Image-based plant disease classification

Deep Learning and Applications Part 1

Università degli Studi di Cagliari

Cocco Mattia 65336 and Lepuri Tomas 65358

1. Introduction

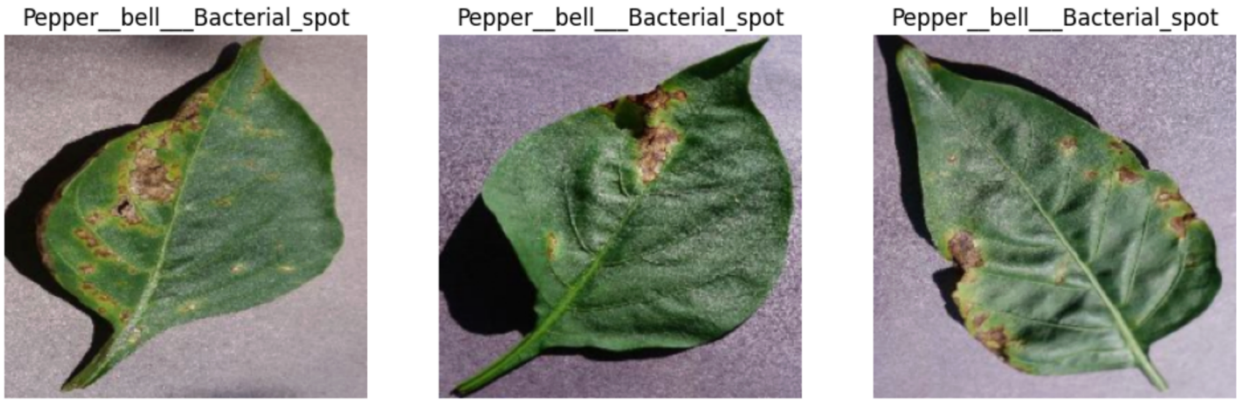
Plant disease detection has always been a relevant task in the agriculture field, since these problems can influence negatively the quality of food, causing economic losses and threatening the health of consumers. Early and accurate identification can help farmers to act tempestively, reducing the spread of infections and the use of pesticides.

Traditionally this task was performed by experts with visual inspection, but this approach can be time-consuming, subjective and not always available. Recent advances in computer vision and **deep learning** made it possible to analyze images and extract discriminative features autonomously, allowing a more scalable solution to the plant disease identification problem.

In this project we investigate the task of **plant disease classification from leaf images**, using different deep learning techniques based on Convolutional Neural Networks and exploiting transfer learning from large-scale datasets.

2. Data

The data employed in this project consist in the **PlantVillage dataset**, available at https://www.kaggle.com/datasets/emmarex/plantdisease .



This dataset contains **20.6k images** of leaf images, picturing three species in healthy and diseased conditions, in a total of **15 classes**, specifically:

- **Pepper bell** (healthy and bacterial spot)
- **Potato** (healthy, early blight and late blight)
- **Tomato** (healthy, bacterial spot, early and late blight, mold, septoria, spider mites, target spot, yellow leaf Curl virus and mosaic virus)

The presence of initial and late stages of the blight disease helps to improve the early detection of this issue, hitting a crucial point made in the introduction.

3. Methods

The approaches selected to solve this problem are mainly three:

- A **custom Convolutional Neural Network**, trained from scratch.
- A **pre-trained ResNet** (18).
- A **fine-tuned ResNet** (18).

During the tests we decided to try various configurations of the networks above, to observe the differences in the results and to overcome problems such as overfitting and underfitting.

3.1. Preprocessing

Before discussing the methods implemented, here are a few **preprocessing** steps that were performed:

- Image **resizing**: all images were resized to a fixed spatial resolution (**224x224**), to ensure a consistent input size and to make our approaches directly comparable.
- **Normalization**: normalization (**channel-wise mean and standard deviation**) was adopted, using ImageNet values for consistency with pre-trained models and for comparability across architectures. To avoid possible discrimination, also dataset-wise mean and std values were tested on the custom CNN.

The transformation and mean/std computation was performed **separately** between the dataset splits, to avoid data leakage and to allow future data augmentation and other actions. The data loader shuffle was set to true for the training set.

Specifically, the dataset was **split** in training set, validation set and test set, following **60-20-20** proportion to allow various tests with different hyperparameters and configurations on the validation set and ensure a fair comparison between our CNN and ResNets on the test set, never used until the very end. A concatenated training-evaluation set was built for final tests.

3.2. Custom Convolutional Neural Network

The first approach is a custom Convolutional Neural Network, consisting in **three blocks** containing:

- **Convolution** (3x3 kernel and 1 padding)
- **ReLU** activation
- **Pooling** (Max, 2x2 filter)

Followed by the classifier:

- A first **Fully Connected** layer
- **ReLU** activation
- A second **Fully Connected** layer

```
# CNN definition
class CustomCNN(nn.Module):
    def __init__(self, num_classes):
        super(CustomCNN, self).__init__()

        self.features = nn.Sequential(
            nn.Conv2d(3, 32, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(2),        # 112x112

            nn.Conv2d(32, 64, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(2),        # 56x56

            nn.Conv2d(64, 128, kernel_size=3, padding=1),
            nn.ReLU(),
            nn.MaxPool2d(2)         # 28x28
        )

        self.classifier = nn.Sequential(
            nn.Linear(128 * 28 * 28, 256),
            nn.ReLU(),
            nn.Dropout(0.5),
            nn.Linear(256, num_classes)
        )
```

We chose **ReLU** and **Max Pooling**, since they have become common practice in image classification tasks.

