Universitat Oberta de Catalunya

Diseases Detection in Citrus Fruits Using Convolutional Neural Networks

Fernando López Laso
Computer Science Degree
Artificial Intelligence

David Isern Alarcón
Carles Ventura Royo

Juny 2018

FITXA DEL TREBALL

Títol del treball:	Diseases Detection in Citrus Fruits Using Convolutional Neural Networks
Nom de l'autor:	Fernando López Laso
Nom del consultor/a:	David Isern Alarcón
Nom del PRA:	Carles Ventura Royo
Data de lliurament (mm/aaaa):	06/2018
Titulació o programa:	Grau Enginyeria Informàtica
Àrea del Treball Final:	Intel·ligència Artificial
Idioma del treball:	Anglès
Paraules clau:	Citrus Diseases, Convolutional Neural Network, Android Development

Resum del Treball

El *Deep Learning* s'està convertint en el darrer lustre en la tècnica més prometedora per a aplicacions de visió per computador. En aquest treball s'empren tècniques d'aprenentatge profund i s'entrena una xarxa neuronal convolucional per a ajudar a la predicció de malalties en fruites. Com aquest camp és extremadament ample, s'ha particularitzat en el cas d'una de les famílies de que combina ésser un mercat econòmic de gran importància, a més de tindre moltíssims problemes de malalties que provoquen pèrdua de valor del fruit entre d'altres. Després, amb la xarxa entrenada i validada, es desenvolupa una aplicació per a dispositius mòbils *Android* per a poder emprar aquella com a eina d'ajuda a la presa de decisions de professionals del sector fitosanitari.

Per a desenvolupar la xarxa s'empra la llibreria PyTorch, la qual es fa servir amb el llenguatge de programació Python, que s'està convertint a l'estàndard *de facto* a la indústria. Per la seua banda, l'aplicació d'Android es desenvolupa amb Kotlin, i per a poder emprar la xarxa neuronal desenvolupada a la primera part de la tesi, es fa servir la llibreria TensorFlow Lite, versió optimitzada del TensorFlow per a dispositius mòbils.

Abstract

The *Deep Learning* is becoming the most promising technique for computer vision applications. In this work, deep learning techniques are used and a convoluted neuronal network is trained to help to predict fruit diseases. As this field is extremely wide, the work is focused in the case of one of the families that combines both being a very important economic market, and having many problems of diseases that cause loss of fruit value among others problems. Then, with the network trained and validated, an application is developed for *Android* mobile devices to be able to use it as a tool to help decision-making of professionals in the phytosanitary sector.

To develop the network, the PyTorch library has been used. This library is programmed with the Python programming language, which is almost the standard *de facto* in the industry. In addition, the Android application is developed with Kotlin, and to be able to use the neuronal network developed in the first part of the thesis in the Android operative system, TensorFlow Lite library has been also used.

Contents

List of Figures **v**

1 Introduction **1**
1.1 Content and justification of the Thesis . . . 1
1.2 Thesis' objectives . . . 2
1.3 Strategy and Methods . . . 2
1.4 Thesis planning . . . 4
1.5 Brief summary of the obtained products . . . 5
1.6 Short description of the rest of the Thesis' chapters . . . 6

2 State of the Art **7**
2.1 Introduction . . . 7
2.2 Fruit Recognition and Classification . . . 7
2.3 Detection and Recognition of Fruits and Plants Diseases . . . 7
2.4 Use of Convolutional Neural Network instead of classical approaches . . . 8
2.5 Related Studies . . . 11

3 Citrus Pests and Diseases **13**
3.1 *Alternaria alternata* . . . 13
3.2 Citrus leprosis virus . . . 14
3.3 *Diaporthe citri* . . . 14
3.4 *Elsinöe spp.* . . . 15
3.5 *Phyllosticta citricarpa* . . . 16
3.6 *Pseudocercospora angolensis* . . . 16
3.7 Septoriosis . . . 17
3.8 *Xanthomonas citri* . . . 17
3.9 Pests, Healthy Fruits and Physiological Disorders . . . 18

4 Data Preparation **19**
4.1 Image sources . . . 19
4.2 Overfitting problem . . . 20
4.3 Data Augmenation . . . 21
4.4 Data Set and Data Loading in PyTorch . . . 21

5 Convolutional Neural Network **25**
5.1 Neural Network . . . 25

5.2 Convolutional Neural Networks 33
5.3 Architectures 36

6 Android Application 39
6.1 Android Development and Framework Basics 39
6.2 Application Architecture 40
6.3 Class Diagrams 41
6.4 Libraries 41
6.5 User Interface 46
6.6 Data Base 48
6.7 Results 50

A Asynchronous work in Android Framework 53
A.1 Introduction 53
A.2 Java Threads 53
A.3 HaMeR framework 54
A.4 AsyncTask 54
A.5 Loaders 54
A.6 Service e IntentService 54
A.7 Anko extensions 55
A.8 Worker Manager 55
A.9 RxJava 55

Bibliography 57

List of Figures

2.1 Performance of different techniques in fungal diseases recognition. 8
2.2 Accuracy for SVM classifier using RGB and HSV images and feature extraction for some apple diseases, from [11]. 9
2.3 Pipeline followed in [11]. 10
2.4 Architecture of Inception-v3 network. 12

3.1 Tangeringe infected by *Alternaria alternata*, from [20]. 13
3.2 Unmature orange fruit with leprosis symptons from [45]. 14
3.3 *Diaporthe citri* over orange, from [3]. 14
3.4 Orange infected by *Elsinöe australis*. 15
3.5 Orange infected by *Elsinöe australis*. 15
3.6 A Valencia sweet orange infected by *Phyllosticta citricarpa*, from [38]. 16
3.7 Orange fruit in Ghana infected by *Pseucocercospora angolensis*, from [21]. 16
3.8 Advanced symptoms of Septoria spot, from [47]. 17
3.9 An advanced stady of the citrus canker, from [39]. 17
3.10 Fruits with damages produced by *Thysanoptera* 18
3.11 Orange with *Diaspididae* female over it. 18
3.12 A hole produced by *Orthoptera*. 18

4.1 Error dicreasing when number of epochs become higuer, from [30]. 20
4.2 Random variation of accuracy in an overfitted model, from [30]. 20

5.1 Perceptron with one input. 25
5.2 Neural Network with four inputs and three neurons in one single hidden layer. . . 26
5.3 Output for different models of activation functions. 28
5.4 Evolution of cost function: batch vs minibatch GD, from Imad Dabbura. 32
5.5 Application of a kernel over an image from the Government of India. We can see how the kernel is applied over neighboring pixels. 34
5.6 Convolution of a single kernel over an RGB image. 34
5.7 Two kernels convolutions. 35
5.8 Accuracy of different pretrained CNN architectures with citrus diseases dataset. . 36
5.9 Training error for CIFAR-10 image set using non-residual networks, from [16]. . . 37
5.10 Basic residual block, from [16]. 37

6.1 MVP with interactions between its parts, from Techyourdance.com. 41

6.2 Class diagram with classes that show elements on the screen and with which the users interact. 42
6.3 Class diagram with classes that represent presenters, model and use cases. 43
6.4 Screens and navigation in the app. 47
6.5 Failed, the good one is the second choice, *Phyllosticta*. Image from the validation set. 51
6.6 Good, pest, *Aonidiella aurantii*, image neither from the validation nor test set. . 51
6.7 Good prediction, melanosis disease produced by *Diaporthe citri*. 51
6.8 Failed, damaged produced by low temperatures. 51
6.9 Right prediction. 51
6.10 Almost a perfect prediction. 51
6.11 Good prediction. 52
6.12 Good prediction with an image not in the dataset. 52
6.13 Good prediction as previous result, now getting the image from the cellphone storage instead of taking a picture. 52

Chapter 1

Introduction

1.1 Content and justification of the Thesis

This work is about the recognition of citrus fruits diseases using Deep Learning. The citrus sector is a big economic sector. If we think about the biggest sub-sector, orange production, the world production in 17/18 season is estimated to be about fifty metric tons millions [31]. We are talking about a billionaire sector, and around the 10% of that billionaire production will be exchanged between countries. All these fruits transactions, specially if the involved countries are in different continents, must meet strict phytosanitary specifications by the destination countries.

These specifications must be checked in both origin and destination, so any aid to the control of the quality of the fruit will represent an improvement on the long chain that could start in a tree in Australia and end in our dish in Europe. This improvement would be in time[1], but also economic, allowing even the producers to reject some productions to exports to some specific countries that would reject the cargo, and reorienting the fruit to other markets that will accept that shipping.

If we thing specifically in our continent, the directive 29/2000 of the European Union Council [5] has a list of those pests and diseases that cannot enter in any territory inside the EU borders, and has also a list about some specific diseases in citrus that, in case of being present in some import, will result in a product rejection or its totally destruction.

On the opposite, when we export vegetable products to other countries, we must take care about following the phytosanitary regulation by the destination countries.

In both cases, imports and exports, not-always well trained phytosanitary inspectors must decide, in seconds, if a sample of a container should be rejected because of some spots in the fruit skin, or in the leafs of a plant, because of some root deformation, etc.

Because of that reasons, an application that aids in the inspectors decision making will be helpful. But this is not the only way an intelligent algorithm could help in this field, and also producers can take profit of a software that helps them to realize which problems exists in their lands, not only for reject them to some markets, but also to have a help for the exact phytosanitary product that should be use.

[1]Being orders of magnitude faster than taking a sample, sending to the lab and wait for culture results.

1.2 Thesis' objectives

The main goal is to get a Convolutional Neural Network that classify with a high confidence citrus diseases, and develop and Android application that allow its use on the field. These goals can be splitted in a series of related tasks:

- Neural Network
  1. data set preparation;
  2. network architecture selection;
  3. network training and validation; and
  4. hyperparameter modification and network parameters (re-training) improvement.
- Mobile Application
  1. project and *gradle* build file definition;
  2. limitation of desirable mobile application features;
  3. class diagram and application architecture selection;
  4. libraries selection; and
  5. implementation and tests.

1.3 Strategy and Methods

1.3.1 Dataset

One of the most important thing in this kind of work is the data we'll use for the neural network training. As is explained in [48], *"In practice, very few people train an entire Convolutional Network from scratch (with random initialization), because it is relatively rare to have a dataset of sufficient size. Instead, it is common to pretrain a ConvNet on a very large dataset (e.g. ImageNet, which contains 1.2 million images with 1000 categories), and then use the ConvNet either as an initialization or a fixed feature extractor for the task of interest."*. This will be our approach for training the network[2]. The reason behind this is clear: a Neural Network cannot be initialized to 0 like other machine learning algorithms like Linear Regression [29], so we must use non-zero values. The random initialization approach has the disadvantage that our training will be longer and less accurate because we won't be able to detect some of the features than images have, features that can be extracted with a very large dataset training.

1.3.2 Technologies

Because of the nature of the products will be released with this thesis, very different technologies have been used in each of the parts in this thesis.

[2]**Chapter 3** is dedicated to the dataset preparation.

Neural Network

Programming Language In the Machine Learning field, and in particular in Deep Learning, Python is almost the standard in the industry. There are other options like Java, C++, Lua or Matlab/Octave, but these, especially the last two, are almost residual in research and information available on the web. So Python has been the choice to this thesis. The version used has been Python 3.6.5, and some other packages has been used to develop all the code, tables and figures that are in this document:

- *numpy*, the standard in numerical computation;
- *matplotlib, seaborn*, for data visualization;
- *pandas*, to work with results obtained in an easy way;
- *pytorch*, deep learning framework; and
- *keras*, a backend over other frameworks like TensorFlow or CNTK.

Deep Learning Framework There are many frameworks in the market that we can use for developing a Neural Network. Between them we have TensorFlow, caffe2, DL4J, CNTK or PyTorch. Between the last one, *PyTorch*, is the selected for developing the model. It has many interesting features that have made it the choosen, among which we can stand out:

- uses python as programming language;
- dynamic computation graph;
- easy to develop models from scratch and to learn, at least easiest thtn other famous frameworks like TensorFlow[3], with an object oriented approach easiest to understand than the declarative that must be used with this; and
- very focus on GPU computation, making very easy to use graphic cards, adding few extra lines to the code.

Cellphone Development

Platform On the other side, for mobile development, Android is the platform to go with. It is the most used, it is the cheaper, and also it is opened in front of iOS. For Android development we have many options, but finally the most common way, the use of the native[4] framework, has been elected.

Programming Language Although Java is the most used yet, the application has been develop in Kotlin. There are some advantages for this election, but the main for this choice is

[3]The use of Keras as a frontend makes easier to use TensorFlow. In fact, it will be used to export the model to TensorFlow.

[4]When we are talking about Android, *native* does not have the meaning of been a compiled language like in iOS because of the use of ART, and specialized virtual machine that make possible the use of this operative system in so many different devices.

the simplicity of the language in comparision with Java, the least verbosity. That's true because:

- easy use of lambda expression;
- extension functions;
- data classes;
- type inference;
- smart casts; and
- null-safety.

Other advantage is the complete integration of Java way in Kotlin and viceversa. Thus, we can call Java code from our application without any modification, and that allows we can work with TensorFlow Lite[5] in an easy manner.

1.3.3 Workstation Specifications

This thesis and all the code developed for it was done in a laptop with the next main hardware specifications:

- processor: Intel(R) Core(TM) i7-4700MQ CPU @ 2.40GHz;
- ram: 16 GB (2x8) DDR3 1600 GHz
- CPU: Nvidia GeForce GTX 765M, 2 GB GDDR5

As we'll be explained later in this thesis, the GPU features, because this is an old one, will make than some of the most famous convolutional network architectures cannot be used.

The operative system was the Ubuntu 16.04.4 with 4.13.0-37-generic kernel. As code editor, both Sublime Text and Emacs25 where used, also with Jupyter notebooks on the web browser. For Android development Android Studio was a mandatory requirement.

1.4 Thesis planning

February, 21th 2018 Project starts.

March, 6th 2018 *PAC 0*

- title, problem to solve and project goals definition.

March, 19th 2018 *PAC 1*

- goals' fine definition;
- work plannig;
- dataset preparation; and

[5]Its API is developed in Java, even though the library itself is programmed in C++.

- state of the art in the thesis field.

April, 30th 2018 *PAC 2*

- dataset augmentation [1 week, 19th to March 26th];
- network architecture selection [1 week, 19th to March 26th];
- training and validation [4 weeks, March 26th to April 23th];
- mobile application structure and architecture selection [2 weeks, March 26th to April 6th]; and

May, 28th 2018 *PAC 3*

- class diagram and use cases definition [2 weeks, 6th to April 23th].
- mobile app implementation and application test [3 weeks, 24th April to May 15th]; and

Juny, 6th 2018 *PAC 4* memory thesis writing.

Juny, 13th 2018 *PAC 5a* thesis presentation preparation.

Juny, 25th 2018 *PAC 5b* thesis public defense.

1.5 Brief summary of the obtained products

After each of the steps that must be made during the development of this thesis, one product will be delivered to feed the next step. Therefore, we can see the process as an stream that starts with the original images and finally ends with the disease prediction in a real scenario.

We can summarize all the products after those steps as follows:

- *Original images → Splitted images.* Using multi-cropped window, the original dataset will be increase, splitting the original images in subimages of 500x500 pixels;
- *Splitted images → Transformed images.* Using `torchvision` and `PIL` packages, the splitted data will be transformed with rotations, flips and bright modifications;
- *Transformed images → PyTorch model.* The transformed images will be use to train and validate a CNN, and this network will be defined as a PyTorch model;
- *PyTorch model → ONNX model.* `ONNX` is an open-source initiative that allows to export the models from an specific machine learning or framework to a common format that can be imported by a different library;
- *ONNX model → TensorFlow model.* The previous model will be imported in TensorFlow, and we will export it to obtaint the `*.pb` file that defines the result we obtained in previous steps using PyTorch;

- *TensorFlow model* $\rightarrow$ *TensorFlow Lite model.* With `toco`, a command line application, we can optimize the TensorFlow model, making it smaller so low resources devices like cellphones can use neural networks without expending too much time; and
- *TensorFlow Lite model* $\rightarrow$ *Classify image.* Finally we'll use the last `*.lite` model to classify the image captured by the cellphone camera.

