11th December 2024

Machine Learning Project

CEBD MOD10 GRB 2242 (Fall 2024)

Leaves Life

Leaf Disease Detection on Images with Neural Networks Project

Mathias BENOIT - Adam CHABA - Eva MAROT - Sacha PORTAL

Table of contents

Git repository: https://github.com/AdamChb/LeavesLife

Introduction

As part of our machine learning course, we have developed a project for the automated detection of plant diseases from images. For this project, we are using the PlantVillage dataset, a database containing thousands of images representing various plant diseases as well as healthy leaves. This dataset allows us to work with a rich and varied dataset, but which can also present imbalances requiring data augmentation.

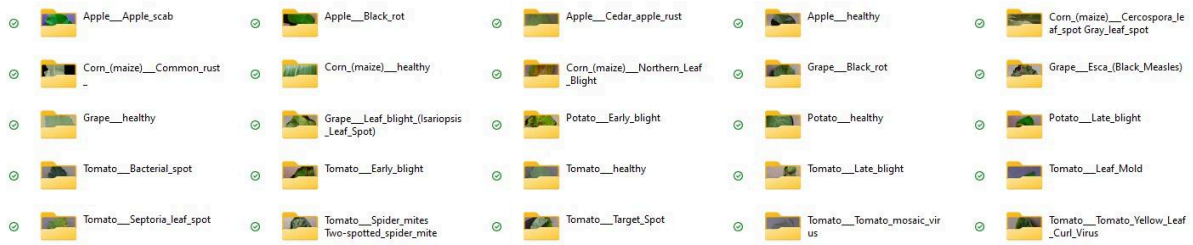
We opted for a convolutional neural network (CNN) model, a complex architecture but the best model for image analysis. It is capable of automatically extracting relevant features to classify images according to plant health status.

Dataset

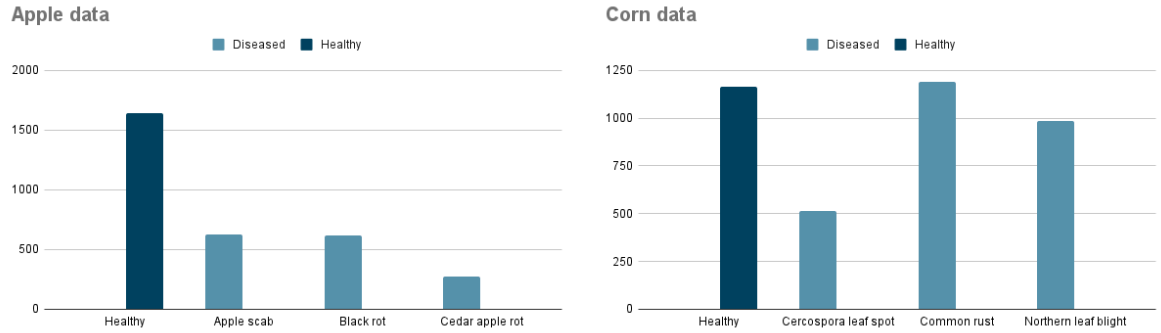
Original dataset

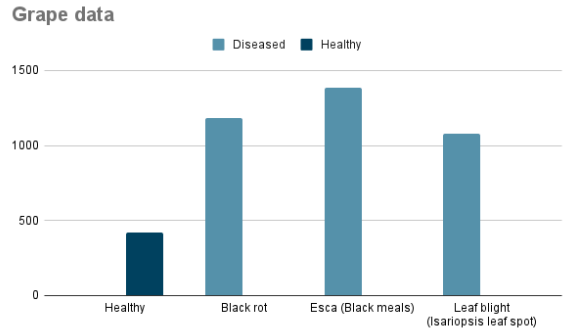
Our dataset offers three versions of each image: color, segmented and grayscale, with 54,305 images in each category. For this project, we have chosen to keep only the color images. We now have 38 directories, featuring 14 plants.

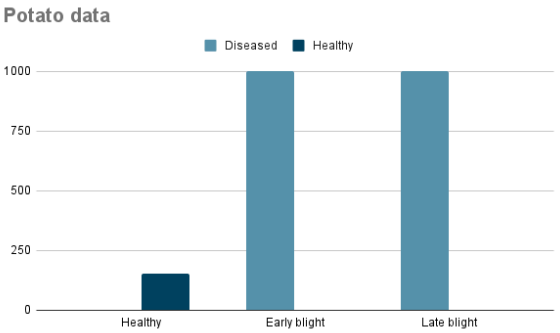
But some plants don't have enough data, so we selected the best ones to use. We chose 5 plants: apple, corn, grape, potato, tomato. Thus we have 31 397 files in 25 folders we will use.

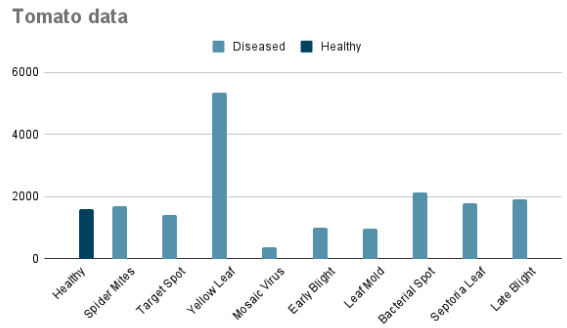


Here you can find the representation of the number of images per category:









Images

The images from the dataset are all .jpg and are 256x256 in size, which is perfect because we did not have to rescale them.

