Master Degree in Artificial Intelligence and Data Engineering

Multimedia Information Retrieval and Computer Vision

Plant Leaves Search Engine

Tommaso Amarante, Lorenzo Bianchi, Iacopo Bicchierini, Edoardo Morucci

Github repository: https://github.com/EdoardoMorucci/Plant-Leaves-Search-Engine—MIRCV

Contents

Chapter 1

Introduction

Plant Leaves Search Engine is a system that can recognise the species given a leaf image by the user. Our engine can recognise leaves from 14 different plants: Apple, Blueberry, Cherry, Corn, Grape, Orange, Peach, Pepper, Potato, Raspberry, Soy Bean, Squash, Strawberry and Tomato. In order to do that we exploit a fine-tuned version of **DenseNet121**, and for the retrieval we use a Vantage Point Tree index.

We decided to extract the features using the finetuned model, to obtain the correct result we deleted the classification layers at the end of the neural network. We exploit these feature to build the VPT index, thanks to the features we retrieved the k nearest neighbours using a distance measure. We compared the query time between the usage of a VPT index and a brute force approach. Finally we have evaluated the pretrained model and the finetuned one using the **MaP** metric.

Chapter 2

DenseNet

The Convolutional Neural Network used as base network for our model is DenseNet.

DenseNet is a type of convolutional neural network that utilises dense connections between layers, through Dense Blocks, where we connect all layers (with matching feature-map sizes) directly with each other. To preserve the feed-forward nature, each layer obtains additional inputs from all preceding layers and passes on its own feature-maps to all subsequent layers. We can say that each layer is receiving a "collective knowledge" from all preceding layers, unlike the traditional CNN where each layer receives informations only from the precedent layer.

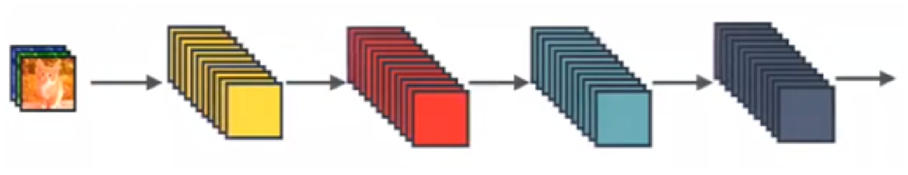


Figure 2.1: Standard CNN approach

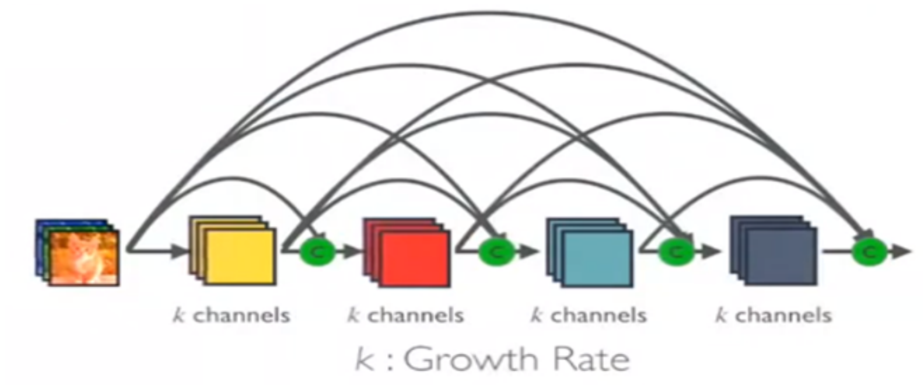


Figure 2.2: DenseNet approach

Chapter 3

Dataset

We have to work with two different datasets. The first one is made of 72k images of leaves divided into 14 classes. Each class represents a different plant. The second dataset, the distractor, is composed of 25k images that represents a sort of noise and is useful when a user insert a non leaf image, the system will return mostly images contained in the distractor.

All the leaves are on a black screen and have different rotations and different zooms.

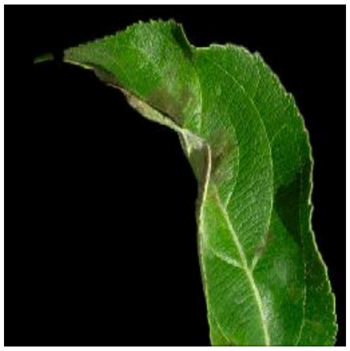


Figure 3.1: Example of a leaf in the dataset

Figure 3.2: Example of a distractor image

Chapter 4

VPT Index

The Vantage Point Tree (VPT) index is an exact similarity searching method that exploits the ball partitioning technique dividing recursively the dataset of interest. At each step of the tree's construction, we choose a pivot from the dataset and then we compute the median m respect to the pivot, the median represents the radius that divides the dataset into two equal parts, half points will go inside the ball and half outside. Since the partitioning is recursive, we repeat the same process to the two subsets created at the previous step, as shown in figure.

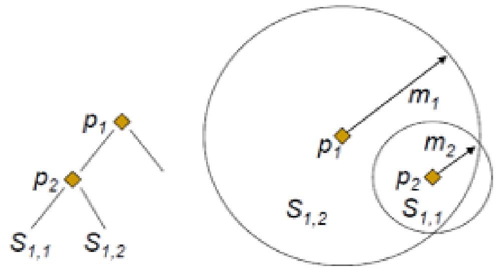


Figure 4.1: Ball partitioning method example.

The VPT is a balanced binary tree that has fixed structure, the stopping condition in the creation is the number of points in a subset to split, if it is under a threshold, we put all the remaining points into one leaf. An example of a small VPT is shown below.

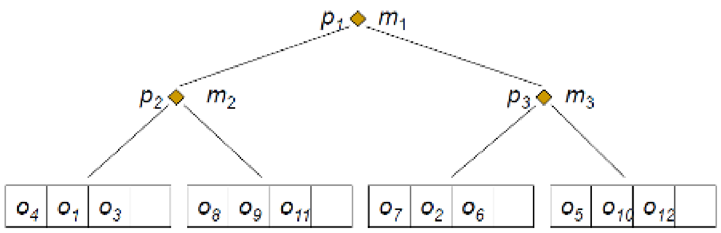


Figure 4.2: Vantage Point Tree example.