So, at the end, the final product will be an android application that will use the optimized TensorFlow Lite model to classify the images we capture with our hand-held devices.

1.6 Short description of the rest of the Thesis' chapters

In the next chapter, *Chapter 2 - State of the Art*, a short description of the current situation in the used of different techniques of computer vision used in fruits and diseases recognition is presented.

Chapter 3 - Citrus Pests and Diseases presents a brief description an one image per class the application we are developing will try to classify.

Chapter 4 - Data Preparation is about the process of preparing the dataset. Here we'll explain which disease we are going to detect, which were the data sources, and also which techniques have been employed trying to augment the size of the dataset.

Chapter 5 - Convolutional Neural Network has a small theory presentation of Convolutional Neural Networks, from neurons to kernels. However, it is focused on the options that have been taken in this project, with some explanations about functions, optimizations, hyperparameter values, etc, used in this thesis.

Finally, in the *Chapter 6 - Android Application*, we'll define the basics of how the application will be developed: application architecture, libraries used, etc.

Chapter 2

State of the Art

2.1 Introduction

The problem we are going to deal with has been subject of some research in the last lustrum, thanks to the improvements and generalization of computer vision techniques.

In agricultural research was (and currently is) usual to find studies that, instead of RGB images, worked with hyperspectral images [15]. But the last years the improvement in machine learning and computer vision techniques and algorithms, also with the higher performance of the computers, have made that almost every new research works with visible images.

2.2 Fruit Recognition and Classification

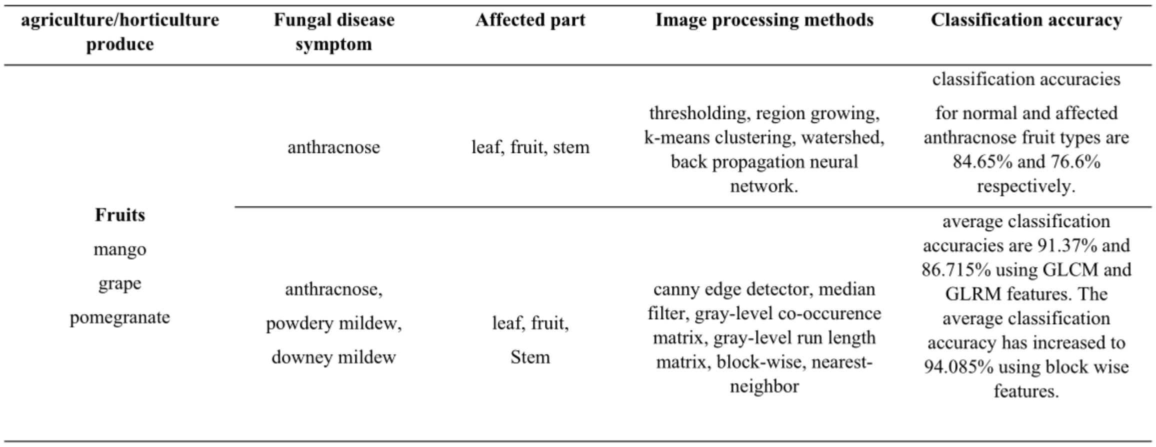
Neural networks has been used in the problem of the fruit recognition [52], and even deep neural networks has been proof their usefulness in more recent papers [37]. But the classical approach to this problem[1] was the use of some unsupervised algorithm for segmentation (usually k-means), followed by feature extraction, and finally classification with Supported Vector Machines or some other variation of SVM [35] using those features[2].

2.3 Detection and Recognition of Fruits and Plants Diseases

For pests and diseases detection the techniques usually used are the same than those explained above, that is, segmentation with k-means, features extraction with some computer vision techniques and classification with SVM [2,6,11,41]. This pipeline is also used in some research about fruit grading [27], where also neural networks have been used satisfactory [23]. Some variations, changing the classification algorithm, can be seen in [23] where "k-nearest neighbour" is used,

[1]And for almost every other problem of object classification.

[2]This has been during many decades, and even in the seminal paper Gradient-Based Learning Applied to Document Recognition [24] written in 1998 we can read "Historically, the need for appropriate feature extractors was due to the fact that the learning techniques used by the classifiers were limited to low-dimensional spaces with easily separable classes".



agriculture/horticulture produce	Fungal disease symptom	Affected part	Image processing methods	Classification accuracy
Fruits mango grape pomegranate	anthracnose	leaf, fruit, stem	thresholding, region growing, k-means clustering, watershed, back propagation neural network.	classification accuracies for normal and affected anthracnose fruit types are 84.65% and 76.6% respectively.
	anthracnose, powdery mildew, downey mildew	leaf, fruit, Stem	canny edge detector, median filter, gray-level co-occurence matrix, gray-level run length matrix, block-wise, nearest-neighbor	average classification accuracies are 91.37% and 86.715% using GLCM and GLRM features. The average classification accuracy has increased to 94.085% using block wise features.

Figure 2.1: Performance of different techniques in fungal diseases recognition.

and in [1, 34] we can read summaries about different techniques used in different papers. In fact, in [34] we can see figure 2.1 1 about the accuracy of diferent techniques used for fungal disease recognition, being the accuracy defined by 2.1.

$$accuracy = \frac{total\ number\ of\ images\ correctly\ classified}{total\ number\ of\ images\ used\ for\ testing} \cdot 100 \tag{2.1}$$

The classical use of *segmentation* → *feature extraction* → *classification* is a complex pipeline, and is very sensible to the extraction algorithm, but with it very good results has been achieved, even with small number of training cases, as we can see in 2.2.

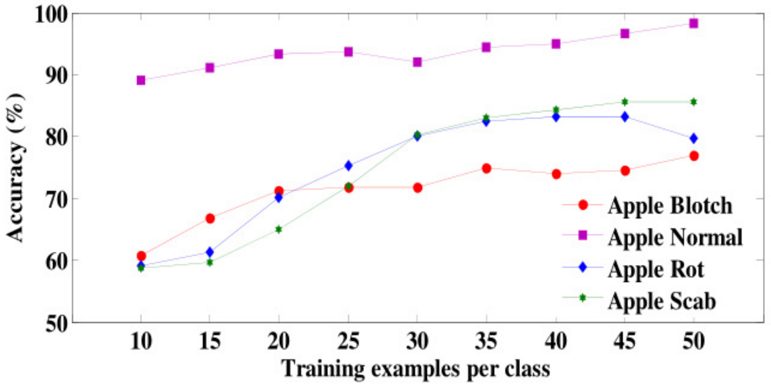
This image is the result of applying the complex pipeline explained in 2.3.

In the next chapter the reasons why we can try with a different approach will be explained, but now we can explain some of the majors problems of this solution and the rest of solutions found in research papers that follow a similar scheme. Thus, we can see that the full algorithm is made with a sequence of other algorithms. And this algorithms need to be adjusted or fine-tuned for every specific problem, testing with different space colors, different number of clusters[3], different clustering and/or classification techniques... Those inconvenient does not exist, or at least are minimized, if we use deep learning.

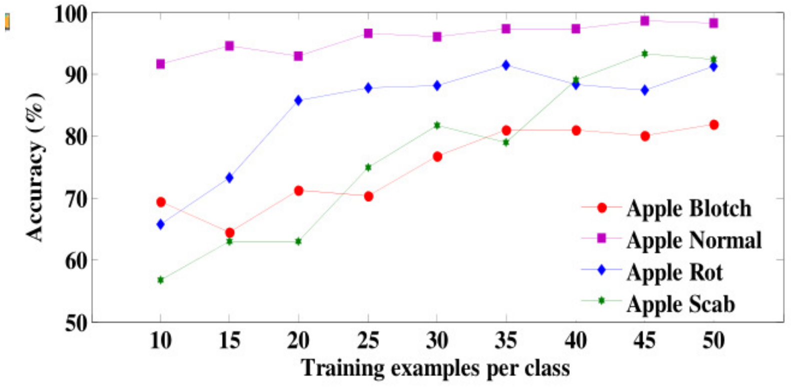
2.4 Use of Convolutional Neural Network instead of classical approaches

As was explained in 2.2 and in 2.3, a classical approach can be defined as an algorithm that first of all extract the features we want to study from the image, and after that uses some classification algorithm to make the prediction. This pipeline has proven to achieve good results in recognition

[3]As we'll see, the use of four clusters for melanosis recognition isn't enough, whereas maybe it is too much for *pseudocercosporas* or damages produced by insects of *orthoptera* genre.



(a) LBP in RGB color space



(b) LBP in HSV color space

Figure 2.2: Accuracy for SVM classifier using RGB and HSV images and feature extraction for some apple diseases, from [11].

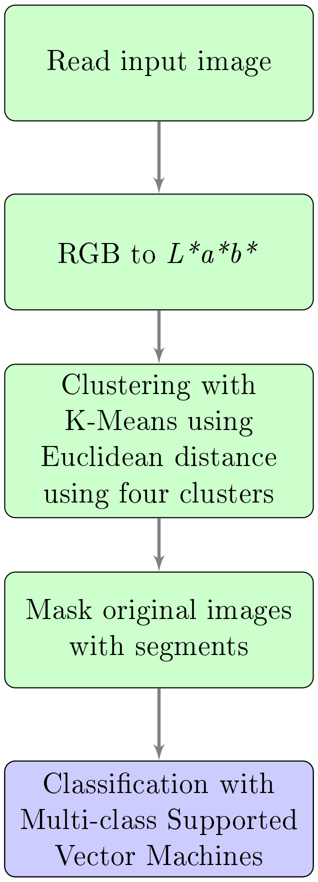

Read input image
RGB to $L*a*b*$
Clustering with K-Means using Euclidean distance using four clusters
Mask original images with segments
Classification with Multi-class Supported Vector Machines

Figure 2.3: Pipeline followed in [11].

tasks, with both accuracy and performance. It has also some other advantages over deep learning techniques, among which we can highlight better performance and the use of small data-sets. But both can be avoided:

- the improvement of the hardware have made than more computational-intensive algorithm like deep neural networks can be trained in personal computers or cloud servers; and
- for some problems, currently there are huge data-sets that can be used to train neural networks, and for those problems like the one we are dealing with, where data-sets doesn't have thousand of images, data-augmentation techniques, and the use of pre-trained networks for either fine-tuning or using as feature extractors, allow to use deep neural networks even with data-sets than previously doesn't fit to this techniques.

So, why do we use CNN instead that pipeline? Those reasons explained above do not clarify why deep learning has been the choice for the thesis. We can summarize them in the next points:

- deep learning are very exciting topic, with researches in few different problems, thus there are many fiels, for example the one this study is about, where its usefulness has not be proven (or not); and
- as a main advantage in front of *classical approaches*, deep neural networks are more robusts and automated; there we need to use different algorithms, or use different features, in different problems, so we can say they are more hand-made algorithms; but here we have a very powerful techniques that, if they work in a problem like citrus diseases recognition, we can be pretty sure that it will work in apple or peaches diseases recognition, so it is easier to generalize and use as it in different but related problems.

As an example of its previous application, we can talk about the study of medical images, and specifically cancer detection from images. And on it convolutional neural networks has demonstrated to be a very good technique, with comparable results, when not better, than those from experienced dermatologist [12][4].

In fact, [12] will be use as a main reference for this thesis. Although skin cancer and fruit diseases are very different topics, for a machine learning algorithm they are very similar, with some spots and imperfections that should be classify over the regular human or fruit skin.

That paper uses Inception-v3 [44] as network architecture without any modification. The primary purpose in this thesis was to use it as starting point. But this concrete network is very complex as we can see at figure 2.4, and because the big quantity of hidden layers and their complixity, the number of parameters is huge, so the hardware used for the thesis wasn't enough to use it because of lack of RAM at the GPU.

2.5 Related Studies

Other papers about deep learning and convolutional neural networks will be used as very important references for this thesis [7, 10, 16, 43], even some on-line courses where these topics are explained in depth [18, 48].

[4]It has to be said than other algorithms, like *Minimum Spanning Trees*, have been object of successful research [9] in medical images recognition.

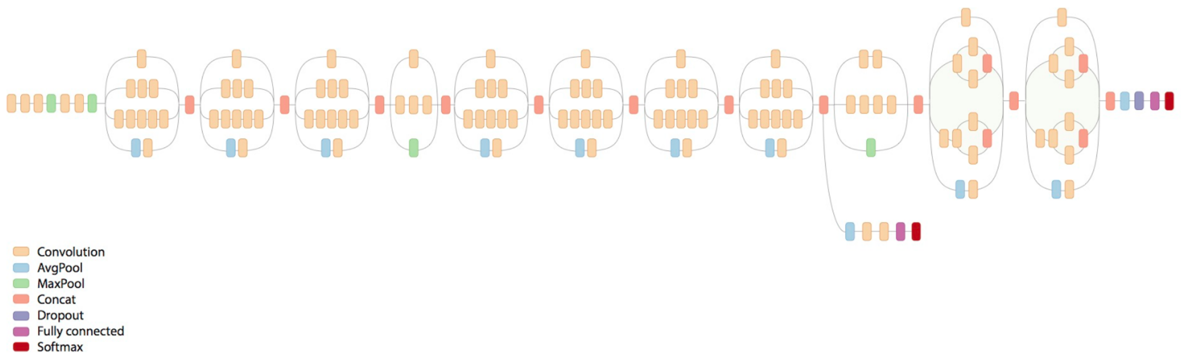


Figure 2.4: Architecture of Inception-v3 network.

Chapter 3

Citrus Pests and Diseases

In the next chapter an small description of the different diseases and classes the application is going to classify can be found.

3.1 *Alternaria alternata*

Fungus disease, typical in tangerine, with some importance also in tangelo. Symptoms in fruits are dark spots with yellow halos when the fruit is young, and from small specks to large pockmarks when the fruit is mature.

The lesions usually affect only the cortex and don't penetrate the locules. If the infection appears in young fruits, in can reduce the performance of the plots, and if it happens in adult fruits, the commercial quality dicrease.

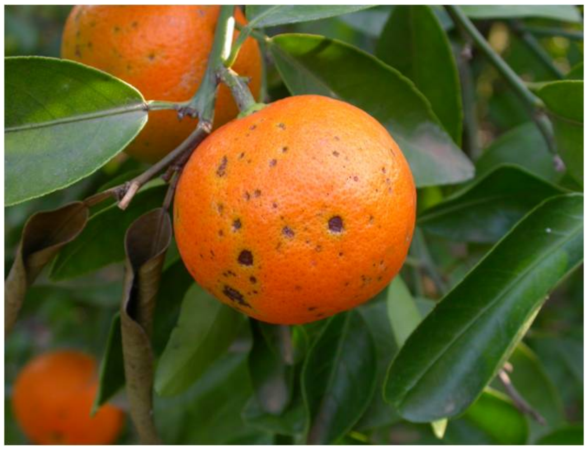


Figure 3.1: Tangeringe infected by *Alternaria alternata*, from [20].

3.2 Citrus leprosis virus

Virosis, caused by different virus transmitted by an acari of *Brevipalpus* genre that only affects every kind of citrus, but no other fruit o plant.

fruit lesions only affect the outer rind. Lesions appear as flat or depressed spots 10-20 mm wide with a necrotic center. It is common for a single fruit to exhibit up to 30 lesions covering a significant portion of the rind. . Over time the lesion becomes brown or blackish, sometimes depressed. Infected fruit tend to change color early and become susceptible to various rots. CiLV also induces premature fruit drop which greatly reduces yield.

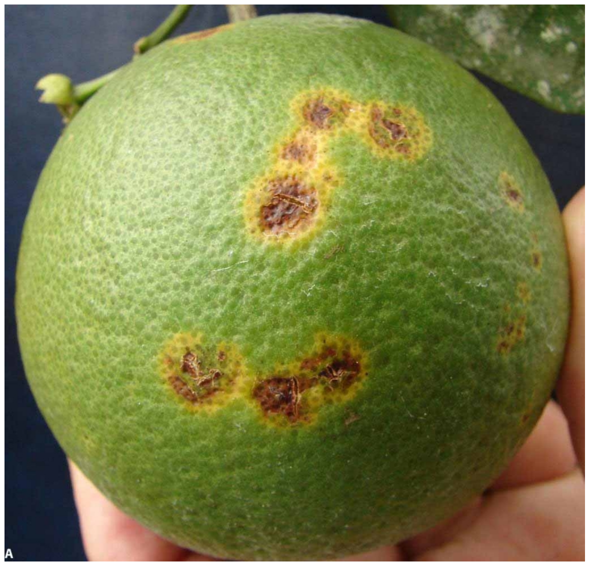


Figure 3.2: Unmature orange fruit with leprosis symptons from [45].

