DataScientest bootcamp project (Data Scientist)

PLANTSPycies

Neural networks models for plant species and disease identification

BECCHERE Giovanni

CHERO Guillaume

PARRA CARRASCOSA Valentin Miguel

15/07/2022

TABLE OF CONTENTS

TABLE OF FIGURES

1 Introduction – General overview

Deep learning has become the most talked-about technology owing to its results which are mainly acquired in applications involving language processing, object detection and image classification (Srivastava et al. 2021). Deep Learning is the most used and most preferred approach to machine learning. It is inspired by the working of the biological brain, and how individual neurons work. It has multiple layers, where the upper layers are built on the outputs from lower layers. Thus, the higher the layer, the more complex is the data it processes (Schulz & Behnke 2012). Deep learning is used in many fields such as: Fraud detection, Vocal AI, Autonomous vehicles, etc.

Image recognition technology appeared in the 1940s, but limited by the technical environment and hardware facilities at that time, so it did not get a long development. Until the 1990s, artificial neural networks were combined with support vector machines to provide support for the development of image recognition techniques, enabling its wide utilization, for example, in license plate recognition, face recognition, object detection, and so on (Li 2022). In recent times, the industrial revolution makes use of computer vision for their work. Automation industries, robotics, medical field, and surveillance sectors make extensive use of deep learning.

Recently, agriculture and food security have become a new application field of image recognition using deep learning. Indeed, modern technologies have given human society the ability to produce enough food to meet the demand of more than 7 billion people. However, food security remains threatened by a number of factors including climate change (Tai et al., 2014), the decline in pollinators (Report of the Plenary of the Intergovernmental Science-PolicyPlatform on Biodiversity Ecosystem and Services on the work of its fourth session, 2016), plant diseases (Strange and Scott, 2005), and others. Plant diseases are not only a threat to food security at the global scale, but can also have disastrous consequences for smallholder farmers whose livelihoods depend on healthy crops.

At all scales, farmers are on the front lines of pest control. This is why, identifying a disease correctly when it first appears is a crucial step for efficient disease management (Mohanty, Hughes, & Salathé, 2016). However, financial capacities of small farmers and giant agribusinesses are not equivalent, although the danger can come from both places. Therefore, it’s now important to develop more applications of pest detection in order to prevent the next global epidemic.

In plants, disease is an abnormal phenotypic state, which shows deviations, called "symptoms", from the expected normal phenotype, and reduces the plant's growth. There are two main types of plant diseases : biotic or infectious diseases (caused by parasitic microorganisms, such as bacteria or fungi) and Abiotic diseases (generally caused by environmental factors, whether natural or man-made). The most common symptom of plant diseases is a change of color or shape on their leaves. To give an example, we can have a look at an apple leaf. Apple trees can suffer from different diseases : for example, cedar-apple rust and the Black Rot of Apple. Gymnosporangium juniperi-virginianae is a plant pathogen that causes cedar-apple rust. In virtually any location where apples grow, apple rust can be a destructive or disfiguring disease. The brightly colored spots produced on the leaves make it easy to identify. Small, yellow-orange spots appear on the upper surfaces of the leaves. The

Black Rot of Apple occurs in early spring when the leaves are unfolding. They appear as small, purple specks on the upper surfaces of leaves and enlarge to circular lesions 1/8 inch to 1/4 inch in diameter. The margins of the lesions remain purple, while the centers turn tan to brown.

		
Healthy	**Rust**	**Black Rot**

Figure 1 - Apple leaves examples

Plant diseases can be detected by the different manifestations on their leaves. Indeed, as you can see on the figure, each disease produces a unique reaction on the plant. Moreover, leaves are also unique for each species. Each plant's species is characterized by an individual shape of leaf. Because species and disease identification is both possible using the leaf, we can use them in our detection model.

The aim of our project is the development of a deep learning model to identify plant species and their disease based on photos.

In the following sections, we are going to describe in detail our dataset. Then, we will build a simple (naïve) model with which we will make our first observation. After this naïve model, we will develop 3 different transfer learning models, based on: VGG16, Resnet 50 and Inception V3. Finally, we will discuss their results and look for explanations regarding their performances.

Srivastava, S., Divekar, A. V., Anilkumar, C., Naik, I., Kulkarni, V., & Pattabiraman, V. (2021). Comparative analysis of deep learning image detection algorithms. Journal of Big Data, 8(1), 1-27.

Schulz, H., & Behnke, S. (2012)

Li, Y. (2022, January). Research and application of deep learning in image recognition. In 2022 IEEE 2nd International Conference on Power, Electronics and Computer Applications (ICPECA) (pp. 994-999). IEEE.

Mohanty, S. P., Hughes, D. P., & Salathé, M. (2016). Using deep learning for image-based plant disease detection. Frontiers in plant science, 7, 1419.

Elaraby, A., Hamdy, W., & Alruwaili, M. (2022). Optimization of deep learning model for plant disease detection using particle swarm optimizer. Computers, Materials & Continua, 71(2), 4019-4031.

2 Datasets Exploration and choice

At the beginning of our project, we had a range of possible datasets we could use:

- Open Images Dataset V6
- COCO (Common Objects in Context)
- V2 Plant Seedlings
- Plant Disease
- PlantVillage Dataset
- New Plant Diseases Dataset

A first exploration of each of the datasets and their characteristics allowed us to choose the best-suited dataset for our project. Let's get to the details.

First of all, we have two huge datasets which are used to identify a plant in a photo among any other objects, persons or animals. These are:

- Open Images Dataset V6[1]: 1.7 million images allowing 600 class object segmentation

- COCO (Common Objects in Context)[2]: 123k images focused in identifying potted plants

Our project's goal is to develop a model capable of identifying species + disease from leaf images, so these 2 datasets were discarded, as they were not useful.

Next, we have four datasets, organized in folders according to the leaf species and its disease, but not for all the datasets. Let's analyze their main characteristics:

- V2 Plant Seedlings[3]: this dataset contains a total of over 5k images from crop and weed seedlings, at different growth stages. These are not totally developed leaves, we have 12 species, but these species are different from the ones we find in the 3 other datasets. Moreover, we get no information about diseases, so this dataset was also discarded

- New Plant Diseases Dataset[4]: we find here a total of almost 88k images, divided in two parts (train and validation). We have a fairly balanced dataset, with a rough number of images per folder between 1600 and 2000 for train set, and 400 to 500 images per folder for the validation set. We have 14 species (apple, blueberry, cherry, corn, grape, orange, peach, pepper, potato, raspberry, soybean, squash, strawberry, and tomato), and a different number of diseases for each species: from 0 (only healthy leaves' images) for the blueberry, orange, raspberry soybean and squash leaves, to 9 diseases for the tomato leaf

- Plant Disease[5]: this dataset includes 54k images, with the same folder structure as New Plant Diseases Dataset, but with a duplicate folder (for peach__healthy) and with an unbalanced number of images per folder (class): it goes from 122 images for Potato__healthy, to 4286 images for Tomato___Tomato_Yellow_Leaf_Curl_Virus.

[1] https://storage.googleapis.com/openimages/web/download.html
[2] https://cocodataset.org/#home
[3] https://www.kaggle.com/datasets/vbookshelf/v2-plant-seedlings-dataset
[4] https://www.kaggle.com/datasets/vipoooool/new-plant-diseases-dataset
[5] https://www.kaggle.com/datasets/saroz014/plant-disease

- PlantVillage Dataset[6]: this is the biggest dataset of these last 3 datasets with the same species folder structure, it has 163k images. It's divided in 3 parts: color + greyscale + segmented (black background), with the same number of images for each of these 3 parts, but umbalanced inside, going from 152 images (Potato___healthy) to 5507 images (Orange___Haunglongbing_(Citrus_greening)).

Our choice is then clear: we'll use New Plant Diseases Dataset. It has a fairly high number of images per class and the dataset can be considered as balanced

The following table helped us compare and choose our dataset:

	OBJECT SEGMENTATION		SPECIES IDENTIFICATION	SPECIES + DISEASES		
Dataset	Open Images Dataset V6	COCO	V2 Plant Seedlings	New Plant Diseases Dataset	Plant Disease	PlantVillage Dataset
Description	segmentation (600 classes). FiftyOne tool	Context 2017 train/val browser	Crop and weed seedlings at different growth stages	Healthy and unhealthy crop leaves	No description available	diseased plant leaf images and corresponding labels
Comments	To identify a plant among other objects	To identify a plant among other objects Focused on "potted plants"	Limited number of species, and different than the other 3 dataset species. No disease information	Train + validation Balanced on train (from 1600 to 2000 images per type) and on validation (400 to 500 images per type)	Déséquilibré sur train : entre 122 images (Potato___healthy) et 4286 images (Tomato___Tomato_Yellow_Leaf_Curl_Virus	(fond noir) Unbalanced number of images : from 152 (Potato___healthy) to 5507 images (Orange___Haunglongbing_(Citrus
Summary	1,743,042 training images	123,287 images	5539 files .png => 5539	87.9k files .jpg => 87.9k	54.3k files .jpg => 54.3k .png => 1 .jpeg => 1	163k files .jpg => 163k .png => 2 .jpeg => 2 Same number of leaves on each folder (color, greyscale et
Folders			Black-grass	Apple___Apple_scab	Apple___Apple_scab	Apple___Apple_scab
			Charlock	Apple___Black_rot	Apple___Black_rot	Apple___Black_rot
			Cleavers	Apple___Cedar_apple_rust	Apple___Cedar_apple_rust	Apple___Cedar_apple_rust
			Common Chickweed	Apple___healthy	Apple___healthy	Apple___healthy
			Common wheat	Blueberry___healthy	Blueberry___healthy	Blueberry___healthy
			Fat Hen	Cherry_(including_sour)___Powdery_milde	Cherry_(including_sour)___Powdery_milde	Cherry_(including_sour)___Powdery_milde
			Loose Silky-bent	Cherry_(including_sour)___healthy	Cherry_(including_sour)___healthy	Cherry_(including_sour)___healthy
			Maize	Corn_(maize)___Cercospora_leaf_spot Gr	Corn_(maize)___Cercospora_leaf_spot Gr	Corn_(maize)___Cercospora_leaf_spot Gr
			Scentless Mayweed	Corn_(maize)___Common_rust_	Corn_(maize)___Common_rust_	Corn_(maize)___Common_rust_
			Shepherd's Purse	Corn_(maize)___Northern_Leaf_Blight	Corn_(maize)___Northern_Leaf_Blight	Corn_(maize)___Northern_Leaf_Blight
			Small-flowered Cranesbill	Corn_(maize)___healthy	Corn_(maize)___healthy	Corn_(maize)___healthy
			Sugar beet	Grape___Black_rot	Grape___Black_rot	Grape___Black_rot
				Grape___Esca_(Black_Measles)	Grape___Esca_(Black_Measles)	Grape___Esca_(Black_Measles)
				Grape___Leaf_blight_(Isariopsis_Leaf_Sp	Grape___Leaf_blight_(Isariopsis_Leaf_Sp	Grape___Leaf_blight_(Isariopsis_Leaf_Sp
				Grape___healthy	Grape___healthy	Grape___healthy
				Orange___Haunglongbing_(Citrus_greeni	Orange___Haunglongbing_(Citrus_greeni	Orange___Haunglongbing_(Citrus_greeni
				Peach___Bacterial_spot	Peach___Bacterial_spot	Peach___Bacterial_spot
				Peach___healthy	Peach___healthy	Peach___healthy
				Pepper,_bell___Bacterial_spot	Peach___healthy (288 files)	Pepper,_bell___Bacterial_spot
				Pepper,_bell___healthy	Pepper,_bell___Bacterial_spot	Pepper,_bell___healthy
				Potato___Early_blight	Pepper,_bell___healthy	Potato___Early_blight
				Potato___Late_blight	Potato___Early_blight	Potato___Late_blight
				Potato___healthy	Potato___Late_blight	Potato___healthy
				Raspberry___healthy	Potato___healthy	Raspberry___healthy
				Soybean___healthy	Raspberry___healthy	Soybean___healthy
				Squash___Powdery_mildew	Soybean___healthy	Squash___Powdery_mildew
				Strawberry___Leaf_scorch	Squash___Powdery_mildew	Strawberry___Leaf_scorch
				Strawberry___healthy	Strawberry___Leaf_scorch	Strawberry___healthy
				Tomato___Bacterial_spot	Strawberry___healthy	Tomato___Bacterial_spot
				Tomato___Early_blight	Tomato___Bacterial_spot	Tomato___Early_blight
				Tomato___Late_blight	Tomato___Early_blight	Tomato___Late_blight
				Tomato___Leaf_Mold	Tomato___Late_blight	Tomato___Leaf_Mold
				Tomato___Septoria_leaf_spot	Tomato___Leaf_Mold	Tomato___Septoria_leaf_spot
				Tomato___Spider_mites Two-spotted_spi	Tomato___Septoria_leaf_spot	Tomato___Spider_mites Two-spotted_spi
				Tomato___Target_Spot	Tomato___Spider_mites Two-spotted_spi	Tomato___Target_Spot
				Tomato___Tomato_Yellow_Leaf_Curl_Vir	Tomato___Target_Spot	Tomato___Tomato_Yellow_Leaf_Curl_Vir
				Tomato___Tomato_mosaic_virus	Tomato___Tomato_Yellow_Leaf_Curl_Vir	Tomato___Tomato_mosaic_virus
				Tomato___healthy	Tomato___Tomato_mosaic_virus	Tomato___healthy
					Tomato___healthy	

Figure 2 - Datasets comparison table

3 Exploratory Data Analysis

Our downloaded dataset was divided in *train* and *validation,* with 38 subfolders each (our 38 classes, combination of species+disease). *Train* folder has a total of 70,295 files and *validation* folder has 17,571 files

We renamed the original *validation* folder as *test*, to later use part of the *train* folder to create de *validation* set (we'll see that in the datasets definition to feed our models).

