POLITECNICO DI TORINO

Corso di Laurea Magistrale
in Ingegneria Matematica

Tesi di Laurea Magistrale

Apples recognition on trees with neural networks

Relatore
Prof. Francesco Vaccarino

Candidata
Debora Cravero

Anno Accademico 2018/2019

Contents

1 Introduction **1**
- 1.1 Objective . . . 1
- 1.2 Introduction to Artificial Intelligence . . . 2
  - 1.2.1 Brief history of AI . . . 3

2 The neural networks **6**
- 2.1 The biological model . . . 6
- 2.2 The artificial neuron's structure . . . 7
- 2.3 The neural network's structure . . . 8
- 2.4 Technical details . . . 9
  - 2.4.1 The activation function . . . 9
  - 2.4.2 Gradient descent . . . 12
  - 2.4.3 Stochastic gradient descent . . . 14
  - 2.4.4 Backpropagation algorithm . . . 15
  - 2.4.5 Regularization techniques . . . 16

3 Types and examples of neural networks **19**
- 3.1 Fully-connected neural networks . . . 19
  - 3.1.1 The Perceptron . . . 20
  - 3.1.2 Multilayer Perceptron . . . 21
  - 3.1.3 Linear associator . . . 22
- 3.2 Recurrent neural networks . . . 23
  - 3.2.1 Hopfield Net . . . 25
  - 3.2.2 Boltzmann machines . . . 26
- 3.3 Self-Organizing Maps: Kohonen Maps . . . 28
- 3.4 Convolutional Neural Networks . . . 29
  - 3.4.1 LeNet-5 . . . 32
  - 3.4.2 AlexNet . . . 33
  - 3.4.3 VGG . . . 35
  - 3.4.4 YOLO . . . 36
  - 3.4.5 YOLOv3 . . . 37
  - 3.4.6 Tiny YOLOv3 . . . 45
- 3.5 Generative Adversarial Networks . . . 46

4 My project **48**
- 4.1 Datasets . . . 48
  - 4.1.1 Training set and validation set . . . 49
  - 4.1.2 Test set . . . 49

4.2 Metrics 50
4.3 Results on YOLOv3 52
4.3.1 Confidence score set to 0.25 53
4.3.2 Fine-tuning YOLOv3 54
4.4 Results on Tiny YOLOv3 58
4.4.1 Fine-tuning Tiny YOLOv3 60
4.4.2 Grid search on Tiny YOLOv3 62
4.4.3 Fine-tuning after grid-search 70
4.4.4 An oddity 71

5 Conclusions **74**

6 Images **76**

Bibliography **129**

List of Figures

1.1 Example of apples with strong difference in sunlight 2
1.2 Examples of apples grouped in a cluster, hidden by leaves or divided by branch 2
1.3 The difference engine by Babbage 3
1.4 Frame of the match Deep Blue vs Kasparov 5

2.1 Human neurons' structure 6
2.2 Artificial neuron's structure 7
2.3 A fully-connected network 9
2.4 Heaviside function 9
2.5 Example of linear ramp 10
2.6 Logistic sigmoid function 10
2.7 Hyperbolic tangent function 11
2.8 ReLU function 11
2.9 Stationary points 13
2.10 Examples of learning rate's choice 13
2.11 Chain rule on a neuron 15

3.1 Example of fully-connected neural network 19
3.2 Perceptron's model 20
3.3 Logic operators 21
3.4 Space's transformation for the XOR problem 22
3.5 General RNN's structure 23
3.6 Different types of RNNs 25
3.7 Hopfield network's structure 25
3.8 Example of application with Hopfield network 26
3.9 General Boltzmann machine's model 27
3.10 Other Boltzmann machines 28
3.11 Kohonen Maps' structure 29
3.12 Example of max-pooling function on 2x2 regions with stride = 2 31
3.13 Examples of different zero-padding, everyone with kernel 3x3 and stride 1 32
3.14 LeNet-5's structure 32
3.15 Table of connections between second and third layer 33
3.16 AlexNet's architecture 34
3.17 VGG-19 35
3.18 VGG's configurations 36
3.19 Example of residual blocks 37
3.20 YOLOv3's architecture 38

3.21 IoU definition . . . 39
3.22 Bounding box coordinates . . . 39
3.23 Combination of COCO and ImageNet with WordNet . . . 42
3.24 Feature pyramid's scheme . . . 43
3.25 Structure of Darknet-53 . . . 44
3.26 Difference between traditional and global averaging approach . . 45
3.27 Tiny YOLOv3 architecture . . . 46
3.28 Example of DCGAN model used to generate cats' fake images . . 47

4.1 Some examples of pictures taken . . . 50
4.2 Example of PR curve with its approximation . . . 51
4.3 Overall results of YOLOv3 . . . 52
4.4 Prediction time with YOLOv3 . . . 53
4.5 Overall results of YOLOv3 with confidence 0.25 . . . 54
4.6 Prediction time of YOLOv3 with threshold of confidence score 0.25 54
4.7 General results after first fine-tuning on YOLOv3 . . . 56
4.8 Plot of average loss error during fine-tuning . . . 57
4.9 Trend of mAP testing weights produced during fine-tuning . . . 57
4.10 Results of the second fine-tuning on YOLOv3 . . . 58
4.11 Prediction time with Tiny YOLOv3 . . . 59
4.12 Results with Tiny YOLOv3 and threshold 0.5 . . . 59
4.13 Results with Tiny YOLOv3 and threshold 0.25 . . . 60
4.14 Fine-tuning loss error and mAP . . . 61
4.15 General statistics after fine-tuning on Tiny YOLOv3 . . . 62
4.16 Behaviour of average loss error and table of its mean over the last 1000 iterations . . . 63
4.17 Trend of average loss error during fine-tuning and its mean on the last 1000 iterations . . . 64
4.18 Trend of average loss error and its mean on the last quarter of re-training changing momentum . . . 65
4.19 Trend of average loss error varying momentum . . . 66
4.20 Development of average loss error varying decay and its mean on last 1000 iterations . . . 67
4.21 Evolution of average loss error with different burn in and its mean on last 1000 iterations . . . 68
4.22 Trend of average loss error using several resolutions and its mean on last 1000 iterations . . . 69
4.23 Trend of average loss error with different mini-batch size and its mean on the last quarter of training . . . 70
4.24 Overall results after grid-search . . . 71
4.25 Overall results on original Tiny YOLOv3 with threshold 0.25 on JPEG images . . . 72
4.26 Overall results on fine-tuned Tiny YOLOv3 with JPEG images . 73

5.1 Recap on YOLOv3 statistics . . . 74
5.2 Recap on Tiny YOLOv3 statistics . . . 74

6.1 First group of pictures with quite good predictions made by YOLOv3 . . . 79

6.2 Second group of pictures with apples misclassificated as oranges by YOLOv3 81
6.3 Third group of pictures with poor detection made by YOLOv3 . 83
6.4 Comparison of detection, setting threshold on confidence score to 0.5 and 0.25 for YOLOv3 88
6.5 Pictures selected for their good predictions on YOLOv3 91
6.6 Pictures with many misclassified apples in the original YOLOv3 93
6.7 Pictures with poor detection in original YOLOv3 95
6.8 Some pictures got after second fine-tuning of YOLOv3 102
6.9 Results with Tiny YOLOv3 using pictures that gave good outcome with YOLOv3 105
6.10 Results with Tiny YOLOv3 using pictures that produced several wrong prediction of oranges with YOLOv3 107
6.11 Some pictures returned by Tiny YOLOv3 after fine-tuning 112
6.12 Some pictures returned after grid-search on Tiny YOLOv3 118
6.13 Detection with Tiny YOLOv3 on pictures in JPEG format 123
6.14 Detection with fine-tuned Tiny YOLOv3 on pictures in JPEG format 128

Abstract

Nowadays it is necessary to optimize the usage of resources in order to reduce costs and waste, especially in the field of agriculture, and technological solutions can help us to pursue this objective, in particular neural networks are a powerful and adaptive tool. There exist many kind of them, each one with a particular structure and features that make it more suitable to accomplish a specific task. In this thesis I am going to investigate about some types and examples of neural network, focusing on YOLO, a particular convolutional neural network designed for object detection that I exploited to detect apples on images taken in an orchard; I improved it with fine-tuning and grid search, getting a remarkable upgrade in performance respect to the initial outcomes. The ability to improve reveals the potential of this device, indeed it could be re-trained on a custom dataset to detect apples or other fruits and used in the area of precision agriculture, like the harvesting of the only fruits within a range of ripeness, the discard of those fruits inner damaged by bacteria or other pathogen and eventually a complete automatic harvesting led by a rover equipped with a mechanic arm; such system would be helpful for reasons of food safety, optimal usage of resources and time and reduction of human physical efforts.

Ringraziamenti

Sono molte le persone che hanno contribuito e mi hanno sostenuta nella realizzazione di questa tesi, perciò vorrei impiegare questo spazio per esprimere tutta la mia gratitudine nei loro confronti.
In primo luogo vorrei ringraziare il mio relatore, il Professore Vaccarino, che ha creduto in me e nelle mie capacità, mi ha fornito indicazioni utili per lo svolgimento di questo progetto e per la vita futura e mi ha fatto conoscere Vittorio Mazzia e Angelo Tartaglia: grazie per la vostra pazienza e disponibilità; i vostri consigli sono stati fondamentali per potermi destreggiare nell'immenso mondo dell'intelligenza artificiale e poter ottenere dei risultati soddisfacenti.
In secondo luogo desidero ringraziare Luca Nazari per l'opportunità offertami nella sua azienda, il mio tutor Alessandro Monge e Andrea Maccagno per avermi guidato durante il tirocinio e tutti gli altri colleghi per avermi aiutato nelle difficoltà quotidiane, per le risate e le partite a calcio balilla durante la pausa pranzo.
Sicuramente devo citare la mia famiglia: i miei genitori, che hanno reso possibili i miei studi, che hanno gioito per i miei successi e mi hanno sostenuto nei momenti difficili, mia sorella Elisa, che arricchisce la mia vita da quando é nata e i miei cani e gatti, che grazie al loro calore e amore incondizionato sanno come risollevarmi l'umore.
Un pensiero speciale va a Giacomo, presente pressoché dall'inizio di questo percorso universitario. Sarebbe riduttivo dirti semplicemente grazie dopo tutto quello che hai fatto e fai per me: senza il tuo supporto morale, le tue esortazioni e i tuoi suggerimenti probabilmente questo lavoro non sarebbe stato portato a termine; per questo ti sono molto riconoscente.
Un omaggio particolare va a Daniela, la mia insegnante di danza orientale, e alle mie compagne di ballo passate e presenti, per le belle serate passate insieme a chiacchierare e ballare: sono state molto utili ad allentare la mia tensione e a ricaricarmi con nuove energie per affrontare al meglio la settimana.
Inoltre ringrazio mio padrino, che é sempre presente nei momenti più importanti della mia vita e i miei amici per il loro sostegno.
Infine complimenti a te, Debora, per aver affrontato e superato gli ostacoli incontrati nel corso degli studi, per la resilienza dimostrata di fronte ai fallimenti e per la perseveranza con cui hai completato questo progetto. Sii orgogliosa di questo lavoro, dei risultati che sei riuscita a raggiungere ed ora va' ... *"verso l'infinito e oltre!"*.

Chapter 1

Introduction

The research in artificial intelligence (AI), thanks to its versatility and potential, has allowed and will allow more and more in the next years to create new tools in order to optimize industrial processes, avoid car accidents, prevent building collapses, report next failures, select products, enhance medical diagnosis... in short AI helps us in many aspects of life.
Indeed, almost every area can benefit from some sort of solution involving artificial intelligence and in this thesis I will show an application of it in the agricultural field, that is the detection of apples on images taken in a orchard with a convolutional neural network.

1.1 Objective

The agricultural field has always known a constant development through the invention of hoe and plough, the study of crop rotation, soil characteristics and weather observation until the introduction of agricultural machinery, fertilizers and OGM. Nowadays the research is focusing on further areas as the precise monitoring of the field and crop in order to optimize the use of irrigation, fertiliser and pesticide, the use of spectral cameras on fruit to detect damages or infections and to determine the actual ripening and an ever more automatic process of harvesting.
In this work I will investigate about how a neural network can be used to detect fruit on trees, in particular apples, with a view to future application on a rover able to navigate independently the field, day and night, to scan trees with multiple cameras, to determine where the ripe fruits are and to take them with a robot arm.
The main problems with this objective are well described in *"Estimation of the number of apples in color images recorded in orchards"* [1] and they can be summed up in two main criticality: illumination and occlusion.

- As regard the first problem, light conditions affect heavily the results of detection, in fact the sunbeams directed frontally on fruits can cause saturation, so in this case the color information is lost. Moreover they can create shaded zones that make very difficult to recognize apples in them. Therefore it is suggested to work at the sunset, under diffusive sunlight or

to put black sheets behind trees and to use artificial illumination in order to avoid strong changes in light on the field.

Figure 1.1: Example of apples with strong difference in sunlight

- The occlusion problem refers to the fact that in a real situation, apples grow quite randomly on the tree, so they can form clusters or hide behind leaves or branches, making their detection harder.

Figure 1.2: Examples of apples grouped in a cluster, hidden by leaves or divided by branch

1.2 Introduction to Artificial Intelligence

Since ancient times among the principal topics of which philosophers and scientists were thinking about, there were mental processes and not only how to follow a proper reasoning, such as using deduction or syllogism, but also how human brain works, if it can be reproduced and how to manufacture instruments able to amplify mental capacity and to simplify some work.
Artificial intelligence fascinated many writers, cartoonists, directors and game directors so that became a recurrent subject in sci-fi genre, hence we can mention in literature masterpiece as "Frankenstein" by Mary Shelley, "The bicentennial man" by Isaac Asimov, "The voyage of the Jerle Shannara" by Terry Brooks and among manga "Ghost in the shell" by Masamune Shirow. In cinematic history we can consider "Metropolis", "2001: a space odyssey", "Blade runner", "Matrix" and "Io, robot", that have handled with the theme of robotics, androids, power detention and coexistence of human beings and robots, "War

games" describes the consequences that can come from the misusing of an artificial intelligence system, in this case the unaware use of a military computer by a student, "Her" and "Trascendence" deals with the possibility of a computer to feel emotions, to have a love affair with it or to transfer a consciousness into it and "Big hero 6" presents Baymax, a kind of friendly robot, created to take care and provide health assistance to the person assigned. Among video games there are "Portal", in which the AI present in the game leads the player through a series of rooms, where he must find a way out, "System shock", in which the player have to combat an evil AI, willing to destroy humanity and in "Hello neighbor" the AI modifies the villain's behaviour according to the past actions made by the player to hinder him.
These are only few examples of the use of the topic "artificial intelligence" throughout culture, now we will explore the principal steps made by researchers in AI field.

1.2.1 Brief history of AI [2, 3]

As regard the history of artificial intelligence we can enumerate Aristotle, who used syllogism as reasoning in 4-th century B.C., Blaise Pascal in 17-th century invented the Pascaline, a mechanical calculator able to do addition and subtraction, George Boole and Charles Babbage lived in 19-th century, the first is famous for his method based on propositional logic and the second built "The difference engine" which computed differential calculus, and finally Alan Turing devoted his live to logic, theory of computation and the creation of intelligent machines according to his well-known "Turing test".

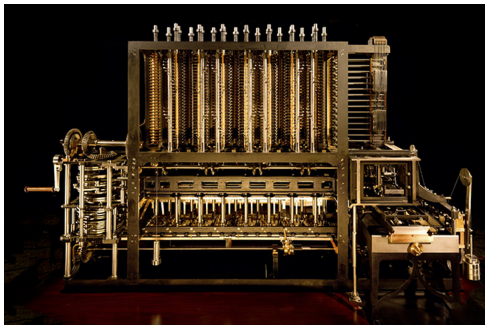


Figure 1.3: The difference engine by Babbage

In the 20-th century, thanks to the progress in neurobiology, information theory and cybernetics, the researchers began to think about the possibility of building an electronic brain and in 1943 Warren McCullouck and Walter Pitt proposed the first work recognised as AI: a model of artificial neuron, that could distinguish two different categories of inputs by testing whether a certain linear function f(x,w) is positive or negative, but the weights had to be set correctly by a human operator.
In the 50's, John McCarthy, Marvin Minsky, Claude Shannon, Nathaniel Rochester, Arthur Samuel, Allen Newell and Herbert Simon asserted themselves with their projects in implementing programs which learn checkers strategies, solve word problems in algebra, prove logical theorems and speak English.

In particular, in 1956 at the Dartmouth conference (conference in which the term "artificial intelligence" was established) the first automatic demonstrator of theorems was presented, the next year Frank Rosenblatt invented the Perceptron, the ancestor of neural networks, made by only one layer and just enough to learn the weights of a linear function without human action and in 1959 Herbert Gelemter designed the "Geometry Theorem Prover", a program that, as the name suggests, is capable of proof theorems of complex geometry.
The expectations in AI were so high that Herbert Simon believed that:

"machines will be capable, within twenty years, of doing any work a man can do" [4, p. 96]

and Marvin Minsky stated that:

"within a generation ... the problem of creating 'artificial intelligence' will substantially be solved" [5, p. 2].

Unfortunately things did not run so smoothly, the issue of teaching a computer to do what people do was much more complicated than expected.
In fact, in 1969 Minsky and Papert in their document "Perceptrons" proved the limitation of the Rosenblatt's Perceptron, presenting as example the failure to solve the trivial XOR problem, due to the non-linearity of the function to learn. Also other promising programs began to show their incapacity to solve more complex tasks and computational costs became unbearable for the technology at the time, thus the majority of researchers turned their efforts towards different areas and started the "AI winter".
In the next years expert systems were developed, such as DENDRAL and MYCIN, respectively able to infer the structure of an organic molecule given its chemical formula and to support medical decision about blood infection based on an incomplete knowledge of symptoms.
In 1975 Paul Werbos managed to solve the XOR problem inventing the back-propagation algorithm applied to a multi-layers perceptron and this algorithm will be used in almost every neural network from then on.
In the 90's the research in AI flourished again, thanks to the increasing computational power and the ongoing scientific discoveries in robotics, medicine, mathematics, physics and economics. In fact it is started again to code about widely differing objectives: logistics, data mining, natural language processing, visual recognition of images, medical diagnosis, etc; this resulted in very effective, high-performance and continually improving programs, to the extent that nowadays many tasks are inconceivable without the aid of computer and it is not foreseen an end to the development of artificial intelligence technology.
Many achievements were presented in public competition, one of the most famous is the ImageNet Large Scale Visual Recognition Challenge (ILSVRC), where each research team tests its algorithm on a given data set of images to classify and who reaches the highest accuracy wins the competition. In such contest was presented AlexNet in 2012, that became a milestone among convolutional neural network (CNN). In the same category we can remember GoogLeNet, ResNet and YOLO.
Whereas for the recognition of handwritten digits, in 1998 Yann LeCun invented LeNet-5, trained on the MNIST dataset.
As regard the development of game strategy, the 11-th May 1997 was a turning

point day for the level of "intelligence" acquired by a machine, indeed Deep Blue, the IBM supercomputer, became the first computer chess-playing system to beat a reigning world chess champion, Garry Kasparov.

Figure 1.4: Frame of the match Deep Blue vs Kasparov

After that, other computers were developed to challenge humans in different games. The most famous matches took place in 2011, when the IBM's question answering system, Watson, defeated the two greatest Jeopardy! champions, Brad Rutter and Ken Jennings, by a significant margin. In March 2016 AlphaGo won 4 out of 5 games of Go in a match with the champion Lee Sedol, beating a professional Go player without handicaps and the next year the same computer program won a three-game match with Ke Jie, ranked number one in the world in Japan Go Association's.
Recently chatbot have appeared, which are softwares able to have some sort of conversation with human, especially created for online assistance.

Chapter 2

The neural networks

The basic unit for artificial intelligence project is the neural network, which is a program designed to fulfill a given task, learning how to do it step by step and improving with time.

2.1 The biological model

When someone thinks about the definition of "artificial intelligence", an answer could be "a machine that can undertake different human tasks that need a strategy to be solved, that can learn from errors to improve and gain experience", therefore it is reasonable to consider to design this machine like a human brain and to reproduce its functionalities.
Indeed a neural networks is a set of neurons connected each other in different ways, that receive inputs, elaborate them, produce an output that spreads through the network and finally take a decision.
Now let's go in more detail on what is a neuron, how it works and how it communicates with the other neurons present in the human brain.

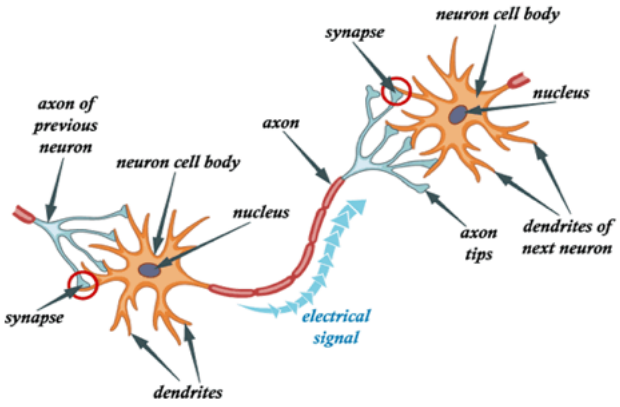


Figure 2.1: Human neurons' structure

A neuron is composed by a central cell body or soma, surrounded by many short branching filaments, named dendrites, and only one long nerve fiber,

named axon. The soma contains the nucleus, which is the processing centre where the information coming from other neural units, passing through dendrites, is computed. Then the process result is one or more neurotransmitters, electric signals made by ions of different chemical, such as sodium, potassium, chloride and calcium, that are sent through the axon, which terminates in synapses, nearby the dendrites of the next neuron. Here a voltage gradient across these membranes is maintained, so that the ions can alter this gradient; if this change is big enough, the ion channels allow the neurotransmitters to pass the membrane, then they are received by the receptors of the next dendrites and the signal is transmitted to the cell body of this new neuron, that will process it and so on.
The signal can be excitatory or inhibitory, which can make more or less likely that the next neuron will produce a similar signal, in order to reinforce the answer to a given stimulus or to relax this reaction.
All these features were reproduced somehow in the artificial neuron's pattern.

2.2 The artificial neuron's structure

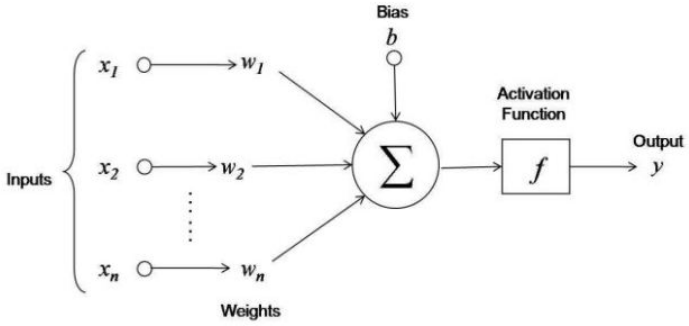


Figure 2.2: Artificial neuron's structure

Each neuron needs:

- inputs x_i
- weights w_i
- bias b
- an activation function f
- a threshold θ
- a cost function C
- an output function o

And it produces an output y.
The inputs x_i can be the outputs of previous neurons that are connected to the actual neuron or can be the inputs of the network itself. Depending on the

objective of our work, the inputs may represent different things, like images, words, texts, sound waves, measured values, etc; consequently they have different forms (numbers or characters) and are often organized in vectors, matrices or tensors.
The weights values change throughout the network training in order to minimize the cost function C by using algorithms like gradient descent or stochastic gradient descent and they have the task of reinforcing or easing the signal received, therefore an high weight represents an excitatory synapse and a low weight an inhibitory synapse.
Then inputs are combined linearly with the weights and it is added a bias:

$$U = \sum_i x_i w_i + b$$

This value is passed to the activation function f, that corresponds in the biological model at the ion channel, that can open the way for the neurotransmitter if the voltage gradient overcomes the threshold θ. Hence if $U \leq \theta, f$ does not "activate", the transmission of that input is stopped (this means that it is considered useless with respect to the task assigned) otherwise the neuron is activated and the input keeps conveying its information through the activation function; finally its output is passed to the output function o (that sometimes is simply the identity function) that returns the final value y:

$$y = o(f(U, \theta))$$

Now the output is spread across the network through the connections between neurons and processed until the reaching of its end and the return the result computed by the network.

2.3 The neural network's structure

A neural network is a set of neurons organized in layers (or levels), where the number of neurons in each layer is set by the programmer and usually varies through the network, also the number itself of layers is not fixed and represents the level of complexity achieved by the network, that is the more layers there are, the more different and elaborate pattern can be distinguished by that model.
The first layer is called input layer because it receives the input and its neurons are the first at processing it, just like the last layer is named output layer because it produces the network's final output. In between these extreme layers stand the hidden layers that contribute at the final result, identifying new patterns, applying linear transformation and changing their weights with the use of a propagation function.
The layers are linked through connections that start from a neuron and end up to another belonging to the next layer and they are associated to a weight that provides the effectiveness of that synapse.
The most basic model is the fully-connected one, that consists of the connection of each neuron to every neuron of the next layer but this makes the model expensive in term of computational costs, especially when the network is very deep and wide, so often techniques, like dropout and norm penalties, are applied to speed up the process or a smaller network can be used, but I will talk about that more in detail later.

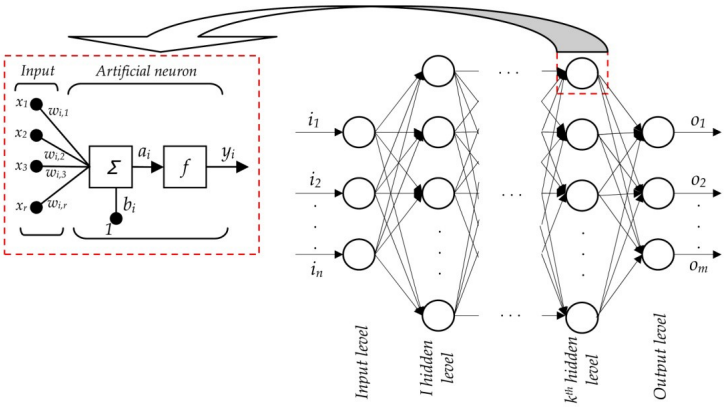


Figure 2.3: A fully-connected network

2.4 Technical details

2.4.1 The activation function

The choice of the activation function determines the evolution of the learning process and thus the effectiveness of the entire algorithm, so it is important to select it correctly, moreover a good activation function should also satisfy some properties, like linearity, continuity, derivability and simplicity in implementation.
Now we are going to outline some of the most used functions for this role. For more details see [6] or [7, chapter 1].

- **Heaviside function**

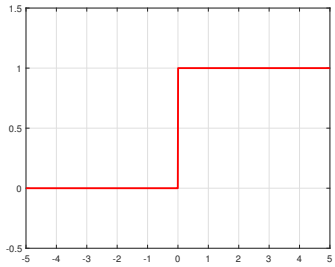


Figure 2.4: Heaviside function

The Heaviside function was the first function used for this purpose because it is a very simple and fast to implement function, it is continuous but not derivable in zero, it makes hard decisions because it is defined as follows:

$$H(x) : \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x \geq 0 \end{cases}$$

therefore its output can only take 0 or 1 values and the turning point is very clear-cut, so it falls easily into error if the value received by the

activation function is near to zero and this was the main reason why it was abandoned and smoother functions are preferred to it.

- **Linear ramp**

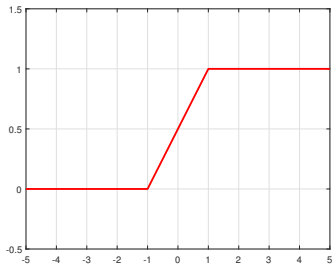


Figure 2.5: Example of linear ramp

The function plotted is an example of the linear ramp:

$$f(x) : \begin{cases} 0 & \text{if } x < a \\ \dfrac{1}{b-a}x - \dfrac{a}{b-a} & \text{if } a \leq x < b \\ 1 & \text{if } x \geq b \end{cases}$$

The linear ramp is continuous but not derivable in two points, that are where the function changes the slope (in our case -1 and 1). It is quite easy to implement and it reaches value one in a less sharp way than in the previous case, so the decision is softer and this made it a good choice as activation function for long time.

- **Logistic sigmoid**

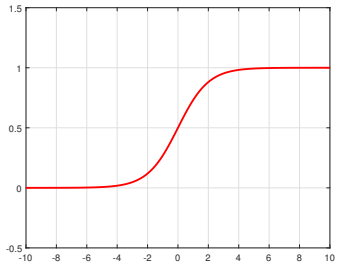


Figure 2.6: Logistic sigmoid function

The sigmoid function has the following form: $\sigma(x) = \dfrac{1}{1+e^{-x}}$

It is a function continuous, differentiable everywhere and its behaviour is similar to that taken by the neuron in biology when it has to decide to transmit the signal or not.
Disadvantages of using this function are its not linearity and the more complexity in implementation than the previous piecewise linear function; indeed it is not very often used as activation function.

- **Hyperbolic tangent**

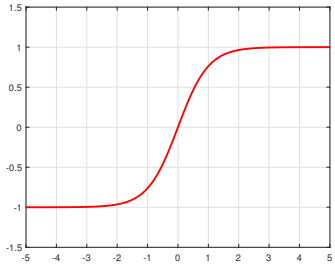


Figure 2.7: Hyperbolic tangent function

The hyperbolic tangent is defined as: $\tanh(x) = \dfrac{\sinh}{\cosh} = \dfrac{e^x - e^{-x}}{e^x + e^{-x}}$

and it is related to the logistic sigmoid by the following dependence:

$$\tanh(x) = 2\sigma(2x) - 1$$

This function is continuous and differentiable everywhere, its peculiarity is the output that can return also negative values because its image is the interval $[-1, 1]$, instead of the interval $[0, 1]$ like the former functions, this allows to avoid "neuron saturation", a phenomenon that happens when most of the domain function returns zero which causes the stop of the learning process for that input, perceived to be useless, but instead of stopping its progress sometimes is preferable to give a little weight to it and the hyperbolic tangent allows this.
This function is not linear and quite complex to compute so it is used only in few cases.

- **Rectified linear unit or ReLU**

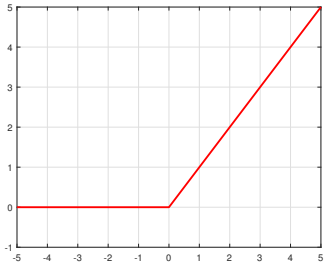


Figure 2.8: ReLU function

A ReLU function has the form: $f(x) = \max\{0, x\}$, it is continuous and not differentiable only in zero, it is piecewise linear, very simple to implement and its image goes to infinite, rather than stop to one as seen before; all these features contribute to make it the most used activation function.
A neuron with this activation function can run into saturation; to avoid this there exist some generalizations of the original ReLU function that

can help for this purpose; some of them consist of adding a non-zero slope α, thus the resulting function is:

$$f(x, \alpha) = \max\{0, x\} + \alpha \min\{0, x\}.$$

Depending on the choice of the value α we can obtain different activation functions:

- **Absolute value rectification**: when $\alpha = -1$ the ReLU function reduces to the absolute value function $f(x) = |x|$. It is usually used for object recognition from images because the neuron should be invariant to a polarity reversal of the input illumination.
- **Leaky ReLU**: α is fixed to a small value, like 0.01
- **Parametric ReLU or PReLU**: α is a learnable parameter

Another generalization of ReLU is the **Maxout unit**: the inputs are divided into groups of k values, each one assigned to a maxout unit that returns the maximum element of the set. These kind of units are useful to learn piecewise linear and convex function with up to k pieces. If k is big enough any convex function can be approximated with a good accuracy.

- **Softmax Unit**

  The softmax function is defined as:

$$softmax(x)_i = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}$$

  This formula can be seen as a probability distribution of a Multinoulli random variable with i possible values over n total trials, so it computes the probability of event i.
  For this reason the softmax function is often used in classification problem, in particular in the output layer because the result represents the probability of the input to belong to the i-th class.
  Moreover this function is continuous, differentiable and invariant to scalar addition.

2.4.2 Gradient descent

The cost function $C(w, b)$ measures the error made by the neural network in processing an input sample; if it returns a small value, you would expect quite accurate results, instead if it returns an high value, it means that the network does not perform well in most cases and it needs some improvements regarding its structure, its parameters or its learning algorithm. The cost function can be defined in several ways, the most used are:

$$\text{Mean Squared Error (MSE): } C(w, b) = \frac{1}{2n} \sum_{i=1}^{n} ||y_i - \hat{y}_i||^2,$$

where y_i is the desired output and $\hat{y}(x_i)$ is the estimated one, and:

$$\text{Cross-entropy (CE): } C(w, b) = -\frac{1}{n} \sum_{i=1}^{n} [y_i \ln \hat{y}_i + (1 - y_i) \ln (1 - \hat{y}_i)].$$

In classification problem it is common to use a binary cross-entropy (BCE) cost function that differs from the above equation only for the values taken by y_i, that are 0 and 1. According to which value y_i takes, the cost function simplifies to one of the two terms and since logarithm is an increasing function, to minimize it is equivalent to minimize its argument, so the formula reduces to:
$\begin{cases} -\frac{1}{n}\sum_{i=1}^{n} \hat{y}_i & \text{if } y_i = 1 \\ -\frac{1}{n}\sum_{i=1}^{n}(1-\hat{y}_i) & \text{if } y_i = 0 \end{cases}$

Indeed the objective is to minimize the cost function and it can be done in multiple ways, one of the first and simplest method used is the gradient descent algorithm.
The theoretical justification behind this algorithm is that if we choose whatever function $f(x) \in \mathbb{R}$ and calculate its derivative $f'(x)$, then

$$f(x - \varepsilon sign(f'(x))) \leq f(x)$$

so we can follow the opposite direction of the derivative, making small steps with opportune values of ε, named also learning rate, until $f'(x) = 0$, i.e. when x is a stationary point.
A stationary point can be a local minimum or maximum, global minimum or maximum or a saddle point; in our case we are looking for a global minimum but it is not always simple to distinguish it from a local minimum or a saddle point, especially when we are working in a very high dimensional space, where is impossible to have a graphical idea of the function evolution.