As Loss Function we opted for the **Cross Entropy Loss** and for optimization we selected **Adaptive moment estimation**, with a **learning rate** of **1e-3**, which later turned out to be the best.

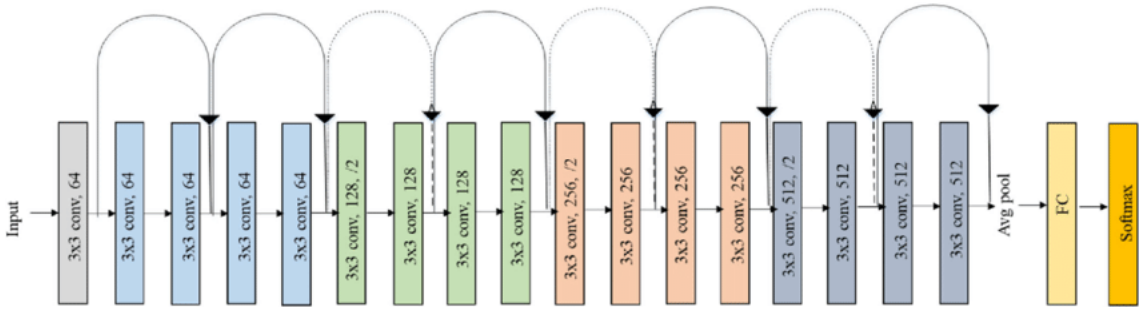
After some tests, **dropout** and training set **data augmentation** were introduced to overcome overfitting, as we will discuss later.

3.3. ResNet

As a second and third approaches we selected a popular CNN, know for the reduction of the degradation problem, affecting deep neural networks: **ResNet**.

We opted for **ResNet-18**, because it is deep enough to capture complex patterns but is significantly lighter than other versions, such as ResNet-50 or ResNet-101.

The architecture of this network consists in 18 convolutional layers, organized in **residual blocks** with **skip connections**, which are the reasons of its success.



Finally, ResNet-18 is widely used as a baseline, especially after pre-training on ImageNet dataset and allowed us to experiment with transfer learning and fine-tuning.

4. Experiments

The following experiments were performed to explore the task of plant disease classification, specifically to:

- Determine whether the selected approaches make sense in this field.
- Check the performance of our custom CNN and technical choices.
- Assess the effectiveness of transfer learning from ImageNet.
- Try to obtain a reasonable performance on the PlantVillage dataset.
- Compare the different solutions.

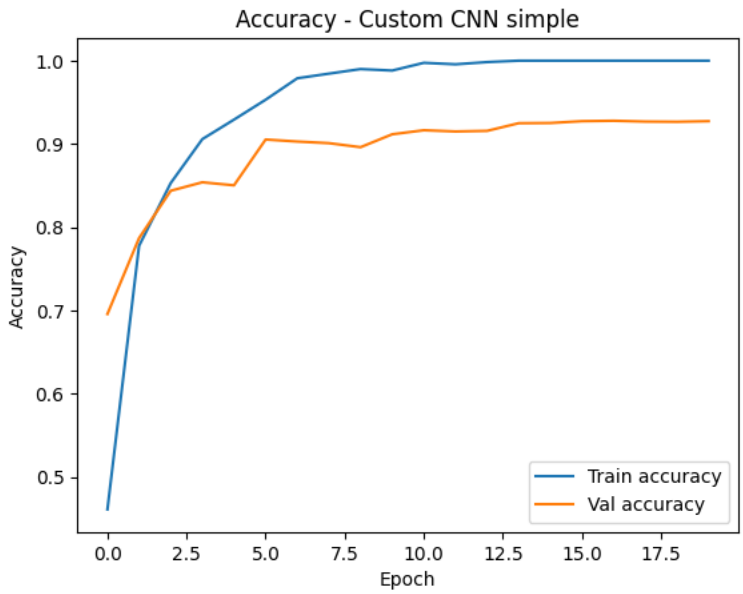
4.1. Custom CNN

After data normalization and dataset split, the first experiment was performed to test our custom CNN (on validation set). The values of mean and standard deviation for the normalization were the ones computed on ImageNet in this phase, to allow direct comparation with ResNet.