[6] https://www.kaggle.com/datasets/abdallahalidev/plantvillage-dataset

We have images with a 256 x 256 x 3 (RGB) size, and .jpg or .JPG extensions. We plotted a bar chart to verify the dataset balance between our classes. Here we can see the example for train dataset folders (.JPG files in orange and .jpg files in blue):

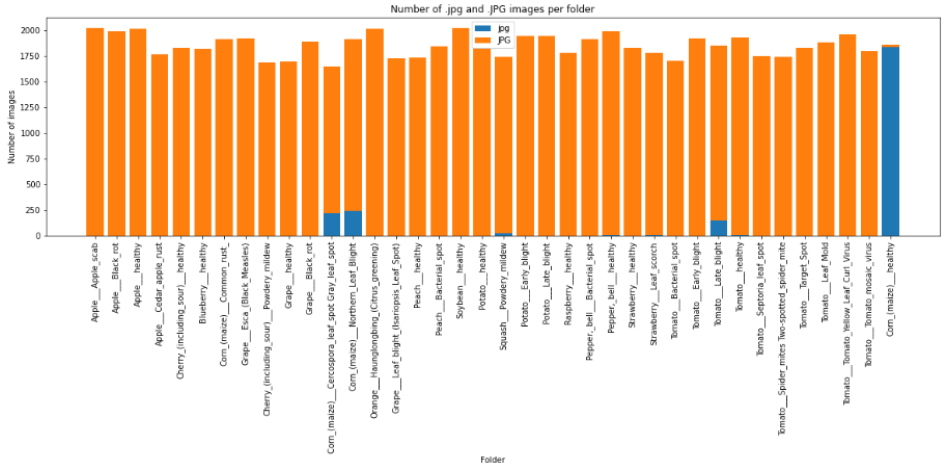

Figure 3 - Number of .jpg and .JPG images per folder (train)

The fact of having two different extensions (.jpg and .JPG) turned out to be a bit tricky, because we had to have that in mind to obtain a balanced reduced set, as we'll explain on the next chapter.

We calculated and plotted the ratio of delta (number of images in each folder - average number of images per folder in the dataset) to this average (in green) and to the total of images in the dataset (in red). We can see that the difference is neglectable respecting to the whole dataset:

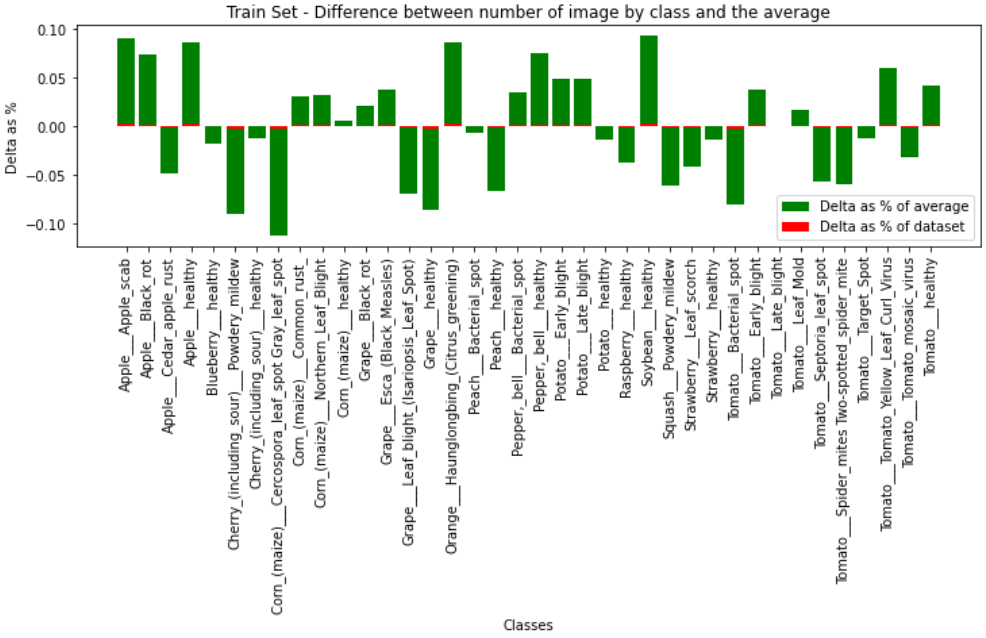

Figure 4 - Ratio of delta (nb of images in each folder - average nb of images per folder in the dataset)

This exploration was made using "*os*", "*PIL*", "*pathlib*", and "*glob*" packages, that allowed us to:

- access and explore directories (*os.listdir*),
- read images (PIL.Image.open(<filename.ext>)
- create new directories (*os.makedirs*),
- create paths from directories (*pathlib.Path*),
- explore files matching a wildcard (*pathlib.Path(*<directory>*).glob(*<wildcard>*))*,

- copy files into a directory (*shutil.copy*(<filename>, < directory>))

Finally, we checked a sample of every class:

Figure 5 - Samples of all 38 classes

3.1 Creation of a reduced dataset

Creating, adjusting and training neural networks can be very time-consuming, because many iterations can be made before finding the hyperparameters that are best-suited for our particular dataset and objective. Many different possible configurations for our different models have been tested, and the long calculation times could be a problem. Thus, we decided to create a reduced size dataset for initial testing and the first model hyperparameters adjustments. Anyway, the final hyperparameter adjustments have been done when testing on the full dataset.

We created 2 new folders (*train_min*, and *test_min*), where we copied, respectively, 200 and 50 images from each class subfolder of the original full train and test datasets (roughly 10% of the full datasets).
This was done taking care of including both type of file extensions (.jpg and .JPG) in the reduced dataset generation code, because some class folders have both of these extensions, and one particular class (*Corn_(maize)__healthy*) has mainly .jpg files.

If we only included .jpg extension in the reduced dataset generation code, we would get empty class folders for most of our classes. And if we only included .JPG extension, we would get an almost empty *Corn_(maize)__healthy* folder.

All these operations between folders were possible using "*os*", "*pathlib*", "*glob*" (as seen in the EDA chapter), and "*shutil*" packages, that allowed us to copy files into a directory (*shutil.copy*(<filename>, < directory>))

4 Convolutional Neural Network model

4.1 Datasets preprocessing

The first model we tried was a simple convolutional neural network, and the first step was to prepare the train, validation and test sets:

- Train: 80% of the original train dataset, with a standard *batch_size* = 32
- Validation: 20% of the original train dataset, with a big batch_size = 512
- Test: 100% of the original validation dataset (that we renamed into "test"), with a big batch_size = 512

From the total of 87866 images, this split creates a final distribution of:
- 56236 images for training set: 64 % overall
- 14059 images for validation set: 16 % overall
- 17571 images for test set: 20 % overall

We chose a bigger batch size for validation dataset to speed up testing, better using the hardware, while it shouldn't make any difference on validation or test results.
These 3 datasets were created with 'categorical' label mode (one-hot encoding), coherent with the activation function choice for the model's output layer (we'll see that later on this chapter)

A rescaling was also carried out, turning the array values in the images from a 0-255 range to a 0-1 range for the model.

4.2 CNN model creation

In order to initialize weights for testing different models, we created a function to build our sequential CNN model. This model is built as a stack of multiple layers of different types with different functionalities. We have 3 types of layers:

- input layers (to take in the raw input),
- hidden layers (to get input from another layer, process it and output to the next one),
- and output layers (to make a prediction).

Hidden layers typically use the same activation function. This function decides whether a neuron should be activated or not by calculating weighted sum and further adding bias with it, introducing non-linearity into the output of a neuron.

The output layer will use a different activation function depending on the type of prediction required by the model. The layers used in the model are:

- **Convolutional layer**: it applies a filter (a set of weights) to the input to extract features (create a feature map). The filter is applied by convolution (an element-wise multiplication of two matrices followed by a sum). Its main parameters are:
    - Number of filters: we can stack multiple convolutional layers, with an increasing number of filters (usually 32, 64, 128, 256 and 512), to learn multiple features in parallel (we'll have 32 to 512 different ways of extracting features from an input)
    - Kernel size: it's the filter size, a 2-tuple specifying the width and height of the 2D convolution window (usually 1x1, 3x3, 5x5 or 7x7). We can start with bigger filter sizes to learn larger features and quickly reduce dimensions, and then reduce the filter size. We'll start with 7x7, then 5x5 and finish with 3x3
    - Activation: function to be applied after filtering. Typically, Rectified Linear Activation (ReLU) for hidden layers, as it's less susceptible to vanishing gradients problem[7] than Logistic (Sigmoid) and Hyperbolic Tangent (Tanh). For output layers, the typical choice is Softmax (outputs a vector of values that sum to 1.0, that can be interpreted as probabilities of class membership). Target labels used to train a model with softmax output layer will be vectors with 1 for the target class and 0 for all other classes.
    - Padding: addition of pixels to the edge to avoid border effect due to filtering. If set to "same", it adds the padding required to the input image to ensure that the output has the same shape as the input.

- **Max pooling layer**: used to reduce the spatial dimensions of the output by taking the maximum value over an input window. We get a summarized version of the features detected in the input. The main parameter is pool_size (window size), typically 2x2 or 3x3

- **Flatten layer**: used to turn the 2D filter maps (the output of the convolutional layers) into a single 1-dimensional array of features. We need to pass a feature vector into a dense layer for classification.

[7] Vanishing gradients: in models with a high number of layers using certain activation functions, the gradients of the loss function approache zero, making the network hard to train.

- **Dense layer**: used to classify images based on output from convolutional layers. Each neuron receives input from all the neurons of the previous layer. Its main hyperparameters are:
    - Units: dimensionality of the output vector => it has to be 38 in our case (the number of plant classes) for the last layer
    - Activation: typically Softmax for the last layer (gives probabilities of class membership)

- **Dropout**: randomly sets input units to 0 (with a frequency of its rate parameter) at each step during training. This helps prevent overfitting. Inputs not set to 0 are scaled up by 1/(1 - rate) such that the sum over all inputs is unchanged. Its main parameter is Rate: Float between 0 and 1, the fraction of the input units to drop.

The model was then defined as shown in the next figure, and compiled using the following parameters:

- Optimizer: *Adam*. The optimizer is used to search through different weights and metrics during training. It defines how the network will be updated based on the loss function. Adam is a popular version of gradient descent because it automatically tunes itself and it achieves good results fast. The following comparison (made by its creators) on an image classification problem, shows why we chose this optimizer:

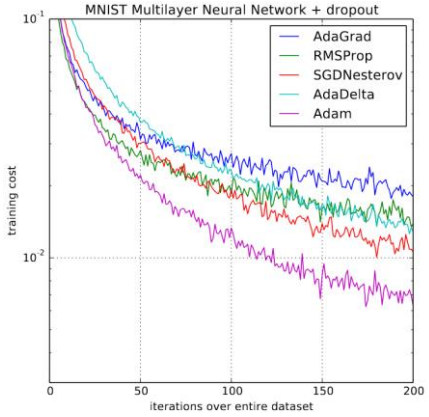

Figure 6 - Adam optimizer vs. others

- Loss: *categorical_crossentropy*. Loss is the quantity that will be minimized during training, categorical_crossentropy is used for a multi-class classification problem
- Metrics: *accur*acy. We define here what we want the model to evaluate and collect during training and testing. We want to evaluate classification accuracy.