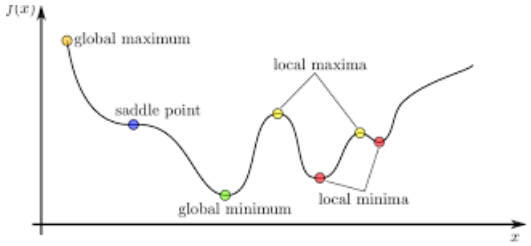


Figure 2.9: Stationary points

That's why the choice of the learning rate is of crucial importance, because it has to be big enough to escape from flat regions, like those that can be near saddle points, to avoid being stuck in a local minimum and to converge in a reasonable time, but it also has to be small enough to not skip the global minimum. It's a trade-off between convergence's speed and accuracy.

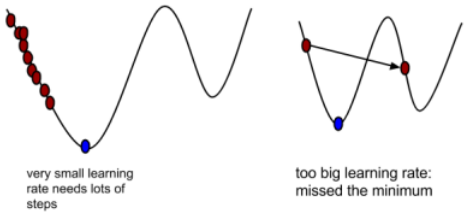


Figure 2.10: Examples of learning rate's choice

Usually the learning rate is set to a small value and decreased during the

learning process or we can use the line search strategy, whereby we can try different values of it and take only the one that gives the smallest objective function value.
In a multidimensional space the derivative has to be replaced by the gradient $\nabla_{x_i} f(x)$, which is a vector containing all the partial derivatives in respect to the component x_i. So a stationary point becomes a point where each partial derivative is zero so its gradient is the null vector and we can reach it moving towards the opposite direction of the gradient:

$$x' = x - \varepsilon \nabla_x f(x)$$

Thus going back to our case, this formula is used during the network training to update weights w_i and bias b_i:

$$w_i' = w_i - \varepsilon \nabla_{w_i} C(w, b)$$

$$b_i' = b_i - \varepsilon \nabla_{b_i} C(w, b)$$

Actually we are never sure to have reached the global minimum and not a local one but we can settle for it if it gives a very low cost function value.

2.4.3 Stochastic gradient descent

The stochastic gradient descent is an enhancement of the previous algorithm to limit the computational cost required to calculate the gradient of the cost function during the training of a large dataset.
In fact we can think about the gradient as an expectation and this value can be estimated using only a small dataset, instead of the entire one. So at each step of the algorithm we sample a mini-batch of examples from the training set of magnitude m $B = (x^1, x^2, ...x^m)$, usually a small number varying from 1 and few hundred, that represents a little part of the original dataset.

$$\nabla C = \frac{1}{n} \sum_{i=1}^{n} \nabla_{x_i} C(w, b) \approx \frac{1}{m} \sum_{j=1}^{m} \nabla_{x^j} C(w, b)$$

Thus the weights and bias are updated according to this new way of calculating the gradient:

$$w_j' = w_j - \varepsilon \frac{1}{m} \sum_{j=1}^{m} \nabla_{w^j} C(w, b)$$

$$b_j' = b_j - \varepsilon \frac{1}{m} \sum_{j=1}^{m} \nabla_{b^j} C(w, b)$$

After the calculation of all weights and bias, a new mini-batch is sorted from the remaining samples of the training set and the resulting estimated gradient, weights and bias are computed repeatedly until all the inputs are used in this algorithm. The union of mini-batch is called epoch and when one of this epoch is completed, a new one is started until all the planned epochs (several thousands) are completed.
The advantage of this algorithm is that does not depend on the size of the training set but only on m, the size of the mini-batch, so it is very used nowadays due to the largeness of the training set needed and the corresponding computational cost requested.

2.4.4 Backpropagation algorithm

As we have seen so far, the information flows from the input layer through the network at the output layer, so this is called forward propagation. But information can also flow backwards and the backpropagation algorithm does just that; this is also to reduce the computational cost of calculating the gradient. Now the gradient is computed using the chain rule, a technique which allows to compose efficiently simple operations.

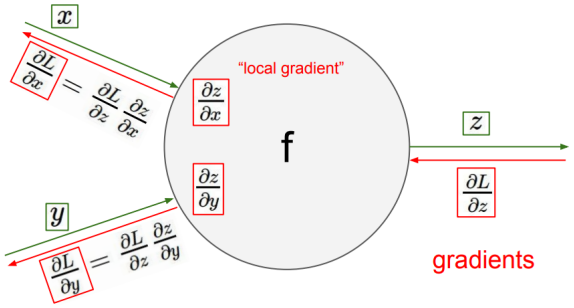


Figure 2.11: Chain rule on a neuron

In fact the cost function of a neuron depends on the following neurons that are linked to it, so the gradient is the product of all these subsequent partial derivatives.

$$\nabla_{z_i^{(l)}} C(w,b) = \sum_i \nabla_{z_i^{(l+1)}} C(w,b) = \sum_i \prod_{j=l+1}^{n} \frac{\partial C(w_i^{(j)}, b_i^{(j)})}{\partial z_i^{(j)}}$$

where $z_i^{(l)}$ represents the i-th neuron in the l-th layer.
Obviously it is more convenient to compute each partial derivative individually, save its value and pass it to its parent node, in this way we can optimize the gradient calculus and avoid repeating the same computations.
In particular, it should be noted how this algorithm updates weights. Let's start from the output layer and make explicit the computation of the cost's gradient in respect to the weights, remembering that $y_i^{(l)} = f\left(\sum_i w_i^{(l)} y_i^{(l-1)}\right)$, where f is the activation function:

$$\nabla_{w_i^{(l)}} C(w,b) = \frac{\partial C(w,b)}{\partial y_i^{(l)}} \frac{\partial y_i^{(l)}}{\partial \left(\sum_i w_i^{(l)} y_i^{(l-1)}\right)} \frac{\partial \left(\sum_i w_i^{(l)} y_i^{(l-1)}\right)}{\partial w_i^{(l)}} =$$

$$= -(d_i^{(l)} - y_i^{(l)}) f'\left(\sum_i w_i^{(l)} y_i^{(l-1)}\right) y_i^{(l-1)}$$

So the adjustment of the weights has to follow the opposite direction of the gradient and to be proportional to the learning rate η:

$$w_i^{(l)}(t+1) = w_i^{(l)}(t) - \eta \, \nabla_{w_i^{(l)}} C(w,b) = w_i^{(l)}(t) + \eta \, \delta_i^{(l)} y_i^{(l-i)},$$

with $\delta_i^{(l)} = (d_i^{(l)} - y_i^{(l)}) f'\left(\sum_i w_i^{(l)} y_i^{(l-1)}\right)$.
This procedure is repeated for the previous layers in a similar way, only a bit

more complex, until the first one is reached.
So in the inner layers we obtain that weights are updated according to:

$$w_i^{(j)}(t+1) = w_i^{(j)}(t) + \eta\ \delta_i^{(j)} y_i^{(j-1)} \qquad j \in [1, l-1]$$

with $\delta_i^{(j)} = -\left(\sum_k \delta_k^{(j+1)} w_k^{(j+1)}\right) f'\left(\sum_i w_i^{(l)} y_i^{(l-1)}\right)$.
If $j = 1$ then we are considering the first layer so new weights depend on the input:

$$w_i^{(1)}(t+1) = w_i^{(1)}(t) + \eta\ \delta_i^{(1)} x_j.$$

The use of this algorithm combined to that of the stochastic gradient descent can make the computational cost in the training phase much lighter than usual. Moreover there exist optimized versions of this algorithm, the most common one uses a momentum parameter.

Backpropagation algorithm optimized with momentum parameter

In the original algorithm the updating depends only on the previous time step, instead in this version it depends on the previous two time steps, that reflects a major attention to the evolution of the cost function.
In practical terms weights change following the equation:

$$w_i^{(j)}(t+1) = w_i^{(j)} + \eta\ \delta_i^{(j)} y_i^{(j-1)} + \alpha\left(w_i^{(j)}(t) - w_i^{(j)}(t-1)\right),$$

where α is the momentum and it belongs to the interval $[0, 1]$; naturally if α is set to zero the algorithm turns back to its original version, instead the more it tends to one, the more regularization is applied, which benefits the convergence process.
As we can notice from the formula above, if weights experienced a big change in the previous steps, this means that the solution is still far, so it is convenient to move fast and in this case the momentum term helps to do a bigger step towards the minimum.
On the contrary, if weights change very little in the last moves, this outline the closeness to at least a local minimum, so it should do a little step in order not to overcome the stationary point; in this case the momentum term gives a very low contribution so the update is determined mainly by the learning term.
In conclusion the momentum term helps the algorithm's convergence, modifying dynamically the magnitude of the step taken according to the recent history in the process.

2.4.5 Regularization techniques

A neural network is considered good when it classifies well not only the training and validation set but especially the test set, which contains samples never seen by the network, so it is important to reach a low test error, even at the expense of the training error; for this purpose regularization techniques were introduced.

Norm penalties

One of the most simple and used strategy of this kind consists in adding a norm penalty $\Omega(w, b)$ to the cost function $C(w, b)$:

$$C(w, b) = C(w, b) + \alpha\Omega(w, b)$$

where $\alpha \in [0, \infty)$ is the hyper-parameter that determines how strong is the penalty imposed to the objective function; obviously if α is set to zero, no regularization is applied.
Furthermore $\Omega(w, b)$ is usually only a function of weights, this means that only weights are regularized and bias are left unchanged.
The most used penalty functions are:

- L^2 **regularization**: $\Omega(w) = \frac{1}{2}\|w\|_2^2$
  This method is also known as weight decay or ridge regression or Tikhonov regularization and it consists of encouraging the weights to go towards the origin.

- L^1 **regularization**: $\Omega(w) = \|w\|_1$
  This kind of regularization tends to reset many components of the vector w, so this corresponds to a features selection.

Data augmentation

In order to obtain good performance, a neural network needs a big amount of training samples consistent to the task assigned and different enough to give a broad spectrum of all the variations present to identify the objective; it is not always possible to get such a big and various dataset, so it is necessary an integration with data augmentation.
This method implies the applying of transformations to the original samples, that increase the variability of the dataset without changing the correct answer. For instance if our dataset is composed by images, we can augment it applying translation, rotation, flip or changing brightness and contrast of each original image.
Another form of data augmentation, very used with sound waves samples, is adding noise to the original dataset because a neural network has to be robust to the presence of noisy samples, so the noise's injection can affect positively its performance.

Early stopping

Early stopping is used when during the training phase, we observe that the training error keeps decreasing while the validation error starts to increase considerably for several epochs, this behaviour can alert the beginning of overfitting, so it is advisable to stop training and keep the parameters achieved when the validation error was minimum.

Bagging or bootstrap aggregating

This technique is classified as an ensemble method because it involves the training of several models and their combination to obtain the final model.
Specifically from the original dataset of n instances k new datasets are created, as large as the original one but where each input is sampled with replacement from the initial set (so these new sets can contain duplicate of the same examples); then each new dataset is trained on a different model and finally tested on the test set to get the best model.
This strategy is effective in reducing the general error because different models

do not make the same error on the same test set, furthermore the training errors are independent so the resulting average model performs at least well as each of its members.
The price to pay for the improvement is the increment in computation and memory, so this is an aspect to take into account if the resources available are limited.

Dropout

Dropout try to approximate bagging over a large number of neural networks, all coming from a bigger one and generated by switching off randomly some of its units. Dropout requires a learning algorithm based on mini-batch to build these sub-networks; each time an input is loaded into a mini-batch a binary mask is sorted and applied on input and hidden layers of the original network. This mask generates the sub-network composed only by the units that are multiplied by one because the others incur into a switch-off with the multiplication by zero, and the sub-network is trained as usual. The masks are sampled independently to each other and the hyper-parameter that defines the probability to include a neuron is set before the beginning of the training, usually an input unit is kept in the sub-network with probability 0.8 and an hidden unit with probability 0.5. Differently from bagging, with this strategy models are no more independent because they share parameters, inheriting them from the parent network; this sharing system allows to train a smaller number of these sub-networks, saving time and computation. Dropout and bagging are often utilised together to take advantage of the efficiency of the first method and the precision of the second one.

Chapter 3

Types and examples of neural networks

In the past decades many different types of neural networks were developed in order to accomplish their various tasks. Below I am going to consider some of these categories and to present their most famous examples. For more information see [8] and [7].

3.1 Fully-connected neural networks

One of the most simple and intuitive kind of neural network is the fully-connected one; each neuron has to be connected to all the neurons present in the next layer.

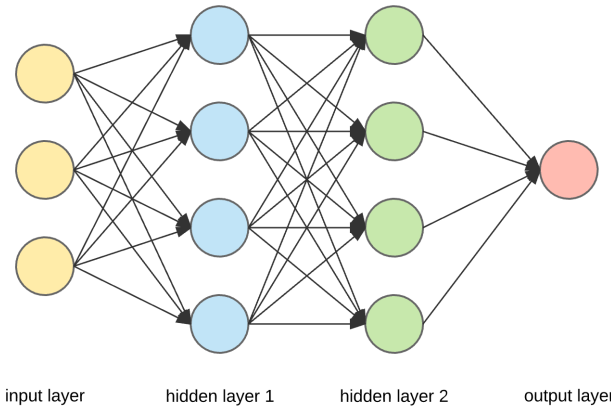


Figure 3.1: Example of fully-connected neural network

This feature makes it likely the most used neural network especially at the start of the AI epoch and nowadays by the beginners of this field, because it gives you the serenity of not worrying about which connections have to be set between neurons, since all the possible connection are switched on and during the training phase the associated weights will change autonomously, strengthening or loosening that connection.

The number of layers and neurons is arbitrary but this has the drawback of making the process very expensive in terms of computational cost if you choose to build a big neural network, because at each connection corresponds at least one parameter and the more deep or large is the network, the more connections are needed. So it is often preferred not to create an entire fully-connected network but to use this structure only in few layers and to lighten the remaining part of the network with strategies that require less connections.

3.1.1 The Perceptron

One of the most famous neural network is the Perceptron, proposed by Frank Rosenblatt in 1958.
It is a binary model able to learn simple patterns, like the logic function AND, OR and NOT. In its original version, it receives only two inputs but it can be generalized to more than that. These inputs x_i are combined together with a linear function $f = \sum_{i=1}^{n} x_i w_i + b$ and according to a threshold λ, it returns the final output $y = \begin{cases} 0 \text{ if } \sum_{i=1}^{n} x_i w_i + b \leq \lambda \\ 1 \text{ if } \sum_{i=1}^{n} x_i w_i + b > \lambda \end{cases}$.

The bias b can be considered as the weight w_0 associated to the input $x_0 = 1$, so we can rewrite the previous formulas in a more compact way: the function is $f = \sum_{i=0}^{n} x_i w_i$ e the output $y = \begin{cases} 0 \text{ if } \sum_{i=0}^{n} x_i w_i \leq \lambda \\ 1 \text{ if } \sum_{i=0}^{n} x_i w_i > \lambda \end{cases}$

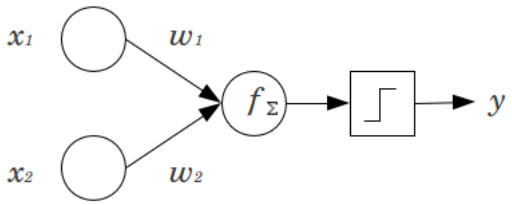


Figure 3.2: Perceptron's model

The big innovation in this model is that the weights w_i are determined by a learning algorithm and not fixed manually anymore, as happened previously.
Initially weights are small random variables, then they are adjusted by a simple but quite effective learning algorithm.
Given the input x and its components x_i, the true output a and the output estimated by the linear function y, we can compute the error with the delta rule, defined as the difference between the output values, $\delta = y - a$ and how much to vary weights, $\Delta w_i = \delta \eta x_i$, where $\eta \in [0, 1]$ is the learning rate.
So, if the predicted output is correct, no changes are made on the weights, otherwise they are updated according to: $w_i = w_i + \Delta w_i$.

Geometrically the correct classification of input consists of finding a line (or a plane if we have three input units, or an hyperplane if we have four or more input units) dividing the space in two parts, each containing only one class, therefore classes have to be linearly separable.
Observing the following plots, we can notice that the classes composed with logic function AND, OR are simply separable (and there is an example of line separator), instead with XOR are not.

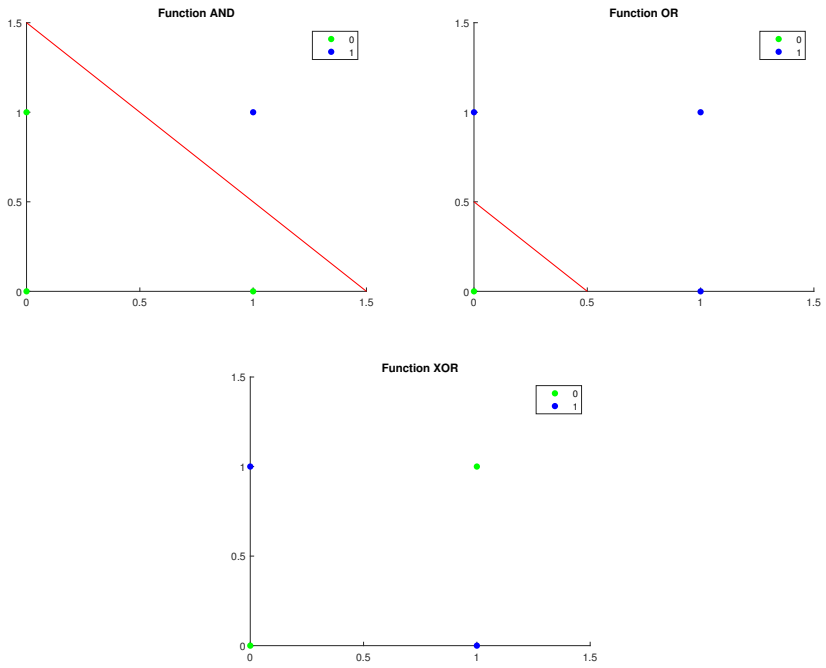


Figure 3.3: Logic operators

Minsky and Papert understood this limitation of the Perceptron and in 1969 they published a book, in which they presented the XOR example and this represented a severe blow for artificial intelligence development.
Later in 1986 Rumelhart, Hinton and Williams proved that this obstacle can be overcome by adding an additional layer to the original Perceptron, thus the Multilayer Perceptron (MLP) was born.

3.1.2 Multilayer Perceptron

The Multilayer Perceptron can be seen as many perceptrons organized in layers, in fact it is a fully-connected network with at least one hidden layer, can have two or more input units and can classify an arbitrary number of classes, reflected in the number of output units.
In contrast to the original perceptron, it follows the backpropagation algorithm for training and the neurons can take both linear and nonlinear activation function, like sigmoid, hyperbolic tangent or logistic function.
The major complexity in structure and flexibility in activation function's choice allows the network to divide the space into more sophisticated parts; in fact

with no hidden layer the network can only distinguish classes linearly separable, introducing one hidden layer it can separate convex regions and with two or more hidden layers it can crop regions arbitrarily complex, in that case the only limitation is given by the number of neurons used.
Resuming the XOR problem, the insertion of an hidden layer with a linear activation function represents a linear transformation of the input space into a new space that makes the classes separable.
One possible numerical solution is that one proposed in [8, p. 171-177], obtained applying weights $w = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$ and bias $b = \begin{bmatrix} 0 \\ -1 \end{bmatrix}$ to the input $x = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix}$ and using ReLU as activation function.
So $xw + b^T = \begin{bmatrix} 0 & -1 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix}$ and employing the ReLU function we get the new coordinates of the original inputs: $\tilde{x} = max(xw + b^T, 0) = \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix}$. Now it's easy to verify that the points are linearly separable in the new configuration.

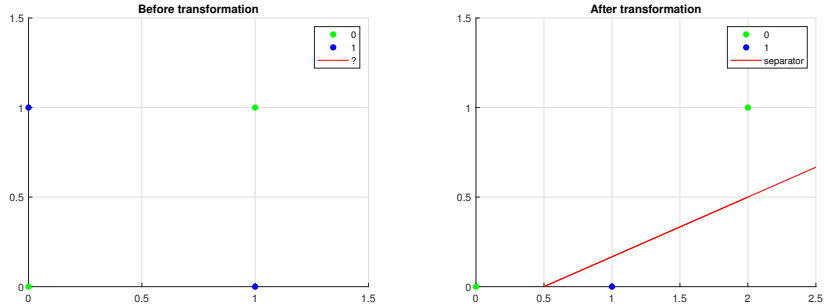


Figure 3.4: Space's transformation for the XOR problem

3.1.3 Linear Associator

The linear associator is an easy and not so modern network able to learn associations between input and output patterns.
It is composed only by input and output units, fully-connected by the weights matrix, can take only the binary values {-1; 1} as input and the learning algorithm follows the Hebb rule or the delta rule, that is the weights are given by the outer product among the input pattern i_p and the output pattern o_p with a learning parameter η: $w = \eta o_p \cdot i_p^T$.
So when a new pattern is tested, we obtain the following output: $o_t = w \cdot i_t = \eta o_p \cdot i_p^T \cdot i_t$, thus the inner product $(i_p^T \cdot i_t) \in [-1, 1]$ gives a similitude measure of the two vectors, in fact if it is equal to:

- 1 it means that the vectors are equal, so they represent the same pattern

- -1 the vectors are opposite and the patterns are complementary
- 0 the vectors are orthogonal and the patterns are unrelated

This argument can be extended to the case of more than one association to learn, where the weights matrix will be a combination of all the associations (i_{p_k}, o_{p_k}): $w = \eta \sum_k o_{p_k} \cdot i_{p_k}^T$ and the output of a test pattern is a combination of the learned patterns: $o_t = \eta \sum_k o_{p_k} \cdot (i_{p_k}^T \cdot i_t)$.
Here lies the limitation of this tool because it can identify correctly a pattern only if the learning vectors are orthogonal to each other, otherwise hybrid patters will be learned and there will be no way to return at the original pattern in the test phase. So the total number of learnable associations is strictly limited by geometrical motivation and that makes this model improper to use in many real application; for this reason some variations and improvements were proposed, like the Hopfield Nets (see subsection 3.2.1).

3.2 Recurrent neural networks

The recurrent neural networks (RNN) are a family of networks designed for processing sequential data, for instance in speech or handwriting recognition, bioinformatics and time series.
It has a main characteristic: the sharing parameters across the network. This allows a more general approach to data, because it can accept input of different length and form; for example in speech recognition the input length is given by the number of words in a sentence, that obviously changes from time to time; moreover a particular word can appear in different positions of the sentences, therefore a model aimed to this problem has to demonstrate flexibility in dealing with inputs so varied. This feature is realized by the presence of backward edges pointing to the hidden units, that make its graph cyclic and no more only feedforward, so that each output is a function of the previous output computed: $h(t) = f(h(t-1),\ x(t))$, where the variable t not necessarily refers to time but to a general step in the learning algorithm.

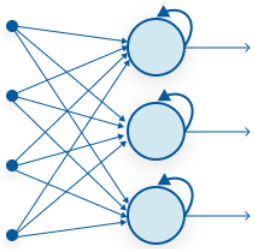


Figure 3.5: General RNN's structure

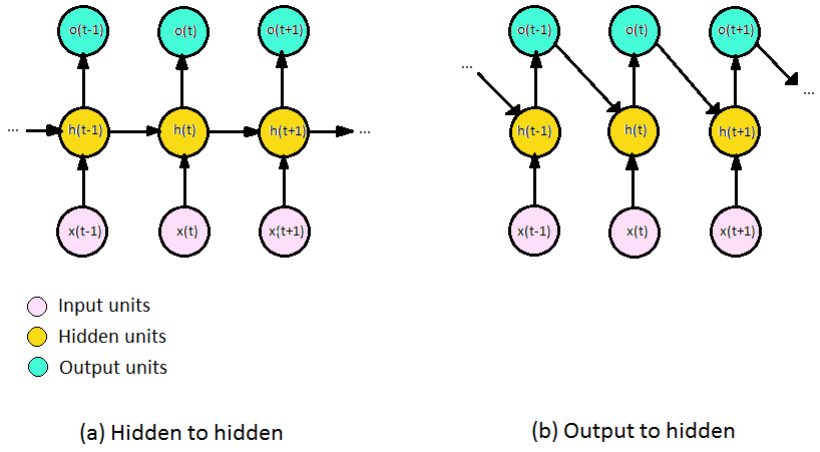
According to which units backward edges link, different RNN can be created, here some examples are presented:

- hidden to hidden: this structure strengthens the training so it is a very powerful network, but also expensive in computational cost, due to its interconnections. Moreover the hidden layer can return output at each time step or only at the final time step

- output to hidden: this model is less effective than the previous one because the task to catch all the necessary information and to transmit it to the next node is assigned to the output units (instead of the hidden units, as in the past case), but it is faster to train because each time step can be computed independently from the others, so this results in a greater parallelization of computation

- bidirectional: there are at least two hidden layers, one that propagates the information forward in time and the other that propagates back the information from next time step, so that at each iteration the output can benefit from both past and future knowledge. This structure is very useful in speech recognition because a single word can have more than one meaning, so looking at the entire sentence is necessary in order to understand which is the corrected one, hence it is important to look not only at the words before the monitored word, but also at the subsequent words.

- recursive: it is a generalization of RNN with a deep tree's structure. The advantage in using a recursive network instead of a traditional recurrent one is the computational cost, if we consider an input of length l, the cost is reduced from $O(l)$ to $O(\log l)$, which also helps in dealing with long-term dependencies. In general it is preferred to use a balanced binary tree but other choices, depending on data, can be done. Actually the best decision regarding the tree structure to use should be inferred by the network itself but this solution often is not feasible. The recursive RNN is suitable for processing data structures, like in natural language and computer vision.



(a) Hidden to hidden

(b) Output to hidden

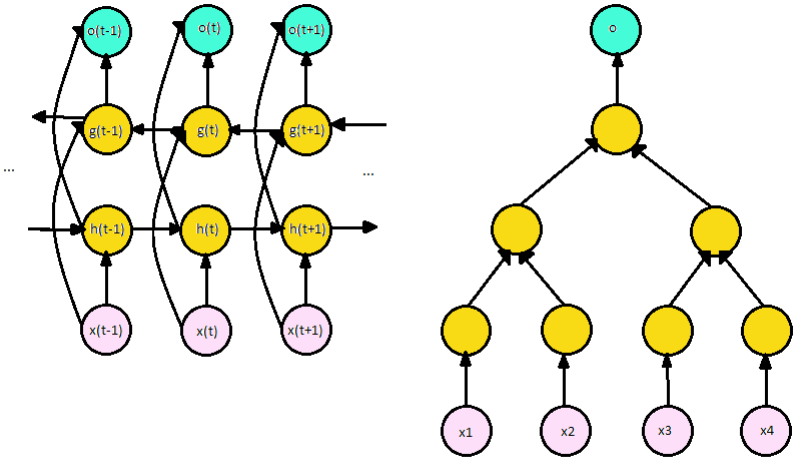


Figure 3.6: Different types of RNNs

3.2.1 Hopfield Net

The Hopfield neural network owes its name to its inventor, John J. Hopfield, who presented it in 1982.

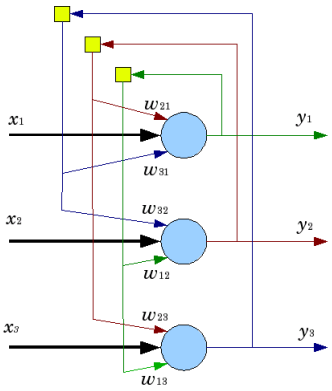


Figure 3.7: Hopfield network's structure

The network is composed by a single layer with a variable amount of recurrent neurons fully connected, that can take the binary values $\{0;1\}$ or $\{-1;1\}$, it is associated with an energy function $E(x)$ as cost function to minimize and the weight matrix is symmetrical with zero values on the diagonal. Its components are updated during the training according to the following algorithm, given the set of k vectors, representing the input patterns x^k:

- $w_{ij} = \sum_k (2x_i^k - 1)(2x_j^k - 1)$ for $i \neq j$ if the input values are $\{0;1\}$
- $w_{ij} = \sum_k x_i^k x_j^k$ for $i \neq j$ if the input values are $\{-1;1\}$

After setting the weights, the output units are initialized with the test pattern: $y_i(0) = x_i$. Then the following computation is iterated until the output converges: $y_j(t+1) = f\left(x_j + \sum_i w_{ij} y_i(t)\right) = f(\tilde{y}_j) = \begin{cases} 1 & \text{if } \tilde{y}_j > \theta_j \\ y_j(t+1) & \text{if } \tilde{y}_j = \theta_j \\ 0 & \text{if } \tilde{y}_j < \theta_j \end{cases}$
where f is a non-linear function with threshold θ. Then the output $y_j(t+1)$ is spread through the network to all the other units.

The convergence of this process is verified by studying the energy function of the network:

$$E(x) = -\frac{1}{2} y W y^T + \theta y^T$$

It is a quadratic, non-increasing and bounded function, so the algorithm leads the network to a local minimum, that is a stable equilibrium point, decreasing its energy.
The Hopfield Nets are more powerful than the simple linear associator because they don't require the input patterns to be orthogonal, but only linearly independent; it is estimated that this kind of network can memorize nearly 0.15N different patterns, where N indicates the amount of neurons used.
For example it can be used to recognise digits or character, as shown in the next example where the network can correctly identify and rebuild the pattern of the digits proposed.

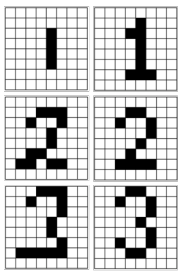


Figure 3.8: Example of application with Hopfield network: in the first column are shown the digits proposed to the network and in the second column there are its answers

Moreover it is also suitable for classification; in this case inputs are elements of the considered classes and the final output has to be compared with the classes to decide which one it belongs to.

3.2.2 Boltzmann machines

The first Boltzmann machine was invented in 1985 by Geoffrey Hinton and Terry Sejnowski [9].
It consists of a stochastic recurrent neural network based on an energy function, so it is kind of a non-deterministic version of the Hopfield net. It is composed by two types of units: visible v and hidden or latent h. The first ones fill in the input layer and the second ones the hidden layer (in the simplest model there is only one hidden layer but there exist more sophisticated models that arrange the latent units in more than one layer, generating a deep neural network).

Typically this network receives n-dimensional binary random inputs x; on

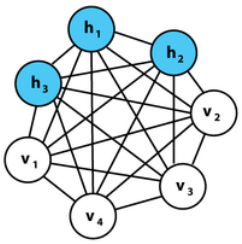


Figure 3.9: General Boltzmann machine's model

them the joint distribution is defined:

$$p(x) = \frac{exp(-E(x))}{P},$$

where $E(x)$ is the energy function and P is the partition function over the input set that specifies the probability measure.
The energy function is given by:

$$E(x) = -x^T W x - b^T x,$$

where W is the weight matrix and b is the bias vector.
We can rewrite this formula according to the decomposition of visible and hidden units:

$$E(x) = -v^T R v - v^T U h - h^T S h - b^T v - c^T h.$$

The training process is driven by a simulated annealing algorithm, based on maximum likelihood condition. This choice makes the updating weight depending only on the statistics of the linking neurons, so the learning rule is said to be "local".
This procedure has a biological explanation, indeed it reflects that is more plausible to think that a neuron cell is triggered by another cell directly connected to it, rather than by a cell further located from it.

As previously said, this is only the original and simpler version of a Boltzmann machine, afterwards other models of these networks were presented. The most used are briefly described below:

- **Restricted Boltzmann machines**
  In the simplest version, it is structured with a layer of visible units and other of hidden units, but this base model can be stacked to another to generate a deeper network. The visible neurons are usually fully-connected to the hidden ones, but there are no connections between units on the same layer, thus it forms a bipartite undirected graph.
  The joint probability function is equal to the previous case:

  $$p(v, h) = \frac{exp(-E(v, h))}{P},$$

  and the energy function reduces to:

  $$E(v, h) = -b^T v - c^T h - v^T W h,$$

  due to conditional independence of the units in the same layer.

- **Deep belief networks**
  It has an hybrid model with undirected connections linking the first two layers and directed connections among the remaining ones; hidden units are arranged in multiple layers, fully-connected to the others but with no inner connections; they usually receive binary inputs and the visible units can deal with binary or real data.
  Due to its complexity in training, it is not very used today but it still remains an important model that gave significant results, like in the recognition of handwritten digits on the MNIST dataset.

- **Deep Boltzmann machines**
  It has an undirected graph, with several hidden layers and no intralayer connections, it is bipartite with the odd and the even layers forming the two groups, so they are mutually conditional independent.
  This feature speeds up the training process because weights can be updated in two steps, one for the odd layers and the second for the even ones.