Examples of images from the dataset

Apple healthy

Grape black rot

Corn northern leaf blight

Potato early blight

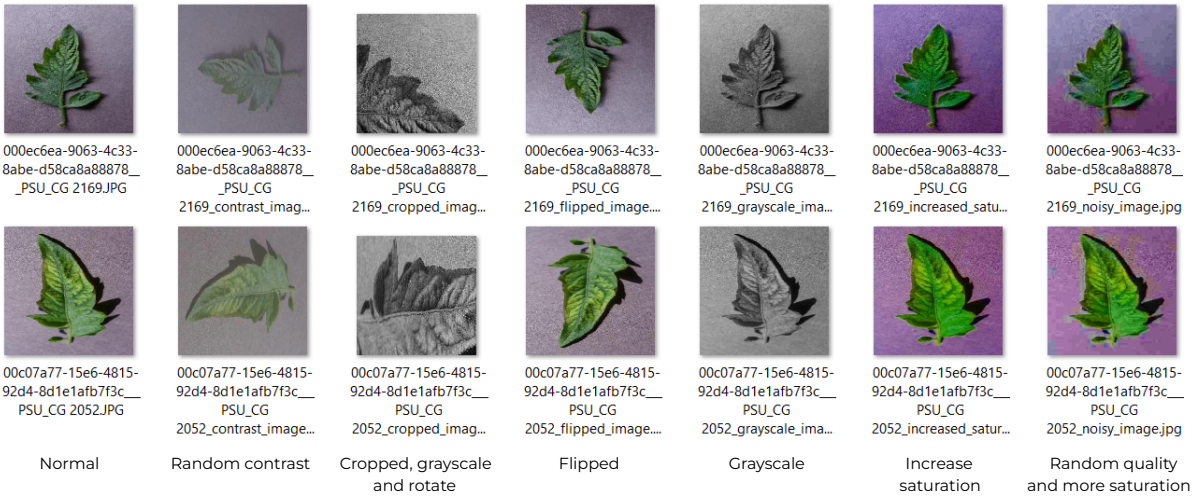
Tomato healthy

Tomato bacterial spot

Data augmentation

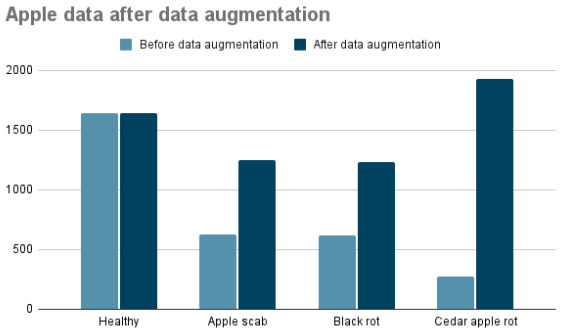
To have more data, we used data augmentation. We wanted to have at least 1000 images per category. So we created 6 images based on one by changing some parameters like saturation, contrast or flipping the image using tensor flow functions. However, if we were very close to having 1000 images in one category, we only created 1 image from another by increasing the saturation.

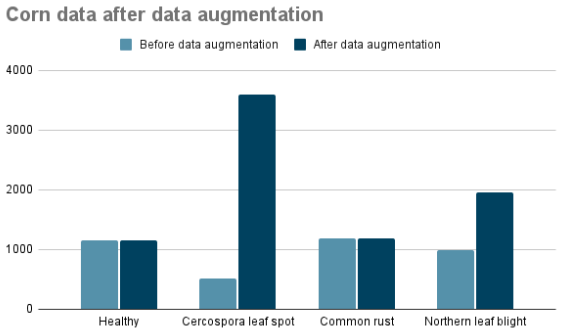
Example of data augmentation:

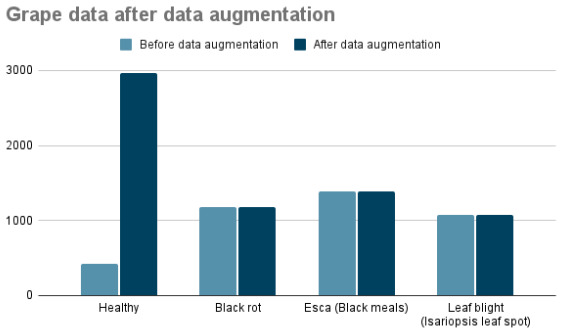


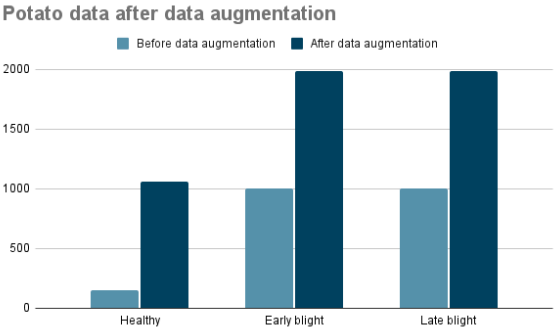
Normal | Random contrast | Cropped, grayscale and rotate | Flipped | Grayscale | Increase saturation | Random quality and more saturation

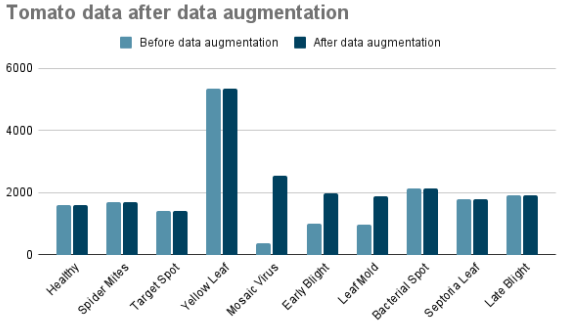
Now, after data augmentation, here are the histograms showing the number of images before and after data augmentation:











This augmentation, on top of providing a larger dataset for the model train into (thus increasing the performance of our model), balances the difference between class populations, so that each disease dataset has enough sample to be correctly recognized. So after data augmentation, we have 47,871 images.

Moreover, we took out 10 images per category and put it in a “test folder”, to make sure that these images aren’t part of the training dataset. These images are the ones that will be used for testing during the presentation. After this step, we have 47,621 images for training and testing, and 250 separated for testing during presentation.

Model choice

In our project, we have chosen to use a CNN (*Convolutional Neural Network*) model for image processing. CNNs are particularly well suited to image datasets thanks to their ability to automatically extract features, such as shapes or colors, and use them efficiently for classification tasks.

What is a convolutional neural network ?

Convolutional Neural Networks (CNN) are deep learning models very useful to analyze images. However, they are considered to be “black box models”, meaning that we can't observe and understand what's going on inside them.

At first glance, their mode of operation is simple: the user provides an input image in the form of a pixel matrix. This matrix has three dimensions: two dimensions for a grayscale image, and a third, of depth 3, to represent the fundamental colors (Red, Green, Blue).