Once the tree is created, we can exploit it to apply a k_{NN} **query**. Since the radius of the query is variable, we need two vectors in memory:

- $d_{NN}[k]$ that stores the distances between the query and the nearest objects. This vector is initialized at d_{MAX} and is ordered based on distances.
- $k_{NN}[k]$ that stores the nearest objects, initially $k_{NN} = null$.

Given a query we start searching the tree starting from the root node, for each node we have three cases:

- if $d(q, p_i) \leq d_{NN}$: add $d(q, p_i)$ to d_{NN} if there is space available or $d(q, p_i)$ is lower than the higher value inside d_{NN}, consequently add on k_{NN} and then reorder the two arrays based on distances inside d_{NN}. Note that if d_{NN} is full we don't add the object switching with the farthest from the query, the same for k_{NN}.
- if $d(q, p_i) - d_{NN} \leq m_i$: search in left subtree
- if $d(q, p_i) + d_{NN} \geq m_i$: search in right subtree

Once a leaf is reached check if one or more objects inside the leaf are closer to the query respect to the objects I've already selected, in this case we switch the leaf's objects with the farthest in d_{NN} and consequently k_{NN}.

4.1 Our implementation

We decided to use three classes: **Object**, **Node** and **VantagePointTree**.

The **Object** is the data structure that contains the informations related to one image:

- $Features$ is the numpy vector extracted using the CNN model
- Id is the identifier used to retrieve the image from the dataframe

It is like the Object o_1 or o_2 but also the pivot p_1 present in image 4.2.

The **Node** is the data structure that will act as the building block for the tree, we differentiate as intern node and leaf node:

- $InternNode$: has the Pivot (that is an Object) and the Median that is the distance that allow to split the received Objects in two sets of equal number of objects.
- $LeafNode$: has a list of Objects (in figure 4.2 we have 4).

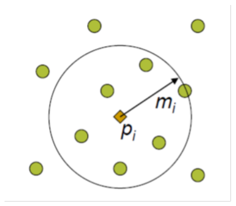


Figure 4.3: Pivot example.

The class Node has two important methods from-dict and to-dict used to serialize and to recreate the node.

The **VantagePointTree** class is used to create the tree structure and perform the kNN search. The attributes stored are:

- Root node
- Distance measure
- Mode pivot choice
- Size of tree

The method $insert$ is recursive and it is used to create the tree structure, a list of objects is passed as parameter. In particular, the insert method finds a pivot, divides the objects into two sets using the median from the pivot and call itself twice passing the sets. Since the creation of the tree could be heavy and takes some time, we add a functionality to store the tree in a dictionary and save it on a disk, in this way we can easily take the dictionary from disk and reconstruct the tree without recomputing the distances.

The **parameters** we can choose when a tree is created are:

- The measure used to compute distances (Euclidean or Manhattan)
- How pivots are selected (random or using outlier heuristic)
- A Boolean to specify if the tree are reconstructed from disk

4.2 Performances of index

To evaluate the performances of the Vantage Point Tree index we have compared the query time using a brute force approach (sequential scan) and using our index.

We performed 50 queries searching for the 5 most similar features' vector, a graph that shows the differences in terms of query time is in the figure below.

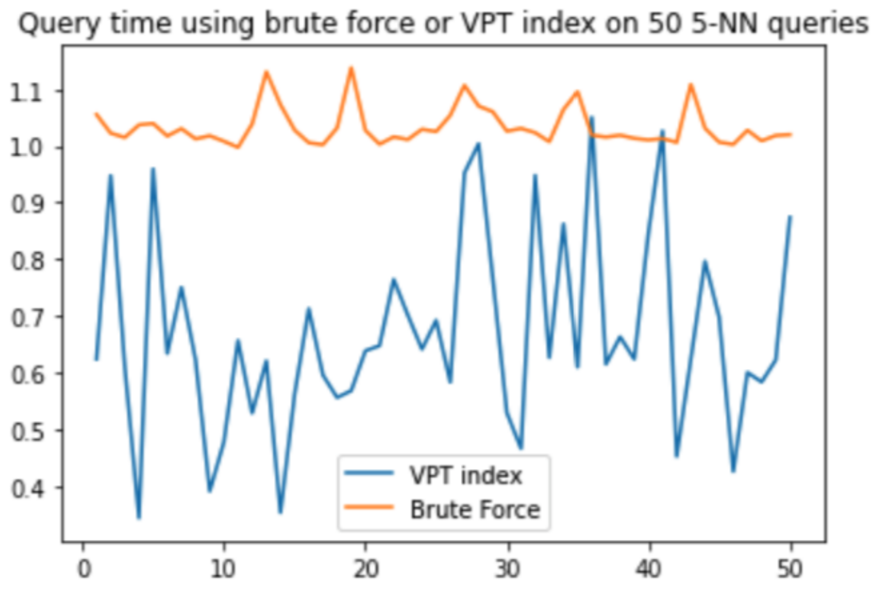


Figure 4.4: Query time using brute force and VPT index.

Considering this test on 50 queries the mean query time is **reduced by 35%** and the distances computed by 49,4%. The choice of using an index is then very important in a real word application where the queries are a lot and the response time is crucial.

	Mean query time	Mean distance computed
Brute force	1,03 sec	97034
VPT index	0,66 sec	49092
Gain	0,37 sec	47.942

Chapter 5

FineTuning

As explained in the previous chapters, the Convolutional Neural Network used as base network is **DenseNet**, and we want to build a fine-tuned model given that network. To prevent huge gradients coming from the newly initialized layers from destroying the weights in the pretrained layers we will initially freeze the layers of the base network and train only the new layers added to build the plant leaves classifier. Only after this preliminary step we are going to fine tune the weights of **DenseNet** (but only of the top layers of the CNN). Given this, our objectives are to build two distinct models:

- One with a **Multilayer Perceptron Network** built over the pretrained **DenseNet** network.
- One as the network of the previous point but with the last **two** blocks of **DenseNet** fine-tuned.