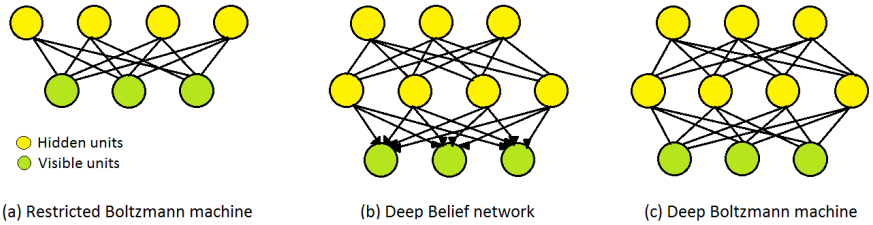


Figure 3.10: Other Boltzmann machines

3.3 Self-Organizing Map: Kohonen Maps

In 1984 Finnish Teuvo Kohonen presented the Kohonen self-organizing network; it consists of a single-layer network, where neurons are fully-connected to the input, output and each other; it learns with a competitive algorithm and it is suitable for pattern recognition and clustering problems.
The units can be organized in line or, more frequently, in a grid and the distance between them determines which weights update during the training phase as shown in the previous image using different colours for the neurons.
In fact the competitive strategy that the network follows is characterized by the search of the most similar neuron to the input sample presented; it represents "the winner" and it is rewarded by updating its weight; but also the neurons near it show a certain similarity to the input, so they deserve some rewards, that will be lower but however sufficient to change their weights.
Specifically the interaction radius R is fixed; which defines how far a neuron can affect the other, so this determines the neighborhood Ω_i of every neuron i, then it is computed the distance between the input x and every neuron with the Euclidean norm: $dist(x, w^{(i)}) = \sqrt{\sum_{j=1}^{n}(x_j - w_j^{(i)})^2}$; the i-th neuron that

minimizes the previous distance is selected and its value is updated by using: $w^{(i)} = w^{(i)} + \eta(x - w^{(i)})$. Also the weights of the i-th neighborhood have to be changed and that can be done by following this rule:

$$\begin{cases} w^{(\Omega_i)} = w^{(\Omega_i)} + \frac{\eta}{2}(x - w^{(\Omega_i)}) & \text{if } R = 1 \\ w^{(\Omega_i)} = w^{(\Omega_i)} + \eta e^{\frac{-\|w^{(i)} - w^{(\Omega_i)}\|^2}{2\sigma^2}} & \text{if } R > 1 \end{cases}$$

In the first case the reward given is exactly half of the winner one and when $R > 1$ to use a Gaussian function is preferred, which spreads the reward around the winner with decreasing intensity.

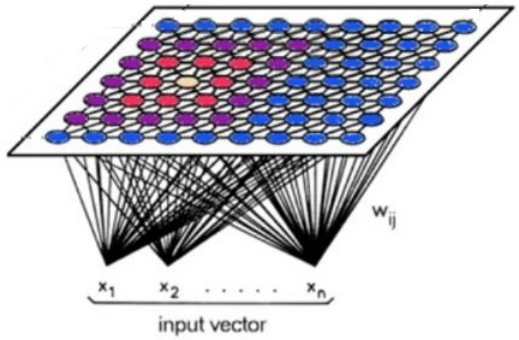


Figure 3.11: Kohonen Maps' structure

The training stops when no significant changes happens to the weights. At this point, if we are considering a pattern recognition problem, each neuron represents an association among input and output and during testing new samples will be identified with the most similar pattern; instead, if we want to create clusters, the final result of the training is indeed the grouping of neurons in classes representing each cluster, so when a new input is tested, the output will be the class at which the most similar neuron belongs to.
This network is a good alternative to k-means because the best k value is not fixed in advance but estimated via learning.

3.4 Convolutional Neural Networks

Convolutional neural networks (CNNs) are suitable for processing data structured in grid-form, like time series and images; for this reason I have chosen a network of this type to develop my project.
The adjective "convolutional" derives from the mathematical operation called convolution, which is performed at least in one layer of a CNN in place of the simple matrix multiplication. In general convolution is an operation between two integrable functions defined on $\mathbb{R}$ and it can be expressed as:

$$z(t) = (f * g)(t) = \int_{-\infty}^{+\infty} f(\tau)g(t - \tau)d\tau,$$

where the first function f is known as input and it is a multidimensional array of data, the second one g as kernel and it is a multidimensional array of parameters and the result z as feature map.
For computational reasons we discretize the domain of the functions f and g, now they have non-zero values only on the same finite set of points; we can rewrite the previous formula as:

$$z(i,j) = (f * g)(i,j) = \sum_m \sum_n f(m,n)g(i-m, j-n).$$

Convolution is a symmetrical operation, so we can swap the indexes of the two functions without changing the result. Sometimes this property is exploited to implement a sort of different version of convolution, that is the cross-correlation function:

$$z(i,j) = (f * g)(i,j) = \sum_m \sum_n f(i+m, j+n)g(m,n).$$

CNNs are based on three important ideas: sparse interactions, parameter sharing and equivariance of representations.

- **Sparse interactions**
  It means that connections among units are sparse thanks to a kernel smaller than the entire input, which means that at each convolution the kernel is applied to a small part of the input, so less parameters are needed and the consequent memory and computational requirements are reduced.
  In practice if we take into account an image as input, it is composed by thousands or millions of pixels but only small groups of them depict meaningful features, like edges or corners, so to highlight them it is needed a kernel of a similar size.

- **Parameter sharing**
  Like the name suggests, it consists of sharing parameters through the network and using them multiple times instead of once, like happens in other traditional neural networks. Also this technique contributes in reducing the total amount of parameters necessary.

- **Equivariance to translation**
  This property is derived directly from the previous characteristic and it states that convolution is invariant to translation, so you can apply a translation then a convolution or vice versa and the result does not change.
  This property is extremely important when processing time-series data or images because a shift in time or space must not alter the skill to detect relevant features, that actually most of the time occur more than once in each input data.
  On the contrary convolution is not invariant in respect to other transformations, like rotation or changes in scale of an image and other expedients are used to get the invariance respect to these transformations.

Going deeper on the structure of a convolutional layer, we can discover that it can be subdivided in three stages: convolution stage, detector stage and pooling stage.

The first step consists in performing several convolution operations, called also filters, in parallel to produce a set of linear activations, each one corresponding to a feature extracted. In the detector phase a non-linear function, often ReLU, is applied to each of the previous output. Finally it is used a pooling function to determine the total output of the convolutional layer.
The pooling function returns an aggregate value of a given position and its neighbourhood. This global value can be achieved computing the maximum, the simple average, a weighted average or the L^2 norm of the values considered. The size of the neighbouring set, called kernel, is an arbitrary choice, usually it takes form of a small square, like 3x3 or 4x4. Then it has to be determined the stride, that is how many locations the algorithm have to skip in horizontal and vertical direction before applying the next pooling function.
The output size depends on the input dimension N, the kernel size K and stride S according to the equation: $\frac{N-K}{S} + 1$, so the choice of a big kernel or a stride greater than one reduces the output size and consequently the computational cost; for instance a stride equal to two nearly halves it, lightening the workload. This is why a stride greater than one is commonly used.

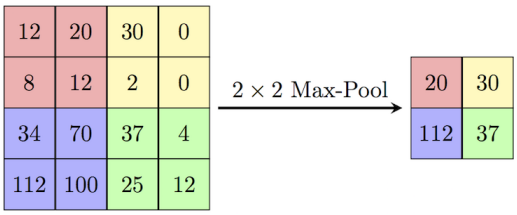


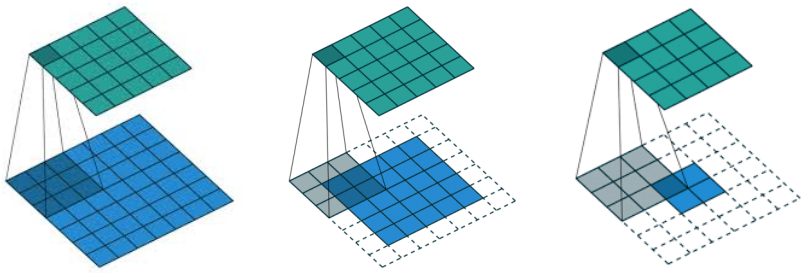
Figure 3.12: Example of max-pooling function on 2x2 regions with stride = 2

In addition to this advantage, the pooling function helps to strengthen the network's ability to be invariant to translation and to achieve invariance towards other transformations, like image rotation.
On the other hand the selection of big values for stride o kernel size causes a drastic reduction in output size, limiting the number of convolutional layers usable in the network, therefore also the number of detectable features. In order to release this restriction, a zero-padding is applied to the input to enlarge it, that consists of adding zeros on the input boundary; the quantity of zeros determines the kind of padding used:

- **Valid convolution:** it is the "basic" version with no zero-padding, so the output pixels depends only by the same number of input pixels because kernel considers no additional padding. With this choice the pixels near the boundary are used less times than the others so their contribution to the output is little.

- **Same convolution:** in this kind of padding enough zeros to keep the size of the output equal to the size of the input are added. For instance, if we use a kernel of size (k x k) and stride equal to one in every direction, we have to add $k - 1$ zeros around the boundary. This choice increases the use of pixels near the boundary respect to the previous case.

- **Full convolution:** is an extreme case that allows each pixel to be considered exactly k times, so the image with stride equal to one will be padded with $k - 1$ zeros on each side.



(a) No zero-padding (b) Same convolution (c) Full convolution

Figure 3.13: Examples of different zero-padding, everyone with kernel 3x3 and stride 1

Actually the most used zero-padding is a tradeoff between valid and same convolution because it combines the advantage of a lower computation burden with a less radical shrinking of input image.
Furthermore, the use of convolution instead of matrix multiplication allows the network to deal with input of different sizes, because kernel can be applied as many times as necessary to process the whole sample.
Now some examples of famous CNNs are presented.

3.4.1 LeNet-5

LeNet-5 [10] is a milestone in handwritten character recognition, actually it has reached an accuracy of 99.2% on the MNIST dataset and it was implemented by Yann LeCun in 1998.
Surprisingly it has a quite simple and not very deep structure, indeed it is composed by only 7 layers, where convolution and average pooling take turns three times, then it ends with a fully-connected layer and an output layer, that returns the final response.

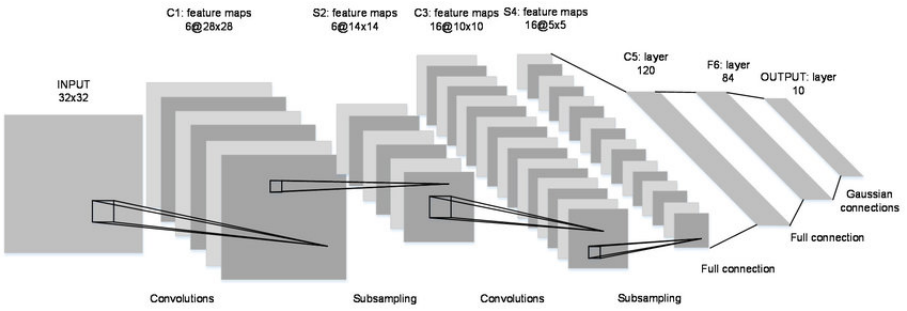


Figure 3.14: LeNet-5's structure

LeNet-5 takes only B/W image of 32x32 pixels as input, it is multiplied subsequently by 6, 16 and 120 kernels in the convolution layers, alternating with two subsampling layers. The convolution layers use a kernel of size 5x5 with stride equal to one and for the average pooling, kernel of 2x2 with stride equal to two. In the fifth layer each feature map is flattened to a single pixel, next they are fully-connected to the next layer that connects to the final layer, composed by 10 units, as the number of digits existing in our alphabet. The softmax function as been selected as activation function in the last layer and the hyperbolic tangent function for the previous layers.
One peculiarity of this network is that second and third layers are not fully-connected to each other but links are carefully chosen, as described in Figure 3.15. The first six feature maps receive three consecutive inputs, the next six four consecutive inputs, the next three four non-consecutive inputs and only the last feature map works with all the outputs of the second layer.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	X				X	X	X			X	X	X	X		X	X
1	X	X				X	X	X			X	X	X	X		X
2	X	X	X				X	X	X			X		X	X	X
3		X	X	X			X	X	X	X			X		X	X
4			X	X	X			X	X	X	X		X	X		X
5				X	X	X			X	X	X	X		X	X	X

Figure 3.15: Table of connections between second and third layer

This decision allows a minor number of connections, so a minor computational cost and also greater prediction accuracy because each unit receives a different combination of inputs that implies the extraction of different features and the improvement of the network's ability to distinguish characters.
Finally, Euclidean radial basis function units are utilized in the last layer to determine which digit the sample represents. They compute the distance between the output of the sixth layer and the parameter vectors, that take binary values -1 or 1 and depict stylized representations of the digits. The character that minimizes this distance will be the network's answer at the recognition question.
LeNet-5 was trained on a modified version of the MNIST in which the original images, normalized to 20x20 pixels, were re-normalized to fit 28x28 pixels and extended with background pixels to 32x32 images, then new images derived from the previous ones were added and they were applied random distortions. It was trained for 20 epochs with a dynamic learning rate, that decreases gradually from 0.0005 to 0.00001 and the remarkable error rate of 0.8% was reached.

3.4.2 AlexNet

AlexNet [11] was created to compete in the ImageNet Large Scale Visual Recognition Challenge (ILSVRC), a competition that requests to recognize 1000 classes of different objects and in 2012, when Alex Krizhevsky entered AlexNet in the challenge, it reached a top-5 error of 15,3%, more than ten percentage points far from the runner-up.

AlexNet is composed by 8 layers, specifically 5 convolutional layers and 3

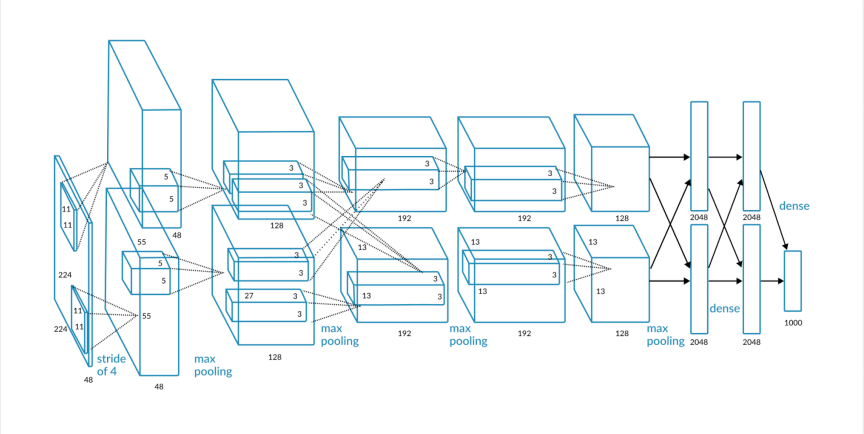


Figure 3.16: AlexNet's architecture

fully-connected layers. It starts receiving 224x224 color images in RGB color code (each color is called channel and they determine the depth of the input, as drawn in Figure 3.16) and applying a convolution of kernel 11x11 with stride 4 on each channel. Then there is a sequence of convolutional layers with max pooling function and eventually are placed fully-connected layers, integrated with a dropout of probability 0.5 in order to reduce overfitting. The output layer has 1000 neurons, as the number of different objects that the network can detect. More details about AlexNet's architecture can be pulled from Figure 3.16.

A ReLU function is used as activation function throughout the network and a softmax function for the output layer. Pooling functions are designed to overlap one another, employing a stride minor to the kernel size and this improves detection accuracy of 0,4% and 0,3% respectively for top-1 and top-5 error rates. The top-1 error rate is the fraction of how many correct labels do not match the predicted labels and top-5 error rate is the fraction of how many correct labels are not among the five labels considered most probable by the network.

As training set it was selected a partition of the ImageNet dataset, that originally contained over 15 millions labeled images belonging to approximately 22000 classes, therefore only the images including the categories present in the ILSVRC were picked up; this resulted in a dataset of 1.2 million training images, 50.000 validation images and 150.000 testing images. Then they were resized to the desired dimensions (256x256), from which random patches of dimension 224x224 are extracted, together with their horizontal reflection, in order to increase the variability in the training set. Another technique to perform data augmentation involves changing the intensity of the RGB channels of the sampled patches.

At this point these patches were used for the training on two GPUs (that explains the split in two pipelines in the figure 3.16) to speed up the training time and reduce the workload on each machine.

It was opted for a stochastic gradient descent with batch size of 128 examples, momentum of 0.9, weight decay of 0.0005 and an initial learning rate of 0.01, that decreases of an order of magnitude for three times.

After 90 epochs of training, AlexNet reached on the ILSVRC-2010 37,5% top-1 error rate and 17.0% top-5 error rate on the test set; a good result especially compared to the best performance achieved during that challenge, respectively 47.1% and 28.2%.

3.4.3 VGG

VGG [12] is the result of the work of Karen Simonyan and Andrew Zisserman. Nowadays it is known as VGG, taking its name from the lab Visual Geometry Group at Oxford, where it was developed, but its original name was ConvNet and it was designed for the ILSVRC-2014.
It is characterized by a series of convolutional layers with an increasing number of filters in each of them, with kernel of small size (usually 3x3) and stride equal to one. It is applied a zero-padding of one pixel to keep the resolution constant. In addition, some convolutional layers are directly connected, others are interspersed by a max-pooling layer with kernel 2x2 and stride equal to two. After that there are three fully-connected layers, implemented with dropout regularization of 0.5 for the first two layers, while the last one has 1000 units to be able to detect all the categories present in the challenge and it uses softmax function to determine the final output. All the previous layers are equipped with ReLU as their activation function.
The network's input is a color image of fixed-size 224x224, with its pixels re-centered on the RGB mean values.
The desired partition of ImageNet dataset (namely only the images containing the classes present in ILSVRC-2014 challenge) was used as training set, then the images were scaled, cropped and submitted to random horizontal flip and random RGB colour shift to augment variability. The network follows mini-batch gradient descent algorithm, with batch size of 256, momentum of 0.9, weight decay of 0.0005 and learning rate starting at 0.01, then decreased by a factor of 10 for three times; it was trained for 74 epochs.
Actually there exist more than one VGG's model, that mainly differ for the number of hidden layers. More details can be found in the comprehensive outline located in the next page (Figure 3.18). The deepest configuration (VGG-19, Figure 3.17) achieved the best performance considering as test set the ILSVRC-2012 dataset, with respectively top-1 and top-5 error rates of 25.5% and 8.0%.

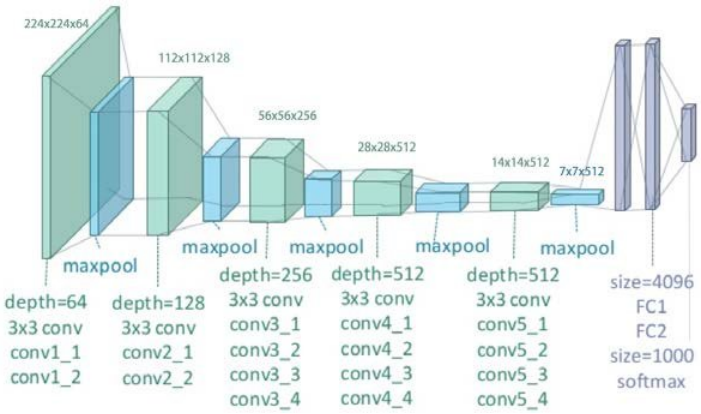


Figure 3.17: VGG-19

ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

Figure 3.18: VGG's configurations

Some years later this project was resumed by Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun to develop a different network, the Residual neural network or, in short, ResNet [13].

3.4.4 YOLO

YOLO [14] stands for "You Only Look Once" and it refers to the fact that this CNN is able to detect objects directly from full images in one evaluation.
To actually succeed in implementing in a single neural network the object detection task was a remarkable result achieved in 2016 by Joseph Redmon, Santosh Divvala, Ross Girshick and Ali Farhadi.
Before this achievement, the existing systems are used to employ a concatenation of structures, where each one has a specific task to fulfill. For instance we can include deformable parts models (DPM), that performs a classifier for each detectable object on a sliding window, which means that at evenly spaced regions of the whole image it evaluates the presence of a specific object; or R-CNN, that generates potential bounding boxes in an image, then a classifier is run on them to determine if an object is present or not, next the remaining bounding boxes are refined, the duplicate detection are eliminated and the

boxes are rescored in respect to the other objects found in the image.
Anyway these pipelines are complex, slow and hard to optimize because they are composed by several parts that have to be trained separately and this affects the overall performance. Instead YOLO manages to generate bounding boxes and to compute class probabilities for them, so its training reflects directly on detection performance and this brings some benefits, like the speed in image processing, so much that this system can claim real-time performance, or the global reasoning, that improves the accuracy in detection and reduces background mistakes, or the generalization of the patterns learnt, that makes the network more robust to custom dataset; whereas its weakness lies in the difficulty of producing accurate object localization, in particular for small objects.
Nevertheless its precision can be enhanced with a specific fine tuning so, given its superiority on many aspects over the current detection system, I have chosen this neural network to conduct my project, specifically I have used the latest version of YOLO and its "reduced" version, that are called respectively YOLOv3 [15] and Tiny YOLOv3.
The complete code is available at *https://github.com/pjreddie/darknet* [16].

3.4.5 YOLOv3

The third version of YOLO strictly follows the novelties and changes made in the second YOLO version, called YOLO9000 [17], combined with a Residual network's structure [13].
Briefly, Residual network or ResNet has two main peculiarities: residual blocks and shortcut connection. Residual blocks are composed by two consecutive convolutional layers with ReLU activation function, followed by a link to the next block and a residual layer that sends the output to a deeper block. Usually it skips the subsequent block and points to the next one, as shown in the figure on the right.
This technique helps to avoid the problem of vanishing gradients and to speed up the learning process. For more details, you can read the report *"Deep residual learning for image recognition"*[13].

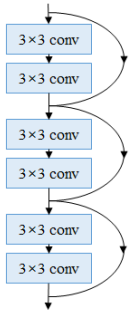


Figure 3.19: Example of residual blocks

Since object detection can be seen as built in two parts (objects classification and localization), also YOLOv3 can be thought as a neural network composed by two elements, each one responsible for a task.
This approach resulted in a very deep neural network, 106 layers, where the first 78 are devoted to features extraction to accomplish the classification task, and the last 28 to identify the position of the objects.

That's an overview of the whole structure.

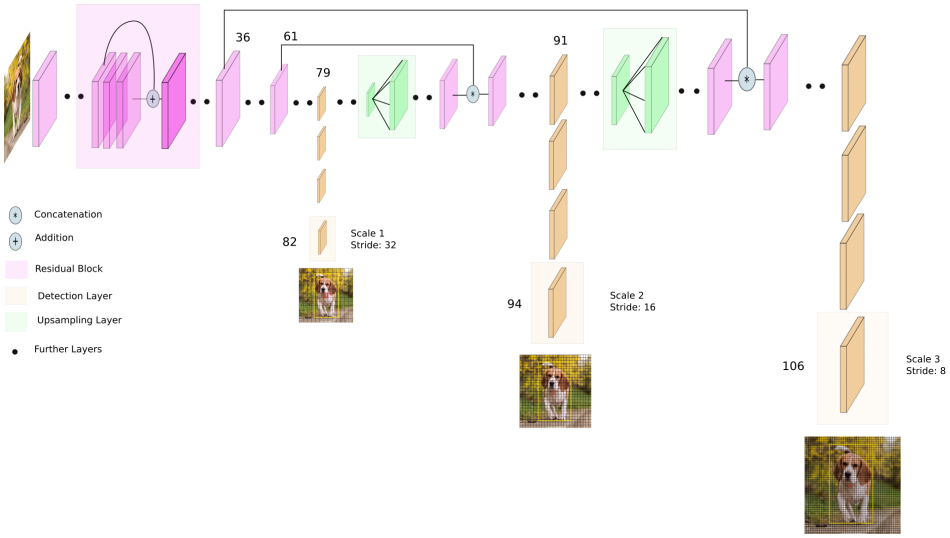


Figure 3.20: YOLOv3's architecture

At first sight we can notice from the above image that YOLOv3 is composed mainly by convolutional layers, some pooling layers that upsample the image and it returns detection at three different scales; but now let's examine what all this means and how it actually works.

Bounding box prediction

In order to identify bounding boxes, after an initial features extraction the network receives an image of default size of 416x416 pixels (but we will see that this resolution is not fixed and can be increased up to 832x832 with an expedient), normalized it taking the top-left corner as origin, then divides it into an (S x S) grid, where each cell is a square of 32x32 pixels, so we obtain a 13x13 grid. Three anchor boxes (prior boxes of different shapes [18]) are centered on each cell and employed as a guideline for drawing a bounding box containing an object for each anchor.
Anchor boxes are rectangles of fixed dimension, centered on the current cell. In Faster R-CNN [18] they are manually chosen, but according to the objects we want to detect (small, thin, large, squared, ...), they can fail or at least make more complex the task of producing precise bounding boxes if they do not fit well the shape of our objects; in other words the Faster R-CNN anchor boxes are inefficient to detect things of various shape and size. So there are basically two options available: let the network learn to adjust boxes during training or optimize them formerly using some machine learning algorithm. Since starting with good priors helps the training in speed and accuracy, instead of learning them during the process, the authors decided to optimize these choices running a k-means clustering on the training set bounding boxes (also known as ground truth boxes), with distance metric:

$$d(ground\ truth\ box, centroid\ box) = 1 - IoU(ground\ truth\ box, centroid\ box),$$

where IoU stands for "intersection over union" and it is the ratio between the intersection of its arguments and the union of them.

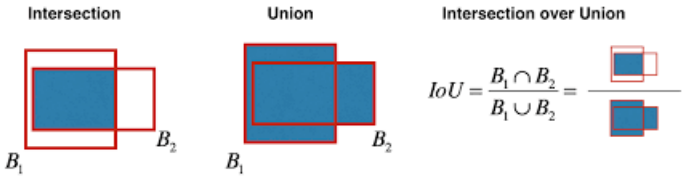


Figure 3.21: IoU definition

With k = 9 the resulting anchor boxes, grouped into three different scale, were: [(10, 13), (16, 30), (33, 23)], [(30, 61), (62, 45), (59, 119)], [(116, 90), (156, 198), (373, 326)]; that were quite different from the priors selected in Faster R-CNN: (188, 111), (113, 114), (70, 92), (416, 229), (261, 284), (174, 332), (768, 437), (499, 501), (355, 715).
The neural network predicts 5 values for each bounding box: t_x, t_y, t_w, t_h, t_o and combining them with the position of the cell considered respect to the image (c_x, c_y), the anchor dimensions p_w, p_h and the sigmoid function, the normalized coordinates of the bounding box respect to the top-left corner of the image and the "objectness score" (a sort of probability of object presence in the bounding box) are obtained:

$$b_x = \sigma(t_x) + c_x \qquad b_y = \sigma(t_y) + c_y \qquad b_w = p_w e^{t_W} \qquad b_h = p_h e^{t_h} \qquad \sigma(t_o)$$

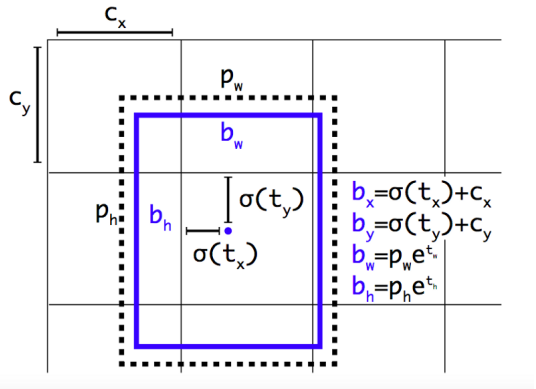


Figure 3.22: Bounding box coordinates

The objectness score $\sigma(t_o)$ gives a measure of how well the predicted bounding box fit the object so it should tend to 1. Then the network predicts conditional class probabilities $\mathbb{P}(class_i|object)$ for each class, that represent the probability of that object of belonging to the class i; if we multiply each one by the objectness score it results in a series of confidence scores, that give an overall measure about the quality and trust of that prediction of being an object of a specific class.
If the objects in images have been labeled, the objecteness score is computed

as $\mathbb{P}(object) * IoU(bb, ground\ truth)$, where IoU(bb, ground truth) is the intersection over union between the predicted bounding box and the ground-truth box and $\mathbb{P}(object)$ is equal to 1 if there is an object and 0 if there is not. If the predicted bounding box overlaps the ground truth box more than any other ones and more than a threshold chosen (0.5 in YOLOv3), then it is associated to the ground truth box and according to the non-maximum suppression principle all the other predictions that overlaps more than 0.5 with the considered ground truth box are discarded, in order to have at most one prediction for every ground truth box. Also predictions with IoU smaller than 0.5 with any ground truth box are discarded.

Loss function

The total error is the sum of a localization error, a classification error and a confidence error, computed over the three bounding box predicted for each of the S^2 cells.

- Localization error measures the distance between the predicted bounding box and its ground truth box associated by calculating its mean squared error (see subsection 2.4.2):

$$\lambda_{coord} \sum_{i=1}^{S^2} \sum_{j=1}^{3} \mathbf{1}_{ij}^{obj} \left[(x_{ij} - b_{x_{ij}})^2 + (y_{ij} - b_{y_{ij}})^2 \right] +$$

$$+\lambda_{coord} \sum_{i=1}^{S^2} \sum_{j=1}^{3} \mathbf{1}_{ij}^{obj} \left[(\sqrt{w_{ij}} - \sqrt{b_{w_{ij}}})^2 + (\sqrt{h_{ij}} - \sqrt{b_{h_{ij}}})^2 \right],$$

  where $\mathbf{1}_{ij}^{obj}$ is equal to 1 if there is an object in the j-th bounding box of i-th cell, otherwise it is 0, λ_{coord} is a parameter set to 5 that helps model stability by increasing the importance of those bounding boxes that contain an object and the square root is applied to both width and height to diversify the weight of errors made on big or small boxes.

- Classification error computes the class probability error on each cell belonging to the ground truth box as binary cross-entropy (see subsection 2.4.2):

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

  $\mathbf{1}_{i}^{obj}$ is equal to 1 if in the cell i there is an object, $p_i(c)$ is the probability of cell i of belonging to the class c as ground truth (so it will be 1 only in one case and 0 for the other categories) and $\hat{p}_i(c)$ is the conditional class probability of class c in cell i.

- Confidence error quantifies the error due to objectness score using binary cross-entropy, where the binary variable and considering both if the object is detected or not:

$$\sum_{i=1}^{S^2} \sum_{j=1}^{B} \mathbf{1}_{ij}^{obj} (C_{ij} - \hat{\sigma}_{ij}(t_o))^2 + \lambda_{noobj} \sum_{i=1}^{S^2} \sum_{j=1}^{B} \mathbf{1}_{ij}^{noobj} (C_{ij} - \hat{\sigma}_{ij}(t_o))^2$$

$\mathbf{1}_{ij}^{obj}$, as previously, is equal to 1 if there is an object in cell i and bounding box j, 0 otherwise, $\mathbf{1}_{ij}^{noobj}$ is defined in the opposite way, C_{ij} is the area of intersection between cell i and ground truth box, $\hat{\sigma}_{ij}(t_o)$ is the objectness score of j-th bounding box of cell i and $\lambda_{noobj} = 0.5$ is a parameter that decreases the weight of cells without objects with the aim of model stability.

So the complete loss function is:

$$\lambda_{coord} \sum_{i=1}^{S^2} \sum_{j=1}^{B} \mathbf{1}_{ij}^{obj} \left[(x_{ij} - b_{x_{ij}})^2 + (y_{ij} - b_{y_{ij}})^2 \right] +$$

$$+\lambda_{coord} \sum_{i=1}^{S^2} \sum_{j=1}^{B} \mathbf{1}_{ij}^{obj} \left[(\sqrt{w_{ij}} - \sqrt{b_{w_{ij}}})^2 + (\sqrt{h_{ij}} - \sqrt{b_{h_{ij}}})^2 \right] +$$

$$+\sum_{i=1}^{S^2} \sum_{j=1}^{B} \mathbf{1}_{ij}^{obj} (C_{ij} - \hat{\sigma}_{ij}(t_o))^2 + \lambda_{noobj} \sum_{i=1}^{S^2} \sum_{j=1}^{B} \mathbf{1}_{ij}^{noobj} (C_{ij} - \hat{\sigma}_{ij}(t_o))^2 +$$

$$+\sum_{i=1}^{S^2} \mathbf{1}_{i}^{obj} \sum_{c \in classes} (p_i(c) - \hat{p}_i(c))^2.$$

Multi-label classification

The choice to use the sigmoid function as classifier instead of the softmax (subsection 2.4.1), as commonly selected, is not random: the application of the softmax function implies that classes are mutually exclusive but in real life an object can belong to several categories (for instance: Border Collie, dog and animal), so to build a more realistic model it is used a multi-label approach. This method allows to merge multiple datasets that have different categories or different levels of details (like a list of mammals and a list of dog breed) and the tool used in this case is WordNet [19]. It is a lexical database which organizes the English vocabulary in a directed graph: words are grouped into set of synonyms (synsets), linked each other by semantic relations.
The datasets utilized in YOLOv3 are ImageNet [20] and COCO [21] and their labels are merged in a tree according to WordNet hierarchy; in the next page is present a piece of the resulting tree.
To perform classification over this new structure it is necessary to compute the conditional probability of every node given its parent node, then to get the absolute probability of a node we have to multiply all the conditional probabilities existing on the path connecting it to the root. During training ground-truth labels are propagated down the tree, so that a node gets the labels of its preceding nodes and during detection the tree is traversed from the root, following the path with highest confidence at each split until some threshold (0.5 in the default case) is reached. At this point it is returned the absolute probability and the specific label of that node as final output.