As mentioned before the loss function is Cross Entropy and the learning rate was chosen among the standard values and set to 1e-3. The training continued for **20 epochs** and these are the results:

```
Epoch 14/20 - Loss: 0.0019 - Train acc: 1.0000 - val acc: 0.9249
Epoch 15/20 - Loss: 0.0008 - Train acc: 1.0000 - val acc: 0.9251
Epoch 16/20 - Loss: 0.0006 - Train acc: 1.0000 - val acc: 0.9273
Epoch 17/20 - Loss: 0.0005 - Train acc: 1.0000 - val acc: 0.9278
Epoch 18/20 - Loss: 0.0004 - Train acc: 1.0000 - val acc: 0.9268
Epoch 19/20 - Loss: 0.0003 - Train acc: 1.0000 - val acc: 0.9266
Epoch 20/20 - Loss: 0.0003 - Train acc: 1.0000 - val acc: 0.9273
```

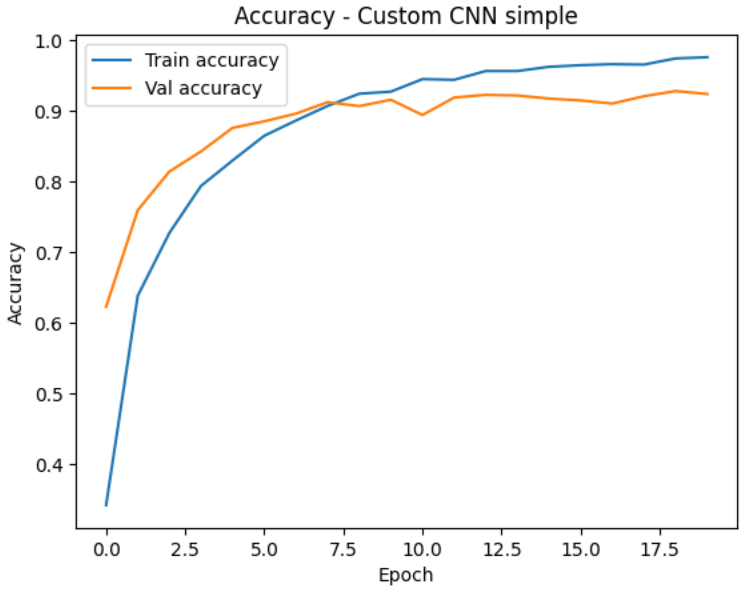
The perfect performance on the training set and the gap between the training and validation accuracy suggested that the network was suffering of a light **overfitting**. The chart makes it clear:



For this reason we decided to:

- Add **data augmentation** to the training set (horizontal flip).
- Add 50% **dropout** to the classifier layer.

After these changes, we repeated the test and the results were better, the gap was reduced and maximum accuracy around **94%** was achieved in the validation set:



We decided to keep the dropout and data augmentation during further tests.

4.1.1. Specific normalization

We also computed mean and standard deviation directly on the PlantVillage dataset and repeated the test. There was no improvement and the validation accuracy stopped at around **91%**:

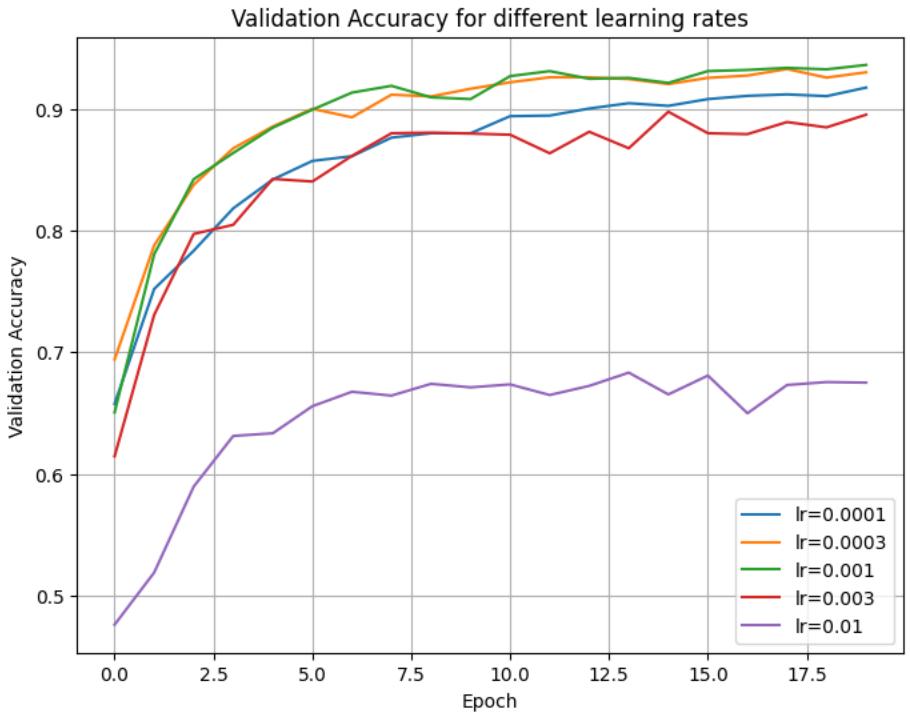
```
Epoch 14/20 - Loss: 0.1925 - Train acc: 0.9347 - val acc: 0.9082
Epoch 15/20 - Loss: 0.1698 - Train acc: 0.9413 - val acc: 0.9123
Epoch 16/20 - Loss: 0.1468 - Train acc: 0.9498 - val acc: 0.9196
Epoch 17/20 - Loss: 0.1374 - Train acc: 0.9528 - val acc: 0.9091
Epoch 18/20 - Loss: 0.1350 - Train acc: 0.9545 - val acc: 0.9050
Epoch 19/20 - Loss: 0.1227 - Train acc: 0.9568 - val acc: 0.9137
Epoch 20/20 - Loss: 0.1153 - Train acc: 0.9593 - val acc: 0.9011
```

The values were computed separately between training and validation set, to avoid **data leakage**.