Unlike other models, it's not necessary to supply the features directly to the model. Instead, the images themselves are given as input, and the model automatically generates the features. This means that we don't know in advance the exact number of features that will be extracted. CNNs contain several hidden layers made up of nodes, and their architecture is generally divided into multiple parts:

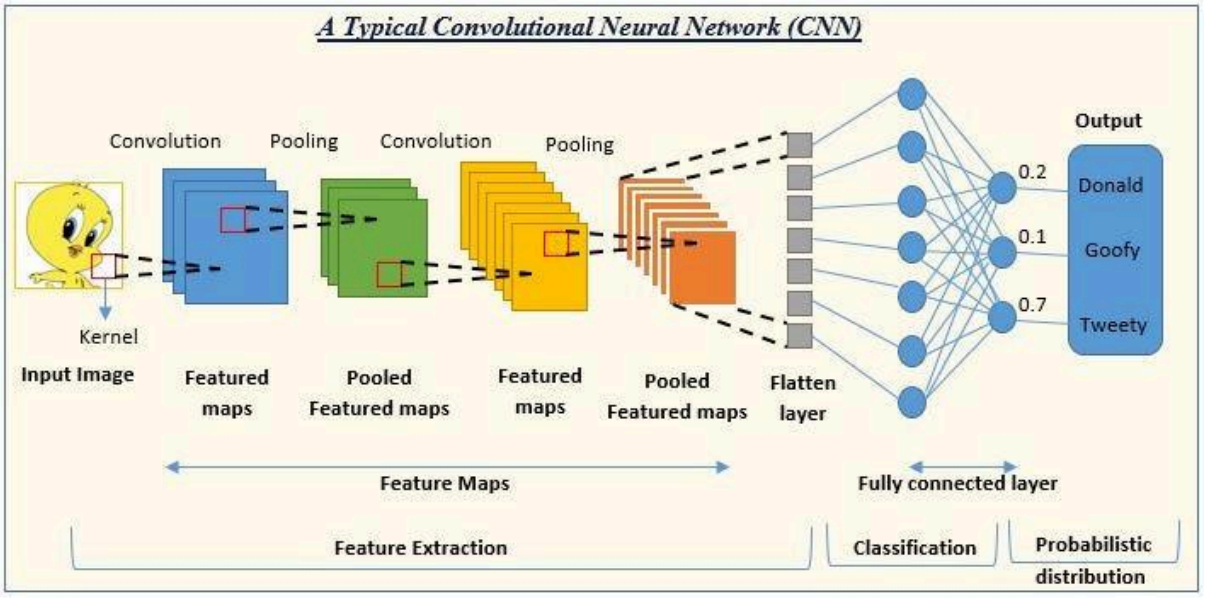
The first part is the convolution part. Its aim is to extract the characteristics specific to each image, while compressing their initial size. The image passes through a succession of filters, generating new images called kernel images. These maps contain different features. For example, for each input image, 32 copies can be created, each capturing specific features.

Pooling is a key step that takes place after convolution to reduce the size of feature maps, while retaining their essential information. This reduces

computational complexity and limits overfitting. After that, what is obtained is then taken as input to this second part, whose role is to classify the image.

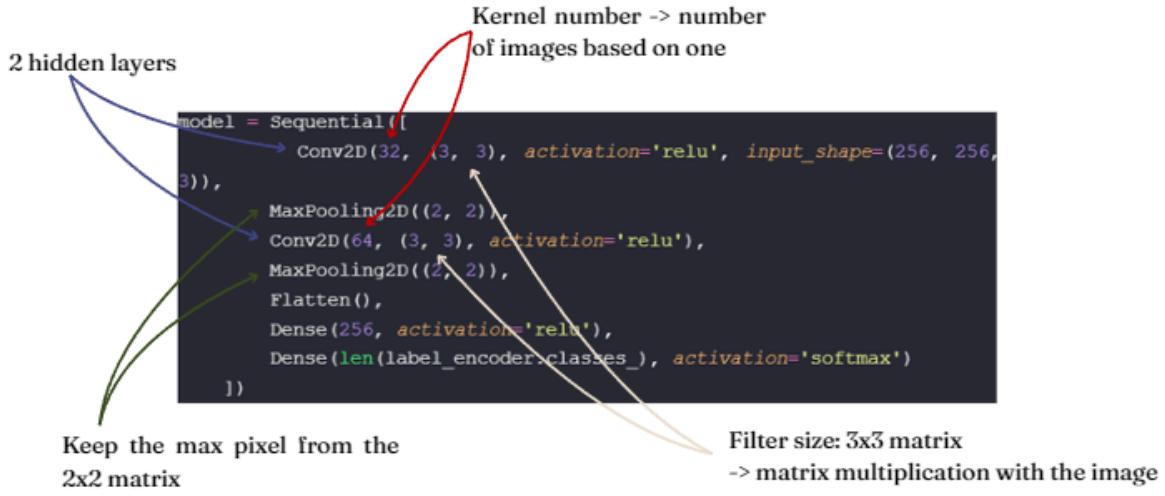
That said, CNNs can be very demanding in terms of computation, often requiring graphics processing units (GPUs) to train models efficiently.

Here is a graphic representation that could help understand what is a convolutional neural network:

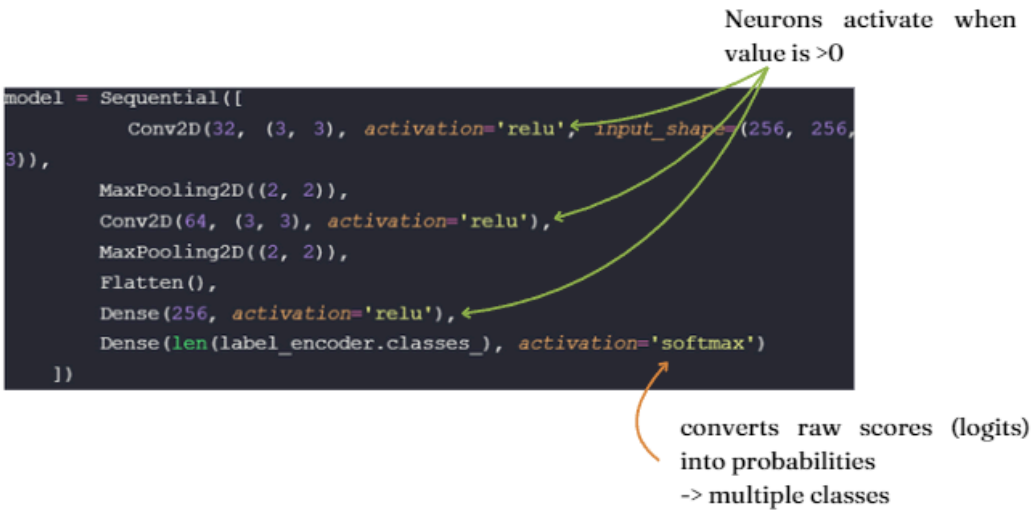


Our code for the model

```
model = Sequential([
          Conv2D(32, (3, 3), activation='relu', input_shape=(256, 256,
3)),
        MaxPooling2D((2, 2)),
        Conv2D(64, (3, 3), activation='relu'),
        MaxPooling2D((2, 2)),
        Flatten(),
        Dense(256, activation='relu'),
        Dense(len(label_encoder.classes_), activation='softmax')
    ])
```



Since the number of kernels is 32, for each image it will create 32 kernel images (copies), with each image containing different features. For the second layer, the kernel number is 64 so we will now have 2048 kernel images (64x32=2048).




```
model.compile(optimizer='adam',loss='categorical_crossentropy',
metrics=['accuracy'])
```

We chose to use Adam because it is the best optimizer!

Number of parameters

We wanted to see the number of parameters using the model.summary() line, and we were shocked to find that our model used 63 millions of parameters to correctly guess the target.

```
model.summary()
```

```
Model compiled.
Model: "sequential_3"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_6 (Conv2D)           (None, 254, 254, 32)      896

 max_pooling2d_6 (MaxPooling  (None, 127, 127, 32)     0
 2D)

 conv2d_7 (Conv2D)           (None, 125, 125, 64)      18496

 max_pooling2d_7 (MaxPooling  (None, 62, 62, 64)       0
 2D)

 flatten_3 (Flatten)         (None, 246016)            0

 dense_6 (Dense)             (None, 256)               62980352

 dense_7 (Dense)             (None, 25)                6425

=================================================================
Total params: 63,006,169
Trainable params: 63,006,169
Non-trainable params: 0
```

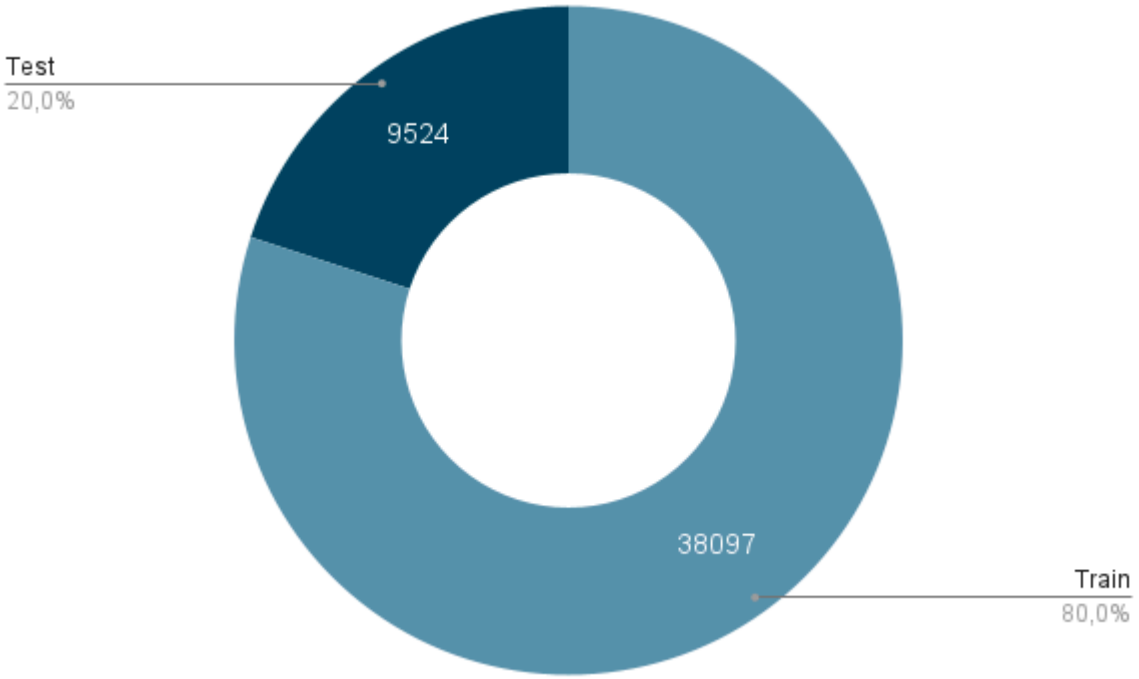
Training and testing

Splitting data

```
print("Splitting data...")
        X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)
    train_dataset = create_dataset(X_train, y_train)
    test_dataset = create_dataset(X_test, y_test)
    print("Data splitted.")
```

Before training, we need to split the data between training and testing data. The training data needed to be bigger, so we chose to do 80% for training and 20% for testing. Since we have 47,621 images, the train dataset contains 38,097 images and the test dataset contains 9524 images.

