematically demonstrated this peculiar capability with a theorem known as *the universal approximation theorem*. This new algorithm has opened the feasibility of several different projects. For example, exploiting the novel backpropagation, Yann Lecun at *AT&T Bell Labs* devised a new type of neural network known as *Convolutional Neural Networks* (CNN) in order to recognize handwritten digits (figure 2.2).
Unfortunately, at that time it was not possible to apply the new technique to larger problems due especially to the calculation power of the computers: neural networks were put aside for a second time. But this time the slowdown concerned only the field of neural networks, and so by the 90s other machine learning algorithms started to gain popularity. For example, *support vector machine* (SVM) proved to be a very promising and usable solution.
Around 2006 Hinton once again introduced a new surprising idea: *unsupervised pre-training* and *deep belief networks*. As explained by him *"the idea is to train a simple two layers unsupervised model like a restricted Boltzmann machine, freeze all the parameters, stick on a new layer on top and train just the parameters for the new layer"*. With unsupervised learning a neural network is shown unlabeled data and asked simply to look for recurring patterns. Finally researchers, using this new methodology, could train more complex and deeper (multiple hidden layers) neural networks creating a completely new field of machine learning known as *Deep Learning*.
After this turn point, many researchers started to gain interest in these novel techniques and several publication started to be published by different research centres around the world.



(A) Graphical representation of LeNet architecture.

(B) LeNet-5 can tell apart two characters even when they strongly overlap.

FIGURE 2.2: LeNet-5 is a convolutional network designed for handwritten and machine-printed character recognition.

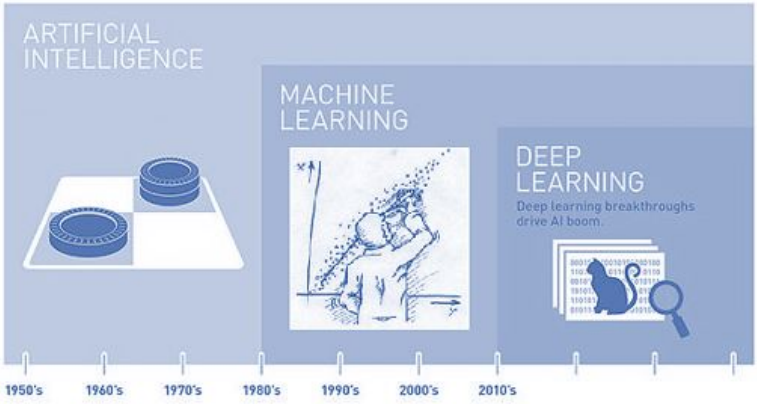


FIGURE 2.3: Representation of how deep learning is collocated in time within the AI research and in space within the machine learning field.

Since that moment neural networks, a little bit at a time, started to surpass traditional methods achieving important results in many different fields and applications: computer vision, speech recognition, medical diagnosis and many others. An important contribution in this last achievements has been given by Fei-Fei Li, a Stanford A.I. professor, convinced that *Data drives learning*. In 2007 she published *"ImageNet"* (Deng et al., 2009), composing an open database, through a collaboration between several research centres, of more than 16 million catalogued images. It was uploaded on the Web in 2009, and the following year she established an annual competition to boost computer-vision breakthroughs, *the Large Scale Visual Recognition Challenge* (LSVRC).
A major improvement has been done by *AlexNet* in 2012. For the first time, they had the groundbreaking idea to train the network using graphics processing units (GPUs). GPUs are parallel floating point calculators and they demonstrate how this hardware can be used to machine learning application with astonishment result also for huge models. Therefore they introduced the concept of *dropout* for the over-fitting problem and the introduction of the *rectified linear activation unit* (ReLU) as models for the neurons.
The foundation *Partnership on AI* on September 2016 testify the complete evolution and relevance of machine learning technique on our life. Corporations like Amazon, Facebook, Google, DeepMind, Microsoft, IBM, Apple joined this partnership with the precise purpose to build a true A.I. for the good of mankind.

2.2 An Introduction to Neural Networks

2.2.1 Thresholded Logic Unit (TLU)

The best way to understand what is a neural network is briefly introducing how biological neurons works.

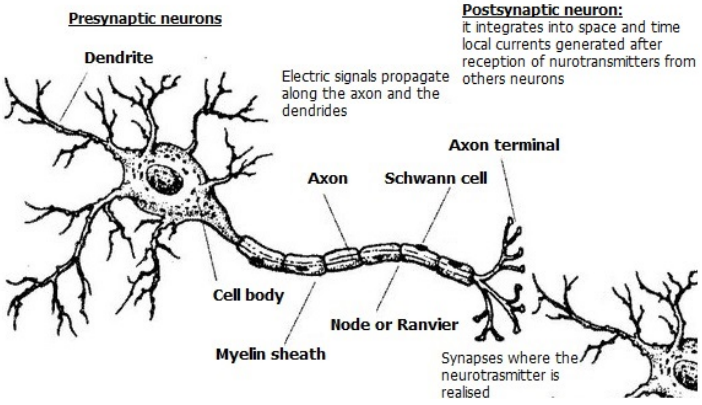
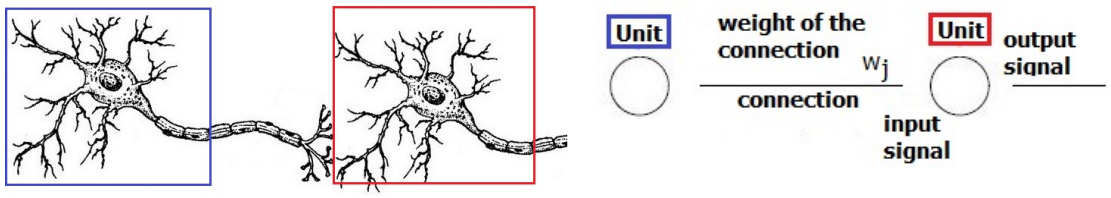


FIGURE 2.4: Presynaptic and postsynaptic neurons.

A brain neuron (figure 2.4) is a cell composed of a *body*, minor extensions called *dendrites* and a major one, the *axon*. At the end of axon is possible to observe a series of little protuberances said *synapses* used as contact elements for other nearby neurons. Neurons are able to transmit an electric signal along their axons. As soon as this signal is close to the synapses, the latter release a certain amount of chemical substance known as *neurotransmitters*. The quantity of neurotransmitter released in the synaptic slit determines the *conductivity of the synapse*, that is how much the synapse boosts or attenuates the electric signal from the axon. Downstream of the synaptic slit, the post-synaptic neuron is endowed with *receptors* able to capture neurotransmitters. When that happens, local currents are generated near the synapses. These currents can sum up in space and time. If the sum exceeds a certain threshold, an impulse of a certain entity and duration, known as *spike*, is generated. This signal goes across the new axon starting over the process.



(A) Two brain neurons connected with their synapses.

(B) A simple neural network composed by two undefined units.

FIGURE 2.5: The different parts of brain neurons are modeled achieving what is called a neural network. The brain cells are highlighted with different colours in order to put in evidence the duality with the positronic representation.

An attempt of reverse engineering of this biological structure is done by neural networks (figure 2.5). In a neural network neurons are represented as *units* able to elaborate input

signals from other neurons. Synapses are described as *connections* among units. The electric signal that goes through, along the axon, is represented with a number, usually between 0 and 1. The synapses conductivity is defined with variables known as *weight of the connection* (w_j). Usually, units perform two very simple operations on input signals: they compute the *activation potential* and with their activation they produce an output signal through an *activation function*. The activation potential is computed as the sum of the weighted signals from other neurons (figure 2.6). The output signal is processed on the base of the activation potential and certain mathematical functions that determine the nature of the unit.
Now it is easy to make a parallelism and understand how the first TLU works. This unit follows the following precise rules:

- Each artificial neuron has a given threshold, θ;
- Each logic unit has a binary output;
- Each neuron has a link with an inhibitory signal;
- A neuron receives many input signals, all having identical weights;
- Without an inhibitory signal, all weighted signals are summed together and the output f is equal to 1 if the sum is greater equal than the threshold.

The following equation shows the characteristics of the thresholded unit function.

$$f = \begin{cases} 1 : \sum_{j=1}^{n} w_j x_j \geq \theta \ \wedge \ no\_inhibition \\ 0 : otherwise \end{cases}$$

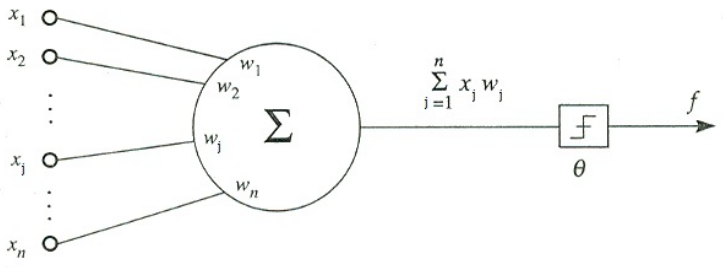


FIGURE 2.6: McCulloch and Pitts's threshold logic unit.

With this unit is trivial to implement any Boolean logic function except for the exclusive-OR that can be achieved starting by a cascade of three neurons. These was a groundbreaking achievement but unfortunately not very practical; the network doesn't display any kind of learning. In order to perform the desired computations, weights and connections have to be set manually. So, it is easy to understand that it does not bring any real advantage over traditional boolean circuits.

2.2.2 The Perceptron

The differences of perceptron with the old TLU can be summarized by the following points.

- Weights and biases are not all identical;
- Weights can be either positive and negative;
- The inhibitory signal is no longer present;

- Perceptron has been introduced with a learning rule.

The activation function, presented in the following equation is very similar to the TLU one.

$$output = \begin{cases} 0 : \sum_j w_j x_j \leq threshold \\ 1 : \sum_j w_j x_j > threshold \end{cases}$$

Now consider w and x as an array and b (*bias*) as the inverse of the threshold the equation can be re-written as:

$$a = \begin{cases} 0 : w \cdot x + b \leq 0 \\ 1 : w \cdot x + b > 0 \end{cases}$$

where a is the activation function. The bias can be think as a threshold that largely influences the output of the unit. In fact with a high value of it, the unit will likely have an output equal to one.

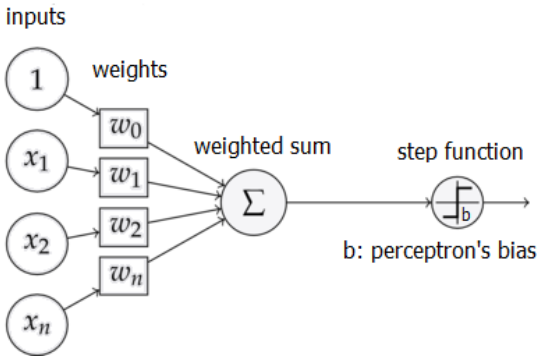


FIGURE 2.7: Rosenblatt's perceptron.

But what is much more important of this new idea is the learning algorithms, which enables to use artificial neurons in a radically different way to conventional logic circuits. This new unit, unlike its predecessors, can learn how to solve problems from its own. So, instead of creating a circuit by hand, Rosemblatt's perceptron is able to adjust its variables to deal with the given problem. The learning procedure is very trivial. Defining the output with $y = y(x_j)$, where x are perceptron's inputs, and a the actual output, is possible to define the *delta error*. Delta error is the difference between the true output and the present one, and the weight modification is proportional to delta

$$\delta = (y(x_j) - a)$$
$$\Delta w_j = \eta \cdot \delta \cdot a$$

where η is the so called learning rate $(0 \div 1)$. It is trivial to understand how this equation works. If the output of the i_{th} neuron is under the desired value, weights of the unit are modified to raise its total input. On the other hand, if it is higher than the desired value, weights are adjusted accordingly to the delta error.

2.2.3 Framework of Neural Networks

The previous architectures have in common the single output. This structure doesn't permits the learning of non-linear function. In fact, functions as XOR cannot be recognised by the perceptron and this situation gives birth to the AI winter. In order to obtain more complex equations neural networks were transformed adding more layers mutually interconnected. The impossibility to learn only problems linearly separable is now not anymore present. The

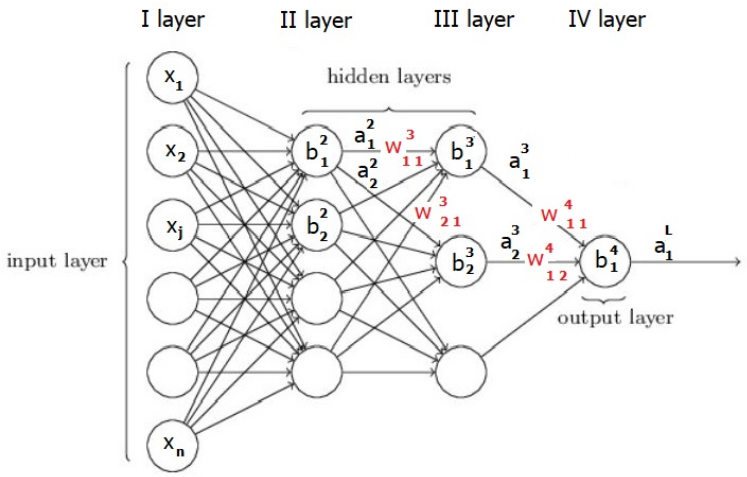


FIGURE 2.8: Nomenclature of a four-layer neural network.

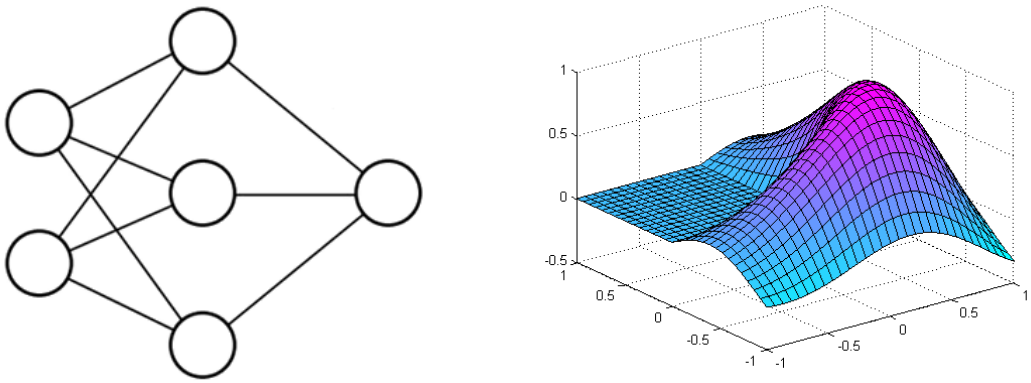
new architecture has many weights and biases that define it and needs obviously a more complex algorithm to be trained. The commonly used notation is presented in figure 2.8. The network used as example is a four layers neural network. The leftmost layer is composed of units known as *input neurons* and together they form the so called *input layer*. On the other side there is a layer known as *output layer* made by *output neurons* and that in this particular case contains only one unit. The two external layers are connected by the intermediate *hidden layers*. Four parameters are necessary and sufficient to describe the neural net:

- x_j is the j_{th} input for the j_{th} input neuron. Usually they are collected in a vector x;
- b_j^l is the j_{th} bias of the l_{th} layer. Also bias are collected in a vector b^l for each layer;
- a_j^l is the activation function of the j_{th} unit in the l_{th} layer. a^l is the related vector;
- w_{jk}^l is used to identify a weight for the link from the k_{th} neuron in the $(l-1)^{th}$ layer to the j_{th} unit in the l_{th} layer. In this case is it possible to compress all weights in a matrix for each layer w^l.

The differences between this structure and the previous perceptron can be showed using the activation function produced at each neuron given by:

$$a_j^l = \sigma(\sum_k w_{jk}^l a_k^{l-1} + b_j^l) \longrightarrow a^l = \sigma(w^l a^{l-1} + b^l)$$

With only one hidden layer, a network is able to compute a huge number of functions but already with two, its capabilities grows exponentially. This is trivial to understand because the second hidden layer allows to sum more complex functions coming from the previous layer. As example adding only one hidden layer the net is able to pass from the impossibility of learn the function XOR to the computation of a complex structure as showed in figure 2.9.



(A) Three layers neural network with two input neurons and one output neuron.

(B) Plot of the possible output graph.

FIGURE 2.9: Visual representation of the computational capabilities of a simple neural network.

2.2.4 Main Activation Functions

Modern neural networks exploit different kind of activation units. Many models have been developed over time. The following are most used ones.

Sigmoid Unit

Learning procedure involves a change of weights and also biases (the perceptron learning procedure implies only weights change). This naturally creates a difference in the output as presented in figure 2.10.

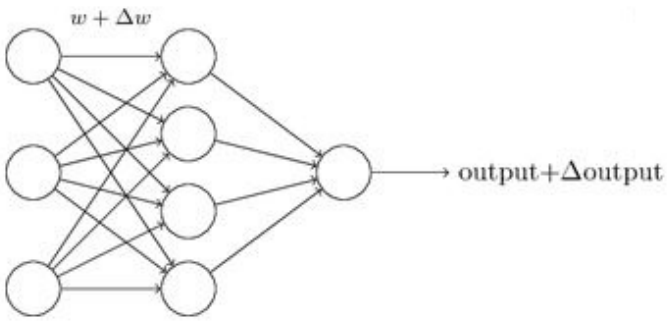
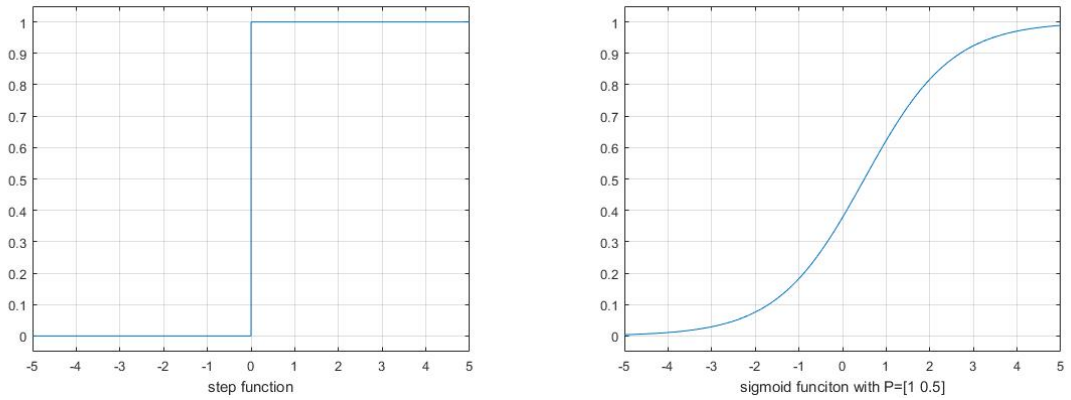


FIGURE 2.10: Weights variation propagate along the network resulting in a variation of the output.

Imaging that the network is made again of perceptrons, and that due to the expression of their activation functions, tiny changes at the input layer of the network can produce a completely different result at the output layer. So, if the output was zero, it could change to one with slightly variations of the values of some variables. The aim of the introduction of the *sigmoid function* is to lessen the effect of small variation and balance them to the final output.

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$



(A) Perceptron activation function. Output can only flip from zero to one depending from the bias.

(B) Sigmoid function output can assume any values between zero and one.

FIGURE 2.11: Step function and sigmoid function plots.

The new activation function results:

$$\sigma(wx + b) = \frac{1}{1 + exp(-\sum_j w_j x_j - b)}$$

This new type of unit overcomes the major problem of the perceptron model. Indeed, the output of the function does not have a discontinuous behaviour anymore, but it responds in a smooth way to input variations. The sigmoid function can be seen as a smoothed perceptron activation function.
The weights and biases change respectively the inclination and the position of the plot as shown in figure 2.12.

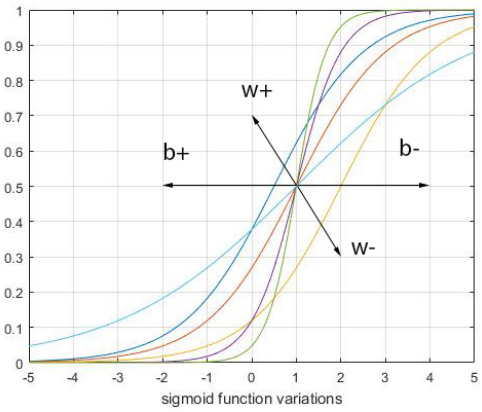


FIGURE 2.12: Depiction of how weights and bias influence the shape of the graph.

Linear Unit

A linear unit is a transfer function which produces an output equal to the activation potential. It does not make any modification to the input $a = \sigma(z) = z$. The resulting graph is a simply straight line going trough the first and third quadrant.

Tanh Unit

The hyperbolic tangent (*tanh*) function could be a great alternative to the sigmoid one. Many papers have demonstrated that this kind of artificial neurons in some situations perform much better than with other activation units. The main difference with the other activation function can be noticed in figure 2.13; in fact the output has a range between -1 and 1 and not between 0 and 1. This must be kept into consideration for the specific application devised. The tanh equation is:

$$tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

so the activation function can be calculated with $tanh(wx + b)$:

$$\sigma(z) = \frac{1 + tanh(z/2)}{2}$$

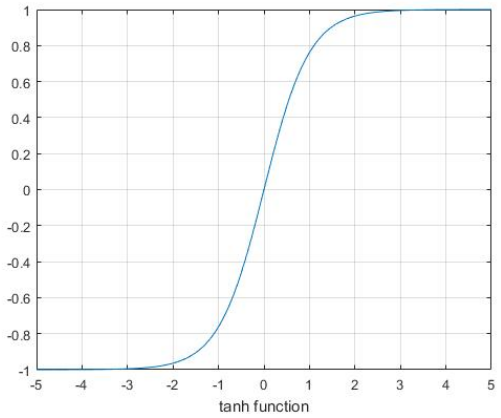


FIGURE 2.13: Plot of the tanh unit activation function. The range of the vertical axis is between -1 and 1.

Rectified Linear Unit (ReLU)

ReLU is also known as a ramp function and is similar to half-wave rectification in electrical engineering. Nowadays, rectifier is the most popular activation function for deep neural networks. This is due to many factors (most important the *vanishing or exploding gradient*). The expression of the ReLU function is:

$$\sigma(z) = max(0, wx + b)$$

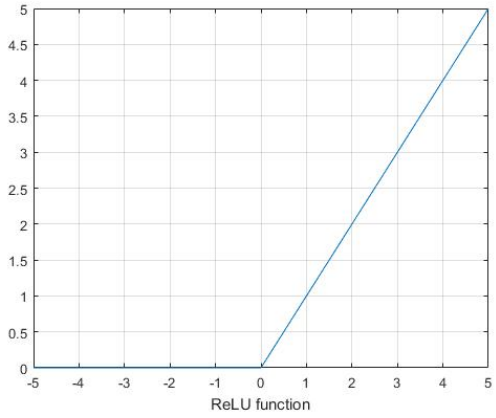


FIGURE 2.14: Plot of the rectified linear unit (ReLu) activation function.

Softmax Unit

These special units are mostly used for the output layer of a neural network, especially for classification problems. For this reason, this unit is always used in the final layer and a^L is a vector made of output components a_j^L and so $z^L = w^L a^{L-1} + b^L$.

$$a_j^L = \sigma(z_j^L) = \frac{e^{z_j^L}}{\sum_k e^{z_k^L}}$$

What is very interesting of this unit is that can be thought as a probability distribution since:

$$\sum_j a_j^L = \frac{\sum_j e^{z_j^L}}{\sum_k e^{z_k^L}} = 1 \tag{2.1}$$

Thus, for this last characteristic, softmax units are increasingly used in the output layer. In this way, it is possible to interpret networks predictions as probabilities, or a fast overview of the confidence of the network in its answer.

2.2.5 Gradient Descent

Far from now is clear how a simple neural network can work. More complex structure needs computation different from the basic ones in order to learn and to give significant results. At this purpose is useful introduce what is one of the central concepts in neural networks: the *cost function*. This is the factor that quantify if the net is near or not from the goals. One of the first, simple and nowadays most used is the mean squared error:

$$C(w,b) = \frac{1}{2n} \sum_x ||y(x) - a||^2 \tag{2.2}$$

where $y(x)$ are the desired output and a the actual one. An hypothetical perfect training algorithm should have variables in a way that $C(w,b) \approx 0$. It means that the net should cyclically adjust the variables in order to minimize the cost function.
In order to find this algorithm it is better to consider a generic case in which the input of the cost function is a n-dimensional vector v. It is not practical and in some cases not even possible to use calculus to minimize the $C(v)$ function. Another approach has to be adopted due to the huge amount of variables given by v. Considering small variations for each components v_j of the vector, C function varies as in equation 2.3.

$$\Delta C \approx \frac{\partial C}{\partial v_1}\Delta v_1 + \frac{\partial C}{\partial v_2}\Delta v_2 + \frac{\partial C}{\partial v_3}\Delta v_3 + \cdots + \frac{\partial C}{\partial v_n}\Delta v_n \tag{2.3}$$

All different Δv can be merged in an array $\Delta v = (\Delta v_1, \Delta v_2)^T$ and all derivatives in $\nabla C = (\frac{\partial C}{\partial v_1}, \frac{\partial C}{\partial v_2})^T$ defining the gradient of C, is possible to obtain equation 2.4.

$$\Delta C \approx \nabla C \cdot \Delta v \tag{2.4}$$

What is very useful in equation 2.4 is that permits to choose Δv so as to make ΔC negative. So keeping in mind that ∇C contains how C varies for each component v_j, is natural to set Δv as in equation 2.5

$$\Delta v = v' - v = -\eta \nabla C \tag{2.5}$$

where η is again the small positive value known as *learning rate*. Substituting 2.5 in 2.4 the following result is obtained:

$$\Delta C \approx -\eta \nabla C \cdot \nabla C = -\eta ||\nabla C||^2 \tag{2.6}$$

Equation 2.7 can be interpreted as an update rule. This gives what is called *gradient descent* algorithm. This powerful methodology cyclically updates v trying to minimize the reference cost function C, so that

$$v \to v' = v - \eta \nabla C \tag{2.7}$$

This type of algorithm is largely used in machine learning and requires only little modification. The main problem related to this computation is referred to the choice of the learning rate (figure 2.15): a high value is able to face large number of variables and do a fast computation but may not never converge to the final solution; on the opposite a small value risk to take too much time for the training and make the algorithm unusable.