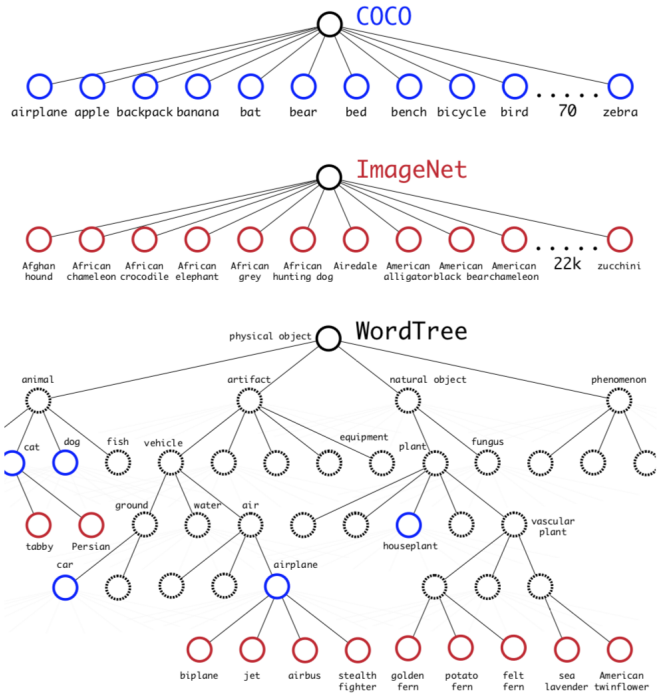


Figure 3.23: Combination of COCO and ImageNet with WordNet

Multi-scale detection

One weakness of the first YOLO version was the struggle in detection of objects of different size, especially the small ones, so in the next versions some changes were introduced to improve its detection performance, namely the generation of predictions at different scales and the use of feature pyramid concept [22].
The first detection is more suitable to identify the larger items because is made only on the 13x13 grid that we have encountered before and it mainly detects coarse-grained features, so at layer 79 after dividing the image in cells, the feature pyramid structure is applied, that is a good component to strengthen detection at different scales because it generates different sized feature maps at multiple levels (that form the "pyramid"), extracting information from each one of them, which are used to determine more bounding boxes and to boost their precision. This construction also involves a bottom-up pathway, a top-down pathway and lateral connections.

- Bottom-up pathway: it consists of creating three feature maps at several scales, each one halved respect to the previous. At every level is associated an anchor of the anchor set assigned to work at this dimension, that operates as described before for the bounding box prediction; then the predicted boxes are passed up to the following level to enrich the structure of details.

- Top-down pathway: it follows the opposite route, going from high to low

pyramid levels, doubling feature maps each time. If on one side this path returns poor detection results, on the other these characteristics are spatially very precise and their locations are sent to the lowest levels.

- Lateral connections: they allow to combine the richer feature maps resulted in the bottom-up pathway with the more accurate feature maps found in the top-down pathway

Then a final layer is queued to return bounding boxes, confidence scores and class prediction for that scale.

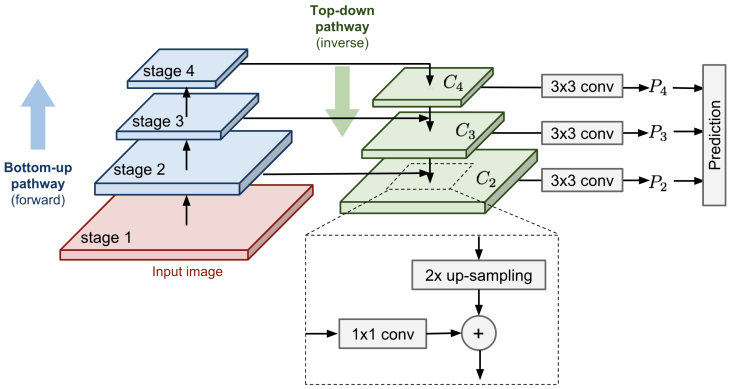


Figure 3.24: Feature pyramid's scheme

This process is iterated at layer 91 and 103, after upsampling the image of a factor of 2 and concatenating it with an earlier layer (the 61-th for the second detection phase and the 36-th for the third), hence a 26x26 cell grid for the second detection scale and 52x52 cell grid for the third are obtained, which are able to seek for finer details of the image and so to discover smaller objects.

The convolutional part

To perform features extraction, required for producing class prediction in the following detection part of the network, it was designed a network taking a cue from YOLO9000 and ResNet named Darknet-53 (because it has 53 convolutional layers). It is structured according to subsequent residual blocks with, towards the end, a global average layer [23], a fully-connected layer containing 1000 units and an output layer that returns the prediction using the softmax function, as we can observe from figure 3.25.

In convolutional layers leaky ReLU is used as activation function (see subsection 2.4.1):

$$\phi(x) = \begin{cases} x & \text{if } x > 0 \\ 0.1x & \text{otherwise} \end{cases}$$

and batch normalization [24] is implemented, a regularization technique that improves convergence, allowing to use greater learning rate, to be less careful about initialization, and makes unnecessary the use of dropout.
Batch normalization applies on SGD algorithm (see subsection 2.4.3) and it

	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 3.25: Structure of Darknet-53

consists of normalizing the input of every mini-batch to have mean of zero and variance of one, so considering a layer with N units and the k-th mini-batch of size m the resulting transformations are:

$$\text{mean: } \mu_k = \frac{1}{m}\sum_{i=1}^{m} x_i$$

$$\text{variance: } \sigma_k^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_k)^2$$

$$\text{normalization: } \hat{x}_i^{(j)} = \frac{x_i^{(j)} - \mu_k^{(j)}}{\sqrt{\sigma_k^{2(j)} + \varepsilon}}, \text{ with } j \in [1, N],\ i \in [1, m]$$

$$\text{normalization transform: } y_i^{(j)} = \gamma^{(j)}\hat{x}_i^{(j)} + \beta^{(j)}, \text{ with } j \in [1, N],\ i \in [1, m]$$

where ε is a constant, arbitrarily small, added for numerical stability, while $\gamma^{(j)}$ and $\beta^{(j)}$ are parameters that have to be learnt by the network, they scale and shift the normalized value for the purpose of keeping the properties of the activation function, i.e. its linearity, derivability, etc.
When we deal with a convolutional layer we have to remember of its invariance to translation and to ensure that our normalization does not alter this aspect, so input are jointly transformed over all the locations present in a feature map of size (p x q) using mini-batch of size (m x p x q) and the learnable parameters $\gamma^{(j)}$ and $\beta^{(j)}$ are related to a single feature map.
Usually for classification in the last convolutional layer feature maps are flattened into a vector, fed into the fully-connected layer and followed by the softmax layer that outputs the predicted class; although for detection is better to

take advantage of the features found in each map, so it is convenient to use a global average pooling, that is a strategy consisting on the computation of the average of each feature map and the transfer of the vector to the next layer. In this way feature maps are enforced to match classes, enhancing correct classification; in addition the lack of parameters to optimize at this stage allows to avoid overfitting, hence global average pooling can be seen as a regularizer.

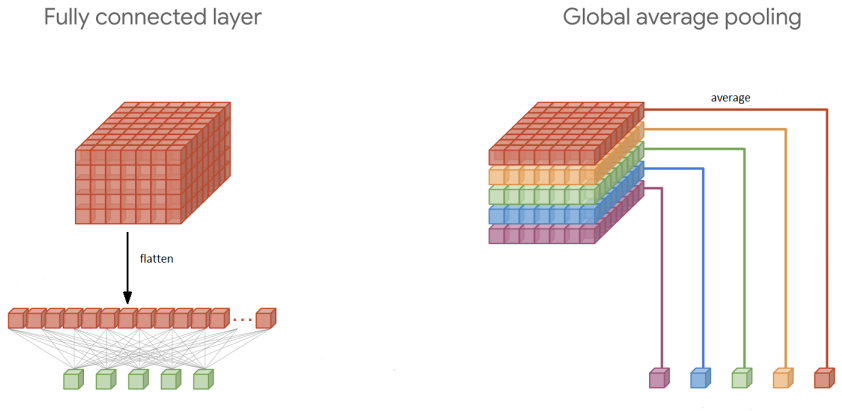


Figure 3.26: Difference between traditional and global averaging approach

Training

YOLOv3 has a flexible structure thanks to its composition exclusively made by convolutional and pooling layers, so it can adapt to images of different size resizing its layout on the fly. This skill is exploited to learn to predict well at various input resolution, therefore during training at every 10 iterations the network changes its dimension picking at random a number between 320 and 832 and divisible by 32.
The network was submitted to two training: one for classification, over the ImageNet dataset containing 1000 categories for 160 epochs, using an initial learning rate of 0.1, decreasing of an order of magnitude for four times, weight decay of 0.0005, momentum of 0.9 and data augmentation like random crops, rotations and shifts in hue, saturation and exposure; the other for detection, over the COCO dataset for 160 epochs, having a starting learning rate of 0.001 then reduced by a factor of 10 two times, weight decay of 0.0005, momentum of 0.9 and data augmentation similar to the previous training. Then YOLOv3 reached mAP of 55.3% on COCO and mAP of 57.9% if the resolution is set to 608x608.

3.4.6 Tiny YOLOv3

Tiny YOLOv3 is basically a small version of YOLOv3: it works in the same way, it alternates convolutional and maxpooling layers for the first 12 layers, that represent the classification part, then detection are produced at two different scale using the feature pyramid model (so in this case the anchors needed are 6, no more 9) and after the first detection scale the feature maps are concatenated

to the outputs of the previous layers (13, 19 and 8), for a total of 23 layers. Its slenderness makes it very fast in processing images of whatever size but at the expense of detection accuracy, thus it requires a fine-tuning for a practical application.

Layer	Type	Filters	Size/Stride	Input	Output
0	Convolutional	16	3 × 3/1	416 × 416 × 3	416 × 416 × 16
1	Maxpool		2 × 2/2	416 × 416 × 16	208 × 208 × 16
2	Convolutional	32	3 × 3/1	208 × 208 × 16	208 × 208 × 32
3	Maxpool		2 × 2/2	208 × 208 × 32	104 × 104 × 32
4	Convolutional	64	3 × 3/1	104 × 104 × 32	104 × 104 × 64
5	Maxpool		2 × 2/2	104 × 104 × 64	52 × 52 × 64
6	Convolutional	128	3 × 3/1	52 × 52 × 64	52 × 52 × 128
7	Maxpool		2 × 2/2	52 × 52 × 128	26 × 26 × 128
8	Convolutional	256	3 × 3/1	26 × 26 × 128	26 × 26 × 256
9	Maxpool		2 × 2/2	26 × 26 × 256	13 × 13 × 256
10	Convolutional	512	3 × 3/1	13 × 13 × 256	13 × 13 × 512
11	Maxpool		2 × 2/1	13 × 13 × 512	13 × 13 × 512
12	Convolutional	1024	3 × 3/1	13 × 13 × 512	13 × 13 × 1024
13	Convolutional	256	1 × 1/1	13 × 13 × 1024	13 × 13 × 256
14	Convolutional	512	3 × 3/1	13 × 13 × 256	13 × 13 × 512
15	Convolutional	255	1 × 1/1	13 × 13 × 512	13 × 13 × 255
16	YOLO				
17	**Route 13**				
18	Convolutional	128	1 × 1/1	13 × 13 × 256	13 × 13 × 128
19	Up-sampling		2 × 2/1	13 × 13 × 128	26 × 26 × 128
20	**Route 19 8**				
21	Convolutional	256	3 × 3/1	13 × 13 × 384	13 × 13 × 256
22	Convolutional	255	1 × 1/1	13 × 13 × 256	13 × 13 × 256
23	YOLO				

Figure 3.27: Tiny YOLOv3 architecture

3.5 Generative Adversarial Networks

Generative adversarial networks (GANs) are the last typology of networks that we will investigate in this work. They were presented in 2004 by Ian Goodfellow and his researchers and they have a very interesting aim: to recreate artificially data similar to the inputs that were showed to during the training [25].
In order to accomplish this task, GANs are composed by two neural networks, named generator and discriminator, that represent the opposing players of a game, in which the generator try to deceive the discriminator by creating false data and presenting to it as genuine. The "game" stops when the generator learns so well to replicate false data that the discriminator is no more able to distinguish them from the true data.
More in detail, it consists of a zero-sum game where generator and discriminator are trained on the dataset Z, that follows a certain distribution $g(z_i, \theta_1)$, then the generator creates new samples x_i according to the input distribution $g(z, \theta_1)$, next true and false data k_i are randomly showed to the discriminator and it produces the probability value $d(k_i, \theta_2)$ that k_i is truly sampled from the training set instead of being artificially drawn by the adversarial network.
The function $v(\theta_1, \theta_2)$ determines the payoff of the discriminator, its opposite $-v(\theta_1, \theta_2)$ the payoff of the generator. Both network try to maximize their payoff

until convergence, that happen when the fake samples are indistinguishable from the real ones for the discriminator, so it return 0.5 as probability of all data. At this point the objective has been reached, thus the discriminator can be discarded and only the generator network is used for further works.
The most common payoff function used is:

$$v(\theta_1, \theta_2) = \mathbb{E}_{k \sim p_{real\ data}} \log d(k, \theta_2) + \mathbb{E}_{k \sim p_{fake\ data}} \log(1 - d(k, \theta_2)),$$

but generally it is difficult or even impossible to maximize if $v(\theta_1, \theta_2)$ is not convex in θ_1 because the convergence process can be unstable and lead to describe circular orbit rather than reach an equilibrium point.
A better formulation of the GAN game to improve its convergence requires that the generator aims to increase the log probability that the discriminator makes an erroneous prediction. The motivation is heuristic: it prevents the generator's cost function in occurring into neuron saturation, even when the discriminator correctly rejects fake samples.
Another approach that gives good results consists in breaking the generation process into steps, more and more detailed, and training the networks on samples that follow a conditional distribution $p(x|y)$.
The choice of proper architecture, hyper-parameters and the use of dropout remain fundamental; for instance very good models are the DCGAN (deep convolutional GAN) and LAPGAN (Laplacian pyramid GAN), as can we see in the example below.

Figure 3.28: Example of DCGAN model used to generate cats' fake images

This kind of networks have huge potential and are often used to replicate images, videos or files audio in several fields, like astronomy, autonomous-driving system and video games.
Unfortunately this potentiality has not always been employed for positive scopes, in fact it can be misused to generate deep fake to incriminate, to discredit or to publicly shame with pornographic files people, often famous or with public offices; for instance in a now popular video you can hear the ex-president of United States Barack Obama to say sentences that actually he has never pronounced [26].

Chapter 4

My project

The objective of my work is to produce an artificial intelligence tool able to detect apples from images of apple trees, therefore I have followed these steps:

- I have chosen a suitable pre-trained neural network
- I have acquired pictures from an orchard to create my test set
- I have tested my dataset on the neural network and produced statistics to determine what is the starting point of my project
- I have created a training set to fine-tune my model and compared the results with the preceding ones
- I have performed a grid-search with cross-validation to optimize model's hyper-parameters
- I have fine-tuned the model with my new configuration on the previous training set and presented the final results.

As I said before, I have selected YOLOv3 and Tiny YOLOv3 as pre-trained neural networks suited for my project for their advantages in terms of completeness on a single device, flexibility to different image sizes, speed processing and good initial precision. YOLOv3 is based on Darknet, an open source neural network framework written in C and CUDA, so it can support GPU computation; I run it on a GPU NVIDIA Quadro M2200, that was provided by Nazari Automazioni s.r.l., the company where I have done an internship to conduct this work. I executed the program on a Linux system by using the command line interface.

4.1 Datasets

To test and re-train these model firstly I had to build datasets suitable for my objective, i.e. to collect pictures containing apples, preferably on trees.

4.1.1 Training set and validation set

Initially I have retrieved from the Open Images v4 dataset [27] only the pictures containing apples labeled individually thank to a toolkit developed by two researchers of Politecnico di Torino: Vittorio Mazzia and Angelo Tartaglia [28]. Then I checked the labelling work made on these images and I found some mistakes (for instance peaches, cherries or plums mislabeled as apples) and stylized apples that are quite different from real fruit (like drawn shapes or the Apple's brand), so I removed wrong labels and misleading images with apple depictions. The remaining pictures formed a training set with 536 samples and a validation set with 23 samples that I used only for the first fine-tuning on YOLOv3.
Next I added to these sets images derived from the same dataset which have multiple apples classified as a unique fruit and I corrected their labelling with LabelImg [29] to have a single apple correspond to each bounding box. Thus I added 313 samples to the training set and 20 samples to the validation set.
Later I also retrieved images from COCO dataset [21], selected those containing apple as label, checked and corrected mislabeling mistakes and eliminated inappropriate pictures, so I obtained 379 training samples and 190 validation samples. Adding up all the previous samples I achieved a training set of 1228 images and a validation set of 233 images that I employed for the rest of my project.

4.1.2 Test set

Since my object is to build a model able to identify apples on trees, it is necessary to test it on images portraying this situation so in a sunny morning of last summer the CEO Luca Nazari, my colleague Andrea Maccagno and I went on an orchard in the countryside near Cuneo to acquire photographs. We used as camera a Canon EOS 60D and we took 617 pictures in total at a resolution of 5184x3456, at different heights from the ground and at a distance of about one meter from the nearest tree trunk depicted. The apple varieties portrayed are: Gala, Crimson Crisp, Golden Delicious, Fuji Raku-Raku and Red Chief.


Figure 4.1: Some examples of pictures taken

Then I labeled all the images with LabelImg, neglecting apples on the background.

4.2 Metrics

In order to determine the quality of a model we have to state the measure in respect to which it can be considered a good or poor algorithm. In classification problems are usually used confusion matrix, accuracy, precision, recall and ROC curve but we are dealing with an object detection problem, that is a bit more complex than a simple classification task, so we have to reassess these metrics:

- True Positive (TP): is an object belonging to the positive category that is correctly classified. In my case, apples correctly detected as apples.
- False Positive (FP): is an object misclassified as positive but it is false. It happens when the neural network mistakes a different object for an apple.
- False Negative (FN): is an object misclassified as negative when it is positive. It represents all the apples not detected.

These measures form the confusion matrix with True Negative (TN), that is an object correctly classified to its negative class, but in our case it does not mean anything because it would include the "not-apple" actually not found, so I kept only the previous measures, not considering TN and its derived metrics accuracy and ROC curve.
A necessary measure to determine if a detection is correct or wrong is the Intersection over Union (IoU), that we have already encountered in YOLO's description. It computes the ratio between the overlapping region and the union region of the predicted box and a ground-truth box and if this ratio is greater or

equal to a threshold then the predicted box is classified as TP if the two boxes have the same label and as FP if labels do not match or if the ratio is lower than the threshold.
Typically the threshold is fixed to 0.5 but it can be increased to a higher value (often 0.75) if we want that only predicted boxes that overlap more strictly the ground truth boxes are considered as TP. FN can be simply obtained by subtracting TP from the total number of ground-truth boxes.
Now that we have defined the basic metrics TP, FP and FN, we can derive other metrics from them.

- Precision: it evaluates how accurate are predictions with the ratio between correct detection and all detection: $\frac{TP}{TP+FP}$
- Recall (or Sensitivity): it represents how good the classifier is to find positives, i.e. correct detections over all ground-truth boxes: $\frac{TP}{TP+FN}$
- F1 score: it computes the harmonic mean of precision and recall to give a more general idea about the accuracy of the model in producing correct detections: $\frac{2 * Precision * Recall}{Precision + Recall}$

In object detection the ROC curve is replaced by the PR curve, where PR stands for Precision-Recall. In the plot are drawn consecutively Precision and Recall of each prediction over all the predictions considered up to that point, so the curve goes up if the prediction is correct or it goes down if it is wrong. Therefore the plot is characterized by a zig-zag pattern with Recall on x-axis and Precision on y-axis. This graph is useful to compute the Average Precision (AV) because it is represented by the area under the PR curve. Since it would be too much complex to estimate the actual area with an integral, the curve it is approximated into a piecewise constant function so that the calculation is reduced to a simple sum of rectangles.

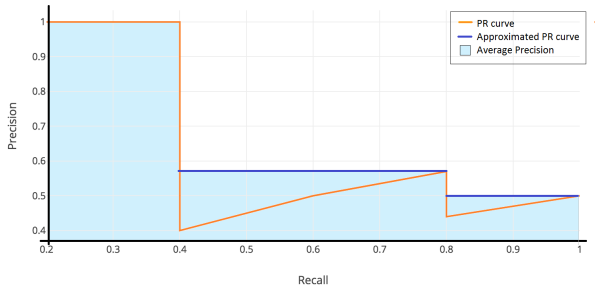


Figure 4.2: Example of PR curve with its approximation

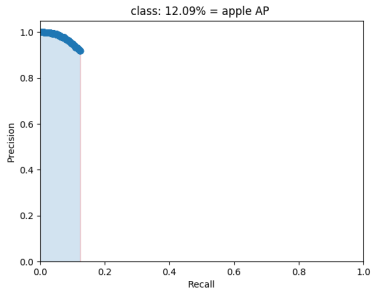
Finally the mean Average Precision or mAP represents the mean of AP computed over all the classes and it is the most used measure for assessing the performance of an object detection model. Since my objective is to detect only one class (apple), AP and mAP coincide. I used the code available at https://github.com/Cartucho/mAP [30] to get the PR curve plot and mAP value.

4.3 Results on YOLOv3

Firstly I processed my test set on YOLOv3. If we consider the complexity of the task assigned due to the presence of foliage and branches that can partially hide apples, the tendency of the fruits to grow close each other that complicates individually detection, the contrast between sun brightness and foliage shade that can greatly vary the hue between near pixels and make green apples similar to leaves or red and yellow ones to oranges and the arbitrariness of angles from which the photos have been taken can cut out partial apples or to highlight some of the previous findings, the results are quite good, especially remembering that this is a general-purpose network, so afterwards it can be forced to focus only on apples, indeed there is considerable room for improvement.
I found three situations looking at the predictions returned by YOLOv3:

- In the first case is perceptible the potential of this neural network, it can detect apples of different sizes, different colors, in different lighting conditions and partially hidden (Figure 6.1).
- To the second case belong pictures with many errors, that are composed almost entirely by the misclassification of apple as orange (Figure 6.2).
- In the third case are present pictures with few or none detection found and this shows the current limitation of YOLOv3 in a real application and the need of a new training (Figure 6.3).

Now let's investigate more in detail these results. I have marked 29356 apples in my test set, YOLOv3 has identified 3960 apples, of which 3641 are true positive respect to IoU ≥ 0.5 and the remaining 319 are false positive. There are other 283 false positive, that are mainly formed by oranges. Then false negatives are 25715, precision 0.858, recall 0.124, F1 score 0.217 and mAP 12.09%.



Ground truth	29356
TP	3641
FP	602
FN	25715
Precision	0.858
Recall	0.124
F1 Score	0.217
mAP	12.09%

Figure 4.3: Overall results of YOLOv3

Regarding time performance YOLOv3 achieves remarkable outcomes; in fact it can process each image in 5/6 seconds and produce the prediction in just over one tenth of a second, precisely the mean prediction time is 0.12 seconds.

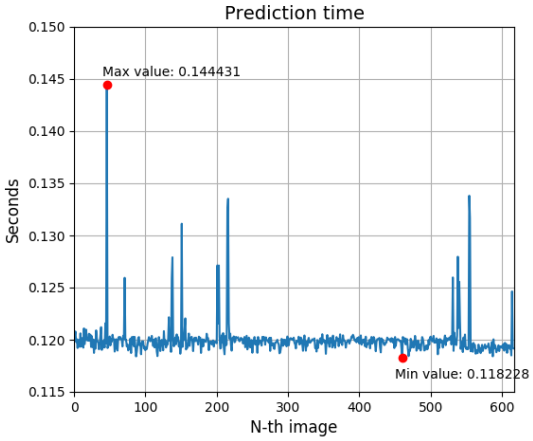


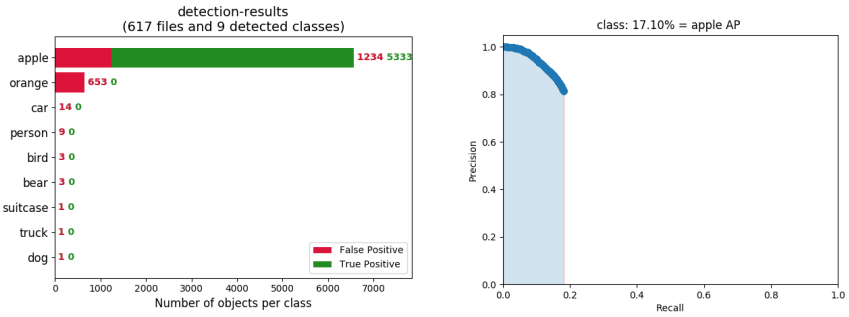
Figure 4.4: Prediction time with YOLOv3

4.3.1 Confidence score set to 0.25

Later I lowered the threshold of confidence score, that determines what is the minimum value accepted to define a prediction, from 0.5 to 0.25. By doing this change I expected an increase in the amount of apples found but also in wrong detection, so I wanted to understand which one of these two aspects would prevail to determine if the choice of a lower threshold can be useful in order to have a better performance.
Comparing pictures with original threshold to those with threshold 0.25, we can note several situations:

- Pictures with almost only apples detected increase their number of apples identified (Figure 6.4 (a))
- Pictures with many misclassified apples worsen their results introducing new wrong predictions (Figure 6.4 (b))
- Pictures with few wrong detection either emphasize this characteristic or keep the ratio between correct and wrong predictions almost constant by raising them proportionally (Figure 6.4 (c))

However the overall outcome is positive because the mAP grows from 12.09% to 17,10% and in general all the other statistics do so, except for precision that suffers from the increment of false positives.



Ground truth	29356
TP	5333
FP	1919
FN	24023
Precision	0.735
Recall	0.182
F1 Score	0.291
mAP	17.10%

Figure 4.5: Overall results of YOLOv3 with confidence 0.25

The performance in time is almost equal to that of the previous case, nearly 6 seconds to process the image through the network and on average 0.12 seconds to output predictions.

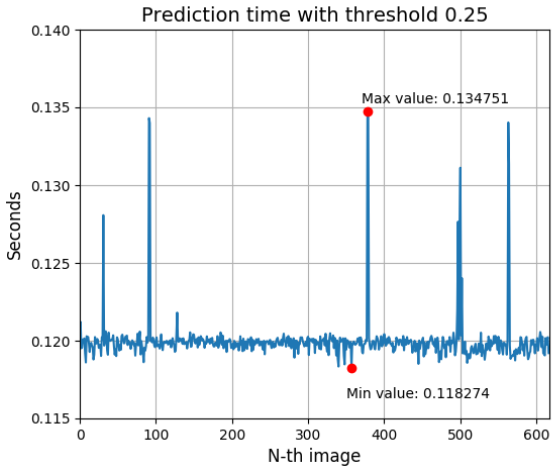


Figure 4.6: Prediction time of YOLOv3 with threshold of confidence score 0.25

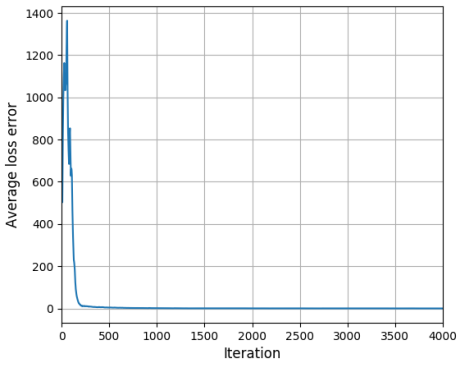
4.3.2 Fine-tuning YOLOv3

Now that we know what is the starting point of YOLOv3, we can try to improve its achievements with a fine-tuning, so I followed the advice provided at

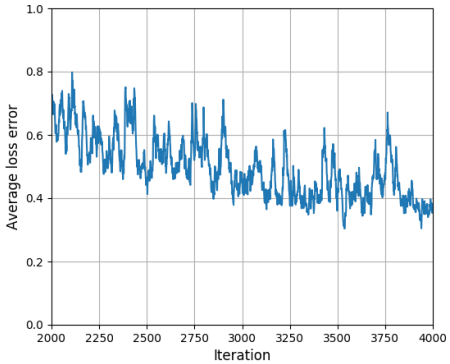
https://github.com/AlexeyAB/darknet#yolo-v3-and-yolo-v2-for-windows-and-linux [31]. Therefore I set class to one, number of filters for detection to 18 (according to the presence of a unique class), height and width to 608 because higher resolution allows more precision, batch size equal to 64 without subdivision due to resources constraints and I used the pre-trained weights of YOLOv3.

First fine-tuning

I remind that the first training set was retrieved from Open Images v4 dataset, dividing it in 536 samples for the actual training set and 23 samples for the validation set. I trained YOLOv3 for 4000 iterations, that corresponds to almost 458 epochs, saving weights at every 200 iterations after the 2000-th iteration and it took nearly 34 hours to complete the training.



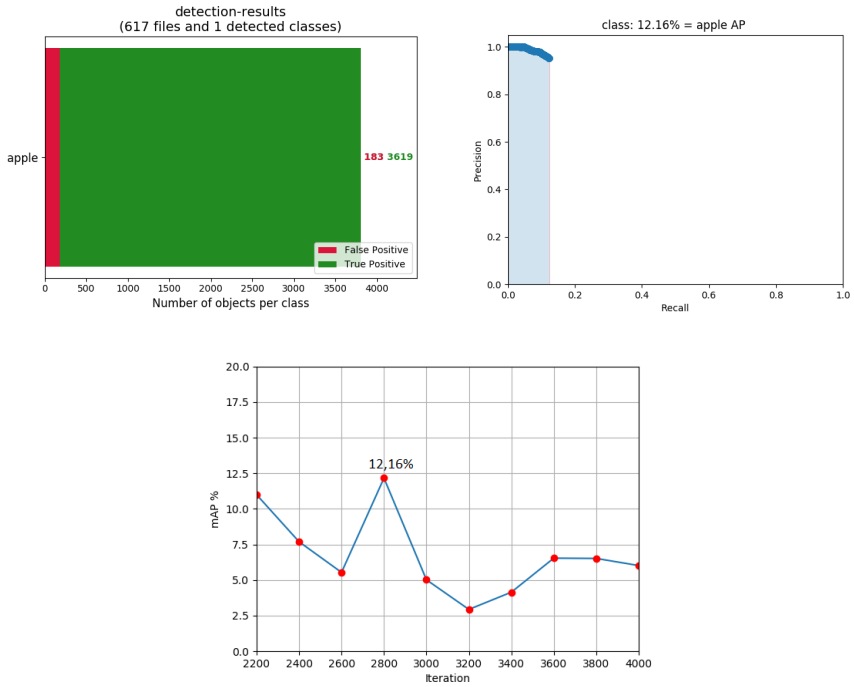
In this graph average loss errors arisen during fine-tuning are drawn. These values are computed according to the formula: average loss error(i + 1) = 0.9 * average loss error(i) + 0.1 * loss error(i + 1), where i refers to the i-th iteration. Error values decrease quickly until they stabilize around 0.4.



Here is presented in detail the trend of average loss errors in the last 2000 iterations and its fluctuating behavior is clear.

Table 4.1: Behaviour of the average loss error during fine-tuning

Nevertheless the training went well and average loss errors achieved small values; results are not much satisfying, probably because the training set was not sufficient for this task, indeed false positives are reduced significantly but true positives are not enough to raise the general statistics and the best result is achieved at iteration 2800 with 12,16% as mAP, considering confidence score down to 0.25. Overall results are summarized in the following page.



Ground truth	29356
TP	3619
FP	183
FN	25737
Precision	0.952
Recall	0.123
F1 Score	0.218
mAP	12.16%

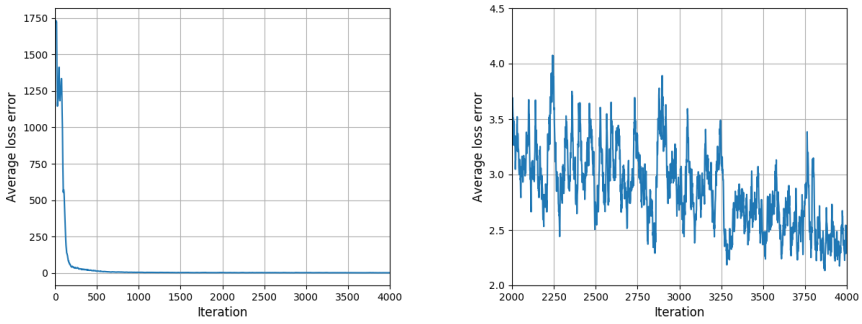
Figure 4.7: General results after first fine-tuning on YOLOv3

Since the network is forced to detect only apples, pictures that contained predictions of oranges now often show predictions of apples (Figure 6.6). As regard other pictures, there are some with a good amount of apples detected (Figure 6.5) and others with little detection (Figure 6.7).

Second fine-tuning

In order to improve detection performance I attempted a second fine-tuning with a larger training, thus I added to the previous dataset new samples re-labeled by myself coming from Open Images v4 dataset and some selected pictures from COCO dataset, for a total of 1228 samples in the training set and 233 samples in the validation set. I kept the configuration script unchanged, having already modified it for the first fine-tuning. I fine tuned YOLOv3 with its pre-trained weights for 4000 iterations, that corresponds to nearly 175 epochs, saving weights at every 200 iterations after the 2000-th. It took almost 35 hours to execute the full fine-tuning.

Since the dataset used is larger and labeled in a more refined way, the task of predicting all the existing apples is more difficult and this aspect reflects on greater average loss errors, that in the end converged over 2.



(a) (b)

Figure 4.8: Plot of average loss error (a) during the whole fine-tuning and (b) between iteration 2000 and 4000

Greater average loss errors do not necessarily mean worse detection performance, indeed with fine-tuned weights YOLOv3 reached better results than the previous cases with over than 7000 apples correctly detected and little more than 200 false positives with a consequent mAP of 24.55%, achieved by weights returned at iteration 3400 and threshold on confidence score of 0.25.