The model summary gives us the output shape and the number of parameters for each layer, and the total number of the model's parameters (which are all trainable in our case). The detail of our model summary and an explanation of output sizes and number of parameters can be found in Appendix 1

4.3 CNN model training and testing

Training a network means finding the best set of weights to map inputs to outputs in our dataset. We have trained our model in the following configurations:

- With reduced dataset:
  - Learning rate = 0.001 + 10 epochs => really bad performance, the model doesn't learn anything. Possible reason: learning rate is too high => the training diverges
  - Learning rate = 0.0001 + 10 epochs => much better result (the model now learns), but still poor (less than 70% in validation).

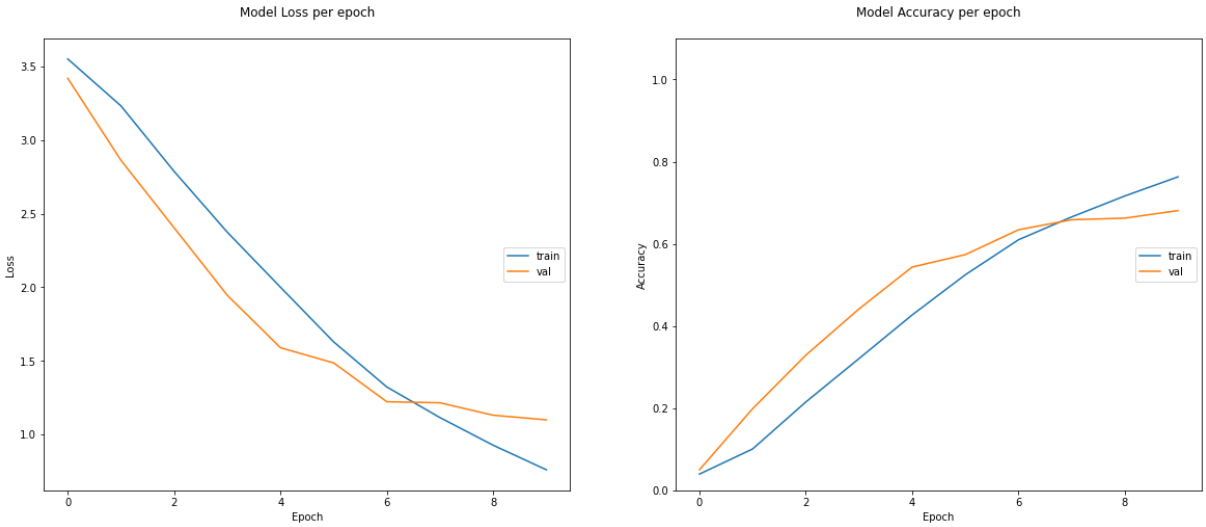

Figure 7 - CNN reduced ds, lr=0.0001, 10 epoch

From the evolution of the curves, we could say that the model just needs more "time" to get a good accuracy => Let's try increasing the number of epochs from 10 to 25.

- Learning rate = 0.0001 + 25 epochs.

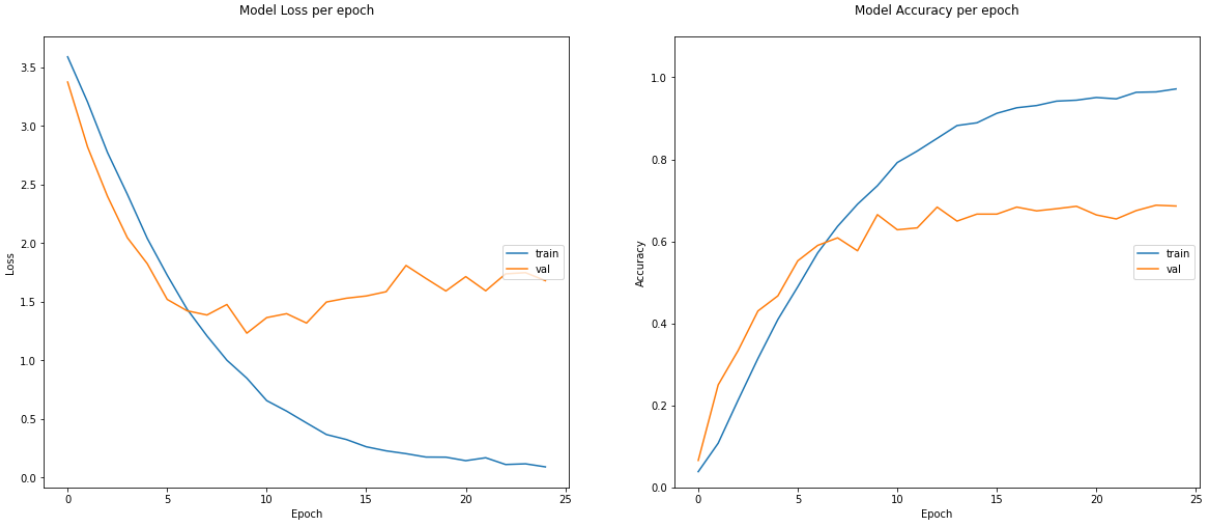

Figure 8 - CNN reduced ds, lr=0.0001, 25 epoch

We can see that the model overfits after 7 epochs. This may mean we need a bigger dataset. We could have avoided here wasting time and resources by stopping the training using callbacks (we will use this on transfer learning).

- With full dataset:
  - Learning rate = 0.0001 + 10 epochs => pretty good results (already 96% validation accuracy with virtually no overfitting 97% accuracy on training)

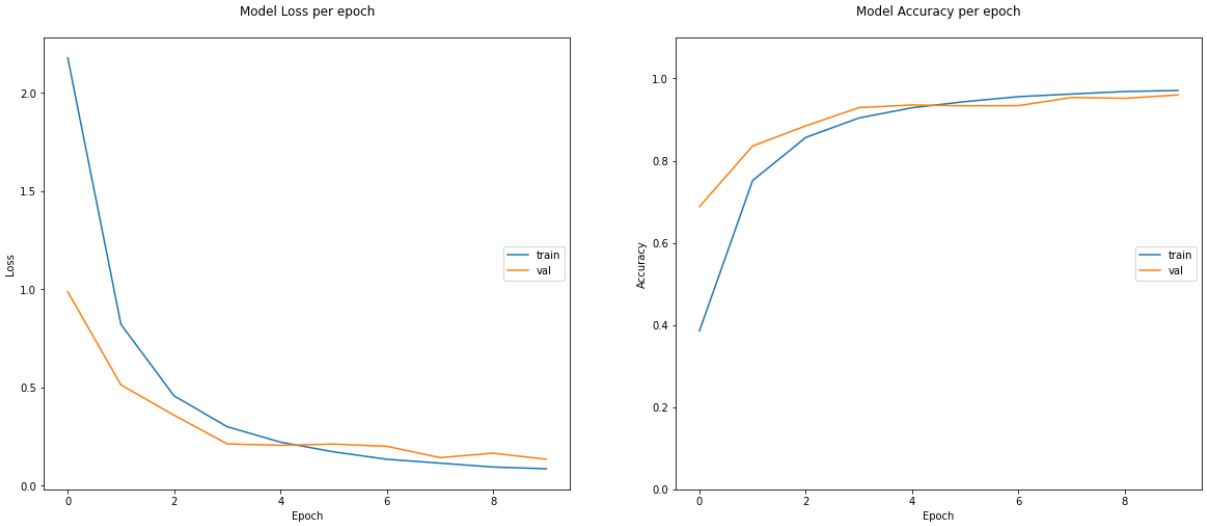

Figure 9 - CNN full ds, lr=0.0001, 10 epoch

  - Learning rate = 0.0001 + 200 epochs => we get our best results for a standard CNN (almost 99% accuracy on training and 97.2% on test dataset). Validation accuracy is also around 97%. We get a small overfit (around 2%)

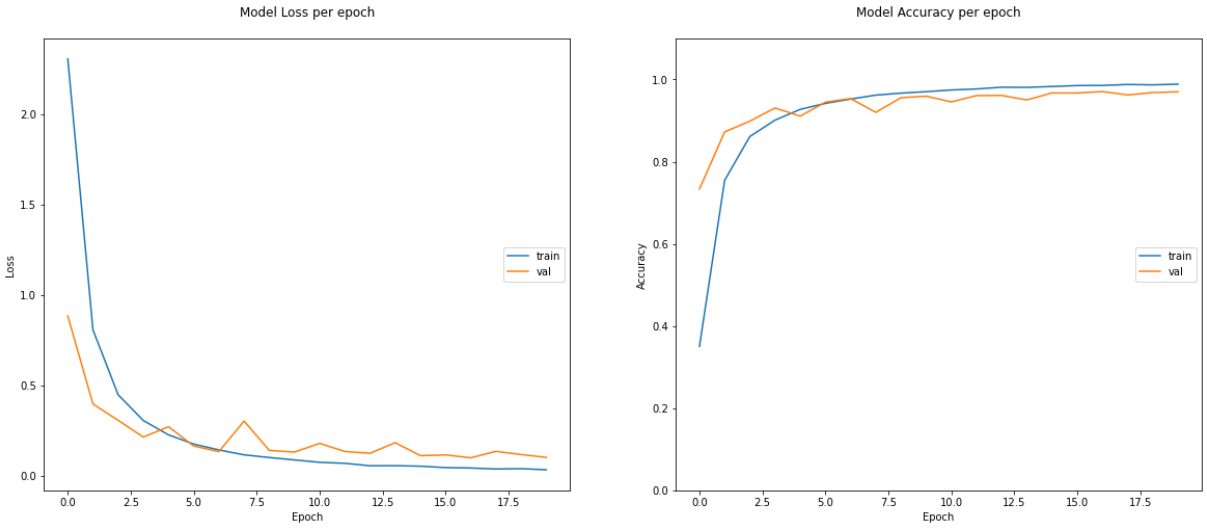

Figure 10 - CNN full ds, lr=0.0001, 20 epoch

If we analyze the confusion matrix, we can see that the most confused classes are:

- Class 7 (Corn_(maize)___Cercospora_leaf_spot Gray_leaf_) vs class 9 (Corn_(maize)___Northern_Leaf_Blight)

- Class 34 (Tomato___Target_Spot) vs class 33 (Tomato___Spider_mites Two-spotted_spider_mite)

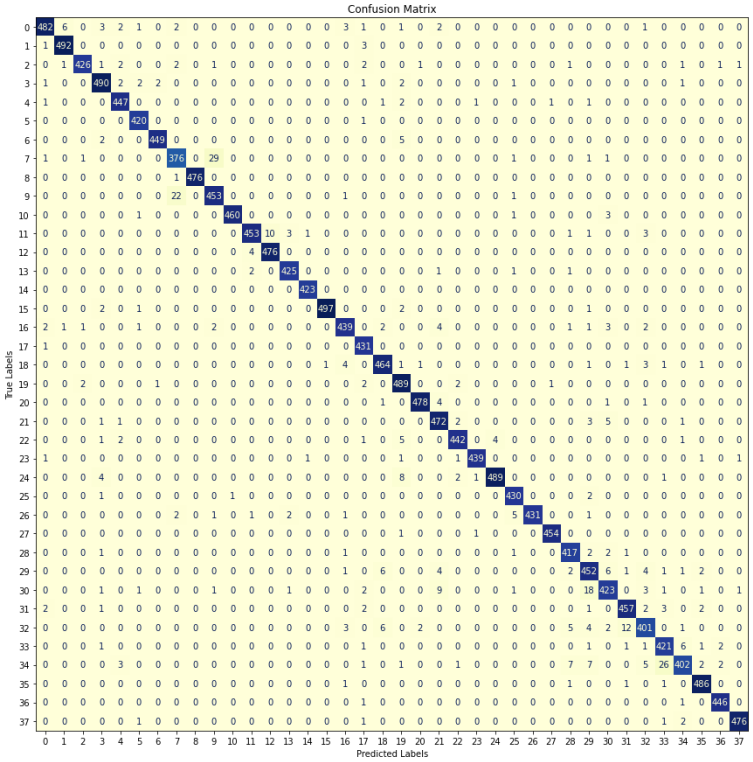

Figure 11 - CNN best model confusion matrix

Let's check the classification report:

	precision	recall	f1-score	support
0.0	0.980	0.956	0.968	504
1.0	0.984	0.992	0.988	496
2.0	0.991	0.968	0.979	440
3.0	0.963	0.976	0.969	502
4.0	0.974	0.985	0.979	454
5.0	0.981	0.998	0.989	421
6.0	0.993	0.985	0.989	456
7.0	0.928	0.917	0.923	410
8.0	1.000	0.998	0.999	477
9.0	0.930	0.950	0.940	477
10.0	0.998	0.989	0.994	465
11.0	0.985	0.960	0.972	472
12.0	0.979	0.992	0.986	480
13.0	0.986	0.988	0.987	430
14.0	0.995	1.000	0.998	423
15.0	0.998	0.988	0.993	503
16.0	0.967	0.956	0.962	459
17.0	0.958	0.998	0.977	432
18.0	0.967	0.971	0.969	478
19.0	0.944	0.984	0.964	497
20.0	0.992	0.986	0.989	485
21.0	0.952	0.973	0.962	485
22.0	0.982	0.969	0.976	456
23.0	0.993	0.987	0.990	445
24.0	0.992	0.968	0.980	505
25.0	0.973	0.991	0.982	434
26.0	1.000	0.971	0.985	444
27.0	0.996	0.996	0.996	456
28.0	0.956	0.981	0.969	425
29.0	0.909	0.942	0.925	480
30.0	0.948	0.914	0.931	463
31.0	0.964	0.972	0.968	470
32.0	0.941	0.920	0.930	436
33.0	0.923	0.968	0.945	435
34.0	0.964	0.880	0.920	457
35.0	0.980	0.992	0.986	490
36.0	0.989	0.996	0.992	448
37.0	0.994	0.990	0.992	481
accuracy			0.972	17571
macro avg	0.972	0.972	0.972	17571
weighted avg	0.973	0.972	0.972	17571