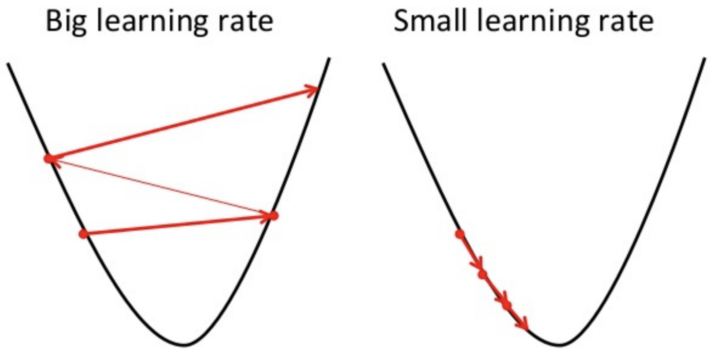


FIGURE 2.15: Representation of big and small learning rate.

The equation obtained so far are for a general case. Using weights and biases in equation 2.7 instead of general variables v, it is possible to obtain the actual algorithm used to train neural networks. In other words, the equation expressing C has components w_j and $b_j$such that

$$w_j \to w'_j = w_j - \eta \frac{\partial C}{\partial w_j} \tag{2.8}$$

$$b_j \to b'_j = b_j - \eta \frac{\partial C}{\partial b_j} \tag{2.9}$$

At this point is worth to take a look again to the formula of the cost function presented in equation 2.2. The cost function in a compact representation can be expressed as $C = \frac{1}{n} \sum_x C_x$, that is simply an average over the elements $C_x = \frac{||y(x) - a||^2}{2}$ where x is an input array of the input layer. Thus, in order to obtain ∇C , the gradients ∇C_x have to be separately computed for each input x and then averaged over all the inputs, so that $\nabla C = \frac{1}{n} \sum_x \nabla C_x$.

2.2.6 Stochastic Gradient Descent

This is the first solution to the application of the gradient descent to a huge training input number. The *stochastic gradient descent* considerably reduce the learning process time; instead of using all training data to compute ∇C, it is possible to estimate its value by adding a small number of ∇C_x. Instead of using all the n training inputs, only a subset of m of them is used.

$$\nabla C = \frac{1}{n}\sum_x \nabla C_x \approx \frac{1}{m}\sum_j \nabla C_j \tag{2.10}$$

The parameter n represents all training inputs and m is the chosen *mini-batch* of input vectors. At this point is possible to rewrite equation 2.9 as

$$w_j \to w'_j = w_j - \frac{\eta}{m}\frac{\partial C}{\partial w_j} \tag{2.11}$$

$$b_j \to b'_j = b_j - \frac{\eta}{m}\frac{\partial C}{\partial b_j} \tag{2.12}$$

The sum is not anymore over all training inputs but only on the inputs of the selected mini-batch. After updating all weights and biases another randomly mini-batch is selected. The gradient stochastic algorithm cyclically takes a new mini-batch, in a random manner, from the training data until all inputs have been considered. At this point a *training epoch* is completed and the algorithm starts the cycle again with a new epoch. The presented algorithm can have mini-batches of different sizes and it is also possible to choose a mini-batch of one only element. This procedure goes under the name of *online* or *incremental learning* and is similar to how human brains work.

2.2.7 Backpropagation Algorithm

Until now the main problem in order to apply gradient descent and thus make the network learn is the computation of the $\frac{\partial C}{\partial w^l_{jk}}$ and $\frac{\partial C}{\partial b^l_j}$. This algorithm is called *backpropagation* because, starting from output layer of a network and going back, it is able to compute the two partial derivatives of the cost function and subsequently compute in a easy way weights and biases. Practically, it is a technique that allows to use gradient descent and naturally all its versions in a practical case. Firstly, the algorithm defines a new quantity know as *output error* δ^l_j. Using this notation, output error refers to the j^{th} neuron of the l^{th} layer. This quantity is defined with equation 2.13. It is the variation of the selected cost function respect to the potential activation function. Again z refers to the j^{th} neuron of the l^{th} layer.

$$\delta^l_j = \frac{\partial C}{\partial z^l_j} \tag{2.13}$$

It is possible to group together all components in a single vector δ^l. So, backpropagation uses this novel quantities δ^l_j, and uses them to compute $\frac{\partial C}{\partial w^l_{jk}}$, $\frac{\partial C}{\partial b^l_j}$. At this point with some mathematical passages it is possible to extract the four fundamental equation of backpropagation.

$$\delta^L = \nabla_a C \odot \sigma'(z^L) \tag{2.14}$$

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l) \tag{2.15}$$

$$\frac{\partial C}{\partial b^l_j} = \delta^l_j \tag{2.16}$$

$$\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j \tag{2.17}$$

Where the last equation represent the selected cost function. Observing the four equations of the backpropagation algorithm, it is possible to understand how it works. Firstly, error values are computed for the output layer and then are back propagated until the input layer through the second equation. The third and fourth equations relate these previously computed errors with the partial derivatives, essential for gradient descent. It is therefore possible to use the results for the two equations 2.11 and 2.12 for the learning process. In conclusion, algorithms such as gradient descent or stochastic gradient descent are always related with backpropagation, that makes computations feasible. For example, using stochastic gradient descent and backpropagation, choosing a mini-batch of m training inputs, the following steps must be performed:

1. A group of m examples is selected from the n available of the dataset;
2. For each x of the m available:
   - x is applied to the input layer;
   - *Feedforward*: following all layers $l = 2, 3, \ldots, L$, potentials $z^{x,l} = w^l a^{x,l-1} + b^l$ and also activation functions are computed $a^{x,l} = \sigma(z^{x,l})$;
   - *Output errors*: $\delta^{x,L}$ is computed as $\delta^{x,L} = \nabla_a C_x \odot \sigma'(z^{x,L})$;
   - *Backpropagation*: for all layers $l = L-1, L-2, \ldots, 2$ are computed $\delta^{x,l} = ((w^{l+1})^T \delta^{x,l+1}) \odot \sigma'(z^{x,l})$.
3. For each layer $l = L, L-1, \ldots, 2$ are updated all variables, weights and biases, following the stochastic gradient descent algorithm;

$$w^l \to w^l - \frac{\eta}{m} \sum_x \delta^{x,l} (a^{x,l-1})^T$$
$$b^l \to b^l - \frac{\eta}{m} \sum_x \delta^{x,l}$$

4. A new mini-batch of input vectors is selected from the $n - m$ available until the epoch is concluded.

With this algorithm and with relatively enough computational power, is possible to train a general multi-layer network. Unfortunately, this is the most basic and primitive algorithm and a lot of possible problems could occur, making difficult the training of the network.

2.3 Training Problems of Neural Networks

Before starting, it is necessary to make some preliminary remarks. The type of neural networks presented so far are called feed-forward neural networks because the signal is always propagated in one direction. But there are many others that are not explained in this chapter like, recurrent neural networks, long short-term memory units, deep belief nets (generative models), Hopfield networks and so on. Moreover, it is important to clarify that the approach used to train the presented feed-forward neural networks is known as *supervised learning*. Nowadays, other two main techniques are vastly used dubbed *unsupervised learning* and *reinforcement learning* respectively.
Instead, in this section we will discuss different problems related with the training of neural networks and their possible solutions.

2.3.1 Neuron Saturation

In order to understand the problem we will observe a very simplified network with only one neuron. As seen in the previous sections, partial derivatives of the selected cost function $\frac{\partial C}{\partial w}$ and $\frac{\partial C}{\partial b}$ influence how network neurons learn. Using again the basic quadratic cost function, adapted for only one neuron $C = \frac{(y-a)^2}{2}$, it is trivial to achieve the following results:

$$\frac{\partial C}{\partial w} = (a-y)\dot{\sigma}(z)x = a\dot{\sigma}(z)$$
$$\frac{\partial C}{\partial b} = (a-y)\dot{\sigma}(z)x = a\dot{\sigma}(z)$$

In this simple case is easy to observe that both weights and bias are driven by the derivative of the activation function. So practically when $\sigma(z)$ is near 1 or 0 its derivative $\dot{\sigma}(z)$ gets very small. This creates a learning slowdown of the network that sometimes prevents any kind of improvement. The problem can be addressed with different approaches.

Learning Slowdown in the Final Layer

In order to solve the problem in the last layer (output layer), different types of cost functions can be used.

- *Cross-entropy cost function.* This is one of the most used cost function. The expression for multiple outputs is given by equation 2.18:

$$C = -\frac{1}{n}\sum_x \sum_y [y_j ln(a_j^L) + (1-y_j)ln(1-a_j^L)] \tag{2.18}$$

  Again, considering a single neuron with multiple inputs and only one output, it can be demonstrated that the gradient takes the form of equation 2.19:

$$\frac{\partial C}{\partial w} = \frac{1}{n}\sum_x x_j(\sigma(z)-y) \tag{2.19}$$

  Equation 2.19 suggests that weights updating is proportional to $(\sigma(z)-y)$ and the activation function, or better its derivative, does not play any role anymore.

- *Log-likelihood cost function.* This cost function is always used in conjunction with an artificial neuron already presented: the softmax unit. As has been previously described, softmax is increasingly exploited for the output layer in a lot of different applications. In fact, due to its formulation 2.1, the output layer values can be described as a probability distribution. The related cost function known as *log-likelihood cost function* is presented in equation 2.20:

$$C = -ln(a_x^L) \tag{2.20}$$

  In 2.20 x is the designated training input to the network and L indicates the output a of the final layer. If the network is very confident of the prediction, it estimates an output value close to one, and the corresponding cost function is very small. Again, it is possible to demonstrate that the gradients assumes the following expressions:

$$\frac{\partial C}{\partial b_j^L} = a_j^L - y_j \tag{2.21}$$
$$\frac{\partial C}{\partial w_{jk}^L} = a_k^{L-1}(a_j^L - y_j) \tag{2.22}$$

  Like the previous method, it is possible to notice that again the two partial derivatives do not have any derivative term and so, the learning slowdown problem is prevented.

Weight and Biases Initialization

With the previous solution there is an high probability to saturate some neurons in the hidden layers. Indeed, changing the cost function only affects the final layer. It is possible to reduce this problem initializing all variables with a Gaussian probability distribution with mean 0 and standard deviation equal to $\frac{1}{\sqrt{n_{in}}}$ where n_{in} is the number of input weights of the selected neuron. With this technique learning slowdown is greatly reduced because all neuron units have less probability to saturate. Instead for biases initialization there are two main approaches: either set at the beginning all biases equal to zero or use the previous approach (mean of zero and standard deviation equal to 1). Indeed, biases affect much less the slowdown problem than weights and so they do not need any adjustment.

2.3.2 Data Overfitting

Another main problem is given by data overfitting. Having a huge amount of variables to train is likely to overfit data and instead of generalize abstract concepts the network could learn peculiarities of the training set. This must be avoided because a network able to make predictions only on data taken from the training set is useless. Fortunately, many different techniques can be used to overcome this important problem.

Dividing Data

This method involves dividing available data in three different categories. *Training data* dedicated to the learning process of the network, *Validation data* to test the network at the end of each epoch and *Test data* in order to make a final check at the end of the training session. So a first signal of overfitting is when the accuracy of the validation data stops increasing and the one of training data keeps increasing. A basic strategy known as *early stopping* consists of stopping training once the classification accuracy has saturated. In general, keeping track during training of these three class of data can be very useful not only for overfitting problem but also to set and find good *hyper-parameters* for the neural network.

Artificially Expanding the Training Data

Modern neural networks have thousands of parameters or even more and so, it is quite easy to overfit training data without having a generalization of the problem. The easiest way to overcome this problem is increasing the number of training example but naturally, this option is not always available for practical reasons. Instead, a group of researchers have developed a technique know as *artificially expanding the training data* in order to enhance the dataset without adding more examples. The basic idea is to expand the available data set by processing images with filters and distortions so applying operations that in a way reflects real world variations. Example of image processing are rotation, translation, skewing and elastic distortion.

Regularization Techniques

The methodologies just explained are one way to face overfitting problem but usually alone are not sufficient. Fortunately, there is a set of techniques, known as *regularization techniques*, that are able to reduce this major problem also with a fixed dataset. Here are presented the two most used but, over the time, many papers have proposed valid alternatives.
The first one is *L1 regularization*. The procedure requires a modification of the selected cost function. For example, using L1 regularization with the cross entropy equation presented in

equation 2.18, a term called *regularization term* is added:

$$C = -\frac{1}{n}\sum_x\sum_y[y_j ln(a_j^L) + (1-y_j)ln(1-a_j^L)] + \frac{\lambda}{n}\sum_w |w| \tag{2.23}$$

where in the second sum w is the number of weights and λ is a new hyper-parameter that has to be set. It is possible to simplify equation 2.23 as:

$$C = C_o + \frac{\lambda}{n}\sum_w |w| \rightarrow \begin{cases} \text{if } \lambda \text{ is small the regularization term can be omitted} \\ \text{if } \lambda \text{ is large the analyzed model learns small weights} \end{cases}$$

Intuitively, the proposed method makes the model learn small weights and they can be large only if they greatly decrease the value of the selected cost function. Moreover, L1 regularization tends to keep the majority of the weights equal to zero and concentrate non-zero variables in a small region of important connections.
The second very popular technique is known as *L2 regularization*. Similar to L1 regularization, it requires to modify the selected cost function adding an extra term.

$$C = C_o + \frac{\lambda}{2n}\sum_w w^2$$

Making all needed computations is possible to demonstrate that gradients assume the following form:

$$\frac{\partial C}{\partial w} = \frac{\partial C_o}{\partial w} + \frac{\lambda}{n}w$$
$$\frac{\partial C}{\partial b} = \frac{\partial C_o}{\partial b_o}$$

Thus, gradient descent algorithms is expressed by

$$w \rightarrow w - \eta\frac{\partial C_o}{\partial w} - \frac{\eta\lambda}{n}w = (1 - \frac{\eta\lambda}{n})w - \eta\frac{\partial C_o}{\partial w}$$

Dropout

It is a radically different technique for regularization. Unlike L1 and L2 regularization, this techniques does not required to make changes to the selected cost function. For each training step only a percentage of neurons are activated (usually 50%). The rest of the neurons are bypassed and their weights and biases aren't updated. At the successive training step the process is repeated with a different neurons randomly picked-up. As it is explained by the same authors *"[...]this technique reduces complex co-adaptation of neurons, since a neuron cannot rely on the presence of particular other neurons"*. With this methodology, therefore, a neuron is forced to learn more robust features that are useful to abstract data. Dropout has been especially useful in training large, deep networks, where overfitting is a major problem.

2.3.3 Vanishing Gradient Problem

In the history part is not explained why after the publication of backpropagation, other machine learning techniques like support vector machines (SVM) or random decision forest took the lead putting aside neural networks. Why researchers have not taken advantage of backpropagation algorithm to build neural networks of many layers? Unfortunately, though backpropagation gives the way to compute the gradient and theoretically propagate it through the network, updating all weights and biases, this does not train the network in a proper way.

For problems related with gradient descent, usually early neurons learn very slowly or do not learn at all. It has already been described how the variables inside a network are updated according to the partial derivatives $\frac{\partial C}{\partial w}$ and $\frac{\partial C}{\partial b}$. Unfortunately, these quantities tend to decrease getting nearer the first layers. This phenomenon is known as the *vanishing gradient problem.*

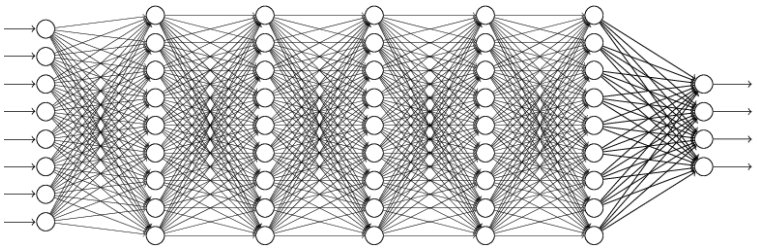


FIGURE 2.16: A seven layer neural network affected by vanishing gradient problem.

The best way to understand this problem is to write the first 2.14 and second equation of backpropagation 2.15 in matrix form. Taking 2.14 is possible to write

$$\delta^L = D(z^L)\nabla_a C \tag{2.24}$$

where $D(z^L)$ is a square matrix with all values equal to zero with the exception of the diagonal whose entries are $\sigma'(z_j^L)$. Thus, in this case instead of the Hadamard product 2.24 has a conventional matrix multiplication. Again from 2.15 is possible to obtain 2.25.

$$\delta^l = D(z^l)(w^{l+1})^T\delta^{l+1} \tag{2.25}$$

Finally combining 2.24 and 2.25 is possible to obtain the following result:

$$\delta^l = D(z^l)(w^{l+1})^T \dots D(z^{L-1})(w^L)^T D(z^L)\nabla_a C \tag{2.26}$$

Looking carefully at equation 2.26 it is possible to notice that, aside $\nabla_a C$, it is made with several terms with the form $(w^j)^T D(z^j)$. Considering the network made of sigmoid neurons, a single entry of the matrix $\sigma'(z_j^L)$ could have maximum value $\sigma'(0) = \frac{1}{4}$ (figure 2.11). For this reason, for a single entry it is possible to write $|w_j\sigma'(z_j)| < 1/4$. Thus, the matrix $D(z^j)$ does not contain entries larger than $\frac{1}{4}$. It is trivial to notice the all terms tend to make the gradient unstable. A neuron in the first layers has a lot of components in its version of equation 2.26 and so, its gradient has a very small value. So, gradient tends to vanish in earlier layers producing an almost null training of the variables or a great slow down in the update of weights and biases during the process.
In conclusion, the problem just presented is only one of many that have prevented researchers in training networks with multiple layers and with complex frameworks. Fortunately, some researchers did not give up and kept searching for solutions for these problems. Nowadays, due to the contribution of these scientists, from some specific research centers, many techniques and algorithms have been devised able to face learning problems of deep and complex neural networks. These new structure have gradually overcome other machine learning techniques and now, as a matter of fact, they are leading almost all the research on artificial intelligence.

2.4 Deep Learning

This section analyzes the last part of the time line in figure 2.1, starting from 2006 when the deep learning started to be a reality. The main idea under deep learning is to break down the

problem in multiple simpler sub-problems. So, it simplifies a complex task in simpler ones in order to achieve a higher accuracy. This approach is not so different from how the human brain works but the complexity is still not even comparable. Anyway, deep learning tries to emulate some peculiar capabilities of it and that is achieved giving a role to the different parts of a network. It is not anymore a matter of stacking multiple fully connected layers but each one has a specific role, finalized to the specific task assigned. Networks with this type of logic and framework are called *deep neural networks*. Again before starting it is necessary to make some preliminary remarks. Deep learning is a huge subject with many fields of application and many different ideas and thus, for the sake of the discussion, only concepts related with image recognition are presented. Classification is a sub-field of pattern recognition and it attempts to assign each input value to one of a given set of classes. However, the majority of concepts can be used for other applications such as speech recognition, natural language processing, object detection or localization, robot navigation systems, self-driving cars and so on. The following points try to summarize the key ideas that made possible the development of deep neural networks:

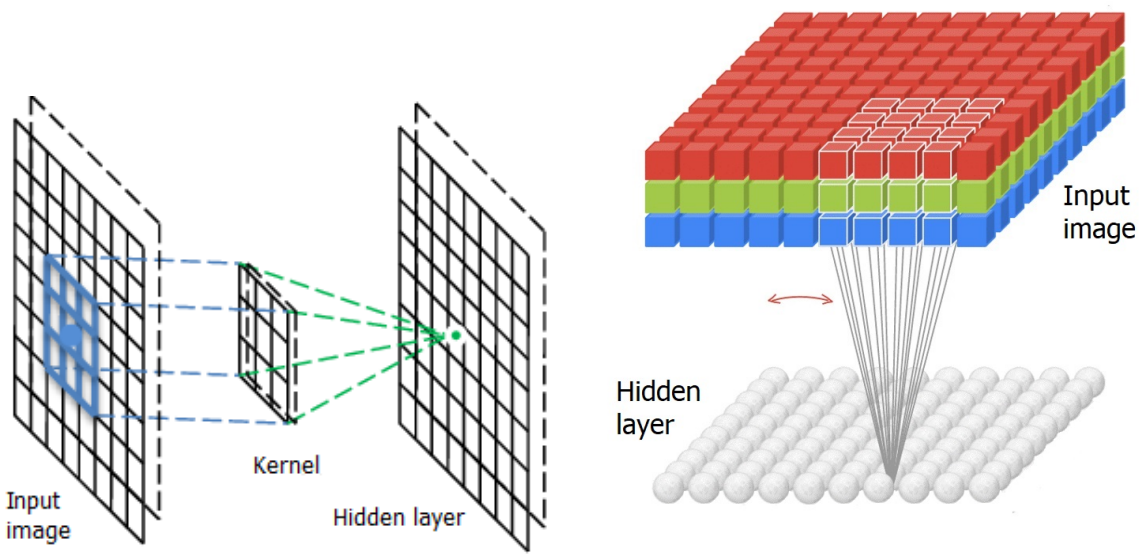
- Reduction of number of parameters using ideas such as convolutional neural networks (CNN), pooling layers, and so on;
- Overfitting reduction through powerful regularization techniques;
- Speed up training using ReLU activation function instead of sigmoid neurons;
- Train models with GPUs instead of CPUs.

2.4.1 Convolutional Neural Networks (CNN)

The network presented in figure 2.16 and all others showed until this section are shown with fully connected layers (all neurons of a layer are connected to all neurons of the following layer). With this approach it is possible to achieve great results but it is not trivial to scale them to more complex problems. The major challenge is that networks with fully connected layers take inputs without giving any particular meaning to them. For example, when they receive as input an image, they do not give importance to the spacial structure of it. Intuitively, fully connected layers lack logic and they require to be specialized for the specific task presented. Moreover, fully connected layers exponentially grow number of parameters and so models are more difficult to train. The main idea is to keep the information of the spatial structure of the input image. This is achieved by convolutional neural networks that have proved to be particularly efficient to classify pictures. In order to better understand how a convolutional layer works it is useful to analyze CNN under its key parameters.

Receptive Field

In a convolutional network an input image is considered with its natural form, a matrix of pixels for black and white pictures and as three distinct matrices for colour images (red, green blue). As usual, the input layer is connected to an hidden layer but, this time, for CNN architecture each neuron is not connected with all neurons of the following layer. With CNN, input neurons are linked with only certain specific hidden units. So, each hidden unit is connected to a selected portion of space of the input layer. This little window is known as *local receptive field*. Each connection of the receptive field has a weight and a single bias. So for example in figure 2.17 (a) is represented a local receptive field of 3x3 (in blue). So, for this specific CNN layer there are 9+1 different variables. This property is known as *shared weights and biases*. In order to compose the hidden layer, the window slides through the input image's pixels creating step by step a layer of hidden neurons. How much the receptive field slides



(A) Monochrome image processed by a filter of 3x3 receptive field.

(B) Colour image processed by a filter of 4x4 receptive field.

FIGURE 2.17: Two CNN layers with different input data.

is said *stride length*. This two terms, stride length and local receptive field dimension, are adjusted with the dimension of the input image. With larger input pictures, it is better have higher values of these two terms. Summing up, basically a CNN layer has a local receptive field of KxK dimension with KxK+1 variables (weights and bias). This square window slides through the input image of a quantity S and at each step applies the equation known as convolution.

$$\sigma\left(b + \sum_{l=0}^{K}\sum_{m=0}^{K} w_{l,m}a_{j+l,k+m}\right) \tag{2.27}$$

As it is possible to observe, equation 2.27 takes into account only one bias and KxK weights. Instead, a_{jk} are the input activation signals. The union between all groups of shared variables, weights and biases, is known as *kernel* or *filter*.

Zero Padding

Taking the first picture presented in 2.17 and supposing to have a input image of 32x32 pixels and a local receptive field of 3x3 the output image will be of 30x30 because is possible to move the receptive field only 31 times. This results in a loss of information that sometimes can be disadvantageous. For this reason, is possible to apply a technique, known as *padding* or *zero padding*, in order to obtain an output volume of the same dimension. To do this, the input image has to be surrounded by a frame of lenght P and all the new grid cells have value zero. As shown in figure 2.18 a zero padding of 2 has been added. In order to compute the dimension of the output image is possible to apply equation 2.28.

$$O = \frac{(I - K + +2P)}{S} + 1 \tag{2.28}$$

where O is the output length/height, I is the input length/height.

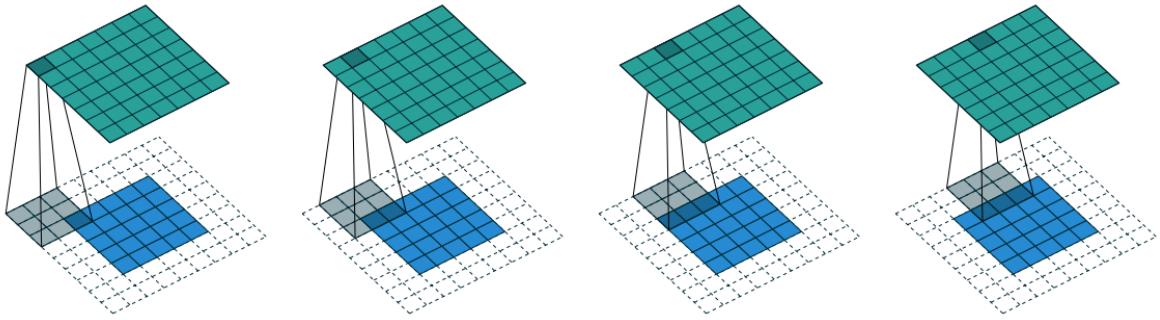


FIGURE 2.18: CNN layer with zero padding equal two.