We decided to keep using the ImageNet-based normalization during the further experiments.

4.1.2. Learning rate exploration

The experiments we just discussed were performed with a learning rate of **1e-3**, which is one of the standard values for the Adam optimizer in this kind of scenarios. However, we repeated the experiments with a few other values, to spot eventual better rates:



This chart describes the validation accuracy with different learning rates in 20 epochs of training and shows that 1e-3 (0.001) is arguably the best one.

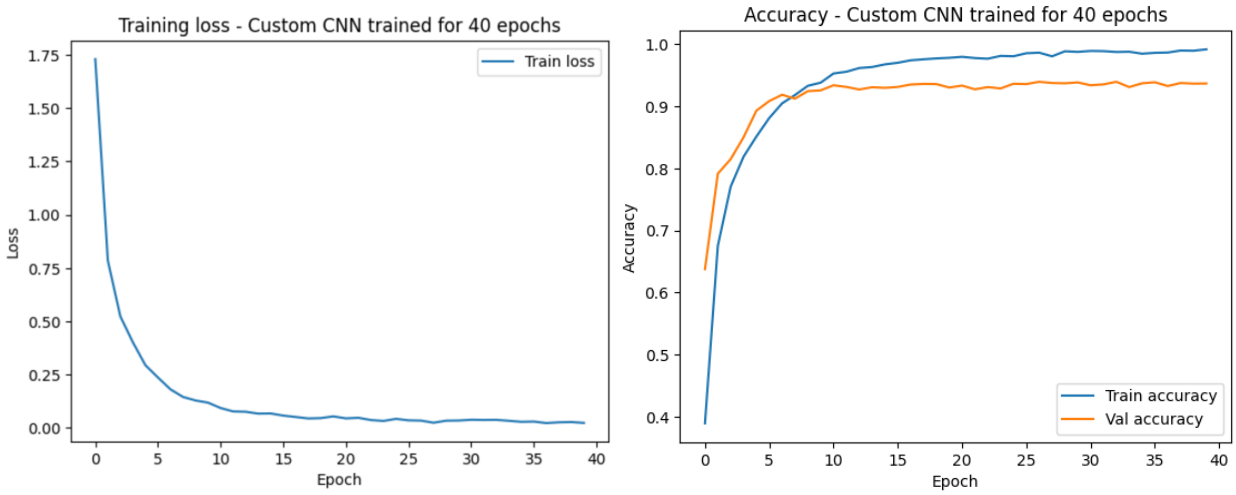
The directions of the top four learning curves may suggest that better accuracy could be obtained increasing the number of epochs.

4.1.3. Increasing number of epochs

Looking for better accuracy, after examining the charts, we repeated the experiment **doubling** the number of epochs.

The validation accuracy reached almost **94%** and the metrics seem stable at this point, but since there is no significant difference, for hardware and comparability reasons we will keep testing with 20 epochs:

```
Epoch 34/40 - Loss: 0.0359 - Train acc: 0.9881 - val acc: 0.9312
Epoch 35/40 - Loss: 0.0478 - Train acc: 0.9850 - val acc: 0.9372
Epoch 36/40 - Loss: 0.0381 - Train acc: 0.9864 - val acc: 0.9389
Epoch 37/40 - Loss: 0.0392 - Train acc: 0.9868 - val acc: 0.9329
Epoch 38/40 - Loss: 0.0291 - Train acc: 0.9901 - val acc: 0.9377
Epoch 39/40 - Loss: 0.0286 - Train acc: 0.9897 - val acc: 0.9368
Epoch 40/40 - Loss: 0.0246 - Train acc: 0.9920 - val acc: 0.9370
```



4.1.4. Custom CNN final test

Finally we tested the Custom CNN with the best configuration (ImageNet normalization, learning rate of 1e-3, dropout…) on the test set.

The test accuracy reached around **95.8%**, which demonstrates that our network is able to generalize quite well in this case:

```
Epoch 14/20 - Loss: 0.0660 - Train acc: 0.9790 - test acc: 0.9477
Epoch 15/20 - Loss: 0.0595 - Train acc: 0.9800 - test acc: 0.9528
Epoch 16/20 - Loss: 0.0591 - Train acc: 0.9801 - test acc: 0.9518
Epoch 17/20 - Loss: 0.0513 - Train acc: 0.9819 - test acc: 0.9487
Epoch 18/20 - Loss: 0.0504 - Train acc: 0.9833 - test acc: 0.9457
Epoch 19/20 - Loss: 0.0474 - Train acc: 0.9836 - test acc: 0.9574
Epoch 20/20 - Loss: 0.0572 - Train acc: 0.9807 - test acc: 0.9586
```

4.2. Pre-trained ResNet

After testing our custom CNN, we attempted to solve the classification problem on this dataset with a popular convolutional network (ResNet-18), **pre-trained** on the **ImageNet** dataset to experiment with **transfer learning**. The experiments were executed directly on the test set, since we used common hyperparameter values for transfer learning and fine-tuning, without trying many different combinations.