Test Accuracy : 97.2 %
Precision Score : 97.2 %
Recall Score : 97.2 %

Figure 12 - CNN best model classification report

We can confirm here that classes 34 and 7 are some of the worst classified (as we easily saw on the confusion matrix): both have F1-score around 92%. But we can also observe here that class 29 (Tomato___Early_blight) is not very well classified by the model (F1-score around 92% as well). This is not so clear when checking the confusion matrix, as errors are spread over multiple other classes, but we can point it out easily with this classification report.

Generally speaking, it looks like classification errors are quite concentrated on tomato leaves, which seems logical, as we have 10 classes (9 diseases).

The accuracy on both the validation and the test sets are around 97 %, which means the model has similar behavior on different "unseen" data, even if it has a small overfitting (+2% accuracy on training set).

All these trainings have been done with a batch size of 32.To see the effect of a high batch size and compare with the transfer learning models, we tried our final model with the full dataset but with batch size = 128.

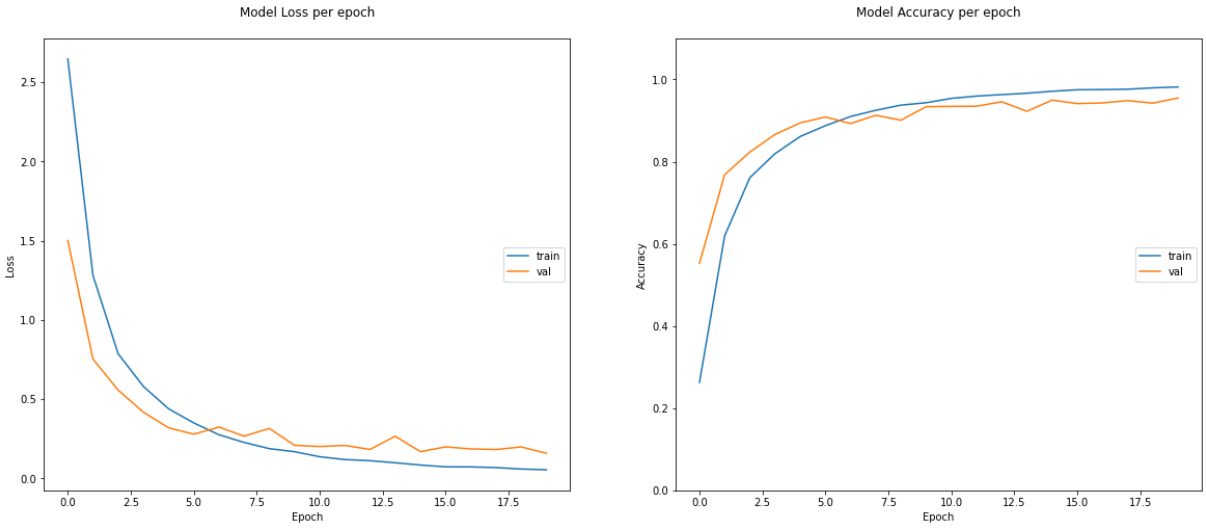

Figure 13 - CNN full ds, lr=0.0001, 20 epoch, batch size = 128

Typically, when using a larger batch there is a degradation in ability of the model to generalize. We can see here that indeed, the effect of changing batch_size from 32 to 128 is a bigger difference between training and validation accuracy (98% for training vs 95% for validation => -3%), and a lower final test accuracy (from 97% to 95% => -2%).

5 Transfer Learning

5.1 General approach, the difficulties we had

After having investigated a first CNN model which was already delivering satisfying results, we decided to investigate the transfer learning technique, looking for a way to reach even better results.

In a nutshell, using transfer learning allows us to save time and computing resources: someone else has already spent the time and computational resources to learn a lot of features and our model will likely benefit from it.

The concept of transfer learning was first introduced, with a first mathematical model and geometric interpretation of transfer learning, by Stevo Bozinovski and Ante Fulgosi in 1976 *(Stevo. Bozinovski and Ante Fulgosi (1976). "The influence of pattern similarity and transfer learning upon the training of a base perceptron B2." (original in Croatian) Proceedings of Symposium Informatica 3-121-5, Bled)*.

Then it was experimentally proved in 1981 with its application in training a neural network on a dataset of images representing letters of computer terminals.

The intuition behind transfer learning, especially in the task of image classification like the one we are dealing with, is that if a model is trained on a sufficiently large and general dataset, it will effectively serve as a generic model for visual recognition. In practice it's possible to leverage on the general feature maps of an already trained model, without having to train a complete new neural network model from scratch. This is especially useful as, to obtain optimal results, a new model needs to be trained on sufficiently large datasets, requiring resources and time.

There are basically two ways of using transfer learning:

- We can simply leverage the knowledge already developed by a model and usually based on a huge dataset, by applying the pretrained model to a specific dataset. Here the goal is to identify the meaningful features for a specific classification problem. The key advantage of doing this is that, rather than re-train the entire model, we only have to add a new classification part and train only this part from scratch on a specific dataset.

- We can also go a step further and fine-tune an existing pretrained model, by limiting the training to significantly fewer parameters of the model, while still leveraging on its already learned knowledge. In practice this consists in labeling most of the existing layers of the pre-trained model as not trainable (freeze) while allowing the training (unfreeze) for some of the higher-level layers, to allow a further and more specific training of the model.

In applying transfer learning to our classification problem, we took as a starting point the example of the notebook related to this technique. Our approach was the one defined in the code below, where the “base_model” is the specific pretrained model to be reused in our transfer learning.

For each base_model we:

- took the weights or the model pre-trained on ImageNet (weights='imagenet');
- excluded the layers at the top of the model (include_top=False) so that the classification of our images in the specific classes was done by the additional layers we specified in the definition of the model;
- specified the input_shape according to our image sizes so that we don’t have to reshape the input depending on the requirement of every pre-trained model;
- started with the pre-trained models completely freezed.

To adapt the complete model to our specific classification problem, on top of the base_model we added the following layers, which we kept always fully trainable, representing the classification part of our network:

- one GlobalAveragePooling2D layer which reduces the spatial dimensions of the output by taking the average over the whole pool;
- two blocks, each composed by one Dense and one Dropout layer: the Dense layer participates in the actual classification, the Dropout layer allows to reach a better generalization and a lower tendency to overfit, by randomly dropping connections between neurons during the training;
- a final Dense layer, which produces the final output in the form a 38 dimension vector, containing the probabilities of an image belonging to the various classes

```
n_class=38
base_model = VGG16(weights='imagenet', include_top=False,
                   classes= n_class,
                         input_shape = (img_height,img_width,3))


# Freeze the layers of base_model
for layer in base_model.layers:
    layer.trainable = False


model = Sequential()
model.add(base_model)
model.add(GlobalAveragePooling2D())
model.add(Dense(1024,activation='relu'))
model.add(Dropout(rate=0.2))
model.add(Dense(512, activation='relu'))
model.add(Dropout(rate=0.2))
model.add(Dense(n_class, activation='softmax'))
```

As in the case of the CNN model, we started working on the reduced dataset. This allowed testing the code and looking at the first results with few quick test runs.

The first pretrained model we looked at was the classical VGG16. The first results we had were disappointing and well below the ones obtained with our CNN model. This was of course possible, but quite unexpected.

To get a better understanding we decided to test another “popular” pretrained model: ResNet50.

When looking at ResNet50 we had again puzzling results: during the training both train and validation accuracy increased slowly but regularly up to around 65% and then just stagnated around that level. This was again possible but unexpected, so we went back to the homeworks by reviewing all the steps. Finally, we realized that, for both VGG16 and ResNet50, we had “simply” not applied the dedicated preprocessing on the datasets.

We made a couple of more tests of transfer learning with ResNet50, after having included the proper preprocessing step. We indeed had good results already on the reduced Dataset.

The conclusion of this first investigation phase was that, once having properly set all conditions, transfer learning looked promising. Therefore, as the next step we decided to test transfer learning with three specific models, adding InceptionV3 to the already mentioned VGG16 and ResNet50.

During the training, we applied some Callbacks. Across the different models we use a total of four Callbacks for different purposes: save the current “best” model or its weights (ModelCheckpoint), avoid stagnation of the metrics (LearningRateScheduler and ReduceLROnPlateau), stop the training to limit the risk of overfitting (EarlyStopping).

We started with a generic version and for each model, based on the first simulations, and we adapted the arguments and/or excluded Callbacks according to the impact on the training.

Finally, for each model we experimented the unfreezing of some of the final layers of the pretrained component, with the aim of improving the results.

5.2 Three Models and their results

5.2.1 VGG16

Overview of VGG16

Karen Simonyan and Andrew Zisserman proposed the idea of the VGG network in 2013 and submitted the actual model based on the idea in the 2014 ImageNet Challenge *(K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. In*

ICLR, 2015). They called it VGG after the department of Visual Geometry Group in the University of Oxford that they belonged to.

The key idea behind VGG was investigating the possibility of creating a deep network by using a combination of several very small convolution filters, rather than larger filters as the ones currently used at the time.

The creators of VGG went quite extreme with this concept of looking at small convolution filters: in the architecture of VGG the image is passed through a stack of convolutional layers, where the used filters have a very small receptive field of 3 X 3, which is the smallest size to capture the notion of left/right, up/down, center.

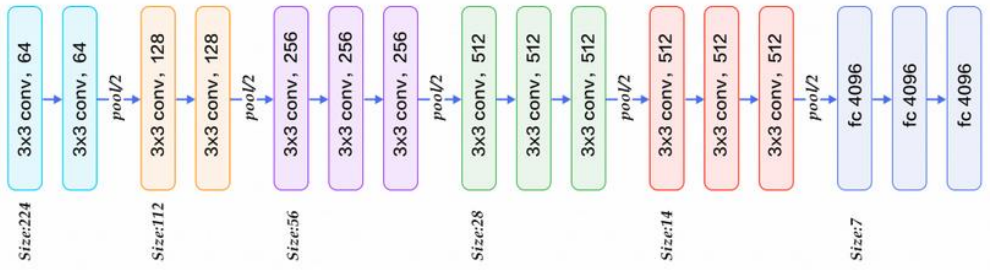

Figure 14 - VGG16 architecture

In fact, replacing a bigger filter with a combination of multiple 3X3 filters didn't increase the complexity of the model, while having more non-linear activation layers made the decision functions more discriminative and so increased the ability of the network to converge faster.

The consistent use of 3 x 3 convolutions across the network made the network very simple, elegant, and easy to work with.

The VGG network secured the first and the second places at the ImageNet Challenge 2014

in the localization and classification tracks respectively.

Our model based on pre-trained VGG16

As discussed in the introduction of transfer learning, first of all we preprocessed the images of the various sets (train, validation and test) with the VGG16 dedicated preprocessing function.

Then we created an instance of a CNN model for which the first layer was a freezed version of VGG16 on top of which we added a few trainable layers to adapt the overall model to our

specific classification problem. The parameters of VGG16 and the additional layers were the ones described in the introduction of transfer learning.

We first trained and evaluated this version, then we looked at possible versions with some layer of VGG16 unfreezed.