Features Map

Every local receptive field, associated to an hidden layer, trains its KxK weights and bias and that variables are tuned to recognize a specific feature. Then, sliding the receptive field over the image, the layer tries to detect the presence of that specific feature inside the input picture. Whereby, a neuron of the layer is activated when its receptive field its excited by the particular feature trained to recognize. But a CNN layer of this type can detect just a simple kind of localized feature. For this reason each CNN layer has a stack of features map N. Each feature map has different shared weights and bias and so a different kernel. To be more

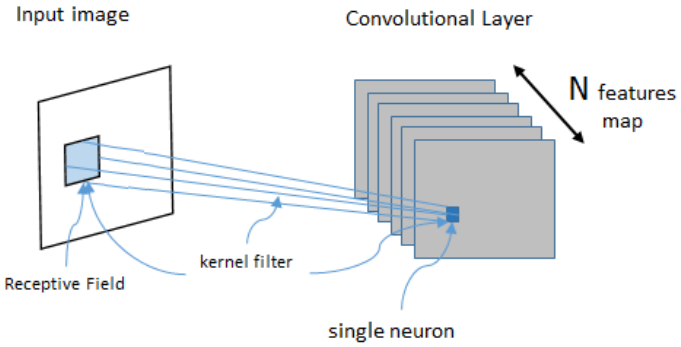


FIGURE 2.19: CNN layer with N feature maps.

precise, each feature map trains its neurons to detect different features. All feature maps are defined by a set of KxK shared variable, weights and biases. In this way a CNN layer it is able to detect contemporaneously from the same image different features. Moreover, with fully connected layers there is a weight and a bias for each connection instead, this approach largely decreases the number of variables to train. For each feature map are needed $K \cdot K + 1$ parameters and with N features map the total number of variable is $(K \cdot K + 1)N$.
Here is better to make a clarification; as for colour input image, also after a CNN layer, instead of a matrix, our data is transformed in a 3 dimensional tensor with dimensions (K, K, N). This means that the convolutional operation 2.27 is not any more done on a square of pixels but on a cube of dimension KxKxN.

2.4.2 Pooling Layer

A convolutional neural network always contains also this kind of layer. More than a layer is an operation made on the tensor output of the CNN layer. Its simple role is to further

reduce the number of model parameters decreasing the dimension of the image going through the network. The idea, as all other parts of a CNN, is supported by a logic framework. Indeed, if a particular feature is detected by the previous convolutional layer it is not anymore important its exact location in the original input image. So, pooling layer, with a very simple mathematical operation, narrows down distance between detected features facilitating the detection of more complex features from deeper convolutional layers. It gets closer features in order to achieve a better overview of the overall image. Depending on mathematical operations they perform, pooling layers can be of different types.

Max Pooling

In max pooling, each unit of the layer scans with its receptive field a portion of the input matrix and outputs the maximum value of it. So, with a certain filter dimension it strides across the matrix with the same length of the dimension. For example, in figure 2.20 the input region is 2x2 and as for a CNN layer the window is slipped with a stride length of two taking every time the maximum value.

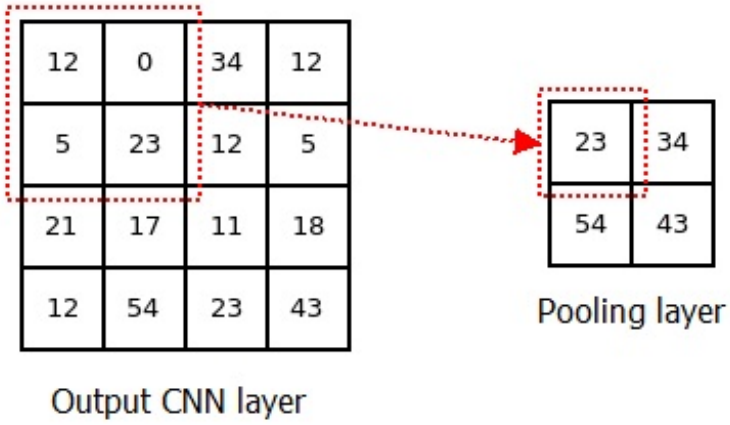


Figure 2.20: Example of max pooling layer.

L2 Pooling

L2 pooling is another common approach. Again, it strides across the image with a certain filter dimension but, in this case, it computes the square root of the sum of the squares of the input values. In some cases, this approach could be more efficient than max pooling due to its more complex operation.

2.4.3 Basic Architecture of a Convolutional Neural Network

Though the network depicted in 2.21 could seem very different from what have been presented so far, its basic working principle is always the same. Weights and biases are the parameters to be trained and gradient descent, or its variations, with backpropagation are used to achieve this purpose. Unlike fully connected layers, CNN gives more attention to the spacial composition of its input and do not treat all inputs in the same manner. Moreover, the problem of recognition is subdivided along the lenght of the network. First layers learn to recognize simple features like lines, circles, edges, while deeper layers, supported by the interposition of pooling layers, detect more complex compositions of features. The image is first analysed through its edges and lines than, their detection is used, in later layers, to find

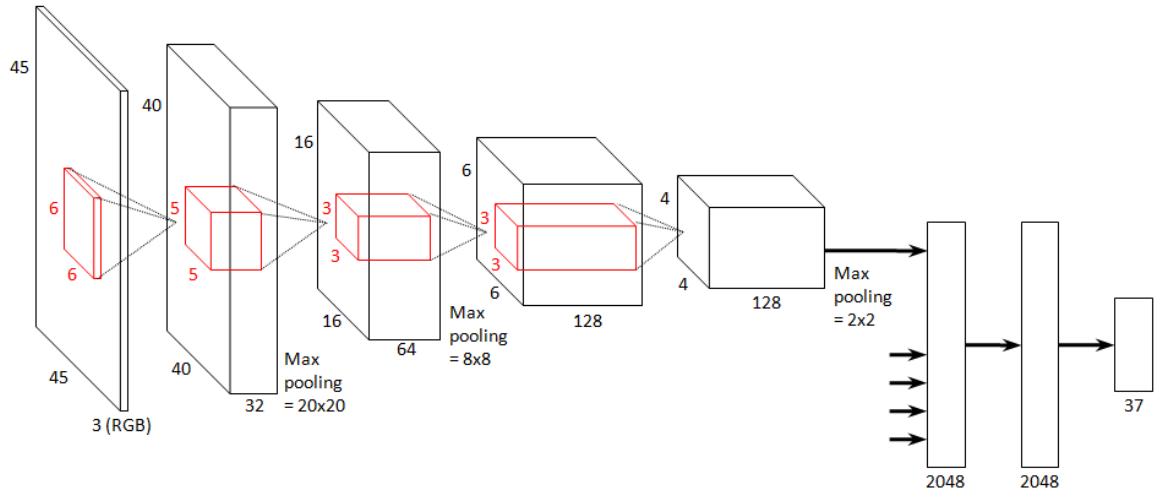


FIGURE 2.21: Example of a common CNN. Zero padding has been set to zero and convolutional and pooling layers has variable local receptive fields.

more abstract patterns like objects and physical elements. In a common CNN network after a cascade of this layers (convolutional and pooling), data is flattened in a one dimensional vector. This vector feeds a fully connected layer, usually characterized with dropout regularization, attached to a final softmax layer. This later is used to make a prediction of the input image. In conclusion, it is possible to think about a cascade of convolutional layers as a detector of increasingly more complex patters and the last fully connected layer as a computational model that takes detected features and associates them to the right outputs. As a matter of fact, this simple logic of neural network performs much better than a trivial network with a stuck of hidden layers. Moreover, substituting sigmoid function units with rectified linear units it is possible to obtain faster training processes. In fact, in the case of high values of z, rectified linear units $max(0, z)$ does not saturate to one like sigmoid neurons. In this way networks can keep learning at the same speed and do not risk a learning slowdown due to the saturation of their units.

Chapter 3

Machine and Deep Learning Platforms

One of the first step done during this thesis work was the research about possible available platforms for the computation of machine and deep learning networks. It should be underlined that most of the solutions found are written or base their functionality on *Python* programming language; this was a choice due to my previous experience with it and the flexibility given by this type of language, different from *Matlab* or *R*.

3.1 Machine Learning

As previously explained, machine learning is the highest branch of A.I. . Most of the algorithms actually used are based on probabilistic and statistical computations that reach high results especially when considering analytic data. In fact many algorithms of this type are actually used by banks or insurance agencies for studying the behaviours of customers and for offering the best tailored solutions. Many of these algorithms can be used with low computational power and processing time almost null; the only request for a meaningful result is the huge amount of data. It is not surprising that nowadays data have become the most valuable good. The libraries and platforms found are:

- Scikit-Learn
- Mllib (Apache Spark)
- WEKA
- H2O
- Accord.net (developed in C#)
- Torch (developed on LuaJIT)
- pyBrain

Among these solutions, having slightly different each other for applications and supported languages, *Scikit-Learn* [6] by *Google* is the most used ones. It is a simple and efficient tool for data mining and data analysis built on *NumPy*, *SciPy*, and *matplotlib*. With this library it is possible to do classification, regression, clustering and preprocessing of data and also compare models and hyper-parameters.

3.2 Deep Learning

When talking about deep learning the algorithms that have to be used are more complex and they require higher computational power. During the first part of the thesis work the following platforms were found:

- Tensor Flow (Google)
- Caffe (Berkeley AI Research)
- pyTorch
- CNTK (Microsoft)
- Nervana Neon (Intel)
- MXNet (Apache incubator)
- Deep Water

The major differences among these platforms are the documentation and hence the support community. All these platforms are open-source because for now the main aim of the growing deep learning community is the development of new solutions and merging this complex world to real life. Obviously this choice implies that the available documentation is not the same for all these solutions: only the most used ones have the necessary support community and so only these ones are really growing.
The most significant ones are *Tensor Flow* [7], *Caffe* [8] and *CNTK* [9]. Tensor Flow is a neural network library created by Google that learns to solve tasks by enhancing and processing of data in different nodes that makes it possible to find a correct result. This machine learning library offers to developers the usage of APIs for Python and C/C++ languages. Thanks to the huge community that uses this platform it was taken into account as first platform for deep learning algorithms used into this thesis. Furthermore, this is one of the two platforms, alongside Caffe, supported by the Movidius (§7.2). Caffe framework is created for commercial use in the first turn. At the same time, it is an open-source, it is written in C++ language and it allows to write user algorithms in Python. Caffe offers a wide toolkit for the development and deployment of modern deep learning algorithms. It is successfully used for speech and images recognition in different fields including fields like astronomy and robotics. It has a clean architecture for instant deployment that perform a quick switching between central and graphics processing units. The open-source code that allows developers not only to control integration but also modify it for their needs. This platform would have been another good solution but it was not taken into account due to its poor community contributions. CNTK is a Computational Network Toolkit developed by Microsoft for deep learning algorithms. The toolkit is used in speech recognition services predominantly, such as Windows Cortana, Skype Translator etc. The toolkit can also be used for automated translation and image recognition tasks resolution. It is developed in C++ language. CNTK allows developers to create distributed neural networks made in the form of the oriented graph. It supports several models of neural networks as feed-forward, convolutional, recurrent neural networks as well as their combinations.
Above all those solutions is placed *Keras* [10]. Keras represents the library that can work with neural networks on a higher level. It simplifies many tasks, it is used in quick experiments and decreases the amount of the same code. It can be used with Tensor Flow, Caffe or CNTK backend: this means that this platform can use the features of the toolkit listed but in a easier way. For this reason it is considered as the best choice for deep learning computations.

Chapter 4

Object Detection

A particular area of Deep Learning is interested into the processing of images in order to detect and recognise object. This area can be reconnected to the theory behind *Computer Vision* and its evolutions. Computer vision is the interdisciplinary field that concerns the creation of algorithms able to understand high-level features from digital images or videos. From what concerns the engineering application, it seeks to automate tasks that the human visual system can do.
This discipline embraces the acquisition, the processing and the analysis of the digital images in order to produce numerical information to be used in the deep learning computation. Computer vision was born in the late 1960s in universities pioneering artificial intelligence but the real foundation of what is used also today is attributable to studies done around 1970s that produced algorithms as extraction of edges, labelling of lines and other techniques.
The innovative techniques developed in the last years permit to create accurate machine learning models that are capable of localizing and identifying multiple objects in a single image. Those two operations are different and often confused with each other. The figure 4.1 explains the light difference.

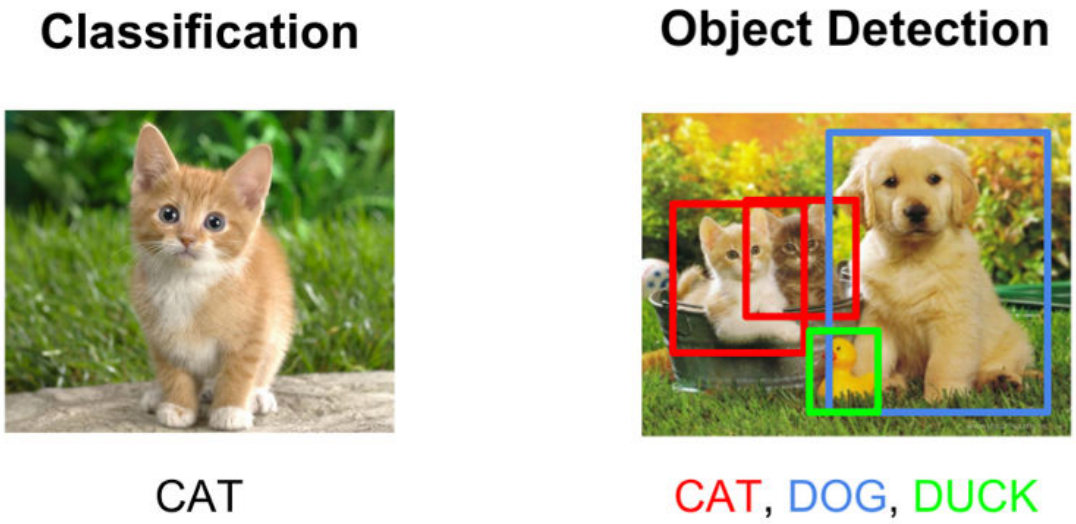


FIGURE 4.1: Object Detection and Object Classification [11].

In general, it can be possible to subdivide two scenarios, the first requires the identification of that image under a specific category, so classification can be the right choice; the second claims to identify the location of the objects recognised into the image and process that information, in those case object detection can be used. Obviously the two methods are far distance each other considering the complexity of the algorithm and results, so is not always possible to make the right choice. Creating accurate machine learning models capable of localizing and identifying multiple objects in a single image remains a core challenge in computer vision. However the concept behind object detection is easy: every object class owns special features that can be used to classify that class, as example all squares have four edges, perpendicular at the corners and with equal side lengths; if the image shows those characteristics, then a square is recognised. The same procedure is applied on all the desired classes. It is important to say that with the usage of deep learning those features are learned by the algorithm and not imposed by the programmer.

Region of Interests (RoIs)

Typically the process of object detection framework can be subdivided into three steps:

1. Creation of the Regions of Interests (RoIs): a model or an algorithm is used to generate a large set of bounding boxes spanning the full image.
2. From each bounding boxes generated, visual features are extracted that are evaluated in order to determine using visual features whether and which objects are present in the proposals.
3. The overlapping boxes that include the same object are combined into a single box that may contain the entire detected element.

An example is showed in figure 4.2.

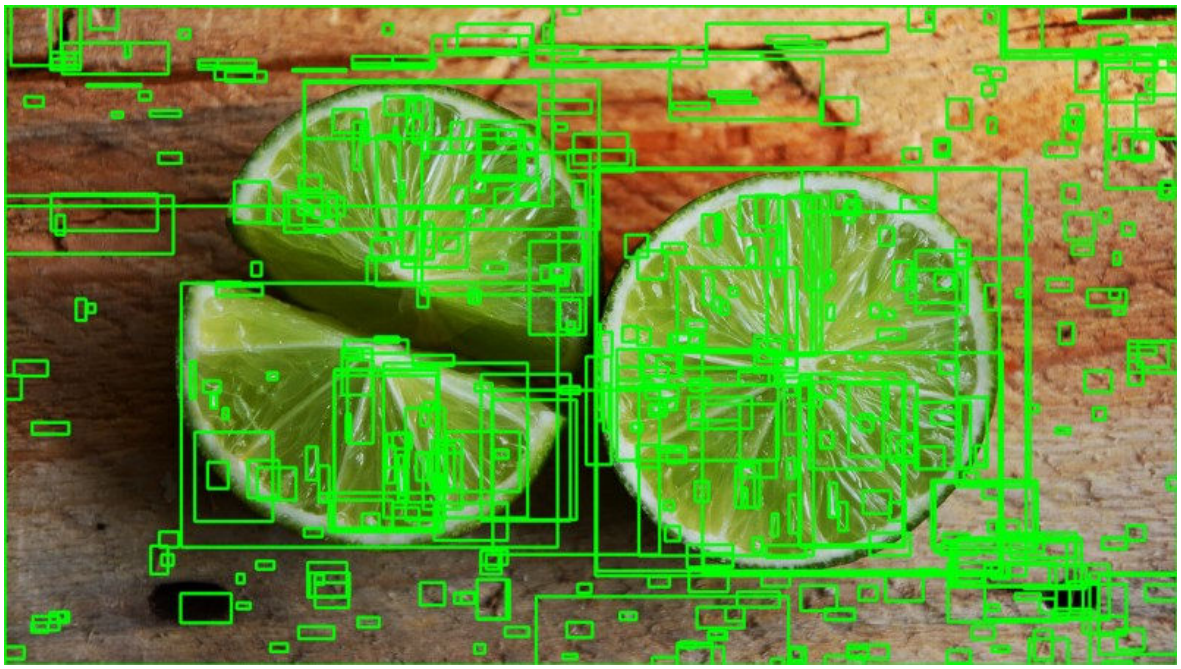


Figure 4.2: An example of created Region of Interest [11].

Image Gradient Vector

The creation of the RoIs can be implemented in various ways evolved during the years. All these methods have their operating principle based on the *Image Gradient Vector*. The aim is the knowledge of the direction of colours changing from one extreme to the other and then measure the gradient on pixels of colours; it should be underlined that the gradient in these cases is discrete due to the fact that each pixel is independent and cannot be further split. The declared vector owns the variation of the colour of the pixel along both directions, x-axis and y-axis. The definition for a pixel localised in the (x, y) location is the following:

$$\nabla f(x,y) = \begin{bmatrix} g_x \\ g_y \end{bmatrix} = \begin{bmatrix} \frac{\partial f}{\partial x} \\ \frac{\partial f}{\partial y} \end{bmatrix} = \begin{bmatrix} f(x,y+1) - f(x,y-1) \\ f(x+1,y) - f(x-1,y) \end{bmatrix}$$

It is noticeable how the partial derivative terms are computed as the colour difference between adjacent pixels; for example $\frac{\partial f}{\partial x}$ is computed as the colour of the the pixel on the left, $f(x, y-1)$, minus the colour of the pixel on the right, $f(x, y+1)$; the same is for the other direction (Figure 4.3).

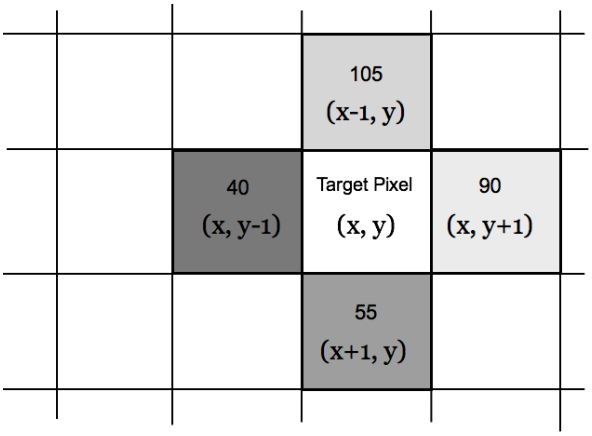


FIGURE 4.3: Gradient representation [11].

As in mathematics also in this case the gradient has two important values used in the algorithm:

- *Magnitude* computed as the L2-norm of the vector: $g = \sqrt{g_x^2 + g_y^2}$
- *Direction* computed as the arctangent of the ratio between the partial derivatives on two directions: $\theta = \arctan(g_y/g_x)$

This process applied to every pixel implies a lot of wasted time. Instead the process can be included into the application of a convolution operator on the entire image matrix; these operators can be different based on the algorithm and are created for a specific purpose as sharpening, blurring and edge detection. Two examples are the *Prewitt operator* and the *Sobel operator*.

4.1 Evolution of Object Detection

Next here are listed the main solutions and applications of object detection during its evolution.

4.1.1 Haar Feature-based Cascade Classifiers

The *Haar Feature-based Cascade Classifiers* is the first real object detection classifier proposed in the paper *"Rapid Object Detection using a Boosted Cascade of Simple Features"* [12] in 2001. This approach bases its function on machine learning using a cascade trained on several positive and negative images and then applied on other test images. The train set is initially supplied to the system which has to extract features from it. For this operation are used Haar features (Figure 4.4) which are applied like a kernel used before. Each feature is a value obtained by the subtraction of pixels sum under the white rectangle from the pixels sum under the black rectangle.

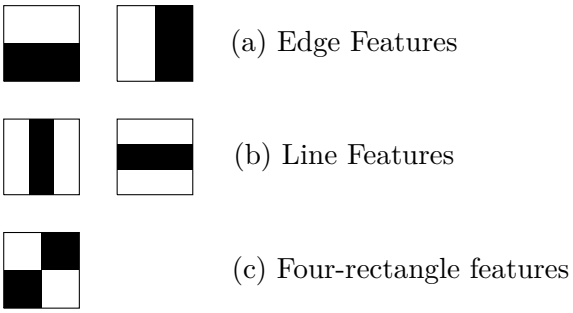


FIGURE 4.4: Haar features representation.

The computation for this calculus is huge (for example a 24x24 window has over 160000 features) but the solution can be found in the *Integral Image*. The integral image contains in position (x, y) of the image the sum of the pixels above, while to the left of (x, y):

$$ii(x, y) = \sum_{x' \leq x, y' \leq y} i(x', y')$$

$$s(x, y) = s(x, y - 1) + i(x, y)$$

$$ii(x, y) = ii(x - 1, y) + s(x, y)$$

where $ii(x, y)$ is the integral image and $i(x, y)$ is the original image, $s(x, y)$ is the cumulative row sum, $s(x, -1) = 0$ and $ii(-1, y) = 0$. This procedure accelerates the calculation and makes possible the integral image computation in one pass only over the original image. However most of the calculated features are useless; taken a particular feature, a rectangle with a blank and a dark area, it has relevance only in a specific zone of the image. An example can be seen in the figure 4.5. If the feature that describes the difference of light between the eyes and nose area is shifted to another point of the image it loses its meaning. Therefore

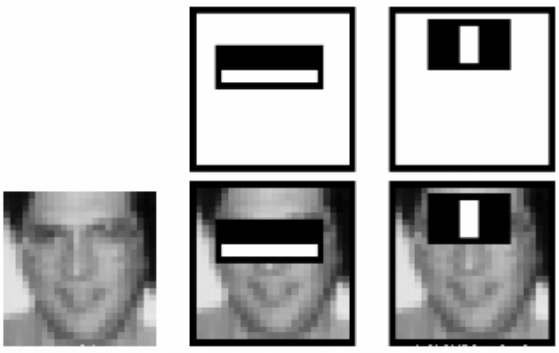


FIGURE 4.5: Haar features applied on face recognition [13].

the features than can be combined into an effective classifier are a very small number, so the goal is to find them out. The solution can be found using a tailored version of *AdaBoost*:

1. Every feature is applied on all the training images dataset;
2. The algorithm associates to each feature the best threshold that is able to classify into positive or negative the prediction;
3. The previous step produces errors or misclassifications than can be used to make a rank of the features and save only the best ones that can guarantee an accurate classification;
4. The procedure is repeated until the required accuracy or error rate is achieved.

The classifier obtained at the end is formed by all the "weak" classifiers that alone are not able to recognise the object in the images but together can reach high accuracy. This procedure may take long time also if the features have been reduced to the minimum number. The problem lies in the application of all the features to all images area. Referring to the figure 4.5 the area containing the face doesn't occupy all the image so remaining zone can be avoided during the previous procedure in order to waste less time. On this concept is based the *Cascade of Classifiers* (figure 4.6). The features are grouped into different steps of

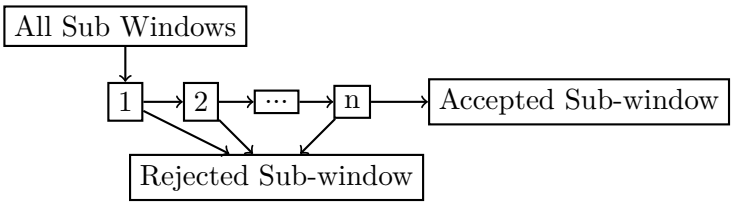


FIGURE 4.6: Haar Cascade.

classifiers so they can be applied one-by-one to the area of images. If the first group doesn't give the result expected that area will be discarded and all the remaining features will not be applied on it. Following this concept, only the area that will reach the last step will be the searched one. This method gives better results in terms of time without losing the previous accuracy.

4.1.2 Histogram of Oriented Gradients (HOG)

A great use of the image gradient vector can be seen into the HOG algorithm [14] in 2005. This method extracts, with good results in terms of final score and processing time, features out of the colors of pixels. The application of the HOG algorithm can be separated into five steps:

1. Preprocessing the image: all the images have to be equals in dimensions of pixels in order to gain higher and meaningful results;
2. Calculate the gradient into the two main directions with a kernel and then use it for the computation of magnitude and direction; the gradient removed a lot of non-essential information (e.g. constant coloured background), but highlighted outlines.
3. The image is divided into many 8x8 pixels cells where each magnitude is changed into a 9 element bucket of unsigned direction; in the case of a magnitude lays between two values of the bucket its magnitude is proportionally split into the two;

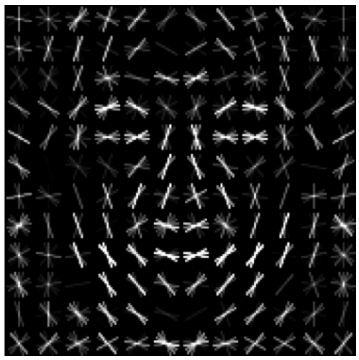


FIGURE 4.7: HOG applied on face recognition dataset.

4. A 2x2 cells block slide over the image where in each region is formed a one-dimensional vector of 36 values from the four histograms of four cells and then is normalized to have a unit weight; the final feature of the HOG is the vector formed by the concatenation of all the block vectors.

The main concept lies on the research of major gradient direction for each small group of pixels that shows the flow from light to dark across the selected image. An example of the resulted image can be seen in figure 4.7.