PS: This explanation is for the last model



Training

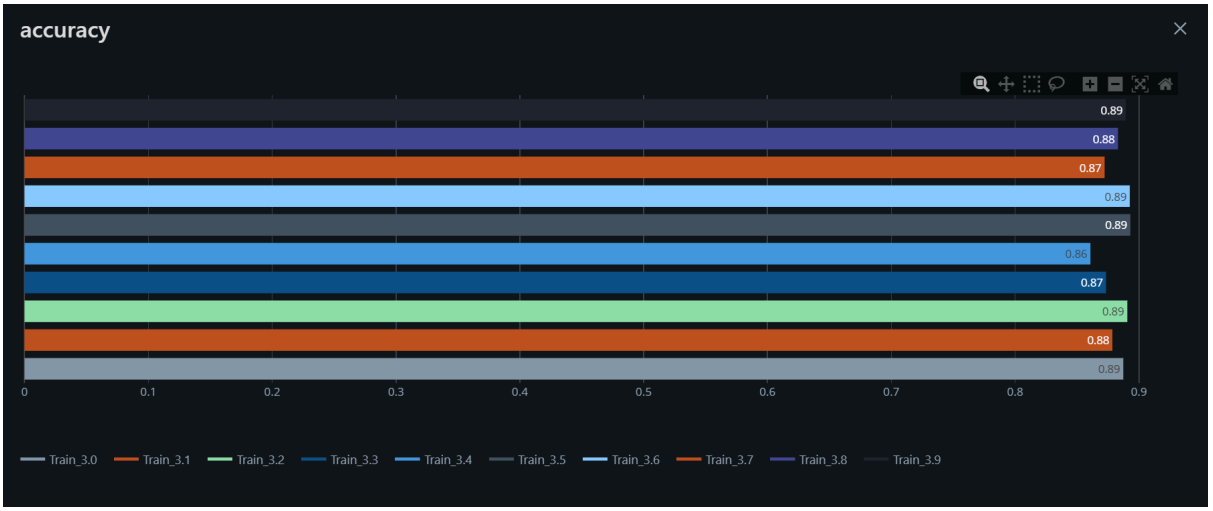
During the project, we used 4 different python codes:

- ➢ Model 0: 25,000 images, no chart or graphics
- ➢ Model 1: 25,000 images, chart and graphs
- ➢ Model 2: 47,621 images, chart and graphs
- ➢ Model 3: 47,621 images, chart and graphs, changes in preprocesses

So the model 0 and 1 are the same, we just have graphics for the model 1.

For all the trains we did, we chose 10 epochs and we have an early stop. So this means that if we have an accuracy n that is lower to accuracy n-1, the training stops and does not continue and train all the epochs.

In addition to that, we decided to create 10 versions of each model, with the exact same parameters but a different seed. That allowed us to take the best model (the one with the highest test accuracy). Here is an example with the 10 versions of the third model, and their accuracies :



As a consequence, we only saved model 3.5 because it had the highest accuracy (0.89).

For our first trains (model 0 and 1), we used, without specifying a seed, the following model :

- One Convolutional layer with 32 3x3 kernel images (and with an input shape equal to the size of the dataset's images)
- A dimension reduction of this layer by taking the max of 2x2 subsets of these kernel images
- Another Convolutional layer with 64 3x3 kernel images
- Another dimension reduction of this layer by taking the max of 2x2 subsets of these kernel images

This first train was in fact successful, because we obtained an accuracy of approximatively 80%. We reused this model using 10 different fixed seeds, and with this model, we achieved with a certain seed an accuracy of almost 86%.

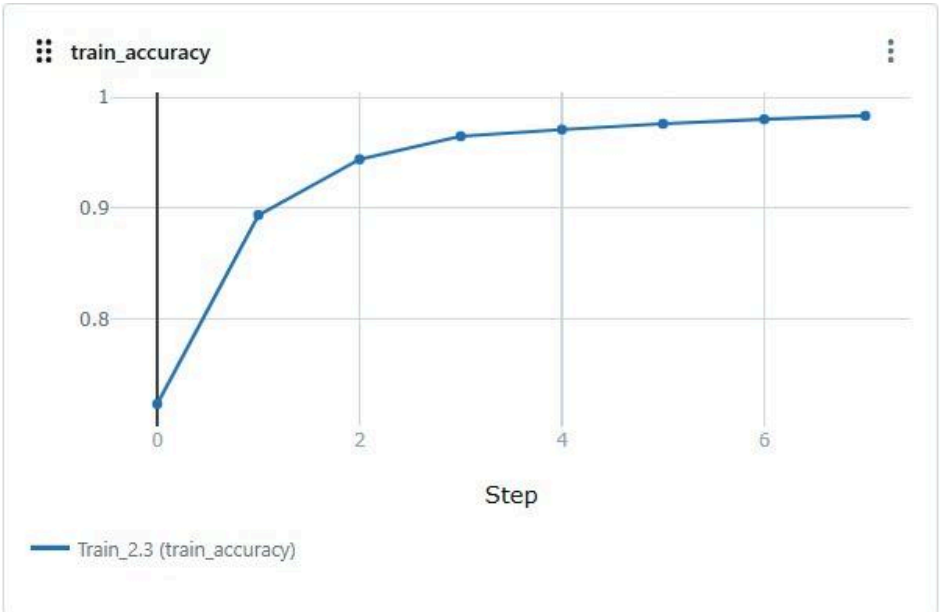
But we observed overfitting, because this accuracy of 86% on test data was achieved with a 94% accuracy on training. We therefore tried to reduce this overfitting, by creating less complex models. We did 3 different tests:

- Reducing the number of kernel images to 16 instead of 32 and 32 instead of 64: we got a test accuracy of 0.81, which is very similar to the unseeded precedent.
- Keeping the first hidden layer with 32 kernel images of size 4x4, but removing the second layer of CNN: the test accuracy diminished to 0.76, and with a greater gap between the test accuracy and the train accuracy.
- Keeping the first hidden layer, this time with 16 kernel images of size 3x3, and removing the second layer of CNN: the test accuracy dropped to 0.37 at its best, while still having a train accuracy of 94%.

As the tests accuracies indicate, each of these modifications failed to reduce the overfitting, so we stuck ourselves to the first model, already performant.

At first, we tried to train the model with a perfectly balanced dataset: exactly 1000 images were used for each class that were chosen randomly. With the other method of loading data that was provided by Tensorflow, we were forced to load the whole dataset, so without the 1000 limit. The model trained using the full dataset (model 2) was even more performant than the previous ones, with a test accuracy that were close to 88%.