To build our two models we removed the fully-connected layer on top and we added **Global Average Pooling layer**, an **Hidden Classifier** with 256 neurons and an **Output layer** with 14 neurons (one for each class we want to classify) and **softmax** as activation function.

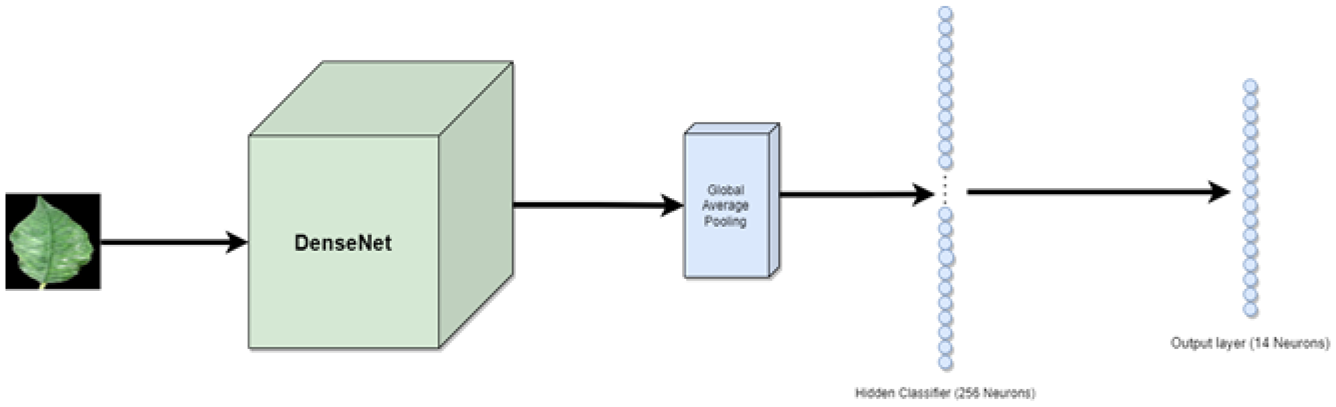
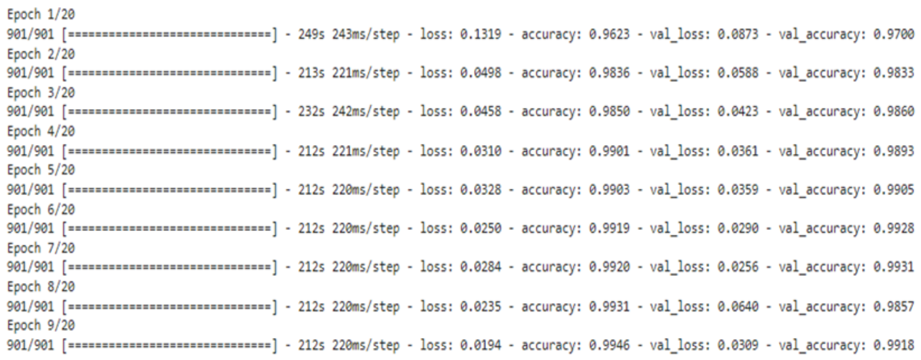


Figure 5.1

The **Dataset** was divided in **Train** set, **Validation** set and **Test** set respectively with a percentage of **80%**, **10%** and **10%**. Since our dataset is large (**72k images**) and, as we will explain later on, we didn't have **overfitting issues** (also thanks to the the **early stopping** technique that we will discuss later on), we did not consider it necessary to use data augmentation techniques. The size of the input image (as required by **DenseNet**) is **224x224** pixels. The **Batch size** used during both the training was 64. After some tests we saw that a lower Batch size leads to increase significantly the training time and to a slightly decrease of the accuracy in each epoch. Instead greater Batch size leads to crash of the Google Colab runtime environment due to RAM problem. The value of 64 was a compromise between this aspects and we think that if we had more computational power, a greater value would lead to slightly better results regarding the accuracy of the model and the training time. The optimizer used was **Adam**, the value of the **Learning rate** was 0.005 and we initially set an high number of epochs (20) but we used the **early stopping** technique to prevent overfitting. The **patience** parameter for the early stopping was 2, so the training is stopped when for two consecutive epochs the accuracy of the validation doesn't improve.

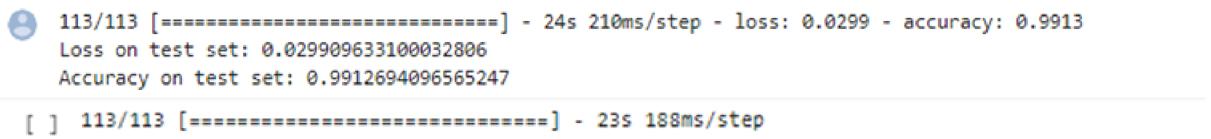
The results of the training of the first model with this configuration are the following:



```
Epoch 1/20
901/901 [==============================] - 249s 243ms/step - loss: 0.1319 - accuracy: 0.9623 - val_loss: 0.0873 - val_accuracy: 0.9700
Epoch 2/20
901/901 [==============================] - 213s 221ms/step - loss: 0.0498 - accuracy: 0.9836 - val_loss: 0.0588 - val_accuracy: 0.9833
Epoch 3/20
901/901 [==============================] - 232s 242ms/step - loss: 0.0458 - accuracy: 0.9850 - val_loss: 0.0423 - val_accuracy: 0.9860
Epoch 4/20
901/901 [==============================] - 212s 221ms/step - loss: 0.0310 - accuracy: 0.9901 - val_loss: 0.0361 - val_accuracy: 0.9893
Epoch 5/20
901/901 [==============================] - 212s 220ms/step - loss: 0.0328 - accuracy: 0.9903 - val_loss: 0.0359 - val_accuracy: 0.9905
Epoch 6/20
901/901 [==============================] - 212s 220ms/step - loss: 0.0250 - accuracy: 0.9919 - val_loss: 0.0290 - val_accuracy: 0.9928
Epoch 7/20
901/901 [==============================] - 212s 220ms/step - loss: 0.0284 - accuracy: 0.9920 - val_loss: 0.0256 - val_accuracy: 0.9931
Epoch 8/20
901/901 [==============================] - 212s 220ms/step - loss: 0.0235 - accuracy: 0.9931 - val_loss: 0.0640 - val_accuracy: 0.9857
Epoch 9/20
901/901 [==============================] - 212s 220ms/step - loss: 0.0194 - accuracy: 0.9946 - val_loss: 0.0309 - val_accuracy: 0.9918
```