The final structure of this freezed version is the following:

```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 vgg16 (Functional)          (None, 8, 8, 512)         14714688

 global_average_pooling2d (G  (None, 512)              0
 lobalAveragePooling2D)

 dense (Dense)               (None, 1024)              525312

 dropout (Dropout)           (None, 1024)              0

 dense_1 (Dense)             (None, 512)               524800

 dropout_1 (Dropout)         (None, 512)               0

 dense_2 (Dense)             (None, 38)                19494

=================================================================
Total params: 15,784,294
Trainable params: 1,069,606
Non-trainable params: 14,714,688
```

Callbacks

Based on the first simulations, we noticed that the model was quickly reaching good results, with the validation metrics quickly stabilizing, despite the continuation of a slow but regular increase of the train metrics. In other words, the issue wasn't identifying weights giving good results but rather looking for fine tuning while limiting the risk of overfitting. So, we decided to use quickly adapting callbacks.

Results

First of all, we looked at the evolution of the loss function and the accuracy during training on both training and validation sets.

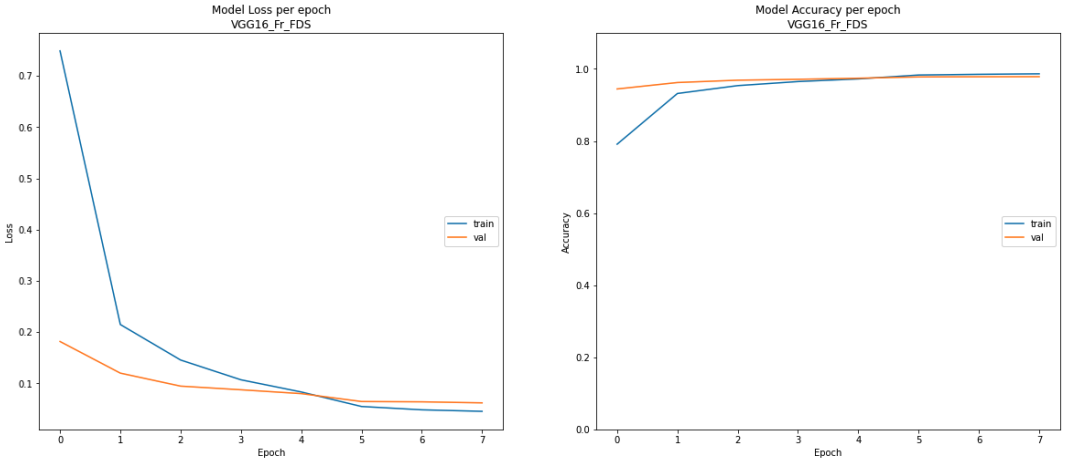

Figure 15 - VGG 16 freezed loss and accuracy curves

We saw that the model converged quite quickly and after only a few epochs was already close to 98% accuracy on the validation set. At that point some slight overfit started taking place as the training accuracy kept improving while validation remained stable. Here EarlyStopping played its role, interrupting the training after 8 epochs. Actually the intervention of the callback could be a bit too early but we did other tests and the best results remained always at around 98%

To evaluate the model we let it make predictions on the test set and looked at the classification report and the confusion matrix

	precision	recall	f1-score	support
0.0	0.982	0.980	0.981	504
1.0	0.986	0.992	0.989	496
2.0	0.991	0.986	0.989	440
3.0	0.986	0.988	0.987	502
4.0	0.996	0.993	0.994	454
5.0	0.995	0.998	0.996	421
6.0	0.987	0.998	0.992	456
7.0	0.964	0.927	0.945	410
8.0	1.000	0.996	0.998	477
9.0	0.937	0.971	0.954	477
10.0	1.000	1.000	1.000	465
11.0	0.981	0.992	0.986	472
12.0	0.992	0.988	0.990	480
13.0	1.000	0.998	0.999	430
14.0	1.000	1.000	1.000	423
15.0	1.000	0.998	0.999	503
16.0	0.981	0.996	0.988	459
17.0	0.995	0.993	0.994	432

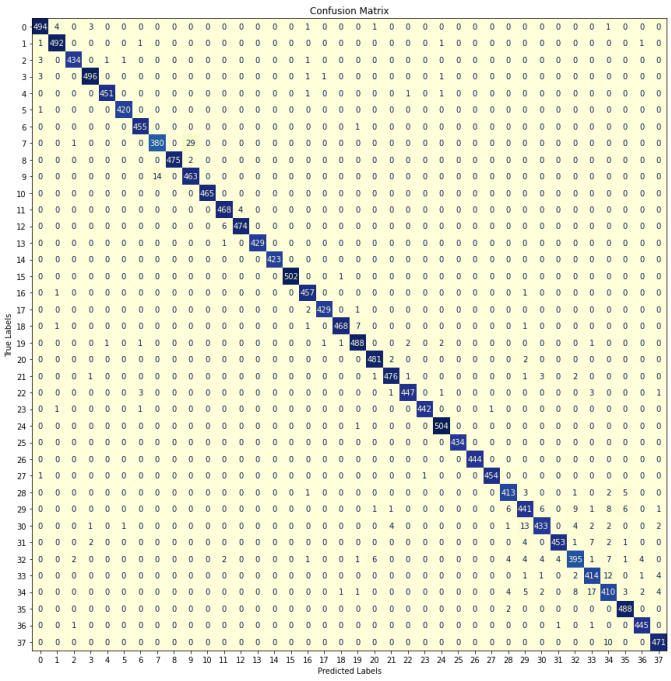

Figure 16 - VGG 16 freezed confusion matrix

```
        18.0      0.994     0.979     0.986       478
        19.0      0.976     0.982     0.979       497
        20.0      0.982     0.992     0.987       485
        21.0      0.983     0.981     0.982       485
        22.0      0.991     0.980     0.986       456
        23.0      0.998     0.993     0.995       445
        24.0      0.988     0.998     0.993       505
        25.0      1.000     1.000     1.000       434
        26.0      1.000     1.000     1.000       444
        27.0      0.998     0.996     0.997       456
        28.0      0.960     0.972     0.966       425
        29.0      0.926     0.919     0.923       480
        30.0      0.964     0.935     0.950       463
        31.0      0.989     0.964     0.976       470
        32.0      0.936     0.906     0.921       436
        33.0      0.926     0.952     0.939       435
        34.0      0.903     0.897     0.900       457
        35.0      0.968     0.996     0.982       490
        36.0      0.982     0.993     0.988       448
        37.0      0.975     0.979     0.977       481

    accuracy                          0.979     17571
   macro avg      0.979     0.979     0.979     17571
weighted avg      0.979     0.979     0.979     17571

Test Accuracy   : 97.93 %
Precision Score : 97.93 %
Recall Score    : 97.93 %
```

As it was already the case with our first model, looking at the F1-score by class we observed that the model struggled in classifying class 7 and 9 (both related to corn's leaves) and the classes from 28 to 37 (the ones related to tomato's leaves). The Confusion Matrix showed that the model sometimes inverted Class 7 and 9, while for the tomato's leaves there was more dispersion of misclassification among the various classes.

The accuracy results were globally already quite good with an accuracy in both training and test close to 98% and a training accuracy only 0.6% higher (very limited overfitting). These results were already very good and, as expected, already above our best previous CNN model.

Unfreezing some layers

In order to see if we could improve our results, we tried to unfreeze some layers of the VGG16 component of the model, while starting from the results of the freezed model. We decided to unfreeze the last 4 layers of our base_model VGG16. The reason for this choice is that

because we instantiated the base_model with include_top = False, the last 4 layers of our base_model were the last convolution block and the final MaxPooling2D. So unfreezing the last 4 layers corresponded to unfreezing the last convolution block (please refer also to the summary of the base_model in the notebook). This translated into adding a bit more of 7 million parameters to the training.

The results after unfreezing, without even changing any parameter/Callback in the training, looked very promising: validation accuracy rised quickly to around 99.5% triggering an early stopping (see picture below). Given the impressive potential result, we didn't do additional fine tuning and looked at the evaluation of this version of the model.

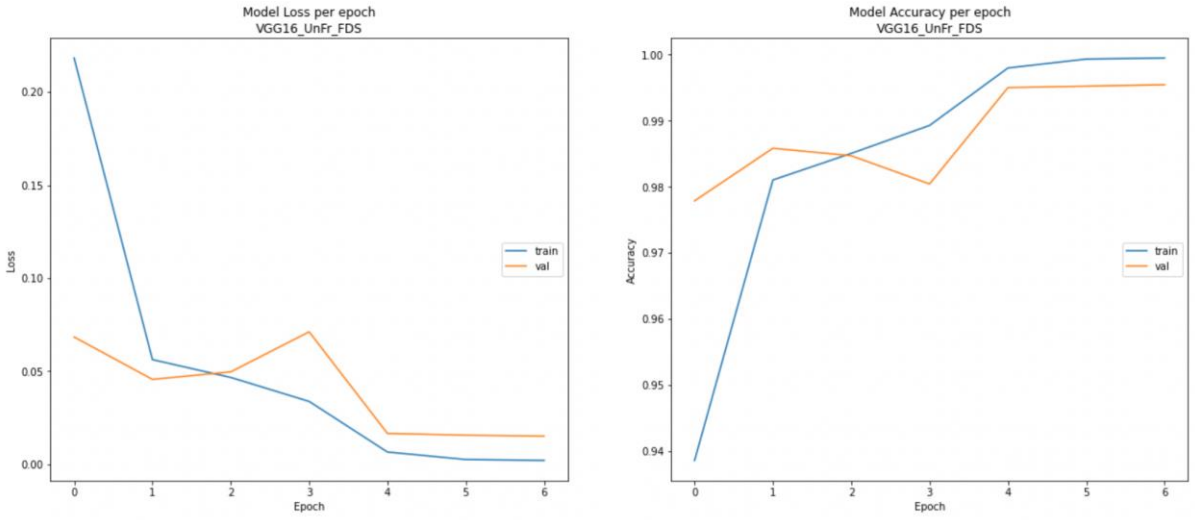

Figure 17 - VGG 16 unfreezed loss and accuracy curves

The evaluation of the model with the predictions on the test set confirmed the improvement that unfreezing brought to our model. The overall results were quite impressive, suggesting that the overall dataset is too good to be true or could be not sufficiently diversified. Accuracy in both training and test was around 99.5% (versus a training accuracy close to 100%).

```
              accuracy                           0.995     17571
             macro avg       0.995     0.995     0.995     17571
          weighted avg       0.995     0.995     0.995     17571


Test Accuracy    : 99.5 %
Precision Score : 99.5 %
Recall Score     : 99.5 %
```

The F1-score by class and the Confusion Matrix gave indications similar to the ones previously observed for VGG16 unfreezed (see above and refer to the notebook for more details).

Conclusion VGG16

We can conclude that our model based on VGG16 performed quite well and showed a nice improvement with the unfreezing of the last convolution block. The results looked even too good, ringing a first bell on the, potentially too high, quality of the dataset.

We appreciated the simplicity of VGG16 which made experimenting and understanding the meaning of our first unfreezing. This came out to be very useful later on when we had to deal with the unfreezing of the more complex structures of ResNet50 and InceptionV3.