4.1.3 Regional CNN (R-CNN)

With the coming of deep learning, CNN became the most used architecture in object recognition field. The main concept explained before (§2.4.3) can be combined to a process called *Selective Search* [15] to reach higher velocity producing R-CNN [16]. This process instead of using directly the proposed RoIs starts to analyse each individual pixel and groups it into small congruent agglomerate and then combines the group that have the closest texture; the algorithm is able to group firstly the small combinations and then combine gradually the larger. The process is showed partially in figure 4.8 where the first row shows the merged area using selective search and the second RoIs computed.

FIGURE 4.8: Example of Selective Search [15].

The processed images are warped to the desired dimension and then fed into the CNN net. Hence the areas of images enclosed to the proposed bounding boxes go through a pre-trained network before the procedure of classification (figure 4.9).

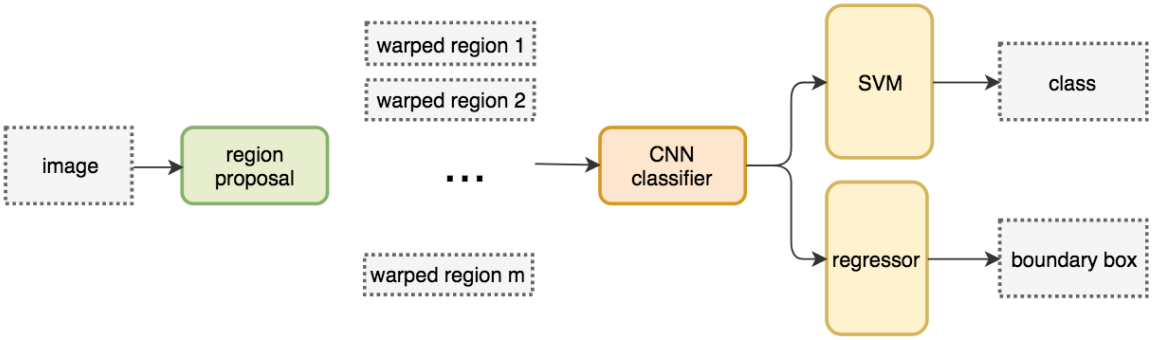


FIGURE 4.9: Procedure of R-CNN [15].

This technique leads to the necessity of a huge amount of time during the training of network due to all region proposals per image to classify; hence it is not compatible with a real time application as the processing of a single image is around 50s. In addition the selective search is not a machine learning algorithm but it is fixed so the production of incoherent candidates is possible.

4.1.4 Fast R-CNN

A great innovation was introduced with the evolution of R-CNN to the "fast" one [17]. Instead of using only the selective search as first step a CNN is adopted applied to feature extraction that are combined with the result of selective search. This method produce RoIs of feature map fed to a fully convoluted network that detects the object after being reduced to a fixed size with a pooling layer (figure 4.10).

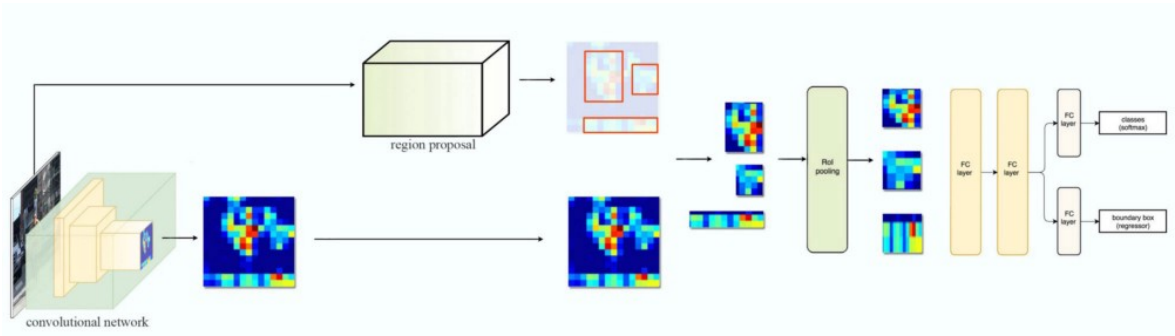


FIGURE 4.10: Procedure of Fast R-CNN.

This network is faster than the formers thanks to the avoidance of the feeding of 2000 regions proposal to the convolutional neural network every time but only once. Fast R-CNN is 10x faster than R-CNN in training and 150x faster in inferencing. Therefore this technique generates a sort of bottleneck during region proposals and this affects the overall performances.

4.1.5 Faster R-CNN

The usage of external algorithm like selective search is still time expensive; 87% of time is occupied by the creation of the region proposal. This last version of R-CNN, faster R-CNN

[18], replaces this step with an internal deep network and uses the RoIs derived directly from the feature maps (figure 4.11). The new network is named *Region Proposal Network* (RPN) and can reach time processing per image of about 10ms, so it is more efficient and faster. RPN predicts the chance to recognise an anchor, a box of different size that may include the desired object, on the background or foreground and refines it.

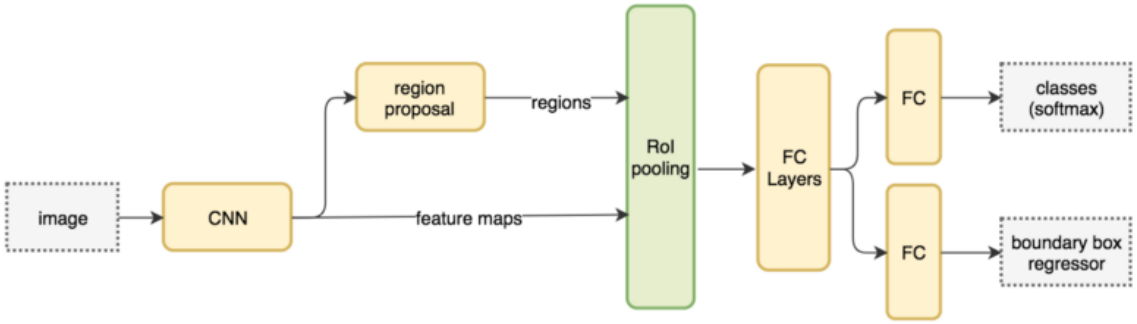


FIGURE 4.11: Procedure of Faster R-CNN.

The rest of the network is the same as the previous one. So Faster R-CNN is composed by a fully convolutional network that is able to propose regions and the previous Fast R-CNN structure to follow.

4.1.6 Single Shot multibox Detector (SSD)

SSD [19] has an architecture that permits to reach high value of mAP (§8.4.2) with an higher frame rate than R-CNN. Single shot means that the picture is processed only one time in a single forward pass over the network.

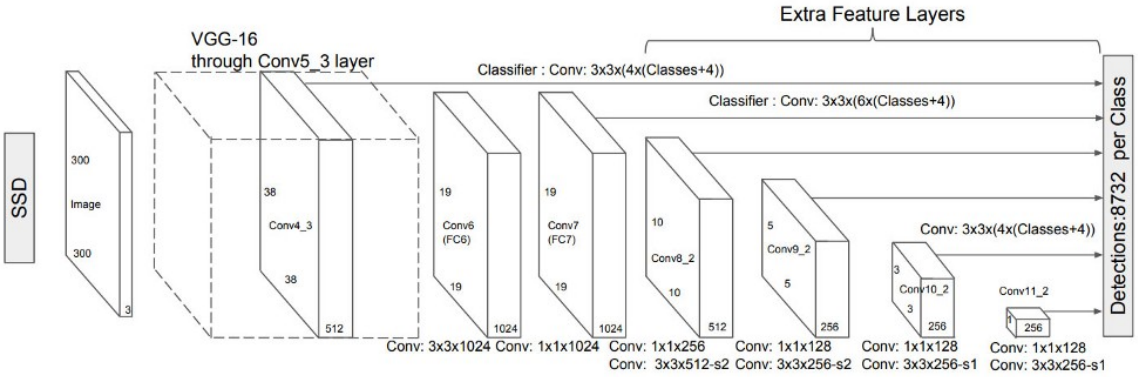


FIGURE 4.12: Structure of SSD.

SSD starts with a *VGG-16* architecture used as base network for its high performance on high quality image classification tasks; it is devoted to the extraction of feature maps. The network is no more a fully connected one but it is used a set of auxiliary convolutional layers that enables the extraction of features at multiple scales and progressively reduces the size of the input image to that of subsequent layers. Rather than calling anchors likes CNN, SSD uses priors a pre-computed with fixed size bounding boxes that try to match the positions of the original ground truth boxes. The priors are selected with the requirements of a minimum IoU (§8.4.1) of 0.5; hence they are submitted to the bounding box regression algorithm that can start from not random coordinates. Therefore this technique starts with the priors as predictions and goes closely to the ground truth bounding boxes.

Chapter 5

Y.O.L.O.

You Only Look Once (YOLO hereon) [20] is an object detection platform used for real-time image processing. It is a project born from the concept of SSD so it analyses the image only one time in a clever way. YOLO is faster than the previous methods and the predictions computed are made from one single network that can be trained end-to-end in order to improve the accuracy; furthermore YOLO demonstrates fewer false positives in background areas due to its direct access to all the images at once and the detection of one object per grid cell enforces the spacial diversity in making predictions.

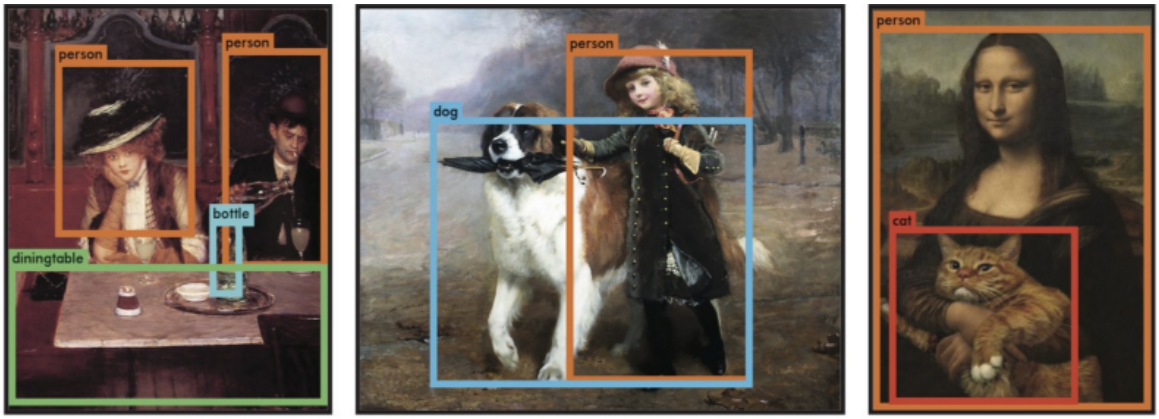
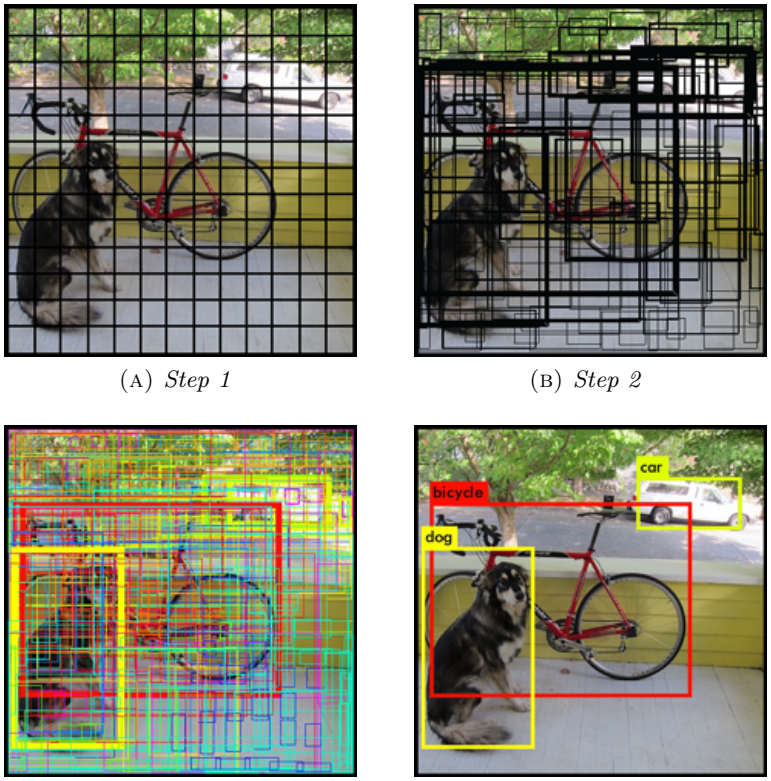


FIGURE 5.1: Example of application of YOLO.

5.1 How it works

As first step YOLO divides up the image into a grid of $S \times S$ cells (5.2a). Each of these cells has to predict B bounding boxes that describe the rectangle enclosing only one object. From all these operations there are also the confidence scores of the system that tells how certain it is to have an object enclosed into the predicted bounding box. The image 5.2b shows the possible boxes with also the confidence: higher confidence is indicated by a fatter drawn box. Hence for each bounding box a classifier predicts a class giving a probability distribution over all the possible ones; it depends on the dataset used for the training phase (PASCAL VOC, COCO, etc.). The confidence score and the class prediction are then combined to produce a result similar to figure 5.2c; this result highlights the differences between higher and lower accuracy in parallel to the different classes. The number of total bounding boxes is given by $S \times S \times B$ but it is easy to see that most of them have very low confidence scores; it is possible to save only from a certain level, as 30% is the default but it depends on the user's requests.

Cleaning the image from all the unsaved boxes produces the final result, an example in figure 5.2d.



(A) *Step 1* (B) *Step 2*

(C) *Step 3* (D) *Step 4*

FIGURE 5.2: YOLO procedure sequence.

5.2 Going deeper in details

YOLO for each cell:

- Predicts B boundary boxes with only one box confidence score individually;
- Classifies one object regardless how many number of boxes B are used;
- Predicts C conditional class probabilities based on the train dataset used.

Each boundary box has five elements describing it: x, y, w, h and the score. The four values that indicates the position and the dimension of the box are normalized by the image width and height in order to be in range $0 \div 1$. Hence the final prediction has a shape defined by:

$$(S, S, B \times 5 + C)$$

Example: dividing the image in 7 x 7 grid and using 2 bounding boxes with PASCAL VOC (20 classes) produce a tensor of:

$$(7, 7, 30)$$

24 convolutional layers followed by 2 fully connected layers compose the network. The convolutional layers are used to reduce the feature maps while the last one is used to output a tensor with the desired shape; this tensor is then flattened and used to generate $(S, S, B \times 5 + C)$ parameters.

Layer	Kernel	Stride	Output
Input			(416, 416, 3)
Convolution	3x3	1	(416, 416, 16)
MaxPooling	2x2	2	(208, 208, 16)
Convolution	3x3	1	(208, 208, 32)
MaxPooling	2x2	2	(104, 104, 32)
Convolution	3x3	1	(104, 104, 64)
MaxPooling	2x2	2	(52, 52, 64)
Convolution	3x3	1	(52, 52, 128)
MaxPooling	2x2	2	(26, 26, 128)
Convolution	3x3	1	(26, 26, 256)
MaxPooling	2x2	2	(13, 13, 256)
Convolution	3x3	1	(13, 13, 512)
MaxPooling	2x2	1	(13, 13, 512)
Convolution	3x3	1	(13, 13, 1024)
Convolution	3x3	1	(13, 13, 1024)
Convolution	1x1	1	(13, 13, 125)

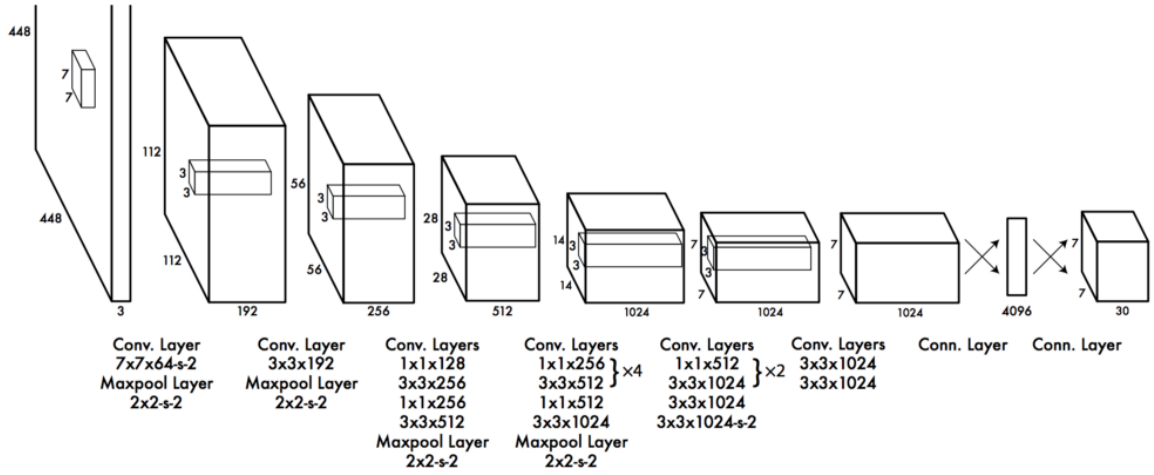


FIGURE 5.3: Structure of YOLO.

5.2.1 Loss Function

As mentioned before YOLO has to predict multiple bounding boxes for each grid cell hence there can be multiple losses for the true positive. The purpose is the selection of only one of them and make it responsible of that object recognition in that cell. Solution is found considering the highest IoU with the ground truth. This method also leads to specialization among the bounding box predictions: certain sizes and aspect ratios became easier to recognise. When only one for each cell prediction is selected the system can compute the loss function by a composition of three factors:

- Classification loss;
- Localization loss;
- Confidence loss.

Classification Loss

If there is an object detected this factor is given at each cell by the squarred error of the class conditional probabilities for any class:

$$\sum_{i=0}^{S^2} \mathbb{1}_i^{obj} \sum_{c \in classes} (p_i(c) - \hat{p}_i(c))^2$$

with $\mathbb{1}_i^{obj}$ equal to 1 if in the cell i it is present an object, otherwise is 0 and $\hat{p}_i(c)$ is the conditional class probability for class c in cell i.

Localization Loss

This parameter measures the errors considering the predicted boundary box locations and sizes only for the box that are detecting an object:

$$\lambda_{coord} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{obj} [(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2] + \lambda_{coord} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{obj} [(\sqrt{w_i} - \sqrt{\hat{w}_i})^2 + (\sqrt{h_i} - \sqrt{\hat{h}_i})^2]$$

with $\mathbb{1}_{ij}^{obj}$ equal to 1 if the j-th boundary box in cell i is the selected one to recognise the object, otherwise is 0 and λ_{coord} is used to increase the weight for the loss in the boundary box coordinates to normalize the error between large and small boxes.

Confidence Loss

This factor is computed in different ways if the object is detected or not:

$$\begin{cases} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{obj} (C_i - \hat{C}_i)^2 & \text{if detected} \\ \lambda_{noobj} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{noobj} (C_i - \hat{C}_i)^2 & \text{if not detected} \end{cases}$$

with $\mathbb{1}_{ij}^{obj}$ equal to 1 if the j-th boundary box in cell i is the selected one to recognise the object, otherwise is 0 ($\mathbb{1}_{ij}^{obj}$ is its complement), $\hat{C}_i$ is the box confidence score of the box j in the cell i and λ_{noobj} decreases the loss when there is the detection of the background.

5.2.2 Non-Maximal Suppression

It is possible that YOLO make a doubles detection of the same object. It is a heuristic result that can be fixed with the appliance of non-maximal suppression. This method removes the duplicate with lower confidence and adds approximately 3% to mAP. The procedure is not standardised; one example can be:

1. Sorting all the prediction using the confidence scores;
2. Starting from the top scores ignoring any prediction with lower IoU with the same class;
3. Repeating the step 2 until all predictions are checked.

5.3 YOLOv2

5.3.1 Accuracy Improvements

The performance of YOLO has been improved in terms of accuracy and processing time with the second version, YOLOv2 [21]. This huge update is given by the following modifications.

	YOLO							YOLOv2
batch norm?		✓	✓	✓	✓	✓	✓	✓	✓
hi-re classifier?			✓	✓	✓	✓	✓	✓	✓
convolutional?				✓	✓	✓	✓	✓	✓
anchor boxes?				✓	✓				
new network?					✓	✓	✓	✓	✓
dimension priors?						✓	✓	✓	✓
location prediction?						✓	✓	✓	✓
passthrough?							✓	✓	✓
multi-scale?								✓	✓
hi-res detector?									✓
Pascal VOC (2007) - mAP	63.4	65.8	69.5	69.2	69.6	74.4	75.4	76.8	78.6

Batch normalisation

The idea behind batch normalisation is that neural network layers work properly when the input data are clean; in fact ideally those data have average value of 0 and a variance very low. This technique permits to do a kind of feature scaling for the data that are between layers stopping the deterioration during the flow through the network. An example is shown in figure 5.4.

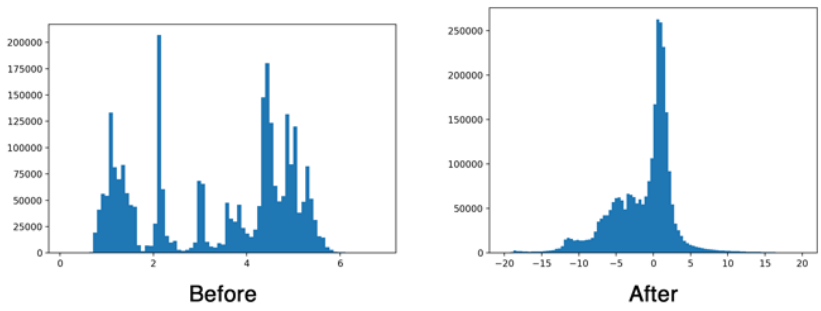


FIGURE 5.4: Example of batch normalisation.

Usually it is applied between the convolutional layer and the activation function and removes the need for dropouts and increase mAP up to 2%.

High-resolution classifier

The training of YOLO is composed by the first training of the classifier and then the end-to-end training of the fully connected layers for object detection. YOLOv1 trains the classifier with 224x224 images and the classifier for object detection with 448x448 pictures. Instead YOLOv2 has as input images for the classifier the same 224x224 images but there is the presence of another retrain of the classifier with the 448x448 images and few epochs. This modification makes the training of the object detector easier and improves the result of mAP by 4%.

Convolutional Layer with Anchor Boxes

As mentioned before, YOLO uses priors for the boundary boxes. Initially these boxes are selected randomly and may work well for some objects and worse for others (figure 5.5) making the gradients unstable with step changes. In fact at first time there is a contradictory response from the system about what shapes use the most.

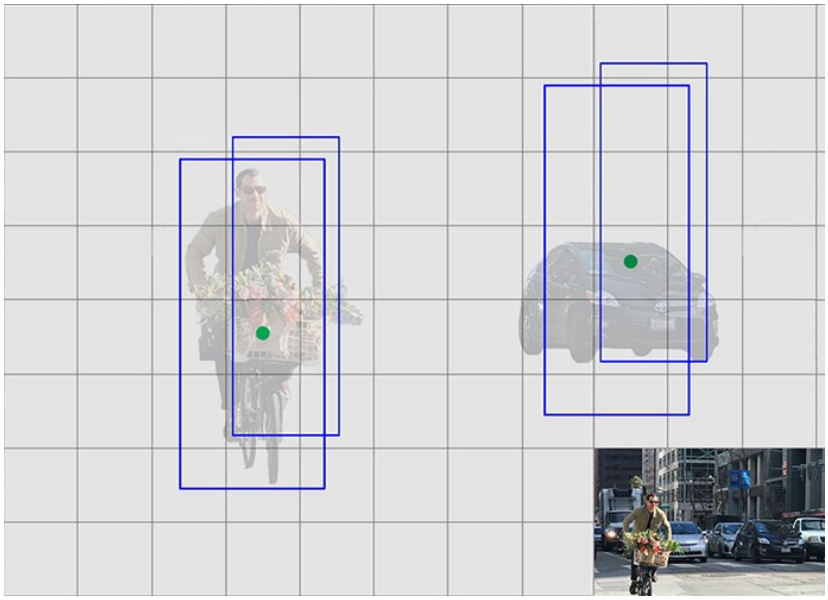


FIGURE 5.5: The shape works well on the object on the left (a man on bicycle) but it is not proper for the object on the right (a car).

In real life application it is possible to notice that each object has a shape more or less constant. This concept can be used in this way: if the priors that include a specific class are constrained between offset values the diversity of the prediction will remain unchanged but after each epoch the prediction will be more focused on a shape increasingly specific. The network has to be modified with the following adjustments:

- The final fully connected layers used for the prediction of boundary boxes are removed;
- The predictions are done at boundary box level and not anymore at cells level;
- Change the image input size from 448x448 to 416x416 to have an odd number of cells in the grid; it makes simpler to determine the belonging of an object to a centre cell (figure 5.6);

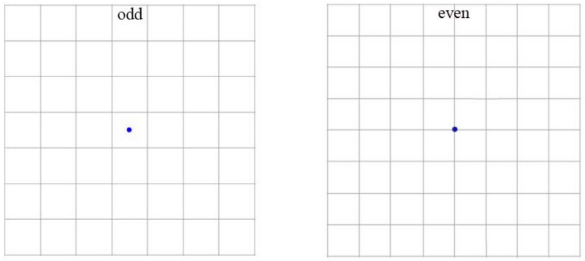


FIGURE 5.6: YOLO grid modification.

- Changing and remove some layers of the network to achieve an output of 13x13.

These changings have negative impact on mAP making it lower of 0.3 but give a better result on recall that improves by 7%; also the accuracy is decreased a little but there is an augment of ground truth object detected.