Figure 5.2

The results on the test set are:



```
113/113 [==============================] - 24s 210ms/step - loss: 0.0299 - accuracy: 0.9913
Loss on test set: 0.029909633100032806
Accuracy on test set: 0.9912694096565247
```

```
[ ]  113/113 [==============================] - 23s 188ms/step
```

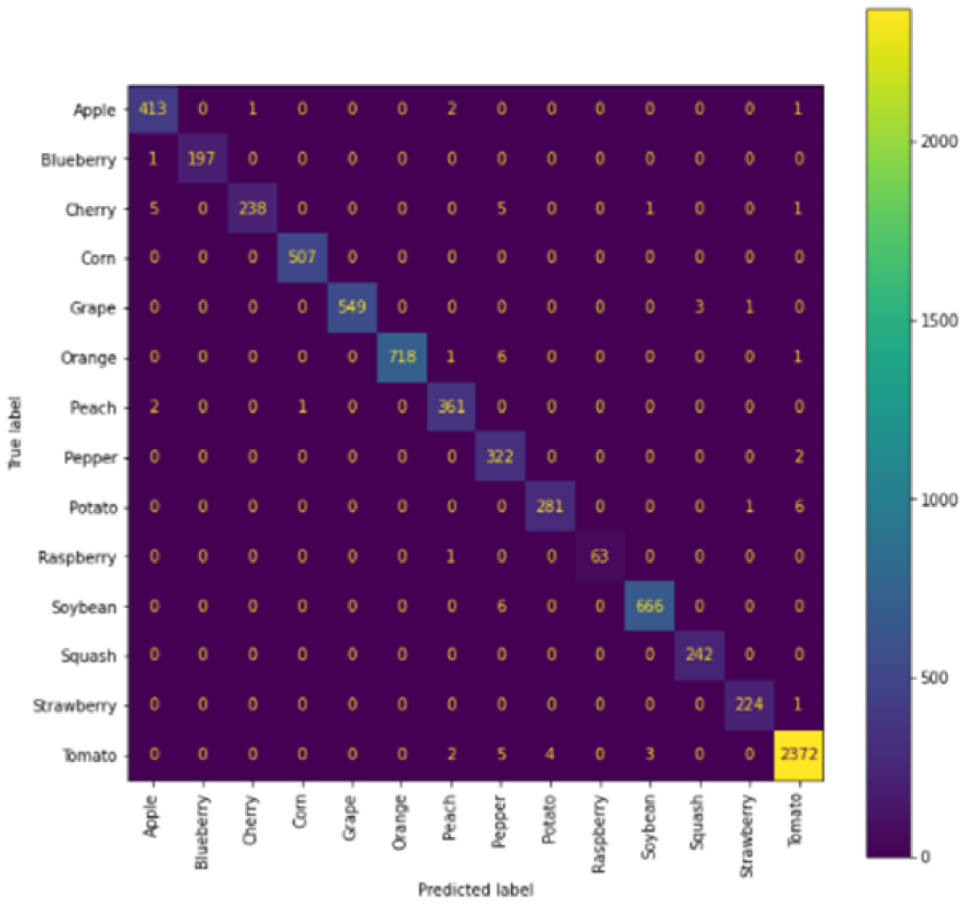
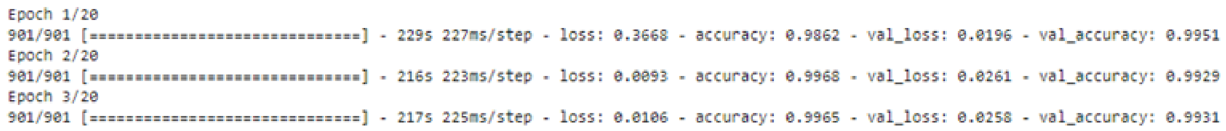


Figure 5.3

After **7** epochs we get the best values for the validation loss and the validation accuracy and later on we're going to load the weights of this epoch using the checkpoint. The value of accuracy on the test set of **0.9913** is incredibly high, and even with the fine-tuning it isn't easy to improve this value. The **Confusion Matrix** is homogenous and the model as no particular difficult to recognise any particular type of leaf compared to others. Before getting this result we tried different values for the **learning rate**. Higher value lead us to lower training time but lower performance on the accuracy of the model, and higher values lead to training time longer but we no improvement on the accuracy, so we found that 0.005 was a correct value. To **fine-tune** the network and so to build the second model we unfrozen the last two blocks of **DenseNet** (except the Batch Normalization layers) before the training, and we re-perform training on the training set with the same configuration

(**Adam as optimizer, learning rate = 0.005, Early stopping**). We decided to **unfreeze only the last two blocks of Densenet** because the time requested to train the model with more free layers increase exponentially and the results obtained with only two layers unfrozen are satisfying.