5.2.2 ResNet50

Overview of ResNet50

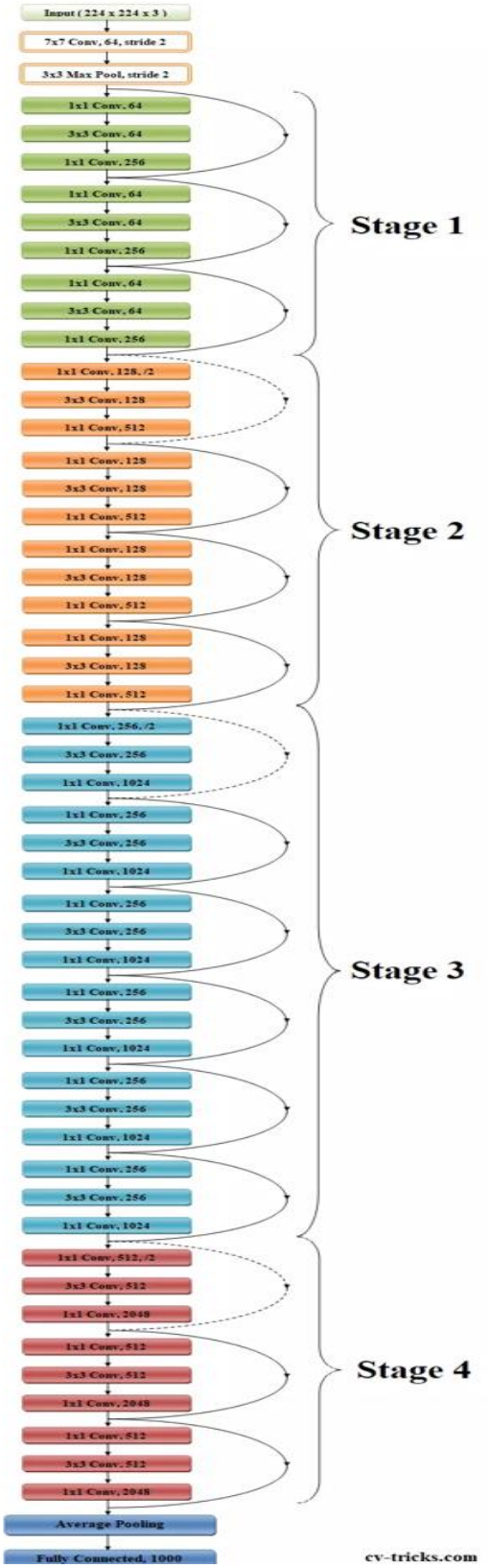

Figure 18 - ResNet50 architecture

ResNet (Residual Networks) is an innovative neural network that was first introduced by Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun *(He, K., Zhang, X., Ren, S. and Sun, J. (2016) Deep Residual Learning for Image Recognition. Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 770-778).*

The key idea of ResNet was the introduction of "Shortcut Connections" and "Residual Learning". A Shortcut Connection is a direct connection that skips over some layers of the model. Residual Learning is the idea that, given an input x, a block of layers, rather than approximating the underlying mapping H(x), could approximate the residual mapping given by the H(x) – x. The creators of ResNet hypothesized, and showed in their paper, that this approach could help in addressing vanishing/exploding gradients, a phenomenon which was becoming an issue for deep learning developments.

ResNet-50 is the 50 layers deep version of ResNet. The picture on the left shows its general architecture including the shortcut connections

Residual learning is applied with shortcut connections from a block to the following one. When the input and output dimensions are the same, the shortcut connection performs identity shortcuts. When the dimensions of input and output are different, the shortcut connection performs projection shortcuts : the dimension is matched by dimension reduction and dimension elevation through a 1 × 1 convolution layer.

In the picture on the left the projection shortcuts are represented with dotted lines while the identity shortcuts with solid lines.

Because of concerns on the training time, ResNet50, like all ResNet with 50 or more layers, is structured in so-called “bottleneck” building blocks. Each block is composed of 3 layers (instead of 2 as it is for ResNet34). The three layers are 1×1, 3×3, and 1×1 convolutions, where the 1×1 layers are responsible for reducing and then increasing (restoring) dimensions, leaving the 3×3 layer a bottleneck with smaller input/output dimensions.

Note that, as proved by the creators of ResNet, the use of parameter-free identity shortcuts, in combination with the bottleneck architectures lead to lower time complexity and model size.

The following picture, extracted from the original article introducing ResNet, represents on the left a building block for ResNet34 and on the right a bottleneck building block for ResNet50.

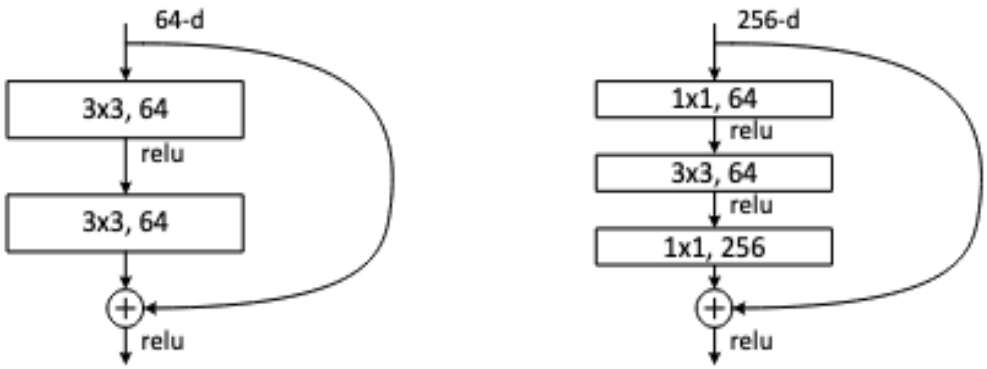

Figure 19 - ResNet34 vs. ResNet50 building blocks

Shortcut connection enables having a deeper network and finally ResNet became the Winner of ILSVRC 2015 in image classification, detection, and localization, as well as Winner of MS COCO 2015 detection, and segmentation**.**

Our model based on pre-trained ResNet50

As discussed in the introduction of transfer learning, first of all we preprocessed the images of the various sets (train,validation and test) with the ResNet50 dedicated preprocessing function.

Then we created an instance of a CNN model for which the first layer was a freezed version of ResNet50 on top of which we added a few trainable layers to adapt the overall model to our specific classification problem. The parameters of ResNet50 and the additional layers are the ones introduced in the introduction of transfer learning

We first trained and evaluated this version, then we looked at possible fine-tuning by unfreezing some layers of ResNet50 unfreezed.

The final structure of this freezed version is the following:

```
Layer (type)                 Output Shape              Param #
=================================================================
 resnet50 (Functional)       (None, 8, 8, 2048)        23587712

 global_average_pooling2d (G  (None, 2048)             0
 lobalAveragePooling2D)

 dense (Dense)               (None, 1024)              2098176

 dropout (Dropout)           (None, 1024)              0

 dense_1 (Dense)             (None, 512)               524800

 dropout_1 (Dropout)         (None, 512)               0

 dense_2 (Dense)             (None, 38)                19494
=================================================================
Total params: 26,230,182
Trainable params: 2,642,470
Non-trainable params: 23,587,712
```

Callbacks

As in the case of VGG16, based on the first simulations, we noticed that the model was quickly reaching strong results, with the validation metrics quickly stabilizing, despite the continuation of a slow but regular increase of the train metrics. So again, we decided to use quickly adapting callbacks.

Results

First of all, we looked at the evolution of the loss function and the accuracy during training on both training and validation sets.

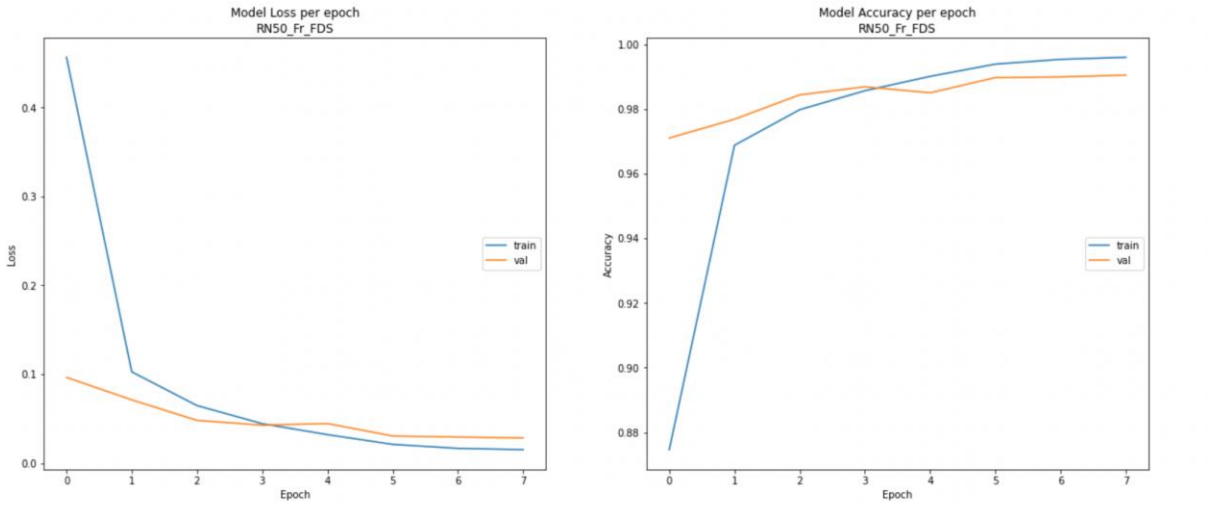

Figure 20 - ResNet50 freezed loss and accuracy curves

We saw that the model converged very quickly and after only a few epochs was already around 99% accuracy on the validation set. At that point some slight overfit started taking place as the training accuracy kept improving while validation remained stable. Here, EarlyStopping played its role, interrupting the training after 8 epochs. Actually, the intervention of the callback could be a bit too early, but we considered that the results, if confirmed by the evaluation of the model, were already very impressive.

To evaluate the model, we let it make predictions on the test set and looked at the classification report and the confusion matrix

	precision	recall	f1-score	support
0.0	0.992	0.990	0.991	504
1.0	0.994	0.994	0.994	496
2.0	0.995	1.000	0.998	440
3.0	0.992	0.998	0.995	502
4.0	0.996	1.000	0.998	454
5.0	0.998	0.998	0.998	421
6.0	1.000	0.998	0.999	456
7.0	0.970	0.937	0.953	410
8.0	0.998	0.996	0.997	477
9.0	0.945	0.975	0.960	477
10.0	1.000	1.000	1.000	465
11.0	0.994	0.996	0.995	472
12.0	0.996	0.994	0.995	480
13.0	1.000	1.000	1.000	430
14.0	1.000	0.998	0.999	423
15.0	1.000	0.998	0.999	503
16.0	0.998	0.991	0.995	459
17.0	0.993	0.998	0.995	432
18.0	0.998	0.996	0.997	478
19.0	0.996	0.994	0.995	497
20.0	1.000	1.000	1.000	485
21.0	0.996	0.979	0.988	485
22.0	0.993	0.996	0.995	456
23.0	1.000	0.998	0.999	445
24.0	0.994	0.998	0.996	505
25.0	1.000	1.000	1.000	434
26.0	1.000	1.000	1.000	444
27.0	1.000	1.000	1.000	456
28.0	0.984	0.993	0.988	425
29.0	0.947	0.969	0.958	480
30.0	0.974	0.963	0.969	463
31.0	0.996	0.985	0.990	470
32.0	0.977	0.968	0.972	436
33.0	0.961	0.970	0.966	435
34.0	0.937	0.937	0.937	457
35.0	0.994	0.996	0.995	490
36.0	0.993	0.996	0.994	448
37.0	0.990	0.988	0.989	481
accuracy			0.989	17571
macro avg	0.989	0.989	0.989	17571
weighted avg	0.989	0.989	0.989	17571

Test Accuracy : 98.9 %

Precision Score : 98.9 %

Recall Score : 98.9 %

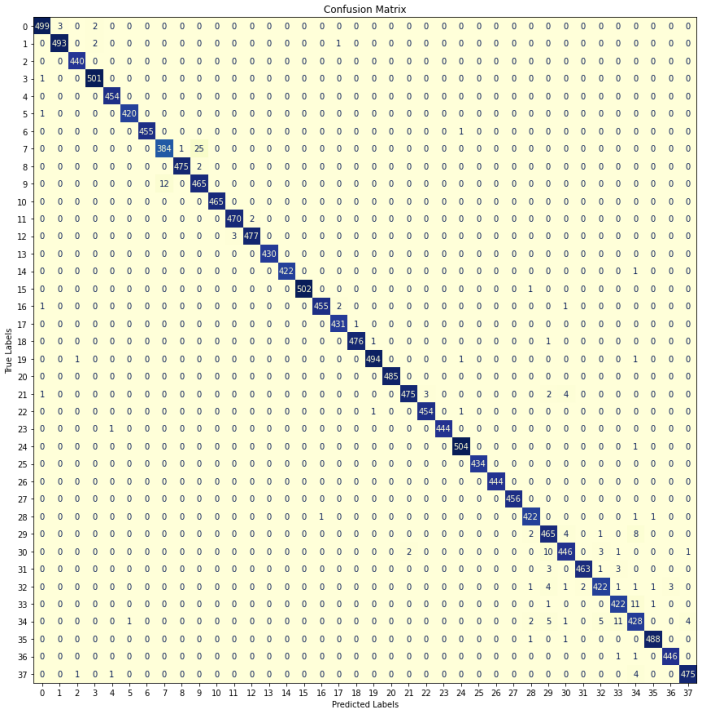

Figure 21 - ResNet50 freezed confusion matrix

The F1-score by class and the Confusion Matrix gave indications similar to the ones previously observed for VGG16 unfreezed (see the in VGG16 section above and refer to the notebook for more details).