After downloading the pre-trained network, we "froze" the weights in every layer, to avoid updates during training and we added a final fully connected layer, specific for our classification problem, which is the only one that received parameters updates.

We set once again:

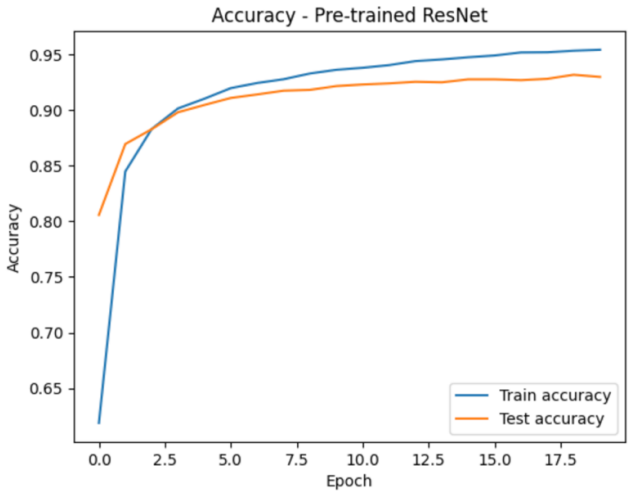
- Loss: **Cross entropy loss**
- Optimizer: **Adam**
- Learning rate: **1e-3**

Batch-size was set to **256** since the beginning, as common practice for ResNets.

The test results show that the pre-trained network is able to use patterns and features learned from ImageNet, achieving a test accuracy of around **93%** (slightly lower than our custom CNN):

```
Epoch 14/20 - Loss: 0.2075 - Train acc: 0.9455 - Test acc: 0.9249
Epoch 15/20 - Loss: 0.2001 - Train acc: 0.9474 - Test acc: 0.9276
Epoch 16/20 - Loss: 0.1942 - Train acc: 0.9490 - Test acc: 0.9276
Epoch 17/20 - Loss: 0.1877 - Train acc: 0.9517 - Test acc: 0.9268
Epoch 18/20 - Loss: 0.1808 - Train acc: 0.9519 - Test acc: 0.9281
Epoch 19/20 - Loss: 0.1770 - Train acc: 0.9533 - Test acc: 0.9317
Epoch 20/20 - Loss: 0.1722 - Train acc: 0.9541 - Test acc: 0.9297
```

The training-test accuracy chart does not show particular overfitting problems:



4.3. Fine-tuned ResNet

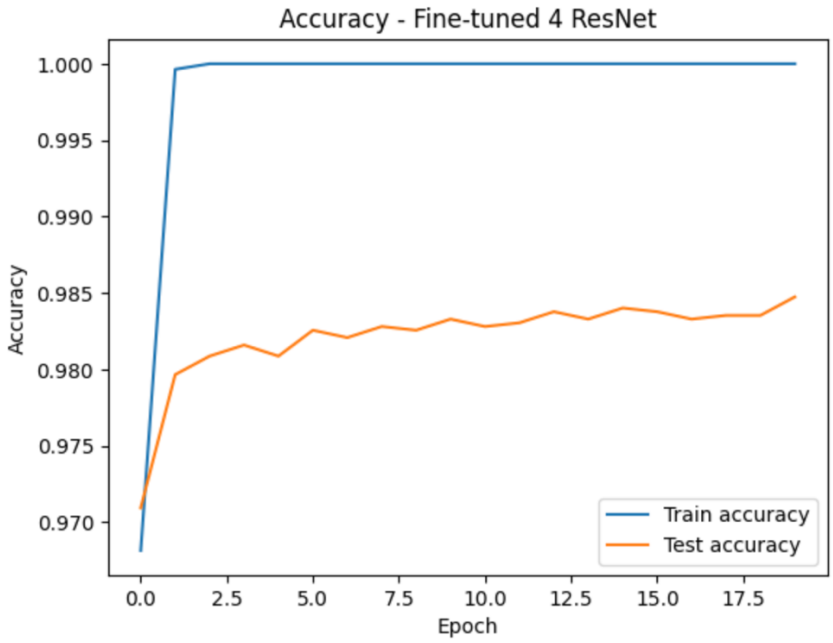
The performance of the pre-trained ResNet was acceptable, but did not even reach the one of our custom CNN, so we made some adjustments to the ResNet-18.

We **enabled the gradients** in the last layer so that, during training, the network could learn dataset-specific patterns. During this test, we decreased Adam **learning rate** to **1e-4**, which is a common practice in fine-tuning, to avoid the "forgetting" of relevant patterns already learned by the CNN.