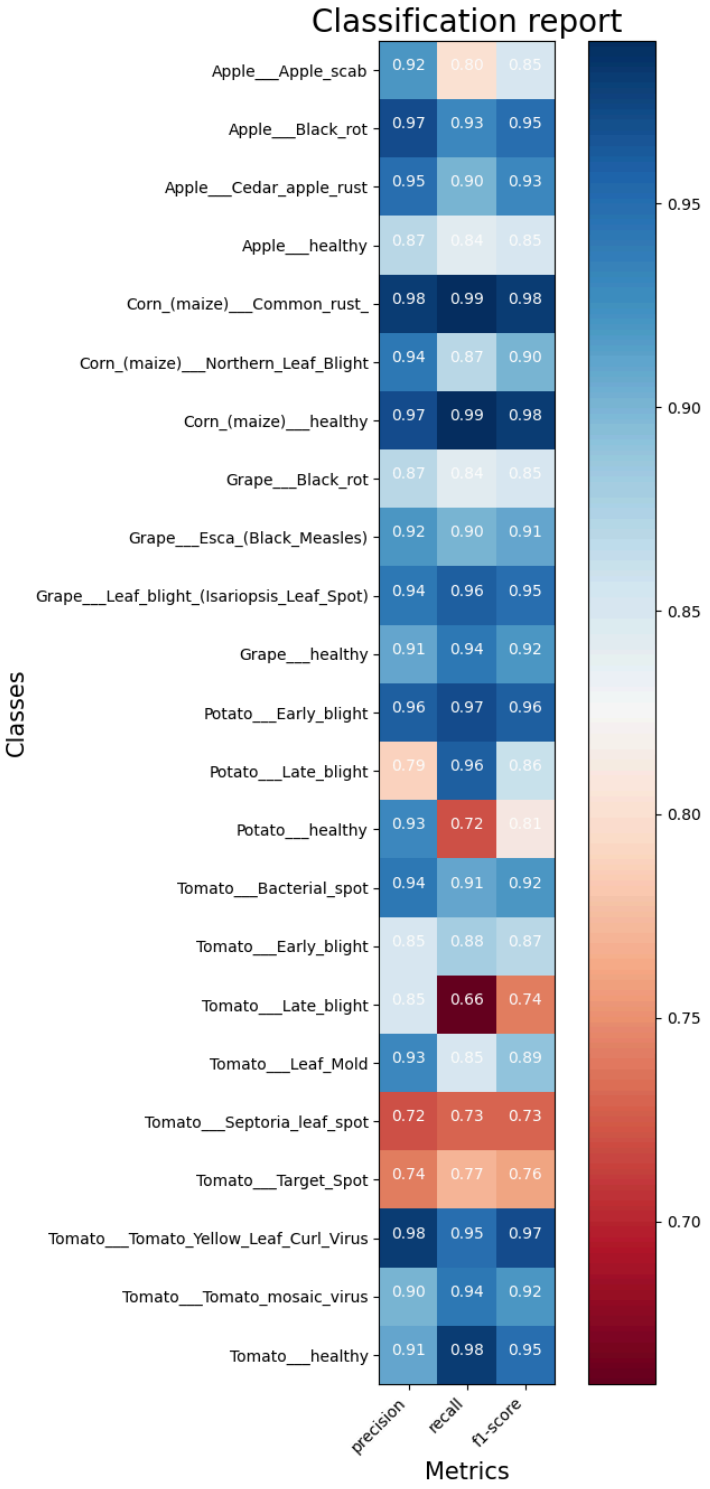
Here is the train accuracy for the same train:

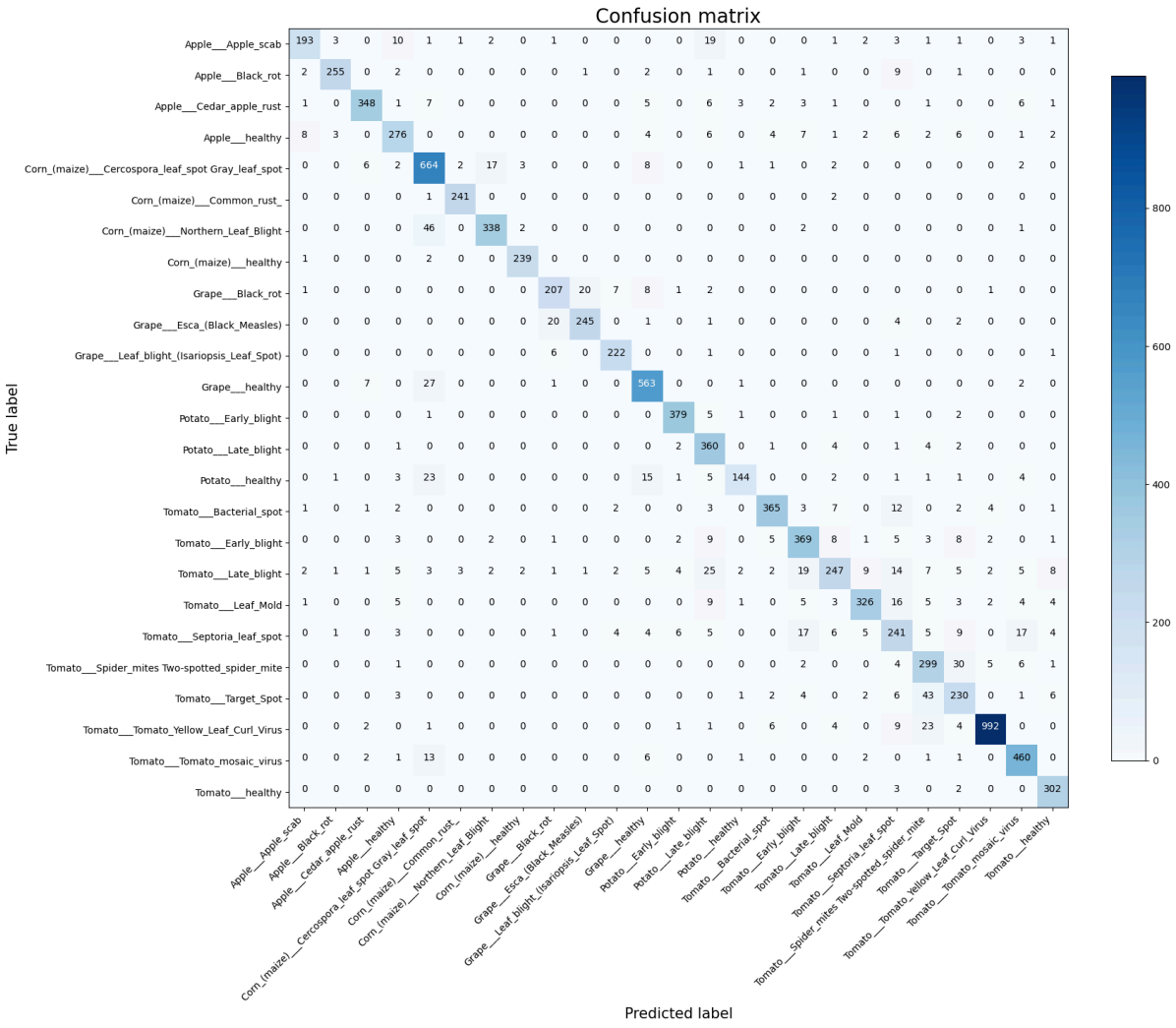


After this, we implemented the final set of trains, the model 3. It is almost the same as model 2, only the preprocess changes, it becomes simplified.