Dimension Clusters

Noticing that the priors have similarities and often are few for each dataset, instead of choosing them by hand it is possible to use a *K-means* clustering on the training dataset and obtain better results. K-means is able to find the top-K boundary boxes that cover the data and they are found considering the centroids (figure 5.7). K-means requires the number K of

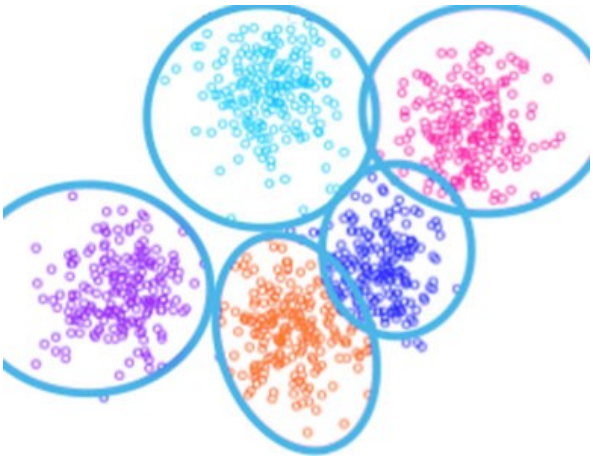


FIGURE 5.7: Five typologies of priors are found with K-means clustering.

clusters so depending on the dataset used and its complexity it has to be tuned. The criteria for the choice is the IoU (figure 5.8) Considering this example of application it is possible

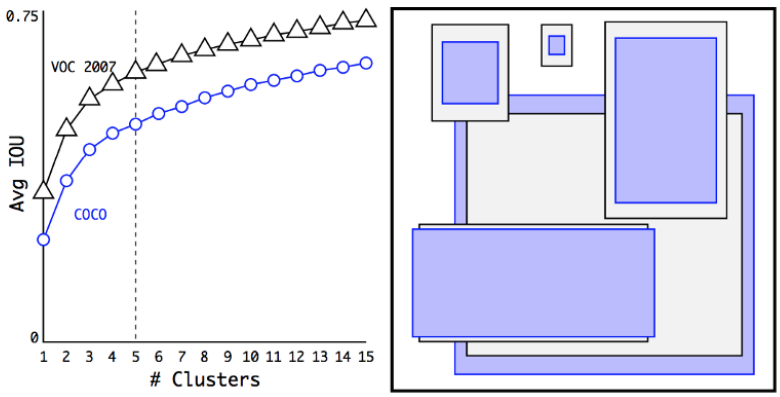


FIGURE 5.8: Clustering box dimensions on VOC and COCO. The right image shows the relative centroids found for VOC and COCO.

to notice a growing of the average IoU of 0.2 but the priors become more "intelligent" and reflect better the objects detected.

Direct Location Prediction

During the first iteration the system suffers instability of the model. It is due to the unconstrained anchor boxes that can end up at any point in the image. The random initialisation leads to a very long convergence time for the system stabilisation. Instead of predicting offset

like the previous version, YOLOv2 predicts location coordinates relative to the location of the grid cell. A logistic activation is able to constrain the network prediction to fall in the specified range (figure 5.9). This procedure is applied using the following formulas:

$$
\begin{aligned}
b_x &= \sigma(t_x) + c_x \\
b_y &= \sigma(t_y) + c_y \\
b_w &= p_w e^{t_w} \\
b_h &= p_h e^{t_h}
\end{aligned}
$$

$$
P_r(object) * IoU(b, object) = \sigma(t_o)
$$

Where:

- t_x, t_y, t_w, t_h are the predicted values by YOLOv2;
- b_x, b_y, b_w, b_h indicates the predicted boundary box;
- (c_x, c_y) is the point at the top-left corner of the prior;
- c_w, c_h are respectively the width and height of the prior;
- $\sigma(t_o)$ is the confidence score associated to the box.

Note that the values of c are normalized by the size of the image.

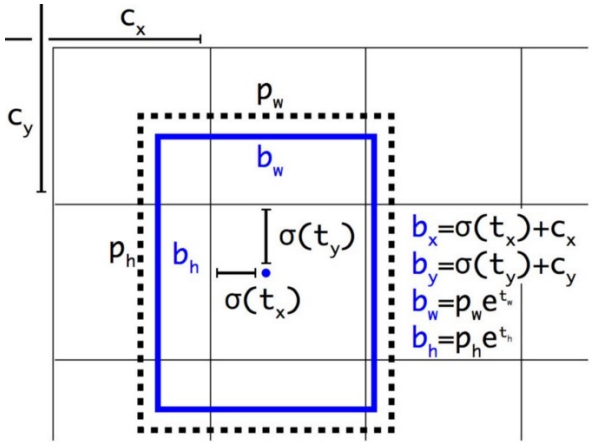


FIGURE 5.9: Visualization of YOLOv2 boxes algorithm. The dotted box is the prior and the blue box is the predicted boundary.

This changing improves the mAP up to 5%.

Fine-Grained Features

YOLO predicts with a 13x13 feature map grid. This is sufficient and works well with medium dimension objects into the image instead small objects give some problems during the recognition. Algorithms as Faster R-CNN and SSD run the proposal networks over various feature maps to get different dimensions object using a range of resolutions. As we said YOLO runs only once so to satisfy this requirement the solution is found adding a passthrough layer that modifies the feature map from the 13x13 dimension into 26x26 resolution. The resulted expanded feature map passes into the detector and gives a 1% performance increase.

Multi-Scale Training

The input images size for YOLO was imposed to be 448x448. This new model changes this requirements with the usage of the priors into 416x416 but since the architecture is formed only by convolutional and pooling layers it is possible to change the sizes on the fly. This gives the possibility to create a version more robust and stable running images of different sizes. Instead of fixing the image input size, the network change the accepted dimension every few iterations. The entire model down-samples by a factor of 32 so it can be possible to train the architecture over images of sizes from 320x320 to 608x608 (320, 352,384,...,608). Every 10 batches the network randomly chooses the dimensions and continue the training. This methodology forces the network to learn to predict well across different input images dimensions hence at various resolutions. With this changing the system can be modelled for lower resolution applications and gives mAP similar to Fast R-CNN but running at more than 90FPS (highest result at the moment for object detection); while for high resolution the medium mAP is about 78.6 (on PascalVoc).

Accuracy Comparison for Different Detectors

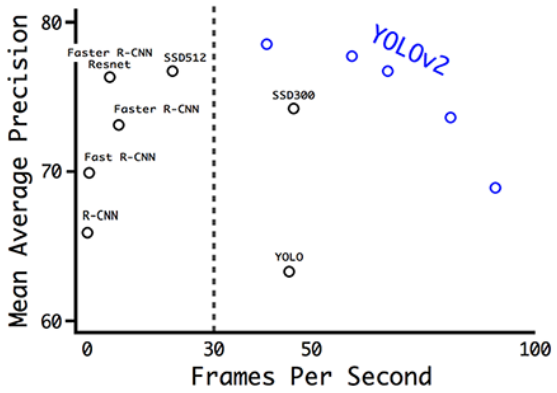


FIGURE 5.10: Accuracy comparison for different detectors.

5.3.2 Speed Improvements

A first changing is about the feature extractor *VGG16*: it tooks 30.69 billion floating point operations for the analysis of a single image (224x224). *GoogLeNet* with a custom architecture processes 8.52 billion operations so it can replace the previous network. The processing time has a huge improvement but the accuracy drops from 90.0% to 88.0%.

Also the classification model is changed. The new classificator called *Darknet-19* is similar to the previous and uses mostly 3x3 filters but doubles the number of channels after every pooling step. Furthermore there is the adding of the global average pooling to make predictions and 1x1 filters that compress the feature map representation between the 3x3 convolutional layers. The full new architecture is showed below:

Layer	Filters	Stride	Output
Convolution	32	3x3	224x224
MaxPooling		2x2/2	112x112
Convolution	64	3x3	112x112
MaxPooling		2x2/2	56x56
Convolution	128	3x3	56x56
Convolution	64	1x1	56x56
Convolution	128	3x3	56x56
MaxPooling		2x2/2	28x28
Convolution	256	3x3	28x28
Convolution	128	1x1	28x28
Convolution	256	3x3	28x28
MaxPooling		2x2/2	14x14
Convolution	512	3x3	14x14
Convolution	256	1x1	14x14
Convolution	512	3x3	14x14
Convolution	256	1x1	14x14
Convolution	512	3x3	14x14
MaxPooling		2x2/2	7x7
Convolution	1024	3x3	7x7
Convolution	512	1x1	7x7
Convolution	1024	3x3	7x7
Convolution	512	1x1	7x7
Convolution	1024	3x3	7x7

The previous cross-out section, *i.e.* the last convolutional layers, is replaced by three 3x3 layers outputting each 1024 channels followed by a final 1x1 convolutional layer that converts the 7x7x1024 output into 7x7x125. Darknet-19 requires only 5.58 billion operations to process an image and can achieve 72.9% top-1 and 91.2% top-5 accuracy on ImageNet.
YOLOv2 is trained on *ImageNet* with 1000 classes and for 160 epochs (in this thesis work has been used COCO). The data for the training are:

- Stochastic gradient descent with starting learning rate of 0.1;
- Polynomial rate decay with a power of 4;
- Weight decay of 0.0005;
- Momentum of 0.9 .

5.3.3 Hierarchical Clasification

Datasets for object detection have far fewer class categories than those for classification. YOLOv2 uses a method that mix images from both classifications and detection datasets during the training. The network is trained end-to-end with object detection samples while the classification loss is backpropagated to train the classifier path. The main problem lies on how to merge different datasets that have not mutual exclusive classes. This problem is solved by the usage of hierarchical classification (figure 5.11).
This new dataset structure is able to link an image recognised as "dog" from *COCO* to "Norfolk terrier" recognise from *ImageNet*. In this way it composes a parent/child structure used also for the final classification result. In fact if the confidence of the children is too low,

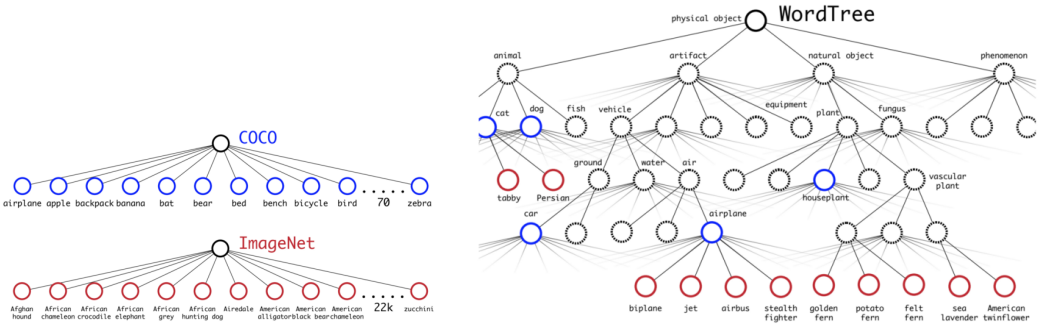


FIGURE 5.11: Structure of COCO, ImageNet and WordTree dataset.

the system gives as output the confidence it has about the parent.Example: if "biplane" has low confidence the system answer with "airplane".
YOLO9000 pushes the usage of WordTree to the limits merging *COCO* detection dataset with the top 9000 classes from the full *ImageNet* release. Using a joint type of training this architecture is able to find object in images using the detection data that came from *COCO* and it is also capable of classification of a large variety of these object using *ImageNet*.

5.4 YOLOv3

YOLOv3 [22] is the latest evolution of this architecture. From the point of view of the class prediction it introduces a huge modification. Most of the current classifiers assume that the output labels are mutually exclusive so the final softmax function that converts scores into probabilities has maximum result 1. YOLOv3 cancelled this concept and became a multi-label classificator. As an example there can be as output simultaneously "vehicle" and "car". So the softmax function cannot be anymore limited to 1 hence it is replaced with an independent logistic regression classifiers that calculate the likeliness of the input belonging to a specific label. Therefore the mean square error used in classification loss is replaced with the cross-entropy loss for each label. Also the cost function changed. It is associated an objectness score equal to 1 to the bounding box prior that overlaps a ground truth object the most; all the other priors that satisfy the requirement of a predefined threshold for the IoU (0.5 as default) the cost function is 0. So each ground truth object is associated to only one boundary box prior and if any of the bounding box satisfies the requirements it incurs in no classification.

5.4.1 Prediction

YOLOv3 generates 3 predictions per location and each of them has a boundary box, an objectness and 80 class scores. This concept is similar to *Feature Pyramid Networks (FPN)*, in fact the sequence of the operations is:

1. Making a prediction in the last feature map layer;
2. Considering 2 layers back and up-samples this layer by 2;
3. Searching for the feature map with higher resolution in the considered layer and merge it with the up-sampled feature map using a technique named element-wise addition;
4. Applying a convolutional filter on the merged map in order to generate the second set of prediction;

5. Replying from step 2 to have a resulted feature map layer with a good high-level structure, semantic information, and good resolution spatial information on object locations.

The priors for YOLOv3 are generated using K-Means clustering and considering *COCO* dataset. The number of preselected cluster is 9: (10x13), (16x30), (33xx3); (30x61), (62x45), (59x119); (116x90), (156x198), (373x326). They are grouped and assigned to a specific feature map to better detect the object.

5.4.2 Feature Extraction

Also the architecture of Darknet-19 experiences an improvement becoming Darknet-53, so much more layers are added (figure 5.12). This network achieves the highest measured floating point operation per second so its application requires for now the utilization of GPU hardware.

	Type	Filters	Size	Output
	Convolutional	32	3 × 3	256 × 256
	Convolutional	64	3 × 3 / 2	128 × 128
1×	Convolutional	32	1 × 1	
	Convolutional	64	3 × 3	
	Residual			128 × 128
	Convolutional	128	3 × 3 / 2	64 × 64
2×	Convolutional	64	1 × 1	
	Convolutional	128	3 × 3	
	Residual			64 × 64
	Convolutional	256	3 × 3 / 2	32 × 32
8×	Convolutional	128	1 × 1	
	Convolutional	256	3 × 3	
	Residual			32 × 32
	Convolutional	512	3 × 3 / 2	16 × 16
8×	Convolutional	256	1 × 1	
	Convolutional	512	3 × 3	
	Residual			16 × 16
	Convolutional	1024	3 × 3 / 2	8 × 8
4×	Convolutional	512	1 × 1	
	Convolutional	1024	3 × 3	
	Residual			8 × 8
	Avgpool		Global	
	Connected		1000	
	Softmax			

FIGURE 5.12: Darknet-53 structure.

5.4.3 Performance

YOLOv3 is the fastest network tested on *COCO* and shows improvement in detecting small objects.

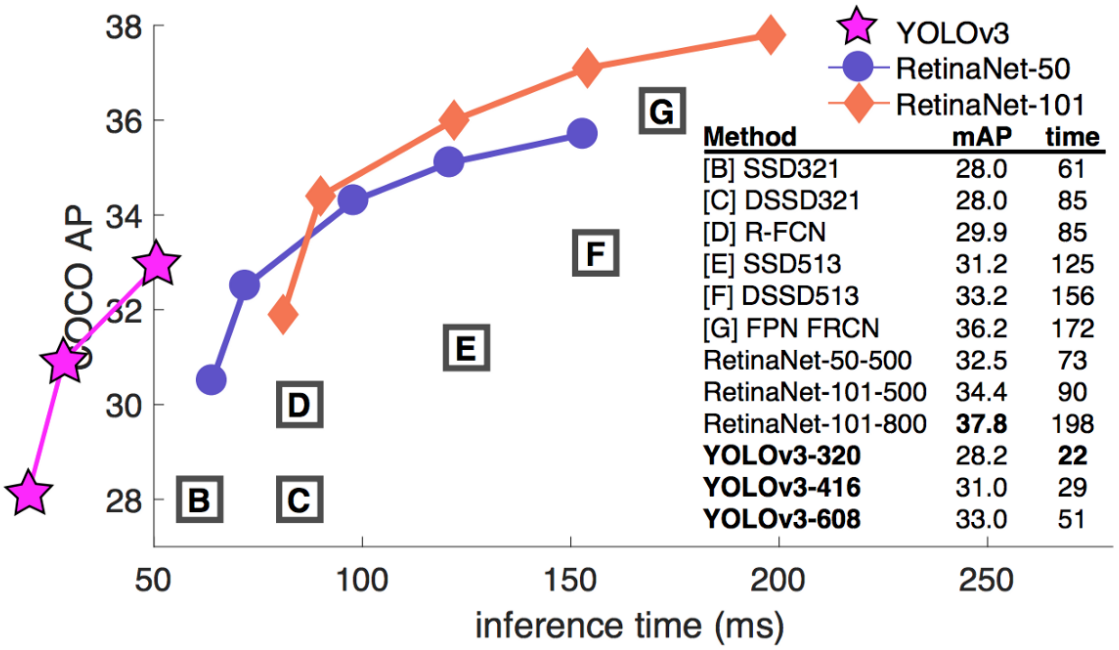


FIGURE 5.13: Performance comparison.

Chapter 6

Mask R-CNN

Object classification projects are able to find into an image a specific object among the trained classes; its successive evolution step, object detection, finds and locates that object using a box that encloses the shape recognised. Mask R-CNN [23] is the last achieved result for computer vision. It introduces the concept of *Image Segmentation*: the object found are no more only enclosed in a box but there is the grouping of the pixels that belong to the same object. The procedure followed by this new algorithm is (figure 6.1):

1. Classification: this image contains a balloon;
2. Object Detection: this image contains seven balloons and the locations are defined;
3. Semantic Segmentation: the seven balloons are formed by this group of pixels;
4. Instance Segmentation: the seven balloons at these seven locations have these seven groups of pixels that define each balloon.

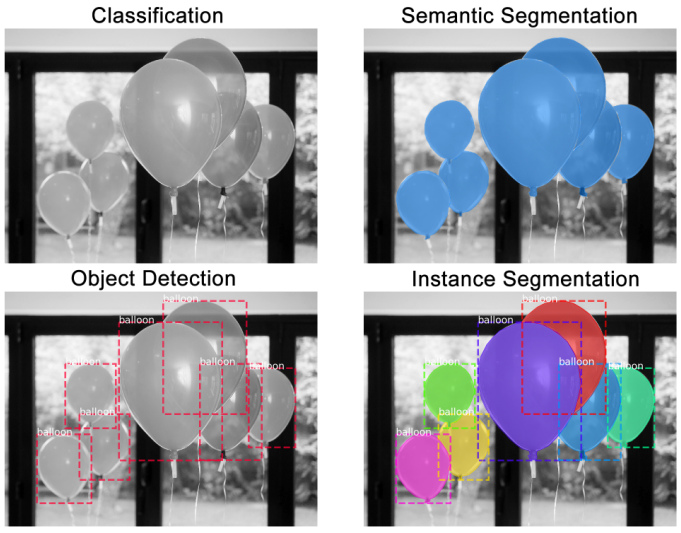


FIGURE 6.1: Example instance segmentation [24].

6.1 How It Works

Mask R-CNN, as the predecessors, is a two stage framework: the first stage is devoted to the scanning of the images and to the generation of the region proposals, *i.e.* areas that may contain an object; the second stage has to classify the region proposals and to make the

new masks for the detected objects. The architecture is slightly modified with respect to the Faster R-CNN; a new section formed by two convolutional networks is added in parallel to the last fully connected network and the masks are created in it.

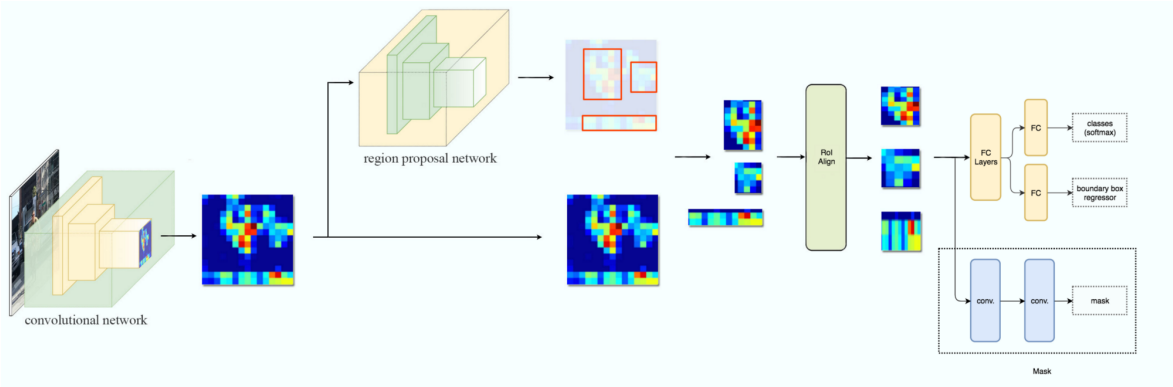


FIGURE 6.2: Mask R-CNN flowchart.

Mask R-CNN starts as usual to process the images with a feature extractor, typically *ResNet50* or *ResNet101*; these layers are first used to detect low level information as edges and corners and then reach a level of detection capable of recognition of high level features (dog, house, pen). The images are in fact converted from the RGB initial form to a feature map of 32x32x2048 that is used as input for the next stages. This backbone is improved by the utilization of the FPN network (figure 6.3) that makes available to all the features map at each level the access to lower and higher level features.

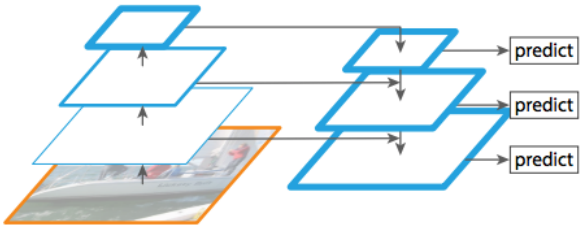


FIGURE 6.3: Features Pyramid Network.

The rest of the network is similar to Faster R-CNN (§4.1.5) until the mask branch.

Mask Representation

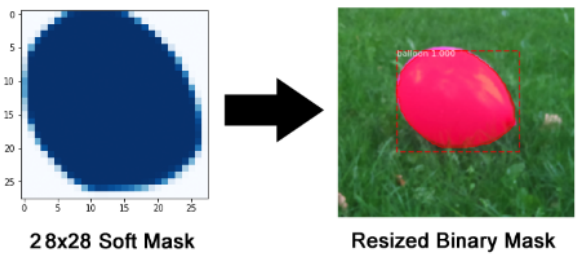


FIGURE 6.4: Mask example.

The mask branch is a convolutional network that uses the positive selected regions from RoIs classifier and creates a shape of the object detected. So the mask is able to encode an input of the object spatial layout so the information extracted can be addressed by the pixel-to-pixel correspondence given by the convolutions. This network has to identify for each RoI an

$m \times m$ mask using a fully convolutional network. With this method each layer in the mask branch can maintain the explicit object spatial layout avoiding the warping into a vector representation with no spatial dimensions. Therefore this network is not complicated as it seems and in fact requires few parameters to work. It is easy to see that the pixel-to-pixel behaviour needs a perfect correlation between the predicted mask and the RoIs of the original image in order to give significant results. This problem is faced by the *RoIAlign*. This layer substitutes the RoIPool and aligns the extracted features with the input.

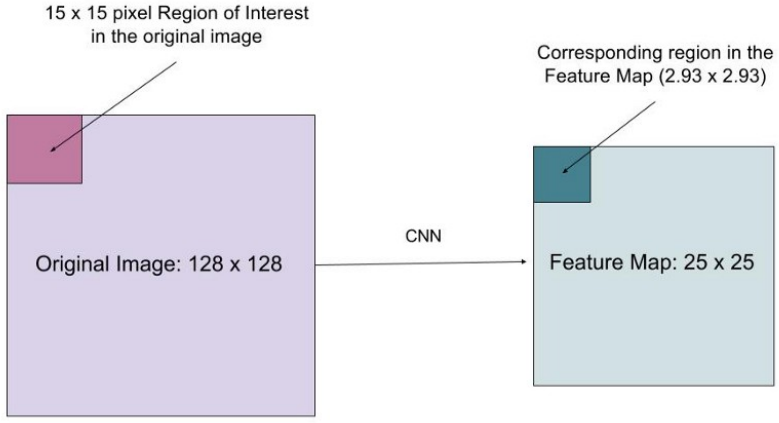
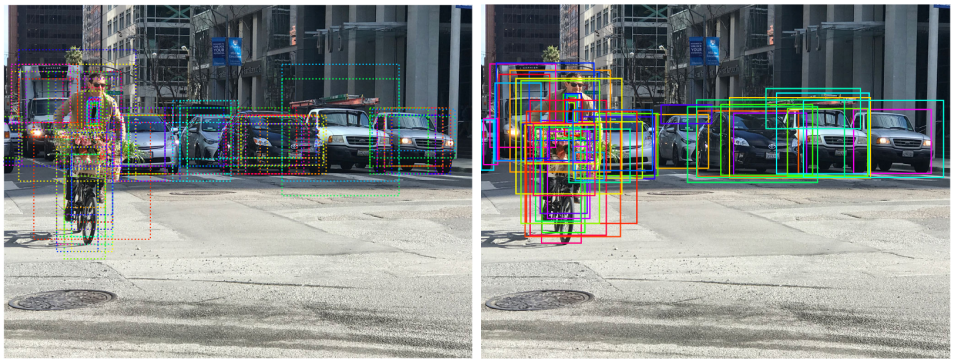


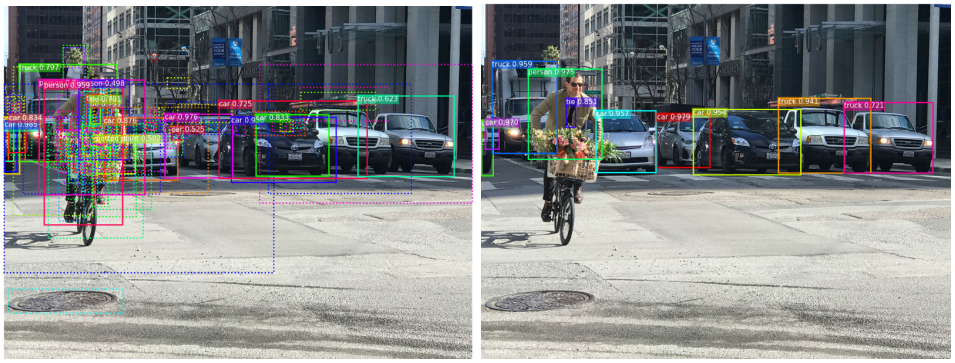
FIGURE 6.5: Original image on left and feature map dimension on right.

The basic idea is the following. Hypothesising that the CNN transforms the original 128x128 image into a feature map of 25x25, the RoI of 15x15 on the upper left is the considered one. This region is warped into the feature map by a ratio of 25/128 pixels so the RoI becomes an area of $15 * 25/128 \simeq 2.93 \rightarrow$ 2.93x2.93 pixels. Old RoIPool rounds this value to 2x2 pixels causing a misalignment; instead RoIAlign avoids the rounding and use a bilinear interpolation. This gives a precise idea of what would be at pixel 2.93.