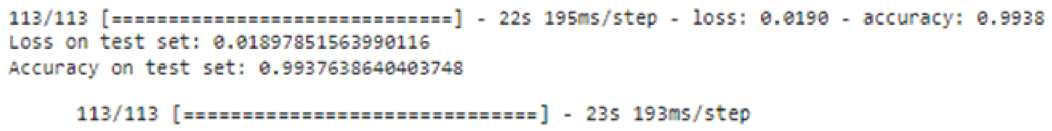
The results of the training are the following:



```
Epoch 1/20
901/901 [==============================] - 229s 227ms/step - loss: 0.3668 - accuracy: 0.9862 - val_loss: 0.0196 - val_accuracy: 0.9951
Epoch 2/20
901/901 [==============================] - 216s 223ms/step - loss: 0.0093 - accuracy: 0.9968 - val_loss: 0.0261 - val_accuracy: 0.9929
Epoch 3/20
901/901 [==============================] - 217s 225ms/step - loss: 0.0106 - accuracy: 0.9965 - val_loss: 0.0258 - val_accuracy: 0.9931
```

Figure 5.4

On the test set we have the following results:



```
113/113 [==============================] - 22s 195ms/step - loss: 0.0190 - accuracy: 0.9938
Loss on test set: 0.01897851563990116
Accuracy on test set: 0.9937638640403748
```

```
      113/113 [==============================] - 23s 193ms/step
```

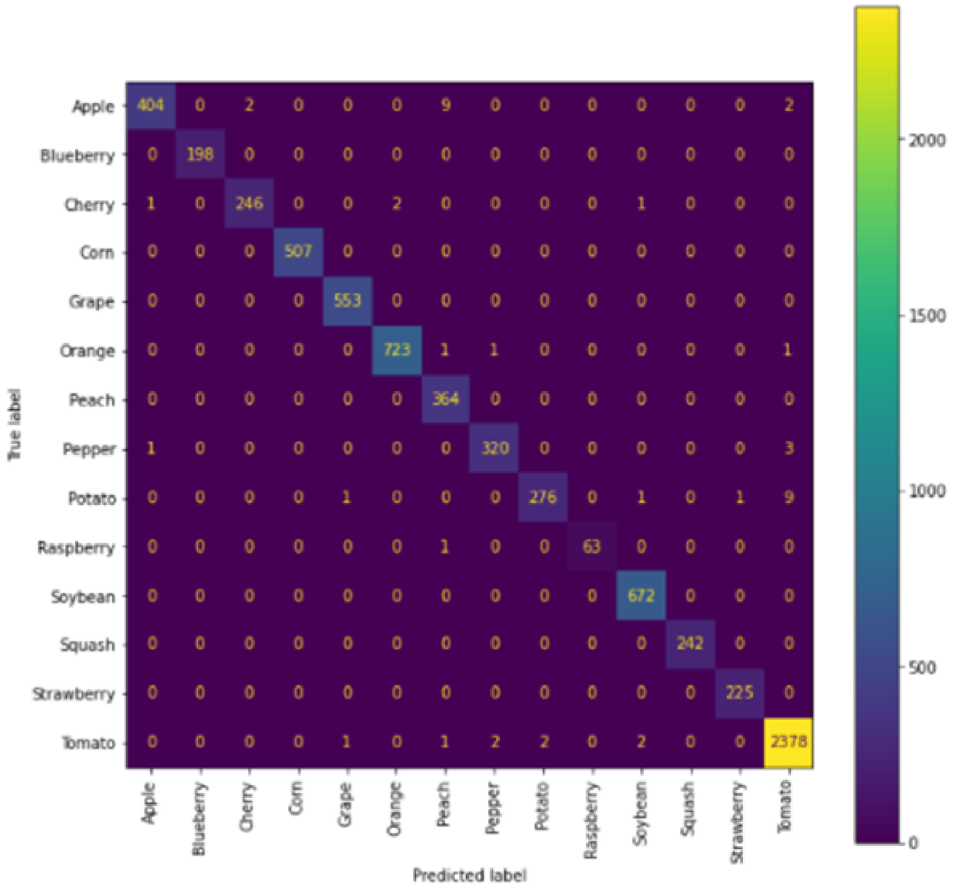


Figure 5.5

The training process of the fine-tuned model compared to the other one is explained in this graph:

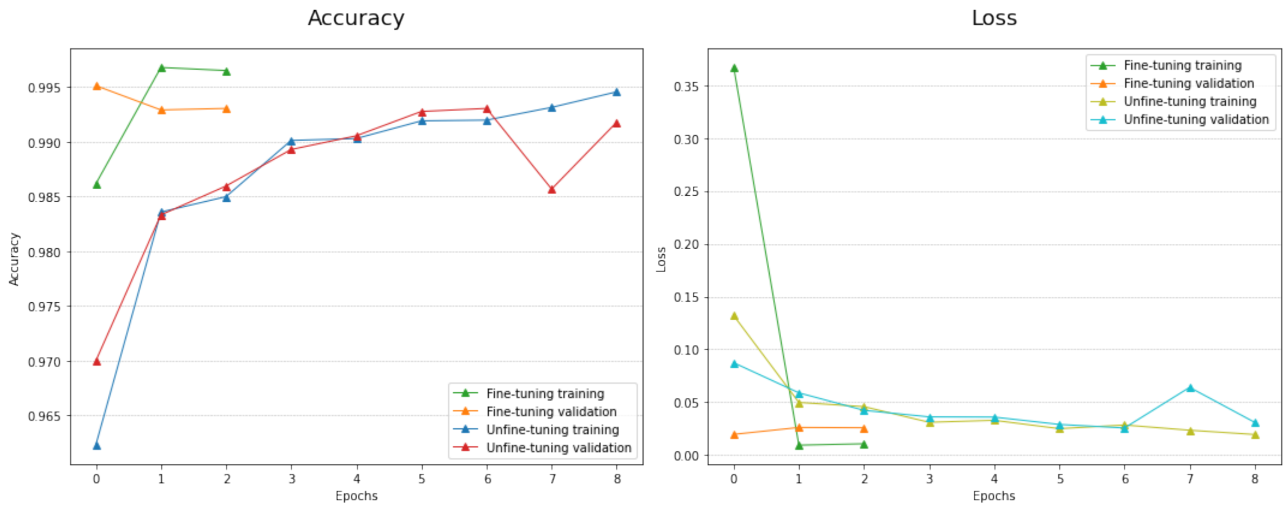


Figure 5.6

After just **one** epoch we get the best values for the validation loss and the validation accuracy, so the **early stoppings** prevent to continue the training with the risk of **overfitting** on the training set. As we can see the accuracy with the fine tuning improves slightly and we obtain the best accuracy in the first epoch. This could be explainable due to the fact that the accuracy reached with the first model was very close to the perfection (**0.9912**), and this values is very difficult to improve, and too much weights assessments on the weights during the training phase would lead to overfitting the model to the training set. With the fine-tuned version we improved this value to **0.993763** and the loss on the test set drop from **0.0299** to **0.01897**, and so we obtained an almost perfect plant leaves classifier.

Chapter 6

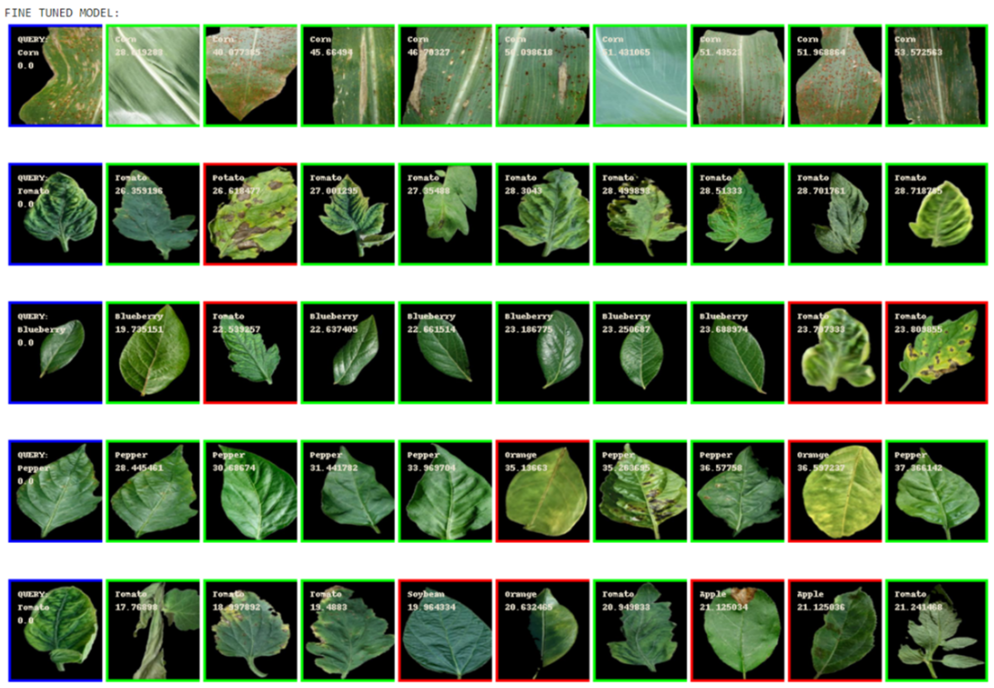
Performance Evaluation

Plant Leaves Search Engine operates with the support of **DenseNet Neural Network** and the **Vantage Point Tree Index**. The first one is useful to assure good results in term of accuracy and the second one to give the results as fast as possible. In this paragraph the performances of these two tools will be evaluated and discussed.

6.1 Model

In order to benchmark the **pretrained** DenseNet with the **finetuned** one, we used the **Mean Average Precision**, computed as shown to us during the lessons. This metric provides a compact view of the quality of retrieval across several queries, in other terms It says how good is our model to give as result, images of leaves of the same plant species of the query. To compute the mAP we compute the Average Precision for each queries and we make a simple mean. We tested the two models using the features extracted by the **test set**, composed by images never seen before by both models, and also using a set composed by **noise images** and **test set**, in order to make the things as difficult as possible for the models. The execution of the function **evaluate_knn** that implements the computation of the mAP can be done in two modes:

- **Verbose**: using a simple output, it shows at the most 10 images from the k nearest neighbors of the query image. The image of the query will be outlined in blue, if the result is relevant in green and red if it is not. For each query the AP will be displayed and mAP will be shown too.
- **Non Verbose**: useful when we make performance evaluation, to not flood the output.



```
FINE TUNED MODEL:

AP@ 10 for each Query:  [1.0, 0.87826278659612, 0.873639455782313, 0.9415178571428572, 0.8857142857142857]
mAP@ 10 : 0.9158268770471153
```

Figure 6.1: Output of evaluate_knn with verbose=True

To actually compare the models we computed the mAP using a number of queries (30), and we also computed this for each k from 1 to 50.

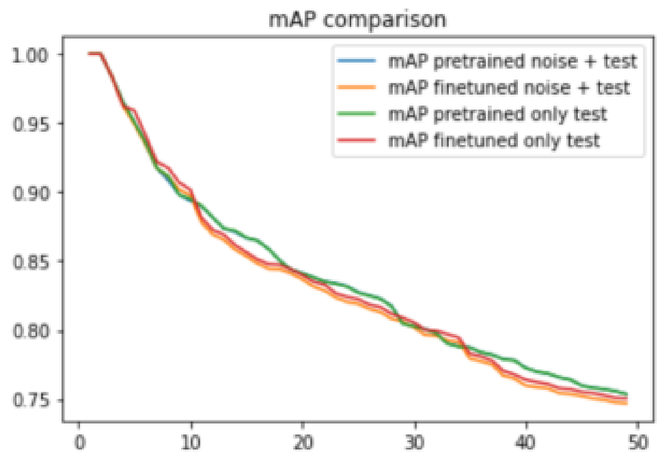


Figure 6.2

The **trend of the mAP**, as it is intuitive, decreases as k increases, furthermore we see how the mAP values for the two features are almost similar. This result is consistent with those seen during the fine tuning process which showed us how the initial model was already able to make a good classification and that the fine tuning only slightly increased the accuracy. Moreover it's worth highlighting how the result using the test and noise is similar with respect to the one made using only the test, this result shows how good our model is at predicting relevant results, also with the presence of a very large number (25k) of noise images with a test set of 7k images.