We kept the model 3.5 because it had the highest testing accuracy (0.89).

For this final model, we produced heatmaps and confusion matrix, so that we could perform deeper visual analysis of the model performance :



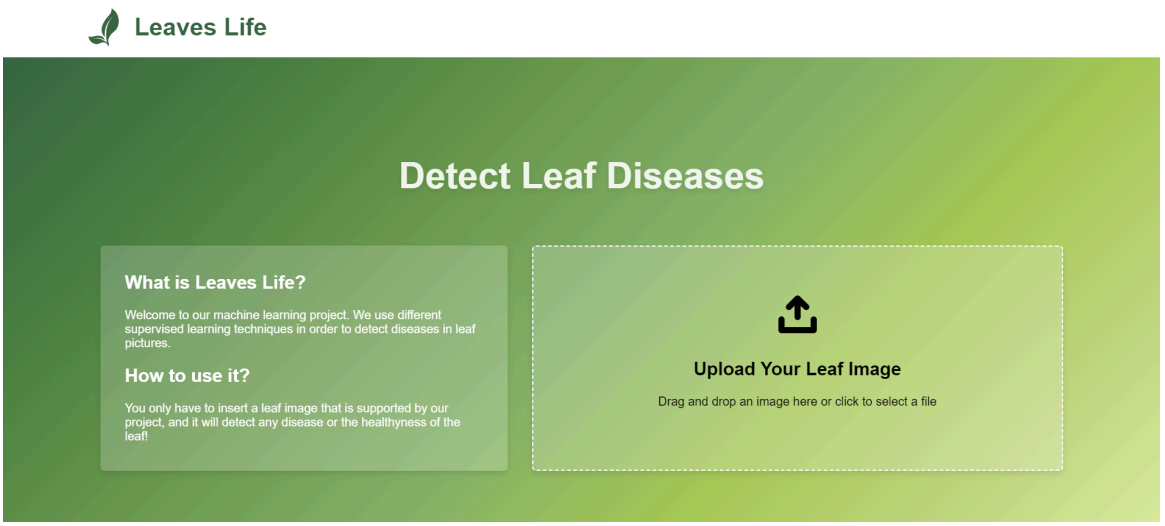


These results indicate that the model is very performant for almost each class, except with 3 tomato diseases that are difficult to clearly separate for the model. For example, Tomato affected by Target Spot and Tomato with Spider mites were greatly mixed up. This problem doesn't come from a lack in the dataset, because the leaves from the other type of plants don't suffer from this small dataset. The answer may come from the natural proximity between the different diseases that make them difficult to differentiate.