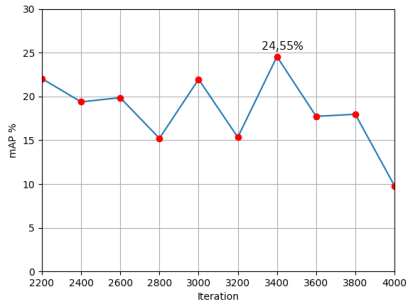
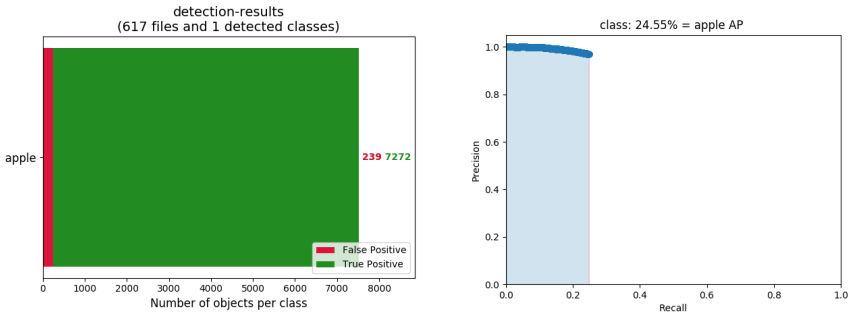


Figure 4.9: Trend of mAP testing weights produced during fine-tuning

I have placed the same pictures I have shown for the first fine-tuning in the last chapter so that they can be compared to check the bounding boxes predicted (Figure 6.8). It is clear from the statistics below the enhancement in detection performance. Also pictures highlight this change, especially those ones with good results since the beginning and there is a little improvement in photos that had little detection.

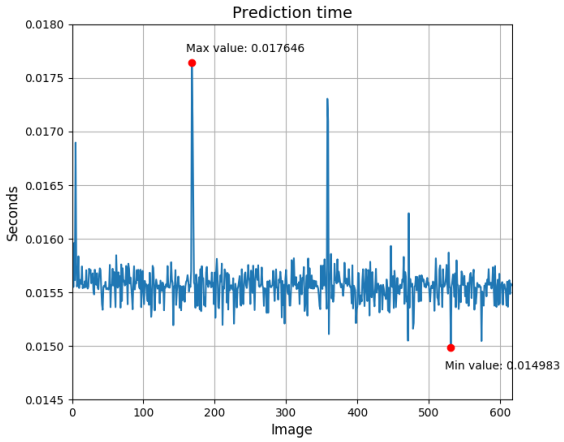


Ground truth	29356
TP	7272
FP	239
FN	22084
Precision	0.968
Recall	0.248
F1 Score	0.394
mAP	24.55%

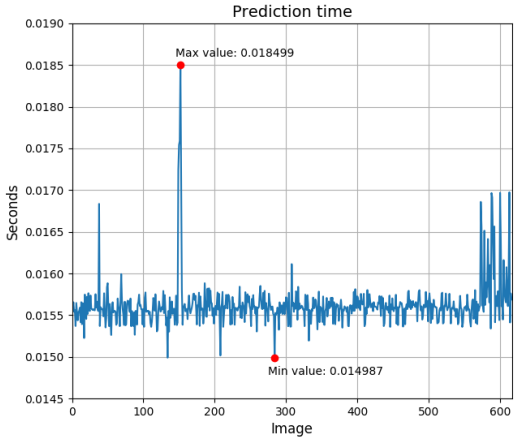
Figure 4.10: Results of the second fine-tuning on YOLOv3

4.4 Results on Tiny YOLOv3

Later I repeated my analysis on Tiny YOLOv3. Since it is a small version of YOLOv3 I expected faster prediction rate but also worst precision in object detection and it is exactly what happened. It took about 3 seconds to process an image of my test set through the network (half of the time taken by YOLOv3) and 0.016 seconds on average to produce predictions (almost one tenth respect to YOLOv3).



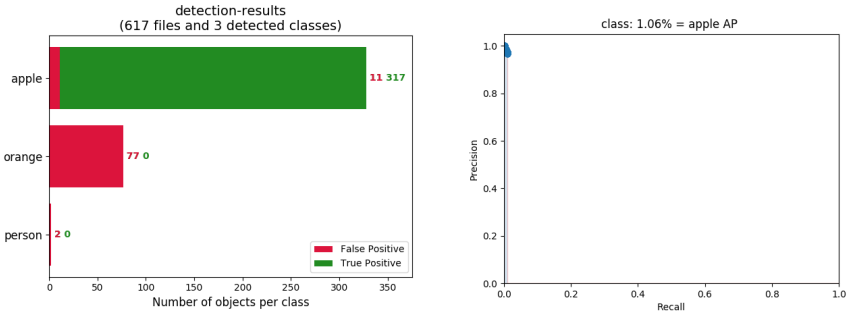
(a) threshold on confidence score: 0.5



(b) threshold on confidence score: 0.25

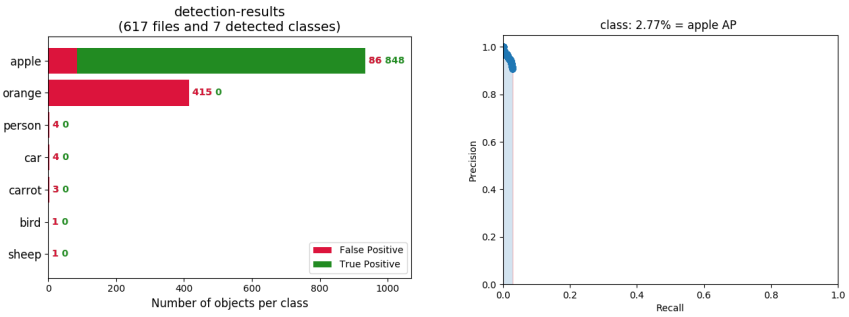
Figure 4.11: Prediction time with Tiny YOLOv3

Unfortunately this speed time is at the expense of detection rate, indeed it is not uncommon to find pictures with no detection at all and it is difficult to have pictures with many apples identified, so mAP is only 1.06% and 2.77% for the lowered threshold on confidence score.



Ground truth	29356
TP	317
FP	90
FN	29039
Precision	0.779
Recall	0.011
F1 Score	0.021
mAP	1.06%

Figure 4.12: Results with Tiny YOLOv3 and threshold 0.5



Ground truth	29356
TP	848
FP	514
FN	28508
Precision	0.623
Recall	0.029
F1 Score	0.055
mAP	2.77%

Figure 4.13: Results with Tiny YOLOv3 and threshold 0.25

I have displayed in chapter 6 the same pictures used previously to show the poor results achieved by Tiny YOLOv3 also with the lowered threshold on confidence score of 0.25 on images that gave good results with YOLOv3 (Figure 6.9). Getting less predicted objects means that there are less misclassified apples also on those pictures that presented many wrong detection of oranges (Figure 6.10).

4.4.1 Fine-tuning Tiny YOLOv3

Given the great performance in time and the much room for improvement, a fine-tuning on Tiny YOLOv3 is in order, so in the configuration file I set the number of classes to one, filter before detection to 18, batch size of 64, mini-batch size of 8 and image resolution to 608x608. Then I fine-tuned my network with its pre-trained weights for 4000 iterations directly on the last dataset that I created for the second fine-tuning on YOLOv3 (that one composed by images coming from Open Images v4 and COCO), I saved weights at every 200 iterations in the second half of re-training and it took 1.5 hours to finish the fine-tuning.
The result of this process was very satisfying because I reached, at iteration 3400 and considering confidence score of at least 0.25, a mAP of 22.62%, a rate similar to that achieved by YOLOv3 after the second fine-tuning (24.55%), but using a not very deep neural network and that took less than 2 hours to re-train.

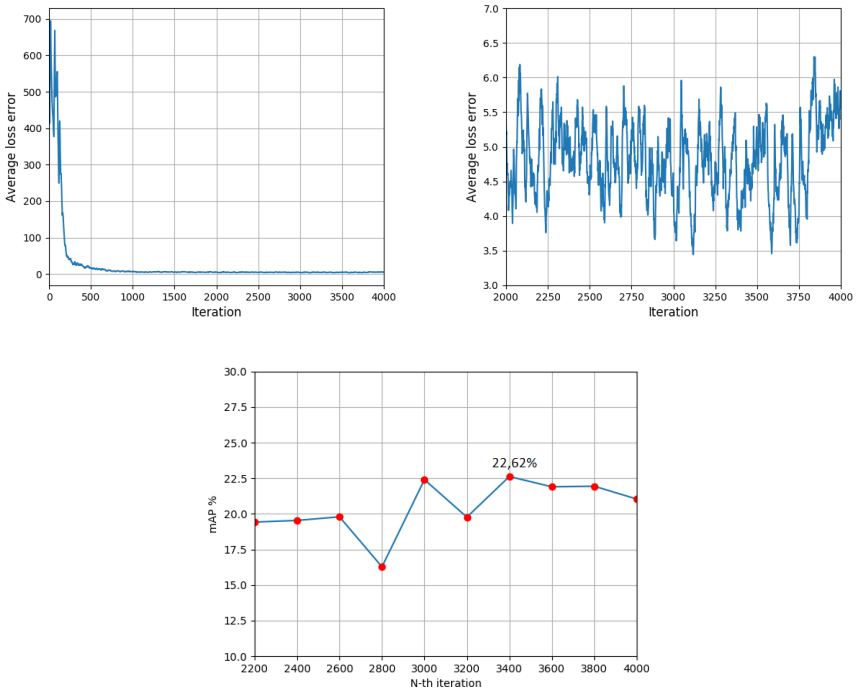
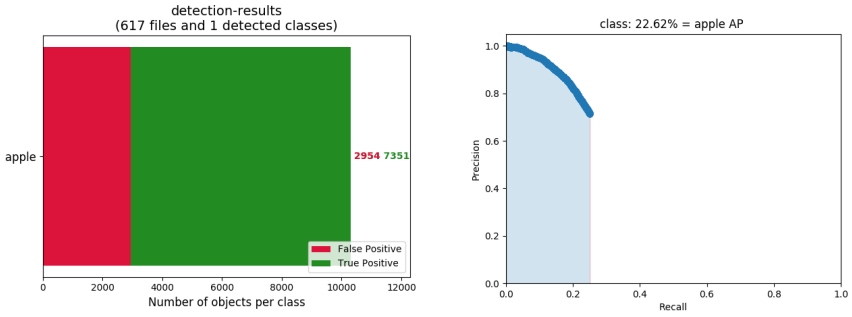


Figure 4.14: Fine-tuning loss error and mAP

Despite Tiny YOLOv3 succeeding in detecting correctly a greater number of apples in respect to the best result achieved by YOLOv3, identifying 7351 true positives; the overall output is worst due to the prediction of several bounding boxes not fitting well apples and the misclassification of many leaves as fruits, that resulted in 2954 false positives, causing a reduction in precision, F1 score and mAP.



Ground truth	29356
TP	7351
FP	2954
FN	22005
Precision	0.713
Recall	0.250
F1 Score	0.371
mAP	22.62%

Figure 4.15: General statistics after fine-tuning on Tiny YOLOv3

The improvement in object detection is also visible on images put in the dedicated chapter, not only on pictures where apples have been easily detectable since the beginning (Figure 6.11 (a)) but especially on those with many fake oranges classified (Figure 6.11 (b)) or those with few or none object detected in the preceding analysis (Figure 6.11 (c)).

4.4.2 Grid search on Tiny YOLOv3

Given the little amount of time spent to execute the fine-tuning on Tiny YOLOv3, it is possible to optimize the hyper-parameters of this neural network with a grid search.
Grid search is a process that allows to compare several values of hyper-parameters and to choose the best configuration according to a selected measure. The values to be tested are picked principally in two ways:

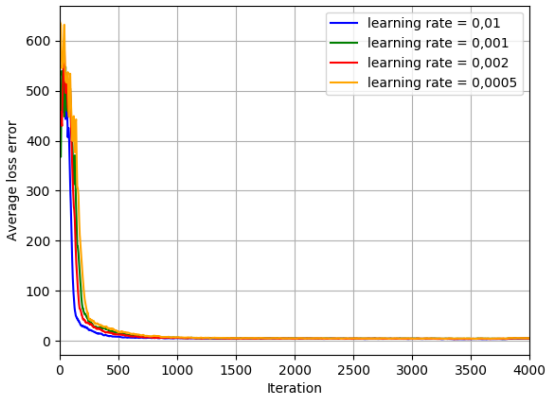
- manually: the developer selects for every hyper-parameter the values that he wants to be tested
- randomly: the developer provides a likely probability distribution for each hyper-parameter and the computer sorts out the values from those distributions to be tested

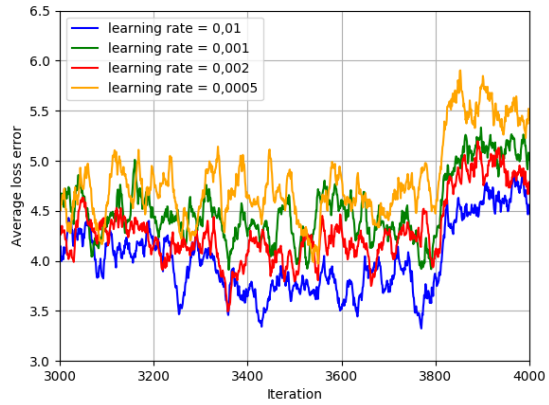
The second method usually outperform the first one but only if the probability distributions are carefully selected, so I preferred to choose manually the values to evaluate.
After that I had to decide if to optimize an hyper-parameter at a time or to test every possible combination of values. The second option gives better results but it is very time consuming so, unless the values to test are few or the available resources are big, it is recommended to optimize parameters one by one and so I did. In addition grid search is often combined with cross-validation to get a more general evaluation of performance, so I implemented a cross-validation k-fold with k = 5, training for 4000 iterations over the union of training and validation set that I used before and I have chosen as performance measure the average over the last 1000 average loss errors of the training.
Now I am going to reproduce the steps I followed to accomplish my grid search.

Learning rate

The most important hyper-parameter of a neural network is the learning rate so I started from here to optimize. In the configuration script learning rate

starts at 0.001 and I decided to evaluate also [0.1, 0.01, 0.002, 0.0005]. During re-training with learning rate = 0.1 the loss error, after an initial drop, increased until it overflowed, so the process could not converge and that value had to be removed.

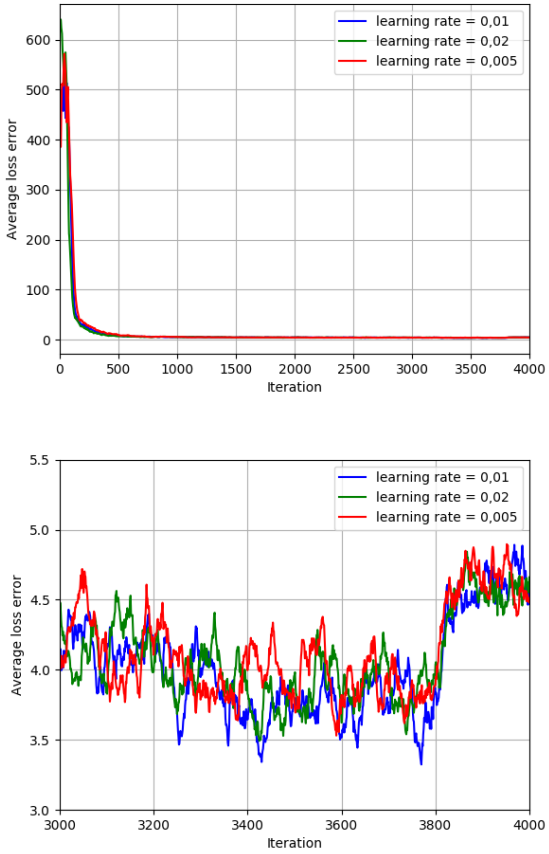




Learning rate	Mean of average loss error
0.01	4.014
0.002	4.302
0.001	4.524
0.0005	4.803

Figure 4.16: Behaviour of average loss error and table of its mean over the last 1000 iterations

From the plot and table above is well visible that the best performance is reached by setting the learning rate to 0.01. Then I examined two values around it: 0.005 and 0.02 to check if they can achieve lower error.
The result is graphically less clear this time because the paths of average loss error overlap each other and a numerical measure is required to determine which value of learning rate is the best to use for further optimization. According to the mean of average loss error on the last quarter of fine-tuning learning rate set to 0.01 is still the best choice, so I opted for it.

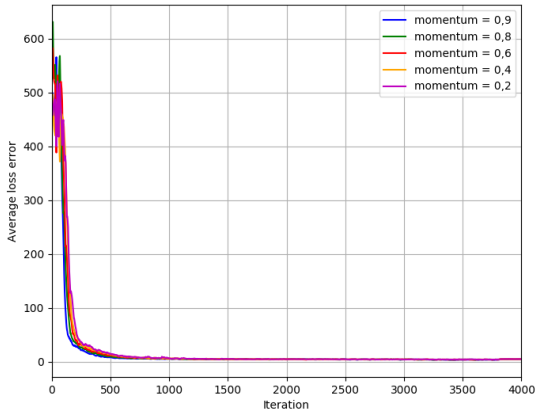


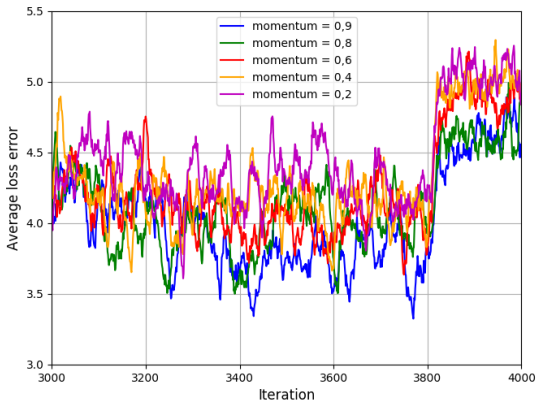
Learning rate	Mean of average loss error
0.02	4.075
0.01	4.014
0.005	4.125

Figure 4.17: Trend of average loss error during fine-tuning and its mean on the last 1000 iterations

Momentum

Another hyperparameter that comes into play in the backpropagation algorithm is the momentum. It can take values between [0; 1] and in Tiny YOLOv3 it is equal to 0.9, thus I tested [0.2, 0.4, 0.6, 0.8].

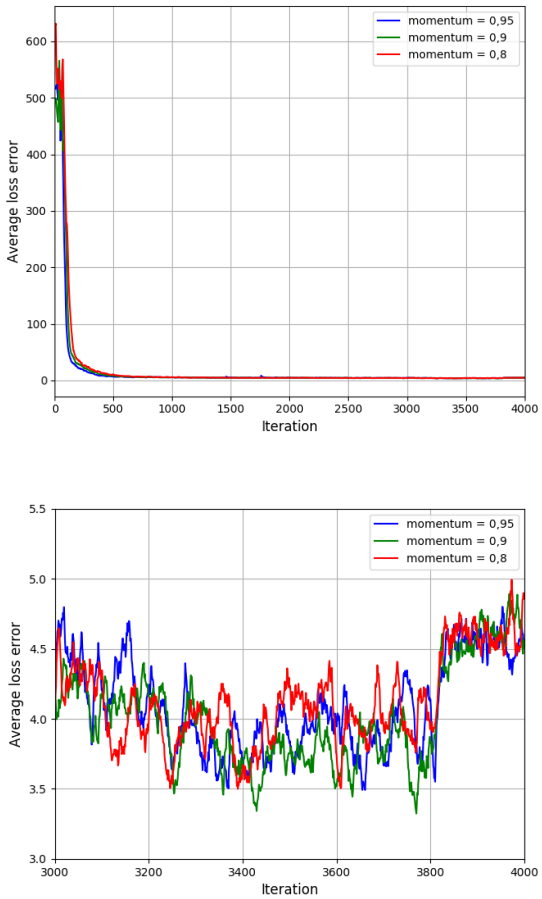




Momentum	Mean of average loss error
0.9	4.014
0.8	4.121
0.6	4.240
0.4	4.318
0.2	4.444

Figure 4.18: Trend of average loss error and its mean on the last quarter of re-training changing momentum

As we can note from the second graph in 4.18, the usage of momentum = 0.9 gives slightly better performances and the computation of means confirms this observation. Then I also checked momentum equal to 0.95 but the mean value of errors is 4.095, greater than the one got by momentum = 0.9, so this parameter can be considered as already optimal and I kept momentum unchanged.

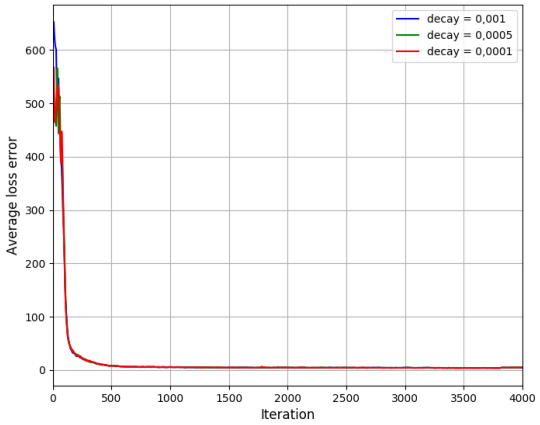


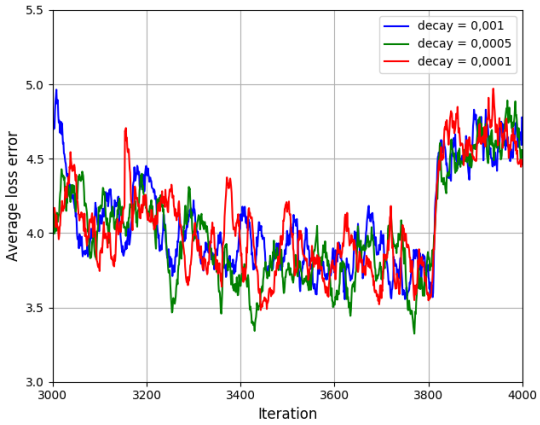
Momentum	Mean on average loss error
0.95	4.095
0.9	4.014
0.8	4.121

Figure 4.19: Trend of average loss error varying momentum

Decay

To cost function is added a weight decay, a penalty term that discourages the network to choose large values for weights in order to avoid overfitting. This term depends on decay, that usually has a very low value, 0.0005 in our case and I decided to evaluate Tiny YOLOv3 with decay equal to 0.001 and 0.0001. The plots of average loss error achieved during training are quite confusing but according to the mean computed on the last 1000 iterations the default decay reaches lower error values so I kept decay = 0.0005 without any further tests.





Decay	Mean on average loss error
0.001	4.074
0.0005	4.014
0.0001	4.065

Figure 4.20: Development of average loss error varying decay and its mean on last 1000 iterations

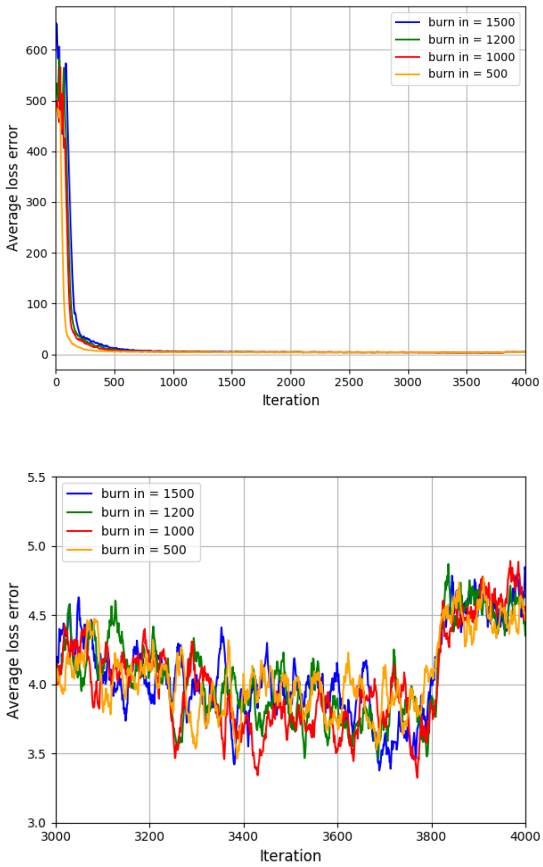
Burn in

As we have seen before in the description of YOLO and other CNN, it is common to start training with a greater learning rate and afterwards to reduce it. However it has been noted empirically that if you choose a lower learning rate for a short period of time at the beginning of training then its speed rises, with a consequent increase in detection rate. The period and magnitude of learning rate reduction is driven by the burn in hyper-parameter.
In Tiny YOLOv3 burn in is set to 1000 and it states that for the first 1000 (or

burn in value) iterations the learning rate is given by:

$$learning\ rate * \left(\frac{current\ iteration}{burn\ in}\right)^4$$

I tested values 500, 1200 and 1500 but without any positive result so I maintained burn in equal to 1000.



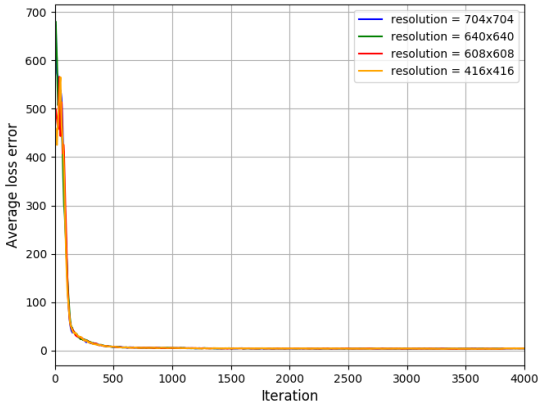
Burn In	Mean on average loss error
1500	4.077
1200	4.066
1000	4.014
500	4.060

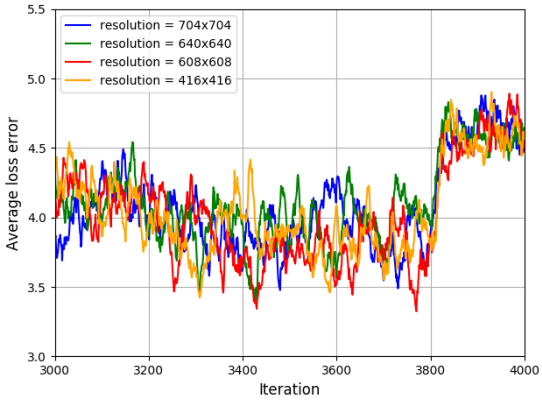
Figure 4.21: Evolution of average loss error with different burn in and its mean on last 1000 iterations

Resolution

Height and width in the configuration file represent the maximal resolution at which an input image can be resized during training. I have already introduced

a sort of optimization by increasing the default size from 416x416 to 608x608, so I checked that my guess was right and I also tested size 640x640, 704x704 and 832x832. With the highest resolution I had an "out of memory" error therefore I discarded it.





Resolution	Mean of average loss error
704x704	4.075
640x640	4.108
608x608	4.014
416x416	4.050

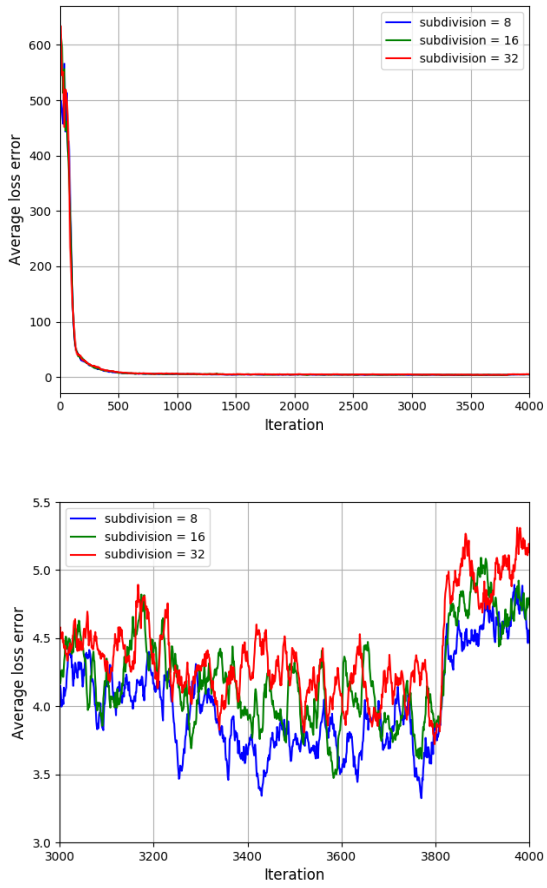
Figure 4.22: Trend of average loss error using several resolutions and its mean on last 1000 iterations

The plot of error development is very chaotic but the mean values reveal that the best resolution between those evaluated is just 608x608.

Subdivision

The subdivision parameter allows to define the size of mini-batches and usually a smaller value gives better results, even though this means to increase, sometimes

significantly, the computational cost. Since 8 is already the lowest possible value, I did not expect to achieve better performance using 16 or 32 but I verified it.



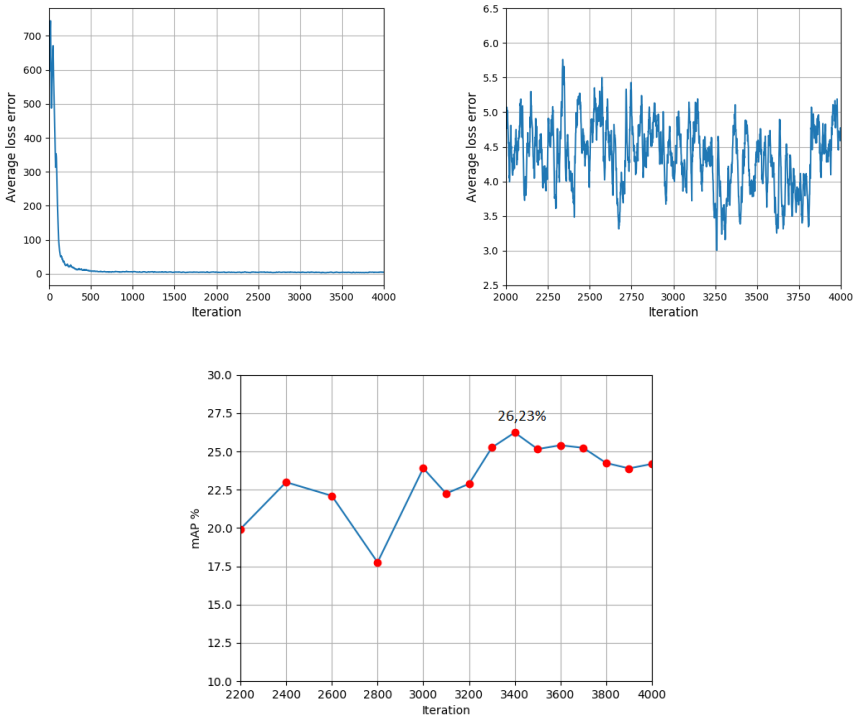
Subdivision	Mean of average loss error
32	4.392
16	4.224
8	4.014

Figure 4.23: Trend of average loss error with different mini-batch size and its mean on the last quarter of training

It is quite clear already from the graph above, that the choice of subdivision equal to 8 is the best available so I kept it.

4.4.3 Fine-tuning after grid-search

I fine-tuned Tiny YOLOv3 with the new configuration obtained by grid-search for 4000 iterations but this time saving weights every 200 iterations between 2000-3000 and then every 100 iterations. I reached the highest mAP of 26.23% at iteration 3400 with threshold on confidence score 0.25 and this result is better than the previous mAP by almost 4 percentage points.



Ground truth	29356
TP	8658
FP	4475
FN	20698
Precision	0.659
Recall	0.295
F1 Score	0.408
mAP	26.23%

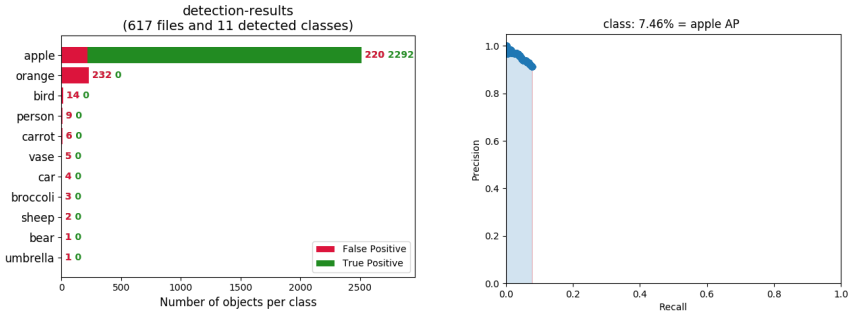
Figure 4.24: Overall results after grid-search

In the last chapter I have listed some pictures already seen to show the final bounding boxes predicted 6.12. Considering the statistics above it can be noted that true positives are grown so a greater amount of apples was detected but also false positives are increased significantly, partly due to bounding boxes not matching well ground-truth, partly for completely wrong classification of leaves as apples. In order to reduce false positives I think it would be a good idea to create training and validation set more appropriate to the task to carry out, i.e. composed only by pictures portraying apples on trees made in a real orchard.