6.2 Vantage Point Tree

We benchmarked the **Vantage Point Tree Index** using two parameters, the **Distance computation** method and the **Pivot choice**. In order to do that we used the **Average Query Time**, the time in which the index gives the result of a kNN query using a set of queries and making the average; and the **Distance Computations Percentage**, that shows how much distances computations are saved thanks to the index. The **Build Time**, the time required to create the index, will be briefly mentioned too.

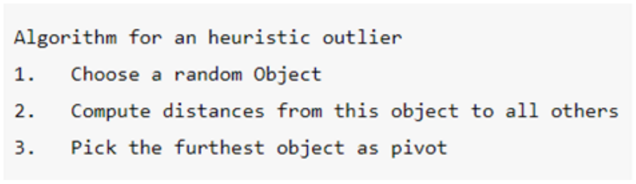
6.2.1 Parameters

The **Distance computation** methods used are 2:

- **Euclidean**: $d(x,y) = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2}$
- **Manhattan**: $d(x,y) = \sum_{i=1}^{n}|x_i - y_i|$

And the **Pivot Choice** available:

- **Random**
- **Heuristic Outlier**



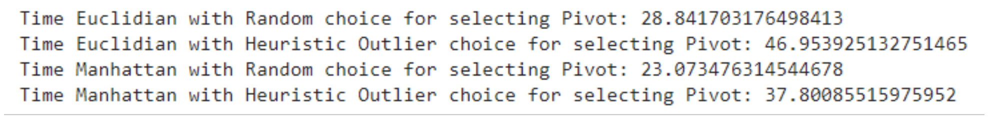
```
Algorithm for an heuristic outlier
1.   Choose a random Object
2.   Compute distances from this object to all others
3.   Pick the furthest object as pivot
```

Figure 6.3

We tried also to find the **Exact Outlier** that has the max average distance from other points, but it has proved to be a non-viable path since computational time, it was predictable, became exponential. In the attempt of finding other parameters which might be related to the Vantage Point Tree, we tried considering the **number of objects needed to stop the insertion** of an other level of the tree and creation of leaf. The number initially chosen was 4 as we had seen during the lessons and sliding possible values from 1 to 1000, the only performance that changed was the build time, and it was expected because the depth of the tree is shortened, but the other performances remain consistent with the initial results. Therefore we decided to not implement nor show the plot because it wasn't remarkable.

6.2.2 Performance measures

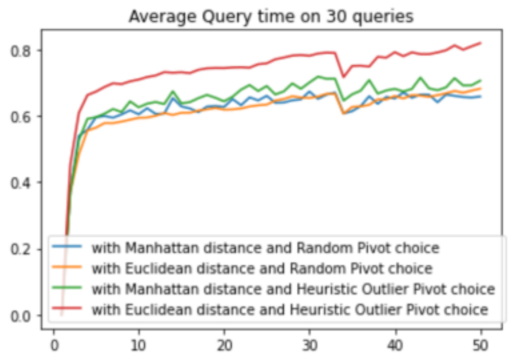
The **build time** is more or less the same, the reason why using **manhattan distance** is slightly faster can be the simpleness of the computation with respect to the **euclidean distance** that includes a square root. The same occurs with the Pivot choice that in **random** is faster than **heuristic outlier**.



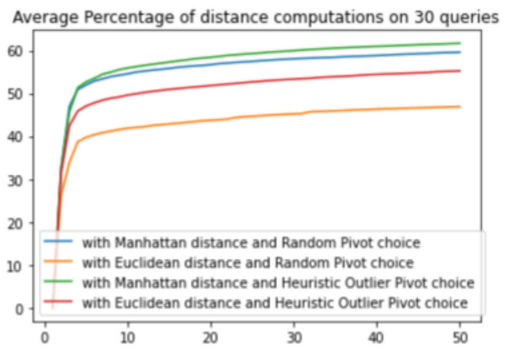
```
Time Euclidian with Random choice for selecting Pivot: 28.841703176498413
Time Euclidian with Heuristic Outlier choice for selecting Pivot: 46.953925132751465
Time Manhattan with Random choice for selecting Pivot: 23.073476314544678
Time Manhattan with Heuristic Outlier choice for selecting Pivot: 37.80085515975952
```

Figure 6.4

The **average query time** varies from 0.6 to 0.8 seconds, increasing **k** the time doesn't get bigger as well because the index saves distance computations as we can see in the plot. In general we can say that using **Random** pivot choice with **Euclidean** distance computation is the best setting of parameters that offers the best query time and saves most computations.



(a)



(b)

Figure 6.5

We can observe that the choice of the Pivot not affect so much the distance computations saved with respect to the distance computation method, furthermore **Heuristic Outlier** seems to offer a less effective Vantage Point Tree Index with respect to **Random**, we expected the opposite result. It is important to mention also that a strict link between **query time** and **distance computations** can't be found, indeed the **Manhattan** solution that provides less distance computations saved, it results to make similar query time with 20% more distances computed and this is because during the query time the distances computed using Manhattan distance is faster. Overall we can say that the aim of the index is accomplished because it saves more or less half of distance computations, making the query time faster.