The difference was immediately clear, since the training accuracy reached 100% rapidly and the test accuracy hit **98%** after the third epoch:

```
Epoch 9/20 - Loss: 0.0004 - Train acc: 1.0000 - Test acc: 0.9826
Epoch 10/20 - Loss: 0.0004 - Train acc: 1.0000 - Test acc: 0.9833
Epoch 11/20 - Loss: 0.0003 - Train acc: 1.0000 - Test acc: 0.9828
Epoch 12/20 - Loss: 0.0003 - Train acc: 1.0000 - Test acc: 0.9830
Epoch 13/20 - Loss: 0.0003 - Train acc: 1.0000 - Test acc: 0.9838
Epoch 14/20 - Loss: 0.0002 - Train acc: 1.0000 - Test acc: 0.9833
Epoch 15/20 - Loss: 0.0002 - Train acc: 1.0000 - Test acc: 0.9840
Epoch 16/20 - Loss: 0.0002 - Train acc: 1.0000 - Test acc: 0.9838
Epoch 17/20 - Loss: 0.0002 - Train acc: 1.0000 - Test acc: 0.9833
Epoch 18/20 - Loss: 0.0001 - Train acc: 1.0000 - Test acc: 0.9835
Epoch 19/20 - Loss: 0.0001 - Train acc: 1.0000 - Test acc: 0.9835
Epoch 20/20 - Loss: 0.0001 - Train acc: 1.0000 - Test acc: 0.9847
```

The gap between training and test accuracy is very small (the focus on this chart is greater than before):

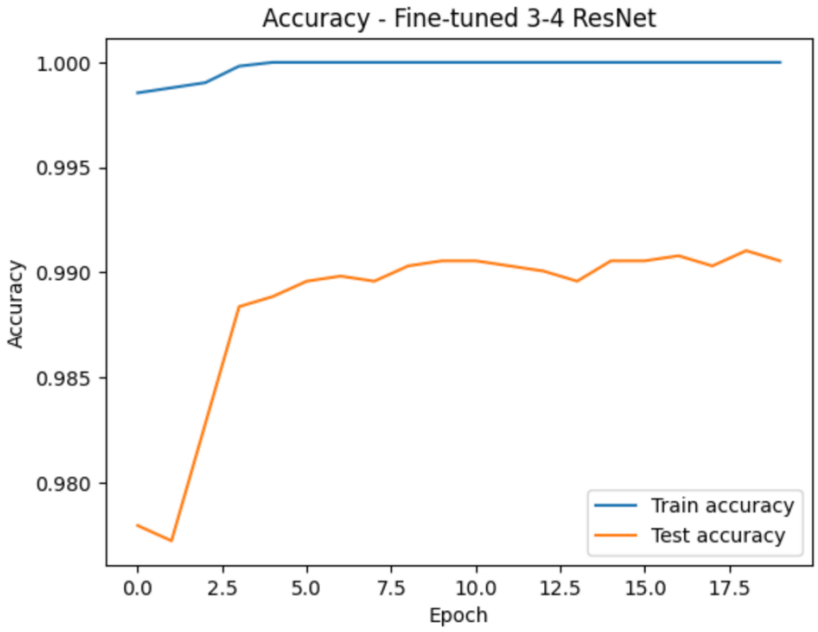


Although these were the best results obtained till now, we tried one last test to see if it was possible to do better and reduce the **gap** between training and test accuracy.

For the final experiment we "unfroze" also the **third layer** and we used a slightly different optimizer, called **AdamW**, which allowed us to set a **weight decay** of **1e-5**: a state-of-art practice on ResNets that aims to penalize big weights, acting as a regularizer and reducing overfitting.

We succeded in our intent, reducing slightly the gap, reaching **loss zero** on the training set and over **99%** test accuracy:

```
Epoch 1/20 - Loss: 0.0051 - Train acc: 0.9985 - Test acc: 0.9780
Epoch 2/20 - Loss: 0.0043 - Train acc: 0.9988 - Test acc: 0.9772
Epoch 3/20 - Loss: 0.0037 - Train acc: 0.9990 - Test acc: 0.9828
Epoch 4/20 - Loss: 0.0008 - Train acc: 0.9998 - Test acc: 0.9884
Epoch 5/20 - Loss: 0.0001 - Train acc: 1.0000 - Test acc: 0.9889
Epoch 6/20 - Loss: 0.0001 - Train acc: 1.0000 - Test acc: 0.9896
Epoch 7/20 - Loss: 0.0001 - Train acc: 1.0000 - Test acc: 0.9898
Epoch 8/20 - Loss: 0.0000 - Train acc: 1.0000 - Test acc: 0.9896
Epoch 9/20 - Loss: 0.0000 - Train acc: 1.0000 - Test acc: 0.9903
Epoch 10/20 - Loss: 0.0000 - Train acc: 1.0000 - Test acc: 0.9906
Epoch 11/20 - Loss: 0.0000 - Train acc: 1.0000 - Test acc: 0.9906
Epoch 12/20 - Loss: 0.0000 - Train acc: 1.0000 - Test acc: 0.9903
Epoch 13/20 - Loss: 0.0000 - Train acc: 1.0000 - Test acc: 0.9901
Epoch 14/20 - Loss: 0.0000 - Train acc: 1.0000 - Test acc: 0.9896
Epoch 15/20 - Loss: 0.0000 - Train acc: 1.0000 - Test acc: 0.9906
Epoch 16/20 - Loss: 0.0000 - Train acc: 1.0000 - Test acc: 0.9906
Epoch 17/20 - Loss: 0.0000 - Train acc: 1.0000 - Test acc: 0.9908
Epoch 18/20 - Loss: 0.0000 - Train acc: 1.0000 - Test acc: 0.9903
Epoch 19/20 - Loss: 0.0000 - Train acc: 1.0000 - Test acc: 0.9910
Epoch 20/20 - Loss: 0.0000 - Train acc: 1.0000 - Test acc: 0.9906
```