3.3 *Diaporthe citri*

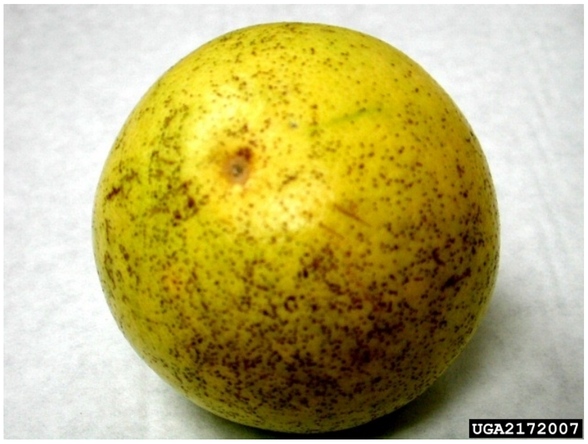


Figure 3.3: *Diaporthe citri* over orange, from [3].

Other fungus disese, its common name is *Melanosi*. Apparently, all citrus species are susceptible to the disease. The disease is characterized by the appearance of suberous crusts in fruits and leaves.

The damages of the melanosis in the fruits are limited to the external part of the crust, reason why, in general, it is considered an important problem only when the production goes to the markets of fresh consumption.

3.4 *Elsinöe spp.*

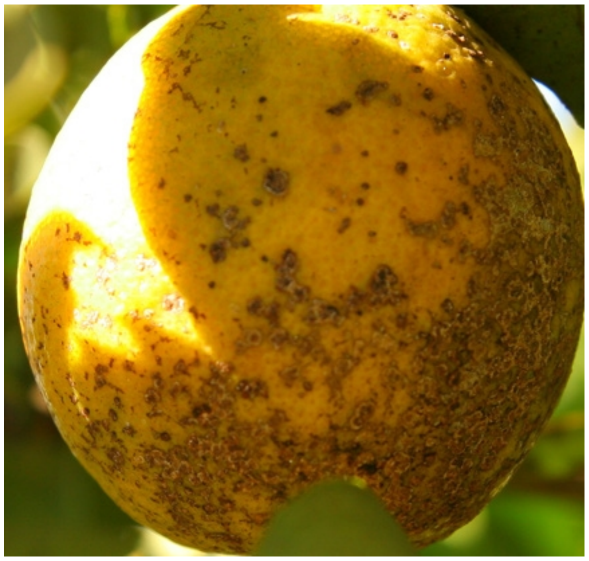


Figure 3.4: Orange infected by *Elsinöe australis.*

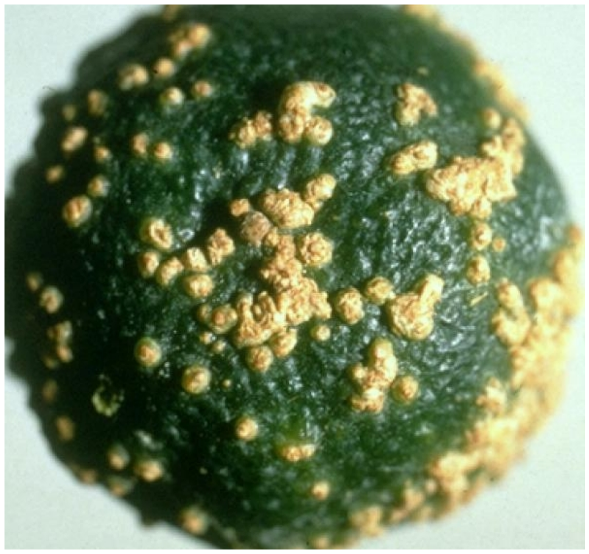


Figure 3.5: Orange infected by *Elsinöe australis.*

This class has been splitted in two, because the sympton differences between both *Elsinöe australis* and *Elsinöe fawcettii*. *Elsinöe spp* is an exotic fungal disease, with an specific article in the EU/2000/29 Council directive.

Although there is little affect on internal fruit quality, fruit are severely blemished rendering them unsellable in the fresh produce market

3.5 *Phyllosticta citricarpa*

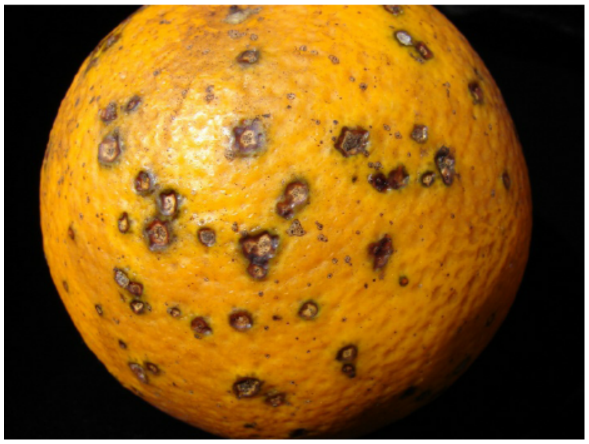


Figure 3.6: A Valencia sweet orange infected by *Phyllosticta citricarpa*, from [38].

The citrus black spot disease is one of the most important exotic pests for world production. It is a quarantine pest in UE, with also its own Council Executive Decision (2016/715) for the control of the importation of citrus from countries where it is present.

Phyllosticta citricarpa is a fungus disease that mainly affects the fruits of *Citrus* genre exclusively, and makes that its value decrease heavily. The lemon fruit is the preferred host, but it also attacks oranges, tangelos and tangerines.

The spot in the most common phase is characteristic, with a depression with usually two or three pycnidias inside it, but in the rest of the phases is very hard to confirm, because the symptoms are variable and can be confused with other diseases, for example with *Alternaria alternata*.

3.6 *Pseudocercospora angolensis*

Like citrus black spot, the cercosporiosis is an exotic disease reflected in the 2000/29/UE Council Directive. It is also like *Phyllosticta* a fungus disease.

The fruits have irregular necrotic lesions in the cortex surrounded by a yellowish halo. These lesions tend to aggregate and reach considerable diameters, which causes cracking of the bark and subsequent dehydration of the internal area of the fruit. In general, the damages of cercosporiosis are very serious, being a limiting factor for the cultivation of citrus fruits in many areas of the African continent.

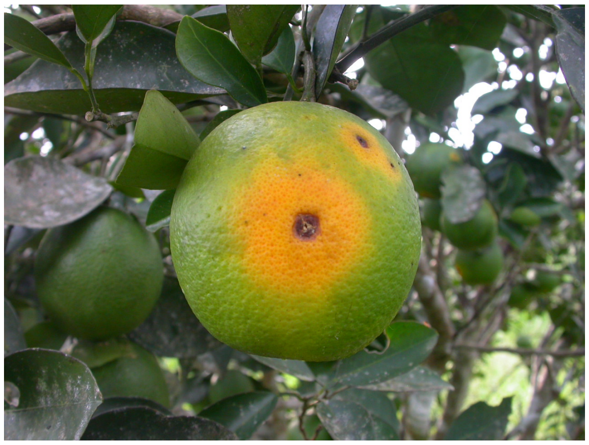


Figure 3.7: Orange fruit in Ghana infected by *Pseucocercospora angolensis*, from [21].

3.7 Septoriosis

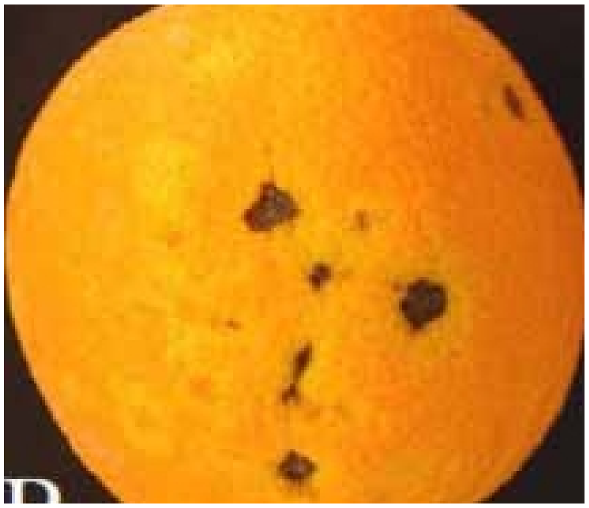


Figure 3.8: Advanced symptoms of Septoria spot, from [47].

Fungus disease that can attack all citrus cultivuar, with Valencia and Navel oranges as preferreded hosts.

Early fruit lesions are small light tan to reddish brown pits 1-2 mm in diameter that extend no deeper than the flavedo. Older lesions are darker sunken 20-30 mm in diameter. Dark fruiting bodies, pycnidia, may develop in these lesions. Lesions may appear in the form of "tear stains" patterns. Spots are more evident on ripe fruit [46].

3.8 *Xanthomonas citri*

Xanthomonas citri is the main causal agent of bacterial citrus canker. The lesions developed on the surface of the fruit, either scattered at isolated points or forming irregular patterns by the union of several lesions. The exudation of resinous substances can be observed in infected young fruits. The lesions never extend through the fruit shell.

Symptoms of citrus canker in fruits may be confused with similar symptoms in the form of scabs or spots caused by other bacteria or fungi that infect citrus or physiological disorders.

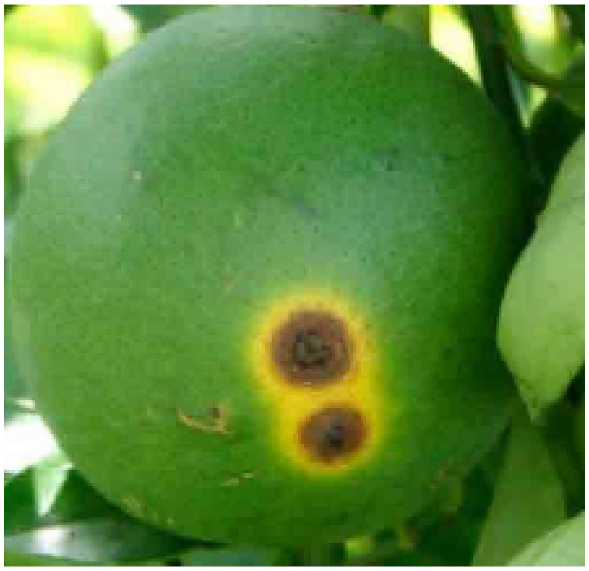


Figure 3.9: An advanced stady of the citrus canker, from [39].

3.9 Pests, Healthy Fruits and Physiological Disorders

Because there are other problems that can cause physical damages on the fruits, and because we must have a control group, there will be other classes that will represent oranges with no problems, those with physiological disorders (because of cold, phytosanitary treatment, etc), and also a class that will represent damages produced by pests, like aphids, thrips, *lepidoptera*, *diaspididae*, etc.

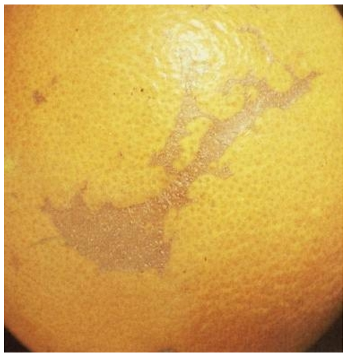


Figure 3.10: Fruits with damages produced by *Thysanoptera*
.

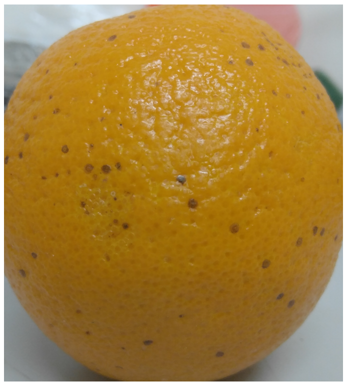


Figure 3.11: Orange with *Diaspididae* female over it.

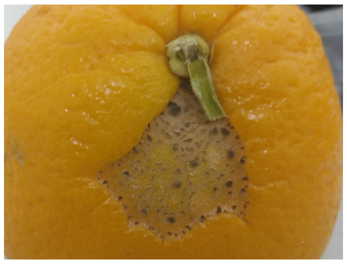


Figure 3.12: A hole produced by *Orthoptera*.

Chapter 4

Data Preparation

4.1 Image sources

One of the most important thing when we are dealing with the neural network training is getting the data we are going to use for training it. Usually there aren't enough amount of data for training, so instead of using random initialization the most common way of acting is pre-training the network with a very large dataset 1.3.

In our case, in our study field, doesn't exist public databases with thousands of images we can use for our work. Even the tens of public images with good quality we can find on the internet are usually for a few well-known diseases. So currently we can only try to diagnose diseases we have some images to train the network with, and then they will be those we are going to carry out the study.

The images that have been used as a main sources are from web pages belonging to some of the most important public departments and institutes around the world, and from some famous image repositories about plants and mycology that exists in the internet. If we don't use these, we don't be sure about the classification of the source images, so we won't be sure about the correctness of our network. In the next list we have those that have been used:

- USDA[1]
- IVIA[2];
- Forestry Images; and
- Mycology collections portal.

More complete collections exist, including USDA and universities private data-sets, but those weren't accessible at the time of this thesis is developed.

[1]United State Department of Agriculture.
[2]Institut Valencià d'Investigacions Agràries.

4.2 Overfitting problem

As was explained at 2.4, when we have few images neural networks haven't been used in a generalized way. That's because of when we train a neural network model with hundreds of images, we have to deal with the problem of overfitting. When we have such a small dataset, it is hard to generalize the model. For these datasets, even when increasing the number of epochs used to train the network reduces the error, the accuracy, despite seeming that should be improved, in fact won't do that and it will fluctuate stochasticaly over a value. Then, we said the network is overfitting [30, Chapter 3].

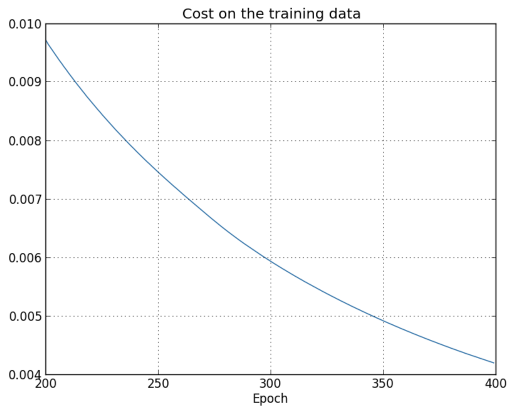


Figure 4.1: Error dicreasing when number of epochs become higuer, from [30].

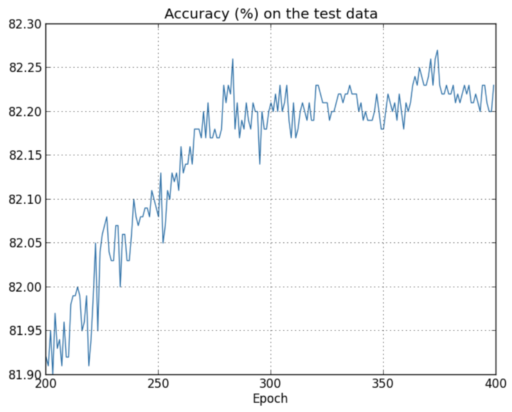


Figure 4.2: Random variation of accuracy in an overfitted model, from [30].

4.2.1 Transfer Learning

But, as we saw, the network is pre-trained previously to the training. That makes possible the use of our small datasets and reduces the over-fitting problem. As we explained, the result of training with small dataset cannot be generalized if they are training from scratch [49]. But using this over training we can generalize our results. This process, known as *Transfer Learning*, has proven, both in research papers [50][3] and in many real projects, its usefulness.