6.2 Visualisation of the Steps



(A) RoIs before refinement. (B) RoIs after refinement.



(C) Classification with RoIs refined. (D) Top boundary box predictions.

(E) Application of the masks.

FIGURE 6.6: Visualization of the steps during the processing of an image with Mask R-CNN.

Chapter 7

Embedded System

One of the main problems for neural networks is the hardware capacity for the training and deployment. The training phase needs a powerful hardware that can take from hours to days for the generation of all the weights of the system. Instead the deployment of the net can be sometimes done in a lightweight way. One of the purpose of this thesis was the application and the usage of object detection on embedded systems. This hardware can be useful for applications on robots, rover and drones due to its minimum requirements of power consumption and its weight. In fact usually with embedded systems one refers to a specific hardware, often with small size, designed to a specific usage. A set of possible hardware was analysed in search of the best one for this type of application and the final system was composed of a *RaspberryPi* with a *Intel Movidius Neural Compute Stick* (Movidius NCS hereon).

7.1 Raspberry Pi

The RaspberryPi is a single-board computer that mounts a kernel Linux-based operating systems, in particular the most used is a dedicated one called *Raspbian*.

FIGURE 7.1: RaspberryPi 3.

Although its little dimensions (85,60mm x 56mm) this computer has powerful hardware. The entire project is based on a Broadcom system-on-a-chip with an ARM processor and a VideoCore GPU. 512 Megabyte of RAM and logic ports complete this powerful system that can be used in infinite ways. Moreover the cost of the board helped the widespread distribution and application making it the most used hardware for simple projects.
Unfortunately, even if this board is extremely powerful for its dimensions, it is not sufficient for the operations needed by the computation of a neural network.

7.2 Movidius NCS

This little device is one of the most recent products of Intel concerning machine learning. It is defined as tiny fanless deep learning USB drive designed to learn A.I. programming. The

Figure 7.2: Movidius Neural Compute Stick.

company added this product under the class of *Vision Processing Unit* (VPU), similar to the classic GPU but for embedded purposes. This little device supports the calculation needed by a convolutional neural network so it can be used for many deep learning applications. Moreover, the stick has lower power consumption so it is perfectly applicable on embedded systems. The VPU includes 4Gbits of LPDDR3 DRAM, imaging and vision accelerators, and an array of 12 VLIW vector processors called *SHAVE* processors. These processors are used to accelerate neural networks by running parts of the neural networks in parallel.

7.2.1 How It Works

The VPU also has a SPARC microprocessor core that runs custom firmware. When the NCS is first plugged in, there is no firmware loaded onto it. The VPU boots from the internal ROM and connects to the host machine as a USB 2.0 device. Applications executing on the host machine communicate to the VPU SOC using the Neural Compute API (NCAPI). When the NCAPI initializes and opens a device, the firmware from the Neural Compute SDK (NCSDK) is loaded onto the NCS. At this time, the NCS resets and reconnects to the host machine as either a USB 2.0 or USB 3.0 device (depending on the host type). It is now ready to accept the neural network graph files and instructions to execute inferences.
A graph file (a particular file produced from the neural network) is loaded into the DRAM attached to the VPU via the NCAPI. A LEON processor coordinates the process receiving the graph file and images for inference via the USB connection. It also parses the graph file and schedules kernels to the SHAVE neural compute accelerator engines. In addition, the LEON processor also takes care of monitoring die temperature and throttling processing on high temperature alerts. The output of the neural network and associated statistics are sent back to the host machine via the USB connection and are received by the host application via the NCAPI.

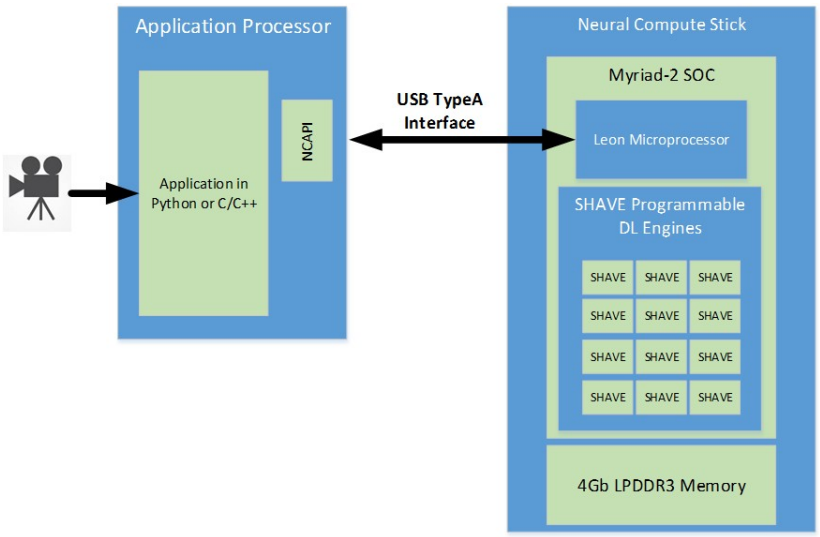


FIGURE 7.3: Workflow from acquisition to processing on NCS.

7.3 Object Detection Application

Combining the previous described hardware it can be possible the deployment of a simple detection algorithm like *Tiny-YOLO*. This is a reduced version of YOLO, formed by nine convolutional layers and two last fully connected layers. Obviously the performance of this type of network are not comparable from the full version of the platform but the easy composition of the net makes them light and fast giving the possibility of a deployment on embedded systems. A simple system has been created for the test phase of the Tiny-YOLO application; the used hardware was:

- RaspberryPi 3;
- Movidius NCS;
- WebCam;
- Powerbank.

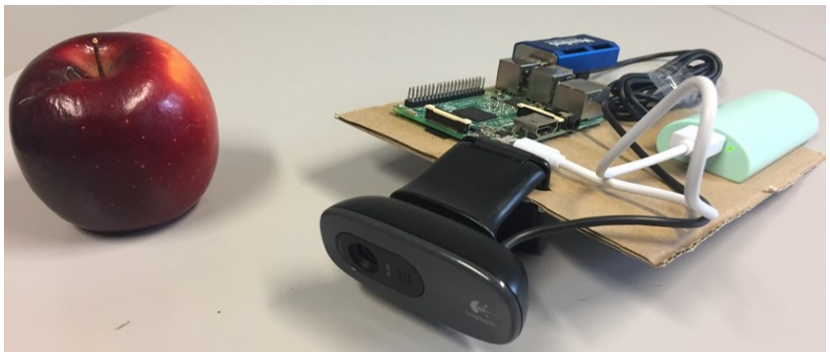


FIGURE 7.4: Hardware system used for the Tiny-YOLO deployment.

Like other applications, also in this case the architecture of Tiny-YOLO has been tested as the original provided and with a retrained model specific for the detection of apples. The full python code is showed in appendix A.

This simple hardware structure has been developed to demonstrate that, also from a low performing system, the acquisition and especially the processing of the images over a neural network can be possible.

7.3.1 Tiny-Y.O.L.O.

With the deployment of original Tiny-YOLO and the help of NCS compute stick the system was able to detect objects with a m-AP around **31.47%** and a FramePerSecond (FPS) around **3FPS**. It is not an incredible result but considering the used hardware and the small network it can be comparable with older nets used.

7.3.2 Tiny-Y.O.L.O (Apple)

This version of the platform has been created by a retrain of the original Tiny-YOLO for the only detection of apples. The retraining phase has been done on a simple laptop without GPU support that has taken around 8 hours to finish. The train dataset was composed by more than 300 images of apples of different species and colours. The result from the retraining was satisfying: the mAP increased to **49.96%** and the processing was done at around **4FPS**.

Chapter 8

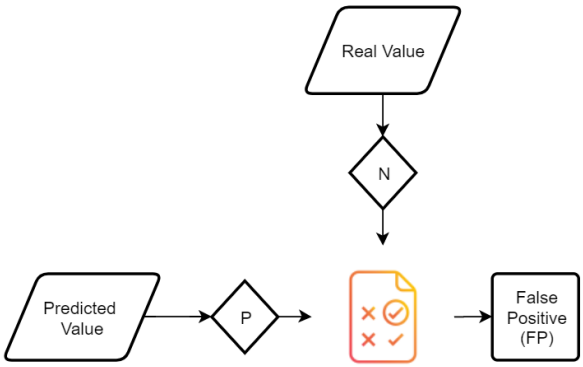
Metrics

Far from now different neural network structures and how they works have been described. Nowadays there are two factions of researchers devoted to machine learning: one that assumes theories about the uselessness of knowing the real "how" a neural network works and the second strongly affirms the necessity of understanding each decision, each parameter and each weight in order to have the total control of the net. It was not underlined until now but effectively the structures of neural networks used for deep learning are so complex that is almost impossible to understand why each node does what it does. Obviously the researchers know the general algorithm functionality but often the weights used for the nodes are not so clear. The point in common between these two groups is the necessity of a final evaluation of the results obtained. It was explained how a neural network learns from the errors but not how the final net with the updated weights and biases is evaluated. During the evolution of this computer branch a lot of methods were developed and some of the most relevant are listed below. Before the listing it is necessary to do a brief reminder. It has been introduced the general division of data when using machine learning algorithms: in fact all the available datasets are split into train, validation and test. All the listed methods use the test group of data to make the evaluation.

8.1 Confusion Matrix

When the object detection was still a future goal the application of machine learning concerned most areas of data analysis, an application highly used also in our time. The first method for the evaluation of results was the usage of a *confusion matrix*. During the evaluation the data into test dataset are given to the network that evaluates them and gives the result. These are the *Predicted* values that goes into the confusion matrix. Test data are those previously classified from the programmer and used as in this case for the confusion matrix. The matrix is composed as explained in the scheme.

		Predicted	
		P	N
Real	P	True Positive (TP)	False Negatives (FN)
	N	False Positive (FP)	True Negatives (TN)



This was an example of a two choices (True, False) neural network. So summarising is possible to say as an example that if the predicted value is True for the system and the real one is

also True the evaluated data give a True Positive in the confusion matrix evaluation; instead if the real value is false the evaluated data give a False Positive. It is easy to note that the system can be considered trustworthy if the values on the diagonal of the matrix are much higher than the others. In fact the sum of numbers on the diagonal is the correct predicted value. In order to have a consistent network this number has to be usually at least the 70% of the total. The explained example refers to a system with only one boolean variable (True or False). The same considerations can be applied to a network with multiple output variables; an example is showed in figure 8.1.

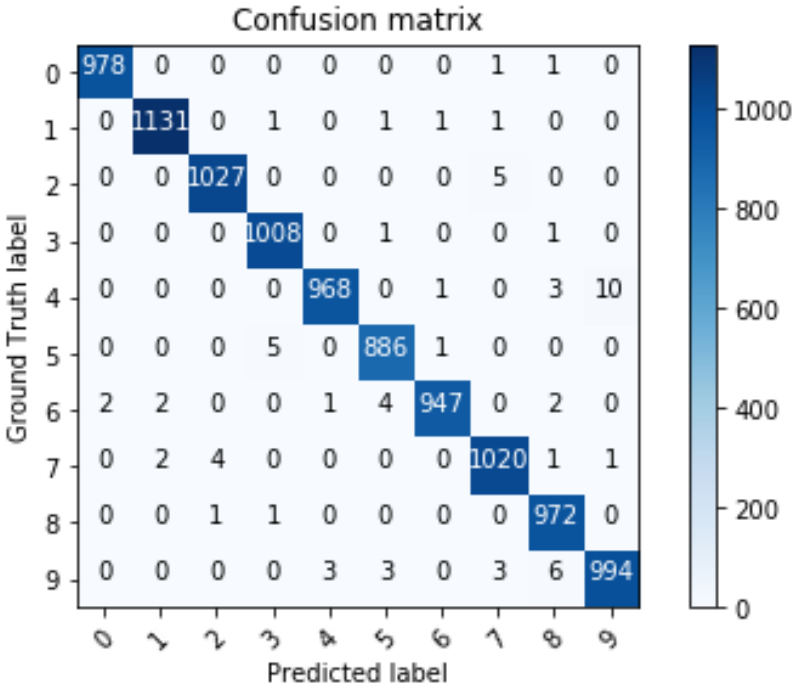


FIGURE 8.1: Confusion matrix with more than one (ten) boolean variable. 9006 Positive (TP and TN) and 69 Negative (FP and FN) values.

8.2 Paired and Combined Criteria

One of the evolutions of the application of the confusion matrix was the adding of paired criteria that take into account not only the single value but a combination. All the following listed values can be considered for network comparisons and are all still used.

- *Precision* (or True Positive Rate): is the proportion of predicted positives which are actual positive
$$\frac{TP}{TP + FP}$$
- *Specificity* (or True Negative Rate): proportion of actual negative which are predicted negative
$$\frac{TN}{TN + FP}$$
- *Recall* (or Sensitivity): is the proportion of the predicted positives out of the possible positives
$$\frac{TP}{TP + FN}$$

- *Positive likelihood*: likelihood that a predicted positive is an actual positive

$$\frac{Sensitivity}{1 - Specificity}$$

- *Negative likelihood*: likelihood that a predicted negative is an actual negative

$$\frac{1 - Sensitivity}{Specificity}$$

Other Combined criteria are:

- *Balanced Classification Rate* (BCR)

$$\frac{1}{2}\left(\frac{TP}{TP + FN} + \frac{TN}{TN + FP}\right)$$

- *Balanced Error Rate* (BER)

$$1 - \text{BER}$$

- *Youden's index*: arithmetic mean between sensitivity and specificity

$$Sensitivity - (1 - Specificity)$$

- *Mattews Correlation Coefficient* (MCC): correlation between the actual and predicted values $(-1 \div 1)$

$$\frac{TP \times TN - FP \times FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}}$$

- *Discriminant power normalised likelihood index*

$$\frac{\sqrt{3}}{\pi}\left(log\left(\frac{Sensitivity}{1 - Specificity}\right) + log\left(\frac{Specificity}{1 - Sensitivity}\right)\right)$$

  results < 1 are poor, > 3 are good and fair otherwise.

8.3 Graphical Tools

From the confusion matrix it is possible to derive two forms of graph useful for the performance evaluation of the model.

ROC Curve

Receiver Operating Characteristics (ROC) curve (an example in figure 8.2) shows the locus of rate of true positive (TP) with respect to the rate of false positive (FP). With this method it is possible to underline the sensitivity of the classifier used in the model. The classifier has to reach the true positive rate of 100% without development on the x-axis, the false positive. This gives a value for the AUC (*Area Under Curve*) equal to 1. This result is impossible to achieve and so good AUC values are over the 0.8 . This method is good also for the evaluation of a *overfitting learning curve*; in fact it is possible to detect if the training phase can be stopped because the maximum level of accuracy is reached and all the remaining epochs could only gives as result an oscillation under that value, with a resulting loss of time.

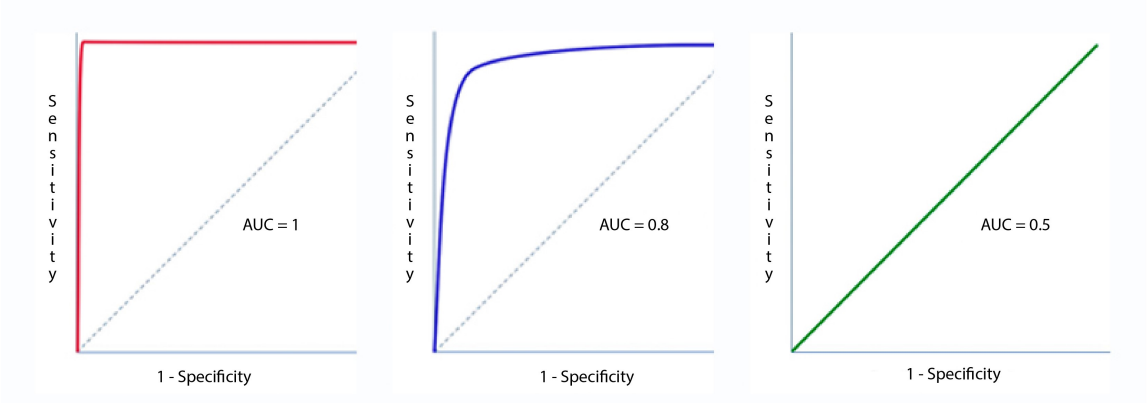


FIGURE 8.2: ROC Curve examples.

8.4 Metrics for Object Recognition

When approaching the object detection analysis, the previous evaluation methods could not be correctly used because the variables in testing are no more only boolean data but pixels. In order to have a unified method for algorithm comparison two main tools have been developed.

8.4.1 IoU

Intersection over Union is the evaluation metric used as base comparator for object detection neural network. It measures the accuracy of an object detector on a particular test dataset. Due to its simple functionality every algorithm that has as output bounding boxes can use it. It requires only two elements for each image under evaluation:

- The *ground-truth* bounding boxes, the bounding boxes created by the programmer that represent the minimum dimension box that includes the object;
- The *predicted* bounding boxes, the output of the model.

Its functionality is trivial and intuitive: IoU measure how much overlap is present between the two selected regions (figure 8.3). Mathematically the IoU is computed as:

$$IoU = \frac{Area\ of\ Overlap}{Area\ of\ Union}$$

so its result is between $0 \div 1$ (figure 8.4); obviously a perfect overlap gives 1 and a completely wrong overlap gives 0 as result. Due to the nature of the dataset and predictions an high result of IoU is over 0.95 but in general results over 0.5 are accepted.

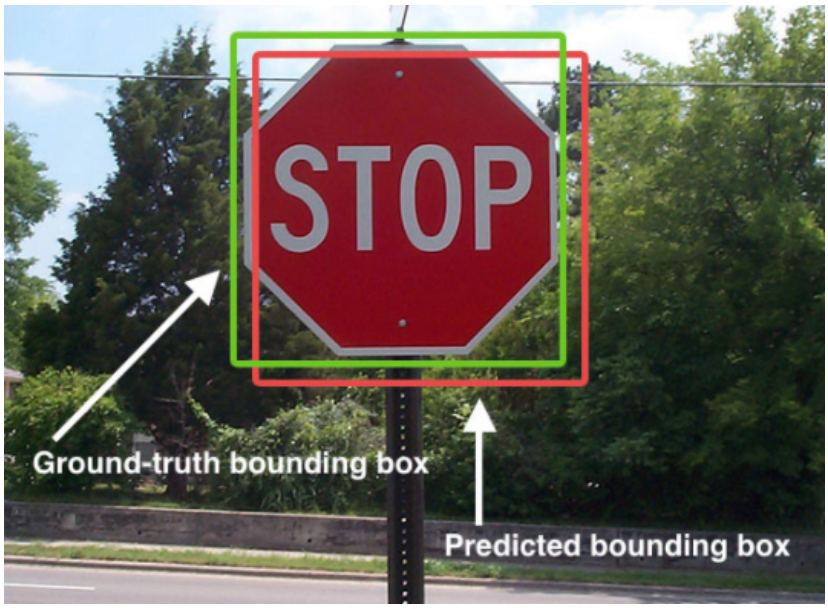
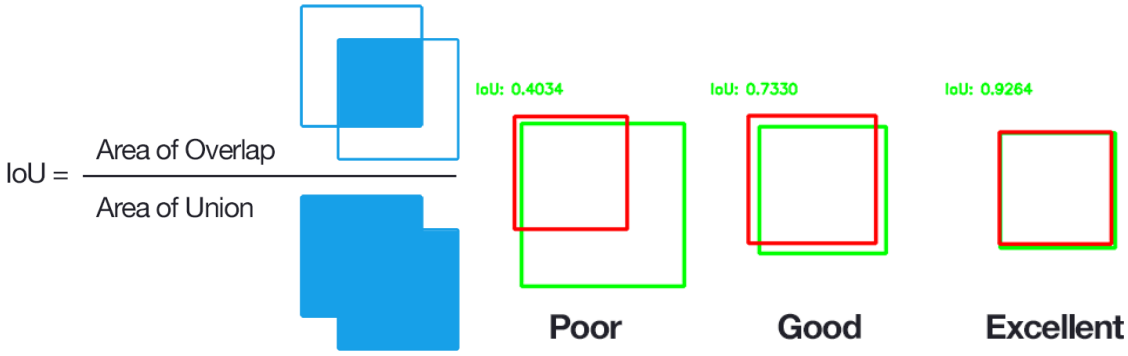


FIGURE 8.3: IoU in a real application of road signal detection. The green box is the ground-truth and the red one is the predicted.



(A) *Visualization of the calculus.*

(B) *Examples of IoU values.*

FIGURE 8.4: IoU visual representation.

Bounding Boxes Data

One of the requirements for the computation of IoU are the ground-truth bounding boxes. Previously it has been said that every algorithm that uses bounding boxes can use IoU as terms for comparisons. This is true but it is necessary a clarification: each algorithm uses different types of annotations and requires its own format for the files in order to be assembled for the IoU. As example VOC and YOLO use a first .txt and a second .xml files for the ground truth, so for each different dataset it is necessary to correctly compose the ground-truth. Several programs try to simplify this process with visual toolkits that permit also different outputs for the selected dataset. For the experiments of this thesis *LabelImg* was used [25], a powerful GitHub project that with a simplified graphical interface (figure 8.5). It gives to the user the total control of the bounding box creation and generates annotation formats for PascalVOC and YOLO.

FIGURE 8.5: Screen-shot of LabelImg program.

This software is used also for the creation of the ground-truth data that can be used for the re-train phase of a network. For this thesis it has been used for the creation of a new dataset used for training different network on the detection of apples (see §9.2).
The second requirements for the IoU are the predicted bounding boxes. One more time the is the problems of compatibility between data. In fact each network can output predicted annotation in various formats. As example YOLO produces a .json format or .txt, PascalVOC produces different .txt and .xml files; all these files have different formatting and have to be pre-processed before being used for the IoU evaluation. Unfortunately not all the networks give the possibility to define the type of output so is an users' task the creation of the correct type of files starting from the predicted output files.

8.4.2 Mean Average Precision

While IoU can be used for the evaluation of a single image prediction, mean Average Precision (mAP) is referred to a general performance of the network. In order to understand this new factor is necessary to introduce step-by-step the components that generate it.

Precision-Recall Curve

In §8.2 were introduced precision and recall factors. They depend from the predefined threshold for the IoU detection (example IoU > 0.5); in fact the True and False results depend on it: a box with IoU under the threshold is considered TN while over TP. This dependency can be used to characterize the performance of a classifier considering how precision and recall change, modifying the threshold. A good classifier precision will stay high as recall increases. A poor classifier will have to take a large hit in precision to get a higher recall. This can be showed in the precision-recall graph (figure 8.6).

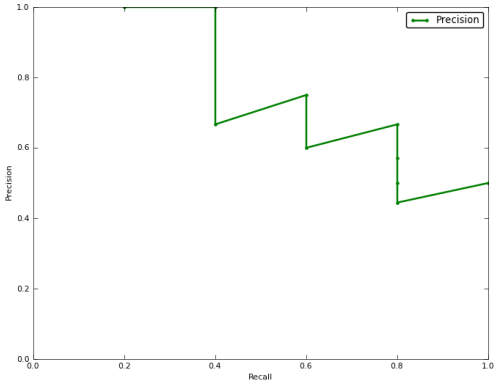


FIGURE 8.6: Example of a Precision-Recall graph. In this case it can achieve 40% recall without sacrificing any precision, but to get 100% recall, its precision drops to 50%.

AP

Obviously comparing graphs from different algorithm is not always easy. In this perspective a value can be useful and this is the role of *Average Precision*. Strictly this value is the area under the precision-recall curve. Having this only element it is easy to compare the algorithms but sometimes it has variations that can confuse the results. Hence the interpolated precision is used. This technique is widely diffused so often with AP is indicated this method. The area considered is no more the perfect area under the graph but the approximated one as in the figure 8.7 (the dotted one).

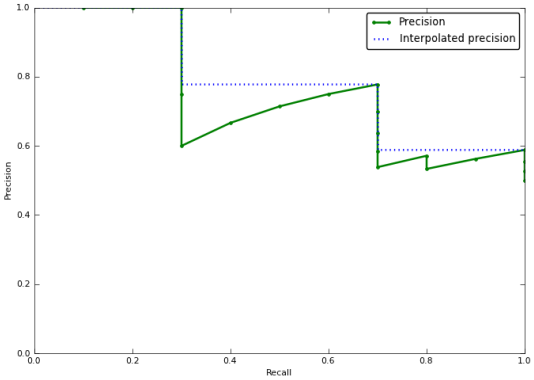
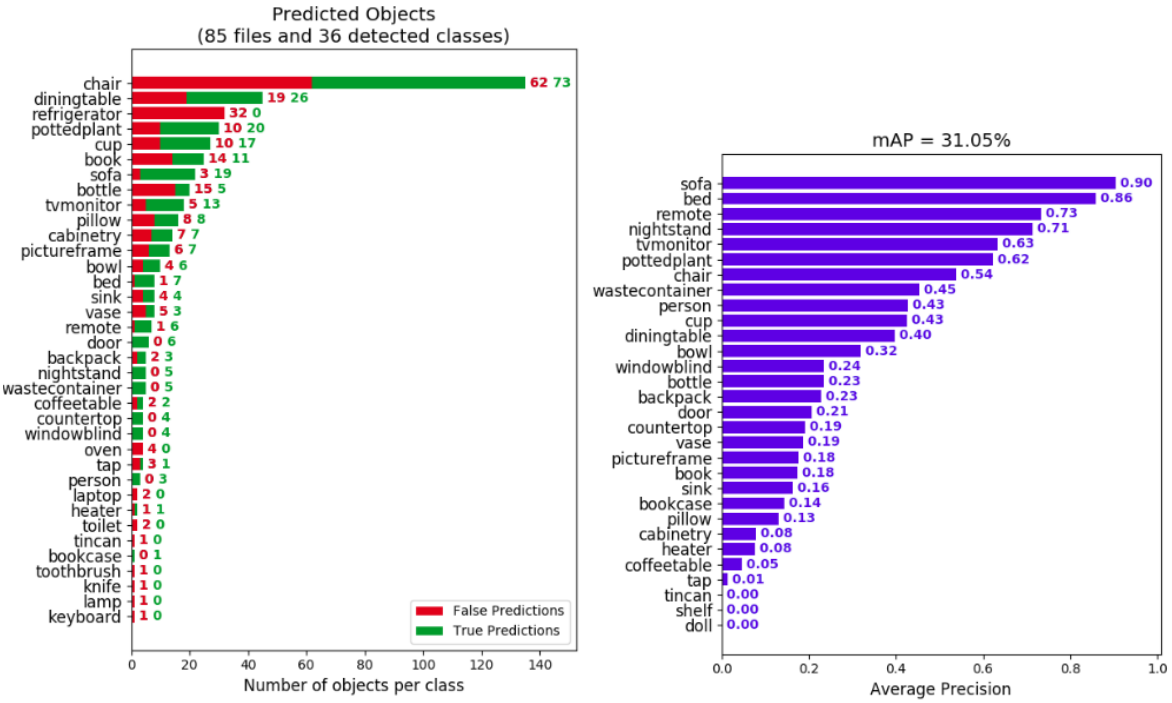


FIGURE 8.7: Example of a Precision-Recall approximated graph.

mAP

Hence when the network has to evaluate under different classes it is possible to produce an AP for each one. The mean between all of the considered classes is the mAP value (figure 8.8).