Chapter 7

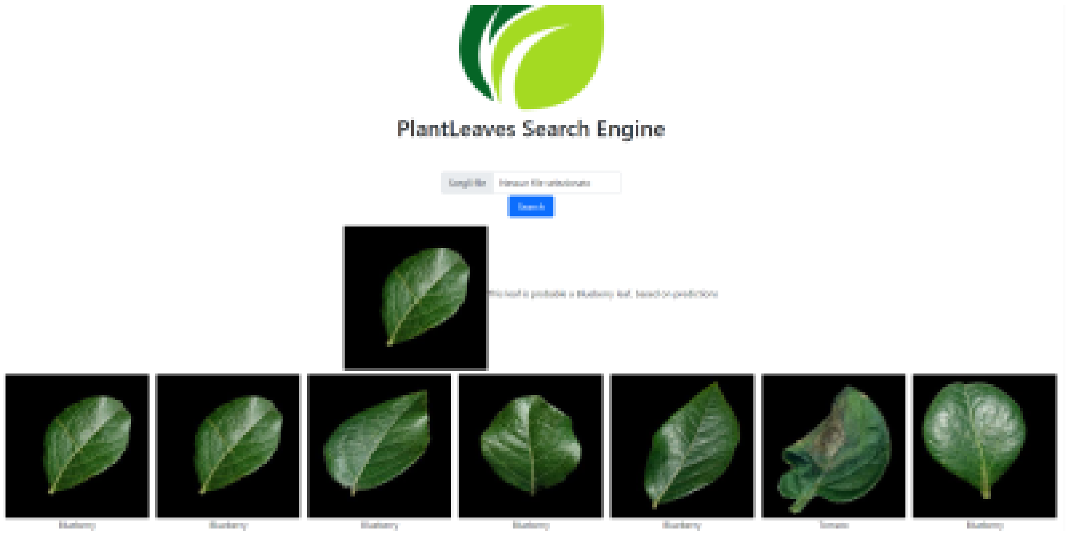
Web Interface

We decided to exploit the **Django** framework to create our web interface. Django is a web framework based on python, because of that it helped us in calling the functions that we had implemented in Google Colab. We have created a simple web page that is hosted in local host, but it can also be hosted in order to be reachable over the internet. We can upload the image that we want to use as a query, after we hit the "search" button we send a POST request to the web server. Django can handle the POST request; it saves the query image and triggers the python functions that process the image to retrieve the total result.

It is important to mention that we downloaded the entire dataset so that it is easy to visualize the 10 most similar images, we also used a Pandas DataFrame to connect each image path to the correct plant label. We need the dataframe to properly show the image label and the prediction on the query that we have uploaded. The dataframe is also useful because the search function returns a list of *ids* of the k nearest neighbours, so we used each id to index a specific row in the dataframe. We used a pregenerated VantagePointTree built with the features that we have extracted using the finetuned model, we simply have to load it from the file. After the load we have to extract the features from the query image that the user has uploaded, and we use that to explore the VPT index to obtain the list of the nearest neighbors of the query.

Django can send the list that contains the result to the frontend HTML page, there we can build the entire page to show the results of the research.

We attach a label containing the species of a plant, we also return a prediction on the query image based on the most frequent label among the results.



This leaf is probable a blueberry leaf, based on predictions

Figure 7.1

Chapter 8

Conclusions

During the project we have learnt how to manage a Convolutional Neural Network, how important is the usage of an index to lower the retrieval time.

Regarding the DenseNet network we observe how it works fine for the majority of the images of leaves, expect when the query image is a rotten leaf, in this particular case it give as result rotten leaves of any species. An explanation that we can give is that our model cares more about the general colour rather than the overall shape of the leaf.

Another important aspect of this work is that the performance of the model with all the weights of DenseNet fixed are extremely near to the perfection. This didn't give us the opportunity to appreciate the performance of fine-tuning on a convolutional neural network.

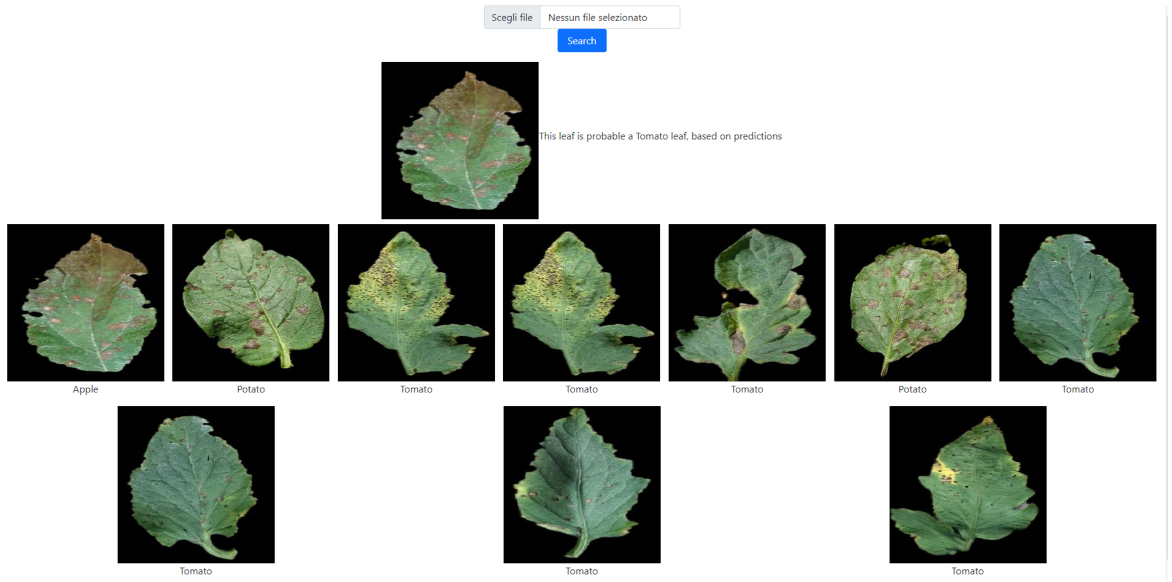


Figure 8.1