This will also increase our performance, reducing the duration of the pre-training each time we want to add some pest or disease in the future, or in the case we have new data to add to the collection.

[3]As is reflected in this paper, *"[...] initializing with transferred features can improve generalization performance even after substantial fine-tuning on a new task, which could be a generally useful technique for improving deep neural network performance".*

4.3 Data Augmenation

We can use other useful technique besides transfer learning. We can multiply the size of the dataset with data augmentation. We will use it for creating images with different modifications. The new images have an obvious disadvantage: they will be redundant. But this method actually works, as can be seen in the literature [7,29]. It doesn't work as well as getting new and different images, but it can be improved our network.

Different modifications can be applied to our data set [7]:

- *crop*: the sizes of the different images we have used is very different. We must use a fixed size that can be also taken by cellphone cameras. Also, this size must be enough to catch common marks and spots diseases and pests make on the fruits. A 300x300 size was chosen, and the first step in the image preprocessing was getting sub-images of this size from the originals;
- *rotation*: it must be done with a uniform random generation, with values between $[0, 360]$ degrees.
- *flipping*: mirroring images is another common technique used for increasing the size of data sets, with also a uniform random generated values between $[{}^{1}/_{1.6}, 1.6]$.
- *shearing*: also a uniform random generation was use, with values in the range $[-20, 20]$;
- *stretching*: finally, the stretching transformation was the only that needs different generator. In this case we must use a log-uniform generator, with values inside the interval $[{}^{1}/_{1.3}, 1.3]$.
- *color shifting*: that will change the value of some (or all) of the channels *(R, B, B)* in our images, with uniform random generation. We can use limit value of 20% trying to not make too darker o too white the image.

Other techniques like resizing or translation weren't used, because either they cannot create new distribution of pixels in local places on images, or they creates very bad quality images, far from those will be taken with cellphones. Also, even when shearing and stretching are useful techniques, they aren't use very often in deep learning [29], so finally we will use a combination of flipping, roration, cropping and color shifting to increase the size of our image set.

The application of this transformation has been made with a combination of shell scripting, and some functions we can find in `torchvision`, a complementary package of the framework `pytorch`.

4.4 Data Set and Data Loading in PyTorch

PyTorch work with directories to split data between classes. Therefore, we'll have a different folder for each of the groups explained in the chapter 3. But, as opposite as other libraries like TensorFlow, we have to split data between test and training data by ourselves:

- with TensorFlow, we have all data in different folders and the library can split it with a default or given percentage[4];

- with PyTorch we'll have two main folders, *training* and *test*, and inside them will have the same sub-folders, each of them corresponding to the same class, but obviously with different images.

PyTorch works with `Dataloaders`. In the next code we can see how we can finally augmented the data set and create them:

```
import torchsample
from torchvision import transforms, datasets

regularization_tuple = [0.485, 0.456, 0.406], [0.229, 0.224, 0.225]

transformations = {
    'training':
        transforms.Compose([
            transforms.RandomResizedCrop(224),
            transforms.RandomHorizontalFlip(),
            transforms.ToTensor(),
            torchsample.transforms.RandomRotate(20),
            torchsample.transforms.RandomRotate(-20),
            transforms.Normalize(regularization_tuple)
        ]),
    'test':
        transforms.Compose([
            transforms.Resize(256),
            transforms.CenterCrop(224),
            transforms.ToTensor(),
            transforms.Normalize(regularization_tuple)
        ]),
}

source_folder = 'diseses_and_pests'

image_datasets = { x: datasets.ImageFolder(
                            os.path.join(data_dir, x), transformations[x
                                ]
                        ) for x in ['training', 'test'] }
dataloaders = { x: torch.utils.data.DataLoader(
                            image_datasets[x],
                            batch_size=4,
                            shuffle=True,
                            num_workers=4
                        ) for x in ['train', 'val']}
class_names = image_datasets['training'].classes

assert class_names == image_datasets['test'].classes
```

First, we need to define the transformations we're going to apply to the image set. We have also to make a transformation to `Tensor`. Tensors are the data structures with which PyTorch work. Each image will be represented with a tensor of *height* $\cdot$ *width* $\cdot$ *number of channels*.

After defining the transformations that will be applied to both training and test test set, we

[4]We can also split data explicitly.

have to defined the folders where images actually are. After that, finally a data-loader for each test and test set will be created.

It is important to explain how PyTorch works. In our dataset there are 544[5] images divide irregularly between 8 classes. The class with the lesser number of images has 16, and the one with the higher has 74. PyTorch, for each epoch, will create a random training set of 370 images. This images will mixed original images with some images created applying transformation defined in the `transformations` dictionary. That means that for a concrete image, in an epoch maybe a flip transformation will be applied, whereas in the next epoch that image may be transformed with a rotation, or again can be flipped, or even could be use without any change.

[5]Training set has approximate 70% of total images, with test set having 30% [29].

Chapter 5

Convolutional Neural Network

In this chapter, we are going to make a brief introduction about Neural Networks, and Convolutional Neural Networks. After that, the architecture and hyperparameter values choices we'll be explained.

5.1 Neural Network

5.1.1 Neurons

A Neural Network is the union of some layers, where each layer will have a number of **neurons**. A neuron is the smaller element of the network, which takes some inputs, and produces one unique value.

One of the first neuron models was the **perceptron**. This model takes one or more binary inputs and produces binary outputs (5.1).

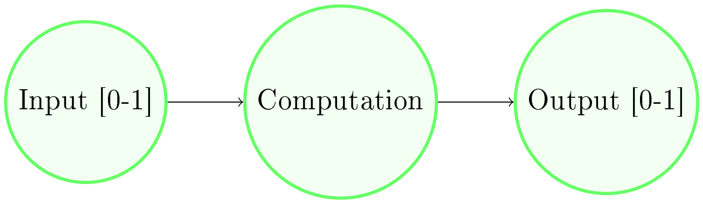


Figure 5.1: Perceptron with one input.

The output will be 0 or 1 depending on if the computed value is bigger or smaller that a threshold (5.1).

$$output = \begin{cases} 0 & \sum_j w_j \cdot x_j \leq threshold \\ 1 & \sum_j w_j \cdot x_j > threshold \end{cases} \tag{5.1}$$

The output is function of the perceptrons weights (w_j), so changing either the weights or the threshold, we'll have a different perceptron model.

5.1.2 Neuron Layers

We can have as many layers between input and output we want, and these layers will be called *hidden layers*. In 5.2 we can see a perceptron network with three layers. The first column of perceptrons will take very simple decisions, by weighing the evidence, that is, the inputs. The hidden layer will make the same but, instead of using the input external evidence, it will use the output of the first layer.

So it will be able to take more complex decisions that the input layer. Finally, the output layer, with one single perceptron, because it will use the output of the hidden layer, will take even more complex decisions. So, the more layers we stack, the more complex decisions the network will produce.

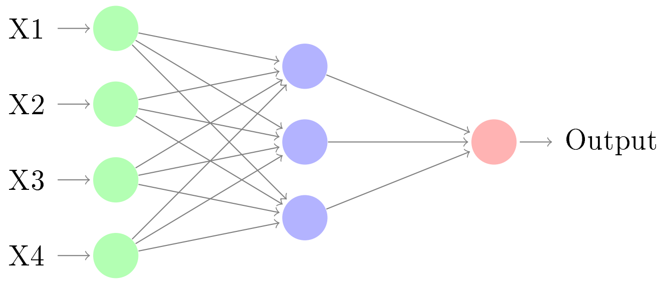


Figure 5.2: Neural Network with four inputs and three neurons in one single hidden layer.

If we have a bias, who will be equal to the negative value of the threshold, we will have then a different equations for the perceptron computation (5.2).

$$output = \begin{cases} 0 & \sum_j w_j \cdot x_j + b \leq 0 \\ 1 & \sum_j w_j \cdot x_j + b > 0 \end{cases} \tag{5.2}$$

We can see the bias as a measure of how easy is to get the perceptron to output 1, or similarly, one easy is to get the perceptron to fire[1].

So, if we find an algorithm that allows us to automatically change the weights and biases of the network in response to an stimuli, we'll theoretically will be able to adopt our network to more complex task. And, because Neural Networks are universal approximation machines, they are capable of handling any kind of problem we can think about, so we, in theory, will be able to use them to solve any problem.

5.1.3 Activation Functions

One problem of perceptron neurons is that, if some of the inputs changes a small value, the output of the network could sometimes to completely flip. That makes the use of this kind of neuron impractical.

[1]The higher the bias is, the easier the perceptron fill fire.

Therefore in practice other kind of neurons are used. Those neurons can produce a value inside a range, usually bigger than zero, and their inputs are also non-binary values. They are defined by the function that produces the output, and that function is called **activation function**. The input is z, defined as $z = w \cdot x + b$[2], and the output is the activated value, a. The most common functions are:

- *sigmoid function*, the most common used activation function in tutorials, books, etc, also called logistic function. Actually is very used in output layers, but it is not in hidden.

$$\sigma(z) = \frac{1}{1+e^{-z}} \tag{5.3}$$

- *linear function*, linear value with unranged values as output.

$$linear(z) = w \cdot z + b \tag{5.4}$$

- *ReLU function*, maybe the most used in papers and production. Very similar to linear, but it cannot give negative output values.

$$z = w \cdot z + b; a = max(0, z) \tag{5.5}$$

- *leaky ReLU*, theoretically better than ReLU, but it is not widely used. Instead of bounding the lesser value to 0, this function allows it to be smaller.

$$z = w \cdot z + b; a = max(c \cdot z, z);\ \ c \approx 0 \tag{5.6}$$

- *tanh function*, similar to sigmoid, it gives us values between -1 and 1. It is very used in hidden layers but not in output: because we want to classify there, it is useful having a function that gives us values between 0 and 1.

$$tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} \tag{5.7}$$

In this project, both *ReLU* and *tanh* have been used, with sigmoid function used in output layer.

[2]Actually is $z = \sum w_j \cdot x_j + b$, but we'll use the simple representation.

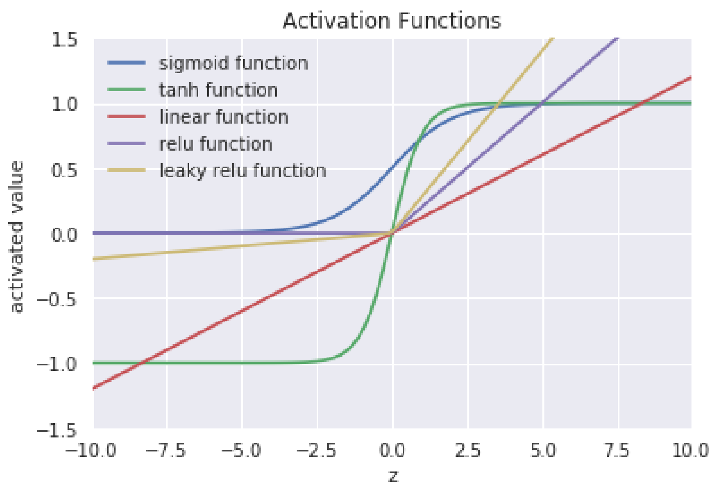


Figure 5.3: Output for different models of activation functions.

5.1.4 Backpropagation

As we said, we need an algorithm which let us find weights and biases. That search is that we call *learning* during the training process of a neural network. The variation of these values will make that, eventually, the output from out network approximates the value of the training inputs. The algorithm we'll use to learn is the *backpropagation algorithm*.

When we train a neural network, what we made is feeding the network with an input. After that, this input will generate an output applying in the hiding layers the equation we saw in 5.2[3]. The final output $\hat{y}$ will be compared with the expected output y.

The difference between those two values will be *backprogated* from the output layer to the input layer through all layers in the network. With this difference, the weights and biases of each hidden layer will be updated. With this update, we are trying to get a better network, that is, have a neural network that, in the next iteration, will make a better prediction[4]. This process will be repeated until the value of the error is lesser than a threshold or we have made a number of iterations equal to limit value we set.

This algorithm has a number of equations about we talk briefly in the next epigraphs.

Loss and Cost Function

We need to quantify the difference between our output ($\hat{y}$) and the reality that training values represents (y). This can be represented with loss functions. Among the different options that

[3]Needless to say adapted to a neural network that is not made of perceptrons, so instead of binary outputs we'll have as outputs values inside a range.

[4]A better prediction means that the difference $\hat{y} - y$ for an step i will be lesser that that difference for the step $i - 1$.

exist, in this thesis *Cross Entropy Loss* has been used (5.8).

$$\mathscr{L}(\hat{y}, y) = -(y \cdot log\hat{y} + (1-y) \cdot log(1-\hat{y})) \tag{5.8}$$

Then, we will compute the *Cost Function* (5.9). This cost function is the average of the loss function above for the entire dataset, and will actually the function we use when we try to improve our network predictions.

$$\mathcal{J}(w,b) = \frac{1}{m}\sum_{i=1}^{m}\mathscr{L}(\hat{y}^{(i)}, y^{(i)}) = \frac{1}{m}\sum_{i=1}^{m}\big[-(y \cdot log\hat{y} + (1-y) \cdot log(1-\hat{y}))\big] \tag{5.9}$$

Gradient Descent

Our objective will be to make $\mathcal{J} \approx 0$. To achieve that, Neural Networks use the *Gradient Descent Algorithm*. The way it works is to repeatedly compute the gradient ∇J, and then moving in the opposite direction.

After we have computed the gradient descent, then we can update the values of weights and biases (5.10), and applying this repeatedly, we will find the minimum of the cost functions, so $\hat{y}$ will be as near to y as it can, so we will be able to say that our network has learnt.

$$\begin{aligned} w_k \rightarrow w'_k &= w_k - \alpha\frac{\partial \mathcal{J}}{\partial w_k} \\ b_l \rightarrow b'_l &= b_l - \alpha\frac{\partial \mathcal{J}}{\partial b_l} \end{aligned} \tag{5.10}$$

With this updated values, we will compute again the new predictions with forward propagation. And with their results, we will again using backpropagation to update, for the next iteration, biases and weights. This process will be follow until:

- the error of the cost functions is lesser than a threshold; or
- the number of iterations we have computed is bigger than a fixed limit we have set.

Derivatives of activation functions

As we saw, we need to compute the partial derivatives of the cost function. This cost is function of the prediction, and this prediction will be also function of the activation functions. So at the end, we need to compute the derivative of the chosen activation function. This is critical, because this value will be computed thousands of times, so choosing a function that has a very simple derivative will increase the performance over other functions with more complex derivatives. Below we can see the derivatives of the activation functions explained in 5.1.3:

- *sigmoid function*
$$\sigma(z) = \frac{1}{1+e^{-z}} \rightarrow \sigma'(z) = a \cdot (1-a) \tag{5.11}$$
- *linear function*
$$linear(z) = w \cdot z + b \rightarrow linear'(z) = 1 \tag{5.12}$$
- *ReLU function*
$$z = w \cdot z + b; a = g(z) = max(0, z) \rightarrow g'(z) = \begin{cases} 0 & if\ z\ <\ 0 \\ 1 & if\ z\ \geq\ 0 \end{cases} \tag{5.13}$$
- *leaky ReLU*
$$z = w \cdot z + b; a = max(c \cdot z, z);\ c \approx 0 \rightarrow g'(z) = \begin{cases} c & if\ z\ <\ 0 \\ 1 & if\ z\ \geq\ 0 \end{cases} \tag{5.14}$$
- *tanh function*
$$tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} \rightarrow tanh'(z) = 1 - a^2 \tag{5.15}$$

Backpropagation algorithm in Python using Numpy We'll see next a version of this algorithm using python with numpy. This code is an adaptation of an original piece code that we can find in [30].

```python
class NeuralNetwork(object):
    def __init__(self, weights, biases, activation_fun=sigmoid):
        self.weights = weights
        self.biases = biases
        self.activation_function = activation_fun
        # 'derivatives' is a dictionary with the derivatives values for
            each
        # 'activation_function' allowed.
        self.activation_function_derivative = derivatives[activation_fun]
        ...

    def backprop(self, x, y):
        # First we have to initialize values
        delta_weights = [np.zeros(w.shape) for w in self.weights]
        delta_biases  = [np.zeros(b.shape) for b in self.biases]

        ## FEEDFORWARD STEP
        # a_0 == x
        activation, activations, zs = x, [x], []
        # In each step of the iteration, we compute the activation value
        # for the next hidden layer, until we arrive to the output layer,
        # that is, the final values of 'self.biases' and 'self.weights'
        for b, w in zip(self.biases, self.weights):
            z = w @ activation + b
            # We use 'zs' for caching results
            zs.append(z)
            activation = self.activation_function(activation)
            activations.append(activation)

        ## BACKPROPAGATION STEP
        # We start from the last (activations[-1]) to the firs layer
        delta = self.cost_derivative(activations[-1], y) * \
                self.activation_function_derivative(zs[-1])
        delta_biases[-1]  = delta
        delta_weights[-1] = delta @ activations[-1].T
        for l in range(self.num_layers-1, 1, -1):
            z = zs[l]
            derivative = self.activation_function_derivatives(z)
            delta = (self.weights[l] @ delta) * derivative
            delta_biases[l]  = delta
            delta_weights[l] = delta @ activations[l].T

        return delta_weights, delta_biases
```

PyTorch backpropagation computation

As we saw, the computation of backpropagation algorithm is both a complex and crucial step. Fortunately, deep learning libraries like PyTorch allow to compute it with few lines of code:

```
import torch
import torch.nn as nn
import torch.optim as optim
...
optimizer = optim.SGD(model.parameters(), lr=0.001, momentum=0.9)
...
output         = model(input)
_, predictions = torch.max(output, 1)
loss           = nn.CrossEntropyLoss(output, labels)
loss.backward()
optimizer.step()
```

In the code above we can see that we only need to call the `backward` function for the loss model we chose, function that computes the gradient for every parameter we want to apply gradient descent.