4.4.4 An oddity

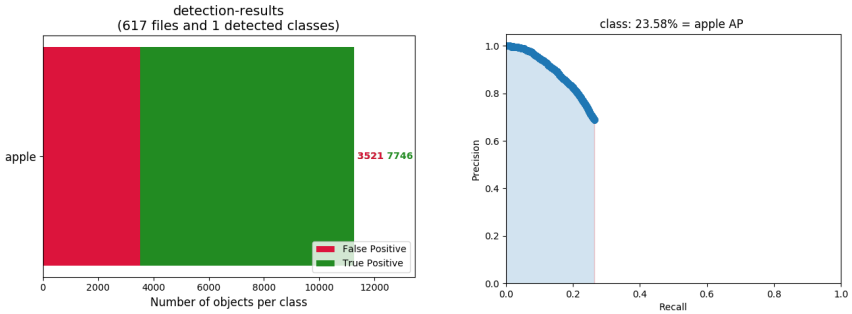
In the description of YOLO's benefits I mentioned its capability to generalize the patterns learnt in order to allow the detection of the same object in several light and exposure conditions or from different points of view, allowing to get an

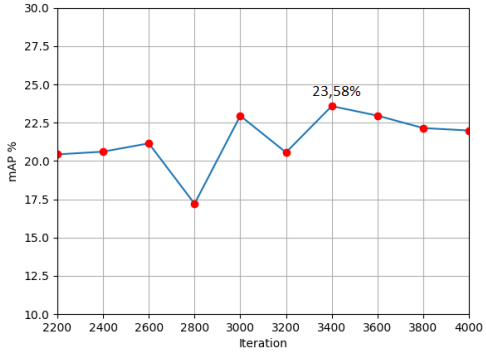
overall higher mAP on different datasets. An example of YOLO robustness can be highlighted by changing the format of the images in the test set. For all my previous analysis I used images in PNG format, but if I convert them into JPEG format I can reach better performance in mAP both with the original weights of Tiny YOLOv3 and with those returned by its fine-tuning, even if every pixel of the new pictures differs from its corresponding pixel in the original photo.



Ground truth	29356
TP	2292
FP	497
FN	27064
Precision	0.8218
Recall	0.0781
F1 Score	0.1426
mAP	7.46%

Figure 4.25: Overall results on original Tiny YOLOv3 with threshold 0.25 on JPEG images





Ground truth	29356
TP	7746
FP	3521
FN	21610
Precision	0.6875
Recall	0.2639
F1 Score	0.3814
mAP	23.58%

Figure 4.26: Overall results on fine-tuned Tiny YOLOv3 with JPEG images

These results can be justified by considering that the neural network was trained and fine-tuned on JPEG images, so probably the patterns learnt are more easily identifiable on pictures of the same type rather than on different formats.
At the end of this thesis I listed some pictures returned by using the original weights 6.13 and fine-tuned weights 6.14 of Tiny YOLOv3 using threshold on confidence score of 0.25. The apples detected in PNG and JPEG images are not exactly the same but globally predictions are similar.

Chapter 5

Conclusions

In this work I have examined the problem of apples detection from images taken in a real orchard using an innovative and high-performing CNN. The initial results showed the difficulty of accomplishing this task by the original YOLOv3 and Tiny YOLOv3 and the necessity of a specific fine-tuning of them in order to get better values of mAP. In summary the main outcomes achieved by YOLOv3 and Tiny YOLOv3, setting always minimum confidence score at 0.25, were:

	Original YOLOv3	After first fine-tuning	After second fine-tuning
Ground truth	29356	29356	29356
TP	5333	3619	7272
FP	1919	183	239
FN	24023	25737	22084
Precision	0.735	0.952	0.968
Recall	0.182	0.123	0.248
F1 Score	0.291	0.218	0.394
mAP	17.10%	12.66%	24.55%

Figure 5.1: Recap on YOLOv3 statistics

	Original Tiny YOLOv3	After fine-tuning	After grid-search
Ground truth	29356	29356	29356
TP	848	7351	8658
FP	514	2954	4475
FN	28508	22005	20698
Precision	0.623	0.713	0.659
Recall	0.029	0.250	0.295
F1 Score	0.055	0.371	0.408
mAP	2.77%	22.62%	26.23%

Figure 5.2: Recap on Tiny YOLOv3 statistics

The improvements introduced by fine-tuning are remarkable, in particular for the Tiny YOLOv3, that reached higher average precision than the complete YOLOv3 and in a time significantly smaller, therefore I think that Tiny YOLOv3 is more suitable for practical applications. Certainly there is still large room for improvement because a mAP of 26.23% is not optimal but considering that the test set is made by pictures taken outside in a real orchard, so with all the illumination problems, the big amount of apples to detect, their position not always clearly visible and that the training set is composed by only few pictures of this kind and the greater part of pictures portrays a little amount of apples well distinguishable in a inner space, hence I think the results can be reassessed as pretty good.

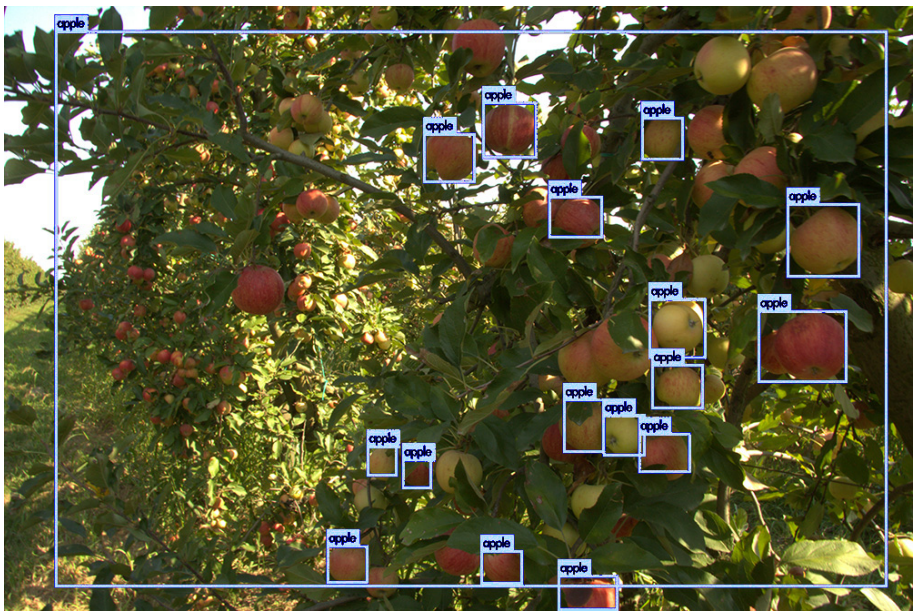
The next step in detecting apples on images should be definitely the collection of new pictures of apple trees in different periods of the day and of the year to use them as training set. This approach should help the neural network to have a more complete knowledge about how apples look like on a tree in different light conditions and at different stage of ripening.
Then a possible future application of this neural network could be the integration in a bigger system aimed at the automatic harvesting of apples or fruit in general. This device should employ the usage of an hyperspectral camera sensitive to near-infra-red or ultra-violet waves because it could determine very precisely the level of ripeness of each fruit detected by the neural network but also the presence of damages caused by weather conditions or by pathogen and hence allow an accurate selection of fruit, as well described in "Recent progress of hyperspectral imaging on quality and safety inspection of fruits and vegetables: a review" [32]. Images can offer information about the position of the fruit only in two dimensions but to harvest it, it is needed to know its correct position in 3-D space so it could be used stereo system or lidar to determine at which depth it is located. Finally a robotic arm, put on a rover able to navigate the orchard autonomously, should reach the fruit and take it in the proper manner without causing damages by breaking some branches or tearing leaves. Some preliminary experiments have been reported in "A survey of computer vision methods for locating fruit on trees" [33] but the way to built a working system of this kind is still long and complex.

Chapter 6

Images

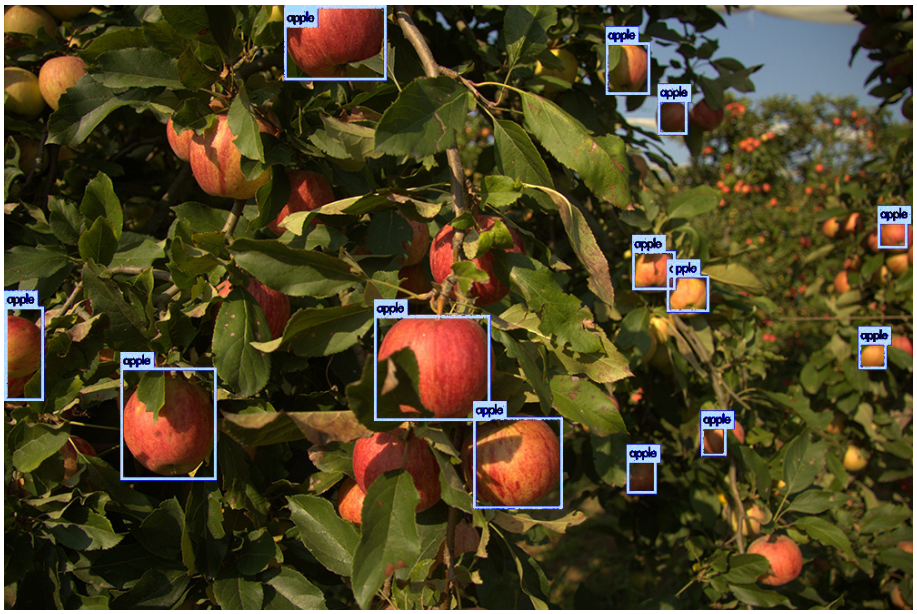
Here are listed the pictures that I have selected to show some output returned by YOLOv3 and Tiny YOLOv3 before and after fine-tuning and optimization. In the original networks I have chosen light blue to mark the apple bounding boxes and after fine-tuning I used light pink to mark them, while the wrong labels are colored with yellow.

First of all are presented some images returned by the original YOLOv3 and they will be used as reference point to the subsequent improvements.




apple
apple
apple
apple


apple
apple
apple
apple
apple
apple
apple
apple
apple
apple


apple
apple
apple
apple
apple
apple
apple
apple
apple
apple
apple
apple
apple


apple
apple
apple
apple
apple
apple

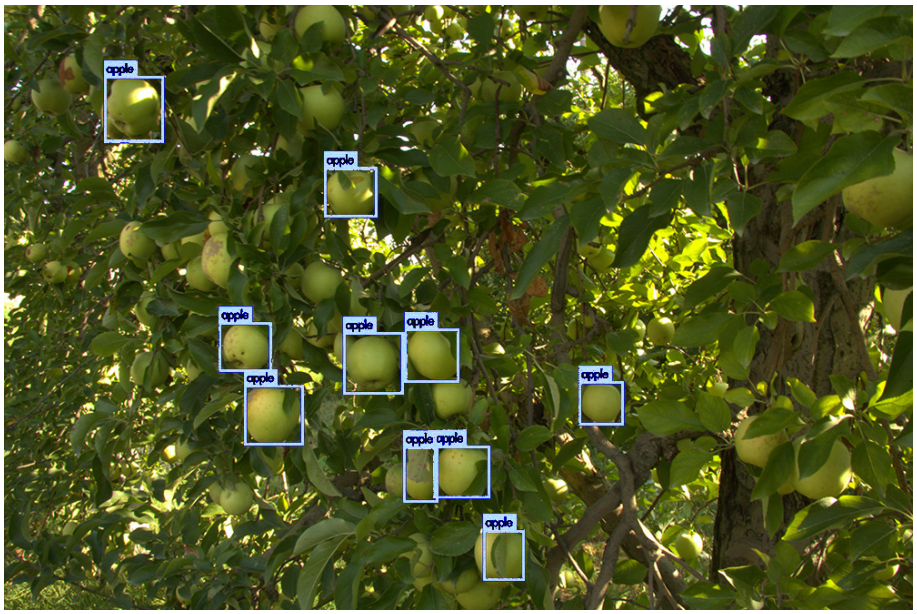
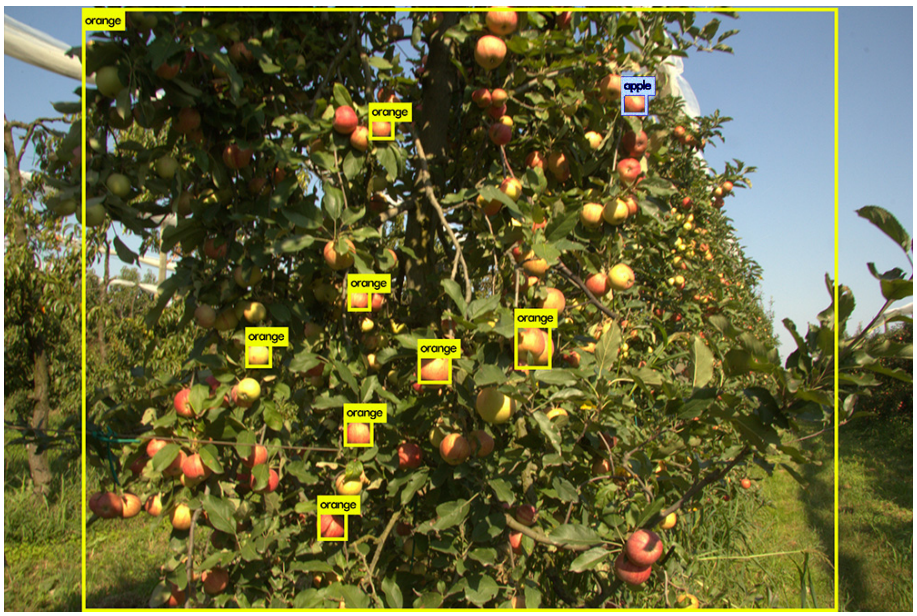


Figure 6.1: First group of pictures with quite good predictions made by YOLOv3




orange
apple
apple
apple
apple
apple
orange
orange
orange
orange
orange

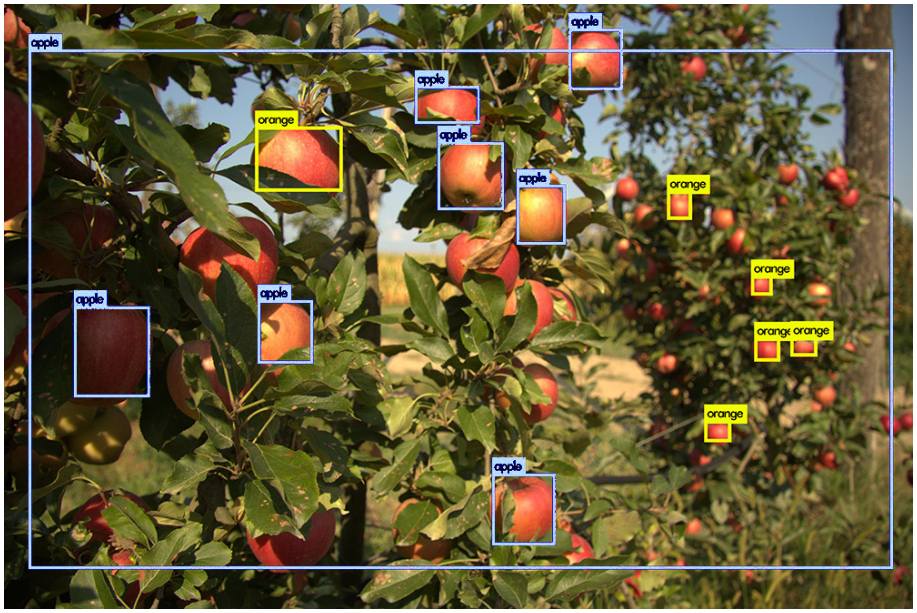

apple
apple
apple
orange
apple
apple
orange
orange
orange orange
orange
apple
apple

Figure 6.2: Second group of pictures with apples misclassificated as oranges by YOLOv3


apple


apple

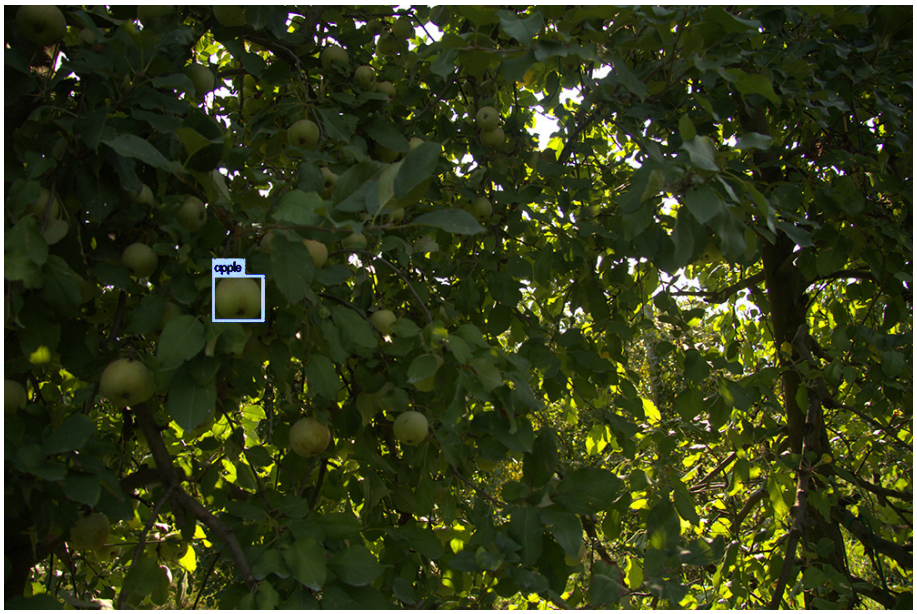
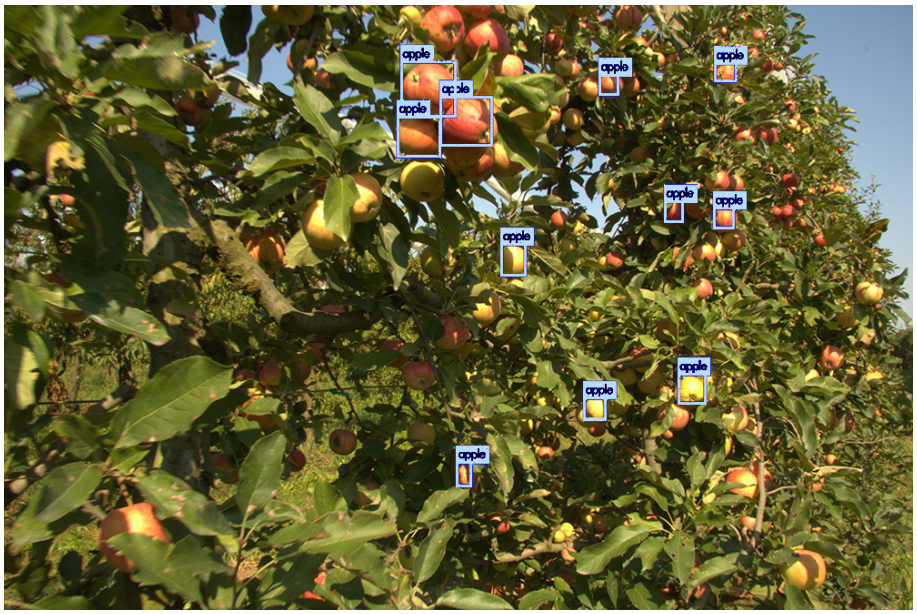


Figure 6.3: Third group of pictures with poor detection made by YOLOv3

Then I compared images with different minimum confidence score; the first picture of each page has the default threshold on confidence score of 0.5 and the second picture has the threshold lowered to 0.25.



(a) Pictures with better classification results reducing the confidence accepted

(b) Picture that have bad results with both threshold settings

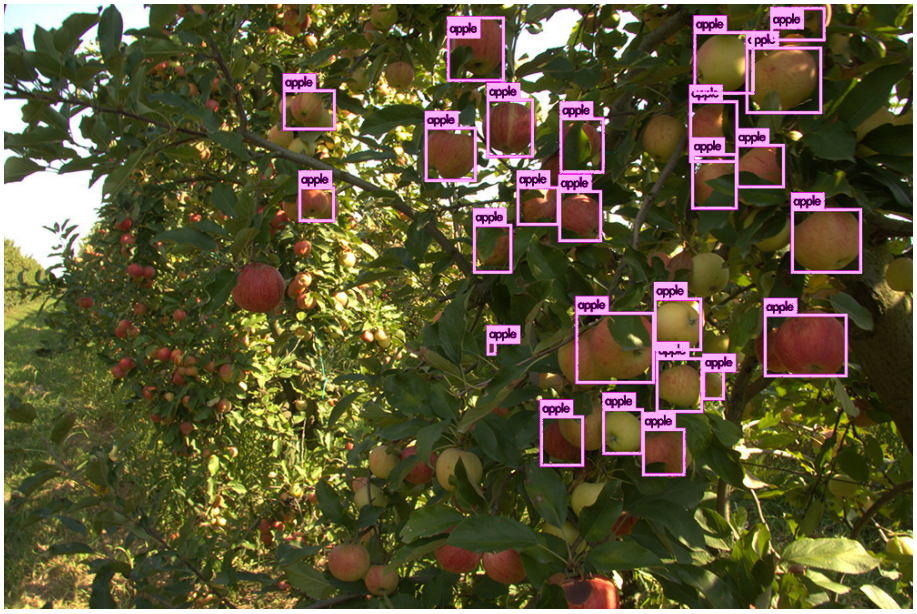

orange
apple
apple
apple
apple
apple
apple
apple

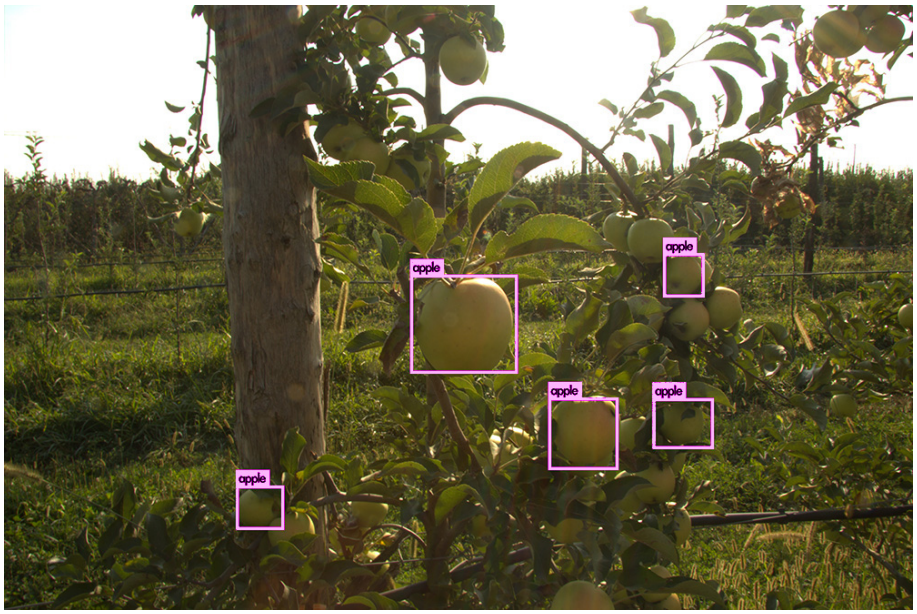

apple, orange
apple
apple
orange
apple
apple, orange
apple
apple
orange
apple
apple
apple
orange
orange
apple
orange
orange
apple
apple
orange
apple
apple
apple
orange
apple
apple
orange

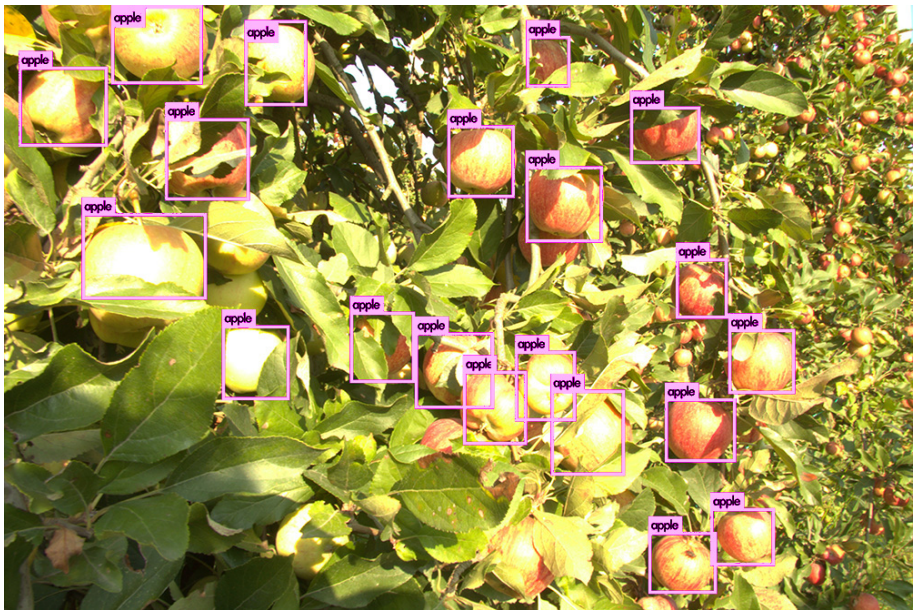
(c) Pictures that increase their number of detection, both correct and wrong

Figure 6.4: Comparison of detection, setting threshold on confidence score to 0.5 and 0.25 for YOLOv3

Here I reproposed the first pictures in order to see the improvement brought by the first fine-tuning of YOLOv3 setting confidence score to 0.25.






apple
apple
apple
apple
apple
apple
apple
apple
apple
apple
apple
apple
apple
apple
apple
apple
apple
apple
apple
apple
apple

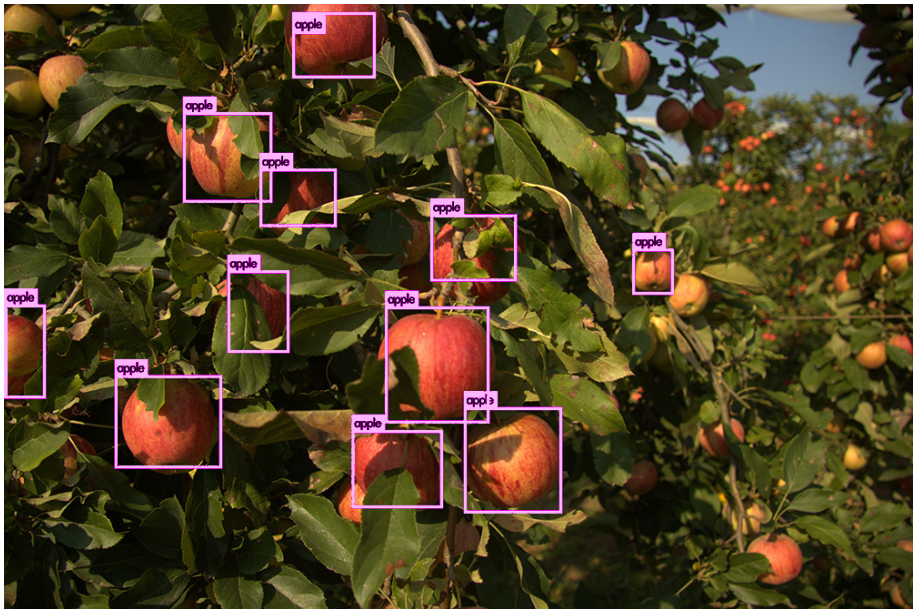

apple
apple
apple
apple
apple
apple
apple
apple
apple
apple
apple

Figure 6.5: Pictures selected for their good predictions on YOLOv3


apple
apple
apple
apple
apple
apple
apple
apple
apple
apple
apple
apple
apple
apple
apple
apple
apple
apple
apple
apple
apple

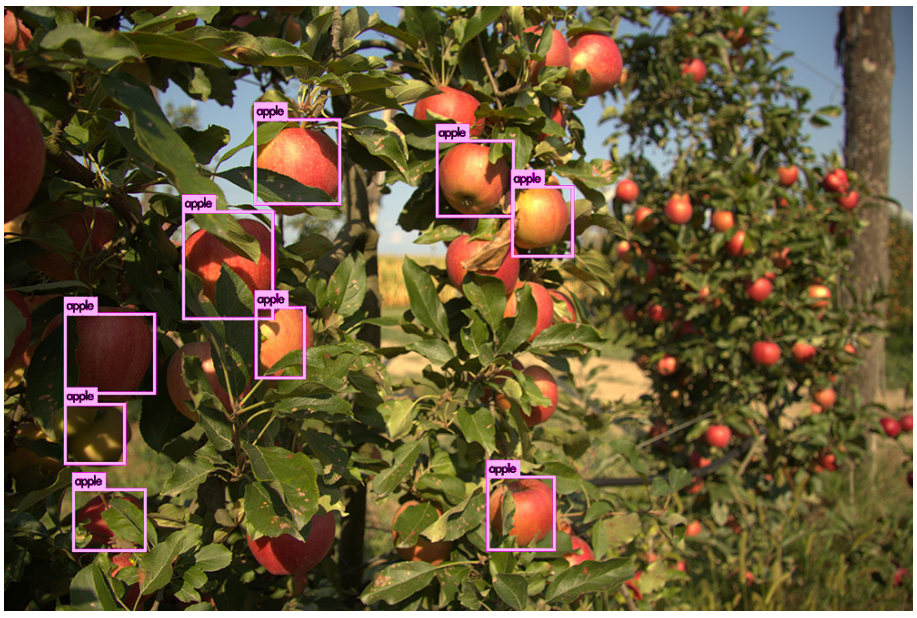


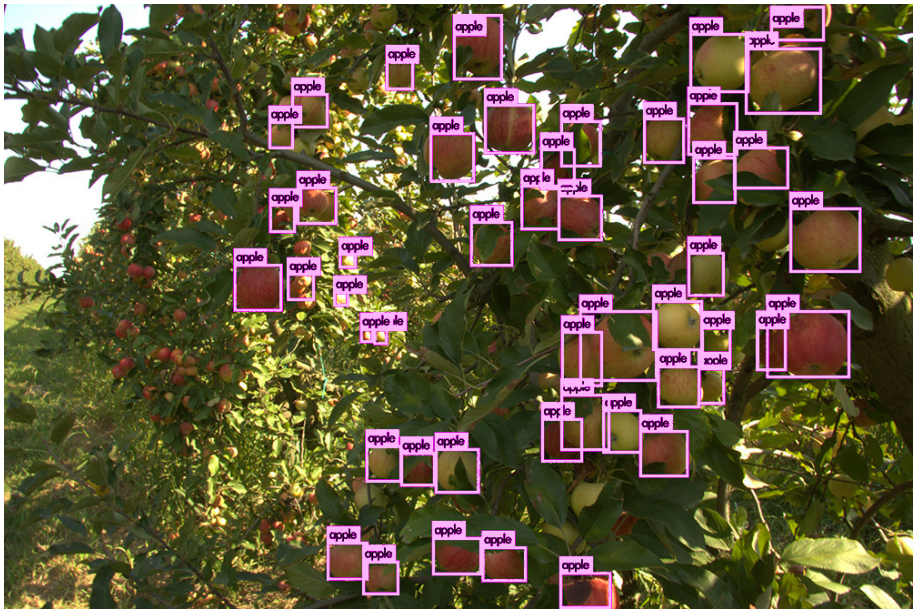
Figure 6.6: Pictures with many misclassified apples in the original YOLOv3

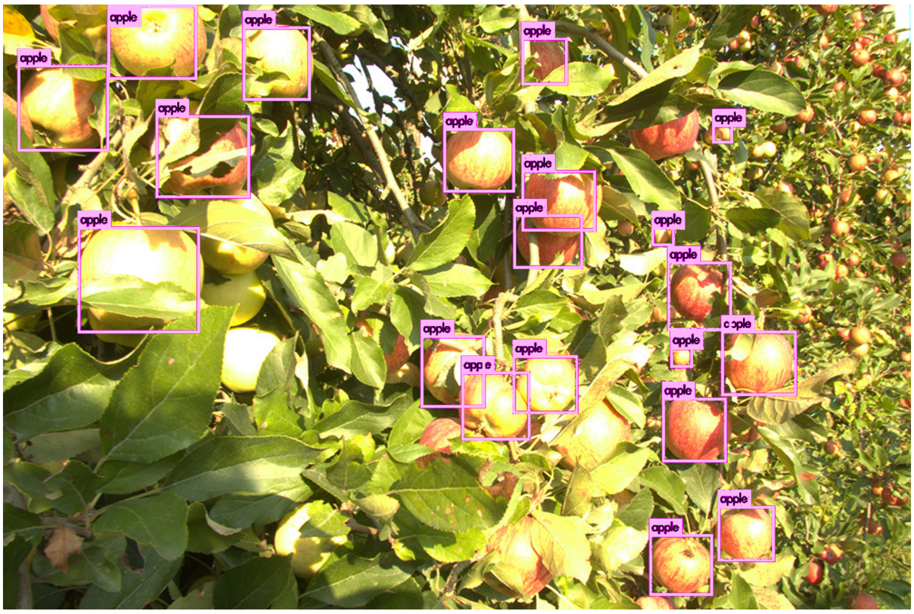

apple
apple
apple
apple
apple

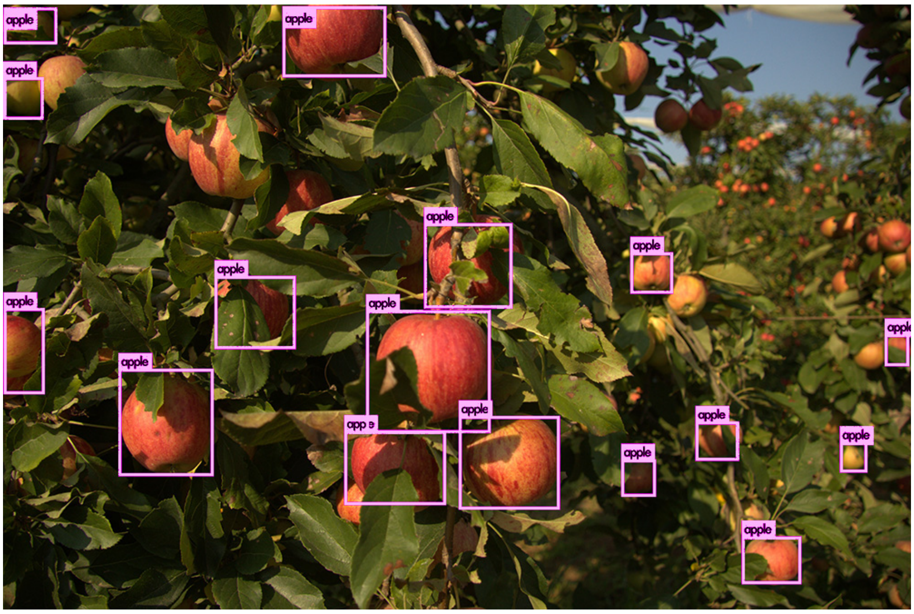

apple

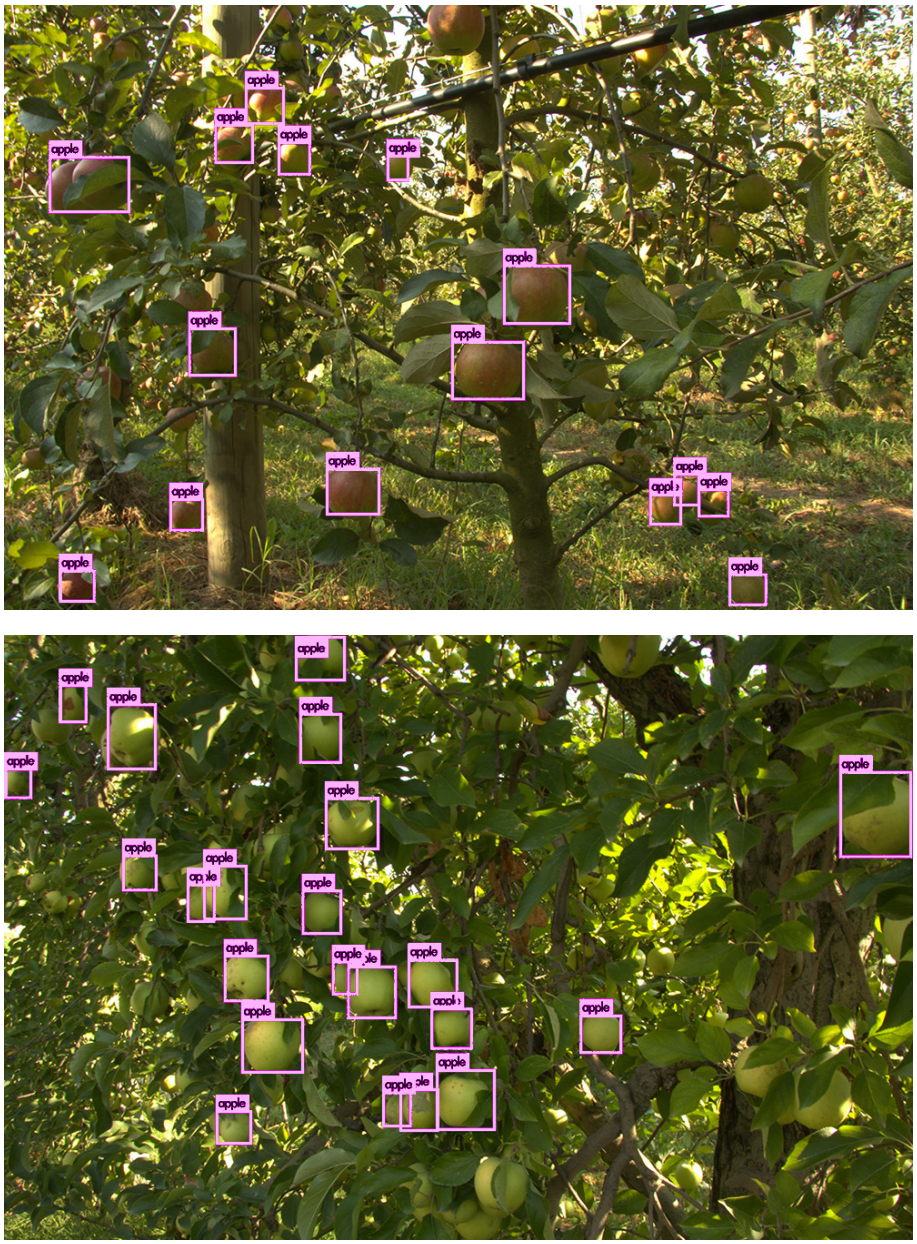
Figure 6.7: Pictures with poor detection in original YOLOv3

After the second fine-tuning on YOLOv3 these are the predictions made on the pictures setting confidence score to 0.25.