The overall results were overall quite good with an accuracy in both training and test close to 99% and a training accuracy only 0.6% higher (very limited overfitting). These results were again quite good and actually better than the ones achieved with VGG16 before unfreezing.

Unfreeze some layers

In order to see if we can improve our results, we tried to unfreeze some layers of the ResNet50 component of the model, while starting from the results of the freezed model. We decided to unfreeze the last 10 layers of our base_model ResNet50. The reason for this choice is that because we instantiated the base_model with include_top = False, the last 10 layers of our base_model included all the components of the last "bottleneck" building block (please refer also to the summary of the base_model in the notebook). So, unfreezing the last 10 layers corresponded to unfreezing the last "bottleneck" building block. This translated into adding around 4.5 million parameters to the training.

When we unfreezed these layers the first results were not really satisfying: the training was stopped after only a few epochs with a validation accuracy in line with the one of the freezed model. The validation loss started rising which in turn quickly triggered the EarlyStopping. It seemed that the aggressive callbacks intervened too early.

So, we did another run after making a couple of changes in the starting learning rate and the Callbacks:

- decrease the starting learning rate to 0.0001;
- remove the LearningRateScheduler as we already had a starting learning rate 10 times lower, and we also expected the ReduceLROnPlateau to be more active because we started from an already very performing model;
- increase the patience of EarlingStopping to 4.

With this new set of Callbacks we reached a validation Accuracy of 99.4% as shown below

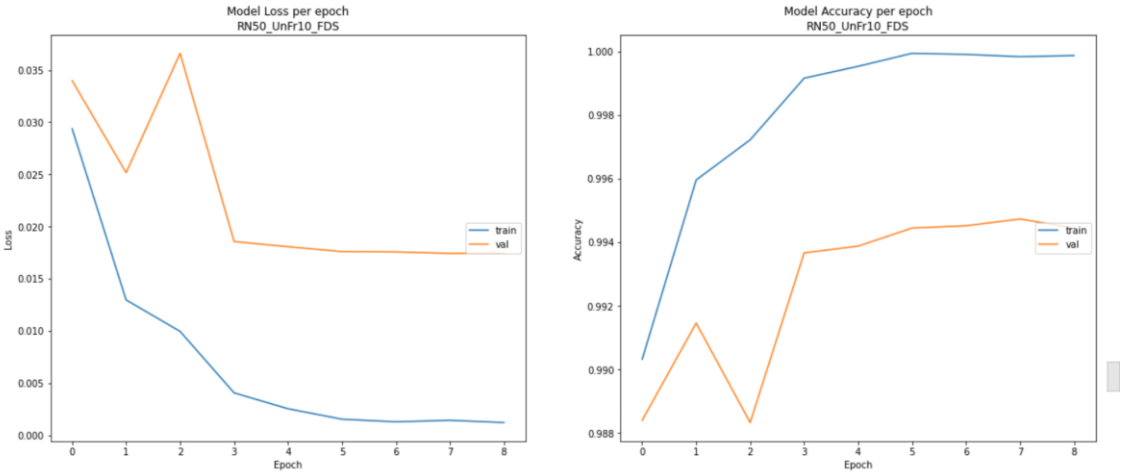

Figure 22 - ResNet50 unfreezed loss and accuracy curves

The evaluation of the model with the predictions on the test set confirmed the improvement that unfreezing brought to our model.

```
              accuracy                           0.994     17571
             macro avg       0.994     0.994     0.994     17571
          weighted avg       0.994     0.994     0.994     17571


Test Accuracy    : 99.4 %
Precision Score : 99.4 %
Recall Score     : 99.4 %
```

The F1-score by class and the Confusion Matrix gave indications similar to the ones previously observed for VGG16 unfreezed (see the in VGG16 section above and refer to the notebook for more details).

Conclusion ResNet50

We can conclude that our model based on ResNet50 performed very well already in the unfreezed version and benefited from a slight improvement when unfreezing the last ”bottleneck” block. As in the case of VGG16 the results looked even too good, ringing a first bell on the, potentially too high, quality of the dataset.

We appreciated looking at the innovative idea of residual learning, which gives the ResNet network an edge versus the VGG network. Also in our specific case, this was likely the reason for the better performance delivered by ResNet50.

5.2.3 InceptionV3

Overview of InceptionV3

Inception v3 is an image recognition model. The model is the culmination of many ideas developed by multiple researchers over the years. The model itself is made up of symmetric and asymmetric building blocks, including convolutions, average pooling, max pooling, concatenations, dropouts, and fully connected layers. Batch normalization is used extensively throughout the model and applied to activation inputs. Loss is computed using Softmax.

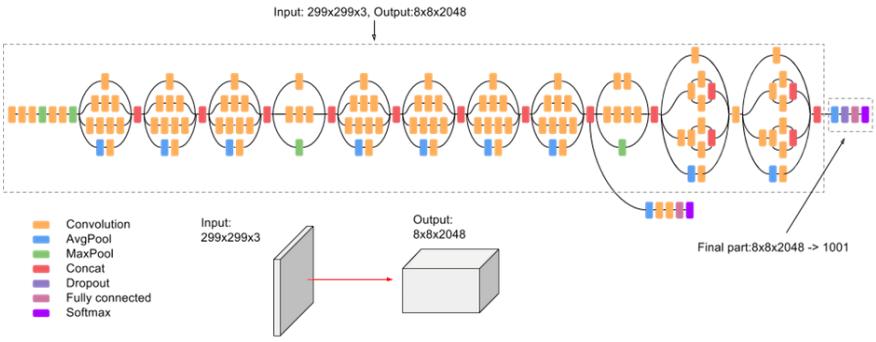

Figure 23 - InceptionV3 layer architecture

Inception models are derived from VGGNet. Although VGGNet has the compelling feature of architectural simplicity, this comes at a high cost: evaluating the network requires a lot of computation. Inception architecture was designed to perform well even under strict constraints on memory and computational budget. This has made it feasible to utilize Inception networks in big-data scenarios, where huge amounts of data need to be processed at reasonable cost or scenarios where memory or computational capacity is inherently limited, for example in mobile vision settings. However, these methods add extra complexity. Works well with small size images.

Our model based on pre-trained InceptionV3

As discussed in the introduction of transfer learning, first we preprocessed the images of the various sets (train, validation and test) with the InceptionV3 dedicated preprocessing function.

Then we created an instance of a CNN model for which the first layer was a freezed version of InceptionV3, on top of which we added a few trainable layers to adapt the overall model to our specific classification problem. The parameters of InceptionV3 and the additional layers were the ones introduced in the introduction of transfer learning.

We first trained and evaluated this version, then we looked at possible fine-tuning by unfreezing some layers of InceptionV3 unfreezed.

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 inception_v3 (Functional)   (None, None, None, 2048)  21802784

 global_average_pooling2d (G  (None, 2048)             0
 lobalAveragePooling2D)

 dense (Dense)               (None, 1024)              2098176

 dropout (Dropout)           (None, 1024)              0

 dense_1 (Dense)             (None, 512)               524800

 dropout_1 (Dropout)         (None, 512)               0

 dense_2 (Dense)             (None, 38)                19494
=================================================================
Total params: 24,445,254
Trainable params: 2,642,470
Non-trainable params: 21,802,784
_________________________________________________________________
```

Callbacks

Based on the test simulation, we noticed that this model was going slowly to stabilization. So, we decided to keep weak callbacks to let the model find the best parameters without stopping it too early. We were able to make this decision because overfitting was very weak.

Results

Results of simulation on the full dataset gave us these visualizations on the loss and accuracy dynamic per epoch.

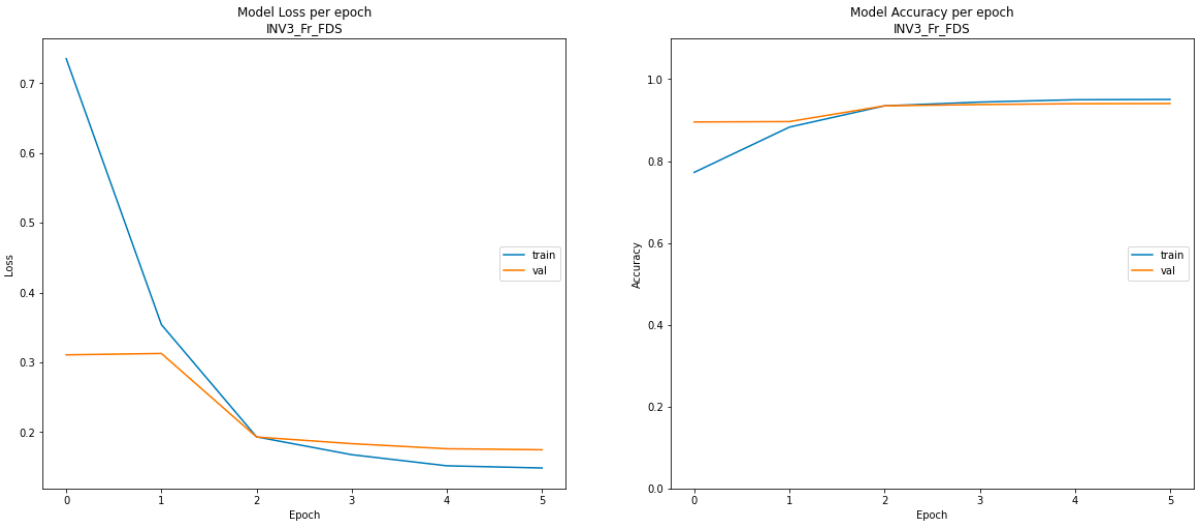

Figure 24 - InceptionV3 freezed loss and accuracy curves

We observed that the model converged, after 5 epochs, to a high value of accuracy. Callbacks were useful to stop simulation before the overfitting increased too much and decreased the learning rate two times.

Prediction

To investigate our model, we looked at the confusion matrix and the classification report. The classification report showed that the accuracy is at 94% on the validation dataset.

	precision	recall	f1-score	support
0.0	0.954	0.948	0.951	504
1.0	0.976	0.988	0.982	496
2.0	0.966	0.973	0.969	440
3.0	0.932	0.948	0.940	502
4.0	0.968	0.985	0.976	454
5.0	0.990	0.990	0.990	421
6.0	0.983	0.991	0.987	456
7.0	0.939	0.871	0.904	410
8.0	0.985	0.987	0.986	477
9.0	0.897	0.952	0.924	477
10.0	0.991	0.996	0.994	465
11.0	0.970	0.951	0.960	472
12.0	0.955	0.971	0.963	480
13.0	0.988	0.991	0.990	430
14.0	1.000	0.998	0.999	423
15.0	0.992	0.990	0.991	503
16.0	0.976	0.965	0.970	459
17.0	0.975	0.981	0.978	432
18.0	0.931	0.954	0.942	478

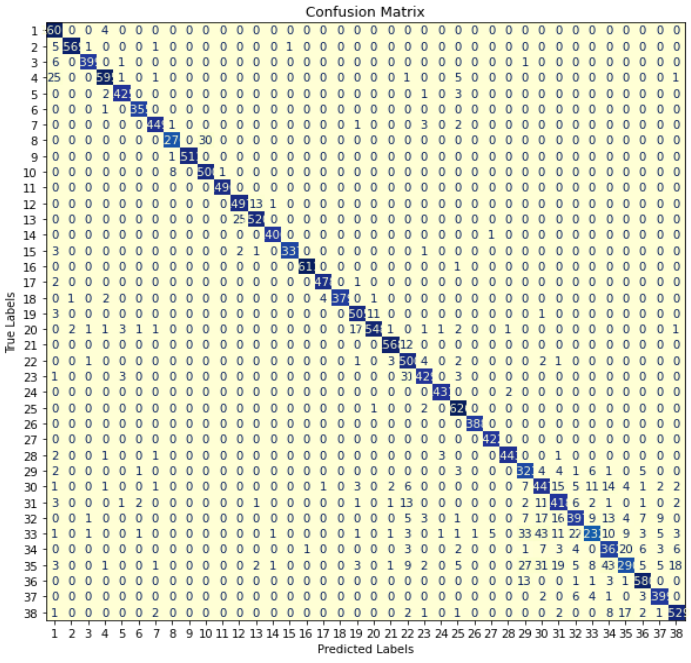

Figure 25 - InceptionV3 freezed confusion matrix

19.0	0.950	0.926	0.938	497
20.0	0.973	0.963	0.968	485
21.0	0.910	0.938	0.924	485
22.0	0.936	0.925	0.931	456
23.0	0.969	0.982	0.975	445
24.0	0.954	0.954	0.954	505
25.0	0.989	1.000	0.994	434
26.0	0.987	0.991	0.989	444
27.0	0.989	0.961	0.974	456
28.0	0.890	0.918	0.904	425
29.0	0.810	0.775	0.792	480
30.0	0.861	0.873	0.867	463
31.0	0.905	0.849	0.876	470
32.0	0.806	0.761	0.783	436
33.0	0.826	0.841	0.834	435
34.0	0.764	0.770	0.767	457
35.0	0.943	0.951	0.947	490
36.0	0.912	0.944	0.928	448
37.0	0.931	0.919	0.925	481
accuracy			0.939	17571
macro avg	0.939	0.939	0.939	17571
weighted avg	0.939	0.939	0.939	17571