5.1.5 Optimization

Below we are going to describe the most common optimization algorithms we can find. Why is important to optimized our computations? Because if we have thousands or even millions of images, the backpropagation step will be very slow, because we won't update the weights and biases until we have computed gradient descent for the whole data set.

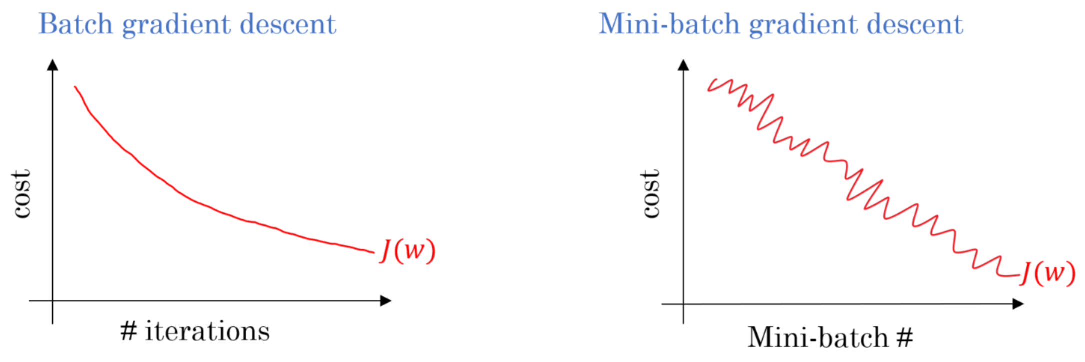


Figure 5.4: Evolution of cost function: batch vs minibatch GD, from Imad Dabbura.

Mini-batch Gradient Descent and Stochastic Gradient Descent

Instead of waiting until all computations have finished, we can update when a small number of images has been used to *learn*. We can split the whole data set in small pieces, which we call *mini-batches*. And, instead of computing gradient descent for the full data set, we can compute it, at each iteration, for a mini-batch, but, the most important, we will update all weights and biases with the values we get using only one batch at each step (also called *epoch*). If we use only, instead of m images for each mini-batch, only one image, we have the *Stochastic Gradient Descent*. This is the one that we have been used in this thesis, as can be seen at the previous line of code `optimizer = optim.SGC(...)`. If m is equal to the size of the full data set, we have the same version of gradient descent as we didn't use any optimization, call *Batch GD*.

The SGD has an obvious disadvantage. It doesn't converge to a global minimum. Instead of that, the cost variation over time has a lot of noise, but the average is also decreasing, and this joined with the huge performance improvement made that SGD or Mini-batch GD are both used instead of the regular version of GD.

Other optimization algorithms

There are several optimizations we can use trying to increase the performance without decreasing a lot the accuracy of our network. Whereas there are a lot of different optimization algorithms [36], the most common and cited in papers are maybe *GD with Momentum*, *RMSProp* and *Adam*[5].

5.1.6 Hyperparameters

In NN there are parameters that can be learnt, but there are also a lot of parameters that we have to chose in order to run the model. We have the learning rate α, the number of layers, number of hidden units, the mini-batch size, etc. We can try randomly with different combinations of

[5]The optimization algorithm we have used is actually a variation of SGD where we use also momentum to improve the performance of the computations.

these, but this could be a huge work. Actually we try with combinations of pairs or triplets of hyperparameters.

In this thesis, we have some of them fixed. As we will see, we have trained a resnet-50 architecture, so the number of layers, hidden units, size of the image, etc, are in fact fixed by this model. So among the rest, we have two that are critical: number of epochs and learning rate.

- *number of epochs*: we can see at 5.8 that after ten epochs, our accuracy is not improving a lot. We could have chose 15 epochs, but because of the small size of the data set, 25 was chosen trying for security; with that value, the model expend approximately 30 minutes to be trained; and
- *learning rate*: instead of fixing the value of learning rate, we can create an *scheduler* that will change it trying to improve the results. That is a feature of PyTorch, and it is very easy to configure:

```
    from torch.optim import lr_scheduler
    ...
    optimizer = optim.SGD(model.parameters(), lr=0.001, momentum=0.9)
    scheduler = lr_scheduler.StepLR(optimizer, step_size=5, gamma=0.1
```

  where *step_size* is the period of learning rate decay, the number of iterations after we apply the gamma value, and the *gamma* is the multiplicative factor of learning rate decay that actually decrease its value.

5.2 Convolutional Neural Networks

We have described regular neural networks. But in this thesis, we are going to use a Convolutional Neural Network (CNN). They are very similar to regular NN, but very focused in Computer Vision tasks. CNN "[...] make the explicit assumption that the input are images, which allows us to encode certain properties into the architecture." [48, CNN for Visual Recognition]. Because of this specific characteristics, they have a better throughput thanks to the huge decrease of the number of parameters of the network.

Instead of connecting all the elements between each layer as an a regular network[6], we have small matrices or kernels, with sizes like of $3x3$, $9x9$, etc, that, applied over a layer, give us the next layer. This process is the *convolution*. This convolution is the responsible of the fewer parameters.

The way the kernels (or filters) are applied can be seen at 5.5. We can see how the application is, at each step that produces a pixel output, over a local region. This is actually the cause of parameters reductions when we compare CNN over FCN, and it has sense because, if we think the way we see, we make relations among near *pixels* or stuffs that are in an image, that usually will have more relation that pixels at the edges of the image.

[6]Because of this, we can call regular networks as Fully-Connected Networks (FCN) as opposition with CNN.

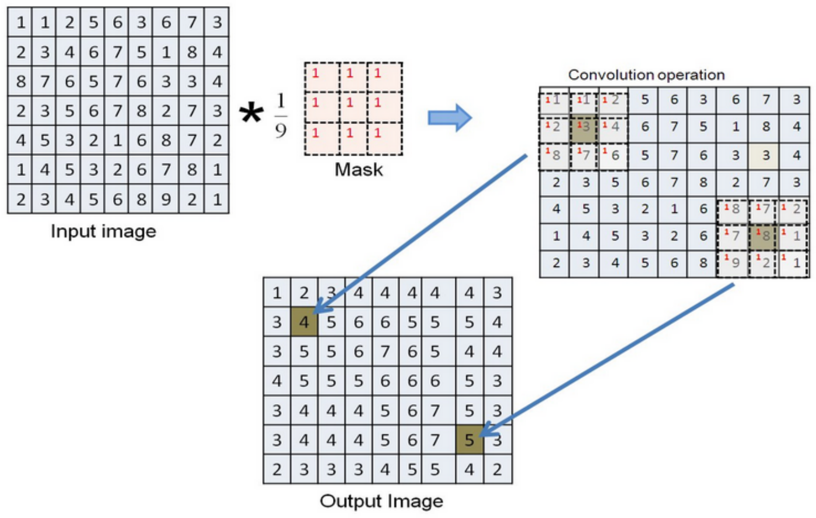


Figure 5.5: Application of a kernel over an image from the Government of India. We can see how the kernel is applied over neighboring pixels.

5.2.1 Kernels

As we work with RGB images, that actually are tensors with a depth of three, we need filters that are tensors too. And their depth must be equal to the depth of the layer they are working with. But, if we multiplied the kernel over the image, we will get a two-dimensional matrix as we can see at 5.6, so in only one step we would have been finished our network. A CNN actually works stacking the results of applying several kernels over each layer, so the result of a convolution is another tensor. At 5.7 we can see how, stacking the result of two convolutions, we have a two-channels[7] tensor as result.

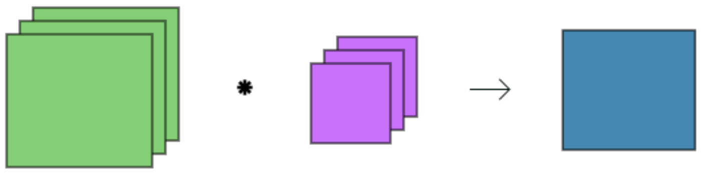


Figure 5.6: Convolution of a single kernel over an RGB image.

[7]Because in CNN we usually work with images, and in images the depth of the tensor, where the tensor is the image, is the number of channels we have, we can talk about depth or number of channels indistinctly.

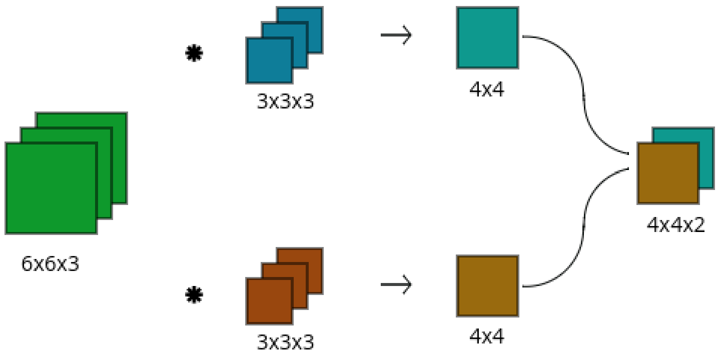


Figure 5.7: Two kernels convolutions.

These kernels play a similar role than the weights matrix in a regular neural network. Therefore, when we train a CNN, we actually will modified the values inside the filters trying to improve the difference between predictions and reality, as we made in NN.

One obvious advantage is that, no matter the size of the input, the number of parameters is a function of the kernel dimensions and the number of filters as we can see in the next example.

Number of parameters computation As an example we can think in an image of $32x32$ pixels in the RGB space color. If the second layer has $28x28x6$ neurons, the total number of parameters will be $32x32x3x28x28x6 \approx 14e^6$.

But if we have a Convolutional Layer with 6 filters of $5x5x3$, we'll have only $(5 \cdot 5+1) \cdot 6 = 156$ parameters[a], and will have as an output also a $28x28x6$ neuron layer. As we saw, the number of parameters only depends on the dimensions of the kernels and their number.

[a]The 1 added is because of the bias.

5.2.2 Padding and Stridded Convolutions

Padding

Every time we applied a filter over a tensor, its size is reduced. If we continue applying kernels, eventually we'll have a very small size tensor. The *padding* is used to avoid this, creating extra columns and rows that will make that the output will have same weight and height than the original input before padding.

If we do not do that, when we apply a kernel with fxf dimensions over an nxn image, we'll have an output of $(n-f+1)x(n-f+1)$. After applying a padding p, then $(n+p)x(n+p)*(fxf) = (n+2p-f+1)x(n+2p-f+1)$.

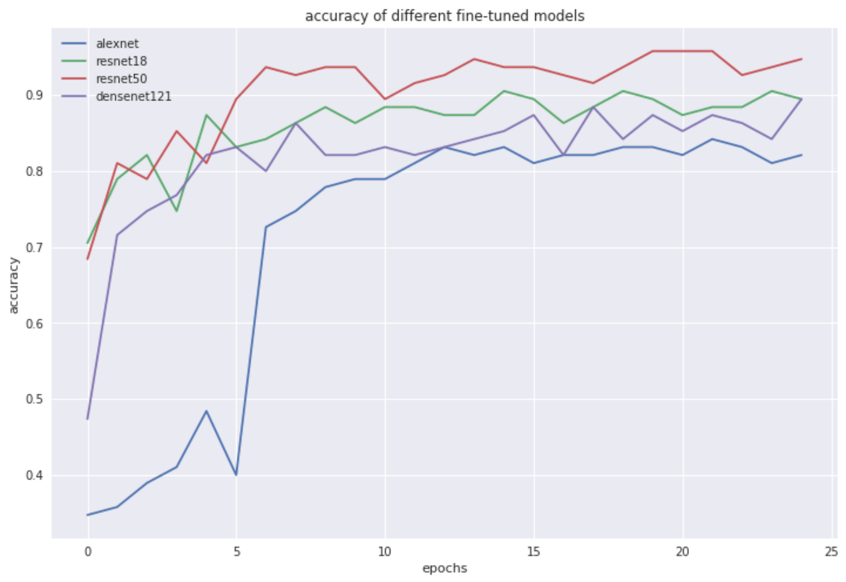


Figure 5.8: Accuracy of different pretrained CNN architectures with citrus diseases dataset.

Stridded convolution

We can move the kernel more than one pixel at each step. If we do that, the size reduction will be bigger. This technique is use also in *Pooling Layers* that we'll explain in the next epigraph.

5.2.3 Layers used in CNN

In CNN, we don't use only Convolutional Layers. Although this thesis' objectives isn't give a full explanation about the theory behind CNN, we are going to give a small explanation about the rest of the layers we'll find in CNN and, actually, in the architecture used in this thesis.

- *RELU layers*: they give as an output a same-size tensor;
- *Fully-connected layers*: usually at the end of CNN, they give us the final classification; and
- *POOL layers*: used to make reduction of the size of the tensors.

5.3 Architectures

As architecture we are talking about how the network is: its layers, how many neurons has each of them, which type, etc. In this thesis, some of the most famous architectures has been used: *alexnet* [10], *resnet* [16] and *densenet* [19]. Some others has been used, like *lenet* [24] or *inception* [43], but because of his simplicity or due to hardware limitations, they haven't been tested.

Between the three that actually were used, we can see their results in 5.8[8]

[8]In this picture we can see that two different versions of *resnet* were used.

Explained them, even represent how they work, is a complex and long task, that exceeds this thesis. Nevertheless, we are going to explain briefly the main features of the architecture used in the thesis, *resnet-50*.

Resnet-50

When we work with very deep networks, we could guess that as more layers we use, a better accuracy we'll achieve. But the reality is different, and there is some number of layers from which the network, instead of getting improved, becomes worse. As we can see at 5.9, there is no benefit stacking more layers.

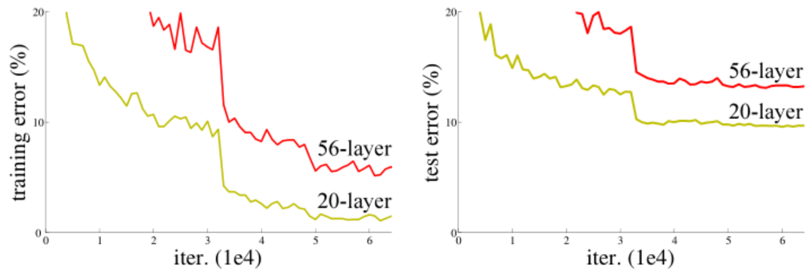


Figure 5.9: Training error for CIFAR-10 image set using non-residual networks, from [16].

One solution to this problem, the solution that residual networks uses, is to use *residual blocks*: we add by-passes that feeds some layers with their own inputs plus the input that comes from previous layers 5.10.