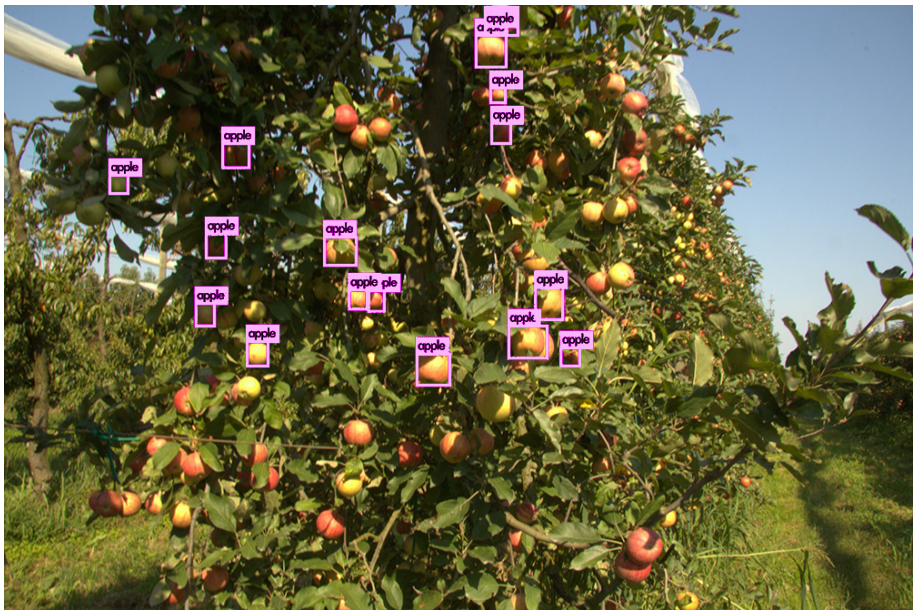



apple
apple
apple
apple
apple
apple
apple
apple
apple
apple
apple
apple
apple
apple
apple
apple
apple


apple
apple
apple
apple
apple
apple
apple
apple
apple
apple
apple
apple
apple
apple



(a) Same images reported in figure 6.5

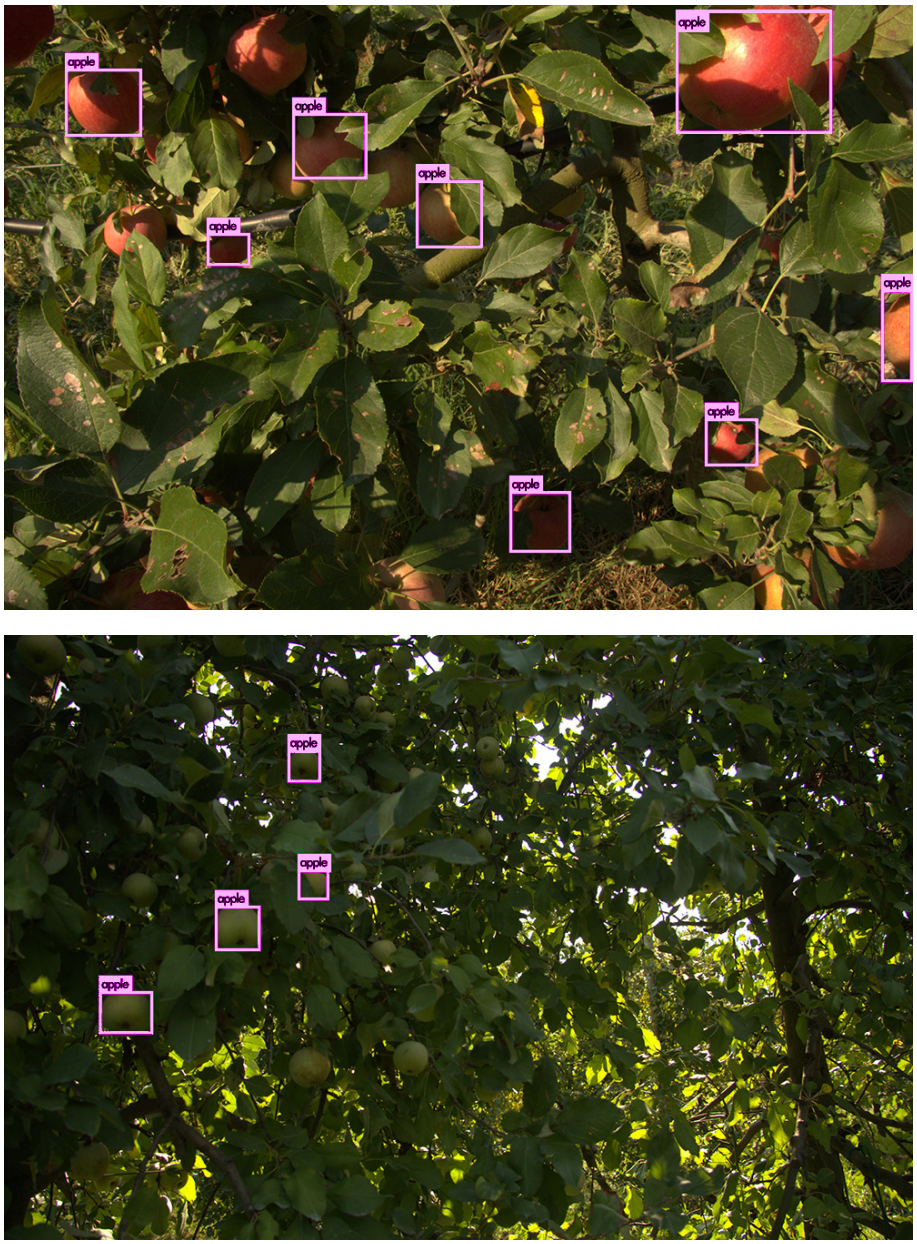

apple


apple

(b) Same images reported in figure 6.6


apple
apple
apple
apple
apple
apple
apple
apple
apple
apple
apple
apple
apple
apple

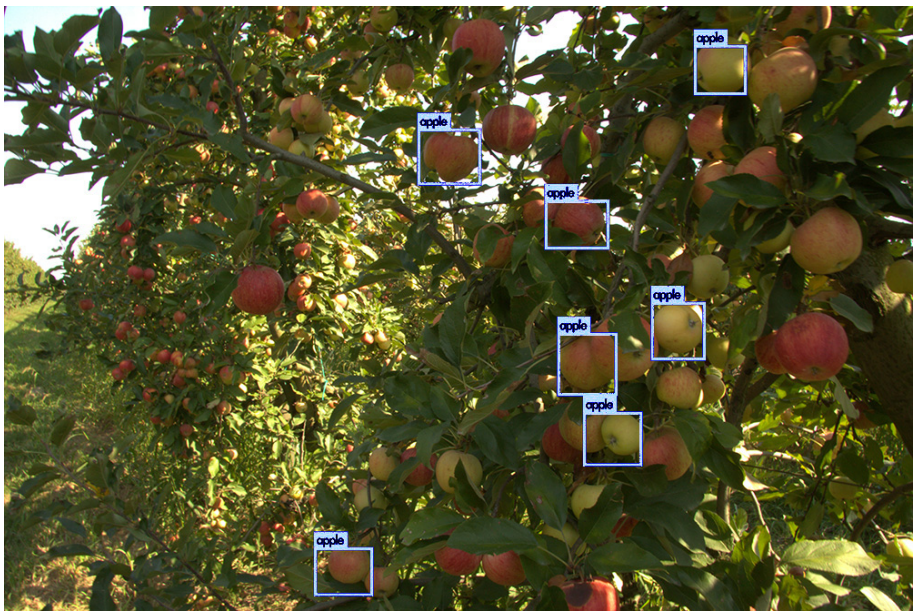

apple
apple
apple

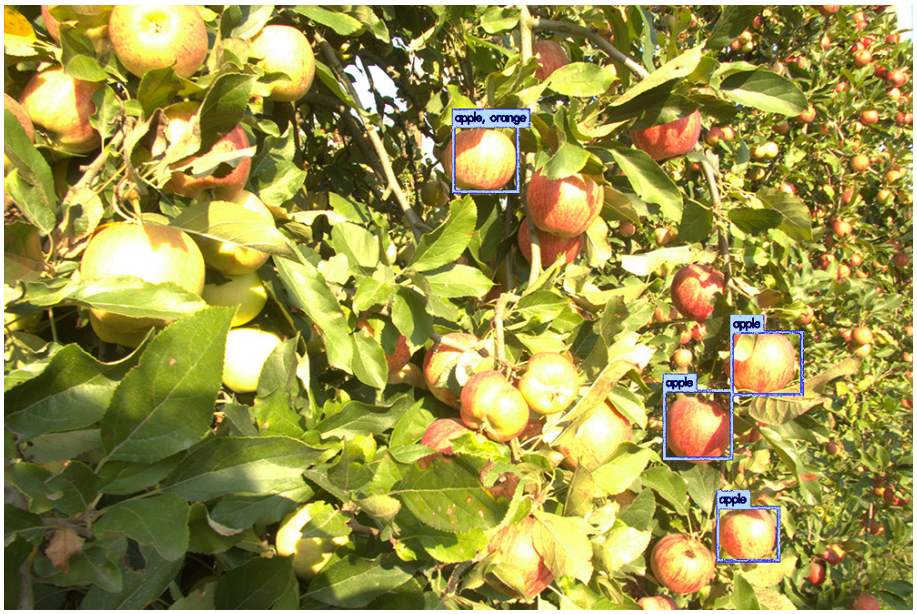


(c) Same images reported in figure 6.7

Figure 6.8: Some pictures got after second fine-tuning of YOLOv3

Pictures processed by Tiny YOLOv3 with minimum confidence score set to 0.25.




apple, orange
apple
apple
apple


apple
orange
apple
apple, orange
apple
orange

Figure 6.9: Results with Tiny YOLOv3 using pictures that gave good outcome with YOLOv3


apple, orange

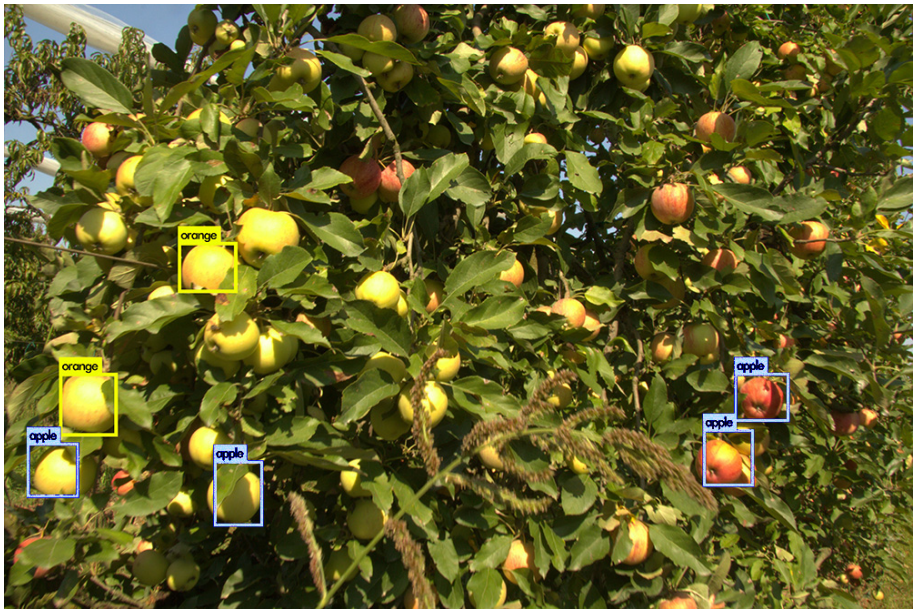
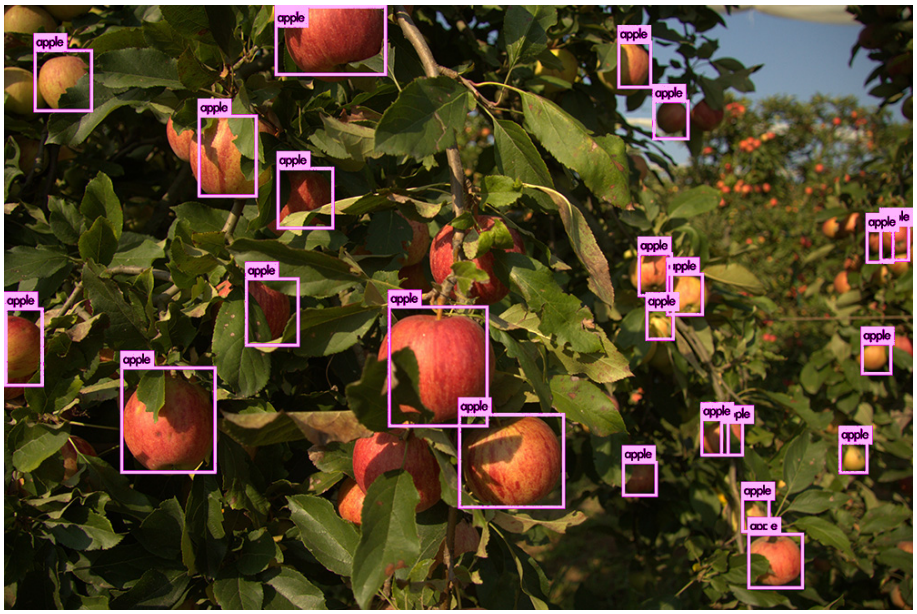
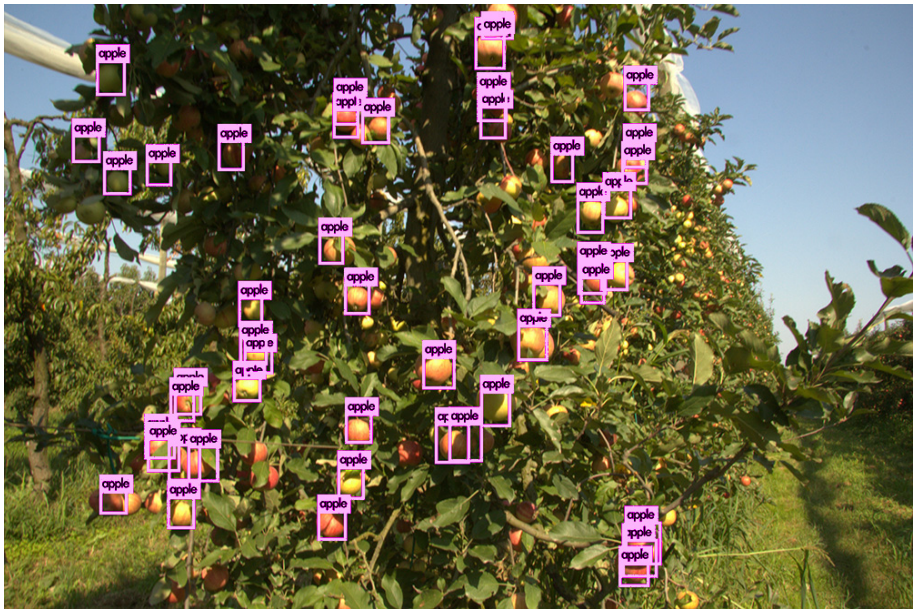

orange
orange
apple
apple
apple
apple

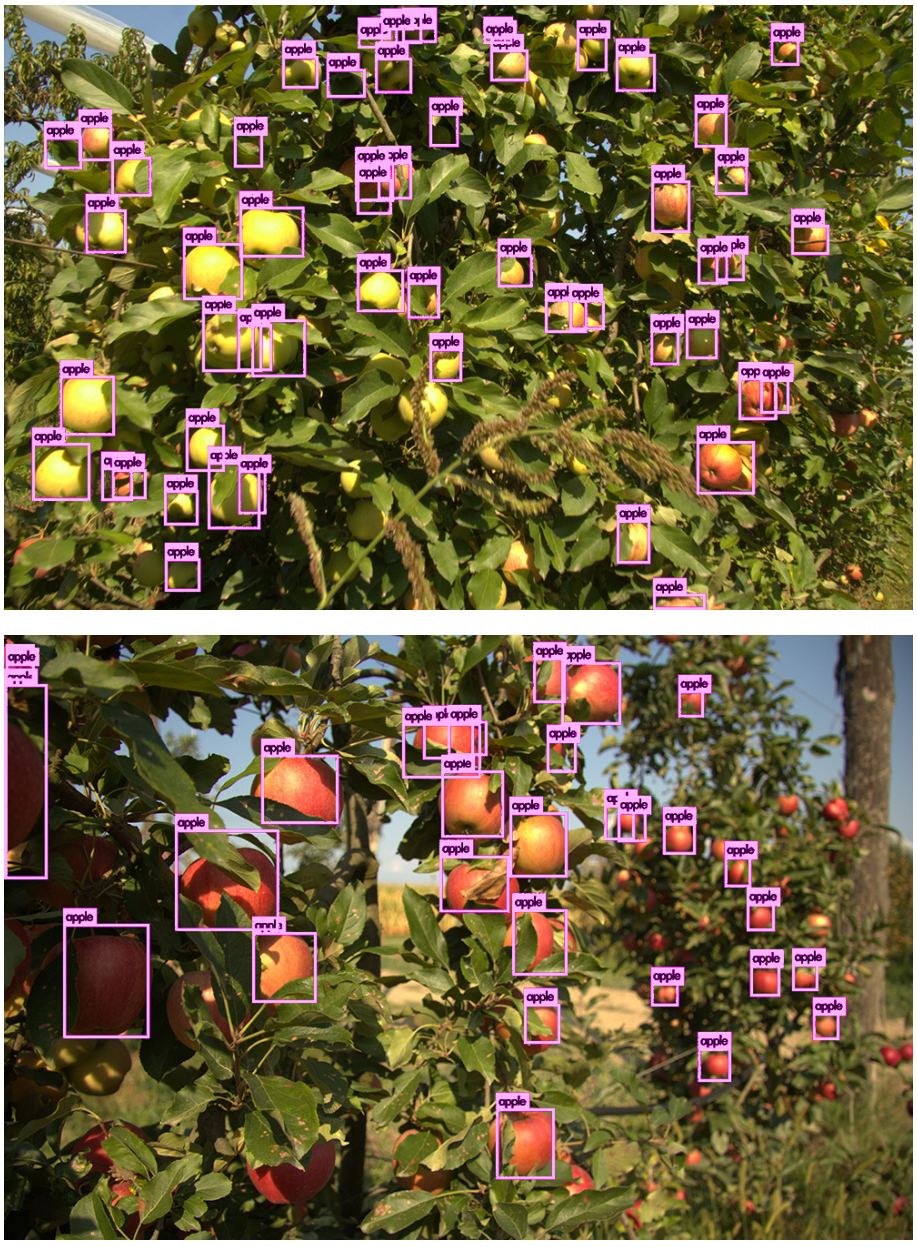
Figure 6.10: Results with Tiny YOLOv3 using pictures that produced several wrong prediction of oranges with YOLOv3

Results returned after fine-tuning on Tiny YOLOv3 and threshold on confidence score of 0.25.



(a) Some pictures on which the detection task is considered "easy" by the network

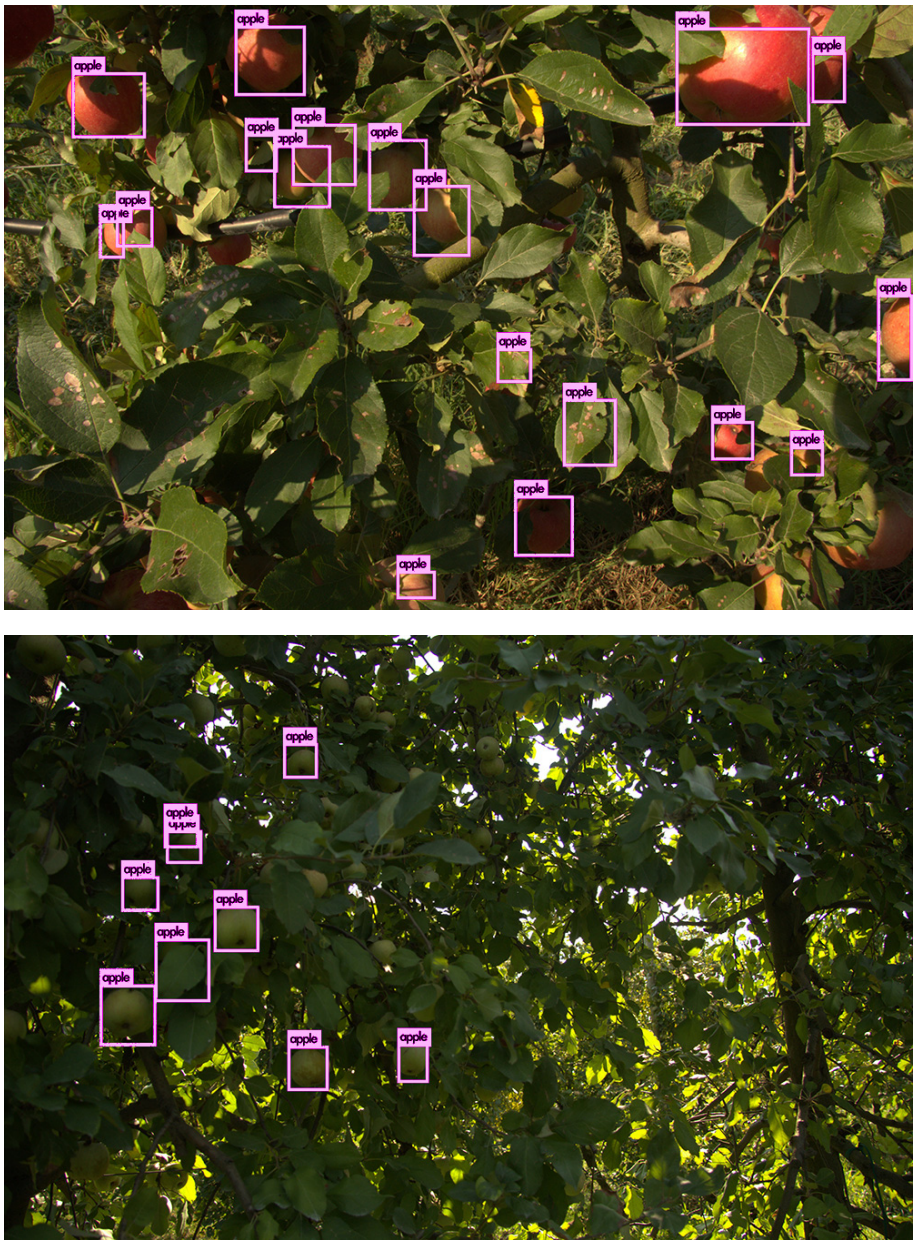




(b) Some pictures that presented many misclassified objects


apple

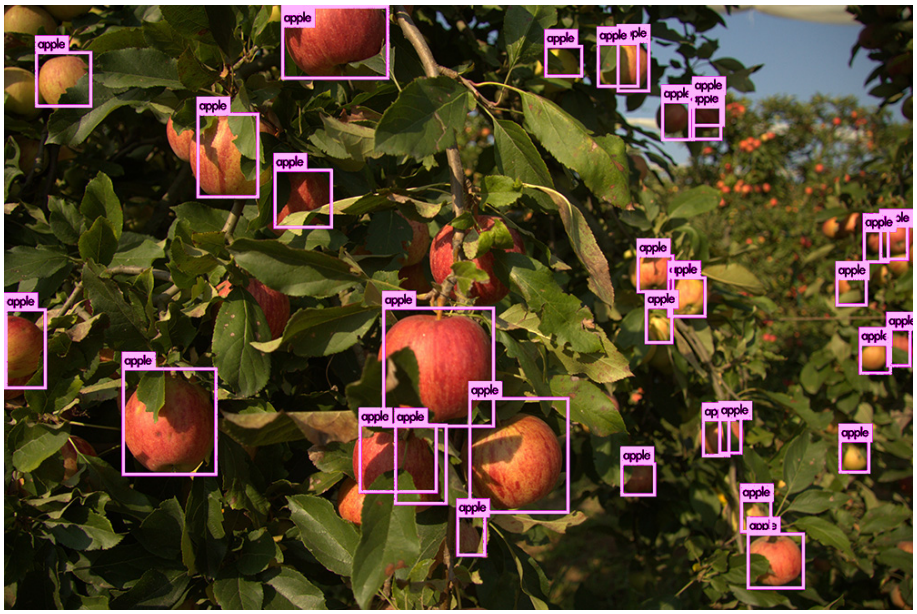

apple



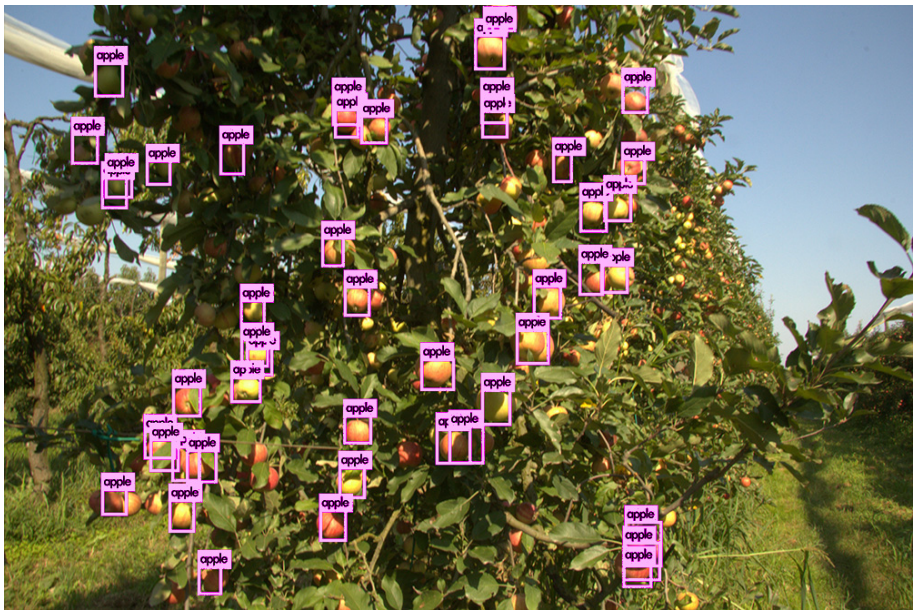
(c) Some pictures that had few or none detected object

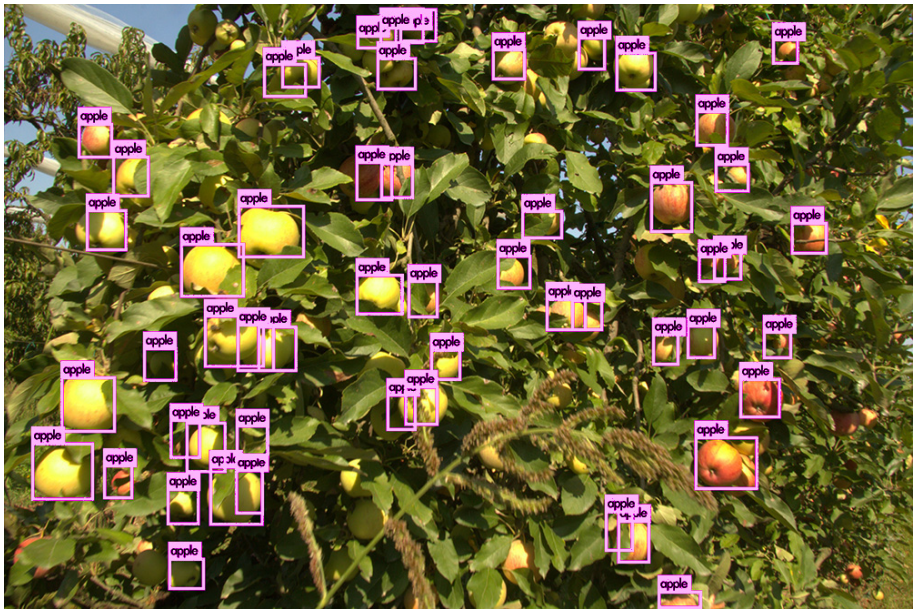
Figure 6.11: Some pictures returned by Tiny YOLOv3 after fine-tuning

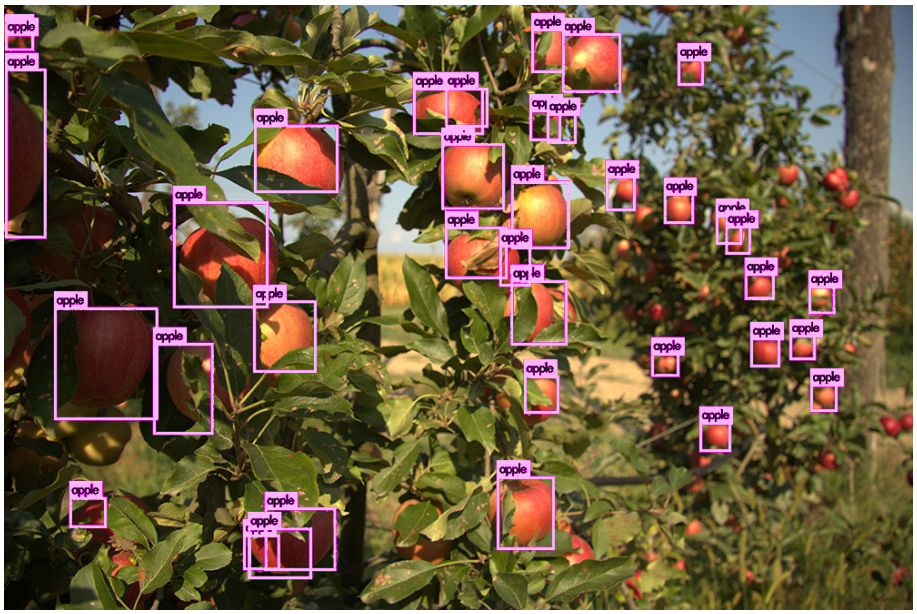
Final pictures returned by the fine-tuned Tiny YOLOv3 using the configuration of hyper-parameters found in the grid search and considering object with at least confidence score of 0.25.