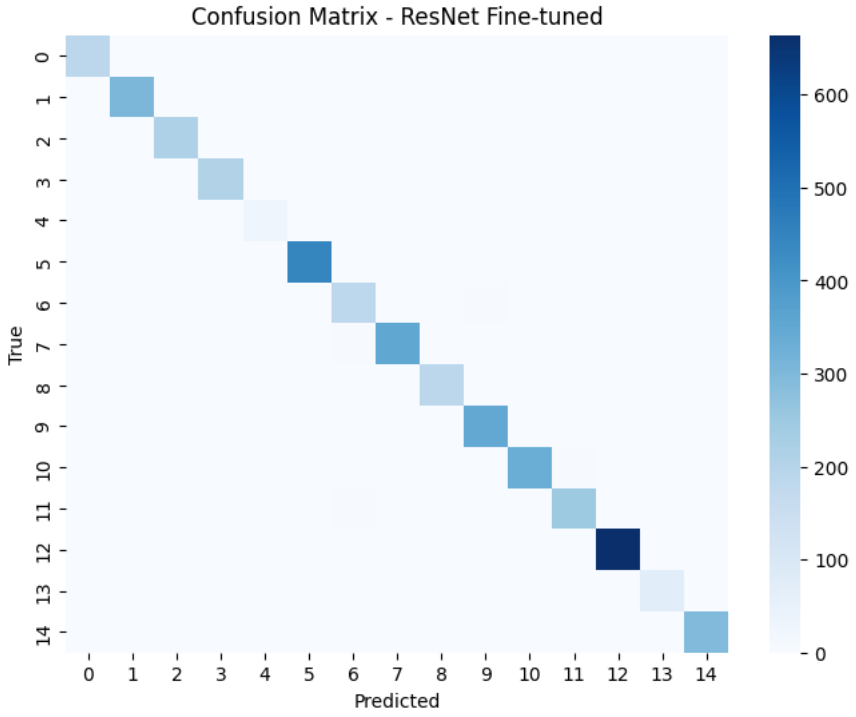
4.3.1. Failure cases analysis

After several experiments, the best performance on the test set was achieved by the fine-tuned ResNet-18, with:

- Over **99% accuracy**
- Only **41 misclassified samples** out of 4128 (test set)

We analyzed these errors to identify eventual patterns that the network did not capture well or particular conditions that were causing failure.

The confusion matrix did not show the information we were looking for because the error rate is too low:



Certain classes have been predicted correctly more than others, just because they are more present in the dataset, for example:

- Class 4 is the less colored because it corresponds to Potato_healthy, which contains the least number of images.
- Class 12 on the other hand is Tomato_yellow_leaf_Curl_virus, which is the largest class in the dataset.

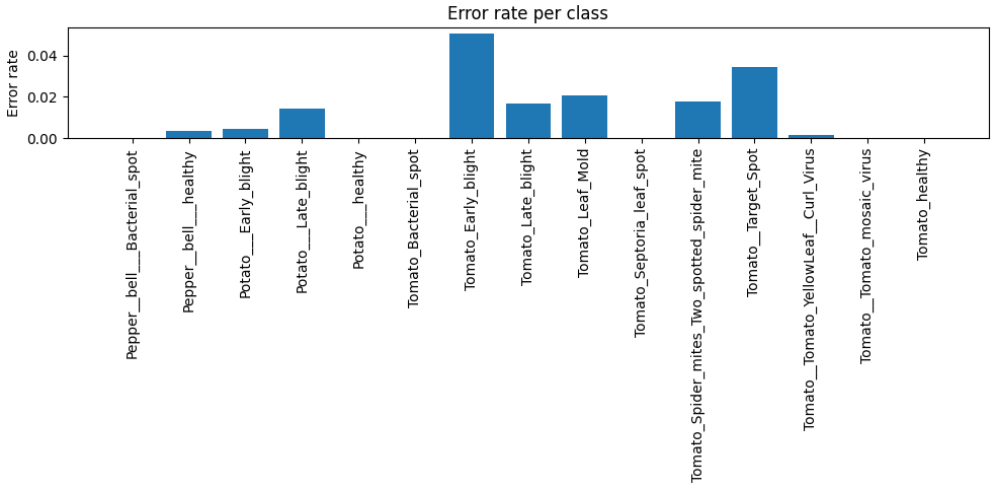
We decided to check the error rate per class and found out that a few specific classes were misclassified more:

```
Error rate per class:
Pepper__bell___Bacterial_spot :   0.00%
Pepper__bell___healthy        :   0.33%
Potato___Early_blight         :   0.47%
Potato___Late_blight          :   1.42%
Potato___healthy              :   0.00%
Tomato_Bacterial_spot         :   0.00%
Tomato_Early_blight           :   5.10%
Tomato_Late_blight            :   1.66%
Tomato_Leaf_Mold              :   2.09%
Tomato_Septoria_leaf_spot     :   0.00%
Tomato_Spider_mites_Two_spotted_spider_mite:   1.77%
Tomato__Target_Spot           :   3.47%
Tomato__Tomato_YellowLeaf__Curl_Virus:   0.15%
Tomato__Tomato_mosaic_virus   :   0.00%
Tomato_healthy                :   0.00%
```