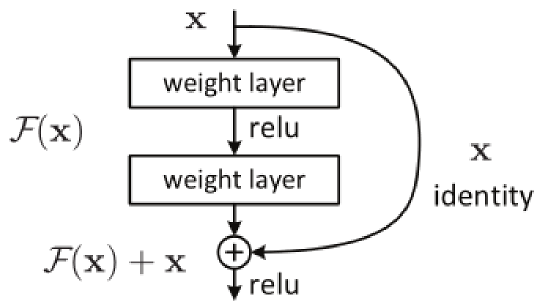

x
weight layer
$\mathcal{F}(\mathbf{x})$
relu
weight layer
x
identity
$\mathcal{F}(\mathbf{x}) + \mathbf{x}$
relu

Figure 5.10: Basic residual block, from [16].

A residual network will be made stacking many of those blocks, and using them will see that deeper network will have if not a better accuracy, it won't be worst than shallower versions.

Chapter 6

Android Application

6.1 Android Development and Framework Basics

The Android operating system is a multi-user Linux system. By default, the system assigns each app a unique Linux user ID[1]. The system sets permissions for all the files in an app so that only the user ID assigned to that app can access them.

Every app runs in its own Linux process, and each of them has its own Virtual Machine, so an app's code runs in isolation from other apps. This processes are started by the Android system when any of the app's components need to be executed, and then shuts down the process when it's no longer needed or when the system must recover memory for other apps.

Android apps can be written using Kotlin, Java, and C++ languages. As we saw in chapter 2, the application developed in this thesis has been written in Kotlin, a relatively new programming language with a complete interoperability with Java code.

The Android SDK tools compile the code along with any data and resource files into an APK file[2], an Android package which is an archive file with an `.apk` suffix. One APK file contains all the contents of an Android app and is the file that Android-powered devices use to install the app.

And Android application can have different components, but some of them are mandatory and has been used in this thesis application:

- *Android manifest file*: the primary task of the manifest is to inform the system about the app's components. Before the Android system can start an app component, the system loads this file to know which components exist. Here, we'll declare the entry point of the app, the permissions, the API level we are compile for, etc.
- *Activity*: they represent a single screen with a user interface. One of this must be the entry point of the application. Each activity is independent of the others.
- *Fragments*: components that allow to develop more flexible UI than these that can be

[1]The ID is used only by the system and is unknown to the app.
[2]In the last Google I/O was presented the new aab format.

developed only with activities, they represent a behavior or a portion[3] of the UI. A fragment must always be hosted in an activity and the fragment's life-cycle is directly affected by the host activity's life-cycle.

- *Xml files*: the UI is defined in *xml* files, which will be translated in the `onCreate` method in the Activity or Fragment that *inflates* them.
- *Resources*: Android apps can used a very large type of resource files. In the app developed with this thesis, `drawable` files[4] and the neural network models themselves[5] are resources.

There could be other components, such as Content Providers, Services, Loaders, etc, which haven't been used in this application, at least at the release that accompany this thesis.

6.2 Application Architecture

One of the biggest problem we will deal when we are developing an Android application is the architecture we'll use with the app. The Android framework uses a variation of the MVC, where `xml` files are the views, the activities and/or fragments are the controllers, and the models are the rest of classes or services[6] that we'll use for retrieve and processing the data and the user actions.

But when we develop a medium or large size application, this approach isn't the most convenient. The controllers became huge classes with even thousands of lines of code. This classes are very hard to test, improve, etc.

Because of that, last years some architectural patterns has been used to develop better apps, at least apps that make the processes of debug, test or improve easier. These architectures are Model-View-Presenter (MVP), Model-View-View-Model (MVVW) and Model-View-Intent (MVI).

The application has been developed with MVP. But the three could be used. The MVVM is in fact the way Google try the developers code their applications, with some additions to the Android framework that makes easier to use it. The MVI is maybe the most recent pattern, and it is yet less used than the others in the industry.

6.2.1 Model View Presenter

The chosen architectural pattern, MVP, is currently the most widely used in Android development. It is conceptually the easiest. We use MVP to improve the *separation of concerns* in business logic[7]. With this pattern, we split the application in:

- *views*: not only the `xml` files but also the activities and fragments;
- *presenters*: there will be classes that will get the user actions and will call the use cases, and after receiving data from those, they will update the UI; and

[3]Especially when we are designing for devices with larger screens like tablets.
[4]Images such as `png` or `jpg`.
[5]Files with `lite` extension produced by the `toco` utility.
[6]Databases, REST services, etc.
[7]Separation of concerns "[...]is a design principle for separating a computer program into distinct sections"

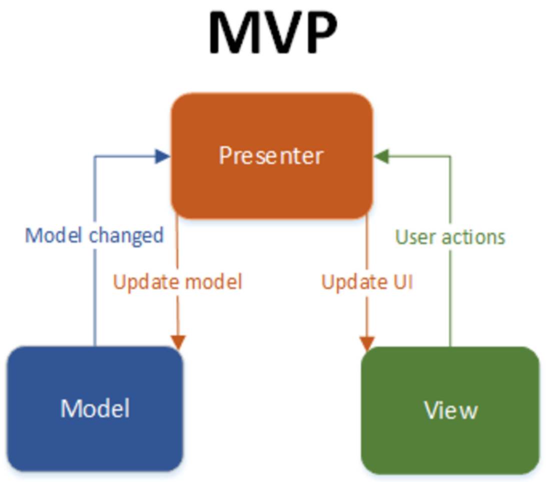


Figure 6.1: MVP with interactions between its parts, from Techyourdance.com.

- *model*: classes (objects) where will store the app state, that will be sent to the presenters after UI requests.

We'll have also `UseCase` classes[8]. With them, we'll make the request to the model from the presenters. We could have use cases for retrieving data from a database, or from a web service, to get some knowledge about the system, etc.

6.3 Class Diagrams

Because the size of the class diagram for the full application, we are going to split it in various parts. First, at 6.2 we can see the classes that represents all the UI, either activities or fragments, with the interfaces that they implement.

The presenters, model and the use cases (business logic) that take the user inputs, make computations and retrieve the info to the user updating the screen can be shown at 6.3.

6.4 Libraries

A small set of libraries have been used in this application. Almost all of them are Android itself libraries that aren't include in the framework from scratch, or Kotlin extensions which make easier the development. In the next sections we are going to give an explanation about their features and why they were chosen instead of other options:

[8]They are also call `Interactor` in the literature.

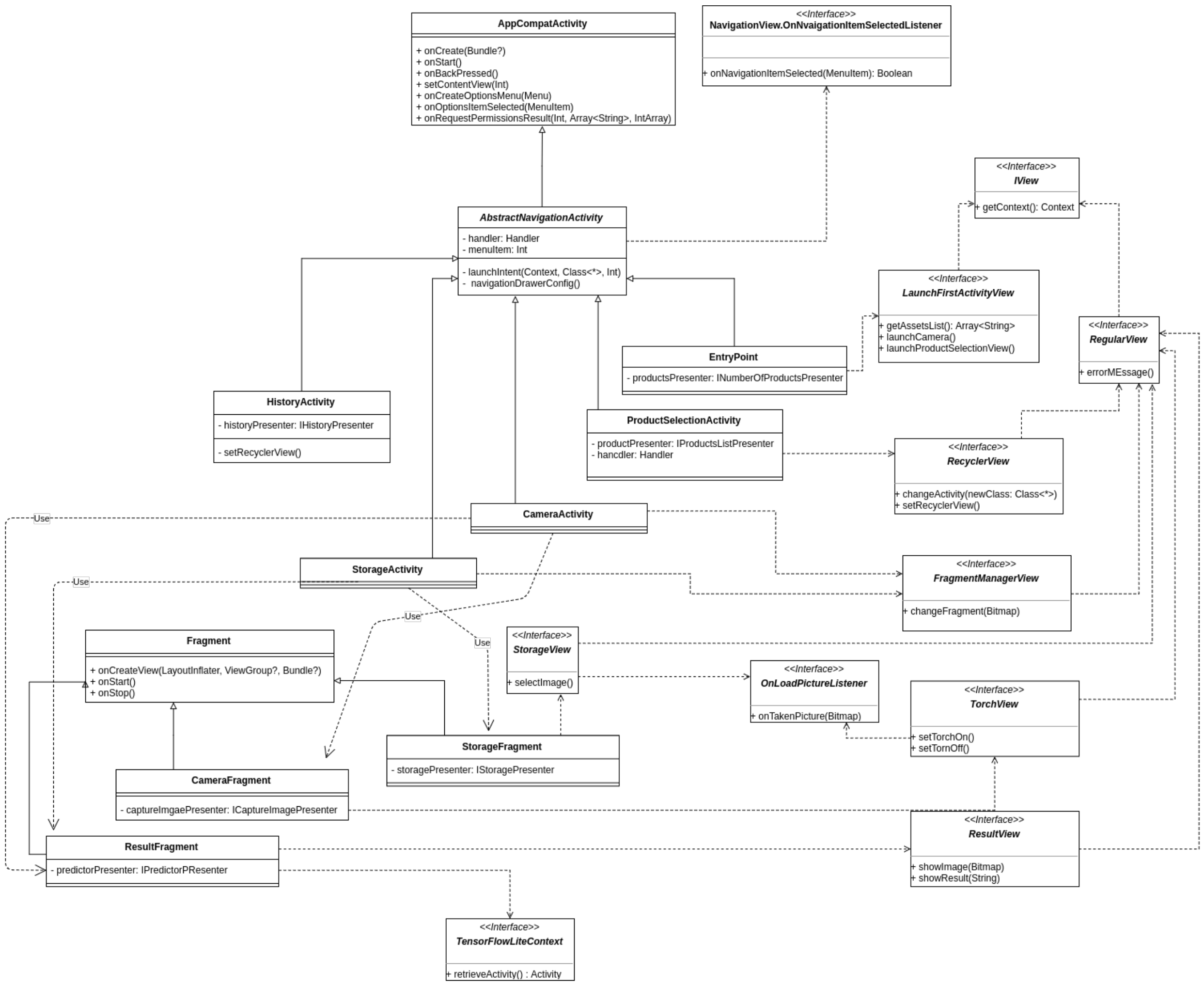


Figure 6.2: Class diagram with classes that show elements on the screen and with which the users interact.

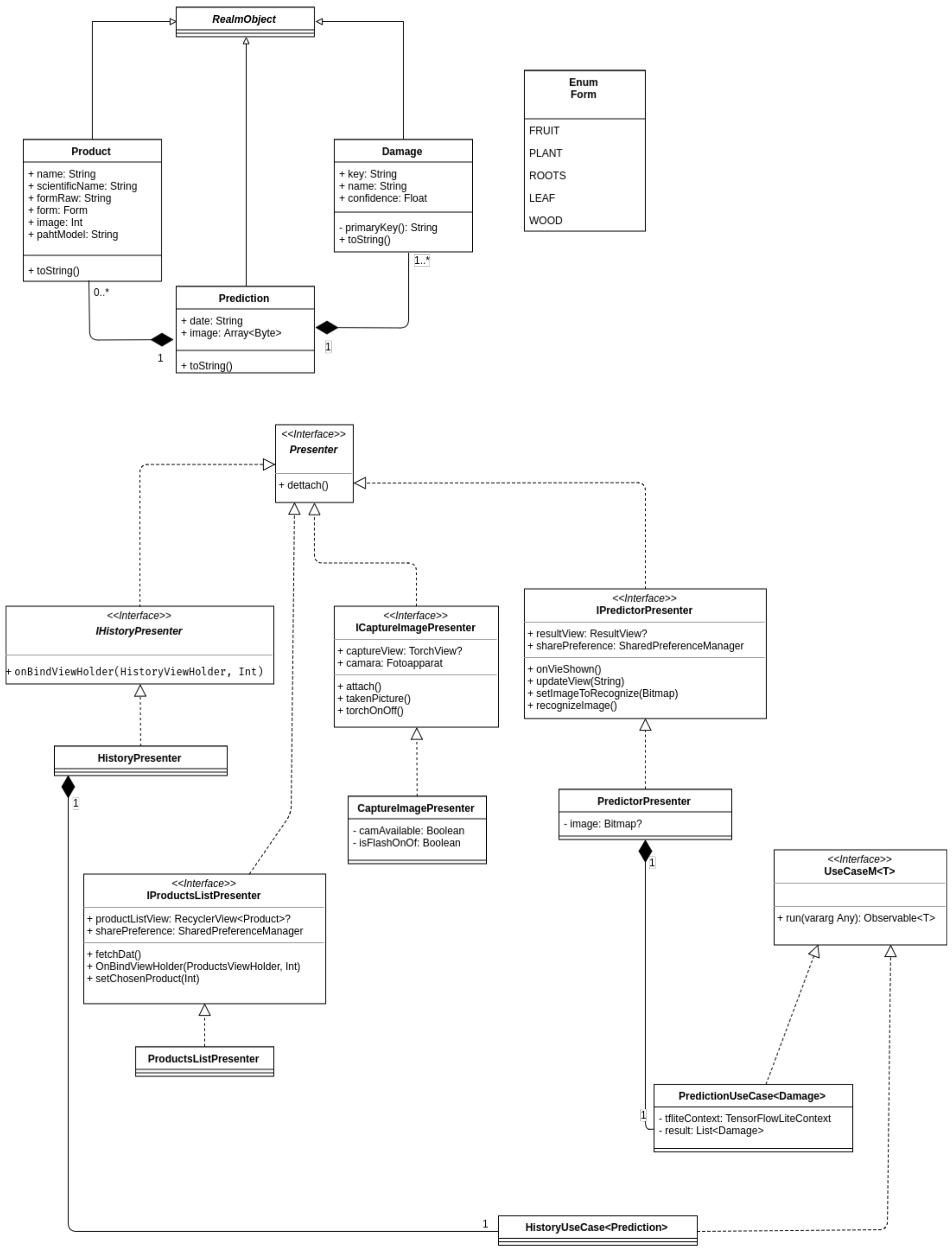


Figure 6.3: Class diagram with classes that represent presenters, model and use cases.

AppCompat-v7 Mostly used in every Android app, it allows to use Fragments[9]. Also, we are going to use `RecyclerView` as a container for the different products we will be able to recognize with the app in the future. This feature is also added with this dependency.

CardView-v7 The items in the `RecyclerView` can be designed from scratch. But a very good alternative is the use of `CardView`. We must also provided the layout for it, but this look and behavior fits nice in an app with a very specific and professional market like this one.

Fotoapparat Developing a camera view with the tools that the Android SDK provides is not actually a trivial job. Fortunately, there are a couple of open-sources options that works really well. Between them[10] the Fotoapparat library has been used. It provides a very easy to use view, that allows to use both cameras, the flash, zoom, etc.

We can instantiate in a very simple way:

```
    val fotoapparat = Fotoapparat(
            context   = context!!,
            view      = camera_view,
            scaleType = ScaleType.CenterCrop
        )
```

where `camera_view` is defined in a layout with the widget `io.fotoapparat.view.CameraView`.

We can also modified the features during the runtime. In the app, we can turn on/off the torch with a simple line of code:

```
    fotoapparat.updateConfiguration(UpdateConfiguration(flashMode=torch())
        )
```

And finally, taking a picture is as simple as writing:

```
    val picture: PhotoResult = fotoapparat.takePicture()
    picture.toBitmap()
            .whenAvailable { bitmapPhoto ->
                bitmapPhoto?.let {
                    callback.onTakenPicture(bitmapPhoto.bitmap)
                }
            }
```

In the code above, we can see how the library itself manage the fact that taking a picture and transforming it is an expensive operation, so it provides the method `whenAvailable` that makes an asynchronous call and wait until the photo is made in a different thread than the UI thread to return it when the result is finally available[11].

Constraint Layout A relatively new layout in the Android framework, the constraint layout allows a faster and easier way to develop nice and flexible UI's than other layout like `FrameLayout` or `RelativeLayout`, even in a graphical way, with points-and-clicks.

[9]The Fragment in the main library is currently deprecated and its use is not advisable.

[10]CameraView or CameraKit were other alternatives, but we can find more in android arsenal.

[11]Because this fact is so important in Android development, we can read more about it and the solutions we took in the app development in the appendix A.