(A) All the detected classes with False and True predictions.

(B) AP for each class and mAP.

FIGURE 8.8: The mAP is calculated over the AP of the classes that achieve a level of IoU over a certain value (usually 0.5).

Chapter 9

Results and Conclusions

Up to now techniques and architectures at different levels of deepening have been explained, in order to make a general knowledge of a wide argument as deep learning. Hence it is possible to underline what has been done for the application of fruits (apple) recognition for precision agriculture.

9.1 Dataset

If the concepts explained before are clear, also the importance of the correct dataset will be obvious. The dataset contains all the resources on which the training phase of the neural network is done. For the experiments of this thesis having a huge images dataset containing multiple classes other than apple one was fundamentals. This is very important because the network learns to identify simple features likes edges and shadows and recognises also elements different from apples and avoid mismatch situations. Below are listed the datasets used during the thesis.

COCO

Common Object in COntext (COCO) is a huge dataset used in several competitions and as base dataset on which all the new neural networks for object detection can be trained. It contains more than 200k labelled images of 80 classes with bounding boxes and masks for each object. Microsoft and Facebook are some of the partners of this project steadily increasing. The neural network used, YOLO and Mask R-CNN, have been trained on COCO as base dataset.

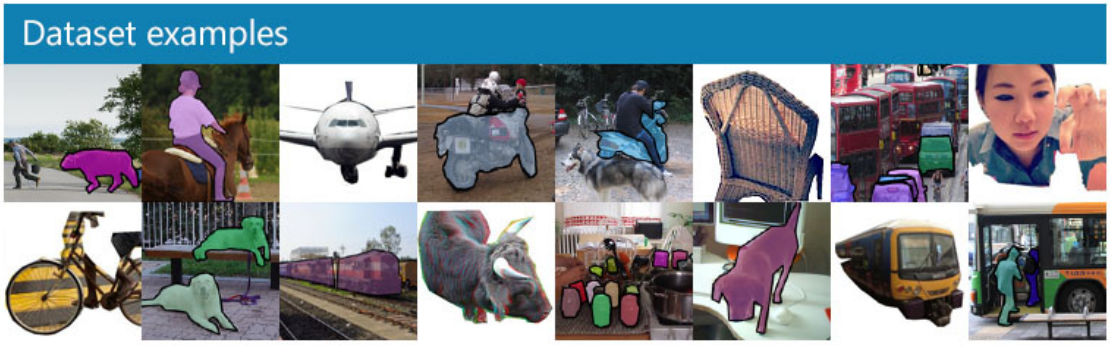


FIGURE 9.1: Example of images into COCO dataset.

ImageNet

ImageNet (figure 9.2) is an image database that contains URLs of figures organized according to the WordNet hierarchy (figure 5.11). It contains 21841 classes (called in this particular case *synsets*) with over 14 million images hand-annotated with the contained object. Over 1

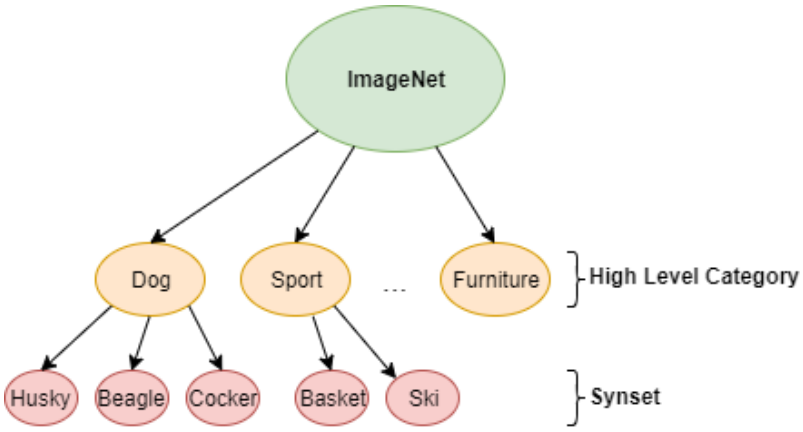


FIGURE 9.2: Final structure ImageNet dataset.

million of these images have bounding boxes that can be used for object detection algorithms. This is a project that continuously grows: the aim of the creators is offering tens of millions of cleanly sorted images; up to now the dataset reaches the average of 1000 images per synset giving to the users the URLs and permissions for research purposes. The competition based on this huge dataset, *ImageNet Large Scale Visual Recognition Challenge* (*ILSVRC*), is the one that signs the start of the current epoch of deep learning. The following table shows the statistics of some high level categories.

High level category	# synsets	Avg # images per synset	Total # images
amphibian	94	591	56K
animal	3822	732	2799K
appliance	51	1164	59K
bird	856	949	812K

TABLE 9.1: Statistic of some high level categories of ImageNet.

Open Images v4

Open Images v4 [26] is a dataset of ∼9 million images that have been annotated with image-level labels and object bounding boxes. The training set of v4 contains 14.6M bounding boxes for 600 object classes on 1.74M images (figure 9.3), making it the largest existing dataset with object location annotations. The boxes have been largely manually drawn by professional annotators to ensure accuracy and consistency. The images are very diverse and often contain complex scenes with several objects (8.4 per image on average). Moreover, the dataset is already subdivided into train, validation and test in order to avoid confusions or overlaps.
The only problem of this dataset regards the download of all the images; in fact the website permits only to differentiate the download between three canonical dataset (train, validation and test) and does not permits the download of a single class or other filters. So also if there is the need of a single class, a user has to download about 500Gb of data. Under this assumption is based my and Vittorio Mazzia work of *OIDv4_Toolkit* [27] (mentioned under

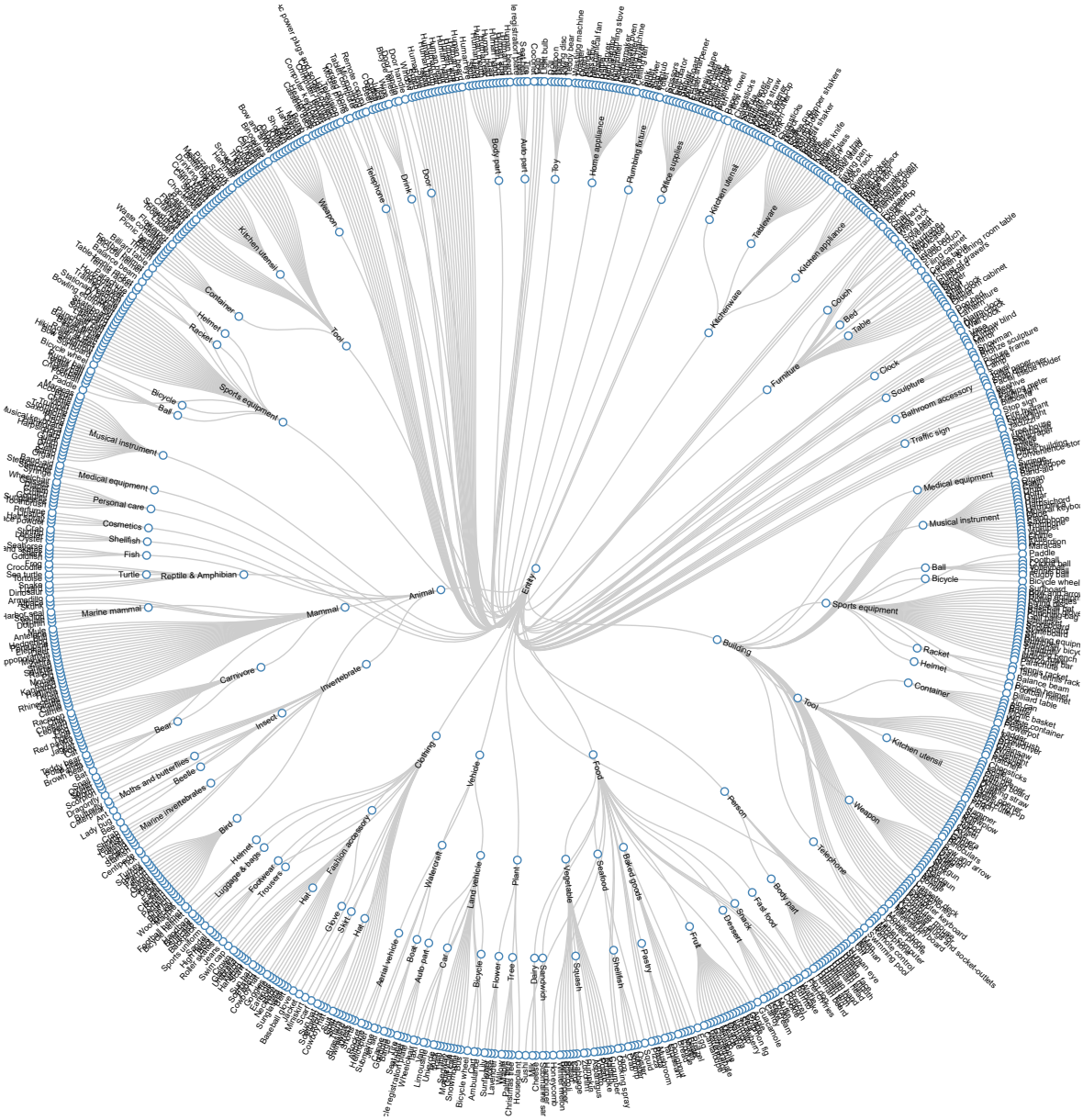


FIGURE 9.3: Dendogram of the classes of Open Images.

the section Extras - Community Contribution of OIDv4). We have developed a python based program that is able to download a single selected class over the three datasets. In this way the retrain of the network has benefited of all the other apple images inside OIDv4 and the overall performance had an increase. At the base of the downloader there is the simple command given by *aws s3* (Amazon Web Service):

```
aws s3 --no-sign-request --only-show-errors cp s3://open-images-dataset/PATH
```

With this command it is possible to download a single image in the selected server knowing its position. The toolkit is able to read the .csv files provided by OIDv4 that contain the name of the class and the name of the image and downloads all the searched ones. Obviously the system is capable in a similar way to download the information related to the labels and generates the .txt files with the correct annotations. With this method a download of 1000 labelled images takes about a minute, far less than doing the work starting from the unlabelled picture. Obviously the dataset does not contain all the possible classes but it

includes many of the most popular. However the OIDv4_Toolkit is much more than a simple downloader. The features of this program are multiple and we are continuously developing it to help all people like us facing this issue. Some of the possibilities that the toolkit offers are:

- Filter the classes: OIDv4 classes have some attributes as *IsGroupOf* or *IsTruncated* that define each picture; the Toolkit is able to detect them and select only the desired images;
- Download a group of classes: in some applications it is useful to train the network detecting multiple classes so the Toolkit permits the simultaneous download of those classes and the creation of the correct labels;
- Visualize the labelled images: the Toolkit has a window able to visualize the downloaded images and the bounding boxes so it is possible to control if the annotations are correct and how the boxes are.

The system is still in development and often there are external contributions or requests that confirm how much the community is growing and that this toolkit maybe can participate to the evolution of object recognition research.

9.2 Re-Training

The detection of apples on trees with high confidence values requires a neural network tailored for the purpose. Training a network starting from scratch is a extremely long work that need huge computational power and time. Moreover the dataset used has to be sufficiently extended including also categories that the system does not have to recognise. The huge request of images generated the idea of "recycling" a trained model and re-use it for different purposes. In *Transfer Learning*, the knowledge of an already trained machine learning model is applied to a different but related problem. The main idea under this method is the fact that all the objects in images are initially defined from simple detections as shadows, edges and colours so the first part of different networks are very similar. Then the objects recognised by the model can be comparable with others, for example a model that detects cars may be trained also for trucks. This assumption gives to the user the possibility that maybe a small dataset and a low-powered computer could have prevented. Transfer learning has been used for the purpose of this thesis for the object detection of apples using the model explained in §5 and §6.
The hardware used for the completion of re-training were the following:

- Private resources:
  - Notebook: the re-training of tiny-YOLOv2 was completed on this device; the time for the completion of 1000 epochs over an i7 of 7^{th}gen was about 13 hours that confirms the negative performance of also a good processor in training neural networks;
  - HPC: *Hactar-PC* is the super-computer that Politecnico di Torino gives to researchers and students; during the usage of this hardware arose many problems. The first huge one was the fact that into this hardware only two graphic units are available and obviously are often busy; secondly the usage of a system like that requires a precise form of scheduling, incompatible with deep learning trials that require fast feedback to the user especially at the beginning where a lot of parameters have to be tuned. As a result of these observations HPC has been only tested but not used for the retraining phase of the thesis.

- Cloud resources:
  - Crestle [28] is one of the websites that give the possibility to use external server with multiple configurations. On Crestle all the popular scientific computing and deep learning packages are pre-installed and configured to run on a GPU. It is one of the most popular online platforms for deep learning computations. The retraining of Mask R-CNN has been done using this resource with 15 epochs (not too much but with good results) and it took 1.5 hours;
  - AWS: *Amazon Web Services* offers a lot of solutions including EC2 instances for deep learning. It has been created a virtual Linux environment that supports GPUs computations. On this environment has been done the re-training of YOLO models: YOLOv2 with 4500 epochs that took about 12 hours and YOLOv3 with 2000 epochs that took about 23 hours. The trainings have been stopped when the average loss values starting to oscillate over the final values.

For the correct re-training of the networks two main actions were needed: the creation of an additional dataset of apples and a slightly modification of the network.

9.2.1 Dataset of Apples

Also even if the COCO dataset used for the original training of YOLO and Mask R-CNN contains "apple" as one of the 80 classes, a new dataset that contains only apples was created and used for the re-training. This aided the growing of the mAP and makes the system more robust. Creating a new dataset consists on selecting a set of figures and makes the bounding area around the apples. The two networks used, YOLO and Mask R-CNN, need different types of annotations for each figure. YOLO, in all its versions, needs bounding boxes defined in a .txt file with a specific format:

```
<object-class> <x> <y> <width> <height>
```

where:

- `<object-class>`: an integer number of object from 0 to (classes-1);
- `<x> <y> <width> <height>` are normalized float values relative to the center of the rectangle bounding box, width and height of image and it can be equal to $[0.0 \div 1.0]$.

Example:

```
0 0.684356 0.357294 0.266734 0.159234
```

The creation of this type of annotation has been possible with the usage of a software named *LabelImg* (figure 8.5) with whom ∼1000 images containing apples have been processed.
A different process has been used for the annotations of Mask R-CNN. In fact this network requires not the bounding boxes but a line over the shape of the apple. For this type of annotation *VGG Image Annotator* (VIA) [29] has been used. This software gives the possibility to make the outline profile and to generate a .json output that can be used for the neural network. As an example (figure 9.4) is reported the same picture of LabelImg but processed for Mask R-CNN in order to see the differences. With this method ∼300 images have been processed and used during the re-training of the network.

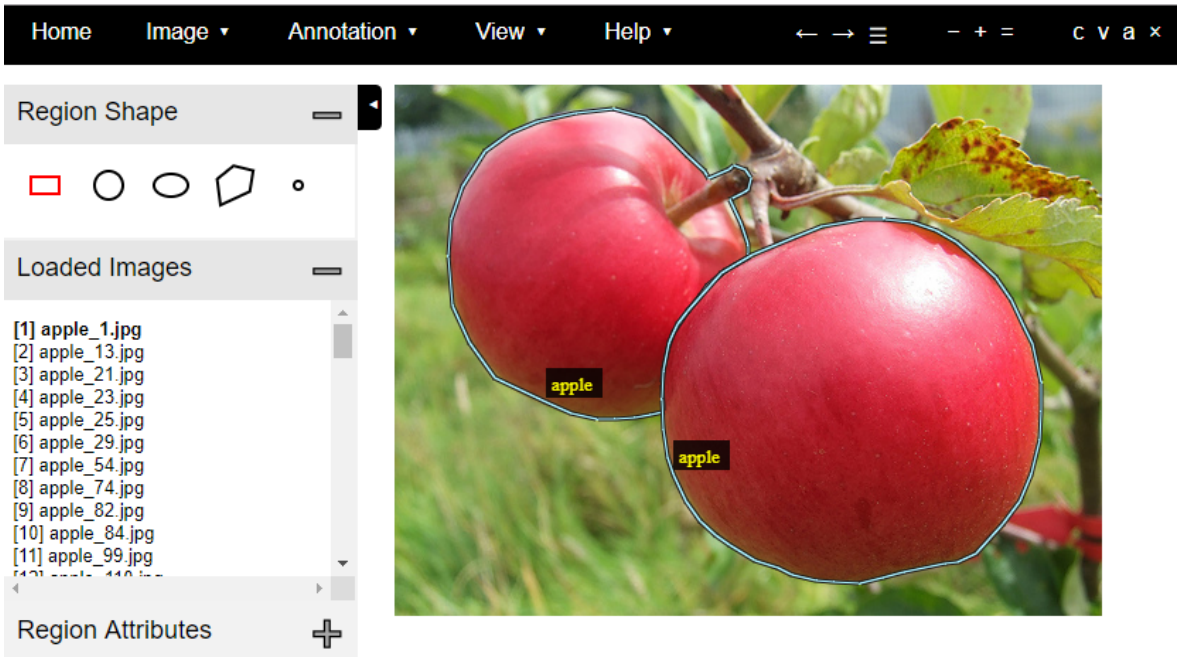


FIGURE 9.4: Annotation example with VIA.

From this software is possible to obtain the .json file structured as following:

```
"apple_1.jpg104549": {
        "fileref": "",
        "size": 104549,
        "filename": "apple_1.jpg",
        "base64_img_data": "",
        "file_attributes": {},
        "regions": {
                "0": {
                        "shape_attributes": {
                        "name": "polygon",
                        "all_points_x": [107, 71,..., 125, 107],
                        "all_points_y": [224, 207,..., 232, 224]
                        },
                        "region_attributes": {
                                "object_name": "apple"
                        }
                },
                "1": {
                        "shape_attributes": {
                                "name": "polygon",
                                "all_points_x": [197, 206, ... , 193,
                                   197],
                                "all_points_y": [275, 294, ... , 263,
                                   275]
                                },
                                "region_attributes": {
                                        "object_name": "apple"
                                }
                        }
                }
}
```

9.2.2 Network Modifications

The network used had to be changed in order to detect only apples. First the number of classes specified in the configuration file was reduced from 80 to 2 (background and apple), in this way the network output referred only to a specific object resulting more performing and accurate. The other modification referred to specific hyper-parameters of the net:

- batch: a batch is a group of images used for the training of each epoch; for YOLO the predefined batch dimension was modified into 64 images with a subdivision of 32;
- input image dimension: each net has a predefined dimension that can vary in order to follow the dimension of the images that have to be used; in this case YOLO input image was imposed to 608x608;
- learning rate: during the training the learning rate changed following the gradient descent theory, so this parameter is reduced at three different steps: until the 200^{th} step is multiplied by 10 (0.0001×10), then until the 5000^{th} step is reduced by 10 and then until the 7000^{th} step is again reduced by 10.

9.3 Results

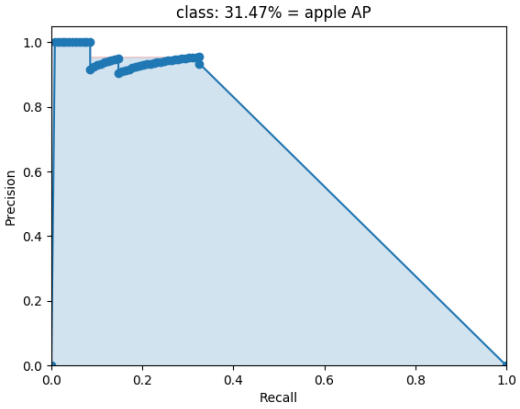
The following table shows the mAP calculated over the different network used and testifies the usefulness of the re-training done.

Architecture	Dataset		
	COCO	Apple	%
Tiny Y.O.L.O.v2	31.47%	49.96%	+58.75%
Y.O.L.O.v2	60%	76.88%	+28.13%
Y.O.L.O.v3	63.2%	70.76%	+11.96%
Mask R-CNN	68.8%	77.65%	+12.86%

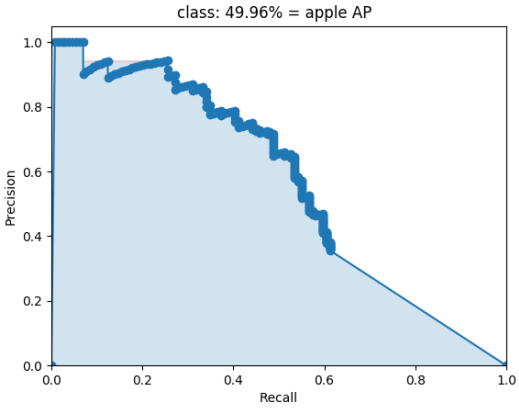
TABLE 9.2: Original network and re-trained one mAP results.

It is possible to underline how the "big" networks are subjected to a minor benefit of the mAP with respect to the smallest one (Tiny YOLOv2). This result can be justified considering the already robust neural networks used that give great outputs already with the original net. The re-train for this case increases the mAP results but above all reduces the processing time of the images, raising the FPS for a real-time application.

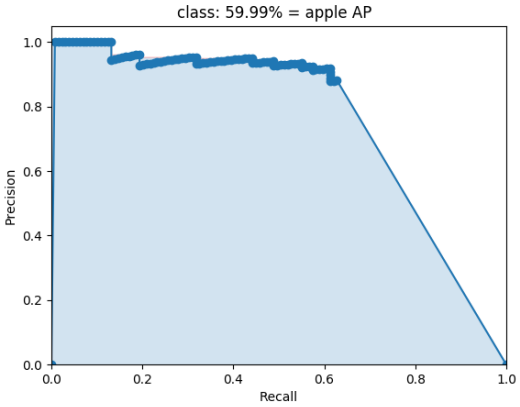
9.3.1 Examples of Processed Images



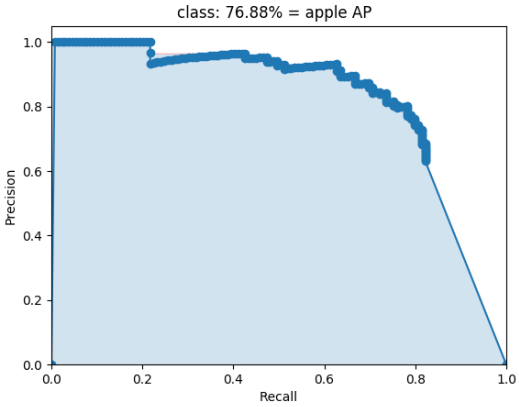
(A) Tiny YOLOv2 COCO



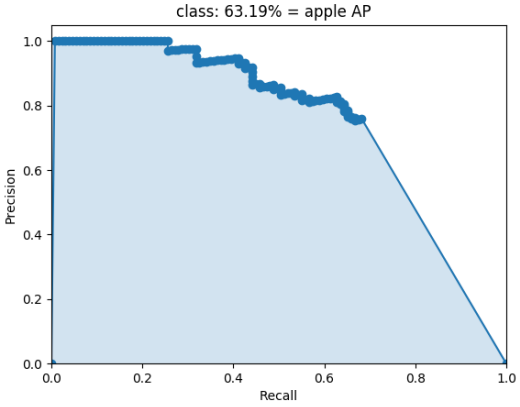
(B) Tiny YOLOv2 Apple



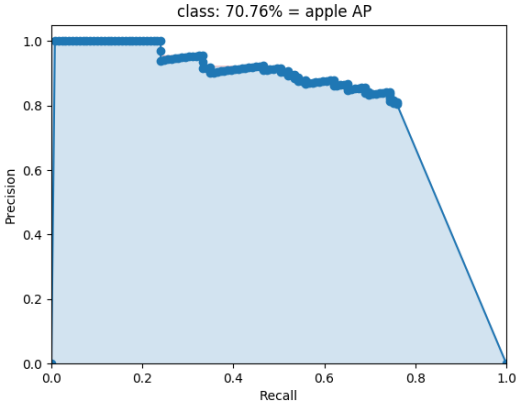
(C) YOLOv2 COCO



(D) YOLOv2 Apple

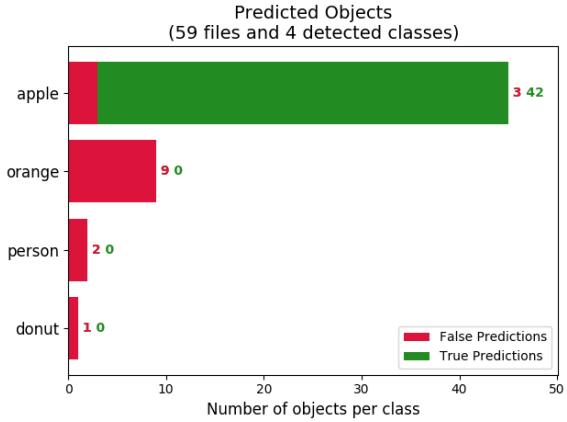


(E) YOLOv3 COCO

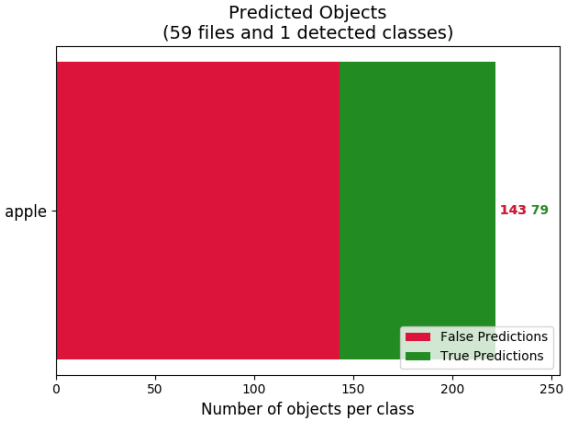


(F) YOLOv3 Apple

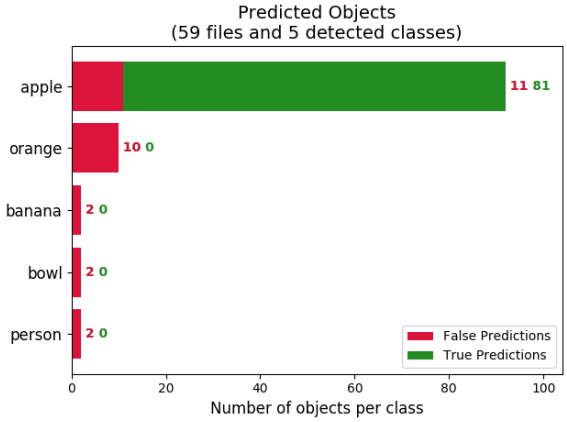
FIGURE 9.5: mAP graphs for the different networks before and after the re-training.