```
Train Accuracy  : 95.1 %
Test Accuracy   : 93.9 %
Precision Score : 93.9 %
Recall Score    : 93.9 %
```

The confusion matrix described that there was very little miss-classification. The model got a little confused with some classes (29 to 34). We can note that these are related to the same species (Tomato) but with different diseases (Early_blight, Late_blight, Leaf_Mold, etc.), which could explain the miss-classification. The model had more troubles in the classification when the species was the same, but diseases were different.

Unfreeze some layers

In order to increase our results, we decided to unfreeze some layers of the inception model. As a first step we unfreezed the last 12 layers because we decided to include the last convolution layer in the unfreezing. This translated into adding around 0.4 million parameters to the training (reaching 3,036,198 parameters in total). Then, we unfreezed from the 23rd last layer, because the last 'package' of convolution layers just started there (please refer also to the summary of the base_model in the notebook). This translated into adding around 2.8 million parameters to the training (reaching 5,462,566 parameters in total).

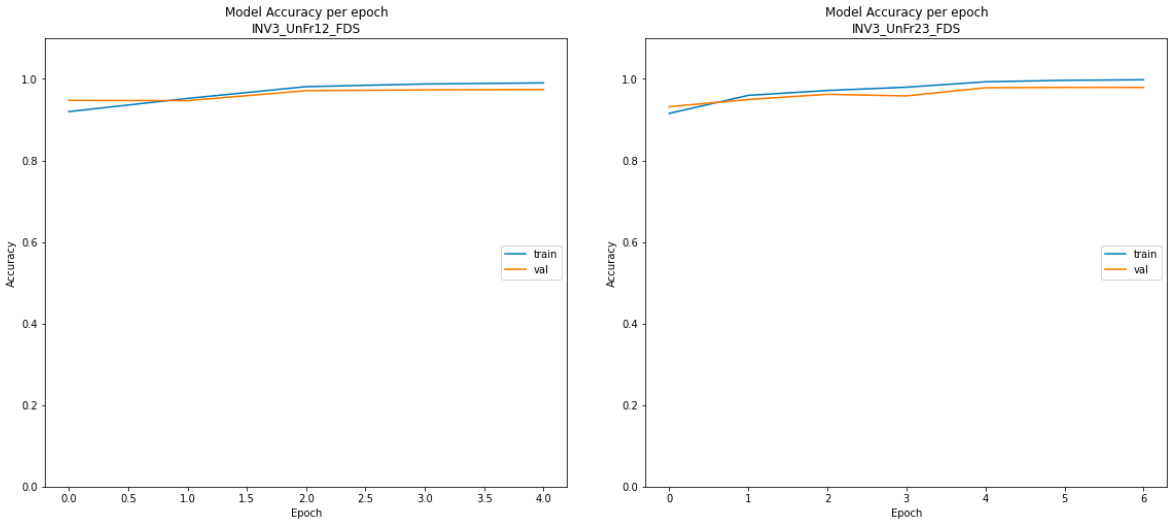

Figure 26 - InceptionV3 unfreezed loss and accuracy curves

```
Test Accuracy    : 97.3 %          Test Accuracy    : 97.9 %
```

We saw that test accuracy increased from 94% to 97.3% on the first unfreezed model (12 layers), and we kept the overfitting very low. Regarding the second unfreezed model (23 layers), we increased the test accuracy to 97.9%. After we unfreezed the model, we observed that it could be a good idea to go further in the number of layers unfreezed (maybe 35). Indeed, even if the number of layers we unfreezed looked high, the number of additional trainable parameters was still low compared to other models.

Conclusion Inception V3

We can conclude that our model based on InceptionV3 didn't reach the accuracy of previous models. One advantage of the inception V3 is the low memory and low resources used, and we indeed observed that the size of the model is lower (Inception V3 : 113Mo < Resnet 50 : 120 Mo and CNN 290Mo). It could have been interesting to compare the speed of simulation, unfortunately the instability of Google Collab made it impossible (even if we have the feeling that inception V3 goes faster).

6 Final conclusions

To identify plant species and plant diseases, we developed a few deep learning models. The first CNN models (CNN) obtained very good results, especially for a naive model. Then we moved toward transfer learning reaching higher accuracy, which is not surprising. The best results were achieved with two models based respectively on Resnet50 and VGG16. More surprisingly, we found out that a third transfer learning model based on InceptionV3 obtained lower accuracy. This could be linked to the fact that the main goal of the inceptionV3 model is to obtain good results with a lower usage of memory and resources.

For each learning transfer model, we experimented the unfreeze of some layers, with the aim of increasing the model's capacity of detecting a pattern. To choose which layers unfreeze, we looked at the structure of the pre-trained model and decided to unfreeze the last n layers (with n specific to the model) in a way that the unfreezing was including complete blocks of convolution. Indeed, this step increased the accuracy of the models, bringing the ones based on ResNet50 and VGG16 close to 99.5% accuracy, and the one based on InceptionV3 close to 98%.

Given the level of results achieved by the unfreezed models, we decided to investigate the possible reasons why. These could come from the quality of the data we had. The dataset was quite balanced, and apparently completed with an augmentation obtained by modification of images (size increasing, flipping, …). In addition, the images were already quite clean: photos were without any background, without other features around and with a good light. This could have led to an excessive homogeneity which made the task for our models much simpler.

Even in this context of very good results we remarked that all models had some difficulties with classifying certain plant diseases, while the classification of the species was almost always good. In particular, looking at the Confusion Matrices and the Classification Reports, we saw that in the case of tomatoes and corn identifying diseases inside the species was more difficult. All models struggled in classifying class 7 and 9 (both related to corn's leaves) and the classes from 28 to 37 (the ones related to tomato's leaves). The Confusion Matrices showed that the models sometimes inverted Class 7 and 9, while for the tomato's leaves there was more dispersion of misclassification among the various classes.

This could be explained by the similarity inside the species across the different diseases which affect a species: same colors, same pattern, etc. In addition, the first CNN model had trouble to classify strawberry and raspberry given their similarity.

An additional parameter which could help us in comparing our models would have been the time requested for the training. Unfortunately, the instability of the Google Collab didn't allow us to investigate that point. The exact same training in different moments took very different time to be completed for apparently no reason. This was really disappointing as the required time for training could have been an interesting aspect when comparing the various models and evaluating the advantages or not of unfreeze models.

Finally, we thought about possible directions for improving the results of our project.

First, we could increase the size of the dataset. We can have more species and more diseases; indeed, some species were only present as healthy. We can also add pictures with a real background. These changes in the dataset would likely lower the accuracy of the models but would make them more “operational”.

Second, we could develop a model with two separate outputs: species and diseases. This could improve the ability to distinguish diseases.

Third, we could investigate the interpretability of the model, to have a deeper understanding of what’s driving the performances. At the end of the project we actually started looking into this by using the GRAD-Cam Interpretability package.

Appendix 1 : Understanding model summary

When building a model, we can get a summary as shown in the image below for our CNN model:

```
model = Sequential()

model.add(Conv2D(32,(7,7),activation="relu",padding="same",in
put_shape=(img_height,img_width,3)))

model.add(Conv2D(32,(7,7),activation="relu",padding="same"))

model.add(MaxPooling2D(3,3))


model.add(Conv2D(64,(5,5),activation="relu",padding="same"))
model.add(Conv2D(64,(5,5),activation="relu",padding="same"))

model.add(MaxPooling2D(3,3))


model.add(Conv2D(128,(3,3),activation="relu",padding="same"))
model.add(Conv2D(128,(3,3),activation="relu",padding="same"))

model.add(MaxPooling2D(2,2))


model.add(Conv2D(256,(3,3),activation="relu",padding="same"))
model.add(Conv2D(256,(3,3),activation="relu",padding="same"))

model.add(Conv2D(512,(3,3),activation="relu",padding="same"))
model.add(Conv2D(512,(3,3),activation="relu",padding="same"))

model.add(Flatten()) model.add(Dense(1568,activation="relu"))

model.add(Dense(512,activation="relu"))

model.add(Dropout(0.5))

model.add(Dense(38,activation="softmax"))
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 256, 256, 32)      4736

conv2d_1 (Conv2D)            (None, 256, 256, 32)      50208

max_pooling2d                (None, 85, 85, 32)        0
(MaxPooling2D)

conv2d_2 (Conv2D)            (None, 85, 85, 64)        51264
conv2d_3 (Conv2D)            (None, 85, 85, 64)        102464

max_pooling2d_1              (None, 28, 28, 64)        0
(MaxPooling2D)

conv2d_4 (Conv2D)            (None, 28, 28, 128)       73856
conv2d_5 (Conv2D)            (None, 28, 28, 128)       147584

max_pooling2d_2              (None, 9, 9, 128)         0
(MaxPooling2D)

conv2d_6 (Conv2D)            (None, 9, 9, 256)         295168
conv2d_7 (Conv2D)            (None, 9, 9, 256)         590080

conv2d_8 (Conv2D)            (None, 9, 9, 512)         1180160
conv2d_9 (Conv2D)            (None, 9, 9, 512)         2359808

flatten (Flatten)            (None, 41472)             0

dense (Dense)                (None, 512)               21234176

dropout (Dropout)            (None, 512)               0

dense_1 (Dense)              (None, 38)                19494

=================================================================
Total params: 26,108,998
Trainable params: 26,108,998
Non-trainable params:  0
```

Figure 27 - CNN model summary

For each layer, we get information about the output and the number of parameters. Here we try to explain how the output shapes and the number of parameters are calculated, with some examples.

Output shape :

- Conv2D layers:

For our first layer, the input shape is (256, 256, 3). The output shape we get from the input layer is: (None, 256, 256, 32), let's see why, starting with padding value.

When we use no padding, the result of convolution filtering is a "shrinked" image (we lose the borders):

[(n x n) image] * [(f x f) filter] —> [(n – f + 1) x (n – f + 1) image]

where **n** is our image size, **f** our filter size and ***** represents convolution.

In our model, we chose padding = "same", to get the same size after filtering, so:

[(n + 2p) x (n + 2p) image] * [(f x f) filter] —> [(n x n) image]

Where **p** is the padding. Therefore: (n + 2p) – f + 1 = n, which gives p = (f – 1) / 2. In our case:

p = (7-1) / 2 = 3

The layers output size is calculated with the following equation:

o = (w - f + 2p)/s + 1

where **o** is the output height/width, **w** is the input height/width, **f** is the filter size, **p** is the padding and **s** is the stride size.

We didn't specify strides in our model, so we have the default size (1,1). So, our output is:

Output height/width = (256 – 7 + 2*3) / 1 + 1 = 256

- MaxPooling2D layers:

The output size after a max pooling layer depends on the size of the pool. In our case, we have (3, 3), so the shape of the data becomes (256/3, 256/3) => (85, 85)

- Flatten layers:

The flatten layer simply flattens the input data, concatenating all existing parameters. In our case, we get a vector with the following length: 9 * 9 * 512 = 41472

Number of parameters:

This value gives us the number of parameters trained for each layer.

- Conv2D Layers

In our model, we have three Conv2D layers, and the calculation of the parameters for these layers follows the same principle, as noted in the formula below. The number 1 denotes the bias that is associated with each filter that we're learning.

Param # = output_channel_number * (input_channel_number * f_height * f_width + 1)

If we apply this formula to our first Conv2D layer (which has an (256, 256, 3) input and (256, 256, 32) output) and 7x7 filter, we get:

Param # = 32 * (3 * 7 * 7 + 1) = 4736

- MaxPooling2D Layers

All MaxPooling2D layers have 0 parameters (this layer doesn't learn anything)

- Flatten Layers

The Flattern layer doesn't learn anything, and thus the number of parameters is 0.

- Dense Layers

We have two Dense layers in our model. The calculation of the parameter numbers uses the following formula.

Param # = output_channel_number * (input_channel_number + biases)

Applying this formula to our model for the first Dense layer:

Param # = 512 * (41472 + 1) = 21,234,176