Picasso A library that allows image management from different sources: drawables, urls, etc. Like *Fotoapparat*, it put the hardest work in background threads. We use it for loading images in the items of the *RecyclerViews*:

```
Picasso.get()
       .load(it.image)
       .resize(500, 500)
       .centerCrop()
       .into(itemView.product_image)
```

Kotlin utilities Here we include the Kotlin extensions and the Anko library. The extensions allow to avoid the use of the `findByView` method, and all the casting operations related to it. Hence, we can substitute this:

```
val button   = findByView(R.id.button_one) as Button
val textView = findByView(R.id.text_view) as TextView

button.text   = "Button"
textView.text = "Text"
```

By:

```
button_one.text = "Button"
text_view.text  = "Text"
```

On the other hand, *Anko* provide very helpful methods for Android development, like co-routines for asynchronous job.

Realm There are many options to add a database to an Android application. We can work at a low level and use *sqlite* directly, even when this is a not recommended practice according to Google. We can also work with *sqlite* in an easy manner with Room or some others third-party libraries that exists in the market.

But there are some other non-sql options, with Realm and ObjectBox being the most used. In this application Realm has been the chosen one[12]. The main reasons to do that is the easy way we can create the database, insert values, modify them, etc.

In 6.6 there is an explanation about how this data base works, how we can create the model, establish relationships between entities and make queries and updates.

TensorFlow Lite Finally, the main library in this app, https://www.tensorflow.org/mobile/tflite/ is a lightweight solution for mobile devices. It only needs to import itself as a dependency[13]. The use is also easy: we need to implement the `Interpreter` class, load the model and labels, and call the `run` method passing it the image and the array where the results will be stored in-place:

[12]We can see here a very good comparison among these options, or even a more complete (without ObjectBox) can be found also in this Github repository.

[13]We can compare this with DL4J, where we need to import the library, ND4J, OpenCV, etc, or with Caffe2, with also many dependencies for Android development.

```
  val interpreter = Interpreter(model)
  interpreter.run(byteBuffer, floatArray)
```

6.5 User Interface

In Android, we have activities and fragments to represent the screen. For the navigation among different options, we can use a `BottomBar Navigation`, or a `DrawerLayout` with a `NavigationView`. We have selected the second one because for the navigation in this applications it seems more natural, but they are almost exchangeable, and very few changes will be necessary to use the other.

Usually in `NavigationView` fragments are used to move between different screens associated with different buttons in the menu. But, because we have some of this with a secondary result, we have chose a different approach, using activities instead of fragments, and sing only those when we'll have an screen change as a result of a user interaction with the first activity. This fact, the use of activities, has made that a small change was needed in order to achieve a smooth navigation view hidden. For this, we have used the next snippet:

```
var handler: Handler?
...
handler.postDelayed( {
  val intent = Intent(ctx, newClass)
  intent.putExtra(ITEM_ID, itemId)
  startActivity(intent)
  finish()
}, Constants.NAVIGATION_VIEW_LAUNCH_DELAY)
```

With that code, we launch the intent that will change between activities after constant time, giving the option to the system to first hide the `NavigationView`.

At 6.4 there is a figure with all the screens we have in the application, and which transitions among them are allowed.

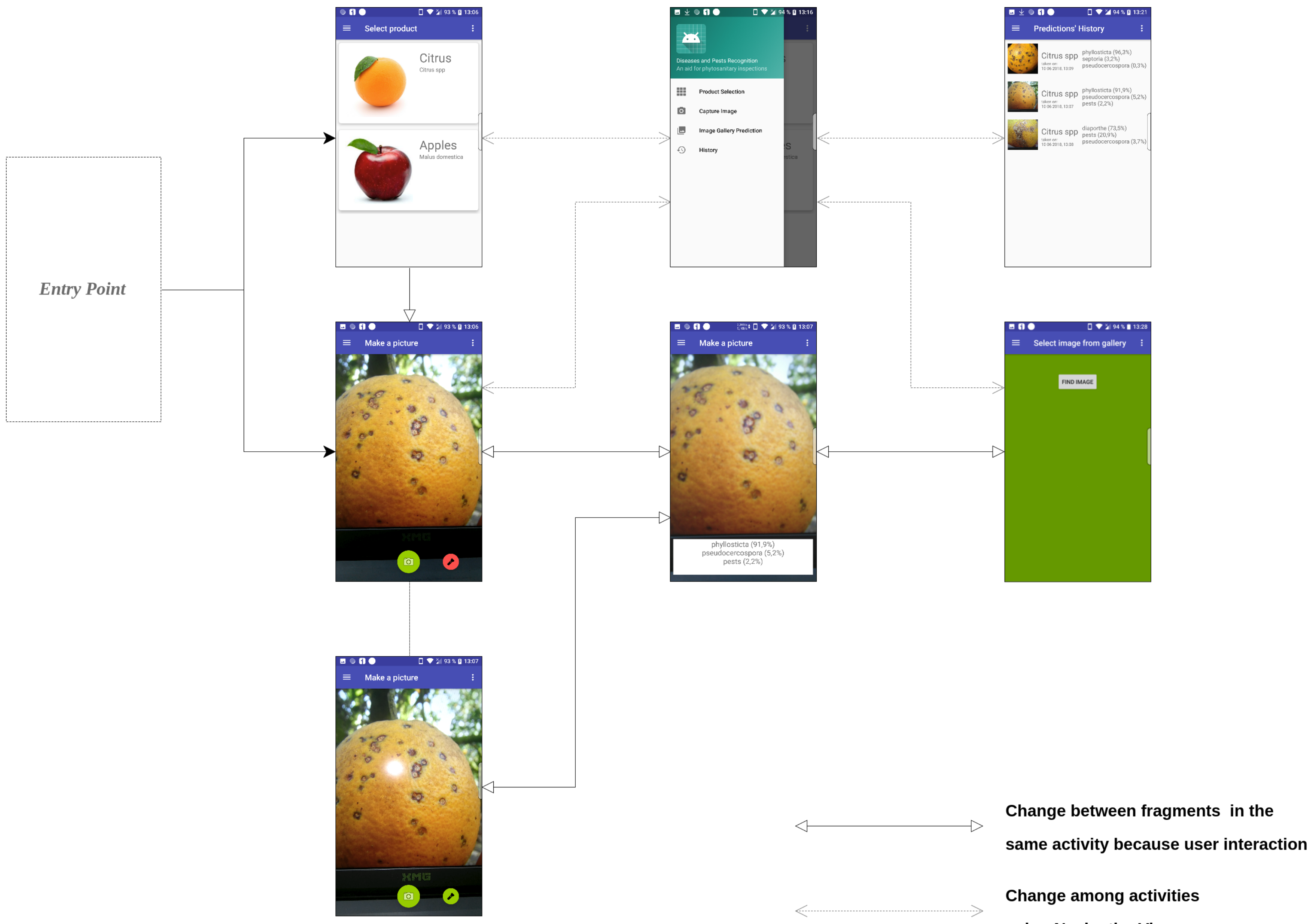


Figure 6.4: Screens and navigation in the app.

6.6 Data Base

First of all, we need to initialize the data base in a class that inherits from the `Application` class:

```
class App : Application() {
    override fun onCreate() {
        super.onCreate()
        Realm.init(this)
    }
}
```

After that, we can now get an instance in whichever point in the code. The only we have to do is getting a default instance[14] with `Realm.getDefaultInstance()`.

With an instance we can now make two things: start a transaction or make a query. For transactions, we have to start and commit it, but for reading we do not need to begin-and-commit:

```
  val realm = Realm.getDefaultInstance()
  realm.beginTransaction()
  \\ Creation, update or delete operations
  realm.commitTransaction()
  realm.close()
```

But kotlin features can simplify this process a lot. Instead of that, we can simply pass a lambda to and write:

```
  Realm.getDefaultInstance().use { realm ->
    realm.executeTransaction {
      for (product in Constants.products) realm.copyToRealm(product)
    }
  }
```

It is very important the thread where we use a realm instance. They cannot be shared between threads, so if we launch different queries in different threads, then we will need to use different instances.

Because realm instances implements closable interfaces, after finishing the use the `close()` method is called.

In the previous code we saw that we copied an object to the database with the `copyToRealm`. This objects cannot be regular POJO objects, and they must inherit `RealmObject` class.

In this project we have used three different classes:

```
open class Product(@PrimaryKey var name: String = "",
                              var scientificName: String = "",
                   var formRaw: String = "",
                   var image: Int = 0,
```

[14]Custom instances, which different names and features can be created, but in this application that was not necessary.

```
                   var pathModel: String = ""): RealmObject() {
    var form: Form?
        get() { return Form.valueOf(formRaw) }
        set(value) { Form.valueOf(formRaw) }

    override fun toString(): String {
        return StringBuilder()
                .append(name)
                .append("␣-␣$scientificName")
                .toString()
    }
}

enum class Form { FRUIT, PLANT, ROOTS, WOOD, LEAF }

open class Damage() : RealmObject() {
    @PrimaryKey var key: String = ""
    var name: String = ""
    var confidence: Float = -1f

    constructor(name: String, confidence: Float): this() {
        this.name = name
        this.confidence = confidence
        this.key = primaryKey()
    }

    private fun primaryKey(): String {
        val df = SimpleDateFormat("dd␣MM␣yyyy,␣HH:mm")
        val date = df.format(Calendar.getInstance().time)

        return "$date␣-␣$name␣-␣${String.format("(%.1f%%) ",␣confidence␣*␣
            100)}"
    }

    override fun toString(): String {
        return StringBuilder()
                .append("$name␣")
                .append(String.format("(%.1f%%)␣", confidence * 100))
                .trim { it <= '␣' }
                .toString()
    }
}

open class Prediction() : RealmObject() {
    var date: String = ""
    var product: Product? = null
    var damages: RealmList<Damage> = RealmList()
    var image: ByteArray? = null

    constructor(product: Product, image: ByteArray) : this() {
        val df = SimpleDateFormat("dd␣MM␣yyyy,␣HH:mm")
        this.date = df.format(Calendar.getInstance().time)
        this.product = product
        this.image = image
    }

    override fun toString(): String {
```

```kotlin
        return StringBuilder()
                .append("$date - ")
                .append("$product - ")
                .append(damages)
                .toString()
    }
}
```

We can see that we don't have getters and setters, only for the `Product.form: Form` field. It could seem that we are going to access directly to the fields, a non-recommended technique in Java. But this is not true, because the Kotlin compiler translates this code to a classical pattern where for each field we'll have a private field and both a getter and a setter.

Realm imposes some limitations to the use of kotlin's features. Thus, we cannot use data classes, and we must make all classes open. Also, we have to declare all variables as mutable, and we have to provided them with default values.

But, in spite of these limitations, the creation of the Realm entities is very easy. We can see also how we can create the relationships:

- *one-to-one*: we only need to have a filed in a class with the same type of the Realm entity[15], as we can see in `Prediction`, which has a `var product: Product? = null` field.
- *one-to-many*: in the same `Prediction` class we can see this relation, in the `var damages: RealmList<Damage> = RealmList()` field.

Retrieving data from the database is also very easy. The only thing we have to do is filter the results of the class that we want. Although in this app very simple queries were needed, we can make them more complex:

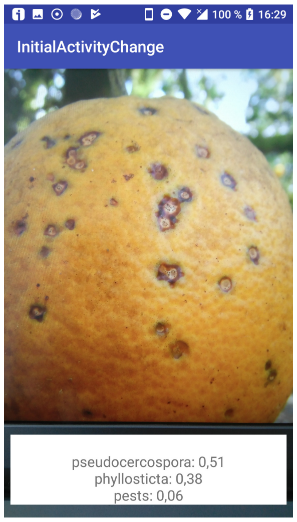
```kotlin
val realm = Realm.getDefaultInstance()
val citrusDamagesMoreThanTen = realm.where(Prediction::class.java)
                    .findAllAsync()
                    .filter{ prediction -> prediction.product.name = "
                        Citrus" }
                    .flatMap { prediction -> prediction.damages }
                    .filter { damage -> damage.confidence > 0.1f }
```

In the previous code, we have got all the predictions stored in the data base in an async way, and after that we have create a list of all citrus predictions, damages with a confidence bigger than ten percent made on citrus.

6.7 Results

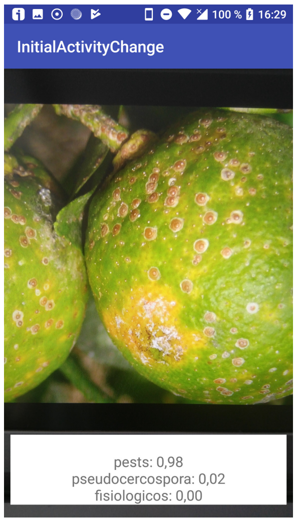
Finally, we can see in the next pictures some results using the app, from images taken with the camera but also using images from the cellphone gallery.

[15]That is, classes inheriting from `RealmObject`.



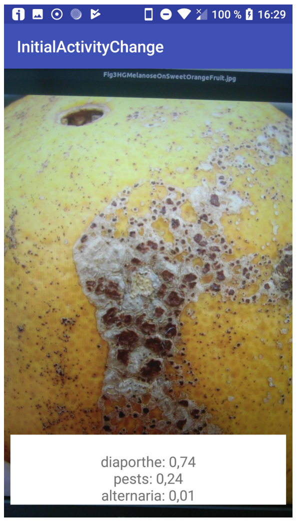
pseudocercospora: 0,51
phyllosticta: 0,38
pests: 0,06

Figure 6.5: Failed, the good one is the second choice, *Phyllosticta*. Image from the validation set.



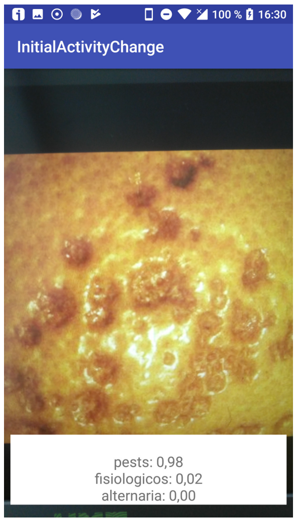
pests: 0,98
pseudocercospora: 0,02
fisiologicos: 0,00

Figure 6.6: Good, pest, *Aonidiella aurantii*, image neither from the validation nor test set.



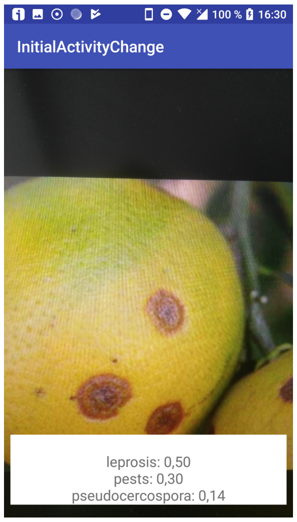
diaporthe: 0,74
pests: 0,24
alternaria: 0,01

Figure 6.7: Good prediction, melanosis disease produced by *Diaporthe citri.*



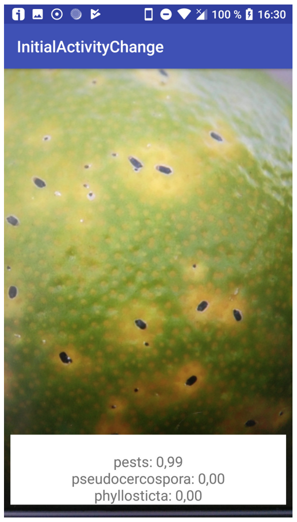
pests: 0,98
fisiologicos: 0,02
alternaria: 0,00

Figure 6.8: Failed, damaged produced by low temperatures.



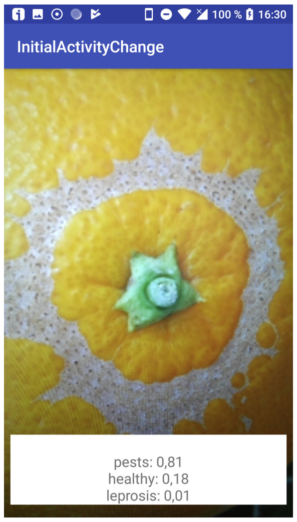
leprosis: 0,50
pests: 0,30
pseudocercospora: 0,14

Figure 6.9: Right prediction.