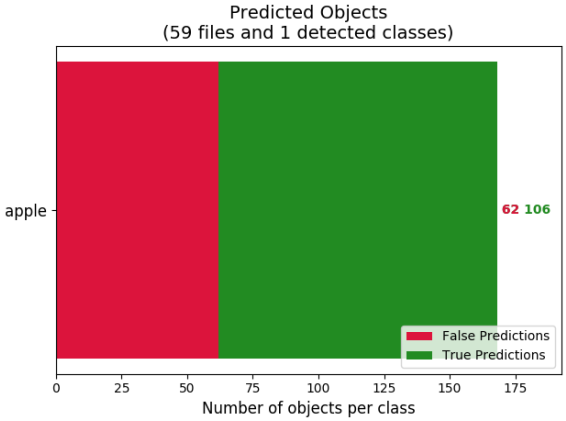
(A) Tiny YOLOv2 COCO



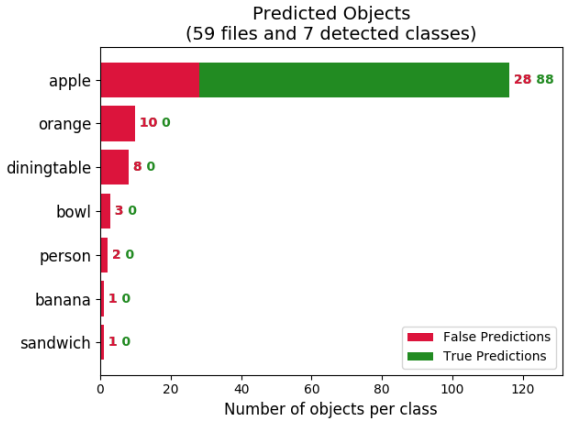
(B) Tiny YOLOv2 Apple



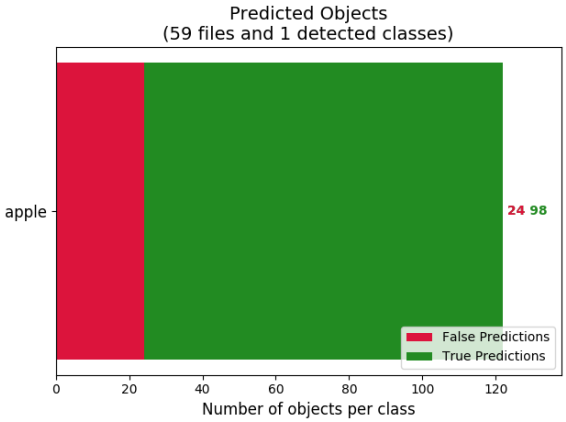
(C) YOLOv2 COCO



(D) YOLOv2 Apple

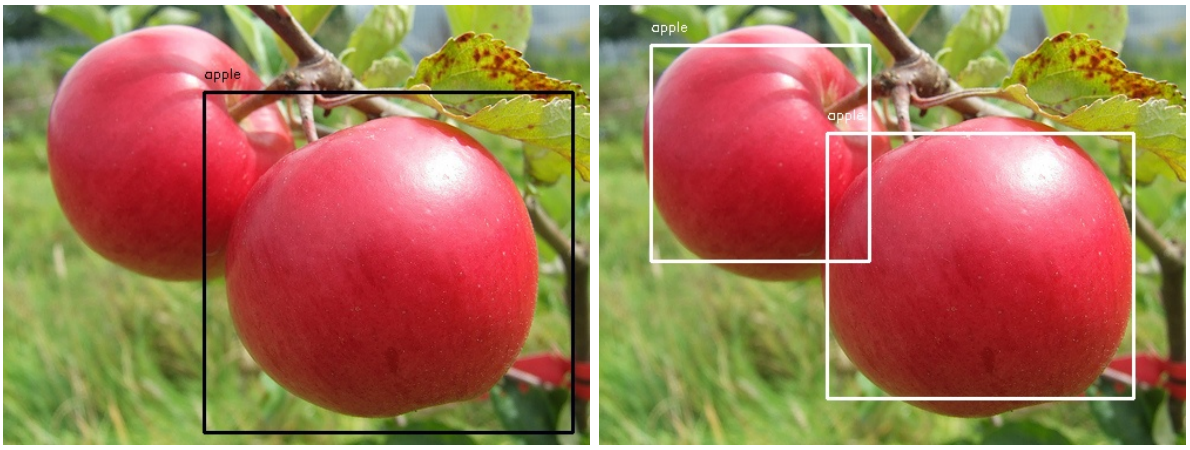


(E) YOLOv3 COCO



(F) YOLOv3 Apple

FIGURE 9.6: Predicted object graphs for the different networks before and after the re-training.



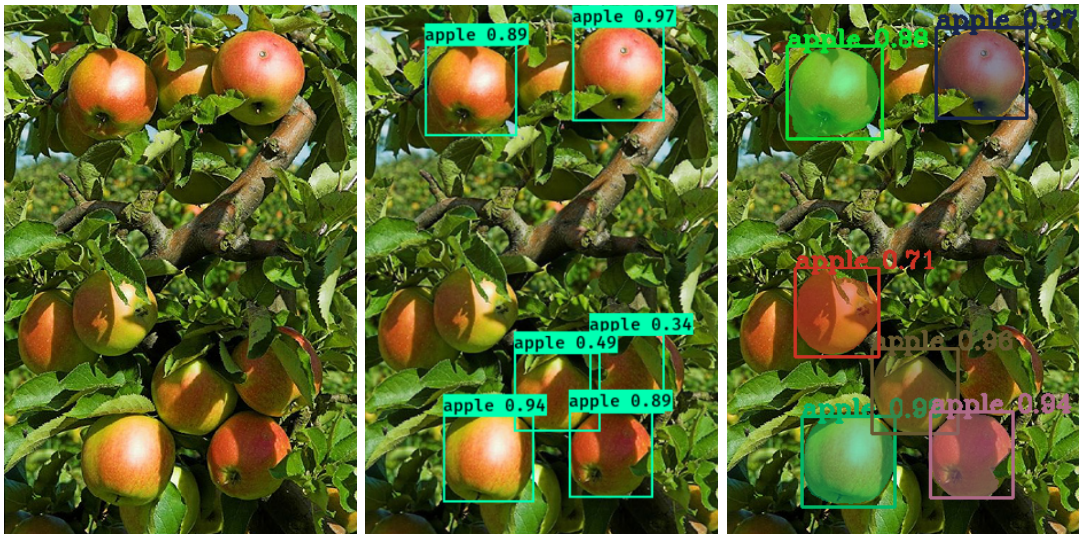
(A) Tiny YOLOv2 COCO

(B) Tiny YOLOv2 APPLE

(C) YOLOv2 COCO

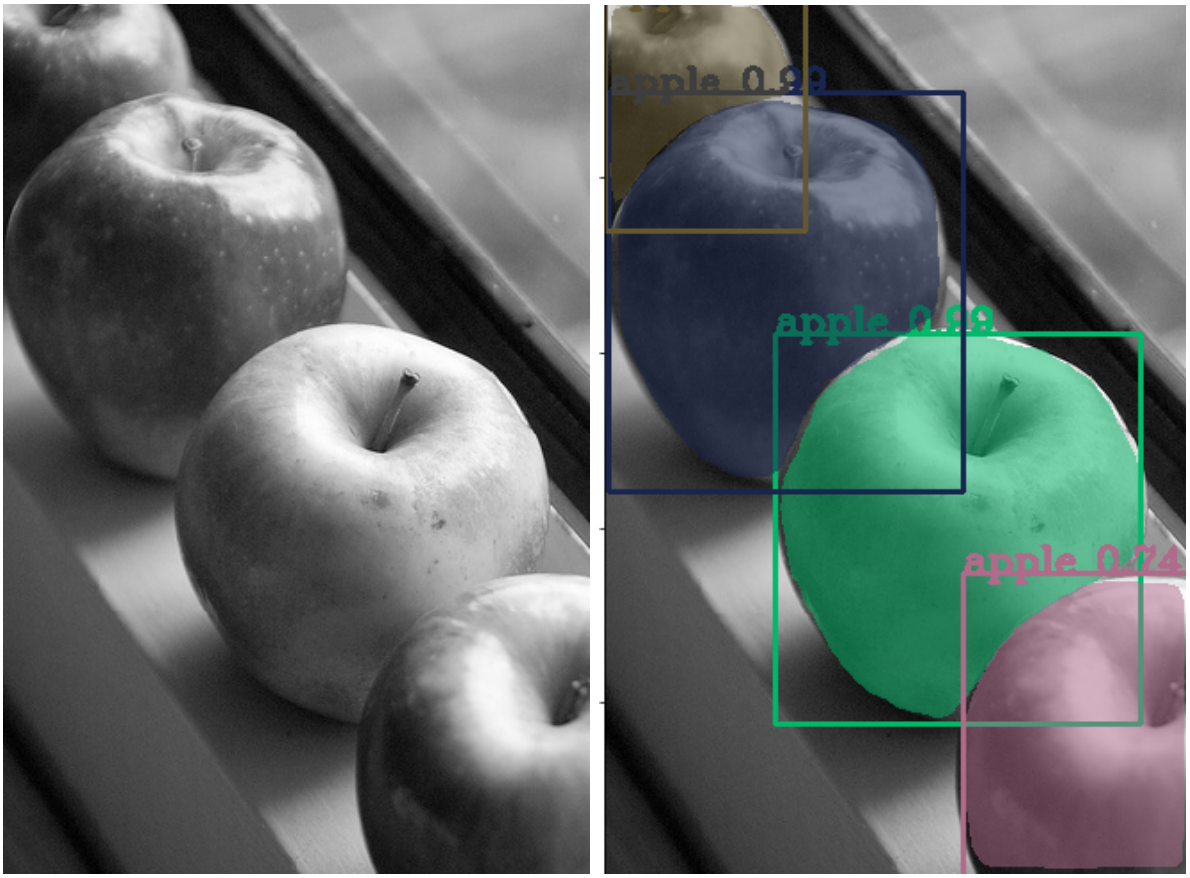
FIGURE 9.7: YOLO outputs.

FIGURE 9.8: YOLOv3 outputs.



(A) Original image (B) YOLOv3 output (C) Mask R-CNN output

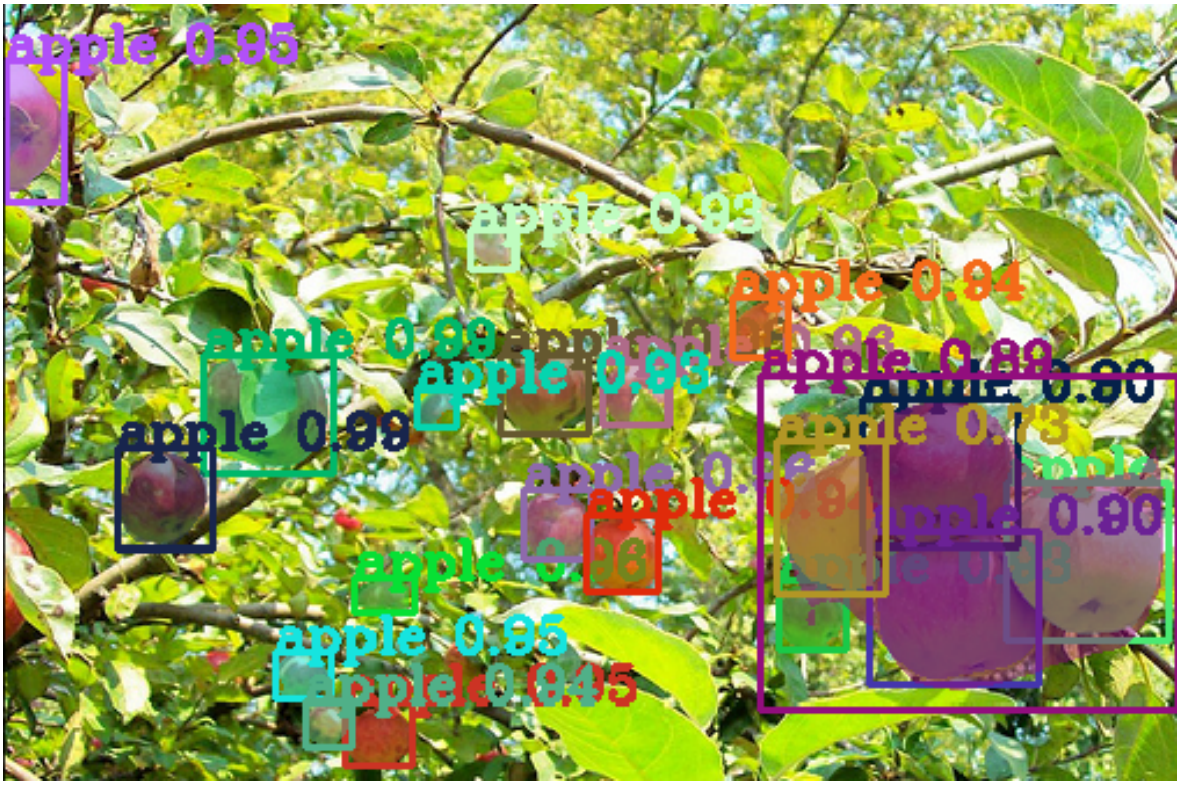
FIGURE 9.9: Same images processed over two different networks.



(A) Original image (B) Mask R-CNN output

FIGURE 9.10: Outputs from Mask R-CNN.

(A) Original image



(B) Mask R-CNN output

FIGURE 9.11: Outputs from Mask R-CNN.

9.4 Conclusions and Future Works

As it is possible to see from the example images the different networks give as output good results that can be useful during the process implied on precision agriculture; the re-training had all better results than the original network testifying the benefit of transfer learning and the works done during the months of this thesis. Next step can be the real application on a rover to pose the basis of auto-harvesting and to use the trained network to make evaluations of agricultural fields. Therefore the same procedure applied can be used for other fruits to generalize the possible application of this methodology. As last project, these results can be combined with a stereo-camera for the computation of the space-positioning of fruits.

Bibliography

[1] Woo Chaw Seng and Seyed Hadi Mirisaee. "A new method for fruits recognition system". In: *Electrical Engineering and Informatics, 2009. ICEEI'09. International Conference on.* Vol. 1. IEEE. 2009, pp. 130–134.

[2] Chomtip Pornpanomchai et al. "Thai fruit recognition system (TFRS)". In: *Proceedings of the First International Conference on Internet Multimedia Computing and Service.* ACM. 2009, pp. 108–112.

[3] S Arivazhagan et al. "Fruit recognition using color and texture features". In: *Journal of Emerging Trends in Computing and Information Sciences* 1.2 (2010), pp. 90–94.

[4] Y Song et al. "Automatic fruit recognition and counting from multiple images". In: *Biosystems Engineering* 118 (2014), pp. 203–215.

[5] Susovan Jana, Saikat Basak, and Ranjan Parekh. "Automatic fruit recognition from natural images using color and texture features". In: *Devices for Integrated Circuit (DevIC), 2017.* IEEE. 2017, pp. 620–624.

[6] *Scikit-Learn.* URL: `http://scikit-learn.org/stable/`.

[7] *Tensor Flow.* URL: `https://www.tensorflow.org/`.

[8] *Caffe.* URL: `http://caffe.berkeleyvision.org/`.

[9] *Microsoft Cognitive Toolkit (CNTK).* URL: `https://www.microsoft.com/en-us/cognitive-toolkit/`.

[10] *Keras: the Python Deep Learning Library.* URL: `https://keras.io/`.

[11] *Going deep into object detection.* URL: `https://towardsdatascience.com/going-deep-into-object-detection-bed442d92b34`.

[12] Paul Viola and Michael Jones. "Rapid object detection using a boosted cascade of simple features". In: *Computer Vision and Pattern Recognition, 2001. CVPR 2001. Proceedings of the 2001 IEEE Computer Society Conference on.* Vol. 1. IEEE. 2001, pp. I–I.

[13] *Haar Classifier in Face Detection.* URL: `https://docs.opencv.org/3.4.1/d7/d8b/tutorial_py_face_detection.html`.

[14] Navneet Dalal and Bill Triggs. "Histograms of oriented gradients for human detection". In: *Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on.* Vol. 1. IEEE. 2005, pp. 886–893.

[15] Koen EA Van de Sande et al. "Segmentation as selective search for object recognition". In: *Computer Vision (ICCV), 2011 IEEE International Conference on.* IEEE. 2011, pp. 1879–1886.

[16] Ross Girshick et al. "Rich feature hierarchies for accurate object detection and semantic segmentation". In: *Proceedings of the IEEE conference on computer vision and pattern recognition.* 2014, pp. 580–587.

[17] Ross Girshick. "Fast r-cnn". In: *Proceedings of the IEEE international conference on computer vision*. 2015, pp. 1440–1448.

[18] Shaoqing Ren et al. "Faster R-CNN: towards real-time object detection with region proposal networks". In: *IEEE Transactions on Pattern Analysis & Machine Intelligence* 6 (2017), pp. 1137–1149.

[19] Wei Liu et al. "Ssd: Single shot multibox detector". In: *European conference on computer vision*. Springer. 2016, pp. 21–37.

[20] Joseph Redmon et al. "You only look once: Unified, real-time object detection". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 779–788.

[21] Joseph Redmon and Ali Farhadi. "YOLO9000: better, faster, stronger". In: *arXiv preprint* (2017).

[22] Joseph Redmon and Ali Farhadi. "Yolov3: An incremental improvement". In: *arXiv preprint arXiv:1804.02767* (2018).

[23] Kaiming He et al. "Mask r-cnn". In: *Computer Vision (ICCV), 2017 IEEE International Conference on*. IEEE. 2017, pp. 2980–2988.

[24] *Splash of Color: Instance Segmentation with Mask R-CNN and TensorFlow*. URL: https://engineering.matterport.com/splash-of-color-instance-segmentation-with-mask-r-cnn-and-tensorflow-7c761e238b46.

[25] *LabelImg, Git code (2015)*. URL: https://github.com/tzutalin/labelImg.

[26] Ivan Krasin et al. "OpenImages: A public dataset for large-scale multi-label and multi-class image classification." In: (2017). URL: https://storage.googleapis.com/openimages/web/index.html.

[27] *OIDv4_ToolKit: Toolkit to download and visualize single or multiple classes from the huge Open Images v4 dataset*. 2018. URL: https://github.com/EscVM/OIDv4_ToolKit.

[28] *Crestle: Effortless Infrastructure for Deep Learning*. https://www.crestle.com/. (Accessed on 10/04/2018).

[29] A. Dutta, A. Gupta, and A. Zissermann. *{VGG} Image Annotator ({VIA})*. 2016. URL: http://www.robots.ox.ac.uk/~vgg/software/via/.

Appendix A

Raspberry Code

```
#---------------------------
#Project: Tiny-Yolov2_Apple on Movidius
#---------------------------
#---------------------------
# Initialization
#---------------------------
import numpy as np
import cv2
import json
import mvnc.mvncapi as mvncapi
import time
from darkflow.cython_utils.cy_yolo2_findboxes import box_constructor
from imutils.video import VideoStream
#---------------------------
# Functions declaration
#---------------------------
def get_meta(meta_file):
    """ Open the meta file of the NN"""
    with open(meta_file, 'r') as fp:
        meta = json.load(fp)
    return meta
def pre_proc_img(img, meta):
    """ Preprocessing a frame
        before feeding into into the NN"""
    w, h, _ = meta['inp_size']
    img = img.astype(np.float32)
    imsz = cv2.resize(img, (w, h))
    imsz = imsz / 255.
    imsz = imsz[:,:,::-1]
    imsz = np.expand_dims(imsz, axis=0)
    return imsz
def findboxes_meta(meta, net_out):
    boxes = list()
    boxes = box_constructor(meta, net_out)
    return boxes
def process_box_meta(meta, b, h, w, threshold):
    max_indx = np.argmax(b.probs)
    max_prob = b.probs[max_indx]
    label = meta['labels'][max_indx]
    if max_prob > threshold:
        left  = int ((b.x - b.w/2.) * w)
        right = int ((b.x + b.w/2.) * w)
        top   = int ((b.y - b.h/2.) * h)
        bot   = int ((b.y + b.h/2.) * h)
        if left  < 0    :  left = 0
        if right > w - 1: right = w - 1
        if top   < 0    :   top = 0
        if bot   > h - 1:   bot = h - 1
        mess = '{}'.format(label)
        return (left, right, top, bot, mess, max_indx, max_prob)
    return None
def procces_out(out, meta, img_orig_dimensions):
    h, w, _ = img_orig_dimensions
```

```
    boxes = findboxes_meta(meta,out)
    threshold = meta['thresh']
    boxesInfo = list()
    for box in boxes:
        tmpBox = process_box_meta(meta, box, h, w, threshold)
        if tmpBox is None:
            continue
        boxesInfo.append({
            "label": tmpBox[4],
            "confidence": tmpBox[6],
            "topleft": {
                "x": tmpBox[0],
                "y": tmpBox[2]},
            "bottomright": {
                "x": tmpBox[1],
                "y": tmpBox[3]}
        })
    return boxesInfo
def add_bb_to_img(img_orig, boxes, fps):
    """ Draws rectangles and text on
        the img_orig"""
    for box in boxes:
        left = box["topleft"]['x']
        right = box["bottomright"]['x']
        top = box["topleft"]['y']
        bot = box["bottomright"]['y']
        label = box["label"]
        confidence = box["confidence"]
        h, w, _ = img_orig.shape
        thick = int((h + w) // 300)
        cv2.rectangle(img_orig,
            (left, top), (right, bot),
            [255, 0, 0], thick)
        info = label + ' ' + '{:.2%}'.format(confidence)
        cv2.putText(
            img_orig, info, (left + 5, top + 14),
            0, 1e-3 * h, [255, 0, 0],
            thick // 3)
        FPS = 'FPS: ' + '{:.2f}'.format(fps)
        cv2.putText(
            img_orig, FPS, (left + 5, bot - 12),
            0, 1e-3 * h, [255, 0, 0],
            thick // 3)
def get_mvnc_device():
    """ Simply checks for all devices
        attached at the Rasp and open them"""
    mvncapi.global_set_option(mvncapi.GlobalOption.RW_LOG_LEVEL, 4)
    # get a list of names for all the devices plugged into the system
    devices = mvncapi.enumerate_devices()
    if (len(devices) < 1):
        print("Error --- no NCS device detected")
        quit()
    # get the first NCS device by its name.
    # For this program we will always open the first NCS device.
    dev = mvncapi.Device(devices[0])
    # try to open the device.  this will throw an exception
    # if someone else has it open already
    try:
        dev.open()
    except:
        print("Error - It hasn't been possible to open the divice")
        quit()
    return dev
def load_graph(dev, GRAPH_FILEPATH):
    """Allocate the graph
       to a certain device dev"""
    with open(GRAPH_FILEPATH, mode='rb') as f:
        graph_file_buffer = f.read()
    graph = mvncapi.Graph('graph1')
    input_fifo, output_fifo = graph.allocate_with_fifos(dev, graph_file_buffer)
    return graph, input_fifo, output_fifo
```

```
#-----------
#  Main
#-----------
if __name__ == '__main__':
    # Read the meta file with all the configurations
    meta_file = 'graph/yolov2-tiny_apple.meta'
    meta = get_meta(meta_file)
    threshold = 0.3
    meta['thresh'] = threshold
    # Initialize the device
    dev = get_mvnc_device()
    # Load the .graph file into the NCS
    graph_file = 'graph/yolov2-tiny_apple.graph'
    graph, input_fifo, output_fifo = load_graph(dev, graph_file)
    # Initialize the camera
    # Chose the type of input camera
    cam = 0
    cam = VideoStream(usePiCamera=cam > 0).start()
    time.sleep(2.0)
    #-----------------
    # Predictions
    #-----------------
    while (True):
        frame = cam.read()
        # Original frame
        frame_orig = np.copy(frame)
        img_orig_dimensions = frame_orig.shape
        # Processing the captured frames
        frame = pre_proc_img(frame, meta)
        # Make a prediction with the Movidius NCS printing the relative fps
        s_time = time.time()
        graph.queue_inference_with_fifo_elem(input_fifo, output_fifo, frame,
        'user object')
        output, user_obj = output_fifo.read_elem()
        e_time = time.time()
        fps = 1/(e_time-s_time)
        print("NCS FPS: {}".format(1/(e_time-s_time)))
        # Reshape the output tensor
        y_out = np.reshape(output, (13, 13,425))
        y_out = np.squeeze(y_out)
        # Use darkflow utils to extrapolate the dictionary
          from the output tensor
        boxes = procces_out(y_out, meta, img_orig_dimensions)
        # Draw boxes and text
        add_bb_to_img(frame_orig, boxes, fps)
        # Display the processed original frames
        cv2.imshow('Tiny-Yolov2 Apple', frame_orig)
        k = cv2.waitKey(1) & 0xFF
        if k == 27:
            break
    # clean up the graph and device
    graph.destroy()
    dev.close()
    dev.destroy()
    cv2.destroyAllWindows()
    cam.stop()
```