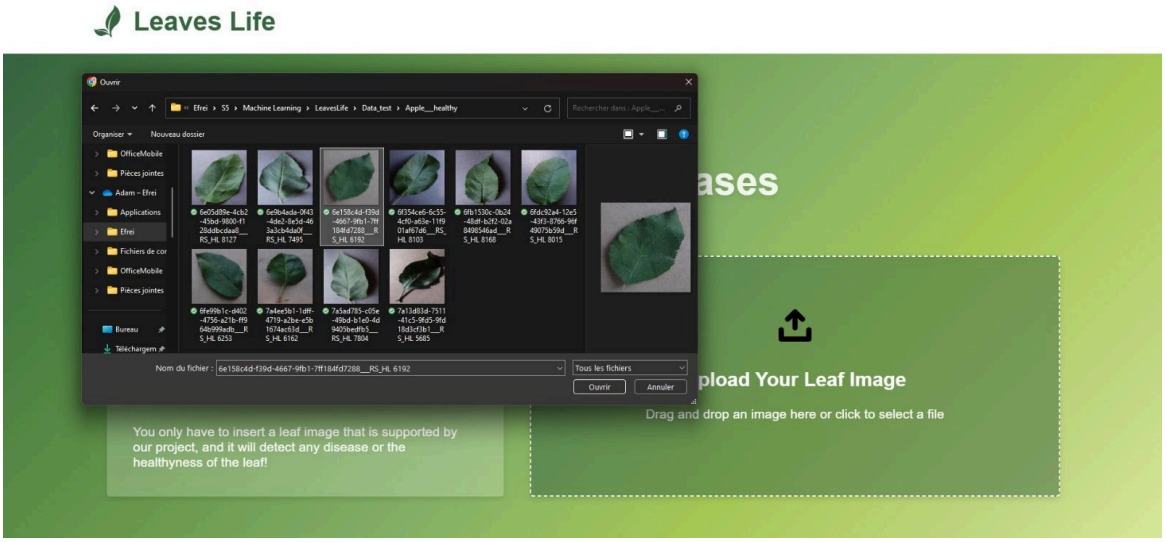
Interface

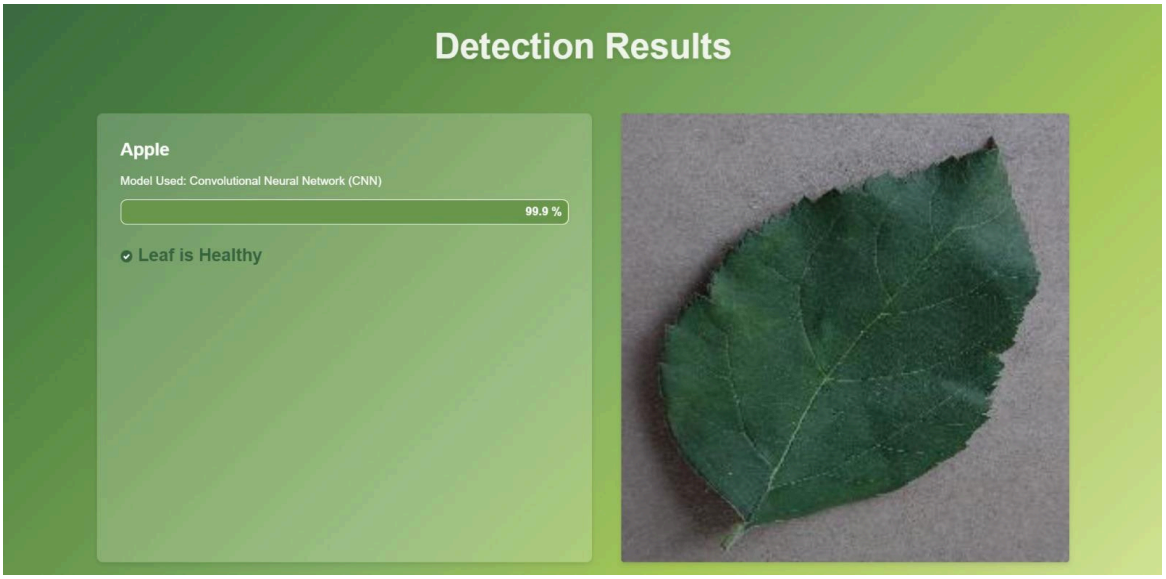
As said earlier, we kept 10 images per class or 250 images in total separated from the training dataset, in order to use them during the presentation. This means that the model has never seen these images.

To test these images, we made a web app where you can upload a leaf image and it will predict the plant and the disease if it has one or if it is healthy otherwise.



For example, if we try with an apple leaf that is healthy, here are the results:





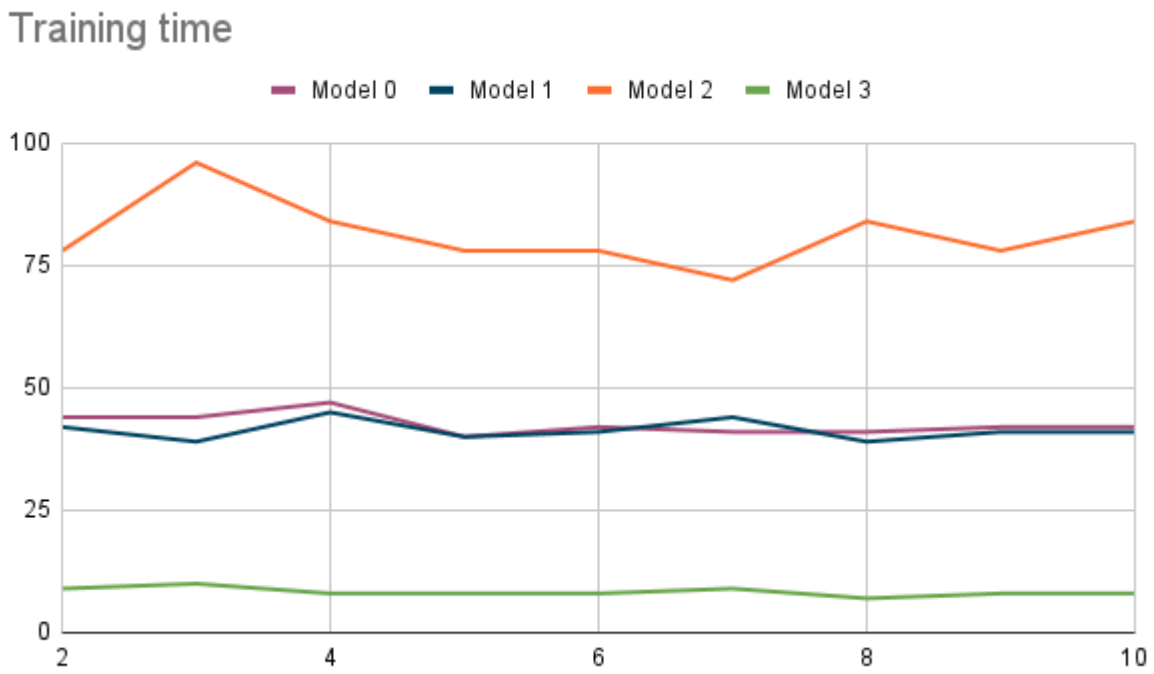
As you can see, the output is well predicted!

Difficulties encountered

Training time

For this project, we did not expect the training time to be this long. We first started to train our model on Adam's laptop. It trained using the CPU, which was very long. The models 0, 1 and 2 were trained this way. The model 2 takes more time because the dataset is bigger. After very long and difficult attempts, we were able to train the model on Mathias' computer in France via Chrome remote desktop. Since he has a NVIDIA GPU, the training took a lot less time, Going from approximately 1h20 to 8 minutes. By the way, this is only the training time without counting the time to load the data, which took like 10 minutes on Adam's laptop and only a few seconds on Mathias' computer.

Fun fact: in total, we trained for 1725 minutes or 28h and 45 minutes.



Training on a GPU

We also struggled a lot when trying to train our model on a GPU. Here are some tips that would have saved us some time.

First of all, we needed to **install compatible versions of CUDA and CUDNN** on the computer. They need to be compatible with both the tensorflow version and the conda libraries (we used CUDA 11.2 and CUDNN 8.1.0).

Then, we decided to use Anaconda, because it is really optimized for machine learning projects, and especially using a GPU. So, in the Anaconda Navigator interface, we created a **python 3.8 environment** (later versions of python didn't work because of many compatibility problems).

In this environment, we needed to install CUDA and CUDNN from the versions that we already installed on the computer:

```
conda install -c conda-forge cudatoolkit=11.2 cudnn=8.1.0
```

After this, we installed tensorflow in the environment (it is really important to **install tensorflow after CUDA and CUDNN**, otherwise it won't work):

```
python -m pip install "tensorflow=2.10"
```

Finally we could install all other useful libraries like mlflow for example.