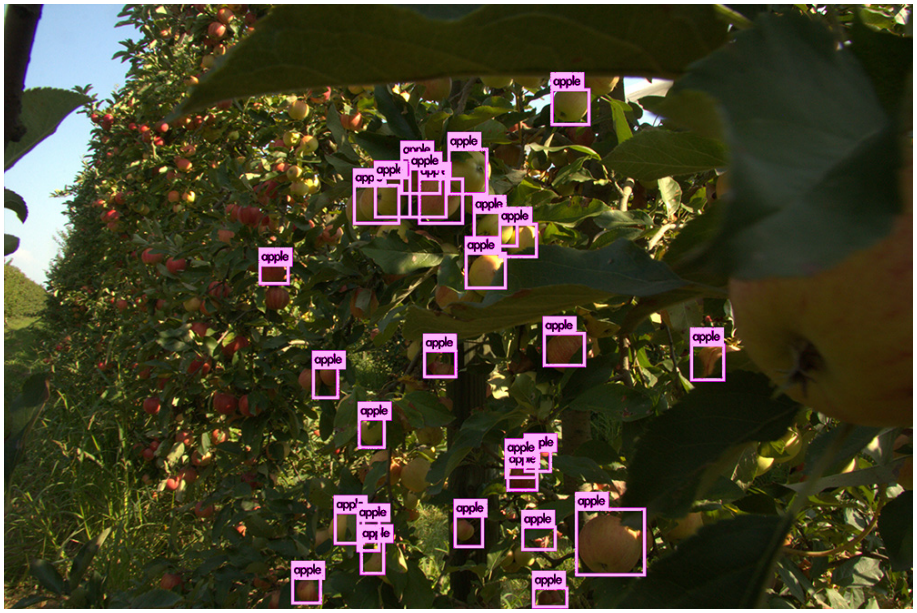

apple


apple

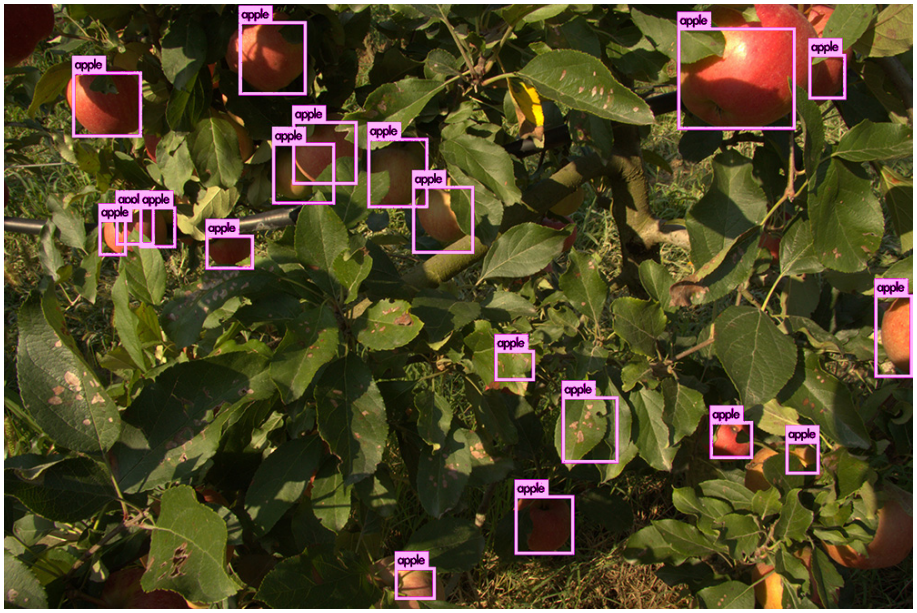









apple
apple
apple
apple
apple
apple
apple
apple
apple
apple
apple
apple
apple
apple
apple


apple
apple
apple
apple
apple
apple
apple
apple
apple
apple
apple
apple
apple
apple
apple
apple
apple

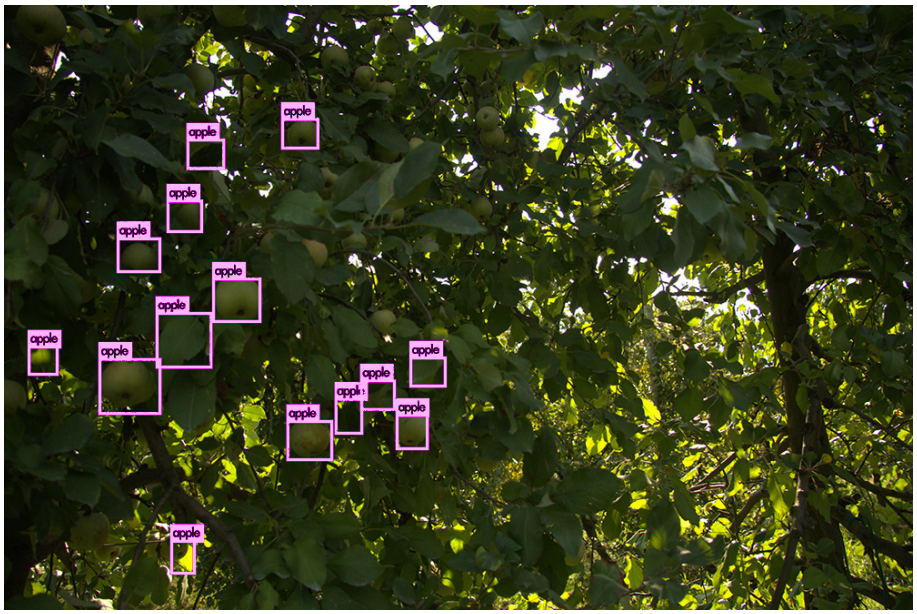
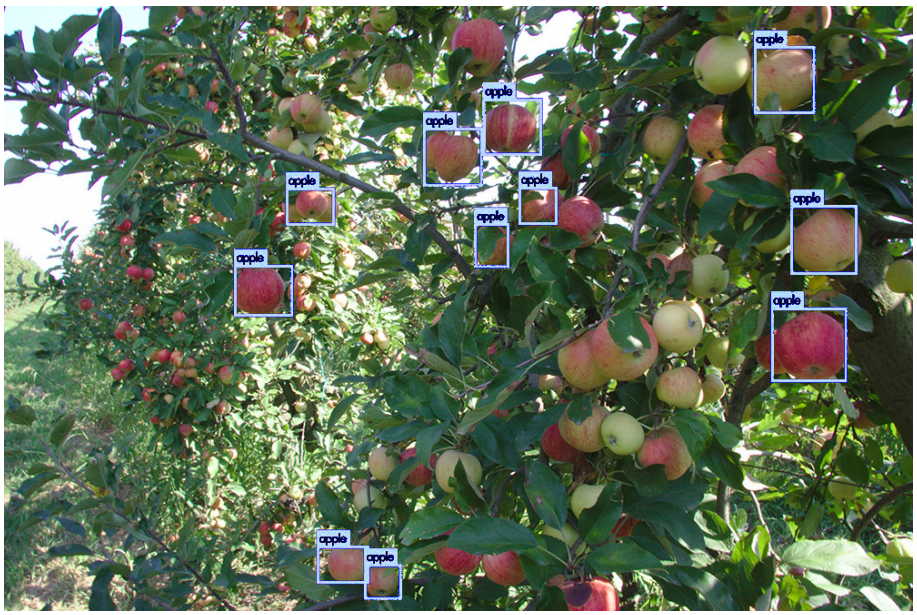
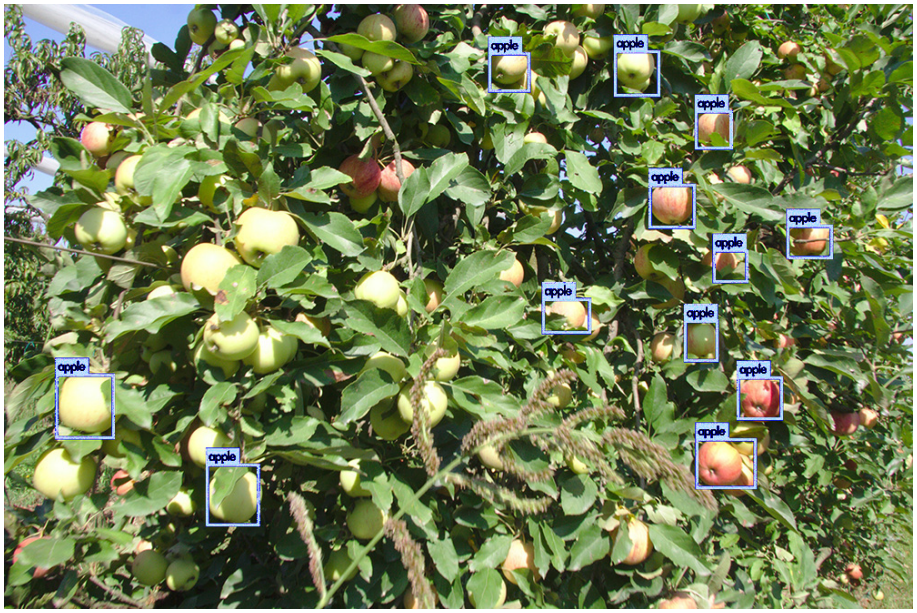


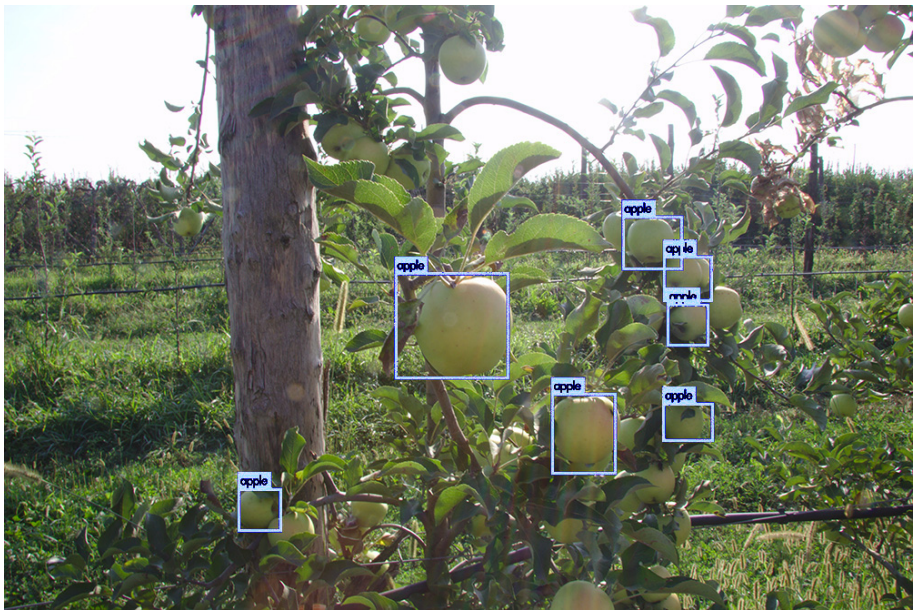
Figure 6.12: Some pictures returned after grid-search on Tiny YOLOv3

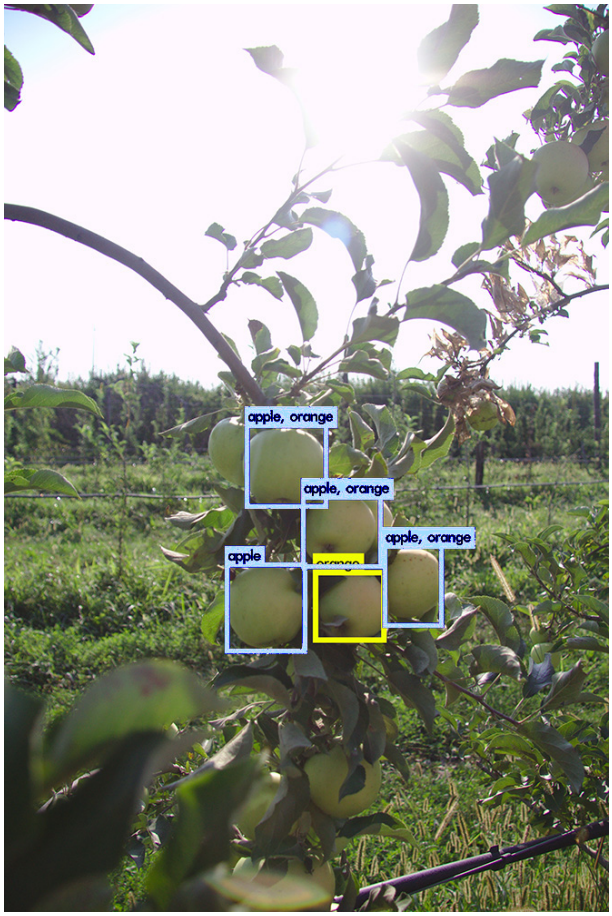
Here are shown the apples detected by Tiny YOLOv3 on JPEG images and confidence score of at least 0.25



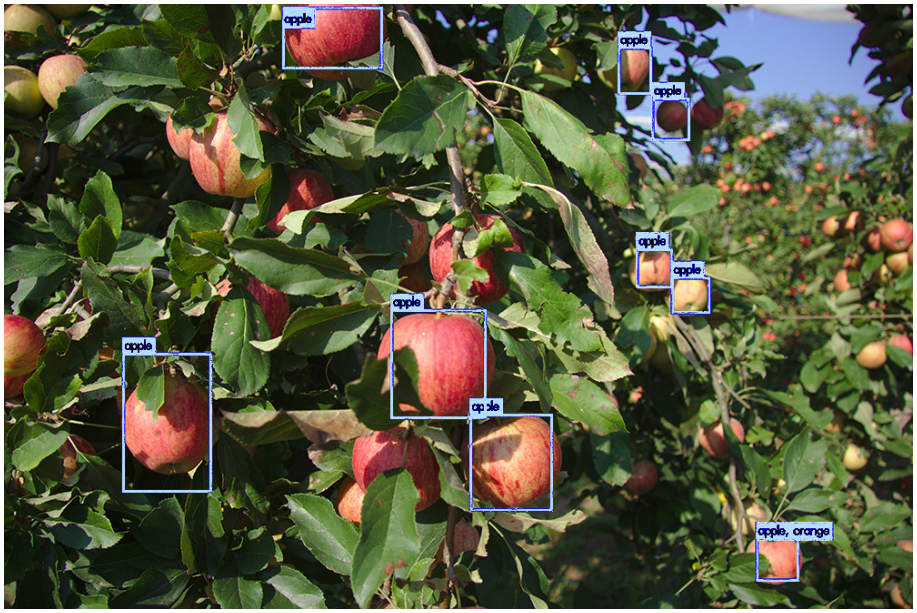

apple
apple
apple
apple
apple
apple
apple
apple
apple
apple
apple

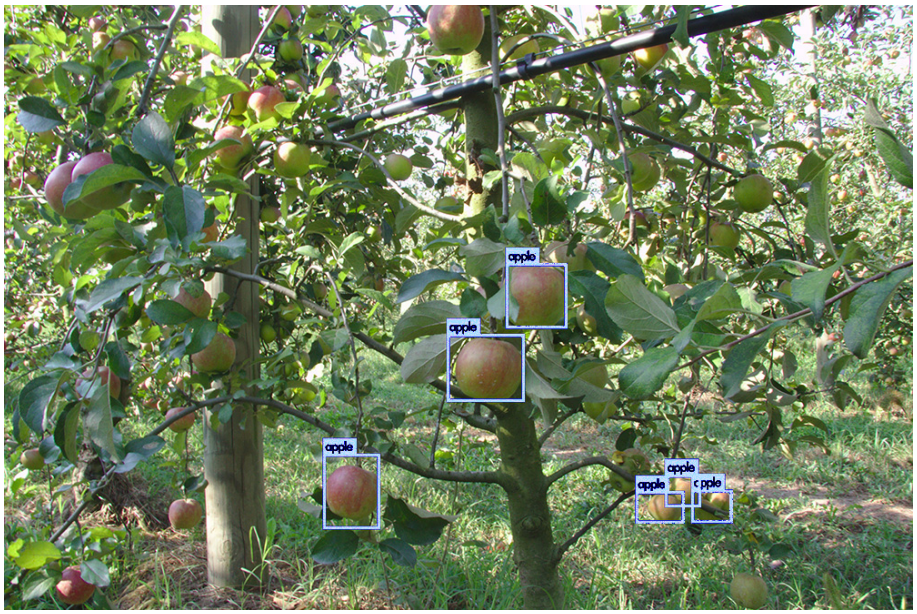

apple
apple
apple
apple
apple
apple
apple
apple
apple
apple
apple
apple


apple
apple
apple
apple
apple


apple, orange
apple, orange
apple, orange
apple


apple
apple


apple
apple
apple
apple
apple
apple
apple
apple, orange


apple
apple
apple
apple
apple
c pple

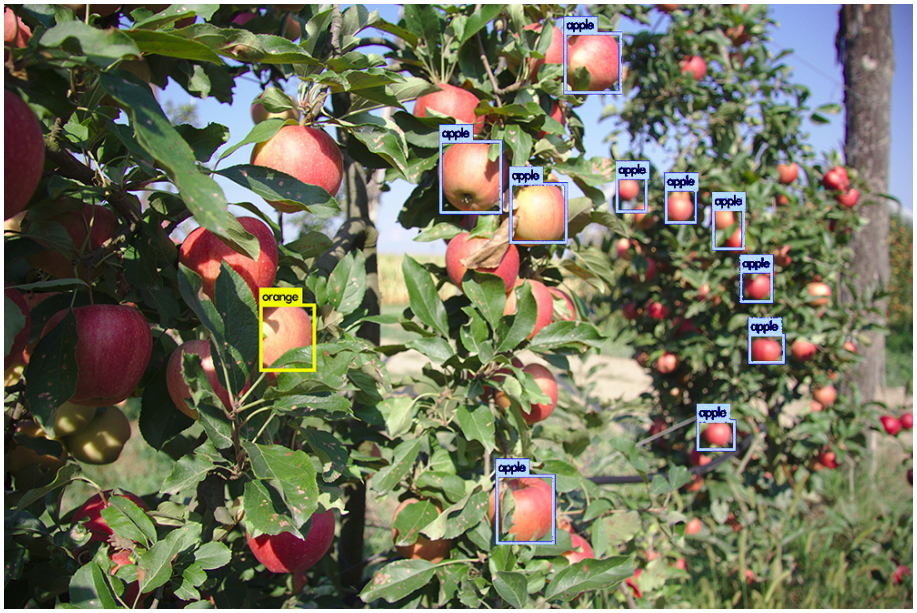
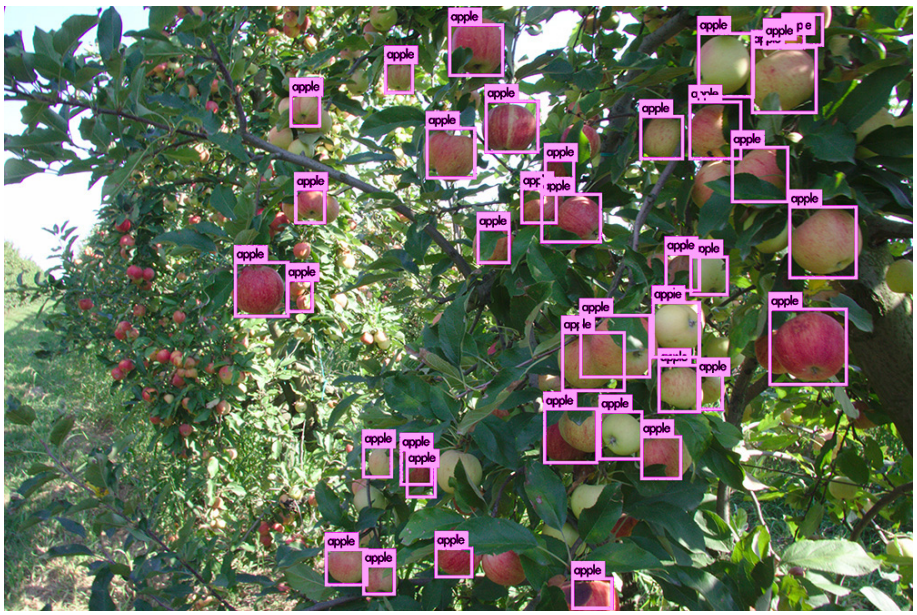
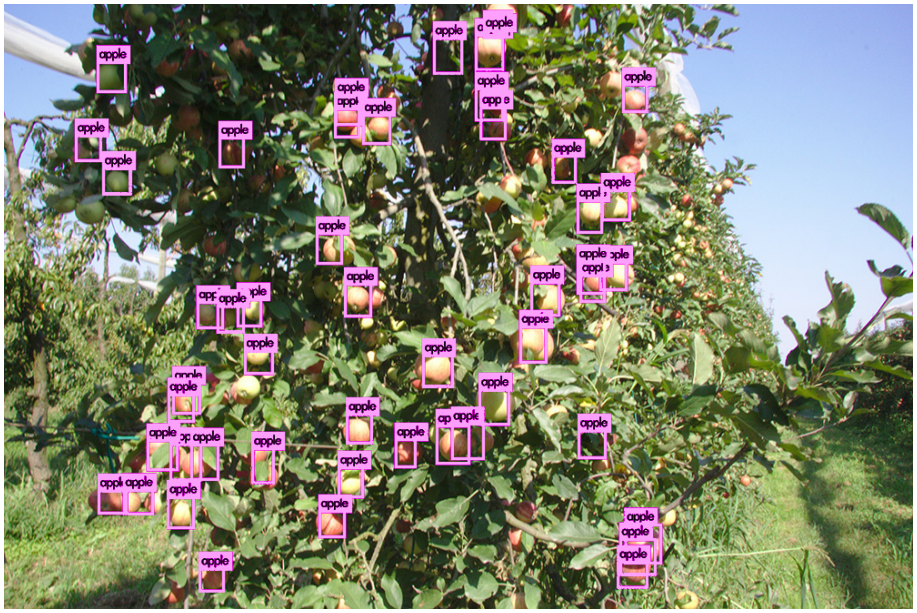
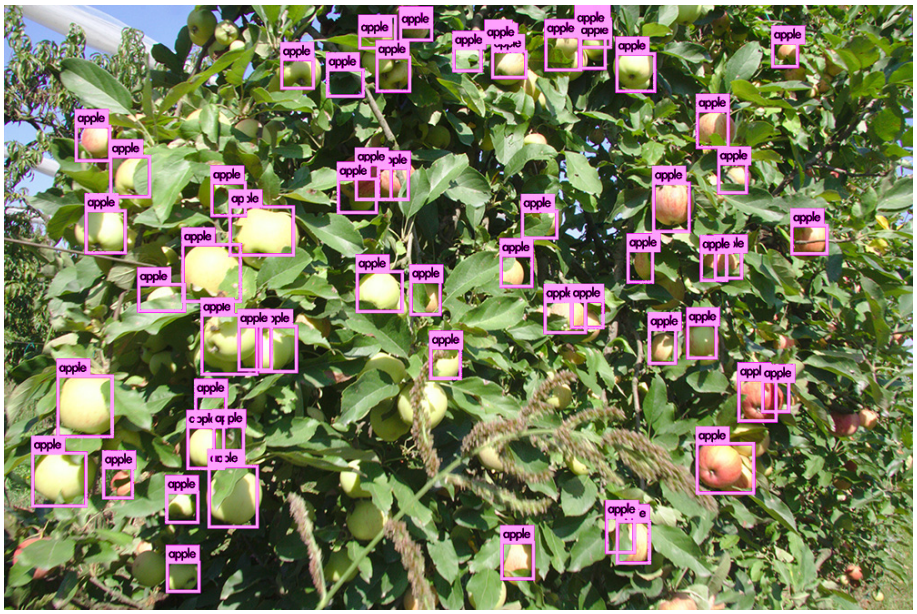

apple
apple
apple
apple
apple
apple
apple
apple
orange
apple
apple

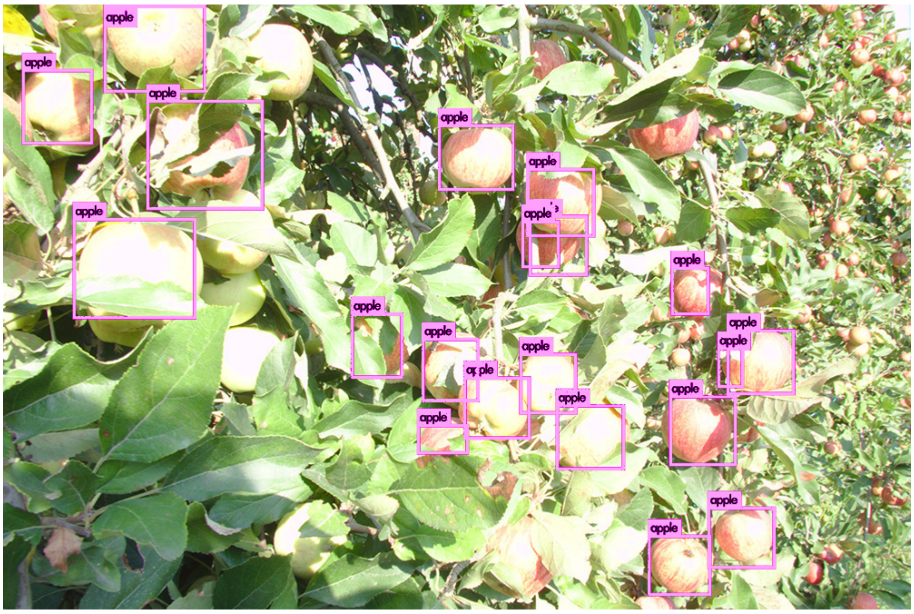
Figure 6.13: Detection with Tiny YOLOv3 on pictures in JPEG format

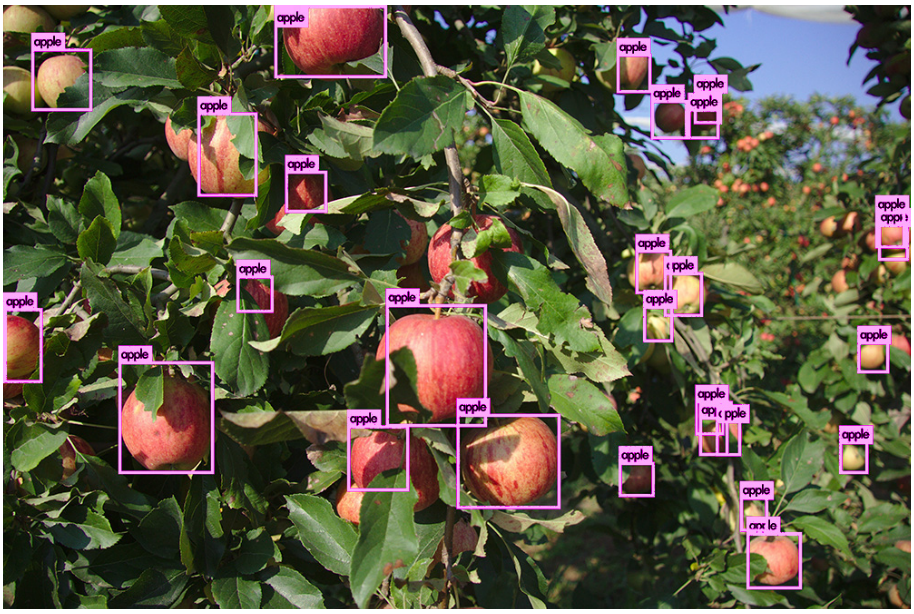
Detection results on JPEG images identified by the fine-tuned Tiny YOLOv3 and confidence score of 0.25.

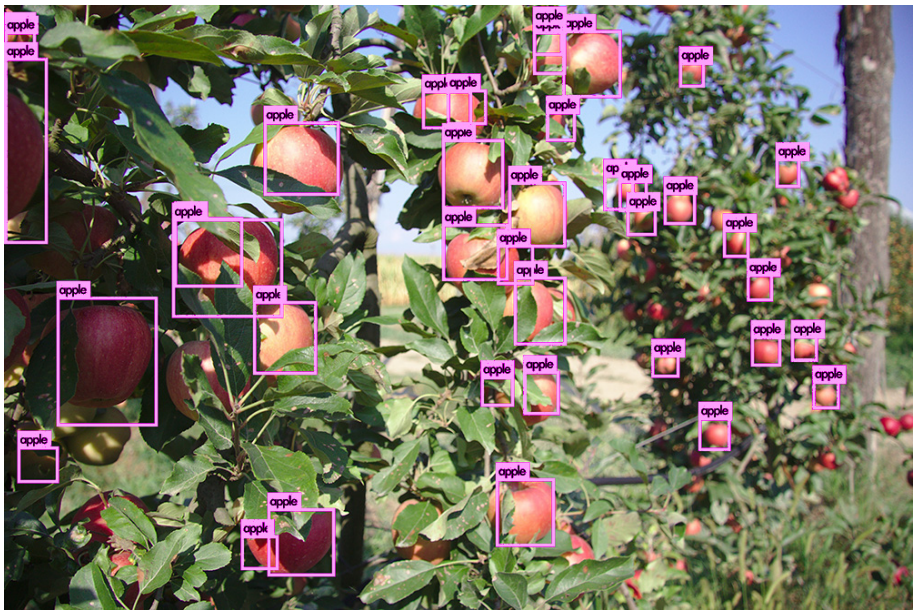







apple


apple


apple


apple

Figure 6.14: Detection with fine-tuned Tiny YOLOv3 on pictures in JPEG format

Bibliography

[1] Oded Cohen, Raphael Linker, and Amos Naor. Estimation of the number of apples in color images recorded in orchards. In *International Conference on Computer and Computing Technologies in Agriculture*, pages 630–642. Springer, 2010.

[2] Wikipedia contributors. *Artificial intelligence — Wikipedia, The Free Encyclopedia*. `https://en.wikipedia.org/w/index.php?title=Artificial_intelligence&oldid=907702868`, 2019.

[3] Marco Somalvico, Francesco Amigoni, and Viola Schiaffonati. *Treccani: La grande scienza. Intelligenza artificiale*. `http://www.treccani.it/enciclopedia/la-grande-scienza-intelligenza-artificiale_%28Storia-della-Scienza%29`, 2003.

[4] Herbert Alexander Simon. *The shape of automation for men and management*, volume 13. Harper & Row New York, 1965.

[5] Marvin Lee Minsky. *Computation*. Prentice-Hall Englewood Cliffs, 1967.

[6] Elio Piccolo. `https://areeweb.polito.it/didattica/gcia/Materiale_Didattico/Lucidi_Corso/6_Lucidi_retineur_nuovi`, 2010.

[7] Ivan Nunes and Hernane Spatti Da Silva. *Artificial neural networks: a practical course*. Springer, 2018.

[8] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

[9] Wikipedia contributors. Boltzmann machine — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Boltzmann_machine&oldid=912503270`, 2019.

[10] Yann LeCun, Léon Bottou, Yoshua Bengio, Patrick Haffner, et al. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[11] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[12] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[13] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.

[14] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 779–788, 2016.

[15] Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement. *arXiv*, 2018.

[16] Darknet. `https://github.com/pjreddie/darknet`.

[17] Joseph Redmon and Ali Farhadi. Yolo9000: better, faster, stronger. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 7263–7271, 2017.

[18] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. In *Advances in neural information processing systems*, pages 91–99, 2015.

[19] George A Miller, Richard Beckwith, Christiane Fellbaum, Derek Gross, and Katherine J Miller. Introduction to wordnet: An on-line lexical database. *International journal of lexicography*, 3(4):235–244, 1990.

[20] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[21] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. Microsoft coco: Common objects in context. In *European conference on computer vision*, pages 740–755. Springer, 2014.

[22] Tsung-Yi Lin, Piotr Dollár, Ross Girshick, Kaiming He, Bharath Hariharan, and Serge Belongie. Feature pyramid networks for object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2117–2125, 2017.

[23] Min Lin, Qiang Chen, and Shuicheng Yan. Network in network. *arXiv preprint arXiv:1312.4400*, 2013.

[24] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[25] Wikipedia contributors. Generative adversarial network — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=Generative_adversarial_network&oldid=912874477`, 2019.

[26] BuzzFeedVideo. You wont believe what obama says in this video! `https://www.youtube.com/watch?v=cQ54GDm1eL0`.

[27] Alina Kuznetsova, Hassan Rom, Neil Alldrin, Jasper Uijlings, Ivan Krasin, Jordi Pont-Tuset, Shahab Kamali, Stefan Popov, Matteo Malloci, Tom Duerig, et al. The open images dataset v4: Unified image classification, object detection, and visual relationship detection at scale. *arXiv preprint arXiv:1811.00982*, 2018.

[28] Angelo Vittorio. Toolkit to download and visualize single or multiple classes from the huge open images v4 dataset. `https://github.com/EscVM/OIDv4_ToolKit`, 2018.

[29] Labelimg. `https://github.com/tzutalin/labelImg`.

[30] map (mean average precision). `https://github.com/Cartucho/mAP`.

[31] Yolo-v3 and yolo-v2 for windows and linux. `https://github.com/AlexeyAB/darknet#yolo-v3-and-yolo-v2-for-windows-and-linux`.

[32] Yuan-Yuan Pu, Yao-Ze Feng, and Da-Wen Sun. Recent progress of hyperspectral imaging on quality and safety inspection of fruits and vegetables: a review. *Comprehensive Reviews in Food Science and Food Safety*, 14(2):176–188, 2015.

[33] AR Jimenez, R Ceres, and JL Pons. A survey of computer vision methods for locating fruit on trees. *Transactions of the ASAE*, 43(6):1911, 2000.