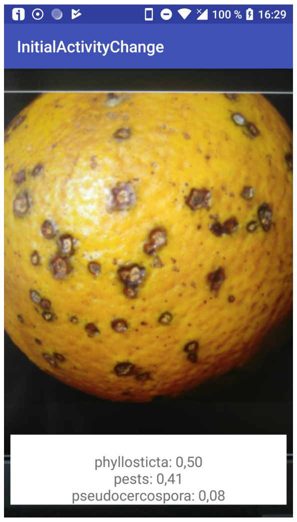
pests: 0,99
pseudocercospora: 0,00
phyllosticta: 0,00

Figure 6.10: Almost a perfect prediction.



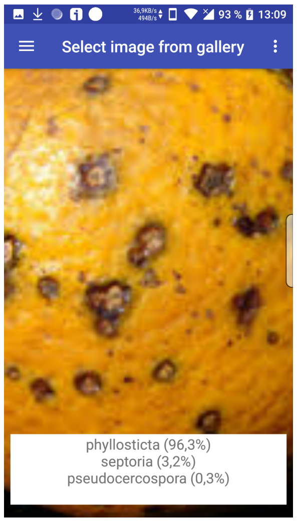
pests: 0,81
healthy: 0,18
leprosis: 0,01

Figure 6.11: Good prediction.



phyllosticta: 0,50
pests: 0,41
pseudocercospora: 0,08

Figure 6.12: Good prediction with an image not in the dataset.



phyllosticta (96,3%)
septoria (3,2%)
pseudocercospora (0,3%)

Figure 6.13: Good prediction as previous result, now getting the image from the cellphone storage instead of taking a picture.

Appendix A

Asynchronous work in Android Framework

Because it is a very important part of the Android development, and because it was very consuming time during the application development, we are going to proceed to give some explanations about the different options that Android provides to do asynchronous work, leaving the UI thread with the lesser work. Not every of the next options have been used in the application development, but some of them where tried until the final options provided directly by some of the libraries or by the Jetbrains Anko library.

A.1 Introduction

During the first times in Android ecosystem applications, one of the main problems where the bad UX experience. One of the main reasons was because of the continuous fails because the main thread, the one which must show the UI, made too much work.

In the current state of the Android development there are many options that allow too avoid this problem, some of them provided by Google and some other by third-party libraries. With this libraries, we'll try to make some of the heaviest work not in the *UI thread* but in some other background threads.

A.2 Java Threads

The lower-level solution, we cannot post info in the UI thread with this method, but it is an easy way to do some job in the background if we don't need to modify the UI directly from other threads.

```
var value : List<String>?
Thread { () ->
    val ls = DataBase.query(query)
    value = ls
}.start()
```

A.3 HaMeR framework

This framework uses the classes *Handler*, *Message* and *Runnable* to do background job and later on showing the results in the UI. When we instantiate the a handler, it takes a reference of the UI thread, and when we `post` inside a background thread, we put the info from one thread to another using a message. In this applications has been used to delay the hiding of the navigation view menu.

```
    ...
    val handler = Handler()
    ...

    handler.postDelayed( {
                val intent = Intent(context, newClass)
                intent.putExtra(ITEM_ID, itemId)
                startActivity(intent)
                finish()
            }, Constants.NAVIGATION_VIEW_LAUNCH_DELAY)
```

A.4 AsyncTask

The first solution provided by Google, it can be use for short-term work. It is more complex and less flexible than the options above, but it provide very clear methods where we should put the code for the background and for the foreground after that has finished.

We must implement a unique method that will run in the background thread, `doInBackground()`, and we can also overwrite other three methods that will run in the main thread: `onPreExecute()`, `onPostExecute()` y `onProgressUpdate()`.

This solution was used in the first version of this application but at the end was substitute by the simpler handler and async method provided by Kotlin.

A.5 Loaders

These are new classes introduced with the API 11, which came to improve some of the defects of the *AsyncTask*. They are usually used for retrieving and storing data in databases.

A.6 Service e IntentService

The *Service* are used for launching processes for not too longer processes, which we can launch and stop whenever we want. They don't need to run in a background thread, indeed they run in the main thread.

If we want longer services, that will run in independent threads, we can use *IntentService*. They cannot be launched and stopped at free, in contrast they run until the activity they are related with ends.

A.7 Anko extensions

Jetbrains, Kotlin developer company, offers some coroutines implementations that we can use to run code in background. One was specified previously with the `async ...` construct. We can use also, instead *handler*, the `UI` coroutine to run some code that was asynchronous in the main thread:

```
async(UI) { list_of_diseases.text = recognizerResults }.
```

A.8 Worker Manager

Recently released by Google, it tries to substitute and simplified almost all of the rest Android frameworks utilities for asynchronous work.

A.9 RxJava

Finally, the RxJava library has been use to make the heavier computation in this app. That's the prediction with the TensorFlow Lite model. The code is easy. First, we have the `UseCase` that defines the computation:

```
override fun run(vararg vs: Any): Observable<String> {
    val model = vs[0] as String
    val image = vs[1] as Bitmap

    var thumbnail: Bitmap

    val scaledImage = Bitmap.createScaledBitmap(
            image, Constants.inputSize, Constants.inputSize, false
    )
    thumbnail = ThumbnailUtils.extractThumbnail(
            image, Constants.inputSize, Constants.inputSize
    )

    val recognizer = Predictor(tfliteContext.retrieveActivity(), model
        )
    result = recognizer.classify(scaledImage)

    recognizer.close()

    return Observable.just(result)
}
```

It returns an `Observable` to which we subscribe in the presenter responsible to updating the screen:

```
val model = sharedPreferencesManager.get(Constants.KEY_MODEL, "")

useCase!!.run(model, image!!)
        .subscribeOn(Schedulers.computation())
```

```
            .observeOn(AndroidSchedulers.mainThread())
            .subscribe( { result -> updateView(result) } )
```

With the `.subscribeOn(...)` method we specified where to we want to run the use case. After that, with `.observeOn(...)` finally we set the thread that will be *observing* if there is any change, so we can use the results emitted by the background thread in this observed thread.

Bibliography

[1] Zalak R. Barot and Narendrasinh Limbad. An approach for detection and classification of fruit disease: A survey. *International Journal of Science and Research*, 4:838–842, December 2015. https://www.ijsr.net/archive/v4i12/8121502.pdf.

[2] Manisha Bhangea and H.A. Hingoliwalab. Smart farming: Pomegranate disease detection using image processing. *Procedia Computer Science*, 58:280–288, August 2015. https://www.sciencedirect.com/science/article/pii/S187705091502133X.

[3] Cesar Calderon. Usda aphis ppq, usda aphis itp. Bugwood.org.

[4] Andrzej Chmielewski. Recyclerview in mvp: Passive view's approach. https://android.jlelse.eu/recyclerview-in-mvp-passive-views-approach-8dd74633158.

[5] Consejo de la Unión Europea. Directiva 2000/29/ce del consejo de 8 de mayo de 2000 relativa a las medidas de protección contra la introducción en la comunidad de organismos nocivos para los vegetales o productos vegetales y contra su propagación en el interior de la comunidad. https://www.boe.es/doue/2000/169/L00001-00112.pdf.

[6] Swati Dewliya and Pratibha Singh. Detection and classification for apple fruit diseases using svm and chain code. *International Research Journal of Engineering and Technology*, 02:2097–2104, August 2015. https://www.irjet.net/archives/V2/i4/Irjet-v2i4338.pdf.

[7] Sander Dieleman. Classifying plankton with deep neural networks. https://benanne.github.io/2015/03/17/plankton.html.

[8] Institut Valencià d'Investigacions Agràries. Gestión integrada de plagas y enfermedades en cítricos - plagas y enfermedades. http://gipcitricos.ivia.es/area/plagas-principales.

[9] Thanh-Toan Do, Ngai-Man Cheung, and Yiren Zhou. Accessible melanoma detection using smartphones and mobile image analysis. arXiV:1411.095532v2 [cs.CV], February 2018. https://arxiv.org/abs/1711.09553.

[10] Alex Drizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.

[11] Shiv Ram Dubey and Anand Singh Jalal. Adapted approach for fruit disease identification using images. arXiv:1405.4930v5 [cs.CV], August 2014. https://arxiv.org/abs/1405.4930.

[12] Andre Esteva, Brett Kuprel, Roberto A. Novoa, Justin Ko, Susan M. Swetter, Helen M.

Blau, and Sebastian Thrun. Dermatologist-level classification of skin cancer with deep neural networks. *Nature*, 542:115–118, 2017. https://www.nature.com/articles/nature21056#references.

[13] Google. Application fundamentals. https://developer.android.com/guide/components/fundamentals?hl=en-419.

[14] Google. Fragments. https://developer.android.com/guide/components/fragments?hl=en-419.

[15] Ronal P. Haff, Sirinnapa Sranwong, Warunee Thanpase, Athit Janhiran, Sumapor Kasemsumran, and Sumio Kawano. Automatic image analysis and spot classification for detection of fruit fly infestation in hyperspectral images of mangoes. *Postharvest Biology and Technology*, 86:23–28, June 2013. https://www.sciencedirect.com/science/article/pii/S0925521413001592.

[16] Kaiming He, Xiangyu Zhang, Shaoquing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. arXiV:1512.03385 [cs.CV], December 2015. https://arxiv.org/abs/1512.03385.

[17] Kaiming He, Xiangyu Zhang, Shaoquing Ren, and Jian Sun. Spatial pyramid pooling in deep convolutional networks for visual recognition. arXiv:1406.4729v4 [cs.CV], April 2015. https://arxiv.org/abs/1406.4729.

[18] Jeremy Howard and Rachel Thomas. Practical deep learning for coders. http://course.fast.ai/.

[19] Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q. Weinberger. Densely connected convolutional networks. arXiv:1608.06993v5 [cs.CV], January 2018. https://arxiv.org/abs/1608.06993.

[20] IVIA. Alternaria alternata. http://gipcitricos.ivia.es/area/plagas-principales/enfermedades/mancha-marron-de-las-mandarinas.

[21] IVIA. Pseudocercospora angolensis. http://gipcitricos.ivia.es/enfermedades-exoticas.html.

[22] Marcin Kitowicz. Recyclerview, swipe to delete. https://github.com/kitek/android-rv-swipe-delete.

[23] Ranjit K.N., Chethan H.K, and Naveena C. Identification and classification of fruit diseases. *International Journal of Engineering Research and Applications*, 6:11–14, July 2016. http://www.ijera.com/papers/Vol6_issue7/Part%20-3/C060703011014.pdf.

[24] Yann LeCun, Léon Botton, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, November 1998.

[25] Min Lin, Qiang Chen, and Shuicheng Yan. Network in network. arXiv:1312.4400v3 [cs.NE], March 2014. https://arxiv.org/abs/1312.4400.

[26] Ammara Masood and Adel Ali Al-Jumaily. Computer aided diagnostic support system for skin cancer: A review of techniques and algorithms. *International Journal of Biomedical Imaging*, 2013:1–22, August 2013. https://www.hindawi.com/journals/ijbi/2013/323268/.

[27] Paymen Moallem, Alireza Serajoddin, and Hossein Pourghassem. Computer vision-based apple grading for golden delicious apples based on surface features. *Information Processing in Agriculture*, 4:33–40, August 2015.

[28] Shara P. Monhanty, David P. Hugues, and Marcel Salathé. Using deep learning for image-based plant disease detection. *Frontiers in Plant Science*, 7:1419, September 2016. https://www.ncbi.nlm.nih.gov/pmc/articles/PMC5032846/.

[29] Andrew Ng. Deep learning specialization. master deep learning, and break into ai. https://www.coursera.org/specializations/deep-learning.

[30] Michael Nielsen. *Neural Networks and Deep Learning*. self-published on-line, 2017. http://neuralnetworksanddeeplearning.com/.

[31] United States Department of Agriculture. Citrus: World markets and trade. https://apps.fas.usda.gov/psdonline/circulars/citrus.pdf.

[32] Unitet State Department of Agriculture. Citrus diseases. http://www.idtools.org/id/citrus/diseases/.

[33] Alireza Pourreza, Wonsuk Lee, Jun Lu, and Pamela Roberts. Development of a multiband sensor for citrus black spot disease detection. *Proceedings of the 13th International Conference on Precision Agriculture*, July 2016. https://www.researchgate.net/publication/307460923_Development_of_a_Multiband_Sensor_for_Citrus_Black_Spot_Disease_Detection.

[34] Jagadeesh D. Pujaria, Rajesh Yakkundimathb, and Abdulmunaf S. Byadgi. Image processing based detection of fungal diseases in plants. *Procedia Computer Science*, 46:1802–1808, February 2015. https://www.sciencedirect.com/science/article/pii/S187705091500201X.

[35] Anderson Rocha, Daniel c. Hauagge, Jacques Wainer, and Siome Goldenstein. Automatic fruit and vegetable classifications from images. *Computers and Electronics in Agriculture*, 70:96–104, January 2010.

[36] Sebastian Ruder. An overview of gradient descent optimization algorithms. http://ruder.io/optimizing-gradient-descent/.

[37] Inkyu Sa, Zongyuan Ge, Feras Dayoub, Ben Upcroft, Tristan Perez, and Chris McCool. Deepfruits: A fruit detection system using deep neural networks. *Sensors*, 1222, August 2016. http://www.mdpi.com/1424-8220/16/8/1222.

[38] Tim Schubert, Bruce Sutton, Ayyamperumal Jeyaprakash, and Charles H. Bronson. Citrus black spot (guignardia citricarpa) discovered in florida. 01 2010.

[39] México Servicio Nacional de Sanidad, Inocuidad y Calidad Agroalimentaria. Xanthomonas citri. https://www.gob.mx/cms/uploads/attachment/file/215782/07_Ficha_T_cnica_-_Cancro_de_los_citricos.pdf.

[40] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. arXiV:1409.1556 [cs.CV], September 2014. https://arxiv.org/abs/1409.1556.

[41] Vijai Singh and A.K. Misra. Detection of plant leaf diseases using image segmentation and

soft computing techniques. *Information processing in Agriculture*, 4:41–49, March 2017. https://www.sciencedirect.com/science/article/pii/S2214317316300154.

[42] SD Pro Solutions. Fruit disease detection using color, texture analysis and ann. https://www.youtube.com/watch?v=2kZIYm5tcTQ/.

[43] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. arXiv:1409.4842 [cs.CV], September 2014. https://arxiv.org/abs/1409.4842.

[44] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. arXiV:1512:00567v3 [cs.CV], 2015. https://arxiv.org/abs/1512.00567.

[45] USDA. Citrus leprosis virus. http://www.idtools.org/id/citrus/diseases/images/fs_images/01Leprosis.jpg.

[46] USDA. Citrus leprosis virus. http://www.idtools.org/id/citrus/diseases/factsheet.php?name=Septoria.

[47] USDA. Septoria citri. http://www.idtools.org/id/citrus/diseases/images/fs_images/Septoria_JimAdaskaveg_Category_2_cs.jpg.

[48] Various. Cs231n: Convolutional neural networks for visual recognition. http://cs231n.stanford.edu/.

[49] Various. Pytorch tutorials. transfer learning tutorial. http://pytorch.org/tutorials/beginner/transfer_learning_tutorial.html.

[50] Jason Yosinski, Jeff Clune, Yoshua Bengio, and Hod Lipson. How transferable are features in deep learning networks? arXiV:1411.1792 [cs.LG], November 2014. https://arxiv.org/abs/1411.1792.

[51] Emny Harna Yossy, Jhonny Pranta, Tommy Wijaya, Heri Hermawan, and Widodo Budiharto. Mango fruit sortation system using neural network and computer vision. *Procedia Computer Science*, 116:596–603, October 2017. https://www.sciencedirect.com/science/article/pii/S0925521413001592.

[52] Yudong Zhang, Shuihua Wang, Genlin Ji, and Preetha Phillips. Fruit classification using computer vision and feedforward neural network. *Journal of Food Engineering*, 143:167–177, December 2014. https://www.sciencedirect.com/science/article/pii/S026087741400291X.