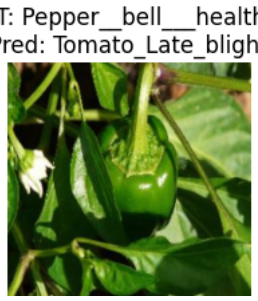
After spotting **Tomato_early_blight**, we immediately thought that the network was having trouble to identify the disease in the early stages, which would have been a major problem in this scenario. After analyzing the **top confusions**, we understood that the diseased plants were not mistaken for healthy ones, but instead, the classifier rarely confused some diseases, for instance:

Prediction	Ground Truth	Number of mispredictions
Tomato Spider Mite	Tomato Target Spot	5
Tomato Late Blight	Tomato Early Blight	5
Tomato Early Blight	Tomato Septoria Spot	3
Tomato Target Spot	Tomato Early Blight	3
Tomato Early Blight	Tomato Late Blight	2

So we hypothesize that the network sometimes has minimal problems with patterns linked to Tomato Blight, Spider Mite and Target Spot, which is also suggested by the chart:



"**Healthy classes**" were not subject of mispredictions. The Pepper_Bell_Early error is caused by an image that does not picture a single leaf:



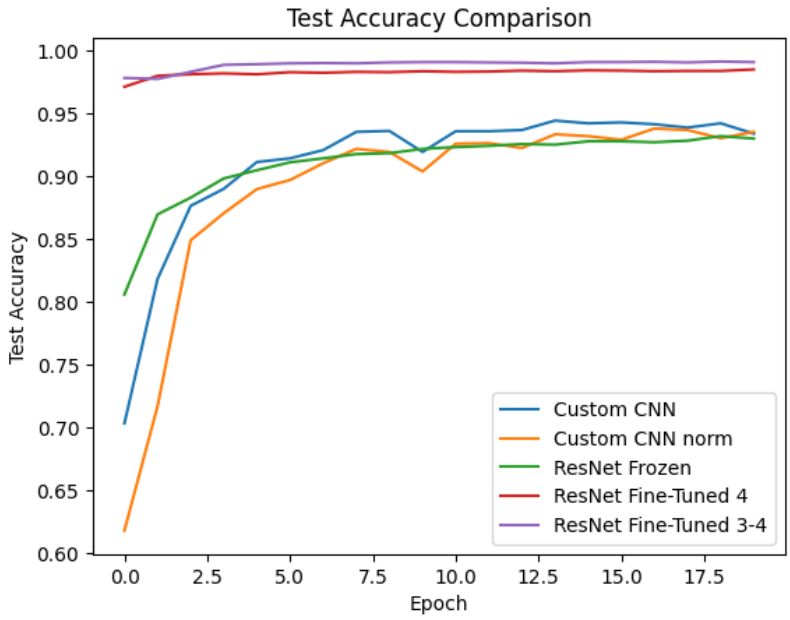
Finally, most errors concern tomato classes, simply because most classes picture tomato diseases.

5. Conclusion

To sum up, we tested various configurations of our custom CNN and ResNet on a dataset examining plant disease classification, drawing the following conclusions:

- All the selected approaches seem to behave reasonably well in this field, test set accuracy was always **superior to 90%** and the less captured patterns caused errors that did not concern healthy-diseased mispredictions. This may results in wrong therapeutic approaches (which is probably better than mistaking an ill plant for a healthy one), but considering the small error-rate, it does not seem enough to explicitly avoid these methods.
- We are satisfied by the behavior of the custom CNN, which reached up to **95% accuracy** on the test set, that is not bad for a small network, trained from scratch on this dataset. The learning rate we chose turned out to be optimal and so did the normalization with ImageNet values. Adam optimizer and Cross Entropy Loss did not create problems either apparently.
- ResNet-18 is able to use the features learned on the ImageNet dataset in this task, reaching up to **93% test accuracy** (training just the FC layer), which means that **transfer learning** was quite successful.
- We managed to achieve excellent performance on the PlantVillage dataset, fine-tuning the ResNet and obtaining the top test accuracy of **99%**.
- Results across the different approaches are comparable, showing superiority of the fine-tuned ResNets, slowly followed by our original CNN:

Approach	Top Accuracy
Custom CNN (dropout + data aug.)	~95.8%
Pre-trained ResNet-18	~93.2%
4^{th} layer fine-tuned ResNet-18	~98.5%
3^{rd}-4^{th} layers fine-tuned ResNet-18 (AdamW)	~99.1%



Finally, possible **future extensions** are many and include:

- Dataset balancing, further data augmentation or generalization strategies.
- Experiments on different datasets.
- Changes to the custom CNN architecture (more layers, different convolutions…)
- Trying different optimizers (simple SGD for instance) or different batch sizes.
- Experimenting transfer learning with other CNNs and examining trade-offs between complexity and effectiveness.

Set up and experiments are available at https://github.com/Matti02co/dla-project-2526 .